Data Structures

ECAP252

Edited by
Balraj Singh

LOVELY
PROFESSIONAL
UNIVERSITY

LOVELY
PROFESSIONAL
UNIVERSITY

Data Structures

Edited By:
Balraj Singh

CONTENT

Unit 01: Basic Concepts

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the need for data structures
- Explain the classification of data structures
- Understand algorithms
- Explain data structure operations

Introduction

Data is any numerical or other information that is represented in a way that can be processed by a computer. The way elements are placed or put together to make a whole is referred to as structure. A data structure is the combination of data and all possible operations that must be performed on that piece of data. Bits, characters, and integers are the most basic data types. Data structures are used to manipulate and assemble information. However, the data available is usually in the amorphous form. When different types of such amorphous data are related to each other, we refer to it as a data structure.

Example: Linked List, Queues, Stacks, Trees etc.

A data structure can be described as a set of domains d, a set of functions F and a set of rules A.

D = {d, F, A}

Where,

D refers to Data structure

d refers to Domain variable

F refers to a set of functions or procedures operating on domain variables.

A is a set of rules governing the operations of these functions on the domain variable.

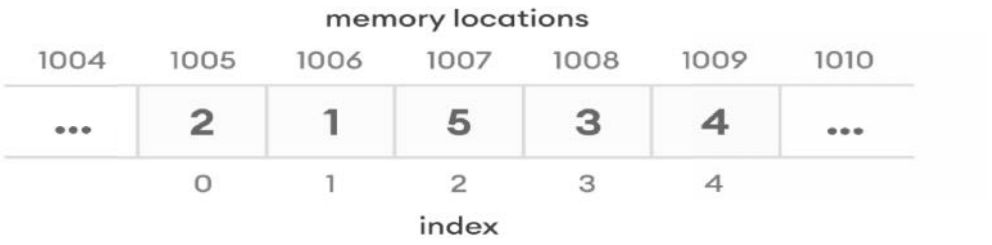
The instructions of a computer program use data to perform certain tasks. Some programs generate data without using any inputs. Some programs generate data using a set of inputs, while some programs use a data set to manipulate the given data. Thus, the data is processed efficiently only by organizing them in a particular structure.

The study of data structures helps to understand the basic concepts involved in organizing and storing data as well as the relationship among the data sets. This in turn helps to determine the way information is stored, retrieved and modified in a computer’s memory. The study of data structures is not limited to the study of data sets. It further extends to the study of representation of data elements. This means that it explains how different types of data are placed in the computer's memory using the binary number system, which forms the storage structure or memory representation. Data structure is implemented in computer programs to manage data. The data is managed using certain logical or mathematical models or concepts. A complex data structure can also be built using simple data structures.

1.1 Data Structures

Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store data sequentially in the memory, then you can go for the Array data structure.



The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory. To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.

1.2 Types of Data Structures

There are two types of data structures:

- Linear data structure
- Non-linear data structure

- **Linear data structures**

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

Following are the popular linear data structures:

1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

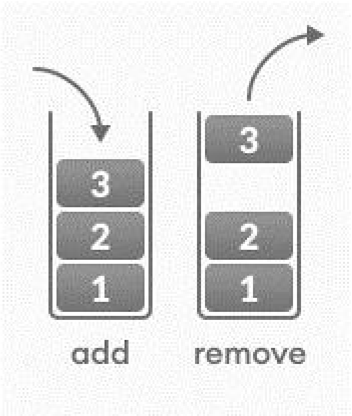
12	2	3	4	85	
0	1	2	3	4	(Index)

Note: An array with each element represented by an index

2. Stack Data Structure

The LIFO technique is used to store elements in a stack data structure. To put it another way, the final element in a stack will be eliminated first.

It works just like a pile of plates where the last plate kept on the pile will be removed first.



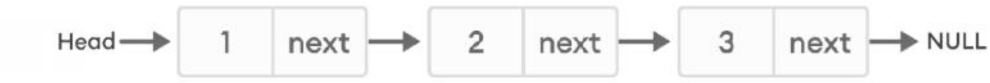
3. Queue Data Structure

Queue is a data structure that is similar to Stacks in that it is an abstract data structure. A queue, unlike stacks, is open on both ends. The one end is always used to input data enqueue), while the other is always used to delete data (dequeue) (dequeue). The Initially-In-First-Out (FIFO) approach is used in Queue, which means that the data item that was stored first would be accessed first. Queue is a data structure that is similar to Stacks in that it is an abstract data structure. A queue, unlike stacks, is open on both ends. The one end is always used to input data (enqueue), while the other is always used to delete data (dequeue) (dequeue). The Initially-In-First-Out (FIFO) approach is used in Queue, which means that the data item that was stored first would be accessed first.

Let us see a real world example of queue data structure

4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.



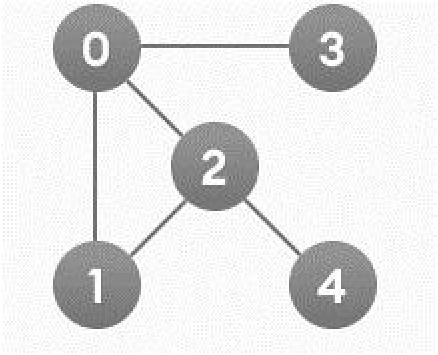
- ## Non linear data structures

Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Non-linear data structures are further divided into graph and tree based data structures.

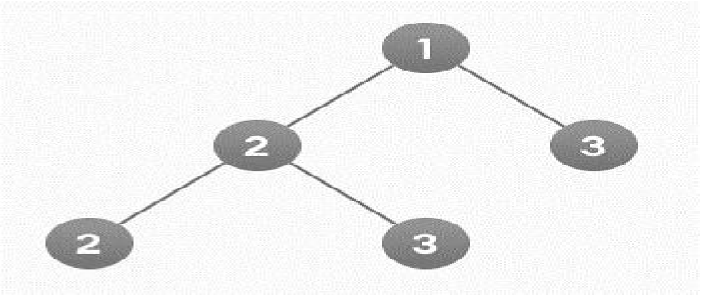
1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.



2. Trees Data Structure

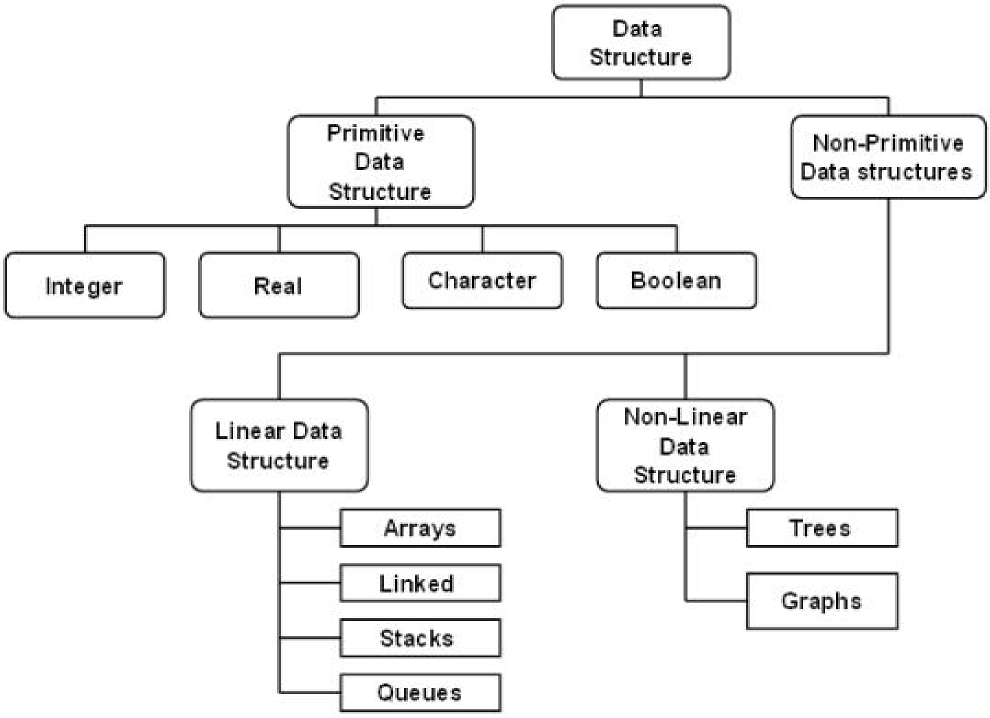
Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.



1.3 Difference between Linear and Nonlinear Data Structures

Main difference between linear and nonlinear data structures lie in the way they organize data elements. In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory. In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them. Non-Linear data structure is that if one element can be connected to more than two adjacent element then it is known as non-linear data structure.

Following diagram shows classification of data structure



1.4 Basic Concepts and Notations of Data Structures

Data structure is a branch of computer science. The study of data structure helps you to understand how data is organized and how data flow is managed to increase efficiency of any process or program. Data structure is the structural representation of logical relationship between data elements. This means that a data structure organizes data items based on the relationship between the data elements.

Example: A house can be identified by the house name, location, number of floors and so on. These structured set of variables depend on each other to identify the exact house. Similarly, data structure is a structured set of variables that are linked to each other, which forms the basic component of a system.

1.5 Data Structures and Algorithms

A data structure is basically an arrangement of data within a computer's memory in computer understandable language. In other words, data is stored in 0 and 1 format and is retrieved in ASCII (American Standard Code for Information Interchange) codes, which is human-understandable format.

The structural and functional aspects of the program depend on the design of the data structure. Thus, a data structure forms the basic building block of a program. Different data structures are used in applications for efficient operation of these applications. The programmers must select the correct data structure to write more efficient programs. This helps to solve the complexity of the problems at a rapid rate.

In computer science, an algorithm is defined as a finite list of distinct instructions for calculating a function. Algorithms are used for data processing, calculation and automated reasoning. An algorithm can also be defined as a set of rules that accurately defines a series of operations.

Example: Instructions for assembling a puzzle can be an example of an algorithm. If you are given a preliminary set of marked pieces, you can follow the instructions given to complete the puzzle.

According to Levitin, algorithms can be defined as, "A sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time."

Most computer programs involve an algorithm and one or more data structures. An appropriate data structure needs to be selected for an algorithm as the efficiency of the algorithm depends on the data structure chosen. By increasing the data storage space, you may not be able to reduce the time needed for processing the data and vice versa.

Example: When we want to print a mailing list alphabetically we need to use a data structure and an algorithm. We first arrange all the names in a data structure (array) and then sort the names alphabetically using an algorithm (sorting).

1.6 The Concept of Data Type

A data type comprises a set of data with values and consists of predefined set of characteristics. To be specific, data is usually stored in a variable, where the value of the variable changes according to the program being executed. The four commonly used data types in C language include int (integer), float (real number), char (character) and pointer. The keywords int, float, char and so on always take lowercase letters. Generally, a data type includes constants and variables. A constant is considered as an entity that does not change in any given program. A variable is an entity that may change from one program to another. It is necessary to specify the variables that will be used in a program. Therefore, type declaration is made by giving the data type and then the variable names. The syntax for declaring the data type and the variable name is as given below:

Syntax:

DATA_TYPE Variable Name

Example:

Int x

Float y

Where x and y are variable name and int and float are data types

Integer Data Type

An integer data type includes only whole numbers. It does not contain any fractional data. It is denoted by the keyword int. It occupies 2 bytes of memory space. Integer data type can either be signed or unsigned. The signed type integer takes both positive and negative values. The range of integer constant is from -32768 to +32767 (-2^15 to +2^15 - 1) for a 16-bit compiler and -128 to 127 (-2^7 to +2^7 - 1) for an 8-bit complier. A 16-bit compiler uses one bit for storing sign and the remaining 15-bits for storing numbers. An 8-bit complier uses one bit for storing sign and the remaining 7-bits for storing numbers. The unsigned integer ranges from 0 to 65535. The signed and unsigned integers are specified as follows:

Syntax:

Unsigned int value;

Signed int value;

The integer data type is denoted by placeholder format string % d, which indicates that the data being used is of integer values.

Example: -

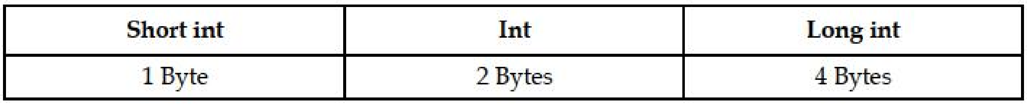
Int x;

Scanf("%d",&x);

Printf("%d",x);

Where scanf() function is used for input data and printf() is used for print output on console screen.

Generally, there are three classes of integers namely, short int, int and long int. As shown in table, short integers occupy only 1 byte, int occupies 2 bytes and the long integers occupy 4 bytes of memory. Long integers can store longer range of values when compared to integer and short integer.



Short int	Int	Long int
1 Byte	2 Bytes	4 Bytes

Floating Point Data Type

The floating point data type contains fractional numbers/real numbers and stores a maximum of six digits after decimal point. The keyword used to denote floating point number is 'float'. It occupies 4 bytes of memory space. The floating point data type is denoted by the placeholder %f, which indicates that the data being used is of floating point values. The three classes of floating point data type are float, double and long double.

Example: -

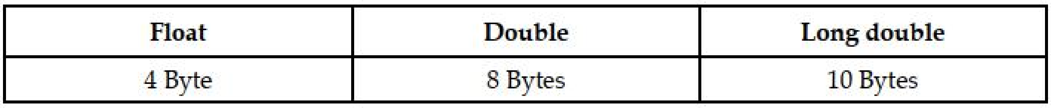
float x;

Scanf("%f",&x);

Printf("%f",x);

Where scanf() function is used for input data and printf() is used for print output on console screen.

As shown in table, float occupies 4 bytes of memory space. Double has longer precision than float and occupies 8 bytes of memory space. The long double further extends the precision and occupies 10 bytes of memory space.



Float	Double	Long double
4 Byte	8 Bytes	10 Bytes

Character Data Type

Character data type consists of a single character. It can store a single special symbol or alphabet placed within single inverted commas. It is denoted by the keyword char. It occupies only 1 byte of memory space. The character data type is denoted by placeholder %c, which indicates that the data being used is of character values.

Example: -

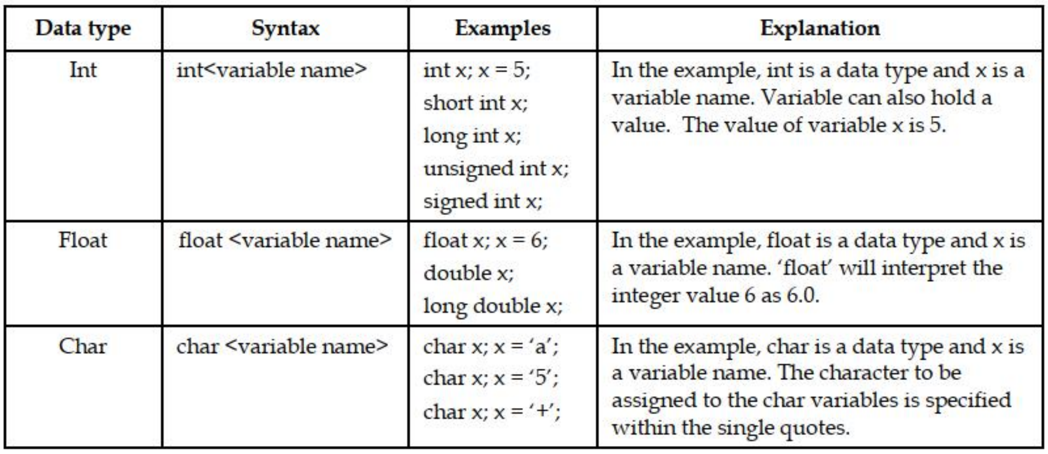
char x;

Scanf("%c",&x);

Printf("%c",x);

Where scanf() function is used for input data and printf() is used for print output on console screen.

Following table gives the syntax and examples of the different data types.



Data type	Syntax	Examples	Explanation

Int	int<variable name>	int x; x = 5;
		short int x;
		long int x;
		unsigned int x;
		signed int x;
			In the example, int is a data type and x is a variable name. Variable can also hold a value. The value of variable x is 5.

Float	float <variable name>	float x; x = 6;
		double x;
		long double x;
			In the example, float is a data type and x is a variable name. 'float' will interpret the integer value 6 as 6.0.

Char	char <variable name>	char x; x = 'a';
		char x; x = '5';
		char x; x = '+';
			In the example, char is a data type and x is a variable name. The character to be assigned to the char variables is specified within the single quotes.

Pointers

A pointer is a reference data structure. A pointer is actually a variable that stores the address of another variable or structure in a program. The pointer variable holds only the memory location and not the actual content. The pointer normally uses the address operator represented by '&' and the indirection operator represented by '*'. The address operator provides the address of the variable and the indirection operator provides the value of the variable which is being pointed by the pointer variable.

1.7 Need for Data Structures

The study of data structure helps programmers to store and manipulate data efficiently. Data structures help to understand the relationship of a data element with other data elements. They also provide various methods to organize and represent the data within the computer's memory.

Data structure is imperative since it governs the types of operations we perform on the data, and the competency of the operations carried out. It also governs how dynamic we can be in dealing with our data.

Note

There are different ways to organize data, for which there is a need for different kinds of data structure.

1.8 Goals of Data Structure

Data structure basically implements two complementary goals. The first goal of data structure is to develop mathematical entities and operations, which can be used to solve particular classes of problems. The second goal is to find out representations for these entities and then implement the operations on those representations. This goal considers implementing the high level data type to solve the problems, which in turn uses existing data types.

The fundamental goal of data structure is to produce solutions that are correct and efficient. This in turn helps to produce quality software. Generally, the production of quality data structure to have quality software implementation involves the following additional goals:

1.9 Correctness

Data structure is designed such that it operates correctly for all kinds of input, which is based on the domain of interest. In other words, correctness forms the primary goal of data structure, which always depends on the specific problems that the data structure is intended to solve.

Example: A data structure designed to store a collection of numbers, in a specific order, must make sure that the numbers are not stored in a haphazard way.

1.10 Efficiency

Data structure also needs to be efficient. It should process the data at high speed without utilizing much of the computer resources such as memory space. In a real time state, the efficiency of a data structure is an important factor that determines the success and failure of the process.

Example: NASA space shuttle requires a high level data structure design, so that it reacts quickly to any changing conditions during a lift-off.

1.11 Advantages of Data Structure

Advantages of Data Structure are:

1. Efficiency
2. Reusability
3. Abstraction

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Algorithms

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

For example, we might need to sort a sequence of numbers into non decreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the sorting problem:

Input: A sequence of n numbers (a_1; a_2; ……; a_n),

Output: A permutation (reordering) (a'_{1}, a'_{2},….,a'_{n}) of the input sequence such that $a'_{1}<a'_{2}$ <…. <a'_{n}

Example: Given the input sequence h31; 41; 59; 26; 41; 58i, a sorting algorithm returns as output the sequence h26; 31; 41; 41; 58; 59i. Such an input sequence is called an instance of the sorting problem. In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, we have a large number of good sorting algorithms at our disposal. Which algorithm is best for a given application depends on—among other factors—the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.

An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

1.12 Qualities of Good Algorithms

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Example: Add two numbers entered by the user

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
    sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Task: Write an Algorithm to find the largest among three numbers.

1.13 Data Structure Operations

Data which appears in the data structures are processed with the help of certain operations. Sometimes two or more of the operations may be used in a given situation.

Example: When you want to delete a record with a given key, you first need to use the search operation to find the location of the record and then use the delete operation.

1.14 Operations on Primitive Data Structure

Various operations can be performed on primitive data structures. Some of these operations are:

1. **Creation Operation**: The creation operation creates a data structure.

Example: Consider an example of integer type data structure.

int a;

Here, declaration of int creates 2 bytes of memory space for variable 'a'. This variable is used to store only integer value.

2. **Destroy Operation**: The destroy operation destroys the data structure. In C language, a function called 'free()' is used to destroy the data structure. This helps in efficient use of memory.

3. **Selection Operation**: The selection operation is used to access data within a data structure. The significance of selection operation is provided in file data structure. Files provide the option of sequential and random access, which totally depend on the nature of files.

4. **Update Operation**: The update operation is used to modify data in data structure.

1.15 Operations on Non-primitive Data Structure

The operations on non-primitive data structure depend on the logical organization of data and their storage structure. Non-primitive data focuses on formation of a set of data elements that are either homogeneous (same data type) or heterogeneous (different data type). Therefore, non-primitive data cannot be operated or manipulated directly by the machine level instructions. Some of the operations on non-primitive data structure are:

1. **Traversing**: Traversing is the method of processing each element exactly once. Traversing is generally done to check the availability of data elements in an array. After carrying out an insertion or deletion operation, you would want to check whether the operation has been successful or not. We use traversing to check if the element is successfully inserted or deleted.

2. **Sorting:** Sorting is the technique of arranging the data elements in some logical order, either ascending or descending order. Some algorithms make use of sorted lists. Therefore, efficient sorting is essential for optimizing these algorithms to ensure that they work correctly.

3. **Merging:** Merging is the method of combining the elements in two different sorted lists into a single sorted list. It is based on the divide-and-conquer algorithm. Merge sort can be considered as the best choice for sorting a linked list as it is easy to implement.

4. **Searching:** Searching is the method of finding the location of an element with a given key value, or finding the location of an element which satisfies a given condition. Searching a data structure allows the efficient retrieval of unambiguous items from a set of items, such as a particular record from a database.

5. **Insertion:** Insertion is the method of adding a new element to the data structure. The insertion process may add a new element in the ith position of the data structure. If sorting also needs to be performed, first we need to assign an item to the given elements and compare it with the previous elements. If the assigned element is smaller than the previous element, we need to swap the positions of both these items. This process is repeated until the correct position of the item is identified.

6. **Deletion:** Deletion refers to removing an item from the structure. When a node is not required in the data structure, it can be removed using the delete operation.

Summary

- Algorithms are used for data processing, calculations, and automated reasoning. An algorithm can be defined as a set of rules that accurately define a series of operations.
- Algorithms are used for data processing, calculations, and automated reasoning. An algorithm can be defined as a set of rules that accurately define a series of operations.
- Two fundamental goals of data structure are correctness and efficiency. Some of the important features of data structures are robustness, adaptability and reusability.
- Data structure can be classified into two categories: primitive data structure and non-primitive data structure.
- Basic data types such as integer, real, character, and Boolean are categorized under primitive data structures. These data types are also known as simple data types because they consist of characters that cannot be divided.
- Non-primitive data structures are further divided into linear and non-linear data structure based on the structure and arrangement of data.

- An Abstract Data Type (ADT) is a technique that is used to specify the logical properties of a data type. It can be considered as a basic mathematical concept used to define the data types.

Keywords

Amorphous: Not having a definite form; shapeless.

Efficiency: Efficiency of a program depends upon the choice of data structures

Creation Operation: The creation operation creates a data structure.

Deletion: Deletion refers to removing an item from the structure.

Self Assessment

1. Which of the following is linear data structure?
A. Trees
B. Graphs
C. Arrays
D. None of these

2. Which of the following is non-linear data structure?
A. Array
B. Linked lists
C. Stacks
D. None of these

3. User defined data type is also called?
A. Primitive
B. Non-primitive
C. Identifier
D. None of these

4. Stack is based on which principle
A. FIFO
B. Push
C. LIFO
D. None of these

5. Describes the running time of an algorithm
A. Asymptotic Notation
B. Time complexity
C. Performance Analysis
D. None of these

6. A procedure for solving a problem in terms of action and their order is called as
A. Process
B. Program instruction
C. Algorithm
D. Template

7. Algorithm can be represented as
A. Pseudocode
B. Flowchart
C. None of the above
D. both Pseudocode and Flowchart

8. Algorithm should have finite number of steps
A. True
B. False

9. Which of the following is not a Characteristics of a Data Structure?
A. Completeness
B. Correctness
C. Time Complexity
D. Space Complexity

10. Which of the following is not a data structure operation
A. Deletion
B. Traverse
C. Code
D. Sorting

11. An algorithm should have _________ well-defined outputs.
A. 0
B. 1
C. 0 or more
D. 1 or more

12. LIFO stands for
A. Last In First Out
B. Late In First Out
C. Light In Figure Out
D. None of the Above

13. Implementation of non – linear data structure is easy
A. True
B. False

14. ____ can be defined as process of combining elements of two data structure
A. Insertion
B. Deletion
C. Sorting
D. Merging

15. _________ can be defined as process of arranging elements of data structure
A. Insertion
B. Deletion
C. Sorting
D. Merging

Answers for Self Assessment

1.	C	2.	D	3.	B	4.	C	5.	A
6.	C	7.	C	8.	A	9.	A	10.	C
11.	D	12.	A	13.	B	14.	D	15.	C

Review Questions

1. Describe the types of Data Structures?
2. What do you mean by linear data structure?
3. How nonlinear data structure are different from linear data structure. Explain your answer with suitable example.
4. What is an algorithm? Write an algorithm to check entered number is even or odd.
5. Explain data structure operations with suitable example.
6. What are the advantages of data structures?
7. Write good qualities of algorithm.
8. What is importance of an algorithm?
9. How linked list is different from array.
10. Give any real life example of data structure.

Further Readings

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Web Links

https://tutorialsinhand.com/tutorials/data-structure-tutorial/data-structure-basics/introduction-to-data-structure.aspx

http://www.cplusplus.com/doc/tutorial/variables/

https://aofa.cs.princeton.edu/home/

Unit 02: Complexity of Algorithms

Contents

Objectives

After studying this unit, you will be able to:

- Understand the basic concepts and notations of data structures
- Explain mathematical notation and functions
- Analyze the algorithmic complexity and time space tradeoff
- Discuss algorithmic analysis

Introduction

Each computer program is a series of instructions that are arranged in a specific order to perform a specific task. A computer program is written to instruct a computer to perform a specific task in order to obtain the desired result. Irrespective of the language used to develop a program, there are some generic steps that can be followed to solve a problem. These generic steps are called algorithms. According to H. Cormen, "Before there were computers, there were algorithms." An algorithm is a set of instructions that performs a particular task. It is considered as a tool that helps to solve a specific computational problem.

Mathematical notation is a system of symbolic representations of mathematical objects and ideas. Mathematical functions appear quite often in the analysis of algorithm along with their notation. Some of the mathematical functions are floor and ceiling functions, summation symbol, factorial, Fibonacci numbers, and so on.

The complexity of an algorithm is a function that describes the efficiency of an algorithm in terms of the amount of data the algorithm must process.

The two main complexity measures of efficiency of an algorithm are:

1. **Time Complexity:** It is a function that describes the time taken by an algorithm to solve a problem.

Example: Big-O notation is used to express the time complexity of an algorithm.

2. **Space Complexity**: It is a function that describes the amount of memory or space required by an algorithm to run. A good algorithm has minimum number of space complexity.

Algorithm analysis is an important part of computational complexity theory. It provides theoretical estimates for the resources that are needed for any algorithm to solve a given problem. These estimates provide an insight into the measures that determine algorithm efficiency. It is necessary to check the efficiency of each of the algorithms in order to select the best algorithm. We can easily measure the efficiency of algorithms by calculating their time complexity. The shorthand way to represent time complexity is asymptotic notation. It is a function that describes the amount of memory or space required by an algorithm to run. A good algorithm has minimum number of space complexity.

Example: Consider the algorithm for sorting a deck of cards. This algorithm continues repeatedly searching through the deck for the lowest card. The square of the number of cards in the deck is the asymptotic complexity of this algorithm.

2.1 Mathematical Notation and Functions

Algorithms are widely used in various areas of study. We can solve different problems using the same algorithm. Therefore, all algorithms must follow a standard. The mathematical notations use symbols or symbolic expressions, which have a precise semantic meaning.

According to Lancelot Hogben, "Every meaningful mathematical statement can also be expressed in plain language. Many plain language statements of mathematical expressions would fill several pages, while to express them in mathematical notation might take as little as one line. One of the ways to achieve this remarkable compression is to use symbols to stand for statements, instructions, and so on.

Asymptotic Notations

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

A problem may have various algorithmic solutions. In order to choose the best algorithm for a particular process, you must be able to judge the time taken to run a particular solution. More accurately, you must be able to judge the time taken to run two solutions, and choose the better among the two.

To select the best algorithm, it is necessary to check the efficiency of each algorithm. The efficiency of each algorithm can be checked by computing its time complexity. The asymptotic notations help to represent the time complexity in a shorthand way. It can generally be represented as the fastest possible, slowest possible or average possible.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

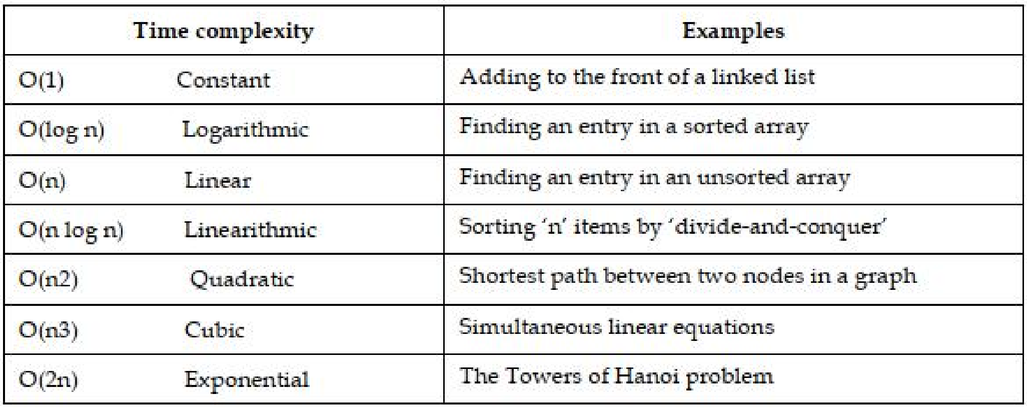
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big-O Notation

'O' is the representation for Big-O notation. Big-O is the method used to express the upper bound of the running time of an algorithm. It is used to describe the performance or time complexity of the algorithm. Big-O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used by the algorithm.

Table gives some names and examples of the common orders used to describe functions. These orders are ranked from top to bottom.



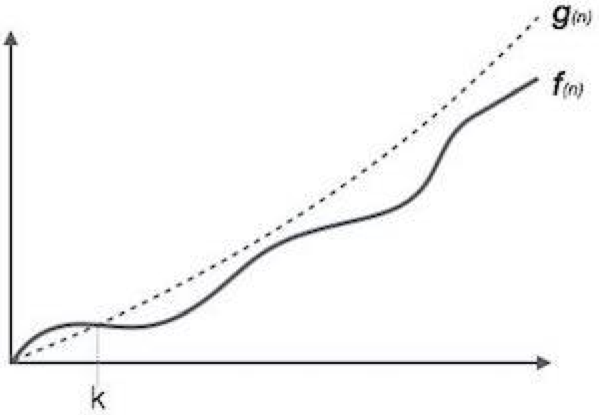
Time complexity		Examples
O(1)	Constant	Adding to the front of a linked list
O(log n)	Logarithmic	Finding an entry in a sorted array
O(n)	Linear	Finding an entry in an unsorted array
O(n log n)	Linearithmic	Sorting 'n' items by 'divide-and-conquer'
O(n2)	Quadratic	Shortest path between two nodes in a graph
O(n3)	Cubic	Simultaneous linear equations
O(2n)	Exponential	The Towers of Hanoi problem

Big-O notation is generally used to express an ordering property among the functions. This notation helps in calculating the maximum amount of time taken by an algorithm to compute a problem. Big-is defined as:

$$f(n) \leq c*g(n)$$

where, n can be any number of inputs or outputs and f(n) as well as g(n) are two non-negativefunctions. These functions are true only if there is a constant c and a non-negative integer n0≥such that,n n0.

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

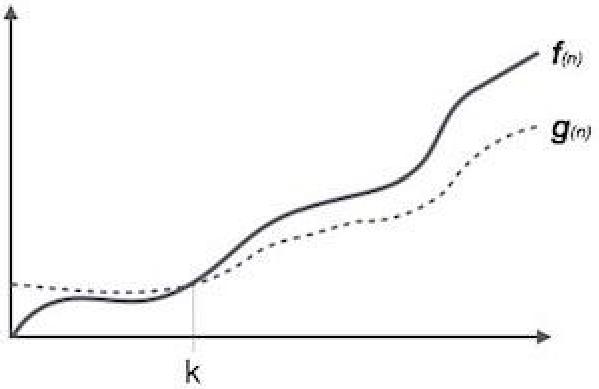


Example for a function f(n)

$O(f(n)) = \{ g(n) :$ there exists $c > 0$ and n_0 such that $f(n) \le c.g(n)$ for all $n > n_0$. $\}$

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

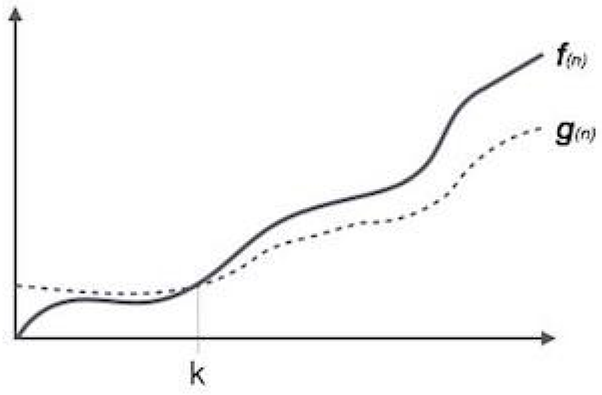


Example for a function f(n)

$\Omega(f(n)) \ge \{ g(n) :$ there exists $c > 0$ and n_0 such that $g(n) \le c.f(n)$ for all $n > n_0$. $\}$

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

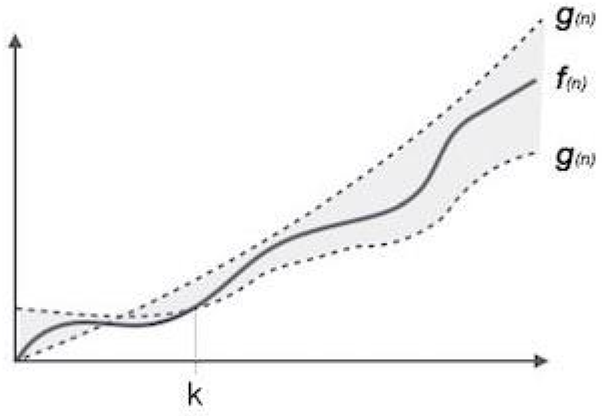


Example for a function f(n)

$\Omega(f(n)) \geq \{ g(n)$: there exists c > 0 and n_0 such that $g(n) \leq c.f(n)$ for all $n > n_0$. }

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



Example for a function f(n)

θ(f(n)) = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n > n0. }

2.2 Common Asymptotic Notations

Following is a list of some common asymptotic notations −

constant	->	O(1)
logarithmic	->	O(log n)
linear	->	O(n)
n log n	->	O(n log n)
quadratic	->	O(n^2)

cubic	->	$O(n^3)$
polynomial	->	$n^{O(1)}$
exponential	->	$2^{O(n)}$

2.3 Mathematical Functions

Mathematical functions express the idea that an input completely determines an output. A function provides exactly one value to each input of a specified type. The value can be real numbers or can be elements from any given sets: the domain and the codomain of the function.

Example

Function f(x)=2x

In this case, the function is assigned to every real number, the real number with twice its value.

Assume x=5, then we can write f(5) = 10

Some of the mathematical functions are described below:

Floor and Ceiling Functions

Floor function is represented as floor(x). Floor function which is also called greatest integer function gives the largest integer less than or equal to x. The range of floor(x) is the set of all integers, but the domain of floor(x) is the set of all real numbers.

Let us take an example to understand the floor function more clearly.

Example

floor(1.01)=1
floor(0)=0
floor(2.9)=2
floor(-3)=-3
floor(-1.1)=-2
Find out floor(x) for various values of x

Ceiling function is represented as ceiling(x). It gives the smallest integer value greater than or equal to x. The domain of ceiling(x) is the set of all real numbers. The range of ceiling(x) is the set of all integers.

Let us consider the following example.

Example

ceiling (1.5)=2

ceiling(0)=0

ceiling(2)=2

ceiling(-3)=-3

ceiling(-1.1)=-1

Find out ceiling(x) for various values of x.

Did you know?

The name and symbol for the floor function and ceiling function was invented by K. E. Iverson (Graham et. al. 1994).

Summation Symbol

Summation symbol is Σ. Summation is the operation of combining a sequence of numbers using addition. The result is the sum or total of all the numbers. Apart from numbers, other types of values such as, vectors, matrices, polynomials, and elements of any additive group can also be added using summation symbol.

Example: Consider a sequence x1, x2, x3……x10. The simple addition of this sequence isx1+x2+x3+x4+x5+x6+x7+x8+x9+x10. Using mathematical notation we canshorten the addition. It can be done by using a symbol to denote “all the way upto” or “all the way down to”.

Then, the expression will be x1+x2+x3+…..+x10. We can also represent theexpression using Greek letter Σ as shown below:

$$\sum_{index\ var\ iable=a}^{b} \mathrm{var}\,iable_{index\ var\ iable}$$

Here, a is the first index and b is the last index. The variables are the numbers that appear constantly in all terms. In the expression,

x1+x2+x3+x4+x5+x6+x7+x8+x9+x10

1 is the first index, 10 is the last index, and x is the variable. If we use i as the index variable then the expression will be

$$\sum_{i=1}^{10} x_i$$

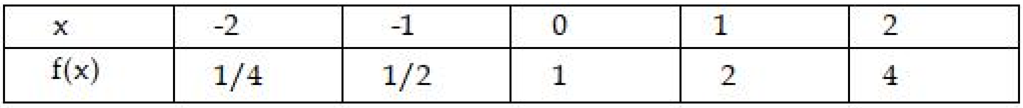
Exponent and Logarithm

Exponential function has the form f(x) =a_x+B where, **a** is the base, **x** is the exponent, and **B** is any expression.

If **a** is positive, the function continuously increases in value. As **x** increases, the slope of the function also increases.

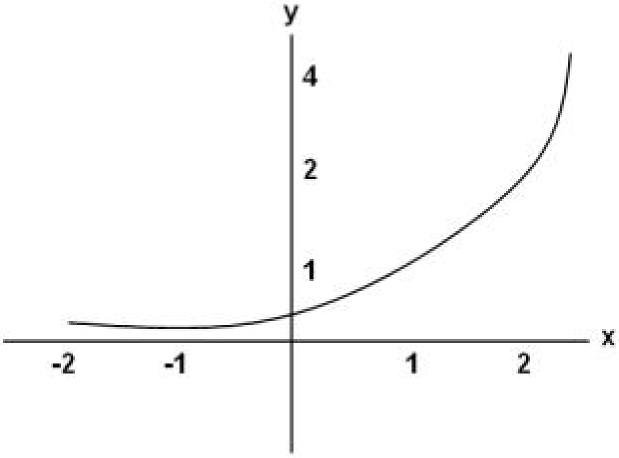
Example: Consider a function. f(x)=2x

Here, we have an exponential function with base 2. Some typical values for this function are:



x	-2	-1	0	1	2
f(x)	1/4	1/2	1	2	4

The graph for y=2x is shown in figure. In the graph as x increases, y also increases, and as x increases the slope of the graph also increases.



A logarithm is an exponent. The logarithmic function is defined as f(x)= logb x. Here, the base of the algorithm is b. The two most common bases which we use are base 10 and base e

Example: Consider the exponential equation 5^2=25 where 5 is base and 2 is exponent.

The logarithmic form of this equation is:

$\text{Log}_5 25=2$

Here, we can say that the logarithm of 25 to the base 5 is 2.

Factorial

The symbol of the factorial function is '!'. The factorial function multiplies a series of natural numbers that are in descending order. The factorial of a positive integer n which is denoted by n! represents the product of all the positive integers is less than or equal to n.

n!=n*(n-1)*(n-2)……2*1

Example

5!=5*4*63*2*1=120

Fibonacci Numbers

In the Fibonacci sequence, after the first two numbers i.e. 0 and 1 in the sequence, each subsequent number in the series is equal to the sum of the previous two numbers. The sequence is named after Leonardo of Pisa, also known as Fibonacci.

Fibonacci numbers are the elements of Fibonacci sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765…….

Sometimes this sequence is given as 0, 1, 1, 2, 3, 5….. There are also other Fibonacci sequences which start with the numbers:

3, 10, 13, 23, 36, 59…….

Fibonacci numbers are the example of patterns that have intrigued mathematicians through the ages. In mathematical terms, the sequence F_n of Fibonacci numbers is defined as:

$$F_n= F_{n-1}+ F_{n-2}$$

Example: Beginning with a single pair of rabbits, if every month each productive pair bears a new pair, who become productive when they are 1 month old, how many rabbits will there be after n months?

Assume that there are x^n pairs of rabbits after n months. The number of pairs in n+1 month is x^{n+1}. Each pair produces a new pair every month but no rabbit dies within that period. New pairs are only born to pairs which are at least 1 month old, so there is an x^{n-1} new pair.

$X^{n+1} = x^n + x^{n-1}$

This equation shows the rules for generating the Fibonacci numbers.

Did you Know?

Fibonacci was the greatest mathematician of his age. He eliminated the use of complex Roman numerals and made mathematics more accessible to the public by bringing the Hindu-Arabic system (including zero) to Western Europe.

2.4 Algorithmic Complexity and Time Space Tradeoff

Complexity is a measure of performance of an algorithm. The complexity of computation is a characterization of time and space requirements, which helps to solve a problem using a specific algorithm. Computational complexity is mostly concerned with the lower bound.

Algorithmic Complexity

We can determine the efficiency of an algorithm by calculating its performance. Following are the two factors that help us to determine the efficiency of an algorithm:

1. Total time required by an algorithm to execute.
2. Total space required by an algorithm to execute.

Thus, the two main considerations required to analyze the program are:

1. Time Complexity
2. Space Complexity

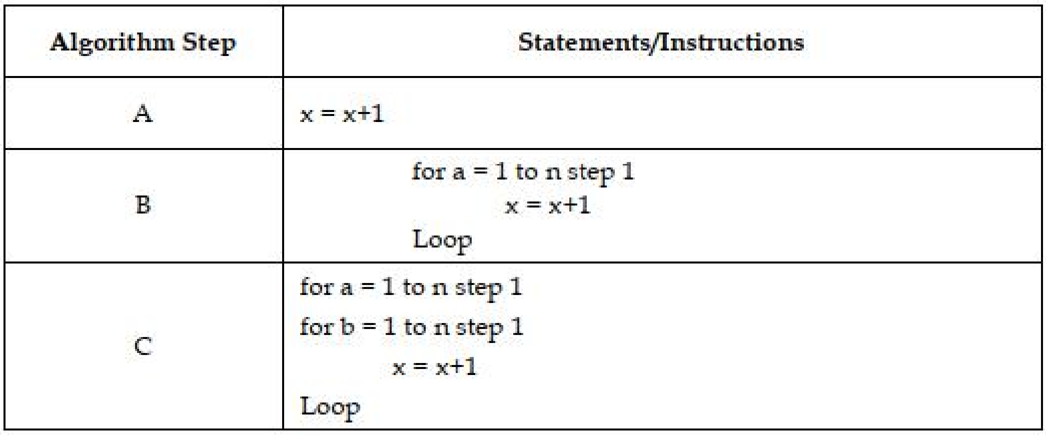
> The amount of computer time required to solve a problem is the time complexity of an algorithm and the amount of memory required to compute the problem is the space complexity of an algorithm.

Time Complexity

Time complexity of an algorithm is the amount of time required by an algorithm to execute. It is always measured using the frequency count of all important statements or the basic instructions. This is because the clock limitation and multiprogramming environment makes it difficult to obtain a reliable timing figure.

The time taken by an algorithm is the sum of compile time and run time. The compile time does not depend on the instance characteristics, as a program once compiled can be run many times without recompiling. Thus, only the run-time of the program matters while calculating time complexity. Let us take an example to get a clear idea of how time complexity of an algorithm is computed.

Table shows analysis of time complexity



Algorithm Step	Statements/Instructions
A	x = x+1

B	for a = 1 to n step 1
	x = x+1
	Loop

C	for a = 1 to n step 1
	for b = 1 to n step 1
	x = x+1
	Loop

1. In step A, there is one independent statement 'x= x+1' and it is not within any loop. Hence, this statement will be executed only once. Thus, the frequency count of step A of the algorithm is 1.
2. In step B, there are three statements out of which 'x = x+1' is an important statement. As the statement 'x = x+1' is contained within the loop, the statement will be executed n number of times.

   Thus, the frequency count of algorithm is n.
3. In step C, the inner and outer loop runs n number of times. Thus, the frequency count is n^2.

During the analysis of algorithm, the focus is on determining those statements that provide the greatest frequency count. The formulas used to calculate the steps executed by an algorithm are:

1 + 2 +……+ n = n(n+1)/2

12+22+…..+ n2 = n(n+1)(2n+1)/6

If an algorithm has input of size n and performs f(n) basic functions, then the time taken to execute those functions will be cf(n), where c is a constant that depends upon the algorithm design.

The time complexity of an algorithm can be further analyzed as best case, worst case and average case time complexity.

1. In best case time complexity, an algorithm will take minimum amount of time to solve a particular problem. In other words, the algorithm runs for a short time.

Example Bubble sort has a best case time complexity of n.

2. In worst case time complexity, an algorithm will take maximum amount of time to solve a particular problem. In other words, algorithm runs for a long time.

Example Quicksort has a worst case time complexity of n^2.

3. In average case time complexity, only certain sets of inputs to the algorithm get the timecomplexity. It specifies the behavior of an algorithm on a particular input.

Example Quicksort has an average case time complexity of n * log(n)

In general, time complexity helps to estimate the number of functions required to solve a problem of size n.

Space Complexity

Space complexity is the amount of memory an algorithm requires to run. The space complexity of an algorithm can be determined by relating the size of a problem (n) to the amount of memory (s) needed to solve that problem. Thus, the space complexity can be computed by using the below two components:

1. Fixed Space Requirement: It is the amount of space acquired by fixed sized structure, variables, and constants.
2. Variable Space Requirement: It is the amount of space required by the structured variables, whose size depends on particular problem instance.

