

Unit I

1

INTRODUCTION TO DATA STRUCTURES

Unit Structure :

- 1.0 Objectives
- 1.1 Data and Information
- 1.2 Data Structure
- 1.3 Classification of Data Structure- Primitive Data types, Abstract Data types, Data type VS File organization
- 1.4 Operation on Data Structure
- 1.5 Importance of Algorithm Analysis
- 1.6 Complexity of an Algorithm- Asymptotic Analysis and Notations, Big O Notation, Big Omega Notation, Big Theta Notation, Rate of Growth and Big O Notation
- 1.7 Summary
- 1.8 References
- 1.9 Questions

1.0 OBJECTIVES

At the end of this unit, the student will be able to

- Explain the use of data structure in computers
- Differentiate between the different types of data structures
- Illustrate the concept of asymptotic notations
- Understand the operations of data structures

1.1 DATA AND INFORMATION

- 1 Data Structure is one of the most fundamental subject in computer science and in-depth understanding of this word i.e Data + Structure is very important especially when you are developing a programs for building efficient systems software and applications.
- 2 Definition- In Computer Science, a data structure is a data organization, management and storage format that enables efficient access and modification. In simple words It is a way in which data is stored on a computer.
- 3 Why do we need Data Structures?
 - 3.1 Data structure is a particular way of storing and organizing information in a computer so that it can be retrieved and used most productively
 - 3.2 As each data structure allow data to be stored in specific manner.

- 3.3 Data Structure allows efficient data search and retrieval.
- 3.4 Specific data structure are decided to work for specific problems.
- 3.5 It allows to manage large amount of databases and indexing services such as hash table.
- 4 Examples of Data Structure are Digital Dictionary, Google Map etc
- 5 For eg Array is a Data Structure- will explain with preceded diagram int number[4]={10,20,30,40}; Here one dimensional array is created which will stored in RAM in rows and column format and that resembles array as a Data structure where 10,20,30,40 represents as value.
- 6 What is Data + Information in terms of Data Structure
- 6.1 Data means an individual unit that contains raw materials which do not carry any specific meaning, Whereas Information means is a group of data that collectively carries a logical meaning.
- 6.2 For eg if we refer to above example of Array, each array value which is stored in row and column is belongs to data part and complete array values can be retrieved at a time is nothing but information, on that note exact difference between data and information is data does not depend on information, but information depend on data.

6.3 Table 1 Shows difference between Data & Information

Points of Comparison	Data	Information
Meaning	Data means raw facts gathered about someone or something which is bare and random	Facts, Concerning a particular event or subject, which are refined by processing is called information
What is it?	It is just text and numbers	It is refined data
Based On	Records and Observation	Analysis
Form	Unorganized	Organized
Useful	May or May not be useful	Always
Specific	No	Yes
Dependency	Does not depend on information	Without data, information cannot be processed

Table 1

1.2 DATA STRUCTURE

1. What is binding between Algorithm, Program and Data Structure?
- 1.1 Algorithm- It outline, the essential of a computational procedure, step by step instructions.

1.2 Program- A implementation of algorithm in some programming language.

1.3 Data Structure- Organization of data needed to solve the program

1.4 for eg Write a program to print 10 numbers

Algorithm would be

Step 1 Take a range of values in a variable A, nothing but an Array
Data Structure

Step 1 Assign a Value to an Array A Such as A[1,2,3,4,5,6,7,8,9,10]

Step 2 Apply loop to print values of Array

Program

```
# include<stdio.h>
#include<conio.h>
Void main
{
    int a[10],i;
    for(i=0;i<10;i++)
    {
        Printf("The value of is ",i);
    }
}
```

Data structure – here array is one type of data structure used here

1.5 Difference between Algorithm and Pseudocode

Table 2 Represents difference between Algorithm and Pseudocode

Sr.No	Algorithm	Pseudocode
1	Systematic logical approach which is a well-defined, step-by-step procedure that allows a computer to solve a problem	It is one of the methods which can be used to represent an algorithm for a program
2	Algorithms can be expressed using natural language, flowcharts etc	Pseudocode allows you to include several control structures such as while, if-then-else, Repeat-until, for and case which is present in many high-level languages

Table 2

1.6 Data Structure mainly specifies the following four things

- ✓ Organization of Data
- ✓ Accessing /methods
- ✓ Degree of Associativity
- ✓ Processing alternatives for information

1.7 To study Data structure in basic terms covers the following points

- ✓ Amount of memory require to store
- ✓ Amount of time require to process
- ✓ Representation of data in memory
- ✓ Operations performed on that data

1.3 CLASSIFICATION OF DATA STRUCTURE- PRIMITIVE DATA TYPES, ABSTRACT DATA TYPES, DATA TYPE VS FILE ORGANIZATION

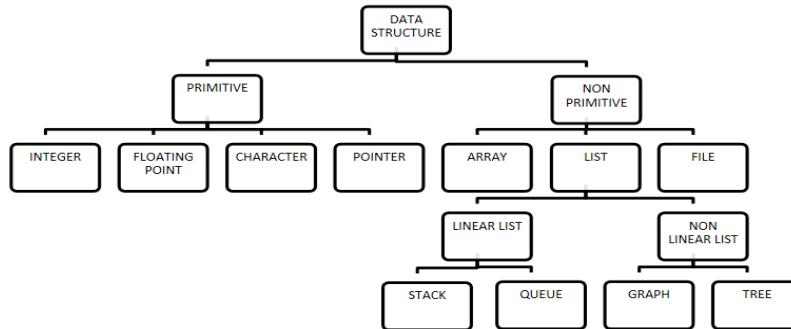


Fig. 1 Classification of Data Structure

- 1 Data structures are normally classified in to two broad categories
 - 1 Primitive Data Structures
 - 2 Non-Primitive Data Structure
- 2 Data types- A particular kind of data item, as defined by the values it can take, the programming language used or the operations that can be performed on it.
- 3 Primitive Data Structure
 - 3.1 Primitive data structure are basic structures and are directly operated upon by machine instructions.
 - 3.2 Primitive data structures have different representations on different computers.
 - 3.3 Integers, floats, character and pointers are examples of primitive data structures.
 - 3.4 These data types are available in most programming languages as built in type
 - 3.4.1 Integer- It is a data type which allows all values without fraction part. We can use it for whole numbers.
 - 3.4.2 Float- It is a data type which use for storing fractional numbers.
 - 3.4.3 Character- It is a data type which is used for character values.
 - 3.4.5 Pointer- A variable that holds memory address of another variable are called pointer.
 - 4 Non Primitive Data type
 - 4.1 These are more sophisticated data structures
 - 4.2 These are derived from primitive data structures.
 - 4.3 The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.

4.4 Example of non-primitive data types are Array, List and Files etc.

4.5 A Non-primitive data type is further divided in to Linear and Non-Linear data structure

1 Array- An array is a fixed size sequenced collection of elements of the same data type.

2 List- An ordered set containing variable number of elements of the same data type .

3 File- A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

5 Linear Data Structures

5.1 A data structure is said to be linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.

5.2 There are two ways to represent a linear data structure in memory,

1 Static Memory Allocation

2 Dynamic Memory Allocation

5.3 The possible operations on the linear data structure are- Traversal, insertion, Deletion, Searching, Sorting and Merging.

5.4 Examples of Linear Data Structures are Stack and Queue

5.4.1 Stack- It is a data structure in which insertion and deletion operation are performed at one end only. The insertion operation is referred to as “PUSH” and deletion operation is referred to as “POP” operation. Stack is also called as Last in First Out(LIFO) data structure.

5.4.2 Queue- The data structure which permits the insertion at one end and deletion at another end known as queue. End at which deletion occurs is known as FRONT end and another end at which insertion occurs as REAR end. It is also called as First in First Out (FIFO) structure.

6 Non Linear data Structures

6.1 These are those data structure in which data items are not arranged in a sequence.

6.2 Examples of Non-Linear Data Structure are Tree and Graph.

6.3 Tree- A tree can be defined as finite set of data items(nodes) in which data items are arranged in branches and sub branches according to requirement. It represents the hierarchical relationship between various elements. Tree consists of nodes connected by edge, the node represented by circle and edge lines connecting to circle.

6.4 Graph- Graph is a collection of nodes(information) and connecting edges(logical relation) between nodes. A tree can be viewed as restricted graph. Graph have many types like un-directed

graph, directed graph, mixed graph, multi graph, simple graph, null graph, weighted graph.

- 7 Table 3 Shows difference between Linear and Non Linear Data Structure

Sr. No	Linear Data Structure	Non-Linear Data Structure
1	Every item is related to its previous and next time	Every item is attached with many other items
2	Data is arranged in linear sequence	Data is not arranged in sequence
3	Data items can be traversed in a single run.	Data cannot be traversed in a single run.
4	Eg Array, Stacks, linked list, queue	Eg Tree, Graph
5	Implementation is easy	Implementation is difficult

Table 3

- 8 Abstract Data type (ADT)- is a type for objects whose behaviour is defined by a set of value and a set of operations.

8.1 The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.

8.2 It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

8.3 It is called “abstract” as it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

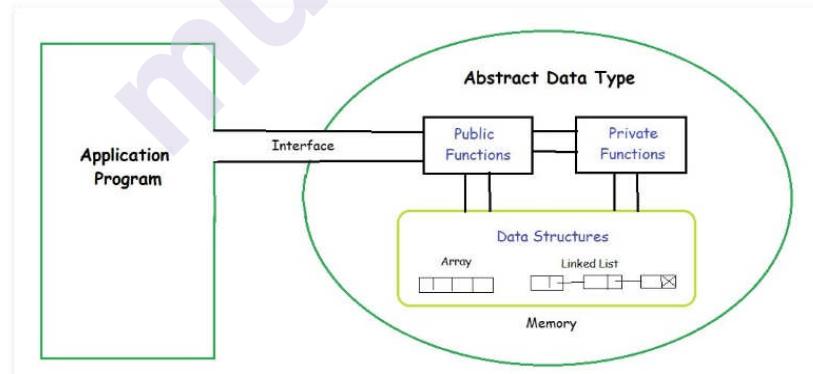


Fig 2 ADT

8.4 for eg Stack ADT

1 In Stack ADT implementation instead of data being stored in each node, the pointer to data is stored.

2 The program allocates memory for the data and address is passed to the stack ADT.

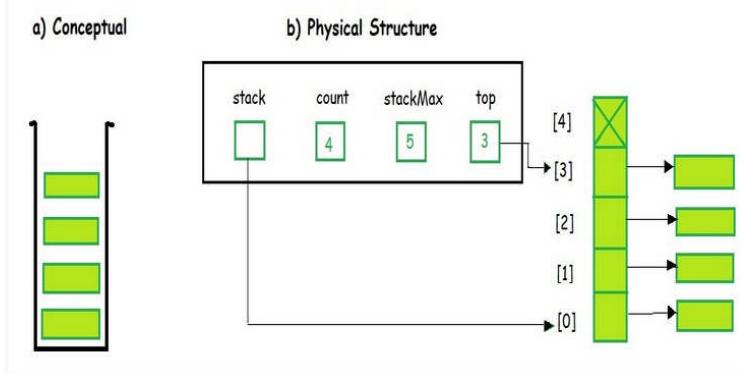


Fig 3 Stack as ADT

- 3 The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- 4 The stack head structure also contains a pointer to top and count of number of entries currently in stack. As Push(), pop(), peek(), size(), isEmpty(), isFull() is already implemented in STACK ADT, only programmer has to use this function for implementing program on stack
- 9 Data Type V/s File Organization- Data type is A particular kind of data item, as defined by the values it can take, the programming language used or the operations that can be performed on it, whereas File organization means how data type values are arranged or organized in file.

1.4 OPERATION ON DATA STRUCTURE

- 1 Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly divided into following types as follows
 - 1.1 Create – The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile time or run time. Malloc() function of C language is used for creation. For eg `ptr=(int*) malloc(100*sizeof(int))`, this syntax will allocate 400 bytes of memory and the pointer ptr holds the address of the first byte in the allocated memory.
 - 1.2 Destroy- This operation destroys memory space allocated for specified data structure. Free() function of C language is used to destroy data structure. For eg syntax of using free() is only write `free(ptr)`.
 - 1.3 Selection- Selection operation deals with accessing a particular data within a data structure. For eg Pseudocode represent below will show how selection works


```
If student's grade is greater than or equal to 60 then
            Print passed
        Else
            Print "failed"
```

In above pseudocode, all students are not passed, only that students are selected which are above 60. So this is selection operation.

1.4 Updation- It updates or modifies the data in the data structure

1.5 Searching-It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.

1.6 Sorting- It is a process of combining the data items of two different sorted list in to a single list.

1.7 Merging- It is a process of combining the data items of two different sorted list in to a single sorted list.

1.8 Splitting- It is a process of partitioning single list to multiple list.

1.9 Traversal- It is a process of visiting each and every node of a list in systematic manner.

1.5 IMPORTANCE OF ALGORITHM ANALYSIS

1 Algorithm

1.1 An essential aspect to data structures is algorithms.

1.2 Data Structures are implemented using algorithms.

1.3 An algorithm is a procedure that you can write as a C function or program, or any other language.

1.4 An algorithm states explicitly how the data will be manipulated.

2 Algorithm Efficiency

2.1 Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.

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

2.3 Usually there are natural units for the domain and range of this function. There are two main complexity measure of the efficiency of an algorithm.

3 Time Complexity

3.1 It is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

3.2 Time can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed or some other natural unit related to the amount of real time the algorithm will be.

4 Space complexity

4.1 It is a function describing the amount of memory space an algorithm takes in terms of the amount of input to the algorithm.

4.2 We always speak of “extra” memory needed, not counting the memory needed to store the input itself. Again we use natural but fixed-length units to measure this.

4.3 We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.

4.4 It is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

5 Worst Case Analysis

5.1 In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know that causes maximum number of operations to be executed.

5.2 For Linear search, the worst case happens when the element to be searched is not present in the array.

5.3 When X element is not present, the search() functions compares it with all the elements of array[] one by one. So, the worst case time complexity of linear search would be.

6 Average Case Analysis

6.1 In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.

6.2 Add all the calculated values and divide the obtained sum by total number of inputs.

6.3 For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by $(n+1)$.

7 Best Case Analysis

7.1 In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the causes minimum number of operations to be executed.

7.2 In the linear search problem, the best case occurs when x is present at the first location.

1.6 COMPLEXITY OF AN ALGORITHM- ASYMPTOTIC ANALYSIS AND NOTATIONS, BIG O NOTATION, BIG OMEGA NOTATION, BIG THETA NOTATION, RATE OF GROWTH AND BIG O NOTATION

1 Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

2 For example- In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e the best case.

3 But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e the worst case.

4 When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

5 There are mainly three asymptotic notations

- 5.1 Big-O Notation
- 5.2 Omega Notation
- 5.3 Theta Notation

6 Big-O Notation(O-Notation)

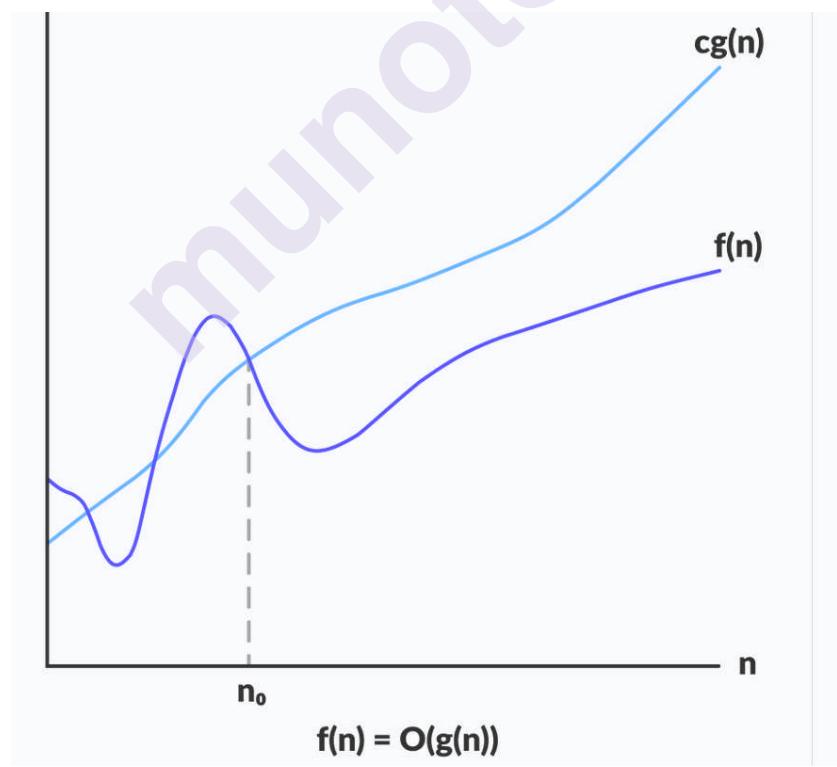
6.1 Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

6.2 $O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$

The above expression can be described as a function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that it lies between 0 and $cg(n)$, for sufficiently large n .

6.3 For any value of n , the running time of an algorithm does not cross the time provided by $O(g(n))$.

6.4 Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.



7 Omega Notation(Ω -Notation)

7.1 Omega notation represents the lower bound of the running time of an algorithm. Thus it provides the best case complexity of an algorithm.

7.2 $\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$

The above expression can be described as a function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c such that it lies above $cg(n)$, for sufficiently large n .

7.3 For any value of n , the minimum time required by the algorithm is given by omega $\Omega(g(n))$.

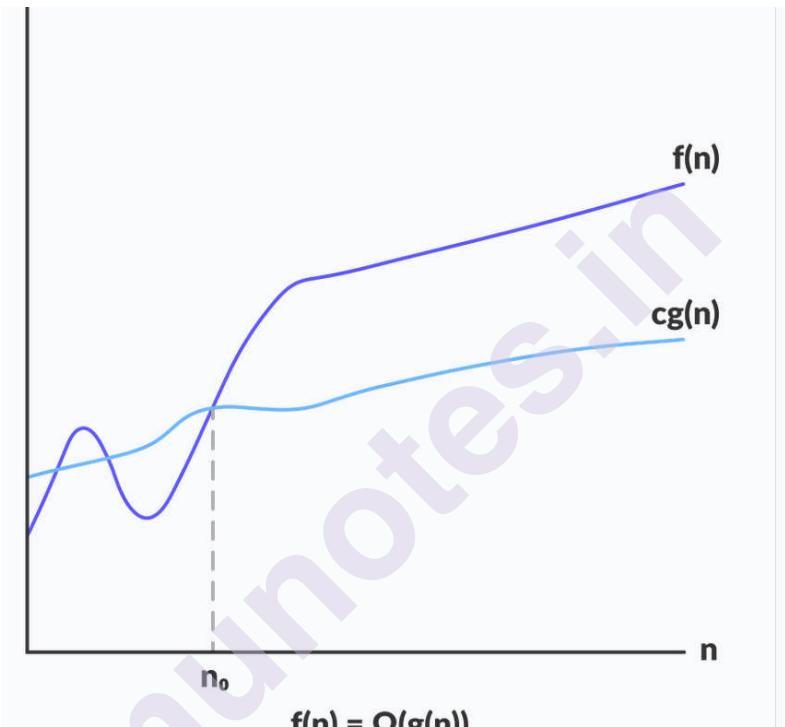


Fig 5 Omega Notation where omega gives the lower bound of a function

8 Theta Notation(Θ -Notation)

8.1 Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

8.2 $\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0 \}$

The above expression can be described as a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exists positive constant c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n .

8.3 If a function $f(n)$ lies anywhere between $c_1g(n)$ and $c_2g(n)$ for all $n \geq n_0$, then $f(n)$ is said to be asymptotically tight bound.

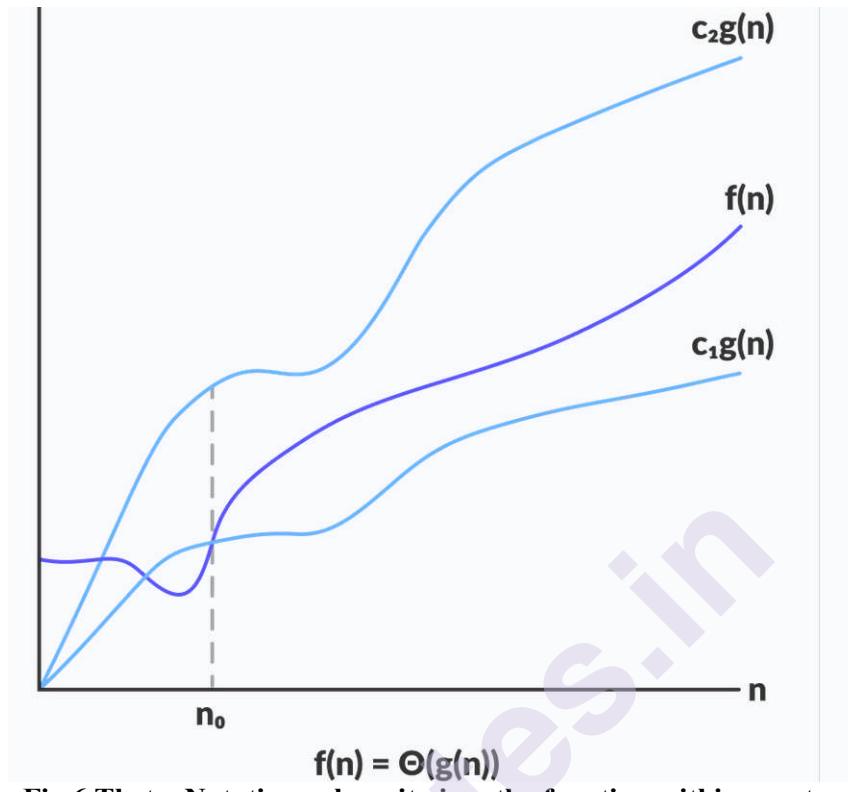


Fig 6 Theta Notation where it gives the function within constant factors

9 Properties of Asymptotic Notations

9.1 General Properties- If $f(n)$ is $O(g(n))$ then $a*f(n)$ is also $O(g(n))$; where a is a constant.

Example: $f(n) = 2n^2+5$ is $O(n^2)$
 then $7*f(n) = 7(2n^2+5) = 14n^2+35$ is also $O(n^2)$.

Similarly this property satisfies for both Θ and Ω notation. We can say - If $f(n)$ is $\Theta(g(n))$ then $a*f(n)$ is also $\Theta(g(n))$; where a is a constant. If $f(n)$ is $\Omega(g(n))$ then $a*f(n)$ is also $\Omega(g(n))$; where a is a constant.

9.2 Transitive Properties

If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n) = O(h(n))$. Example: if $f(n) = n$, $g(n) = n^2$ and $h(n)=n^3$ n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$

9.3 Reflexive Properties

Reflexive properties are always easy to understand after transitive. If $f(n)$ is given then $f(n)$ is $O(f(n))$. Since maximum value of $f(n)$ will be $f(n)$ itself Hence $x = f(n)$ and $y = O(f(n))$ tie themselves in reflexive relation always.

Example: $f(n) = n^2$; $O(n^2)$ i.e $O(f(n))$ Similarly this property satisfies for both Θ and Ω notation.

9.4 Symmetric Properties

If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$. Example: $f(n) = n^2$ and $g(n) = n^2$, then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$. This property only satisfies for theta notation.

9.5 Transpose symmetric properties

If $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$. Example: $f(n) = n$, $g(n) = n^2$, then n is $O(n^2)$ and n^2 is $\Omega(n)$. This property only satisfies for Omega and O notation.

- 10 Practice Problem-In this problem you will compute the asymptotic time complexity of the following divide-and-conquer algorithm.
You may assume that n is a power of 2. (NOTE: It doesn't matter what this does!)

```
float useless(A) {
    n = A.length;
    if (n==1) {
        return A[0];
    }
    let A1,A2 be arrays of size n/2
    for (i=0; i <= (n/2)-1; i++) {
        A1[i] = A[i];
        A2[i] = A[n/2 + i];
    }
    for (i=0; i<=(n/2)-1; i++) {
        for (j=i+1; j<=(n/2)-1; j++) {
            if (A1[i] == A2[j])
                A2[j] = 0;
        }
    }

    b1 = useless(A1);
    b2 = useless(A2);
    return max(b1,b2);
}
```

The solution to above problem is

Clearly $T(1) = \Theta(1)$. The first loop used for splitting has time complexity

$\Theta(n)$ and the second has time complexity $\Theta(n^2)$. There are 2 recursive calls of size $n/2$. Finally constant time ($\Theta(1)$) is spent combining. Hence we get the recurrence.

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= 2 T(n/2) + \Theta(n^2) \text{ for } n \geq 2 \end{aligned}$$

By the master method we get that $T(n) = \Theta(n^2)$. Hence this given algorithm is a $\Theta(n^2)$ algorithm.

1.7 SUMMARY OF CHAPTER

- In this chapter we discuss about what is difference between Data and information.
- This chapter also gives definition of Data structure along with operation on Data Structure like Insertion, search, updation, deletion.
- Also seen the concept of why algorithm analysis is important with respect to best case, worst case and average case analysis and with respect to time and space complexity.
- Asymptotic notation can be illustrated in this chapter with function growth rate and example and also discussed usage of BigO, Theta and Omega in finding out algorithm complexity.

1.8 REFERENCES

Textbook

- 1 An Introduction to Data Structure with Applications, Jean – Paul Tremblay and Paul Sorenson Tata MacGraw Hill 2 nd 2007.
- 2 Schaum's Outlines Data structure Seymour Lipschutz Tata McGraw Hill 2 nd 2005.

Useful Links

- 1 Nptel Course of Data Structure-
<https://nptel.ac.in/courses/106/102/106102064/>

1.9 MISCELLANEOUS QUESTIONS

- Q1 Discuss different types of Data Structures
- Q2 Define Data Structure and its importance
- Q3 What is Best case, Average Case and Worst case w.r.t to algorithm
- Q4 Compare and Contrast between Big O notation, Theta Notation and Omega Notation
- Q5 Difference between Linear and Non Linear Data Structure



2

ARRAY

Unit Structure :

- 2.0 Objectives
- 2.1 Introduction
- 2.2 One Dimensional Array- Memory Representation, Traversing, Insertion, Deletion, Searching, Sorting, Merging of Arrays.
- 2.3 Multidimensional Arrays- Memory Representation, General Multidimensional arrays
- 2.4 Sparse Arrays- Sparse Matrix, Memory Representation of special kind of matrices
- 2.5 Advantages and Limitations of Arrays
- 2.6 Summary
- 2.7 References
- 2.8 Questions

2.0 OBJECTIVES

At the end of this unit, the student will be able to

- ✓ Describe the memory representation of one dimensional array and the operation on array.
- ✓ Illustrate the concept of M-Dimensional array.
- ✓ Explain the need of Sparse array.
- ✓ Compare and contrast between different types of arrays.

2.1 INTRODUCTION OF ARRAYS

- An array is a data structure used to process multiple elements with the same data type when a number of such elements are known.
- Arrays form an important part of almost all-programming languages.
- It provides a powerful feature and can be used as such or can be used to form complex data structures like stacks and queues.
- An array can be defined as an infinite collection of homogeneous(similar type) elements.
- This means that an array can store either all integers, all floating point numbers, all characters, or any other complex data type, but all of same type.
- Arrays are always stored in consecutive memory locations.
- Types of Arrays

There are two types of arrays

- One Dimensional Arrays
- Two Dimensional Arrays

2.2 ONE DIMENSIONAL ARRAY- MEMORY REPRESENTATION, TRAVERSING, INSERTION, DELETION, SEARCHING, SORTING, MERGING OF ARRAYS

1. One Dimensional Arrays

- A one-dimensional array is one in which only one subscript specification is needed to specify a particular element of the array.
- A one-dimensional array is a list of related variables. Such lists are common in programming.
- One-dimensional array can be declared as follows :
Data_type var_name[Expression];

2. Initializing One-Dimensional Array

2.1 ANSI C allows automatic array variables to be initialized in declaration by constant initializers as we have seen we can do for scalar variables.

2.2 These initializing expressions must be constant value; expressions with identifiers or function calls may not be used in the initializers.

2.3 The initializers are specified within braces and separated by commas as shown in below declarations

```
int ex[5] = { 10, 5, 15, 20, 25 } ;
char word[10] = { 'h', 'e', 'l', 'l', 'o' } ;
```

2.4 Example

// Program to print the element of Array

```
#include <stdio.h>
int main()
{
    /*an array with 5 rows and 2 columns*/
    int a[5][2] = {{0,0},{1,2},{2,4},{3,6},{4,8}};
    int i,j;
    /* Output each array elements value*/
    for(i=0; i<5; i++)
    {
        for(j=0; j<2; j++)
        {
            printf("a[%d][%d]=%d",i,j,a[i][j]);
        }
    }
    return 0;
}
```

```
}
```

Output

```
a[0][0] = 0  
a[0][1] = 0  
a[1][0] = 1  
a[1][1] = 2  
a[2][0] = 2  
a[2][1] = 4  
a[3][0] = 3  
a[3][1] = 6  
a[4][0] = 4
```

3 Memory Representation of 1-D array:

3.1 One-dimensional arrays are allocated in a contiguous block of memory.

3.2 All the elements are stored next to each other.

3.3 Example: int a[4]= {10,20,30,40}

3.4 Each element in an array has a unique subscript value from 0 to size of array.

3.5 Example: a[0] = 10 , a[1]=20 , a[3] = 30 , a[4] = 40

3.6 As there are 4 integer elements , the array occupies total of $4 \times 2 = 8$ bytes

3.7 let the memory location start at value 100, so the memory representation will look like as depict in below diagram

a[0]	a[1]	a[2]	a[3]	Unique subscript value
10	20	30	40	
100	102	104	108	Memory Location

Fig 1 Memory Representation of 1D Array

4. Operation of One Dimensional Array

4.1 Traversing

1 In traversing operation of an array, each element of an array is accessed exactly for once for processing. This is also called visiting of an array.

2 Let LA is a Linear Array (unordered) with N elements.

//Write a Program which perform traversing operation.

Note- Use online compiler GDB compiler for running this program

```
#include <stdio.h>  
int main()  
{  
    int i,n; //Declaration of Variable used for inputting Value and for  
    iteration
```

```

printf("Enter the size of array");
scanf("%d",&n);
int LA[n];//Declaration of array
printf("The array elements are:\n");
for(i=0;i<n;i++)
{
    scanf("%d", &LA[i]);
}

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

```

Output

Enter the size of array 5

5

The array elements are:

1
2
3
4
5

LA[0] = 1
LA[1] = 2
LA[2] = 3
LA[3] = 4
LA[4] = 5

4.2 Insertion Operation

1. Insert operation is to insert one or more data elements into an array. Based on the requirement, new element can be added at the beginning, end or any given index of array.

2. Here, we see a practical implementation of insertion operation, where we add data at the end of the array.

3. Example

//Write a program to perform Insertion operation.

```

//Program for inserting an Array
#include <stdio.h>
void main()
{
    int LA[] = {5,6,7,8,9}// Declaration & Initialization of Array
    int item = 4, k = 0, n = 5; //Iterator Variable
    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]);
    }
}

```

Output

The original array elements are:

```

LA[0] = 5
LA[1] = 6
LA[2] = 7
LA[3] = 8
LA[4] = 9

```

The array elements after insertion:

```

LA[0] = 4
LA[1] = 5
LA[2] = 6
LA[3] = 7
LA[4] = 8
LA[5] = 9

```

4.3 Deletion Operation

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

2 Example

Consider LA is a linear array with N elements and K is a positive integer such that K<=N.

//Write a program to perform deletion operation.

```

//Program for deleting an element from an array
#include <stdio.h>
void main()
{
    int LA[] = {1,2,3,4,5}; // Declaring and Initialization of Array
    int k = 3, n = 5; // Set the range variables
    int i, j; // Declare the Iterator
    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]);
    }
}

```

Output

The original array elements are:

```

LA[0] = 1
LA[1] = 2
LA[2] = 3
LA[3] = 4
LA[4] = 5

```

The array elements after deletion:

```

LA[0] = 1
LA[1] = 2
LA[2] = 4
LA[3] = 5

```

4.4 Update Operation

1. Update operation refers to updating an existing element from the array at a given index.

2. Example

Consider LA is a linear array with N elements and K is a positive integer such that $K \leq N$.

```

//Write a program to perform updation operation.
#include <stdio.h>
void main()
{
    int LA[] = {10,11,12,13,14};
    int k = 3, n = 5, item = 15;
    int i, j;
    printf("The original array elements are:\n");
    for(i = 0; i<n; i++)
    {
        printf("LA[%d] = %d \n", i, LA[i]);
    }
    LA[k-1] = item;
}

```

```

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

```

Output

The original array elements are:

```

LA[0] = 10
LA[1] = 11
LA[2] = 12
LA[3] = 13
LA[4] = 14

```

The array elements after updation:

```

LA[0] = 10
LA[1] = 11
LA[2] = 15
LA[3] = 13
LA[4] = 14

```

4.5 Searching an Element in an Array

- 1 Using searching operation on array, an element is searched in an array using key as variable . If element is found in an array at particular location means element is present, if not means elements is not present in an array.
- 2 There are two types of Search methods a)Linear Search b) Binary search
- 3 For Eg Write a program to perform linear search on given array

```

//Write a program to perform Search operation.
#include <stdio.h>
int main()
{
    int a[10],i,n,key;
    printf("Enter size of the array : ");
    scanf("%d", &n);
    printf("Enter elements in array : ");
    for(i=0; i<n; i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Enter the key : ");
    scanf("%d", &key);

    for(i=0; i<n; i++)
    {
        if(a[i]==key)
        {
            printf("element found ");
            return 0;
        }
    }
}

```

```

    }

    printf("element not found");
}
//output
Enter size of the array : 5
Enter elements in array : 1
2
3
4
5
Enter the key : 6
element not found

```

4.6 Sorting Operation in an array

1 With this operation, the array elements are ordered either in ascending or descending order. There many sorting techniques, that you will study later on. But overhere simple bubble sort technique is implemented

2 For eg Write a program for sorting elements in an array

```

//Write a program to perform Sort operation.
#include <stdio.h>
void main()
{
    int i, j, a, n;

    printf("Enter the value of N \n");

    scanf("%d", &n);
    int number[30];
    printf("Enter the numbers \n");

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

        scanf("%d", &number[i]);
    for (i = 0; i < n; ++i)

    {
        for (j = i + 1; j < n; ++j)

        {
            if (number[i] > number[j])

```

```

        a = number[i];
        number[i] = number[j];
        number[j] = a;
    }

}

printf("The numbers arranged in ascending order are given below
\n");
for (i = 0; i < n; ++i)
    printf("%d\n", number[i]);

}

//output
Enter the value of N
5
Enter the numbers
5
4
3
2
1
The numbers arranged in ascending order are given below
1
2
3
4
5

```

4.6 Merging Operation in an Array

- 1 With this operation, the two arrays be merged together in to an one array.
- 2 For eg Write a program to merge two array in to one

```

//Write a program to perform Merge operation.
#include<stdio.h>
int main()
{
    int size1, size2, i, k, merge[100];
    printf("Enter Array 1 Size: ");
    scanf("%d", &size1);
    printf("Enter Array 1 Elements: ");
    int arr1[50];
    for(i=0; i<size1; i++)
    {
        scanf("%d", &arr1[i]);
        merge[i] = arr1[i];
    }
}

```

```

}
k = i;
printf("\nEnter Array 2 Size: ");
scanf("%d", &size2);
int arr2[50];
printf("Enter Array 2 Elements: ");
for(i=0; i<size2; i++)
{
    scanf("%d", &arr2[i]);
    merge[k] = arr2[i];
    k++;
}
printf("\nThe new array after merging is:\n");
for(i=0; i<k; i++)
    printf("%d ", merge[i]);
return 0;
}
//output
Enter Array 1 Size: 5
Enter Array 1 Elements: 1
2
3
4
5
Enter Array 2 Size: 5
Enter Array 2 Elements: 6
7
8
9
10
The new array after merging is:
1 2 3 4 5 6 7 8 9 10

```

2.3 MULTIDIMENSIONAL ARRAYS- MEMORY REPRESENTATION, GENERAL MULTIDIMENSIONAL ARRAYS

- Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts.
- Two dimensional array in which elements are stored column by column is called as column major matrix.
- Two dimensional array in which elements are stored row by row is called as row major matrix.
- First subscript denotes number of rows and second subscript denotes the number of columns.

- The simplest form of the Multi Dimensional Array is the Two Dimensional Array. A Multi Dimensional Array is essence a list of One Dimensional Arrays.

Two dimensional arrays can be declared as follows :

```
int int_array[10]; // A normal one dimensional array
int int_array2d[10][10]; // A two dimensional array
```

Initializing a Two Dimensional Array

A 2D array can be initialized like this :

```
int array[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
```

Program

```
#include <stdio.h>
void printarr(int a[][]);
void printdetail(int a[][]);
void print_usingptr(int a[][]);
main()
{
    int a[3][2]; \ A
    for(int i = 0;i<3;i++)
        for(int j=0;j<2 ;j++)
    {
        {
            a[i]=i;
        }
    }
    printdetail(a);
}
void printarr(int a[][])
{
    for(int i = 0;i<3;i++)
        for(int j=0;j<2;j++)
    {
        {
            printf("value in array %d,a[i][j]);
        }
    }
}
void printdetail(int a[][])
{
    for(int i = 0;i<3;i++)
        for(int j=0;j<2;j++)
    {
        {
            printf( "value in array %d and address is %8u, a[i][j],&a[i][j]);
        }
    }
}
void print_usingptr(int a[][])
{
    int *b; \ B
    b=a; \ C
    for(int i = 0;i<6;i++) \ D
    {
```

```

printf("value in array %d and address is %16lu,*b,b);
b++;// increase by 2 bytes \ E
}
}

```

Explanation

- Statement A declares a two-dimensional array of the size 3×2 .
- The size of the array is 3×2 , or 6.
- Each array element is accessed using two subscripts.
- You can use two for loops to access the array. Since i is used for accessing a row, the outer loop prints elements row-wise, that is, for each value of i , all the column values are printed.
- You can access the element of the array by using a pointer.
- Statement B assigns the base address of the array to the pointer.
- The for loop at statement C increments the pointer and prints the value that is pointed to by the pointer. The number of iterations done by the for loop, 6, is equal to the array.
- Using the output, you can verify that C is using row measure form for storing a two dimensional array.

2.4 SPARSE ARRAYS- SPARSE MATRIX, MEMORY REPRESENTATION OF SPECIAL KIND OF MATRICES

1 A sparse array is an array of data in which many elements have a value of zero. This is in contrast to a dense array, where most of the elements have non-zero values or are “full” of numbers. A sparse array may be treated differently than a dense array in digital data handling.

2 Sparse Matrix Representation

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

3Triplet Representation (Array Representation)

3.1 In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0^{th} row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix. For example, consider a matrix of size 5×6 containing 6 number of non-zero values. This matrix can be represented as shown in the image

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

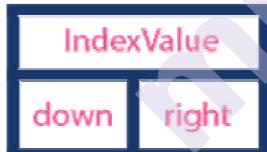
Fig 1 Array Representation of Sparse Matrix

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow a similar pattern.

4 Linked Representation

4.1 In linked representation, we use a linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**. Header node consists of three fields and element node consists of five fields as shown in the image.

Header Node



Element Node



Fig 2 Header and Element Node of Linked Representation

4.2 Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image.

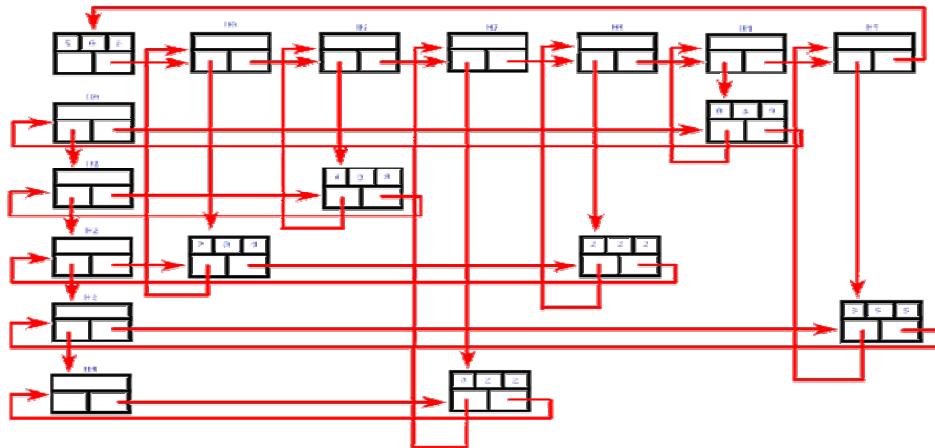


Fig 3 Linked Representation of Sparse Matrix

In the above representation, H0, H1,..., H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements). In this representation, in each row and column, the last node right field points to its respective header node.

