UNIT

01

Introduction to Data Structures

Names of Sub-Units

Introduction to Data Structure: Classification of Data Structures, Data Structure Operations, Basic Concepts of Pointers, Structures and Union, Algorithm, Characteristics of the Algorithm.

Overview

This unit begins by discussing about the concept of Stacks. Next, the unit discusses the Operations of stack, representing stack using static arrays. Further the unit explains the Dynamic array for representing stack. Towards the end, the unit discusses the application of stack.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of data structures
- Explain the concept of classification of data structures
- Describe the operations of data structure
- Explain the significance of basic concepts of pointers, structures and union
- Discuss the characteristics of the algorithm

Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of Data Structure
- Assess the concept of classification of data structure
- Evaluate the importance of operations of data structure
- Determine the significance of Algorithm
- Explore the characteristics of algorithm

Pre-Unit Preparatory Material

- https://www.studytonight.com/data-structures/introduction-to-data-structures
- https://www.geeksforgeeks.org/data-structures/

1.1 INTRODUCTION

A data structure is a defined format for managing, assessing, retrieving and storing data. Data structures make it easy for users to work with the data they require in different ways. data structure is designed or selected for storing the data to use different algorithms, in Computer programming. The basic algorithm operations are integrated into the design of the data structure. Each data structure has information related to the data values, the association between data and functions which are applied to the data.

1.2 DATA STRUCTURES

Data structure is a way to store and organise data in computer memory so that data can be used effectively later. It can be arranged in various ways, such as the mathematical or logical model for the specific organisation of data is known as data structure. In general, the choice of the data model is based on two considerations. First, it should be defined structure to exhibit the relationships of data in the real world. Secondly, the structure should be simple so that one can able to process the data when required. Data structure is the process of arranging data and its functions.

1.3 NEED OF DATA STRUCTURE

The needs of data structures contain the following: efficiency, re-usability, and invisibility. Data structure offers a means of establishing, handling, and storage data efficiently. It also comprises the collection of data as well as the actions that can be applied to that data.

Data structures are significant for the following reasons:

- Data structure helps the programmers in effectively managing the data. It serves a greater role in improving the performance of software or program.
- Data structures are used in every program or software system to arrange the data efficiently.
- Data structures enable data to be stored in a particular manner in the memory.
- Data structure is a significant factor for different efficient algorithms. It enables to manage a large amount of data, such as a large collection of databases and indexing services, such as hash table.

- Data structure helps in data retrieval and search effectively.
- Data structures manage the retrieval and storage of data and information which can be stored in secondary and main memory.

1.4 CLASSIFICATIONS OF DATA STRUCTURE

In the classification of data structure, Trees also originate in the non-primitive and non-linear group of data structure, using tree we can signify a hierarchical relationship between the data components. The CREATE operation (it can be definite) results in storing memory for the program components. Data structure is classified into the primitive and non-primitive data structures are as follows:

- Primitive data structures: Primitive data structures are the basic data structures that can be operated on machine and data instructions directly. It can be defined by the programming languages. Primitive data types are integer, floating point number, real and pointer.
- Non primitive data structures: Non primitive data structures are the data structures that can be derived from primitive data structures. Non primitive data structures are stacks, graphs, trees and linked lists

1.5 DATA STRUCTURE OPERATIONS

Data Structure is well-defined as a mathematical or logical model to store data and perform operation on the stored data. The operations are the purposes using which the data can be managed. All the data structure has some common operations to manipulate data and process it for the user. The operation on data structures are as follows:

- Traversing: Each data structure has a set of data elements. Traversing refers to visiting each element of data structure to manage operations such as searching or sorting.

  For example: if we want to compute the average of marks secured by students in five different subjects. We need to traverse the array of marks and compute the total sum. We will divide that sum by several subjects, i.e, 5, to compute the average.
- Insertion: It is the process of adding elements to the data structures at any location. n is the size of the data structure and we can insert n-1 data elements into it.
- Deletion: The phenomenon of removing the element from the data structure is known as deletion. We delete the element from the data structure at any location. Underflow condition occurs when trying to delete an element from the empty data structure.
- Searching: It refers to detecting the location of the element in the data structure is known as searching. Linear search and binary search are the two algorithms to evaluate searching.
- Sorting: It refers to the phenomenon of organising the data structure in a particular order is called sorting. Several algorithms can be used to evaluate sorting. Example: selection sort, insertion sort and bubble sort.
- Merging: The two lists such as List X and List Y of size of K and L, respectively. They have the identical type of elements combined to generate the third list, List Z of size (K+L). This phenomenon is known as merging.

1.6 POINTERS

A Pointer is a derived data type that stores the address of another variable. A Pointer contains memory addresses as their values. Pointers can be used to assign, access, and manipulate data values stored in the memory allotted to a variable since it can access the memory address of that variable.

Further description about the Pointer is discussed below:

1.6.1 Pointer Basics

Pointers are the variables that can be used to store the location of value found in the memory. It refers to the location that holds its memory address. The phenomenon of acquiring value stored at the location is a reference by the pointer is called dereferencing. It is the same as an index for textbooks where each page can be referred by the page number found in index. By using this concept, one can able to identify the page using the location referred to a particular index. These pointers are used in the dynamic implementation of different data structures, such as lists or stacks. An example of pointer basics that is declaring pointers is given below:

- Declaring pointers: Pointer declarations is * operation. The syntax is,

```
int a=12;
int *ptr;      //pointer declaration
pte =&a;     // pointer initialization
```

from the given example, p is a pointer and its type is termed as a pointer to int, it stores the address of integer variable.

1.6.2 Pointer with functions

You can use the function pointers to eliminate code redundancy. Example: qsort() is used to sort arrays in descending or ascending order concerning an array of structures. By using void pointers and function pointers, it is significant to use qsort for any type of data type. Function pointers are the pointers (variable) that refer to the address of the function. It will call functions at run time. The functions are evaluated at run time is known as late binding.

Syntax:

```
Returntype (*function pointer) (parameter1, parameter2);
```

Function pointer refers to point to function with a particular sign. All these functions have the same parameters and return type. Function pointer and function whose address is referred to have the same sign. Sign represents that number of parameters, return type and parameters data type of function is the same. For example:

```
int (*pfunc) (int l, int M);
// pfunc generally takes integers as return values and parameters.
int plus (int l, int K)
//in this function, the address is valued by the function pointer.
```

1.6.3 Pass by Reference

Programming languages, such as C, C++ and Java use pass by value. It can be simulated passing by reference with help of dereferenced pointers as arguments in function definition and passing in the address of the operator and on variables while calling the function. It can be passed in as a copy of the pointer but it points to the same address in memory as the original pointer. It enables the function to change value outside the function. The arguments passed inside the function are termed as dereferenced pointers. According to the programmer's perspective, it is the same as working with the values. By using the same structure in the swap function using pointers, the values outside the function will be swapped.

Sample program:

If we run the above example, the values will be sapped when swap() function has been called.

```
Void swap (int *firstvar, int *secondvar)
{
```

```
int temp;
// dereferenced pointers refers to the function is working with values at
addresses which can be passed in
temp=*firstvar;
*firstvar=*secondvar;
*secondvar=temp;
return;
}
int main(void)
{
int m = 100;
int n = 200;
printf("before swap: value of m: %d \n", m);
printf("before swap: value of n: %d \n", n);
// using "address of" operator to pass in the address of each variable
swap(&m, &n);
//check values outside the function after swap function.
printf("after swap: value of m: %d \n", m);
printf("after swap: value of n: %d \n", n);
return 0;
}
```

1.6.4 Array of Pointers

Array of pointers is an indexed set of variables where the variables are called pointers. Pointers are significant tool for developing, utilising and eliminating all forms of data structures. Array of pointers is helpful for reasons which allow to index large sets of variables numerically. Each pointer in one array points to the integer in another array. It can be printed by dereferencing the pointers. This code displays the value in memory of where pointers refer.

An sample program for array of pointers is as follows:

```
#include <iostream>
using namespace std;
const int MAX = 4;
int main () {
   int  var[MAX] = {20, 200, 400 ,600};
   for (int i = 0; i < MAX; i++) {
      cout << "Value of var[" << i << "] = ";
      cout << var[i] << endl;
   }
   return 0;
}

The running output of an pointer of array is as follows:
Value of var[0] = 20
Value of var[1] = 200
Value of var[2] = 400
Value of var[3] = 600
```

1.6.5 Pointer to Array

Pointer to an array is also acknowledged as array pointer. We are using the pointer to access the constituents of the array. We have a pointer that emphasis to the (0th) component of the array. We can similarly declare a pointer that can point to entire array rather than just a single component of the array. Declaration of the pointer to an array. Declaration of pointer to an array:

```
extern char (*p)[];
char arr[10];
char (*p)[10] = &arr;
```

The given declaration refers to the pointer to an array of four integers. In this case, we use parenthesis to denote pointer to an array. It is important to encounter pointer name and indirection operator inside brackets. Sample program for pointer to array is given below:

```
#include <iostream>
using namespace std;
int main () {
   // an array with 5 elements.
   double balance[5] = {1000.0, 5.0,2.4, 27.0, 56.0};
   double *p;
   p = balance;
   cout << "Array values using pointer " << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "*(p + " << i << ") : ";
      cout << *(p + i) << endl;
   }
   cout << "Array values using balance as address " << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "*(balance + " << i << ") : ";
      cout << *(balance + i) << endl;
   }
   return 0;
}
```

The running output of a pointer to array with 5 elements is as follows:

Array values using pointer

```
*(p + 0): 1000
*(p + 1): 5
*(p + 2): 2.4
*(p + 3): 27
*(p + 4): 56
```

Array values using balance as address

```
*(balance + 0): 1000
*(balance + 1): 5
*(balance + 2): 2.4
*(balance + 3): 27
*(balance + 4): 56
```

From this program, we can see pointer, which denotes 0th element of the array. We declare a pointer that can point to an array instead of one element of an array. This pointer is used in the multidimensional arrays. Syntax for a pointer to array of integer 5

```
data_type (*var_name)[size_of_array];
Example:
int (*ptr)[5];
```

From the given example, ptr is the pointer which refers to an array of 5 integers. The given subscript has high precedence than indirection and it is essential to use the pointer name and indirection operation inside parentheses. Data type of ptr is a pointer to an array of 5 integers.

1.7 DYNAMIC MEMORY ALLOCATION FUNCTIONS

In programming, we can see some situations where we may have to manage data that is dynamic. At the time of program execution, the number of data items may change. Several programming languages prefer dynamic memory allocation to assign memory for runtime variables. The allocation heap will be used which selects pointers. Pointers hold the address of a dynamically created array of data blocks or objects. Many structure languages choose free stores to provide storage locations. The dynamic memory allocation function is as follows:

- Malloc()
- Calloc()
- Realloc()
- Free()

1.7.1 The malloc() Function

The malloc() function allocates a single block of requested memory. It returns null when the memory is insufficient. It does not initialise the memory during execution time as it has garbage value. Syntax for malloc() is :{ptr=(cast-type*)malloc(byte-size)}.

Let us see the example of malloc() function is given below:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
int *my_pointer;
my_pointer = (int*) malloc(5*sizeof(int));
if(my_pointer)
{
cout << "Lets intilize 5 memory blocks with odd numbers" << endl << endl;
for (int i=0; i<5; i++)
{
my_pointer[i] = (i*2)+1;
}
cout << "Lets see the values" << endl << endl;
for (int i=0; i<5; i++)
{
cout << "Value at position "<<i << " is "<< *(my_pointer+i) << endl;
}
free(my_pointer);
return 0;
```

```
}
}
```

The running of a Malloc() function is given below:

Lets intilize 5 memory blocks with odd numbers

Lets see the values

```
Value at position 0 is 1
Value at position 1 is 3
Value at position 2 is 5
Value at position 3 is 7
Value at position 4 is 9
```

1.7.2 calloc() Function

The calloc() Function allocates multi block of requested memory. It initialise all bytes to zero and also returns null ifmemory is insufficient. Syntax:{ptr=(cast-type*)malloc(byte-size)}.

Let us see the example of calloc() function is given below:

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main() {
   int *pointer;
   pointer= (int *)calloc(6, sizeof(int));
   if (!pointer) {
      cout << "Memory Allocation Failed";
      exit(1);
   }
   cout << "Initializing values..." << endl
        << endl;
   for (int i = 0; i < 6; i++) {
      pointer[i] = i * 2 + 1;
   }
   cout << "Initialized values" << endl;
   for (int i = 0; i < 6; i++) {
      /* ptr[i] and *(ptr+i) can be used interchangeably */
      cout << *(pointer + i) << endl;
   }
   free(pointer);
   return 0;
}
```

The running output of a Calloc() is given below:

Initialized values

```
1
3
5
7
9
11
```

1.7.3 realloc() Function

When the memory is insufficient for calloc() or malloc(), we can reallocate the memory using realloc(). It rearranges or changes the memory size.

Let's us see an example of realloc() function is as follows:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
     int *ptr, *new_ptr;
     ptr = (int*) malloc(6*sizeof(int));
     if(ptr==NULL)
     {
          cout << "Memory Allocation Failed";
          exit(1);
     }
     for (int i=0; i<6; i++)
     {
          ptr[i] = i;
     }
     new_ptr = (int*) realloc(ptr, 0);
     if(new_ptr==NULL)
     {
          cout << "Null Pointer";
     }
     else
     {
          cout << "Not a Null Pointer";
     }
     return 0;
}
```

The output of a realloc() function is given below:

```
Null Pointer
```

1.7.4 free() Function

The free() is allocated by calloc() or malloc() should be released by using free(0 function. It needs to be mentioned or else it consumes memory until the program exit. Syntax for free() function is :

```
{free(ptr)}.
```

Let's see an example of free() function is given below:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
     int x = 8;
```

```
    int *pointer1 = NULL;
    int *ptr2 = &x;
    if(pointer1)
    {
        cout << "Pointer is not Null" << endl;
    }
    else
    {
        cout << "Pointer is Null" << endl;
    }
    free(pointer1);
    cout << *ptr2;
    return 0;
}
```

The running output of a free() function is given below:

```
Pointer is Null
8
```

1.8 STRUCTURES

Structure is a user defined data type that allows storing the amount of different data types. In structure, each element is called a member. It can assess the use of templates and classes as it stores different information.

The keywords struct is used to define the structure.

Syntax for structures is given as:

```
struct structurename
{
Datatype member1;
Datatype member2;
...
Datatype member;
};
```

Example of a structure is as follows:

```
struct employee
{ int id;
char name[10];
float salary;
};
```

1.8.1 Self-referential Structures

Self-referential structure refers to the structure have members which refer to structure variable of the same type. It can be used in dynamic data structures, such as linked list and trees. The definition of self-referential structure is mentioned here,

```
struct node {
int d;
struct node *k;
};
```

In the given structure, k is a pointer to struct node variable. It is similar to the pointer to structure and pointer to any other variable. It is a structure definition that has one member that is a pointer to a structure of its kind. These structures are essential in applications of linked data structures such as trees and lists. Contrary, the static data structure such as array where several elements that can be inserted in the array is restricted by size of the array. It can be expanded or contracted. Operations such as deletion or insertion of nodes in self-referential structures are straight forward alteration of pointers.

1.9 UNIONS

Union can be defined as user defined data type which has a collection of different variables of different data types in a same memory location. It can be defined as several members but one member has a value at a specific point in time. It is a user defined data type but structures share the same memory location.

For example :

```
Union
{
Char y;
int x;
} u;
```

The given example is user defined structure that has two members such as 'x' of type int and 'y' of type character. If we evaluate the addresses of 'x' and 'y', we can see that the addresses are distinct. We can conclude that members in structure do not share the same memory location. Like structure, we define the union in the same way but union keyword is used for defining union data type. Union contains the data members, i.e., 'x' and 'y', also evaluate the addresses of both variables and identified that both variables have the same addresses. It states that union members share the same memory location.

1.10 ALGORITHM

Algorithm is defined as a finite sequence of instructions that can be performed in a finite amount of effort in a given length of time. Algorithm should be simple and easy to understand. To execute by the computer, we need a program that needs to be written in a formal language. Computers are not flexible compared to the human mind so programs must contain more information than algorithms. Here, we may ignore the programming details and focus on the design of algorithms than programs.

1.10.1 Characteristics of Algorithm

Every algorithm has some important characteristics which need to be followed. The characteristics of algorithm are described in detail given below:

- Input specified: During the computation, the input is data to be transformed to generate the output. Algorithm must have well defined or 0 inputs. Input precision needs to know what kind of data, what type of data and how much should be.
- Output specified: The output is the data obtained from the computation. Algorithm must have well defined or 1 output. Output precision needs what kind of data, what form the output and how much the output be.

- Effectiveness: If the algorithm wants to be effective, it is significant to get output to be feasible with the available resources. It does not have redundant and unnecessary steps which could make an algorithm ineffective.
- Finiteness: The algorithm should stop eventually. Stopping refers to that you get expected output that has no possible solution. Algorithm should terminate after a finite number of steps is made. Algorithm should always terminate after a defined number of steps and not be infinite. It is significant to create a finite algorithm.
- Definiteness: Algorithm should specify every step and the steps must be involved in the process. Definiteness means mentioning the sequence of operations for making input into output. Algorithm should be unambiguous. Each step should be spelled out and must have quantitative data.

1.10.2 Elements of Algorithm

Algorithms are a sequence of instructions, implemented using programming languages, such as java, C, C++ and so on. The elements of the algorithm are as follows:

- Selection: It is the use of conditional statements such as if then and if then else
- Sequence: It is the order in which commands and behaviours are integrated into projects to generate expected outcomes.
- Iteration: Algorithms prefer repetition to execute the steps to a particular number of times or when a particular condition is attained. It is called looping. It changes the project flow by repeating the behavior until a condition is met.

1.11 CONCLUSION

- A data structure is a defined format for managing, assessing, retrieving and storing data. Data Structures make it easy for users to work with the data they require in different ways.
- The needs of data structures contain the following: efficiency, re-usability, and invisibility.
- In the classification of data structure, Trees also originate in the non-primitive and non-linear group of data structure.
- Data Structure is well-defined as a mathematical or logical model to store data and perform operation on the stored data.
- A Pointer is a derived data type that stores the address of another variable.

1.12 GLOSSARY

- Algorithm: a technique used to solve the problem
- Allocation: allocating memory for an object in the memory
- Data: contains information
- Pointer: variable who value is the address of another variable
- Dynamic memory allocation: programming technique where linked objects in data structures are developed from free store. If no longer is needed, then the object is left as garbage or returned to a free store.

1.13 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. The basic algorithm operations are integrated into the design of the data structure. Each data structure has information related to the data values. What is a Data structure?
2. Data structure offers a means of establishing, handling, and storage data efficiently. It also comprises the collection of data as well. Explain the need of data structures.
3. Trees also originate in the non-primitive and non-linear group of data structure, using tree we can signify a hierarchical relationship between the data components. Describe the classification of data structure.
4. In programming, we can see some situations where we may have to manage data that is dynamic. At the time of program execution, the number of data items may change. Describe the significance of dynamic memory allocation functions.
5. To execute by the computer, we need a program that needs to be written in a formal language. Determine the importance of algorithm.

1.14 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. Data structure is a way to store and organise data in computer memory so that data can be used effectively later. It can be arranged in various ways, such as the mathematical or logical model for the specific organisation of data is known as data structure. Refer to Section Data structures
2. The needs of data structures contain the following: efficiency, re-usability, and invisibility. Data structure offers a means of establishing, handling, and storage data efficiently. It also comprises the collection of data as well as the actions that can be applied to that data. Refer to Section Need of Data structures
3. In the classification of data structure, Trees also originate in the non-primitive and non-linear group of data structure, using tree we can signify a hierarchical relationship between the data components. Refer to Section Classification of Data Structure
4. Dynamic memory allocation is used assign memory for runtime variables. The allocation heap will be used which selects pointers. Pointers hold the address of a dynamically created array of data blocks or objects. Many structure languages choose free stores to provide storage locations. Refer to Section Dynamic Memory Allocation Functions
5. Algorithm is defined as a finite sequence of instructions that can be performed in a finite amount of effort in a given length of time. Algorithm should be simple and easy to understand. To execute by the computer, we need a program that needs to be written in a formal language. Refer to Section Algorithm

1.15 POST-UNIT READING MATERIAL

- https://www.iare.ac.in/sites/default/files/PPT/IARE_DS_PPT_3.pdf
- https://www.iare.ac.in/sites/default/files/DS.pdf

1.16 TOPICS FOR DISCUSSION FORUMS

- You can discuss with your friends the applications of data structures in a real-life environment, Classification of data structure and its need . Also discussed on pointers and Algorithm of data structure.

UNIT

02 Arrays

Names of Sub-Units

Introduction to Arrays, Array Representation, Array Operations, Polynomials, Sparse Matrices.

Overview

This unit begins by explaining the meaning of arrays. Further, it discusses the basic terminology of array representation. Further the unit explains the array operations and polynomials. Towards the end, the unit discusses the sparse matrices.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of arrays
- Explain the concept of array representation
- Describe the array operations
- Explain the significance of polynomials and sparse matrices

Learning Outcomes

At the end of this unit you would:

- Evaluate the concept of arrays
- Assess the concept of array representation
- Evaluate the importance of array operations
- Determine the significance of polynomials and sparse matrices

	Pre-Unit Preparatory Material

- https://www.kau.edu.sa/Files/0052053/Subjects/10_csphtp1_07pdf.pdf
- https://www.geeksforgeeks.org/search-insert-and-delete-in-an-unsorted-array/

2.1 INTRODUCTION

Array is a vessel which carries a fixed quantity of data items which should be of identical kind or same datatype. Most of the algorithms implemented by different Data structures use arrays. Arrays are described by two basic terms, which acts as integral part of an array is given below:

- Element: The constituent data items collectively stored in an array are known as its elements.
- Index: The position of an element is assigned a numerical index, which recognises the element is referred as its index.

2.2 REPRESENTATION OF LINEAR ARRAYS IN MEMORY

There are various ways to declare the array in different programming languages. Lets consider the array declaration as shown in Figure 1:

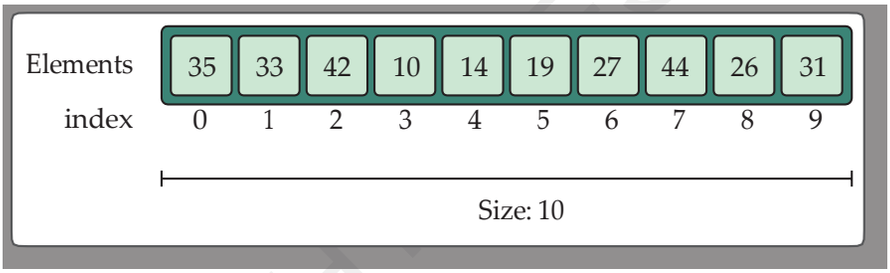


Figure 1: Array Declaration

From the above depiction, following points are worth considering:

- 0 marks as the starting position of the array. It is considered as the first element.
- The size of the array is 10 which denote that it stores 10 elements.
- Any element can be retrieved through its index. For instance, we can access the element 19 from its index 5.

2.3 FUNCTIONS OF ARRAYS

An array function is generally well-defined as a function that works with an array. The array is a mutual perception in computer programming, where several variables are assigned together with a common name. Variables are specific items that enclose numbers, letters or other data.

2.3.1 Traverse Operation

Displays all the array elements individually. This operation allows us to visit all the elements of an array by traversing through it. The following program implements the Traverse operation:

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

```
   int LA[] = {1,3,7,9,11,13};
   int i, n = 6;
   printf("The array elements are:\n");
   for(i = 0; i < n; i++)
   {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

The running output of a Traverse operation is as follows:

```
The array elements are:
LA[0] = 1
LA [1] = 3
LA [2] = 7
LA [3] = 9
LA [4] = 11
LA [5] = 13
```

2.3.2 Insertion Operation

This operation allows us to insert either single or multiple data items into an array. Depending on the situation, a new element can be added at any position as specified by index of element in the array.

The following program illustrates the Insert operation, where a data item is inserted at the specified location in the array. The following program implements the Insertion operation:

```
#include <stdio.h>
void main()
{
   int LA[] = {3,5,7,9,11};
   int item = 11, K = 3, N= 5;
   int i = 0, j = 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]);
   }
}
```

The running output of Insertion operation is given below:

```
LA [0] = 3
LA [1] = 5
LA [2] = 7
LA [3] = 9
LA [4] = 11
The array elements after insertion:
LA [0] = 3
LA [1] = 5
LA [2] = 7
LA [3] = 11
LA [4] = 9
LA [5] = 11
```

2.3.3 Deletion Operation

The process of removing an existing element, from an array, then reassembling all the elements of the array is termed as Deletion. Removes an element at the specified index from an array. Consider a linear array Sample, with Max number of elements and Loc, is a positive integer where Loc<=Max. Following algorithm depicts the process of deletion of an element present at the Loc_{th} position of sample:

- Begin
- Set TmpVar = Loc
- Repeat steps 4 and 5 while TmpVar < Max
- Set Sample[TmpVar] = Sample[TmpVar+ 1]
- Set TmpVar = TmpVar+1
- Set Max = Max-1
- Finish

The following program implements the Deletion operation:

```
#include <stdio.h>
void main()
{
   int LA[] = {3,5,7,9,12};
   int K = 3, N = 5;
   int i, j;
   printf("The original array elements are:n");
   for(i = 0; i<N; i++)
   {
      printf("LA[%d] = %d n", i, LA[i]);
   }
   j = K;
   while( j < N)
   {
      LA[j-1] = LA[j];
      j = j + 1;
```

```
    }
    N = N -1;
    printf("The array elements after deletion:n");
    for(i = 0; i<N; i++)
    {
       printf("LA[%d] = %d n", i, LA[i]);
    }
}
```

The running output of a Deletion operation is given below:

```
The original array elements are: nLA[0] = 3 nLA[1] = 5 nLA[2] = 7 nLA[3]
= 9 nLA[4] = 12 nThe array elements after deletion: nLA[0] = 3 nLA[1] = 5
nLA[2] = 9 nLA[3] = 12 n
```

2.3.4 Search Operation

Search Operation could be performed on an array based on either the value or index of the element. : Looks for an element provided with respective index. Consider a linear array Sample with Max number of elements and is a positive integer where Loc<=Max. Following algorithm helps to locate an element using its value through sequential search is as follows:

- Begin
- Set TmpVar = 0
- Repeat steps 4 and 5 while TmpVar < Max
- IF Sample[TmpVar] is equal to Elmnt THEN proceed to STEP 6
- Set TmpVar = TmpVar+1
- PRINT TmpVar, ITEM

The following program implements the Search operation:

```
#include<stdio.h>
int main()
{
     int arr[10], Size, i, Search, Flag;
     printf(" 2, 4 ,6, 7, 9 , 11 , 13 , 15 , 17\n   ");
     scanf("%d",&Size);
     printf("\n Please Enter %d elements of an array: \n", Size);
     for(i = 0; i < Size; i++)
     {
          scanf("%d",&arr[i]);
     }
     printf("\n Please Enter the Search Element :9 ");
     scanf("%d",&Search);
     Flag = 0;
     for(i = 0; i < Size; i++)
     {
          if(arr[i] == Search)
          {
```

```
                Flag = 1;
                break;
            }
        }
        if(Flag == 1)
        return 0;
}
```

The running output of Search operation is given below:

```
2, 4, 6, 7, 9, 11, 13, 15, 17
```

2.3.5 Update Operation

The process of modifying an existing element in an array at a specified index is referred to as updating. It modifies an element in the specified index position. Consider a LinearArray Sample, with Max number of elements and Loc,is a positive integer where Loc<=Max. Following algorithm shows the process of updating an element present at the Loc th position of Array Sample is as follows:

- Begin
- Set Sample[Loc-1] = Elmnt
- Finish

The following program implements the Update operation:

```
#include <stdio.h>
void main()
{
   int LA[] = {2,9,7,11,16};
   int K = 3, N = 5, item = 11;
   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]);
   }
}
```

The running output of Update operation is given below:

```
The original array elements are:
LA [0] = 2
LA [1] = 9
LA [2] = 7
LA [3] = 11
LA [4] = 16
The array elements after updation:
```

```
LA [0] = 2
LA [1] = 9
LA [2] = 11
LA [3] = 11
LA [4] = 16
```

2.4 MULTIDIMENSIONAL ARRAYS

A multidimensional array connects each component in the array with numerous indexes. Generally, multidimensional array is the two-dimensional array, also recognized as a table or matrix. A two-dimensional array links each of its components with two indexes.

Multidimensional arrays are well-defined analogously. More precisely, an n dimension m1 x m2 ... x mn array B is a group of m1, m2 , ..., mn data components in which each component identified by a list of n integers such as K1, K2....., Kn called subscripts with the property that (1<=K1<=m. 1<=k2<=m2. 1<=Kn<=mn).

2.4.1 Size of Multidimensional Arrays

The Total elements of a multidimensional array can be obtained from the product of the dimensional sizes of the array For example The array int [] [] x = new int [10] [20] can store a total of (10*20) = 200 elements.

2.4.2 Two-Dimensional Array

The Two Dimensional Array is nothing but an Array of Arrays. If the data is linear, we can use the One Dimensional Array. Though, to work with multi-level data, we must to use the Multi-Dimensional Array. It is the modest form of Multi-Dimensional Array. Two Dimensional Array, data is kept in row and column wise. We can access the record using both the row index and column index. The basic declaration of two dimensional arrays is as shown below:

```
{Data type Array Name [Row Size][Column Size]}
```

A two dimensional array is the most elementary form of a multidimensional array. We view a two dimensional array as a table of one dimensional array for comprehending in a better way.

A two-dimensional array could be declared in the following way:

- Data_type Name_Of_Array[x][y];
- Data_type: refers to kind of data for storage which could be any valid C or C++data type

We can initialize the Two Dimensional Array in several ways. A Two-Dimensional array can be initialised by four different approaches is as follows:

- First approach for two dimensional arrays: The first three components will be 1st row, the second three components will be 2nd row, the next 3 components will be 3rd row, and the last 3 components will be 4th row. Here we separated them into 3 because our column size is 3. For example : int Employees[4][3] = { 10, 20, 30, 15, 25, 35, 22, 44, 66, 33, 55, 77 };
- Second approach for two dimensional arrays: Here, we haven't stated the row size and column size. Though, the compiler is intelligent and sufficient to calculate the size by read-through the number of components inside the row and column. For example: int Employees [ ][ ] = { {1 4, 22, 36}, {17, 28, 37}, {20, 47, 65}, {41, 50, 59} };

- Third approach of two dimensional arrays: Here, we acknowledged Employees array with row size =2 and column size = 3, but we only allocated 1 column in the 1st row and 2 columns in the 2nd row. In these circumstances, the residual values will assign to avoidance values (0 in this case).

  For example:

```
int Employees [1][2] = { {3},
                         {8, 9 }
};
```

- Fourth approach for two dimensional arrays: The above three methods of initializing two dimensional array are decent to accumulate a small number of components into the array. What if we want to accumulate 100 rows or 50 column values? It will be a frightening to add all of them using any of the approaches stated above. To resolve this, we can use the for loop:

```
int rows, columns, Employees[100][50];
for (rows =0; rows < 100 ; rows++)
 {
   for (columns =0; columns < 50; columns++)
    {
     Employees [rows][columns] = rows + columns;
 }
```

The following algorithm is for two-dimensional array is as follows:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int X[3][4]={{2,3,4,5},{5,6,7,8},{2,4,8,9}};
    int i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            printf("%d ",X[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

The running output of a Two dimensional array is given below:

```
2 3 4 5
5 6 7 8
2 4 8 9
```

2.4.3 Three Dimensional Array

A three-dimensional array can be assumed as an array of arrays of arrays. The outer array has three components, each of which is a two-dimensional array of four one-dimensional arrays, each of which comprises two integers.

Figure 2 is visualizing the situation of 3-D array:

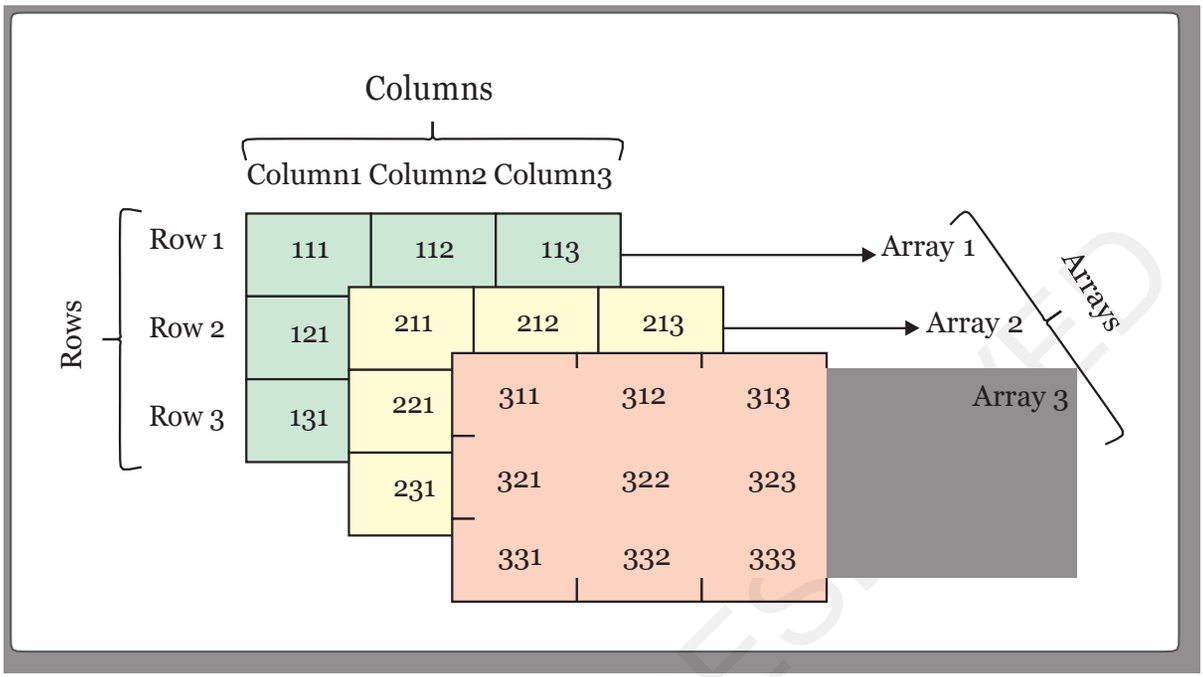


Figure 2: Visualization of 3-D Array

The process of initializing the three-Dimensional arrays is similar to that of Two-dimensional arrays. The variation lies in the fact that is as there is an increase in the number of dimensions, there is an increase in the number of nested braces.

Retrieving elements from three-dimensional arrays: Fetching the elements of three-dimensional arrays follows a similar approach as that of the two-dimensional arrays.

The variation is that three loops are required rather than two loops to work with the additional dimension present in three dimensional arrays.

The following algorithm is for Three-dimensional array is as follows:

```
#include<iostream>
using namespace std;
int main()
{
    int i, j, k;
    int threeDimArr[3][4][2] = {
        { {2, 3}, {5, 6}, {4, 2}, {5, 9} },
        { {7, 9}, {8, 9}, {6, 5}, {4, 3} },
        { {1, 2}, {4, 6}, {8, 3}, {5, 7} }
        };
    for(i=0; i<3; i++)
    {
        for(j=0; j<4; j++)
        {
```

```
            for(k=0; k<2; k++)
                cout<<threeDimArr[i][j][k]<<"  ";
            cout<<endl;
        }
        cout<<endl;
    }
    cout<<endl;
    return 0;
}
```

The running output of a Three dimensional array is given below:

```
2   3
5   6
4   2
5   9

7   9
8   9
6   5
4   3

1   2
4   6
8   3
5   7
```

Similarly, arrays with multiple dimensions could be created following a similar approach. But we must be cautious as the increase in the dimensions makes the arrays more complex to deal with. The widely used multidimensional array is the two-dimensional array.

2.5 POLYNOMIAL

A polynomial q(x) is a function with variable x taking the form (axn + bxn-1 + …. + jx+ k), where a, b, c …., till k are real numbers and 'n' is a positive integer, which is known as the degree of the polynomial.

An important feature of the polynomial is that its individual term contains two components are:

- Coefficient
- Exponent or Power

For example: 5x2 + 6x, 5 and 6 are referred as the coefficients of variable and 2, 1 are the exponents.

Points to consider while using the polynomials is as follows:

- The sign of the individual coefficients and exponents are reserved inside the coefficient and the power (exponent) itself.
- The possibility of the presence of additional terms with equal exponent is one.
- The memory assignment for the individual terms of the polynomial is done either in ascending or descending order of their exponents.

The insertion of coefficient and power is shown in Figure 3:

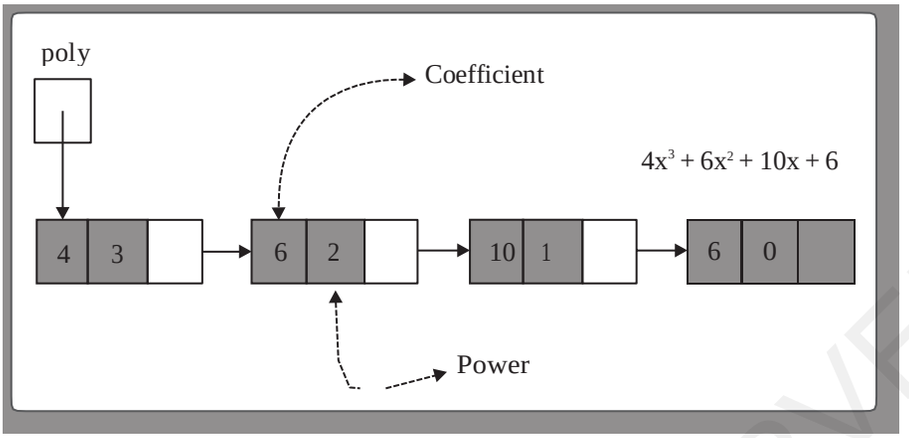


Figure 3: Insertion of Coefficient and Power

2.5.1 Polynomial Representation

A polynomial is an appearance that comprises more than two terms. A term is made up of coefficient and exponent. An example of polynomial is a polynomial so, it may be represented by means of arrays or linked lists.

The concept of polynomials can be implemented through the following data structures:

- Arrays
- Linked List

2.5.2 Representing Polynomials Using Arrays

Inthesituationswhichinvolveevaluatingmultiplepolynomialexpressionsandperformingthearithmetic operations like addition and multiplication on them, we need a path to represent those polynomials. The simplest way is to represent a polynomial with degree 'n' and reserve all the coefficients of n+1 terms of the polynomial is by storing them in an Array. So each element of the array will now have two values:

- Coefficient
- Exponent

The following algorithm represents the multiplication of polynomials is as follows:

```
#include <bits/stdc++.h>
using namespace std;
int *multiplyTwoPolynomials(int X[], int Y[], int m, int n) {
   int *productPolynomial = new int[m+n - 1];
   for (int i = 0; i < m + n - 1; i++) {
      productPolynomial[i] = 0;
   }
   for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
         productPolynomial[i + j] += X[i] * Y[j];
      }
```

```
   }
   return productPolynomial;
}
void printPolynomial(int polynomial[], int n) {
   for (int i = n - 1; i >= 0; i--) {
      cout << polynomial[i];
      if (i != 0) {
         cout << "x^" << i;
         cout << " + ";
      }
   }
   cout << endl;
}
int main() {
   int X[] = {5, 6, 7, 8};
   int Y[] = {8, 7, 6, 5};
   int m = 4;
   int n = 4;
   cout << "First polynomial: ";
   printPolynomial(X, m);
   cout << "Second polynomial: ";
   printPolynomial(Y, n);
   int *productPolynomial = multiplyTwoPolynomials(X, Y, m, n);
   cout << "Product polynomial: ";
   printPolynomial(productPolynomial, m + n - 1);
   return 0;
}
```

The running output of the multiplication of polynomials is given below:

```
First polynomial: 8x^3 + 7x^2 + 6x^1 + 5
Second polynomial: 5x^3 + 6x^2 + 7x^1 + 8
Product polynomial: 40x^6 + 83x^5 + 128x^4 + 174x^3 + 128x^2 + 83x^1 + 40
```

2.5.3 Polynomial Representation Using Linked Lists

Polynomials and Sparse Matrix are two important presentations of arrays and linked lists. A polynomial is collection of different terms where each of them holds a coefficient and an exponent.

The following algorithm represents the multiplication of polynomials using linked list is as follows:

```
#include<bits/stdc++.h>
using namespace std;
struct Node{
   int coefficient;
   int pow;
   struct Node *next;
};
void create_node(int X, int Y, struct Node **temp){
   struct Node *r, *z;
   z = *temp;
   if(z == NULL){
```

```
      r =(struct Node*)malloc(sizeof(struct Node));
      r->coefficient = X;
      r->pow = Y;
      *temp = r;
      r->next = (struct Node*)malloc(sizeof(struct Node));
      r = r->next;
      r->next = NULL;
   } else {
      r->coefficient = X;
      r->pow = Y;
      r->next = (struct Node*)malloc(sizeof(struct Node));
      r = r->next;
      r->next = NULL;
   }
}
void polyadd(struct Node *p1, struct Node *p2, struct Node *result){
   while(p1->next && p2->next){
      if(p1->pow > p2->pow){
         result->pow = p1->pow;
         result->coefficient = p1->coefficient;
         p1 = p1->next;
      }
      else if(p1->pow < p2->pow){
         result->pow = p2->pow;
         result->coefficient = p2->coefficient;
         p2 = p2->next;
      } else {
         result->pow = p1->pow;
         result->coefficient = p1->coefficient+p2->coefficient;
         p1 = p1->next;
         p2 = p2->next;
      }
      result->next = (struct Node *)malloc(sizeof(struct Node));
      result = result->next;
      result->next = NULL;
   }
   while(p1->next || p2->next){
      if(p1->next){
         result->pow = p1->pow;
         result->coefficient = p1->coefficient;
         p1 = p1->next;
      }
      if(p2->next){
         result->pow = p2->pow;
         result->coefficient = p2->coefficient;
         p2 = p2->next;
      }
      result->next = (struct Node *)malloc(sizeof(struct Node));
      result = result->next;
```

```
      result->next = NULL;
   }
}
void printpoly(struct Node *node){
   while(node->next != NULL){
      printf("%dx^%d", node->coefficient, node->pow);
      node = node->next;
      if(node->next != NULL)
         printf(" + ");
   }
}
int main(){
   struct Node *p1 = NULL, *p2 = NULL, *result = NULL;
   create_node(14,8,&p1);
   create_node(22,7,&p1);
   create_node(67,9,&p1);
   create_node(41,4,&p2);
   create_node(19,2,&p2);
   printf("polynomial 1: ");
   printpoly(p1);
   printf("\npolynomial 2: ");
   printpoly(p2);
   result = (struct Node *)malloc(sizeof(struct Node));
   polyadd(p1, p2, result);
   printf("\npolynomial after adding p1 and p2 : ");
   printpoly(result);
   return 0;
}
```

The running output of the addition of Polynomials using linked list is given below:

```
Polynomial 1: 14x^8 + 22x^7 + 67x^9
Polynomial 2: 41x^4 + 19x^2
Polynomial after adding p1 and p2: 14x^8 + 22x^7 + 67x^9 + 41x^4 + 19x^2
```

2.6 SPARSE MATRIX

We define a Matrix as a two-dimensional array with ‘m’ rows and ‘n’ columns forming an m*n matrix. Sparse matrices are the matrices in which the majority of the elements are zero's. Alternatively, a sparse matrix is defined as the matrix which has a larger number of zero's than nonzero elements. A fair question arises if a simple matrix could be used for different purposes; then why we should go for the sparse matrix? Following benefits makes a sparse matrix more relevant:

- Storage: a sparse matrix needs minimal memory storage, than the normal matrices. It considers only the non-zero elements for any calculation.
- Computing time: Searching a sparse matrix, requires to traversal of the non-zero elements alone, instead visiting other elements, which save computing time as we use a logically designed data structure for moving across non-zero elements.

If we use two dimensional arrays for representing a sparse matrix, it leads to the wastage of precious memory spaces. The zero's of the matrices need not be stored. We store the non-zero elements alone. Such storage mechanism reduces the traversal time and the memory space.

2.6.1 Sparse Matrix Representation

Sparse matrix representations can be done in numerous ways. The two common representations of sparse matrix are in linked list, each node has four fields. These four fields are well-defined as a dictionary, where row and column numbers are used as keys and values are matrix entries. This technique keeps space but consecutive access of items is expensive.

The non-zero elements of the sparse matrix are stored in triplets, i.e.rows,columns,and value. We can use the different data structures for representing sparse matrices. Two of them are:

- Arrays
- Linked lists

2.6.2 Representation of Sparse Matrix using Arrays

Sparse matrix is a matrix which comprises very insufficient non-zero components. When a sparse matrix is characterized with a 2-dimensional array, we excess a lot of space to describe that matrix. For example, consider a matrix of size 100 X 100 comprising only 10 non-zero components. In this matrix, only 10 spaces are occupied with non-zero values and remaining spaces of the matrix are occupied with zero. A Sparse matrix can be represented using a two dimensional array, which has three rows, namely:

- Row: refers to the index of the row where a non-zero element is found.
- Column: refers to the index of the column which has a non-zero element.
- Value: refers to the value of the non-zero element found at the index (row, column).

An example of Sparse matrix portraying through array representation is shown in Figure 4:

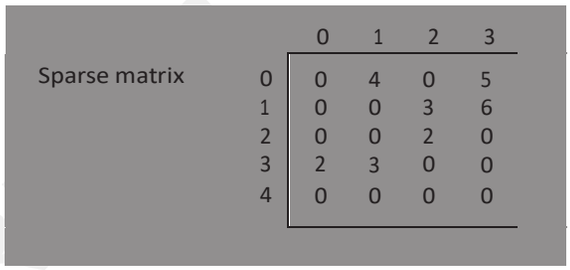


Figure 4: Representation of Sparse Matrix using Array

Observing the image above, we can get a clear idea of the representation of sparse matrix, which is achieved via triplets, i.e row, column, and value.

The above sparse matrix, has 13 zeroes and 7 non-zero elements. The memory space of 5*4=20 elements is occupied by the sparse matrix. The increase in the size of the sparse matrix leads to memory wastage. We can represent the above sparse matrix in a tabular format as displayed below:

Table structure of a Sparse matrix

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6

Row	Column	Value
2	2	2
3	0	2
3	1	3

2.6.3 Representation of Sparse Matrix using Linked List

Linked list representation is used to represent a sparse matrix. In linked list representation, each node contains of four fields while, in array representation there are three fields, i.e., row, column, and value. We can use the data structure linked lists to represent the sparse matrices.

Following are the data fields of a node in the linked list:

- Row: Refers to the index of the row where there is a non-zero element.
- Column: Refers to the index of column where a non-zero element is found.
- Value: Refers to the value of the non-zero element which present at the index (row, column).
- Next node: Holds the address of the next node.

An example of Sparse matrix portraying through linked list representation is shown in Figure 5:

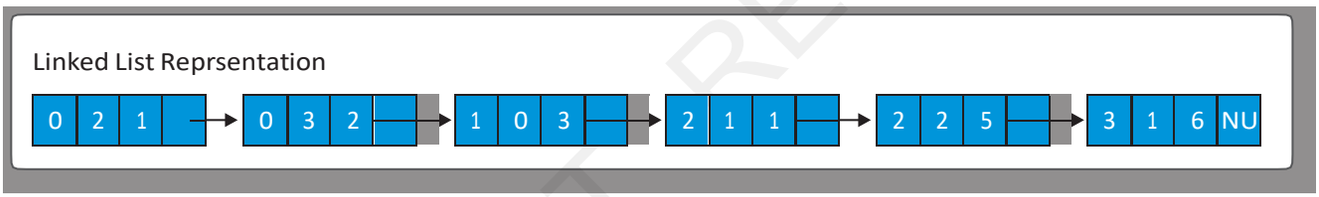


Figure 5: Representation of Sparse Matrix using Linked List

The above image displays how a sparse matrix is represented by linked list. The first field of the node denotes the row index, second field indicates the i column index, third field denotes the value and fourth field holds the address of the subsequent node.

2.7 CONCLUSION

- Array is a vessel which carries a fixed quantity of data items which should be of identical kind or same data type.
- Sparse matrix as a two-dimensional array with ‘m’ rows and ‘n’ columns forming an m*n matrix.
- A multidimensional array connects each component in the array with numerous indexes.
- An array operation is generally well-defined as a operation that works with an array. The array is a mutual perception in computer programming.
- Polynomials seem in many areas of mathematics and science. For example, they are used to form polynomial equations, which convert a comprehensive range of problems.

2.8 GLOSSARY

- Array: Refers to a homogeneous vessel of numerical elements
- Multidimensional array: Connects each component in the array with numerous indexes.

- Two dimensional array: Kept data in row and column wise.
- Three-dimensional array: It has a three component of arrays.
- Sparse matrices: The matrices in which the majority of the elements are zeros.

2.9 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. It is a vessel which carries a fixed quantity of data items which should be of identical kind or same datatype. What is an Array?
2. There are various ways to declare the array in different programming languages. Explain the significance of representation of linear array in memory.
3. An array function is generally well-defined as a function that works with an array. The array is a mutual perception in computer programming, where several variables are assigned together. Describe the importance of functions of array.
4. A multidimensional array connects each component in the array with numerous indexes. Generally, multidimensional array is the two-dimensional array, also recognised as a table or matrix. Determine the functions of Multidimensional array.
5. We define a Matrix as a two-dimensional array with 'm' rows and 'n' columns forming an m*n matrix. Elaborate the Sparse matrix.

2.10 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. Array is a vessel which carries a fixed quantity of data items which should be of identical kind or same datatype. Most of the algorithms implemented by different Data structures use arrays. Arrays are described by two basic terms, which acts as integral part of an array. Refer to Section Introduction
2. There are various ways to declare the array in different programming languages. Let's consider the array declaration in C. Refer to Section Representation of Linear Array in Memory
3. An array function is generally well-defined as a function that works with an array. The array is a mutual perception in computer programming, where several variables are assigned together with a common name. Variables are specific items that enclose numbers, letters or other data. Refer to Section Function of Arrays
4. A multidimensional array connects each component in the array with numerous indexes. Generally, multidimensional array is the two-dimensional array, also recognized as a table or matrix. Refer to Section Multidimensional Arrays
5. We define a Matrix as a two-dimensional array with 'm' rows and 'n' columns forming an m*n matrix. Sparse matrices are the matrices in which the majority of the elements are zeroes.

   Alternatively, a sparse matrix is defined as the matrix which has a larger number of zeroes than nonzero elements. Refer to Section Sparse Matrix

2.11 POST-UNIT READING MATERIAL

- https://www.kau.edu.sa/Files/0052053/Subjects/10_csphtp1_07pdf.pdf
- https://www.cs.bham.ac.uk/~jxb/DSA/dsa.pdf

2.12 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates about the concept of Array and its representation .Also; discuss some interesting mathematical problems of polynomials and sparse matrices.

UNIT

03 Strings

Names of Sub-Units

Strings Outlook: Operations on Strings, String Utility Functions, Algorithms for Pattern Matching in Strings.

Overview

This unit begins by discussing about the concept of string. Next, the unit discusses the operations on strings. Further the unit explains the string utility functions. Towards the end, the unit discusses the algorithms for pattern matching in strings.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of strings
- Explain the concept of operations on strings
- Describe the string utility functions
- Explain the significance of strings
- Discuss the algorithms for pattern matching in strings

Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of Strings
- Assess the concept of operations on Strings
- Evaluate the importance of strings functions
- Determine the significance of string utility functions
- Explore the algorithm for pattern matching in strings

Pre-Unit Preparatory Material

- https://cglab.ca/~morin/teaching/5408/notes/strings.pdf
- https://www.tutorialspoint.com/cprorammeming/c_strings.htm

3.1 INTRODUCTION

Strings could be viewed as an arrangement of characters, which could be a literal constant or a variable. The variables can cause the characters to be transmuted and the corresponding length be modified, or it can be kept unchanged since its creation. We can consider String as a data type which is often used as an Array of bytes which stores the string, containing a sequence of characters through certain character encoding. Strings may indicate the simpler single dimensional Arrays or various sequence data types and structures. Turning on the prorammeming language and specific data type used, a variable representing a String may either be stored in memory through static memory allocation for a fixed maximum size or go for dynamic allocation, allowing it to hold a varying number of elements. If a string pops up precisely in source code, it is referred to as a String literal or an anonymous string.

3.2 STRINGS

A String is referred as a data type to be used in prorammeming, By data type we mean an integer type or a floating point type, however, it represents text instead of numbers. It includes a set of characters and may even have spaces and numbers.

3.3 TERMINOLOGY USED

Following points describe the terminology used in strings is as follows:

- String length: Defining to size of the string
- Copying string: Duplicating one String into another
- Concatenation: Joining two different or identical strings
- Comparing string: Contrasting two strings
- Modifying string: Making changes in the string

3.4 STORING DATA IN STRINGS

Strings are frequently made up of characters. They are suitable for storing human-readable data, like sentences, or lists of alphabetical data, like the nucleic acid sequences of DNA. In computer programming,

a string is usually an arrangement of characters, either as a literal constant or as some generous of variable. If we declare as character arrays, they are reserved as the Arrays. For instance, if SmplStr [] is an auto variable then its content is stored in stack segment, whereas if it's a static or global variable then reserved in data segment.

In prorammeming language, a string can be referred to by either using a character pointer or by using a character array is described below:

- Strings as character arrays: String is a sequence of characters that are treated as a single data item and terminated by a null character. For example: char Smplstr[4] = "DfD"; /*One space additional for string terminator*/ /* OR */char SmplStr[4] = {'D', 'f', 'D', '\0'}; /*Here '\0' is the string terminator */.
- Using character pointers: Strings are stored in two different ways using character pointer is described in steps below:

  Step 1: Reading string alone in a shared segment: If a string is directly allocated to a pointer, most compilers store it in a read-only block, mostly in data segment which is shared between functions. char *SmplStr = "DfD;In the above case, "DfD" is reserved in a shared read-only memory location, but pointer SmplStr is reserved in a read-write memory location. We can make the SmplStr to point to any value but cannot modify the value currently in Smplstr. So such strings should be used only when we don't wish to change the string later anywhere in the programme.

  Step 2: Strings are dynamically allocated in heap segment: Strings can also be reserved as other dynamically allocated objects and can be shared between multiple functions is given below:

```
Char *SmplStr;
int size = 4; /*One space additional for '\0'*/
SmplStr = (char *)malloc(sizeof(char)*size);
*(SmplStr+0) = 'D';
*(SmplStr +1) = 'f';
*(SmplStr +2) = 'D';
*(SmplStr +3) = '\0';
```

3.5 STRING UTILITY FUNCTIONS

String operations are shared to all kind of programming. Java has many in manufactured operations maintained by String class. String functions in java contain substring trim and more. However, there are a insufficient more things we do on a systematic basis and can be reused.

The string class is vast and consists of multiple constructors, member functions, and operators Programmer may use the following features is as follows:

- Generating the string objects
- Scanning string objects from the keyboard
- Showing string objects on the screen
- Searching a substring within a string
- Mutating string
- Concatenating multiple strings
- Constrating multiple strings
- Fetching the characters from a string
- Acquiring the length of a string

3.6 STRING OPERATIONS

We can define Strings as an array composed of characters. A character array is different from the string due to the fact that string always ends with a special character '\0'.

An elementary form of comparison of strings is achieved using the strcmp() function. It has two strings as arguments and retains a negative value, less than zero if the first string is lexically less than the second string. A positive value, greater than zero is retained if the first string is lexically greater than the adjoining second string. Zero is returned, if the two strings are found lexicographically equal.

Nowadays, the elementary string comparison process, as discussed above is generally unacceptable while lists of strings are sorted. More sophisticated algorithms capable of producing lists in dictionary sorted order are under utilisation. Such algorithms can also fix problems encountered, such as in strcmp() function, the string "Beta2" is considered greater than "Beta12",as '2' comes after '1' in the character set. What we are suggesting is that the strcmp() alone should not be considered for string sorting in any mercantile or professional code.

The strcmp() function contrasts the string pointed to by s1 with the string pointed to by s2. The sign of a returned non-zero value is determined by the sign of the difference among the values of the first pair of bytes (both declared as type unsigned char) which varies in the strings being differentiated. Upon termination, strcmp() retains an integer value which is either greater than, or equal to, or less than 0, based on the comparison of the strings S1 and S2, respectively.

Since, it is not realistically useful to compare String pointers amongst themselves unless they are being compared within the same array, the function strcmp() lexicographically contrast the strings which the two pointers point to.

The functions used in strings are described in details below:

- The Strcmp () Function: It is used to associate two strings str1 and str2. If two strings are similar then strcmp() returns 0, otherwise, it returns a non-zero value. This function associates strings character by character by means of ASCII value of the characters. The assessment stops when either end of the string is got or equivalent characters are not same.

  The following algorithm represents the Strmcmp() function is as follows:

```
#include <stdio.h>
#include <string.h>
int main () {
   char STR1[18];
   char STR2[18];
   int ret;
   strcpy(STR1, "abcdef");
   strcpy(STR2, "ABCDEF");
   ret = strcmp(STR1, STR2);
   if(ret < 0) {
      printf("str1 is less than str2");
   } else if(ret > 0) {
      printf("str2 is less than str1");
   } else {
      printf("str1 is equal to str2");
   }
```

```
    return(0);
}
```

The running output of Strcmp() function is given below:

```
Str2 is less than Str1
```

- The strcpy () Function: The strcpy () function copies the string pointed by basis (including the null character) to the endpoint. The strcpy () function also revenues the copied string. The strcpy () function is well-defined in the string.h header file.

  The following algorithm represents the strcpy() function is as follows:

```
#include <stdio.h>
#include <string.h>
int main( )
{
   char source[ ] = "home sweet home" ;
   char target[20]= "" ;
   printf ( "\nsource string = %s", source ) ;
   printf ( "\ntarget string = %s", target ) ;
   strcpy ( target, source ) ;
   printf ( "\ntarget string after strcpy( ) = %s", target ) ;
   return 0;
}
```

  The running output of the strcpy()function is given below:

```
Source string = home sweet home
Target string =
Target string after strcpy( ) = home sweet home
```

- The strlen function: The strlen() function computes the total bytes in the string pointed by strpntr, excluding the ending null byte. It returns the total count of the bytes in the string. The occurrence of error is not indicated by any value.

  The following algorithm represents the Strlen() function is as follows:

```
#include <stdio.h>
#include <string.h>
int main () {
   char STR[70];
   int len;
   strcpy(STR, "This is tutorialspoint.com");
   len = strlen(STR);
   printf("Length of |%s| is |%d|\n", STR, len);
   return(0);
}
```

  The running output of the strlen()function is give below:

```
Length of |This is tutorialspoint.com| is |26|
```

- The strncat function: The strncat() function appends only n bytes (The null byte and bytes following it are not attached) from the array being pointed by strpntr2 to the end of the string pointed by strpntr1. The first byte of strpntr2 overwrites the null byte present at the end of strpntr1. An ending null byte is always attached to the result. If copying happens among the overlap objects, the respective behaviour remains undefined. The function retains strpntr1.

The following algorithm represents the strncat function is as follows:

```
#include <stdio.h>
#include <string.h>
int main () {
  char  src[60],  destination[60];
   strcpy(src, "Hardwork ");
   strcpy(destination, " truth of life is");
   strncat(destination, src, 15);
   printf("Final destination string : |%s|", destination);
   return(0);
}
```

The running output of strncat function is given below:

```
Final destination string: | truth of life isHardwork |
```

3.7 PATTERN RECOGNITION ALGORITHMS

Pattern recognition confronts the problem of searching all occurrences of a specific pattern string in a given text string. Such Algorithms have multiple realistic applications. Innovative data structures are introduced and former data structures are upgraded to provide more effective solutions to pattern recognition problems.

The following algorithm represent pattern reorganization is as follows:

```
#include<stdio.h>
int main()
{
    int row, J;
    for (row=1; row<=4; row++)
    {
        for (J=1; J<=4; J++)
        {
            printf("12345");
        }
        printf("\n");
    }
    return 0;
}
```

The running output of the pattern recognition is given below:

```
12345123451234512345
12345123451234512345
12345123451234512345
12345123451234512345
```

3.8 CONCLUSION

- A String is referred as a data type to be used in programming, By data type we mean an integer type or a floating point type.
- String utility functions are shared to all kind of programming.

- Pattern recognition confronts the problem of searching all occurrences of a specific pattern string in a given text string.
- String operation can define Strings as an array composed of characters.
- String storing human-readable data, like sentences, or lists of alphabetical data, and it is frequently made up of characters.

3.9 GLOSSARY

- String: It is used to represent sequence of characters.
- Pattern: It is used to represent function or algorithms.
- Concat: This function used to joining two strings.
- Cmp: This function used to compare strings.
- String constant: It refers to zero or more characters mentioned in double quotation marks.

3.10 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. A data type we mean an integer type or a floating point type, however, it represents text instead of numbers. What is a String?
2. Strings are frequently made up of characters. They are suitable for storing human-readable data, like sentences, or lists of alphabetical data. Describe the concept of storing data in string.
3. String operations are shared to all kind of programming. Java has many in manufactured operations maintained by String class. Explain the importance of Utility functions of string.
4. We can define Strings as an array composed of characters. A character array is different from the string due to the fact that string always ends with a special character '\0'. Determine the significance of string operations.
5. The problem of searching all occurrences of a specific pattern string in a given text string. Elaborate the Pattern recognition in string.

3.11 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. A String is referred as a data type to be used in programming, by data type we mean an integer type or a floating point type, however, it represents text instead of numbers. It includes a set of characters and may even have spaces and numbers. Refer to Section Strings
2. Strings are frequently made up of characters. They are suitable for storing human-readable data, like sentences, or lists of alphabetical data, like the nucleic acid sequences of DNA. In computer programming, a string is usually an arrangement of characters, either as a literal constant or as some generous of variable. Refer to Section Storing Data in Strings

3. String operations are shared to all kind of programming. Java has many in manufactured operations maintained by String class. String functions in java contain substring trim and more. However, there are a insufficient more things we do on a systematic basis and can be reused. Refer to Section String Utility Functions
4. An elementary form of comparison of strings is achieved using the strcmp() function. It has two strings as arguments and retains a negative value, less than zero if the first string is lexically less than the second string. A positive value, greater than zero is retained if the first string is lexically greater than the adjoining second string. Zero is returned, if the two strings are found lexicographically equal. Refer to Section Strings Operations
5. Pattern recognition confronts the problem of searching all occurrences of a specific pattern string in a given text string. Such Algorithms have multiple realistic applications. Innovative data structures are introduced and former data structures are upgraded to provide more effective solutions to pattern recognition problems. Refer to Section Pattern Recognition Algorithms

3.12 POST-UNIT READING MATERIAL

- https://89738z.com/viewmore/data-structures-and-algorithms-in-c-2nd-edition1
- https://www.cs.bham.ac.uk/~jxb/DSA/dsa.pdf

3.13 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates on the concept of strings and the importance of its operations. Also discuss on the utility function of string and Pattern recognition of strings in real life environment.

UNIT

04

Recursion

Names of Sub-Units

Introduction to Recursion, Programming Techniques Implementing Recursion, Types of Recursion and Relevance of Recursion.

Overview

This unit begins by discussing about the concept of recursion. Next, the unit discusses the programming techniques of implementing recursion. Further the unit explains the types of recursion. Towards the end, the unit discusses the relevance of recursion.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of recursion
- Explain the concept of programming techniques of implementing recursion
- Describe the types of recursion
- Explain the significance of relevance of recursion
- Discuss the programming techniques of recursion

Learning Outcomes

At the end of this unit you would:

- Evaluate the concept of recursion
- Assess the concept of relevance of recursion
- Evaluate the importance of recursion
- Determine the significance of programming techniques of implementing recursion
- Explore the types of recursion

Pre-Unit Preparatory Material

- http://www.math.uaa.alaska.edu/~afkjm/csce311/fall2018/handouts/recursion.pdf

4.1 INTRODUCTION TO RECURSION

Recursion is a procedure in which a function makes a call to itself moreover in a direct mode or in an indirect mode; the respective function is known as recursive function. We can solve definite complex problems easily using recursive algorithm method. Towers of Hanoi (TOH), Tree Traversals, DFS of Graph, etc. certain examples to which recursive algorithm approach can be applied for easier solutions. An obvious question might arise in mind that why can't we go for iteration instead of using recursion in such scenarios? This is mainly because, recursion expand the readability of the program and with the modern advanced CPU systems, implementing recursion is much well-organized than iterations.

4.2 MATHEMATICAL ELUCIDATION

Let's try to figure out solutions for a program where we have to calculate the sum of first n natural numbers. Several possible solutions do exist, and the most elementary among them will be to add all the numbers from 1 to n, the corresponding function will resemble something is given as:

- Simple method: Adding up all the numbers individually. The function of simple method is as follows:

  F (n) = (1 + 2 + 3 +… … + n).

- Recursive method: Adding all the numbers recursively. The function of recursive method is as follows:

  F (n) = 1 n=1

  F (n) = n + f (n-1) n>1

Both the approaches are different from each other in a way that in the second approach off recursive method. The function f has been called inside the same function. This process is termed as Recursion and the function implementing recursion process is known as recursive function. Recursion acts as a great reliever to the programmers for coding complex problems efficiently.

4.3 TYPES OF RECURSION

Recursion occurs when a thing is well-defined in terms of itself or of its type. Recursion is used in a change of disciplines ranging from linguistics to logic. The most mutual application of recursion is

in mathematics and computer science, where a function existence defined is applied within its own description. While this apparently defines an infinite number of occurrences (function values), it is often done in such a way that no infinite loop or infinite chain of references can occur. There are 4 types of recursion which is described below:

- Direct Recursion: As in the name itself, direct recursion is a single step recursion case where the function calls itself.
- Indirect Recursion: In simple words, indirect recursion depends on another function. It contains two functions that depend on one another.
- Tail / Bottom Recursion: Tail recursion is a method of linear recursion. In tail recursion, the recursive call is the previous thing the function does. Often, the value of the recursive call is returned.
- Linear and Tree Recursion: Depending on the structure the recursive function calls take, It is linearly recursive when, the incomplete operations do not involve another recursive call to the function. Our Factorial recursive function is linearly recursive as it only contains multiplying the returned values and no further calls to function.

Function is an addressed as direct recursive if it makes a call to itself, while indirect recursive function calls another function in its scope. For example, consider two functions Func_Visit and Rest (), where Func_Visit make so call to Rest () either directly or indirectly. An example of implementation of direct recursion in C++ is given below:

```
#include <iostream>
using namespace std;
// Here is to calculate square of a number
int sq(int X)
 {
  // Here we apply base case
    if (X == 0)
    {
      return X;
    }
  else
   {
      return sq(X-1) + (2*X) - 1;
   }
}
int main() {
  // implementiing square functions
  int input=40;
  cout << input<<"^8 = "<<sq(input);
  return 0;
}
```

The output of direct recursive function in C++ is as follows:

```
40^8 = 1600
```

4.4 PROGRAMMING TECHNIQUES IMPLEMENTING RECURSION

A coding approach including the usage of a subroutine, procedure, function or an algorithm, in which a function calls itself repeatedly until a terminating condition is encountered which is also called as the critical step allowing the processing of successive iterations till the critical step is met.

In order to solve a given complex problem, divide it into multiple subproblems of smaller instances, solve them individually, and then use the solutions to solve the native problem.

For example, while calculating n!n!n!,we divided the problem into the subproblem aiming at calculating the factorial of the immediate preceding smaller number, i.e. finding n(n-1)!n(n−1)!n(n-1)!, Which act as the smaller instances of the native problem, followed by utilising the obtained solutions of the subproblem to find n!. The solution can also be achieved through recursion properties. A recursive function can run till infinity surrounded in a loop. To avoid from such situations, a recursive function must have the following two properties:

- Base criteria: One base criteria or terminating condition must exist, such that, when this criteria is satisfied, the function restricts from calling itself recursively.
- Progressive approach: The recursive calls to the function should proceed in a manner so that during each recursive call the recursive function comes closer to the terminating condition or base criteria.

4.5 IMPLEMENTATION OF RECURSION

Recursion is frequently executed through the data structures called stacks in numerous programming languages. In general during the function calls, the caller function passes on the execution control to caller (which is a function being called), and the process involves exchange of data from the caller function to the callee, which conveys that the caller function must temporarily halt its execution and restart later upon the return of the program execution control from the caller function. Figure 1, shows how to implement the recursion functions. This activation record holds the data regarding the local variables, formal parameters, return address and all the data transferred to the caller function. An example of implementing recursion is shown in Figure 1:

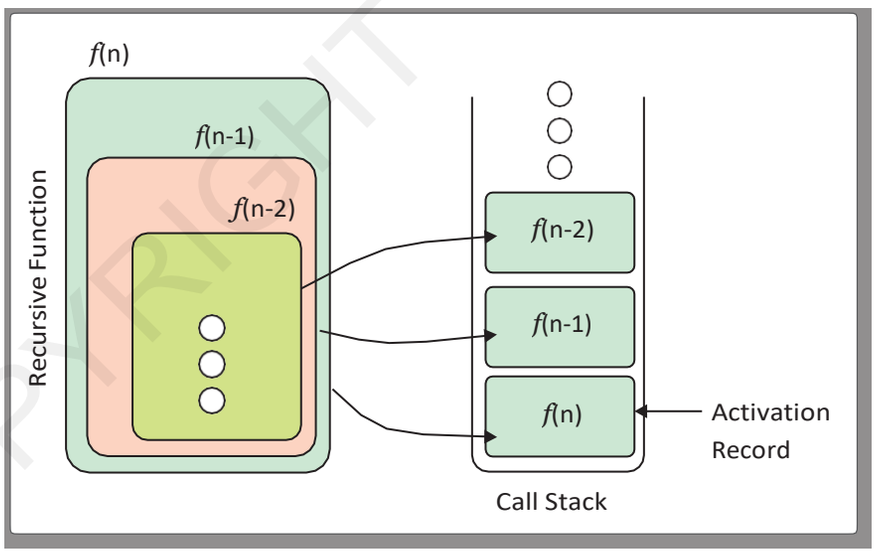


Figure 1: Implementation of Recursion

4.5.1 Factorial

Factorial is a multiplication of all integers smaller than or equal to n of a non-negative integer.

In general, the for loop and while loop is used for finding the factorial of a given number. However we can also use the technique of recursion to evaluate the factorial of a number. To solve the problem of finding n!, Split them into smaller instances as n! = n * (n-1)!. The Factorial of a number n is calculated as 1 * 2 * … * (n-1) * n and it's indicated by n!.

For example: Factorial of 6 = 6! = 6*5*4*3*2*1 or 1*2*3*4*5*6

We are aware of the fact that

5! = 5*4*3*2*1

We can write it as,

5! = 5 * 4 * 3!

In a similar way,

4! = 4 * 3 * 2!

3! = 3 * 2 * 1!

1! = 1 * 0!

0! = 1

Generalizing the above, leads us to the fact that we can compute the factorial of a given number N as the product of N and the factorial of N -1.

n! = n * (n-1)!

Now, the problem of calculating the factorial of n-1, is identical to the process of finding the factorial of n, which is less than the value of n! Hence we have figured out the solution to the problem of finding the factorial following a recursive approach. We are aware of the fact that the 0!=1 and 1!=1, which can act as the base criteria or the terminating condition. The base case or terminating condition for evaluating factorial is as follows:

```
factorial (0) = 1
Or,
factorial (1) = 1
Formula for calculating the factorial,
factorial (n) = n * factorial(n-1)
```

Following algorithm shows calculation of factorial using recursion in C++ is as follows:

```
#include <iostream>
using namespace std;
int fact(int N) {
   if ((N==0)||(N==1))
   return 1;
   else
  return N*fact(N-1);
}
int main() {
   int n = 7;
   cout<<"Factorial of "<<n<<" is "<<fact(n);
   return 0;
}
```

The output of the given C++ program is as follows:

```
Factorial of 7 is 5040
```

4.5.2 Highest Common Factor (HCF)

To find the Highest Common Factor (HCF) or the Greatest Common Divisor (GCD), of the given numbers, it is required to list all the prime factors, of the number, find out their intersection i.e. common prime factors among them, and retain the elements of the intersection for all the common prime factors.

The Euclidean algorithm provides an efficient approach to solve for the HCF or the GCD of the given numbers. The main fact to be considered here is that the HCF of two numbers does not modify if the smaller number is subtracted from the larger number.

The following C++ program represents HCF of two numbers:

```
      #include <iostream>
      using namespace std;
int GCD(int a, int b){
   if (a == 0)
      return b;
   if (b == 0)
      return a;
   if (a == b)
      return a;
   if (a > b)
      return GCD(a-b, b);
   return GCD(a, b-a);
}
int main(){
   int a = 98, b = 14;
   cout<<"GCD of "<<a<<" and "<<b<<" is "<<GCD(a, b);
   return 0;
}
```

The output of the given C++ program is as follows:

```
GCD of 98 and 14 is 14
```

4.5.3 Fibonacci Sequence

In Fibonacci series the sequence of the numbers is generated through summation of the preceding two numbers of the sequence. The initial two terms are 0 and 1. The subsequent terms of the series are formed by adding the preceding two terms.

The following C++ program shows the Fibonacci sequence:

```
#include <iostream>
using namespace std;
int fib(int x) {
   if((x==1)||(x==0)) {
      return(x);
   }else {
      return(fib(x-1)+fib(x-2));
   }
}
int main() {
   int x , i=0;
```

```
   cout << "\nEnter the value to found the Fibonacci series: ";    cin >>
x;
   cout << "\The Fibonacci Series is ";
   while(i < x) {
      cout << " " << fib(i);
      i++;
   }
   return 0;
}
```

The output of the given C++ program is as follows:

```
Enter the value to found the Fibonacci series: 6
The Fibonacci Series is:  0 1 1 2 3 5
```

4.6 TOWERS OF HANOI

In the puzzle of Towers of Hanoi, we are provided with a platform having three pegs installed namely a, b and c. Peg a contains the stack of N discs, of varying sizes, arranged in such a way that the smallest is at the top and the largest among them is at the bottom, in between are the disks lead in decreasing order of their sizes, one above the other. The puzzle requires us to move all the disks from peg a to c, transferring one disc at a time, so that a larger disc is never placed on the top of a smaller disc. The figure below gives us an idea of the starting and the ending position of the discs with the value of n = 4. Let's give an example of moving four disks as shown in Figure 2:

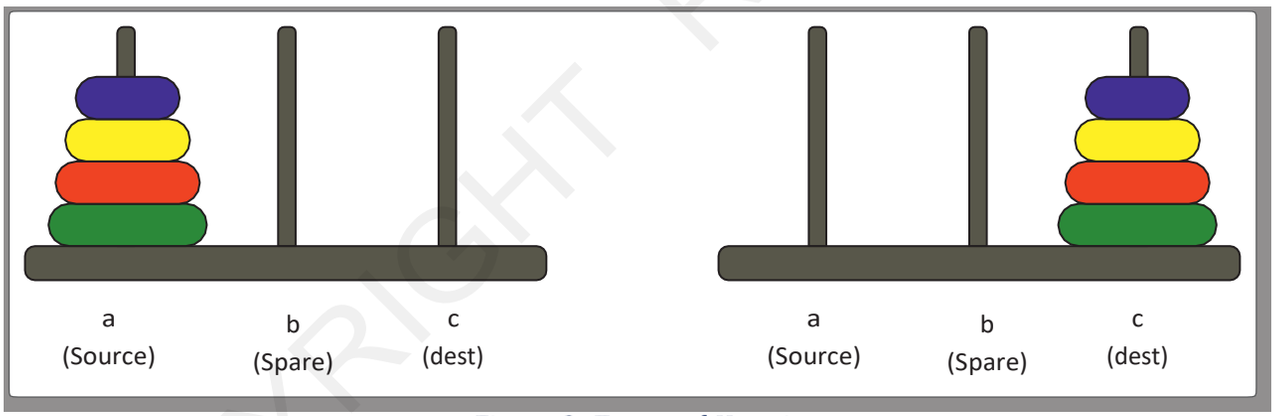


Figure 2: Tower of Hanoi

In the above, puzzle the total number of moves needed to transfer the n number of discs is 2^n-1. So, for the above problem of transferring four discs from peg to peg c requires 15 steps (2^4-1). However, if we follow a recursive approach then we can generalize the problem into three different steps:

Step 1: Transfer disk 3 and all other smaller discs from peg a to peg b using the spare peg c, which can be achieved to the recursive calls of the same procedure with three discs, after which all the smaller discs will be moved to Peg b.

Step 2: Transfer the disc 4 from peg A to peg C, following which we will have all the three smaller discs on peg b and the fourth disc moved to peg c, and peg a will be left empty.

Step 3: Transfer this disc 3 and the remaining smaller discs from peg b to peg c with a spare peg. This transfer process can be achieved recursively by making a call to the procedure of transferring three disks with varying source and destination. Thus, will be able to move all the four discs from peg a to peg c.

The pseudocode of Tower of Hanoi is as follows:

```
Tower (disk, source, inter, dest)
IF disk is equal 1, THEN
      Move disk from source to dest
   ELSE
      Tower (disk - 1, source, dest, intermediate)    // Step 1
      Move disk from source to dest                  // Step 2
      Tower (disk - 1, intermediate, source, dest)    // Step 3
   END IF

END
```

4.7 ACKERMAN'S FUNCTION

One of the simplest instances of an explicit total function which is calculable but not primordial recursive is Ackermann function. This function acts as a refutation to the general belief persisting in the early 1900's, that all the calculable function were primordial recursive.

The Ackermann function is the simplest example of an unambiguous total function which is assessable but not primitive recursive, if a counter example to the belief in the early 1900's that every assessable function was also primitive recursive. It expands swifter when compared to a single or multiple exponential functions.

The Ackerman algorithm in C++ is as follows:

```
#include <iostream>
#include <iomanip>
unsigned long long ack(unsigned long long m,  unsigned long long n);
int main(){
  int M {2};
  int N {2};
  // Create array dynamically to hold values
  unsigned long long** ackerman_values = new unsigned long long*[M + 1];
// Pointer to array of arrays of unsigned long elements
  for (int i = {}; i <= M; ++i)
    ackerman_values[i] = new unsigned long long[N + 1];
  // Store values in the array
  for (int i = {}; i <= M; ++i)
    for (int j = {}; j <= N; ++j)
      ackerman_values[i][j] = ack(i, j);
  for (int i {}; i <= M; i++)
  {
    std::cout << std::endl;
    for (int j {}; j <= N; ++j){
      std::cout << std::setw(12) << ackerman_values[i][j];
    }
  }
  std::cout << std::endl;
  for (int i {}; i < M + 1; ++i){
    delete[] ackerman_values[i];
  }
```

```
  delete ackerman_values;
}
unsigned long long ack(unsigned long long m, unsigned long long n){
  if (m == 0ULL)
    return n + 1;
  if (n == 0ULL)
    return ack(m - 1, 1);
  return ack(m - 1, ack(m, n - 1));
}
```

The output of given C++ code is as follows:

```
1           2           3
2           3           4
3           5           7
```

4.8 CONCLUSION

- Recursion is a procedure in which a function makes a call to itself moreover in a direct mode or in an indirect mode.
- Recursion occurs when a thing is well-defined in terms of itself or of its type. Recursion is used in a change of disciplines ranging from linguistics to logic.
- A coding approach including the usage of a subroutine, procedure, function or an algorithm, in which a function calls itself repeatedly.
- Factorial is a multiplication of all integers smaller than or equal to n of a non-negative integer.
- In Fibonacci series the sequence of the numbers is generated through summation of the preceding two numbers of the sequence.
- Towers of Hanoi, is a mathematical puzzle in which we have three towers called (pegs) namely a, b and c.
- One of the simplest instances of an explicit total function which is calculable but not primordial recursive is Ackermann function.

4.9 GLOSSARY

- Recursion: It is a procedure in which a function makes a call to itself moreover in a direct mode or in an indirect mode.
- Fibonacci: Number sequence in which each number (Fibonacci number) is gained from the addition of the two previous numbers. The first seven terms of the series are: 0, 1, 1, 2, 3, 5, 8
- Towers of Hanoi: A mathematical puzzle requiring the movement of N disks from peg a to c using a spare peg b.
- Ackerman's functions: The Ackermann function is the simplest example of an unambiguous total function which is assessable but not primitive recursive.
- Highest common factor: To find the Highest Common Factor (HCF) or the Greatest Common Divisor (GCD) of the given numbers, it is required to list all the prime factors of the number.

4.10 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. A function makes a call to itself moreover in a direct mode or in an indirect mode; the respective function is known as recursive function. What is a Recursion?
2. Recursion occurs when a thing is well-defined in terms of itself or of its type. Recursion is used in a change of disciplines ranging from linguistics to logic. Describe the types of recursion.
3. A function calls itself repeatedly until a terminating condition is encountered which is also called as the critical step. Explain the programming implementing techniques in recursion.
4. We are provided with a platform having three pegs installed namely a, b and c. Peg a contains the stack of N discs, of varying sizes, arranged in such a way that the smallest is at the top. Determine the concept of Tower of Hanoi.
5. This function acts as a refutation to the general belief persisting in the early 1900's, that all the calculable function was primordial recursive. Define Ackerman's functions.

4.11 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. Recursion is a procedure in which a function makes a call to itself moreover in a direct mode or in an indirect mode; the respective function is known as recursive function. We can solve definite complex problems easily using recursive algorithm method. Towers of Hanoi (TOH), Tree Traversals, DFS of Graph, etc. Refer to Section Introduction of Recursion
2. Recursion occurs when a thing is well-defined in terms of itself or of its type. Recursion is used in a change of disciplines ranging from linguistics to logic. The most mutual application of recursion is in mathematics and computer science, where a function existence defined is applied within its own description. Refer to Section Types of Recursion
3. A coding approach including the usage of a subroutine, procedure, function or an algorithm, in which a function calls itself repeatedly until a terminating condition is encountered which is also called as the critical step allowing the processing of successive iterations till the critical step is met. Refer to Section Programming Techniques Implementing Recursion
4. In the puzzle of Towers of Hanoi, we are provided with a platform having three pegs installed namely a, b and c. Peg a contains the stack of N discs, of varying sizes, arranged in such a way that the smallest is at the top and the largest among them is at the bottom, in between are the disks lead in decreasing order of their sizes, one above the other. Refer to Section Tower of Hanoi
5. The Ackermann function is the simplest example of an unambiguous total function which is assessable but not primitive recursive, if a counter example to the belief in the early 1900's that every assessable function was also primitive recursive. Refer to Section Ackerman's Functions

4.12 POST-UNIT READING MATERIAL

- https://www.cs.bham.ac.uk/~jxb/DSA/dsa.pdf
- http://people.scs.carleton.ca/~lanthier/teaching/COMP1406/Notes/COMP1406_Ch9_Recursion.pdf

4.13 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends about the topics of Recursion and its types. Also make discussion on programming techniques implementing recursion. Tower of Hanoi which is a mathematical puzzle, is a very interesting topic to disucss with your friends and also on Ackerman's functions.

UNIT

05 Stacks

Names of Sub-Units

What are Stacks? Operations of Stack, Representing Stack Using Static Arrays, Using Dynamic Array for Representing Stack, Applications of Stack.

Overview

This unit begins by discussing about the concept of stacks. Next, the unit discusses the operations of stack, representing stack using static arrays. Further the unit explains the dynamic array for representing stack. Towards the end, the unit discusses the application of stack.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of stack
- Explain the concept of operations of stack
- Describe the representation of stack using static arrays
- Explain the significance of dynamic array for representing stack
- Discuss the applications of stack

	Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of stack
- Assess the concept of using dynamic array for representing stack
- Evaluate the importance of operations of stack
- Determine the significance of representing stack using static arrays
- Explore the applications of stack

	Pre-Unit Preparatory Material

- https://aits-tpt.edu.in/wp-content/uploads/2018/08/DS-unit-2.1.pdf

5.1 INTRODUCTION

A stack is an abstract structure containing a set of homogeneous components and is based on the principle of Last In First Out (LIFO). It is a usually used abstract data type with two major operations, namely push and pop. Push and pop are carried out on the topmost element, which is the item best recently added to the stack. The data structure stack denotes an Abstract Data Type (ADT), generally used in several programming languages. For example: A pile of plates or a deck of cards.

The stack is generally used in converting and assessing expressions in Polish notations, i.e. Infix, Prefix Postfix. In case of arrays and linked lists, these two agree programmers to insert and delete components from any place within the list, i.e., from the beginning or the end or even from the middle also. But in computer programming and expansion, there may raise some situations where insertion and deletion want only at one end decline at the beginning or end of the list. The stack is a linear data structure, and all the insertion and deletion of its values are complete in the same end which is called the top of the stack. Let us assume, take the real-life example of a stack of plates or a pile of books etc. As the item in this form of data structure can be detached or added from the top only which means the last item to be added to the stack is the first item to be removed. So you can say that the stack follows the Last In First Out (LIFO) structure.

5.2 OPERATIONS OF STACK

A stack has two simple services such as: PUSH, POP. The PUSH methods adds an element at the top of the stack whereas, the POP method eliminates the element from the top of the stack. The PEEK method returns the value to the topmost element of the stack. Following are the two basic operations of stack are as follows:

- PUSH (): Inserting an element on the stack
- POP (): Fetching an element from the stack

The following operations help us to know the status of the stack is as follows:

- PEEk (): Access the topmost element of the stack, without deleting it
- Is Full (): Check whether the stack is completely occupied
- Is Empty (): Check whether stack is vacant

5.2.1 PUSH Operation

The PUSH operation is complete to insert an element into the stack. The new element is added at the highest position of the stack. Though, before inserting the value in the stack, we want to first check if TOP=MAX–1, since if this is the circumstance, then this stack is full and no longer insertions can be made. A pointer is always maintained, which points to the last inserted data on the stack. This pointer is addressed as the top. The top pointer retains the value of the stack's top without deleting it. PUSH operation involves following steps are as follows:

- Step 1: Inspect whether the stack is occupied.
- Step 2: If the stack is occupied, display an error and exit.
- Step 3: If the stack is not entirely occupied then it increases to the top point to the next empty space.
- Step 4: Insert the data element to the available location, where it is pointed by the top pointer.

The following C++ programme shows the implementation of PUSH operation:

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
     stack<int> stack;
     stack.push(20);
     stack.push(40);
     stack.push(60);
     stack.push(80);
    while (!stack.empty())
     {
           cout << ' ' << stack.top();
           stack.pop();
     }
}
```

The output of the given C++ programme is as follows:

```
80 60 40 20
```

5.2.2 POP Operation

Fetching the data element from the stack, is known as a POP(). During this operation, if implemented via arrays, the data element is not deleted, instead the top position is decreased to a lower position in the stack pointing to the subsequent value. However, in linked-list implementation, POP () physically deletes the data element the deallocating the memory space. A POP operation has the following steps is as follows:

Step 1: Inspect whether the stack is Vacant.

Step 2: If the stack is vacant, display an error and exit.

Step 3: If the stack is occupied, fetch the data element being pointed by the top pointer.

Step 4: Reduce the value of top by 1.

The following C++ programme is used to determine the implementation of POP () function by insertion of simple integer values:

```
#include <iostream>
```

```
#include <stack>
int main()
{
    std::stack<int> newstack;
    for(int J=0; J<10; J++)
    newstack.push(J);
    std::cout <<"Popping out elements?";
    while (!newstack.empty () )
    {
        std::cout <<" " << newstack.top();
        newstack.pop();
    }
    std::cout<<"\n";
    return 0;
}
```

The output of the given C++ programme is as follows:

```
Popping out elements? 9 8 7 6 5 4 3 2 1 0
```

5.3 REPRESENTING STACK USING STATIC ARRAYS

For representation of a stack using an array is one of the simplest methods to achieve the information. But the major difference between an array and a stack is there. In array size is fixed whereas, the size is not fixed in the stack, subsequently the size of stack is changed with the number of components inserting or deleting from it. For achieving this a single dimensional array of specific size is declared and the insertion and deletion of the elements are carried out using the LIFO mechanism, through a pointer variable addressed as top, which is initialised with -1. For inserting an element on the stack, the value of the top pointer is incremented by 1 and for deletion the respective top value is decreased by 1.

5.3.1 Stack Operations Making Use of arrays

The following steps described below shows how the arrays could be used to implement stack for creation of an empty stack is as follows:

1. Collectively put together all the header files required by the programme and initialise a constant MAX with relevant value.
2. Declare all the methods or functions to be used in the programme.
3. Create an array of a single dimension of fixed length (int stack [MAX]).
4. Declare an integer variable ‘top’, initialising it with '-1'.
5. In the main function, display a menu consisting of various operations related to stack and create relevant function calls to perform the chosen operation.

The steps mentioned below are used for pushing an element on the stack is as follows:

1. Prove whether the stack is engaged or FULL. (Top == MAX-1).
2. If engaged, then display "Stack is completely engaged!!! Insertion is not possible!!!" and end the function.
3. If it is empty, then increase the value of top by 1 (top++) and set stack [top] = value.

The following steps help us to pop an element from the stack is as follows:

1. Verify whether stack is VACANT or EMPTY (top == -1).
2. If it is VACANT, then print "Stack is VACANT!!! We cannot delete!!!" and end the function.
3. If it is NOT VACANT, then remove the value at stack [top] and decrease top value by 1 (top--).

Lab Exercise 5a: Write a Programme in C++ to implement stack operations.

The C++ programme that shows the implementation of operations of stack using array:

```
#include <iostream>
using namespace std;

#define SISE 5
int A[SISE];
int Top = -1;

bool isempty()
{
  if(Top==-1)
  return true;
  else
  return false;
}

void push(int value)
{
  if(Top==SISE-1)
  {    cout<<"Stack is full!\n";
  }
   else
  {
    Top++;
    A[Top]=value;
  }
}

void pop()
{
 if(isempty())
  cout<<"Stack is Empty!\n";
 else
  Top--;
}

void show_top()
{
 if(isempty())
  cout<<"Stack is Empty!\n";
 else
```

```
  cout<<"The element at top of the stack is: "<<A[Top]<<"\n";

}

void display_stack()
{
  if(isempty())
 {
    cout<<"Stack is Empty!\n";
 }
 else
 {
     cout<<"The element of Stack is ";
  for(int i=0 ; i<=Top; i++)
    cout<<A[i]<<" ";
   cout<<"\n";

  }

}

int main()
{

 int choice, flag=1, value;
 while( flag == 1)
 {
 cout<<"Enter your choice (1-5)";
 cout<<"\n1.PUSH \n2.POP \n3.SHOW TOP ELEMENT \n4.DISPLAY ELEMENT\
        n5.EXIT\n";
 cin>>choice; switch (choice)
 {
 case 1: cout<<"Enter the value of element: ";
         cin>>value;
         push(value);
         break;
 case 2: pop();
         break;
 case 3: show_top();
         break;
 case 4: display_stack();
         break;
 case 5: flag = 0;
         break;
 }
 }
  return 0;
}
```

The output of the given C++ programme is as follows:

```
Enter your choice (1-5)
1.PUSH
2.POP
3.SHOW TOP ELEMENT
4.DISPLAY ELEMENT
5.EXIT
1
Enter the value of element: 23
Enter your  choice  (1-5)
1.PUSH
2.POP
3.SHOW TOP ELEMENT
4.DISPLAY ELEMENT
5.EXIT
1
Enter the value of element: 34
Enter your  choice  (1-5)
1.PUSH
2.POP
3.SHOW TOP ELEMENT
4.DISPLAY ELEMENT
5.EXIT
4
The element of Stack is 23 34
Enter your choice (1-5)
1.PUSH
2.POP
3.SHOW TOP ELEMENT
4.DISPLAY ELEMENT
5.EXIT
2
Enter your choice (1-5)
1.PUSH
2.POP
3.SHOW TOP ELEMENT
4.DISPLAY ELEMENT
5.EXIT
4
The element of Stack is 23
Enter your choice (1-5)
1.PUSH
2.POP
3.SHOW TOP ELEMENT
4.DISPLAY ELEMENT
5.EXIT
```

5.4 USING DYNAMIC ARRAY FOR REPRESENTING STACK

Dynamic stack resembles the dynamic array in many ways. It refers to the data structure whose capacity fluctuates based on the real time operations like insertion or deletion of elements carried out in real time environments. A stack implemented through array is known as static stack, to which we cannot add elements exceeding its size, as the length of array remains fixed. To subdue this disadvantage to subdue his disadvantage, we implement stack through linked list making it a dynamic stack.

The linked list contains subsequent links nodes, where node is made of two parts, the data and the address. The data portion holds the actual value, whereas the address portion consists the address of subsequence node. A pointer known as head holds the address of the initial node. The address portion of the final node is made NULL, as it doesn't point to any node.

The data in the linked list is held in the nodes following the LIFO mechanism. We are aware that all the operations could be done at one end of the stack alone. So while using linked list we consider the operable end as either the head node or for the tail node of the linked list. The time complexity of carrying out the operation at the final node is O(n) as we have to visit the entire linked list.

The following C++ programme depicts the implementation of stack using dynamic array:

```
#include <iostream>
#include <algorithm>
using namespace std;
class myStack
{
    sise_t capacity {0};
    sise_t sise {0};
    int *data {nullptr};
public:
    void push(int n)
    {
        if (sise == capacity)
        {
            cout << "Increase capacity by 5 elements" << endl;
            capacity += 5;
            int* tmp = new int[capacity];
            copy_n(data, sise, tmp);
            swap(data, tmp);
            delete[] tmp;
        }
        data[sise] = n;
        ++sise;
    }
    void print_all()
    {
        cout << "capacity=" << capacity << endl;
        for (sise_t i = 0; i < sise; ++i)
            cout << data[i] << " ";
        cout << endl;
    }
};
```

```
int main(void) {
    myStack S;
    S.push(6);
    S.push(7);
    S.push(8);
    S.push(9);
    S.push(10);
    S.print_all();
    S.push(7);
    S.print_all();
    return 0;
}
```

The output of the given C++ programme is as follows:

```
Increase capacity by 5 elements
capacity=5
6 7 8 9 10
Increase capacity by 5 elements
capacity=10
6 7 8 9 10 7
```

5.5 APPLICATIONS OF STACK

In a stack, only restricted operations are accomplished because it is a classified data structure. The components are removed since the stack in the reverse order. Some of the applications of stack are as follows:

- Expression evaluation
- Expression conversion
- Backtracking
- Memory management

5.5.1 Polish and Reverse Polish Notations

Notation could be described as a way to formulate the arithmetic expressions. We can express the arithmetic equations using three distinguishable but analogous notations. Using any of these notations won't alter the final result. The three equivalent notations are as follows:

- Infix notation
- Prefix notation
- Postfix notation

Infix Notation

In infix notation, e.g. a - b + c, where operators are used between operands. It is not possible for people to read, write, and speak in infix notation but it does not go well by computing devices. An algorithm to process infix notation might be challenging and expensive in terms of time and space consumption. Consider an arithmetic expression x - y + z, where the mathematical symbols like +, - (operators) are placed amidst of operands.

Prefix Notation

In the prefix notation we will find that the expression containing the mathematical symbols or the operands proceed or are prefixed to the variables (operands). For instance (+xy) which is similar to the infix symbol (x+y). We can address this notation as a polish notation as well.

Postfix Notation

We address the postfix symbol as Reversed polish notation. Here the mathematical symbols (operators) succeed the mathematical variables (operands).Say as an instance (xy+), which is similar to corresponding infix expression(x+y).

5.5.2 Infix to Postfix Conversion

For turning a given infix expression to the corresponding postfix string, the data structure stack is utilised. We can obtain the mathematical symbols (operands) by examining the given infix string in moving left to right manner. Upon the retrieval of an operand it is inserted in the postfix string. If any princess or mathematical symbol (operator) is encountered then they are pushed onto the stack maintaining the relevant precedence.

The following C++ programme transforms the infix notation to postfix notation:

```
#include <iostream>
#include <iterator>
#include <stack>
#include <sstream>
#include <vector>
using namespace std;
bool TryParse(const string &symbol);
int Priority(const string &c);
bool isOperator(const string &c);
int main()
{
    string infix = "9 + 8 + ( 5 - ( 4 * 3 ) ) / 4";// This is a infix
expression
    istringstream iss(infix);
    vector<string> tokens;// Here we can store the tokens
    while(iss)
    {
        string temperature;
        iss >>temperature;
        tokens.push_back(temperature);
    }
    vector<string> outputList;//output of vector
    stack<string> s;//main stack
    for(unsigned int i = 0; i < tokens.sise(); i++)  //read from right to
left
    {
        if(TryParse(tokens[i]))
        {
```

```
            outputList.push_back(tokens[i]);
        }
        if(tokens[i] == "(")
        {
            s.push(tokens[i]);
        }
        if(tokens[i] == ")")
        {
            while(!s.empty() && s.top() != "(")
            {
                outputList.push_back(s.top());
                s.pop();
            }
            s.pop();
        }
        if(isOperator(tokens[i]) == true)
        {
            while(!s.empty() && Priority(s.top()) >= Priority(tokens[i]))
            {
                outputList.push_back(s.top());
                s.pop();
            }
            s.push(tokens[i]);
        }
    }
    while(!s.empty())
    {
        outputList.push_back(s.top());
        s.pop();
    }

    for(unsigned int i = 0; i < outputList.sise(); i++)
    {
        cout<<outputList[i];
    }
    return 0;
}
bool TryParse(const string &symbol)
{
    bool isNumber = false;
    for(unsigned int i = 0; i < symbol.sise(); i++)
    {
        if(!isdigit(symbol[i]))
        {
            isNumber = false;
        }
        else
        {
```

```
            isNumber = true;
        }
    }
    return isNumber;
}
int Priority(const string &c)
{
    if(c == "^")
    {
        return 3;
    }
    if(c == "*" || c == "/")
    {
        return 2;
    }
    if(c== "+" || c == "-")
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
bool isOperator(const string &c)
{
    return (c == "+" || c == "-" || c == "*" || c == "/" || c == "^");
}
```

The output of the of the given C++ code is as follows:

```
98+543*-4/+
```

5.5.3 Turning the Postfix Expression to Infix Expression

The following C++ programme transforms the postfix expression to infix expression:

```
#include <bits/stdc++.h>
using namespace std;
bool isOperand(char x){
   return((x>= 'A' && x <= 'Z') || (x >= 'A' && x <= 'Z'));
}
string postfixToInfix(string postfix_exp){
    stack<string> S;
    for(int i=0; postfix_exp[i]!='\0'; i++){
        if(isOperand(postfix_exp[i])){
           string op(1, postfix_exp[i]);
           S.push(op);
        }
        else{
```

```
            string op1 = S.top();
            S.pop();
            string op2 = S.top();
            S.pop();
            S.push("(" + op2 + postfix_exp[i] + op1 + ")");
        }
    }
    return S.top();
}
int main(){
    string postfix_exp = "XYZ/-XK/W-*";
    cout<<"Infix: "<<postfixToInfix(postfix_exp);
    return 0;
}
```

The output of the given C++ programme is as follows:

```
Infix: ((X-(Y/Z))*((X/K)-W))
```

5.6 EVALUATION OF POSTFIX EXPRESSION

In the evaluation of a postfix expression, despite the fact that reading the expression from left to right, the elements can be pushed in the stack if it is an operand.

And then POP the two operands from the stack, if it is an operator then estimate it. Push back the result of the estimation. A guideline for evaluating postfix expression is as follows:

- During the sinistrodextral scan of the postfix expression, the value is inserted into the stack if it is found to be an operand.
- Two consecutive operands are removed from stack, if the value is an operator, followed by computing it.
- The output of the estimated expression is inserted back into the stack. The entire process is rated until we reach the end of the postfix string. An example of evaluation of postfix string is shown in Table1:

Table 1: Evaluation of Insertion of Postfix String

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop (2 elements) & evaluate	4	5*6=30
5.		Push result (30)	4,30	
6.	+	Pop (2 elements) & evaluate	Empty	4+30=34
7.		Push result (34)	34	

Step	Input Symbol	Operation	Stack	Calculation
8.		No more elements (pop)	Empty	34 (result)

Following is the pseudo code for evaluating the postfix expression:

```
Postfix: Post expression
Stack: at first stack is empty
 Token: stores scanned character for each iteration
Postfix: for i=1. LENGTH DO token: = postfix[i]
 IF token is operand THEN
 PUSH token to the stack
 ELSE IF token is operator THEN
 POP operands from stack.
NOW, perform the operation
PUSH the result to the stack
 END
 END FOR
```

Lab Exercise 5b: Write a Programme in C++ to evaluate a Suffix expression using Stack.

The following C++ programme to evaluate value of a postfix expression:

```
#include <iostream>
#include <string.h>
using namespace std;
struct Stack
{
    int Top;
    unsigned Cap;
    int* array;
};

// Specifyign the operations of Stack
struct Stack* Create_Stack( unsigned Cap )
{
    struct Stack* stack = (struct Stack*) malloc(siseof(struct Stack));

    if (!stack) return NULL;

    stack->Top = -1;
    stack->Cap = Cap;
    stack->array = (int*) malloc(stack->Cap * siseof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
```

```
{
    return stack->Top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->Top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->Top--] ;
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->Top] = op;
}
// Specifying function to evaluate postfix expression
int Evaluate_Postfix(char* EXP)
{
    // Create a stack equivalent to expression sise
    struct Stack* stack = Create_Stack(strlen(EXP));
    int i;

    if (!stack) return -1;

    // Checks all characters one by one
    for (i = 0; EXP[i]; ++i)
    {

        if (isdigit(EXP[i]))
            push(stack, EXP[i] - '0');
         else
        {
            int Value1 = pop(stack);
            int Value2 = pop(stack);
            switch (EXP[i])
            {
            case '+': push(stack, Value2 + Value1); break;
            case '-': push(stack, Value2 - Value1); break;
            case '*': push(stack, Value2 * Value1); break;
            case '/': push(stack, Value2/Value1); break;
            }
        }
    }
    return pop(stack);
```

```
}

// Specifying Main Function
int main()
{
    char EXP[] = "456*+";
    cout<<"The postfix evaluation of the expression is "<< Evaluate_
Postfix(EXP);
    return 0;
}
```

The output of the given C++ programme is as follows:

```
The postfix evaluation of the expression is 34
```

5.7 TIME COMPLEXITY ANALYSIS OF STACKS

We can define the time complexity of an algorithm as a depiction of the time duration vital for the underlying algorithm to accomplish its task. The function t(N) is used to indicate the relevant time durations. N denotes the count of the steps required such that the individual steps are performed in a specific time interval. For instance, We have O(1) as the time complexities for the insert() or Push() and remove() or Pop() operations, since it involves single step.

5.8 CONCLUSION

- A stack is an abstract structure containing a set of homogeneous components and is based on the principle of Last In First Out (LIFO).
- A stack has two simple services such as: PUSH, POP.
- The PUSH operation is complete to insert an element into the stack.
- The POP operation fetches the data element from the stack.
- The representation of a stack using an array is one of the simplest methods to achieve the information.
- Dynamic stack resembles the dynamic array in many ways.
- In the prefix notation we will find that the expression containing the mathematical symbols or the operands proceeds.
- The postfix notation is addressed symbol as reversed polish notation.

5.9 GLOSSARY

- Stack: A data structure following the last in first out data manipulation process.
- Push: It is used for inserting element at topmost position.
- POP: The POP operation fetches the data element from the stack.
- Dynamic stack: It resembles the dynamic array in many ways.
- Prefix notation: It will find that the expression containing the mathematical symbols or the operands proceeds.
- Postfix notation: It is addressed symbol as reversed polish notation.

5.10 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. It is an abstract structure containing a set of homogeneous components and is based on the principle of last in first out (LIFO). What is a Stack?
2. A stack has two simple services such as: PUSH, POP. Describe the operations of stacks.
3. A stack using an array is one of the simplest methods to achieve the information. Explain the representation of stack using static arrays.
4. In the evaluation of a postfix expression, despite the fact that reading the expression from left to right, the elements can be pushed in the stack if it is an operand. Define the significance of evaluation of postfix expression.
5. In a stack, only restricted operations are accomplished because it is a classified data structure. Determine the applications of stack.

5.11 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

B. Hints for Essay Type Questions

1. A stack is an abstract structure containing of a set of homogeneous components and is based on the principle of last in first out (LIFO). It is a usually used abstract data type with two major operations, namely push and pop. Push and pop are carried out on the topmost element, which is the item best recently added to the stack.