Therefore, to calculate the space complexity of an algorithm we have to consider the following two factors:

1. Constant characteristics

2. Instant characteristics

Thus, the space requirement S(p) is given as:

S(p) = C + Sp

Here, C is the constant (required fixed space) and Sp is the space that depends on a particular instance of variables.

Let us take an example to determine the space complexity of the variables used in a program.

Example

Algorithm: To compute the sum of three elements

//Input: x, y, and z are of integer type

Input x, y, z

//Output: The sum of three integers is returned

return x+y+z

Thus, if each of the input elements occupies 2 bytes of memory space,

then the inputs x, y, z will require a total memory size of 6 bytes.

In general, space complexity helps to define the amount of memory required to solve a particular problem.

Time Space Tradeoff

Most of the algorithms are constructed to work with inputs of arbitrary length. Usually, the efficiency of an algorithm is stated as a function relating to time complexity or space complexity.

Time space tradeoff in context with algorithms relates to the execution of an algorithm. The execution of an algorithm can be done in a short time by using more memory, because execution time increases with less memory. Therefore, proper selection of one alternative over the other is the tradeoff.

Problems like sorting or matrix-multiplication have many choices of algorithms. Some of the choices are extremely space-efficient and some are extremely time-efficient. Research in time-space tradeoff lower bounds seeks to prove that for certain problems, no algorithms exist that achieve lesser time complexity and space complexity simultaneously.

2.5 Algorithmic Analysis

Analysis of an algorithm is required to determine the amount of resources such as time and storage necessary to execute the algorithm. Usually, the efficiency or running time of an algorithm is stated as a function which relates the input length to the time complexity or space complexity.

Algorithm analysis framework involves finding out the time taken and the memory space required by a program to execute the program. It also determines how the input size of a program influences the running time of the program.

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big-O notation, Omega notation, and Theta notation are used to estimate the complexity function for large arbitrary input.

2.6 Types of Analysis

The efficiency of some algorithms may vary for inputs of the same size. For such algorithms, we need to differentiate between the worst case, average case and best case efficiencies.

Best Case Analysis

If an algorithm takes the least amount of time to execute a specific set of input, then it is called the best case time complexity. The best case efficiency of an algorithm is the efficiency for the best case input of size n. Because of this input, the algorithm runs the fastest among all the possible inputs of the same size.

To analyze the best case efficiency, we have to first determine the kind of inputs for which the count C(n) will be the smallest among all possible inputs of size n. Then, we ascertain the value of C(n) on the most convenient inputs.

Example: In case of sequential search, the best case for lists of size n is when their first elements are equal to the search key. Then,

Cbest (n) = 1

Average Case Analysis

If the time complexity of an algorithm for certain sets of inputs are on an average, then such a time complexity is called average case time complexity.

Average case analysis provides necessary information about an algorithm's behavior on a typical or random input. You must make some assumption about the possible inputs of size n to analyze the average case efficiency of algorithm.

Example: Assume that in case of sequential search, the probability of successful search is equal to t i.e. $0 \le t \le 1$, and the probability of the first match occurring in the ith position of the list is the same for all values of i. From these assumptions we can easily find out the average number of key comparisons Cavg (n).

In case of successful search, the probability of the first match occurring in the ith position of the list is t/n for all values of i and the comparison made by the algorithm is also i.

In case of unsuccessful search, the number of comparison is n with the probability of (1 - t) . Therefore, we can write:

$$
\begin{aligned}
C_{avg}(n) &= [1.\frac{t}{n} + 2.\frac{t}{n} + \ldots.. + i.\frac{t}{n} + \ldots. + n.\frac{t}{n}] + n.(1-t) \\
&= \frac{t}{n}[1+2+\ldots i+\ldots+n] + n(i-t) \\
&= \frac{t}{n}\frac{n(n+2)}{2} + n(1-t) \\
&= \frac{t(n+1)}{2} + n(1-t)
\end{aligned}
$$

For t=1, the average number of key comparisons made by sequential search is (n + 1) / 2 which means the algorithm inspects on an average about half of the list's elements. For t=0, the average number of key comparisons is n which means the algorithm inspects all n element on all such inputs.

Worst Case Analysis

If an algorithm takes maximum amount of time to execute for a specific set of input, then it is called the worst case time complexity. The worst case efficiency of an algorithm is the efficiency for the worst case input of size n. The algorithm runs the longest among all the possible inputs of the similar size because of this input of size n.

To determine the worst case efficiency of an algorithm, we have to analyze the algorithm to identify the kind of input suitable for the largest value of the basic operation's count C(n) among all possible inputs of size n. Then, we can compute the worst case value Cworst(n)

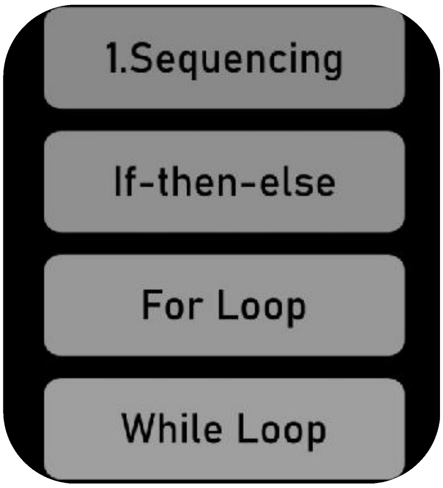
Example: In case of sequential search, if the search element key is present at the nth position of the list, then the basic operations and time required to execute the algorithm is more. Thus, it gives the worst case time complexity. Worst case time complexity is represented as:

Cworst(n)=n

2.7 Control Structures

To analyse a programming code or algorithm, we must notice that each instruction affects the overall performance of the algorithm and, therefore, each instruction must be analysed separately to analyse overall performance.

Some algorithm control structures are



1. **Sequencing**: - Suppose our algorithm is divided into two parts, A and B. A takes time t_A and B takes time t_B for computation. The total computation "t_A + t_B" is according to the sequence rule. According to the maximum rule, this computation time is (max (t_A,t_B)).

Example

Let

t_A =O (n) and t_B = θ (n2).

Then, the total computation time can be calculated as

Computation Time = t_A + t_B

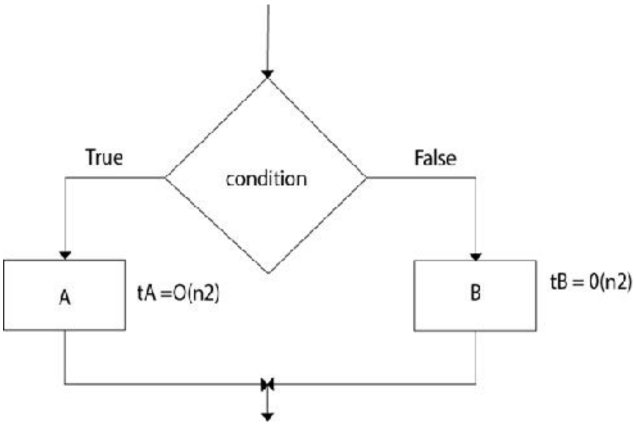
= (max (t_A ,t_B))

= (max (O (n), θ (n2)) = θ (n2))

2. **If-then-else:**-The total time computation is according to the condition rule-"if-then-else." According to the maximum rule, this computation time is max (t_A ,t_B).

Example

- Suppose t_A = O (n2) and t_B = θ (n2)
- Calculate the total computation time for the following:



- Total Computation = (max (tA,tB))
- max (O (n2), θ (n2) = θ (n2)

3. **For loop: -** For loop used when programmer need continuous execution of an algorithm in specific time.

The syntax of loop is

```
For (initialization; condition; updation) {
Statement(s);
}
```

4. **While loop:-**The simple technique for analysing the loop is to determine the function of the variable involved whose value decreases each time around. It is necessary that the value must be a positive integer to terminate the loop.

Task: Program to demonstrate the working of loops control structure.

Summary

- A computer program is written as a sequence of steps that needs to be performed to obtain the desired result.
- Mathematical notation is a system of symbolic representations of mathematical objects and ideas.
- Some of the mathematical functions are floor and ceiling functions, summation symbol, factorial, Fibonacci numbers, and so on.
- The complexity of an algorithm is a function which describes the efficiency of an algorithm in terms of the amount of data the algorithm must process.
- The efficiency of each algorithm can be checked by computing its time complexity.
- The asymptotic notations help to represent the time complexity in a shorthand way. It can generally be represented as fastest possible, slowest possible, or average possible.

- The floor and ceiling functions give the nearest integer up or down.
- In the Fibonacci sequence, after the first two numbers, i.e., 0 and 1 in the sequence, each subsequent number in the series is equal to the sum of the previous two numbers.
- Analysis of an algorithm is required to determine the amount of resources such as, time and storage required to execute the algorithm.

Keywords

Lower Bound: A mathematical argument which means that you can't hope to go faster than a certain amount.

Memory: An internal storage area in the computer.

Notation: The activity of representing something by a special system of characters.

Upper Bound: A number equal to or greater than any other number in a given set.

Self Assessment

1. Space complexity of an algorithm is the maximum amount of _______ required by it during execution.

A. Time
B. Operations
C. Memory space
D. None of the above

2. To measure Time complexity of an algorithm Big O notation is used which:

A. describes limiting behaviour of the function
B. characterises a function based on growth of function
C. upper bound on growth rate of the function
D. all of the mentioned

3. How is time complexity measured?

A. By counting the number of statements in an algorithm
B. By counting the number of primitive operations performed by the algorithm on a given input size
C. By counting the size of data input to the algorithm
D. None of the above

4. Data space is

A. Amount of space used by the variables and data types
B. Amount of space used by the variables and constants
C. Amount of space used by the constants and data types
D. None of above

5. Which of the following case does not exist in complexity theory

A. Best case
B. Average case
C. Worst case
D. Null case

6. Which of the following are types of assumption notations.

A. Big Theta

B. Big Oh

C. Big Omega

D. All of above

7. Which of the following best describes the useful criterion for comparing the efficiency of algorithms?

A. Time

B. Memory

C. Both of the above

D. None of the above

8. Which are not looping structures?
A. For loop
B. While loop
C. Do...while loop
D. if…else

9. The first expression in a for… loop is
A. Step value of loop
B. Value of the counter variable
C. Condition statement
D. None of the above

10. Which of the following control structures is an exit-controlled loop?
A. For loop
B. While loop
C. Const and Goto
D. Do-While loop

11. Which of the following is an invalid if-else statement?
A. if (if (a == 1)){}
B. if (func1 (a)){}
C. if (a){}
D. if ((char) a){}

12. Which of the following statement about for loop is true?
A. Index value is retained outside the loop
B. Index value can be changed from within the loop
C. Goto can be used to jump, out of the loop
D. All of these

13. Some algorithm control structures are
A. Sequencing, if-else, for loop, while loop
B. Insertion, deletion
C. Sorting and merging
D. All of above

14. Loops in C Language are implemented using?
A. While Block
B. For Block
C. Do While Block
D. All the above

15. Which loop is guaranteed to execute at least one time?

A. For

B. While

C. do while

D. None of the above

Answers for Self Assessment

1.	C	2.	D	3.	D	4.	B	5.	D
6.	D	7.	C	8.	D	9.	B	10.	D
11.	A	12.	D	13.	A	14.	D	15.	C

Review Questions

1. "Mathematical notation is a system of symbolic representations of mathematical objects and ideas." Discuss.
2. "To select the best algorithm, it is necessary to check the efficiency of each algorithm." Justify.
3. "Big-O notation describes the performance or time complexity of an algorithm." Comment.
4. "The omega notation can be defined as $f(n) \geq c * g(n)$." Describe.
5. "Floor function gives the largest integer lesser than or equal to x." Describe with an example.
6. Describe modular arithmetic with the help of a 12 hour clock.
7. "Efficiency of an algorithm can be determined by calculating its performance." Comment.
8. "Time complexity of an algorithm is the amount of time required by an algorithm to execute." Discuss with an example.
9. "Time space tradeoff in context of algorithms relates to the execution of an algorithm." Comment.
10. "Analysis of an algorithm is required to determine the amount of resources it requires." Discuss.

Further Readings

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Web Links

https://devopedia.org/algorithmic-complexity#:~:text=Algorithmic%20complexity%20is%20a%20measure,for%20large%20values%20of%20n.&text=Algorithmic%20complexity%20is%20also%20called%20complexity%20or%20running%20time.

Unit 03: Introduction to Pointers

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concepts of pointers
- Identify pointer increment and scale factors
- Pointer expressions
- Pointers and arrays
- NULL Pointer
- Self-Referential structure

Introduction

Computers use their memory for storing instructions of the programs as well as the values of the variables. Since memory is a sequential collection of storage cells each cell has an address associated with it. Whenever we declare a variable, the system allocates, somewhere in the memory, a memory location and a unique address is assigned to this location. Whenever a value is assigned to this variable the value gets stored in the location having a unique address in the memory associated with that variable. Therefore, the values stored in memory can be manipulated using their addresses. Pointer is an extremely powerful mechanism to write efficient programs. Incidentally, this feature makes C stand out as the most powerful programming language. Pointers are the topic of this unit.

3.1 Pointers

A memory variable is merely a symbolic reference given to a memory location. Now let us consider that an expression in a C program is as follows:

```
int a = 10, b = 5, c;

c = a + b;
```

The above expression implies that a, b and c are the variables which can hold the integer data. Now from the above mentioned statement let us assume that the variable 'a' occupies the address 3000 in the memory, 'b' occupies 3020 and the variable 'c' occupies 3040 in the memory. Then the compiler will generate the machine instruction to transfer the data from the location 3000 and 3020 into the CPU, add them and transfer the result to the location 3040 referenced as c. Hence

we can conclude that every variable holds two values:

Address of the variable in the memory (l-value)

Value stored at that memory location referenced by the variable. (r-value)

Pointer is nothing but a simple data type in C programming language, which has a specialcharacteristic to hold the address of some other memory location as its r-value. C programminglanguage provides '&' operator to extract the address of any object. These addresses can be storedin the pointer variable and can be manipulated.

The syntax for declaring a pointer variable is,

```
<data type> *<identifier>;
```

Example

```
int n;

int *ptr; /* pointer to an integer*/
```

The following statement assigns the address location of the variable n to ptr, and ptr is a pointer to n.

```
ptr=&n;
```

Since a pointer variable points to a location, the content of that location is obtained by prefixing the pointer variable by the unary operator * (also called the indirection or dereferencing operator) like, *<pointer_variable>.

Example:

```
# include<stdio.h>

main()

{

int a=10, *ptr;

ptr=&a; /* ptr points to the location of a */

printf("The value of a pointed by the pointer ptr is: %d", *ptr);

/* printing the value of a pointed by ptr through the pointer ptr*/
```

}

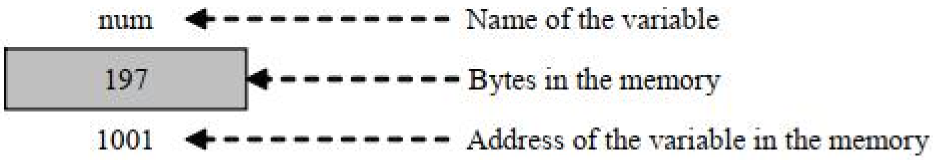
A null value can be assigned to a pointer when it does not point to any data or in the other words, as a good programming habit every pointer should be initialized with the null value. A pointer with a null value assigned to it is nothing but a pointer which contains the address zero.

The precedence of the unary operators '&' and '*' are same in C language. Here as a special case we can mention that '&' operator cannot be used or applied to any arithmetic expression, it can only be used with an operand which has unique address.

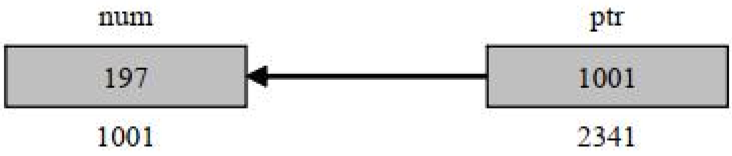
Pointer is a variable which can hold the address of a memory location. The value stored in a pointer type variable is interpreted as an address. Consider the following declarative statement:

int num = 197;

This statement instructs the compiler to reserve a 2-byte memory location (assuming that the target machine stores an int type in two bytes) and to put the value 84 in that location. Assume that a system allocates memory location 1001 for num. diagrammatically it can be shown as:



As the memory addresses are numbers, they can be assigned to some other variable. Let ptr be the variable which holds the address of variable num. We can access the value of num by the variable ptr. Thus, we can say "ptr points to num". Diagrammatically, it can be shown as:



3.2 Accessing the Address of a Variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable?

This can be done with the help of the operator & available in C. The operator & immediately preceding a variable return the address of the variable associated with it.

Example: The statement

P = &quantity;

Would assign the address 5000 to the variable p. The & operator can be remembered as 'address of'.

The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

& 125 (pointing at constant).

Int x[10];

&x (pointing at array names).

&(x+y) (pointing at expressions).

If x is an array, then expression such as

&x[0] and &x[i+3]

are valid and represent the addresses of 0th and (i+3)th elements of x

3.3 Pointer Declaration

Since pointer variables contain address that belongs to a separate data type, they must be declared as pointers before we use them. Pointers can be declared just a any other variables. The declaration of a pointer variable takes the following form:

data_type *pt_name;

The above statement tells the compiler three things about the variable pt_name.

1. The asterisk (*) tells that the variable pt_name is a pointer variable.

2. pt_name needs a memory location.

3. pt_name points to a variable of type data type.

Example: The statement

int *p;

declares the variable p as a pointer variable that points to an integer data type (int). The type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer.

Given below are some more examples of pointer declaration

Pointer declaration	**Interpretation**
Int *rollnumber;	Create a pointer variable rollnumber capable of pointing to an integer type vari able or capable of holding the address of an integer type variable
char *name;	Create a pointer variable name capable of pointing to a character type variable or capable of holding the address of a character type variable
float *salary;	Create a pointer variable salary capable of pointing to a float type variable or capable of holding the address of a float type variable

3.4 Address Operator - &

Once a pointer variable has been declared, it can be made to point to a variable by assigning the address of that variable to the pointer variable. The address of a variable can be extracted using address operator - &.

An expression having & operator generates the address of the variable it precedes. Thus, for example,

&num

produces the address of the variable num in the memory. This address can be assigned to any pointer variable of appropriate type (i.e., the data type of variable num) using an assignment statement such as p = &num; which causes p to point to num. That is, p now contains the address of num.

The assignment shown above is known as pointer initialization. Before a pointer is initialized, it should not be used. A pointer variable can be initialized in its declaration itself.

int x;

int *p = &x;

statement declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. This is an initialization of p, not *p. On the contrary, the statement

int *p = &x, x;

is invalid because the target variable x is not declared before the pointer.

3.5 Indirection Operation - *

Since a pointer type variable contains an assigned address of another variable the value stored in the target variable can be obtained using this address. The value store in a variable can be referred to using a pointer variable pointing to this variable using indirection operator (*).

Example: Consider the following code.

```
int x = 109;

int *p;

p = &x;
```

Then the following expression

```
*p
```

Represents the value 109.

3.6 Pointer Variables

The actual address of a variable is not known immediately. We can determine the address of a variable using 'address of' operator (&). We have already seen the use of 'address of' operator in the scanf( ) function.

Another pointer operator available in C is "*" called "value a address" operator. It gives the value stored at a particular address. This operator is also known as 'indirection operator'.

Example:

```
main( )

{

int i = 3;

printf ("\n Address of i: = %u", & i); /* returns the address * /

printf ("\t value i = %d", * (&i)); /* returns the value of address of i */

}
```

3.7 Initialization of Pointer Variables

Since pointer variables contain address that belong to a separate data type, they must be declared as pointers before we use them.

The declaration of a pointer variable takes the following form:

```
data_type *pt_name
```

This tells the compiler three things about the variable pt_name.

1. The asterisk (*) tells that the variable pt_name is a pointer variable.

2. pt_name needs a memory location.

3. pt_name points to a variable of type data type.

Example: int *p; declares the variable p as a pointer variable that points to an integer data type. The type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer.

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement such as p = &quantity; which causes p to point to quantity. That is, p now contains the address of quantity. This is known as pointer initialization. Before a pointer is initialized, it should not be used. A pointer variable can be initialized in its declaration itself.

Example: int x, *p=&x; statement declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. This is an initialization of p, not *p. On the contrary, the statement int *p = &x, x; is invalid because the target variable x is declared first.

3.8 Accessing a Variable through its Pointer

Consider the following statements:

int q, * i, n;

q = 35;

i = & q;

n = * i;

i is a pointer to an integer containing the address of q. In the fourth statement we have assignedthe value at address contained in i to another variable n. Thus, indirectly we have accessed thevariable q through n. using pointer variable i.

3.9 Pointer Expression

Like other variables, pointer variables can be used in expressions. Arithmetic and comparison operations can be performed on the pointers. For example, if p1 and p2 are properly declared and initialized pointers, then following statements are valid.

y = *p1 * *p2; /multiply values stored in variables pointed to by *p1/and *p2

sum = sum + *p1; /increment sum by the value stored in the variable/pointed to by p1

The pointer may point to any location in the memory therefore you should be careful while using pointers in your programs.

3.10 Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer

Following arithmetic operations are possible on the pointer in C language:

Increment

Decrement

Addition

Subtraction

Comparison

Increment:

It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

Example:

If an integer pointer that stores address 1000 is incremented, then it will increment by 2(size of an int) and the new address it will points to 1002. While if a float type pointer is incremented then it will increment by 4(size of a float) and the new address will be 1004.

Decrement:

It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores address 1000 is decremented, then it will decrement by 2(size of an int) and the new address it will points to 998. While if a float type pointer is decremented then it will decrement by 4(size of a float) and the new address will be 996.

Example:

Program to illustrate pointer increment/decrement

```
#include <stdio.h>

// Driver Code
int main()
{
        // Integer variable
        int N = 4;

        // Pointer to an integer
        int *ptr1, *ptr2;

        // Pointer stores
        // the address of N
        ptr1 = &N;
        ptr2 = &N;

        printf("Pointer ptr1 "
                "before Increment: ");
        printf("%p \n", ptr1);

        // Incrementing pointer ptr1;
        ptr1++;

        printf("Pointer ptr1 after"
                " Increment: ");
        printf("%p \n\n", ptr1);

        printf("Pointer ptr1 before"
                " Decrement: ");
        printf("%p \n", ptr1);

        // Decrementing pointer ptr1;
        ptr1--;


        printf("Pointer ptr1 after"
                " Decrement: ");
        printf("%p \n\n", ptr1);


        return 0;
}
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]

3.11 Pointer and Arrays

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

The array declared as:

static int x[5] = {1, 2, 3, 4, 5}; is stored as follows:

Elements x[0] x[1] x[2] x[3] x[4]

Value 1 2 3 4 5

Address 1000 1002 1004 1006 1008

The name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 1000, the location where x[0] is stored. That is,

x = &x[0] = 1000

If we declare p as an integer pointer, then we can make the pointer p to point to the array x by the

assignment statement

p = x ;

which is equivalent to

p = &x[0];

Now we can access every value of x using p++ to move from one element to another. The relationship between p and x is shown below:

p = &x[0] (=1000)

p+1 = &x[1] (=1002)

p+2 = &x[2] (=1004)

p+3 = &x[3] (=1006)

The address of an element is calculated using its index and the scale factor of the data type, i.e., Address of x[3] = Base Address + (3 × Scale Factor of int) = 1000 + (3 × 2) = 1006

When handling arrays, instead of using array indexing, we can use pointers to access arrayelements, as *(p+3) gives the value of x[3]. The pointer accessing method is much faster than array indexing. &x[i] and (x+i) both represent the address of the ith element of x. x[i] and *(x+i) both represent the contents of that address, the value of the ith element of x. The two terms are interchangeable.

When assigning a value to an array element such as x[i], the left side of the assigned statement may be written as either x[i] or as *(x+i). Thus, a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element. While assigning an address to an identifier, a pointer variable must appear on the left side of the assignment statement. Expressions such as x, (x+1) and &x[i] cannot appear on the left side of an assignment statement because it is not possible to assign an arbitrary address to an array name or an array element.

3.12 Array of Pointers

A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. In such situations the newly defined array will have one less dimension than the original multi-dimensional array. Each pointer will indicate the beginning of a separate (n - 1) dimensional array.

In general terms, a two dimensional array can be defined as one dimensional array of pointers by writing

data_type *array[expression1];

rather than the conventional array definition data_type array[expression1] [expression2]; Similarly, a n dimensional array can be defined as a (n-1) dimensional array of pointers by writing

data_type *array[expression1][expression2]...[expressionn-1];

rather than the conventional array definition data_type array[expression1] [expression2]...

[expressionn];

In these declarations data_type refers to the data type of the original n dimensional array, array is the array name, and expression1, expression2, . . ., expression n are positive-valued integer expressions that indicate the maximum number of elements associated with each subscript.

The array name and its preceding asterisk are not enclosed in parentheses in this type of declaration. Thus, a right-to-left rule first associates the pairs of square brackets with array, defining the named object as an array. The preceding asterisk then establishes that the array will contain pointers.

Moreover, note that the last (the rightmost) expression is omitted when defining an array of pointers, whereas the first (the leftmost) expression is omitted when defining a pointer to a group of arrays.

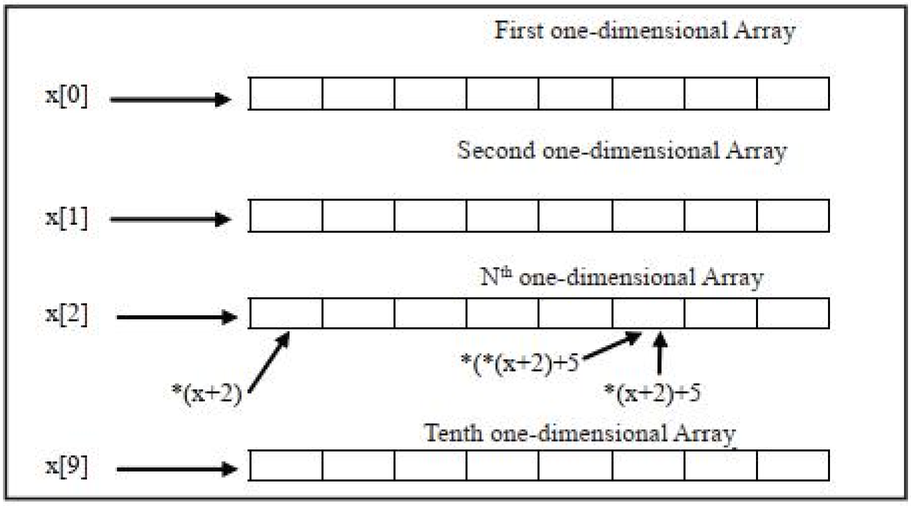
When a n dimensional array is expressed in this manner, an individual array element within the n dimensional array can be accessed by a single use of the indirection operator. The following example illustrates how this is done.

Suppose that x is a two dimensional integer array having 10 rows and 20 columns, we can define x as a one dimensional array of pointers by writing int *x[10];

Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row, and so on. The number of elements within each row is not explicitly specified.

An individual array element, such as x[2][5], can be accessed by writing *(x[2] + 5). In this expression, x[2] is a pointer to the first element in row 2, so that (x[2] + 5) points to element 5 (actually, the sixth element) within row 2. The object of this pointer, *(x[2] + 5), therefore, refers to x[2] [5].

These relationships are illustrated below:


First one-dimensional Array
x[0]
Second one-dimensional Array
x[1]
Nth one-dimensional Array
x[2]
*(x+2)
((x+2)+5
*(x+2)+5
Tenth one-dimensional Array
x[9]

3.13 Pointers and Functions

We can use function pointers to avoid code redundancy. For example a simple qsort() function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use qsort for any data type.

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type. Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function.

```
#include <stdio.h>
#include <time.h>

void getSeconds(unsigned long *par);

int main () {

  unsigned long sec;
getSeconds( &sec );

  /* print the actual value */
printf("Number of seconds: %ld\n", sec );

  return 0;
}

void getSeconds(unsigned long *par) {
  /* get the current number of seconds */
  *par = time( NULL );
return;
}
```

Lab Exercise

```
// Program to demonstrate working of Pointers


#include<stdio.h>
int main(){

int n=123;
int *ptr=&n;
int **nn=n;

//*ptr=&n;
printf("Original value of variable n is = %d\n",n);
printf("Address of n is = %p\n",ptr);
printf("Address of n in decimal number is = %d\n",ptr);
printf("Value of  %d",nn);
return 0;
}
```

Lab Exercise

```
// Program to access value using Pointers


#include<stdio.h>
int main(){
int x;
printf("Enter Value of x\n");
scanf("%d",&x);
int *p;
p=&x;
printf("value of x entered by user is %d\n",x);
printf("Address of  variable x is %p\n",p);
printf("Getting value from pointer variable  %d\n",*p);
printf("Address of variable x is %d\n",p);
printf("Address of variable x is %u\n",p);
*p=500;
printf("New value of x is %d\n",x);
return 0;

}
```

Lab Exercise

```
// Program for Pointer Arithmatics


#include<stdio.h>
int main(){
int a,b,sum,sub,mul,div;
int *ptr1,*ptr2;
printf("Enter first number");
scanf("%d",&a);
printf("Enter second number");
scanf("%d",&b);
ptr1=&a;
ptr2=&b;
sum=*ptr1+*ptr2;
sub=*ptr1-*ptr2;
mul=*ptr1* *ptr2;
div= *ptr1/ *ptr2;
printf("Using pointers, all arithmetic operations
performed\n");
printf("Sum of first and second number is =%d\n",sum);
printf("Subtraction of first and second number is %d\n",sub);
printf("Product of first and second number is %d\n",mul);
printf("Division of first and second number is %d\n",div);
printf("Address of first and second number a=%p, b=%p
",ptr1,ptr2);
return 0;
}
```

3.14 NULL Pointer

A Null Pointer is a pointer that does not point to any memory location. It stores the base address of the segment. The null pointer basically stores the Null value while void is the type of the pointer.

If we do not have any address which is to be assigned to the pointer, then it is known as a null pointer. When a NULL value is assigned to the pointer, then it is considered as a **Null pointer**.

Example:

```
#include<stdio.h>
int main(){

int *ptr;
ptr=NULL;
printf("Value of null pointer is %d",ptr);
return 0;
}
```

3.15 Structure Definition

A structure is a collection of variables referenced under one name providing a convenient means of keeping related information together. The structure definition creates a format that may be used to declare structure variables in a program later on.

The general format of structure definition is as follows:

struct tag_name

{

data_type member1;

data_type member2;

};

A keyword struct declares a structure to hold the details of fields of different datatypes. At this time, no variable has actually been created. Only a format of a new data type has been defined.

Consider the following example:

struct addr

{

```
char name [30];
char street [20];
char city [15];
char state [15];
int pincode;
};
```

The keyword struct declares a structure to hold the details of fine fields of address, namely, #name, street, city, state, pin code. The first four members are character array and fifth one is an integer.

Creating Structure Variables

The structure declaration does not actually create variables. Instead, it defines data type only. For actual use a structure variable needs to be created. This can be done in two ways:

1. Declaration using tagname anywhere in the program.

Example: struct book

```
{
        char name [30];
        char author [25];
        float price;
        }
        struct book book1, book2;
```

2. It is also allowed to combine structure declaration and variable declaration in one statement.

This declaration is given below:

```
        struct person
        {
        char * name;
        int age;
        char *address;
        }
        p1, p2, p3;
```

While declaring structure variables along with their definition, the use of tag name is optional.

```
        struct
        {
        char *name;
        int age;
        char *address;
        }
        p1, p2, p3;
```

Giving Values to Members

As the members are not themselves variables they should be linked to the structure variables. The Link between a member and a variable is established using member operator '.' which is also known as dot operator.

This can be explained using following example:

Example: / * Program to define a structure and assign value to members*/

```
struct book
{
char * name;
int pages;
char *author;
};
main( )
{
struct book b1;
printf (“\n Enter Values:”);
scanf (“%s %d %s”, b1.name, &b1.page, b1.author);
printf (“%s, %d, %s, b1.name, b1.page, b1.author);
}
```

3.16 Structure Pointers

A complete structure can be transferred to a function by passing a structure-type pointer as an argument. In principle, this is similar to the procedure used to transfer an array to a function.

However, we must use explicit pointer notation to represent a structure that is passed as an argument. A structure passed in this manner will be passed by reference rather than by value.

Hence, if any of the structure members are altered within the function, the alterations will be recognized outside the function.

Example:

```
#include <stdio.h>
typedef struct
{
char *name;
int acct_no;
char accttype;
float balance;
}
record;
/* transfer a structure-type pointer to a function */
main( )
{
void adjust (record *pt); /* function declaration * /
static record customer = {“Smith”, 3333, ‘C’, 33.33};
printf (“%s %d %c %.2f\n”, customer.name, customer.acct_no,
customer.acct_type, customer.balance);
adjust (&customer);
printf (“%s %d %c %.2f\n”, customer.name, customer.acct_no,
customer.acct_type, customer.balance);
}
void adjust (record *pt)
{
pt->name = “Jones”;
pt->acct_not = 9999;
pt->acct_type = ‘R’;
pt->balance = 99.99;
return;
}
```

This program illustrates the transfer of a structure to a function by passing the structure’s address (a pointer) to the function. In particular, customer is a static structure of type record, whose members are assigned an initial set of values. These initial values are displayed when the program begins to execute. The structure’s address is then passed to the function adjust where different values are assigned to the member of the structure.

Within adjust, the formal argument declaration defines pt as a pointer to a structure of type record. Also, nothing is explicitly returned from adjust to main. Within main, the current values assigned to the members of customer are again displayed after adjust has been accessed. Thus, the program illustrates whether or not the changes made in adjust carry over to the calling portion of the program.

Executing the program results in the following output:

```
Smith 3333 C 33.33
Jones 9999 r 99.99
```

The value assigned to the members of customer within adjust are recognized within main. A pointer to a structure can be returned from a function to the calling portion of the program. This feature may be useful when several structures are passed to a function, but only one structure is returned.

As we define a pointer pointing to int, or a pointer pointing to a char, similarly, we can have a pointer pointing to a struct. Such pointers are known as ‘structure pointers’. The program given below demonstrates the usage of structure pointer.

```
main( )
{
struct emp
{
char empname [25];
char company [25];
int empno;
};
static struct emp emp1 = {“Prashant”, “SOCEM’, 101};
struct emp *ptr;
ptr = &emp1;
printf (“%s %s %d\n”, emp1.empname,emp1.company,emp1.empno);
printf (“%s %s %d\n”, ptr->company, ptr->empno);
}
```

In the above program, two types of operators are used to refer to structure elements:

1. Dot Operator

2. Arrow Operator

When the structure is referred to by its name, the structure elements are addressed using dot

operators.

Example: b1.name

When the structure is referred to by the pointer to structure, the structure elements are addressed

using arrow operators.

Example: ptr->name

On the left hand side of ‘.’ structure operator, there must always be a structure variable, whereas on the right hand side of the ‘->’ operator there must always be a pointer to a structure.

The following program demonstrates the passing of address of a structure variable to a function.

```
struct emp
{
char empname [25];
int empno;
}
main( )
{
static struct emp emp1 = {Prashant”,”socem”, 101};
display (&emp1);
}
display (e)
struct emp *e; /*pointer to a structure */
{
printf (“%s \n%s\n%d”, e->empname, e->empno);
}
```

```
Output: Prashant
SOCEM
101
```

In the above example, -> operator is used to access the structure elements using pointer to structure.

3.17 Self-Referential Structure

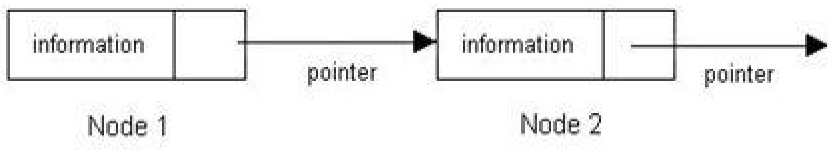
A self-referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind. A chain of such structures can thus be expressed as follows.

```
struct name {
            member 1;
```

```
                    member 2;

                    . . .

                    struct name *pointer;

};
```

The above illustrated structure prototype describes one node that comprises of two logical segments. One of them stores data/information and the other one is a pointer indicating where the next component can be found. .Several such inter-connected nodes create a chain of structures.

The following figure depicts the composition of such a node. The figure is a simplified illustration of nodes that collectively form a chain of structures or linked list.



Such self-referential structures are very useful in applications that involve linked data structures, such as lists and trees. Unlike a static data structure such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self-referential structure can dynamically be expanded or contracted. Operations like insertion or deletion of nodes in a self-referential structure involve simple and straight forward alteration of pointers.

Example

```
#include<stdio.h>
struct data{
int a;
char c;
struct data *ptr;
};
int main(){
struct data data1;
struct data data2;
data1.a=100;
data1.c='A';
data1.ptr=NULL;
data2.a=200;
data2.c='B';
data2.ptr=NULL;
data1.ptr=&data2;
printf("Value of data 1 direct from structure %d %c \n",data1.a,data1.c);
printf("Values of data1 reference to the data2   %d   %c", data1.ptr->a,data1.ptr->c);

return 0;
}
```

Summary

- Pointers are often passed to a function as arguments by reference. This allows data items within the calling function to be accessed, altered by the called function, and then returned to the calling function in the altered form.
- There is an intimate relationship between pointers and arrays as an array name is really a pointer to the first element in the array.
- Access to the elements of array using pointers is enabled by adding the respective subscript to the pointer value (i.e. address of zeroth element) and the expression preceded with an indirection operator.
- As pointer declaration does not allocate memory to store the objects it points at, therefore, memory is allocated at run time known as dynamic memory allocation.
- Self-referential structure are special structure that can hold links to the other structures.

Keywords

Array of Pointer: A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays.

Pointer: It is a variable which can hold the address of a memory location rather than the value at the location.

Pointer Expression: Like other variables, pointer variables can be used in expressions. Arithmetic and comparison operations can be performed on the pointers

Self-Assessment

1. Which one is incorrect statement?

A. int x=90;
B. int *ptr1,*ptr2;
C. ptr1=&x;
D. ptr2=ptr1;

2. Which of the following symbol is used for declare a pointer.

A. #
B. @
C. $
D. *

3. What are the applications of pointers?

A. Implement data structure
B. Dynamic memory allocation
C. Accessing array and functions
D. Above all

4. What are the correct statements about pointers?

A. Pointer is a variable that stores the address of another variable
B. Pointer can also be used to refer to another pointer function
C. Pointers assign and releases the memory as well
D. All of above.

5. What format specifier is used for pointers?

A. %c
B. %d

C. %p
D. %s

6. What is the output of following program?

```
#include<stdio.h>

int main(){

int *ptr=NULL;

printf("Value of null pointer is= %d",ptr);

return 0;

}
```

A. 1
B. 2
C. 0
D. 4

7. Which one is incorrect statement?

A. int x=90;

B. int *ptr1,*ptr2;

C. ptr1=&x;

D. ptr2=ptr1;

8. What type of arithmetic operations can be performed on pointers?

A. Addition

B. Subtraction

C. Multiply

D. Above all

9. What is the output of following program?

```
#include<stdio.h>
int main(){
     int x=90,y=10,result;
    int *ptr1,*ptr2;
    ptr1=&x;
    ptr2=&y;
    result=*ptr1**ptr2;
printf("Product of x and y  using pointers  is %d\n",result);
return 0;
}
```

A. 100

B. 80

C. 900

D. 10

10. Which one is incorrect statement?

A. int a=10,*ptr;

B. ptr=/a;

C. ptr--;

D. above all

11. What are the different operations that can be performed on pointers?

A. Decrement

B. Addition

C. Subtraction

D. Above all

12. Structure can be nested by

A. separate structure

B. Embedded structure

C. both separate structure and Embedded structure

D. none of above

13. Self Referential Structures are_____

A. Have one or more pointers which point to the same type of structure, as their member.

B. Can dynamically be expanded or contracted

C. Both of them

D. None of above

14. What are the types of Self Referential Structures?

A. Structure with Single Link

B. Structure with Multiple Links

C. Both Structure with Single Link and Structure with Multiple Links

D. None of the above

15. What are the correct statements about pointers?

A. Pointer is a variable that stores the address of another variable

B. Pointer can also be used to refer to another pointer function

C. Pointers assign and releases the memory as well

D. All of above.

Answers for Self Assessment

1.	C	2.	D	3.	D	4.	D	5.	C
6.	C	7.	C	8.	D	9.	C	10.	B
11.	D	12.	C	13.	C	14.	C	15.	D

Review Questions

1. Define 'Pointer'. List down the various advantages of using pointers in a C program.

2. How pointer are initialized and implemented in C? Write a program to explain the concept.

3. Explain with the help of a C program, the concept of Pointer Arithmetic in C.

4. How printer in C incorporates the concept of Arrays? Write a suitable program to demonstrate the concept.

5. Differentiate the followings:

(a) Pointer and arrays

(b) Pointer to a variable and pointer to a pointer

(c) Pointer and variable

(d) Value in a function and address in a function

6. Twenty-five numbers are entered from the keyboard into an array. Write a program to find out how many of them are positive, how many are negative, how many are even and how many odd.

7. Write a function to calculate the factorial value of any integer entered through the keyboard.

8. Write a program that demonstrate the working of self-referential structure.

9.Explain pointers and functions with suitable example.

10.As pointer declaration does not allocate memory to store the objects it points at, therefore, Memory is allocated at run time known as dynamic memory allocation.

Further Readings

Books Ashok N. Kamthane, "Programming with ANCI & Turbo C", Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web links

https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/

https://www.careerride.com/C-self-referential-structure.aspx

Unit 04: Arrays

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain arrays
- Describe two dimensional array
- Describe array initialization

Introduction

An array is a group of data items of same data type that share a common name. Ordinary variables are capable of holding only one value at a time. If we want to store more than one value at a time in a single variable, we use arrays.

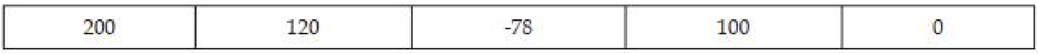
An array is a collective name given to a group of similar quantities. Each member in the group is referred to by its position in the group.

Arrays are allotted the memory in a strictly contiguous fashion. The simplest array is one dimensional array which is simply a list of variables of same data type. An array of one-dimensional arrays is called a two-dimension array.

4.1 Arrays

Arrays are allocated the memory in a strictly contiguous fashion. The simplest array is one dimensional array which is a list of variables of same data type. An array of one dimensional arrays is called a two dimensional array; array of two dimensional arrays is three dimensional array and so on.

The members of the array can be accessed using positive integer values (indicating their order in the array) called subscript or index. Look at an array of integers as shown below:



200	120	-78	100	0
a[0]	a[1]	a[2]	a[3]	a[4]

The description of this array is listed below:

Name of the array: a

Data type of the array: integer

Number of elements: 5

Valid index values: 0, 1, 2, 3, 4

Value stored at the location a[0] : 200

Value stored at the location a[1] : 120

Value stored at the location a[2] : -78

Value stored at the location a[3] : 100

Value stored at the location a[4] : 0

4.2 Advantages of Arrays

Arrays offer a number of advantages, some of which are elucidated below:

1. If only a limited number of variables of a particular data type is required ion a program, one can choose the variable names to suite the situation. Let us say we require five integer type variables, we can define them as follows:

```
int v_one, v_two, v_three, v_four, v_five;
```

Now, consider if we require hundred integer type variables, is the above approach convenient? Obviously not. We can, instead, use an array of integer type having 100 elements as shown below:

```
int num[100];
```

2. Array elements can be accessed using index. Therefore, all the elements can be processed in a desired manner in a single for loop that runs for each element, as shown below:

```
for(i=0; i<100; i++)
num[i]=num[i]+10;
```

In a single for loop, all the elements have been incremented by 10.

3. Since array elements are physically created contiguously in the memory, they can be accesses using pointers (as you will learn later). Therefore, there are more than one way to reference array elements.

4.3 Types of Arrays

According the number of subscripts required to access an array element, arrays can be of

Following types:

1. One-dimensional array

2. Multi-dimensional array

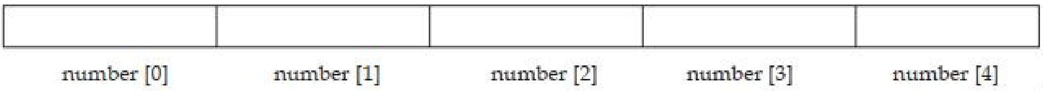
1. One-dimensional Array

A list of items can be given one variable name using only one subscript and such a variable is called a one-dimensional array.

Example: If we want to store a set of five numbers by an array variable number. Then it will be accomplished in the following way:

int number [5];

This declaration will reserve five contiguous memory locations capable of storing an integer type value each, as shown below:

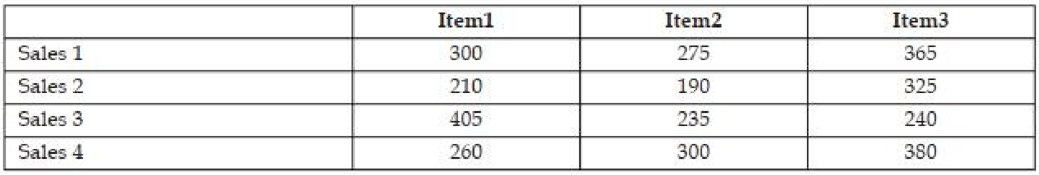


As C performs no bounds checking, care should be taken to ensure that the array indices are within the declared limits. Also, indexing in C begins from 0 and not from 1.

2. Two-dimensional and Multi-dimensional Array

It is possible to have an array of more than one dimensions. Two dimensional array (2-D array) is an array of number of 1-dimensional arrays.

A two dimensional array is also called a matrix. Consider the following table:



	Item1	Item2	Item3
Sales 1	300	275	365
Sales 2	210	190	325
Sales 3	405	235	240
Sales 4	260	300	380

This is a table of four rows and three columns. Such a table of items can be defined using two dimensional arrays.

General form of declaring a 2-D array is

data_typearray_name [row_size] [colum_size];

Example: int marks [4] [2];

It will declare an integer array marks of four rows and two columns. An element of this array can be accessed by the manipulation of both the indices. printf ("%d", marks [2] [1]) will print the element present in third row and second column.

C allows arrays of three or more dimensions. Multi-dimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript.

The general form of a multi-dimensional array is

data_typearray_name [s1] [s2] [s3] . . . [sm];

E.g.: int survey [3] [5] [12];

float table [5] [4] [5] [3];

Here, survey is a 3-dimensional array declared to contain 180 integer type elements. Similarly, table is a 4-dimensional array containing 300 elements of floating point type.

Let us consider some applications of multidimensional array programming.

Tasks: Write a program to find transpose of matrix.

Example - Sorting an integer array.