2.5 ADVANTAGES AND LIMITATIONS OF ARRAYS

1 Advantages

- 1.1 Arrays represent multiple data items of the same type using a single name.
- 1.2 Arrays allocate memory in contiguous memory locations for all its elements. Hence there is no chance of extra memory being allocated in case of arrays. This avoids memory overflow or shortage of memory in arrays.
- 1.3 In arrays, the elements can be accessed randomly by using the index number.
- 1.4 Using arrays, other data structures like linked lists, stacks, queues, trees, graphs etc can be implemented.
- 1.5 Two-dimensional arrays are used to represent matrices.

2 Disadvantages

- 2.1 The number of elements to be stored in an array should be known in advance.
- 2.2 An array is a static structure (which means the array is of fixed size). Once declared the size of the array cannot be modified. The memory which is allocated to it cannot be increased or decreased.

- 2.3 Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations and the shifting operation is costly.
- 2.4 Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem.

2.6 SUMMARY OF CHAPTER

- This chapter highlights the declaration of 1D array with the program illustrating for operation on array like insertion, updation, deletion, searching, sorting, merging.
- Two D array is part of multidimensional array and explained through a program on matrix.
- Here we also did a discussion about types of Sparse matrix i.e Array and Linked representation.
- We seen some advantages and limitations of arrays.

2.7 REFERENCES

Textbook

- 1 An Introduction to Data Structure with Applications, Jean – Paul Tremblay and Paul Sorenson Tata MacGraw Hill 2 nd 2007.
- 2 Schaum's Outlines Data structure Seymour Lipschutz Tata McGraw Hill 2 nd 2005.

Useful Links

- 1 Nptel Course of Data Structure-
<https://nptel.ac.in/courses/106/102/106102064/>

2.8 MISCELLANEOUS QUESTIONS

- Q1 Discuss various operation performed on an array?
- Q2 List advantages and Disadvantages of Array?
- Q3 Illustrate the concept of Sparse Matrix?
- Q4 Define Array?



Unit II

3

LINKED LIST-I

Unit Structure :

- 3.0 Objectives
- 3.1 Introduction
- 3.2 Concept on Linked List
- 3.3 Representation of Linked List node
- 3.4 Types of Linked List
- 3.5 Singly Linked List
 - 3.5.1 Creation of linked list.
 - 3.5.2 Insertion of any element in the linked list.
 - 3.5.3 Deletion any element from the linked list.
 - 3.5.4 Traversing/Display of the linked list.
- 3.6 Program for the Implementation of Singly Linked List:
- 3.7 Applications of Linked List
 - 3.7.1 Searching a node in list
 - 3.7.2 Merging of two linked List
 - 3.7.3 Copy one singly linked list with other list
 - 3.7.4 Reversing the Linked List
 - 3.7.5 Splitting of two linked list
- 3.8 Summary
- 3.9 List of References
- 3.10 Bibliography
- 3.11 Unit End Exercises

3.0 OBJECTIVES

After going through this unit, you will be able to:

- Define linked List, Memory representation of Linked List
- Types of Linked List-Singly, Doubly, Circular Linked List
- Implementation of Singly Linked List
- Explain the applications of Linked List

3.1 INTRODUCTION

List refers to linear collection of data elements, which contain first element, second element and so on. Data processing are storing and processing data organized into list.

For example:

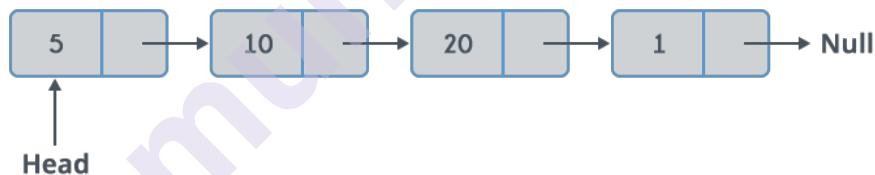
Cat, Dog, Lion, Elephant

We can add more elements in the list at any place beginning, at middle position or at end position.

A linked list is ordered collection of data in which each element contains the data and link to its successor and predecessor.

3.2 CONCEPT ON LINKED LIST

- A linked list is a non-sequential collection of data element. It is a dynamic data structure. For every data element in a linked list, there is an associated pointer that would give the memory location of the next data element in the linked list.
- The data elements in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data elements is easier as each data element contains the address of the next data element.
- It is a data Structure which consists of **group of nodes** that forms a sequence.
- It is very common data structure that is used to create tree, graph and other abstract data types.
- Linked list comprise of group or **list of nodes** in which each node have link to next node to form a chain.



- A linked list is a **linear collection of data-elements**, called '**nodes**'. The linear order is maintained by pointers.

Data	Address of next node
------	----------------------

Fig.3.2.1.Address of Node

Advantages of Linked List:

- Linked list is dynamic in nature which allocates the memory when required.
- Grow and shrink the linked list during run time
- Memory Utilization
- Insertion and Deletion Operations are quite easy

Disadvantages of Linked List:

- Memory Usage-The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Time Consuming
- Heap Space Restriction
- Reverse Traversing is difficult.
- Data accessing is very slow.
- No cache Memory Support

Applications of Linked List:

- In System Programming
- In operating System
- Used in radix and bubble sorting.
- In a FAT file system, the metadata of a large file is organized as a linked list of FAT entries.
- To model the different ADT for other data structure as like stack, queue, tree & Graph etc.

Difference between Array and Linked List

Parameter	Array	Linked List
Size	Specified during declaration.	No need to specify; grow and shrink during execution.
Storage Allocation	Element location is allocated during compile time.	Element position is assigned during run time.
Order of the elements	Stored consecutively	Stored randomly
Accessing the element	Direct or randomly accessed, i.e., Specify the array index or subscript.	Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer.
Insertion and deletion of element	Slow relatively as shifting is required.	Easier, fast and efficient.
Searching	Binary search and linear search	
Memory required	less	More
Memory Utilization	Ineffective	Efficient

3.3 REPRESENTATION OF LINKED LIST NODE

Let's see how each node of the linked list is represented. Each node consists:

- A data element
- An address of another node

We wrap both the data item and the next node reference in a struct as:

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

typedef struct node *node; //Define node as pointer of data type struct
node
```

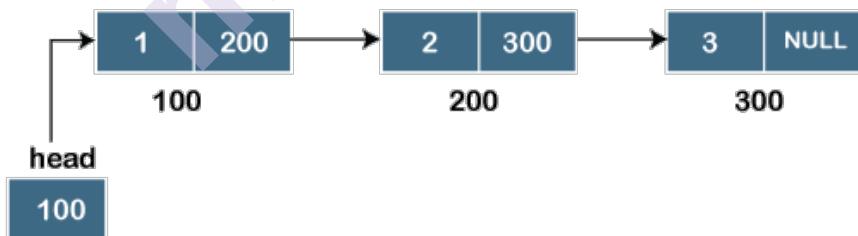
- The above definition is used to create every node in the list. The **data** field stores the element and the **next** is a pointer to store the address of the next node.

3.4 TYPES OF LINKED LIST

- **Singly Linked List –**

Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in the sequence of nodes. Elements are navigated in forward direction.

The operations we can perform on singly linked lists are insertion, deletion and traversal. The representation of the singly linked list as shown below:

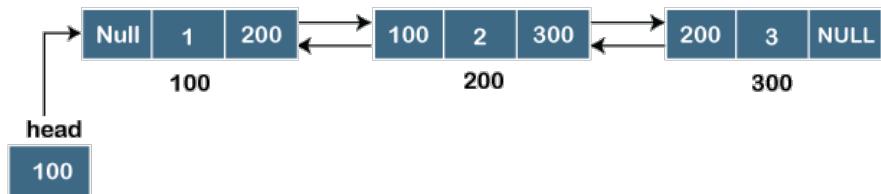


Representation of the node in a singly linked list-

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

- **Doubly Linked List –**

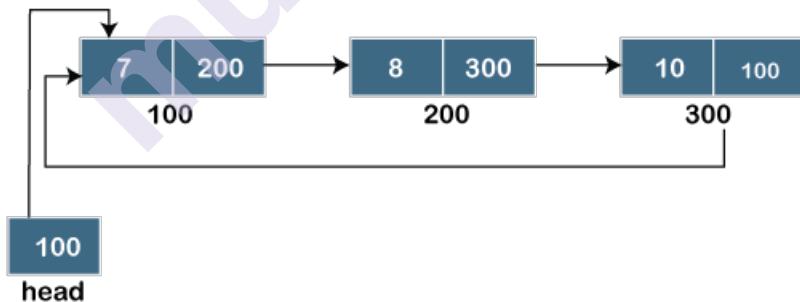
The node in a doubly-linked list has two address parts; one part stores the address of the next while the other part of the node stores the previous node's address. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node. Elements can be navigated forward and backward. The representation of the doubly linked list as shown below:



Representation of the node in a doubly linked list

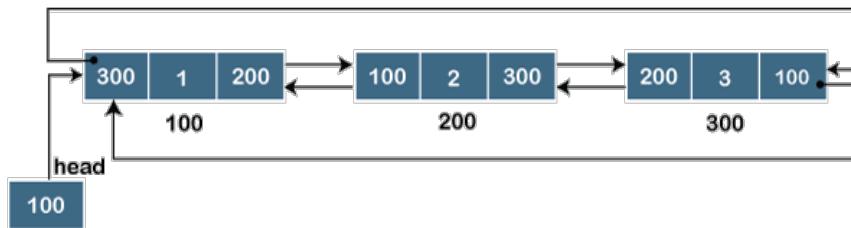
```
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}
```

- **Circular Linked List –** A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



- **Doubly Circular linked list**

The doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the **NULL** value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.



Basic Operations on Linked List:

Following are the basic operations supported by a list.

- Insertion – To add a node at the given position.
- Deletion – Deletes an element from the list.
- Display – Displays the complete list.
- Search – Searches an element using the given key.

Memory Allocation and reallocation in Linked List:

Unlike an array, in linked list individual elements are stored anywhere in memory. Each data element is called a node. A node contain data and address (next) fields. Every nodes holds a pointer (next) to next node in the list.

The memory is allocated for new node dynamically at runtime. A linked list maintains the data elements in a logical order rather than in physical order.

Dynamic memory management allows us to allocate additional memory space or to release unwanted space during program execution.

The functions use for dynamic memory management are as follows-

malloc ()-Allocates as specified number of bytes in memory

calloc ()-Allocates space for an array of elements

free ()- Frees previously allocated space

realloc ()-Modifies the size of previously allocated space

struct node

{

int data;

struct node *next;

} N;

N *n;

n=(*N) malloc (sizeof (N))

free(n)

malloc () is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request.

It is defined by:

void *malloc (number_of_bytes)

void * is returned this pointer can be converted to any type.

For example: char *cp;
cp = (char *) malloc (100);

Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes.

For example,

```
int *ip;
ip = (int *) malloc (100*sizeof(int));
```

free() is the opposite of **malloc()**, which de-allocates memory. The argument to free() is a pointer to a block of memory in the heap a pointer which was obtained by a malloc() function.

The syntax is: free (ptr);

The advantage of free() is simply memory management when we no longer need a block.

3.5 SINGLY LINKED LIST

Singly linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any singly linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.

Implementation of Singly Linked List:

Before writing the code to build the above list, we need to create a **start** node, used to create and access other nodes in the linked list. The following structure definition will do

- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list? This is called as self-referential structure.
- Initialize the start pointer to be NULL.

struct slink list { int data; struct link list* next; }; Type def struct link list node; node *start = NULL;	Node Structure: <table border="1"><tr><td>data</td><td>next</td></tr></table> Empty List: <table border="1"><tr><td>NULL</td></tr></table> Start=NULL	data	next	NULL
data	next			
NULL				

Figure 3.5.1. Structure definition, singly link node and empty list

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

3.5.1. Creation of linked list.

3.5.2. Insertion of any element in the linked list.

3.5.3. Deletion any element from the linked list.

3.5.4. Traversing/Display of the linked list.

3.5.1. Creating a node for Singly Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node.

node* getnode() { node* newnode; newnode = (node *)malloc(sizeof(node)); printf("\n Enter data: "); scanf("%d", &newnode -> data); newnode -> next = NULL; return newnode; }	newnode <table border="1"><tr><td>20</td><td>NULL</td></tr></table> 1000	20	NULL
20	NULL		

Figure 3.5.1.1.illustrates the creation of a node for singly linked list.

We can use the following algorithm steps to create a node of the singly linked list...

Step 1: create the new node using getnode().

newnode = getnode();

Step 3 : If the list is empty, assign new node as start.

start = newnode;

Step 3 : If the list is not empty, follow the steps given below:

- The next field of the new node is made to point the first node (i.e.start node) in the list by assigning the address of the first node.

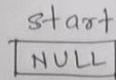
- The start pointer is made to point the new node by assigning the address of the new node.

Step 4: • Repeat the above steps ‘n’ times.

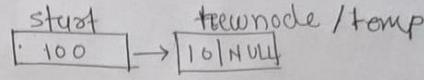
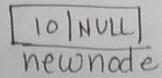
Creation a list of 5 nodes.

$n=5$

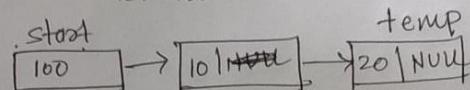
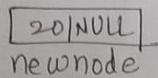
1. Initially



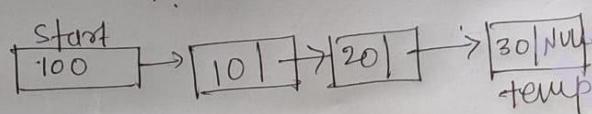
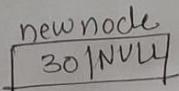
2. $i=0$



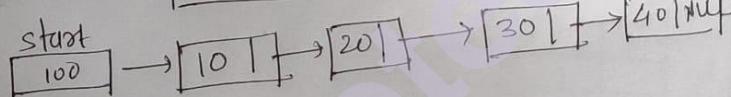
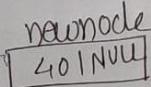
3. $i=1$



4. $i=2$



5. $i=3$



6. $i=4$

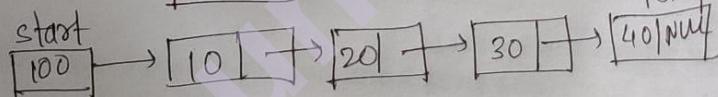
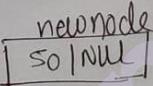


Fig.3.5.1.2.Pictorial Representation of Creation of singly linked list

The function createslist(), is used to create 'n' number of nodes:

```
void createslist(int n)
{
    int i;
    node * newnode;
    node *temp;
    for( i = 0; i < n ; i++)
    {
        newnode = getnode();
        if(start == NULL)
        {
            start = newnode;
        }
        else
```

```

    {
        temp = start;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newnode;
    }
}
}
}

```

3.5.2.Insertion of any element in the linked list.

In a singly linked list, the insertion operation can be performed in three ways. They are as follows...

- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

Inserting At Beginning of the list

The following algorithm steps are to be followed to insert a new node at the beginning of the list:

Step 1• Create the new node using getnode().
`newnode = getnode();`

Step 2• If the list is empty then $start = newnode$.

Step 3• If the list is not empty, follow the steps given below:

`newnode -> next = start;`
`start = newnode;`

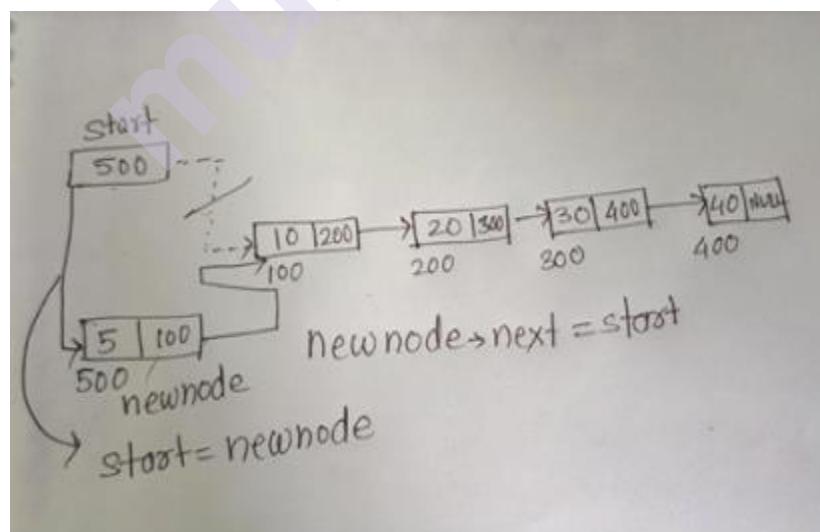


Fig.3.5.2.1.Inserting node at beginning

The function insert_at_beg(), is used for inserting a node at the beginning

```

void insert_at_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        newnode -> next = start;
        start = newnode;
    }
}

```

Inserting At End of the list

The following algorithm steps are followed to insert a new node at the end of the list:

Step 1• Create the new node using getnode()
`newnode = getnode();`

Step 2• If the list is empty then *start = newnode*.

Step 3• If the list is not empty follow the steps given below:

```

temp = start;
while(temp -> next != NULL)
    temp = temp -> next;
temp -> next = newnode;

```

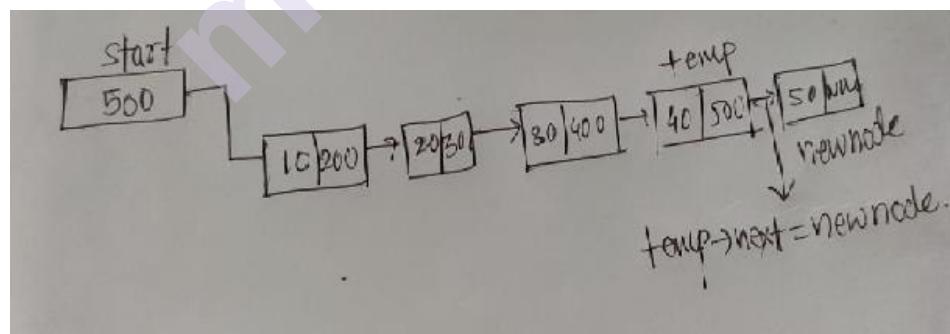


Fig.3.5.2.2.Inserting node at end

The function insert_at_end(), is used for inserting a node at the end.

```

void insert_at_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
    {

```

```

        start = newnode;
    }
else
{
    temp = start;
    while(temp -> next != NULL)
        temp = temp -> next;
    temp -> next = newnode;
}
}

```

Inserting a node at intermediate position:

The following algorithm steps are followed, to insert a new node in an intermediate position in the list:

Step 1: Create the new node using getnode().

```
newnode = getnode();
```

Step 2: Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

Step 3: Store the starting address (which is in start pointer) in temp and temp1 pointers. Then traverse the temp pointer upto the specified position followed by temp1pointer.

Step 4: After reaching the specified position, follow the steps given below:

```
temp -> next = newnode; newnode -> next = temp;
```

- Let the intermediate position be 3.

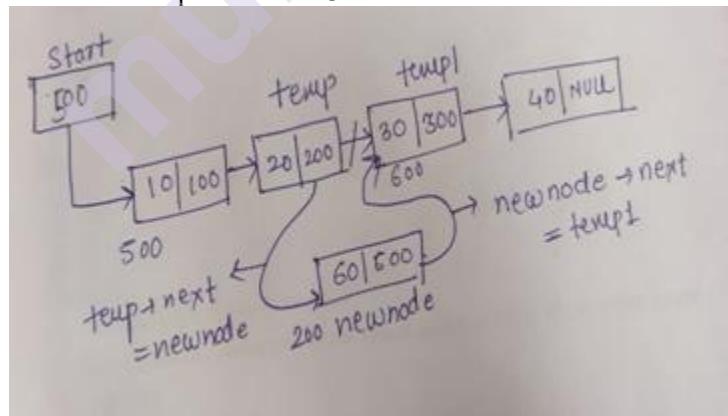


Fig.3.5.2.3.Inserting node at specified position

The function `insert_at_mid()`, is used for inserting a node in the intermediate position.

```

void insert_at_mid()
{
    node *newnode, *temp1, *temp;
    intpos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos> 1 && pos< nodectr)
    {
        temp = temp1 = start;
        while(ctr<pos)
        {
            temp = temp1;
            temp1 = temp1 -> next;
            ctr++;
        }
        temp -> next = newnode;
        newnode -> next = temp1;
    }
    else
    {
        printf("position %d is not a middle position", pos);
    }
}

```

3.5.3. Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.

• **Deleting a node at the beginning:**

Deleting a node from the beginning of the list is the simplest operation of all. Since the first node of the list is to be deleted, therefore, we just need to make the start, point to the next of the start. This will be done by using the following statements.

Step 1: If list is empty then display ‘Empty List’ message.

Step 2: If the list is not empty, follow the steps given below:

```

temp = start;
start = start -> next;
free(temp);

```

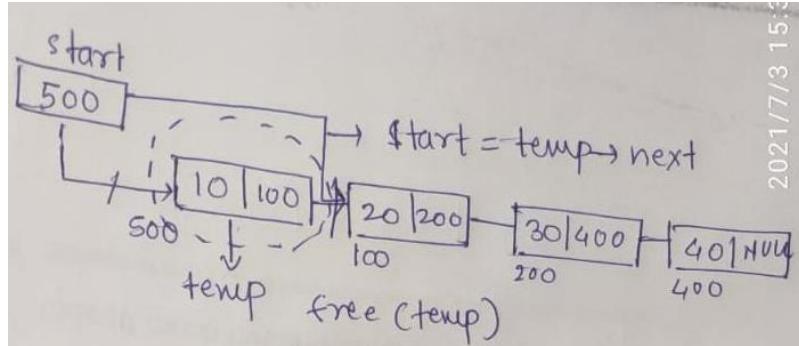


Fig. 3.5.3.1.Delete node from beginning position

The function `delete_at_beg()`, is used for deleting the first node in the list.

```
void delet_eat_beg()
{
node *temp;
if(start == NULL)
{
printf("\n No nodes are exist..");
return ;
}
else
{
    temp = start;
    start = temp -> next;
    free(temp);
    printf("\n Node deleted ");
}
}
```

• Deleting a node at the end.

The following algorithm steps are followed to delete a node at the end of the list:

- If list is empty then display ‘Empty List’ message.
- If the list is not empty, then follow the steps given below:

```
temp = temp1 = start;
while(temp -> next != NULL)
{
    temp1 = temp;
    temp = temp -> next;
}
temp1 -> next = NULL;
free(temp);
```

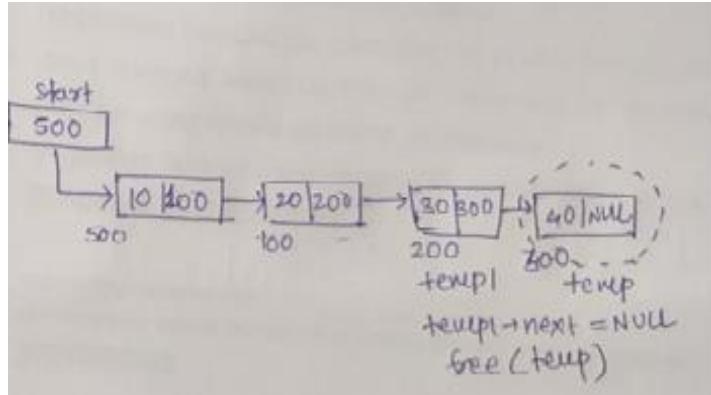


Fig. 3.5.3.2.Delete node from end position

The function `delete_at_last()`, is used for deleting the last node in the list.

```
void delete_at_last()
{
    node *temp, *temp1;
    if(start == NULL)
    {
        printf("\n Empty List..");
        return ;
    }
    else
    {
        temp = temp1 = start;
        while(temp -> next != NULL)
        {
            temp1 = temp;
            temp = temp -> next;
        }
        temp1 -> next = NULL;
        free(temp);
        printf("\n Node deleted ");
    }
}
```

- **Deleting a node at intermediate position.**

The following algorithm steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display ‘Empty List’ message
- If the list is not empty, follow the steps given below.

```
if(pos> 1 && pos<nodectr)
{
    temp = temp1 = start; ctr = 1;
    while(ctr<pos)
    {
        temp1 = temp;
        temp = temp -> next;
        ctr++;
    }
```

```

temp1 -> next = temp -> next;
free(temp);
printf("\n node deleted..");
}

```

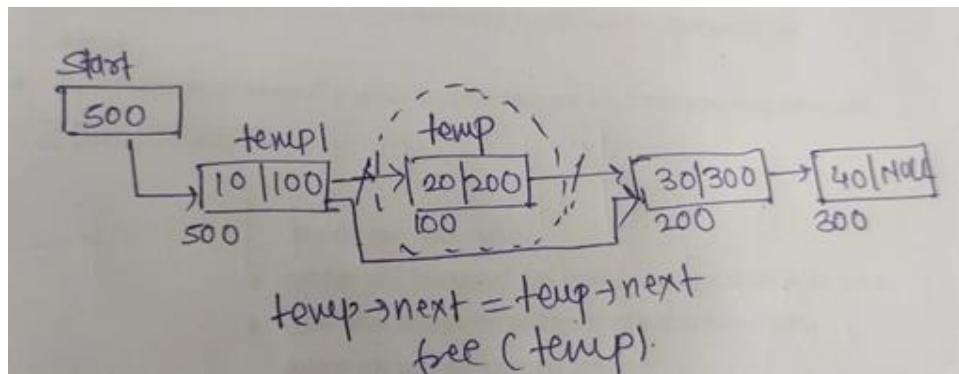


Fig. 3.5.3.3.Delete node from specified position

The function `delete_at_mid()` is used for deleting the intermediate node in the list.

```

void delete_at_mid()
{
    intctr = 1, pos, nodectr;
    node *temp, *temp1;
    if(start == NULL)
    {
        printf("\n Empty List..");
        return ;
    }
    else
    {
        printf("\n Enter position of node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos>nodectr)
        {
            printf("\n This node does not exist");
        }
        if(pos> 1 && pos< nodectr)
        {
            temp = temp1 = start;
            while(ctrl<pos)
            {
                temp1 = temp;
                temp = temp -> next;
                ctr++;
            }
            temp1 -> next = temp -> next;
            free(temp);
            printf("\n Node deleted..");
        }
    }
}

```

```

    {
        printf("\n Invalid position..");
        getch();
    }
}
}

```

3.5.4.Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

- Assign the address of start pointer to a temporary pointer.
- Display the information from the data field of each node.

The function `traverse()` is used for traversing and displaying the information stored in the list from left to right.

```

void traverse()
{
    node *temp;temp=start;
    printf("\n The contents of List (Left to Right): \n");
    if(start==NULL)
        printf("\nEmptyList");
    else
    {
        while(temp!=NULL)
        {
            printf("%d->",temp->data);
            temp=temp->next;
        }
    }
    printf("NULL");
}

```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list using recursion.

```

int countnode(node *ll)
{
    if(ll==NULL)
        return 0;
    else
        return(1 + countnode(ll - > next));
}

```

3.6 PROGRAM FOR THE IMPLEMENTATION OF SINGLY LINKED LIST

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
struct slinklist
{
    int data;
    structslinklist *next;
};
typedef struct slinklist node;
node *start = NULL;
int menu()
{
    intch;
    clrscr();
    printf("\n 1.Create a list ");
    printf("\n-----");
    printf("\n 2.Insert a node at beginning ");
    printf("\n 3.Insert a node at end");
    printf("\n 4.Insert a node at middle");
    printf("\n-----");
    printf("\n 5.Delete a node from beginning");
    printf("\n 6.Delete a node from Last");
    printf("\n 7.Delete a node from Middle");
    printf("\n-----");
    printf("\n 8.Traverse the list (Left to Right");
    printf("\n-----");
    printf("\n 9. Count nodes ");
    printf("\n 10. Exit ");
    printf("\n\n Enter your choice: ");
    scanf("%d",&ch);
    returnnch;
}
node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> next = NULL;
    returnnewnode;
}
int countnode(node *ptr)
{
    int count=0;
    while(ptr != NULL)
    {
        count++;
        ptr = ptr -> next;
    }
}
```

```

        return (count);
    }
void createslist(int n)
{
    int i;
    node *newnode;
    node *temp;
    for(i = 0; i < n; i++)
    {
        newnode = getnode();
        if(start == NULL)
        {
            start = newnode;
        }
        else
        {
            temp = start;
            while(temp -> next != NULL)
                temp = temp -> next;
            temp -> next = newnode;
        }
    }
}
void traverse()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right): \n");
    if(start == NULL)
    {
        printf("\n Empty List");
        return;
    }
    else
    {
        while(temp != NULL)
        {
            printf("%d-->", temp -> data);
            temp = temp -> next;
        }
    }
    printf(" NULL ");
}
void insert_at_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
}

```

```

    }
else
{
    newnode -> next = start;
    start = newnode;
}
}
void insert_at_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        temp = start;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newnode;
    }
}
void insert_at_mid()
{
    node *newnode, *temp1, *temp;
    int pos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos > 1 && pos < nodectr)
    {
        temp = temp1 = start;
        while(ctr < pos)
        {
            temp = temp1;
            temp1 = temp1 -> next;
            ctr++;
        }
        temp -> next = newnode;
        newnode -> next = temp1;
    }
    else
    {
        printf("position %d is not a middle position", pos);
    }
}

void delete_at_beg()

```

```

{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes are exist..");
        return ;
    }
    else
    {
        temp = start;
        start = temp -> next;
        free(temp);
        printf("\n Node deleted ");
    }
}
void delete_at_last()
{
    node *temp, *temp1;
    if(start == NULL)
    {
        printf("\n Empty List..");
        return ;
    }
    else
    {
        temp = temp1=start;
        while(temp -> next != NULL)
        {
            temp1= temp;
            temp = temp -> next;
        }
        temp1 -> next = NULL;
        free(temp);
        printf("\n Node deleted ");
    }
}
void delete_at_mid()
{
    intctr = 1, pos, nodectr;
    node *temp, *temp1;
    if(start == NULL)
    {
        printf("\n Empty List.. ");
        return ;
    }
    else
    {
        printf("\n Enter position of node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);

```

```

        if(pos>nodectr)
        {
            printf("\nThis node doesnot exist");
        }
        if(pos> 1 &&pos<nodectr)
        {
            temp = temp1 = start;
            while(ctrl<pos)
            {
                temp1 = temp;
                temp = temp -> next;
                ctrl++;
            }
            temp1 -> next = temp -> next;
            free(temp);
            printf("\n Node deleted..");
        }
        else
        {
            printf("\n Invalid position..");
            getch();
        }
    }
}

void main(void)
{
    intch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch(ch)
        {
        case 1:
            if(start == NULL)
            {
                printf("\n Number of nodes you want to create: ");
                scanf("%d", &n);
                createslist(n);
                printf("\n List created..");
            }
            else
                printf("\n List is already created..");
                break;
        case 2:
            insert_at_beg();
            break;
        case 3:
            insert_at_end();
            break;
        }
    }
}

```

```

case 4:
    insert_at_mid();
    break;
case 5:
    delete_at_beg();
    break;
case 6:
    delete_at_last();
    break;
case 7:
    delete_at_mid();
    break;
case 8:
    traverse();
    break;
case 9:
    printf("\n No of nodes : %d ", countnode(start));
    break;
case 10 :
    exit(0);
}
getch();
}
}

```

3.7 APPLICATIONS OF LINKED LIST

3.7.1 Searching a node in list:

Check whether the given key is present or not in the linked list.
Consider the Linked List : 10 20 30 40 NULL.

Input-20

Output=Search Found

Input-100

Output=Search Not Found

```

int search Node (node * l, int key)
{
    node *temp = l;
    while(temp != NULL)
    {
        if(temp->data == key)
            return 1;
        temp = temp->next;
    }
    return -1;
}

```

3.7.2 Merging of two linked List:

Given two linked list with Node values sorted in ascending order.

The linked List is-

1->2->3

4->5->6

Merging of two linked list is-

1->2->3->4->5->6

```
/* merges the two linked lists, restricting the common elements to occur
only once in the final list */
void merge ( struct node *l1, struct node *l2, struct node **newnode )
{
    struct node *l3=NULL ;

    /* if both lists are empty */
    if( l1 == NULL &&l2 == NULL )
        return ;

    /* traverse both linked lists till the end. If end of any one list is
reached loop is terminated */
    while ( l1 != NULL &&l2 != NULL )
    {
        /* if node being added in the first node */
        if( *newnode == NULL )
        {
            *newnode = malloc(sizeof( struct node )) ;
            l3= *newnode ;
        }
        else
        {
            l3->next= malloc(sizeof( struct node )) ;
            l3 = l3 ->next ;
        }
        if( l1 -> data <l2 -> data )
        {
            l3 -> data = l1 ->data ;
            l1 = l1 ->next ;
        }
        else
        {
            if( l2 -> data <l1 -> data )
            {
                l3 -> data = l2->data ;
                l2 = l2 ->next ;
            }
            else
            {
                if( l1 -> data == l2 -> data )
                {
                    l3 -> data = l2 ->data ;
                    l1 = l1 ->next ;
                }
            }
        }
    }
}
```

```

        l2 = l2->next ;
    }
}
}

/* if end of first list has not been reached */
while ( l1 != NULL )
{
    l3 ->next = malloc(sizeof( struct node ) );
    l3 = l3 ->next ;
    l3 -> data = l1 ->data ;
    l1 = l1 ->next ;
}

/* if end of second list has been reached */
while ( l2 != NULL )
{
    l3 ->next = malloc(sizeof( struct node ) );
    l3 = l3 ->next ;
    l3 -> data = l2 ->data ;
    l2 = l2 ->next ;
}
l3 ->next = NULL ;
}

```

3.7.3 Copy one singly linked list with other list:

```

void copy(node* l1, node*l2)
{
    node* temp1, *temp2;
    temp1=l1;
    l2=(node*)malloc(sizeof(node));
    temp2=l2;
    while(temp1!=NULL)
    {
        temp2->data=temp1->data;
        temp2->next=(node*)malloc(sizeof(node));
        temp2=temp2->next;
        temp1=temp1->next;
    }
    temp2=NULL;
    printf("\n The list is copied:");
    while(l2->next!=NULL)
    {
        printf("%d",l2->data);
        l2=l2->next;
    }
}

```

3.7.4 Reversing the Linked List:

The linked list is-
10->20->34->NULL

The reverse Linked List is-
34->20->10->NULL

```
void reverse()
{
    Node* temp1,*temp2,*temp3;
    temp1=start;
    If(temp1==NULL)
    {
        printf("\n The List is empty");
        getch();
    }
    else
    {
        temp2=NULL;
        while(temp1!=NULL)
        {
            temp3=temp2;
            temp2=temp1;
            temp1=temp1->next;
            temp2=temp2->next;
        }
        start=temp2;
    }
    printf("\n the List is reversed");
}
```

3.7.5 Splitting of two linked list:

List is :
1 2 3 4 5 6 7

Enter node after which u want to Split : 5

List is :
1 2 3 4 5

List is :
6 7

```
void Split(struct node *start, int value, struct node **start1)
{
    struct node *temp=start;
    while(temp!=NULL)
    {
        if(temp->data==value)
```

```

        break;
        temp=temp->link;
    }
    if(temp==NULL)
    {
        printf("\nValue does not exist\n");
        return;
    }
    *start1=temp->link;
    temp->link=NULL;
}/*End of Split()*/

```

3.8 SUMMARY

- The malloc() is used for allocation of anode and free() is used for de-allocation operation.
 - The singly linked list has only forward pointer and no backward link is provided. Hence the traversing of the list is possible only in one direction.
 - Backward traversing is not possible.
 - Insertion and deletion operations are less efficient because for inserting the element at desired position the list needs to be traversed.
 - Similarly traversing of the list is required for locating the element which needs to be deleted.
-

3.9 LIST OF REFERENCES

<https://sites.google.com/site/datastructuresite/Home-Page/books>
<https://www.hackerearth.com/practice/data-structures/linked-list/singly-linked-list/tutorial/>

3.10 BIBLIOGRAPHY

1. Yashavant Kanetkar Data Structures Through C ,BPB Publications, ISBN: 9788176567060,
2. Goodrich, Tamassia, Goldwasser, —Data Structures and Algorithms in C++ || , Wiley publication, ISBN-978-81-265-1260-7
3. D S Malik, Data Structures Using java, Thomson, India Edition 2006.
4. Sahni S, Data Structures, Algorithms and Applications in java, McGraw-Hill, 2002.
5. SamantaD, Classic Data Structures, Prentice-Hall of India, 2001.
6. Tremblay P, and Sorenson P G, Introduction to Data Structures with Applications, Tata McGraw-Hill,

7. Jean-Paul Tremblay and Paul G. Sorenson, An Introduction to Data Structures with Applications, Tata McGraw Hill
8. Tanenbaum, Data Structures using C & C++, PHI
9. Robert L. Kruse, Data Structures and Program Design in C, PHI
10. Seymour Lipschutz, “Data structures with C”, Schaum’s Publication 5.
Aaron Tanenbaum, “Data Structures using C”, Pearson Education

3.11 UNIT END EXERCISES

1. What is Linked List? State its types.
2. What are the advantages of a Linked List over an Array?
3. What is linked list? How it is different from array? Explain the different types of linked list.
4. Explain the advantages and disadvantages of linked list.
5. Explain applications of Linked list.
6. Write the function to insert and delete a node in the beginning of a Singly Linked List.
7. Write a C program to create and display Singly Linked List.
8. Write a C program to reverse a Singly Linked List.
9. Write a C program to merging of two Singly Linked List.
10. Write a C program to splitting Singly Linked List into two linked list.
11. Write a C program to search a node in Singly Linked List.



LINKED LIST-II

Unit Structure :

- 4.0 Objectives
- 4.1 Introduction
- 4.2 Doubly Linked List
- 4.3 Applications of doubly linked list
- 4.4 Comparison between singly linked list and doubly linked list
- 4.5 Operations on doubly linked list
 - 4.5.1 Creation of doubly linked list.
 - 4.5.2 Insertion of any element in the linked list.
 - 4.5.3 Deletion any element from the linked list.
 - 4.5.4 Traversing/Display of the linked list.
- 4.6 Program on implementation of Doubly Linked List
- 4.7 Circular Linked List
- 4.8 Applications of circular linked list
- 4.9 Operations on doubly linked list
 - 4.9.1 Creation of circular linked list.
 - 4.9.2 Insertion of any element in the linked list.
 - 4.9.3 Deletion any element from the linked list.
 - 4.9.4 Traversing/Display of the linked list.
- 4.10 Program on implementation of Circular Singly Linked List
- 4.11 Representation of linked list using header node
- 4.12 Representation of Polynomial using Linked List
- 4.13 Representation of Sparse Matrix using Linked List
- 4.14 Summary
- 4.15 List of References
- 4.16 Bibliography
- 4.17 Unit End Exercises

4.0 OBJECTIVES

After going through this unit, you will be able to:

- Implementation of doubly linked list with various operations
- Implementation of circular linked list with various operations
- Discuss applications of doubly and circular linked list
- Explain the representation of polynomial and sparse matrix using Linked List

4.1 INTRODUCTION

The **linked list** is a **non primitive data structure** which is free from fixed memory size restriction. It is a linear collection of data element and dynamic in nature. A linked list element cannot be stored in a consecutive memory. It is static free and user can add any number of elements when required. The linked list is a data structure and used to create other data structures, like stacks, queues, etc. It allows the insertion and deletion of nodes at any point in the list.

4.2 DOUBLY LINKED LIST

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- Prev link.
- Data.
- Next link.

The prev link points to the predecessor node and the next link points to the successor node. The data field stores the required data. Many applications require searching forward and backward through nodes of a list. For example searching for a name in a telephone directory is popular example.

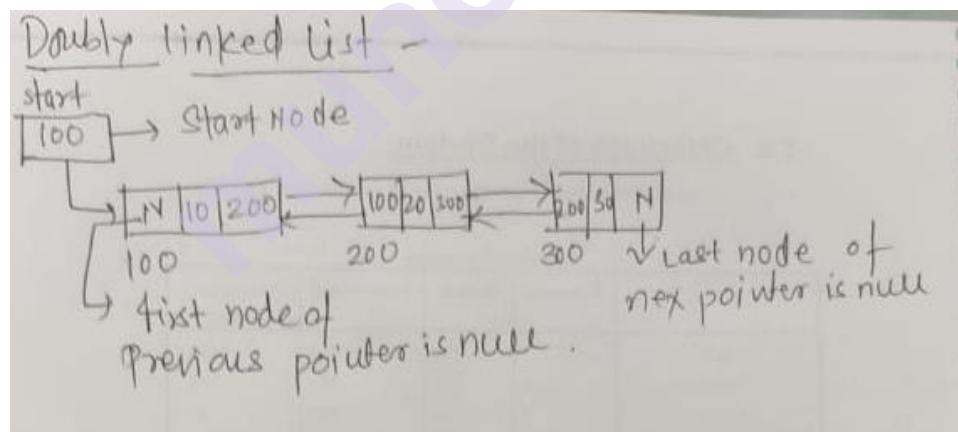


Fig.4.1.Representation of doubly linked list

Implementation of Doubly Linked List:

Before writing the code to build the above list, we need to create a **start** node, used to create and access other nodes in the linked list. The following structure definition will do

- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
- Initialize the start pointer to be NULL.