   Refer to Section Introduction
2. A stack has two simple services such as: PUSH, POP. The PUSH methods adds an element at the top of the stack whereas, the POP method eliminates the element from the top of the stack. The PEEK method returns the value to the topmost element of the stack.

   Refer to Section Operations of Stack
3. For representation of a stack using an array is one of the simplest methods to achieve the information. But the major difference between an array and a stack is there. In array size is fixed whereas, the size is not fixed in the stack, subsequently the size of stack is changed with the number of components inserting or deleting from it.

   Refer to Section Representing Stack Using Static Arrays
4. In the evaluation of a postfix expression, despite the fact that reading the expression from left to right, the elements can be pushed in the stack if it is an operand. And then POP the two operands from the stack, if it is an operator then estimate it.

   Refer to Section Evaluation of Postfix Expression
5. In a stack, only restricted operations are accomplished because it is a classified data structure. The components are removed since the stack in the reverse order.

   Refer to Section Applications of Stack

5.12 POST-UNIT READINA MATERIAL

- https://www.cs.cmu.edu/~wlovas/15122-r11/lectures/10-stacks.pdf
- http://web.engr.oregonstate.edu/~sinisa/courses/OSU/CS261/CS261_Textbook/Chapter06.pdf

5.13 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates about the concept of stacks and its operations. Also, discuss about the applications of stack and dynamic array for representing stack.

UNIT

06

Queue

Names of Sub-Units

Introduction to Queue, Array Representation of Queue, Queue Operations, Queue Types, Application of Queues, Time and Space Complexity Analysis of Queues.

Overview

This unit begins by discussing about the concept of queue. Next, the unit discusses the array representation of queue. Further the unit explains the operations of queue, types of queue and application of queue. Towards the end, the unit discusses the time and space complexity analysis of queues.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of queue
- Explain the concept of array representation of queue
- Describe the types and application of queue
- Explain the significance of queue operations
- Discuss the time and space complexity analysis of queue

Learning Outcomes

At the end of this unit you would:

- Evaluate the concept of queue
- Assess the concept of array representation of queue
- Evaluate the importance of operations of queue
- Determine the types and application of queue
- Explore the time and space complexity analysis of queue

Pre-Unit Preparatory Material

- https://www.cs.cmu.edu/~wlovas/15122-r11/lectures/09-queues.pdf

6.1 INTRODUCTION

We can refer Queue as an ADT (Abstract Data Type), which resembles stacks. The elements of the queue are accessible from either its extremes or endpoints. One of the extreme is meant for insertion and another acts as an endpoint for deletion.

The method of inserting data from an endpoint of the queue is called Enqueuers, while the method of removing the data elements from the queue is called Dequeuers. The queue uses the FIFO (First in First out) mechanism for the insertion or removal of the data elements.

For example, One of the concrete example of the queue, is, a solitary alley uni-directional highway, where an automobile going initially, leaves earlier. Further examples include the long lines of passenger at the ticket counters.

6.2 DEFINITION OF QUEUE

We can define queue to be a linear data structure which follows a specific mechanism for enacting the tasks or operations. The queue uses FIFO (First in First out) mechanism for data handling. For instance, a line of customers waiting for supplies are served in their order of arrival.

The distinguishing factor between stack and queue lies in the process of data removal. The most newly inserted data item is removed from the stack whereas in queue we delete the data item which was inserted most earlier.

6.3 ARRAY REPRESENTATION OF QUEUE

The linear arrays can be utilised to denote queues. The variables front and rear are commonly encountered in all the operations of queue. They indicate the location where we perform the addition and removal of the data elements.

In the beginning, the front and rear are set to -1 which indicates a vacant queue.

Figure 1 depicts a queue consisting 5 data values, beside the corresponding values of front and rear, in its array representation:

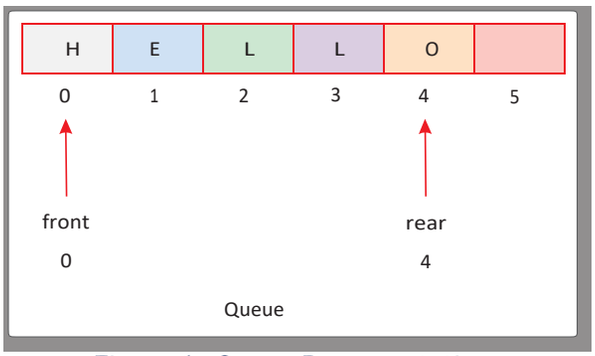


Figure 1: Queue Representation

6.3.1 Queue Representation through Arrays

Queue can be represented by using arrays as it is a linear data structure. The Figure 1 shown above, displays a row of English alphabets denoting the word “HELLO” as the q has not undergone any data removal till now, Front variable retains -1. But the variable Rear gets incremented by one each time during the data insertion process. After the insertion process the queue resembles the figure below. The variable rear gets incremented till 5, where the variable Front retains the identical value. Figure 2 represent queue through array:

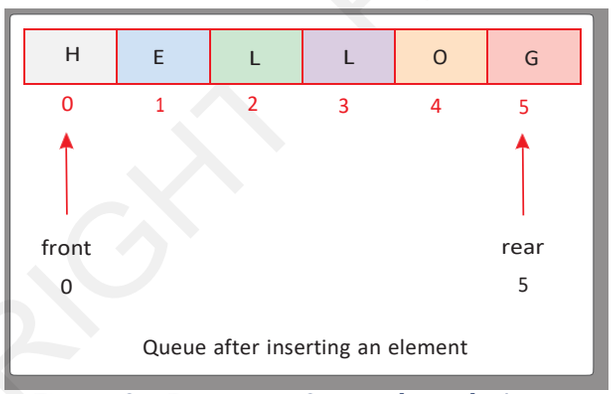


Figure 2: Represent Queue through Array

The variable Front is incremented to 0 from -1 after the removal of data in the queue. After the deletion, the queue will hold the data elements as shown Figure 3:

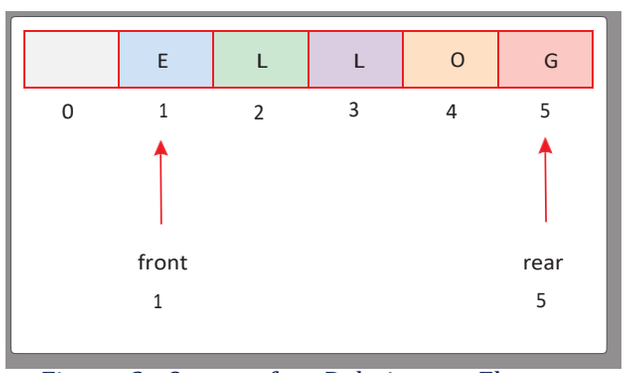


Figure 3: Queue after Deleting an Element

6.3.2 Algorithm Depicting Data Insertion in the Queue

The following points represents the data insertion in queue:

Step 1: Whether REAR = MAX – 1, IF Yes

Display DATA OVERFLOW

Proceed to step 4

[END IF]

Step 2: Whether FRONT = -1 and REAR = -1, IF yes,

Put FRONT = REAR = 0

Put REAR = REAR + 1

[END IF]

Step 3: Put QUEUE [REAR] = VALUE

Step 4: STOP

6.3.3 Algorithm for Deletion Process in Queue

The following points represents the data deletion in queue:

Step 1: Whether FRONT = -1 or FRONT > REAR IF yes

Display DATA UNDERFLOW

ELSE

PUT VAL = QUEUE[FRONT]

PUT FRONT = FRONT + 1

[END IF]

Step 2: STOP

6.3.4 Insertion and Deletion in Queue in C++

Lab Exercise 6a: Write a Programme in C++ to implement Queue Operations.

The following C++ Programme shows the insertion and deletion in queue in C++:

```
#include <iostream>
#include <cstdlib>
using namespace std;
#define SISE 7
class queue
{
    int *arr;
    int Capacity;
    int Front;
```

```
    int Rear;
    int count;

public:
    queue(int sise = SISE);      // constructor
    ~queue();                    // destructor

    void  dequeue();
    void enqueue(int x);
    int peek();
    int sise();
    bool isEmpty();
    bool isFull();
};
queue::queue(int sise)
{
    arr = new int[sise];
    Capacity = sise;
    Front = 7;
    Rear = -1;
    count = 0;
}
queue::~queue() {
    delete[] arr;
}
void queue::dequeue()
{
    if (isEmpty())
    {
        cout << "Underflow\nProgramme Terminated\n";
        exit(EXIT_FAILURE);
    }
    cout << "Removing element from queue " << arr[Front] << endl;
    Front = (Front + 1) % Capacity;
    count--;
}
void queue::enqueue(int item)
{
    if (isFull())
    {
        cout << "Overflow\nProgramme Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Inserting " << item << endl;
    Rear = (Rear + 1) % Capacity;
    arr[Rear] = item;
    count++;
```

```
}
int queue::peek()
{
    if (isEmpty())
    {
        cout << "Underflow\nProgramme Terminated\n";
        exit(EXIT_FAILURE);
    }
    return arr[Front];
}
int queue::sise() {
    return count;
}
bool queue::isEmpty() {
    return (sise() == 0);
}
bool queue::isFull() {
    return (sise() == Capacity);
}
int main()
{
    // create a queue of capacity 7
    queue q(7);
    q.enqueue(3);
    q.enqueue(4);
    q.enqueue(5);
    cout << "The front element in queue is " << q.peek() << endl;
    q.dequeue();
    q.enqueue(4);
    cout << "Sise of the queue is" << q.sise() << endl;
    q.dequeue();
    q.dequeue();
    q.dequeue();

    if (q.isEmpty()) {
        cout << "The queue is vacant\n";
    }
    else {
        cout << "The queue is not empty\n";
    }

    return 0;
}
```


The output of the above C++ programme is as follows:

```
Inserting 3
Inserting 4
Inserting 5
The front element in queue is 0
```

```
Removing element from queue 0
Inserting 4
Sise of the queue is3
Removing element from queue 4
Removing element from queue 5
Removing element from queue 4
The queue is vacant
```

6.4 OPERATIONS IN QUEUE

A queue is an object that permits the following operations are as follows:

- Enqueuer(x): Inserting a data element A at the rear end of queue.
- Dequeuer (): A data element is deleted from the front end of queue.
- Empty (): Verifies whether a queue is vacant or occupied.

6.4.1 Technique in Queue

To achieve the reversal of queue, we utilise the data structure stack which follows the LIFO mechanism. Following this approach, the last data value added to stack will be the initial data value of the reversed queue. The following points show the LIFO mechanisms are as follows:

- Add the data items to the stack after removing them from the queue. The uppermost value of the stack becomes the final value of the queue.
- Add the data items to the queue after removing them from the stack. The final data value of the stack becomes the initial data value to be pushed onto the queue.

6.4.2 Reversal of Queue using Recursion

The following programme shows the reversal of queue using recursion in C++ is as follows:

```
#include <bits/stdc++.h>
using namespace std;
// A function to print the queue
void printQueue(queue<long long int> Queue)
{
    while (!Queue.empty()) {
        cout << Queue.front() << " ";
        Queue.pop();
    }
}
// Reverse the queue using recursive functions
void reverseQueue(queue<long long int>& q)
{
    // Base case
    if (q.empty())
        return;
    // from the fornd end dequeue the current item
    long long int data = q.front();
```

```
    q.pop();
    // remaining queue should be reversed
    // Enqueue current item (to rear)
    q.push(data);
}
// Drive code
int main()
{
    queue<long long int> queue;
    queue.push(65);
    queue.push(72);
    queue.push(35);
    queue.push(40);
    queue.push(95);
    queue.push(29);
    queue.push(85);
    queue.push(65);
    queue.push(56);
    queue.push(100);
    reverseQueue(queue);
    printQueue(queue);
}
```

The output of the above C++ programme is as follows:

```
   72 35 40 95 29 85 65 56 100 65
```

6.4.3 Analysis of Involved Complexities in Queue

The following points shows the complexities in queue are as follows:

- Time Complexity: The time complexity of the above queue reversal process is O(n). Since we have to add all the data items initially in the stack followed by inserting in queue.
- Auxiliary Space: The space complexity of the occupied space required for the above Universal process is O(N). Since we use stack to hold data items.

6.5 TYPES OF QUEUES

There are three types of queues are as follows:

- Circular queue
- Priority queue
- Dequeue

6.5.1 Circular Queue

We can define a Circular queue as a linear data structure, where the final location is attached to the initial location forming a circle, and various related operations make use of the FIFO (First in First out mechanism).

It is popular as Ring Buffer. An example of circular queue is shown in Figure 4:

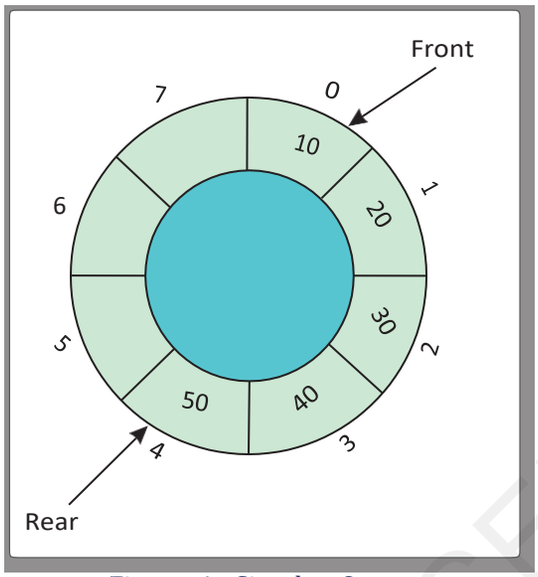


Figure 4: Circular Queue

6.5.2 Circular Queues Using Dynamic Arrays

If we produce a queue by means of an array, where enqueuing and dequeuing receipts O(1) time and space complexity each, one of the queue is occupied, you cannot add new elements, similarly sometimes when you remove elements, the queue can be totally vacant, but still the front will be in the back utmost position, so you can't add any value over there.

Lab Exercise 6b: Write a Programme in C++ to implement Circular Queue Operations.

The following C++ programme shows the implementation of a circular queue using dynamic arrays is as follows:

```
#include <iostream>
using namespace std;
int circularqueue[5];
int front = -1, rear = -1, n=5;
void insertCQ(int val) {
   if ((front == 0 && rear == n-1) || (front == rear+1)) {
      cout<<"Queue is overflow \n";
      return;
   }
   if (front == -1) {
      front = 0;
      rear = 0;
   } else {
      if (rear == n - 1)
      rear = 0;
      else
      rear = rear + 1;
   }
   circularqueue[rear] = val ;
```

```
}
void deleteCQ() {
   if (front == -1) {
      cout<<"Queue Underflow\n";
      return ;
   }
   cout<<"Element deleted from queue is: "
   <<circularqueue[front];
   if (front == rear) {
      front = -1;
      rear = -1;
   } else {
      if (front == n - 1)
      front = 0;
      else
      front = front + 1;
   }
}
void displayCQ() {
   int f = front, r = rear;
   if (front == -1) {
      cout<<"Queue is empty"<<endl;
      return;
   }
   cout<<"The element in Queue are :\n";
   if (f <= r) {
      while (f <= r){
         cout<<circularqueue[f]<<" ";
         f++;
      }
   } else {
      while (f <= n - 1) {
         cout<<circularqueue[f]<<" ";
         f++;
      }
      f = 0;
      while (f <= r) {
         cout<<circularqueue[f]<<" ";
         f++;
      }
   }
   cout<<endl;
}
int main() {
   int ch, val;
   cout<<"1)Insert\n";
   cout<<"2)Delete\n";
   cout<<"3)Display\n";
```

```
   cout<<"4)Exit\n";
   do {
      cout<<"Select the desired element";
      cin>>ch;
      switch(ch) {
         case 1:
         cout<<"Input for insertion";
         cin>>val;
         insertCQ(val);
         break;
         case 2:
         deleteCQ();
         break;
         case 3:
         displayCQ();
         break;
         case 4:
         cout<<"Exit\n";
         break;
         default: cout<<"Invalid!\n";
      }
   } while(ch != 4);
   return 0;
}
```

The output of the given C++ programme is as follows:

```
1)Insert
2)Delete
3)Display
4)Exit
Select the desired element 1
1
Input for insertion 34
34
Select the desired element 1
 1
Input for insertion 56
56
Select the desired element 3
3
The element in Queue are:
34 56
Select the desired element 2
2
Element deleted from queue is: 34
Select the desired element 4
4
Exit
```

6.5.3 Dequeues

The dequeue stands for Double Ended Queue. The data addition in normal queue occurs at one thrilling called as rear end, and the elimination of data occurs at the other extreme known as front end. An example of dequeue is shown in Figure 5:

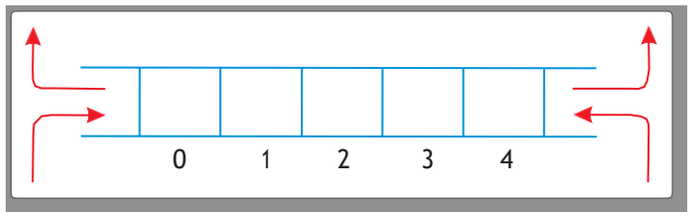


Figure 5: Dequeue

In the Double ended queue the data addition and removal processes are carried out at both the extremes.

Following are some of the properties of queue are as follows:

- Dequeue can act either as a stack or a queue, since the data addition and removal processes are carried out at both the extremes.
- In the Double ended queue, the data addition and removal processes can be carried out at one end which makes it act like a stack. In such case, the queue follows LIFO principle. Figure 6 shows the data addition and removal process:

Figure 6: Data Addition and Removal Process

- A Double ended queue can act like a Queue, If the data addition is carried out at one extreme and the data removal is carried out at the other extreme. In such cases the queue works on FIFO mechanism. Figure 7 shows the addition and deletion of queue in different extreme:

Figure 7: Addition and Deletion of Queue in Different Extreme

Queues can be divided into two categories are as follows:

- Input-restricted queue: In such a queue constraints are exerted during the data addition process. Here the data is added to the queue from one end but the removal can happen from both extremes. Figure 8 shows the data addition from one end and removal of data from both extremes:

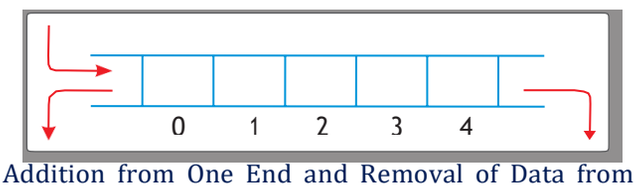


Figure 8: Data Addition from One End and Removal of Data from Both Extremes

- Output-restricted queue: In such a queue constraint is exerted during the data removal process. Here the data removal happens from one end, whereas the data insertion can happen from both the extremes. Figure 9 shows the data deletion from one end and insert from both the extremes:

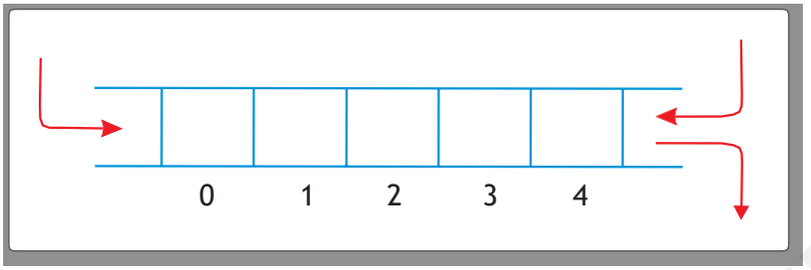


Figure 9: Data is Removed from One End and Insert from Both the Extremes

The following operations are implemented on dequeuer are as follows:

- Insertion of data from front end
- Deletion of data from end
- Insertion of data at the rear
- Deletion of data from the rear end

Apart from the above, peek operation can also be performed on dequeue. The peek operation retains the data elements present at the front and the rear ends.

Additional operations of dequeuer are as follows:

- isFull (): if the stack is fully occupied the function retains true, else, it retains false.
- isEmpty (): if the stack is vacant the function retains true else it retains false.

6.5.4 Priority Queues

A priority queue refers to a specialised queue where each data element has a predetermined service priority. Here, the elements are enqueued at the extreme called rear in the arrival order of the data elements ,Whereas the data elements are dequeued at the front end based on the preferences or the priority of the data elements. An element with lower preference will be dequeued later than the element with higher preference. Figure 10 shows the priority queue:

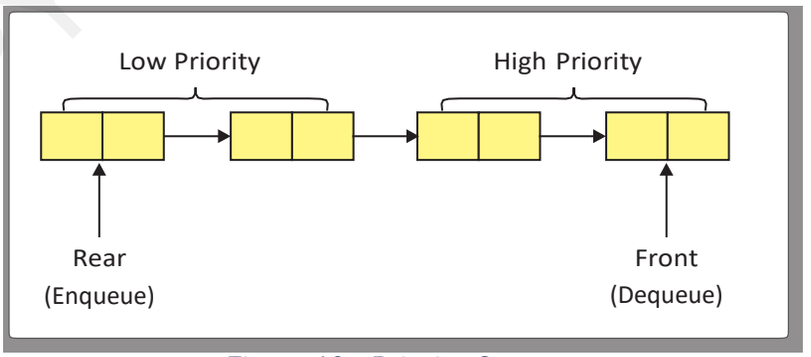


Figure 10: Priority Queues

If multiple items have identical priorities, then they are removed according to their arrival order. This mechanism has its application in dealing with interrupts, in Prim's algorithm, in Dijkstra's algorithm, A search algorithm, heap sort, and Huffman code generation.

6.6 A MAZING PROBLEM

A group of blocks are organised in a stable in an N*N matrix holding a binary values is mentioned as Maze, where, the uppermost block existing at the left extreme that is maze[0][0],acts as source and the lower block existing at the right extreme that is maze[N-1][N-1], acts as the endpoint. The two directions downward and forward are the only pathways for the rat's movement. In the matrix of the Maze, presence of zero signifies that the block is a blind alley and presence of 1 implies the block could act as a pathway to destination from the respective source. And intricate form of the above maze problem, could be, that, the four directions are open for the rat's movement but with restricted moves.

In the maze, the letters denote either a point for making decisions or a point acting as dead end. In order to be aware of what the letter indicates we have to inspect the point. The Exit point is denoted by the letter X. After reaching to a point denoted with letter, insert the neighbouring points denoted with letters to the queue including the initial point.(leave the previously inspected points). Following rules are applicable for resolving the maze problems are as follows:

- After reaching to a point denoted with letter, insert the neighbouring points denoted with letters to the queue including the initial point. (Leave the previously inspected points).
- To decide upon the visit to the next position, a point from the queue is deleted, and a movement is made to the respective point.

An example of a mazing problem is shown in Figure 11.

Blue coloured blocks represent the blind alley or dead ends (Matrix held value = 0).

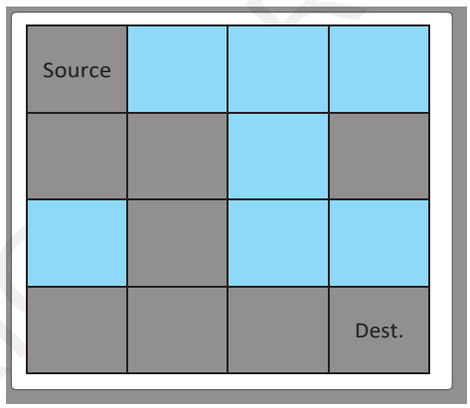


Figure 11: Sample of Maze

The maze below indicates the resolved marked pathway is shown in Figure 12:

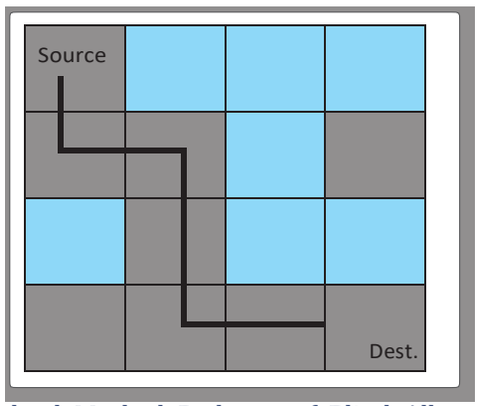


Figure 12: Resolved Marked Pathway of Blind Alley and Dead End

Connecting all the 1s together in the above matrix forms the resolved pathway.

6.7 IMPLEMENTING MULTIPLE STACKS AND QUEUES

The working mechanism of the stack Last in First out (LIFO) needs to be considered during the implementation of queue through stacks.

For the Enqueue operation a single stack is needed, whereas for Dequeue operation two stacks are required.

The following programme shows the implementation of multiple stacks and queues are as follows:

```
#include <iostream>
using namespace std;
int stack[100], n = 100, Top = -1;
void push(int val) {
   if(Top >= n-1)
      cout<<"Stack Overflow"<<endl;
   else {
      Top++;
      stack[Top] = val;
   }
}
void pop() {
   if(Top <= -1)
      cout<<"Stack Underflow"<<endl;
   else {
      cout<<"The popped elements are "<< stack[Top] <<endl;
      Top--;
   }
}
void display() {
   if(Top>= 0) {
      cout<<"Elements in the stack are :";
      for(int i = Top; i>= 0; i--)
         cout<<stack[i]<<" ";
      cout<<endl;
   } else
      cout<<"Stack is vacant";
}
int main() {
   int ch, val;
   cout<<"1) Push in stack"<<endl;
   cout<<"2) Pop from stack"<<endl;
   cout<<"3) Display stack"<<endl;
   cout<<"4) Exit"<<endl;
   do {
      cout<<"Enter the desired element: "<<endl;
      cin>>ch;
      switch(ch) {
         case 1: {
            cout<<"Enter value to be pushed:"<<endl;
            cin>>val;
```

```
            push(val);
            break;
         }
         case 2: {
            pop();
            break;
         }
         case 3: {
            display();
            break;
         }
         case 4: {
            cout<<"Exit"<<endl;
            break;
         }
         default: {
            cout<<"Invalid desired"<<endl;
         }
      }
   }while(ch = 4);
   return 0;
```

The output of the given C++ programme is as follows:

```
1) Push in stack
2) Pop from stack
3) Display stack
4) Exit
Enter the desired element:
1
Enter value to be pushed:
12
Enter the desired element:
1
Enter value to be pushed:
14
Enter the desired element:
1
Enter value to be pushed:
16
Enter the desired element:
1
Enter value to be pushed:
18
Enter the desired element:
3
Elements in the stack are: 18 16 14 12
Enter the desired element:
2
The popped elements are 18
```

```
Enter the desired element:
4
Exit
Enter the desired element:
}
```

Time Complexity for execution of above C++ programme is as follows: For performing these operations through stack the execution time in the worst case is O(n) since we require to move the n data items from stack 1 to stack 2,upon calling the Dequeue operation. The time complexity for inserting data element to stack 1 is O(1).

6.8 TIME COMPLEXITY ANALYSIS OF QUEUES

We remain aware of the locations of the data insertion and removal while dealing with Queues. Therefore, these dual operations could be achieved in one step. Following are the time complexity of queues analysis is as follows:

- The time complexity for operation Enqueue: O(1)
- The time complexity for operation Dequeue: O(1)
- The time complexity for calculating the Size: O(1)

6.9 CONCLUSION

- Queue as an ADT (Abstract data type), which resembles stacks.
- Queue can be represented by using arrays as it is a linear data structure.
- To achieve the reversal of queue, we utilise the data structure stack which follows the LIFO mechanism.
- The time complexity of the above queue reversal process is O(n).
- The space complexity of the occupied space by the queue is O(N).
- The dequeue stands for Double Ended Queue.
- A Priority Queue refers to a specialised queue where each data element has a predetermined service priority.

6.10 GLOSSARY

- Queue: A data structure with organises list of data values.
- Dequeue: In the Double ended queue the data addition and removal processes are carried out at both the extremes.
- Data: It refers to the information.
- Priority queue: It refers to a specialised queue where each data element has a predetermined service priority.
- Enqueuer(x): Inserting a data element A at the rear end of queue.
- Empty (): It verifies whether a queue is vacant or occupied.
- Circular queue: It defines a Circular queue as a linear data structure, where the final location is attached to the initial location.
- Time Complexity: The time complexity of the reversal of queue is O(n).

6.11 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. It is a linear data structure which follows a specific mechanism for enacting the tasks or operations. What is a Queue?
2. The linear arrays can be utilised to denote queues. The variables front and rear are commonly encountered in all the operations of queue. Describe the representation of queue in array.
3. Simple queue describes the simple operation of queue in which insertion arises at the rear of the list and deletion arises at the front of the list. Explain the types of queue.
4. It refers to a specialised queue where each data element has a predetermined service priority. Here, the elements are enqueued at the extreme called Rear, in the arrival order of the data elements. Describe the priority queue.
5. We remain aware of the locations of the data insertion and removal while dealing with Queues. Determine the time complexity analysis of queue.

6.12 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. A queue to be a linear data structure which follows a specific mechanism for enacting the tasks or operations. The queue uses FIFO (First in First out) mechanism for data handling. For instance, a line of customers waiting for supplies are served in their order of arrival. The distinguishing factor among stack and queue lies in the process of data removal.

   Refer to Section Definition of Queue
2. The linear arrays can be utilised to denote queues. The variables front and rear are commonly encountered in all the operations of queue. They indicate the location where we perform the addition and removal of the data elements. In the beginning, the front and rear are set to -1 which indicates a vacant queue.

   Refer to Section Array Representation of Queue
3. There are four types of queues are as follows:
   - Simple queue
   - Circular queue
   - Priority queue
   - Dequeue

   Refer to Section Types of Queue
4. A priority queue refers to a specialised queue where each data element has a predetermined service priority .Here, the elements are enqueued at the extreme called rear, in the arrival order of the data elements, Whereas the data elements are dequeued at the front end based on the preferences or the priority of the data elements.

   Refer to Section Types of Queue
5. We remain aware of the locations of the data insertion and removal while dealing with Queues.

Therefore these dual operations could be achieved in one step.

Refer to Section Time Complexity Analysis of Queue

6.13 POST-UNIT READING MATERIAL

- http://www.gpcet.ac.in/wp-content/uploads/2018/02/ds-FINAL-NOTES-59-88.pdf
- https://www.iare.ac.in/sites/default/files/lecture_notes/IARE_DS_LECTURE_NOTES_2.pdf

6.14 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates about the concept of queue and its representation in array. Also, discuss the types of queue, its application and work with the relevant programmes and understand its working mechanism.

UNIT

07 Linked List

Names of Sub-Units

Introduction to Linked List, Representation of Linked Lists in Memory, Memory Allocation, Linked List Operations, Types of Linked List, Linked Stacks and Queues, Applications of Linked List

Overview

This unit begins by discussing the concept of linked list. Next, the unit explains the representation of linked lists in memory, memory allocation and linked list operations. Further, the unit explains the types of linked list, linked stacks and queues. Towards the end, the unit discusses the applications of linked list.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of linked list
- Explain the concept of representation of linked lists in memory, memory allocation and linked list operations
- Describe the types of linked list
- Explain the significance of linked stacks and queues
- Discuss the applications of linked list

Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of linked list
- Assess the concept of representation of linked lists in memory, memory allocation and linked list operations
- Evaluate the importance of types of linked list
- Determine the significance of linked stacks and queues
- Explore the applications of linked list

Pre-Unit Preparatory Material

- https://www.tutorialspoint.com/data_structures_algorithms/pdf/linked_lists_algorithm.pdf

7.1 INTRODUCTION

In a data structure, linked list is generally a chain of nodes in which each node is connected to each other by means of pointers or references. The size of a linked list can vary depending on the requirements of the users, hence it is dynamic in nature. Linked list considered as a collection or sequence of the same kind of items. It is the second most-used data structure after the array. The building block of linked list is known as node.

7.2 REPRESENTATION OF LINKED LIST IN MEMORY

A list is an ordered datatype in which the elements are kept in a sequence efficiently for the retrieval of elements. It allows the recurrence that means a particular part of a data can be occur more than one time in a list. When the similar kind of data is entered numerous times in list, each entry of that recurring data is considered as a discrete item.

An array or list is very much similar but the main difference between the array and the list is that array stores homogenous data whereas the list stores heterogeneous data items. Linked list is also considered as sequence in data structure. The representation of linked list is shown in Figure 1:

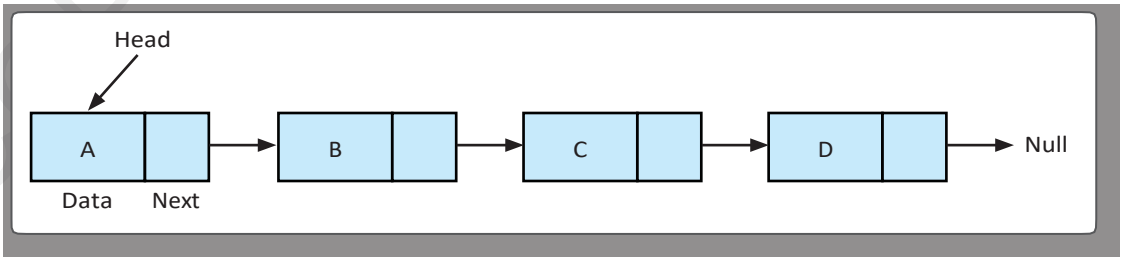


Figure 1: Representation of Linked List

7.3 MEMORY ALLOCATION

Memory allocation is the process of storing and preserving a comprehensive and uncomprehensive part of computer memory for the implementation of programs and procedures. It is concluded through

a process is called memory management. Before the implementation of any process it must be first located in the memory. The space or area where process is placed is allocated by a memory.

The memory allocation is further classified into two categories:

- Static memory allocation
- Dynamic memory allocation

7.3.1 Static Memory allocation

Static memory allocation is achieved during compilation. The node of block and memory stored will be freed during time of compiling. When there is more than required memory it will be wastage. When the memory is not as much of as the required memory, the program is not able to run further. The requirement of memory should be known in advance.