```
# include <stdio.h>
void main( )
{
int arr [5];
int i, j; temp;
printf (“\n Enter the elements of the array:”};
scanf (“%d”, &arr [i]);
for (i = 0; i< = 4; i ++);
{
for (J = 0; J < = 3; J ++)
if (arr [J] >arr [J+1])
{
temp = arr [J];
arg [J] = arr [J+1];
arr [J+1] = temp;
}
}
printf (“\ n The Sorted array is:”);
for (i = 0; i< 5; i++)
printf (“\ t %d”, arr [i]);
}
```

Example - Accept character string and find its length.

We will solve this question by looping instead of using Library function strlen( ).

```
# include <stdio.h>
void main( )
{
char name [20];
int i, len;
printf (“\n Enter the name:”);
scanf (“%s”, name);
for (i = 0; name [i] ! = ‘\0’; i++);
Len = i - 1;
print f(“\n Length of array is % d”, len);
```

Character Arrays

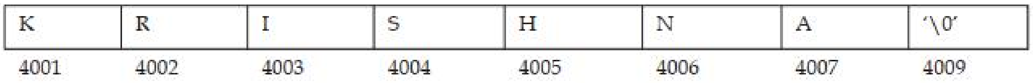
Just as a group of integers can be stored in an integer array, group of characters can be stored in a character array or “strings”. The string constant is a one dimensional array of characters terminated by null character (‘\0’). This null character ‘\0’ (ASCII value0) is different from ‘O’

(ASCII value 48).

The terminating null character is important because it is the only way the function that works with string can know where the string ends.

Example: Static char name [ ] = {‘K’, ‘R’, ‘I’, ‘S’, ‘H’, ‘\0’};

This example shows the declaration and initialization of a character array. The array elements of a character array are stored in contiguous locations with each element occupying one byte of memory.



K	R	I	S	H	N	A	'\0'
4001	4002	4003	4004	4005	4006	4007	4009

Notes

1. Contrary to the numeric array where a 5 digit number can be stored in one array cell, in the character arrays only a single character can be stored in one cell. So in order to store an array of strings, a 2-dimensional array is required.
2. As scanf( ) function is not capable of receiving multi word string, such strings should be entered using gets( ).

Task: Point out the errors, if any, in this program:

```
main(){

            int i, a - 2, b - 3 ;
            int arr.[ 2 +3] ;
             for (i0;i<a+b;i++ )
            {
                        scanf( "%d", &rarr[i] ) ;
                         printf ( " \ n%d", arr[i] ) ;
}
}
```

4.4 Array Declaration

Arrays are defined in the same manner as ordinary variables, except that each array name must be accompanied by the size specification.

The general form of array declaration is:

```
            data_typearray_name [size];
```

data-type specifies the type of array, size is a positive integer number or symbolic constant that indicates the maximum number of elements that can be stored in the array.

Example: float height [50];

This declaration declares an array named height containing 50 elements of type float. The compiler will interpret first element as height [0]. As in C, the array elements are induced

for 0 to [size-1].

Two dimensional arrays can be declared similarly, as shown below:

```
            data_typearray_name[size1][size2];
```

For instance, the following array (named b) is array of 2 arrays of integer type of size 5 elements:

int b[2][5];

The array b has 10 (2 * 5) elements, each capable of storing an integer type data, referenced as:

b[0][0] b[0][1] b[0][2] b[0][3] b[0][4]

b[1][0] b[1][1] b[1][2] b[1][3] b[1][4]

Multidimensional arrays can be declared on the similar lines. A three dimensional array (named c) of int type has been declared below:

Int c[2][2][5];

The array c has 20 (2 * 2 * 5) elements, each capable of storing an integer type data, referenced as:

c[0][0][0] c[0][0][1] c[0][0][2] c[0][0][3] c[0][0][4]

c[0][1][0] c[0][1][1] c[0][1][2] c[0][1][3] c[0][1][4]

c[1][0][0] c[1][0][1] c[1][0][2] c[1][0][3] c[1][0][4]

c[1][1][0] c[1][1][1] c[1][1][2] c[1][1][3] c[1][1][4]

4.5 Array Initialization

One-dimensional Array

The elements of an array can be initialized in the same way as the ordinary variables, when they are declared. Given below are some examples which show how the arrays are initialized.

static int num [6] = {2, 4, 5, 45, 12};

static int n [ ] = {2, 4, 5, 45, 12};

static float press [ ] = {12.5, 32.4, -23.7, -11.3};

In these examples note the following points:

1. Till the array elements are not given any specific values, they contain garbage value.

2. If the array is initialized where it is declared, its storage class must be either static or extern.

If the storage class is static, all the elements are initialized by 0.

3. If the array is initialized where it is declared, mentioning the dimension of the array is optional.

Two-dimensional Arrays

Two dimensional arrays may be initialized by a list of initial values enclosed in braces following their declaration.

E.g.: static int table[2][3] = {0, 0, 0, 1, 1, 1};

initializes the elements of the first row to 0 and the second row to one. The initialization is done by row.

The aforesaid statement can be equivalently written as

static int table[2][3] = {{0, 0, 0}, {1, 1, 1}};

by surrounding the elements of each row by braces.

We can also initialize a two dimensional array in the form of a matrix as shown below:

static int table[2][3] = {{0, 0, 0},

{1, 1, 1}};

The syntax of the above statement. Commas are required after each brace that closes off a row, except in the case of the last row.

If the values are missing in an initializer, they are automatically set to 0. For instance, the statement

static int table [2] [3] = {{1, 1},

{2}};

will initialize the first two elements of the first row to one, the first element of the second row to two, and all the other elements to 0.

When all the elements are to be initialized to 0, the following short cut method may be used.

static int m [3] [5] = {{0}, {0}, {0}};

The first element of each row is explicitly initialized to 0 while other elements are automatically initialized to 0.

While initializing an array, it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional. Thus, the following declarations are acceptable.

static int arr [2] [3] = {12, 34, 23, 45, 56, 45};

static int arr[ ] [3] = {12, 34, 23, 45, 56, 45 };

Multi-dimensional Array

Example: Example of initializing a 4-dimensional array:

static int arr [3] [4] [2] = {{{2, 4}, {7, 8}, {3, 4}, {5, 6},},

{{7, 6}, {3, 4}, {5, 3}, {2, 3}, },

{{8, 9}, {7, 2}, {3, 4}, {6, 1}, } };

In this example, the outer array has three elements, each of which is a two dimensional array of

four rows, each of which is a one dimensional array of two elements.

Tasks: Write a program to find inverse of matrix.

4.6 Accessing Elements of an Array

Once an array is declared, individual elements of the array are referred using subscript or index number. This number specifies the element's position in the array. All the elements of the array are numbered starting from 0. Thus number [5] is actually the sixth element of an array.

Consider the program given above. It has entered 6 values in the array num. Now to read values from this array, we will again use for Loop to access each cell. The given program segment explains the retrieval of the values from the array.

```
for (count = 0; count < 6; count ++)
{
printf ("\n %d value =", num [count]);
}
```

Data can be inserted into array by treating the array elements just like any other variable. If an integer value is to be read from keyboard into an array element (say c[2][3][0]), the following code snippet would do the job:

Scanf("%d", &c[2][3][0]);

In order to read values in the entire array for loop may be used as explained by the following examples:

main( )

```
{
int num [6];
int count;
for (count = 0; count < 6; count ++)
{
printf ("\n Enter %d element:" count+1);
scanf ("%d", &num [count]);
}
}
```

In this example, using the for loop, the process of asking and receiving the marks is accomplished. When count has the value zero, the scanf( ) statement will cause the value to be stored at num [0].

This process continues until count has the value greater than 5.

Case Study: Each element of the array has a memory address. The following program prints an array limit value and an array element address.

Program:

```
#include <stdio.h>
void printarr(int a[]);
main()
{
int a[5];
for(int i = 0;i<5;i++)
{
a[i]=i;
}
printarr(a);
}
void printarr(int a[])
{
for(int i = 0;i<5;i++)
{
printf("value in array %d\n",a[i]);
}
}
void printdetail(int a[])
{
for(int i = 0;i<5;i++)
{
printf("value in array %d and address is %16lu\n",a[i],&a[i]);
\\ A
}
}
```

Explanation

1. The function printarr prints the value of each element in arr.

2. The function printdetail prints the value and address of each element as given in statement A. Since each element is of the integer type, the difference between addresses is 2.

3. Each array element occupies consecutive memory locations.

4. You can print addresses using place holders %16lu or %p.

Questions

1. Write a program to add two 6 x 6 matrices.

2. Write a program to multiply any two 3 x 3 matrices.

3. Write a program to sort all the elements of a 4 x 4 matrix.

4. Write a program to obtain the determinant value of a 5 x 5 matrix.

4.7 Passing Array as an Argument to Function

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Method -1

Formal parameters as a pointer −

```
void myFunction(int *param) {
   .
   .
   .
}
```

Method -2

Formal parameters as a sized array −

```
void myFunction(int param[10]) {
   .
   .
   .
}
```

Method -3

Formal parameters as an unsized array −

```
void myFunction(int param[]) {
   .
   .
   .
}
```

Example:Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows −

```
double getAverage(int arr[], int size)
 {
   int i;
   double avg;
   double sum = 0;

   for (i = 0; i< size; ++i) {
      sum += arr[i];
   }
   avg = sum / size;
   return avg;
}
```

Now, let us call the above function as follows −

```
#include <stdio.h>
 double getAverage(int arr[], int size);
int main () {
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
   avg = getAverage( balance, 5 ) ;
printf( "Average value is: %f ", avg );
      return 0;
}
```

Tasks: Write a program to print sum of two matrices.

Summary

- An array is a group of memory locations related by the fact that they all have the same name and same data type.
- An array including more than one dimension is called a multidimensional array.
- The size of an array should be a positive number. If an array in declared without a size and in initialized to a series of values it is implicitly given the size of number of initializers.
- Array subscript always starts with 0. Last element's subscript is always one less than the size of the array e.g., an array with 10 elements contains element 0 to 9. Size of an array must be a constant number.

Keywords

Array: A user defined simple data structure which represents a group of same type of variables having same name each being referred to by an integral index

Multidimensional array: An array in which elements are accessed using multiple indices

One dimensional array: An array in which elements are accessed using a single index

Subscript/Index: The integral index by which an array element is accessed

Two dimensional array: An array in which elements are accessed using two indices

SelfAssessment

1. What is an Array?

A. A group of elements of same data type.
B. An array contains more than one element.
C. Array elements are stored in memory in continuous or contiguous locations.
D. All the above.

2. An array Index starts with?

A. 1
B. 0
C. -1
D. 2

3. Arrays can

A. store data elements of same data type at contiguous memory location.
B. be used for CPU scheduling.
C. be used for reverse data elements, sort data elements etc.
D. All of above

4. How many kinds of elements an array can have?

A. Char and int type
B. Only char type
C. Only int type
D. All of them have same type

5. Choose the correct statement

A. Array stores data of the same type
B. Array can be a part of a structure
C. Array of structure is allowed
D. All of the above

6. Two dimensional arrays in C

A. An array of arrays is known as two dimensional array.
B. An array of loops
C. An array of tokens
D. All of above

7. Choose the correct syntax for two dimensional array
A. data_type name_of_array;
B. data_type name_of_array[rows][columns];
C. data_type [rows][columns];
D. name_of_array[rows][columns];

8. What will be the output of the following C code?

```
#include <stdio.h>
   void main()
   {
      int a[2][3] = {1, 2, 3, 4, 5};
      int i = 0, j = 0;
      for (i = 0; i< 2; i++)
      for (j = 0; j < 3; j++)
      printf("%d", a[i][j]);
   }
```

A. 1 2 3 4 5 0
B. 1 2 3 4 5 junk
C. 1 2 3 4 5 5
D. Run time error

9. How do you initialize an array in C programming?
A. int arr[3] = (1,2,3);
B. int arr(3) = {1,2,3};
C. int arr[3] = {1,2,3};
D. int arr(3) = (1,2,3);

10. Matrix can be represented using one dimension array.
A. True
B. False

11. What are the advantages of arrays?
A. Objects of mixed data types can be stored
B. Elements in an array cannot be sorted
C. Index of first element of an array is 1
D. Easier to store elements of same data type

12. What is the maximum number of dimensions an array in C may have?
A. Two
B. Eight
C. Sixteen
D. Theoretically no limit. The only practical limits are memory size and compilers

13. Array can be considered as set of elements stored in consecutive memory locations but having __________.
A. Same data type
B. Different data type
C. Same scope

D. None of these

14. Array is an example of _______ type memory allocation.
A. Compile time
B. Run time
C. All of above
D. None of the above

15. Array elements assessed using index of array.
A. True
B. False

Answers forSelf Assessment

1.	D	2.	B	3.	D	4.	D	5.	D
6.	A	7.	B	8.	A	9.	C	10.	B
11.	D	12.	D	13.	A	14.	A	15.	A

Review Questions

1. Explain the usefulness of Arrays in C.
2. What do you mean by 'Array'? How it can be declared & initialized in a C program?
3. Draw a diagram to represent the internal storage of an Array.
4. Describe the different types of Array. Give suitable programs.
5. Find the smallest number in an array using pointers.
6. If an array arr contains n elements, then write a program to check if arr[0] = arr[n-1], arr[1] = arr[n-2] and so on.
7. Write a program to copy the contents of one array into another in the reverse order.
8. Write a program to pick up the largest number from any 5 row by 5 column matrix.
9. Write a program to obtain transpose of a 4 x 4 matrix. The transpose of a matrix is obtainedby exchanging the elements of each row with the elements of the corresponding column.
10. Write a program that interchanges the odd and even components of an array.

Further Readings

Books Ashok N. Kamthane, “Programming with ANCI & Turbo C”, Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web Links

https://www.cs.uic.edu/~jbell/CourseNotes/C_Programming/Arrays.html

https://www.javatpoint.com/c-array

Unit 05: Operations on Arrays

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain arrays
- Describe two dimensional array
- Describe array initialization

Introduction

An array is a group of data items of same data type that share a common name. Ordinary variables are capable of holding only one value at a time. If we want to store more than one value at a time in a single variable, we use arrays.

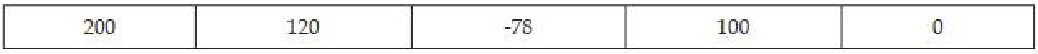
An array is a collective name given to a group of similar quantities. Each member in the group is referred to by its position in the group.

Arrays are allotted the memory in a strictly contiguous fashion. The simplest array is one dimensional array which is simply a list of variables of same data type. An array of one dimensional arrays is called a two dimension array.

5.1 Arrays

Arrays are allocated the memory in a strictly contiguous fashion. The simplest array is one dimensional array which is a list of variables of same data type. An array of one dimensional arrays is called a two dimensional array; array of two dimensional arrays is three dimensional array and so on.

The members of the array can be accessed using positive integer values (indicating their order in the array) called subscript or index. Look at an array of integers as shown below:



200	120	-78	100	0

a[0] a[1] a[2] a[3] a[4]

The description of this array is listed below:

Name of the array : a

Data type of the array : integer

Number of elements : 5

Valid index values : 0, 1, 2, 3, 4

Value stored at the location a[0] : 200

Value stored at the location a[1] : 120

Value stored at the location a[2] : -78

Value stored at the location a[3] : 100

Value stored at the location a[4] : 0

5.2 Advantages of Arrays

Arrays offer a number of advantages, some of which are elucidated below:

1. If only a limited number of variables of a particular data type is required ion a program, one can choose the variable names to suite the situation. Let us say we require five integer type variables, we can define them as follows:

```
int v_one, v_two, v_three, v_four, v_five;
```

Now, consider if we require hundred integer type variables, is the above approach convenient? Obviously not. We can, instead, use an array of integer type having 100 elements as shown below:

```
int num[100];
```

2. Array elements can be accessed using index. Therefore, all the elements can be processed in a desired manner in a single for loop that runs for each element, as shown below:

```
for(i=0; i<100; i++)
num[i]=num[i]+10;
```

In a single for loop, all the elements have been incremented by 10.

3. Since array elements are physically created contiguously in the memory, they can be accesses using pointers (as you will learn later). Therefore, there are more than one way to reference array elements.

5.3 Types of Arrays

According the number of subscripts required to access an array element, arrays can be of

Following types:

1. One-dimensional array

2. Multi-dimensional array

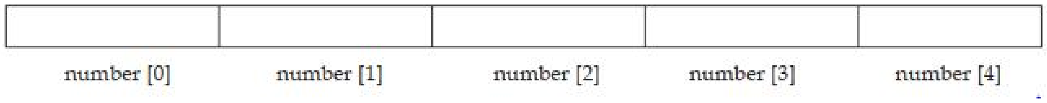
1. One-dimensional Array

A list of items can be given one variable name using only one subscript and such a variable is called a one dimensional array.

Example: If we want to store a set of five numbers by an array variable number. Then it will be accomplished in the following way:

```
int number [5];
```

This declaration will reserve five contiguous memory locations capable of storing an integer type value each, as shown below:

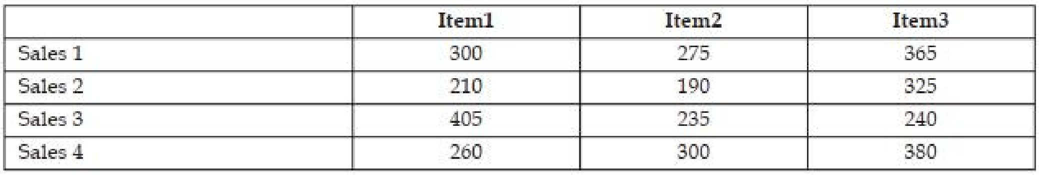


As C performs no bounds checking, care should be taken to ensure that the array indices are within the declared limits. Also, indexing in C begins from 0 and not from 1.

2. Two-dimensional and Multi-dimensional Array

It is possible to have an array of more than one dimensions. Two dimensional array (2-D array) is an array of number of 1-dimensional arrays.

A two dimensional array is also called a matrix. Consider the following table:



	Item1	Item2	Item3
Sales 1	300	275	365
Sales 2	210	190	325
Sales 3	405	235	240
Sales 4	260	300	380

This is a table of four rows and three columns. Such a table of items can be defined using two dimensional arrays.

General form of declaring a 2-D array is

data_typearray_name [row_size] [colum_size];

Example: int marks [4] [2];

5.4 <u>Operations on Array</u>

1. Traversing an array
2. Inserting an element in an array
3. Searching an element in an array
4. Deleting an element from an array
5. Merging two arrays

1. Traversing an Array

Traversing an array means accessing each and all elements of the array for a particular purpose. Traversing the data elements of an array can include print every element, calculating the total number of elements, or conducting any process on these elements.

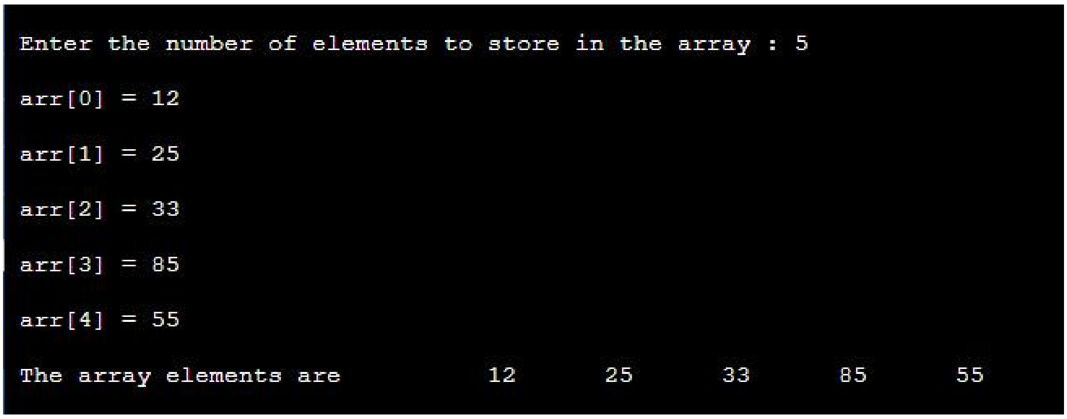
Lab Exercise

Program to read and display numbers:

```
#include <stdio.h>

int main()

{

int i, n, arr[20];

printf("\n Enter the number of elements to store in the array : ");

scanf("%d", & n);

for(i=0;i<n;i++)

{
```

```
printf("\n arr[%d] = ", i);
scanf("%d",&arr[i]);
}
printf("\n The array elements are ");
for(i=0;i<n;i++)
printf("\t %d", arr[i]);
return 0;
}
```

Output



```
Enter the number of elements to store in the array : 5

arr[0] = 12

arr[1] = 25

arr[2] = 33

arr[3] = 85

arr[4] = 55

The array elements are          12       25       33       85       55
```

2. Inserting an element in an array

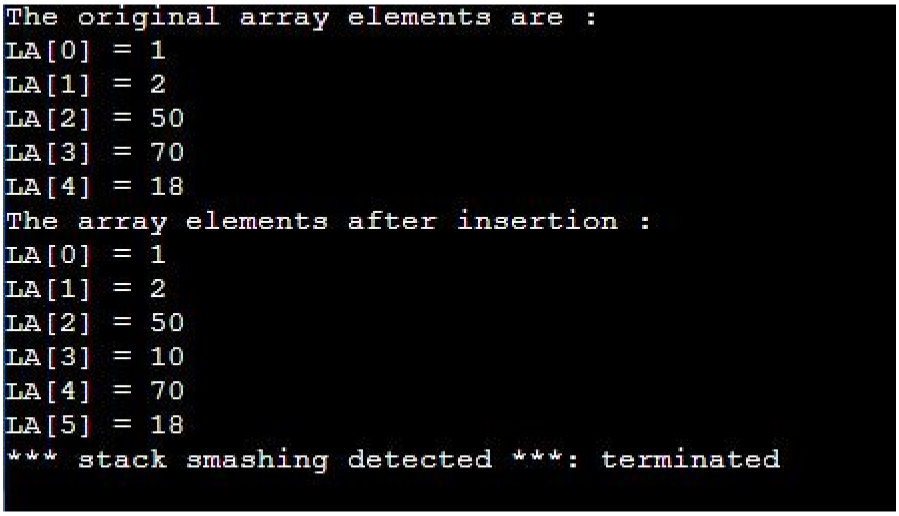
If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple. We just have to add 1 to the upper_bound and assign the value. Algorithm to insert an element in the middle of an array.

Lab Exercise

Program to insert element in between of the array

```
#include <stdio.h>

#include<stdio.h>

int main() {
  int LA[] = {1,2,50,70,18};
  int item = 10, k = 3, n = 5;
  int i = 0, j = n;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }

  n = n + 1;
while( j>= k) {
    LA[j+1] = LA[j];
    j = j - 1;
  }

  LA[k] = item;

printf("The array elements after insertion :\n");

for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
  return 0;
}
```

Output



```
The original array elements are :
LA[0] = 1
LA[1] = 2
LA[2] = 50
LA[3] = 70
LA[4] = 18
The array elements after insertion :
LA[0] = 1
LA[1] = 2
LA[2] = 50
LA[3] = 10
LA[4] = 70
LA[5] = 18
*** stack smashing detected ***: terminated
```

Task

1. Make an application that perform all the operations on array.
2. Write a program to perform deletion on an array.

3. Searching an element in an array

Searching refers to find desired key element from an array.

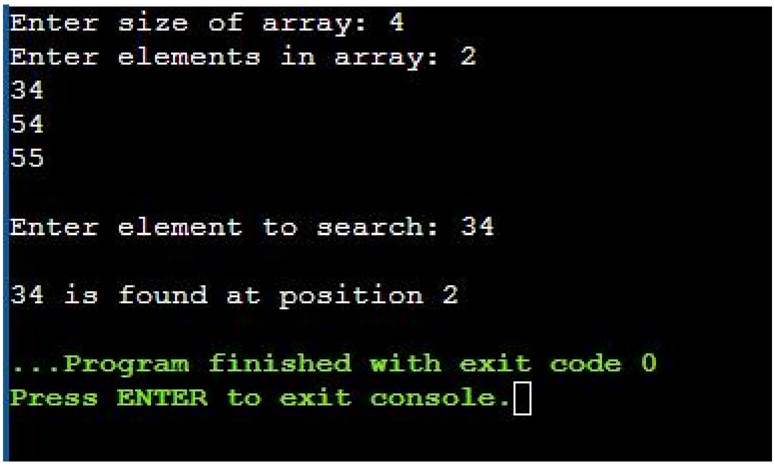
Lab Exercise

Program to search an element from array

```
#include <stdio.h>
#define MAX_SIZE 100  // Maximum array size
```

```
int main()
{
    int arr[MAX_SIZE];
    int size, i, toSearch, found;
printf("Enter size of array: ");
scanf("%d", &size);
printf("Enter elements in array: ");
for(i=0; i<size; i++)
    {
scanf("%d", &arr[i]);
    }
printf("\nEnter element to search: ");
scanf("%d", &toSearch);
    found = 0;
for(i=0; i<size; i++)
    {

        if(arr[i] == toSearch)
        {
            found = 1;
break;
        }
    }

if(found == 1)
    {
printf("\n%d is found at position %d", toSearch, i + 1);
    }
    else
    {
printf("\n%d is not found in the array", toSearch);
    }

    return 0;
}
```

Output



```
Enter size of array: 4
Enter elements in array: 2
34
54
55

Enter element to search: 34

34 is found at position 2

...Program finished with exit code 0
Press ENTER to exit console.
```

4. Deleting an element from an array

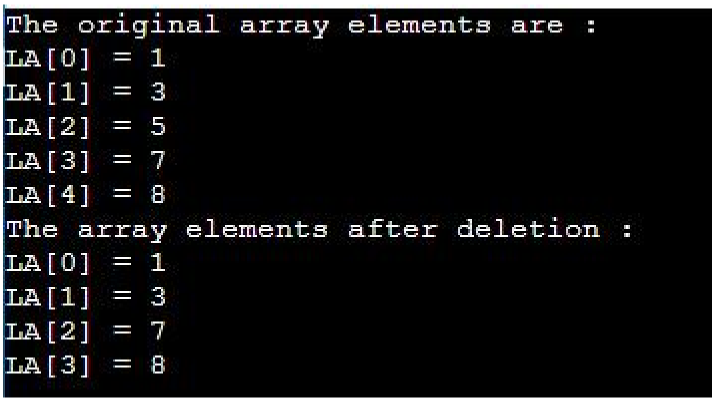
Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Lab Exercise

Program to delete an element from array

```
#include <stdio.h>
int main() {
  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5;
  int i, j;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }

  j = k;

while( j< n) {
    LA[j-1] = LA[j];
    j = j + 1;
  }

  n = n -1;

printf("The array elements after deletion :\n");
for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
  return 0;
}
```

Output



```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

5. Merging two arrays

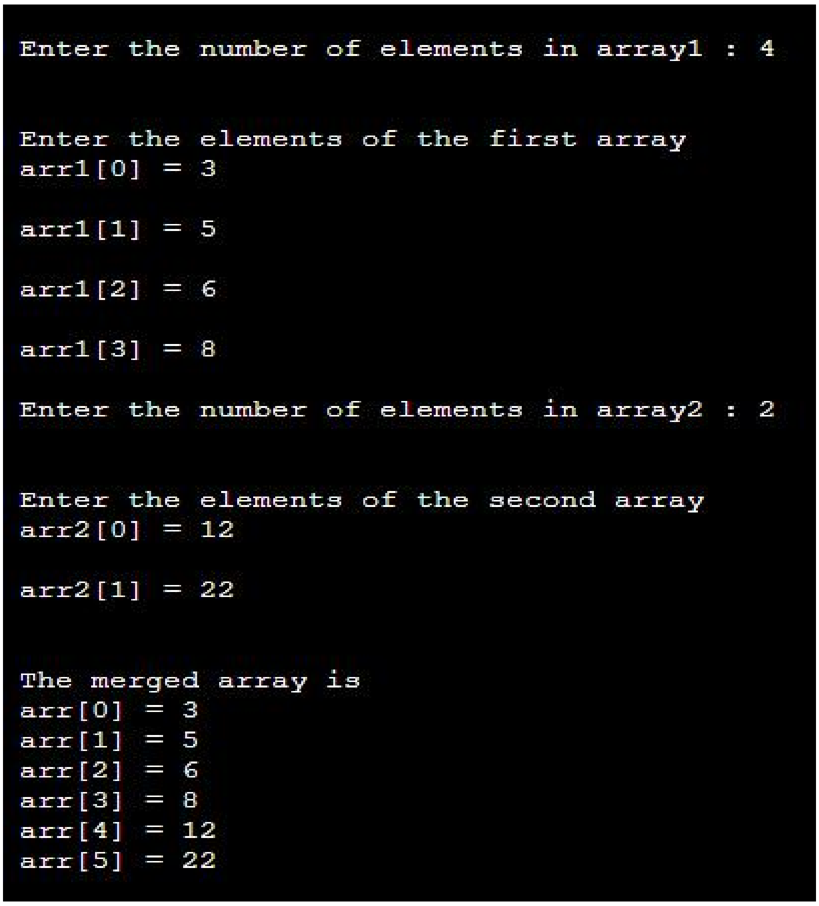
Merging two arrays in a third array means first copying the elements of the first array into the third array and then copying the elements of the second array in the third array. Therefore, the merged array contains the elements of the first array followed by the elements of the second array.

Lab Exercise

Program to merge two arrays

```
#include <stdio.h>

int main()
{
int arr1[10], arr2[10], arr3[20];
int i, n1, n2, m, index=0;

printf("\n Enter the number of elements in array1 : ");
scanf("%d", & n1);
printf("\n\n Enter the elements of the first array");
for(i=0;i<n1;i++)
{
printf("\n arr1[%d] = ", i);
scanf("%d", & arr1[i]);
}
printf("\n Enter the number of elements in array2 : ");
scanf("%d", & n2);
printf("\n\n Enter the elements of the second array");
for(i=0;i<n2;i++)
{
printf("\n arr2[%d] = ", i);
scanf("%d", & arr2[i]);
}
m = n1+n2;
for(i=0;i<n1;i++)
{
arr3[index] = arr1[i];
index++;
}
for(i=0;i<n2;i++)
{
arr3[index] = arr2[i];
index++;
}
printf("\n\n The merged array is");
for(i=0;i<m;i++)
printf("\n arr[%d] = %d", i, arr3[i]);
return 0;
}
```

Output



```
Enter the number of elements in array1 : 4


Enter the elements of the first array
arr1[0] = 3

arr1[1] = 5

arr1[2] = 6

arr1[3] = 8

Enter the number of elements in array2 : 2


Enter the elements of the second array
arr2[0] = 12

arr2[1] = 22


The merged array is
arr[0] = 3
arr[1] = 5
arr[2] = 6
arr[3] = 8
arr[4] = 12
arr[5] = 22
```

Summary

- An array is a group of memory locations related by the fact that they all have the same name and same data type.
- An array including more than one dimension is called a multidimensional array.
- The size of an array should be a positive number. If an array in declared without a size and in initialized to a series of values it is implicitly given the size of number of initializers.
- Array subscript always starts with 0. Last element's subscript is always one less than the size of the array e.g., an array with 10 elements contains element 0 to 9. Size of an array must be a constant number.

Keywords

- **Array**: A user defined simple data structure which represents a group of same type of variables having same name each being referred to by an integral index
- **Multidimensional array**: An array in which elements are accessed using multiple indices
- **One dimensional array**: An array in which elements are accessed using a single index
- **Subscript/Index**: The integral index by which an array element is accessed
- **Two dimensional array**: An array in which elements are accessed using two indices.

SelfAssessment

1. Choose the correct statement

A. Search in array is delete an element from array
B. Search in array is insert an element in array
C. Search in array is find an element from array
D. None of above

2. Choose the correct statement

A. Search in array is delete an element from array
B. Search in array is insert an element in array
C. Search in array is update an element in array
D. None of above

3. Searching is a process in which we find element in array

A. True
B. False

4. What are the disadvantages of arrays?

A. Data structure like queue or stack cannot be implemented
B. There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size
C. Index value of an array can be negative
D. Elements are sequentially accessed

5. Traversal is process of visit each element of an array

A. True

B. False

6. A ___ is required to perform traversal of an array

A. Loop

B. Switch Statement

C. Constant

D. All of above

7. Insertion in array insert a new element into array

A. True

B. False

8. Choose the correct statement

A. We can't perform insertion on array

B. We can't perform deletion on array

C. We can't traverse an array elements

D. None of above

9. Choose the right statement

A. Insert an element at beginning is possible

B. Insert an element at end is possible

C. Insert an element at given location is possible

D. All of above

10. Deletion of an element from the array reduces the size of the array by________.

A. one

B. Two

C. Three

D. Four

11. Deletion of an element from the array is

A. Remove value from array

B. Insert Value in array

C. Merge an array

D. All of above

12. After performing deletion on array, its required to re-organizing all elements of array

A. True

B. False

13. Which of the following is allowed in case of arrays

A. Insertion in Array

B. Deletion in Array

C. Concatenate two Arrays

D. All of above

14. To merge two arrays we need at least three array variables

A. True

B. False

15. To perform merge operation on array, minimum ____ arrays required

A. 2

B. 1

C. -2

D. 0

Answers for SelfAssessment

1.	D	2.	C	3.	A	4.	B	5.	A
6.	A	7.	A	8.	D	9.	D	10.	A
11.	A	12.	A	13.	S	14.	A	15.	A

Review Questions

1. Write a program that perform insertion on an array.
2. Discuss in detail operations of an array.
3. What do you mean by 'Array'? How it can be declared & initialized in a C program?
4. Draw a diagram to represent the internal storage of an Array.
5. Describe the different types of Array. Give suitable programs.
6. Delete the element from an array.
7. If an array arr contains n elements, then write a program to check if arr[0] = arr[n-1], arr[1] = arr[n-2] and so on.
8. Write a program to merge different arrays.
9. Write a program to pick up the largest number from any 5 row by 5 column matrix.
10. Write a program that interchanges the odd and even components of an array.

Further Readings

Books Ashok N. Kamthane, "Programming with ANCI & Turbo C", Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web Links

https://www.cs.uic.edu/~jbell/CourseNotes/C_Programming/Arrays.html

https://www.javatpoint.com/c-array

Unit 06: Linked Lists

CONTENTS

Objectives

After studying this unit, you will be able to:

- Dynamic Memory Management
- Understand the basics of linked list
- Discuss the operations of linked lists

Introduction

A data structure consists of a group of data elements bound by the same set of rules. The data elements also known as members are of different types and lengths. We can manipulate data stored in the memory with the help of data structures. The study of data structures involves examining the merging of simple structures to form composite structures and accessing definite components from composite structures. An array is an example of one such composite data structure that is derived from a primitive data structure.

An array is a set of similar data elements grouped together. Arrays can be one-dimensional or multidimensional. Arrays store the entries sequentially. Elements in an array are stored in continuous locations and are identified using the location of the first element of the array.

6.1 Linked list

Linked lists are the most common data structures. They are referred to as an array of connected objects where data is stored in the pointer fields. Linked lists are useful when the number of elements to be stored in a list is indefinite.

Concept of Linked Lists

An array is represented in memory using sequential mapping, which has the property that elements are fixed distance apart. But this has the following disadvantage. It makes insertion or deletion at any arbitrary position in an array a costly operation, because this involves the movement of some of the existing elements.

When we want to represent several lists by using arrays of varying size, either we have to represent each list using a separate array of maximum size or we have to represent each of the lists using one single array. The first one will lead to wastage of storage, and the second will involve a lot of data movement.

So, we have to use an alternative representation to overcome these disadvantages. One alternative is a linked representation. In a linked representation, it is not necessary that the elements be at a fixed distance apart. Instead, we can place elements anywhere in memory, but to make it a part of the same list, an element is required to be linked with a previous element of the list. This can be done by storing the address of the next element in the previous element itself. This requires that every element be capable of holding the data as well as the address of the next element. Thus, every element must be a structure with a minimum of two fields, one for holding the data value, which we call a data field, and the other for holding the address of the next element, which we call link field.

Therefore, a linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.

This program uses a strategy of inserting a node in an existing list to get the list created. An insert function is used for this. The insert function takes a pointer to an existing list as the first parameter, and a data value with which the new node is to be created as a second parameter, creates a new node by using the data value, appends it to the end of the list, and returns a pointer to the first node of the list. Initially the list is empty, so the pointer to the starting node is NULL.

Therefore, when insert is called first time, the new node created by the insert becomes the start node. Subsequently, the insert traverses the list to get the pointer to the last node of the existing list, and puts the address of the newly created node in the link field of the last node, thereby appending the new node to the existing list. The main function reads the value of the number of nodes in the list. Calls iterate that many times by going in a while loop to create the links with the specified number of nodes.

6.2 Dynamic Memory Allocation

There are several limitations in static memory allocation:

This is done in RAM dedicated solely to a programme, which is frequently limited in capacity. The size of a static array is fixed. We won't be able to expand it to accommodate situations that require more elements. As a result, we'll likely to declare larger arrays than necessary, resulting in memory waste. We also can't lower array size to conserve memory when fewer array elements are necessary. Advanced data structures such as linked lists, trees, and graphs, which are needed in most real-world programming situations, are not possible (or efficient) to develop.

C has a feature called dynamic allocation that is quite unique (amongst high level languages). It allows us to design data types and structures of any size and duration to meet the needs of our programmes. Dynamic memory allocation occurs when memory is allocated at runtime, that is, when a programme is running. Pointers and four common library functions are used in dynamic memory management.

Programming language (C) provides several functions for memory allocation and management, namely, malloc, calloc, realloc and free.Functions are defined in the <stdlib.h> header file.

Dynamic Memory Management Functions

Function	Typical call	Description
malloc	malloc (sz)	Allocate a block of size *sz* bytes from memory heap and return a pointer to the allocated block e.g., ptr = (cast.type*) malloc (byte_size);
calloc	calloc *in (sz)*	Allocate a block of size *n* x *sz* bytes from memory heap, initialize it to zero and return a pointer to the allocated block e.g., ptr = (cast_type*) calloc (n, elem_size);
realloc	realloc *(bl,; sz)*	Adjust the size of the memory block *blk* allocated on the heap to *sz,* copy the contents to a new location if necessary and return a pointer to the allocated block e.g., ptr = realloc (ptr, newsize);
free	free *(blk)*	Free block of memory *blk* allocated from memory heap e.g., free (ptr);

Advantages of Dynamic Memory allocation

- When we don't know how much memory will be required for the software ahead of time.
- When we need data structures that don't have a memory restriction.
- When you wish to make better use of your memory space.
- For example, if you allocate memory space for a 1D array like array[20] and only use 10 memory spaces, the remaining 10 memory spaces will be squandered, and this wasted memory will be unavailable to other programme variables.
- Insertions and deletions in dynamically constructed lists may be done quickly and easily by manipulating addresses, whereas insertions and deletions in statically allocated memory result in additional movements and memory waste.
- Dynamic memory allocation is required when using the concepts of structures and linked lists in programming.

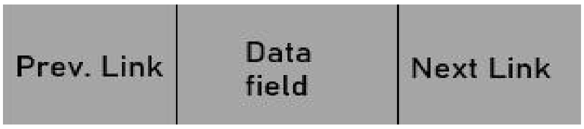
6.3 Types of linked list

1. Single linked list
2. Double linked list
3. Circular linked list

A doubly-linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links that are references to the previous and to the next node in the sequence of nodes.

In the single linked list each node provides information about where the next node is in the list. It faces difficulty if we are pointing to a specific node, then we can move only in the direction of the links. It has no idea about where the previous node lies in memory. The only way to find the node which precedes that specific node is to start back at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a single linked list. Since in order to easily delete an arbitrary node one must know the preceding node. This problem can be avoided by using Doubly Linked List, we can store in each node not only the address of next node but also the address of the previous node in the linked list. A node in Doubly Linked List has three fields

1. Data
2. Previous Link
3. Next Link



Implementation of Doubly Linked List

Structure of a node of Doubly Linked List can be defined as:

```
struct node
{
int data;
struct node *llink;
struct node *rlink;
}
```

Circular Linked List

Circular Linked List is another remedy for the drawbacks of the Single Linked List besides Doubly Linked List. A slight change to the structure of a linear list is made to convert it to circular linked list; link field in the last node contains a pointer back to the first node rather than a Null.

6.4 Representation of Linked List

Because each node of an element contains two parts, we have to represent each node through a structure.

While defining linked list we must have recursive definitions:

```
struct node
{
int data;
struct node * link;
}
```

Here, link is a pointer of struct node type i.e. it can hold the address of variable of struct node type. Pointers permit the referencing of structures in a uniform way, regardless of the organization of the structure being referenced. Pointers are capable of representing a much more complex relationship between elements of a structure than a linear order.

Initialization:

```
main()
{
struct node *p, *list, *temp;
list = p = temp = NULL;
.
.
.
}
```

Example:

Program:

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p, int n)
{
struct node *temp;
/* if the existing list is empty then insert a new node as the
starting node */
if(p==NULL)
{
p=(struct node *)malloc(sizeof(struct node)); /* creates new
node data value passes
as parameter */
if(p==NULL)
{
printf(“Error\n”);
exit(0);
}
p-> data = n;
p-> link = p; /* makes the pointer pointing to itself because
it is a circular list*/
}
else
{
temp = p;
/* traverses the existing list to get the pointer to the last node
of it */
while (temp->link != p)
temp = temp->link;
temp-> link = (struct node *)malloc(sizeof(struct node)); /*
creates new node using
data value passes as
parameter and puts its
address in the link field
```

```
of last node of the
existing list*/
if(temp -> link == NULL)
{
printf("Error\n");
exit(0);
}
temp = temp->link;
temp-> data = n;
temp-> link = p;
}
return (p);
}
void printlist( struct node *p )
{
struct node *temp;
temp = p;
printf("The data values in the list are\n");
if(p!= NULL)
{
do
{
printf("%d\t",temp->data);
temp=temp->link;
} while (temp!= p);
}
else
printf("The list is empty\n");
}
void main()
{
int n;
int x;
struct node *start = NULL ;
printf("Enter the nodes to be created \n");
scanf("%d",&n);
while ( n -- > 0 )
{
printf( "Enter the data values to be placed in a node\n");
scanf("%d",&x);
```

```
start = insert ( start, x );
}
printf("The created list is\n");
printlist( start );
}
```

6.5 Deleting the Specified Node in Singly Linked List

To delete a node, first we determine the node number to be deleted (this is based on the assumption that the nodes of the list are numbered serially from 1 to n). The list is then traversed to get a pointer to the node whose number is given, as well as a pointer to a node that appears before the node to be deleted. Then the link field of the node that appears before the node to be deleted is made to point to the node that appears after the node to be deleted, and the node to be deleted is freed.

Lab Exercise:

```
# include <stdio.h>
# include <stdlib.h>
struct node *delet( struct node *, int );
int length ( struct node * );
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *p, int n)
{
struct node *temp;
if(p==NULL)
{
p=(struct node *)malloc(sizeof(struct node));
if(p==NULL)
{
printf("Error\n");
exit(0);
}
p-> data = n;
p-> link = NULL;
}
else
{
temp = p;
while (temp->link != NULL)
temp = temp->link;
temp-> link = (struct node *)malloc(sizeof(struct node));
if(temp -> link == NULL)
{
printf(“Error\n”);
exit(0);
}
temp = temp->link;
temp-> data = n;
temp-> link = NULL;
}
return (p);
}
void printlist( struct node *p )
{
printf(“The data values in the list are\n”);
while (p!= NULL)
{
printf(“%d\t”,p-> data);
p = p->link;
}
}
void main()
{
int n;
int x;
struct node *start = NULL;
printf(“Enter the nodes to be created \n”);
scanf(“%d”,&n);
while ( n- > 0 )
{
printf( “Enter the data values to be placed in a node\n”);
scanf(“%d”,&x);
start = insert ( start, x );
}
printf(“ The list before deletion id\n”);
printlist( start );
printf(“% \n Enter the node no \n”);
scanf( “ %d”,&n);
start = delet (start , n );
```

```
printf(“ The list after deletion is\n”);
printlist( start );
}
/* a function to delete the specified node*/
struct node *delet( struct node *p, int node_no )
{
struct node *prev, *curr ;
int i;
if (p == NULL )
{
printf(“There is no node to be deleted \n”);
}
else
{
if ( node_no> length (p))
{
printf(“Error\n”);
}
else
{
prev = NULL;
curr = p;
i = 1 ;
while ( i<node_no )
{
prev = curr;
curr = curr->link;
i = i+1;
}
if ( prev == NULL )
{
p = curr ->link;
free ( curr );
}
else
{
prev -> link = curr ->link ;
free ( curr );
}
}
}
return(p);
}
/* a function to compute the length of a linked list */
int length ( struct node *p )
{
int count = 0 ;
while ( p != NULL )
{
count++;
p = p->link;
}
return ( count ) ;
}
```

6.6 Inserting a Node after the Specified Node in a Singly Linked List

To insert a new node after the specified node, first we get the number of the node in an existinglist after which the new node is to be inserted. This is based on the assumption that the nodes ofthe list are numbered serially from 1 to n. The list is then traversed to get a pointer to the node,whose number is given. If this pointer is x, then the link field of the new node is made to point tothe node pointed to by x, and the link field of the node pointed to by x is made to point to the newnode. Figures 2.3 and 2.4 show the list before and after the insertion of the node, respectively.

Insertion in Linked List can happen at following places:

At the beginning of the linked list.

At the end of the linked list.

At a given position in the linked list.

Algorithm: Insertion at beginning

```
Step 1: IF PTR = NULL
Write OVERFLOW
    Go to Step 7
   [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET NEW_NODE → NEXT = HEAD
Step 6: SET HEAD = NEW_NODE
Step 7: EXIT
```

Deletion from a Linked List

Delete from beginning

Delete from end

Delete from middle/ given position

Find the previous node of the node to be deleted.

Change the next pointer of the previous node

Free the memory of the deleted node.

In case of first node deletion, we need to update the head of the linked list.

Algorithm: Deletion at beginning

```
Step 1: IF HEAD = NULL
Write UNDERFLOW
    Go to Step 5
   [END OF IF]
Step 2: SET PTR = HEAD
Step 3: SET HEAD = HEAD -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Searching in linked list

Searching is performed to find the location of a particular element in the list. Traversing is performed in the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm: Searching in linked list

```
Step 1: SET PTR = HEAD
Step 2: Set I = 0
STEP 3: IF PTR = NULL
  WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF
STEP 4: REPEAT STEP 5 TO 7 UNTIL PTR != NULL
STEP 5: if ptr → data = item
  write i+1
 End of IF
STEP 6: I = I + 1
STEP 7: PTR = PTR → NEXT
[END OF LOOP]
STEP 8: EXIT
```

6.7 Linked List Common Errors

Here is summary of common errors of linked lists. Read these carefully, and read them againwhen you have problem that you need to solve.