<pre>structdlink list { int data; structdlink list* next, * prev; }; typedef structdlinklist node; node *start = NULL;</pre>	<p>Node Structure:</p> <table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td style="padding: 5px;">prev</td><td style="padding: 5px;">data</td><td style="padding: 5px;">next</td></tr> </table> <p>Empty List: NULL</p> <p>Start=NULL</p>	prev	data	next
prev	data	next		

Figure 4.2 Structure definition, doubly link node and empty list

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction. There are various operations which can be performed on doubly linked list.

4.3 APPLICATIONS OF DOUBLY LINKED LIST

1. Represents a deck of cards in a game.
2. Undo and redo operations in text editors
3. A music player which has next and previous button uses doubly linked list
4. It is used to implement rewind and forward functions in the playlist.

4.4. COMPARISON BETWEEN SINGLY LINKED LIST AND DOUBLY LINKED LIST

Singly Linked List	Doubly Linked List					
<p>Singly Linked List is a collection of nodes and each node is having one data field and next link field</p> <p>Example:</p> <table border="1" style="display: inline-table; width: fit-content;"> <tr> <td style="padding: 5px;">Data</td> <td style="padding: 5px;">Next field</td> </tr> </table>	Data	Next field	<p>Doubly Linked List is a collection of nodes and each node is having one data field and one previous link field and one next link field</p> <p>Example:</p> <table border="1" style="display: inline-table; width: fit-content;"> <tr> <td style="padding: 5px;">Previous</td> <td style="padding: 5px;">Data</td> <td style="padding: 5px;">Next</td> </tr> </table>	Previous	Data	Next
Data	Next field					
Previous	Data	Next				
<p>The elements can be accessed using next link</p>	<p>The elements can be accessed using both previous link as well as next link</p>					
<p>It is not required extra field .Hence node takes less memory in SLL</p>	<p>One filed is required to store previous link .Hence node takes more memory in DLL</p>					
<p>Less efficient access to elements</p>	<p>More efficient access to elements</p>					

4.5 OPERATIONS ON DOUBLY LINKED LIST

A list of all such operations is given below.

- 4.5.1. Creation of doubly linked list.
- 4.5.2. Insertion of any element in the linked list.
- 4.5.3. Deletion any element from the linked list.
- 4.5.4. Traversing/Display of the linked list.

4.5.1. Creation of doubly linked list.

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set prev field to NULL and next field also set to NULL

node* getnode() { node* newnode; newnode = (node *) malloc(sizeof(node)); printf("\n Enter data: "); scanf("%d", &newnode -> data); newnode -> next = NULL; newnode->prev=NULL; return newnode; }	newnode <table border="1"><tr><td>NULL</td><td>20</td><td>NULL</td></tr></table> 1000	NULL	20	NULL
NULL	20	NULL		

Creating a Doubly Linked List with ‘n’ number of nodes:

The following algorithm steps are to be followed to create ‘n’ number of nodes:

- Get the new node using getnode().
 newnode =getnode();
- If the list is empty then start =temp= newnode
- If the list is not empty, follow the steps given below:
 temp-> next=newnode;
 newnode->prev=temp;
- Repeat the above steps ‘n’ times.

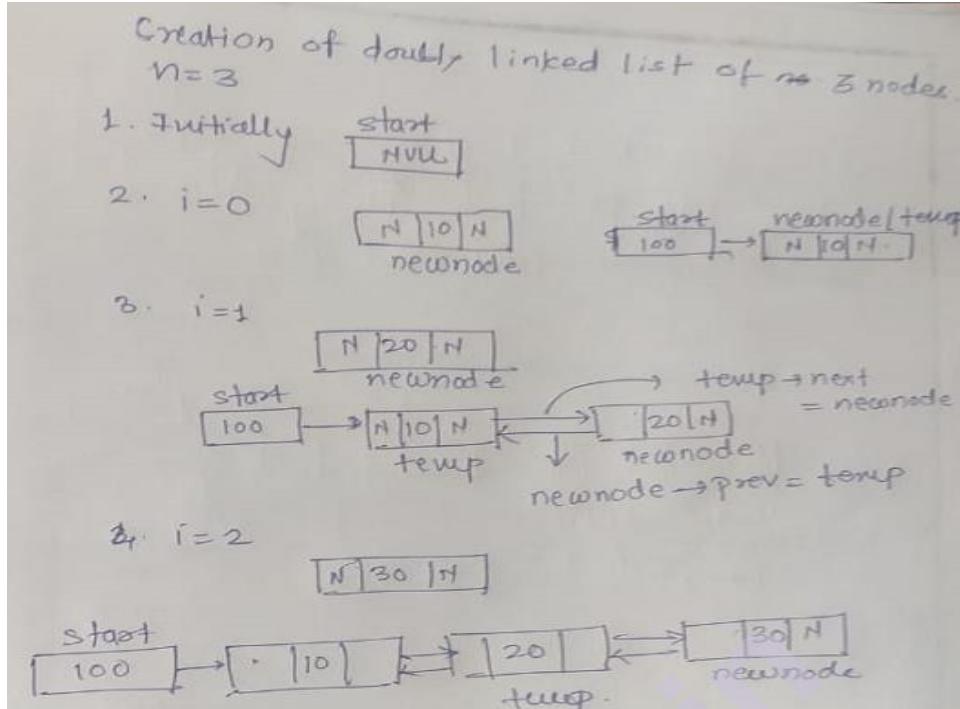


Fig.4.3 Creation of doubly linked list for 'n' nodes

4.5.2. Insertion of any element in the linked list.

Inserting a node in the doubly linked list, there are 3 cases-

- Inserting a node at beginning position
- Inserting node at last position
- Inserting node at intermediate position.
- **Inserting a node at the beginning:**

The following algorithm steps are to be followed to insert a newnode at the beginning of the list:

- Create the newnode using getnode().
 $newnode = getnode();$
- If the list is empty then $start = newnode$.
- If the list is not empty, follow the steps given below:

```

newnode->next= start;
start->prev=newnode;
start=newnode;
```

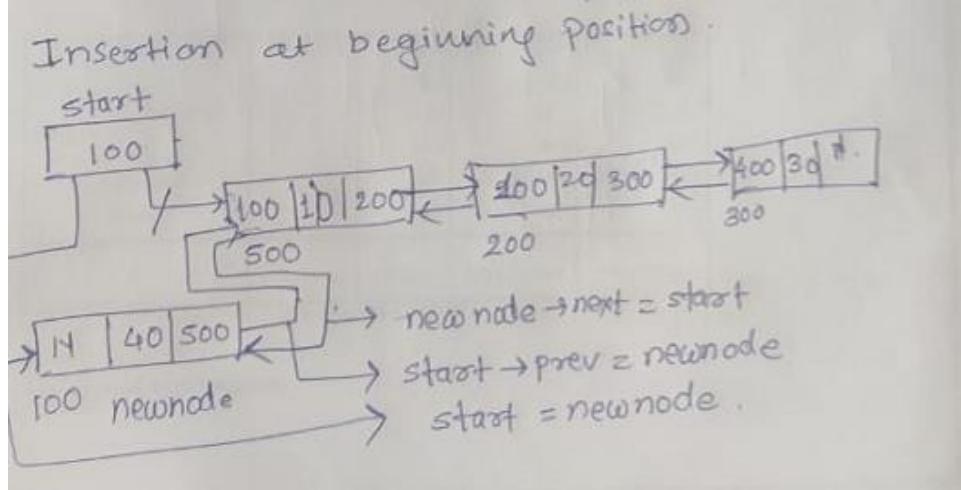


Fig.4.4.Shows inserting a node into the double linked list at the beginning.

- **Inserting a node at the end:**

The following algorithm steps are followed to insert a newnode at the end of the list:

- Create the new node using getnode()
 $\text{newnode}=\text{getnode}();$
- If the list is empty then $\text{start}=\text{newnode}$.
- If the list is not empty follow the steps given below:
 $\text{temp}=\text{start};$
 $\text{while}(\text{temp}->\text{next} \neq \text{NULL})$
 $\quad \text{temp}=\text{temp}->\text{next};$
 $\quad \text{temp}->\text{next}=\text{newnode};$
 $\quad \text{newnode}->\text{prev}=\text{temp};$

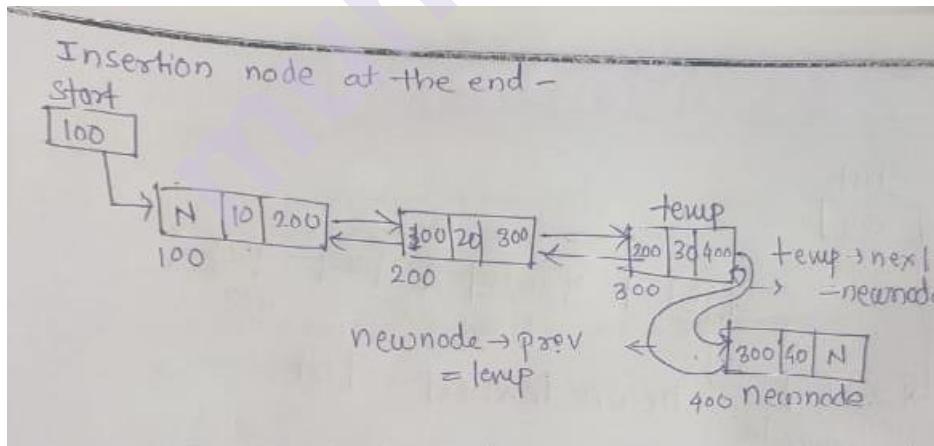


Figure 4.5 shows inserting a node into the double linked list at the end.

- **Inserting a node at an intermediate position:**

The following algorithm steps are followed, to insert a new node in an intermediate position in thelist:

- Create the newnode using getnode().
 $\text{newnode}=\text{getnode}();$

- Check that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:
 $\text{newnode} \rightarrow \text{next} = \text{temp} \rightarrow \text{next};$
 $\text{newnode} \rightarrow \text{prev} = \text{temp};$
 $\text{temp} \rightarrow \text{next} \rightarrow \text{prev} = \text{newnode};$
 $\text{temp} \rightarrow \text{next} = \text{newnode};$

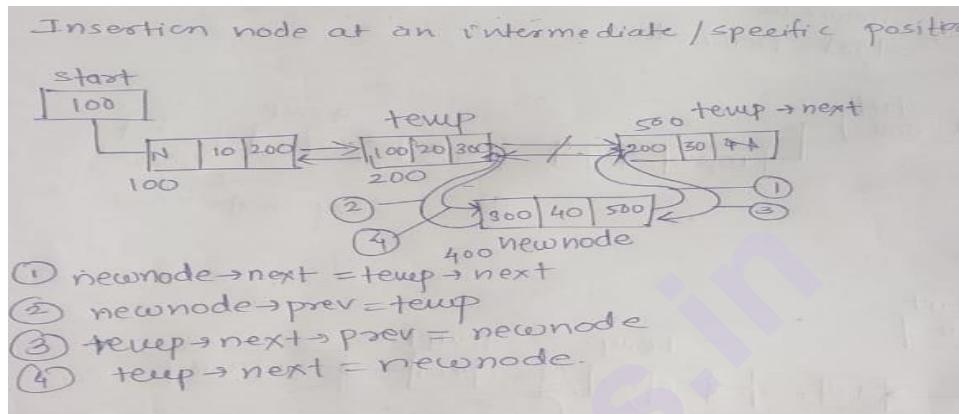


Figure 4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

4.5.3. Deletion any element from the linked list

Deleting a node in the doubly linked list, there are 3 cases-

- Delete a node from beginning position
- Delete a node from last position
- Delete a node from intermediate position.

• Deleting a node at the beginning:

The following algorithm steps are followed, to delete a node at the beginning of the list:

- If list is empty then display ‘Empty List’ message.
- If the list is not empty, follow the steps given below:
 $\text{temp} = \text{start};$
 $\text{start} = \text{start} \rightarrow \text{next};$
 $\text{start} \rightarrow \text{prev} = \text{NULL};$
 $\text{free}(\text{temp});$

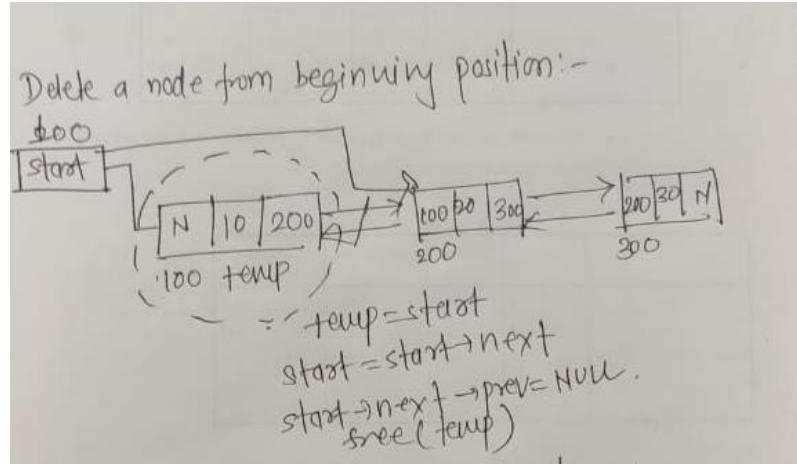


Figure 4.7 Delete a node from beginning position

- **Deleting a node at the end:**

The following steps are followed to delete a node at the end of the list:

- If list is empty then display ‘Empty List’ message
- If the list is not empty, follow the steps given below:

```

temp=start;
while(temp->next !=NULL)
{
    temp=temp->next;
}
temp->prev->next = NULL;free(temp);

```

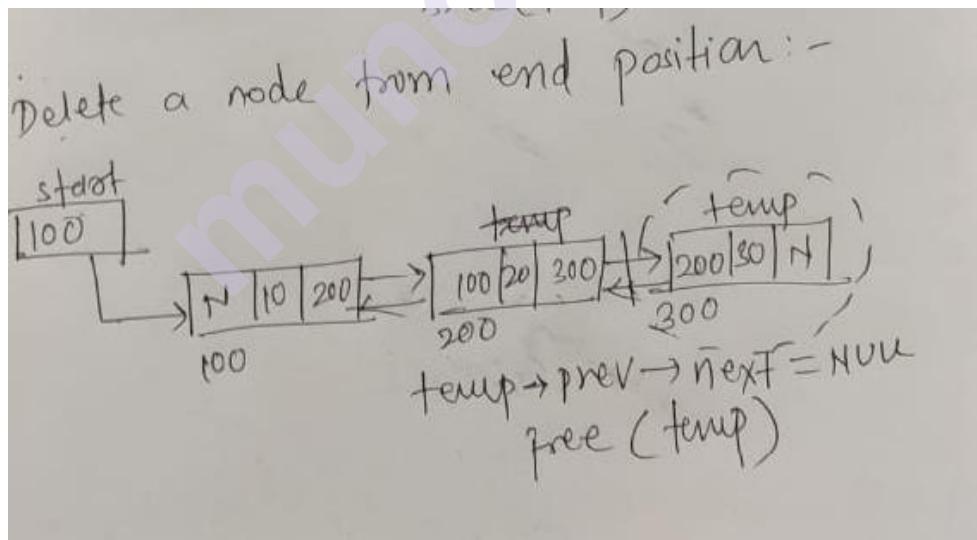


Figure 4.8.showsdeleting a node at the end of a double linked list.

- **Deleting a node at Intermediate position:**

The following steps are followed, to delete a node from an intermediate position in the list(List must contain more than two nodes).

- If list is empty then display ‘Empty List’ message.
- If the list is not empty, follow the steps given below:

- Get the position of the node to delete.
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid.

Then perform the following steps:

```
if(pos>1&&pos<nodectr)
```

```
{
```

```
    temp = start;i= 1;
```

```
    while(i<pos)
```

```
{
```

```
    temp = temp ->next;i++;
```

```
}
```

```
    temp -> next ->prev = temp ->prev;
```

```
    temp ->prev -> next = temp -> next;
```

```
    free(temp);
```

```
    printf("\nnode deleted..");
```

```
}
```

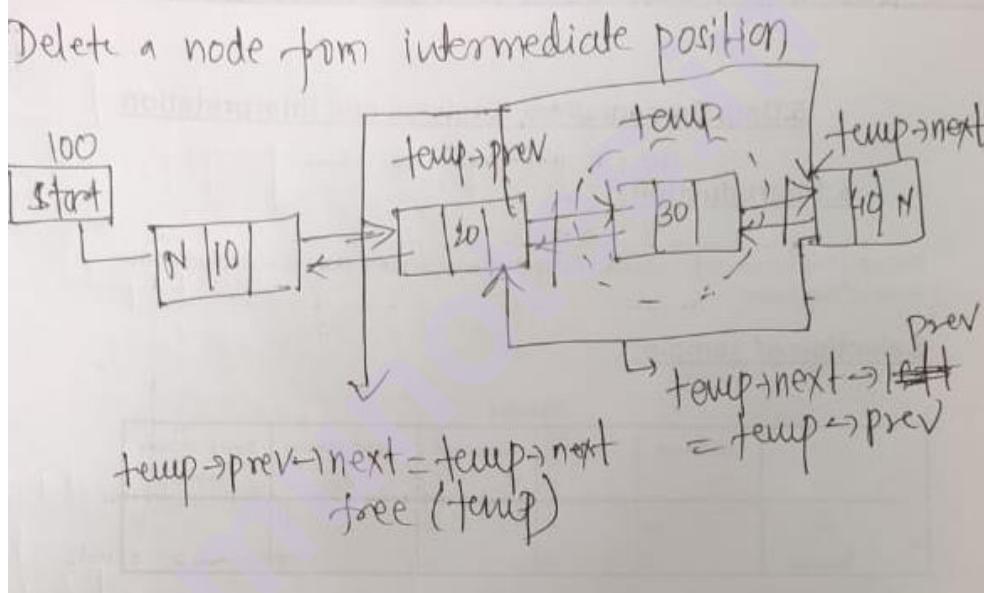


Figure 4.9 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

4.6 PROGRAM ON IMPLEMENTATION OF DOUBLY LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct dlinklist
{
    structdlinklist *prev;
    int data;
    structdlinklist *next;
};
```

```

typedef struct dlinklist node;
node *start = NULL;
node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode ->prev = NULL;
    newnode -> next = NULL;
    return newnode;
}
int countnode(node *start)
{
    if(start == NULL)
        return 0;
    else
        return 1 + countnode(start -> next);
}
int menu()
{
    intch;
    clrscr();
    printf("\n 1.Create");
    printf("\n-----");
    printf("\n 2. Insert a node at beginning ");
    printf("\n 3. Insert a node at end");
    printf("\n 4. Insert a node at middle");
    printf("\n-----");
    printf("\n 5. Delete a node from beginning");
    printf("\n 6. Delete a node from Last");
    printf("\n 7. Delete a node from Middle");
    printf("\n-----");
    printf("\n 8. Traverse the list in forward direction ");
    printf("\n 9. Traverse the list from backward direction ");
    printf("\n-----");
    printf("\n 10.Count the Number of nodes in the list");
    printf("\n 11.Exit");
    printf("\n\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}
void createdlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    for(i = 0; i < n; i++)
    {
        newnode = getnode();

```

```

if(start == NULL)
    start = newnode;
else
{
    temp = start;
    while(temp -> next)
        temp = temp -> next;
    temp -> next = newnode;
    newnode -> prev = temp;
}
}

void traverse_forward()
{
    node *temp;
    temp = start;
    printf("\n The contents of List: ");
    if(start == NULL )
        printf("\n Empty List");
    else
    {
        while(temp != NULL)
        {
            printf("\t %d ", temp -> data);
            temp = temp -> next;
        }
    }
}

void traverse_backward()
{
    node *temp;
    temp = start;
    printf("\n The contents of List: ");
    if(start == NULL)
        printf("\n Empty List");
    else
    {
        while(temp -> next != NULL)
            temp = temp -> next;
    }
    while(temp != NULL)
    {
        printf("\t%d", temp -> data);
        temp = temp -> prev;
    }
}

void dll_insert_beg()
{
    node *newnode;
    newnode = getnode();

```

```

if(start == NULL)
    start = newnode;
else
{
    newnode -> next = start;
    start -> prev = newnode;
    start = newnode;
}
}

void dll_insert_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
        start = newnode;
    else
    {
        temp = start;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newnode;
        newnode -> prev = temp;
    }
}

void dll_insert_mid()
{
    node *newnode,*temp;
    intpos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos - nodectr >= 2)
    {
        printf("\n Position is out of range..");
        return;
    }
    if(pos > 1 && pos < nodectr)
    {
        temp = start;
        while(ctr < pos - 1)
        {
            temp = temp -> next;
            ctr++;
        }
        newnode -> prev = temp;
        newnode -> next = temp -> next;
        temp -> next -> prev = newnode;
        temp -> next = newnode;
    }
}

```

```

else
    printf("position %d of list is not a middle position ", pos);
}
void dll_delete_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty list");
        getch();
        return ;
    }
    else
    {
        temp = start;
        start = start -> next;
        start ->prev = NULL;
        free(temp);
    }
}
void dll_delete_last()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty list");
        getch();
        return ;
    }
    else
    {
        temp = start;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp ->prev -> next = NULL;
        free(temp);
        temp = NULL;
    }
}
void dll_delete_mid()
{
    int i = 0, pos, nodectr;
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty List");
        getch();
        return;
    }
    else

```

```

{
    printf("\n Enter the position of the node to delete: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos>nodectr)
    {
        printf("\nthis node does not exist");
        getch();
        return;
    }
    if(pos> 1 && pos<nodectr)
    {
        temp = start;
        i= 1;
        while(i <pos)
        {
            temp = temp -> next;
            i++;
        }
        temp -> next ->prev = temp ->prev;
        temp ->prev -> next = temp -> next;
        free(temp);
        printf("\n node deleted..");
    }
    else
    {
        printf("\n It is not a middle position..");
        getch();
    }
}
void main(void)
{
    intch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch(ch)
        {
            case 1 :
                printf("\n Enter Number of nodes to create: ");
                scanf("%d", &n);
                createdlist(n);
                break;
            case 2 :
                dll_insert_beg();
                break;
            case 3 :
                dll_insert_end();

```

```

        break;
case 4 :
    dll_insert_mid();
    break;
case 5 :
    dll_delete_beg();
    break;
case 6 :
    dll_delete_last();
    break;
case 7 :
    dll_delete_mid();
    break;
case 8 :
    traverse_forward ();
    break;
case 9 :
    traverse_backward();
    break;
case 10 :
    printf("\n Number of nodes: %d", countnode(start));
    break;
case 11:
    exit(0);
}
getch();
}
}

```

4.7. CIRCULAR SINGLY LINKED LIST

The linked list where the last node points the start node is called circular linked list. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Hence we can move from last node to the start node of the list very efficiently. Hence accessing of any node is much faster than singly linked list. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular singly linked list is shown in figure 3.6.1.

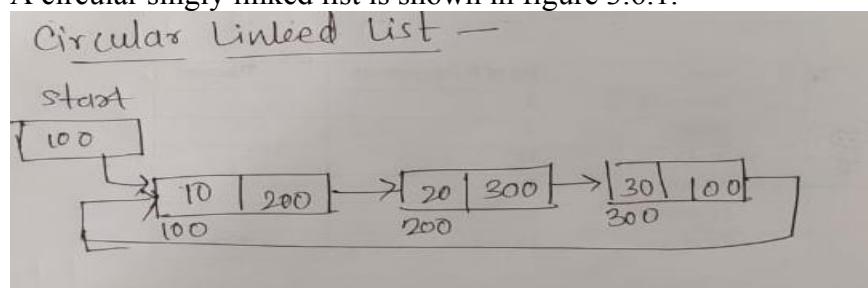


Figure 4.7.1.Circular Singly Linked List

Advantages of circular linked list:

- Accessibility of a number node in the list
- No null link problem
- Merging and splitting operations implemented easily
- Saves time when you want to go from last node to first node

Disadvantages of circular linked list

- The list will go infinite loop, if proper case is not taken.
- It is not easy to reverse the list elements.

4.8 APPLICATIONS IN CIRCULAR LINKED LIST

1. It is used in operating system. When multiple applications are running in PC operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application.
2. To repeat songs in the music player
3. Escalator which will run circularly uses the circular linked list

4.9 OPERATIONS IN A CIRCULAR SINGLE LINKED LIST ARE

- 4.9.1. Creation of node in circular linked list
- 4.9.2. Insertion a node in circular linked list
- 4.9.3. Delete a node in circular linked list
- 4.9.4. Displaying/Traversing a node in circular linked list

4.9.1.Creating a circular single Linked List with ‘n’ number of nodes:

The following algorithm steps are to be followed to create ‘n’ number of nodes:

- Create the new node using getnode().
newnode = getnode();
- If the list is empty, assign new node as start.
start = newnode;
- If the list is not empty, follow the steps given below:
temp = start;
while(temp -> next != NULL)
temp = temp -> next;
temp -> next = newnode;
- Repeat the above steps ‘n’ times.
- newnode -> next = start;

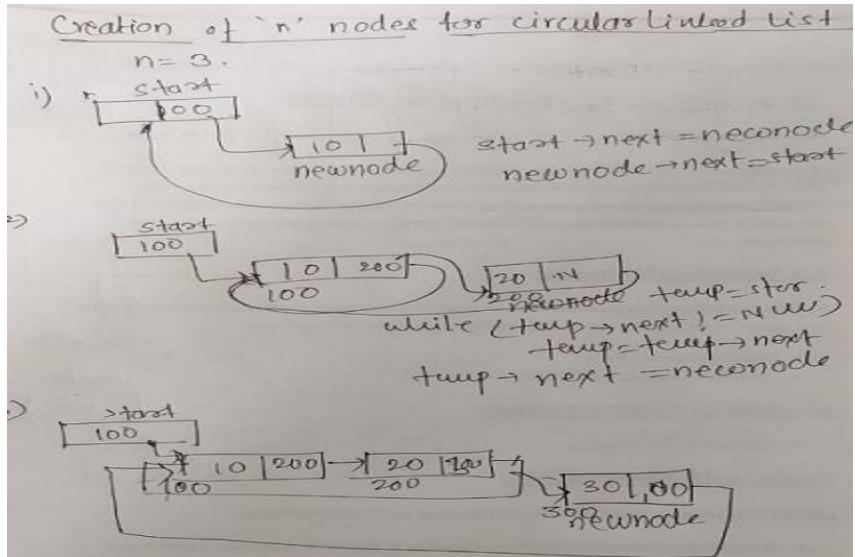


Fig.4.10. Creation of circular linked list

The function createlist(), is used to create ‘n’ number of nodes:

4.9.2. Insertion a node in circular linked list

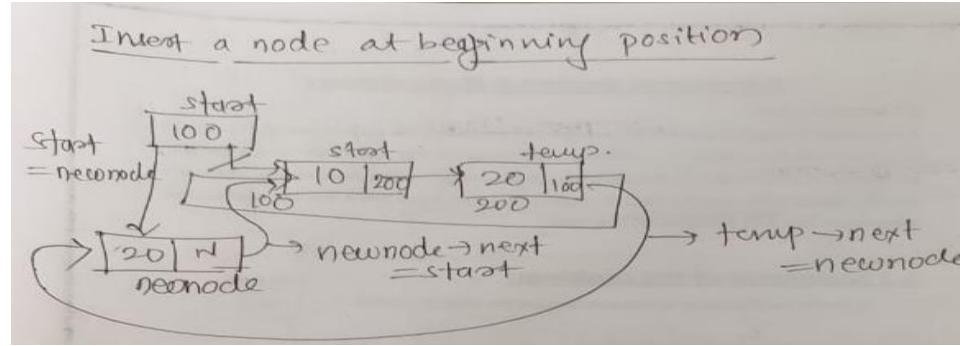
Inserting a node in the doubly linked list, there are 3 cases-

- Inserting a node at beginning position
- Inserting node at last position
- Inserting node at intermediate position.

• Inserting a node at the beginning:

The following algorithm steps are to be followed to insert a new node at the beginning of the circular list:

- create the new node using getnode().
 $\text{newnode} = \text{getnode}();$
- If the list is empty, assign new node as start.
 $\text{start} = \text{newnode};$
 $\text{newnode} \rightarrow \text{next} = \text{start};$
- If the list is not empty, follow the steps given below:
 $\text{temp} = \text{start};$
 $\text{while}(\text{temp} \rightarrow \text{next} \neq \text{start})$
 $\quad \text{temp} = \text{temp} \rightarrow \text{next};$
 $\quad \text{newnode} \rightarrow \text{next} = \text{start};$
 $\quad \text{start} = \text{newnode};$
 $\quad \text{temp} \rightarrow \text{next} = \text{start};$



4.11. shows inserting a node into the circular single linked list at the beginning.

- **Inserting a node at the end:**

The following algorithm steps are followed to insert a new node at the end of the list:

- create the new node using getnode().
 $\text{newnode} = \text{getnode}();$
- If the list is empty, assign new node as start.
 $\text{start} = \text{newnode};$
 $\text{newnode} \rightarrow \text{next} = \text{start};$
- If the list is not empty follow the steps given below:
 $\text{temp} = \text{start};$
 $\text{while}(\text{temp} \rightarrow \text{next} \neq \text{start})$
 $\quad \text{temp} = \text{temp} \rightarrow \text{next};$
 $\quad \text{temp} \rightarrow \text{next} = \text{newnode};$
 $\quad \text{newnode} \rightarrow \text{next} = \text{start};$

The function `cll_insert_end()`, is used for inserting a node at the end.

Figure 4.12 shows inserting a node into the circular single linked list at the end.

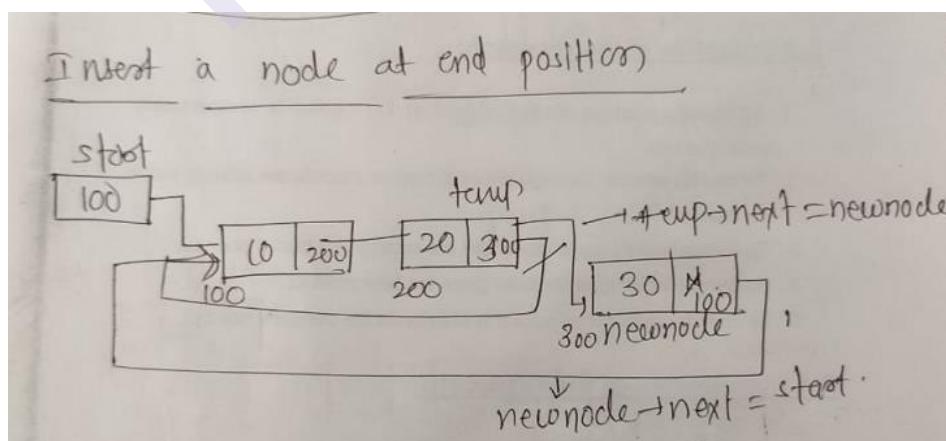


Figure 4.12 Inserting a node at the end

- **Inserting a node at intermediate position:**

The following algorithm steps are followed, to insert a new node in an intermediate position in the list:

Step 1: Create the new node using getnode().

```
newnode = getnode();
```

Step 2: Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

Step 3: Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

Step 4: After reaching the specified position, follow the steps given below:

```
temp1 -> next = newnode;
```

```
newnode -> next = temp;
```

- Let the intermediate position be 3.

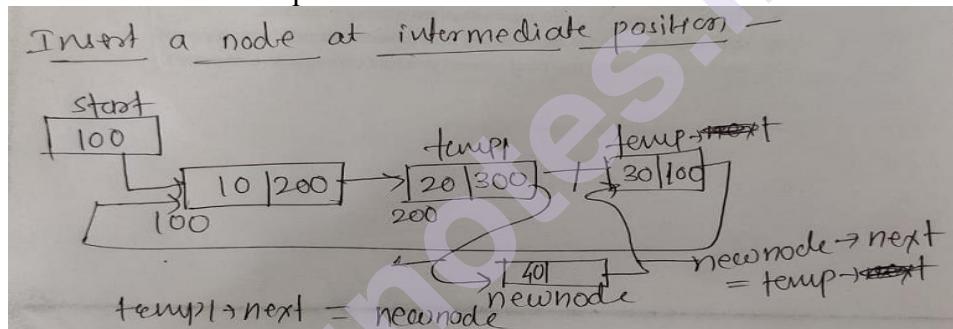


Figure 4.13 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

4.9.3 Deletion any element from the linked list

Deleting a node in the doubly linked list, there are 3 cases-

- Delete a node from beginning position
- Delete a node from last position
- Delete a node from intermediate position.

- **Deleting a node at the beginning:**

The following algorithm steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message 'Empty List'.

- If the list is not empty, follow the steps given below:

```
temp1 = temp = start;
while(temp1 -> next != start)
    temp1 = temp1 -> next;
    start = start -> next;
    temp1 -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL*.

The function `cll_delete_beg()`, is used for deleting the first node in the list

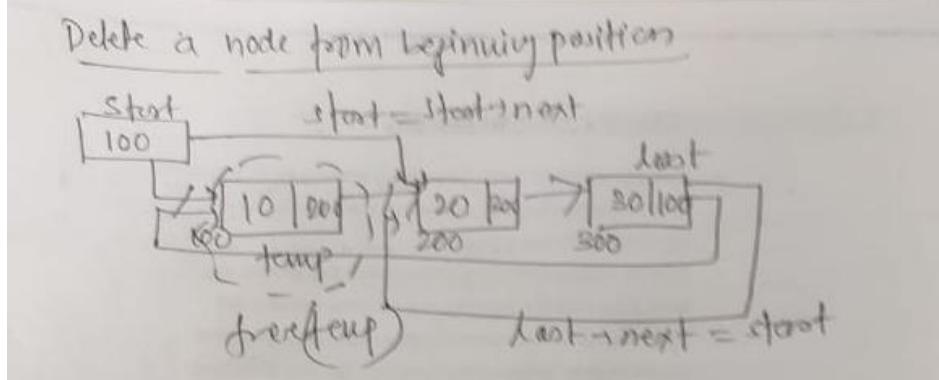


Figure 4.14 Deleting a node at beginning.

- **Deleting a node at the end:**

The following algorithm steps are followed to delete a node at the end of the list:

- If the list is empty, display a message ‘Empty List’.

- If the list is not empty, follow the steps given below:

```

temp = start;
temp1 = start;
while(temp -> next != start)
{
    temp1 = temp;
    temp = temp -> next;
}
temp1 -> next = start;

```

- After deleting the node, if the list is empty then `start = NULL`.

The function `cll_delete_last()`, is used for deleting the last node in the list.

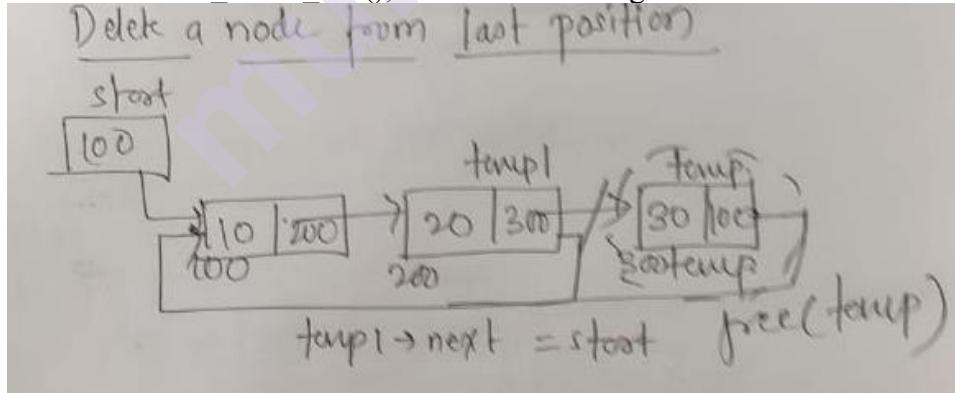


Figure 4.15 Deleting a node at the end

- **Deleting a node at intermediate position.**

The following algorithm steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display ‘Empty List’ message
- If the list is not empty, follow the steps given below.

```

if(pos> 1 && pos<nodectr)
{
temp = temp1 = start; ctr = 1;
while(ctr<pos)
{
    temp1 = temp;
    temp = temp -> next;
    ctr++;
}
temp1 -> next = temp -> next;
free(temp);
printf("\n node deleted..");
}

```

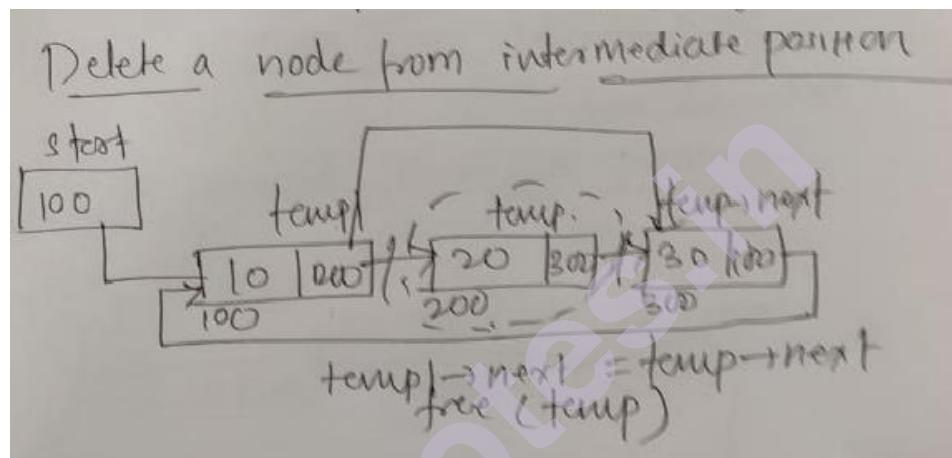


Figure 4.16 .Deleting a node in intermediate position

4.9.4 Traversing a circular single linked list from left to right:

The following algorithm steps are followed, to traverse a list from left to right:

- If list is empty then display ‘Empty List’ message.
- If the list is not empty, follow the steps given below:

```

temp = start;
do
{
    printf("%d", temp -> data);
    temp = temp -> next;
} while(temp != start);

```

4.10 PROGRAM ON IMPLEMENTATION OF CIRCULAR SINGLE LINKED LIST

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct cslinklist
{

```

```

int data;
struct cslinklist *next;
};

typedef struct cslinklist node;
node *start = NULL;
int nodectr;
node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> next = NULL;
    return newnode;
}
int menu()
{
    int ch;
    clrscr();
    printf("\n 1. Create a list ");
    printf("\n\n-----");
    printf("\n 2. Insert a node at beginning ");
    printf("\n 3. Insert a node at end");
    printf("\n 4. Insert a node at middle");
    printf("\n\n-----");
    printf("\n 5. Delete a node from beginning");
    printf("\n 6. Delete a node from Last");
    printf("\n 7. Delete a node from Middle");
    printf("\n\n-----");
    printf("\n 8. Display the list");
    printf("\n 9. Exit");
    printf("\n\n-----");
    printf("\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}
void createlist(int n)
{
    int i;
    node *newnode,*temp;
    nodectr = n;
    for(i = 0; i < n ; i++)
    {
        newnode = getnode();
        if(start == NULL)
        {
            start = newnode;
        }
        else
        {

```

```

        temp = start;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newnode;
    }
}
newnode ->next = start; /* last node is pointing to starting node */
}
void display()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right): ");
    if(start == NULL )
        printf("\n Empty List");
    else
    {
        do
        {
            printf("\t %d ", temp -> data);
            temp = temp -> next;
        } while(temp != start);
    printf(" NULL ");
    }
}
void cll_insert_beg()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
        newnode -> next = start;
    }
    else
    {
        temp = start;
        while(temp -> next != start)
            temp= temp -> next;
        newnode -> next = start;
        start = newnode;
        temp -> next = start;
    }
    printf("\n Node inserted at beginning..");
    nodectr++;
}
void cll_insert_end()
{
    node *newnode, *temp;
    newnode = getnode();

```

```

if(start == NULL )
{
    start = newnode;
    newnode -> next = start;
}
else
{
    temp = start;
    while(temp -> next != start)
        temp = temp -> next;
    temp -> next = newnode;
    newnode -> next = start;
}
printf("\n Node inserted at end..");
nodectr++;
}

void cll_insert_mid()
{
    node *newnode, *temp, *prev;
    int i, pos ;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    if(pos> 1 &&pos<nodectr)
    {
        temp = start;
        temp1= temp;
        i= 1;
        while(i <pos)
        {
            temp1 = temp;
            temp = temp -> next;
            i++;
        }
        temp1 -> next = newnode;
        newnode -> next = temp;
        nodectr++;
        printf("\n Node inserted at middle..");
    }
    else
    {
        printf("position %d of list is not a middle position ", pos);
    }
}

void cll_delete_beg()
{
    node *temp, *last;
    if(start == NULL)
    {
        printf("\n No nodes exist..");
    }
}

```

```

        getch();
        return ;
    }
else
{
    last = temp = start;
    while(last -> next != start)
        last= last -> next;
    start = start -> next;
    last -> next = start;
    free(temp);
    nodectr--;
    printf("\n Node deleted..");
    if(nodectr == 0)
        start = NULL;
}
}

void cll_delete_last()
{
    node *temp,*prev;
    if(start == NULL)
    {
        printf("\n No nodes exist..");
        getch();
        return ;
    }
else
{
    temp = start;
    temp1 = start;
    while(temp -> next != start)
    {
        temp1= temp;
        temp = temp -> next;
    }
    temp1 -> next = start;
    free(temp);
    nodectr--;
    if(nodectr == 0)
        start = NULL;
    printf("\n Node deleted..");
}
}

void cll_delete_mid()
{
    int i = 0, pos;
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n No nodes exist..");

```