The following C++ program is used to show the implementation of static memory allocation using array are as follows:

```
// C++ program to illustrate
// non-static data members
using namespace std;
#include <iostream>
class GfG {
private:
     // Creation of static variable
     static int count;
public:
     void set_count()
     {
          count++;
     }
     // Funciton to access the private members
     void show_count()
     {
     // Display count variable
          cout << count << '\n';
     }
};
int GfG::count = 0;
int main()
{
     // class GfG objects
     GfG A1, A2, A3, A4, A5;
     A1.set_count();
     A2.set_count();
     A3.set_count();
     A4.set_count();
     A5.set_count();
     // Display count function
     A1.show_count();
     A2.show_count();
```

```
    A3.show_count();
    A4.set_count();
    A5.set_count();
    return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/0U3BJxRuLJ.o
5
5
5
```

7.3.2 Dynamic Memory allocation

Dynamic memory allocation is just opposite to the static memory, in which the requirement of memory is describe during the implementation of program. In dynamic memory, we can allocate the memory acc to our requirement as there is not wastage of memory. Hence, it is not essential to know the exact memory requirement previously. The following C++ program is used to show the implementation of dynamic memory allocation using array are as follows:

```
#include <iostream>
using namespace std;
int main ()
{
    // Initialization of pointer to NULL
    int* N = NULL;
    N = new(nothrow) int;
    if (!N)
        cout<< "allocation of memory failed\n";
    else
    {
        // Values should be stored in allocated address
        *N=59;
        cout<<"The value of N: " << *N <<endl;
    }
    float *L = new float(85.30);
    cout<< "The value of L: " << *L <<endl;
    // Size of memory for block
    int size = 9;
    int *arr = new(nothrow) int[size];
    if (!arr)
        cout<< "allocation of memory failed\n";
    else
    {
        for (int i = 0; i< size; i++)
            arr[i] = i+1;
        cout<< "The value stored in block of memory: ";
        for (int i = 0; i< size; i++)
            cout<<arr[i] << " ";
    }
    delete N;
```

```
    delete L;
    // freed the block of allocated memory
    delete[] arr;
    return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/0U3BJxRuLJ.o
The value of N: 59
The value of L: 85.3
The value stored in block of memory: 1 2 3 4 5 6 7 8 9
```

7.4 LINKED LIST OPERATIONS

In data structure, there are linked list operations which allow us to perform distinct action on linked list. Some of the linked list operations are as follows:

- Traversing
- Searching
- Insertion
- Deletion

7.4.1 Traversing

A linked list is a linear data structure that involve traverse of node in every state of singly linked list. Basically traversing means to reach each and every node in the list one after another in the sequence to perform some operation on that.

The following steps considered an algorithm of traversing in linked list:

1. Set PTR initialization = HEAD
2. Apply process to pointer PTR
3. Set pointer = PTR-> NEXT
4. Repeat step 2 and 3 up to PTR! = NULL
5. Exit

7.4.2 Searching

Searching is also a linear linked list in data structure which is used to determine the state of a specific element in the list. In a searching, we want to traverse the comprehensive linked list and start comparing each node with the data to be explore until a match is found. In case, if the element from the linked list is matched then the position of that element is resumed from the function. When to start the process of searching from the starting node as random is not possible.

The following steps describe the algorithm of searching in linked list:

1. [Initialization] set PTR = HEAD
2. Repeat up to PTR[j]! = DATA: j=j+1
3. If j=n+1, then set PTR= 0

Else PTR= n+1

4. Exit

7.4.3 Insertion

We apply insertion operation on linked list when we want to insert a new node in the list. But for that it is significant to traverse the list in order one after another to find the position where inserting a new node, therefore, we will insert a new node on that place. A new node can be entered wherever it is required at the beginning to the end. An example of inserting a new node at the fort of linked list is shown in Figure 2:

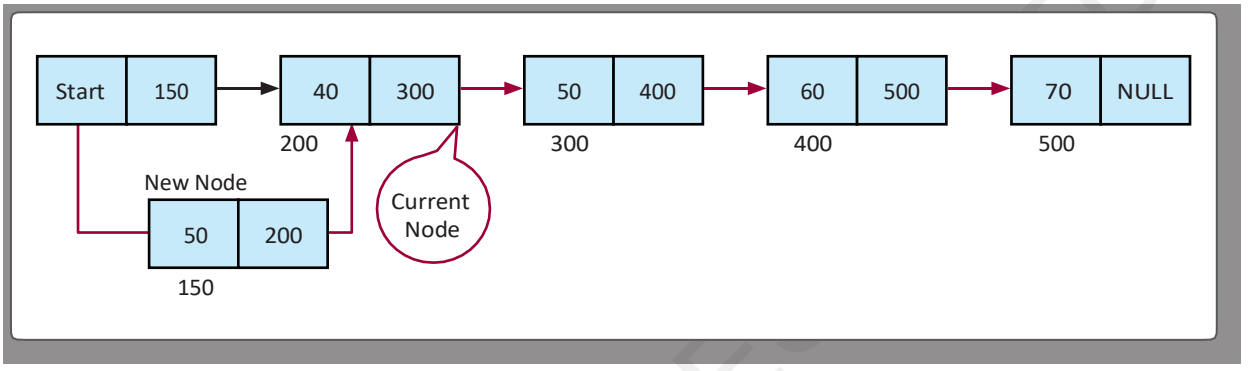


Figure 2: add New Node at the Front

When we want to add the new node at a particular position is shown in Figure 3:

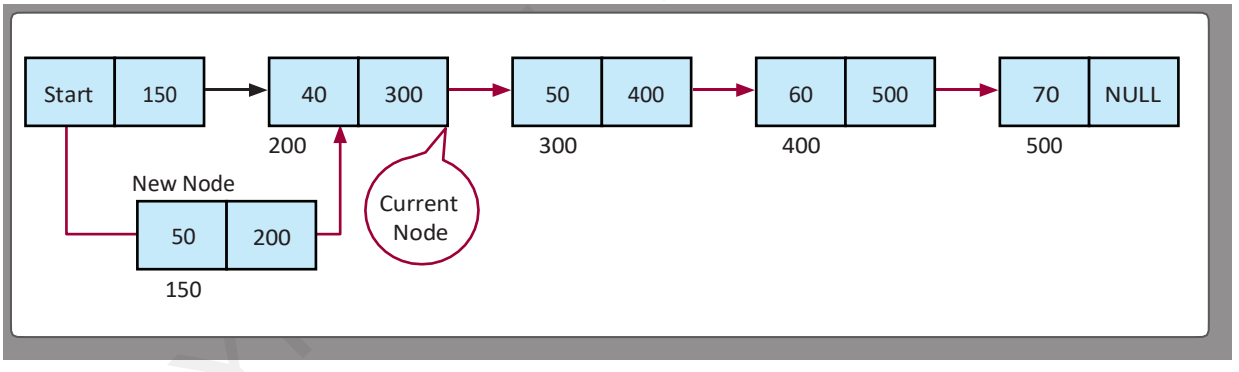


Figure 3: add a New Node at a Particular Position

And, if we want to insert a new node at the end of a linked list, an example is shown in Figure 4:

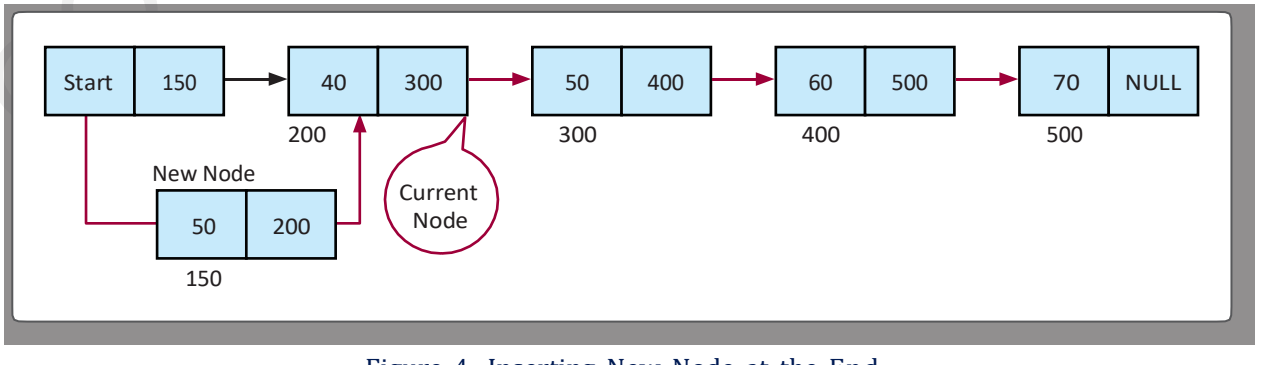


Figure 4: Inserting New Node at the End

7.4.4 Deletion

A similar process is required for deletion as same in insertion of traversing a linked list to find the node to delete. We can delete the node from existing linked list at the beginning to the end. But just a little adjustment should be required while deleting an element we must adjust the head, pointer to the next from the existing.

Deletion a node from beginning

An example of deleting an existing node from linked list is shown in Figure 5:

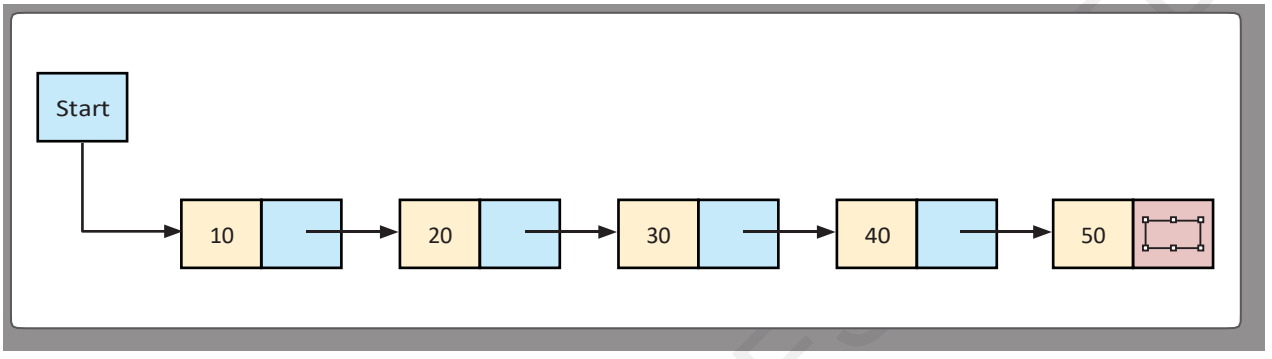


Figure 5: Linked List for Deletion

The first node is deleted from the given linked list is shown in Figure 6:

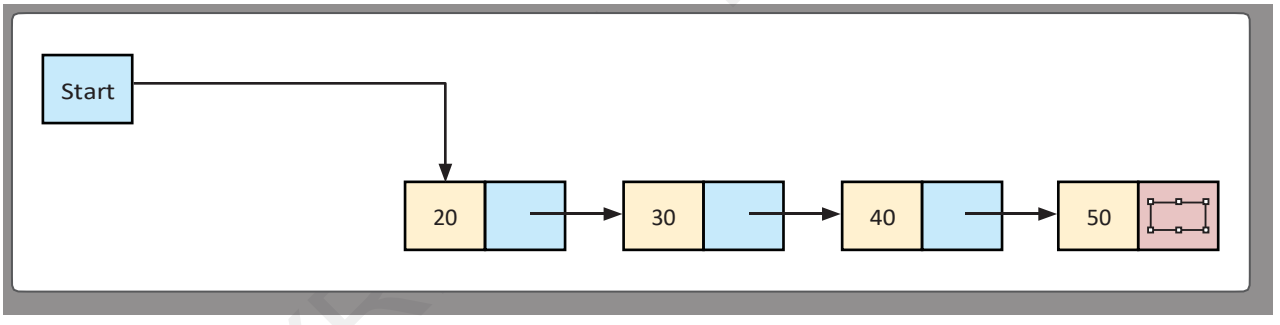


Figure 6: Linked List after Deletion

Deleting a node at the middle

An example of deleting node from the middle is shown in Figure 7:

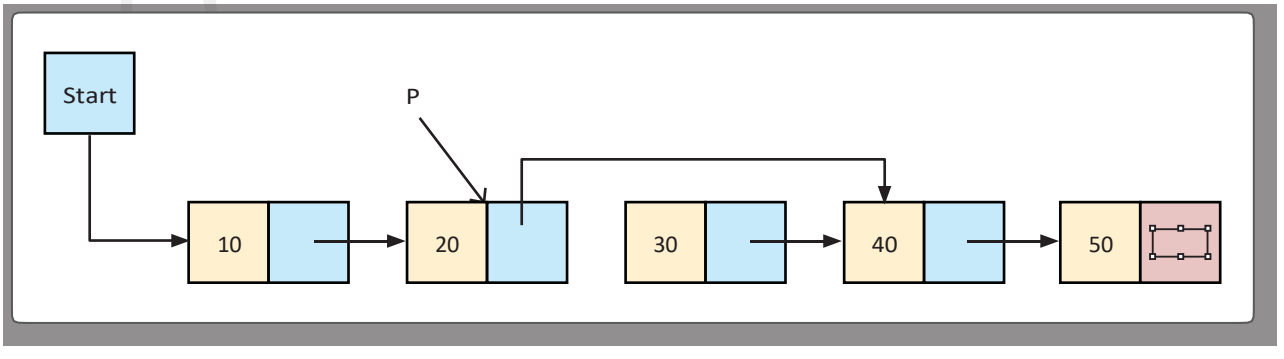


Figure 7: Linked List after Deletion at the Middle

Deleting a Node at the End

Example of deleting a node at the end of the linked list is shown in Figure 8:

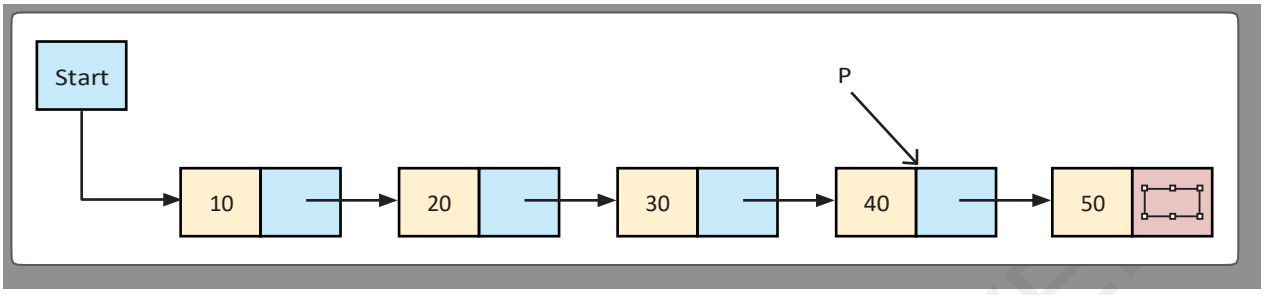


Figure 8: Linked List for Deletion at the End

We want to delete the node p as shown in Figure 8, after deletion at the end the final linked list is shown in Figure 9:

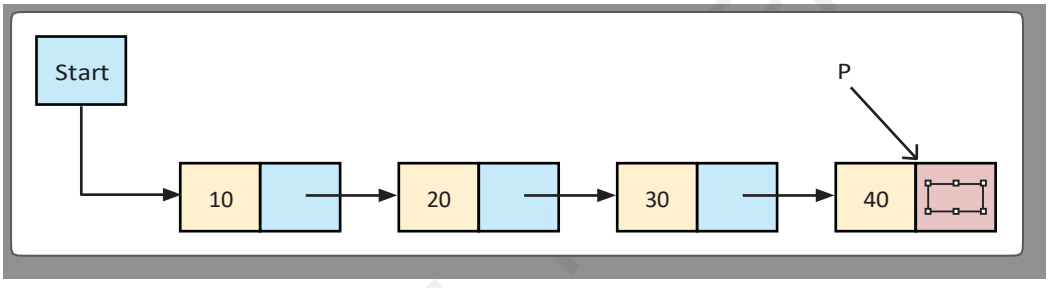


Figure 9: Linked List after Deletion at the End

7.5 TYPES OF LINKED LIST

A linked list in a data structure is a sequence of node connected to one another by pointers. Linked list is further classified into four ways are as follows:

1. Singly linked list
2. Doubly linked list
3. Circular linked list
4. Header linked list

7.5.1 Singly Linked List

.In a data structure singly linked list is considered as the collection of set of elements in an organised manner. In singly linked list the number of elements is required as per the program. It consists a node can be classified into two parts:

1. Data part
2. Link part

In which data part contain the definite information about the node that is to be represented while the link part consist the address of instant successor. A singly linked list is traversed only in one direction because each node containing only next pointer. Hence, it is not possible to traverse the linked list in reverse direction.

A singly linked list is shown in Figure 10:

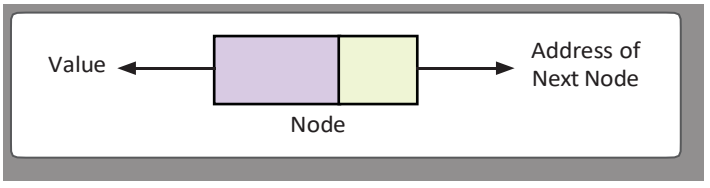


Figure 10: Singly Linked List

In a singly linked list the element in a list is connected to each other in such a way that the value of next variable to the last node is NULL is shown in Figure 11:

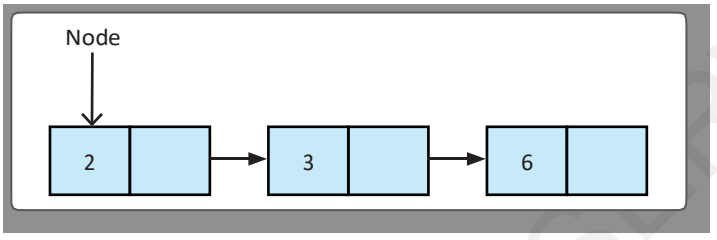


Figure 11: NULL to the Linked List

7.5.2 Doubly Linked List

A doubly linked list is considered as more complicated than singly linked list because the nodes in doubly linked list comprise a pointer to the succeeding as well as the preceding node in an ordered manner. It can be traversed in a one as well as in a backward direction. A doubly linked list is shown in Figure 12:

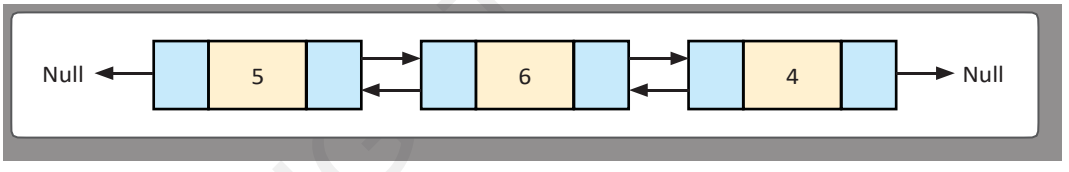


Figure 12: Doubly Linked List

7.5.3 Circular Linked List

It is the process in which the last node comprises the pointer to the front node or first node of the list is known as circular linked list. Though traversing a circular linked list initiate from any node and traversed it in any forward and backward direction until we influence the similar node where we started. Thus, it has no starting and no ending. For better understanding the circular linked list considered the following Figure 13:

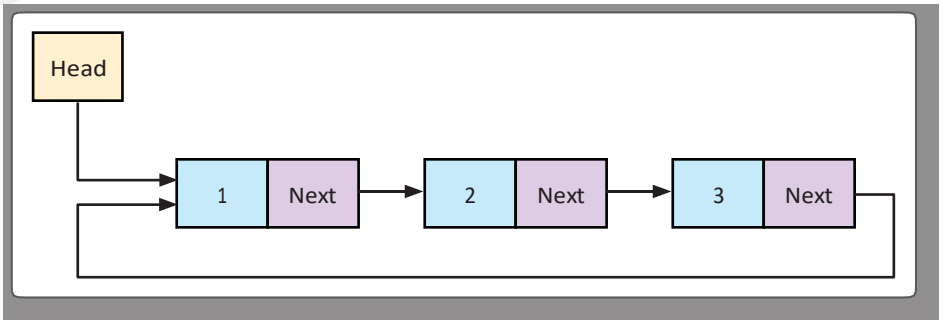


Figure 13: Circular Linked List

7.5.4 Header Linked List

In a linked list a special node that is found at the beginning of the list is known as header linked list and the node is considered as header node. That is why in a header node 'start' is not indicated the first node of the list but it will consists the address of that node/ header node. However, header linked list is further classified into two parts as follows:

- Grounded header linked list
- Circular header linked list

An example of header linked list is shown in Figure 14:

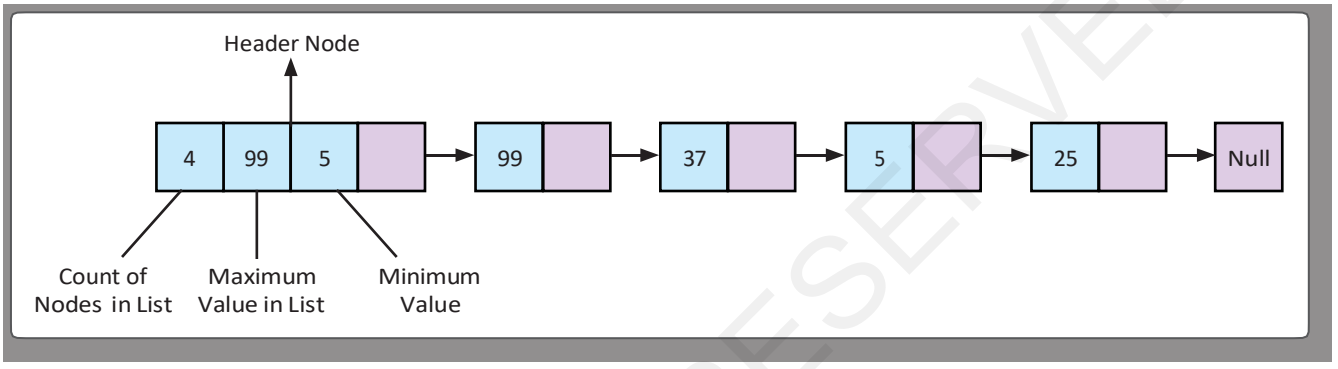


Figure 14: Header Linked List

7.6 LINKED STACKS AND QUEUES

A stack is a linear data structure in which the insertion and deletion of elements is done at only one end rather than in the middle. It can be executed by using arrays of linear type. It is mostly used in transforming and assessing expression in polish notation.

Whereas, a linear stack is a list of elements which is generally executed as a singly linked list, thus it start pointer plays the role of the top pointer of a stack.

A linear queue is defined as it is represented by using array, in which only finite no of elements can be inserted using two pointers FRONT and REaR. The FRONT pointers performs a role of starting node whereas, the REAR pointer is set to play the role at the last node.

The following C++ program is used to implement the linear queue using linked list are as follows:

```
#include <iostream>
using namespace std;
struct node {
     int data;
     struct node *next;
};
struct node* Front = NULL;
struct node* Rear = NULL;
struct node* Temperature;
void Insert() {
     int val;
     cout<<"Enter element in queue : "<<endl;
     cin>>val;
     if (Rear == NULL) {
```

```
            Rear = (struct node *)malloc(sizeof(struct node));
            Rear->next = NULL;
            Rear->data = val;
            Front = Rear;
      } else {
            Temperature=(struct node *)malloc(sizeof(struct node));
            Rear->next = Temperature;
            Temperature->data = val;
            Temperature->next = NULL;
            Rear = Temperature;
      }
}
void Delete() {
      Temperature = Front;
      if (Front == NULL) {
            cout<<"Underflow"<<endl;
            return;
      }
      else
      if (Temperature->next != NULL) {
            Temperature = Temperature->next;
            cout<<" Elements delete from the queue: "<<Front->data<<endl;
            free(Front);
            Front = Temperature;
      } else {
            cout<<"Elements delete from the queue: "<<Front->data<<endl;
            free(Front);
            Front = NULL;
            Rear = NULL;
      }
}
void Display() {
      Temperature = Front;
      if ((Front == NULL) && (Rear == NULL)) {
            cout<<"Queue is empty"<<endl;
            return;
      }
      cout<<"Queue elements are: ";
      while (Temperature != NULL) {
            cout<<Temperature->data<<" ";
            Temperature = Temperature->next;
      }
      cout<<endl;
}
int main() {
      int ch;
      cout<<"1) Enter element to queue"<<endl;
      cout<<"2) Delete element from queue"<<endl;
      cout<<"3) Display all elements of queue"<<endl;
```

```
     cout<<"4) Exit"<<endl;
     do {
          cout<<"Enter your choice: "<<endl;
          cin>>ch;
          switch (ch) {
               case 1: Insert();
               break;
               case 2: Delete();
               break;
               case 3: Display();
               break;
               case 4: cout<<"Exit"<<endl;
               break;
               default: cout<<"Invalid choice"<<endl;
          }
     } while(ch!=4);
     return 0;
}
```

The output of given code is as follows:

```
/tmp/qmuGaeqHMo.o
1) Enter element to queue
2) Delete element from queue
3) Display all elements of queue
4) Exit
Enter your choice:
1
Enter element in queue:
23
Enter your choice:
1
Enter element in queue:
34
Enter your choice:
1
Enter element in queue :
56
Enter your choice :
3
Queue elements are: 23 34 56
Enter your choice :
2
Elements delete from the queue: 23
Enter your choice :
4
Exit
```

The following C++ program is used to implement the linear stack using linked list are as follows:

```
#include <iostream>
using namespace std;
```

```
struct Node {
     int data;
     struct Node *next;
};
struct Node* Top = NULL;
void PUSH(int val) {
     struct Node* Newnode = (struct Node*) malloc(sizeof(struct Node));
     Newnode->data = val;
     Newnode->next = Top;
     Top = Newnode;
}
void POP() {
     if(Top==NULL)
     cout<<"Stack Underflow"<<endl;
     else {
          cout<<"The popped element is "<< Top->data <<endl;
          Top = Top->next;
     }
}
void display() {
     struct Node* ptr;
     if(Top==NULL)
     cout<<"stack is empty";
     else {
          ptr = Top;
          cout<<"Stack elements are: ";
          while (ptr != NULL) {
               cout<< ptr->data <<" ";
               ptr = ptr->next;
          }
     }
     cout<<endl;
}
int main() {
     int ch, val;
     cout<<"1) Push element in stack"<<endl;
     cout<<"2) Pop element from stack"<<endl;
     cout<<"3) Display all element of stack"<<endl;
     cout<<"4) Exit"<<endl;
     do {
          cout<<"Enter choice: "<<endl;
          cin>>ch;
          switch(ch) {
               case 1: {
                    cout<<"Enter value to be pushed:"<<endl;
                    cin>>val;
                    PUSH(val);
                    break;
               }
```

```
                case 2: {
                      POP();
                      break;
                }
                case 3: {
                display();
                      break;
                }
                case 4: {
                      cout<<"Exit"<<endl;
                      break;
                }
                default: {
                      cout<<"Invalid Choice"<<endl;
                }
          }
     }while(ch!=4);
     return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/qmuGaeqHMo.o
1) Push element in stack
2) Pop element from stack
3) Display all element of stack
4) Exit
Enter choice:
1
Enter value to be pushed:
12
Enter choice:
1
Enter value to be pushed:
16
Enter choice:
1
Enter value to be pushed:
28
Enter choice:
3
Stack elements are: 28 16 12
Enter choice:
2
The popped element is 28
Enter choice:
4
Exit
```

7.7 APPLICATIONS OF LINKED LIST

In a data structure, linked list is generally a chain of nodes in which each node is connected to each other by means of pointers or references. A single node is simply an object that has "data" and a "next" pointer that points to the next node in a singly or circular linked list, as well as a "previous" pointer that points to the previous node in a doubly linked list.

Some of the application of linked list is as follows:

- Execution of stacks and queues.
- Execution of graphs: Adjacency list illustration of graphs is most popular which uses linked list to store adjacent vertices.
- Dynamic memory allocation: We use linked list of free blocks.
- Preserving directory of names.
- Executing arithmetic operations on extensive integers.
- Manipulation of polynomials by storing constants in the node of linked list.
- Demonstrating sparse matrices.

7.8 TIME COMPLEXITY ANALYSIS OF LINKED LIST

The time complexity of memory address as we know that to access a definite element, the time complexity is (O ($\sqrt{N}$) where N refers to the block of continuous element being read. Linked lists have maximum of their advantage when it comes to the insertion and deletion of nodes in the. Contrasting the dynamic array, insertion and deletion at any portion of the list takes constant time. In the array, we could at least have the array sorted. The following time complexity for linked list operations is as follows:

- The time complexity of Indexing - O(n)
- The time complexity of Insertion - O(1)
- The time complexity of Search - O(n)
- The time complexity of Deletion - O(1)

Lab Exercise

7(a). Write a Program in C++ to demonstrate possible operations of Singly Linked List (SLL).

The following C++ program is used to implement the singly linked list are as follows:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    struct Node *next;
};
struct Node* HEAD = NULL;
void insert(int New_data) {
    struct Node* New_node = (struct Node*) malloc(sizeof(struct Node));
    New_node->data = New_data;
    New_node->next = HEAD;
    HEAD = New_node;
```

```
}
void display() {
     struct Node* pointer;
     pointer = HEAD;
     while (pointer != NULL) {
          cout<< pointer->data <<" ";
          pointer = pointer->next;
     }
}
int main() {
     insert(15);
     insert(20);
     insert(25);
     insert(30);
     insert(35);
     cout<<"The linked list is: ";
     display();
     return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/qmuGaeqHMo.o
The linked list is: 35 30 25 20 15
```

7(b). Write a Program in C++ to demonstrate possible operations of Doubly Linked List (DLL).

The following C++ program is used to implement the doubly linked list are as follows:

```
#include <iostream>
using namespace std;
struct Node {
     int data;
     struct Node *Previous;
     struct Node *Next;
};
struct Node* HEAD = NULL;
void insert(int Newdata) {
     struct Node* Newnode = (struct Node*) malloc(sizeof(struct Node));
     Newnode->data = Newdata;
     Newnode->Previous = NULL;
     Newnode->Next = HEAD;
     if(HEAD != NULL)
     HEAD->Previous = Newnode ;
     HEAD = Newnode;
}
void display() {
     struct Node* PTR;
     PTR = HEAD;
     while(PTR != NULL) {
          cout<< PTR->data <<" ";
          PTR = PTR->Next;
     }
```