1. Allocating a new node to step through the linked list; only a pointer variable is needed.

2. Confusing the and the -> operators.

3. Not setting the pointer from the last node to 0 (null).

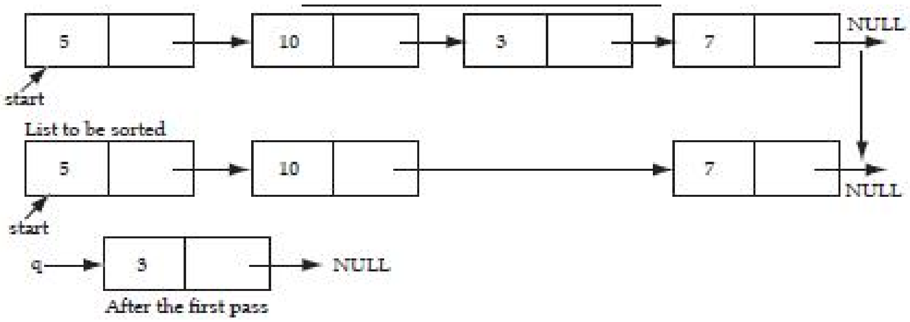
4. Not considering special cases of inserting/removing at the beginning or the end of thelinked list.

5. Applying the delete operator to a node (calling the operator on a pointer to the node)before it is removed. Delete should be done after all pointer manipulations are completed.

6. Pointer manipulations that are out of order. These can ruin the structure of the linked list.

Sorting and Reversing a Linked List

To sort a linked list, first we traverse the list searching for the node with a minimum data value.Then we remove that node and append it to another list which is initially empty. We repeat thisprocess with the remaining list until the list becomes empty, and at the end, we return a pointerto the beginning of the list to which all the nodes are moved.



Sorting of linked list

To reverse a list, we maintain a pointer each to the previous and the next node, then we make thelink field of the current node point to the previous, make the previous equal to the current, and thecurrent equal to the next.

6.8 Arrays vs. Linked list

Array	Linked list
Data elements are stored in contiguous locations in memory.	New elements can be stored anywhere and a reference is created for the new element using pointers.
Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	Insertion and Deletion operations are fast and easy in a linked list.
Memory is allocated during the **compile time** (*Static memory allocation*).	Memory is allocated during the **run-time** (*Dynamic memory allocation*).
Size of the array must be specified at the time of array declaration/initialization.	Size of a Linked list grows/shrinks as and when new elements are inserted/deleted.

Summary

- An array is a set of same data elements grouped together. Arrays can be one-dimensional ormultidimensional.
- A linear or one-dimensional array is a structured collection of elements (often called as array elements) that are accessed individually by specifying the position of each element with a single index value.
- Multidimensional arrays are nothing but "arrays of arrays". Two subscripts are used to refer to the elements.
- The operations that are performed on an array are adding, sorting, searching, and traversing.
- Traversing an array refers to moving in inward and outward direction to access each element in an array.
- Linked list is a technique of dynamically implementing a list using pointers. A linked list contains two fields namely, data field and link field.
- A singly-linked list consists of only one pointer to point to another node and the last node always points to NULL to indicate the end of the list.
- A doubly-linked list consists of two pointers, one to point to the next node and the other to point to the previous node.
- In a circular singly-linked list, the last node always points to the first node to indicate the circular nature of the list.
- A circular doubly-linked list consists of two pointers for forward and backward traversal and the last node points to the first node.
- Searching operation involves searching for a specific element in the list using an associated key.
- Insertion operation involves inserting a node at the beginning or end of a list.
- Deletion operation involves deleting a node at the beginning or following a given node or at the end of a list.

Keywords

Non-linear Data Structure: Every data item is attached to several other data items in a way thatis specific for reflecting relationships. The data items are not arranged in a sequential structure.

Searching: Finding the location of the record with a given key value, or finding the locations ofall records, which satisfy one or more conditions.

Traversing: Accessing each record exactly once so that certain items in the record may beprocessed.

Circular Linked List: A linear linked list in which the last element points to the first element, thus,forming a circle.

Doubly Linked List: A linear linked list in which each element is connected to the two nearestelements through pointers.

SelfAssessment

1. A linear collection of data elements where the linear node is given by means of pointer is called?

A. Linked list

B. Node list

C. Primitive list

D. None of these

2. Which of the following are type of linked list

A. Single Linked List

B. Circular Linked List

C. Double Linked List

D. All of above

3. Linked list is considered as an example of ___________ type of memory allocation.

A. Static

B. Compile time

C. Heap

D. Dynamic

4. Among 4 header files, which should be included to use the memory allocation functions like malloc(), calloc(), realloc() and free()?

A. #include<string.h>

B. #include<stdlib.h>

C. #include<memory.h>

D. All of above

5. DMA stands for

A. Dynamite Memory Access

B. Dynamic Memory Available

C. Direct Memory Access

D. None of Above

6. Dynamic memory allocation is

A. More efficient

B. Less efficient

C. Don't know

D. None of above

7. Which function is used to delete the allocated memory space?

a) Dealloc()

b) free()

c) Both A and B

d) None of the above

8. Which of the following advantages of linked list over arrays?

A. Dynamic Size

B. Ease of insertion and deletion

C. All of above

D. None of above

9. Linked list can be represented in the memory using
A. One array
B. Two arrays
C. Six arrays
D. None of above

10. Which of the following operations is performed for visit nodes in linked list?
A. Deletion
B. User Define Function
C. Traversing
D. None of above

11. Which of the following operations is performed for visit nodes in linked list?
A. Deletion
B. User Define Function
C. Traversing
D. None of above

12. Searching in a linked list is
A. Find element in linked list
B. Find element in linked list and move element to another location
C. Find element in linked list, if match found then the address of the node is returned otherwise we process the next node
D. None of above

13. Insertion into linked list is process of
A. Create a new linked list
B. Insert a new node to linked list
C. All of above
D. None of above

14. Insertion in a linked list can be done at
A. The beginning of linked list
B. The end of linked list
C. A particular position in linked list
D. All of above

15. Deleting a node at the beginning is
A. Delete the first node of linked list
B. Delete all nodes of liked list
C. All of above
D. None of above

16. Which of the following operation remove node from linked list
A. Traversing
B. Inversing
C. Deletion
D. Insertion

Answers for Self Assessment

1.	A	2.	D	3.	D	4.	B	5.	C
6.	A	7.	B	8.	C	9.	B	10.	C

11. C 12. C 13. B 14. D 15. A

16. C

Review Questions

1. Define array and its types.
2. Give an example of multidimensional array.
3. Discuss any two types of array initialization methods with example.
4. Discuss different sorting methods.
5. Write a program to sort the elements of a linked list.
6. Differentiate between array and linked list with suitable example.
7. Discuss different operation performed with linked list.
8. Discuss advantages of linked list as compared to arrays.

Further Readings

Books Ashok N. Kamthane, "Programming with ANCI & Turbo C", Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web Links

https://www.cs.uic.edu/~jbell/CourseNotes/C_Programming/Arrays.html

https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm

Unit 07: Doubly Linked Lists

CONTENTS

Objectives

After studying this unit, you will be able to:

- Understand the circular linked list
- Discuss the operations of doubly linked lists
- Discuss the operations ofcircular linked lists

Introduction

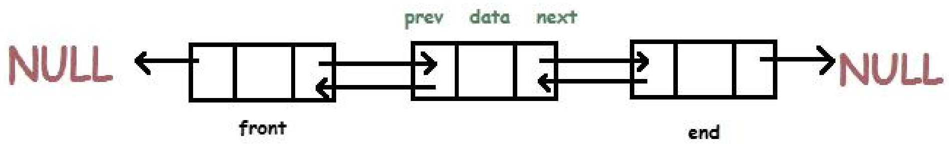
Data Structures are the programmatic way of storing data so that data can be used efficiently. Almost every enterprise application uses various types of data structures in one or the other way. This tutorial will give you a great understanding on Data Structures needed to understand the complexity of enterprise level applications and need of algorithms, and data structures.

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward or backward easily as compared to Single Linked List.

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

7.1 Doubly Linked List

Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The first node of the list has its previous link pointing to NULL similarly the last node of the list has its next node pointing to NULL.

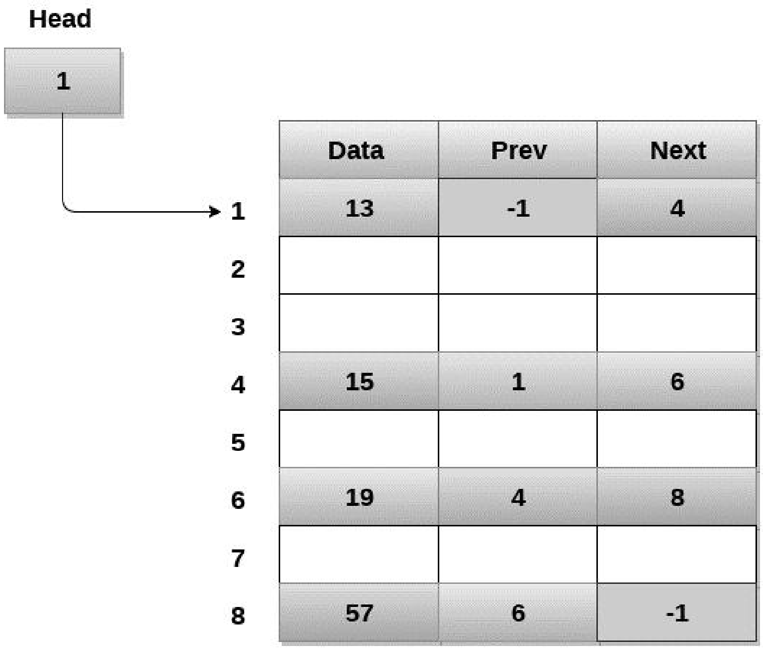


Memory Representation of a doubly linked list

Memory The graphic below is a representation of a doubly linked list. In general, a doubly linked list takes up more space for each node, making fundamental operations like insertion and deletion take longer. However, because the list keeps pointers in both directions, we can simply change the list's elements (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the prev of the list contains null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

Advantages and Disadvantages of Doubly Linked List over Singly Linked List

- Traversal can be done in both directions (from the start node to the end node as well as from the end node to the start node) in a Doubly Linked list. But this is not possible in a Singly Linked List and it can only be traversed only in one direction.
- Deletion and insertion operations are easy to implement in a Doubly LL than a Singly LL. For example, in a singly linked list, to delete a node, the pointer to the previous node is needed for which the list is to be traversed. In a Doubly LL, we just need to know the pointer of the node to be deleted.

- Memory has to be allocated for both the next and previous pointers in a node. Hence, the occupation of memory is higher in Doubly LL.
- Both the pointers will have to be modified if any kind of operation is performed like insertion, deletion, etc in case of Doubly LL.

Algorithm to create a Doubly linked list

```
Algorithm to create Doubly Linked list
Begin:
alloc (head)
  If (head == NULL) then
    write ('Unable to allocate memory')
  End if
  Else then
    read (data)
head.data ← data;
head.prev ← NULL;
head.next ← NULL;
    last ← head;
    write ('List created successfully')
  End else
End
```

Did you know?

The singly linked list had a single pointer pointing to the next node. But a doubly linked list contains two pointers. One pointer points to the next node and one pointer to the previous node. Thus, a doubly linked list is a two-way chain.

Every node in a doubly linked list has-

- Data
- Address of the next node
- Address of the previous node

Basic Operations on Double Linked List

The basic set of operations that we can perform on a doubly linked list are:

1. Traverse forward: It means visiting each node from the beginning till the end with the help of the 'Next' pointer.

2. Traverse backwards: This operation is performed to visit each node but in the reverse direction. It will traverse the list from the end to the beginning.

3. Insertion: This operation inserts an element at any given position in the list.

4. Deletion: Deletion operation helps in deleting a node from the linked list.

5. Display forward: This operation is used to print/display all the data elements of the linked list from beginning till the end.

6. Display backwards: it will display the elements from the end to the beginning.

7. Search: Search operation helps in searching an element by traversing it.

7.2 Traversal in a Doubly Linked List

Traversal refers to linearly visiting each node. In a doubly linked list, we can traverse in both forward and backward directions.

Following program has traverse function that used to visit each node in a program.

Lab Exercise

```
//Program
#include<stdio.h>
#include<stdlib.h>
void create(int);
int traverse();
struct node
{
   int data;
   struct node *next;
   struct node *prev;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
printf("1.Insert In Doubly Linked List\n2.Traverse Doubly Linked List\n3.Exit\n4.Enter your choice?");
scanf("%d",&choice);
      switch(choice)
      {
         case 1:
printf("\nEnter the item\n");
scanf("%d",&item);
         create(item);
break;
         case 2:
traverse();
break;
         case 3:
exit(0);
break;
        default:
printf("\nPlease enter valid choice\n");
     }

}while(choice != 3);
}
void create(int item)
{

  struct node *ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
  {
printf("\nOVERFLOW\n");
  }
  else
  {
  if(head==NULL)
  {
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
     head=ptr;
  }
  else
  {
ptr->data=item;printf("\nPress 0 to insert more ?\n");
ptr->prev=NULL;
ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
printf("\nNode Inserted\n");
}
}
int traverse()
{
   struct node *ptr;
if(head == NULL)
   {
```

```
printf("\nEmpty List\n");
   }
   else
   {
ptr = head;
while(ptr != NULL)
      {
printf("%d\n",ptr->data);
ptr=ptr->next;
      }
   }
}
```

7.3 Insertion in a Linked List

Insertion in a linked list occurs at three different positions:

1. Insertion at the beginning of the list

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

Allocate the space for the new node in the memory. This will be done by using the following statement.

ptr = (struct node *)malloc(sizeof(struct node));

Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
      head=ptr;
```

In the second scenario, the condition head == NULL become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

This will be done by using the following statements.

```
ptr->next = head;
head→prev=ptr;
```

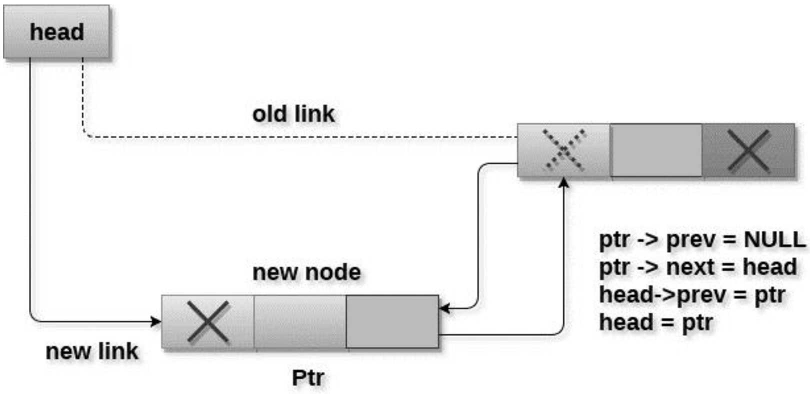
Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

ptr→prev =NULL

head = ptr

Algorithm:

```
Step 1: IF ptr = NULL
  Write OVERFLOW
 Go to Step 9
 [END OF IF]
Step 2: SET NEW_NODE = ptr
Step 3: SET ptr = ptr -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET head -> PREV = NEW_NODE
Step 8: SET head = NEW_NODE
Step 9: EXIT
```



Insertion into doubly linked list at beginning

Lab Exercise

```
#include<stdio.h>
#include<stdlib.h>
void insertbeginning(int);
struct node
{
    int data;
    struct node *next;
    struct node *prev;
```

```
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
printf("\nEnter the item which you want to insert?\n");
scanf("%d",&item);
insertbeginning(item);
printf("\nPress 0 to insert more ?\n");
scanf("%d",&choice);
}while(choice == 0);
}
void insertbeginning(int item)
{

  struct node *ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
  {
printf("\nOVERFLOW");
  }
  else
  {


  if(head==NULL)
  {
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
     head=ptr;
  }
  else
  {
ptr->data=item;
ptr->prev=NULL;
ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
}


}
```

2. Insertion after a particular node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Use the following steps for this purpose.

Allocate the memory for the new node. Use the following statements for this.

```
ptr = (struct node *)malloc(sizeof(struct node));
```

Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.

```
temp=head;
   for(i=0;i<loc;i++)
  {
      temp = temp->next;
if(temp == NULL)
      {
           Return 0;
      }
  }
```

The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node therefore we need to make a fer pointer adjustments here. Make the next pointer of ptr point to the next node of temp.

```
ptr → next = temp → next;
```

make the prev of the new node ptr point to temp.

```
ptr → prev = temp;
```

make the next pointer of temp point to the new node ptr.

```
temp → next = ptr;
```

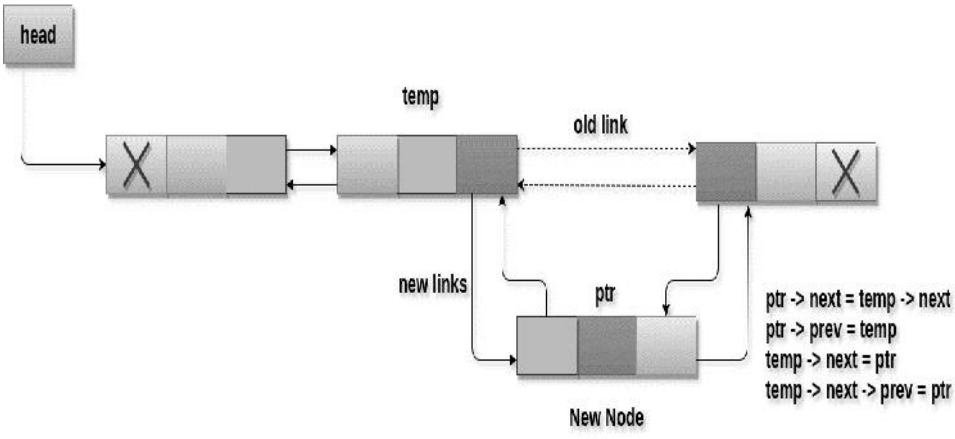
make the previous pointer of the next node of temp point to the new node.

```
temp → next → prev = ptr;
```

Algorithm

```
Step 1: IF PTR = NULL
   Write OVERFLOW
   Go to Step 15
 [END OF IF]
```

```
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET TEMP = START
Step 6: SET I = 0
Step 7: REPEAT 8 to 10 until I
Step 8: SET TEMP = TEMP -> NEXT
STEP 9: IF TEMP = NULL
STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
   GOTO STEP 15
   [END OF IF]
 [END OF LOOP]

Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT
Step 12: SET NEW_NODE -> PREV = TEMP
Step 13 : SET TEMP -> NEXT = NEW_NODE
Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE
Step 15: EXIT
```



Insertion into doubly linked list after specified node

Lab Exercise

```
#include<stdio.h>
#include<stdlib.h>
void insert_specified(int);
void create(int);
struct node
{
   int data;
```

```
    struct node *next;
    struct node *prev;
};
struct node *head;
void main ()
{
    int choice,item,loc;
    do
    {
printf("\nEnter the item which you want to insert?\n");
scanf("%d",&item);
if(head == NULL)
        {
            create(item);
        }
        else
        {
insert_specified(item);
        }
printf("\nPress 0 to insert more ?\n");
scanf("%d",&choice);
}while(choice == 0);
}
void create(int item)
    {
   struct node *ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
   {
printf("\nOVERFLOW");
   }
   else
   {



   if(head==NULL)
   {
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
       head=ptr;
```

```
  }
  else
  {
ptr->data=item;printf("\nPress 0 to insert more ?\n");
ptr->prev=NULL;
ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
printf("\nNode Inserted\n");
}

}
void insert_specified(int item)
{

  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  struct node *temp;
  int i, loc;
if(ptr == NULL)
  {
printf("\n OVERFLOW");
  }
  else
  {
printf("\nEnter the location\n");
scanf("%d",&loc);
     temp=head;
     for(i=0;i<loc;i++)
     {
        temp = temp->next;
if(temp == NULL)
        {
printf("\ncan't insert\n");
return;
        }
     }
ptr->data = item;
ptr->next = temp->next;
ptr ->prev = temp;
```

```
        temp->next = ptr;
        temp->next->prev=ptr;
printf("Node Inserted\n");
   }
}
```

3. Insertion at the end of the list

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

Allocate the memory for the new node. Make the pointer ptr point to the new node being inserted.

```
ptr = (struct node *) malloc(sizeof(struct node));
```

Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
   head=ptr;
```

In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer temp to head and traverse the list by using this pointer.

```
Temp = head;
while (temp != NULL)
{
temp = temp → next;
   }
```

the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e.ptr.

```
temp→next =ptr;
```

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

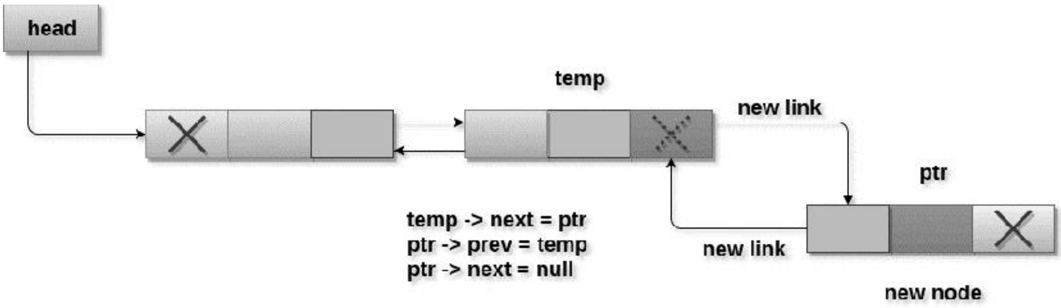
```
ptr → prev = temp;
```

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

```
ptr → next = NULL
```

Algorithm

```
Step 1: IF PTR = NULL
 Write OVERFLOW
  Go to Step 11
 [END OF IF]

Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET TEMP = START
Step 7: Repeat Step 8 while TEMP ->NEXT != NULL
Step 8: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE
Step 10C: SET NEW_NODE -> PREV = TEMP
Step 11: EXIT
```



Insertion into doubly linked list at the end

Lab Exercise

```
#include<stdio.h>
#include<stdlib.h>
void insertlast(int);
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};
```

```
struct node *head;
void main ()
{
  int choice,item;
  do
  {
printf("\nEnter the item which you want to insert?\n");
scanf("%d",&item);
insertlast(item);
printf("\nPress 0 to insert more ?\n");
scanf("%d",&choice);
}while(choice == 0);
}
void insertlast(int item)
{
    struct node *ptr = (struct node *) malloc(sizeof(struct node));
  struct node *temp;
if(ptr == NULL)
  {
printf("\nOVERFLOW");

  }
  else
  {

ptr->data=item;
if(head == NULL)
    {
ptr->next = NULL;
ptr->prev = NULL;
      head = ptr;
    }
    else
    {
     temp = head;
     while(temp->next!=NULL)
     {
       temp = temp->next;
     }
     temp->next = ptr;
ptr ->prev=temp;
ptr->next = NULL;
      }
printf("\nNode Inserted\n");


  }
}
```

7.4 Deletion from Doubly Linked Lists

Deletion in a linked list occurs at three different positions:

1. Deletion at beginning

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

```
Ptr = head;
   head = head → next;
```

now make the prev of this new head node point to NULL. This will be done by using the following statements.

```
head → prev = NULL
```

Now free the pointer ptr by using the free function.

```
free(ptr)
```

Algorithm

```
STEP 1: IF HEAD = NULL
WRITE UNDERFLOW
GOTO STEP 6
STEP 2: SET PTR = HEAD
STEP 3: SET HEAD = HEAD → NEXT
STEP 4: SET HEAD → PREV = NULL
STEP 5: FREE PTR
STEP 6: EXIT
```

2. Deletion at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation cannot be carried on.

- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
    if(ptr->next != NULL)
    {
ptr = ptr ->next;
    }
.
```

- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of ptr to NULL.

  ptr → prev → next = NULL

  free the pointer as this the node which is to be deleted.

  free(ptr)

Algorithm

```
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 7
[END OF IF]

Step 2: SET TEMP = HEAD
Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL
Step 4: SET TEMP = TEMP->NEXT
[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL
Step 6: FREE TEMP
```

3. Deletion of the node having given data

In order to delete the node after the specified data, we need to perform the following steps.

Copy the head pointer into a temporary pointer temp.

temp = head

Traverse the list until we find the desired data value.

while(temp -> data != val)

```
temp = temp ->next;
```

Check if this is the last node of the list. If it is so then we can't perform deletion.

```
if(temp -> next == NULL)
   {
return;
   }
```

Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
   }
```

Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

```
ptr = temp ->next;
      temp -> next = ptr ->next;
ptr -> next ->prev = temp;
      free(ptr);
```

Algorithm

```
Step 1: IF HEAD = NULL
   Write UNDERFLOW
  Go to Step 9
 [END OF IF]

Step 2: SET TEMP = HEAD
Step 3: Repeat Step 4 while TEMP ->DATA != ITEM
Step 4: SET TEMP = TEMP -> NEXT
  [END OF LOOP]

Step 5: SET PTR = TEMP -> NEXT
Step 6: SET TEMP -> NEXT = PTR -> NEXT
Step 7: SET PTR -> NEXT -> PREV = TEMP
Step 8: FREE PTR
Step 9: EXIT
```

Lab Exercise

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void delete_specified();
struct node
{
   int data;
   struct node *next;
   struct node *prev;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
printf("1.Append List\n2.Delete node\n3.Exit\n4.Enter your choice?");
scanf("%d",&choice);
      switch(choice)
      {
         case 1:
printf("\nEnter the item\n");
scanf("%d",&item);
         create(item);
break;
         case 2:
delete_specified();
break;
         case 3:
exit(0);
break;
         default:
printf("\nPlease enter valid choice\n");
      }

}while(choice != 3);
}
void create(int item)
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
  {
printf("\nOVERFLOW\n");
```

```
 }
 else
 {

      if(head==NULL)
 {
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
    head=ptr;
 }
 else
 {
ptr->data=item;
ptr->prev=NULL;
ptr->next = head;
    head->prev=ptr;
    head=ptr;
 }
printf("\nNode Inserted\n");
}

}
void delete_specified( )
{
  struct node *ptr, *temp;
  int val;
printf("Enter the value");
scanf("%d",&val);
  temp = head;
while(temp -> data != val)
  temp = temp ->next;
if(temp -> next == NULL)
  {
printf("\nCan't delete\n");
  }
  else if(temp -> next -> next == NULL)
  {
    temp ->next = NULL;
printf("\nNode Deleted\n");
```

```
    }
    else
    {
ptr = temp ->next;
        temp -> next = ptr ->next;
ptr -> next ->prev = temp;
        free(ptr);
printf("\nNode Deleted\n");
    }
}
```

7.5 Singly Linked List Vs Doubly Linked List

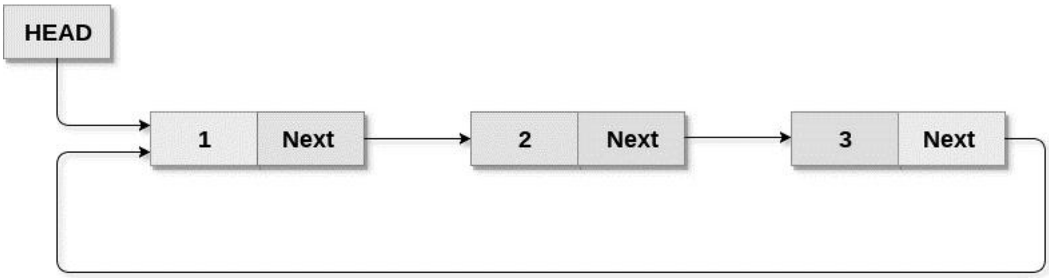
Singly Linked List	Doubly Linked List
Each node consists of a data value and a pointer to the next node.	Each node consists of a data value, a pointer to the next node, and a pointer to the previous node.
Traversal can occur in one way only (forward direction).	Traversal can occur in both ways.
It requires less space.	It requires more space because of an extra pointer.
It can be implemented on the stack.	It has multiple usages. It can be implemented on the stack, heap, and binary tree.

7.6 Circular Linked List

In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Following operations performed on circular linked list

Lab Exercise

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
while(choice != 7)
   {
printf("\n*********Main Menu*********\n");
printf("\nChoose one option from the following list ...\n");
      printf("\n==============================================================\n");
printf("\n1.Insert in begining\n2.Insert at last\n3.Delete from Beginning\n4.Delete from
last\n5.Search for an element\n6.Show\n7.Exit\n");
printf("\nEnter your choice?\n");
scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
beginsert();
break;
         case 2:
lastinsert();
break;
```

```
      case 3:
begin_delete();
break;
      case 4:
last_delete();
break;
      case 5:
search();
break;
      case 6:
display();
break;
      case 7:
exit(0);
break;
      default:
printf("Please enter valid choice..");
    }
  }
}
void beginsert()
{
  struct node *ptr,*temp;
  int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
  {
printf("\nOVERFLOW");
  }
  else
  {
printf("\nEnter the node data?");
scanf("%d",&item);
ptr -> data = item;
if(head == NULL)
    {
      head = ptr;
ptr -> next = head;
    }
    else
    {
      temp = head;
      while(temp->next != head)
        temp = temp->next;
ptr->next = head;
      temp -> next = ptr;
      head = ptr;
    }
printf("\nnode inserted\n");
  }

}
void lastinsert()
{
  struct node *ptr,*temp;
  int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
  {
printf("\nOVERFLOW\n");
  }
  else
  {
printf("\nEnter Data?");
scanf("%d",&item);
ptr->data = item;
if(head == NULL)
    {
      head = ptr;
ptr -> next = head;
    }
    else
    {
      temp = head;
while(temp -> next != head)
      {
        temp = temp ->next;
      }
      temp -> next = ptr;
ptr -> next = head;
```

```
        }

printf("\nnode inserted\n");
    }

}

void begin_delete()
{
    struct node *ptr;
if(head == NULL)
    {
printf("\nUNDERFLOW");
    }
    else if(head->next == head)
    {
        head = NULL;
        free(head);
printf("\nnode deleted\n");
    }

    else
{  ptr = head;
while(ptr -> next != head)
ptr = ptr ->next;
ptr->next = head->next;
        free(head);
        head = ptr->next;
printf("\nnode deleted\n");

    }
}
void last_delete()
{
    struct node *ptr, *preptr;
    if(head==NULL)
    {
printf("\nUNDERFLOW");
    }
    else if (head ->next == head)
    {
        head = NULL;
        free(head);
printf("\nnode deleted\n");

    }
    else
    {
ptr = head;
while(ptr ->next != head)
        {
preptr=ptr;
ptr = ptr->next;
        }
preptr->next = ptr ->next;
        free(ptr);
printf("\nnode deleted\n");

    }
}

void search()
{
    struct node *ptr;
    int item,i=0,flag=1;
ptr = head;
if(ptr == NULL)
    {
printf("\nEmpty List\n");
    }
    else
    {
printf("\nEnter item which you want to search?\n");
scanf("%d",&item);
if(head ->data == item)
        {
printf("item found at location %d",i+1);
        flag=0;
        }
        else
    {
    while (ptr->next != head)
    {
      if(ptr->data == item)
      {
printf("item found at location %d ",i+1);
        flag=0;
break;
      }
      else
      {
        flag=1;
      }
i++;
ptr = ptr ->next;
    }
    }
if(flag != 0)
    {
printf("Item not found\n");
    }
  }

}

void display()
{
  struct node *ptr;
ptr=head;
if(head == NULL)
  {
printf("\nnothing to print");
  }
  else
  {
printf("\n printing values ... \n");

while(ptr -> next != head)
    {
```

```
printf("%d\n", ptr -> data);
ptr = ptr ->next;
       }
printf("%d\n", ptr -> data);
   }


}
```

Summary

- Linked list is a technique of dynamically implementing a list using pointers. A linked list contains two fields namely, data field and link field.
- A singly-linked list consists of only one pointer to point to another node and the last node always points to NULL to indicate the end of the list.
- A doubly-linked list consists of two pointers, one to point to the next node and the other to point to the previous node.
- In a circular singly-linked list, the last node always points to the first node to indicate the circular nature of the list.
- A circular doubly-linked list consists of two pointers for forward and backward traversal and the last node points to the first node.
- Searching operation involves searching for a specific element in the list using an associated key.
- Insertion operation involves inserting a node at the beginning or end of a list.
- Deletion operation involves deleting a node at the beginning or following a given node or at the end of a list.

Keywords

Non-linear Data Structure: Every data item is attached to several other data items in a way thatis specific for reflecting relationships. The data items are not arranged in a sequential structure.

Searching: Finding the location of the record with a given key value, or finding the locations ofall records, which satisfy one or more conditions.

Traversing: Accessing each record exactly once so that certain items in the record may beprocessed.

Circular Linked List: A linear linked list in which the last element points to the fi rst element, thus,forming a circle.

Doubly Linked List: A linear linked list in which each element is connected to the two nearestelements through pointers.

SelfAssessment

1. Which of the following statements about a doubly linked list is not correct?

A. We can navigate in both the directions
B. It requires more space than a singly linked list
C. The insertion and deletion of a node take a bit longer
D. None of above

2. What is a memory efficient double linked list?
A. Each node has only one pointer to traverse the list back and forth
B. The list has breakpoints for faster traversal
C. An auxiliary singly linked list acts as a helper list to traverse through the doubly linked list
D. A doubly linked list that uses bitwise AND operator for storing addresses

3. Traversing doubly linked list refer to visit each element of list.
A. True
B. False

4. In doubly linked lists, traversal can be performed?
A. Only in forward direction
B. Only in reverse direction
C. In both directions
D. None of the above

5. What is the worst case time complexity of inserting a node in a doubly linked list?
A. O(nlogn)
B. O(logn)
C. O(n)
D. O(1)

6. What is the functionality of following code

```
public class insertFront(int data)

{

    Node node = new Node(data, head, head.getNext());

    node.getNext().setPrev(node);

    head.setNext(node);

    size++;

}
```

A. Insert a node at the beginning of the list
B. Delete Node
C. Traverse List
D. None of above

7. Insertion into doubly linked list is
A. Insert new node
B. Insert USB stick into CPU
C. All of above
D. None of above

8. Which of the following operations does a doubly linked list execute more efficiently than a singly linked list?
A. Deleting a node whose location in given
B. Searching of an unsorted list for a given item
C. Inverting a node after the node with given location
D. Traversing a list to process each node

9. Deletion at the beginning in the doubly linked list is possible.
A. True
B. False

10. Which of the following basic operation of doubly linked list is responsible for delete element from list?
A. Insert a node
B. Insert node at end
C. Delete a node
D. None of above

11. Deletion at the given position in the doubly linked list is possible.
A. True
B. False

12. Is there a linked list version in which the list's last node points to the list's beginning node?
A. Singly linked list
B. Doubly linked list
C. Circular linked list
D. Multiply linked list

13. In circular linked list, insertion of node requires modification of?
A. One pointer
B. Two pointer
C. Three pointer
D. Requires no modification

14. A variant of the linked list in which none of the node contains NULL pointer is?
A. Singly linked list
B. Doubly linked list
C. Circular linked list
D. None of the above

15. Which of the following are disadvantages of circular linked list?
A. Depending on the implementation, inserting at start of the list would require doing a search for last node which could be expensive.
B. Finding the end of the list and loop control is harder (no NULL's to mark the beginning and end).
C. All of above
D. None of above

Answers for Self Assessment

1.	D	2.	A	3.	A	4.	C	5.	C
6.	A	7.	A	8.	A	9.	A	10.	C
11.	A	12.	C	13.	B	14.	C	15.	C

Review Questions

1. Define circular linked list.
2. Give an example of doubly linked list.

3. Discuss any two operations on doubly linked list.
4. Discuss different delete operation of doubly linked.
5. Write a program to traverse doubly linked list.
6. Differentiate between singly linked list and doubly linked list.
7. Write a program to delete element from a doubly linked list.

Further Readings

Books Ashok N. Kamthane, "Programming with ANCI & Turbo C", Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web Links

https://www.studytonight.com/data-structures/doubly-linked-list

https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm

Unit 08: Introduction to Stacks

CONTENTS

Objectives

After studying this unit, you will be able to:

- Learn fundamentals of stacks
- Explain the basic operations of stack
- Explain the implementation and applications of stacks

Introduction

Stacks are simple data structures and an important tool in programming language. Stacks are linear lists which have restrictions on the insertion and deletion operations. These are special cases of ordered list in which insertion and deletion is done only at the ends.

The basic operations performed on stack are push and pop. Stack implementation can be done in two ways - static implementation or dynamic implementation. Stack can be represented in the memory using a one-dimensional array or a singly linked list.

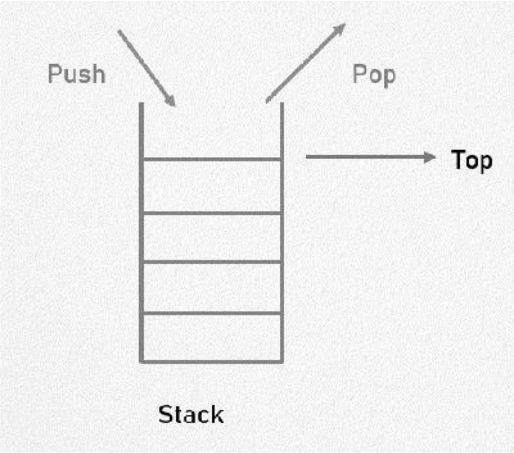
Stack is another linear data structure having a very interesting property. Unlike arrays and link lists, an element can be inserted and deleted not at any arbitrary position but only at one end. Thus, one end of a stack is sealed for insertion and deletion while the other end allows both the operations.

8.1 Stack Structure

The stack data structure is used to maintain records of a file in which the order among the records of file is not important. Figure 7.1 displays the structure of a stack where stack is like a hollow cylinder with a closed bottom end and an open top end. In the stack data structure, the records are added and deleted at the top end. Last-In-First-Out (LIFO) principle is followed to retrieve records from the stack. The records added last are accessed first.

A stack is a linear data structure in as much as its member elements are ordered as 1st, 2nd,…. and last. However, an element can be inserted in and deleted from only one end. The other end remains sealed. This open end to which elements can be inserted and deleted from is called stack top or top of the stack. Consequently, the elements are removed from a stack in the reverse order of insertion.

A stack is said to possess LIFO (Last In First Out) property. A data structure has LIFO property if the element that can be retrieved first is the one that was inserted last.



Basic Operations of Stack

The basic operations of stack are to:

1. Insert an element in the stack (Push operation)

2. Delete an element from the stack (Pop operation)

Push Operation

The procedure to insert a new element to the stack is called push operation. The push operation adds an element on the top of the stack. 'Top' refers to the element on the top of stack. Push makes the 'Top' point to the recently added element on the stack. After every push operation, the 'Top' is incremented by one. When the array is full, the status of stack is FULL and the condition is called stack overflow. No element can be inserted when the stack is full

Algorithm to Implement Push Operation on Stack

```
PUSH (STACK, n, top, item) /* n = size of stack*/
if (top = n) then STACK_FULL; /* checks for stack overflow */
else
{ top = top+1; /* increases the top by 1 */
STACK [top] = item ;} /* inserts item in the new top position */
end PUSH
```

Pop Operation

The procedure to delete an element from the top of the stack is called pop operation. After every pop operation, the 'Top' is decremented by one. When there is no element in the stack, the status of the stack is called empty stack or stack underflow. The pop operation cannot be performed when it is in stack underflow condition.

Algorithm to Implement Pop Operation in a Stack

```
POP (STACK, top, item)
if (top = 0) then STACK_EMPTY; /* check for stack underflow*/
else { item = STACK [top]; /* remove top element*/
top = top – 1; /* decrement stack top*/
}
end POP
```

8.2 Implementation of Stacks

There are two basic methods for the implementation of stacks – one where the memory is used statically and the other where the memory is used dynamically.

Array-based Implementation

A stack is a sequence of data elements. To implement a stack structure, an array can be used as it is a storage structure. Each element of the stack occupies one array element. Static implementation of stack can be achieved using arrays. The size of the array, once declared, cannot be changed during the program execution. Memory is allocated according to the array size. The memory requirement is determined before the compilation. The compiler provides the required memory. This is suitable when the exact number of elements is known. The static allocation is an inefficient memory allocation technique because if fewer elements are stored than declared, the memory is wasted and if more elements need to be stored than declared, the array cannot expand. In both the cases, there is inefficient use of memory.

The following pseudo-code shows the array-based implementation of a stack. In this, the elements of the stack are of type T.

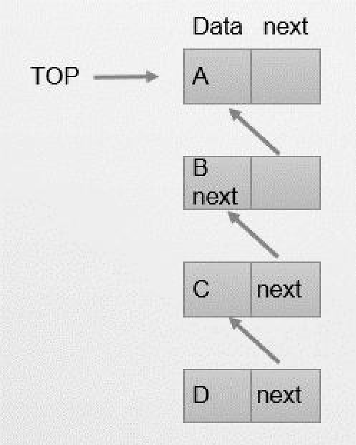
```
struct stk
{ T array[max_size];
/* max_size is the maximum size */
int top = -1;
/* stack top initially given value -1 */
} stack;
void push(T e)
/*inserts an element e into the stack s*/
{
if (stack.top == max_size)
printf("Stack is full-insertion not possible");
else
{
stack.top = stack.top + 1;
stack.array[stack.top] = e;
}
}
T pop()
/*Returns the top element from the stack */
{
T x;
if(stack.top == -1)
printf("Stack is empty");
else
{
x = stack.array[stack.top];
stack.top = stack.top - l;
return(x);
}
}
booleanempty()
/* checks if the stack is empty * /
{
boolean empty = false;
if(stack.top == -1)
empty = true else empty = false;
return(empty);
}
void initialise()
/* This procedure initializes the stack s * /
{
stack.top = -1;
}
```

The above implementation strategy is easy and fast since it does not have run-time overheads. At the same time it is not flexible since it cannot handle a situation when the number of elements exceeds max_size. Also, let us say, if max_size is derided statically to 100 and a stack actually has only 10 elements, then memory space for the rest of the 90 elements would be wasted.

Linked List Representation of Stacks

The array representation of stacks is easy and convenient. However, it allows the representation of only fixed sized stacks. The size of the stack varies during program application for different applications. Representing stack using linked list can solve this problem. A singly linked list can be used to represent any stack. In a singly linked list, the data field represents the ITEM and the LINK field points to the next item.

In linked list implementation of the stack, we need to create nodes and the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Here the memory is used dynamically. For every push operation, the memory space for one element is allocated at run-time and the element is inserted into the stack. For every pop operation, the memory space for the deleted element is de-allocated and returned to the free space pool. Hence the shortcomings of the array-based implementation are overcome. But since, this allocates memory dynamically, the execution is slowed down.

The following pseudo-code is for the pointer-based implementation of a stack. Each element of the stack is of type T.

```
struct stk
{
T element;
struct stk *next;
};
struct stk *stack;
void push(struct stk *p, T e)
{
struct stk *x;
x = new(stk);
x.element = e;
x.next = NULL;
p = x;
}
```

Here the stack full condition is checked by the call to new which would give an error if no memory space could be allocated.

```
T pop(struct stk *p)
{
struct stk *x;
if (p == NULL)
printf("Stack is empty");
else
{ x = p;
x = x.next;
return(p.element);
}
booleanempty(sstructstk *p)
{
if (p == NULL)
return(true);
else
return(false);
}
void initialize(struct stk *p)
{
p = NULL;
}
```

8.3 Applications of Stacks

There are numerous applications of the stack data structure in computer algorithms. It is used to store return information in the case of function/procedure/subroutine calls. Hence, one would find a stack in architecture of any Central Processing Unit (CPU). In this section, we would just illustrate a few of them.

Expression Evaluation and Conversion

Parenthesis Checking

Backtracking

Function Call

String Reversal

Memory Management

Syntax Parsing

Parenthesis checker

Parenthesis checker is used for balanced Brackets in an expression. The balanced parenthesis means that when the opening parenthesis is equal to the closing parenthesis, then it is a balanced parenthesis.

(a+b*(c/d))

[10+20*(6+7)]

(x+y)/(c-d)

Balanced parenthesis

A = (50+25)

In the above expression there is one opening and one closing parenthesis means that both opening and closing brackets are equal; therefore, the above expression is a balanced parenthesis.

Unbalanced parenthesis

A= [(15+25)

The above expression has two opening brackets and one closing bracket, which means that both opening and closing brackets are not equal; therefore, the above expression is unbalanced.

Algorithm

- Initialize a character stack.
- Now traverse the expression string exp.
- If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.
- If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then balanced else brackets are not balanced.
- After complete traversal, if there is some starting bracket left in stack then not balanced

Expression conversion and evaluation

Arithmetic expressions can be represented in 3 forms:

- Infix notation
- Postfix notation (Reverse Polish Notation)
- Prefix notation (Polish Notation)

Infix Notation

Infix Notation can be represented as:

operand1 operator operand1

Example: 15 + 26

a + b

Postfix Notation

Postfix Notation can be represented as

operand1 operand2 operator

Example: 15 29 +

a b +

Prefix notation

Prefix notation can be represented as

operator operand1 operand2

Example: + 10 20

+ a b

Infix notation is used most frequently in our day to day tasks. Machines find infix notations tougher to process than prefix/postfix notations. Hence, compilers convert infix notations to prefix/postfix before the expression is evaluated.