```

        getch();
        return ;
    }
else
{
    printf("\n Which node to delete: ");
    scanf("%d", &pos);
    if(pos>nodectr)
    {
        printf("\nThis node does not exist");
        getch();
        return;
    }
    if(pos> 1 && pos<nodectr)
    {
        temp=start;
        temp1 = start;
        i= 0;
        while(i < pos - 1)
        {
            temp1 = temp;
            temp = temp -> next ;
            i++;
        }
        temp1 -> next = temp -> next;
        free(temp);
        nodectr--;
        printf("\n Node Deleted..");
    }
    else
    {
        printf("\n It is not a middle position..");
        getch();
    }
}
void main(void)
{
    int result;
    intch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch(ch)
        {
        case 1 :
            if(start == NULL)
            {
                printf("\n Enter Number of nodes to create: ");

```

```

        scanf("%d", &n);
        createlist(n);
        printf("\nList created..");
    }
else
    printf("\n List is already Exist..");
    break;
case 2 :
    cll_insert_beg();
    break;
case 3 :
    cll_insert_end();
    break;
case 4 :
    cll_insert_mid();
    break;
case 5 :
    cll_delete_beg();
    break;
case 6 :
    cll_delete_last();
    break;
case 7 :
    cll_delete_mid();
    break;
case 8 :
    display();
    break;
case 9 :
    exit(0);
}

```

4.11 REPRESENTATION OF LINKED LIST USING HEADER NODE

A header linked list is a type of linked list which always contains a special node called the header node at the very beginning of the linked list. It is an extra node kept at the front of a list.

Such a node does not represent an item in the linked list. The information part of this node can be used to store any global information about the whole linked list. A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list.

For example, the picture below

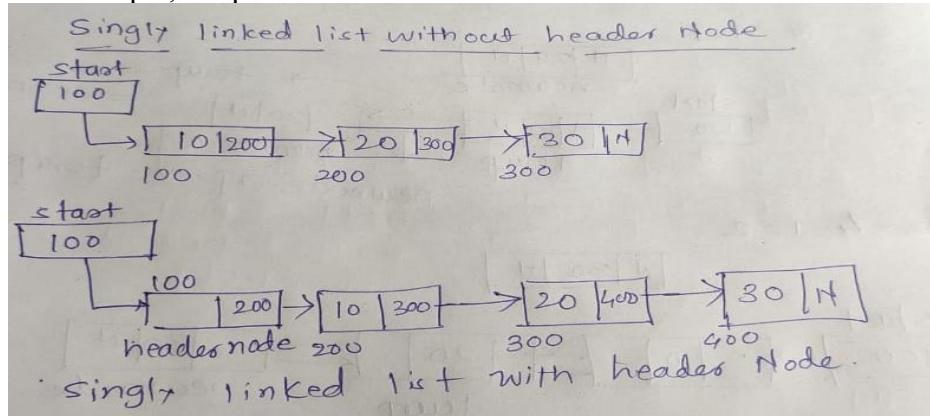


Fig.4.17 Representation of Polynomial using Linked List

Advantages of header node-

- In header node, you can maintain count variable which gives number of nodes in the list. You can update header node count member whenever you add /delete any node. It will help in getting list count without traversing list.
- In addition to address of first node you can also store the address of last node. So that if you want to insert node at the rear end you don't have to iterate from the beginning of the list. Also in the case of DLL(Doubly Linked List) if you want to traverse from rear end it helps.
- We can also use it to store the pointer to the current node in the linked list which eliminates the need of an external pointer during the traversal of the linked list.
- While inserting a new node in the linked list we don't need to check whether start node is null or not because the header node will always exist and we can insert a new node just after that.

4.12 REPRESENTATION OF POLYNOMIAL USING LINKED LIST

A polynomial is composed of different terms where each of them holds a coefficient and an exponent.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts: one is the coefficient and other is the exponent $13x^2 + 36x$, here 13 and 36 are coefficients and 2, 1 is its exponential value.

The sign of each coefficient and exponent is stored within the coefficient and the exponent itself

Additional terms having equal exponent is possible one.

The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent. The linked list

can be used to represent a polynomial of any degree. Simply the information field is changed according to the number of variables used in the polynomial. If a single variable is used in the polynomial the information field of the node contains two parts: one for coefficient of variable and the other for exponent of variable.

Let us consider an example to represent a polynomial using linked list as follows:

Polynomial: $3x^3 - 4x^2 + 2x - 9$

Polynomial can be represented in the following ways.

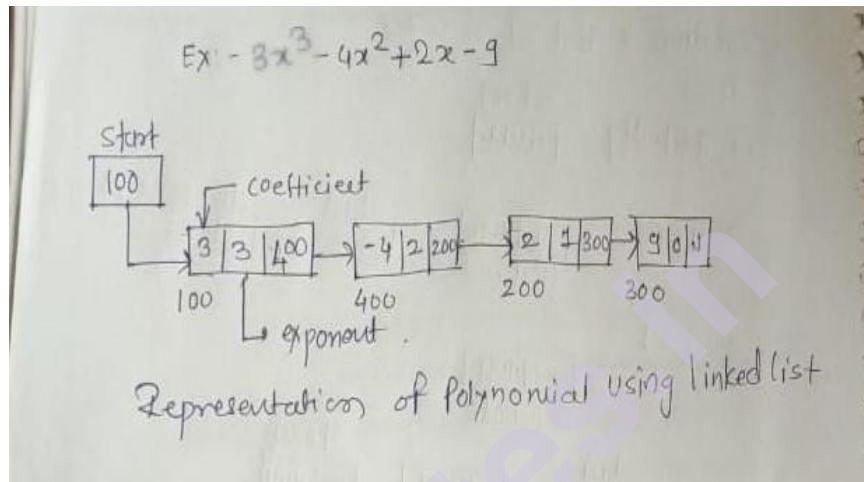


Fig.4.18 Representation of Polynomial using Linked List

Representation of a polynomial using the linked list is beneficial when the operations on the polynomial like addition and subtractions are performed. The resulting polynomial can also be traversed very easily to display the polynomial.

The linked list is used for polynomial arithmetic because it can have separate coefficient and exponent fields for representing each term of polynomial. Hence there is no limit for exponent. We can have any number as an exponent. The arithmetic operation on any polynomial of arbitrary length is possible using linked list.

Program for polynomial creation with help of linked list:

```
#include <conio.h>
#include <stdio.h>
#include <malloc.h>
struct link
{
    Int coef; int expo;
    struct link *next;
};
typedef struct link node;
node * getnode()
{
    node *newnode;
```

```

newnode =(node *) malloc( sizeof(node) );
printf("\n Enter Coefficient : ");
fflush(stdin);
scanf("%f",&newnode ->coef);
printf("\n Enter Exponent : ");
fflush(stdin);
scanf("%d",&newnode ->expo);
newnode ->next = NULL;
returnnewnode;
}
node * create_poly (node *p )
{
charch;
node *temp,*newnode;
while( 1 )
{
    printf ("\n Do U Want polynomial node (y/n): ");
    ch = getche();
    if(ch == 'n')
        break;
    newnode = getnode();
    if( p == NULL )
        p= newnode;
    else
    {
        temp = p;
        while(temp->next != NULL )
            temp = temp->next;
        temp->next = newnode;
    }
}
return p;
}
void display (node *p)
{
node *t = p;
while (t != NULL)
{
    printf("+ %.2d", t ->coef);
    printf("X^ %d", t -> expo);
    t=t -> next;
}
}
void main()
{
node *poly1 = NULL , *poly2 = NULL,*poly3=NULL;
clrscr();
printf("\nEnter First Polynomial..(in ascending-order of exponent)");
poly1 = create_poly (poly1);

```

```

printf("\nEnter Second Polynomial..(in ascending-order of
exponent)");
poly2 = create_poly (poly2);
clrscr();
printf("\n Enter Polynomial 1: ");
display (poly1);
printf("\n Enter Polynomial 2: ");
display (poly2);
getch();
}

```

4.13 REPRESENTATION OF SPARSE ARRAY USING LINKED LIST

A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing $m \times n$ matrix. Sparse matrices are those matrices that have the maximum elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Advantages of sparse matrix instead of over a simple matrix

- **Storage:** As we know, a sparse matrix that contains lesser non-zero elements than zero so less memory can be used to store elements. It evaluates only the non-zero elements.
- **Computing time:** In the case of searching n sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. The zeroes in the matrix are of no use to store zeroes with non-zero elements. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

Representation of Sparse Matrix

The non-zero elements can be stored with triples, i.e., rows, columns, and value. The sparse matrix can be represented in the following ways:

- Array representation
- Linked list representation
- **Array Representation**

The 2 d array can be used to represent a sparse matrix in which there are three rows named as:

1. Row: It is an index of a row where a non-zero element is located.

2. Column: It is an index of the column where a non-zero element is located.
3. Value: The value of the non-zero element is located at the index (row, column).

Let us consider 4X4 matrix elements are-

1	0	0	-4
0	6	0	0
0	0	7	0
0	5	0	0

As we can observe above, that sparse matrix is represented using triplets, i.e., row, column, and value. In the above sparse matrix, there are 11 zero elements and 5 non-zero elements. If the size of the sparse matrix is increased, then the wastage of memory space will also be increased. The above sparse matrix can be represented in the tabular form shown as below:

Row	Col	Value
4	4	5
0	0	1
0	3	-4
1	1	6
2	2	7
3	1	5

In the above table structure, the first column is representing the row number, the second column is representing the column number and third column represents the non-zero value at index(row, column). The size of the table depends upon the number of non-zero elements in the sparse matrix.

- **Linked List Representation**

In linked list representation, linked list data structure is used to represent a sparse matrix. In linked list representation, each node consists of four fields whereas, in array representation, there are three fields, i.e., row, column, and value. The following are the fields in the linked list:

- Row: It is an index of row where a non-zero element is located.
- Column: It is an index of column where a non-zero element is located.
- Value: It is the value of the non-zero element which is located at the index (row, column).
- Next node: It stores the address of the next node.

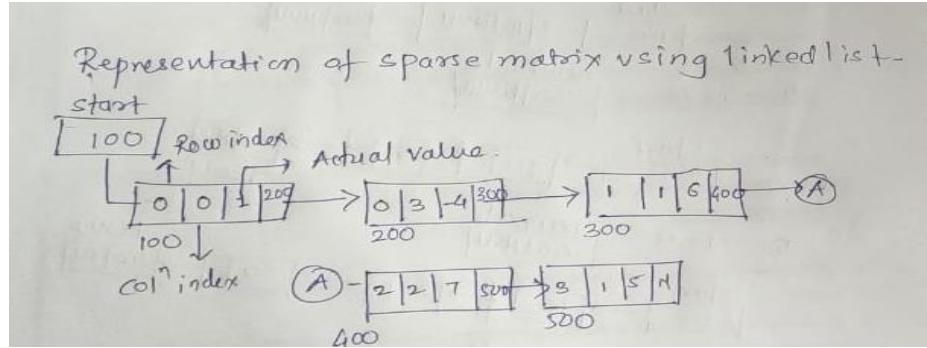


Fig.4.19. Representation of Sparse Matrix using Linked List

4.14 SUMMARY

- The doubly linked list has two pointer fields. One field is previous link field and another is next link field. By using these two pointers we can access any node efficiently whereas in singly linked list only one pointer field which stores forward pointer.
- In circular list the next pointer of last node points to start node, whereas in doubly linked list each node has two pointers. The main advantage of circular list over doubly linked list is that with the help of single pointer field we can access start node quickly. So amount of memory get saved.
- The header node is the first node of linked list. This node is useful for getting starting address of linked list.
- Polynomials and Sparse Matrix are two important applications of arrays and linked lists.

4.15 LIST OF REFERENCES

- <https://sites.google.com/site/datastructuresite/Home-Page/books>
- <https://www.hackerearth.com/practice/data-structures/linked-list/singly-linked-list/tutorial/>
- <https://dscet.ac.in/questionbank/cse/third-sem/CS8391-DATA-STRUCTURES.pdf>

4.16 Bibliography

- Yashavant Kanetkar Data Structures Through C ,BPB Publications, ISBN: 9788176567060,
- Goodrich, Tamassia, Goldwasser, —Data Structures and Algorithms in C++ || , Wiley publication, ISBN-978-81-265-1260-7
- D S Malik, Data Structures Using java, Thomson, India Edition 2006.
- Sahni S, Data Structures, Algorithms and Applications in java, McGraw-Hill, 2002.
- Samanta D, Classic Data Structures, Prentice-Hall of India, 2001.

6. Tremblay P, and Sorenson P G, Introduction to Data Structures with Applications, Tata McGraw-Hill,
 7. Jean-Paul Tremblay and Paul G. Sorenson, An Introduction to Data Structures with Applications, Tata McGraw Hill
 8. Tanenbaum, Data Structures using C & C++, PHI
 9. Robert L. Kruse, Data Structures and Program Design in C, PHI
 10. Seymour Lipschutz, “Data structures with C”, Schaum’s Publication 5.
Aaron Tanenbaum, “Data Structures using C”, Pearson Education
-

4.17 UNIT END EXERCISES

1. What is a Doubly Linked List? Write a ‘C’ program to add new node at the end of a Doubly Linked List.
2. What is difference between the singly and doubly link list. Mention the significance of a circular linked list.
3. What is doubly linked list? What advantages does a doubly linked list have over linear linked lists?
4. How a linked list can be used to represent a polynomial of type:-
 $9x^2y^2 - 8xy^2 + 10xy + 9y^2$
5. Describe the applications of linked list on polynomial Expressions.
6. Describe the operations are performed on Circular Linked List.
7. Explain the operations are performed on the singly linked list.
8. What is header node and what is the use of it?
9. Write a C function to find product of all elements of Linked List?
10. What are advantages of doubly linked list over singly linked list?
11. Why the linked list is used for representation of polynomial?
12. What are advantages of circular linked list over doubly linked list?
13. Write program to implement circular linked list.
14. Write program to implement doubly linked list.
15. What is Circular Linked List? State the advantages and disadvantages of Circular Link List Over Doubly Linked List and Singly Linked List



Unit III

5

STACK

Unit Structure

- 5.0 Objectives
- 5.1 Introduction
- 5.2 Operations on the Stack Memory Representation of Stack
- 5.3 Array Representation of Stack
- 5.4 Applications of Stack
 - 5.4.1 Evaluation of Arithmetic Expression
 - 5.4.2 Matching Parenthesis
 - 5.4.3 Infix and postfix operations
 - 5.4.4 Memory management
 - 5.4.5 Recursion
- 5.5 Summary
- 5.6 References
- 5.7 Unit End Exercise

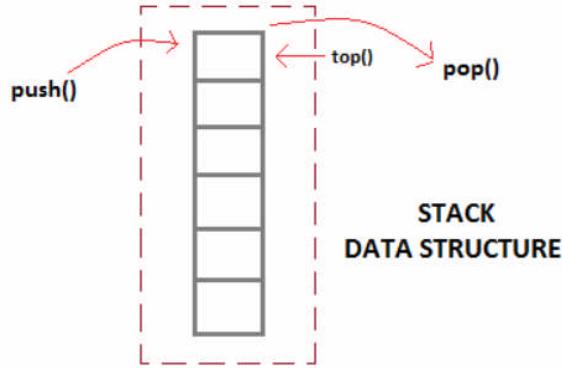
5.0 OBJECTIVES

After reading through this chapter, you will be able to –

- To understand the basic concepts about stacks.
- To impart the operations on the stack memory representation.
- To understand the array representation of stack.
- To understand the applications of stack.
- To understand the evaluation of arithmetic operations on stack.
- To understand the matching parenthesis.
- To understand the infix and postfix operations on stack.

5.1 INTRODUCTION

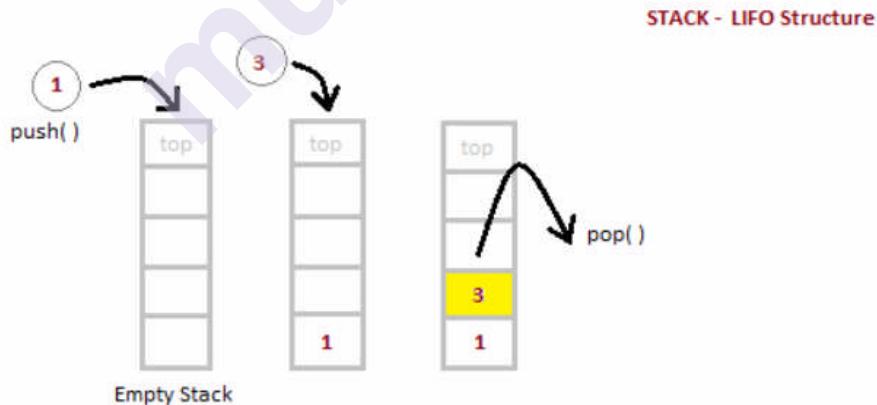
- ➔ Stack is a simple data structure that allows adding and removing elements in a particular order.
- ➔ Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



- Stack is an ordered list of similar data type.
- Stack is a LIFO (Last in First out) structure or we can say FILO (First in Last out).
- push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called Top.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

5.2 OPERATIONS ON THE STACK MEMORY REPRESENTATION OF STACK

→ Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.
The "pop" operation removes the item on top of the stack.

Basic Operations:

- ➔ Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations
 - push() – Pushing (storing) an element on the stack.
 - pop() – Removing (accessing) an element from the stack.
- ➔ When data is Pushed onto stack. To use a stack efficiently, we need to check the status of stack as well.
- ➔ For the same purpose, the following functionality is added to stacks
 - peek() – get the top data element of the stack, without removing it.
 - isFull() – check if stack is full.
 - isEmpty() – check if stack is empty.
- ➔ At all times, we maintain a pointer to the last Pushed data on the stack. As this pointer always represents the top of the stack, hence named top.
- ➔ The top pointer provides top value of the stack without actually removing it.

Algorithm for PUSH operation:

1. Check if the stack is full or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element.

Algorithm for POP operation:

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

➔ Below we have a simple C++ program implementing stack data structure while following the object-oriented programming concepts.

```
# include<iostream>
using namespace std;
class Stack
{
    int top;
public:
    int a[10]; //Maximum size of Stack
Stack()
{
    top = -1;
}
// declaring all the function
void push(int x);
int pop();
void isEmpty();
};
```

```

// function to insert data into stack
void Stack::push(int x)
{
    if(top >= 10)
    {
        cout << "Stack Overflow \n";
    }
    else
    {
        a[++top] = x;
        cout << "Element Inserted \n";
    }
}
// function to remove data from the top of the stack
int Stack::pop()
{
    if(top < 0)
    {
        cout << "Stack Underflow \n";
        return 0;
    }
    else
    {
        int d = a[top--];
        return d;
    }
}

// function to check if stack is empty
void Stack::isEmpty()
{
    if(top < 0)
    {
        cout << "Stack is empty \n";
    }
    else
    {
        cout << "Stack is not empty \n";
    }
}

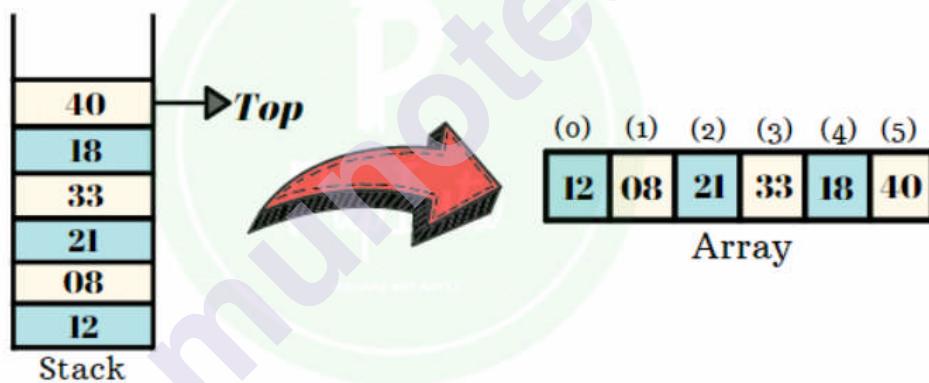
// main function
int main() {
    Stack s1;
    s1.push(10);
    s1.push(100);
}

```

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

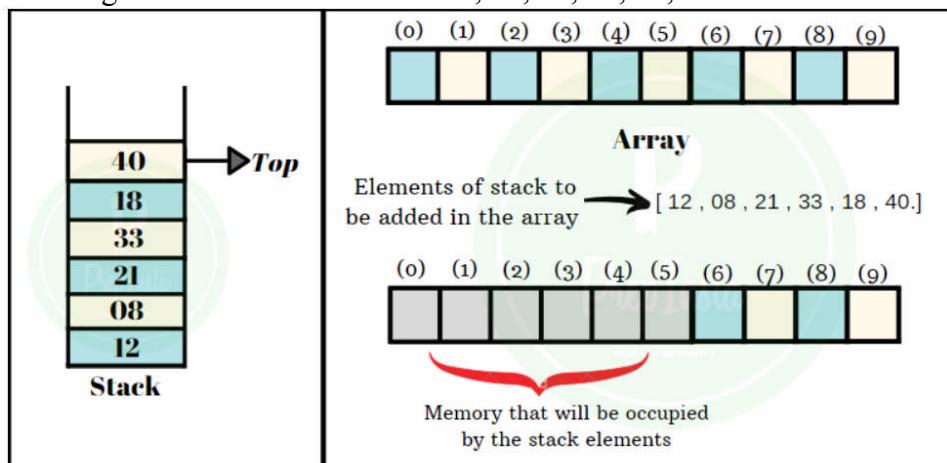
5.3 ARRAY REPRESENTATION OF STACK

- A stack is a data structure that can be represented as an array. Let us learn more about Array representation of a stack.
- An array is used to store an ordered list of elements. Using an array for representation of stack is one of the easy techniques to manage the data.
- There is a major difference between an array and a stack.
 - Size of an array is fixed.
 - While, in a stack, there is no fixed size since the size of stack changed with the number of elements inserted or deleted to and from it.
- The difference, an array can be used to represent a stack by taking an array of maximum size big enough to manage a stack.



Example:

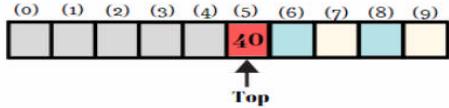
We are given a stack of elements: 12, 08, 21, 33, 18, 40.



Step1:

- Push (40)
- Top = 40
- Element is inserted at a[5]

Push (40)



Step2:

- Push (18)
- Top = 18
- Element is inserted at a[4]

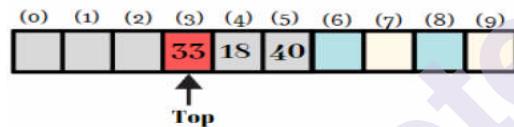
Push (18)



Step3:

- Push (33)
- Top = 33
- Element is inserted at a[3]

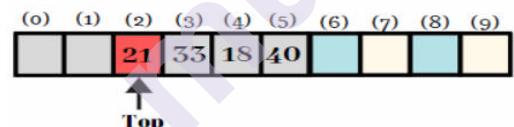
Push (33)



Step 4:

- Push (21)
- Top = 21
- Element is inserted at a[2]

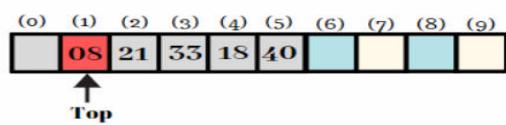
Push (21)



Step 5:

- Push (8)
- Top = 8
- Element is inserted at a[1]

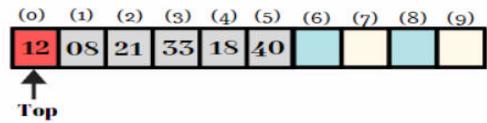
Push (08)



Step 6:

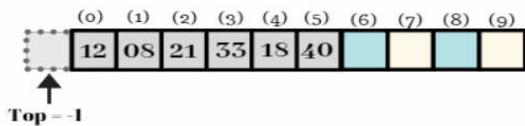
- Push (12)
- Top = 12
- Element is inserted at a[0]

Push (12)



Step 7:

- Top = -1
- End of array



Code of stack (using structure)

```
// Program for Implementation of stack (array) using structure
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack
struct Stack {
    int top;
    int maxSize;
    int* array;
};

struct Stack* create(int max)
{
    struct Stack* stack = (struct Stack*) malloc (sizeof (struct Stack));
    stack->maxSize = max;
    stack->top = -1;
    stack->array = (int*) malloc(stack->maxSize * sizeof(int));
    //here above memory for array is being created
    // size would be 10*4 = 40
    return stack;
}

// Checking with this function is stack is full or not
// Will return true if stack is full else false
//Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
    if(stack->top == stack->maxSize - 1){
        printf ("Will not be able to push maxSize reached\n");
    }
    // Since array starts from 0, and maxSize starts from 1
    return stack->top == stack->maxSize - 1;
}

// By definition the Stack is empty when top is equal to -1
// Will return true if top is -1
int isEmpty (struct Stack* stack)
{
```

```

        return stack->top == -1;
    }

// Push function here, inserts value in stack and increments stack top by 1
void push (struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("We have pushed %d to stack\n", item);
}

// Function to remove an item from stack. It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

int main ()
{
    struct Stack* stack = create (10);

    push(stack, 5);
    push(stack, 10);
    push(stack, 15);

    int flag=1;
    while(flag)
    {
        if(! isEmpty(stack))
            printf("We have popped %d from stack\n", pop(stack));
        else
            printf("Can't Pop stack must be empty\n");
            printf("Do you want to Pop again? Yes: 1 No: 0\n");
            scanf("%d",&flag);
    }
    return 0;
}
Output:
We have pushed 5 to stack

```

We have pushed 10 to stack
 We have pushed 15 to stack
 We have popped 15 from the stack
 Do you want to Pop again? Yes: 1 No: 0
 0

5.4 APPLICATIONS OF STACK

→ In a stack, only limited operations are performed because it is restricted data structure. The elements are deleted from the stack in the reverse order.

Following are the applications of stack:

1. Evaluation of Arithmetic Expression
2. Matching Parenthesis
3. Expression Conversion
 - i. Infix to Postfix
 - ii. Postfix to Infix
 - iii. Prefix to Infix
4. Memory Management

5.4.1 Evaluation of Arithmetic Expression:

→ Stack data structure is used for evaluating the given expression. For example, consider the following expression

$$5 * (6 + 2) - 12 / 4$$

→ Since parenthesis has the highest precedence among the arithmetic operators,

$(6 + 2) = 8$ will be evaluated first. Now, the expression becomes

$$5 * 8 - 12 / 4$$

→ * and / have equal precedence and their associativity is from left-to-right. So, start evaluating the expression from left-to-right.

$$5 * 8 = 40 \text{ and } 12 / 4 = 3$$

→ Now, the expression becomes

$$40 - 3$$

→ And the value returned after the subtraction operation is 37.

5.4.2 Matching Parenthesis:

→ One of the most important applications of stacks is to check if the parentheses are balanced in a given expression. The compiler generates an error if the parentheses are not matched.

→ Here are some of the balanced and unbalanced expressions:

BALANCED EXPRESSION	UNBALANCED EXPRESSION
$(a+b)$	$(a+b$
$[(c-d)*e]$	$[(c-d * e]$
$\{()\}[]$	$\{[()]\}$

→ Consider the above-mentioned unbalanced expressions:

- The first expression $(a + b$ is unbalanced as there is no closing parenthesis given.
- The second expression $[(c - d * e)]$ is unbalanced as the closed round parenthesis is not given.
- The third expression $\{ [(]) \}$ is unbalanced as the nesting of square parenthesis and the round parenthesis are incorrect.

→ Steps to find whether a given expression is balanced or unbalanced

- Input the expression and put it in a character stack.
- Scan the characters from the expression one by one.
- If the scanned character is a starting bracket ('(' or '{' or '['), then push it to the stack.
- If the scanned character is a closing bracket (')' or ')' or ']'), then pop from the stack and if the popped character is the equivalent starting bracket, then proceed. Else, the expression is unbalanced.
- After scanning all the characters from the expression, if there is any parenthesis found in the stack or if the stack is not empty, then the expression is unbalanced.

5.4.3 Expression Conversion:

→ There are three popular methods used for representation of an expression

Infix	$A + B$	Operator between operands.
Prefix	$+ AB$	Operator before operands.
Postfix	$AB +$	Operator after operands.

i. Conversion of Infix to Postfix:

Step 1: Consider the next element in the input.

Step 2: If it is operand, display it.

Step 3: If it is opening parenthesis, insert it on stack.

Step 4: If it is an operator, then

- If stack is empty, insert operator on stack.
- If the top of stack is opening parenthesis, insert the operator on stack
- If it has higher priority than the top of stack, insert the operator on stack.
- Else, delete the operator from the stack and display it, repeat Step 4.

Step 5: If it is a closing parenthesis, delete the operator from stack and display them until an opening parenthesis is encountered. Delete and discard the opening parenthesis.

Step 6: If there is more input, go to Step 1.

Step 7: If there is no more input, delete the remaining operators to output.

Example: Suppose we are converting $3*3/ (4-1) +6*2$ expression into postfix form.

Expression	Stack	Output
3	Empty	3
*	*	3
3	*	33
/	/	33*
(/(33*
4	/(33*4
-	/(-	33*4
1	/(-	33*41
)	-	33*41-
+	+	33*41-/
6	+	33*41-/6
*	+*	33*41-/62
2	+*	33*41-/62
	Empty	33*41-/62*+

So, the Postfix Expression is $33*41-/62*+$

ii. Conversion of Postfix to Infix:

Following is an algorithm for Postfix to Infix conversion:

Step 1: While there are input symbol left.

Step 2: Read the next symbol from input.

Step 3: If the symbol is an operand, insert it onto the stack.

Step 4: Otherwise, the symbol is an operator.

Step 5: If there are fewer than 2 values on the stack, show error /* input not sufficient values in the expression */

Step 6: Else,

a. Delete the top 2 values from the stack.

b. Put the operator, with the values as arguments and form a

string.

c. Encapsulate the resulted string with parenthesis.

d. Insert the resulted string back to stack.

Step 7: If there is only one value in the stack, that value in the stack is the desired infix string.

Step 8: If there are more values in the stack, show error /* The user input has too many values */

Example: Suppose we are converting efg-+he-sh-o+/* expression into Infix.

Following table shows the evaluation of Postfix to Infix

efg-+he-sh-o+/*	NULL
fg-+he-sh-o+/*	“e”
g-+he-sh-o+/*	“f” “e”
-+he-sh-o+/*	“g” “f” “e”
+he-sh-o+/*	“f”-“g” “e”
he-sh-o+/*	“e+f-g”
e-sh-o+/*	“h” “e+f-g”
-sh-o+/*	“e” “h” “e+f-g”
sh-o+/*	“h-e” “e+f-g”
h-o+/*	“s” “h-e” “e+f-g”
-o+/*	“h” “s” “h-e” “e+f-g”
o+/*	“h-s” “h-e” “e+f-g”
+/*	“o” “s-h” “h-e” “e+f-g”
/*	“s-h+o” “h-e” “e+f-g”
*	“(h-e)/(s-h+o)” “e+f-g”
NULL	“(e+f-g) * (h-e)/(s-h+o)”

So, the Infix Expression is $(e+f-g) * (h-e)/(s-h+o)$

iii. Prefix to Infix:

→ Algorithm for Prefix to Infix Conversion

Step 1: The reversed input string is inserted into a stack ->
prefixToInfix(stack)

Step 2: If stack is not empty,

a. Temp -> pop the stack

b. If temp is an operator,

 Write a opening parenthesis “(“ to show -> prefixToInfix(stack)

 Write temp to show -> prefixToInfix(stack)

 Write a closing parenthesis “)” to show

c. Else If temp is a space ->prefixToInfix(stack)

d. Else, write temp to show if stack.top! = space -
>prefixToInfix(stack)

Example: Suppose we are converting /-bc+-pqr expression from Prefix to Infix.

→ Following table shows the evaluation of Prefix to Infix Conversion

Expression	Stack
-bc+-pqr	Empty
/-bc+-pq	“q” “r”
/-bc+-	“p” “q” “r”
/-bc+	“p-q” “r”
/-bc	“p-q+r”
/-b	“c” “p-q+r”
/-	“b” “c” “p-q+r”
/	“b-c” “p-q+r”
NULL	“((b-c)/((p-q)+r))”

So, the Infix Expression is $((b-c)/((p-q)+r))$

5.4.4 Memory management:

→ The assignment of memory takes place in contiguous memory blocks.
We call this stack memory allocation because the assignment takes place
in the function call stack.

- The size of the memory to be allocated is known to the compiler. When a function is called, its variables get memory allocated on the stack.
- When the function call is completed, the memory for the variables is released. All this happens with the help of some predefined routines in the compiler.
- The user does not have to worry about memory allocation and release of stack variables.

```
int main ()
{
    // All these variables get memory
    // allocated on stack
    int f;
    int a[10];
    int c = 20;
    int e[n];
}
```

- The memory to be allocated to these variables is known to the compiler and when the function is called, the allocation is done and when the function terminates, the memory is released.

5.4.5 Recursion:

- Some computer programming languages allow a module or function to call itself. This technique is known as recursion.
- In recursion, a function α either calls itself directly or calls a function β that in turn calls the original function α. The function α is called recursive function.

- Example: a function calling itself.

```
int function (int value) {
    if (value < 1)
        return;
    function (value - 1);
    printf ("%d ", value);
}
```

- Example – a function that calls another function which in turn calls it again.

```
int function1(int value1) {
    if (value1 < 1)
        return;
    function2(value1 - 1);
    printf ("%d ", value1);
}
int function2(int value2) {
    function1(value2);
```

}

→ Properties:

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have

- Base criteria – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- Progressive approach – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

5.5 SUMMARY

→ In this chapter we studied Operations on the Stack Memory Representation like push (), pop (), peek (), isFull (), isEmpty () .

→ we are focused on array representation of stack; A stack is a data structure that can be represented as an array. Let us learn more about Array representation of a stack.

→ Also, we studied various applications of stack like evaluation of arithmetic expression,

Matching Parenthesis, Infix and postfix operations, Memory management and Recursion

5.6 REFERENCES

1. <https://www.tutorialspoint.com/>
2. <https://www.studytonight.com/>
3. <https://prepinsta.com/>
4. <https://www.tutorialride.com/>
5. <https://www.faceprep.in/>

5.7 UNIT END EXERCISE

1) Complete the class with all function definitions for a stack class stack

```
{  
int data [10];  
int top;  
public:  
stack () {top=-1;}  
void push ();  
void pop ();  
}
```

2) Change the following infix expression to postfix expression.

(A + B) *C+D/E-F

3) Evaluate the following postfix expression using a stack and show the contents of stack after execution of each operation:
50,40, +,18, 14, -, *, +

4) Each node of a STACK contains the following information, in addition to required pointer field:

- i) Roll number of the student
- ii) Age of the student

Give the structure of node for the linked stack in question TOP is a pointer which points to the topmost node of the STACK. Write the following functions.

- i) PUSH () - To push a node to the stack which is allocated dynamically
- ii) POP () - To remove a node from the stack and release the memory.



6

QUEUE

Unit Structure

- 6.0 Objectives
- 6.1 Introduction
- 6.2 Queue
- 6.3 Operations on the Queue
- 6.4 Memory Representation of Queue
- 6.5 Array representation of queue
- 6.6 Linked List Representation of Queue
- 6.7 Circular Queue
- 6.8 Deque
- 6.9 Priority Queue
- 6.10 Applications of Queues
- 6.11 Summary
- 6.12 Bibliography
- 6.13 Unit End Exercise

6.0 OBJECTIVES

- To understand the queue.
- To be able to implement queue and deque.
- To be able to recognize problem properties where queues, and deques are appropriate data structures.
- To be able to understand the applications of Queue.

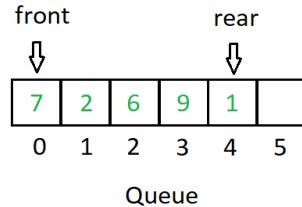
6.1 INTRODUCTION

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

6.2 QUEUE

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).

A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is **First In First Out (FIFO)**. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

6.3 OPERATIONS ON THE QUEUE

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

enqueue() – add (store) an item to the queue.

dequeue() – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient.

These are –

peek() – Gets the element at the front of the queue without removing it.

isfull() – Checks if the queue is full.

isempty() – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueueing (or storing) data in the queue we take help of rear pointer.

Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

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.

Implementation of enqueue() in C programming language –

Example

```
int enqueue(int data)
    if(isfull())
        return 0;

    rear = rear + 1;
    queue[rear] = data;

    return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation –

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.

Implementation of dequeue() in C programming language –

Example

```
int dequeue() {
    if(isempty())
        return 0;

    int data = queue[front];
    front = front + 1;

    return data;
}
```

6.4 MEMORY REPRESENTATION OF QUEUE

Like Stacks, Queues can also be represented in memory in two ways.

Using the contiguous memory like an array

Using the non-contiguous memory like a linked list

Using the Contiguous Memory like an Array

In this representation the Queue is implemented using the array.

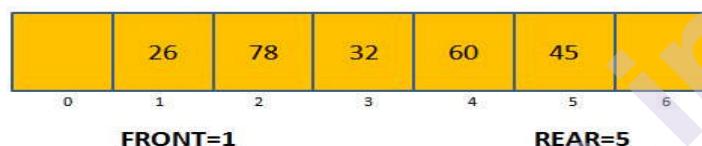
Variables used in this case are

QUEUE- the name of the array storing queue elements.

FRONT- the index where the first element is stored in the array representing the queue.

REAR- the index where the last element is stored in array representing the queue.

MAX- defining that how many elements (maximum count) can be stored in the array representing the queue.



Using the Non-Contiguous Memory like a Linked List

In this representation the queue is implemented using the dynamic data structure Linked List.

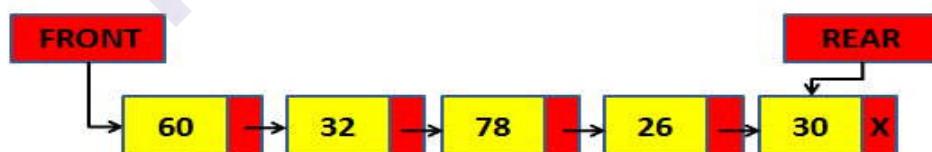
Using linked list for creating a queue makes it flexible in terms of size and storage.

You don't have to define the maximum number of elements in the queue.

Pointers (links) to store addresses of nodes for defining a queue are.

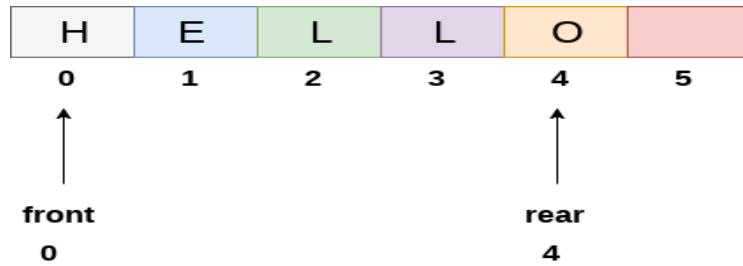
FRONT- address of the first element of the Linked list storing the Queue.

REAR- address of the last element of the Linked list storing the Queue.



6.5 ARRAY REPRESENTATION OF QUEUE

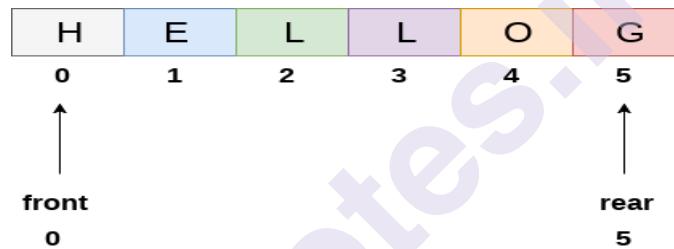
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



Queue

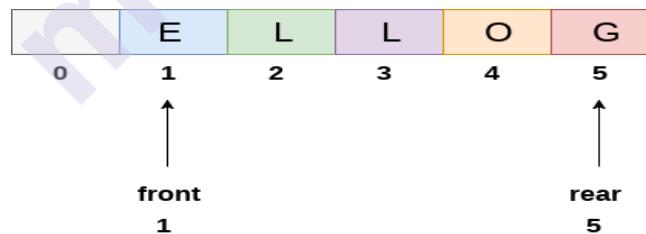
The above figure shows the queue of characters forming the English word "**HELLO**". Since, No deletion is performed in the queue till now, therefore the value of front remains -1 .

However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

6.6 LINKED LIST REPRESENTATION OF QUEUE

The array implementation can not be used for the large scale applications where the queues are implemented.

One of the alternative of array implementation is linked list implementation of queue.

The storage requirement of linked representation of a queue with n elements is $O(n)$ while the time requirement for operations is $O(1)$.