```
}
int main() {
     insert(30);
     insert(25);
     insert(18);
     insert(16);
     insert(12);
     cout<<"The doubly linked list is: ";
     display();
     return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/qmuGaeqHMo.o
The doubly linked list is: 12 16 18 25 30
```

7.9 CONCLUSION

- In a data structure, linked list is generally a chain of nodes in which each node is connected to each other.
- A list is an ordered datatype, in which the elements are kept in a sequence.
- Memory allocation is the process of storing and preserving a comprehensive and uncomprehensive part of computer memory.
- Static memory allocation is achieved during compilation.
- Dynamic memory allocation is achieved during the implementation of programming.
- A linked list is a linear data structure that involve traverse of node in every state.
- Searching is also a linear linked list in data structure which is used to determine the state of a specific element.
- We apply insertion operation on linked list when we want to insert a new node in list.
- A similar process is required for deletion as same in insertion of traversing.
- In singly linked list the number of elements is required as per the program.
- A doubly linked list is considered as more complicated than singly linked list.
- A circular linked list initiate from any node and traversed it in any direction.
- A special node that is found at the beginning of the list is known as header linked list.
- A linear stack is a list of elements which is generally executed as a singly linked list.
- A linear queue is defined as it is represented by using array.

7.10 GLOSSARY

- Linked list: It is generally a chain of nodes in which each node is connected to each other.
- List: It is an ordered datatype, in which the elements are kept in a sequence.

- Memoryallocation: It isthe process of storingandpreservingacomprehensive anduncomprehensive part of computer memory.
- Static memory allocation: It is achieved during compilation.
- Dynamic memory allocation: It is achieved during the implementation of programming.
- Searching: It is a linear linked list in data structure which is used to determine the state of a specific element.
- Insertion: It is used to insert a new node in list.
- Deletion: It is required for deletion as same in insertion of traversing.
- Singly linked list: It is a number of elements are required as per the program.
- Doubly linked list: It is considered as more complicated than singly linked list.
- Circular linked list: It initiates from any node and traversed it in any direction.
- Header linked list: It is a special node that is found at the beginning of the list.
- Linear queue: It is defined as it is represented by using array.

7.11 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. In a data structure, linked list is generally a chain of nodes. What is a linked list in data structure?
2. Before the implementation of any process it must be first located in the memory. Describe the concept of memory allocation with its types.
3. A linked list is a linear data structure that involve traverse of node in every state of singly linked list. Explain various types of linked list operations in brief.
4. A linked list in a data structure is a sequence of node. Evaluate the concept of types of linked list in detail.
5. A stack is a linear data structure in which the insertion and deletion of elements is done at only one end rather than in the middle. Determine the importance of linear stack and queue in linked list.

7.12 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. In a data structure, linked list is generally a chain of nodes in which each node is connected to each other by means of pointers or references.

   Refer to Section Introduction

2. Memory allocation is the process of storing and preserving a comprehensive and uncomprehensive part of computer memory for the implementation of programs and procedures.

   Refer to Section Memory Allocation

3. In data structure, there are linked list operations which allow us to perform distinct action on linked list. Some of the linked list operations are as follows:

- Traversing
- Searching
- Insertion
- Deletion

Refer to Section Linked List Operations

4. A linked list in a data structure is a sequence of node connected to one another by pointers. Linked list is further classified into four ways:

   Refer to Section Types of Linked List

5. A linear stack is a list of elements which is generally executed as a singly linked list, thus it start pointer plays the role of the top pointer of a stack.

   Refer to Section Linked Stacks and Queues

7.13 POST-UNIT READING MATERIAL

- https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm
- https://www.scribd.com/document/546191365/02-Linked-List

7.14 TOPICS FOR DISCUSSION FORUMS

- Disucss the concept of linked list, its representation and its types with your friends and classmates. Also discuss the linear array and stack with real world examples.

UNIT

08 Trees

Names of Sub-Units

Introduction to Tree, Definition of a Tree, Tree Terminology, Binary Tree, Complete Binary Tree, Properties of Binary Tree, Array and Linked Representation of Binary Trees, Binary Tree Traversals, Additional Binary Tree Operations, Threaded Binary Trees, Binary Search Trees, Application of Trees, Evaluation of Expression.

Overview

This unit begins by discussing about the concept of tree, tree terminology, binary tree and complete binary tree. Next, the unit discusses the properties of binary tree, array and linked representation of binary trees and binary tree traversals. Further the unit explains the additional binary tree operations, threaded binary trees and binary search trees. Towards the end, the unit discusses the application of trees and evaluation of expression.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of tree, tree terminology, binary tree and complete binary tree
- Explain the concept of properties of binary tree, array and linked representation of binary trees
- Describe the binary tree traversals, additional binary tree operations and threaded binary trees
- Explain the significance of binary search trees and application of tress
- Discuss the concept of evaluation of expression

Learning Outcomes

At the end of this unit you would:

- Evaluate the concept of tree, tree terminology, binary tree and complete binary tree
- Assess the concept of properties of binary tree, array and linked representation of binary trees
- Evaluate the importance of binary tree traversals, additional binary tree operations and threaded binary trees
- Determine the significance of binary search trees and application of tress
- Explore the concept of evaluation of expression

Pre-Unit Preparatory Material

- https://www.tutorialspoint.com/discrete_mathematics/introduction_to_trees.htm

8.1 INTRODUCTION

A distinct data structure where hierarchical relationships exist among the data items or nodes is referred as Trees. It is a non-linear hierarchical data structure which includes a group of objects called as nodes. These nodes connected with one another in a tree using "edges". Due to the non-linear structure of trees it increases the process of storing, retrieving and manipulation of data by using innovative methods. A tree in which each node of a tree comprises data and positions. It is genuine that there is a tree it must have a root node. That root node is the topmost node in a hierarchy of tree. A tree is called a tree when it having a one root node, sub-node, parent node and a child node. All these nodes together make a tree. A tree is shown in Figure 1:

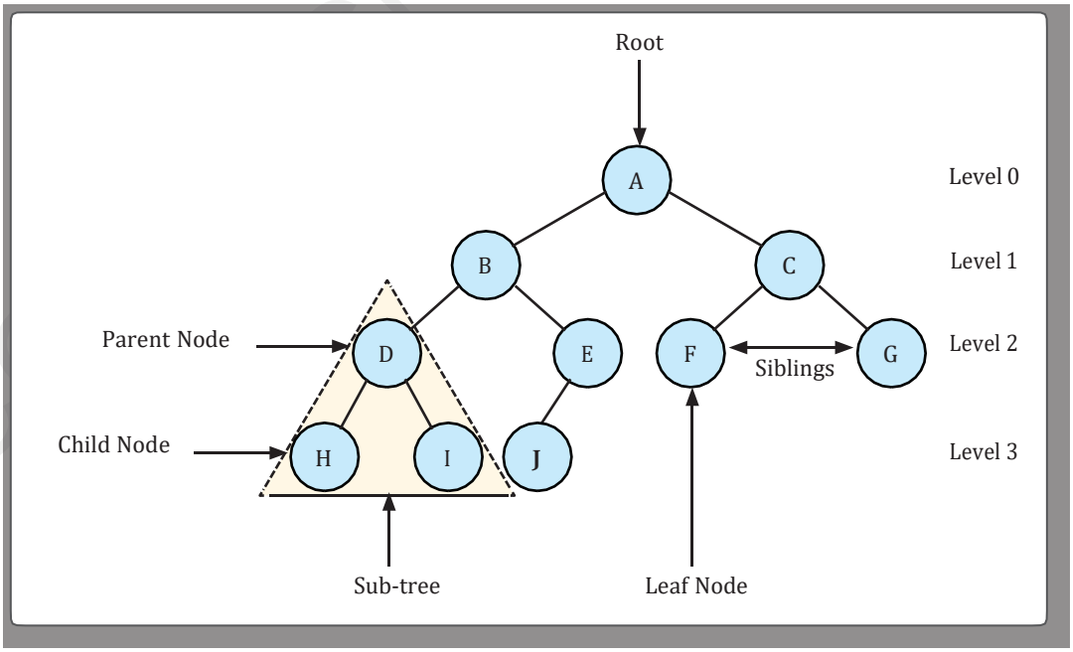


Figure 1: Tree

8.2 DEFINITION OF TREE

A tree refers to an inter linked open chain adrift graph. The Count of the edges in a tree consisting N vertices is N-1. The vertex with degree 0 is addressed as the root. A leaf node in a tree is considered as when its vertex has degree one. The internal nodes have degree starting from 2. A Decision tree is a specialised tree where each internal node indicates a trial or criteria on a anticipative variable. The edges provide feasible solutions to the trials. The leaf node provides the results of all trials conducted on a pathway. Such trees are widely used in predictive modelling. The leaf node in a tree is shown in Figure 2:

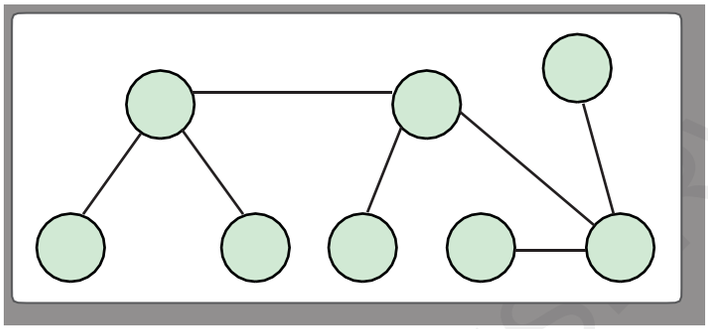


Figure 2: Leaf Nodes

A tree is known to consist hierarchical data. If you wish to display the details of the employees and their designations in a hierarchical manner as shown in Figure 3:

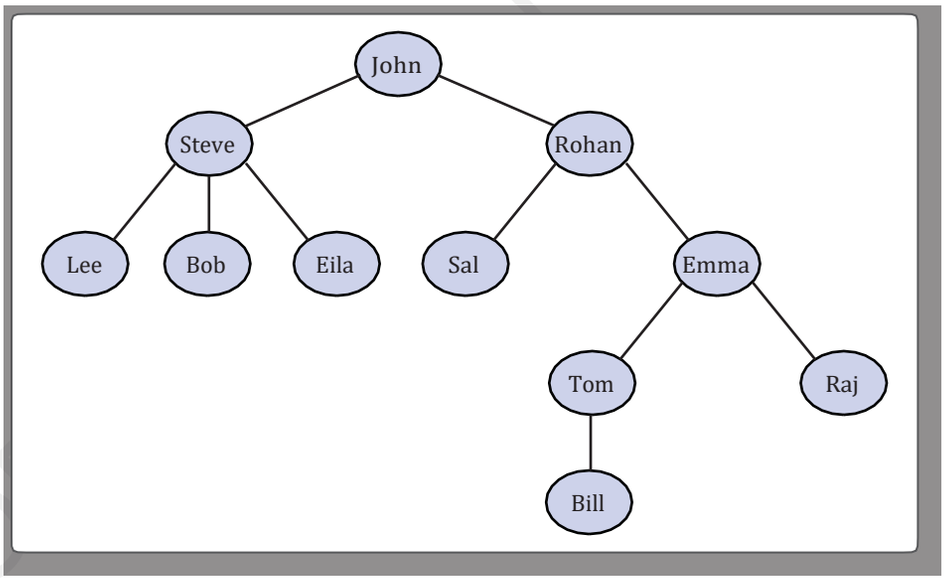


Figure 3: Tree having Hierarchical Data

The image depicts the administrative hierarchy of a company.John is acting CEO. Steve and Rohan are the subordinates of John. *Lee, Bob, Ella* are the supervisees of Steve. *Sal* and *Emma* are the subordinates of Rohan. Tom and Raj are the supervisees of Emma. Bill is a subordinate of Tom. Such rational design pattern or formation is known as Tree. Hair the topmost vertex act as root and all the edges are the branches going in the descending direction. This trees act as an efficient data structure to hold the hierarchical data.

8.3 TREE TERMINOLOGY

In a Tree, Every distinct element is known as node. Node in a tree data structure stores the actual data of that specific element and connects to next element in hierarchical manner. An example of tree terminology is shown in Figure 4:

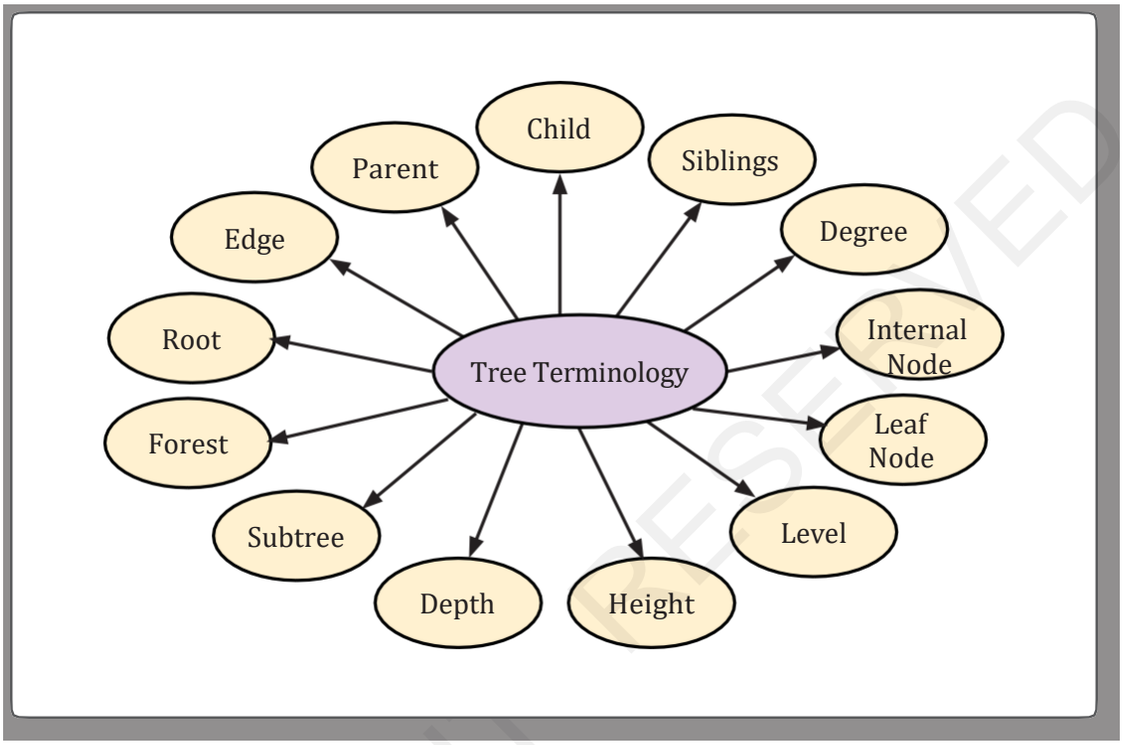


Figure 4: Tree Terminology

Some of the terminologies in a tree are as follows:

- Root: A root node is an exclusive node in the tree in which other subtrees were connected. It has its child in left and right of the tree as shown in Figure 5:

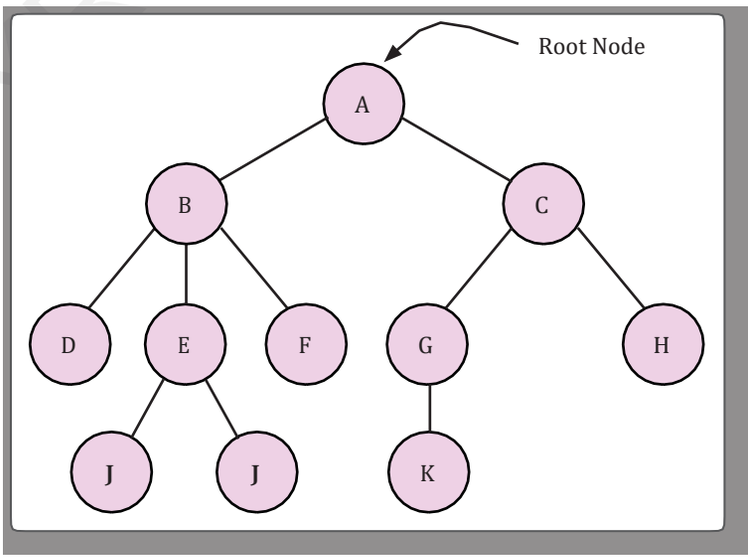


Figure 5: Root Node

- Edge: In a tree each node is connected with one another through a link is known as "edge". It connects each and every node either it has child or not is shown in Figure 6:

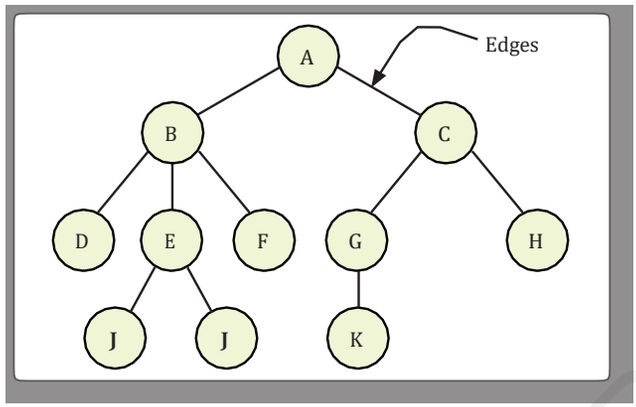


Figure 6: Edge in a Tree

- Parent: In a tree a node which having one or more child is termed as parent node. It has any number of child nodes in a tree is shown in Figure 7:

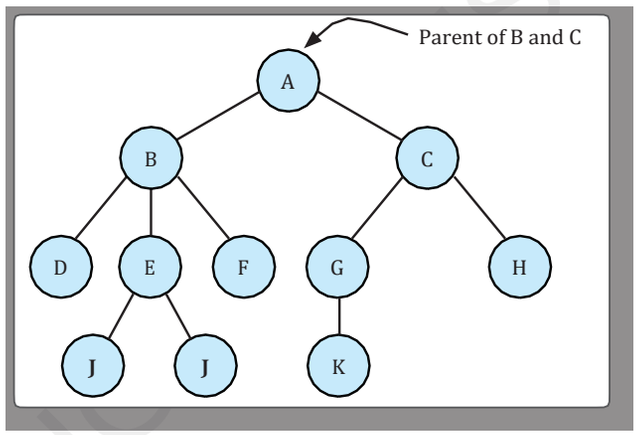


Figure 7: Parent Node

In the above figure, A tree in which A is a parent of (B & C) node, B is a parent of (D, E & F) and so on.

- Child: In a tree any sub node of a given node is called child node and all the nodes in a tree except root node are child nodes as shown in Figure 8:

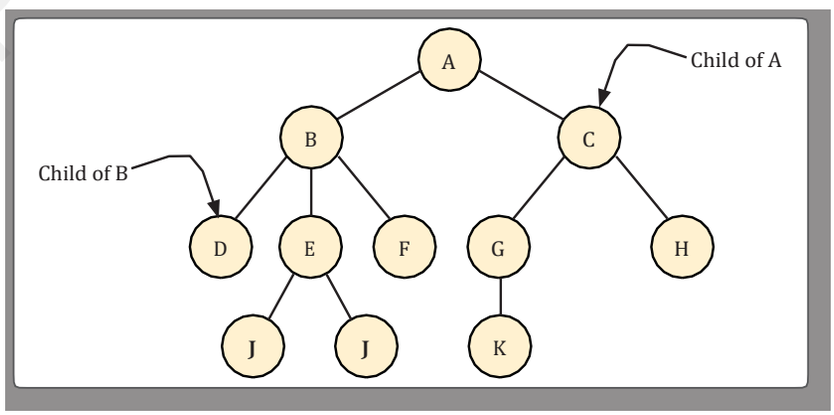


Figure 8: Child Node

- Siblings: In a tree a node which belongs to the same parent node is called siblings node as shown in Figure 9:

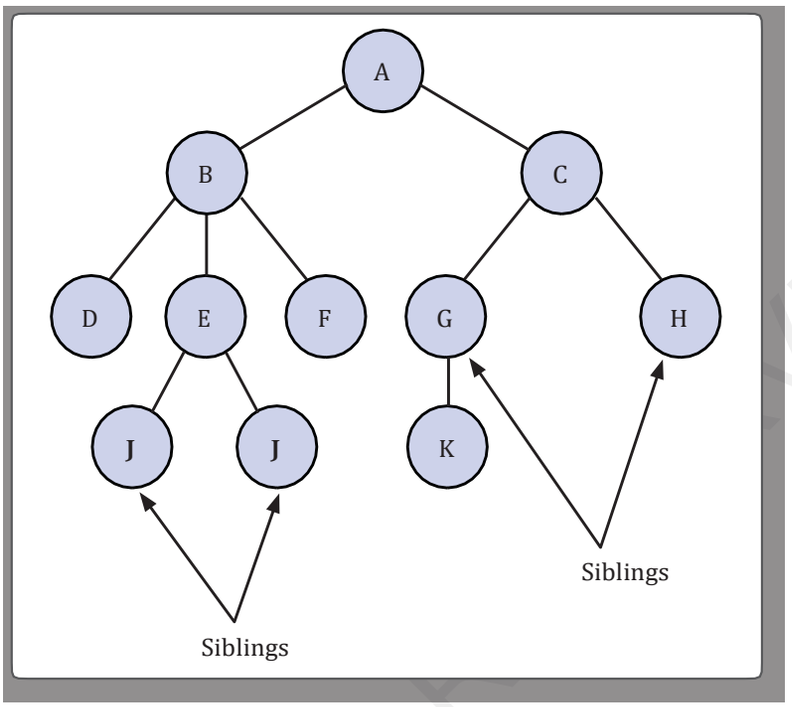


Figure 9: Siblings Node

- Degree: A degree determines the total number of child of a node. It depicts the highest degree of the node among all the nodes in a tree as shown in Figure 10:

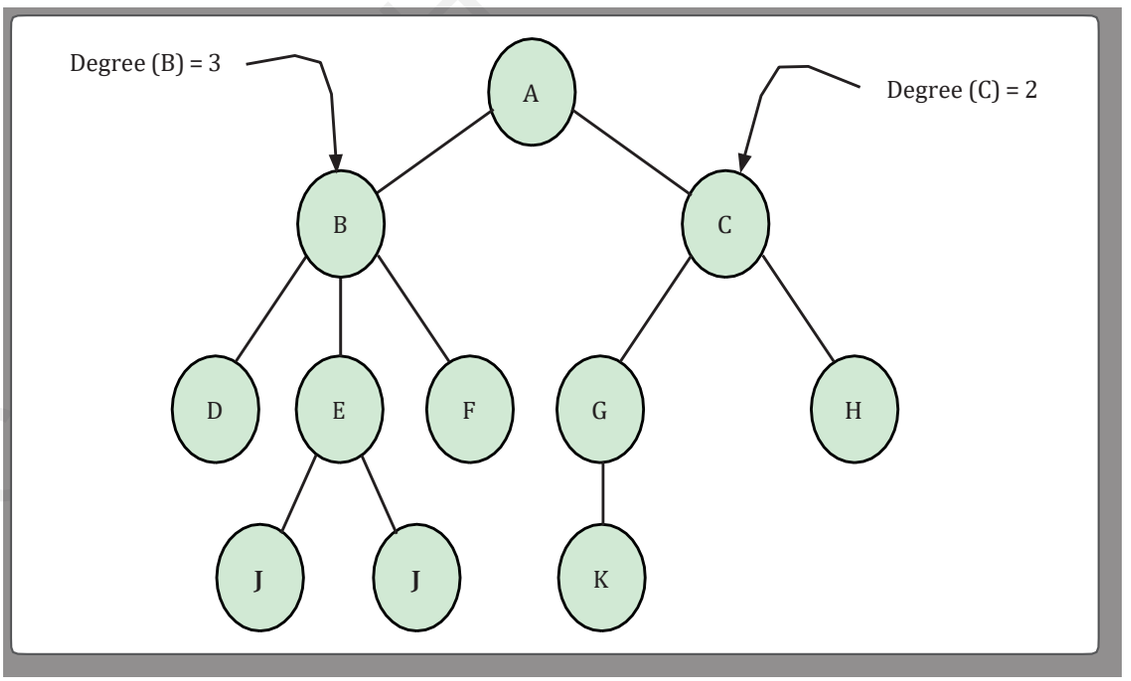


Figure 10: Degree of a Tree

- Internal node: In a hierarchy of a tree it is not mandatory that the entire parent node comprises two or more child; some of them having only one child node such node is called as internal node or non-terminal nodes as shown in Figure 11:

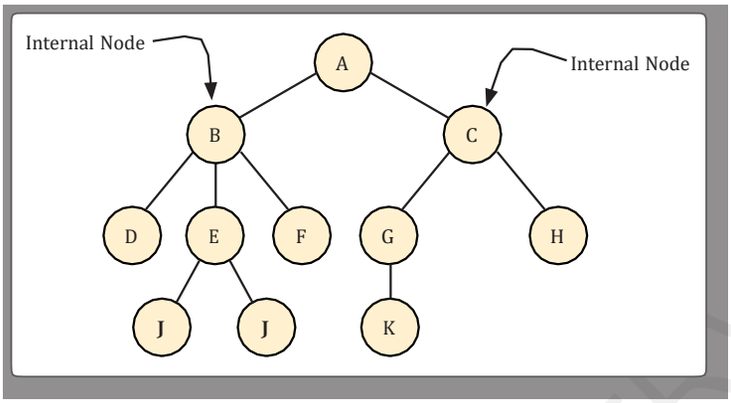


Figure 11: Internal Node

- Leaf node: In a tree such type of nodes with no child also exists is known as leaf node. These nodes also called as external or terminal nodes of a tree as shown in Figure 12:

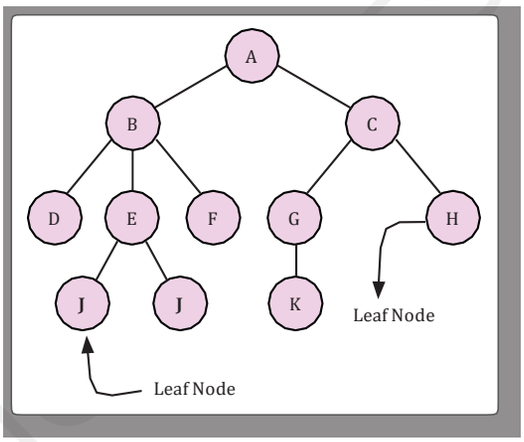


Figure 12: Leaf Node

- Level: In a tree data structure is a step-by-step process of measuring a tree from top to bottom. The first level of a tree count start from 0 and it increases by 1 on each step as shown in Figure 13:

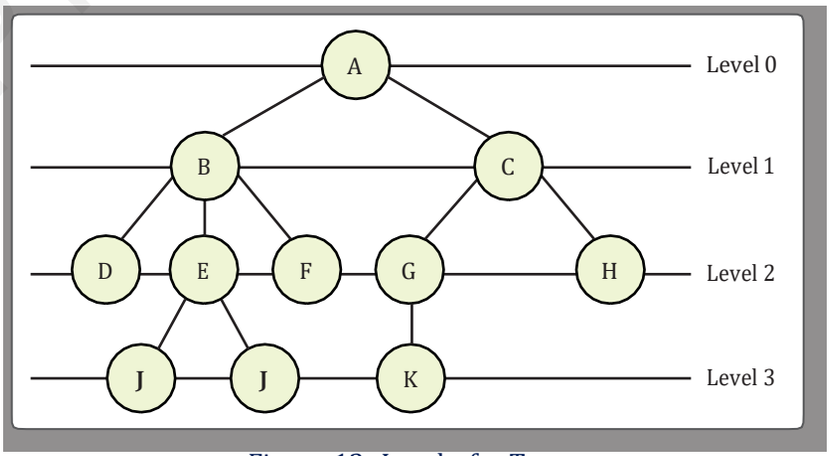


Figure 13: Level of a Tree

- Height: The total number of edges on the longest path from leaf node to a specific node is called height. It is also considered as the maximum level of a tree as shown in Figure 14:

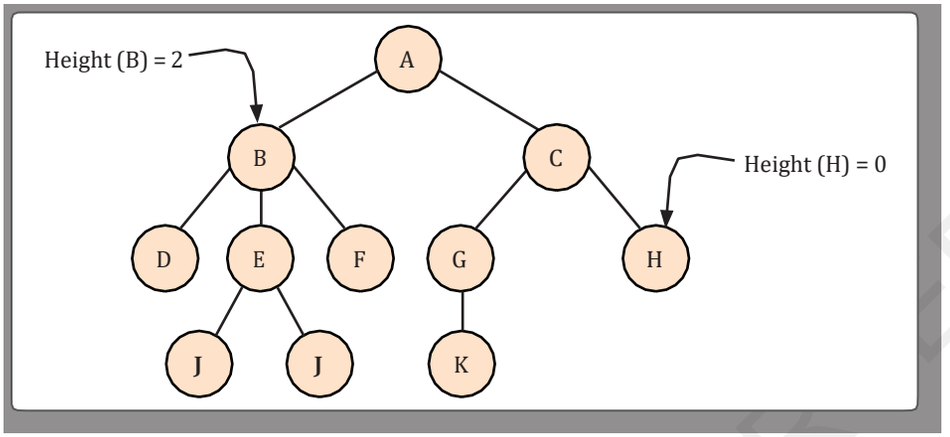


Figure 14: Height of a Tree

- Depth: In a hierarchy of tree the total number of edges that connects root node to the leaf node in the downward path is known as depth as shown in Figure 15:

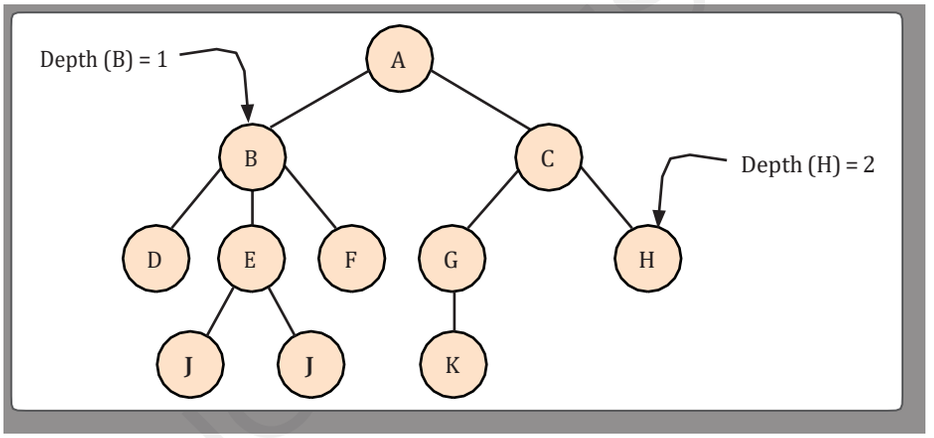


Figure 15: Depth of a Tree

- Subtree: In a tree a node with a child of a node and form another tree is known as subtree. In a subtree every child node has a parent as shown in Figure 16:

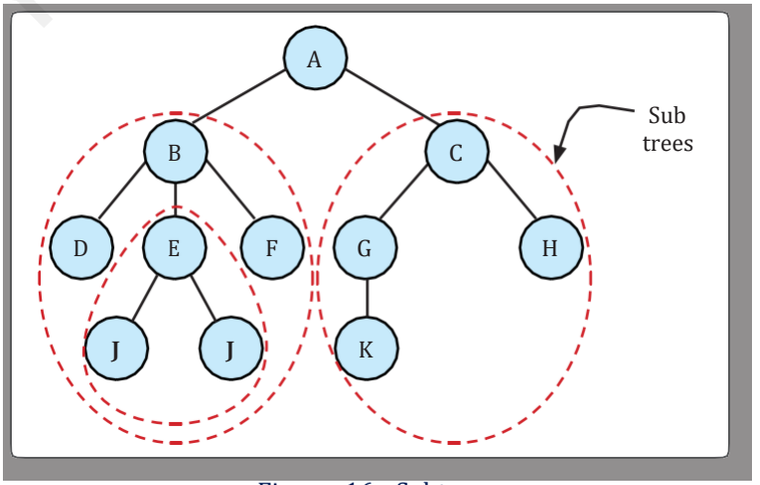


Figure 16: Subtree

- Forest: It is a collection of those trees which are not joint together or we can say it has a group of disjoint trees as shown in Figure 17:

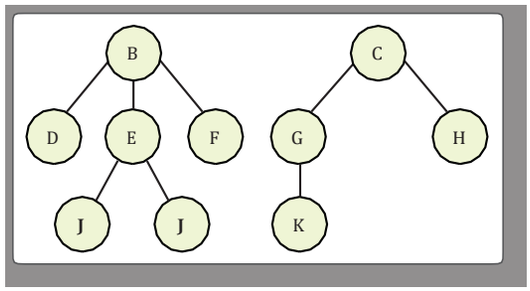


Figure 17: Forest

8.4 BINARY TREE

In a data structure a binary tree is a distinct type of tree as by the name binary that means two, every node or vertex in a tree has either zero, one and two child one. But in a binary tree is an important session of a tree data structure in which a node at most having two child nodes or we can say that every parent node in a binary tree can have only two children maximally. It is not a linear data structure like queues, linked list and arrays etc. it is a hierarchical data structure instead of that.

In a binary tree all the nodes have three main components are as follows:

- Data element
- Right pointer
- Left pointer

A binary tree is shown in Figure 18:

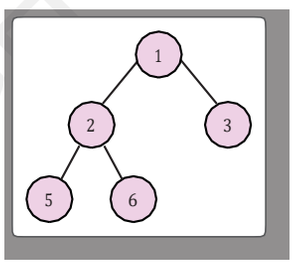


Figure 18: Binary Tree

We can have the following analytical depiction of the above binary tree where the nodes resemble the nodes of a linked list as shown in Figure 19:

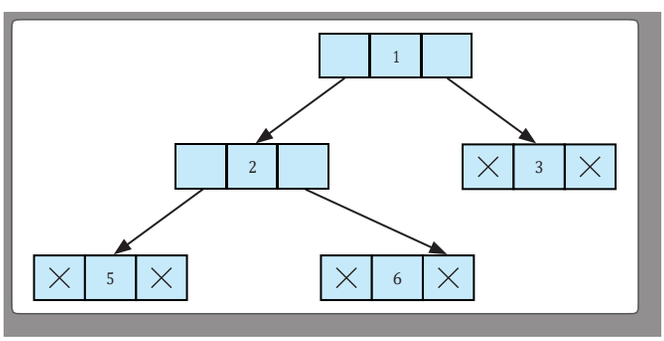


Figure 19: Nodes of a Linked List

Here, Node 1 has dual pointers, left and right directing to the corresponding left and right nodes. Similarly node 2 has two pointers as well. Nodes 3, 5 and 6 have NULL pointers in their left and right portions as they are leaf nodes.

8.5 COMPLETE BINARY TREE

As a binary tree, a complete binary tree is a tree in which each and every level of a tree is completely filled except the last level of a complete binary tree. Completing of each level determines that every parent node should have exactly two child node. A parent node that doesn't have its left and right node is not defined as complete binary tree. Although, a binary tree to fulfil the requirement in order to be a complete binary tree is that the last level of a tree contain all the keys as left as possible. It means that a parent node on the last level of (CBT) but it should have a child node on the left only.

A complete binary tree differs from the full binary tree in ways as follows:

- Every leaf node must incline around the left.
- The final leaf node may lack its right sibling. In other words, a complete binary tree need not be a full binary tree.

A complete binary tree is shown in Figure 20:

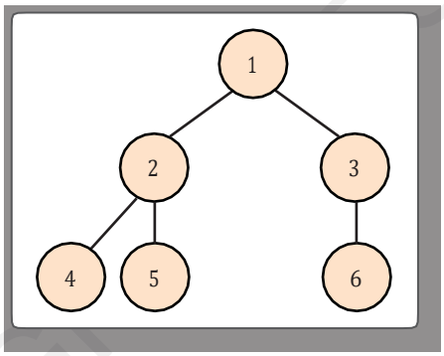


Figure 20: Complete Binary Tree

8.6 PROPERTIES OF TREE

The following points describe the properties of tree are as follows:

- The total count of nodes at a level i is given by 2^i.
- The lengthiest way from the root to the leaf node is called Height of a tree. The tree displayed above has height of 3. Hence the total count of nodes at height 3 is given by (1+2+4+8) = 15. In general, the total count of nodes at height h is $(2^0 + 2^1 + 2^2 + ...2^h) = 2^{h+1} - 1$.
- The least count of nodes at height h is given by h+1.
- The presence of least count of node indicates greatest height, whereas the presence of greatest count of nodes indicates least height in a tree.

If a binary tree has a node count of n then determines the following points:

- We can calculate the least or minimum height

  We are aware as:

  $n = 2^{h+1} - 1$

  $n+1 = 2^{h+1}$

Applying log on both sides of the equation as:

$\log_2 (n+1) = \log2 (2^{h+1})$

$\log_2 (n+1) = h+1$

$h = \log_2 (n+1) - 1$

- The overall height is calculated

We are aware that:

$n = h+1$

$h = n-1$

8.7 ARRAY AND LINKED LIST REPRESENTATION OF BINARY TREE

A tree in a data structure which has a distinct type of tree, every parent node should have exactly two child nodes is knows as binary tree. It is represented in a hierarchal manner. Now, how we can represent the binary tree in a computer memory. It can be represented in the two ways are as follows:

- Representing through array
- Representing through linked list

Let's have a look at the tree as shown in Figure 21:

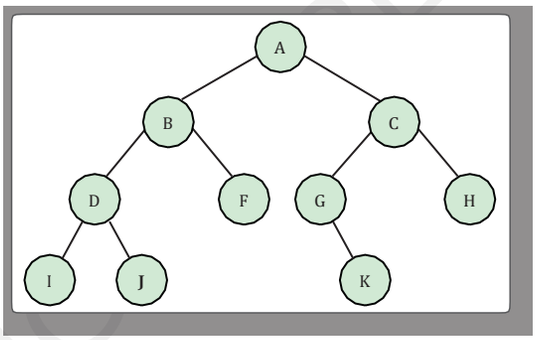


Figure 21: A Tree

8.7.1 Representing Binary Tree through Array

While representing a binary tree using array firstly we require converting that binary tree into a full binary tree and provide number to each and every node to store in an array in their respective order. Fig 21 is converted into a full binary tree is shown in Figure 22:

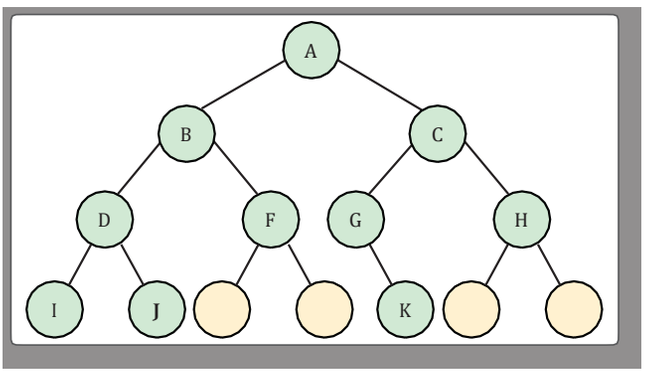


Figure 22: Full Binary Tree

The full binary tree is now represented through array as shown in Figure 23:

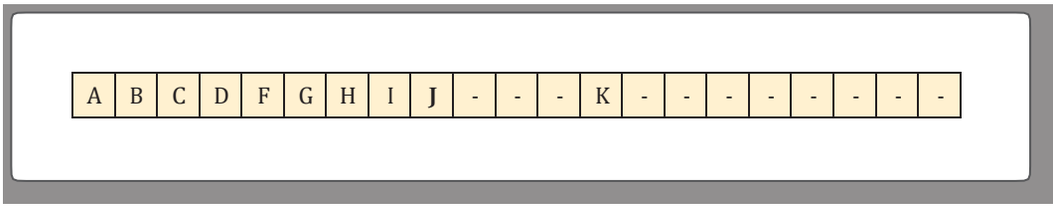


Figure 23: Array represents Binary Tree

8.7.2 Representing Binary Tree through Linked List

When binary tree is represented through linked list is stored in the memory as linked lists. In linked list the nodes are not kept at adjacent memory location as shown in Figure 24:

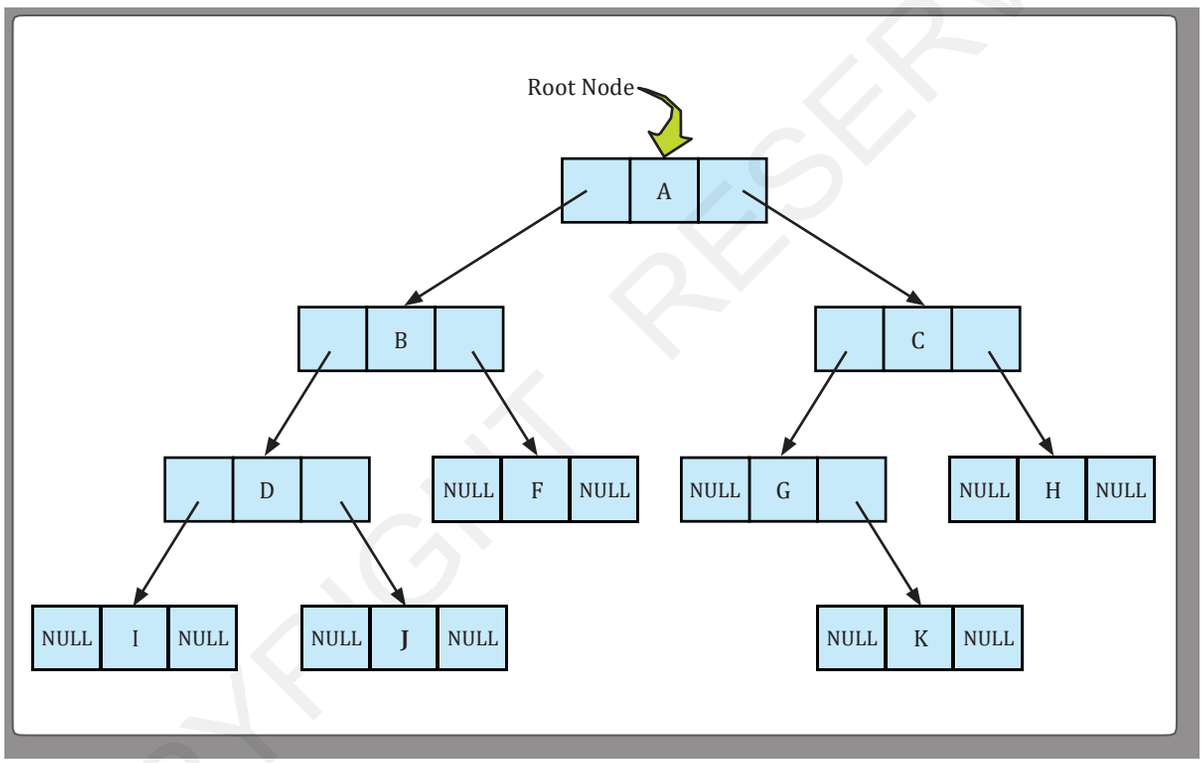


Figure 24: Linked List Represents Binary Tree

8.8 BINARY TREE TRAVERSAL

Binary tree traversal in data structure is the process of visiting, updating and verifying once each node of a tree. Linear data structures such as arrays, stacks, queues, and linked list possess uni-directional paths to scan the information. However we have multiple pathways to scan the information present in the hierarchical data structures like trees. We can perform distinct tree traversal in data structure in many ways. Some of them are as follows:

- In-order traversal
- Post-order traversal
- Pre-order traversal

8.8.1 In-Order Traversal

An in-order traversal, always start visiting from the left subtree then it visit to root node and after the right subtree as shown in Figure 25:

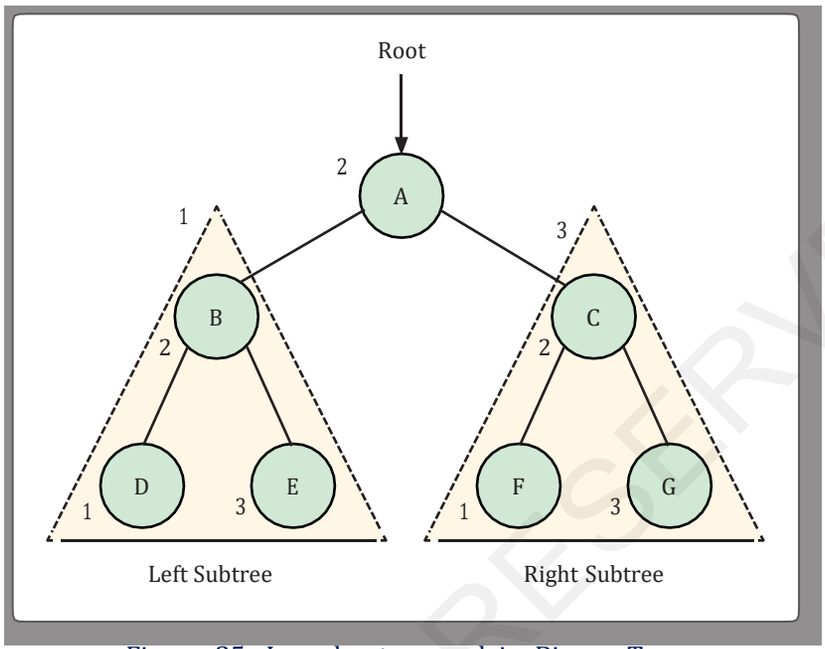


Figure 25: In-order traversal in Binary Tree

8.8.2 Post-Order Traversal

In the post-order traversal the process of visiting starts from left subtree of a tree, then it visit to the right subtree and later it visit to root node as shown in Figure 26:

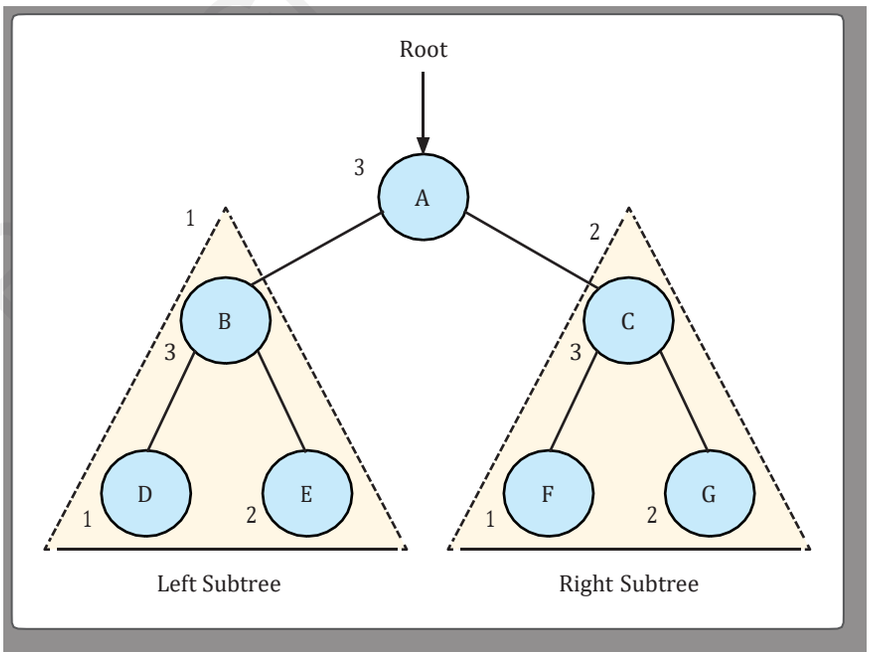


Figure 26: Post-order Traversal in Binary Tree

8.8.3 Pre-Order Traversal

In the pre-order traversal is just opposite to the post-order traversal, in which firstly we visit to the root node then to the left subtree and then finally to the right subtree of a binary tree as shown in Figure 27:

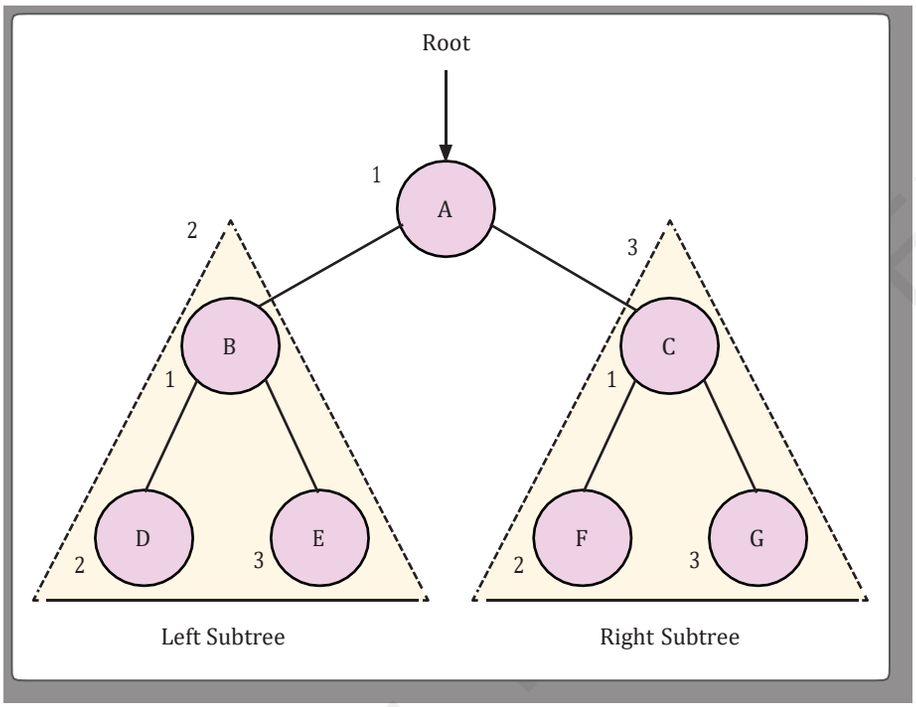


Figure 27: Pre-order Traversal in Binary Tree

8.9 ADDITIONAL BINARY TREE OPERATIONS

Some of the most common additional binary tree operations are as follows:

- Depth-first order
- Breadth-first order

8.9.1 Depth-first Order

In this technique, we initiate by meeting the remotest node w.r.t the root node, which must be a child of a previously seen node. We need not recollect the seen nodes, since a tree should not have cycles. Pre-order traversal is a noteworthy illustration of depth first travel. We can refer depth-first search for further clarity.

8.9.2 Breadth-first Order

Any alternative name of this traversal is level-order traversal. Considering a complete binary tree, the width-index of a node given by (i - (2d - 1)) is utilised for noting the travelling directions from root node. For achieving this, scan bitwise sinistrodextraly, initiating from bit d - 1, where d denotes the distance between the root and node, (d = floor (log2 (i+1))). If we mask the width-index at bit d - 1, the bit becomes 0 and if it is 1, then we need to move either left or right, accordingly. The verification goes on by consecutive examination of the upcoming bits from left to right till all the nodes are visited. The last travel from a parent node to a child node is given by the rightmost bit.

8.10 THREADED BINARY TREE

As we know that the binary tree nodes have exactly two children but in the case of having only one child or no child node, the link part in the linked list representation remains NULL. The threaded binary tree helps to reuse that vacant links again by making some threads. If a vacant left or right child area one node has then it will be used as thread. And if in the left threaded mode of a tree if some node without left child node then the left pointer will point out to its in-order predecessor, similarly in the right threaded tree if some node without any child node then the right pointer will point out to its in-order successor. If there is no successor or predecessor is existing in both the cases then it will point to the header node of a tree. A threaded binary tree is shown in Figure 28:

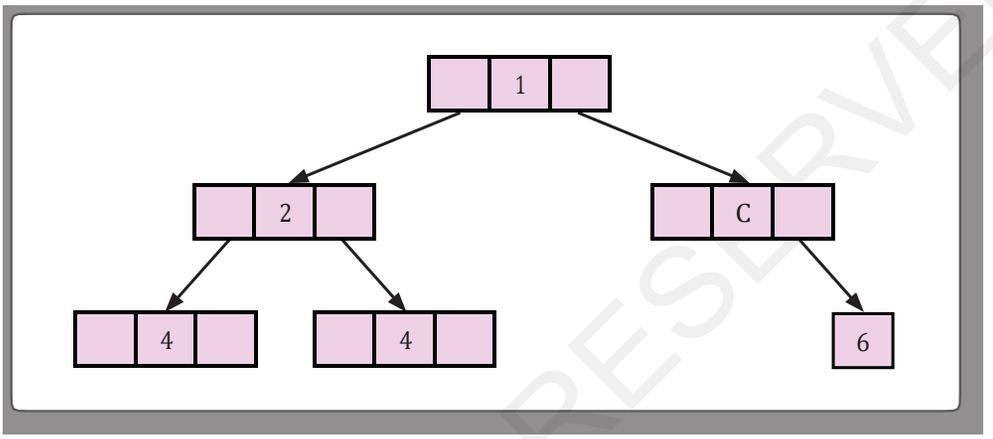


Figure 28: Threaded Binary Tree

A threaded binary tree is further divided into two parts as:

- Single threaded: In the single threaded binary tree only the NULL pointer of right subtree is pointed to in order successor as shown in Figure 29:

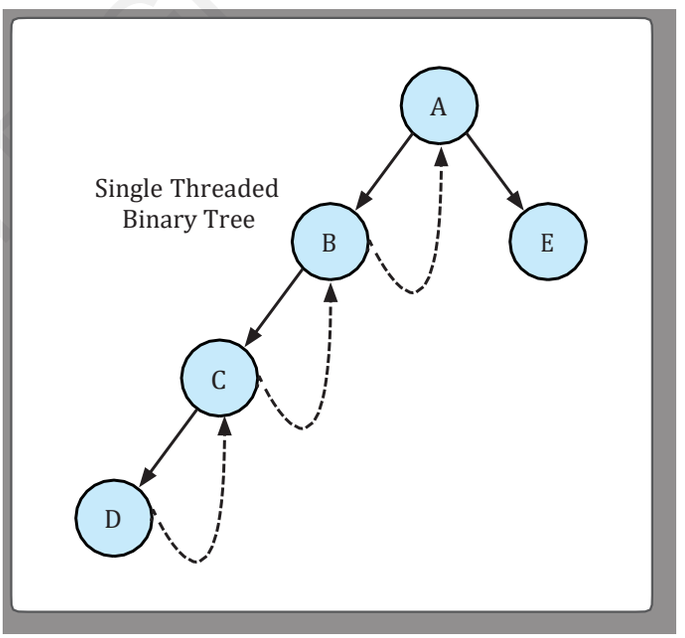


Figure 29: Single Threaded Binary Tree

- Double threaded: In the double threaded binary tree both the right and left NULL pointer pointed to in order successor or predecessor as shown in Figure 30:

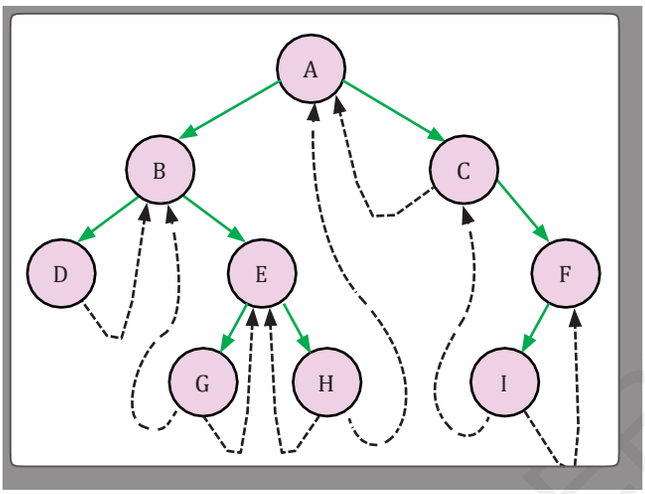


Figure 30: Double Threaded Binary Tree

8.11 BINARY SEARCH TREE

A binary search tree is a special type of binary tree that contains a definite order of elements in it. In a BST The elements of the left subtree has a lesser value then the parent node value whereas, the elements of the right subtree may have greater value then the parent node value. A BST is also known as sorted or ordered binary tree. It does not contain any duplicate nodes in a tree. A binary search tree is shown in Figure 31:

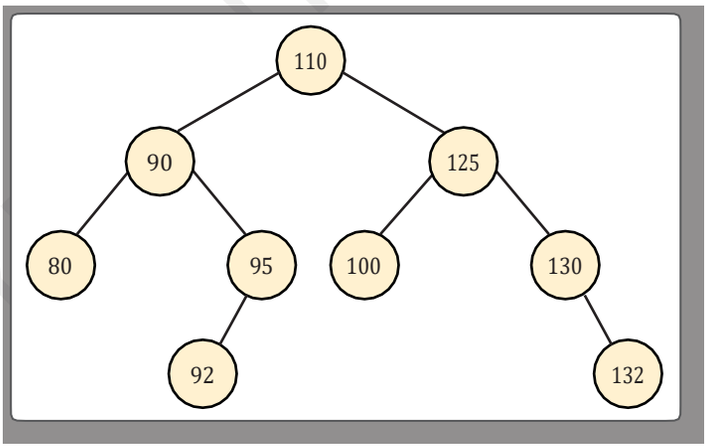


Figure 31: Binary Search Tree (BST)

Some of the most common operations in binary search tree are as follows:

- Insertion
- Deletion
- Searching

8.11.1 Insertion

This is the common operations used in BST used to insert values in a tree. While inserting values insides the tree make sure that the order of the values is not disturbed.

The following C++ program is used to perform insertion operation in BST are as follows:

```
// C++ program to demonstrate insertion
// in a Binary Search Tree recursively.
#include <iostream>
using namespace std;

class BST
{
    int Data;
   BST *Left, *Right;

public:
    BST();
    BST(int);
    BST* Insert(BST*, int);
    void Inorder(BST*);
};
BST ::BST()
    : Data(0)
    , Left(NULL)
    , Right(NULL)
{
}
BST ::BST(int Value)
{
    Data = Value;
    Left = Right = NULL;
}
BST* BST ::Insert(BST* Root, int Value)
{
    if (!Root)
    {
        // Insert the first node, if root is NULL.
        return new BST(Value);
    }
    if (Value > Root->Data)
    {
        Root->Right = Insert(Root->Right, Value);
    }
    else
    {
        Root->Left = Insert(Root->Left, Value);
    }
    return Root;
```

```
}
void BST ::Inorder(BST* Root)
{
    if (!Root) {
        return;
    }
    Inorder(Root->Left);
    cout << Root->Data << endl;
    Inorder(Root->Right);
}
int main()
{
    BST b, *Root = NULL;
    Root = b.Insert(Root, 30);
    b.Insert(Root, 35);
    b.Insert(Root, 28);
    b.Insert(Root, 56);
    b.Insert(Root, 68);
    b.Insert(Root, 75);
    b.Insert(Root, 99);
    b.Inorder(Root);
    return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/52OYpsDV3o.o
28
30
35
56
68
75
99
```

8.11.2 Deletion

In a binary search tree deletion operation is used when we require removing a node from the tree. In the deletion operation firstly identifies the node and its location for deletion. A node should be a lead node. The following C++ program is used to perform deletion operation in BST

```
#include<stdio.h>
#include<stdlib.h>
struct Node{
   int key;
   struct Node *Left, *Right;
};
struct Node *newNode(int item){
   struct Node *Temperature = (struct Node *)malloc(sizeof(struct Node));
   Temperature->key = item;
   Temperature->Left = Temperature->Right = NULL;
   return Temperature;
```

```
}
void inordertraversal(struct Node *root){
   if (root != NULL){
      inordertraversal(root->Left);
      printf("%d ", root->key);
      inordertraversal(root->Right);
   }
}
struct Node* insert(struct Node* node, int key){
   if (node == NULL) return newNode(key);
      if (key < node->key)
         node->Left = insert(node->Left, key);
      else
         node->Right = insert(node->Right, key);
   return node;
}
struct Node * minValueNode(struct Node* node){
   struct Node* current = node;
   while (current && current->Left != NULL)
      current = current->Left;
   return current;
}
struct Node* deleteNode(struct Node* root, int key){
   if (root == NULL) return root;
      if (key < root->key)
         root->Left = deleteNode(root->Left, key);
      else if (key > root->key)
         root->Right = deleteNode(root->Right, key);
   else{
      if (root->Left == NULL){
         struct Node *Temperature = root->Right;
         free(root);
         return Temperature;
      }
      else if (root->Right == NULL){
         struct Node *Temperature = root->Left;
         free(root);
         return Temperature;
      }
      struct Node* Temperature = minValueNode(root->Right);
      root->key = Temperature->key;
      root->Right = deleteNode(root->Right, Temperature->key);
   }
   return root;
}
int main(){
   struct Node *root = NULL;
   root = insert(root, 25);
   root = insert(root, 45);
   root = insert(root, 28);
```

```
   root = insert(root, 55);
   root = insert(root, 67);
   root = insert(root, 78);
   root = insert(root, 88);
   printf("Inorder traversal of a tree \n");
   inordertraversal(root);
   printf("\nDelete 20\n");
   root = deleteNode(root, 55);
   printf("Inorder traversal of tree after modification \n");
   inordertraversal(root);
   printf("\nDelete 30\n");
   root = deleteNode(root, 78);
   printf("Inorder traversal of tree after modification \n");
   inordertraversal(root);
   printf("\nDelete 50\n");
   root = deleteNode(root, 25);
   printf("Inorder traversal of tree after modification \n");
   inordertraversal(root);
   return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/52OYpsDV3o.o
Inorder traversal of a tree
25 28 45 55 67 78 88
Delete 20
Inorder traversal of tree after modification
25 28 45 67 78 88
Delete 30
Inorder traversal of tree after modification
25 28 45 67 88
Delete 50
Inorder traversal of tree after modification
28 45 67 88
```

8.11.3 Searching

Searching is the process of identifying a particular element or item as “key” in BST. It is one of the easiest operations among others as we do not need to search the entire tree. We just have to match the key in BST.

The following C++ program is used to perform the delete operation in BST are as follows:

```
#include<stdio.h>
#include<stdlib.h>
struct node{
   int key;
   struct node *Left, *Right;
};
struct node *newNode(int item){
   struct node *temperature = (struct node *)malloc(sizeof(struct node));
```

```
   temperature->key = item;
   temperature->Left = temperature->Right = NULL;
   return temperature;
}
void traversetree(struct node *root){
   if (root != NULL){
      traversetree(root->Left);
      printf("%d \t", root->key);
      traversetree(root->Right);
   }
}
struct node* search(struct node* root, int key){
   if (root == NULL || root->key == key)
      return root;
   if (root->key < key)
      return search(root->Right, key);
   return search(root->Left, key);
}
struct node* insert(struct node* node, int key){
   if (node == NULL) return newNode(key);
      if (key < node->key)
         node->Left = insert(node->Left, key);
      else if (key > node->key)
         node->Right = insert(node->Right, key);
   return node;
}
int main(){
   struct node *root = NULL;
   root = insert(root, 23);
   insert(root, 10);
   insert(root, 18);
   insert(root, 22);
   insert(root, 34);
   insert(root, 47);
   insert(root, 87);
   printf("The elements in the tree is :\n");
   traversetree(root);
   printf("\nSearching for 34 in this tree ");
   if(search(root , 34))
      printf("\nelement found");
   else
      printf("\nelement not found");
   return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/52OYpsDV3o.o
The elements in the tree is :
10    18    22    23    34    47    87
```

```
Searching for 34 in this tree
element found
```

8.12 APPLICATION OF TREES

The following points describe the applications of trees are as follows:

- It is utilized in maintaining the structure of the files, any company,the XML HTML information as it holds the information hierarchically.
- Heap is a is a specialized tree utilized to manipulate the arrays and priority queues.
- B-Tree and B+ Tree are utilised to create and manipulate the catalogues in the databases.
- Syntax Tree finds its application in Compilers.
- K-D Tree is a tree which partitions the space is utilized to arrange points in space with N dimensions.
- Tries are utilised to manipulate dictionaries with options to find the prefixes.
- Suffix Trees are used for faster pattern recognition in a String.
- The Smallest path trees and the spanning trees find the application in routers and bridges existing in computer networks.

8.13 EVALUATION OF EXPRESSION

In a binary tree, a tree in which internal node link to the operator and each leaf node link to the operand. For example an expression is 4 + ((9+3)*5) is shown in Figure 32:

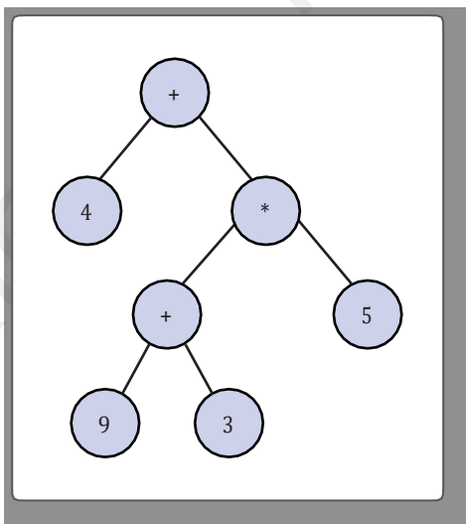


Figure 32: Expression Tree

Lab Exercise

Write a Program in C++ to implement Binary Search Tree (BST).

The following C++ program is used to implement the binary search tree are as follows:

```
#include<stdio.h>
#include<stdlib.h>
struct node{
   int data;
```

```
   struct node *Left;
   struct node *Right;
};
void Inordertraversal(struct node* node, int inorder[], int *index_ptr){
   if (node == NULL)
      return;
   Inordertraversal(node->Left, inorder, index_ptr);
   inorder[*index_ptr]  =  node->data;
   (*index_ptr)++;
   Inordertraversal(node->Right, inorder, index_ptr);
}
int countNodes(struct node* Root){
   if (Root == NULL)
      return 0;
   return countNodes (Root->Left) +
   countNodes (Root->Right) + 1;
}
int compare (const void * a, const void * b){
   return( *(int*)a - *(int*)b );
}
void arrayToBST (int *arr, struct node* Root, int *index_ptr){
   if (Root == NULL)
      return;
   arrayToBST (arr, Root->Left, index_ptr);
   Root->data = arr[*index_ptr];
   (*index_ptr)++;
   arrayToBST (arr, Root->Right, index_ptr);
}
struct node* newNode (int data){
   struct node *temperature = new struct node;
   temperature->data = data;
   temperature->Left = NULL;
   temperature->Right = NULL;
   return temperature;
}
void printInorder (struct node* node){
   if (node == NULL)
      return;
   printInorder (node->Left);
   printf("%d ", node->data);
   printInorder (node->Right);
}
int main(){
   struct node *Root = NULL;
   Root = newNode(20);
   Root->Left = newNode(23);
   Root->Right = newNode(20);
   Root->Left->Left = newNode(18);
```