The precedence of operators needs to be taken care as per hierarchy

(^) > (*) > (/) > (+) > (-)

Brackets have the highest priority.

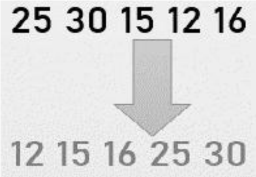
To evaluate an infix expression, We need to perform 2 steps:

- Convert infix to postfix
- Evaluate postfix

Task: Write a program that demonstrate working of PUSH and POP operation.

Sorting

A Sorting process is used to rearrange a given array or elements based upon selected algorithm/ sort function.



Quick Sort is used for sorting a list of data elements.Quicksort is a sorting algorithm based on the divide and conquer approach.An array is divided into subarrays by selecting a pivot element.During array dividing, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element

There are many different versions of quick Sort that pick pivot in different ways.

Always pick first element as pivot.

Always pick last element as pivot

Pick a random element as pivot.

Pick median as pivot.

Algorithm

```
quickSort(arr, beg, end)
  if (beg < end)
pivotIndex = partition(arr,beg, end)
quickSort(arr, beg, pivotIndex)
quickSort(arr, pivotIndex + 1, end)
partition(arr, beg, end)
  set end as pivotIndex
pIndex = beg - 1
  for i = beg to end-1
  if arr[i] < pivot
    swap arr[i] and arr[pIndex]
pIndex++
  swap pivot and arr[pIndex+1]
return pIndex + 1
```

8.4 Tower of Hanoi

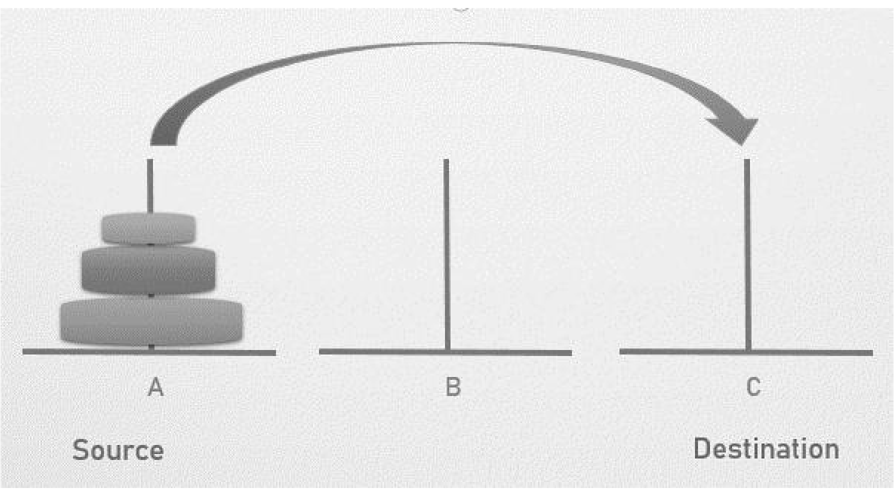
The Tower of Hanoi, is a mathematical problem which consists of three rods and multiple disks.Initially, all the disks are placed on one rod, one over the other in ascending order of size similar to a cone-shaped tower.

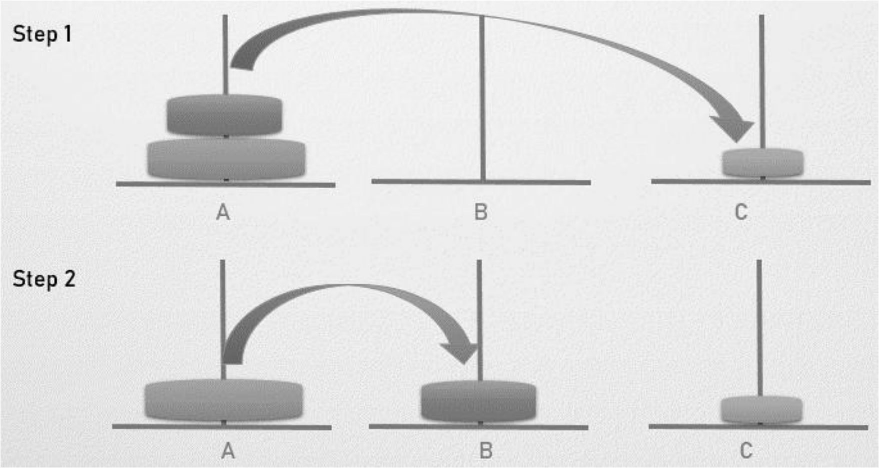
The objective of this problem is to move the stack of disks from the source to destination, following these rules:

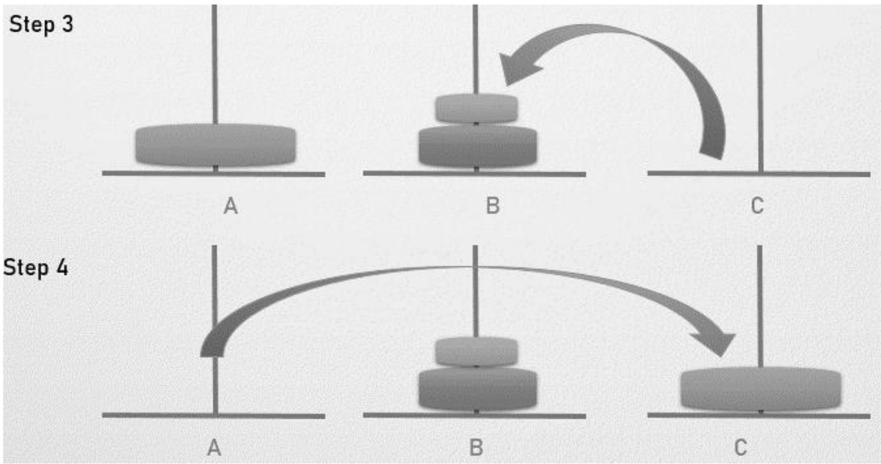
1. Only one disk can be moved at a time.

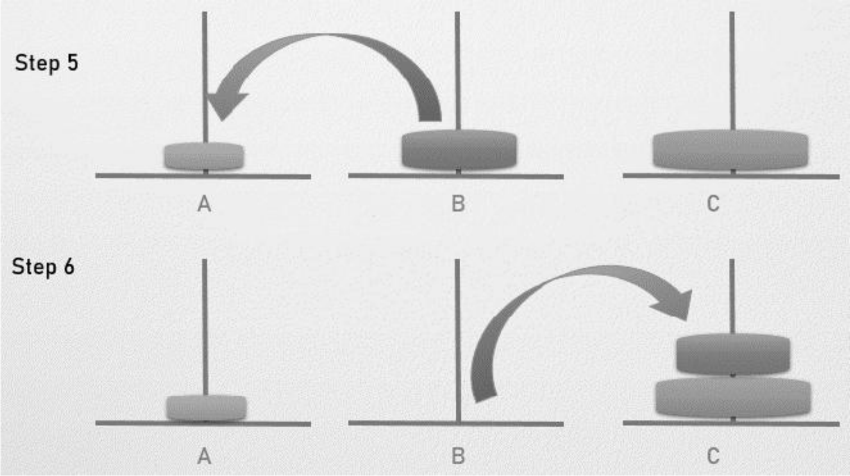
2. Only the top disk can be removed.

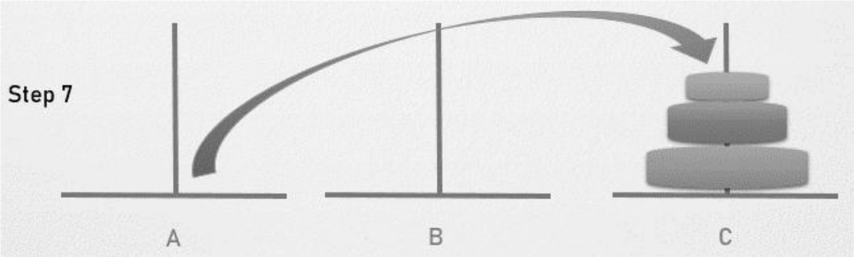
3. No large disk can sit over a small disk.


A
B
C
Source
Destination


Step 1
A
B
C
Step 2
A
B
C


Step 3
A
B
C
Step 4
A
B
C





Iterative Algorithm

1. At First Calculate the number of moves required i.e. "pow(2,n) - 1" where "n" is number of discs.
2. If the number of discs i.e n is even then swap Destination Rod and Auxiliary Rod.
3. for i = 1 upto number of moves:

    Check if "i mod 3" == 1:

    Perform Movement of top disc between Source Rod and Destination Rod.

    Check if "i mod 3" == 2:

    Perform Movement of top disc between Source Rod and Auxiliary Rod.

    Check if "i mod 3" == 0:

    Perform Movement of top disc between Auxiliary Rod and Destination Rod.

Simulating Recursive Function using Stack

A recursive solution to a problem is often more expensive than a non-recursive solution, both in terms of time and space. Frequently, this expense is a small price to pay for the logical simplicity and self-documentation of the recursive solution. However, in a production program (such as a compiler, for example) that may be run thousands of times, the recurrent expense is a heavy burden on the system's limited resources.

Thus, a program may be designed to incorporate a recursive solution in order to reduce the expense of design and certification, and then carefully converted to a non-recursive version to be put into actual day-to-day use. As we shall see, in performing such as conversion it is often possible to identify parts of the implementation of recursion that are superfluous in a particular application and thereby significantly reduce the amount of work that the program must perform.

Suppose that we have the statement

rout (x); where route is defined as a function by the header

rout(a); x is referred to as an argument (of the calling function), and a is referred to as a parameter(of the called function).

What happens when a function is called? The action of calling a function may be divided intothree parts:

1. Passing Arguments

2. Allocating and initializing local variables

3. Transferring control to the function.

1. ***Passing arguments***: For a parameter in C, a copy of the argument is made locally within the function, and any changes to the parameter are made to that local copy. The effect to this scheme is that the original input argument cannot be altered. In this method, storage for the argument is allocated within the data area of the function.

2. ***Allocating and initializing local variables***: After arguments have been passed, the local variables of the function are allocated. These local variables include all those declared directly in the function and any temporaries that must be created during the course of execution.

3. ***Transferring control to the function***: At this point control may still not be passed to the function because provision has not yet been made for saving the return address. If a function is given control, it must eventually restore control to the calling routine by means of a branch. However, it cannot execute that branch unless it knows the location to which it must return. Since this location is within the calling routine and not within the function, the only way that the function can know this address is to have it passed as an argument. This is exactly what happens. Aside from the explicit arguments specified by the programmer, there is also a set of implicit arguments that contain information necessary for the function to execute and return correctly. Chief among these implicit arguments is the return address. The function stores this address within its own data area. When it is ready to return control to the calling program, the function retrieves the return address and branches to that location. Once the arguments and the return address have been passed, control may be transferred to the function, since everything required has been done to ensure that the function can operate on the appropriate data and then return to the calling routine safely.

Task: Write a program to implement stack using linked list.

Summary

- A stack is a linear data structure in which allocation and deallocation are made in a last-in-first-out (LIFO) method.
- The basic operations of stack are inserting an element on the stack (push operation) and deleting an element from the stack (pop operation).
- Stacks are represented in main memory by using one-dimensional array or a singly linked list.
- To implement a stack structure, an array can be used as its storage structure. Each element of the stack occupies one array element. Static implementation of stack can be achieved using arrays.
- Stack is used to store return information in the case of function/procedure/subroutine calls. Hence, one would find a stack in architecture of any Central Processing Unit (CPU).
- In infix notation operators come in between the operands. An expression can be evaluated using stack data structure.

Keywords

LIFO: (Last In First Out) the property of a list such as stack in which the element which can be retrieved is the last element to enter it.

Pop: Stack operation retrieves a value form the stack.

Infix: Notation of an arithmetic expression in which operators come in between their operands.

Postfix: Notation of an arithmetic expression in which operators come after their operands.

Prefix: Notation of an arithmetic expression in which operators come before their operands.

Push: Stack operation which puts a value on the stack.

Stack: A linear data structure where insertion and deletion of elements can take place only at one end.

SelfAssessment

1. What type of data structure does a Stack is?

A. Linear
B. Non-linear
C. Both Linear and Non-Linear
D. None of above

2. It is impossible to do ___ operation on empty stack.

A. PUSH
B. POP
C. STATUS
D. None

3. Can we delete a node at front of a stack by using POP operation?

A. True
B. False

4. At which position in the stacks, the operations are being done.

A. TOP
B. SIZE
C. POP
D. PUSH

5. Other names for the insertion and deletion operations in Stacks?

A. PUSH – insertion, POP –Deletion.
B. PUSH – Deletion, POP – Insertion.
C. Both A and B are valid.
D. None.

6. Which of the following is an application of stack?

A. Finding factorial
B. tower of Hanoi
C. infix to postfix
D. all of the above

7. A pointer variable which contains the location at the top element of the stack is called …..

A. Top

B. Last

C. Final

D. End

8. ………. is the term used to delete an element from the stack.

A. Push

B. Pull

C. Pop

D. Pump

9. The elements are removal from a stack in ………. order.

A. Reverse

B. Hierarchical

C. Alternative

D. Sequential

10. In the linked representation of the stack ……… behaves as the top pointer variable of stack.

A. Stop pointer

B. Begin pointer

C. Start pointer

D. Avail pointer

11. Choose the correct statement

A. Linked list allocates the memory dynamically. However, the time complexity in both the scenario is the same for all the operations, i.e. push, pop and peek.

B. Array allocates the memory dynamically. However, the time complexity in both the scenario is the same for all the operations, i.e. push, pop and peek.

C. All of above

D. None of above

12. Stack can be implemented using

A. Array

B. Linked list

C. Both array and linked list

D. None of above

13. The representation of stack can be done in ____________.

A. One way

B. Two ways

C. Three

D. None

14. Select true statement for implementation of stack using array is

A. The stack is formed by using the array.

B. All the operations regarding the stack are performed using arrays.

C. All of the above

D. None of the above

15. Which of the following is true about linked list implementation of stack?

A. In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end.

B. In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from the beginning.

C. Both of the above

D. None of the above

Answers for Self Assessment

1.	A	2.	B	3.	A	4.	A	5.	A
6.	D	7.	A	8.	C	9.	A	10.	C
11.	A	12.	C	13.	B	14.	C	15.	D

Review Questions

1 What do you mean by stack? Explain different applications of stack.

2 What are the advantages of implementing a stack using dynamic memory allocation method?

3 Explain concept of tower of Hanoi.

4 what are the different methods for implementing stacks?

5 Give an example of push and pop operation using stack.

6 Write an algorithm to reverse an input string of characters using a stack.

Further Readings

Books Ashok N. Kamthane, “Programming with ANCI & Turbo C”, Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web Links

www.en.wikipedia.org

https://www.javatpoint.com/data-structure-stack

https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

Unit 09: Introduction to Queues

CONTENTS

Objectives

After studying this unit, you will be able to:

- Learn implementation of queues
- Explain priority queue
- Discuss applications of queues

Introduction

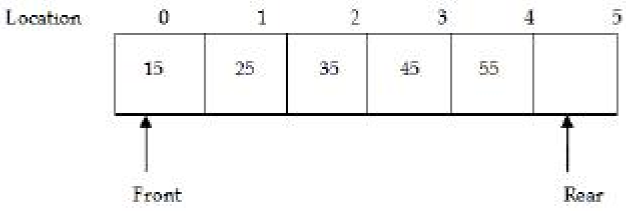
A queue is a linear list of elements that consists of two ends known as front and rear. We can delete elements from the front end and insert elements at the rear end of a queue. A queue in an application is used to maintain a list of items that are ordered not by their values but by their sequential value.

The queue abstract data type is also a widely used one with applications very common in real life. An example comes from the operating system software where the scheduler picks up the next process to be executed on the system from a queue data structure. In this unit, we would study the various properties of queues, their operations and implementation strategies.

9.1 Fundamentals of Queues

A queue is an ordered collection of items in which deletion takes place at one end, which is called the front of the queue, and insertion at the other end, which is called the rear of the queue. The queue is a 'First In First Out' system (FIFO). In a time-sharing system, there can be many tasks waiting in the queue, for access to disk storage or for using the CPU. The queues in a bank, or railway station counter are examples of queue. The first person in the queue is the first to be attended.

The two main operations in the queue are insertion and deletion of items. The queue has two pointers, the front pointer points to the first element of the queue and the rear pointer points to the last element of the queue.



Basic Operations of Queue

The basic operations of queue are insertion and deletion of items which are referred as enqueue and dequeue respectively. In enqueue operation, an item is added to the rear end of the queue. In dequeue operation, the item is deleted from the front end of the queue.

Insert at Rear End

To insert an item into the queue, first it should be verified whether the queue is full or not. If the queue is full, a new item cannot be inserted into the queue. The condition FRONT=NULL indicates that the queue is empty. If the queue is not full, items are inserted at the rear end. When an item is added to the queue, the value of rear is incremented by 1.

In queue we need to maintain two data pointers, front and rear. Operations on queue are comparatively difficult to implement than that of stacks

Step 1 − Check if the queue is full.

Step 2 − If the queue is full, produce overflow error and exit.

Step 3 − If the queue is not full, increment rear pointer to point the next empty space.

Step 4 − Add data element to the queue location, where the rear is pointing.

Step 5 − return success.

Algorithm: Enqueue operation

```
procedure enqueue(data)
    if queue is full
    return overflow
  endif
    rear ← rear + 1
  queue[rear] ← data
  return true
  end procedure
```

Delete from the Front End

To delete an item from the stack, first it should be verified that the queue is not empty. If the queue is not empty, the items are deleted at the front end of the queue. When an item is deleted from the queue, Dequeue operation include two tasks: access the data where front is pointing and remove the data after access.

Step 1 − Check if the queue is empty.

Step 2 − If the queue is empty, produce underflow error and exit.

Step 3 − If the queue is not empty, access the data where front is pointing.

Step 4 − Increment front pointer to point to the next available data element.

Step 5 − Return success. the value of the front is incremented by 1.

Algorithm: Dequeue operation

```
procedure dequeue
    if queue is empty
    return underflow
  end if
  data = queue[front]
  front ← front + 1
  return true
end procedure
```

Example:

```
/*Program of queue using array*/
/*insertion and deletion in a queue*/
/*insertion and deletion in a queue*/
# include <stdio.h>
# define MAX 50
int queue_arr[MAX];
int rear = -1;
int front = -1;
void ins_delete();
void insert();
void display();
void main()
{
int choice;
while(1)
{
printf("1.Insert\n");
printf("2.Delete\n");
printf("3.Display\n");
printf("4.Quit\n");
printf("Enter your choice : \n");
scanf("%d",&choice);
switch(choice)
{
case 1 : insert();
break;
case 2 :ins_delete();
break;
case 3: ins_display();
break;
case 4: exit(1);
default:
```

```
printf("Wrong choice\n");
}/*End of switch*/
}/*End of while*/
}/*End of main()*/
void insert()
{
int added_item;
if (rear==MAX-1)
printf("Queue overflow\n");
else
{
if (front==-1) /*If queue is initially empty */
front=0;
printf("Enter an element to add in the queue : ");
scanf("%d", &added_item);
rear=rear+1;
queue_arr[rear] = added_item ;
}
} /*End of insert()*/
void ins_delete()
{
if (front == -1 || front > rear)
{
printf("Queue underflow\n");
return ;
}
else
{
printf("Element deleted from queue is : %d\n", queue_arr[front]);
front=front+1;
}
} /*End of delete() */
void display()
{
int i;
if (front == -1)
printf("Queue is empty\n");
else
{
printf("Elements in the queue:\n");
for(i=front;i<= rear;i++)
```

```
printf("%d ",queue_arr[i]);
printf("\n");
}
} /*End of display() */
```

Output:

```
1. Insert
2. Delete
3. Display
4. Quit
Enter your choice: 1
Enter an element to add in the queue: 25
Enter your choice: 1
Enter an element to add in the queue: 36
Enter your choice: 3
Elements in the queue: 25, 36
Enter your choice: 2
Element deleted from the queue is: 25
```

In this example:

1. The preprocessor directives #include are given. MAXSIZE is defined as 50 using the #define statement.

2. The queue is declared as an array using the declaration int queue_arr[MAX].

3. In the while loop, the different options are displayed on the screen and the value entered in the variable choice is accepted.

4. The switch case compares the value entered and calls the method corresponding to it. If the value entered is invalid, it displays the message "Wrong choice".

5. Insert method: The insert method inserts item in the queue. The if condition checks whether the queue is full or not. If the queue is full, the "Queue overflow" message is displayed. If the queue is not full, the item is inserted in the queue and the rear is incremented by 1.

6. Delete method: The delete method deletes item from the queue. The if condition checks whether the queue is empty or not. If the queue is empty, the "Queue underflow" message is displayed. If the queue is not empty, the item is deleted and front is incremented by 1.

7. Display method: The display method displays the contents of the queue. The if condition checks whether the queue is empty or not. If the queue is not empty, it displays all the items in the queue.

9.2 Types of Queue

Simple Queue

In a simple queue, insertion takes place at the rear and removal occurs at the front. It follows the FIFO (First in First out) rule.

Circular Queue

In a circular queue, the last element points to the first element making a circular link. In a circular queue, the rear end is connected to the front end forming a circular loop. An advantage of circular queue is that, the insertion and deletion operations are independent of one another. This prevents an interrupt handler from performing an insertion operation at the same time when the main function is performing a deletion operation.

Double ended queue

Double ended queue is also known as deque. It is a type of queue where the insertions and deletions happen at the front or the rear end of the queue. The various operations that can be performed on the double ended queue are:

1. Insert an element at the front end

2. Insert an element at the rear end

3. Delete an element at the front end

4. Delete an element at the rear end

Example:

Program for Implementation of Circular Queue.

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
int Q_F(int COUNT)
{
return (COUNT==SIZE)? 1:0;
}
int Q_E(int COUNT)
{
return (COUNT==0)? 1:0;
}
void rear_insert(int item, int Q[], int *R, int *COUNT)
{
if(Q_F(*COUNT))
{
printf("Queue overflow");
return;
}
*R=(*R+1) % SIZE;
Q[*R]=num;
*COUNT+=1;
}
void front_delete(int Q[], int *F, int *COUNT)
{
if(Q_E(*COUNT))
{
printf("Queue underflow");
return;
}
printf("The deleted element is %d\n", Q[*F]);
*F=(*F+1) % SIZE;
```

```
*COUNT-=1;
}
void display(int Q[], int F, int COUNT)
{
int i,j;
if(Q_E(COUNT))
{
printf("Queue is empty\n");
return;
}
printf("The contents of the queue are:\n");
i=F;
for(j=1;j<=COUNT; j++)
{
printf("%d\n", Q[i]);
i=(i+1) % SIZE;
}
printf("\n");
}
void main()
{
int choice, num, COUNT, F, R, Q[20];
clrscr();
F=0;
R=-1;
COUNT=0;
for(;;)
{
printf("1. iInsert at front\n");
printf("2. Delete at rear end\n");
printf("3. Display\n");
printf("4. Exit\n");
scanf("%d", &choice);
switch(choice)
{
case 1: printf("Enter the number to be inserted\n");
scanf("%d", &num);
rear_insert(num, Q, &R, &COUNT);
break;
case 2: front_delete(Q, &F, &COUNT);
break;
case 3: display(Q, F, COUNT);
break;
default: exit(0);
}
}
}
```

Output:

```
1. Insert at rear end
2. Delete at front end
3. Display
4. Exit
1
Enter the number to be inserted
50
1. Insert at rear end
2. Delete at front end
3. Display
4. Exit
1
Enter the number to be inserted
60
1. Insert at rear end
2. Delete at front end
3. Display
4. Exit
3
The contents of the queue are 50 60
1. Insert at rear end
2. Delete at front end
3. Display
4. Exit
2
The element deleted is 50
```

In this example:

1. The header files are included and a constant value 5 is defined for variable SIZE using #define statement. The SIZE defines the size of the queue.

2. A queue is created using an array named Q with an element capacity of 20. A variable named COUNT is declared to store the count of number elements present in the queue.

3. Four functions are created namely, Q_F(), Q_E(), rear_insert(), front_delete(),and display(). The user has to select an appropriate function to be performed.

4. The switch statement is used to call the rear_insert(), front_delete(), and display() functions.

5. When the user enters 1, rear_insert() function is called. In the rear_insert() function, the if loop checks if the count is full. If the condition is true, then the program prints a message "Queue is empty". Else, it checks for the value of R and assigns the element (num) entered by the user to R. Initially, when there are no elements in the queue, the value of R will be 0. After every insertion, the variable COUNT is incremented.

6. When the user enters 2, the front_delete() function is called. In this function, the if loop checks if the variable COUNT is empty. If the condition is true, then the program prints a message "Queue underflow". Else, the element in the 0th

7. When the user enters 3, the display() function is called. In this function, the if loop checks if the value of COUNT is 0. If the condition is true, the program prints a message "Queue is empty". Else, the value of F is assigned to the variable i. The for loop then displays the elements present in the queue. position will be deleted. The size of F is computed and the COUNT is set to 1.

8. When the user enters 4, the program terminates.

Priority Queue

In priority queue, the elements are inserted and deleted based on their priority. Each element is assigned a priority and the element with the highest priority is given importance and processed first. If all the elements present in the queue have the same priority, then the first element is given importance.

A priority queue is an abstract data type in which each element is associated with a priority value.Elements are served on the basis of their priority.An element with high priority is dequeued before an element with low priority.If two elements have the same priority, they are served according to their order in the queue.

The priority queue moves the highest priority elements at the beginning of the priority queue and the lowest priority elements at the back of the priority queue.It supports only those elements that are comparable. Priority queue in the data structure arranges the elements in either ascending or descending order.

Types of Priority Queue

Ascending Order Priority Queue

An ascending order priority queue gives the highest priority to the lower number in that queue

Example:

List: 5 6 20 22 10

Arrange these numbers in ascending order.

List 5 6 10 20 22

Descending Order Priority Queue

A descending order priority queue gives the highest priority to the highest number in that queue.

Example:

List: 5 6 35 22 10

Arrange these numbers

List: 35 22 10 6 5

Priority Queue Operations

Inserting an Element into the Priority Queue

Deleting an Element from the Priority Queue

Peeking from the Priority Queue (Find max/min)

Extract-Max/Min from the Priority Queue

Priority queue can be implemented using

Array

Linked list

Heap data structure

Binary search tree

Priority Queue Applications

Dijkstra's algorithm: To find shortest path in graph.

Prim's Algorithm: Prim's algorithm uses the priority queue to the values or keys of nodes and draws out the minimum of these values at every step.

Data Compression: Huffman codes use the priority queue to compress the data.

Operating Systems: load balancing and interrupt handling in an operating system

9.3 Implementation of Queues

There are two possible implementation strategies – one where the memory is used statically and the other where memory is used dynamically.

Queue can be implemented using:

1. ***Array***
2. ***Linked List***

Queue implementation Using Array

To represent a queue we require a one-dimensional array of some maximum size say n to hold the data items and two other variables front and rear to point to the beginning and the end of the queue.

Queue implemented using array stores only fixed number of data values. Two variables front andrear, that are implemented in queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue.

Initially both front and rear are set to -1.For insert a new value into the queue, increment rear value by one and then insert at that position. For delete a value from the queue, then delete the element which is at front position and increment front value by one.

Enqueue operation

Enqueue() function is used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue.

Algorithm: Enqueue operation

```
Step 1: IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 3: Set QUEUE[REAR] = NUM
Step 4: EXIT
```

Dequeue operation

Dequeue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The Dequeue() function does not take any value as parameter.

Algorithm: Dequeue operation

```
Step 1: IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]
Step 2: EXIT
```

Queue implementation Using Linked list

Due to the drawbacks of array. The array implementation cannot be used for the large scale applications where the queues are implemented.The alternative of array implementation is linked list implementation of queue. In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insert operation

There can be the two scenario of inserting this new node ptr into the linked queue.In the first scenario, we insert element into an empty queue. In this case, the condition front = NULL becomes true.In the second case, the queue contains more than one element. The condition front = NULL becomes false.

Algorithm

```
Step 1: Allocate the space for the new node PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]
Step 4: END
```

Delete operation

Deletion operation removes the element that is first inserted among all the queue elements. The condition front == NULL becomes true if the list is empty. Otherwise, we will delete the element that is pointed by the pointer front.

Algorithm

```
Step 1: IF FRONT = NULL
Write " Underflow "
Go to Step 5
[END OF IF]
```

```
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

9.4 Applications of Queues

One major application of the queue data structure is in the computer simulation of a real-world situation. Queues are also used in many ways by the operating system, the program that schedules and allocates the resources of a computer system. One of these resources is the CPU (Central Processing Unit) itself. If you are working on a multi-user system and you tell the computer to run a particular program, the operating system adds your request to its "job queue". When your request gets to the front of the queue, the program you requested is executed. Similarly, the various users for the system must share the I/O devices (printers, disks etc.). Each device has its own queue of requests to print, read or write to these devices. The following subsection discusses one application of the queues – the priority queue. It is used in time-sharing multi-user systems where programs of high priority are processed first arid programs with the same priority form a standard queue.

In Operating systems:

a) Semaphores

b) FCFS (first come first serve) scheduling,

c) Spooling in printers

d) Buffer for devices like keyboard

In Networks:

a) Queues in routers/ switches

b) Mail Queues

Queues are used in operating systems for handling interrupts.

Queues are used as buffers in most of the applications like MP3 media player, CD player, etc

When a resource is shared among multiple consumers.

CPU scheduling,

Disk Scheduling.

Summary

- A queue is an ordered collection of items in which deletion takes place at the front and insertion at the rear of the queue.
- In a memory, a queue can be represented in two ways; by representing the way in which the elements are stored in the memory, and by naming the address to which the front and rear pointers point to.
- The different types of queues are double ended queue, circular queue, and priority queue.
- The basic operations performed on a queue include inserting an element at the rear end and deleting an element at the front end.
- A priority queue is a collection of elements such that each element has been assigned a priority. An element of higher priority is processed before any element of lower priority.
- Two elements with the same priority are processed according to the order in which they were inserted into the queue.

Keywords

FIFO: (First In First Out) The property of a linear data structure which ensures that the element retrieved from it is the first element that went into it.

Front: The end of a queue from where elements are retrieved.

Queue: A linear data structure in which the element is inserted at one end while retrieved from another end.

Rear: The end of a queue where new elements are inserted.

Dequeue: Process of deleting elements from the queue.

Enqueue: Process of inserting elements into queue.

SelfAssessment

1. A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as a?

A. Queue

B. Stack

C. Tree

D. Linked list

2. Process of inserting an elements at the end of queue is known as?

A. Dequeue

B. Enqueue

C. Push

D. Pop

3. The maximum size of the queue?

A. Can be changed

B. Cannot be change

C. Independent

D. None of these

4. Application of queue is

A. Serving request of a single shared resource, like a printer, CPU task scheduling

B. Call center phone system using queue to hold people calling

C. Handling of interrupt in real time system

D. All of above

5. Which of the following is not a basic operation of a queue?

A. enqueue()

B. dequeue()

C. input()

D. peek()

6. peek() function is

A. This function helps to add the data at the front of queue.

B. This function helps to delete the data at the front of queue.

C. This function helps to see the data at the front of queue.

D. None of above

7. isempty() function is

A. If the value of front is less than MIN or 0, it tells that queue is empty.

B. If the value of rear is less than MIN or 0, it tells that queue is empty.

C. All of above

D. None of above

8. What kind of a data structure does a queue is?

A. Linear

B. Non-linear

C. Both Linear and Non Linear

D. None

9. What is the operation we perform on Queues?

A. FIRST-IN –LAST-OUT

B. FIRST-IN-FIRST-OUT

C. Sum

D. All of above

10. Queue can be implemented

A. Sequential

B. Linked

C. Both sequential and linked

D. Neither sequential nor linked

11. Sequential implementation of queue means

A. Queue is implemented using array

B. Queue is implemented using linked list

C. All of above

D. None of above

12. Sequential implementation of queue means

A. Queue is implemented using array

B. Queue is implemented using linked list

C. All of above

D. None of above

13. In linked list implementation of queue, if only front pointer is maintained, which of the following operation take worst case linear time?

A. Insertion

B. Deletion

C. To empty a queue

D. Both insertion and deletion

14. In Queue, ENQUEUE means____ whereas DEQUEUE refers____.

A. An insertion operation, a deletion operation.
B. End of the queue, defining a queue.
C. Traverse operation, Insert operation
D. None of above

15. push() and pop() functions are found in
A. Queue
B. Stack
C. Tree
D. All of above

Answers for Self Assessment

1.	A	2.	B	3.	B	4.	D	5.	C
6.	C	7.	A	8.	A	9.	B	10.	C
11.	A	12.	B	13.	D	14.	A	15.	B

Review Questions

1 “Using double ended queues is more advantageous than using circular queues. “Discuss

2 “Stacks are different from queues.” Justify.

3 “Using priority queues is advantageous in job scheduling algorithms. “Analyze

4 Can a basic queue be implemented to function as a dynamic queue? Discuss

5 Describe the application of queue.

6 How will you insert and delete an element in queue?

7 Explain dynamic memory allocation advantages.

Further Readings

Books Ashok N. Kamthane, “Programming with ANCI & Turbo C”, Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

https://www.geeksforgeeks.org/

https://www.javatpoint.com/data-structure-queue

https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm

Unit 10: Introduction to Queues

CONTENTS

Objectives

After studying this unit, you will be able to:

- Learn implementation of queues
- Explain priority queue
- Discuss applications of queues

Introduction

A queue is a linear list of elements that consists of two ends known as front and rear. We can delete elements from the front end and insert elements at the rear end of a queue. A queue in an application is used to maintain a list of items that are ordered not by their values but by their sequential value.

The queue abstract data type is also a widely used one with applications very common in real life. An example comes from the operating system software where the scheduler picks up the next process to be executed on the system from a queue data structure. In this unit, we would study the various properties of queues, their operations and implementation strategies.

10.1 Circular Queue

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as Ring Buffer.
- It is an abstract data type.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

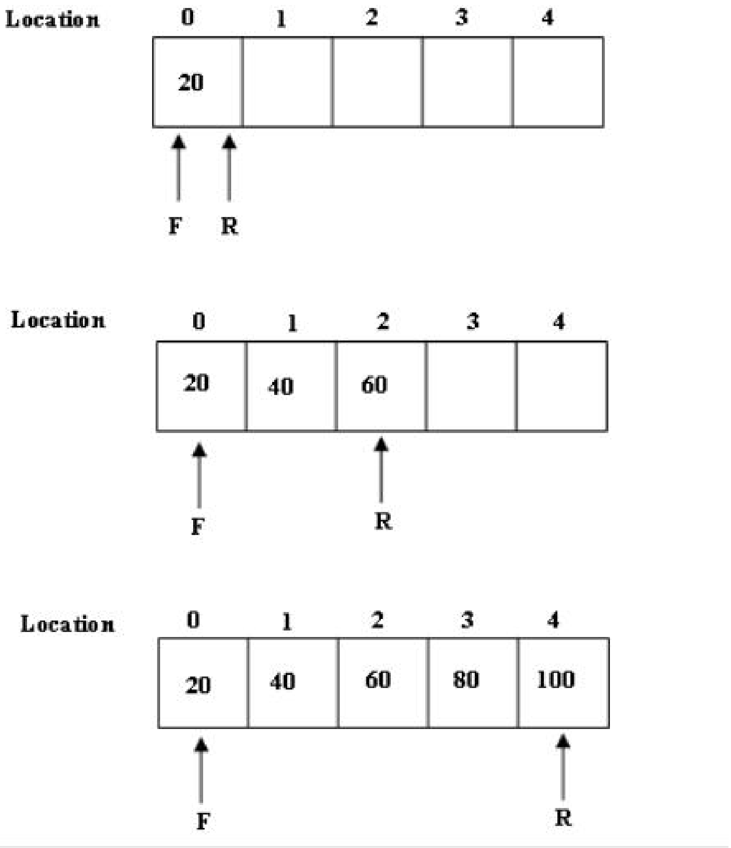
In a circular queue, the rear end is connected to the front end forming a circular loop. An advantage of circular queue is that, the insertion and deletion operations are independent of one another. This prevents an interrupt handler from performing an insertion operation at the same time when the main function is performing a deletion operation. The figure 8.5 depicts a circular queue. The queue elements are stored in an array. The front end of the queue is represented as F and the rear end is represented as R. Before inserting an element into the queue, the R pointer should be set to -1. The value of R is then incremented to insert the elements. In the first figure of figure 8.5, only one element (20) is present in the queue. Hence, the value of F and R pointer will be 0. In the second figure of figure 8.5, two elements are added (40 and 60) to the queue. This can be done by incrementing the R pointer. The following statement depicts the increment operation:

R=(R+1) % SIZE

Here,

SIZE is the queue size. In this case, the size is 5.

In the third figure of figure 8.5, elements 80 and 100 are added to the queue. Now the R value will be 5.Since, the value of SIZE is also 5, R will point to 0.



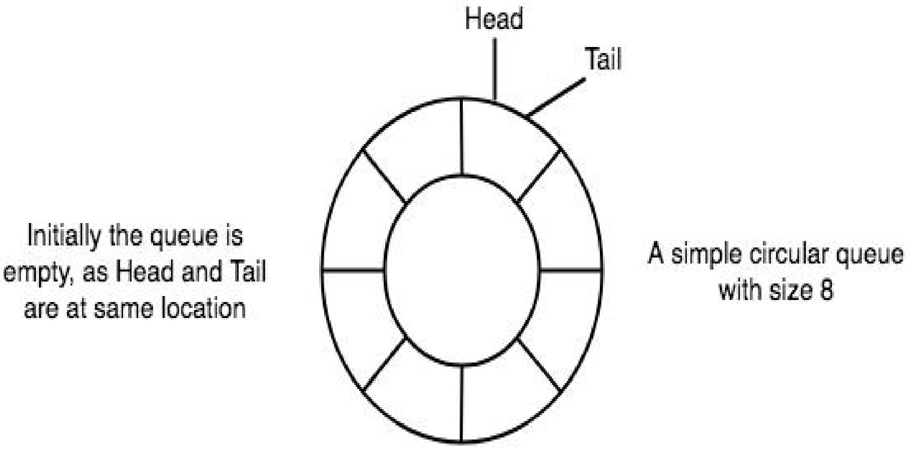
The above figure shows the structure of circular queue. It stores an element in a circular way and performs the operations according to its FIFO structure.

Did you know?

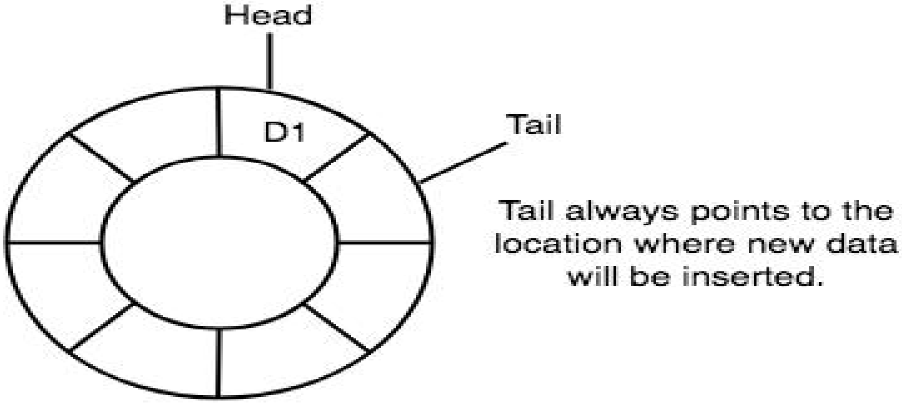
A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a Ring Buffer.

Features of Circular Queue

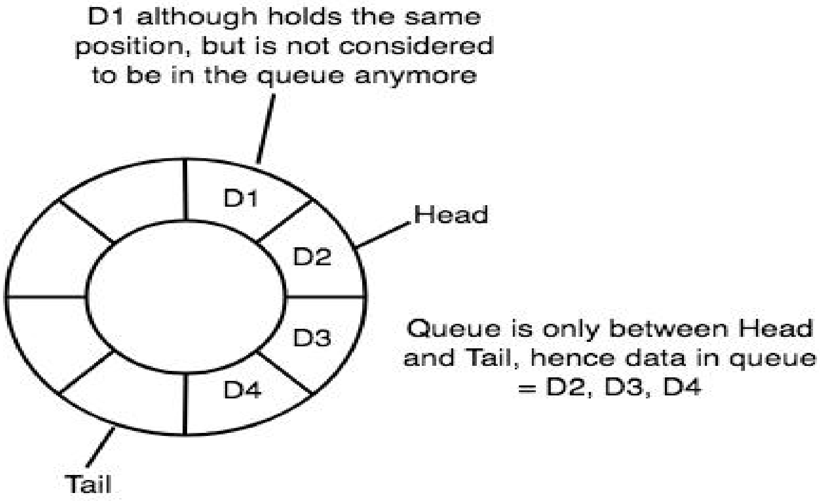
1. In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.
2. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.


Head
Tail
Initially the queue is empty, as Head and Tail are at same location
A simple circular queue with size 8

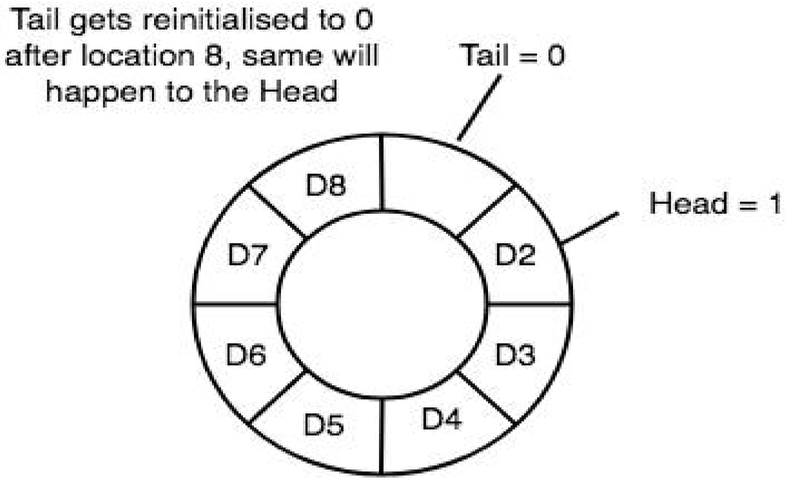
3. New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.


Head
D1
Tail
Tail always points to the location where new data will be inserted.

4. In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when dequeue is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.

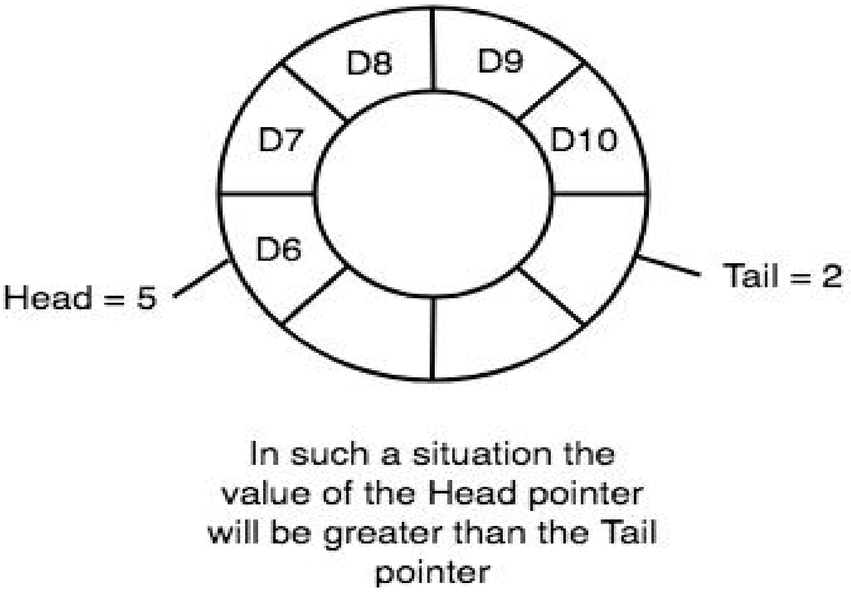

D1 although holds the same position, but is not considered to be in the queue anymore
D1
Head
D2
D3
D4
Queue is only between Head and Tail, hence data in queue = D2, D3, D4
Tail

5. The head and the tail pointer will get reinitialised to 0 every time they reach the end of the queue.



Tail gets reinitialised to 0 after location 8, same will happen to the Head

6. Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. Sounds odd? This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialised upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer

Application of Circular Queue

- Round robin execution of Jobs in a multi programming OS.
- Looped execution of the slides of a presentation.
- Assigning turns to play for multiplayer gaming systems.
- Process management tasks by Operating System.
- Browsing through the open windows applications using alt + tab (Microsoft Windows).
- Time division multiplexing in case of Networks and Communications.

Memory Representation of Circular Queue

Circular Queues can also be represented in memory in two ways

1. Using the contiguous memory like an array.
2. Using the non-contiguous memory like a linked list.

Circular Queue using an Array

QUEUE- the name of the array storing queue elements.

MAX- defining that how many elements (maximum count) can be stored in the array representing the queue.

FRONT- the index where the first element is stored in the array representing the queue. If the last element is deleted from index MAX, then the FRONT variable is updated with 0 (first index of the array).

REAR- the index where the last element is stored in array representing the queue. If last stored value is at MAX and FRONT is not 0 then the new element will be stored at 0 index.

– In this representation the circular queue is implemented using a Linked List. Using linked list for creating a queue makes it flexible in terms of access.

– Pointers (links) to store addresses of nodes for defining a circular queue are

Using Linked List

- FRONT- address of the first element of the Linked list storing the Queue.
- REAR- address of the last element of the Linked list storing the Queue. LINK part of the REAR node stores the address of the first node.