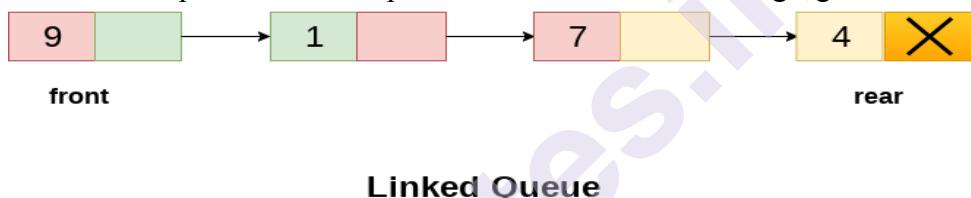
In a linked queue, each node of the queue consists of two parts i.e. data part and the link part.

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.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

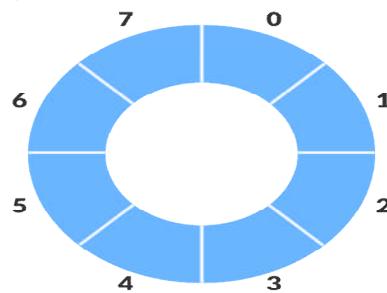
The linked representation of queue is shown in the following figure.



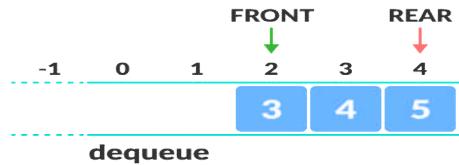
Linked Queue

6.7 CIRCULAR QUEUE

A circular queue is the extended version of a regular queue where the last element is connected to the first element. Thus forming a circle-like structure.



The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.



Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements).

This reduces the actual size of the queue.

How Circular Queue Works

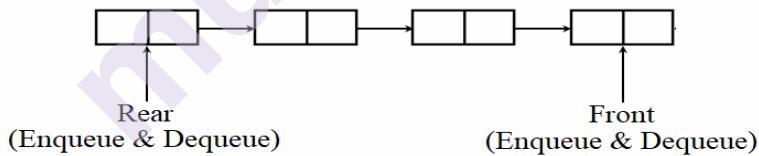
Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

Here, the circular increment is performed by modulo division with the queue size. That is,

if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)

6.8 DEQUE

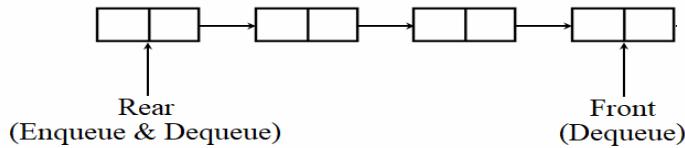
A deque is also a special type of queue. In this queue, the **enqueue and dequeue operations take place at both front and rear**. That means, we can insert an item at both the ends and can remove an item from both the ends. Thus, it may or may not adhere to the FIFO order:



It's used to save browsing history, perform undo operations, implement A-Steal job scheduling algorithm, or implement a stack or implement a simple queue.

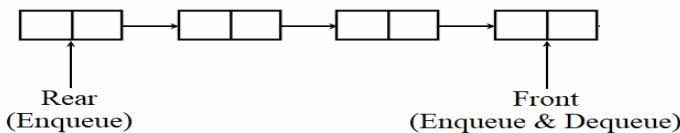
Further, it has two special cases: **input-restricted deque** and **output-restricted deque**.

In the first case, the enqueue operation takes place only at the rear, but the dequeue operation takes place at both rear and front:



An input-restricted queue is useful when we need to remove an item from the rear of the queue.

In the second case, the enqueue operation takes place at both rear and front, but the dequeue operation takes place only at the front:



An output-restricted queue is handy when we need to insert an item at the front of the queue.

6.9 PRIORITY QUEUE

Priority Queue is an extension of queue with following properties.

Every item has a priority associated with it.

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.

In the below priority queue, element with maximum ASCII value will have the highest priority.

Priority Queue		
	Initial Queue = {}	
Operation	Return value	Queue Content
insert (C)		C
insert (O)		C O
insert (D)		C O D
remove max	O	C D
insert (I)		C D I
insert (N)		C D I N
remove max	N	C D I
insert (G)		C D I G

Atypical priority queue supports following operations.

- insert(item, priority):** Inserts an item with given priority.
get Highest Priority (): Returns the highest priority item.
delete Highest Priority(): Removes the highest priority item.

How to implement priority queue?

Using Array: A simple implementation is to use array of following structure.

```
struct item {  
    int item;  
    int priority;  
}
```

insert() operation can be implemented by adding an item at end of array in O(1) time.

Applications of Priority Queue:

- 1) CPU Scheduling
- 2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
- 3) All queue applications where priority is involved.

6.10 APPLICATIONS OF QUEUES

Queues are used in various applications in Computer Science-

- Job scheduling tasks of CPU.
- Printer's Buffer to store printing commands initiated by a user.
- Input commands sent to CPU by devices like Keyboard and Mouse.
- Document downloading from internet.
- User Requests for call center services.
- Order Queue for Online Food Delivery Chains.
- Online Cab Booking applications.

6.11 SUMMARY

A queue is kind of like the opposite of a stack. Instead of having a “last-in, first-out” structure, it is “first-in, first-out”. Inserting A, B, and then C, will make the first removed A, then B, then C. The first item inserted is called the “head” of the queue, and the last item is the “tail”.

6.12 BIBLIOGRAPHY

1. <https://csveda.com/queue-applications-introduction-and-memory-representation/>
2. https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm
3. <https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/>
4. <https://www.javatpoint.com/array-representation-of-queue>
5. <https://www.programiz.com/dsa/circular-queue>
6. <https://www.baeldung.com/cs/types-of-queues>

6.13 UNIT END EXERCISE

1. Explain Queue data structure in details?
2. Explain briefly operations on queue?
3. What is circular queue ? how to implement it?
4. Write short notes on :
 - a) Priority queue
 - b) Dequeue
 - c) Enqueue
- 5) Explain Applications of Queue and Dequeue



Unit IV

7

SORTING AND SEARCHING TECHNIQUES

Unit Structure :

- 7.0 Objectives
- 7.1 Sorting
 - 7.1.1 Bubble
 - 7.1.2 Selection
 - 7.1.3 Insertion
 - 7.1.4 Merge Sort
- 7.2 Searching
 - 7.2.1 Indexed Sequential Searches
 - 7.2.2 Binary Search
- 7.3 Summary
- 7.4 List of References
- 7.5 Model Questions

7.0 OBJECTIVES

This chapter would make you understand the following concepts:

- Sorting and its types, also its time complexity.
- Searching and its types also its advantages and disadvantages.

7.1 SORTING

Sorting is a process, in which we can perform on data structure in order to arrange data elements in ascending and descending order. There are different methods that can be used to arrange data in ascending or descending order. There are two types of sorting.

Internal sorting:

Internal sorting is used when the data to be sort, all stored in main memory. Bubble sort, selection sort, Insertion sort, Quick sort, Radix sort are internal sorting techniques.

External sorting:

If all the data that is to be sorted do not fit entirely in the main memory, external sorting is required. An external sort requires the use of external memory such as disks or tapes during sorting. In external sorting, some part of the data is loaded into the main memory, sorted using any

internal sorting technique and written back to the disk in some intermediate file. This process continues until all the data is sorted. Merge sort is an example of external sorting.

7.1.1 Bubble sort

- The simplest sorting algorithm is bubble sort. A bubble sort is an internal exchange sort. This sorting technique is named bubble because of the logic is similar to the bubble in water. When a bubble are formed it is small at the bottom and when it moves up it becomes bigger and bigger i.e. bubbles are in ascending order of their size from the bottom to the top.
- The key idea of the bubble sort is to make pairwise comparisons and exchange the position of the pair if they are out of order.
- This sorting method proceeds by scanning through the elements one pair at a time, and swapping any adjacent pairs it finds to be out of order.
- Bubble sort starts by comparing first element with second element; If first element is greater than second element then it will swap both the elements and then move on to compare second element with third element and so on.
- Let Array[Max] is integer array, N is size of array. Bubble sort is simplest sorting algorithm where after every pass the largest elements of array bubbles up toward the last place/index. If we have total n elements in array then to sort all these elements we need to repeat this process for n-1 times. In each pass, sorting starts from starting index and end on (N-pass-1) element.

Bubble Sort Algorithm

- Let Array[Max] is array, N is size of array.
 1. Repeat for pass=1 to N-1.
 2. Repeat for j=0 to N-pass-1.
 3. If array[j]>array[j+1].
 4. Interchange a[j] with a[j+1].
 5. Print sorted Array.
- Time complexity of bubble sort in worst case is O(N²), in Best case is O(N) and in Average case is O(N²).
Program:- Bubble sort
Input:- array of numbers
Output:- numbers arranged in ascending order.

Bubble sort example:- consider the following numbers

75	16	80	10	18	6
----	----	----	----	----	---

Pass 1: compare the elements j and j+1 and do swap / no swap operations

Pass	j	Elements						operation
1	0	75	16	80	10	18	6	Swap 75 and 16.
	1	16	75	80	10	18	6	No swap for 75 and 80.
	2	16	75	80	10	18	6	Swap 80 and 10.
	3	16	75	10	80	18	6	Swap 80 and 18.
	4	16	75	10	18	80	6	Swap 80 and 6.
		16	75	10	18	6	80	Pass 1 finish.

At the end of pass first largest element is filtered and placed at the end
compare the elements j and j+1 and do swap / no swap operations

Pass 2:

Pass	j	Elements						operation
2	0	16	75	10	18	6	80	No swap for 16 and 75.
	1	16	75	10	18	6	80	Swap 75 and 10.
	2	16	10	75	18	6	80	Swap 75 and 18.
	3	16	10	18	75	6	80	Swap 75 and 6.
		16	10	18	6	75	80	Pass 2 finish.

At the end of pass second largest element is filtered and placed at its proper place

Pass 3:

Pass	j	Elements						operation
3	0	16	10	18	6	75	80	swap 16 and 10.
	1	10	16	18	6	75	80	No swap for 16 and 18.
	2	10	16	18	6	75	80	Swap 18 and 6.
		10	16	6	18	75	80	Pass 3 finish.

At the end of pass third largest element is filtered and placed at the end

Pass 4:

Pass	j	Elements						operation
4	0	10	16	6	18	75	80	No Swap for 10 and 16.
	1	10	16	6	18	75	80	Swap 16 and 6.
		10	6	16	18	75	80	Pass 4 finish.

At the end of pass fourth largest element is filtered and placed at the end

Pass 5:

Pass	j	Elements						operation
5	0	10	6	16	18	75	80	swap 10 and 6.
		6	10	16	18	75	80	Pass 5 finish.

At the end of pass fifth largest element is filtered and placed at the end and finally sorted

Advantages of Bubble sort

- 1) Bubble sort is easy and simple to implement.
- 2) Elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

Disadvantages of Bubble sort

- 1) Bubble sort is slow as it takes $O(n!)$ comparison to complete sorting.
- 2) It is not sufficient for large lists.

Bubble sort program:-

```
// program for implementation of Bubble Sort
void bubbleSort(int arr[], int n)
{
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] < arr[j+1])
            {
                // swap temp and arr[i]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
}
```

Output:- run:

Sorted array

9 5 4 3 2 1 1

BUILD SUCCESSFUL (total time: 0 seconds)

7.1.2 Selection sort

- Selection sort performs in-place comparison which means that the array is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
- During Iteration 1, smallest element in the array (index 0 to last index) is selected and that selected element is replaced with first position element.
- During Iteration 2 smallest element from sub array (starting from index 1 to last index) is selected and that selected element is replaced with second position element and so on.

- In other words , the idea of the selection sort is to search the smallest element in the list and exchange it with the element in the first position, Then, find the second smallest element and exchange it with the element in the second position, and so on until the entire array is sorted.
- In selection sort, iteration 1 looks for smallest element in the array and replace first position with that smallest element. After that iteration 2 look for smallest element present in the sub array, starting from index 1, till the last index and replace second position with that smallest element. This is repeated, until the array is completely sorted.

Selection sort algorithm

- Let array {Max} integer array, N is size of array.
- 1. Repeat for i=0 to N-2.
- 2. Set min=Array[i] and loc=i.
- 3. Repeat for j=i+1 to N-1.
- 4. IF MIN>Array[j] then.
- 5. Set MIN=[j] and loc=j;
- 6. Interchange Array [j] with Array [loc].
- Time complexity of selection sort in worst case is O(N²), in best case is O(N) and in average case is O(N²).

Selection sort example:- consider the following numbers

75	16	80	10	18	6
----	----	----	----	----	---

Following procedure is used:

Pass	I/O	Elements						operation
Pass 1	Input	75	16	80	10	18	6	MIN=6 Swap 75 with 6.
	Output	6	16	80	10	18	75	After Pass 1 smallest element i.e. 6 is at its proper position.
Pass 2	Input	6	16	80	10	18	75	MIN = 10 Swap 16 with 10.
	Output	6	10	80	16	18	75	After Pass 2 second smallest element i.e. 10 is at its proper position.
Pass 3	Input	6	10	80	16	18	75	MIN = 16 Swap 80 with 16.
	Output	6	10	16	80	18	75	After Pass 3 16 is at its proper position.
Pass 4	Input	6	10	16	80	18	75	MIN = 18 Swap 80 with 18.
	Output	6	10	16	18	80	75	After Pass 4 18 is at its proper position.
Pass 5	Input	6	10	16	18	80	75	MIN = 75 Swap 80 with 75.
	Output	6	10	16	18	75	80	After pass 5 75 is at its proper position

Selection sort program:-

```
void sort(int arr[], int n)
{
    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++)
    {

```

```

// Find the minimum element in unsorted array
int min_idx = i;
for (int j = i+1; j < n; j++)
    if (arr[j] > arr[min_idx])
        min_idx = j;

// Swap the found minimum element with the first
// element
int temp = arr[min_idx];
arr[min_idx] = arr[i];
arr[i] = temp;
}
}

```

run:

Sorted array

9 5 4 2 1 1

BUILD SUCCESSFUL (total time: 0 seconds)

Advantages of selection sort.

1. Selection sort is simple and easy to implement.
2. It gives 60% performance improvement over bubble sort.

Disadvantages of selection sort.

Selection sort performs poor when dealing with large number of elements.

7.1.3 Insertion Sort

- Insertion sort is different from other sorting algorithms as sorting starts from the index 1(not 0), i.e. the element at index 1 compared with index 0 first. If element at index 1 is smaller than element at index 0 then we insert the element at index 1 to index 0.

Insertion sort algorithm

- Let array [Max] is integer array , N is size of array
- 1. Repeat for i=to N-1.
- 2. Set temp=Array[i]and j=i-1.
- 3. While j>=0 and array [j]>temp do.
- 4. Array [j+1] = Array[j].
- 5. J=j-1.
- 6. Array [j+1]=temp.

Insertion sort example:- consider the following numbers

75	16	80	10	18	6
----	----	----	----	----	---

Following procedure is used:

	I/O	Elements						operation
Pass 1	Input	75	16	80	10	18	6	Compare 16 with 75 and insert 16 at index 0.
	Output	16	75	80	10	18	6	
	Output	16	75	80	10	18	6	
Pass 2	INPUT	16	75	80	10	18	6	Compare 80 with 75, no change. Compare 80 with 16, no change.
	output	16	75	80	10	18	6	
Pass 3	Input	16	75	80	10	18	6	Compare 10 with 80. Compare 10 with 75. Compare 10 with 16. Insert 10 at index 0.
	Output	10	16	75	80	18	6	
	Output	10	16	75	80	18	6	
Pass 4	Input	10	16	75	80	18	6	Compare 18 with 80. Compare 18 with 75. Compare 18 with 16. Compare 18 with 10. Insert 18 at index 2.
	Output	10	16	18	75	80	6	
Pass 5	Input	10	16	18	75	80	6	Compare 6 with 80. Compare 6 with 75. Compare 6 with 18. Compare 6 with 16. Compare 6 with 10. Insert 6 at index 0.
	Output	6	10	16	18	75	80	Sorted string.

Insertion sort program:

```

void sort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

```

run:

5 6 11 12 13

BUILD SUCCESSFUL (total time: 0 seconds)

Radix sort example:- consider the following numbers

75	16	80	10	18	6
----	----	----	----	----	---

Pass 1: Bucket sort the numbers while scanning the input from left to right one's digit (LSD).

At the end of this pass 1 remove the elements from the buckets from 0 to 9 in first out manner. The elements will be 721,13 ,123, 35, 537, 27, 438, 9

Pass 2: now it will scan as per pass 1, but the ten's place (i.e next LSD)

At the end of this pass 2 remove the elements from the buckets from 0 to 9 in first out manner. The elements will be 9,13,7,21,123,27,35,537,438.

Pass 3: now it will scan as per pass 1, but the hundred's place (i.e next MSD)

At the end of this pass 3 remove the elements from the buckets from 0 to 9 in first out manner. The elements will be 9,13,27,35,123,438,537,721. And the numbers are sorted.

0	1	2	3	4	5	6	7	8	9
721			13		35		537	468	
			123				27		

0	1	2	3	4	5	6	7	8	9
9	13	721	35						
		123	537						
		27	438						

0	1	2	3	4	5	6	7	8	9
9	123			438	537		721		
13									
27									
35									

7.1.4 MergeSort:

Algorithm:Input:- array of numbers

Output:-numbers arranged in ascending order

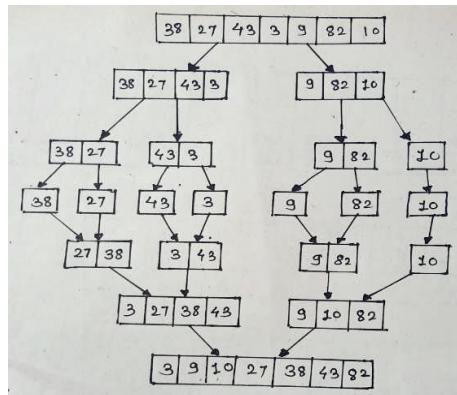
1. MergeSort(array A, int p, int r) {
2. if (p < r) { // we have at least 2 items

```

q = (p + r)/2
3. MergeSort(A, p, q) // sort A[p..q]
4. MergeSort(A, q+1, r) // sort A[q+1..r]
5. Merge(A, p, q, r) // merge everything together
}
}

```

Example:



```

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int [n1];
    int R[] = new int [n2];

    /*Copy data to temp arrays*/
    for (int i=0; i<n1; ++i)
        L[i] = arr[l + i];
    for (int j=0; j<n2; ++j)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarry array
    int k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {

```

```

        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}

/* Copy remaining elements of L[] if any */
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

/* Copy remaining elements of R[] if any */
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Find the middle point
        int m = (l+r)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m+1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

```

Advantages - Merge Sort

1. Merge sort algorithm is best case for sorting slow-access data e.g) tape drive.

2. Merge sort algorithm is better at handling sequential - accessed lists.

Disadvantages - Merge Sort

1. The running time of merge sort algorithm is $O(n \log n)$ which turns out to be the worse case.

2. Merge sort algorithm requires additional memory space of $O(n)$ for the temporary array TEMP.

7.2 SEARCHING

Searching is the process of discovery some particular element in the list. If the element is existing in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful. There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the organization of the list.

7.2.1 Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found, then location of the item is returned otherwise the algorithm return NULL. Linear search is mostly used to search an unordered list in which the items are not sorted.

Features of Linear Search Algorithm

It is used for unsorted and unordered small list of elements.

It has a time complexity of $O(n)$, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.

It has a very simple implementation.

The algorithm of linear search is given as follows

Algorithm

LINEAR_SEARCH (A, N, VAL)

Step 1: [INITIALIZE] SET POS = -1

Step 2: [INITIALIZE] SET I = 1

Step 3: Repeat Step 4 while $I \leq N$

Step 4: IF $A[I] = VAL$

SET POS = I

PRINT POS

Go to Step 6

[END OF IF]
 SET I = I + 1
 [END OF LOOP]
 Step 5: IF POS = -1
 PRINT " VALUE IS NOT PRESENTIN THE ARRAY "
 [END OF IF]
 Step 6: EXIT

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space	O(1)		

Program

```

int[] arr = {1, 3, 5, 8, 4, 3, 5, 0, 4, 9};
int item,flag=0;
item =3
for(int i = 0; i<10; i++)
{
  if(arr[i]==item)
  {
    flag = i+1;
    break;
  }
  else
    flag = 0;
}
if(flag != 0)
{
  printf("Item found at location %d" ,flag);
}
else
  printf("Item not found");

}
}

```

Output:

```

Enter Item ?
3
Item found at location 2
Enter Item ?
2
Item not found
Binary Search

```

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

7.2.2 Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Features of Binary Search

It is great to search through large sorted arrays.

It has a time complexity of $O(\log n)$ which is a very good time complexity. We will discuss this in details in the Binary Search tutorial.

It has a simple implementation.

Binary search algorithm is given below.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET BEG = lower_bound

END = upper_bound, POS = - 1

Step 2: Repeat Steps 3 and 4 while BEG <= END

Step 3: SET MID = (BEG + END)/2

Step 4: IF A[MID] = VAL

SET POS = MID

PRINT POS

Go to Step 6

ELSE IF A[MID] > VAL

SET END = MID - 1

ELSE

SET BEG = MID + 1

[END OF IF]

[END OF LOOP]

Step 5: IF POS = -1

PRINT "VALUE IS NOT PRESENT IN THE ARRAY"

[END OF IF]

Step 6: EXIT

Complexity

SN	Performance	Complexity
1	Worst case	$O(\log n)$
2	Best case	$O(1)$
3	Average Case	$O(\log n)$
4	Worst case space	$O(1)$

Example

Let us consider an array $\text{arr} = \{1, 5, 7, 8, 13, 19, 20, 23, 29\}$. Find the location of the item 23 in the array.

In 1st step :

BEG = 0

END = 8

MID = 4

$a[\text{mid}] = a[4] = 13 < 23$, therefore

in Second step:

Beg = mid + 1 = 5

End = 8

mid = 13/2 = 6

$a[\text{mid}] = a[6] = 20 < 23$, therefore;

in third step:

beg = mid + 1 = 7

End = 8

mid = 15/2 = 7

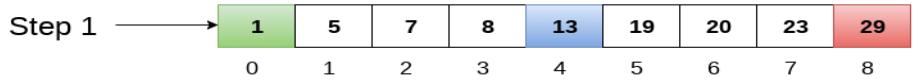
$a[\text{mid}] = a[7]$

$a[7] = 23 = \text{item}$;

therefore, set location = mid;

The location of the item will be 7.

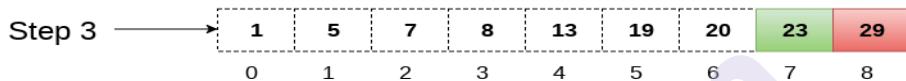
Item to be searched = 23



a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6



a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7



a [mid] = 23
23 = 23
loc = mid

Return location 7

```
int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};  
int item=45  
int location = -1;  
System.out.println("Enter the item which you want to search");  
location = binarySearch(arr,0,9,item);  
if(location != -1)  
printf("the location of the item is %d",location);  
else  
    printf("Item not found");  
}  
static 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);
    }

}

}

Output:
```

Enter the item which you want to search

45

the location of the item is 5

7.3 SUMMARY:

- The sorting algorithms are used to arrange data in order and we use array as a data storage structure.
- All the algorithms in this chapter execute in $O(N^2)$ time. Nevertheless, some can be substantially faster than others.
- An invariant is a condition that remains unchanged while an algorithm runs.
- The bubble sort is the least efficient, but the simplest, sort.
- The insertion sort is the most commonly used of the $O(N^2)$ sorts described in this chapter.
- None of the sorts in this chapter require more than a single temporary variable, in addition to the original array.
- Linear search is a very simple search algorithm which is a type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
- Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

7.4 LIST OF REFERENCES

- Data Structures using C Balguruswamy McGraw Hill Education, New Delhi 2013, ISBN: 978-1259029547
- Data Structures using C ISRD Group McGraw Hill Education, New Delhi 2013, ISBN: 978-12590006401
- <http://nptel.ac.in/courses/106102064/1>
- www.oopweb.com/algorithms
- www.studytonight.com/data-structures/
- <http://www.academictutorial.com/data-structures>

7.5 MODEL QUESTIONS:

1. Arrange the following numbers in ascending as well as descending order with the help of the specified algorithms and show each pass in detail.

Numbers: - 123,456,200,23,56,12

Algorithms: -

- Bubble sort
 - Selection
 - Merge
 - Insertion
2. Write the short note on the above (ques 2) sorting algorithms with example.
 3. Write down the complexity for any 2 algorithms from above (ques 2).
 4. Write the advantages and disadvantages of any 3 algorithms from above (ques 2).
 5. Write down the algorithm for any 3 algorithms from above (ques 2).
 6. Explain Linear and Binary Search.
 7. Trace 2,45,56,67,78,99,100 using Linear and Binary Search



TREES

Unit Structure :

- 8.0 Objectives:
- 8.1 Introduction to Tree
 - 8.1.1 Binary Tree
 - 8.1.2 Properties of Binary Tree
 - 8.1.3 Memory Representation of Binary Tree
 - 8.1.4 Operations Performed on Binary Tree
 - 8.1.5 Reconstruction of Binary Tree from its Traversals
- 8.2 Huffman Algorithm
- 8.3 Binary Search Tree
 - 8.3.1 Operations on Binary Search Tree
- 8.4 Heap
 - 8.4.1 Memory Representation of Heap
 - 8.4.2 Operation on Heap, Heap Sort
- 8.5 Summary
- 8.6 List of References
- 8.7 Model Questions

8.0 OBJECTIVES

This chapter would make you understand the following concepts:

- Introduction to Trees.
- Binary Trees, memory representations, operations and its properties.
- Two types of traversals.
- Huffman algorithm, its benefits and its implementation.
- Binary Search Tree and Operations on Binary Search Tree.
- Heap, Memory Representation of Heap and Operations on Heap.
- Heap Sort and its complexity.

8.1 INTRODUCTION TO TREE

In all the previous topics we have seen data structures like stack, queue and linked list which follows a specific order and are called as linear data structures. Now we are going to understand non-linear data structures do not form a sequence and thus called as non-linear data structures. This hierarchical structure which has relationship between data

elements is called as tree. Tree Data structures not only stores data in hierarchical manner but have number of applications, ease and efficiency of various data structure operations like searching, sorting ,etc compare to linear data structure.

Tree Data Structure:

Tree is a collection of finite set “T” of one or more nodes such that there is a node assigned as the root of the tree and other nodes are divided into $n \geq 0$ disjoint sets T_1, T_2, \dots, T_n and each of this node is tree in turn. Nodes in root are called as sub-tree or child of the root.

Definition of Tree:

A tree is a non-linear data structure used to represent the hierarchical structure of one or more elements known as nodes.

Tree holds values and has either zero or more than zero references which are referring to other nodes known as child nodes. A tree can have zero or more child nodes, which is at one level below it and each child node can have only one parent node which above it. The node at the top of the tree is known as root of the tree and the node at the lowest level is known as leaf node.

Ex:- a tree of an organization

Tree with one or more child nodes are called as internal nodes and the node without child is called as external nodes.

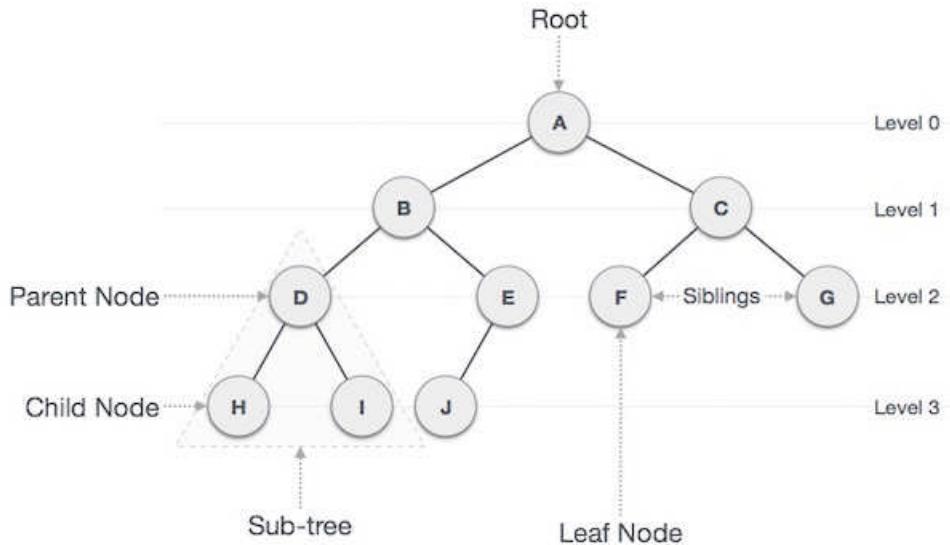
Applications of Trees:

- Hierarchical Management:- a tree like structure helps us to understand the protocol easily. Example :-Board of management.
- Effortlessness searching:- as data is in sorted manner so searching is easier compare to other linear data structures. Example:- searching a file in linux .
- Easy Manipulations: as data is two linear so can be used to process .Example :- sorting according to date_of_joining.

8.1.1 Binary Trees:

A binary tree has a special requirement that each node can have a maximum of two children.

One of the advantage of using binary tree over both static linear data structure i.e ordered array and a dynamic linear data structure i.e linked list will have searching faster.



8.1.2 Properties of Binary Tree

Following are the important terms with respect to tree.

1. Root – The mother node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
2. Path – Path refers to the sequence of nodes along the edges of a tree.
3. Node – Every individual element in tree is called as node. Every information is stored in the node with the value and link to other nodes.
4. Parent node – Any node excluding the root node has one edge upward to a node called parent node.
5. Child node – The node below a given node connected by its edge downward is called its child node.
6. Siblings – Nodes which belongs to same node is called as siblings of each other.
7. Leaf – The node which does not have any child node is called the leaf node.
8. Subtree – Subtree denotes the descendants of a node.
9. Visiting – Visiting means checking the value of a node when control is on the node.
10. Traversing – Traversing means passing through nodes in a specific order.
11. Levels – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
12. Keys – Key represents a value of a node based on which a search operation is to be carried out for a node.

13. Predecessor : Every node in binary tree, except the root has unique parent called predecessor of
14. Descendents : The descendents of a node are all those nodes which are reachable from that node Eg E,F and H are descendents in of B in fig:5.1
15. Ancestors: The ancestors of a nodes are all the node along the path from the root to that node B and A are ancestor of E in fig:5.1
16. In-degree: The in-degree of a node is the total number of edges coming to the node.
17. Out degree: The out degree of a node is the total number of edges going outside from the node
18. Level: Level is a rank of a tree order. The whole tree structure is leveled. The level of root node is always at 0.the immediate children of root are at level 1 and their immediate children are at level 2 and so on as shown in fig.5.1.
19. Depth of Tree :the depth of a tree is the longest path from the root to leaf node. In a tree data structure ,the total number of edges from node to a particular node is called as depth of that node.
20. Height :The highest number of nodes that is possible in a way starting from the first node {root} to a leaf node is called the height of a tree.

8.1.3Memory Representation of Binary Tree

- In this representation we use singly linked list. In this representation each node requires three fields, (see, Fig.)
- 1) One for the link of the left child,
 - 2) Second field for representing the information associated with the node, and information associated with the node, and
 - 3) The Third is used to represent the link of the right child.

When a node has no child then the corresponding pointer fields are null. The left and right field of a node is pointer to left and the right child of that node.

- Example : Fig 5.21 shows an example of linked representation of tree.
- Following is the code to declarant a tree data structure in java. Each node of binary tree,(as the root of some sub-tree)has both left and right sub tree, which we can access through pointer by declaring as follows:

```
Class Node
{
    intdata;
    Node left, right
}
```

8.1.4 Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree but few are the following:-

1. Insertion Operation
2. Deletion Operation

1. Insertion Operation-

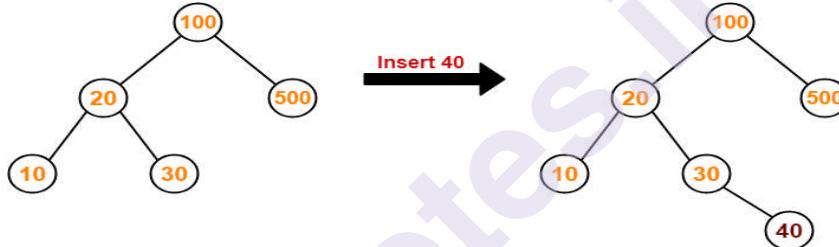
The insertion of a new key always considers the child of some leaf node.

For finding out the suitable leaf node,

- Search the key to be inserted from the root node till some leaf node is reached.
- Once a leaf node is reached, insert the key as child of that leaf node.

Example-

Consider the following example where key = 40 is inserted in the given BST-



We start from 40 to the root node 100.

- As $40 < 100$, so we search in 100's left subtree.
- As $40 > 20$, so we search in 20's right subtree.
- As $40 > 30$, so we add 40 to 30's right subtree.

2. Deletion Operation-

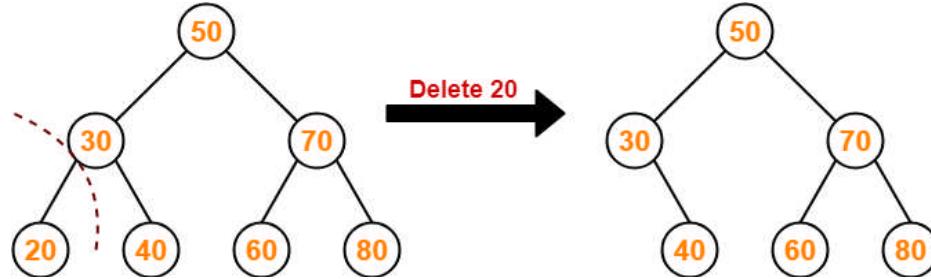
When it comes to deleting a node from the binary search tree, following three cases are possible-

Case-01: Deletion Of A Node Having No Child (Leaf Node)-

Just remove / disconnect the leaf node that is to be deleted from the tree.

Example-

Consider the following example where node with value = 20 is deleted from the BST-

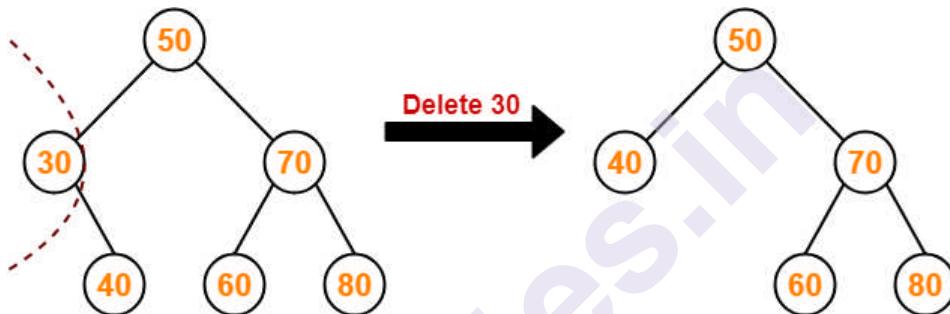


Case-02: Deletion Of A Node Having Only One Child-

Just make the child of the deleting node, the child of its grandparent.

Example-

Consider the following example where node with value = 30 is deleted from the BST-



Case-02: Deletion Of A Node Having Two Children-

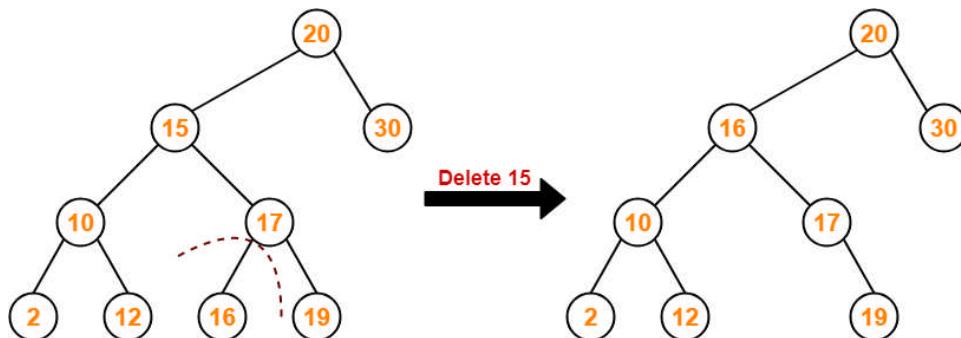
A node with two children may be deleted from the BST in the following two ways-

Method:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as inorder successor.
- Replace the deleting element with its inorder successor.

Example-

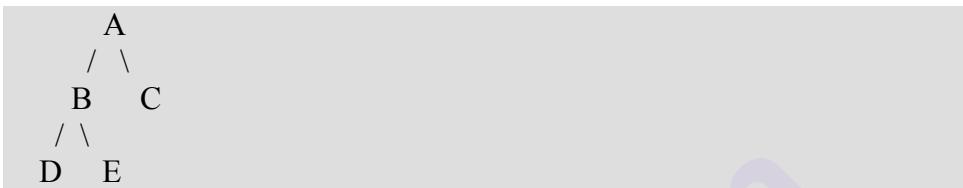
Consider the following example where node with value = 15 is deleted from the BST-



8.1.5 Reconstruction of Binary Tree from its Traversals:

Algorithm for Inorder traversal:

- 1) Create stack S.
- 2) Set current node as root
- 3) Push the current node to S and set current = current->left until current = NULL
- 4) If current = NULL and stack is not empty then
 - a) Pop the top item.
 - b) Print the popped item, set current = popped-item->right
 - c) Go to step 3.
- 5) If current is NULL and stack is empty then done.



Algorithm for preorder traversal:

- 1) Create an stack nodeStack and push root node to stack.
- 2) Do following while nodeStack is not empty.
 - I. Pop an item from stack and print it.
 - II. Push right child of popped item to stack
 - III. Push left child of popped item to stack

Algorithm for posttraversal

1. Create a stack
2. Do following while root != NULL
 - a) Push root's right child and then root to stack.
 - b) Set root as root's left child.
3. Pop an item from stack and set it as root.
 - a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.
 - b) Else print root's data and set root as NULL.
4. Repeat steps 2 and 3 while stack is not empty.

8.2 HUFFMAN ALGORITHM

Huffman coding is a lossless data compression algorithm. In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. Most frequent characters have the smallest codes and longer codes for least frequent characters.

There are mainly two parts. First one to create a Huffman tree, and another one to traverse the tree to find codes.

For an example, consider some strings “YYYZXXYYX”, the frequency of character Y is larger than X and the character Z has the least frequency. So the length of the code for Y is smaller than X, and code for X will be smaller than Z.

Complexity for assigning the code for each character according to their frequency is $O(n \log n)$

Input and Output

Input:

A string with different characters, say
“ACCEBFFFFAAXXBLKE”

Output:

Code for different characters:

Data: K, Frequency: 1, Code: 0000
Data: L, Frequency: 1, Code: 0001
Data: E, Frequency: 2, Code: 001
Data: F, Frequency: 4, Code: 01
Data: B, Frequency: 2, Code: 100
Data: C, Frequency: 2, Code: 101
Data: X, Frequency: 2, Code: 110
Data: A, Frequency: 3, Code: 111

Algorithm

Huffman Coding (string)

Input: A string with different characters.

Output: The codes for each individual characters.

Begin

 define a node with character, frequency, left and right child of the node for Huffman tree.

 create a list ‘freq’ to store frequency of each character, initially, all are 0
 for each character c in the string do

 increase the frequency for character ch in freq list.

 done

 for all type of character ch do

 if the frequency of ch is non zero then

 add ch and its frequency as a node of priority queue Q.

 done

 while Q is not empty do

 remove item from Q and assign it to left child of node

 remove item from Q and assign to the right child of node

 traverse the node to find the assigned code

```

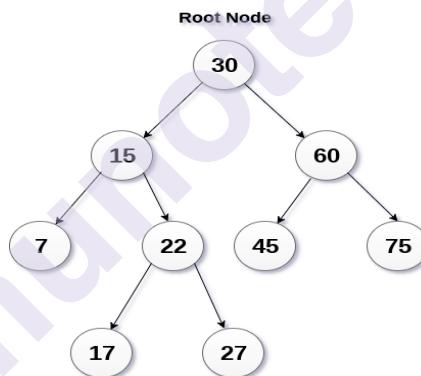
done
End
traverseNode(n: node, code)
Input: The node n of the Huffman tree, and the code assigned from the
previous call

Output: Code assigned with each character
if a left child of node n ≠φ then
    traverseNode(leftChild(n), code+'0') //traverse through the left child
    traverseNode(rightChild(n), code+'1') //traverse through the right child
else
    display the character and data of current node.