Circular Queue Operations

- Two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last elements of the queue
- Initially, set value of FRONT and REAR to -1

Enqueue Operation

- Check if the queue is full
- For the first element, set value of FRONT to 0
- Circularly increase the REAR index by 1 (i.E. If the rear reaches the end, next it would be at the start of the queue)
- Add the new element in the position pointed to by REAR

Deqeue Operation

- Check if the queue is empty
- Return the value pointed by FRONT
- Circularly increase the FRONT index by 1
- For the last element, reset the values of FRONT and REAR to -1

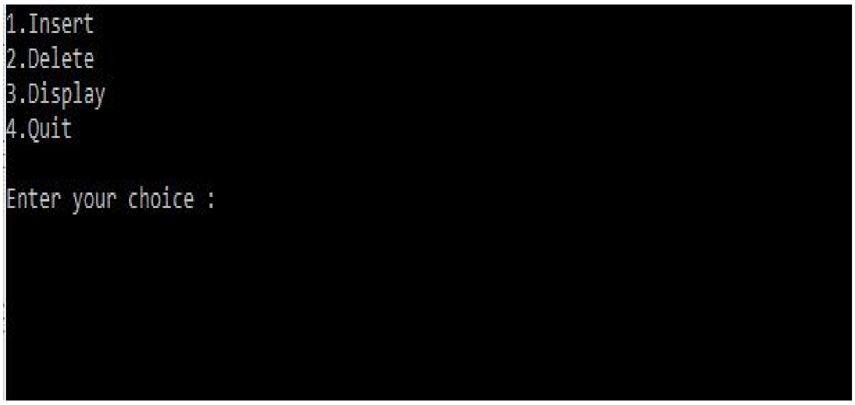
Lab Exercise

```
#include<stdio.h>

# define MAX 5

int cqueue_arr[MAX];

int front = -1;

int rear = -1;

void insert(int item)
```

```
{
if((front == 0 && rear == MAX-1) || (front == rear+1))
{
printf("Queue Overflow \n");
return;
}
if(front == -1)
{
front = 0;
rear = 0;
}
else
{
if(rear == MAX-1)
rear = 0;
else
rear = rear+1;
}
cqueue_arr[rear] = item ;
}
void deletion()
{
if(front == -1)
{
printf("Queue Underflow \n");
return ;
}
printf("Element deleted from queue is : %d \n",cqueue_arr[front]);
if(front == rear)
{
front = -1;
rear=-1;
}
else
{
if(front == MAX-1)
front = 0;
else
front = front+1;
}
```

```
}
void display()
{
int front_pos = front,rear_pos = rear;
if(front == -1)
{
printf("Queue is empty \n");
return;
}
printf("Queue elements : ");
if( front_pos<= rear_pos )
while(front_pos<= rear_pos)
{
printf("%d ",cqueue_arr[front_pos]);
front_pos++;
}
else
{
while(front_pos<= MAX-1)
{
printf("%d ",cqueue_arr[front_pos]);
front_pos++;
}
while(front_pos<= rear_pos)
{
printf("%d ",cqueue_arr[front_pos]);
front_pos++;
}
}
printf("null");
}
int main()
{
int choice,item;
do
{
printf("\n1.Insert\n");
printf("2.Delete\n");
printf("3.Display\n");
printf("4.Quit\n");
printf("\nEnter your choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1 :
printf("Input the element for insertion in queue : ");
scanf("%d", &item);
insert(item);
break;
case 2 :
deletion();
break;
case 3:
display();
break;
case 4:
break;
default:
printf("Wrong choice \n");
}
}while(choice!=4);
return 0;
}
```

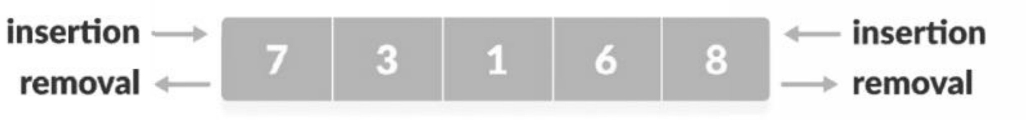
Output



```
1.Insert
2.Delete
3.Display
4.Quit

Enter your choice :
```

10.2 Deque

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First in First Out).



Types of Deque

- **Input Restricted Deque**

In this deque, input is restricted at a single end but allows deletion at both the ends.

- **Output Restricted Deque**

In this deque, output is restricted at a single end but allows insertion at both the ends.

Operations performed on deque

There are the following operations that can be applied on a deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque –

- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Applications of deque

- An internet browser's history.
- Another common application of the deque is storing a computer code application's list of undo operations.
- Have you ever see Money-Control App, it'll show the stocks you last visited, it'll take away the stocks when a while and can add the most recent ones.
- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Lab Exercise

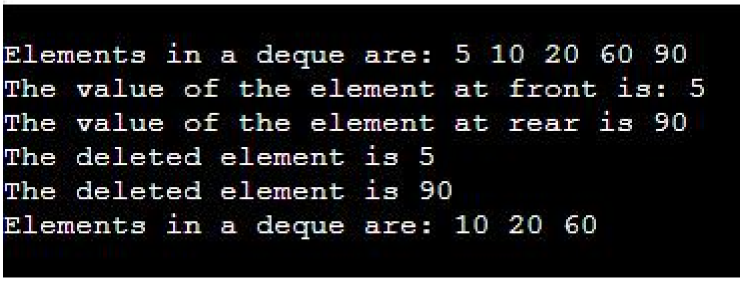
```
#include <stdio.h>
#define size 5
int deque[size];
int f = -1, r = -1;
// insert_front function will insert the value from the front
void insert_front(int x)
{
if((f==0 && r==size-1) || (f==r+1))
```

```
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        f=r=0;
        deque[f]=x;
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}

// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
```

```
      deque[r]=x;
    }

}

            // display function prints all the value of deque.
void display()
{
    int i=f;
printf("\nElements in a deque are: ");

    while(i!=r)
    {
printf("%d ",deque[i]);
i=(i+1)%size;
    }
     printf("%d",deque[r]);
}

            // getfront function retrieves the first value of the deque.
void getfront()
{
    if((f==-1) && (r==-1))
    {
printf("Deque is empty");
    }
    else
    {
printf("\nThe value of the element at front is: %d", deque[f]);
    }

}

// getrear function retrieves the last value of the deque.
void getrear()
{
    if((f==-1) && (r==-1))
    {
printf("Deque is empty");
    }
```

```
   else
   {
printf("\nThe value of the element at rear is %d", deque[r]);
   }

}

// delete_front() function deletes the element from the front
void delete_front()
{
   if((f==-1) && (r==-1))
   {
printf("Deque is empty");
   }
   else if(f==r)
   {
printf("\nThe deleted element is %d", deque[f]);
      f=-1;
      r=-1;

   }
    else if(f==(size-1))
    {
printf("\nThe deleted element is %d", deque[f]);
       f=0;
    }
    else
    {
printf("\nThe deleted element is %d", deque[f]);
        f=f+1;
    }
}

// delete_rear() function deletes the element from the rear
void delete_rear()
{
   if((f==-1) && (r==-1))
   {
printf("Deque is empty");
   }
   else if(f==r)
   {
printf("\nThe deleted element is %d", deque[r]);
      f=-1;
      r=-1;

   }
    else if(r==0)
    {
printf("\nThe deleted element is %d", deque[r]);
       r=size-1;
    }
    else
    {
printf("\nThe deleted element is %d", deque[r]);
        r=r-1;
    }
}

int main()
{
insert_front(10);
insert_front(5);
insert_rear(20);
insert_rear(60);
insert_rear(90);
display();  // Calling the display function to retrieve the values of deque
getfront();  // Retrieve the value at front-end
getrear();  // Retrieve the value at rear-end
delete_front();
delete_rear();
display(); // calling display function to retrieve values after deletion
   return 0;
}
```

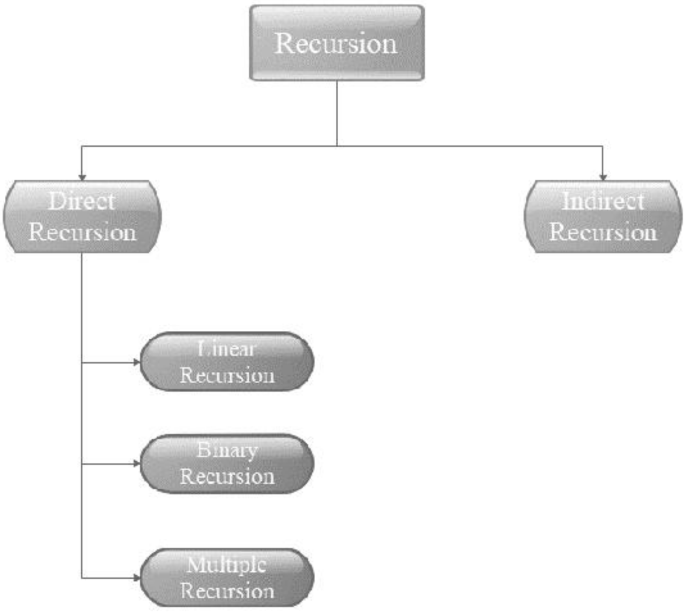
Output



```
Elements in a deque are: 5 10 20 60 90
The value of the element at front is: 5
The value of the element at rear is 90
The deleted element is 5
The deleted element is 90
Elements in a deque are: 10 20 60
```

10.3 Recursion

- Recursion is one of the most powerful tools in a programming language, but one of the most threatening topics-as most of the beginners and not surprising to even experience students feel.
- When function is called within the same function, it is known as recursion in C. The function which calls the same function, is known as recursive function.
- Recursion is defined as defining anything in terms of itself. Recursion is used to solve problems involving iterations, in reverse order.



Disadvantages of Recursion

1. The computer may run out of memory if the recursive calls are not checked.
2. It is not more efficient in terms of speed and execution time.
3. According to some computer professionals, recursion does not offer any concrete advantage over non-recursive procedures/functions.
4. Recursive solution is always logical and it is very difficult to trace.(debug and understand).
5. Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
6. Recursion uses more processor time.

7. Recursion is not advocated when the problem can be through iteration.
8. Recursion may be treated as a software tool to be applied carefully and selectively.

Difference between recursion and iteration

Iteration	Recursion
In iteration,a problem is converted into a train of steps that are finished one at a time, one after another	Recursion is like piling all of those steps on top of each other and then quashing the mall into the solution.
With iteration,each step clearly leads on to the next, like stepping stones across a river	In recursion, each step replicates itself at a smaller scale, so that all of them combined together eventually solve the problem.
Any iterative problem is solved recursively	Not all recursive problem can solved by iteration
It does not use Stack	It uses Stack

Summary

- A queue is an ordered collection of items in which deletion takes place at the front and insertion at the rear of the queue.
- In a memory, a queue can be represented in two ways; by representing the way in which the elements are stored in the memory, and by naming the address to which the front and rear pointers point to.
- The different types of queues are double ended queue, circular queue, and priority queue.
- The basic operations performed on a queue include inserting an element at the rear end and deleting an element at the front end.
- Two elements with the same priority are processed according to the order in which they were inserted into the queue.

Keywords

FIFO: (First In First Out) the property of a linear data structure which ensures that the element retrieved from it is the first element that went into it.

Front: The end of a queue from where elements are retrieved.

Queue: A linear data structure in which the element is inserted at one end while retrieved from another end.

Rear: The end of a queue where new elements are inserted.

Dequeue: Process of deleting elements from the queue.

Enqueue: Process of inserting elements into queue.

SelfAssessment

1. Circular Queue is also known as ________

A. Ring Buffer

B. Square Buffer

C. Rectangle Buffer

D. Curve Buffer

2. A data structure in which elements can be inserted or deleted at/from both ends but not in the middle is?

A. Queue

B. Circular queue

C. Dequeue

D. Priority queue

3. Choose the application of circular queue

A. Looped execution of the slides of a presentation.

B. Assigning turns to play for multiplayer gaming systems.

C. Process management tasks by Operating System.

D. All of above

4. Circular queues can also be represented in memory using following ways.

A. Using contiguous memory allocation

B. Using non-contiguous memory allocation

C. Circular queue cannot be represented in memory

D. Using contiguous and non-contiguous memory allocation

5. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location of the queue is full.

A. True

B. False

6. A data structure in which elements can be inserted or deleted at/from both ends but not in the middle is?

A. Queue

B. Circular queue

C. Dequeue

D. Priority queue

7. Similarly in DEQUEUEs, insertion is performed at ___ end whereas the deletion is performed at __ end.

A. FRONT, REAR

B. REAR, FRONT.

C. FRONT, REAR & REAR, FRONT

D. None of the above

8. What the applications are of dequeue?

A. A-Steal job scheduling algorithm

B. Can be used as both stack and queue

C. To find the maximum of all sub arrays of size k

D. All of above

9. What is a dequeue?

A. A queue with insert/delete defined for both front and rear ends of the queue

B. A queue implemented with a doubly linked list

C. A queue implemented with both singly and doubly linked lists

D. None of the mentioned

10. What is the functionality of the following piece of code?

```
public void function(Object item)
{
    Node temp=new Node(item,trail);
    if(isEmpty())
    {
      head.setNext(temp);
      temp.setNext(trail);
    }
    else
    {
      Node cur=head.getNext();
      while(cur.getNext()!=trail)
      {
                cur=cur.getNext();
      }
      cur.setNext(temp);
    }
    size++;
}
```

A. Insert at the front end of the dequeue

B. Insert at the rear end of the dequeue

C. Fetch the element at the rear end of the dequeue

D. Fetch the element at the front end of the dequeue

11. A function is called indirect recursive ______

A. If it calls the same function.

B. If it calls another function.

C. Execute other function

D. Above all

12. Function which call itself is called______

A. Static function

B. Auto function

C. Recursive function

D. All of above

13. Which of the following statements is true?

A. Recursion is always better than iteration

B. Recursion uses more memory compared to iteration

C. Recursion uses less memory compared to iteration

D. Iteration is always better and simpler than recursion

14. Recursion is similar to which of the following?

A. Switch Case

B. Loop

C. If-else

D. if el if else

15. When any function is called from main(), the memory is allocated to it on the stack.

A. True

B. False

C. Can be true or false

D. Neither true nor false

Answers for Self Assessment

1.	A	2.	B	3.	D	4.	D	5.	A
6.	C	7.	C	8.	D	9.	A	10.	B
11.	B	12.	C	13.	B	14.	B	15.	A

Review Questions

1 "Using double ended queues is more advantageous than using circular queues. "Discuss

2 "Stacks are different from queues." Justify.

3 Write a program that implement circular queue data structure.

4 Can a basic queue be implemented to function as a dynamic queue? Discuss

5 Describe the application of circular queue.

6 How will you insert and delete an element in circular queue?

7 Explain dynamic memory allocation advantages.

Further Readings

Books Ashok N. Kamthane, "Programming with ANCI & Turbo C", Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web links

https://www.programiz.com/dsa/circular-queue

https://www.geeksforgeeks.org/

https://www.javatpoint.com/data-structure-queue

Unit 11: Trees

CONTENTS

Objectives

After studying this unit, you will be able to:

- Define trees
- Explain different types of trees
- Discuss the applications of trees

Introduction

We know that a data structure is a collection of data pieces labelled with the same name. A data structure can be thought of as a collection of rules for keeping data together. Data structures are used in almost all computer programmes. Algorithms are incomplete without data structures. It can be used to manage massive databases with large amounts of data. Data structures are prioritised in several modern programming languages over algorithms.

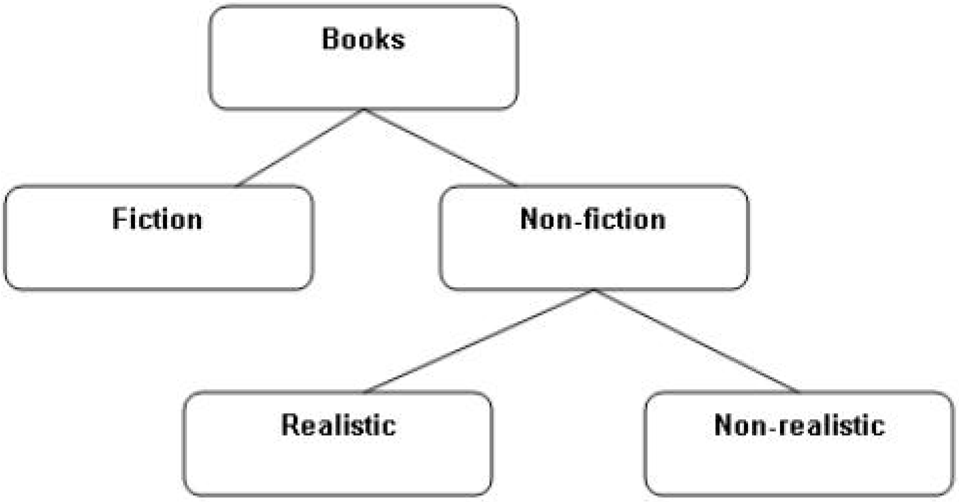
There are many data structures that help us to manipulate the data stored in the memory, which we have discussed in the previous units. These include array, stack, queue, and linked-list.

Choosing the best data structure for a program is a challenging task. Similar tasks may require different data structures. We derive new data structures for complex tasks using the already existing ones. We need to compare the characteristics of the data structures before choosing the right data structure. A tree is a hierarchical data structure suitable for representing hierarchical information. The tree data structure has the characteristics of quick search, quick inserts, and quick deletes.

11.1 Trees

A tree is a very important data structure as it is useful in many applications. A tree structure is amethod of representing the hierarchical nature of a structure in a graphical form. It is termed as "treestructure" since its representation resembles a tree. However, the chart of a tree is normally upsidedown compared to an actual tree, with the root at the top and the leaves at the bottom.

Figure shows tree structure of books.



In the hierarchical organization of books shown in figure 10.1, Books is the root of the tree. Books can be classified as Fiction and Non-fiction. Non-fiction books can be further classified as Realistic and Non-realistic, which are the leaves of the tree. Thus, it forms a complete tree structure.

Trees are primarily treated as data structures rather than as data types.

A tree is a widely-used data structure that depicts a hierarchical tree structures with a set of linked nodes. The elements of data structure in a tree are arranged in a non-linear fashion i.e., they use two dimensional representations. Thus, trees are known as non-linear data structures. This data structure is more efficient in inserting additional data, deleting unnecessary data, and searching new data.

Did you Know?

- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially.
- In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size.
  - But, it is not acceptable in today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move sub trees around with minimum effort

Representation of Tree in Graphs

A graph G consists of a set of objects V = {v1, v2, v3 …} called vertices (points or nodes) and a set of objects E = {e1, e2, e3 ….} called edges (lines or arcs).

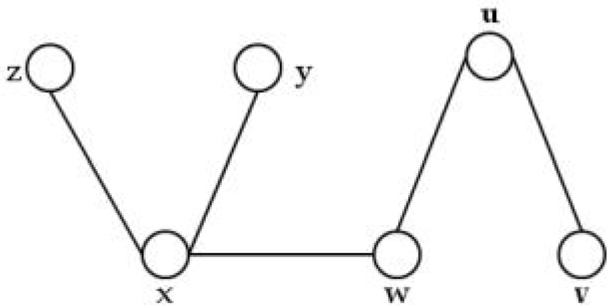
The set V (G) is called the vertex set of G and E (G) is the edge set.

The graph is denoted as G = (V, E)

Let G be a graph and {u, v} an edge of G. Since {u, v} is 2-element set, we write {v, u} instead of {u, v}.

This edge can be represented as uv or vu.

If e = uv is an edge of a graph G, then u and v are adjacent in G and e joins u and v.

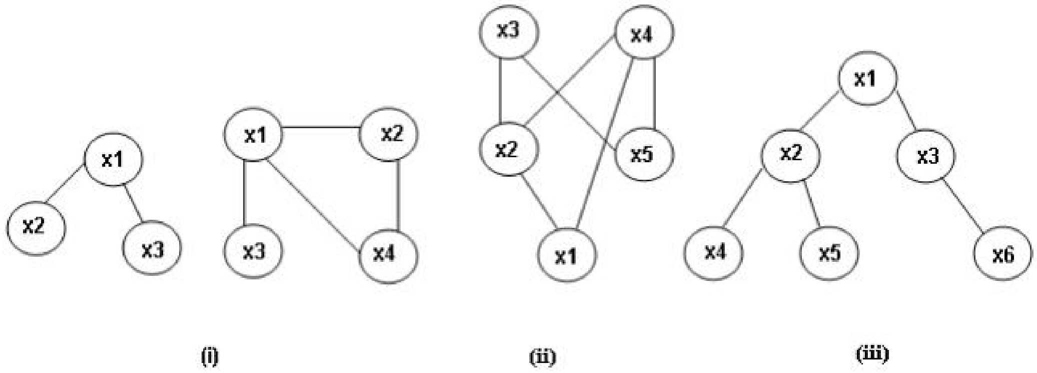


This graph G is defined by the sets:

V (G) = {u, v, w, x, y, z} and E(G) = { uv, uw, wx, xy, xz}

Every graph has a diagram associated with it. The vertex u and an edge e are incident with each other as are v and e. If two distinct edges e and f are incident with a common vertex, then they are adjacent edges.

Following figure depicts three examples of graphs. Graphs, unlike trees, can have sets of nodes that are disconnected from other sets of nodes.



In figure 10.3, the graph (i) has two different, unconnected set of nodes. Graphs can also contain cycles. Graph (ii) has several cycles. One such path is from x1 to x2 to x4 and back to x1. Another one is from x1 to x2 to x3 to x5 to x4 and back to x1. (There are also cycles in graph (ii).) Graph (iii) does not have any cycles and all nodes are reachable. Therefore, it is a tree.

Tree Terminologies

Node

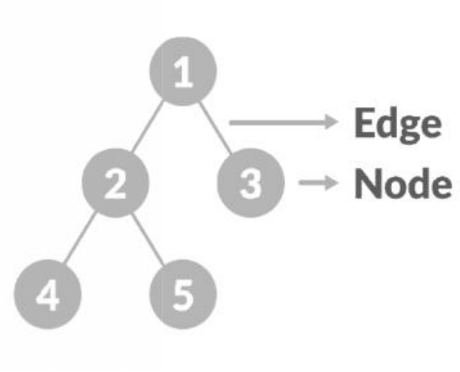
A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called leaf nodes or external nodes that do not contain a link/pointer to child nodes.

The node having at least a child node is called an internal node.

Edge

It is the link between any two nodes.



Root

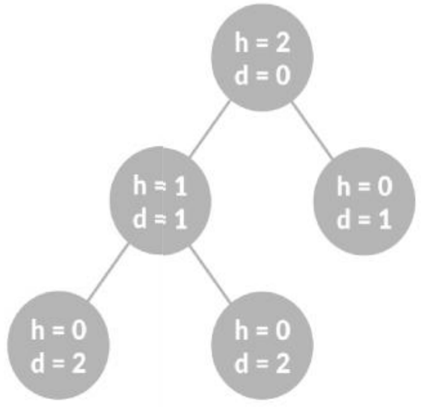
It is the topmost node of a tree.

Height of a Node

The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

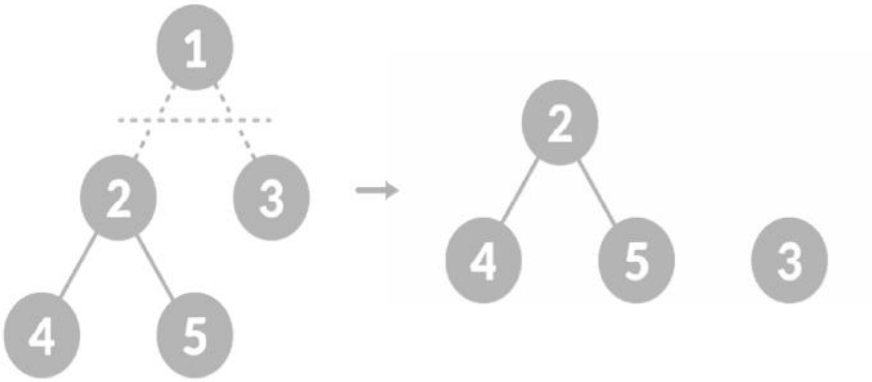


Degree of a Node

The degree of a node is the total number of branches of that node.

Forest

A collection of disjoint trees is called a forest.



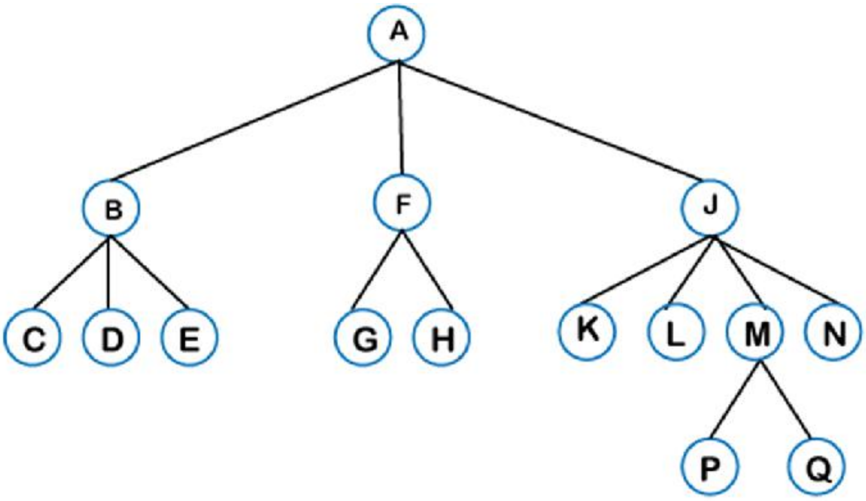
Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.

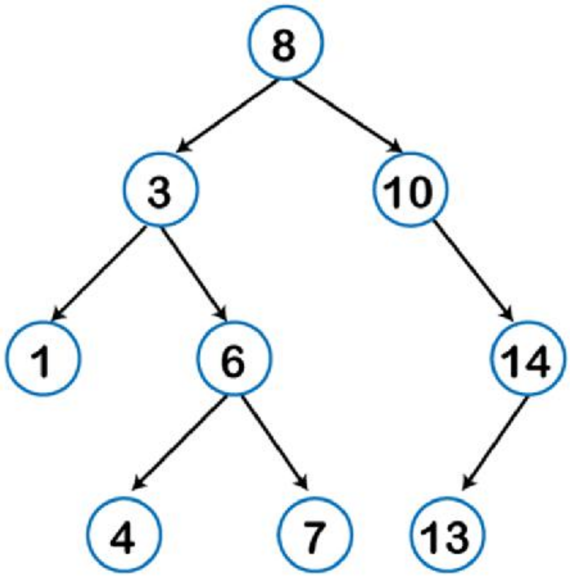
11.2 Types of Trees

1. General Tree
2. Binary Tree
3. Binary Search Tree
4. AVL Tree

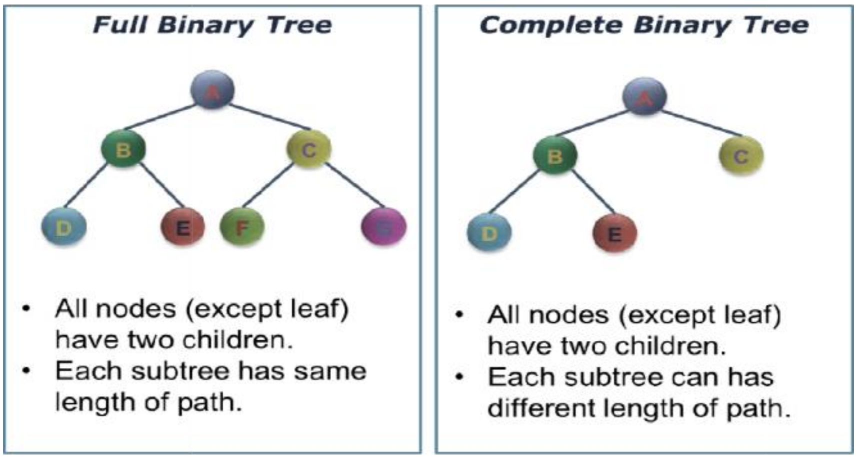
1. **General Tree:** In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as subtrees.



2. **Binary Tree***:* Binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



Types of Binary Tree



Full Binary Tree

- All nodes (except leaf) have two children.
- Each subtree has same length of path.

Complete Binary Tree

- All nodes (except leaf) have two children.
- Each subtree can has different length of path.

Binary Search Tree

Binary search tree is a non-linear data structure in which one node is connected to n number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child.

AVL Tree

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the binary tree as well as of the binary search tree.It is a self-balancing binary search tree that was invented by Adelson VelskyLindas.Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the balancing factor.

11.3 Representation of Binary Tree

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Linked List Representation
2. Array Representation (Sequential representation)

1. Linked representation

Binary trees in linked representation are stored in the memory as linked lists. These lists have nodes that aren't stored at adjacent or neighboring memory locations and are linked to each other through the parent-child relationship associated with trees.

In this representation, each node has three different parts –

- pointer that points towards the right node,
- pointer that points towards the left node,
- data element.

This is the more common representation. All binary trees consist of a root pointer that points in the direction of the root node. When you see a root node pointing towards null or 0, you should know that you are dealing with an empty binary tree. The right and left pointers store the address of the right and left children of the tree.

2. Sequential representation

Although it is simpler than linked representation, its inefficiency makes it a less preferred binary tree representation of the two. The inefficiency lies in the amount of space it requires for the storage of different tree elements. The sequential representation uses an array for the storage of tree elements.

The number of nodes a binary tree has defines the size of the array being used. The root node of the binary tree lies at the array's first index. The index at which a particular node is stored will define the indices at which the right and left children of the node will be stored. An empty tree has null or 0 as its first index.

11.4 Binary Search Tree

In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.

Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.

Key property is value at node

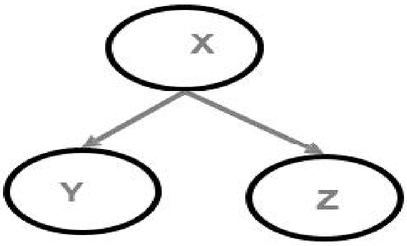
Smaller values in left subtree.

Larger values in right subtree.

Example

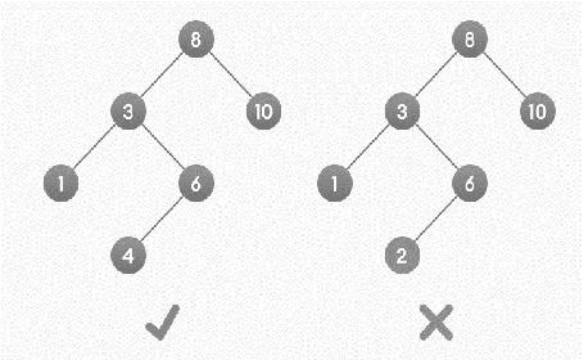
X > Y

X < Z



The properties that separate a binary search tree from a regular binary tree

1. All nodes of left subtree are less than the root node.
2. All nodes of right subtree are more than the root node.
3. Both subtrees of each node are also BSTs i.e. they have the above two properties.



A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree.

Types of Binary Trees

1. Complete binary tree: All the levels in the trees are full of last level's possible exceptions. Similarly, all the nodes are full, directing the far left.
2. Full binary tree: All the nodes have 2 child nodes except the leaf.
3. Balanced or Perfect binary tree: In the tree, all the nodes have two children. Besides, there is the same level of each sub node.

Advantages of binary search tree

- Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
- The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o(log2n) time. In worst case, the time it takes to search an element is 0(n).
- It also speed up the insertion and deletion operations as compare to that in array and linked list.

Binary Search Tree Applications

- In multilevel indexing in the database.
- For dynamic sorting.
- For managing virtual memory areas in Unix kernel.

Difference between BT (Binary Tree) and BST (Binary Search Tree)

A binary tree is simply a tree in which each node can have at most two children.

A binary search tree is a binary tree in which the nodes are assigned values, with the following restrictions:

1. No duplicate values.
2. The left subtree of a node can only have values less than the node.
3. The right subtree of a node can only have values greater than the node and recursively defined.
4. The left subtree of a node is a binary search tree.
5. The right subtree of a node is a binary search tree.

Example

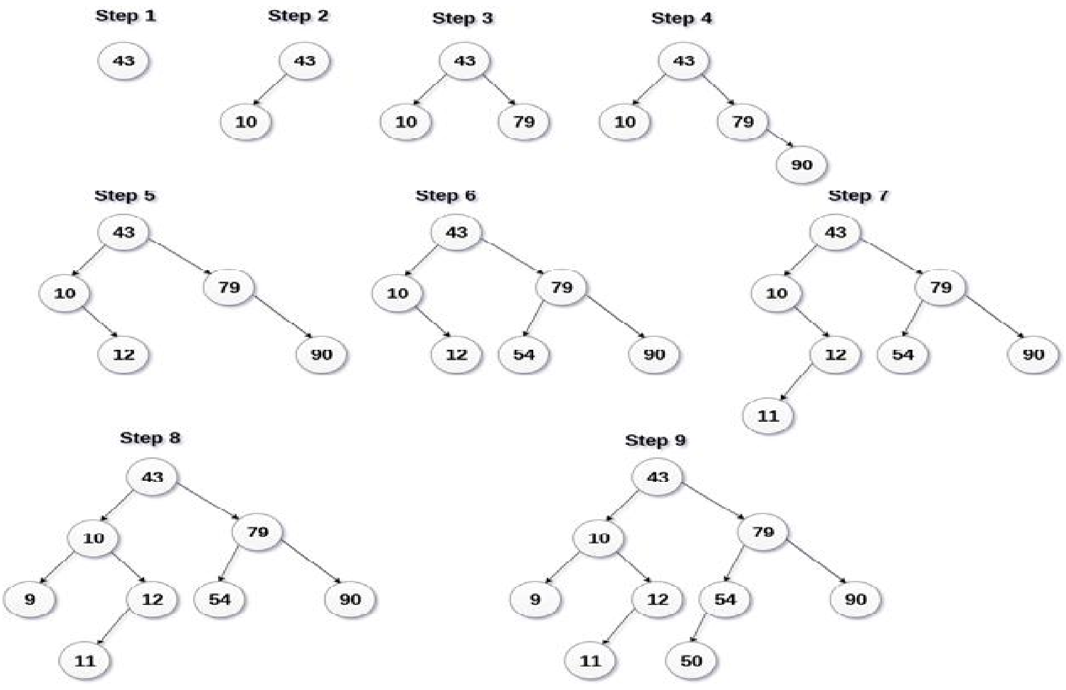
Create the binary search tree using the following data elements

43, 10, 79, 90, 12, 54, 11, 9, 50

Insert 43 into the tree as the root of the tree.

Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.

Otherwise, insert it as the root of the right of the right sub-tree.



Binary search Tree Creation

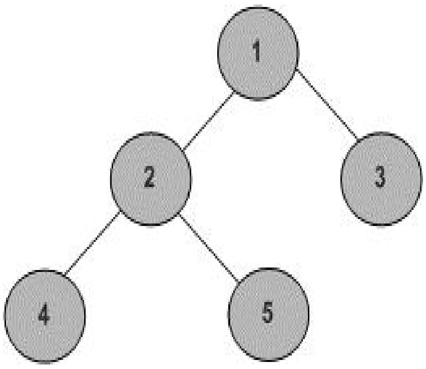
11.5 Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. We cannot randomly access a node in a tree.

Ways of traversal are

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

Example



Inorder (Left, Root, Right) : 4 2 5 1 3

Preorder (Root, Left, Right) : 1 2 4 5 3

Postorder (Left, Right, Root) : 4 5 2 3 1

Inorder algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Preorder algorithm

Until all nodes are traversed −

Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Postorder algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

Tree Traversal Methods

There are two methods for traversal

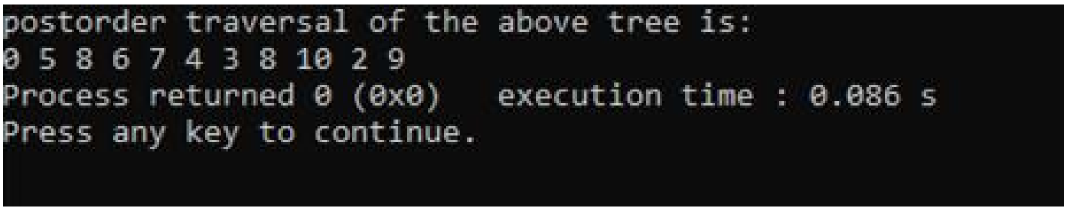
1. Recursive
2. Non Recursive

Example

```
// Postorder Traversal in using Recursion
#include <stdio.h>
#include <stdlib.h>
struct tree {
   int val;
   struct tree* left;
   struct tree* right;
};
typedef struct tree TreeNode;
TreeNode* newTree(int data)
{
TreeNode* root = (TreeNode*)malloc(sizeof(TreeNode));
   root->val = data;
```

```
    root->left = NULL;
    root->right = NULL;
    return (root);
}
void postorder(TreeNode* root)
{
    if (root == NULL)
return;
postorder(root->left);
postorder(root->right);
    printf("%d ", root->val);
}
int main()
{
TreeNode* t = newTree(9);
    t->left = newTree(3);
    t->left->left = newTree(0);
    t->left->right = newTree(4);
    t->left->right->left = newTree(5);
    t->left->right->right = newTree(7);
    t->left->right->right->left = newTree(8);
    t->left->right->right->right = newTree(6);
    t->right = newTree(2);
    t->right->left = newTree(8);
    t->right->right = newTree(10);
    printf("postorder traversal of the above tree is:\n");
postorder(t);
    return 0;
}
```

Output



```
postorder traversal of the above tree is:
0 5 8 6 7 4 3 8 10 2 9
Process returned 0 (0x0)   execution time : 0.086 s
Press any key to continue.
```

Example

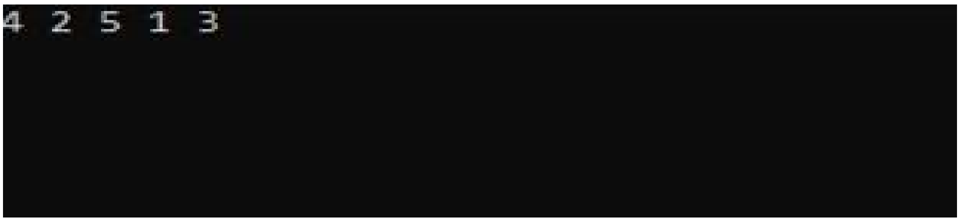
```
// Inorder Tree Traversal without Recursion
#include<stdio.h>
```

```
#include<stdlib.h>
#define bool int
struct tNode
{
int data;
struct tNode* left;
struct tNode* right;
};
struct sNode
{
struct tNode *t;
struct sNode *next;
};
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);
void inOrder(struct tNode *root)
{
struct tNode *current = root;
struct sNode *s = NULL;
bool done = 0;
while (!done)
{
        if(current != NULL)
    {
        push(&s, current);
    current = current->left;
    }
    else
    {
    if (!isEmpty(s))
    {
        current = pop(&s);
        printf("%d ", current->data);
    current = current->right;
    }
    else
        done = 1;
    }
}
```

```
}
void push(struct sNode** top_ref, struct tNode *t)
{
struct sNode* new_tNode =
                                    (structsNode*) malloc(sizeof(struct sNode));
if(new_tNode == NULL)
{
            printf("Stack Overflow \n");
            getchar();
            exit(0);
}
new_tNode->t = t;
new_tNode->next = (*top_ref);
(*top_ref) = new_tNode;
}
bool isEmpty(struct sNode *top)
{
return (top == NULL)? 1 : 0;
}
struct tNode *pop(struct sNode** top_ref)
{
struct tNode *res;
struct sNode *top;
if(isEmpty(*top_ref))
{
            printf("Stack Underflow \n");
            getchar();
            exit(0);
}
else
{
            top = *top_ref;
            res = top->t;
            *top_ref = top->next;
            free(top);
            return res;
}
}
struct tNode* newtNode(int data)
{
```

```
struct tNode* tNode = (struct tNode*)
malloc(sizeof(struct tNode));
tNode->data = data;
tNode->left = NULL;
tNode->right = NULL;
return(tNode);
}
int main()
{
struct tNode *root = newtNode(1);
root->left          = newtNode(2);
root->right         = newtNode(3);
root->left->left = newtNode(4);
root->left->right = newtNode(5);
inOrder(root);
getchar();
return 0;
}
```

Output



```
4 2 5 1 3
```

Task

- Program to demonstrate working of traversal using recursive way.
- Program to demonstrate working of traversal using non recursive way.

11.6 Searching

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in the binary search tree is pretty easy due to the fact that, elements in BST are stored in a particular order.

Following are steps involved in searching

- Compare the element with the root of the tree.
- If the item is matched, then return the location of the node.
- Otherwise, check if the item is less than the element present at the root. If so, then move to the left sub-tree.
- If not, then move to the right sub-tree.
- Repeat this procedure recursively until a match is found.
- If an element is not found, it returns NULL.

Algorithm

Search (ROOT, ITEM)

```
Step 1: IF ROOT -> DATA = ITEM OR ROOT = NULL
    Return ROOT
   ELSE
   IF ROOT < ROOT -> DATA
   Return search(ROOT -> LEFT, ITEM)
  ELSE
   Return search(ROOT -> RIGHT,ITEM)
  [END OF IF]
  [END OF IF]
Step 2: END
```

11.7 Insertion and Deletion in Binary Search Trees

Insertion in BST

Insertion Operation is performed to insert an element in the Binary Search Tree.The insertion of a new key always takes place as the child of some leaf node.

Deletion in BST

In a binary search tree, the deletion operation is performed with O(n) time complexity. Deleting a node from Binary search tree includes following three cases.

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

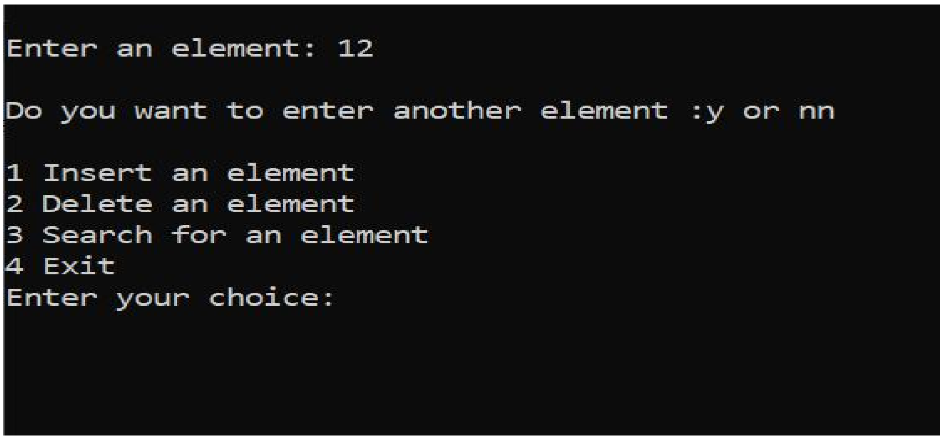
Example

```
// Program to perform insertion, deletion and searching in Tree
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int info;
            struct node*left;
            struct node*right;
};
typedef struct node BST;
BST *LOC, *PAR;
```

```
void search(BST *root, int item)
{
   BST *save,*ptr;
   if (root == NULL)
   {
      LOC = NULL;
      PAR=NULL;
   }
if (item == root -> info)
   {
   LOC = root;
   PAR = NULL;
return;
   }
   if (item < root->info)
   {
      save = root;
ptr = root->left;
   }
   else
   {
      save = root;
ptr = root ->right;
   }
   while( ptr != NULL)
   {
      if (ptr -> info == item)
      {
         LOC = ptr;
         PAR = save;
return;
      }
if(item <ptr->info)
      {
         save = ptr;
ptr = ptr->left;
      }
      else
      {
         save = ptr;
ptr = ptr->right;
      }
   }
   LOC = NULL;
   PAR = save;
return;
}
struct node* findmin(struct node*r)
{
            if (r == NULL)
                        return NULL;
            else if (r->left!=NULL)
                        return findmin(r->left);
            else if (r->left == NULL)
                        return r;
}
struct node*insert(struct node*r, int x)
{
            if (r == NULL)
            {
         r = (struct node*)malloc(sizeof(struct node));
         r->info = x;
         r->left = r->right = NULL;
         return r;
            }
            else if (x < r->info)
         r->left = insert(r->left, x);
            else if (x > r->info)
         r->right = insert(r->right, x);
            return r;
}
struct node* del(struct node*r, int x)
{
            struct node *t;
            if(r == NULL)
                        printf("\nElement not found");
            else if (x < r->info)
         r->left = del(r->left, x);
            else if (x > r->info)
         r->right = del(r->right, x);
```

```
        else if ((r->left != NULL) && (r->right != NULL))
{
      t = findmin(r->right);
      r->info = t->info;
      r->right = del(r->right, r->info);
        }
        else
        {
      t = r;
      if (r->left == NULL)
        r = r->right;
      else if (r->right == NULL)
        r = r->left;
      free(t);
        }
        return r;
}
int main()
{
  struct node* root = NULL;
  int x, c = 1, z;
  int element;
  char ch;
  printf("\nEnter an element: ");
scanf("%d", &x);
  root = insert(root, x);
  printf("\nDo you want to enter another element :y or n");
scanf(" %c",&ch);
  while (ch == 'y')
  {
    printf("\nEnter an element:");
scanf("%d", &x);
    root = insert(root,x);
    printf("\nPress y or n to insert another element: y or n: ");
scanf(" %c", &ch);
  }
  while(1)
  {
    printf("\n1 Insert an element ");
    printf("\n2 Delete an element");
    printf("\n3 Search for an element ");
    printf("\n4 Exit ");
    printf("\nEnter your choice: ");
scanf("%d", &c);
    switch(c)
    {
      case 1:
        printf("\nEnter the item:");
scanf("%d", &z);
        root = insert(root,z);
break;
      case 2:
        printf("\nEnter the info to be deleted:");
scanf("%d", &z);
        root = del(root, z);
                break;
case 3:
        printf("\nEnter element to be searched:  ");
scanf("%d", &element);
        search(root, element);
        if(LOC != NULL)
          printf("\n%d Found in Binary Search Tree !!\n",element);
        else
          printf("\nIt is not present in Binary Search Tree\n");
break;
      case 4:
        printf("\nExiting...");
                return;
      default:
        printf("Enter a valid choice: ");
    }
  }
  return 0;
}
```

Output



```
Enter an element: 12

Do you want to enter another element :y or nn

1 Insert an element
2 Delete an element
3 Search for an element
4 Exit
Enter your choice:
```

Did you Know?