```

8.3 BINARY SEARCH TREE

Binary Search tree can be defined as a class of binary trees, in which the nodes are organized in a specific order. This is also called ordered binary 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. This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

Benefits of using binary search tree

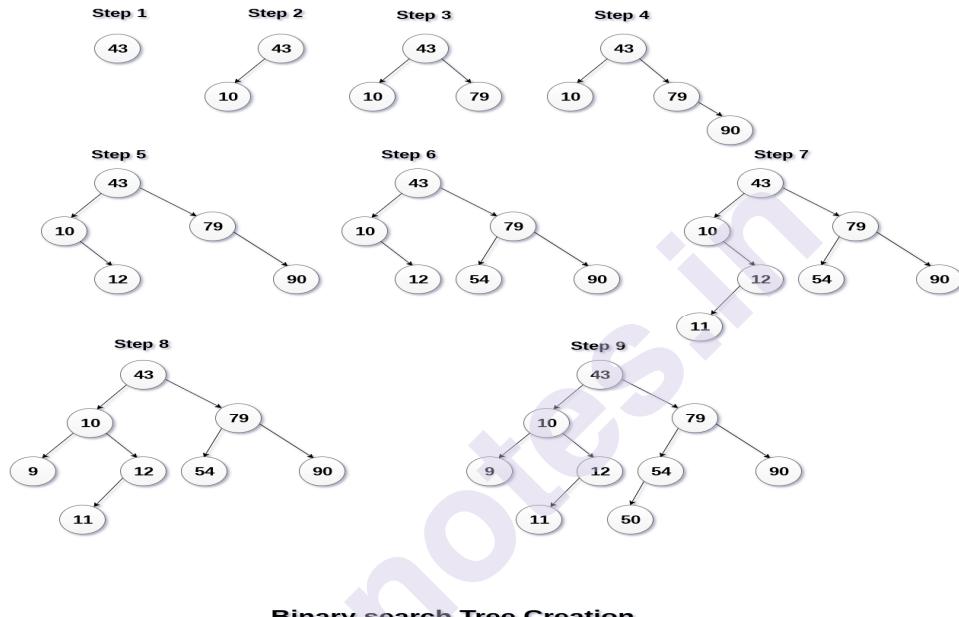
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the preferred element.
2. The binary search tree is considered as well-organized data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $O(\log_2 n)$ time. In worst case, the time it takes to search an element is $O(n)$.

- 3. It also speed up the insertion and deletion operations as compare to that in array and linked list.
- Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown below



8.3.1 Operations on Binary search trees (insertion and deletion)

- A Binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (data/info) and satisfies the following condition;
 1. The key in the left child of a node, is less than the key in its parent node.
 2. The key in the right child of a node, is greater than the key in its parent node.
 3. The left and right sub-trees of node are again binary search trees.
- The definition ensures that no two entries in a binary search tree can have equal keys, following is the example of binary search trees where all keys in left subtree is less than root and all keys in right subtree are greater than root.
- In order to create a binary search tree on given data use following steps:
 - Step1: Read a data in item variable.
 - Step2: Allocate memory for a new node and store the address in pointer root.
 - Step3: Store the data x in the node root.

Step4:If(data<root.data).

Step5:Recursively create the left subtree of root and make it the left child of root.

Step6:If(data>root.data).

Step7:Recursively create the right subtree of root and make it the right child of root.

- Below is a recursive algorithm to perform above mentioned steps.

Algorithm:create(root,data)

Root is initially NULL, data is key which you want to insert

1. If root is NULL
2. create a node with data
3. return
4. else
5. If data is greater than root.data
6. root.left=create(root.left,data)
7. else
8. Root,right=create(root.left,data)

Example:

- Let create a binary search tree for sequence:20,17,29,22,45,9,19

- Step1:The first item is 20 and this is the root node, so begin the tree.

- Step2:Sequence 20,17,29,22,45,9,19.

This is a binary search tree,so there are two child nodes available, the left and the right.The next number is 17,the rule is applied(left is less than parent node) and so it has to be the left node,like this,

- Step3:Sequence 20,17,29,22,45,9,19.

This next number is 29, this is higher than the root node so it goes to the RIGHT sub-tree which happens to be empty at this stage,so the tree now looks like this,

- Step4:Sequence 20,17,29,22,45,9,19.

The next number is 22.This is more than the root and so need to be on the RIGHT sub-tree.The first node is already occupied.so the rule is applied again to that node,22 comes before 29 and so it needs to be on the LEFT sub-tree of that node like this,

- Step5:Sequence 20,17,29,22,45,9,19.

The next number is 45, this is more than the root and more than the first right node,so it is placed on the right side of the tree like this,

- Step6:Sequence 20,17,29,22,45,9,19.

The next number is 9 which is less than the root, the first left node is occupied and 9 is less than that node too,so it is placed on the left sub-tree, like this,

- Step7:Sequence 20,17,29,22,45,9,19.

The next number is 19, which is less than the root, so it will need to be in the left sub-tree. It is greater than the occupied 17 node and so it is placed in the right sub-tree, like this.

8.4 HEAP

A binary heap is a heap data structure that takes the form of a binary tree. Binary heaps are a common way of applying priority queues. The binary heap was announced by J. W. J. Williams in 1964.

A binary heap is defined as a binary tree with two constraints:-

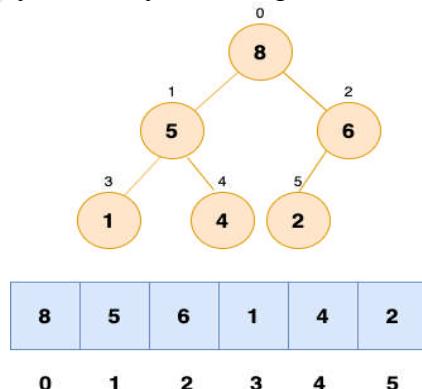
Form property: a binary heap is a complete binary tree and all levels of the tree, except possibly the last one are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

Heap property: the key stored in each node is either greater than or equal to or less than or equal to the keys in the node's children, according to some total order.

Max-heaps are the parent key which are greater than or equal to (\geq) the child keys and those where it is less than or equal to are called min-heaps. Effective algorithms are known for the two operations needed to implement a priority queue on a binary heap: inserting an element, and removing the smallest or largest element from a min-heap or max-heap, respectively. Binary heaps are also commonly employed in the heapsort sorting algorithm, which is an in-place algorithm because binary heaps can be implemented as an implicit data structure, storing keys in an array and using their relative positions within that array to represent child-parent relationships.

8.4.1 Memory Representation of Heap

Binary Heap is a Complete Binary Tree, it can be easily represented as an array and array-based representation is space-efficient.



8.4.2 Heap operations

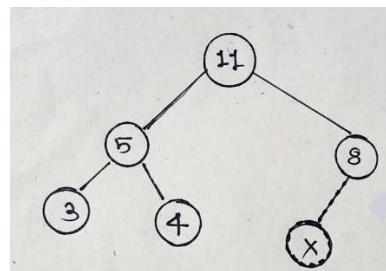
Both the insert and remove operations modify the heap to adapt to the shape property first, by adding or removing from the end of the heap.

Then the heap property is restored by traversing up or down the heap. Both operations have complexity of an $O(\log n)$ time.

Insert

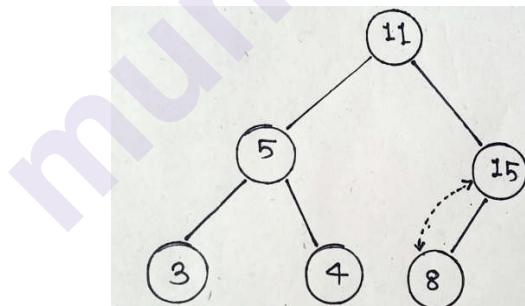
To add an element to a heap we must perform an up-heap operation (also known as bubble-up, percolate-up, sift-up, trickle-up, swim-up, heapify-up, or cascade-up), by following this algorithm:

- Add the element to the bottom level of the heap at the most left.
- Compare the added element with its parent; if they are in the correct order, stop.
- If not, swap the element with its parent and return to the previous step.



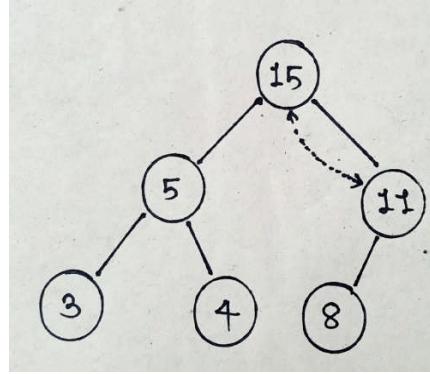
The number of operations required depends only on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a worst-case time complexity of $O(\log n)$ but an average-case complexity of $O(1)$.

As an example of binary heap insertion, say we have a max-heap



and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since $15 > 8$, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:

However the heap property is still violated since $15 > 11$, so we need to swap again:



which is a valid max-heap. There is no need to check the left child after this final step: at the start, the max-heap was valid, meaning $11 > 5$; if $15 > 11$, and $11 > 5$, then $15 > 5$, because of the transitive relation.

Delete

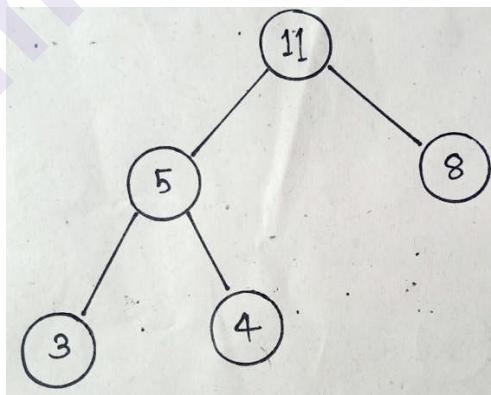
The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called down-heap (also known as bubble-down, percolate-down, sift-down, sink-down, trickle down, heapify-down, cascade-down, and extract-min/max).

Replace the root of the heap with the last element on the last level.

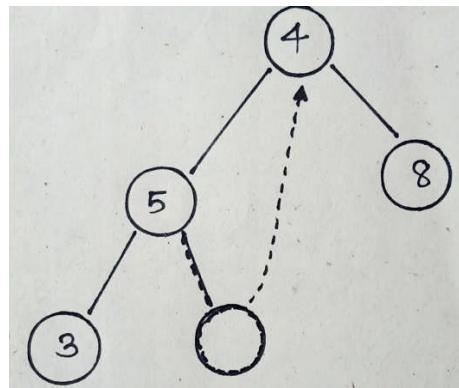
Compare the new root with its children; if they are in the correct order, stop.

If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

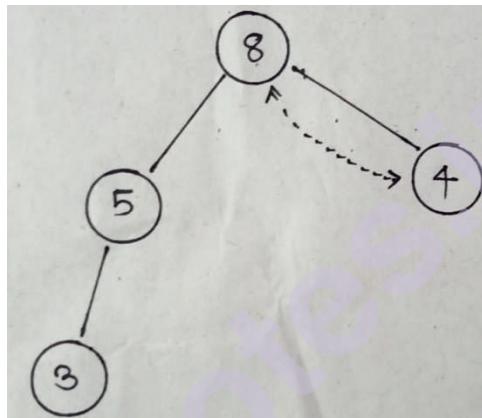
So, if we have the same max-heap as before



We remove the 11 and replace it with the 4.



Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:



The downward-moving node is swapped with the larger of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position. This functionality is achieved by the Max-Heapify function as defined below in pseudocode for an array-backed heap A of length `heap_length[A]`. Note that "A" is indexed starting at 1.

Max-Heapify (A, i):

```

left ←  $2 \times i$  // ← means "assignment"
right ←  $2 \times i + 1$ 
largest ← i

```

```

if left ≤ heap_length[A] and A[left] > A[largest] then:
    largest ← left

```

```

if right ≤ heap_length[A] and A[right] > A[largest] then:
    largest ← right

```

```

if largest ≠ i then:
    swap A[i] and A[largest]
    Max-Heapify (A, largest)

```

Heap Sort

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- o Build a heap H, using the elements of ARR.
- o Repeatedly delete the root element of the heap formed in phase 1.

Complexity

Complexity	Best Case	Average Case	Worst case
Time Complexity	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Space Complexity		$O(1)$	

Algorithm

HEAP_SORT(ARR, N)

- o **Step 1:** [Build Heap H]
Repeat for i=0 to N-1
 CALL INSERT_HEAP(ARR, N, ARR[i])
 [END OF LOOP]
- o **Step 2:** Repeatedly Delete the root element
Repeat while N > 0
 CALL Delete_Heap(ARR,N,VAL)
 SET N = N+1
 [END OF LOOP]
- o **Step 3:** END

```
1. #include<stdio.h>
2. int temp;
3. void heapify(int arr[], int size, int i)
4. {
5.     int largest = i;
6.     int left = 2*i + 1;
7.     int right = 2*i + 2;
8.     if(left < size && arr[left] > arr[largest])
9.         largest = left;
10.    if(right < size && arr[right] > arr[largest])
11.        largest = right;
12.    if(largest != i)
13.    {
14.        temp = arr[i];
15.        arr[i]= arr[largest];
16.        arr[largest] = temp;
17.        heapify(arr, size, largest);
18.    }
19. }
20. void heapSort(int arr[], int size)
```

```

21. {
22.     int i;
23.     for (i = size / 2 - 1; i >= 0; i--)
24.         heapify(arr, size, i);
25.     for (i=size-1; i>=0; i--)
26.     {
27.         temp = arr[0];
28.         arr[0]= arr[i];
29.         arr[i] = temp;
30.         heapify(arr, i, 0);
31.     }
32. }
33. void main()
34. {
35.     int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};
36.     int i;
37.     int size = sizeof(arr)/sizeof(arr[0]);
38.     heapSort(arr, size);
39.     printf("printing sorted elements\n");
40.     for (i=0; i<size; ++i)
41.     printf("%d\n",arr[i]);
42. }
```

Output:printing sorted elements

```

1
1
2
2
2
3
4
10
23
100
```

8.5 SUMMARY:

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- A binary tree is a special type of tree in which every node or vertex has either no child node or one child node or two child nodes.
- Binary traversals are pre-order,inorder and postorder.
- Stack is the approach to traverse tree without recursion
- A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

- Binary search tree (BST) is a special kind of binary tree where each node contains only larger values in its right subtree and Only smaller values in its left subtree.

8.6 LIST OF REFERENCES

- Data Structures using C Balguruswari JL_ McGraw Hill Education, New Delhi 2013, ISBN: 978-1259029547
- Data Structures with C (S1E) (Schaums Outline Series. Lipschutz McGraw Hill Education, New Delhi 2013, ISBN: 978-0070701984
- Practical C programming Steve Oualline O'Reilly Media
- <https://www.khanacademy.org/>
- https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm
- <http://www.javatpoint.com/tree>

8.7 MODEL QUESTIONS:

1. Define Binary Tree and state its properties.
2. Explain the memory representation of Binary Tree
3. Explain Traversals of Binary Tree.
4. State and explain Huffman algorithm.
5. Explain binary search Tree
6. Explain heap data structure.
7. Explain the memory representation of heap.
8. Create a binary Tree from 12,56,3,4,67,34,89,10.



ADVANCED TREE STRUCTURES

Unit Structure :

- 9.0 Objectives:
- 9.1 Advanced Tree Structures
 - 9.1.1 Red Black Tree
 - 9.1.2 Operations Performed on Red Black Tree
- 9.2 AVL Tree
 - 9.2.1 Operations performed on AVL Tree
- 9.3 2-3 Tree
- 9.4 B-Tree
- 9.5 Summary
- 9.6 List of References
- 9.7 Model Questions

9.0 OBJECTIVES:

- Advanced Tree Structures and its types.
- AVL Tree and Operations performed on AVL Tree
- 2-3 Tree and its operations
- B-Tree Introduction

9.1 ADVANCED TREE STRUCTURES

Data Structures are used to store and manage data in an efficient and organised way for faster and easy access and modification of Data. Some of the basic data structures which we already have discussed are Arrays, LinkedList, Stacks, Queues etc. In this chapter we will discuss some of the complex and advanced Data Structures like Disjoint Sets, Red Black, 2-3 tree, Self-Balancing Trees, Segment Trees, Tries etc.

9.1.1 Red-Black tree

The Red-black tree is a self-balanced binary search tree in which each node contains one extra bit of information that denotes the color of the node. The color of the node could be either Red or Black, depending on the bit information stored by the node.

Properties

The following are the properties associated with a Red-Black tree:

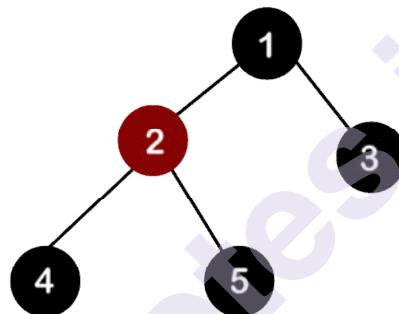
The root node of the tree should be Black.

A red node can have only black children. Or, we can say that two adjacent nodes cannot be Red in color.

If the node does not have a left or right child, then that node is said to be a leaf node. So, we put the left and right children below the leaf node known as nil

The black depth or black height of a leaf node can be defined as the number of black that we encounter when we move from the leaf node to the root node. One of the key properties of the Red-Black tree is that every leaf node should have the same black depth.

Let's understand this property through an example.



In the above tree, there are five nodes, in which one is a Red and the other four nodes are Black. There are three leaf nodes in the above tree. Now we calculate the black depth of each leaf node. As we can observe that the black depth of all the three leaf nodes is 2; therefore, it is a Red-Black tree.

If the tree does not obey any of the above three properties, then it is not a Red-Black tree.

Height of a red-black tree

Consider h as the height of the tree having n nodes. What would be the smallest value of n ? The value of n is the smallest when all the nodes are black. If the tree contains all the black nodes, then the tree would be a complete binary tree. If the height of a complete binary tree is h , then the number of nodes in a tree is:

$$n = 2^h - 1$$

What would be the largest value of n ?

The value of n is largest when every black node has two red children, but the red node cannot have red children, so that it will have black children. In this way, there are alternate layers of black and red children. So, if the

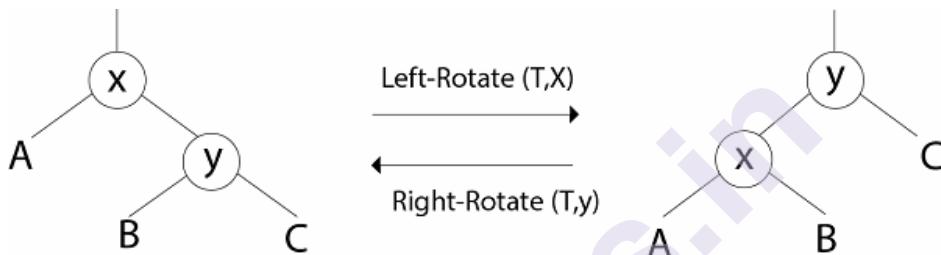
number of black layers is h , then the number of red layers is also h ; therefore, the tree's total height becomes $2h$. The total number of nodes is: $n = 2^h - 1$

9.1.2 Operations Performed on Red Black Tree

The search-tree operations TREE-INSERT and TREE-DELETE, when runs on a red-black tree with n keys, take $O(\log n)$ time. Because they customize the tree, the conclusion may violate the red-black properties. To restore these properties, we must change the color of some of the nodes in the tree and also change the pointer structure.

1. Rotation:

Restructuring operations on red-black trees can generally be expressed more clearly in details of the rotation operation.



Clearly, the order (Ax By C) is preserved by the rotation operation. Therefore, if we start with a BST and only restructure using rotation, then we will still have a BST i.e. rotation do not break the BST-Property.

LEFT ROTATE (T, x)

1. $y \leftarrow \text{right}[x]$
 1. $y \leftarrow \text{right}[x]$
 2. $\text{right}[x] \leftarrow \text{left}[y]$
 3. $p[\text{left}[y]] \leftarrow x$
 4. $p[y] \leftarrow p[x]$
 5. If $p[x] = \text{nil}[T]$
then $\text{root}[T] \leftarrow y$
else if $x = \text{left}[p[x]]$
then $\text{left}[p[x]] \leftarrow y$
else $\text{right}[p[x]] \leftarrow y$
 6. $\text{left}[y] \leftarrow x$.
 7. $p[x] \leftarrow y$.

2. Insertion:

- o Insert the new node the way it is done in Binary Search Trees.
- o Color the node red
- o If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can decision from a parent and a child both having a red color. This type of discrepancy is determined by the location of the node concerning grandparent, and the color of the sibling of the parent.

RB-INSERT (T, z)

1. $y \leftarrow \text{nil}$ [T]
2. $x \leftarrow \text{root}$ [T]
3. while $x \neq \text{NIL}$ [T]
 4. do $y \leftarrow x$
 5. if $\text{key}[z] < \text{key}[x]$
 6. then $x \leftarrow \text{left}[x]$
 7. else $x \leftarrow \text{right}[x]$
 8. $p[z] \leftarrow y$
 9. if $y = \text{nil}$ [T]
 10. then $\text{root}[T] \leftarrow z$
 11. else if $\text{key}[z] < \text{key}[y]$
 12. then $\text{left}[y] \leftarrow z$
 13. else $\text{right}[y] \leftarrow z$
 14. $\text{left}[z] \leftarrow \text{nil}$ [T]
 15. $\text{right}[z] \leftarrow \text{nil}$ [T]
 16. $\text{color}[z] \leftarrow \text{RED}$
 17. RB-INSERT-FIXUP (T, z)

After the insert new node, Coloring this new node into black may violate the black-height conditions and coloring this new node into red may violate coloring conditions i.e. root is black and red node has no red children. We know the black-height violations are hard. So we color the node red. After this, if there is any color violation, then we have to correct them by an RB-INSERT-FIXUP procedure.

RB-INSERT-FIXUP (T, z)

1. while $\text{color}[p[z]] = \text{RED}$
2. do if $p[z] = \text{left}[p[p[z]]]$
3. then $y \leftarrow \text{right}[p[p[z]]]$
4. If $\text{color}[y] = \text{RED}$
5. then $\text{color}[p[z]] \leftarrow \text{BLACK}$ //Case 1
6. $\text{color}[y] \leftarrow \text{BLACK}$ //Case 1
7. $\text{color}[p[z]] \leftarrow \text{RED}$ //Case 1
8. $z \leftarrow p[p[z]]$ //Case 1
9. else if $z = \text{right}[p[z]]$
10. then $z \leftarrow p[z]$ //Case 2
11. LEFT-ROTATE (T, z) //Case 2
12. $\text{color}[p[z]] \leftarrow \text{BLACK}$ //Case 3

13. $\text{color}[p[p[z]]] \leftarrow \text{RED}$ //Case 3
14. $\text{RIGHT-ROTATE}(T, p[p[z]])$ //Case 3
15. else (same as then clause)
With "right" and "left" exchanged
16. $\text{color}[\text{root}[T]] \leftarrow \text{BLACK}$

3. Deletion:

First, search for an element to be deleted

- o If the element to be deleted is in a node with only left child, swap this node with one containing the largest element in the left subtree. (This node has no right child).
- o If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right subtree (This node has no left child).
- o If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.
- o The item to be deleted is now having only a left child or only a right child. Replace this node with its sole child. This may violate red constraints or black constraint. Violation of red constraints can be easily fixed.
- o If the deleted node is black, the black constraint is violated. The elimination of a black node y causes any path that contained y to have one fewer black node.
- o Two cases arise:
 - o The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
 - o The replacing node is black.

The strategy RB-DELETE is a minor change of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotation to restore the red-black properties.

RB-DELETE (T, z)

1. if $\text{left}[z] = \text{nil}[T]$ or $\text{right}[z] = \text{nil}[T]$
2. then $y \leftarrow z$
3. else $y \leftarrow \text{TREE-SUCCESSOR}(z)$
4. if $\text{left}[y] \neq \text{nil}[T]$
5. then $x \leftarrow \text{left}[y]$
6. else $x \leftarrow \text{right}[y]$
7. $p[x] \leftarrow p[y]$

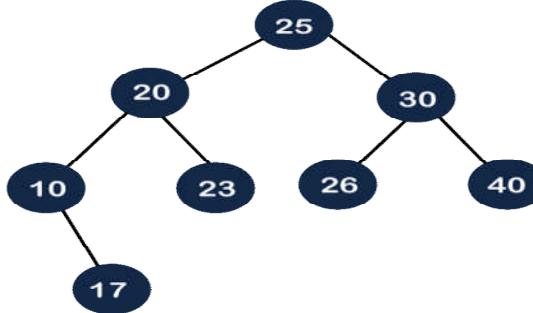
8. if $p[y] = \text{nil}$ [T]
9. then $\text{root}[T] \leftarrow x$
10. else if $y = \text{left}[p[y]]$
11. then $\text{left}[p[y]] \leftarrow x$
12. else $\text{right}[p[y]] \leftarrow x$
13. if $y \neq z$
14. then $\text{key}[z] \leftarrow \text{key}[y]$
15. copy y 's satellite data into z
16. if $\text{color}[y] = \text{BLACK}$
17. then RB-delete-FIXUP (T, x)
18. return y

RB-DELETE-FIXUP (T, x)

1. while $x \neq \text{root}[T]$ and $\text{color}[x] = \text{BLACK}$
2. do if $x = \text{left}[p[x]]$
3. then $w \leftarrow \text{right}[p[x]]$
4. if $\text{color}[w] = \text{RED}$
5. then $\text{color}[w] \leftarrow \text{BLACK}$ //Case 1
6. $\text{color}[p[x]] \leftarrow \text{RED}$ //Case 1
7. LEFT-ROTATE ($T, p[x]$) //Case 1
8. $w \leftarrow \text{right}[p[x]]$ //Case 1
9. If $\text{color}[\text{left}[w]] = \text{BLACK}$ and $\text{color}[\text{right}[w]] = \text{BLACK}$
10. then $\text{color}[w] \leftarrow \text{RED}$ //Case 2
11. $x \leftarrow p[x]$ //Case 2
12. else if $\text{color}[\text{right}[w]] = \text{BLACK}$
13. then $\text{color}[\text{left}[w]] \leftarrow \text{BLACK}$ //Case 3
14. $\text{color}[w] \leftarrow \text{RED}$ //Case 3
15. RIGHT-ROTATE (T, w) //Case 3
16. $w \leftarrow \text{right}[p[x]]$ //Case 3
17. $\text{color}[w] \leftarrow \text{color}[p[x]]$ //Case 4
18. $\text{color}[p[x]] \leftarrow \text{BLACK}$ //Case 4
19. $\text{color}[\text{right}[w]] \leftarrow \text{BLACK}$ //Case 4
20. LEFT-ROTATE ($T, p[x]$) //Case 4
21. $x \leftarrow \text{root}[T]$ //Case 4
22. else (same as then clause with "right" and "left" exchanged)
23. $\text{color}[x] \leftarrow \text{BLACK}$

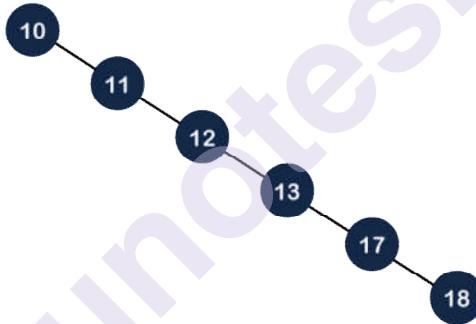
9.2 AVL TREE

An AVL tree is a self-balancing binary search tree if we observe the below case, which is a binary search tree and a balanced tree.



In the above tree, the worst-case time complexity for searching an element is $O(h)$, i.e., the height of the tree. In the above case, the number of comparisons made to search 17 element is 4, and the height of the tree is also 4.

If we consider the skewed binary tree, as shown below:



In the above right skewed tree, the number of comparisons made to search 17 element is 5, and the number of elements present in the tree is also 5. Therefore, we can say that if the tree is a skewed binary tree then the time complexity would be $O(n)$.

Now, we have to balance these skewed trees so that they will have the time complexity $O(h)$. There is a term known as a balance factor, which is used to self -balance the binary search tree. The balance factor can be computed as:

Balance factor = height of left subtree-height of right subtree

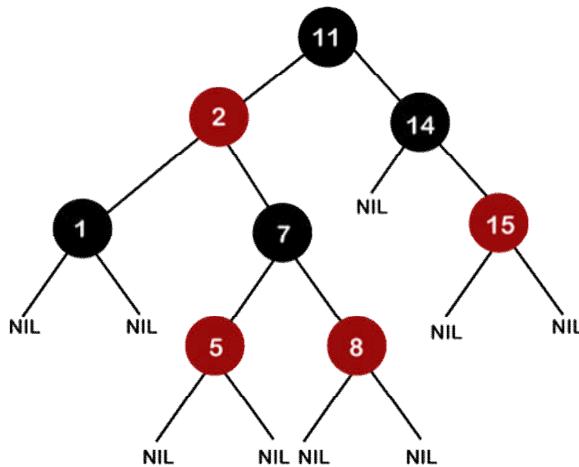
The value of the balance factor could be either 1, 0 or -1. If each node in the binary tree is having a value of either 1, 0, or -1, then that tree is said to be a balanced binary tree or AVL tree.

The tree is known as a perfectly balanced tree if the balance factor of each node is 0. The AVL tree is an almost balanced tree because each node in the tree has the value of balance factor either 1, 0 or -1.

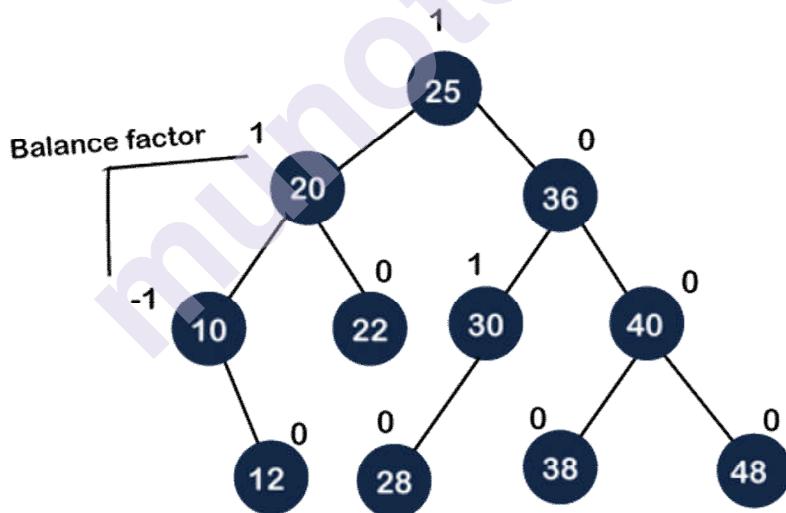
Rule of the AVL tree:

Every node should have the balance factor either as -1, 0 or 1.

Example



In the above figure, we need to check whether the tree is a Red-Black tree or not. In order to check this, first, we need to check whether the tree is a binary search tree or not. As we can observe in the above figure that it satisfies all the properties of the binary search tree; therefore, it is a binary search tree. Secondly, we have to verify whether it satisfies the above-said rules or not. The above tree satisfies all the above five rules; therefore, it concludes that the above tree is a Red-Black tree.



In the above figure, we need to check whether the tree is an AVL tree or not. As each node has a value of balance factor either as -1, 0, or 1, so it is an AVL tree.

9.2.1 Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with $O(\log n)$ time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

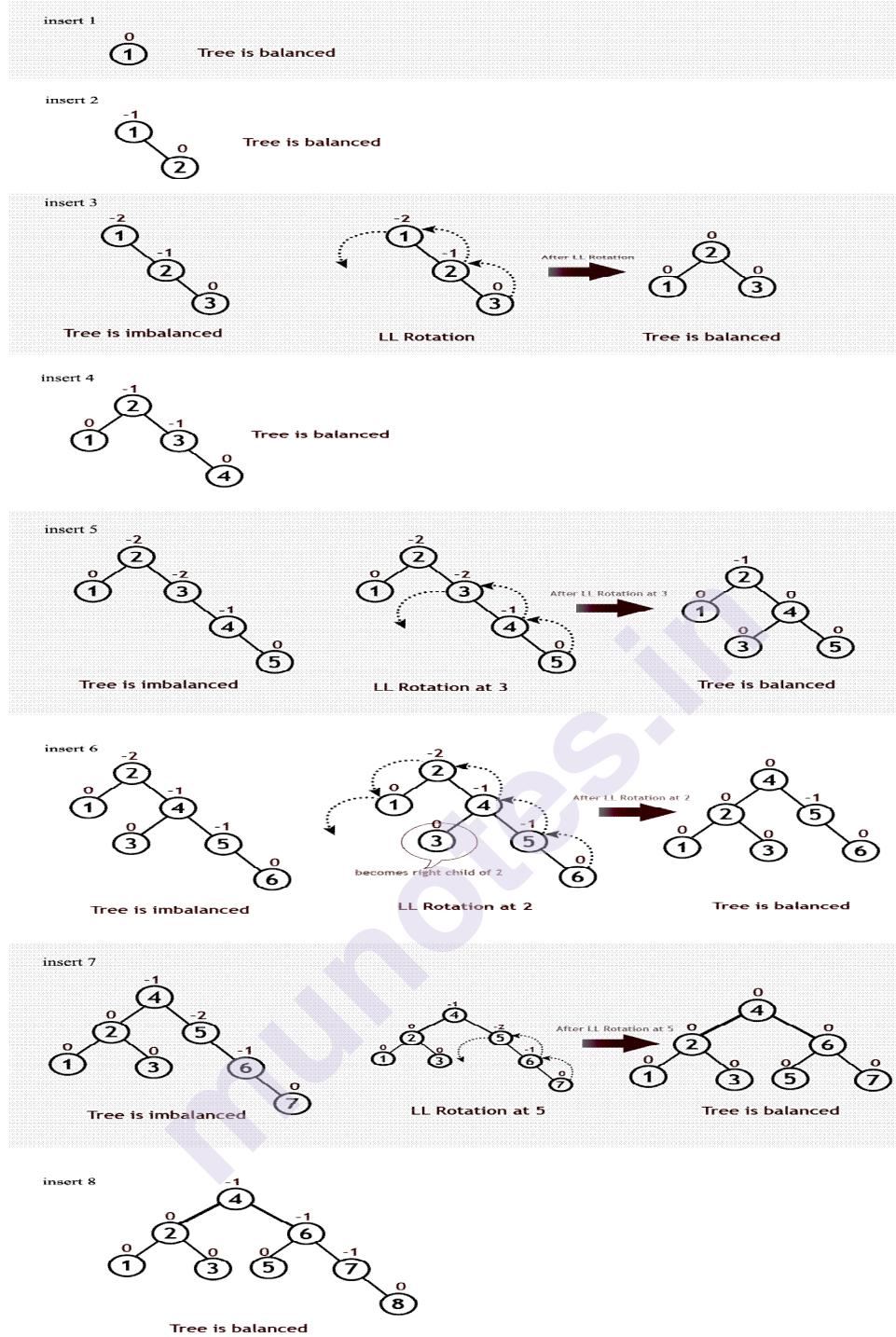
- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the value of root node in the tree.
- Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
- Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
- Step 5 - If search element is smaller, then continue the search process in left subtree.
- Step 6 - If search element is larger, then continue the search process in right subtree.
- Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with $O(\log n)$ time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the **Balance Factor** of every node.
- Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.



Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

Time complexity

In the Red-Black tree case, the time complexity for all the operations, i.e., insertion, deletion, and searching is O(logn).

In the case of AVL tree, the time complexity for all the operations, i.e., insertion, deletion, and searching is O(logn).

9.3 2-3 Tree

A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.

Here are the properties of a 2-3 tree:

- each node has either one value or two values a node with one value is either a leaf node or has exactly two children (non-null).
- Values in left subtree < value in node < values in right subtree a node with two values is either a leaf node or has exactly three children (non-null).
- Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
- all leaf nodes are at the same level of the tree

Insertion

The insertion algorithm into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

If the tree is empty, create a node and put value into the node

Otherwise find the leaf node where the value belongs.

If the leaf node has only one value, put the new value into the node

If the leaf node has more than two values, split the node and promote the median of the three values to parent.

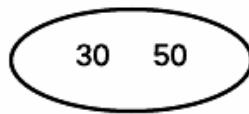
If the parent then has three values, continue to split and promote, forming a new root node if necessary

Example:

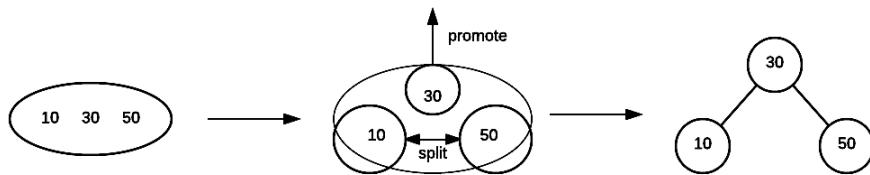
Insert 50



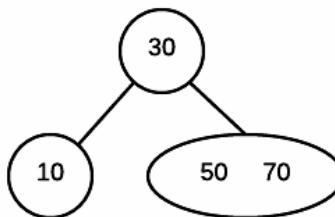
Insert 30



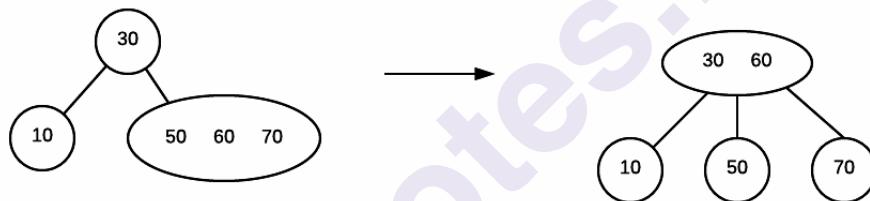
Insert 10



Insert 70



Insert 60



Time Complexity

The property of being perfectly balanced, enables the **2-3 Tree** operations of insert, delete and search to have a time complexity of $O(\log(n))$.

9.4 B Tree

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most $m-1$ keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

Every node in a B-Tree contains at most m children.

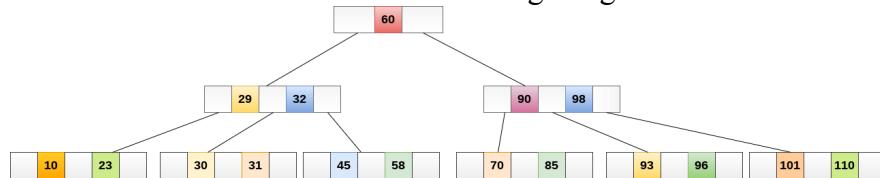
Every node in a B-Tree except the root node and the leaf node contain at least $m/2$ children.

The root nodes must have at least 2 nodes.

All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have $m/2$ number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

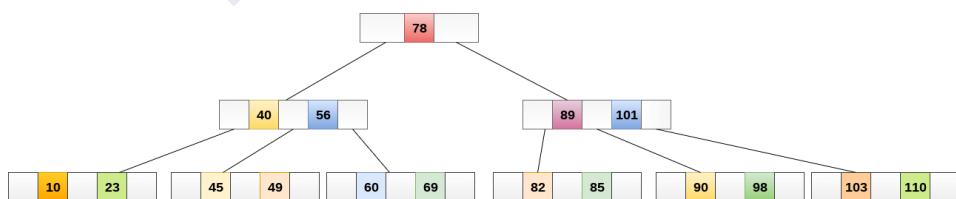
Compare item 49 with root node 78. since $49 < 78$ hence, move to its left sub-tree.

Since, $40 < 49 < 56$, traverse right sub-tree of 40.

$49 > 45$, move to right. Compare 49.

match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes $O(\log n)$ time to search any element in a B tree.



Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.

If the leaf node contain less than $m-1$ keys then insert the element in the increasing order.

Else, if the leaf node contains $m-1$ keys, then follow the following steps.

Insert the new element in the increasing order of elements.

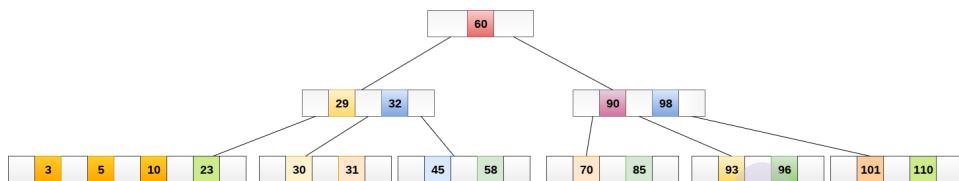
Split the node into the two nodes at the median.

Push the median element upto its parent node.

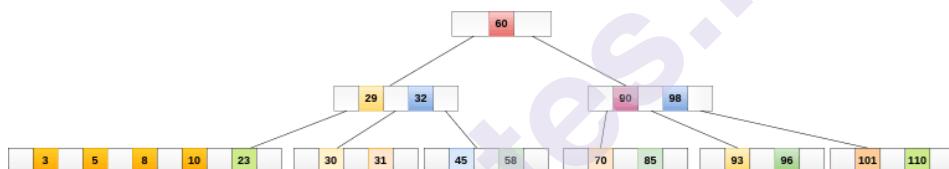
If the parent node also contain $m-1$ number of keys, then split it too by following the same steps.

Example:

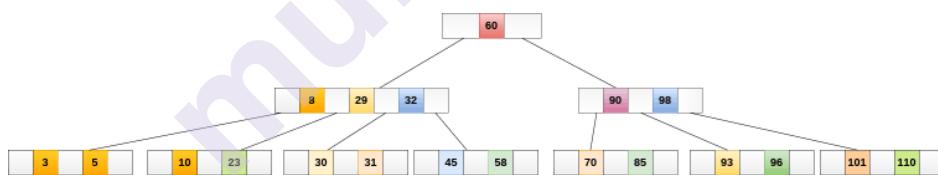
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than $(5 - 1 = 4)$ keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

Locate the leaf node.

If there are more than $m/2$ keys in the leaf node then delete the desired key from the node.

If the leaf node doesn't contain $m/2$ keys then complete the keys by taking the element from eight or left sibling.

If the left sibling contains more than $m/2$ elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.

If the right sibling contains more than $m/2$ elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

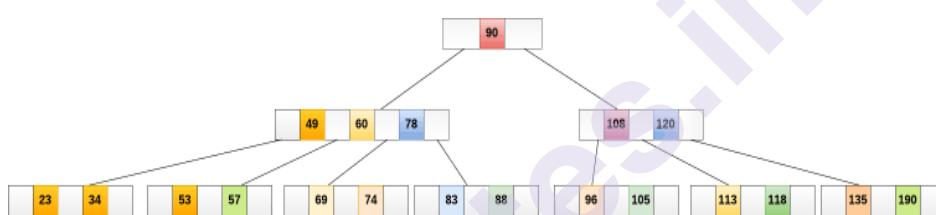
If neither of the sibling contain more than $m/2$ elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.

If parent is left with less than $m/2$ nodes then, apply the above process on the parent too.

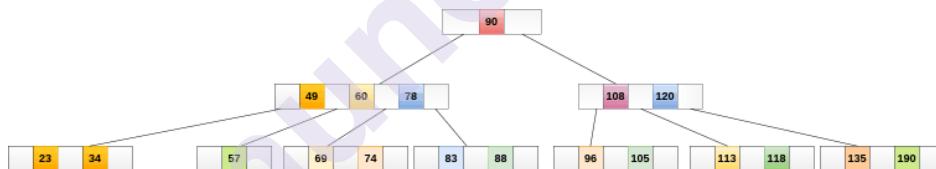
If the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

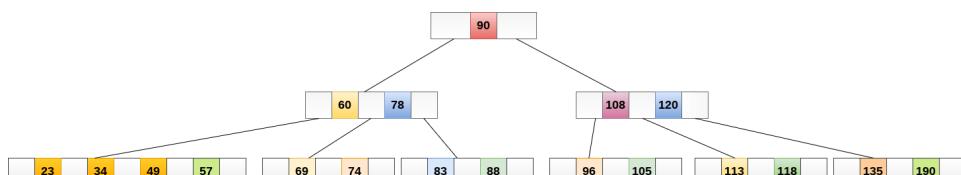


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

9.5 SUMMARY:

- Advanced Data Structures are used to store and manage data in an efficient and organized way for faster and easy access and modification of data like Disjoint Sets, Red Black, 2-3 tree, Self-Balancing Trees, Segment Trees, Tries etc.
- The Red-black tree is a self-balanced binary search tree in which each node contains one extra bit of information that denotes the color of the node. The color of the node could be either Red or Black, depending on the bit information stored by the node.
- A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.
- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.

9.6 LIST OF REFERENCES

- Data Structures using C Balguruswamy McGraw Hill Education, New Delhi 2013, ISBN: 978-1259029547
- Data Structures with C (S1E) (Schaums Outline Series. Lipschutz McGraw Hill Education, New Delhi 2013, ISBN: 978-0070701984
- https://www.tutorialspoint.com/advanced_data_structures/index.asp
- <https://www.javatpoint.com/tree>

9.7 MODEL QUESTIONS:

1. Explain Red Black Tree with an example.
2. State and explain various operations Performed on Red Black Tree.
3. Explain AVL Tree with an example.
4. State and explain various Operations performed on AVL Tree.
5. Explain 2-3 Tree with an example.
6. Explain B-Tree with an example.
7. Draw the complete binary tree of height 3 on the keys {1, 2, 3... 15}. Add the NIL leaves and color the nodes in three different ways such that the black heights of the resulting trees are: 2, 3 and 4.
8. Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.



Unit V

10

HASHING TECHNIQUES

Unit Structure:

- 10.0 Objective
- 10.1 Introduction
- 10.2 Why Hashing?
- 10.3 Hash function
 - 10.3.1 Need for a good hash function
 - 10.3.2 Different hashing functions
- 10.4 Hash Table
 - 10.4.1 A simple Hash Table in operation
- 10.5 Collision
- 10.6 Indexing
 - 10.6.1 Types of Indexes
- 10.7 Rehashing
 - 10.7.1 Why rehashing?
 - 10.7.2 How Rehashing is done?
- 10.8 Summary
- 10.9 Questions
- 10.10 Reference for further reading

10.0 OBJECTIVE

This chapter would make you understand the following concepts:

- Hash function
- Address calculation techniques
- Common hashing functions Collision resolution
- Linear probing, Quadratic
- Double hashing, Bucket hashing
- Deletion and rehashing

10.1 INTRODUCTION

We have already seen a number of different ways of storing items in a computer: arrays and variants thereof (e.g., sorted and unsorted arrays, heap trees), linked lists (e.g., queues, stacks), and trees (e.g., binary search trees, heap trees). We have also seen that these approaches can perform quite differently when it comes to the particular tasks we expect to carry out on the items, such as insertion, deletion and searching, and that the

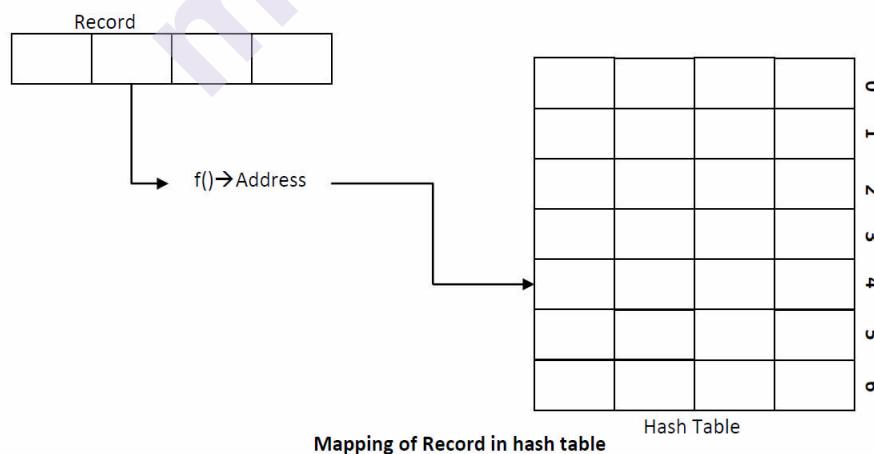
best way of storing data does not exist in general, but depends on the particular application. This chapter looks at another way of storing data, that is quite different from the ones we have seen so far. The idea is to simply put each item in an easily determined location, so we never need to search for it, and have no ordering to maintain when inserting or deleting items. This has impressive performance as far as time is concerned, but that advantage is payed for by needing more space (i.e., memory), as well as by being more complicated and therefore harder to describe and implement.

10.2 WHY HASHING?

Sequential search requires, on the average $O(n)$ comparisons to locate an element. So many comparisons are not desirable for a large database of elements. Binary search requires much fewer comparisons on the average $O(\log n)$ but there is an additional requirement that the data should be sorted. Even with best sorting algorithm, sorting of elements require $O(n \log n)$ comparisons. There is another widely used technique for storing of data called hashing. It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order ($O(1)$). In its worst case, hashing algorithm starts behaving like linear search.

Best case timing behaviour of searching using hashing = $O(1)$ Worst case timing Behavior of searching using hashing = $O(n)$

In hashing, there cord for a key value "key", is directly referred by calculating the address from the key value. Address or location of an element or record, x , is obtained by computing some arithmetic function $f.f(key)$ gives the address of x in the table.



Hashing is implemented in two steps:

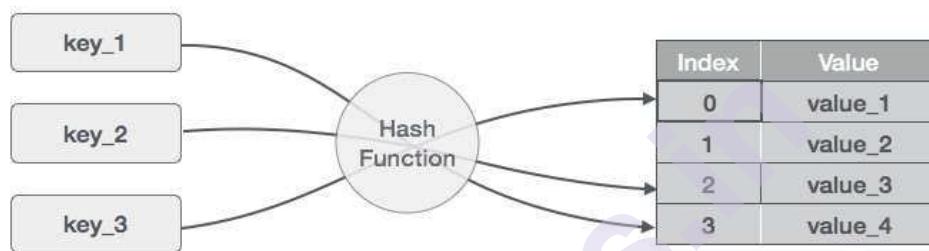
An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.

The element is stored in the hash table where it can be quickly retrieved using hashed key.

```
hash=hashfunc(key)
index = hash % array_size
```

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and $\text{array_size} - 1$) by using the modulo operator (%).

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key, value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)

Sr. No.	Key	Hash	Array Index
1	1	$1 \% 20 = 1$	1
2	2	$2 \% 20 = 2$	2
3	42	$42 \% 20 = 2$	2
4	4	$4 \% 20 = 4$	4
5	12	$12 \% 20 = 12$	12
6	14	$14 \% 20 = 14$	14
7	17	$17 \% 20 = 17$	17
8	13	$13 \% 20 = 13$	13
9	37	$37 \% 20 = 17$	17

10.3 HASH FUNCTION

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

Easy to compute: It should be easy to compute and must not become an algorithm in itself.

Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.

Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

10.3.1 Need for a good hash function

Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.

10.3.2 Different hashing functions

1. Division-Method
2. Mid square Methods
3. Folding Method
4. Digit Analysis
5. Length Dependent Method
6. Algebraic Coding
7. Multiplicative Hashing

1. Division-Method

In this method we use modular arithmetic system to divide the key value by some integer divisor (may be table size).

It gives us the location value, where the element can be placed.

We can write, $L = (K \bmod m) + 1$

where $L \Rightarrow$ location in table / file $K \Rightarrow$ key value

$m \Rightarrow$ table size/number of slots in file

Suppose, $k = 23, m = 10$ then

$L = (23 \bmod 10) + 1 = 3 + 1 = 4$, The key whose value is 23 is placed in 4th location.

2. Mid square Methods

In this case, we square the value of a key and take the number of digits required to form an address, from the middle position of squared value.

Suppose a key value is 16, then its square is 256. Now if we want address of two digits, then you select the address as 56 (i.e. two digits starting from middle of 256).

3. Folding Method

Most machines have a small number of primitive data types for which there are arithmetic instructions.

Frequently key to be used will not fit easily into one of these data types.

It is not possible to discard the portion of the key that does not fit into such an arithmetic data type.

The solution is to combine the various parts of the key in such a way that all parts of the key affect for final result such an operation is termed folding of the key.

That is the key is actually partitioned into number of parts, each part having the same length as that of the required address. Add the value of each parts, ignoring the final carry to get the required address.

This is done in two ways:

a. **Fold-shifting** : Here actual values of each parts of key are added.

Suppose, the key is : 12345678, and the required address is of two digits,

Then break the key into :12,34,56,78.

Add these, we get $12+34+56+78=180$, ignore first 1 we get 80 as location

- b. Fold-boundary :** Here there versed values of outer parts of key are added.

Suppose, the key is:12345678, and the required address is of two digits,

Then break the key into :21,34,56,87.

Add these, we get $21+34+56+87=198$, ignore first 1 we get 98 as location

4. Digit Analysis

This hashing function is a distribution-dependent. Here we make a statistical analysis of digits of the key, and select those digits (of fixed position) which occur quite frequently. Then reverse or shifts the digits to get the address. For example, if the key is : 9861234. If the statistical analysis has revealed the fact that the third and fifth position digits occur quite frequently, then we choose the digits in these positions from the key. So we get, 62. Reversing it we get 26 as the address.

5. Length Dependent Method

In this type of hashing function we use the length of the key along with some portion of the key j to produce the address, directly.

In the indirect method, the length of the key along with some portion of the key is used to obtain intermediate value.

6. Algebraic Coding

Here an bit key value is represented as a polynomial.

The divisor polynomial is then constructed based on the address range required.

The modular division of key-polynomial by divisor polynomial, to get the address-polynomial.

Let $f(x)$ = polynomial of n bit key = $a_1 + a_2 x + \dots + a_n x^{n-1}$
 $d(x)$ = divisor polynomial = $x^l + d_1 + d_2 x + \dots + d_l x^{l-1}$
then the required address polynomial will be $f(x) \bmod d(x)$

7. Multiplicative Hashing

This method is based on obtaining an address of a key, based on the multiplication value.

If k is the non-negative key, and a constant c , ($0 < c < 1$), compute $k \cdot c \bmod 1$, which is a fractional part of kc .

Multiply this fractional part by m to get the address ($0 \leq h(k) < m$)

10.4 HASH TABLE

The underlying idea of a hash table is very simple, and quite appealing: Assume that, given a key, there was a way of jumping straight to the entry for that key. Then we would never have to search at all, we could just go there! Of course, we still have to work out a way for that to be achieved. Assume that we have an array data to hold our entries. Now if we had a function $h(k)$ that maps each key k to the index (an integer) where the associated entry will be stored, then we could just look up $\text{data}[h(k)]$ to find the entry with the key k . It would be easiest if we could just make the data array big enough to hold all the keys that might appear. For example, if we knew that the keys were the numbers from 0 to 99, then we could just create an array of size 100 and store the entry with key 67 in $\text{data}[67]$, and so on. In this case, the function h would be the identity function $h(k) = k$. However, this idea is not very practical if we are dealing with a relatively small number of keys out of a huge collection of possible keys. For example, many American companies use their employees' 9-digit social security number as a key, even though they have nowhere near 10^9 employees. British National Insurance Numbers are even worse, because they are just as long and usually contain a mixture of letters and numbers. Clearly it would be very inefficient, if not impossible, to reserve space for all 10^9 social security numbers which might occur. Instead, we use a non-trivial function h , the so-called hash function, to map the space of possible keys to the set of indices of our array. For example, if we had to store entries about 500 employees, we might create an array with 1000 entries and use three digits from their social security number (maybe the first or last three) to determine the place in the array where the records for each particular employee should be stored.

10.4.1 A simple Hash Table in operation

Let us assume that we have a small data array we wish to use, of size 11, and that our set of possible keys is the set of 3-character strings, where each character is in the range from A to Z. Obviously, this example is designed to illustrate the principle typical real-world hash tables are usually very much bigger, involving arrays that may have a size of thousands, millions, or tens of millions, depending on the problem. We now have to define a hash function which maps each string to an integer in the range 0 to 10. Let us consider one of the many possibilities. We first map each string to a number as follows:

Each character is mapped to an integer from 0 to 25 using its place in the alphabet (A is the First letter, so it goes to 0, B the second so it goes to 1, and so on, with Z getting value 25). The string X1X2X3 therefore gives us three numbers from 0 to 25, say k_1 , k_2 , and k_3 . We can then map the whole string to the number calculated as

$$k = k_1 * 26^2 + k_2 * 26^1 + k_3 * 26^0 = k_1 * 26^2 + k_2 * 26 + k_3$$

That is, we think of the strings as coding numbers in base 26. Now it is quite easy to go from any number k (rather than a string) to a number

from 0 to 10. For example, we can take the remainder the number leaves when divided by 11. This is the C or Java modulus operation $k \% 11$. So our hash function is

$$h(X_1X_2X_3) = (k_1 * 26^2 + k_2 * 26 + k_3) \% 11 = k \% 11$$

This modulo operation, and modular arithmetic more generally, are widely used when constructing good hash functions. As a simple example of a hash table in operation, assume that we now wish to insert the following three-letter airport acronyms as keys (in this order) into our hash table: PHL, ORY, GCM, HKG, GLA, AKL, FRA, LAX, DCA. To make this easier, it is a good idea to start by listing the values the hash function takes for each of the keys

Code	PHL	ORY	GCM	HKG	GLA	AKL	FRA	LAX	DCA
$h(X_1X_2X_3)$	4	8	6	4	8	7	5	1	1

It is clear already that we will have hash collisions to deal with.

We naturally start off with an empty table of the required size, i.e. 11

--	--	--	--	--	--	--	--	--	--

Clearly we have to be able to tell whether a particular location in the array is still empty, or whether it has already been filled. We can assume that there is a unique key or entry (which is never associated with a record) which denotes that the position has not been filled yet. However, for clarity, this key will not appear in the pictures we use. Now we can begin inserting the keys in order. The number associated with the first item PHL is 4, so we place it at index 4, giving

				PHL					
--	--	--	--	-----	--	--	--	--	--

Next is ORY, which gives us the number 8, so we get:

				PHL				ORY	
--	--	--	--	-----	--	--	--	-----	--

Then we have GCM, with value 6, giving:

				PHL		GCM		ORY	
--	--	--	--	-----	--	-----	--	-----	--

Then HKG, which also has value 4, results in our first collision since the corresponding position has already been filled with PHL. Now we could, of course, try to deal with this by simply saying the table is full, but this gives such poor performance (due to the frequency with which collisions occur) that it is unacceptable.

10.5 COLLISION

This approach sounds like a good idea, but there is a pretty obvious problem with it: What happens if two employees happen to have the same three digits? This is called a collision between the two keys. Much of the remainder of this chapter will be spent on the various strategies for dealing with such collisions. First of all, of course, one should try to avoid collisions. If the keys that are likely to actually occur are not evenly spread throughout the space of all possible keys, particular attention should be

paid to choosing the hash function h in such a way that collisions among them are less likely to occur. If, for example, the first three digits of a social security number had geographical meaning, then employees are particularly likely to have the three digits signifying the region where the company resides, and so choosing the first three digits as a hash function might result in many collisions. However, that problem might easily be avoided by a more prudent choice, such as the last three digits.

There are several strategies for collision resolution. The most commonly used are:

1. **Buckets.**
2. **Separate chaining** - used with open hashing
3. **Open addressing** - used with closed hashing

1. **Buckets.** One obvious option is to reserve a two-dimensional array from the start. We can think of each column as a bucket in which we throw all the elements which give a particular result when the hash function is supplied, so the FIfth column contains all the keys for which the hash function evaluates to 4. Then we could put HKG into the slot 'beneath' PHL, and GLA in the one beneath ORY, and continue Filling the table in the order given until we reach:

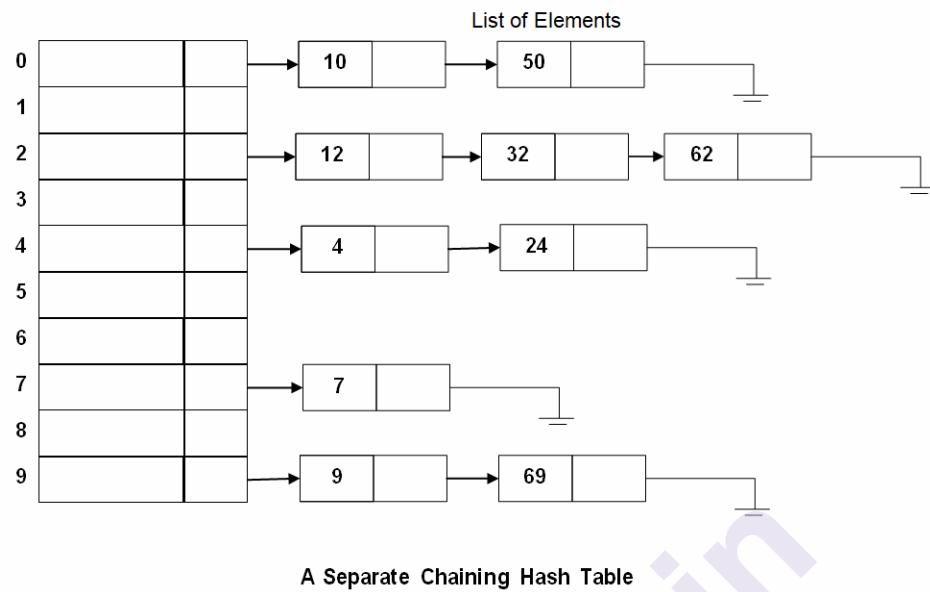
0	1	2	3	4	5	6	7	8	9	10
	LAX			PHL	FRA	GCM	AKL	ORY		
	DCA			HKG				GLA		

The disadvantage of this approach is that it has to reserve quite a bit more space than will be eventually required, since it must take into account the likely maximal number of collisions. Even while the table is still quite empty overall, collisions will become increasingly likely. Moreover, when searching for a particular key, it will be necessary to search the entire column associated with its expected position, at least until an empty slot is reached. If there is an order on the keys, they can be stored in ascending order, which means we can use the more efficient binary search rather than linear search, but the ordering will have an overhead of its own. The average complexity of searching for a particular item depends on how many entries in the array have been filled already. This approach turns out to be slower than the other techniques we shall consider, so we shall not spend any more time on it, apart from noting that it does prove useful when the entries are held in slow external storage.

2. Separate chaining

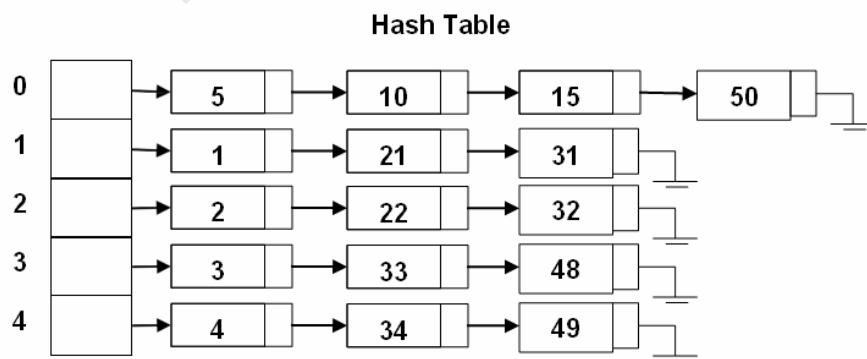
In this strategy, a separate list of all elements mapped to the same value is maintained. Separate chaining is based on collision avoidance. If memory space is tight, separate chaining should be avoided. Additional memory space for links is wasted in storing address of linked elements. Hashing function should ensure even distribution of elements among

buckets; otherwise the timing behaviour of most operations on hash table will deteriorate.



Example : The integers given below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function. 1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50 An element can be mapped to a location in the hash table using the mapping function **key % 10**.

Hash Table Location	Mapped element
0	5, 10, 15, 50
1	1, 21, 31
2	2, 22, 32
3	3, 33, 48
4	4, 34, 49



3. Open Addressing

Separate chaining requires additional memory space for pointers. Open addressing hashing is an alternate method of handling collision. In

open addressing, if a collision occurs, alternate cells are tried until an empty cell is found.

- I. Linear probing
- II. Quadratic probing
- III. Double hashing.

I. Linear Probing

In linear probing, whenever there is a collision, cells are searched sequentially (with wrap around) or an empty cell. Fig. shows the result of inserting keys {5,18,55,78,35,15} using the hash function ($f(key) = key \% 10$) and linear probing strategy.

	Empty Table	After 5	After 18	After 55	After 78	After 35	After 15
0							15
1							
2							
3							
4							
5		5	5	5	5	5	5
6				55	55	55	55
7						35	35
8			18	18	18	18	18
9					78	78	78

Linear probing is easy to implement but it suffers from "**primary clustering**"

When many keys are mapped to the same location (clustering), linear probing will not distribute these keys evenly in the hash table. These keys will be stored in neighbourhood of the location where they are mapped. This will lead to clustering of keys around the point of collision

II. Quadratic probing

One way of reducing "primary clustering" is to use quadratic probing to resolve collision.

Suppose the "key" is mapped to the location j and the cell j is already occupied. In quadratic probing, the location $j, (j+1), (j+4), (j+9), \dots$ are examined to find the first empty cell where the key is to be inserted.

This table reduces primary clustering.

It does not ensure that all cells in the table will be examined to find an empty cell. Thus, it may be possible that key will not be inserted even if there is an empty cell in the table.

III. Double Hashing

This method requires two hashing functions $f1(key)$ and $f2(key)$.

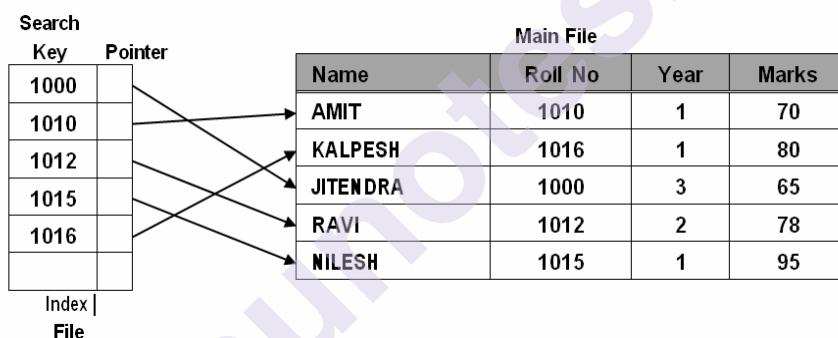
Problem of clustering can easily be handled through double hashing. Function f_1 (key) is known as primary hash function. In case the address obtained by f_1 (key) is already occupied by a key, the function f_2 (key) is evaluated.

The second function f_2 (key) is used to compute the increment to be added to the address obtained by the first hash function f_1 (key) in case of collision.

The search for an empty location is made successively at the addresses f_1 (key) + f_2 (key), f_1 (key) + 2 f_2 (key), f_1 (key) + 3 f_2 (key),...

10.6 Indexing

Indexing is used to speed up retrieval of records. It is done with the help of a separate sequential file. Each record of in the index file consists of two fields, a key field and a pointer into the main file. To find a specific record for the given key value, index is searched for the given key value. Binary search can be used to search in index file. After getting the address of record from index file, the record in main file can easily be retrieved.



Index file is ordered on the ordering key Roll No. each record of index file points to the corresponding record. Main file is not sorted.

Advantages of indexing over sequential file:

Sequential file can be searched defectively on ordering key. When it is necessary to search for a record on the basis of some other attribute than the ordering key field, sequential file representation is inadequate.

Multiple indexes can be maintained for each type of field used for searching. Thus, indexing provides much better flexibility.

An index file usually requires less storage space than the main file. A binary search on sequential file will require accessing of more blocks. This can be explained with the help of the following example. Consider the example of a sequential file with $r = 1024$ records of fixed length with record size $R = 128$ bytes stored on disk with block size $B = 2048$ bytes.

Number of blocks required to store the file =

Number of block accesses for searching a record= $\log_2 64 = 6$

Suppose, we want to construct an index on a key field that is V = 4 bytes long and the block pointer is P = 4 bytes long.

A record of an index file is of the form $< V, Pj >$ and it will need 8 bytes per entry.

Total Number of index entries = 1024

Number of blocks b' required to store the file =

Number of block accesses for searching a record = $\log_2 4 = 2$

With indexing, new records can be added at the end of the main file. It will not require movement of records abs in the case of sequential file. Updation of index file requires fewer block accesses compare to sequential file

10.6.1 Types of Indexes:

- ✓ Primaryindexes
- ✓ Clusteringindexes
- ✓ Secondaryindexes

1. Primary Indexes (Indexed Sequential File):

An indexed sequential file is characterized by Sequential organization (ordered on primary key)

Indexed on primary key An indexed sequential file is both ordered and indexed.

Records are organized in sequence based on a key field, known as primary key.

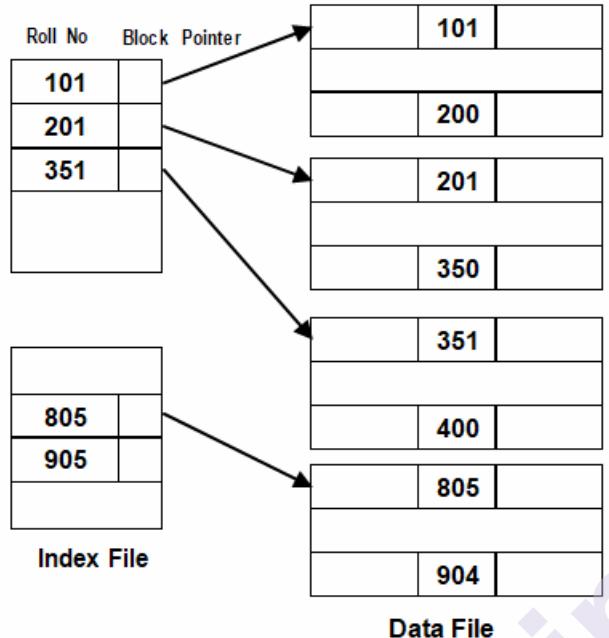
An index to the file is added to support random access. Each record in the index file consists of two fields : a key field, which is the same as the key field in the main file.

Number of records in the index file is equal to the number of blocks in the main file (data file) and not equal to the number of records in the main file (data file). To create a primary index on the ordered files how n in the Fig. we use the roll no field as primary key. Each entry in the index file has roll no value and a block pointer. The first three index entries are as follows.

```
<101,addressofblock1>
<201,addressofblock2>
<351,addressofblock3>
```

Total number of entries in index is same as the number of disk blocks in the ordered data file.

A binary search on the index file requires very few block accesses

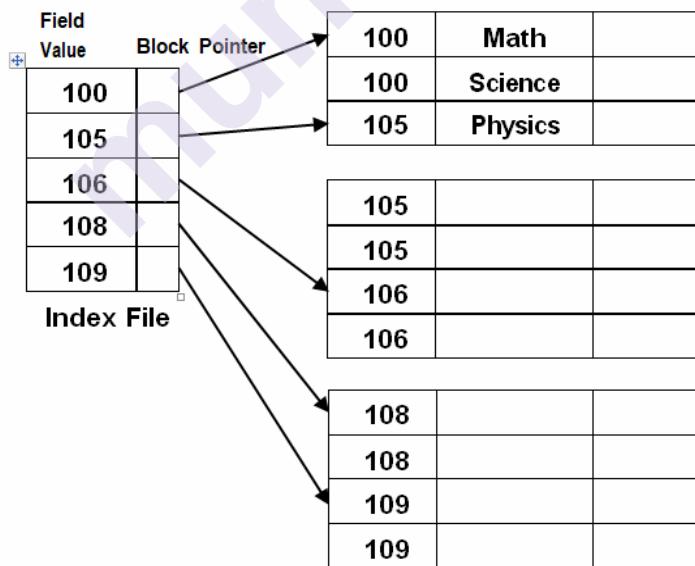


Primary Index on ordering key field roll number

2. Clustering Indexes

If records of a file are ordered on a non-key field, we can create a different type of index known as clustering index.

A non-key field does not have distinct value for each record. A Clustering index is also an ordered file with two fields.



Field Clustering Data File

Example of clustering index on roll no

3. Secondary indexes (Simple Index File)

While the hashed, sequential and indexed sequential files are suitable for operations based on ordering key or the hashed key. Above

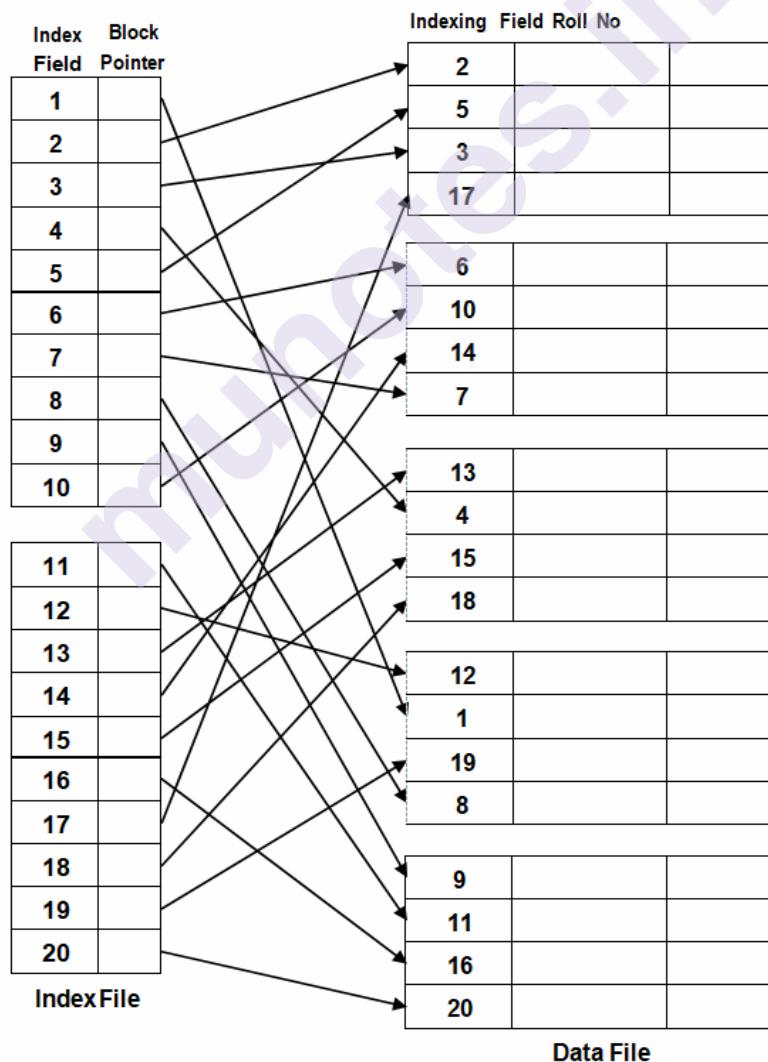
file organizations are not suitable for operations involving a search on a field other than ordering or hashed key.

If searching is required on various keys, secondary indexes on these fields must be maintained. A secondary index is an ordered file with two fields. Some non-ordering field of the data file.

A block pointer There could be several secondary indexes for the same file.

One could use binary search on index file as entries of the index file are ordered on secondary key field. Records of the data files are not ordered on secondary key field. A secondary index requires more storage space and longer search time than does a primary index.

A secondary index file has an entry for every record where as primary index file has an entry for every block in data file. There is a single primary index file but the number of secondary indexes could be quite a few.



A secondary index on a non-ordering key field

10.7 REHASHING

As the name suggests, **rehashing means hashing again**. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

10.7.1 Why Rehashing?

Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases as explained above. This might not give the required time complexity of $O(1)$.

Hence, rehash must be done, increasing the size of the bucket Array so as to reduce the load factor and the time complexity.

10.7.2 HOW REHASHING IS DONE?

Rehashing can be done as follows:

- For each addition of a new entry to the map, check the load factor.
- If it's greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucket array.
- Then traverse to each element in the old bucket Array and call the insert() for each so as to insert it into the new larger bucket array.

10.8 SUMMARY

In this chapter of Hashing function we discussed about impotence of Hashing, Hashing Function and how it works. Real world example and advantages and disadvantages of Hashing. Students who will choose Database or Data science as career will get benefites of this chapter.

10.9 QUESTIONS

1. The integers given below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function. 1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50
2. Write a difference between Separate Chaining and Open Addressing.
3. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function $h(k) = k \bmod 10$ and linear probing. What is the resultant hash table?

4. Hash the following in the table of size 11. Use any two collision resolution techniques. 23, 55, 10, 71, 67, 32, 100, 18, 10, 90, 44.
5. What is hashing? Explain hashing function.
6. Explain Linear Probing with example.
7. What do you mean by Collision? How to avoid?
8. How Rehashing is done?
9. Give importance of Indexing.
10. Explain Different Hashing Techniques.

10.10 REFERENCE FOR FURTHER READING

<https://www.cs.bham.ac.uk/~jxb/DSA/dsa.pdf>

<http://www.darshan.ac.in/Upload/DIET/Documents/CE/>

<https://www.computer-pdf.com/programming/781-tutorial-data-structure-and-algorithm-notes.html>

<https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables>

<https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/>



11

GRAPHS

Unit Structure:

- 11.0 Objective
- 11.1 Introduction
- 11.2 Some Important Terminology
- 11.3 Memory Representation of Graph
 - 11.3.1 Adjacency Matrix Representation
 - 11.3.2 Adjacency List Representation
- 11.4 Basic Operations
- 11.5 Graph Traversal –systematically visiting all vertices
 - 11.5.1 Depth First Traversal (DFS)
 - 11.5.2 Breadth First Search (BFS)
- 11.6 Applications of the Graph
- 11.7 Reachability
- 11.8 Shortest Path Problems
 - 11.8.1 Shortest paths -Dijkstra's algorithm
 - 11.8.2 Shortest paths Floyd's algorithm
- 11.9 Spanning trees
 - 11.9.1 Prim's Algorithm - A greedy vertex-based approach.
 - 11.9.2 Kruskal's algorithm -A greedy edge-based approach.
- 11.10 Summary
- 11.11 Questions
- 11.12 Reference for further reading

11.0 OBJECTIVE

This chapter would make you understand the following concepts:

- Graph,
- Graph Terminology,
- Memory Representation of Graph,
- Adjacency Matrix Representation of Graph,
- Adjacency List or Linked Representation of Graph,
- Operations Performed on Graph,
- Graph Traversal,
- Applications of the Graph,

- Reachability,
 - Shortest Path Problems,
 - Spanning Trees.
-

11.1 INTRODUCTION:

Graph G is a pair (V, E) , where V is a finite set of vertices and E is a finite set of edges.

We will often denote $n = |V|$, $e = |E|$.

A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and the edges by lines.

A graph G consist of an on-empty set V called these to f nodes (points, vertices) of the graph, a set E which is these to fedges and a mapping from these to fedges E to a set of pairs of elements of V.

It is also convenient to write a graph as $G = (V, E)$.

Notice that definition of graph implies that to every edge of a graph G, we can associate a pair of nodes of the graph. If an edge XCE is thus associated with a pair of nodes (u, v) where $u, v \in V$ then we says that edge x connect u and v.

11.2 SOME IMPORTANT TERMINOLOGY

a. Adjacent Nodes

Any two nodes which are connected by an edge in a graph are called adjacent node.

b. Directed & Undirected Edge

In a graph $G = (V, E)$ an edge which is directed from one end to another end is called a directed edge, while the edge which has no specific direction is called undirected edge.

c. Directed graph(Digraph)

A graph in which every edge is directed is called directed graph or digraph.

d. Undirected graph

A graph in which every edge is undirected is called undirected graph.

e. Mixed Graph

If some of the edges are directed and some are undirected in graph then the graph is called mixed graph.

f. Loop(Sling)

An edge of a graph which joins a node to it self is called a loop (sling).

g. Parallel Edges

In some directed as well as undirected graphs, we may have certain pairs of nodes joined by more than one edges, such edges are called Parallel edges.

h. Multi graph

Any graph which contains some parallel edges is called multi graph.

i. Weighted Graph

A graph in which weights are assigned to every edge is called weighted graph.

j. Isolated Node

In a graph a node which is not adjacent to any other node is called isolated node.

k. Null Graph

A graph containing only isolated nodes are called null graph. In other words set of edges in null graph is empty.

l. Path of Graph

Let $G = (V, E)$ be a simple digraph such that the terminal node of any edge in these quence is the initial node of the edge, if any appearing next in these quence defined as path of the graph.

m. Length of Path

The number of edges appearing in these quence of the path is called length of path.

n. Degree of vertex

Then o of edges which have V as their terminal node is call as in degree of node V

Then o of edges which have V as their initial node is call as out degree of node V

Sum of in degree and out degree of node V is called its Total Degree or Degree of vertex.

o. Simple Path (Edge Simple)

A path in a diagraph in which the edges are distinct is called simple path or edge simple.

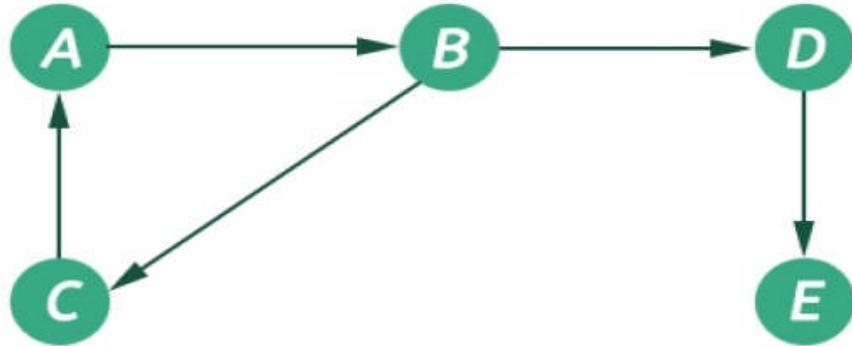
p. Elementary Path (Node Simple)

A path in which all the nodes through which it traverses are distinct is called elementary path.

q. Cycle(Circuit)

A path which originates and ends in the same node is called cycle (circuit).

11.3 MEMORY REPRESENTATION OF GRAPH



In this graph, there are five vertices and five edges. The edges are directed. As an example, if we choose the edge connecting vertices B and D, the source vertex is B and destination is D. So we can move B to D but not move from D to B.

The graphs are non-linear, and it has no regular structure. To represent a graph in memory, there are few different styles. These styles are –

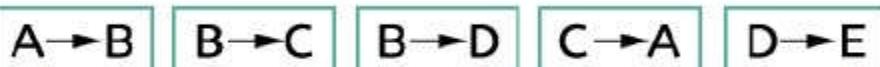
1. Adjacency matrix representation
2. Edge list representation
3. Adjacency List representation

11.3.1 Adjacency Matrix Representation

We can represent a graph using Adjacency matrix. The given matrix is an adjacency matrix. It is a binary, square matrix and from ith row to jth column, if there is an edge, that place is marked as 1. When we will try to represent an undirected graph using adjacency matrix, the matrix will be symmetric.