Binary Search Tree time complexities

Search Operation - O(n)

Insertion Operation - O(1)

Deletion Operation - O(n)

Summary

- A tree structure is a way of presenting the hierarchical nature of a structure in a graphical form.
- The trees can be represented as graphs.
- The three types of graphs are directed graphs, undirected graphs, and mixed graphs.
- The different kinds of trees include binary tree, binary search tree, 2-3 tree, and Huffman tree.
- The two ways to represent trees are linked representation and array representation.
- The applications of trees include expression trees, game trees, and decision trees.
- To evaluate an expression tree, we recursively evaluate the left and right sub-trees and then apply the operator at the root.

Keywords

Binary Search Tree:- A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node.

Complete Binary Tree:-A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

Tree: - A tree data structure is a non-linear data structure because it does not store in a sequential manner.

SelfAssessment

1. Root is

A. The topmost node of a tree.

B. The child node of a tree.

C. The positive node of a tree.

D. None of above.

2. Degree of a Node is
A. The degree of a node is the X number of branches of that node.
B. The degree of a node is the X+Y number of branches of that node.
C. The degree of a node is the total number of branches of that node.
D. All of above.

3. Which of the following is not a type of tree data structure?

A. General Tree
B. Primary Tree
C. Binary Tree
D. Binary Search Tree

4. Tree is a nonlinear data structure.
A. True
B. False

5. Choose the right statement about binary tree.
A. Every node in a binary tree has a left and right reference along with the data element.
B. A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent.
C. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.
D. All of above.

6. Which of the following is not component of binary tree.
A. Data element
B. Pointer to left subtree
C. Pointer to right subtree
D. Super tree

7. Binary tree can be represented using
A. Arrays
B. Linked list
C. Both Array and Linked list
D. None of above

8. Which of the following is application of binary search tree.
A. In multilevel indexing in the database.
B. For dynamic sorting.
C. For managing virtual memory areas in Unix kernel.
D. All of above.

9. Which of the following is way of tree traversal?
A. In-order Traversal
B. Pre-order Traversal
C. Post-order Traversal

D. All of above

10. Select the traversal method in which root is visited after visit left sub-tree and right sub-tree.
A. In order Traversal
B. Pre-order Traversal
C. Post-order Traversal
D. None of above

11. Choose the tree traversal method from following
A. Recursive
B. Non Recursive
C. Both recursive and non-recursive
D. None of above

12. Post order traversal is

A. Traverse the left sub-tree, than traverse the right sub-tree, finally traverse the root
B. Traverse the root, than traverse the right sub-tree, finally traverse the left sub-tree
C. Traverse the left sub-tree, than traverse the right sub-tree
D. Traverse root only

13. Node within a data structure.

A. True
B. False

14. ______ is time complexity for insertion operation.
A. O(n)
B. O(1)
C. O(n)
D. None of above

15. O(n) is time complexity for
A. Search Operation
B. Insertion Operation
C. Deletion Operation
D. Merge operation

Answer for Self Assessment

1.	A	2.	C	3.	B	4.	A	5.	D
6.	D	7.	C	8.	D	9.	D	10.	C
11.	C	12.	A	13.	A	14.	B	15.	C

Review Questions

1. "Choosing the best data structure for a program is a challenging task". Discuss.

2. "The trees can be represented as graphs". Justify.

3. "There are different kinds of trees and it is important to understand the difference between them".

Analyze.

4. Explain binary search tree.

5. What is searching in tree?

6. How insertion performed in tree.

7. Differentiate between in-order traversal and pre-order traversal.

Further Readings

Books Ashok N. Kamthane, "Programming with ANCI & Turbo C", Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web links

https://www.gatevidyalay.com/tree-data-structure-tree-terminology/

https://www.javatpoint.com/searching-in-binary-search-tree

Unit 12: Graphs

CONTENTS

Objectives

After studying this unit, you will be able to:

- Understand breadth first search
- Learn depth first search
- Discuss network flow problems andwarshall's algorithm
- learntopological sort

Introduction

Graph traversal entails visiting each vertex and edge in a predetermined order. You must verify that each vertex of the graph is visited exactly once when utilizing certain graph algorithms. The sequence in which the vertices are visited is crucial, and it may be determined by the algorithm or question you're working on.It's critical to keep track of which vertices have been visited throughout a traversal. Marking vertices is the most popular method of tracking them.

In Graph traversal visiting every vertex and edge exactly once in a well-defined order.In graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited may depend upon the algorithm or type of problem going to solve.

Two common elementary algorithms for tree-searching are

– Breadth-first search (BFS)

– Depth-first search (DFS).

Both of these algorithms work on directed or undirected graphs. Many advanced graph algorithms are based on the ideas of BFS or DFS. Each of these algorithms traverses edges in the graph, discovering new vertices as it proceeds. The difference is in the order in which each algorithm discovers the edges.

Notes:

- Graph is an abstract data type.
- It is a pictorial representation of a set of objects where some pairs of objects are connected by links.
- Graph is used to implement the undirected graph and directed graph concepts from mathematics.
- It represents many real life application. Graphs are used to represent the networks. Network includes path in a city, telephone network etc.
- It is used in social networks like Facebook, LinkedIn etc.

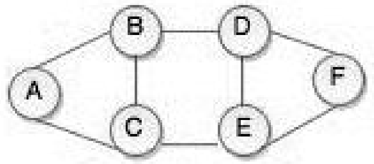
Graph Components

Graph consist of two components

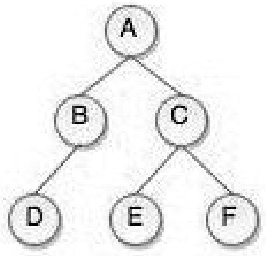
1. Vertices
2. Edges

- Graph is a set of vertices (V) and set of edges (E).
- V is a finite number of vertices also called as nodes.
- E is a set of ordered pair of vertices representing edges.

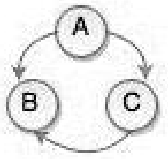
Example: In Facebook, each person is represented with a vertex or a node. Each node is a structure and contains the information like user id, user name, gender etc.



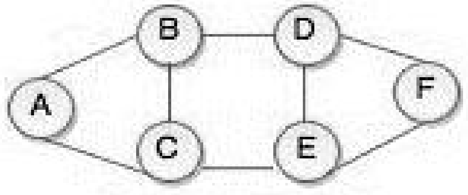
Graph 1



Graph 2



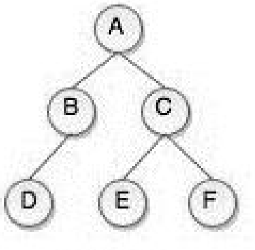
Graph 3



Graph 1

V = {A, B, C, D, E, F}

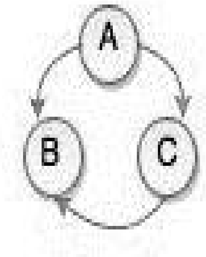
E = {(A, B), (A, C), (B, C), (B, D), (D, E), (D, F), (E, F)}



Graph 2

V = {A, B, C, D, E, F}

E = {(A, B), (A, C), (B, D), (C, E), (C, F)}



Graph 3

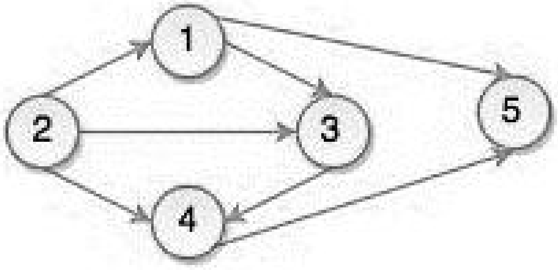
V = {A, B, C}

E = {(A, B), (A, C), (C, B)}

12.1 Types of Graphs

Directed Graph

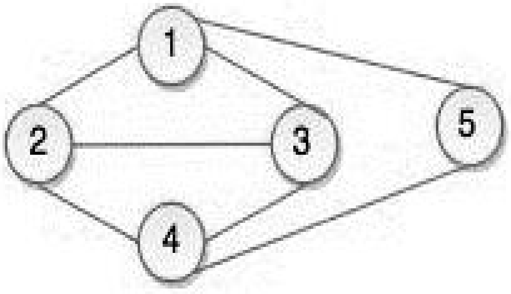
- If a graph contains ordered pair of vertices, is said to be a Directed Graph.
- If an edge is represented using a pair of vertices (V1, V2), the edge is said to be directed from V1 to V2.
- The first element of the pair V1 is called the start vertex and the second element of the pair V2 is called the end vertex.



- Set of Vertices V = {1, 2, 3, 4, 5, 5}
- Set of Edges W = {(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

Undirected Graph

- If a graph contains unordered pair of vertices, is said to be an Undirected Graph.
- In this graph, pair of vertices represents the same edge.
- In an undirected graph, the nodes are connected by undirected arcs.
- It is an edge that has no arrow. Both the ends of an undirected arc are equivalent, there is no head or tail.



- Set of Vertices V = {1, 2, 3, 4, 5}
- Set of Edges E = {(1, 2), (1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

12.2 Graph Terminology

- ***Path***

– A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

- ***Closed Path***

– A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V_0=V_N.

- ***Simple Path***

– If all the nodes of the graph are distinct with an exception V_0=V_N, then such path P is called as closed simple path.

- ***Cycle***

– A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

- ***Connected Graph***

– A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

- ***Complete Graph***

– A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

- ***Weighted Graph***

– In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

- ***Digraph***

– A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

- ***Loop***

– An edge that is associated with the similar end points can be called as Loop.

- *Adjacent Nodes*

– If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

- *Degree of the Node*

– A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

12.3 Applications of Graph

– In Computer science graphs are used to represent the flow of computation.

– Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.

– In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.

12.4 Graph Representation

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.There are two ways to store Graph into the computer's memory.

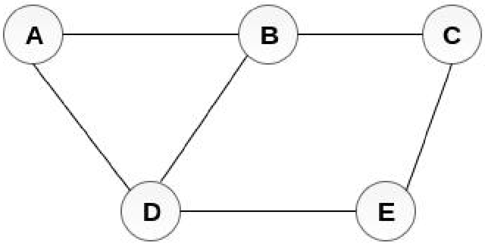
1. Sequential Representation
2. Linked Representation

1. Sequential Representation

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension n x n.

Adjacency Matrix

- An undirected graph and its adjacency matrix representation is shown in the following figure.

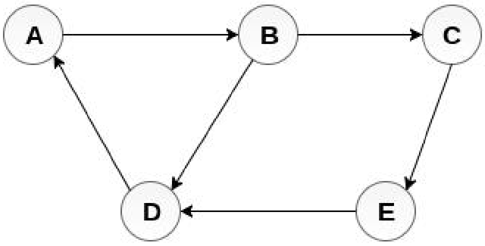


Undirected Graph

$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix

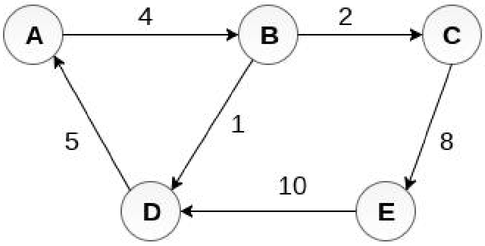
- A directed graph and its adjacency matrix representation is shown in the following figure.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

Directed Graph **Adjacency Matrix**

- The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

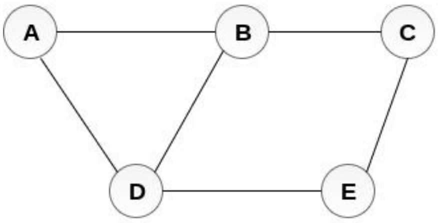
Weighted Directed Graph **Adjacency Matrix**

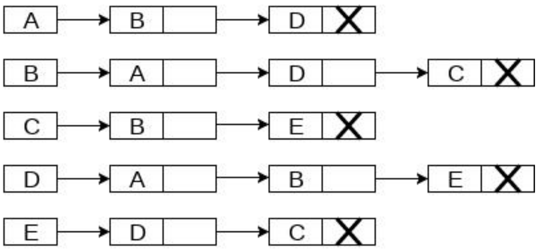
2. Linked Representation

In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Linked Representation

- Consider the undirected graph shown in the following figure and check the adjacency list representation.

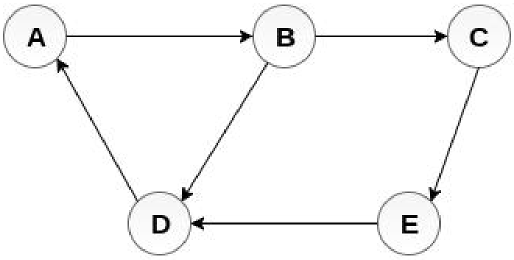




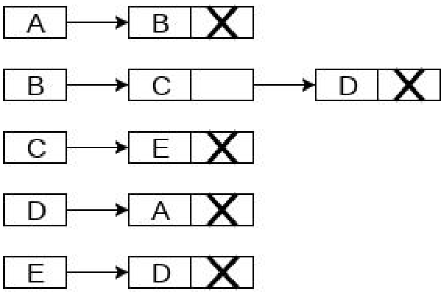
Undirected Graph **Adjacency List**

- An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node.

- If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.
- Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.

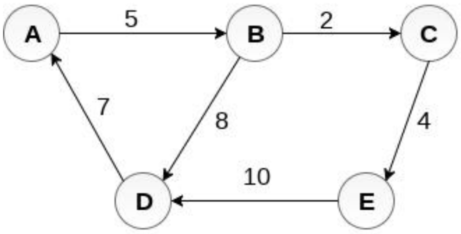


Directed Graph

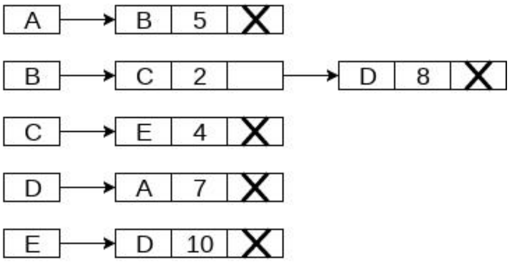


Adjacency List

- In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



Adjacency List

12.5 Breadth First Search (BFS)

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

For using BFS algorithm user should know about data structure queue and its relevant operations like en-queue and de-queue.

Algorithm: Breadth First Search

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
```

```
Step 4: Dequeue a node N. Process it
            and set its STATUS = 3
            (processed state).
Step 5: Enqueue all the neighbours of
            N that are in the ready state
            (whose STATUS = 1) and set
            their STATUS = 2
            (waiting state)
[END OF LOOP]
            Step 6: EXIT
```

Example: Breadth First Search

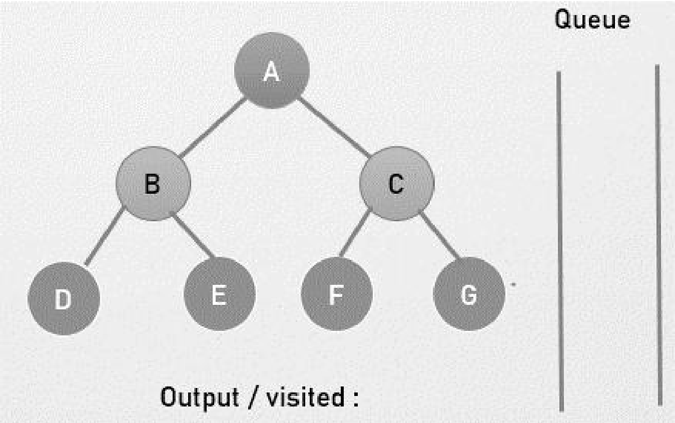


Fig (a)

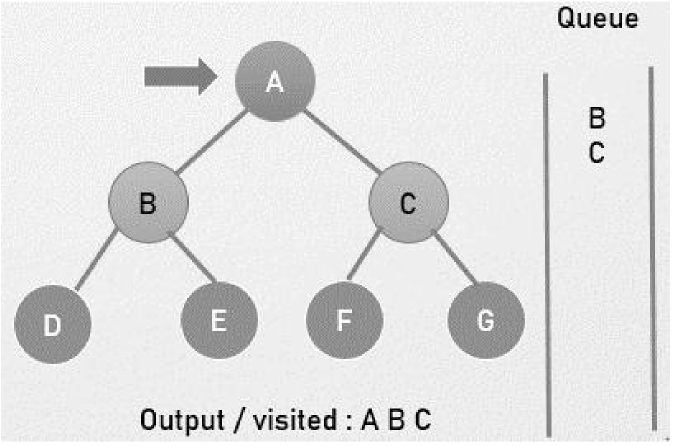


Fig (b)

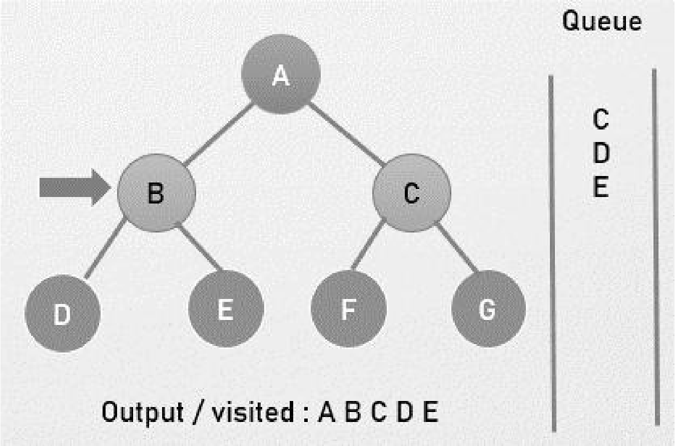


Fig (c)

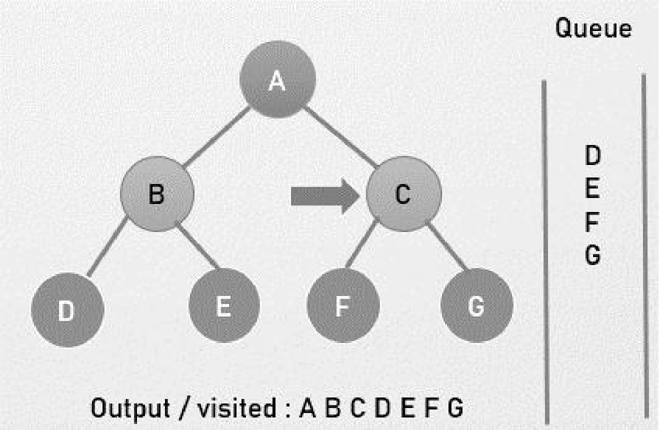


Fig (d)

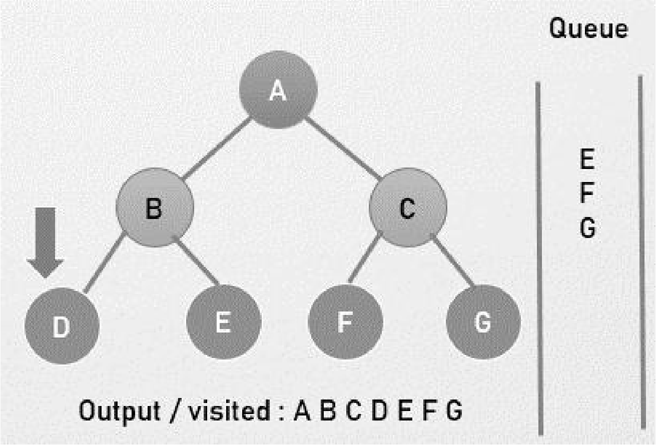


Fig (e)

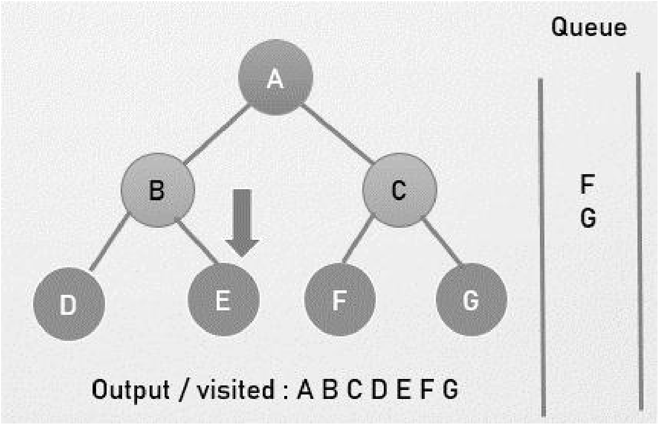


Fig (f)

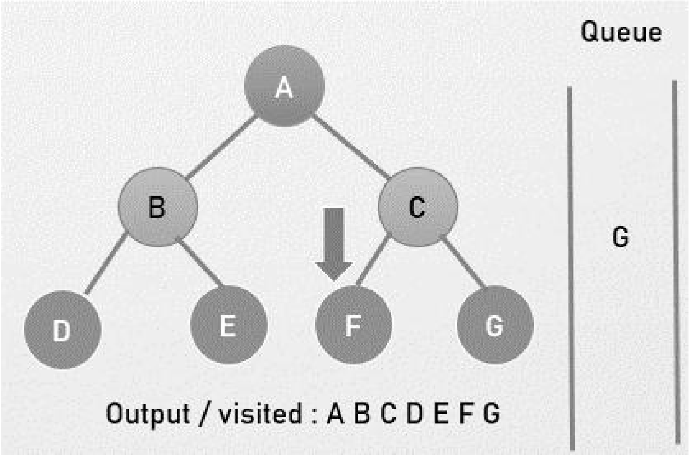


Fig (g)

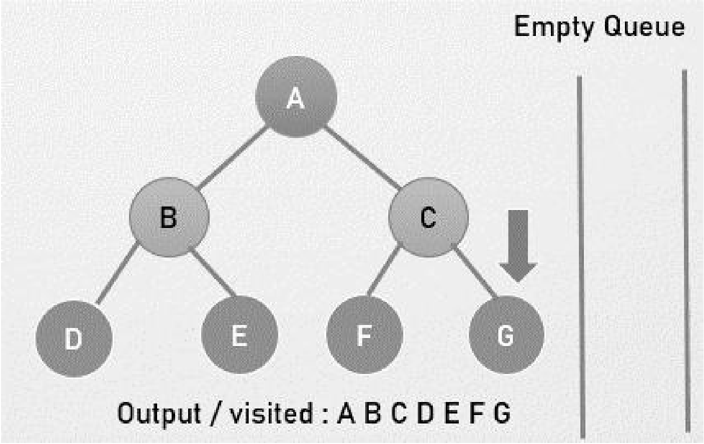


Fig (h)

Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

BFS Applications

- Path finding algorithms

- To build index by search index
- Cycle detection in an undirected graph
- For GPS navigation
- In minimum spanning tree
- Social networking websites

12.6 Depth First Search

Depth first search is another way of traversing graphs, which is closely related to preorder traversal of a tree. Recall that preorder traversal simply visits each node before its children. It is most easy to program as a recursive routine.

DFS traversal is a recursive algorithm for searching all the vertices/ nodes of a graph or tree using stack data structure.In Depth First Search (DFS) algorithm traverses a graph in a depth ward motion.The DFS algorithm use the concept of backtracking.Depth-first search (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.Often implemented recursively. Many graph algorithms involve visiting or marking vertices.

Steps for DFS

Step 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Step 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Step 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

For using DFS algorithm user should know about data structure Stack (Last In First Out) and its relevant operations like Push and Pop.

Algorithm: Depth First Search

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their
        STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Example: Depth First Search

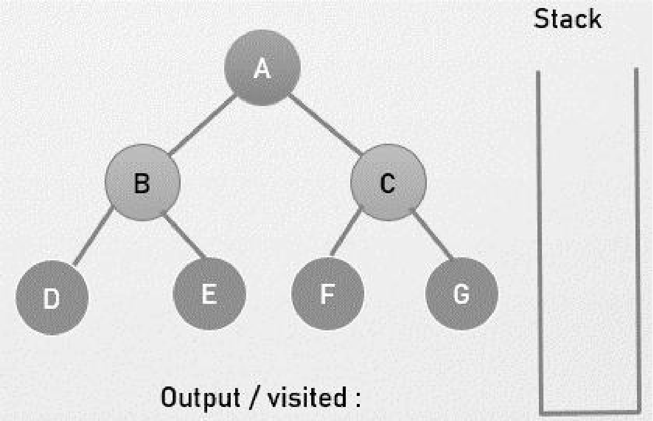


Fig (a)

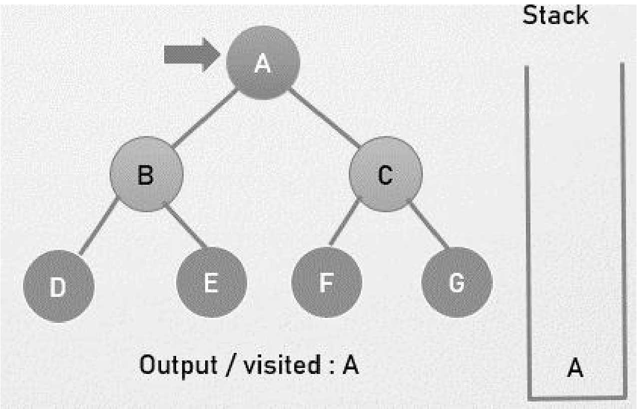


Fig (b)

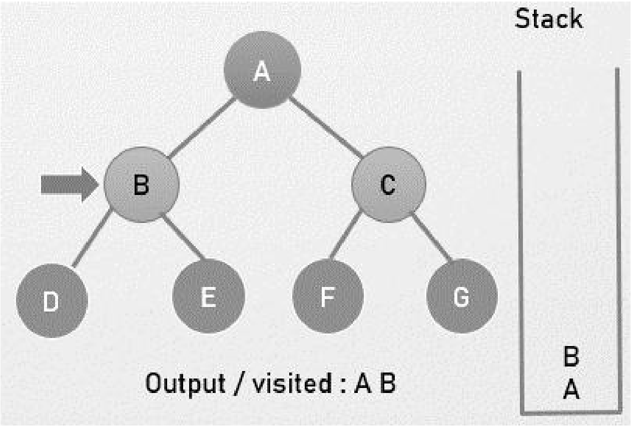


Fig (c)

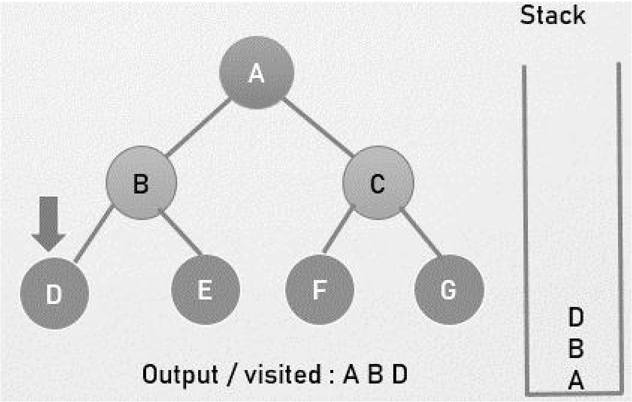


Fig (d)

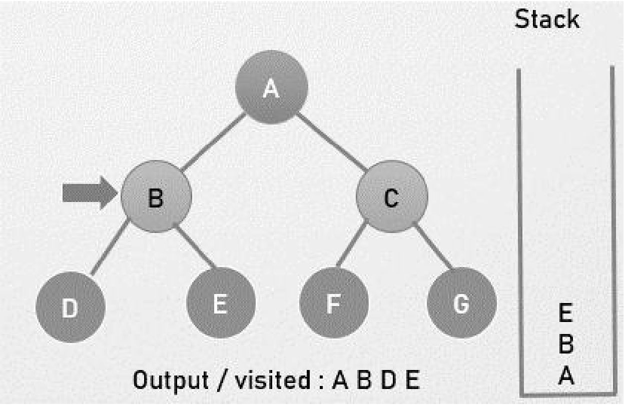


Fig (e)

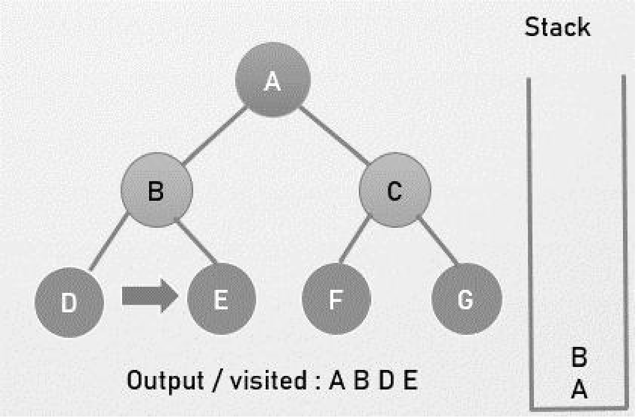


Fig (f)

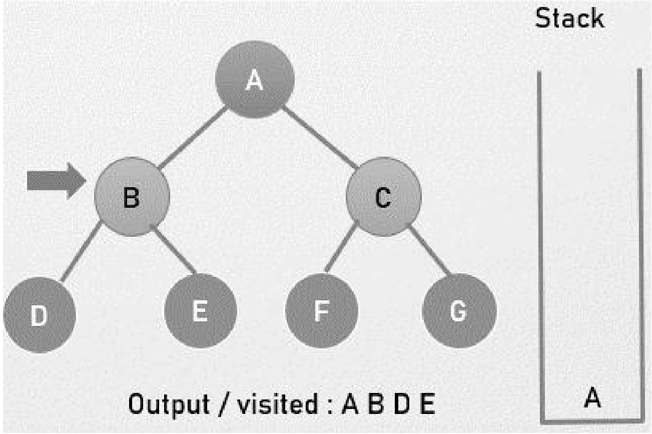


Fig (g)

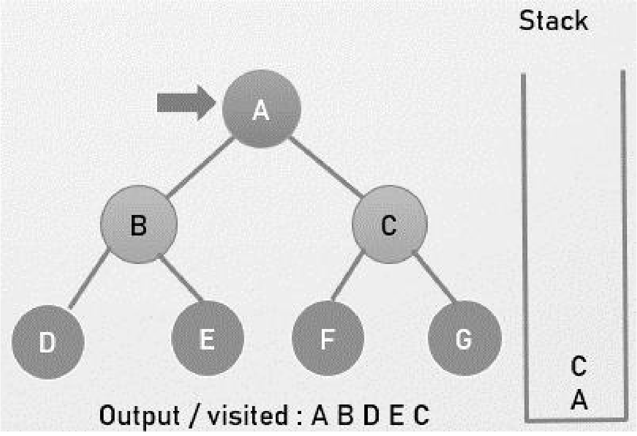


Fig (h)

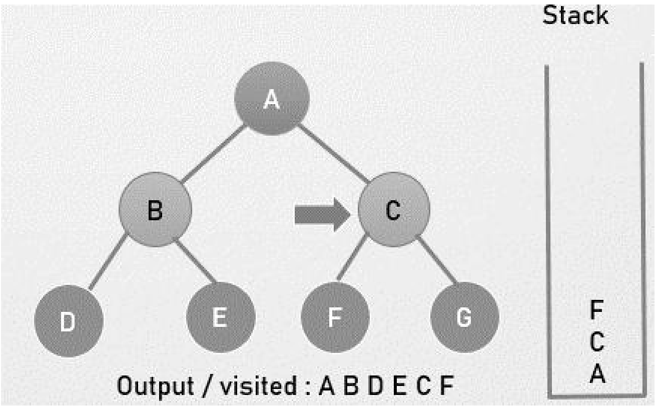


Fig (i)

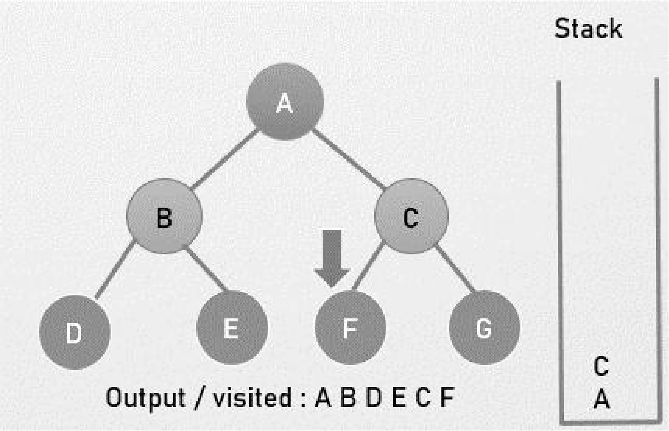


Fig (j)

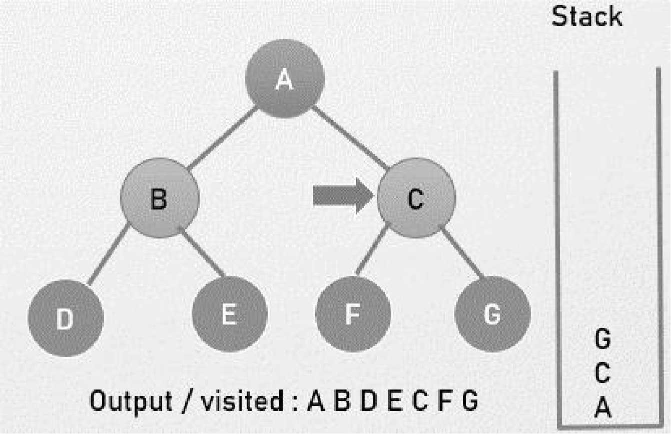


Fig (k)

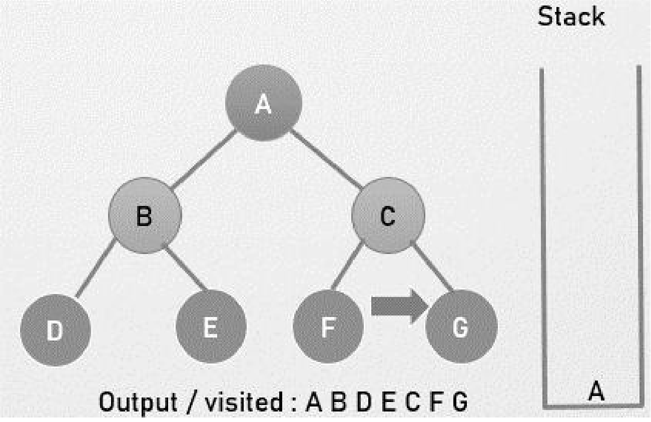


Fig (l)

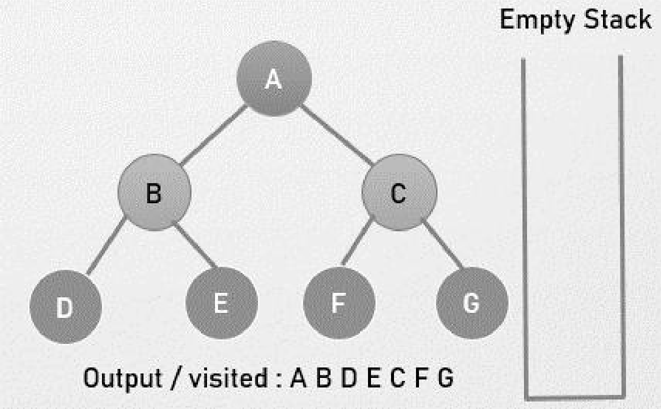


Fig (m)

Algorithm Complexity

Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

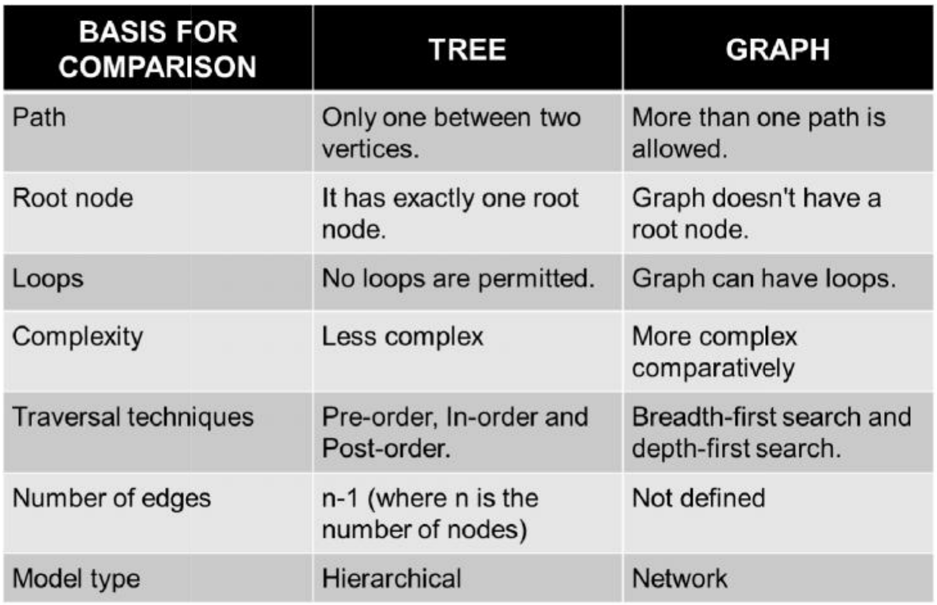
Space Complexity: O(V).

DFS Applications

- Mapping Routes and Network Analysis.
- Path Finding.

- Cycle detection in graphs.
- Topological Sorting.
- Solving puzzle.

12.7 Difference Between Tree and Graph



BASIS FOR COMPARISON	TREE	GRAPH
Path	Only one between two vertices.	More than one path is allowed.
Root node	It has exactly one root node.	Graph doesn't have a root node.
Loops	No loops are permitted.	Graph can have loops.
Complexity	Less complex	More complex comparatively
Traversal techniques	Pre-order, In-order and Post-order.	Breadth-first search and depth-first search.
Number of edges	n-1 (where n is the number of nodes)	Not defined
Model type	Hierarchical	Network

Summary

- Graphs provide in excellent way to describe the essential features of many applications.
- Graphs are mathematical structures and are found to be useful in problem solving. They may be implemented in many ways by the use of different kinds of data structures.
- Graph traversals, Depth first as well as Breadth First, are also required in many applications.
- Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighboring nodes.
- DFS traversal is a recursive algorithm for searching all the vertices/ nodes of a graph or tree using stack data structure.

Keywords

Breadth-first search: -Breadth-first search is an algorithm for searching a tree data structure for a node that satisfies a given property.

Depth-first search:-Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children.

Graph: - A graph can be defined as a collection of vertices and the edges that connect them. A graph is a cyclic tree in which the vertices (Nodes) preserve any complex relationship between them rather than having a parent-child relationship.

Self Assessment

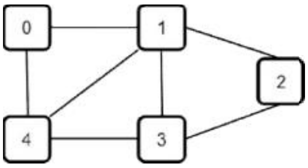
1. A simple graph does not have which of the following properties?
A. Must be connected
B. Must be unweighted
C. Must have no loops or multiple edges
D. Must have no multiple edges

2. A graph consist of the following components
A. Vertices
B. Edges
C. Both edges and vertices
D. None of above

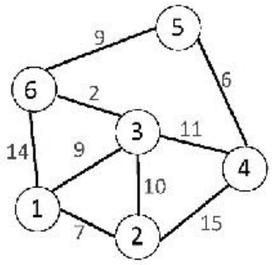
3. Which of the following is true?
A. A graph may contain no edges and many vertices
B. A graph may contain many edges and no vertices
C. A graph may contain no edges and no vertices
D. A graph may contain no vertices and many edges

4. Which of the following is not a graph?

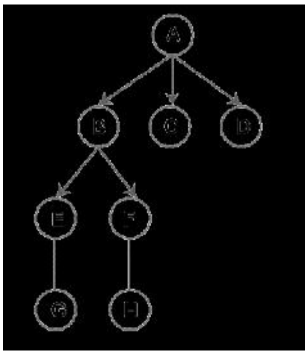
a)

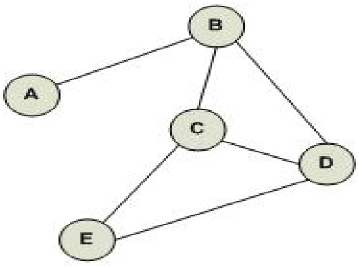


b)



c)



d) 

5. directed graph is
A. A graph contains ordered pair of vertices.
B. A graph contains 2 pair of vertices.
C. A graph contains vertices.
D. None of the above

6. The Depth First Search traversal of a graph will result into?
A. Linked List
B. Tree
C. Queue
D. Array

7. The data structure which is being used in DFS is ______.
A. Stack
B. Tree
C. Queue
D. None of above

8. Choose the incorrect statement about DFS and BFS from the following.
A. BFS is equivalent to level order traversal in trees
B. DFS is equivalent to post order traversal in trees
C. DFS and BFS code has same time complexity
D. DFS is implemented using stack

9. Choose the correct statement from following
A. Depth-first search is an algorithm for insert node into tree or graph data structures.
B. Depth-first search is an algorithm for traversing or searching tree or graph data structures.
C. DFS is delete first style from data structure.
D. Depth-first search is an algorithm for count elements in the data structure.

10. In Depth First Search, how many times a node is visited?
A. Once
B. Twice
C. Thrice
D. Equivalent to number of in-degree of the node

11. The Data structure used in standard implementation of Breadth First Search is?
A. Stack
B. Queue

C. Linked List

D. Tree

12. What can be the applications of Breadth First Search?

A. Finding shortest path between two nodes

B. Finding bipartiteness of a graph

C. GPS navigation system

D. All of the mentioned

13. Choose the correct statement from following

A. Breadth-first search is an algorithm for searching a tree data structure for a node that satisfies a given property.

B. Breadth-first search is an algorithm used for insertion.

C. Breadth-first search is an algorithm for delete an element from array.

D. All of above

14. Which of the following is not an application of Breadth First Search?

A. Finding shortest path between two nodes

B. Finding bipartiteness of a graph

C. GPS navigation system

D. Path Finding

15. The data structure which is being used in DFS is stack.

A. True

B. False

Answersfor self Assessment

l.	A	2.	C	3.	B	4.	D	5.	A
6.	B	7.	A	8.	B	9.	B	10.	D
11.	B	12.	D	13.	A	14.	D	15.	B

Review Questions

1. “A graph in which each edge is assigned a direction is called a directed graph”. Discuss withexample.
2. “A graph consists of a set of vertices and edges/arcs”. Explain with diagram.
3. Write and explain breadth first search algorithm.
4. Write and explain depth first search algorithm.
5. Differentiate between tree and graph.
6. Discuss in detail applications of graph data structure.
7. Explain how graphs are different from tree.

Further Readings

Books Ashok N. Kamthane, “Programming with ANCI & Turbo C”, Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web Links

https://www.simplilearn.com/tutorials/data-structure-tutorial/graphs-in-data-structure

https://www.freecodecamp.org/news/a-gentle-introduction-to-data-structures-how-graphs-work-a223d9ef8837/

https://www.tutorialspoint.com/data_structures_algorithms/graph_data_structure.htm

Unit 13: Searching

CONTENTS

Objectives

After studying this unit, you will be able to:

- Understand Searching
- Explain linear search
- Analyze complexity
- Define binary search

Introduction

The process of discovering the location LOC of an element in a list is referred to as searching in a data structure. This is a crucial feature of many data structure algorithms, because we can only conduct one operation on an element if and only if we discover it. To determine whether an element is present in a collection of objects, various algorithms have been defined. Both internal and external data structures can be used with this approach. Any algorithm's efficiency is improved by the efficiency of searching for an element.

Did you Know?

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.

13.1 Search Techniques in Data structure

1. Linear Search
2. Binary Search

1. Linear Search

This is the traditional technique for searching an element in a collection of elements. In this type of search, all the elements of the list are traversed one by one to find if the element is present in the list or not. One example of such an algorithm is a linear search. This is a straightforward and basic algorithm. Suppose ARR is an array of n elements, and we need to find location LOC of element ITEM in ARR. For this, LOC is assigned to -1, which indicates that ITEM is not present in ARR. While comparing ITEM with data at each ARR location, and once ITEM == ARR[N], LOC is updated with location N+1. Hence we found the ITEM in ARR.

Algorithm

LSEARCH(ARR, N, ITEM, LOC) Here ARR Is the array of N number of elements, ITEM holds the value we need to search in the array and algorithm returns LOC, the location where ITEM is present in the ARR. Initially, we have to set LOC = -1.