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

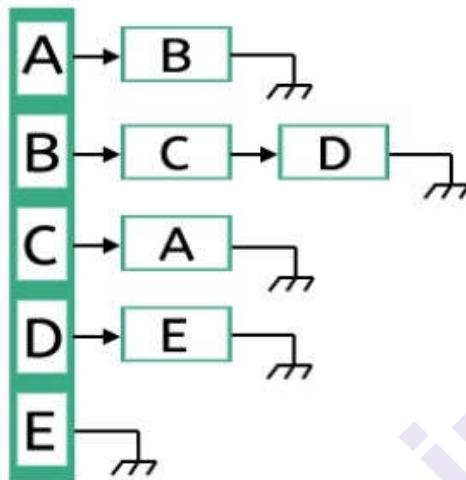
11.3.2 Edge List Representation



Graphs can be represented using one dimensional array also. This is called the edge list. In this representation there are five edges are present, for each edge the first element is the source and the second one is the destination. For undirected graph representation the number of elements in the edge list will be doubled.

11.3.3 Adjacency List Representation

This is another type of graph representation. It is called the adjacency list. This representation is based on Linked Lists. In this approach, each Node is holding a list of Nodes, which are Directly connected with that vertices. At the end of list, each node is connected with the null values to tell that it is the end node of that list.



11.4 BASIC OPERATIONS

Following are the basic primary operations that can be performed on a Graph:

- ✓ **Add Vertex** – Adds a vertex to the graph.
- ✓ **Add Edge** – Adds an edge between the two vertices of the graph.
- ✓ **Display Vertex** – Displays a vertex of the graph.

11.5 GRAPH TRAVERSAL –SYSTEMATICALLY VISITING ALL VERTICES

In order to traverse a graph, i.e. systematically visit all its vertices, we clearly need a strategy for exploring graphs which guarantees that we do not miss any edges or vertices. Because, unlike trees, graphs do not have a root vertex, there is no natural place to start a traversal, and therefore we assume that we are given, or randomly pick, a starting vertex i . There are two strategies for performing graph traversal. The first is known as breadth first traversal: We start with the given vertex i . Then we visit its neighbours one by one (which must be possible no matter which implementation we use), placing them in an initially empty queue. We then remove the first vertex from the queue and one by one put its neighbours at the end of the queue. We then visit the next vertex in the queue and again put its neighbours at the end of the queue. We do this until the queue is empty. However, there is no reason why this basic algorithm should ever terminate. If there is a circle in the graph, like A, B, C in the first unweighted graph above, we would revisit a vertex we have already visited, and thus we would run into an infinite loop (visiting A's neighbours puts B onto the queue, visiting that (eventually) gives us C,

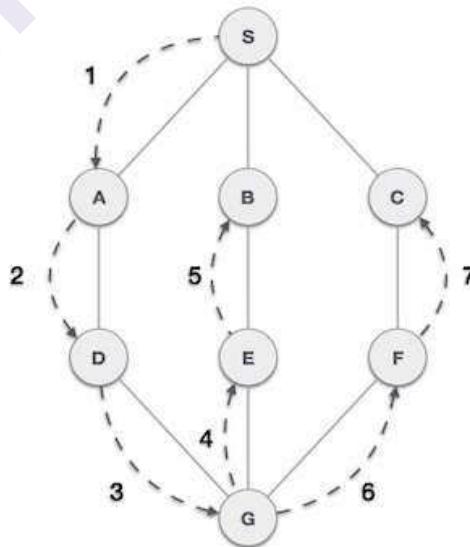
and once we reach C in the queue, we get A again). To avoid this we create a second array done of booleans, where $\text{done}[j]$ is true if we have already visited the vertex with number j , and it is false otherwise. In the above algorithm, we only add a vertex j to the queue if $\text{done}[j]$ is false. Then we mark it as done by setting $\text{done}[j] = \text{true}$. This way, we will not visit any vertex more than once, and for a finite graph, our algorithm is bound to terminate. In the example we are discussing, breadth first search starting at A might yield: A, B, D, C, E.

To see why this is called breadth first search, we can imagine a tree being built up in this way, where the starting vertex is the root, and the children of each vertex are its neighbours (that haven't already been visited). We would then first follow all the edges emanating from the root, leading to all the vertices on level 1, then find all the vertices on the level below, and so on, until we find all the vertices on the 'lowest' level. The second traversal strategy is known as depth first traversal: Given a vertex i to start from, we now put it on a stack rather than a queue (recall that in a stack, the next item to be removed at any time is the last one that was put on the stack). Then we take it from the stack, mark it as done as for breadth first traversal, look up its neighbours one after the other, and put them onto the stack. We then repeatedly pop the next vertex from the stack, mark it as done, and put its neighbours on the stack, provided they have not been marked as done, just as we did for breadth first traversal. For the example discussed above, we might (starting from A) get: A, B, C, E, D.

Again, we can see why this is called depth first by formulating the traversal as a search tree and looking at the order in which the items are added and processed.

11.5.1 Depth First Traversal

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 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.)

Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

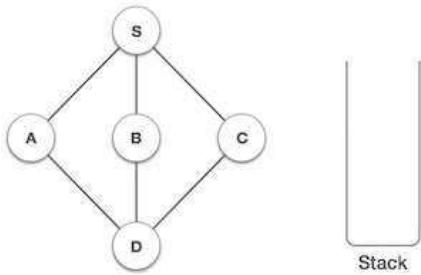
Steps	Traversal	Description
1.	 	Initialize the stack.
2.	 	Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.	 	Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A . Both S and D are adjacent to A but we are concerned for unvisited nodes only.

4.		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose B , mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7.		Only unvisited adjacent node is from D is C now. So we visit C , mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Depth First Traversal in C

We shall not see the implementation of Depth First Traversal (or Depth First Search) in C programming language. For our reference purpose, we shall follow our example and take this as our graph model –



Implementation in C

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex {
    char label;
    bool visited;
}; //stack variables
int stack[MAX];
int top = -1;
//graph variables
//array of vertices
struct Vertex* lstVertices[MAX];
//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;
//stack functions
void push(int item) {
    stack[++top] = item;
}
int pop() {
    return stack[top--];
}
int peek() {
    return stack[top];
}
bool isStackEmpty() {
    return top == -1;
} //graph functions
//add vertex to the vertex list
void addVertex(char label) {
    struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));

```

```

vertex->label = label;
vertex->visited = false;
lstVertices[vertexCount++] = vertex;
}
//add edge to edge array
void addEdge(int start,int end) {
adjMatrix[start][end] = 1;
adjMatrix[end][start] = 1;
}
//display the vertex
void displayVertex(int vertexIndex) {
printf("%c ",lstVertices[vertexIndex]->label);
}
//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
int i;
for(i = 0; i<vertexCount; i++) {
if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false) {
return i;
}
}
return -1;
}
void depthFirstSearch() {
int i;
//mark first node as visited
lstVertices[0]->visited = true;
//display the vertex
displayVertex(0);
//push vertex index in stack
push(0);
while(!isEmpty()) {
//get the unvisited vertex of vertex which is at top of the stack
int unvisitedVertex = getAdjUnvisitedVertex(peek());
//no adjacent vertex found
if(unvisitedVertex == -1) {
pop();
} else {
lstVertices[unvisitedVertex]->visited = true;
displayVertex(unvisitedVertex);
push(unvisitedVertex);
}
}

```

```

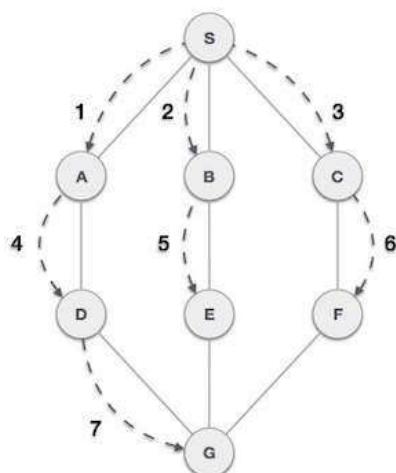
}

//stack is empty, search is complete, reset the visited flag
for(i = 0;i <vertexCount;i++) {
    lstVertices[i]->visited = false;
}
}int main() {
int i, j;
for(i = 0; i<MAX; i++) // set adjacency {
    for(j = 0; j<MAX; j++) // matrix to 0
        adjMatrix[i][j] = 0;
    }
addVertex('S'); // 0
addVertex('A'); // 1
addVertex('B'); // 2
addVertex('C'); // 3
addVertex('D'); // 4
addEdge(0, 1); // S - A
addEdge(0, 2); // S - B
addEdge(0, 3); // S - C
addEdge(1, 4); // A - D
addEdge(2, 4); // B - D
addEdge(3, 4); // C - D
printf("Depth First Search: ");
depthFirstSearch();
return 0;
}

```

11.5.2 Breadth First Search (BFS)

Algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

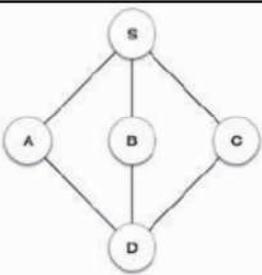
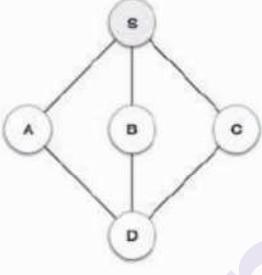
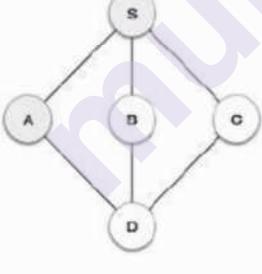
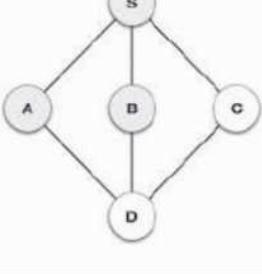


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.

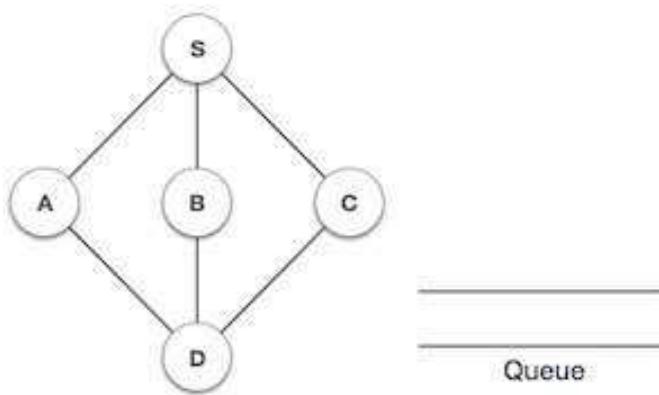
Steps	Traversal	Description
1.	 Queue	Initialize the queue.
2.	 Queue	We start from visiting S (starting node), and mark it as visited.
3.	 Queue	We then see an unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A , mark it as visited and enqueue it.
4.	 Queue	Next, the unvisited adjacent node from S is B . We mark it as visited and enqueue it.

5.	 	Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.
6.	 	Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7.	 	From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Breadth First Traversal in C

We shall not see the implementation of Breadth First Traversal (or Breadth First Search) in C programming language. For our reference purpose, we shall follow our example and take this as our graph model –



Implementation in C

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex {
    char label;
    bool visited;
};
//queue variables
int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;
//graph variables
//array of vertices
struct Vertex* lstVertices[MAX];
//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;
//queue functions
void insert(int data) {
    queue[++rear] = data;
    queueItemCount++;
}
int removeData() {
    queueItemCount--;
    return queue[front++];
}
bool isQueueEmpty() {
    return queueItemCount == 0;
}
//graph functions
//add vertex to the vertex list
void addVertex(char label) {
    struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
    vertex->label = label;
    vertex->visited = false;
    lstVertices[vertexCount++] = vertex;
}
//add edge to edge array

```

```

voidaddEdge(intstart,int end) {
adjMatrix[start][end] = 1;
adjMatrix[end][start] = 1;
}
//display the vertex
voiddisplayVertex(intvertexIndex) {
printf("%c ",lstVertices[vertexIndex]->label);
}
//get the adjacent unvisited vertex
intgetAdjUnvisitedVertex(intvertexIndex) {
int i;
for(i = 0; i<vertexCount; i++) {
if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
return i;
}
return -1;
}
voidbreadthFirstSearch() {
int i;
//mark first node as visited
lstVertices[0]->visited = true;
//display the vertex
displayVertex(0);
//insert vertex index in queue
insert(0);
intunvisitedVertex;
while(!isQueueEmpty()) {
//get the unvisited vertex of vertex which is at front of the queue
inttempVertex = removeData();
//no adjacent vertex found
while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
lstVertices[unvisitedVertex]->visited = true;
displayVertex(unvisitedVertex);
insert(unvisitedVertex);
}
}
//queue is empty, search is complete, reset the visited flag
for(i = 0;i<vertexCount;i++) {
lstVertices[i]->visited = false;
}
}

```

```

int main() {
    int i, j;
    for(i = 0; i<MAX; i++) // set adjacency {
        for(j = 0; j<MAX; j++) // matrix to 0
            adjMatrix[i][j] = 0;
    }
    addVertex('S'); // 0
    addVertex('A'); // 1
    addVertex('B'); // 2
    addVertex('C'); // 3
    addVertex('D'); // 4
    addEdge(0, 1); // S - A
    addEdge(0, 2); // S - B
    addEdge(0, 3); // S - C
    addEdge(1, 4); // A - D
    addEdge(2, 4); // B - D
    addEdge(3, 4); // C - D
    printf("\nBreadth First Search: ");
    breadthFirstSearch();
    return 0;
}

```

If we compile and run the above program, it will produce the following result –

Breadth First Search: S A B C D

11.6 APPLICATIONS OF THE 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.

In World Wide Web, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.

In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

Thus the development of algorithms to handle graphs is of major interest in the field of computer science.

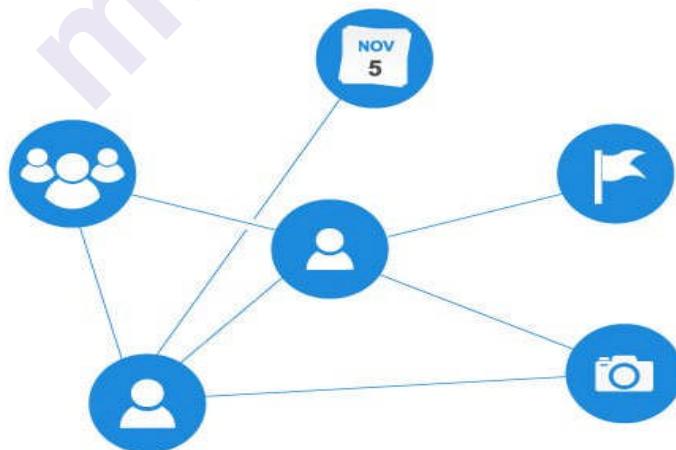
Social graphs draw edges between you and the people, places and things you interact with online.

Facebook's Graph API

Facebook's Graph API is perhaps the best example of application of graphs to real life problems. The Graph API is a revolution in large-scale data provision.

On The Graph API, everything is a vertex or node. This are entities such as Users, Pages, Places, Groups, Comments, Photos, Photo Albums, Stories, Videos, Notes, Events and so forth. Anything that has properties that store data is a vertice.

And every connection or relationship is an edge. This will be something like a User posting a Photo, Video or Comment etc., a User updating their profile with a their Place of birth, a relationship status Users, a User liking a Friend's Photo etc.



The Graph API uses this collections of vertices and edges (essentially graph data structures) to store its data. The Graph API is also a GraphQL API. This is the language it uses to build and query the schema.

The Graph API has come into some problems because of it's ability to obtain unusually rich info about user's friends.

Knowledge Graphs

Google's Knowledge Graph

A knowledge graph has something to do with linking data and graphs...some kind of graph-based representation of knowledge. It still isn't what is can and can't do yet. Recommendation Engines

Yelp's Local Graph

Yelps has been slowly phasing out their old Fusion API for a GraphQL API.

The Local Graph API promises to make it easier for developers to integrate Yelp's data and share great local businesses through their apps. Graph QL leverages the power of graph data structures by modeling the business problem as a graph within its schema. The most common use case for GraphQL is operating on graph data structures. Both Apollo Client and Relay operate on GraphQL data as a normalized graph. On the Local Graph API, Yelp represents your business as a vertice with name, id, alias, is_claimed, is_closed etc. graph properties. Yelp also creates additional vertices for Place (as custom type Location in GraphQL schema,), Categories (as custom type Category in GraphQL schema), Review (as type Review) and Hours (as type Hours) Yelp creates edges with relationships such as the location of a business with a certain name, the opening hours of a business, the reviews of a business, the category of a business. Using the local graph feature, a yelp app can uses your location to match recommendations of businesses close to you. In this case your location and the location of the business are both vertices while the recommendation is the edge.

Path Optimization Algorithms

Path optimizations are primarily occupied with finding the best connection that fits some predefined criteria e.g. speed, safety, fuel etc or set of criteria e.g prodecures, routes.

In unweighted graphs, the Shortest Path of a graph is the path with the least number of edges. Breadth First Search (BFS) is used to find the shortest paths in graphs—we always reach a node from another node in the fewest number of edges in breadth graph traversals.

Flight Networks

For flight networks, efficient route optimizations perfectly fit graph data structures. Using graph models, airport procedures can be modelled and optimized efficiently. Computing best connections in flight networks is a key application of algorithm engineering. In flight network, graph data structures are used to compute shortest paths and fuel usage in route planning, often in a multi-modal context.

The vertices in flight networks are places of departure and destination, airports, aircrafts, cargo weights.

The flight trajectories between airports are the edges. Turns out it's very feasible to fit graph data structures in route optimizations because of precompiled full distance tables between all airports.

Entities such as flights can have properties such as fuel usage, crew pairing which can themselves be more graphs.

GPS Navigation Systems

Car navigations also use Shortest Path APIs. Although this is still a type of a routing API it would differ from the Google Maps Routing API because it is single-source (from one vertex to every other i.e. it computes locations from where you are to any other location you might be interested in going.)

BFS is used to find all neighbouring locations.

11.7 REACHABILITY

In graph theory, reachability refers to the ability to get from one vertex to another within a graph. A vertex s can reach a vertex t (and t is reachable from s) if there exists a sequence of adjacent vertices (i.e. a path) which starts with s and ends with t . In an undirected graph, reachability between all pairs of vertices can be determined by identifying the connected components of the graph. Any pair of vertices in such a graph can reach each other if and only if they belong to the same connected component; therefore, in such a graph, reachability is symmetric (s reaches t iff t reaches s). The connected components of an undirected graph can be identified in linear time. The remainder of this article focuses on the more difficult problem of determining pairwise reachability in a directed graph (which, incidentally, need not be symmetric).

For a directed graph, $G = (V, E)$ with vertex set V and edge set E , the reachability relation of G is the transitive closure of E , which is to say the set of all ordered pairs (s, t) of vertices in V for which there exists a sequence of vertices $v_0 = s, v_1, v_2, \dots, v_k = t$ such that the edge (v_{i-1}, v_i) is in E for all $1 \leq i \leq k$.

11.8 SHORTEST PATH PROBLEMS

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

The problem of finding the shortest path between two intersections on a road map may be modeled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.

11.8.1 Shortest paths -Dijkstra's algorithm

A common graph based problem is that we have some situation represented as a weighted di-graph with edges labelled by non-negative numbers and need to answer the following question:

For two particular vertices, what is the shortest route from one to the other? Here, by "shortest route" we mean a path which, when we add up the weights along its edges, gives the smallest overall weight for the path. This number is called the length of the path. Thus, a shortest path is one with minimal length. Note that there need not be a unique shortest path, since several paths might have the same length. In a disconnected graph there will not be a path between vertices in different components, but we can take care of this by using 1 once again to stand for "no path at all".

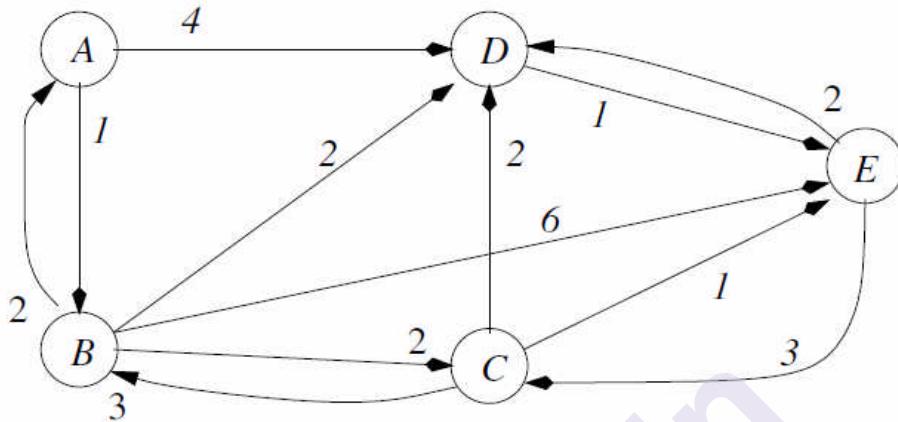
Dijkstra's algorithm. The first version of such an algorithm is not as efficient as it could be, but it is relatively simple and certainly correct. (It is always a good idea to start with an inefficient simple algorithm, so that the results from it can be used to check the operation of a more complex efficient algorithm.) The general idea is that, at each stage of the algorithm's operation, if an entry $D[u]$ of the array D has the minimal value among all the values recorded in D , then the overestimate $D[u]$ must actually be tight, because the improvement algorithm discussed above cannot possibly find a shortcut.

The following algorithm implements that idea:

```
// Input: A directed graph with weight matrix 'weight' and
// a start vertex 's'.
// Output: An array 'D' of distances as explained above.
// We begin by building the distance overestimates.
D[s] = 0 // The shortest path from s to itself has length zero.
for ( each vertex z of the graph ) {
    if ( z is not the start vertex s )
        D[z] = infinity // This is certainly an overestimate.
}
// We use an auxiliary array 'tight' indexed by the vertices,
// that records for which nodes the shortest path estimates
// are "known" to be tight by the algorithm.
for ( each vertex z of the graph ) {
    tight[z] = false
}
// We now repeatedly update the arrays 'D' and 'tight' until
// all entries in the array 'tight' hold the value true.
repeat as many times as there are vertices in the graph {
    find a vertex u with tight[u] false and minimal estimate D[u]
    tight[u] = true
    for ( each vertex z adjacent to u )
        if ( D[u] + weight[u][z] < D[z] )
            D[z] = D[u] + weight[u][z] // Lower overestimate exists.
}
// At this point, all entries of array 'D' hold tight estimates.
```

The time complexity of this algorithm is clearly $O(n^2)$ where n is the number of vertices, since there are operations of $O(n)$ nested within the repeat of $O(n)$.

A simple example. Suppose we want to compute the shortest path from A (node 0) to E (node 4) in the weighted graph we looked at before:



A direct implementation of the above algorithm, with some code added to print out the status of the three arrays at each intermediate stage, gives the following output, in which ∞ is used to represent the infinity symbol

Computing shortest paths from A

	A	B	C	D	E
D	0	∞	∞	∞	∞
tight	no	no	no	no	no
pred.	none	none	none	none	none

Vertex A has minimal estimate, and so is tight. Neighbour B has estimate decreased from ∞ to 1 taking a shortcut via A. Neighbour D has estimate decreased from ∞ to 4 taking a shortcut via A.

	A	B	C	D	E
D	0	1	∞	4	∞
tight	yes	no	no	no	no
pred.	none	A	none	A	none

Vertex B has minimal estimate, and so is tight. Neighbour A is already tight.

Neighbour C has estimate decreased from ∞ to 3 taking a shortcut via B.
 Neighbour D has estimate decreased from 4 to 3 taking a shortcut via B.
 Neighbour E has estimate decreased from ∞ to 7 taking a shortcut via B.

	A	B	C	D	E
D	0	1	3	3	7
tight	yes	yes	no	no	no
pred.	none	A	B	B	B

Vertex C has minimal estimate, and so is tight.

Neighbour B is already tight.

Neighbour D has estimate unchanged.

Neighbour E has estimate decreased from 7 to 4 taking a shortcut via C.

	A	B	C	D	E
D	0	1	3	3	4
tight	yes	yes	yes	no	no
pred.	none	A	B	B	C

Vertex D has minimal estimate, and so is tight.

Neighbour E has estimate unchanged.

	A	B	C	D	E
D	0	1	3	3	4
tight	yes	yes	yes	yes	no
pred.	none	A	B	B	C

Vertex E has minimal estimate, and so is tight.

Neighbour C is already tight.

Neighbour D is already tight.

	A	B	C	D	E
D	0	1	3	3	4
tight	yes	yes	yes	yes	yes
pred.	none	A	B	B	C

End of Dijkstra's computation.

A shortest path from A to E is: A B C E.

11.8.2 Shortest paths Floyd's algorithm

If we are not only interested in finding the shortest path from one specific vertex to all the others, but the shortest paths between every pair of vertices, we could, of course, apply Dijkstra's algorithm to every starting vertex. But there is actually a simpler way of doing this, known as Floyd's algorithm. This maintains a square matrix 'distance' which contains the overestimates of the shortest paths between every pair of

vertices, and systematically decreases the overestimates using the same shortcut idea as above. If we also wish to keep track of the routes of the shortest paths, rather than just their lengths, we simply introduce a second square matrix 'predecessor' to keep track of all the 'previous vertices'. In the algorithm below, we attempt to decrease the estimate of the distance from each vertex s to each vertex z by going systematically via each possible vertex u to see whether that is a shortcut; and if it is, the overestimate of the distance is decreased to the smaller overestimate, and the predecessor updated:

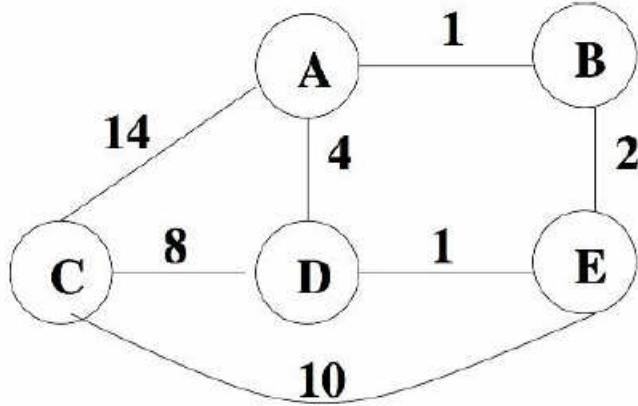
```
// Store initial estimates and predecessors.
for ( each vertex s ) {
    for ( each vertex z ) {
        distance[s][z] = weight[s][z]
        predecessor[s][z] = s
    }
}
// Improve them by considering all possible shortcuts u.
for ( each vertex u ) {
    for ( each vertex s ) {
        for ( each vertex z ) {
            if ( distance[s][u]+distance[u][z] < distance[s][z] ) {
                distance[s][z] = distance[s][u]+distance[u][z]
                predecessor[s][z] = predecessor[u][z]
            }
        }
    }
}
```

As with Dijkstra's algorithm, this can easily be adapted to the case of non-weighted graphs by assigning a suitable weight matrix of 0s and 1s. The time complexity here is clearly $O(n^3)$, since it involves three nested for loops of $O(n)$.

This is the same complexity as running the $O(n^2)$ Dijkstra's algorithm once for each of the n possible starting vertices. In general, however, Floyd's algorithm will be faster than Dijkstra's, even though they are both in the same complexity class, because the former performs fewer instructions in each run through the loops. However, if the graph is sparse with $e = O(n)$, then multiple runs of Dijkstra's algorithm can be made to perform with time complexity $O(n^2 \log_2 n)$, and be faster than Floyd's algorithm.

A simple example.

Suppose we want to compute the lengths of the shortest paths between all vertices in the following undirected weighted graph:



We start with distance matrix based on the connection weights, and trivial predecessors:

Start

	A	B	C	D	E
A	0	1	14	4	∞
B	1	0	∞	∞	2
C	14	∞	0	8	10
D	4	∞	8	0	1
E	∞	2	10	1	0

	A	B	C	D	E
A	A	A	A	A	A
B	B	B	B	B	B
C	C	C	C	C	C
D	D	D	D	D	D
E	E	E	E	E	E

Then for each vertex in turn we test whether a shortcut via that vertex reduces any of the distances, and update the distance and predecessor arrays with any reductions found. The five steps, with the updated entries in quotes, are as follows:

A :

	A	B	C	D	E
A	0	1	14	4	∞
B	1	0	'15'	'5'	2
C	14	'15'	0	8	10
D	4	'5'	8	0	1
E	∞	2	10	1	0

	A	B	C	D	E
A	A	A	A	A	A
B	B	B	'A'	'A'	B
C	C	'A'	C	C	C
D	D	'A'	D	D	D
E	E	E	E	E	E

B :

	A	B	C	D	E
A	0	1	14	4	'3'
B	1	0	15	5	2
C	14	15	0	8	10
D	4	5	8	0	1
E	'3'	2	10	1	0

	A	B	C	D	E
A	A	A	A	A	'B'
B	B	B	A	A	B
C	C	A	C	C	C
D	D	A	D	D	D
E	'B'	E	E	E	E

C :

	A	B	C	D	E
A	0	1	14	4	3
B	1	0	15	5	2
C	14	15	0	8	10
D	4	5	8	0	1
E	3	2	10	1	0

	A	B	C	D	E
A	A	A	A	A	B
B	B	B	A	A	B
C	C	A	C	C	C
D	D	A	D	D	D
E	B	E	E	E	E

D :

	A	B	C	D	E
A	0	1	'12'	4	3
B	1	0	'13'	5	2
C	'12'	'13'	0	8	'9'
D	4	5	8	0	1
E	3	2	'9'	1	0

E :

	A	B	C	D	E
A	0	1	12	4	3
B	1	0	'11'	'3'	2
C	12	'11'	0	8	9
D	4	'3'	8	0	1
E	3	2	9	1	0

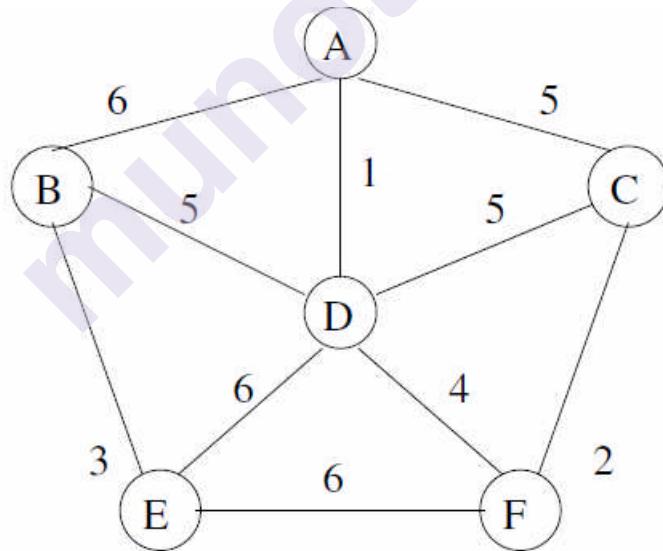
	A	B	C	D	E
A	A	A	'D'	A	B
B	B	B	'D'	A	B
C	'D'	A	C	C	'D'
D	D	A	D	D	D
E	B	E	'D'	E	E

	A	B	C	D	E
A	A	A	D	A	B
B	B	B	D	'E'	B
C	D	'E'	C	C	D
D	D	'E'	D	D	D
E	B	E	D	E	E

The algorithm finishes with the matrix of shortest distances and the matrix of associated predecessors. So the shortest distance from C to B is 11, and the predecessors of B are E, then D, then C, giving the path C D E B. Note that updating a distance does not necessarily mean updating the associated predecessor { for example, when introducing D as a shortcut between C and B, the predecessor of B remains A.

11.9 SPANNING TREES

We now move on to another common graph-based problem. Suppose you have been given a weighted undirected graph such as the following:



We could think of the vertices as representing houses, and the weights as the distances between them. Now imagine that you are tasked with supplying all these houses with some commodity such as water, gas, or electricity. For obvious reasons, you will want to keep the amount of digging and laying of pipes or cable to a minimum. So, what is the best pipe or cable layout that you can find, i.e. what layout has the shortest overall length? Obviously, we will have to choose some of the edges to dig along, but not all of them. For example, if we have already chosen the

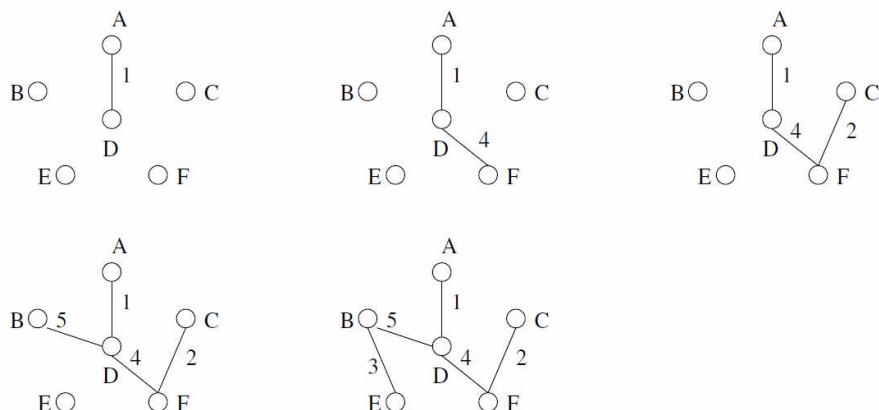
edge between A and D, and the one between B and D, then there is no reason to also have the one between A and B. More generally, it is clear that we want to avoid circles. Also, assuming that we have only one feeding-in point (it is of no importance which of the vertices that is), we need the whole layout to be connected. We have seen already that a connected graph without circles is a tree.

Hence, what we are looking for is a minimal spanning tree of the graph. A spanning tree of a graph is a subgraph that is a tree which connects all the vertices together, so it 'spans' the original graph but using fewer edges. Here, minimal refers to the sum of all the weights of the edges contained in that tree, so a minimal spanning tree has total weight less than or equal to the total weight of every other spanning tree. As we shall see, there will not necessarily be a unique minimal spanning tree for a given graph.

11.9.1 Prim's Algorithm - A greedy vertex-based approach.

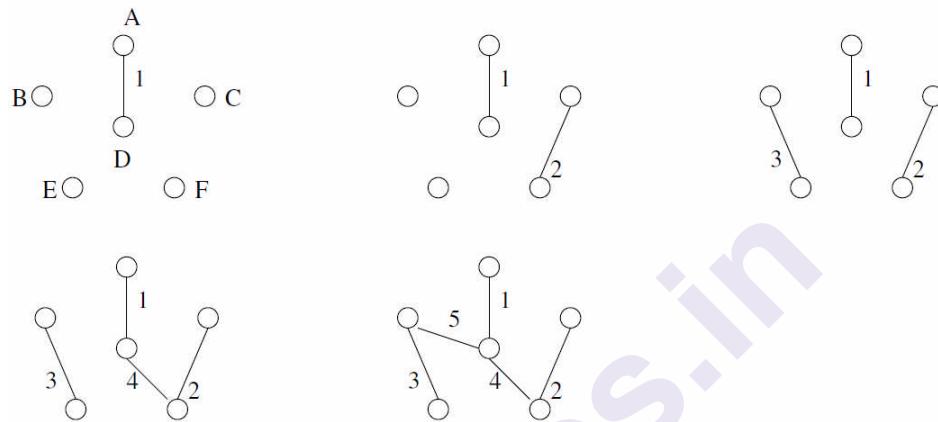
Suppose that we already have a spanning tree connecting some set of vertices S. Then we can consider all the edges which connect a vertex in S to one outside of S, and add to S one of those that has minimal weight.

This cannot possibly create a circle, since it must add a vertex not yet in S. This process can be repeated, starting with any vertex to be the sole element of S, which is a trivial minimal spanning tree containing no edges. This approach is known as Prim's algorithm. When implementing Prim's algorithm, one can use either an array or a list to keep track of the set of vertices S reached so far. One could then maintain another array or list closest which, for each vertex i not yet in S, keeps track of the vertex in S closest to i. That is, the vertex in S which has an edge to i with minimal weight. If closest also keeps track of the weights of those edges, we could save time, because we would then only have to check the weights mentioned in that array or list. For the above graph, starting with $S = \{g\}$, the tree is built up as follows:



11.9.2 Kruskal's algorithm -A greedy edge-based approach.

This algorithm does not consider the vertices directly at all, but builds a minimal spanning tree by considering and adding edges as follows: Assume that we already have a collection of edges T. Then, from all the edges not yet in T, choose one with minimal weight such that its addition to T does not produce a circle, and add that to T. If we start with T being the empty set, and continue until no more edges can be added, a minimal spanning tree will be produced. This approach is known as Kruskal's algorithm. For the same graph as used for Prim's algorithm, this algorithm proceeds as follows:



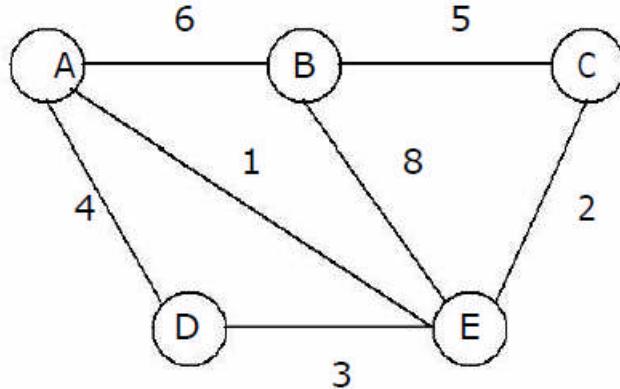
In practice, Kruskal's algorithm is implemented in a rather different way to Prim's algorithm. The general idea of the most efficient approaches is to start by sorting the edges according to their weights, and then simply go through that list of edges in order of increasing weight, and either add them to T, or reject them if they would produce a circle. There are implementations of that which can be achieved with overall time complexity $O(e \log_2 e)$, which is dominated by the $O(e \log_2 e)$ complexity of sorting the e edges in the first place. This means that the choice between Prim's algorithm and Kruskal's algorithm depends on the connectivity of the particular graph under consideration. If the graph is sparse, i.e. the number of edges is not much more than the number of vertices, then Kruskal's algorithm will have the same $O(n \log_2 n)$ complexity as the optimal priority queue based versions of Prim's algorithm, but will be faster than the standard $O(n^2)$ Prim's algorithm. However, if the graph is highly connected, i.e. the number of edges is near the square of the number of vertices, it will have complexity $O(n^2 \log_2 n)$ and be slower than the optimal $O(n^2)$ versions of Prim's algorithm.

11.10 SUMMARY

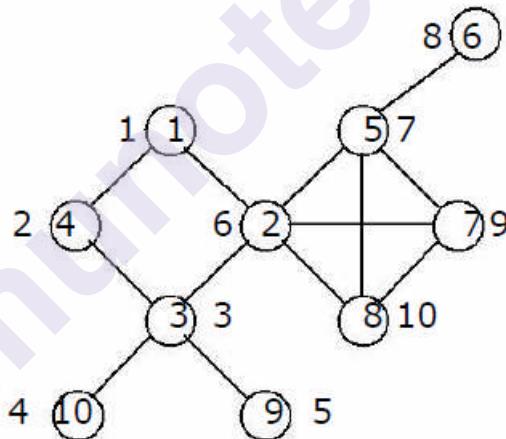
From this chapter students got the idea about graphs and its type. Implementation of graphs in their real life. As well as they understood At what situation which traversal algorithm they suppose to choose.

11.11 QUESTIONS

- Describe the algorithm to find a minimum spanning tree T of a weighted graph G .
- Find the minimum spanning tree T of the graph shown below.



- For the graph given below find the following:
Linked representation of the graph. Adjacency list. Depth first spanning tree. Breadth first spanning tree. Minimal spanning tree using Kruskal's and Prim's algorithms.



- Show that the sum of degrees of all vertices in an undirected graph is twice the number of edges.
- Explain how existence of a cycle in an undirected graph may be detected by traversing the graph in a depth first manner.
- Give an example of a connected directed graph so that a depth first traversal of that graph yields a forest and not a spanning tree of the graph.
- Write a $-C$ function to find out whether there is a path between any two vertices in a graph (i.e. to compute the transitive closure matrix of a graph)
- Construct a weighted graph for which the minimal spanning trees produced by Kruskal's algorithm and Prim's algorithm are different.
- Write a difference between Kruskal's algorithm and Prim's algorithm

9. Explain Floyd's algorithm.
10. What are Applications of the Graph

11.12 REFERENCE FOR FURTHER READING

<https://www.cs.bham.ac.uk/~jxb/DSA/dsa.pdf>

<http://www.darshan.ac.in/Upload/DIET/Documents/CE/>

<https://www.computer-pdf.com/programming/781-tutorial-data-structure-and-algorithm-notes.html>

[https://www.tutorialspoint.com/graph-and-its-representations](https://www.tutorialspoint.com/graph_and_its_representations)

<https://www.geeksforgeeks.org/applications-of-graph-data-structure/>

<https://leapgraph.com/graph-data-structures-applications/>