```
1. Set LOC = -1,i=1
2. Repeat while DATA[i] != ITEM:
        i=i+1
3. If i=N+1 ,then Set LOC =0
        Else LOC = N+1
4. Exit.
```

Example: Let's say, below is the ARR with 5 elements. And we need to find whether ITEM= 12 is present in this array or not.

22	32	12	35	65

In the start, LOC =-1

Step 1: ITEM != 22 thus we move to next element.

22	32	12	35	65

12==

Step 2: ITEM != 32 thus we move to next element.

22	32	12	35	65

12==

Step 5: Hence ITEM == ARR[2] thus LOC updated to 3.

22	32	12	35	65

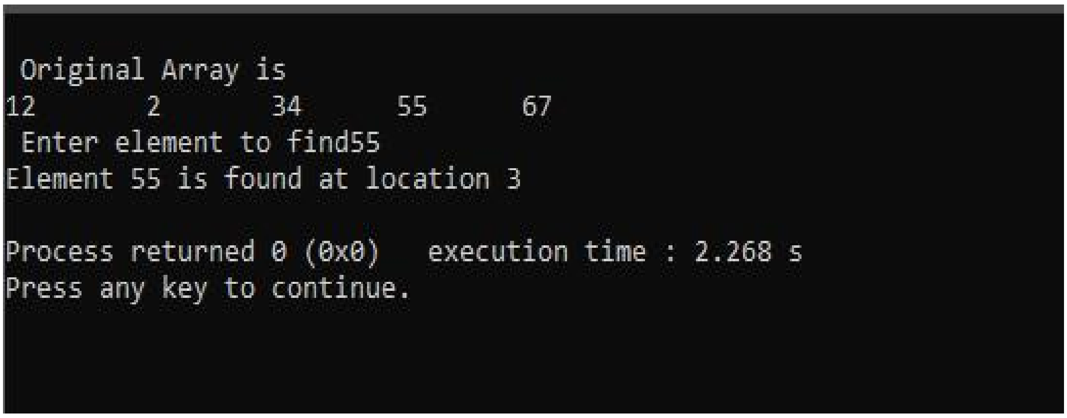
12==

Lab Exercise

```
// Program
#include<stdio.h>
int main(){
```

```
int a[5]={12,2,34,55,67};
int i, toSearch,flag=0;
printf("\n Original Array is \n");
for(i=0;i<5;i++){
  printf("%d \t ", a[i]);
}
printf("\n Enter element to find");
scanf("%d",&toSearch);
for(i=0;i<5;i++){
  if(a[i]==toSearch)
  {
    flag=1;
break;
  }
  else
    flag=0;
  }
  if(flag!=0)
    printf("Element %d is found at location %d\n",toSearch,i);
  else
    printf("\n Item not found\n");
return 0;
}
```

Output



```
 Original Array is
12      2       34      55      67
 Enter element to find55
Element 55 is found at location 3

Process returned 0 (0x0)   execution time : 2.268 s
Press any key to continue.
```

13.2 The Complexity of Sequential Search

Here are the complexities of the linear search given below.

Space complexity

As linear search algorithm does not use any extra space, thus its space complexity = O(n) for an array of n number of elements.

Time Complexity

Worst-case complexity: O(n) – This case occurs when the search element is not present in the array.

Best case complexity: O(1) – This case occurs when the first element is the element to be searched.

Average complexity: O(n) – This means when an element is present somewhere in the middle of the array.

2. Binary Search

This is a technique to search an element in the list using the divide and conquer technique. This type of technique is used in the case of sorted lists. Instead of searching an element one by one in the list, it directly goes to the middle element of the list, divides the array into 2 parts, and decides element lies in which sub-array the element exists.

Suppose ARR is an array with sorted n number of elements present in increasing order. With every step of this algorithm, the searching is confined within BEG and END, which are the beginning and ending index of sub-arrays. The index MID defines the middle index of the array where,

MID = INT(beg + end)/2

It needs to be checked if ITEM < ARR[N} where ITEM is the element that we need to search in ARR.

- If ITEM = ARR[MID] then LOC = MID and exit .
- If ITEM < ARR[MID} then ITEM can appear in the left sub-array, then BEG will be the same and END = MID -1 and repeat.
- If ITEM > ARR[MID] then ITEM can appear in the right subarray then BEG = MID+1 and END will be the same and repeat.

After this MID is again calculated for respective sub-arrays, if we didn't find the ITEM, the algorithm returns -1 otherwise LOC = MID.

Algorithm:

BSEARCH(ARR, LB, UB, ITEM, LOC) Here, ARR is a sorted list of elements, with LB and UB are lower and upper bounds for the array. ITEM needs to be searched in the array and algorithm returns location LOC, index at which ITEM is present else return -1.

```
1. Set BEG = LB, END = UB and MID = INT([BEG+END]/2)
2. Repeat step 3 and 4 while BEG <= END and ARR[MID] != ITEM
3. IF ITEM< ARR[MID] then:
    Set END = MID-1
   Else:
    Set BEG = MID+1
4. Set MID = INT(BEG+END)/2
5. IF ARR[MID] = ITEM then:
    Set LOC = MID
```

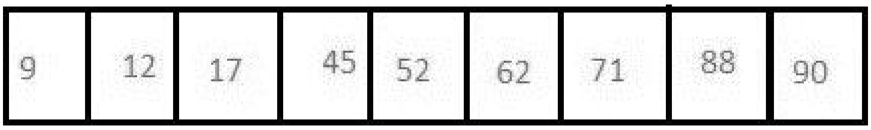
Else:

Set LOC = NULL

6. Exit.

Example:

Let’s say here, ITEM = 62

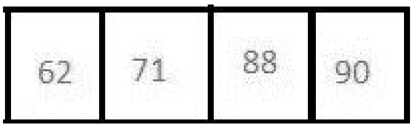


9	12	17	45	52	62	71	88	90

BEG = 1 and END =9 Hence MID = (1+9)/2 = 5

ARR[MID] = 52

Step 1: ARR[MID] < ITEM : thus END =9 and BEG = MID +1 = 6. Thus our new sub-array is,



62	71	88	90

Step 2: Now BEG =6 and END =9 thus MID = INT([6+9]/2)= 6

NOW ARR[6] =ITEM. Thus LOC = MID

Thus LOC = 6

The complexity of Binary Search

Here are the complexities of the binary search given below.

Worst Case: O(nlogn)

Best Case: O(1)

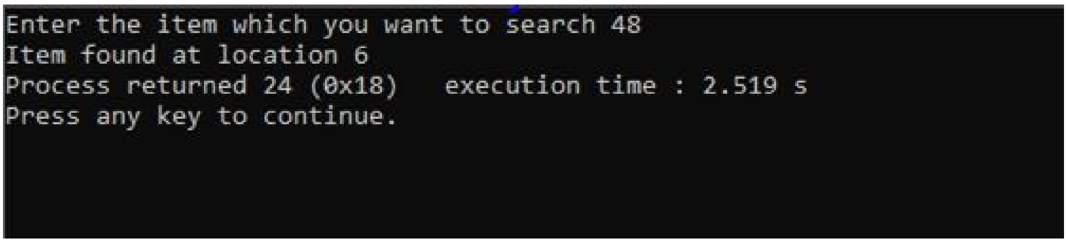
Average Case: O(nlogn)

Lab Exercise

```
// Program
#include<stdio.h>
int binarySearch(int[], int, int, int);
void main ()
{
   int arr[10] = {10, 12, 21, 23, 40, 48, 50, 78, 90, 96, 100};
   int item, location=-1;
   printf("Enter the item which you want to search ");
scanf("%d",&item);
```

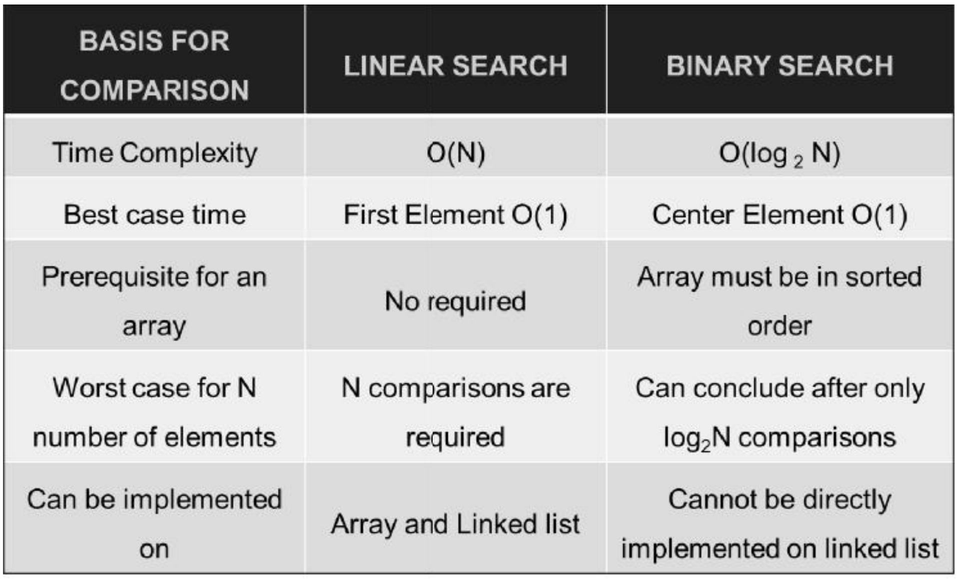
```
    location = binarySearch(arr, 0, 9, item);
    if(location != -1)
    {
        printf("Item found at location %d",location);
    }
    else
    {
        printf("Item not found");
    }
}
int binarySearch(int a[], int beg, int end, int item)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
            return binarySearch(a,beg,mid-1,item);
        }

    }
    return -1;
}
```

Output



```
Enter the item which you want to search 48
Item found at location 6
Process returned 24 (0x18)   execution time : 2.519 s
Press any key to continue.
```

Linear Search Vs. Binary Search



BASIS FOR COMPARISON	LINEAR SEARCH	BINARY SEARCH
Time Complexity	O(N)	O(log $_2$ N)
Best case time	First Element O(1)	Center Element O(1)
Prerequisite for an array	No required	Array must be in sorted order
Worst case for N number of elements	N comparisons are required	Can conclude after only log$_2$N comparisons
Can be implemented on	Array and Linked list	Cannot be directly implemented on linked list

Key Differences Between Linear Search and Binary Search

1. Linear search is iterative in nature and uses a sequential approach. On the other hand, Binary search implements a divide and conquer approach.
2. The best-case time in linear search is for the first element, i.e., O (1). As against, in binary search, it is for the middle element, i.e., O (1).
3. Linear search can be implemented in an array as well as in a linked list, whereas binary search cannot be implemented directly in a linked list.
4. Linear search is easy to use, and there is no need for any ordered elements. On the other hand, the binary search algorithm is tricky, and elements are necessarily arranged in order.

Summary

- Search is a programme that allows you to look for papers, files, and other forms of information. A binary search is a form of advanced search method for finding and retrieving data from a sorted list of things.
- Binary search is commonly known as a half-interval search or a logarithmic search.
- A binary search is not suitable for unsorted data.
- The linear search is simple to use, or we could say less complex, because the components in a linear search can be put in any sequence, whereas the elements in a binary search must be arranged in a specific order.
- The implementation of binary search is limited as it can be implemented only on those data structures that have two-way traversal.
- In a linear search, the worst- case scenario for finding the element is O(n).

Keywords

Searching: Searching in data structure refers to the process of finding location LOC of an element in a list.

Linear Search: This is the traditional technique for searching an element in a collection of elements. In this type of search, all the elements of the list are traversed one by one to find if the element is present in the list or not.

Binary Search: This is a technique to search an element in the list using the divide and conquer technique. This type of technique is used in the case of sorted lists. Instead of searching an element one by one in the list, it directly goes to the middle element of the list, divides the array into 2 parts, and decides element lies in which sub-array the element exists.

Complexity: Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

Self Assessment

1. Which of the following is not a searching technique?

A. Linear Search
B. Binary Search
C. Queue Search
D. None of above

2. Where is linear searching used?

A. When the list has only a few elements
B. When performing a single search in an unordered list
C. Used all the time
D. When the list has only a few elements and when performing a single search in an unordered list

3. What is the best case for linear search?

A. O(nlogn)
B. O(logn)
C. O(n)
D. O(1)

4. What is the worst case for linear search?

A. O(nlogn)
B. O(logn)
C. O(n)
D. O(1)

5. _________ Search is often called sequential search.

A. Binary Search
B. Linear Search
C. Both linear search and binary search
D. None of above

6. Linear search is mostly used to search an unordered list in which the items are not sorted.

A. True
B. False
C. Sometimes true

D. Sometimes false

7. Finding the location of a given item in a collection of items is called......
A. Discovering
B. Finding
C. Searching
D. Mining

8. Which of the following algorithm type is iterative in nature?

A. Binary Search
B. Linear Search
C. Both binary search and linear search
D. None of the above

9. The binary search algorithm uses
A. Linear way to search values
B. Divide and conquer method
C. Bubble sorting technique
D. None of them

10. Best case complexity of binary search algorithm is
A. O(1)
B. O(log n)
C. O(log n)
D. O(1)

11. Average case complexity of binary search algorithm is
A. O(1)
B. O(log n)
C. O(log n)
D. O(1)

12. Average case complexity and worst case complexity of binary search algorithm are same
A. True
B. False

13. Worst case complexity of binary search algorithm is
A. O(1)
B. O(log n)
C. O(log n)
D. O(1)

14. Which of the following is right statement for binary search
A. Binary search is a fast search algorithm with a run-time complexity of (log n).
B. Binary search is a fast search algorithm with a run-time complexity of (m log n).
C. Binary search is a fast search algorithm with a run-time complexity of (x log n).

D. Binary search is a fast search algorithm with a run-time complexity of (log X).

15. Which from the following technique is used for finding a value from an array?

A. Linear Search

B. Binary Search

C. Bubble sort

D. All above

Answer for Self Assessment

1.	C	2.	D	3.	D	4.	C	5.	B
6.	A	7.	C	8.	B	9.	B	10.	A
11.	B	12.	A	13.	C	14.	A	15.	D

Review Questions

1. What is searching? Explain different types of searching in data structure.
2. Differentiate between linear search and binary search.
3. Linear search also known as sequential search. Comment.
4. What is the role of searching in data structure?
5. What are applications of searching?

Further Readings

Books Ashok N. Kamthane, “Programming with ANCI & Turbo C”, Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

YashvantKanetkar, Let us C

Web Links

https://www.javatpoint.com/ds-linear-search-vs-binary-search

https://en.wikipedia.org/

https://onlinedegree.iitm.ac.in/

Unit 14: Sorting

CONTENTS

Objectives

After studying this unit, you will be able to:

- Understand Sorting
- Define sorting algorithms
- Analyze sorting techniques

Introduction

Sorting is the process of arranging data in a preferred order in a data structure. Sorting data makes it easier to swiftly and simply search through it. A dictionary is the most basic example of sorting. When you wanted to look up a term in a dictionary before the Internet, you had to do it in alphabetical order. This made things a lot easier.

Imagine the stress of having to sift through a large book containing all of the English words from around the world in a jumbled sequence! It's the same dread that an engineer will feel if their data isn't organized and categorized.

14.1 Sorting Algorithm

Sorting Algorithms are methods of reorganizing a large number of items into some specific order such as highest to lowest, or vice-versa, or even in some alphabetical order.

These algorithms take an input list, processes it (i.e, performs some operations on it) and produce the sorted list.

The most common example we experience every day is sorting clothes or other items on an e-commerce website either by lowest-price to highest, or list by popularity, or some other order.

Did you know?

- Sorting is the process of ordering or placing a list of elements from a collection in some kind of order. It is nothing but the storage of data in sorted order.
- Sorting can be done in ascending and descending order.
- It arranges the data in a sequence, which makes searching easier.

Example: Let's suppose you have an array of strings: [h,j,k,i,n,m,o,l]

Now, sorting would yield an output array in alphabetical order.

Output: [h,i,j,k,l,m,n,o]

Sorting Categories

There are two different categories in sorting:

Internal sorting: If the input data is such that it can be adjusted in the main memory at once, it is called internal sorting.

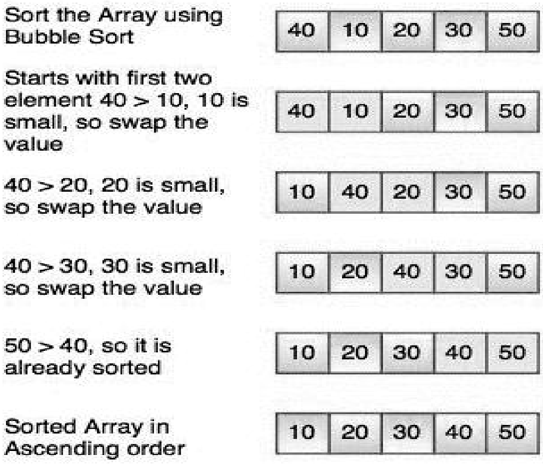
External sorting: If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting.

Types of Sorting in Data Structure

1. Quick Sort
2. Bubble Sort
3. Merge Sort
4. Insertion Sort
5. Selection Sort
6. Heap Sort
7. Radix Sort
8. Bucket Sort

14.2 Bubble Sort

Bubble sort, also referred to as comparison sort, is a simple sorting algorithm that repeatedly goes through the list, compares adjacent elements and swaps them if they are in the wrong order. This is the simplest algorithm and inefficient at the same time. Yet, it is very much necessary to learn about it as it represents the basic foundations of sorting.


Sort the Array using Bubble Sort 40 10 20 30 50
Starts with first two element 40 > 10, 10 is small, so swap the value 40 10 20 30 50
40 > 20, 20 is small, so swap the value 10 40 20 30 50
40 > 30, 30 is small, so swap the value 10 20 40 30 50
50 > 40, so it is already sorted 10 20 30 40 50
Sorted Array in Ascending order 10 20 30 40 50

The diagram represents how bubble sorting actually works. This sort takes O (n2) time. It starts with the first two elements and sorts them in ascending order.Bubble sort starts with the first two elements. It compares the elements to check which one is greater.In the above diagram, element 40 is greater than 10, so these values must be swapped. This operation continues until the array is sorted in ascending order.

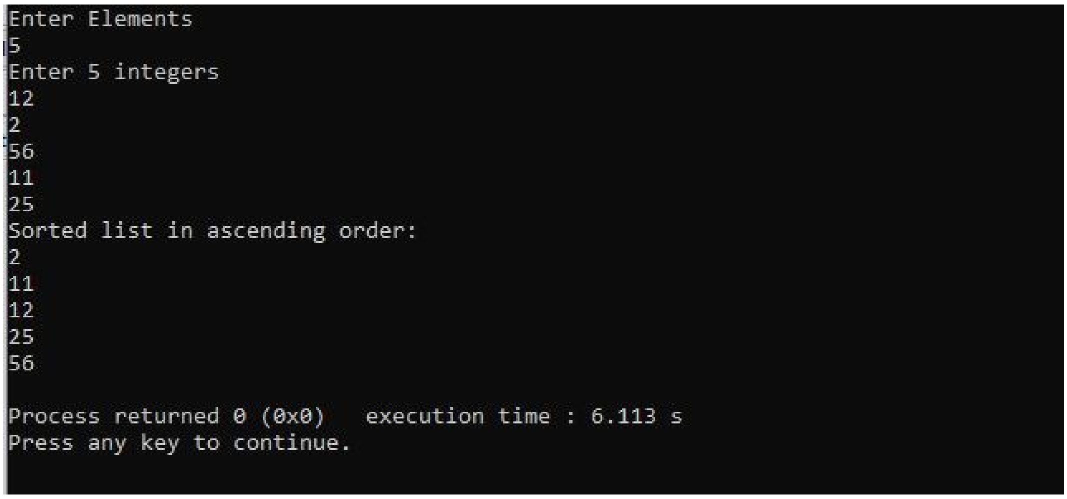
Algorithm

```
bubbleSort( Arr[], totat_elements)


  for i = 0 to total_elements - 1 do:
    swapped = false


    for j = 0 to total_elements - i - 2 do:



      if Arr[j] > Arr[j+1] then


        swap(Arr[j], Arr[j+1])
        swapped = true
      end if


    end for


    if(not swapped) then
      break
    end if


  end for


end
```

Lab Exercise:

```
#include <stdio.h>
void bubble_sort(long [], long);
int main()
{
  long array[100], n, c, d, swap;
  printf("Enter Elements\n");
  scanf("%ld", &n);
  printf("Enter %ld integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%ld", &array[c]);
  bubble_sort(array, n);
  printf("Sorted list in ascending order:\n");
  for ( c = 0 ; c < n ; c++ )
     printf("%ld\n", array[c]);
  return 0;
}
void bubble_sort(long list[], long n)
{
  long c, d, t;
  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        t        = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

Output



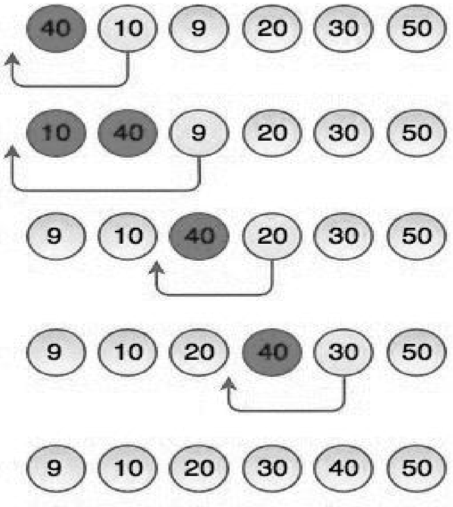
```
Enter Elements
5
Enter 5 integers
12
2
56
11
25
Sorted list in ascending order:
2
11
12
25
56

Process returned 0 (0x0)   execution time : 6.113 s
Press any key to continue.
```

14.3 Insertion Sort

Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.

Did you Know?

- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- Insertion sort has one of the simplest implementations.
- This sort is efficient for smaller data sets, but it is insufficient for larger lists.
- It has less space complexity than bubble sort.
- It requires a single additional memory space.
- Insertion sort does not change the relative order of elements with equal keys because it is stable.



The above diagram represents how insertion sorting works. Insertion sorting works like the way we sort playing cards in our hands. It always starts with the second element as the key. The key is to compare it with the elements ahead of it and put it in the right place.In the above figure, 40 has nothing before it. Element 10 is compared to 40 and is inserted before 40. Element 9 is smaller than 40 and 10, so it is inserted before 10 and this operation continues until the array is sorted in ascending order.

Algorithm

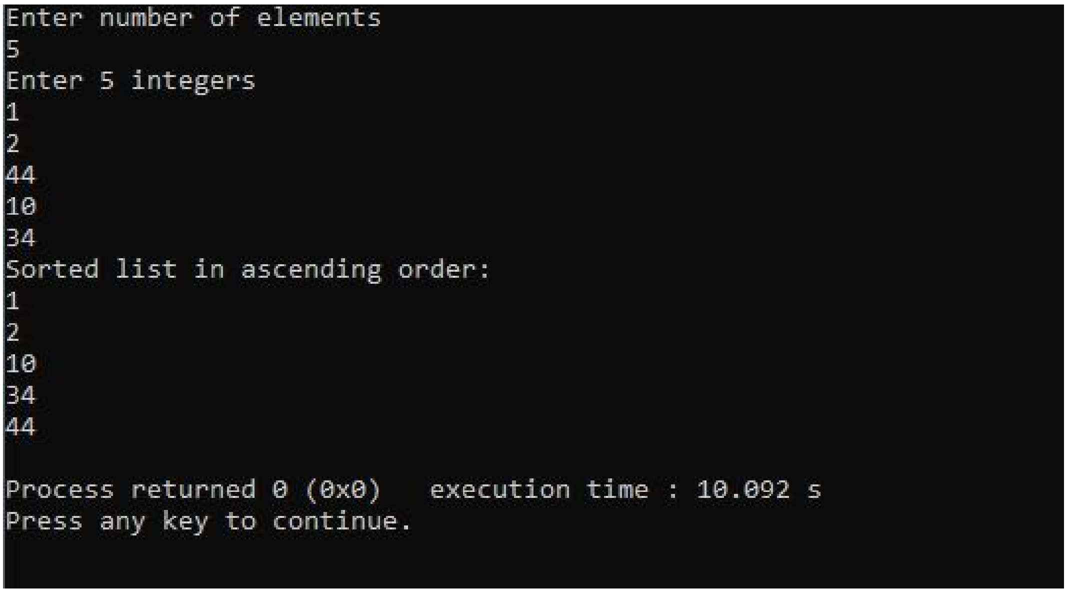
```
INSERTION-SORT(A)
  for i = 1 to n
          key ← A [i]
          j ← i – 1
           while j > = 0 and A[j] > key
                      A[j+1] ← A[j]
                      j ← j – 1
          End while
          A[j+1] ← key
 End for
```

Lab Exercise

```
#include <stdio.h>
int main()
{
 int n, array[1000], c, d, t;
 printf("Enter number of elements\n");
 scanf("%d", &n);
 printf("Enter %d integers\n", n);
 for (c = 0; c < n; c++)
 {
   scanf("%d", &array[c]);
 }
 for (c = 1 ; c <= n - 1; c++)
 {
d = c;
   while ( d > 0 && array[d] < array[d-1])
   {
      t           = array[d];
      array[d]    = array[d-1];
      array[d-1] = t;
      d--;
```

```
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c <= n - 1; c++)
  {
     printf("%d\n", array[c]);
  }
  return 0;
}
```

Output



```
Enter number of elements
5
Enter 5 integers
1
2
44
10
34
Sorted list in ascending order:
1
2
10
34
44

Process returned 0 (0x0)   execution time : 10.092 s
Press any key to continue.
```

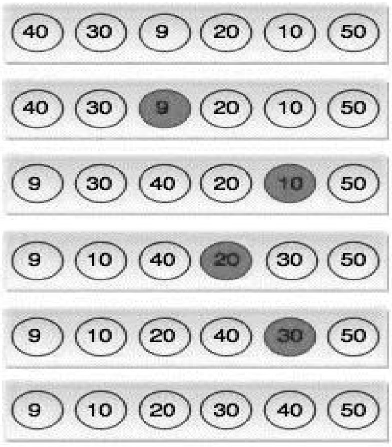
14.4 Selection Sort

Selection sort is a simple comparison-based sorting algorithm. It is in-place and needs no extra memory. The idea behind this algorithm is pretty simple. We divide the array into two parts: sorted and unsorted. The left part is sorted subarray and the right part is unsorted subarray. Initially, sorted subarray is empty and unsorted array is the complete given array.

We perform the steps given below until the unsorted subarray becomes empty:

1. Pick the minimum element from the unsorted subarray.
2. Swap it with the leftmost element of the unsorted subarray.
3. Now the leftmost element of unsorted subarray becomes a part (rightmost) of sorted subarray and will not be a part of unsorted subarray.

A selection sort works as follow

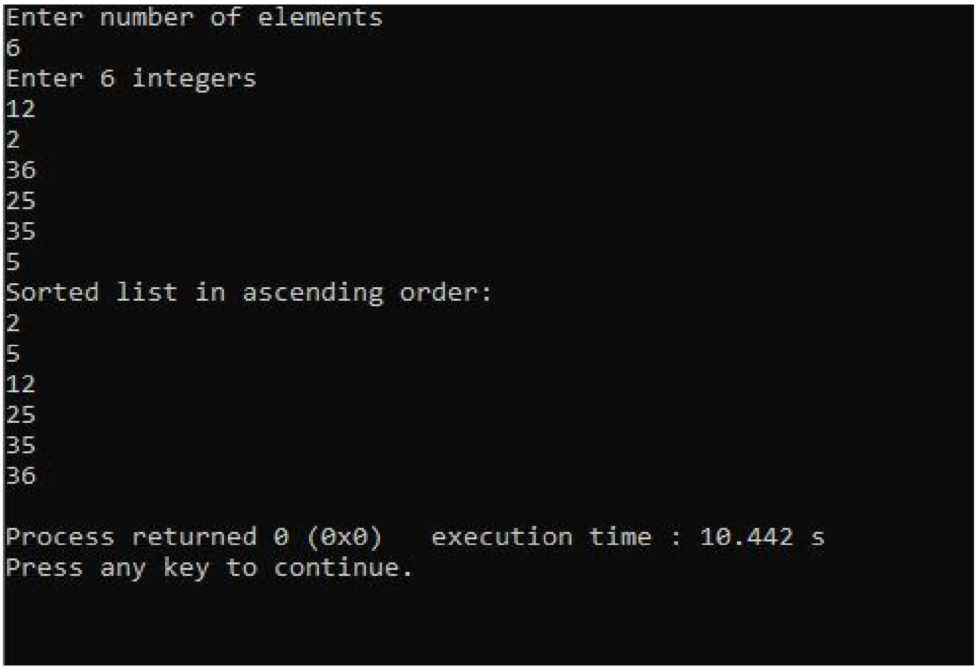


In the above diagram, the smallest element is found in first pass that is 9 and it is placed at the first position. In second pass, smallest element is searched from the rest of the element excluding first element. Selection sort keeps doing this, until the array is sorted.

Lab Exercise

```
#include <stdio.h>
  int main()
  {
    int array[100], n, c, d, position, swap;
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for ( c = 0 ; c < n ; c++ )
      scanf("%d", &array[c]);
    for ( c = 0 ; c < ( n - 1 ) ; c++ )
    {
      position = c;
      for ( d = c + 1 ; d < n ; d++ )
      {
        if ( array[position] > array[d] )
          position = d;
      }
if ( position != c )
      {
        swap = array[c];
        array[c] = array[position];
        array[position] = swap;
      }
    }
    printf("Sorted list in ascending order:\n");
    for ( c = 0 ; c < n ; c++ )
      printf("%d\n", array[c]);
    return 0;
  }
```

Output



```
Enter number of elements
6
Enter 6 integers
12
2
36
25
35
5
Sorted list in ascending order:
2
5
12
25
35
36

Process returned 0 (0x0)   execution time : 10.442 s
Press any key to continue.
```

14.5 Shell Sort

Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions. In general, Shell sort performs the following steps.

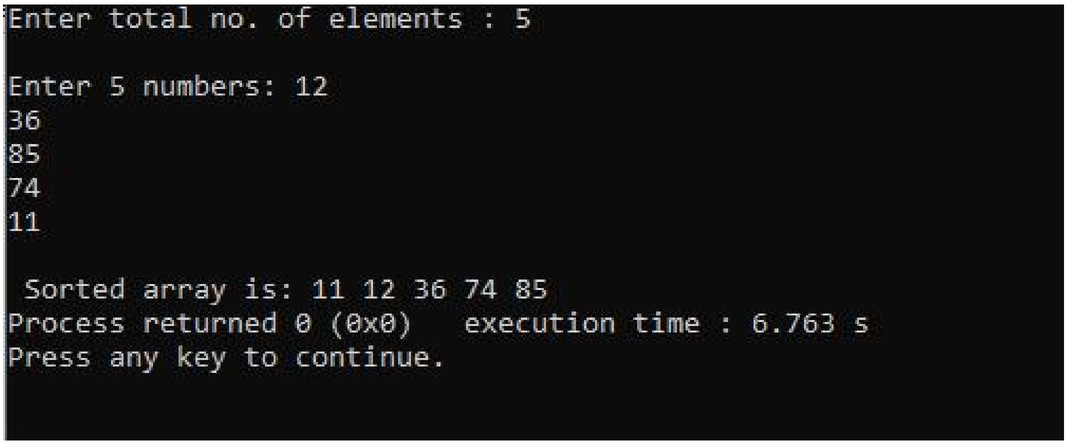
Step 1: Arrange the elements in the tabular form and sort the columns by using insertion sort.

Step 2: Repeat Step 1; each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

Lab Exercise

```
#include <stdio.h>
void shellsort(int arr[], int num)
{
int i, j, k, tmp;
for (i = num / 2; i > 0; i = i / 2)
{
for (j = i; j < num; j++)
{
for(k = j - i; k >= 0; k = k - i)
{
if (arr[k+i] >= arr[k])
break;
else
{
tmp = arr[k];
arr[k] = arr[k+i];
arr[k+i] = tmp;
}
}
}
}
}
int main()
{
int arr[30];
int k,  num;
printf("Enter total no. of elements : ");
scanf("%d", &num);
printf("\nEnter %d numbers: ", num);
for (k =  0 ; k < num; k++)
{
scanf("%d", &arr[k]);
}
shellsort(arr, num);
printf("\n Sorted array is: ");
for (k = 0; k < num; k++)
printf("%d ", arr[k]);
return 0;
}
```

Output



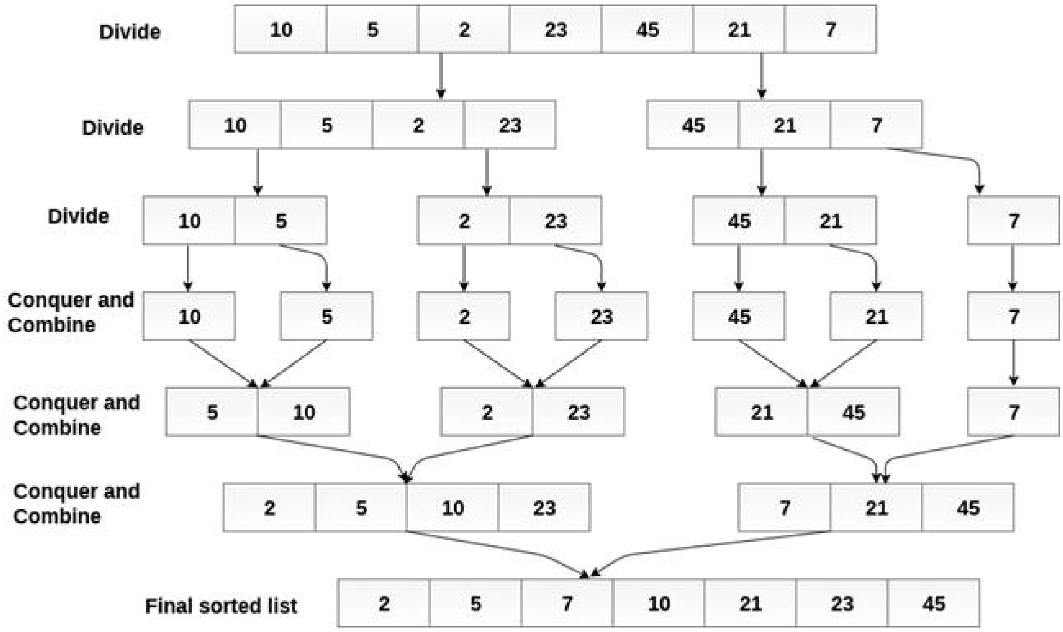
```
Enter total no. of elements : 5

Enter 5 numbers: 12
36
85
74
11

 Sorted array is: 11 12 36 74 85
Process returned 0 (0x0)   execution time : 6.763 s
Press any key to continue.
```

14.6 Merge Sort

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

Example - A = {10, 5, 2, 23, 45, 21, 7}



Algorithm

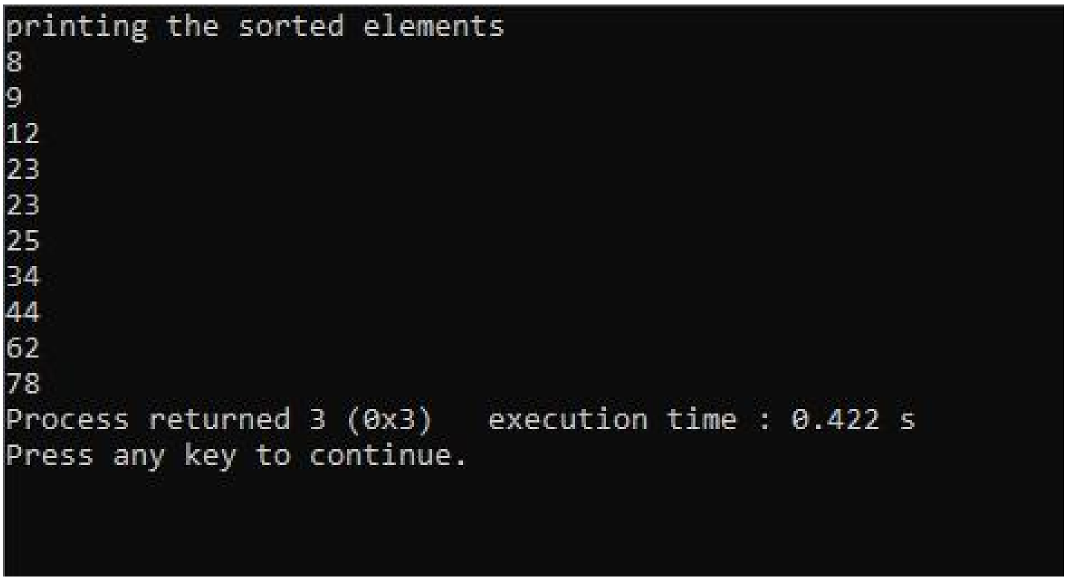
```
Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
IF ARR[I] < ARR[J]
SET TEMP[INDEX] = ARR[I]
SET I = I + 1
ELSE
SET TEMP[INDEX] = ARR[J]
SET J = J + 1
[END OF IF]
SET INDEX = INDEX + 1
[END OF LOOP]
Step 3: [Copy the remaining
elements of right sub-array, if
any]
```

```
IF I > MID
Repeat while J <= END
SET TEMP[INDEX] = ARR[J]
SET INDEX = INDEX + 1, SET J = J + 1
[END OF LOOP]
[Copy the remaining elements of
left sub-array, if any]
ELSE
Repeat while I <= MID
SET TEMP[INDEX] = ARR[I]
SET INDEX = INDEX + 1, SET I = I + 1
[END OF LOOP]
[END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K = 0
Step 5: Repeat while K < INDEX
SET ARR[K] = TEMP[K]
SET K = K + 1
[END OF LOOP]
Step 6: Exit
```

Lab Exercise

```
#include<stdio.h>
void mergeSort(int[],int,int);
void merge(int[],int,int,int);
void main ()
{
    int a[10]= {12, 9,8, 25, 23, 44, 62, 78, 34, 23};
    int i;
    mergeSort(a,0,9);
    printf("printing the sorted elements");
    for(i=0;i<10;i++)
    {
        printf("\n%d",a[i]);
    }
}
void mergeSort(int a[], int beg, int end)
{
    int mid;
    if(beg<end)
    {
        mid = (beg+end)/2;
        mergeSort(a,beg,mid);
        mergeSort(a,mid+1,end);
        merge(a,beg,mid,end);
    }
}
void merge(int a[], int beg, int mid, int end)
{
    int i=beg,j=mid+1,k,index = beg;
    int temp[10];
    while(i<=mid && j<=end)
    {
        if(a[i]<a[j])
        {
            temp[index] = a[i];
            i = i+1;
        }
        else
        {
            temp[index] = a[j];
            j = j+1;
        }
        index++;
    }
    if(i>mid)
    {
        while(j<=end)
        {

temp[index] = a[j];
            index++;
            j++;
        }
    }
    else
    {
        while(i<=mid)
        {
            temp[index] = a[i];
            index++;
            i++;
        }
    }
    k = beg;
    while(k<index)
    {
        a[k]=temp[k];
        k++;
    }
}
```

Output



```
printing the sorted elements
8
9
12
23
23
25
34
44
62
78
Process returned 3 (0x3)   execution time : 0.422 s
Press any key to continue.
```

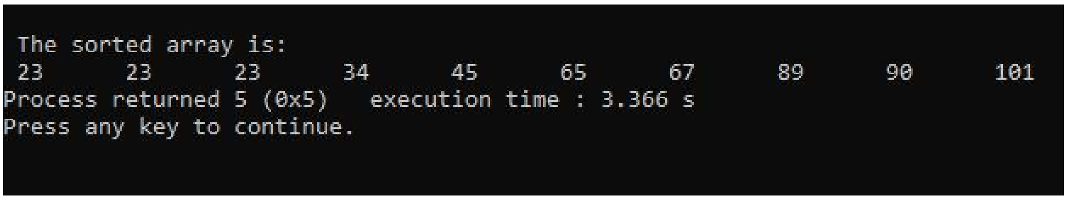
14.7 Radix Sort

Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

Lab Exercise

```
#include <stdio.h>
int largest(int a[]);
void radix_sort(int a[]);
void main()
{
    int i;
    int a[10]={90,23,101,45,65,23,67,89,34,23};
    radix_sort(a);
    printf("\n The sorted array is: \n");
    for(i=0;i<10;i++)
        printf(" %d\t", a[i]);
}
int largest(int a[])
{
    int larger=a[0], i;
    for(i=1;i<10;i++)
    {
        if(a[i]>larger)
        larger = a[i];
    }
    return larger;
}
void radix_sort(int a[])
{
    int bucket[10][10], bucket_count[10];
    int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
    larger = largest(a);
    while(larger>0)
    {
        NOP++;
        larger/=10;
    }
    for(pass=0;pass<NOP;pass++)
    {
        for(i=0;i<10;i++)
        bucket_count[i]=0;
        for(i=0;i<10;i++)
        {
            remainder = (a[i]/divisor)%10;
            bucket[remainder][bucket_count[remainder]] = a[i];
            bucket_count[remainder] += 1;
        }
               i=0;
        for(k=0;k<10;k++)
        {
            for(j=0;j<bucket_count[k];j++)
            {
                a[i] = bucket[k][j];
            i++;
          }
        }
        divisor *= 10;
}
}
```

Output



```
 The sorted array is:
 23      23      23      34      45      65      67      89      90      101
Process returned 5 (0x5)   execution time : 3.366 s
Press any key to continue.
```

Difference between Searching and Sorting Algorithm:

Searching Algorithm	**Sorting Algorithm**
Searching Algorithms are designed to retrieve an element from any data structure where it is used.	A Sorting Algorithm is used to arranging the data of list or array into some specific order.
These algorithms are generally classified into two categories i.e. Sequential Search and Interval Search.	There are two different categories in sorting. These are Internal and External Sorting.
The worst-case time complexity of searching algorithm is O(N).	The worst-case time complexity of many sorting algorithms like Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort is O(N^2).

Insertion Sort Vs. Selection Sort

BASIS FOR COMPARISON	**INSERTION SORT**	**SELECTION SORT**
Basic	The data is sorted by inserting the data into an existing sorted file.	The data is sorted by selecting and placing the consecutive elements in sorted location.
Nature	Stable	Unstable

Process to be followed	Elements are known beforehand while location to place them is searched.	Location is previously known while elements are searched.

Insertion Sort Vs. Selection Sort

BASIS FOR COMPARISON	**INSERTION SORT**	**SELECTION SORT**
Immediate data	Insertion sort is live sorting technique which can deal with immediate data.	It can not deal with immediate data, it needs to be present at the beginning.
Best case complexity	O(n)	O(n^2)

Comparison of Sorting Methods

Sorting Method	**Time Complexity Worst Case**	**Time Complexity Average Case**	**Time Complexity Best Case**	**Space Complexity**
Bubble Sort	n(n-1)/2 = O(n^2)	n(n-1)/2 = O(n^2)	n(n-1)/2 = O(n^2)	Constant
Insertion Sort	n(n-1)/2 = O(n^2)	n(n-1)/4 = O(n^2)	O(n)	Constant
Selection Sort	n(n-1)/2 = O(n^2)	n(n-1)/2 = O(n^2)	n(n-1)/2 = O(n^2)	Constant
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Depends

Summary

- A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements.
- Internal sorting: If the input data is such that it can be adjusted in the main memory at once, it is called internal sorting.

- External sorting: If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting.
- Selection sort is a simple comparison-based sorting algorithm. It is in-place and needs no extra memory.

Keywords

- **Merge sort** is a good choice if you want a stable sorting algorithm. Also, merge sort can easily be extended to handle data sets that can't fit in RAM, where the bottleneck cost is reading and writing the input on disk, not comparing and swapping individual items.
- **Radix sort** looks fast, with its O(n)O(n) worst-case time complexity. But, if you're using it to sort binary numbers, then there's a hidden constant factor that's usually 32 or 64 (depending on how many bits your numbers are). That's often way bigger than O(\lg(n))O(lg(n)), meaning radix sort tends to be slow in practice.
- **Merge Sort -** Merge sort (also commonly spelled mergesort) is an efficient, general-purpose, comparison-based sorting algorithm.
- **Heap Sort-** heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.
- Sorting: Sorting is the technique of arranging the data elements in some logical order, either ascending or descending order. Some algorithms make use of sorted lists. Therefore, efficient sorting is essential for optimizing these algorithms to ensure that they work correctly.
- **Internal sorting:** If the input data is such that it can be adjusted in the main memory at once, it is called internal sorting.
- **External sorting:** If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting.

Self Assessment

1. Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position.

A. True
B. False
C. Sometimes true
D. Sometimes false

2. In the following scenarios, when will you use selection sort?

A. The input is already sorted
B. A large file has to be sorted
C. Large values need to be sorted with small keys
D. Small values need to be sorted with large keys

3. What is the worst case complexity of selection sort?

A. O(nlogn)
B. O(logn)

C. O(n)
D. O(n2)

4. Shell sort algorithm is an example of?
A. External sorting
B. Internal sorting
C. In-place sorting
D. Bottom-up sorting

5. Shell short is also used for searching
A. True
B. False

6. Which of the following method is used for sorting in merge sort?
A. Merging
B. Partitioning
C. Selection
D. Exchanging

7. What will be the best case time complexity of merge sort?
A. O(n log n)
B. O(n2)
C. O(n2 log n)
D. O(n log n2)

8. Which of the following is a stable sorting algorithm?
A. Merge sort
B. Typical in-place quick sort
C. Heap sort
D. Selection sort

9. Which of the following sorting algorithm is in-place
A. Counting sort
B. Radix sort
C. Bucket sort
D. None

10. Best case complexity of radix sort is
A. (n log n)
B. n log n
C. (nk)
D. (n + k)

11. Which of the following is not a sorting technique?
A. Bubble Sort
B. Linear Sort
C. Selection Sort
D. Merge Sort

12. Bubble sort is

A. Bubble sort is a type of sorting.

B. It is used for sorting 'n' (number of items) elements.

C. It compares all the elements one by one and sort them based on their values.

D. All of above

13. What is the average case complexity of bubble sort?

A. O(nlogn)

B. O(logn)

C. O(n)

D. O(n2)

14. Which of the following is correct with regard to insertion sort?

A. Insertion sort is stable and it sorts In-place

B. Insertion sort is unstable and it sorts In-place

C. Insertion sort is stable and it does not sort In-place

D. None of above

15. Insertion sort sorting method sorts the array by shifting elements one by one.

A. True

B. False

Answers for Self Assessment

l.	A	2.	C	3.	D	4.	B	5.	A
6.	A	7.	A	8.	A	9.	B	10.	A
11.	B	12.	D	13.	D	14.	A	15.	A

Review Questions

1. What is sorting? How it is different from searching.
2. Discuss the advantages of using sorting in the data structure.
3. Differentiate between selection sort and merge sort.
4. Write a program that demonstrates the working of merge sort.
5. Write a program that demonstrates the working of selection sort.
6. Write a program that demonstrates the working of bubble sort.
7. Differentiate between insertion sort and selection sort.
8. Write and explain insertion sort algorithm. What is the complexity of the algorithm?

Further Readings

Books Ashok N. Kamthane, "Programming with ANCI & Turbo C", Pearson Education, Year of Publication, 2008

Lipschutz. Seymour. Data Structures with C. Delhi: Tata McGraw Hill

Reddy.A.M Padma (2006). Data Structures Using C. Bangalore: Sri Nandi Publications

Greg W Scragg, Genesco Suny, Problem Solving with Computers, Jones and Bartlett, 1997.

R.G. Dromey, Englewood Cliffs, N.J., How to Solve it by Computer, Prentice-Hall International, 1982.

Yashvant Kanetkar, Let us C

Web Links

https://www.geeksforgeeks.org/sorting-algorithms/https://betterexplained.com/articles/sorting-algorithms/

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in