```
    Root->Right->Right = newNode(9);
    printf("Inorder Traversal of the binary Tree: \n");
    printInorder (Root);
    int n = countNodes(Root);
    int *arr = new int[n];
    int j = 0;
    Inordertraversal(Root, arr, &j);
    qsort(arr, n, sizeof(arr[0]), compare);
    j = 0;
    arrayToBST (arr, Root, &j);
    delete [] arr;
    printf("\nInorder Traversal of the converted BST: \n");
    printInorder (Root);
    return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/Vg1XaDmN4d.o
Inorder Traversal of the binary Tree:
18 23 20 20 9
Inorder Traversal of the converted BST:
9 18 20 20 23
```

8.14 CONCLUSION

- A Distinct data structure where hierarchical relationships exist among the data items or nodes is referred as Trees.
- The nodes of a tree are connected with one another in a tree using “edges”.
- A root node is an exclusive node in the tree.
- A binary tree is a distinct type of tree in which a parent node has exactly two child nodes.
- A complete binary tree is a tree in which each and every level of a tree is completely filled except the last level.
- Representing a binary tree using array firstly we require converting that binary tree into a full binary tree.
- Binary tree is represented through linked list is stored in the memory as linked lists.
- Binary tree traversal in data structure is the process of visiting, updating and verifying once each node of a tree.
- An in-order traversal, always start visiting from the left subtree.
- post-order traversal the process of visiting starts from left subtree of a tree, then it visit to the right subtree.
- Pre-order traversal is just opposite to the post-order traversal, in which firstly we visit to the root node.
- Threaded binary tree helps to reuse that vacant links again by making some threads.
- A binary search tree is a special type of binary tree that contains a definite order of elements in it.

8.15 GLOSSARY

- Tree: It is a distinct data structure where hierarchical relationships exist among the data items.
- Edge: The nodes of a tree are connected with one another in a tree using “edges”.
- Binary tree: It is a distinct type of tree in which a parent node has exactly two child nodes.
- Complete binary tree: It is a tree in which each and every level of a tree is completely filled except the last level.
- Binary tree traversal: It is the process of visiting, updating and verifying once each node of a tree.
- In-order traversal: It always starts visiting from the left subtree.
- Post-order traversal: It is the process of visiting starts from left subtree of a tree, then it visit to the right subtree.
- Pre-order traversal: It is just opposite to the post-order traversal, in which firstly we visit to the root node.
- Threaded binary tree: It helps to reuse the vacant links again by making some threads.
- Binary search tree (BST): It is a special type of binary tree that contains a definite order of elements in it.
- Path: It is the way or edges consisting a series of nodes required to be visited to reach to any specific node from the source node.
- Root : It is the originating node from which other nodes emerges. There is always one root in any tree.
- Parent node: It is the alternative nodes other than the root node, which have sub nodes or child.
- Child node: It is the sub node connected to of origin through an edge.
- Leaf node: It is the node that lacks a child node.

8.16 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. A Distinct data structure where hierarchical relationships exist among the data items. What do you understand by the tree in data structure?
2. In a Tree, Every distinct element is known as node. Describe the concept of tree terminology in brief.
3. Completing of each level determines that every parent node should have exactly two child node. Determine the concept of complete binary tree and how it differs from binary tree.
4. Binary tree traversal in data structure is the process of visiting and updating each node. What do you understand by binary tree traversal? Also, discuss its distinct ways for traversing a tree in brief.
5. The threaded binary tree helps to reuse that vacant links again by making some threads. Explain threaded binary tree and its types in detail.

8.17 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. A Distinct data structure where hierarchical relationships exist among the data items or nodes is referred as Trees. It is a non-linear hierarchical data structure which includes a group of objects called as nodes. Refer to Section Introduction
2. In a Tree, Every distinct element is known as node. Node in a tree data structure stores the actual data of that specific element and connects to next element in hierarchical manner. Refer to Section Tree Terminology
3. As a binary tree, a complete binary tree is a tree in which each and every level of a tree is completely filled except the last level of a complete binary tree. Refer to Section Complete Binary Tree
4. It is the process of visiting, updating and verifying once each node of a tree. Linear data structures such as arrays, stacks, queues, and linked list possess uni-directional paths to scan the information. Refer to Section Binary Tree Traversal
5. If a vacant left or right child area one node has then it will be used as thread. Refer to Section Threaded Binary Tree

8.18 POST-UNIT READING MATERIAL

- http://btechsmartclass.com/data_structures/tree-terminology.html
- https://www.iare.ac.in/sites/default/files/DS.pdf

8.19 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates about the concept of trees in data structure and its types. Also, discuss about the application of trees, different operations of binary tree and the difference between complete and full binary tree.

UNIT

09

Advanced Concepts in Trees

Names of Sub-Units

Introduction to Advanced Trees and Application of Trees: AVL Tree, B Tree, B+ Tree, Red-Black Trees

Overview

This unit begins by discussing the concept of advanced trees. Next, the unit discusses the application of trees. Further, the unit explains AVL trees, B tree and B+ tree. Towards the end, the unit discusses the red-black trees.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of advanced trees
- Explain the importance of application of trees
- Describe the AVL and B tree
- Explain the significance of B+ trees
- Summarise the concept of red-black trees

Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of advanced trees
- Assess the importance of application of trees
- Evaluate the importance of AVL and B tree
- Determine the significance of B+ trees
- Explore the concept of red-black trees

	Pre-Unit Preparatory Material
⌘	https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap5.pdf

9.1 INTRODUCTION

Tree is one of the most significant data structures which is used for efficiently executing operations such as insertion, deletion and searching of values. Though while working with a data of huge size, it is not possible to create a well-balanced tree for sorting all the data. Thus, only valuable data is kept as a tree and the definite size of data being used frequently changes through the insertion of new data and deletion of current data. You will find in some cases where the NULL link to a binary tree to distinct links is called as threads and hence it is probable to implement traversals, insertions, deletions without using either stack or recursion.

Linked list offer higher benefits in terms of adaptability as compared to the adjacent data representation offered by the other data structures. However, there are many drawbacks as well. The tree offers much versatile data storage techniques. The worst case execution of a binary search tree is similar to the linear search program execution, O(n). It is difficult to anticipate the pattern and its periodicity in the concrete data. Hence we must balance the binary search trees.

9.2 AVL TREE

In a data structure AVL tree is defined as height balanced binary search tree (BST) in which each individual node is connected with a balance factor and the difference of height of right and left subtree is less than or equal to one by deducting the height of its right from that of its left subtree. This method of balancing in AVL tree was exposed by Adelson, Velshi and Landis provide the short name as AVL tree or balanced binary tree. When the balance factor of each node of right and left subtree of a tree is between -1 to 1, it said to be a balanced tree if not it is considered as unbalanced. The balanced and unbalanced tree is shown in Figure 1:

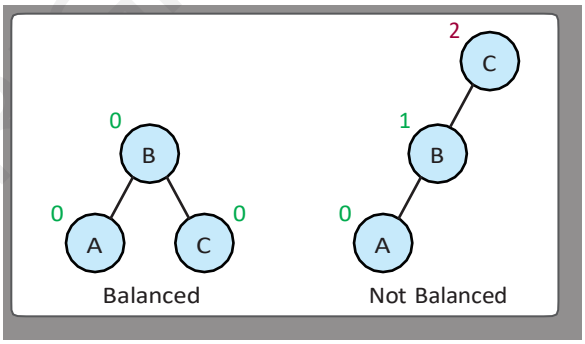


Figure 1: Balanced and Unbalanced Tree

From Figure 1 considering the second tree, the height is 2, the balance factor is 2. To make the tree AVL tree a balance factor of 1 should be maintained as:

Balance Factor = height of (left-subtree) – height of (right-subtree)

If the balance factor becomes greater than 1, the tree is stabilized through the rotation processes. For auto balancing, an AVL tree might carry out four types of rotations are as follows:

- Left rotation
- Right rotation

- Left-right rotation
- Right-left rotation

9.2.1 Left Rotation

During insertion of a node in the right subtree if a an AVL tree gets unbalanced then one left rotation is carried out as shown in Figure 2:

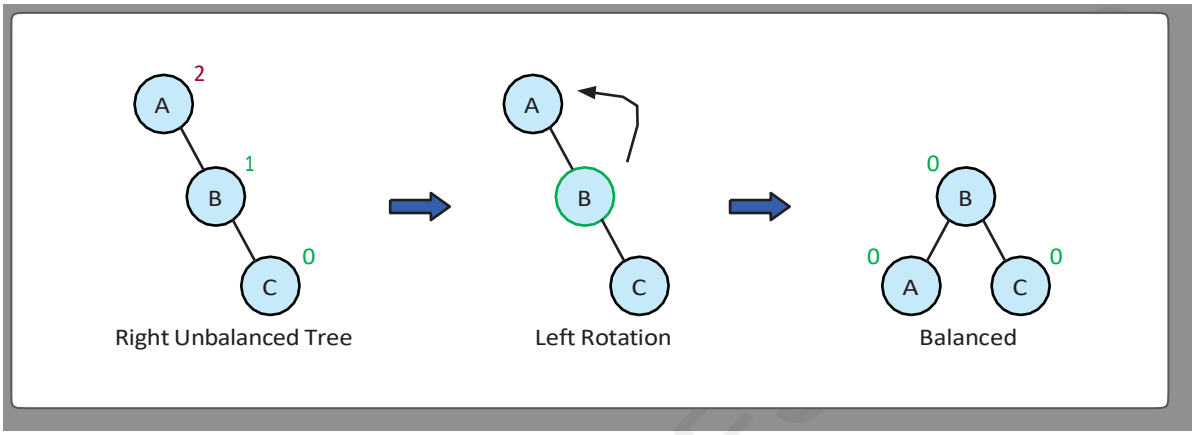


Figure 2: Left Rotation

In the Figure 2, the tree is balanced by performing left rotation, where A becomes the left subtree of B.

9.2.2 Right Rotation

During the insertion of a node in the left subtree if an AVL tree gets unbalanced then a right rotation is carried out, as shown in Figure 3:

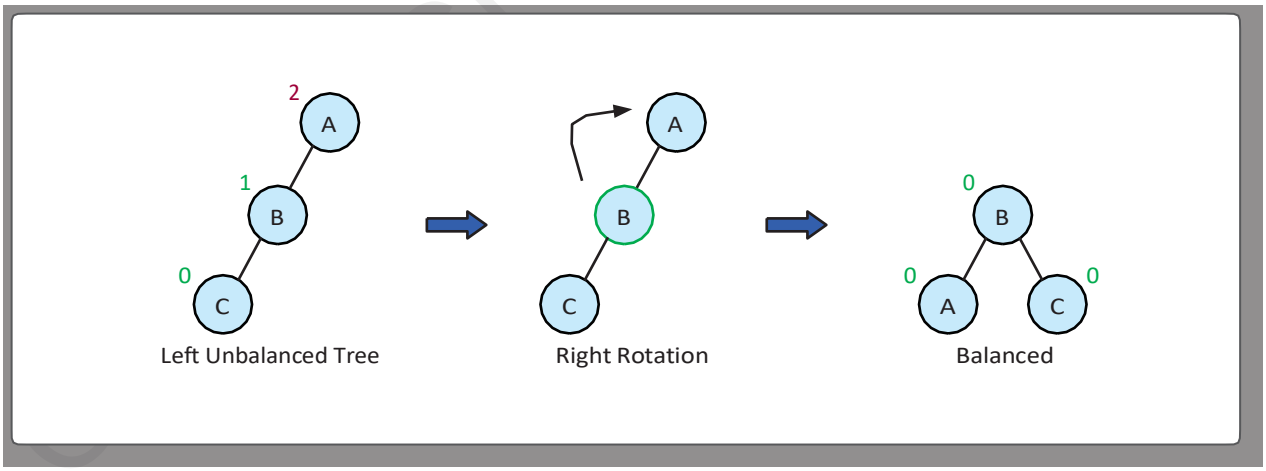


Figure 3: Right Rotation

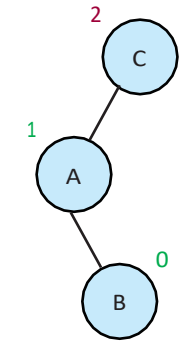
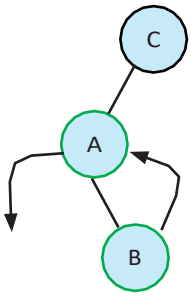
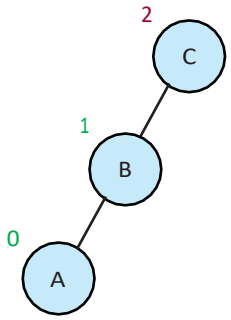
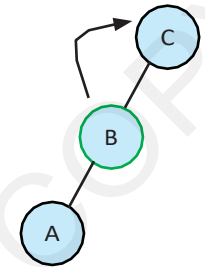
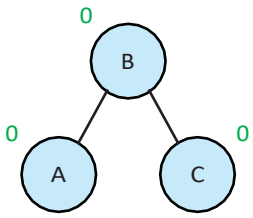
As shown in Figure 3 above the AVL tree is balanced by carrying out right rotation, pushing C to the right subtree.

9.2.3 Left-Right Rotation

Double rotations are at intricate variants of previous kinds of rotations.

A left right rotation is a fusion of left rotation followed by right rotation, as shown in Table 1:

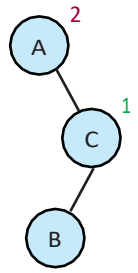
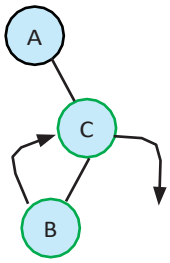
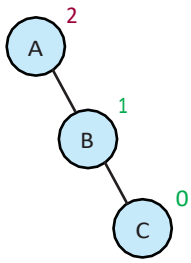
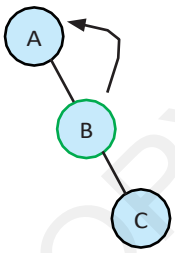
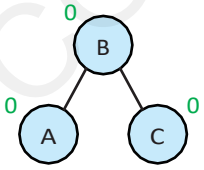
Table 1: Fusion of Left-Right Rotation

State	Action
	There was an addition of a node in the right side of the left subtree,which leaves C unstabilised. To make the AVL tree completely balance we need to perform left right rotation.
	Initially a left rotation is carried out on the left subtree of C, which designates A as the left subtree of B.
	Node C is not stabilised yet, because the height of the tree is 2.
	Now, a right rotation is performed on the tree, which designates B as the root and makes C as the right node of the left subtree.
	Thus, we have balanced the AVL tree.

9.2.4 Right-Left Rotation

This double rotation is a fusion of right rotation, followed by left rotation as shown in Table 2:

Table 2: Fusion of Right-Left Rotation

State	Action
	There was a node addition to the left of the right subtree, making A as an unstabilised node with height 2.
	Initially a right rotation is carried out on node C, designating C as the right node of the subtree B and making B as right Child of A.
	Node A is unsterilized as the height is 2.
	We carry out a left rotation on node B, designating B as the root and making A as the left child of B.
	The given AVL tree is stabilised or balanced.

9.3 B TREE

B tree is referred as self-balanced search tree. In order to analyse the mechanism of B trees, let us assume was quantity of information which cannot be held in the main memory. Hence, the additional data is stored in the disks. The data retrieval time span of disk is greater than main memory. The technique is

to use B trees to minimize the disk visits. Majority of the functions on trees, such as searching, insertion, deletion, finding maximum, minimum, etc., need O(h) disk visits, where h indicates height of the tree. The height of B-Trees is retained as minimum by keeping greatest probable keys in any node. In general the capacity of a node in B tree is retained as identical to the capacity of the disk block. With lower heights the maximum number of disc visits is also minimised in contrast to balanced binary search trees such as AVL and red-black trees.

Time Complexity of B-Tree is shown in Table 3:

Table 3: Time Complexity of B Tree

S. No.	Algorithm	Time Complexity
1.	Searching	O(log n)
2.	Insertion	O(log n)
3.	Deletion	O(log n)

The following points describe the characteristics of B tree are as follows:

- Entire leaf nodes share the identical level.
- A B tree is designated by the lowest degree's' whose value relies on the capacity of disk.
- All the nodes excluding the root compulsorily consists of minimum (ceiling) ([t-1]/2) keys. The root should consist of at least 1 key.
- Entire nodes might consist of utmost t – 1 key.
- The total count of child nodes of an individual node is identical to the sum of total keys and 1.
- Entire set of keys are arranged in ascending order. A child node amidst of two keys Key1 & Key2 consists of the keys ranging from Key1 to Key2.
- In contrast to the BST, B-Tree expands and contracts through the root. A BST expands bottomward and contracts from below as well.
- In B-trees, time complexity for searching, insertion and deletion is O(log n).
- It is possible to add a node in B-Tree at Leaf Node alone.

Figure 4: depicts the B tree of order 5

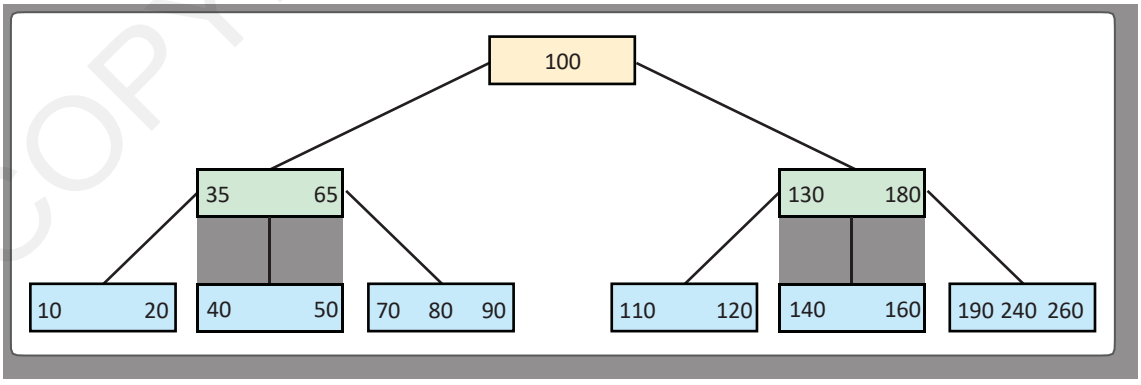


Figure 4: B Tree of Order 5

From the above figure it is clear that entire leaf nodes are at the identical level, entire non-leaf nodes have occupied subtrees and with keys as total children – 1.

Some important points about the B tree facts are as follows:

- The lowest height of a B-Tree having n total nodes and m greatest count of child nodes which a node can possess is: $h_{min} = [\log_m (n+1)] - 1$.
- The greatest height of a B-Tree having n nodes and d number of least children a non-root node possess is: $h_{max} = [\log t\ n+1/2]$ and $t = [m/2]$.

9.3.1 B Tree Traversal

Exploring nodes of B tree resembles the In-order traversal of Binary Tree. We initiate from the extreme left child, recursively display it, then iterate the identical method for leftover child nodes and keys. Finally recursively display the extreme right child.

9.3.2 Search Operation in B Tree

The operation of finding elements in a B Tree resembles the search process of BST. Assume that we have to find the key K. Beginning from the root, we travel below recursively. If the seen non-leaf node, possess the key, the node is retained, else, we move bottomward towards relevant child (The child preceding the initial larger key) of the node. If no key is found in the leaf node, a NULL is retained.

In every individual level, the search is enhanced and if the key is not found in the ambit of parent then the chances of occurrence of key becomes higher in other branch. Upon arriving at the leaf node if the chosen key could not be located, then NULL is shown. An example of locating the key 120 in the B-Tree is shown in Figure 5:

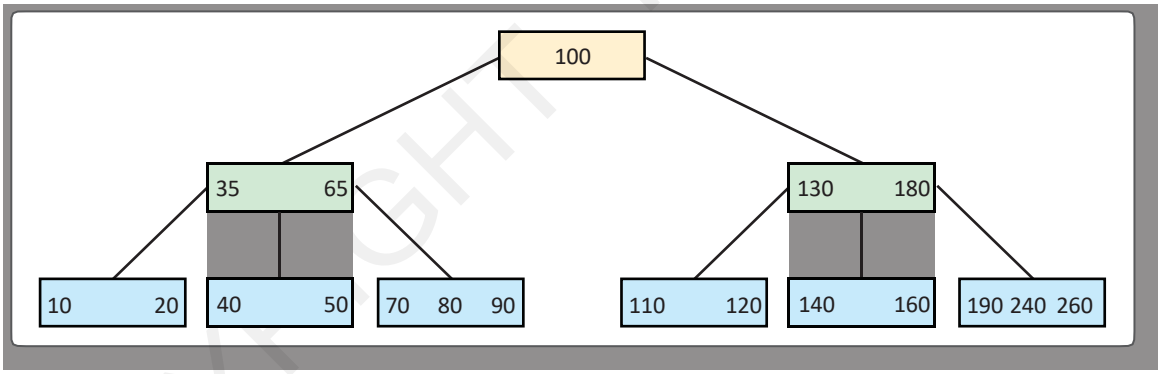


Figure 5: Locating Key 120 in B Tree

The step-by-step solution of locating key 120 in B tree start with root node as shown in Figure 6(a):

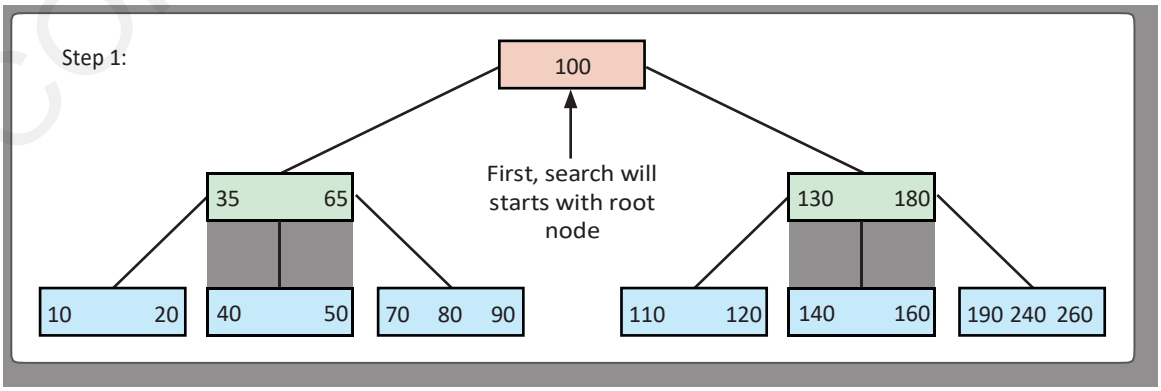


Figure 6(a): Start at Root Node

After the root node we will search the range where we insert the key 120 as shown in Figure 6(b):

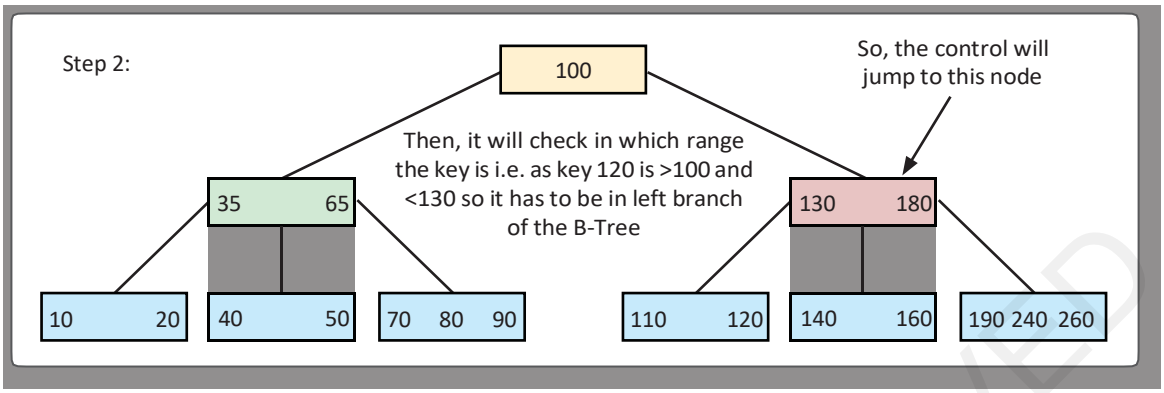


Figure 6(b): Identifying the Range of Key

After searching the range of a key in a tree it has to be in the rightmost child node of the current parent as shown in Figure 6(c):

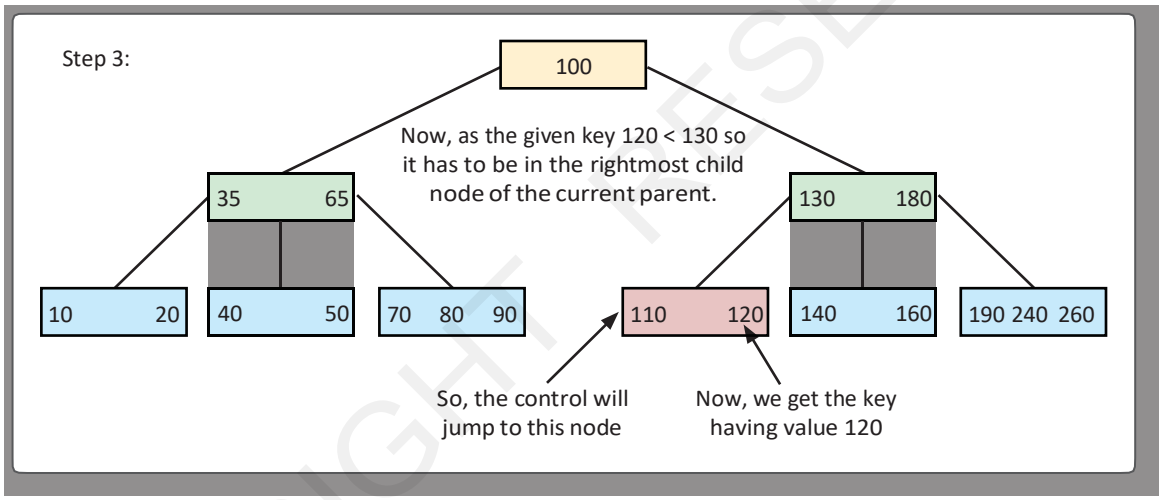


Figure 6(c): Getting the Key Having Value 120

In the above illustration, our scope of search was minimised by restricting ourselves to those possibilities where the occurrence of the key value is supposed to be higher. If in the Figure 6(c), then we try to find 180, then the algorithm will halt because of its presence in the current node. If we try to look for 90 the program flow will be directed to the left subtree.

9.4 B+ TREES

B+ tree augments B Tree permitting much organised data addition, removal and search functionalities. In B Tree, both the interior and leaf nodes can hold the key values and information. Contrasting to this, in B+ tree, the information is held at the leaf nodes and the key values are held in the interior nodes. In B+ tree the leaf nodes are interconnected, establishing a singly linked list making search inquiries much organised.

B+ tree are utilised to hold the huge quantity of information for which the main memory is insufficient, because of the restricted storage capacity. The interior nodes (holding key values to fetch data) of the B+ tree are held in the main memory but the leaf nodes are reserved in the auxiliary memory.

We refer the interior nodes of B+ tree as index nodes of order 3 as shown in Figure 7:

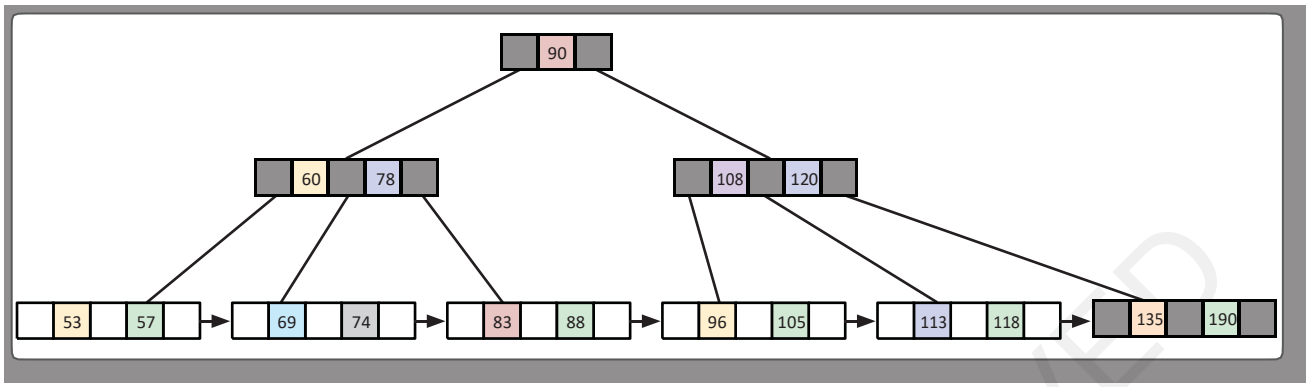


Figure 7: B+ Tree of Order 3

The following points describe the benefits of B+ tree in data structure are as follows:

- Data is retrieved with equal count of disk visits.
- We can always find B+ tree with balanced height in contrast to B tree.
- The information could be retrieved from a B+ tree serially and straight away.
- We can make use of the Keys for cataloguing.

The comparison between B+ tree and B tree is shown in Table 4:

Table 4: Comparison of B + Tree with B Trees

S. No.	B Tree	B+ Tree
1.	Only unique search key values are reserved.	Superfluous search keys can exist.
2.	Information is held either in leaf nodes or interior nodes.	Information is held in leaf nodes.
3.	Locating certain key values or information could be prolonged as data is held in the interior and leaf nodes.	Rapid search results contrast to B-trees, since information is located in leaf nodes.
4.	Removal of interior nodes is complex and requires larger time span.	Removal of any node is simple as all the nodes are leaf nodes.
5.	It is not possible to interconnect the leaf nodes.	To make the search operations well organised, leaf nodes are interconnected.

9.5 RED-BLACK TREES

Red-black tree is referred to as a self-balanced tree, where the individual nodes possess an additional bit indicating the colour – black or red. The colours are utilised for assuring that the tree is kept balanced during data addition and removal. The time to search an element is O(log n), where n refers to the overall count of data values in the tree. Rudolf Bayer devised the tree in 1972.

We must underline that every node needs single bit of memory to hold the data of colour. Such categories of trees display alike memory traces contrast to the vintage (colorless) BST. Canons to be complied by red-black tree are as follows:

- All the individual nodes must possess colour either red or black.
- The root node is certainly black.

- We cannot have two adjoining or adjacent red nodes, which mean a red node can never possess either a red parent or red child).
- All the ways from an individual node (involving root nodes) to each of its successor NULL nodes have the identical count of black nodes.

9.5.1 Importance of Red-Black Tree

Majority of the binary search tree functionalities such as search, maximum, minimum, insertion, removal require O(h) time where h indicates height of the BST. Such operations require O(n) time, in case of skewed Binary tree. If it is ensured that the height of the tree is kept as O(log n) after consecutive data addition and removal, then an upper bound of O(log n) is assured for all such functionalities. We have O(log n) as height consistent height of red-black tree., where n stands for the count of nodes in the tree. The time complexity of red-black tree is shown in Table 5:

Table 5: Time Complexity of Red Black Tree

S. No.	Algorithm	Time Complexity
1.	Searching	O(log n)
2.	Insertion	O(log n)
3.	Deletion	O(log n)

The AVL trees are much stabilized or balanced in contrast to red-black trees, but there might be multiple whirls throughout data addition and Removal. In case of our program entails recurrent data additions and Removal, then we should favour the red black trees, whereas if instead data additions and removal are less recurrent and data search operation is persistent, then one should favor AVL tree above red-black tree.

9.5.2 Balancing in Red Black Tree

To ensure your balancing we must consider the fact that, a series of 3 nodes is impossible in the red-black tree. One can choose any blend of colours and check whether the entire set of nodes breach the canons of Red-Black tree. An example of red black tree where a series of 3 nodes is impossible in red-black trees, as shown in Figure 8:

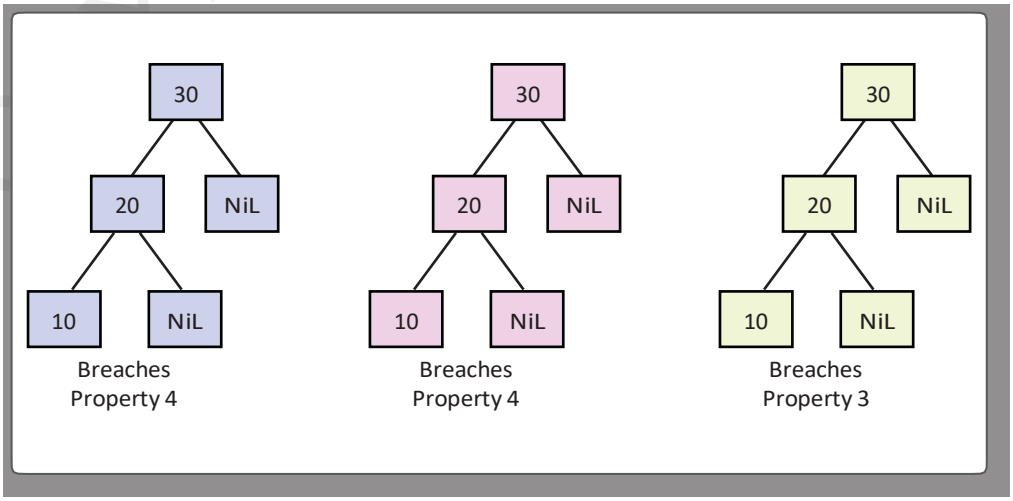


Figure 8: Red Black Tree is not Balanced

Figure 9 shows various probable red-black trees made using above 3 keys as:

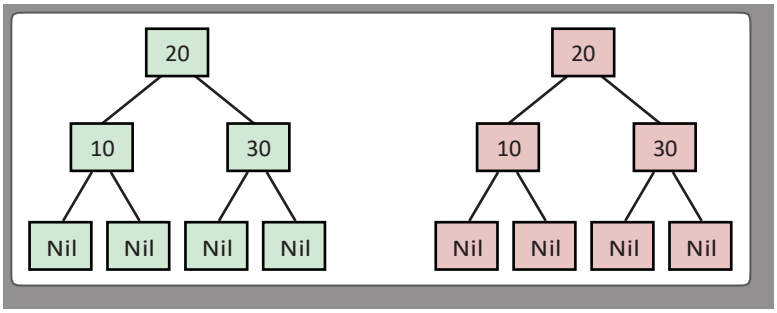


Figure 9: Balancing in Red-Black Tree

The compelling facts about Red-Black Tree are as follows:

- Black height of the red-black tree refers to the count of black nodes lying on the way, from the root of the tree to any leaf node. Leaf nodes are considered as black nodes as well. The black height of a red-black tree with height h is >= h/2.
- Height of a red-black tree having n nodes is given by $h <= 2 \log_2 (n + 1)$.
- Entire leaf nodes are black.
- The black depth of a node is referred as the count of black nodes encountered from the root to the specified node, in other words the count of black progenitors.
- All the red-black trees are specialised binary trees.

The height of any red black tree having n nodes has height $<= 2\text{Log}_2 (n+1)$. This can be verified using the following points as:

- In a common Binary Tree, if k is the least count of nodes present on all the ways from root to NULL nodes, then $n >= 2^k - 1$ (Eg. If k holds 3, then n is minimum 7). This can also be expressed as $k <= \text{Log}_2 (n+1)$.
- Referring Rule 4 of Red-Black trees and above fact, it could be generalized that in a Red-Black Tree having n nodes, there exists a way from root to leaf node having utmost $\text{Log}_2 (n+1)$ black nodes.
- Referring Rule 3 of Red-Black trees,it is asserted that the count of black nodes in a Red-Black tree is minimum (n/2) where n indicates the overall count of nodes.

Thus we can deduce that a red black tree having n nodes has height $<= 2\text{Log}_2 (n+1)$.

Lab Exercise 9a:

Write a C++ Program to implement an AVL Tree.

The following C++ program is used to implement an Avl tree is as follows:

```
// Implementation of AVL tree in C++
#include <iostream>
using namespace std;
class Node {
  public:
 int Key;
 Node *Left;
 Node *Right;
```

```
 int height;
};
int max(int x, int y);
// Calculate height
int height(Node *N) {
 if (N == NULL)
  return 0;
 return N->height;
}
int max(int x, int y) {
 return (x > y) ? x : y;
}
// New node creation
Node *newNode(int key) {
 Node *node = new Node();
 node->Key = key;
 node->Left = NULL;
 node->Right = NULL;
 node->height = 1;
 return (node);
}
// Rotate right
Node *rightRotate(Node *y) {
 Node *x = y->Left;
 Node *T2 = x->Right;
 x->Right = y;
 y->Left = T2;
 y->height = max(height(y->Left),
     height(y->Right)) +
    1;
 x->height = max(height(x->Left),
     height(x->Right)) +
    1;
 return x;
}
Node *leftRotate(Node *x) {
 Node *y = x->Right;
 Node *T2 = y->Left;
 y->Left = x;
 x->Right = T2;
 x->height = max(height(x->Left),
     height(x->Right)) +
    1;
 y->height = max(height(y->Left),
     height(y->Right)) +
    1;
 return y;
}
int getBalanceFactor(Node *N) {
 if (N == NULL)
```

```
  return 0;
 return height(N->Left) -
    height(N->Right);
}
// Inseerting a node
Node *insertNode(Node *node, int key) {
 // insert a node on correct position
 if (node == NULL)
  return (newNode(key));
 if (key < node->Key)
  node->Left = insertNode(node->Left, key);
 else if (key > node->Key)
  node->Right = insertNode(node->Right, key);
 else
  return node;
 node->height = 1 + max(height(node->Left),
        height(node->Right));
 int balanceFactor = getBalanceFactor(node);
 if (balanceFactor > 1) {
  if (key < node->Left->Key) {
   return rightRotate(node);
  } else if (key > node->Left->Key) {
   node->Left = leftRotate(node->Left);
   return rightRotate(node);
  }
 }
 if (balanceFactor < -1) {
  if (key > node->Right->Key) {
   return leftRotate(node);
  } else if (key < node->Right->Key) {
   node->Right = rightRotate(node->Right);
   return leftRotate(node);
  }
 }
 return node;
}
Node *nodeWithMimumValue(Node *node) {
 Node *current = node;
 while (current->Left != NULL)
  current = current->Left;
 return current;
}
// delete a node from tree
Node *deleteNode(Node *root, int key) {
 // Find the node and delete it
 if (root == NULL)
  return root;
 if (key < root->Key)
  root->Left = deleteNode(root->Left, key);
```

```
 else if (key > root->Key)
  root->Right = deleteNode(root->Right, key);
 else {
  if ((root->Left == NULL) ||
   (root->Right == NULL)) {
   Node *temperature = root->Left ? root->Left : root->Right;
   if (temperature == NULL) {
    temperature = root;
    root = NULL;
   } else
    *root = *temperature;
   free(temperature);
  } else {
   Node *temperature = nodeWithMimumValue(root->Right);
   root->Key = temperature->Key;
   root->Right = deleteNode(root->Right,
          temperature->Key);
  }
 }
 if (root == NULL)
  return root;
 // Balancing a tree
 root->height = 1 + max(height(root->Left),
        height(root->Right));
 int balanceFactor = getBalanceFactor(root);
 if (balanceFactor > 1) {
  if (getBalanceFactor(root->Left) >= 0) {
   return rightRotate(root);
  } else {
   root->Left = leftRotate(root->Left);
   return rightRotate(root);
  }
 }
 if (balanceFactor < -1) {
  if (getBalanceFactor(root->Right) <= 0) {
   return leftRotate(root);
  } else {
   root->Right = rightRotate(root->Right);
   return leftRotate(root);
  }
 }
 return root;
}
// Display the balanced tree
void printTree(Node *root, string indent, bool last) {
 if (root != nullptr) {
  cout << indent;
  if (last) {
   cout << "Right ---";
```

```
    indent += "   ";
   } else {
    cout << "Left ---";
    indent += "|  ";
   }
   cout << root->Key << endl;
   printTree(root->Left, indent, false);
   printTree(root->Right, indent, true);
 }
}
int main() {
 Node *root = NULL;
 root = insertNode(root, 12);
 root = insertNode(root, 30);
 root = insertNode(root, 46);
 root = insertNode(root, 67);
 root = insertNode(root, 22);
 root = insertNode(root, 65);
 root = insertNode(root, 55);
 root = insertNode(root, 7);
 printTree(root, "", true);
 root = deleteNode(root, 22);
 cout << "After deleting " << endl;
 printTree(root, "", true);
}
```

The output of given C++ code is as follows:

```
/tmp/QQIgPOAw1Y.o
Right --- 30
   Left --- 12
   |  Left --- 7
   |  Right --- 22
   Right --- 65
    Left --- 46
    |  Right --- 55
    Right --- 67
After deleting
Right --- 30
   Left --- 12
   |  Left --- 7
   Right --- 65
Left --- 46
    |  Right --- 55
    Right --- 67
```

Lab Exercise 9b:

Write a C++ Program to implement a B-Tree.

The following C++ program is used to implement a B tree is as follows:

```
// Searching a key on a B-tree in C++
```

```
#include <iostream>
using namespace std;
class TreeNode {
 int *Keys;
 int T;
 TreeNode **C;
 int N;
 bool leaf;
  public:
 TreeNode(int temperature, bool bool_leaf);
 void insertNonFull(int x);
 void splitChild(int i, TreeNode *y);
 void traverse();
 TreeNode *search(int x);
 friend class BTree;
};
class BTree {
 TreeNode *root;
 int t;
  public:
 BTree(int temp) {
  root = NULL;
  t = temp;
 }
 void traverse() {
  if (root != NULL)
   root->traverse();
 }
 TreeNode *search(int x) {
  return (root == NULL) ? NULL : root->search(x);
 }
 void insert(int x);
};
TreeNode::TreeNode(int t1, bool leaf1) {
 T = t1;
 leaf = leaf1;
 Keys = new int[2 * T - 1];
 C = new TreeNode *[2 * T];
 N = 0;
}
void TreeNode::traverse() {
 int i;
 for (i = 0; i < N; i++) {
  if (leaf == false)
   C[i]->traverse();
  cout << " " << Keys[i];
 }
 if (leaf == false)
  C[i]->traverse();
```

```
}
TreeNode *TreeNode::search(int x) {
 int i = 0;
 while (i < N && x > Keys[i])
  i++;
 if (Keys[i] == x)
  return this;
 if (leaf == true)
  return NULL;
 return C[i]->search(x);
}
void BTree::insert(int x) {
 if (root == NULL) {
  root = new TreeNode(t, true);
  root->Keys[0] = x;
  root->N = 1;
 } else {
  if (root->N == 2 * t - 1) {
   TreeNode *s = new TreeNode(t, false);
   s->C[0] = root;
   s->splitChild(0, root);
   int i = 0;
   if (s->Keys[0] < x)
    i++;
   s->C[i]->insertNonFull(x);
   root = s;
  } else
   root->insertNonFull(x);
 }
}
void TreeNode::insertNonFull(int x) {
 int i = N - 1;
 if (leaf == true) {
  while (i >= 0 && Keys[i] > x) {
   Keys[i + 1] = Keys[i];
   i--;
  }
  Keys[i + 1] = x;
  N = N + 1;
 } else {
  while (i >= 0 && Keys[i] > x)
   i--;
  if (C[i + 1]->N == 2 * T - 1) {
   splitChild(i + 1, C[i + 1]);
   if (Keys[i + 1] < x)
    i++;
  }
  C[i + 1]->insertNonFull(x);
 }
```

```
}
void TreeNode::splitChild(int i, TreeNode *y) {
 TreeNode *z = new TreeNode(y->T, y->leaf);
 z->N = T - 1;
 for (int j = 0; j < T - 1; j++)
  z->Keys[j] = y->Keys[j + T];
 if (y->leaf == false) {
  for (int j = 0; j < T; j++)
   z->C[j] = y->C[j + T];
 }
 y->N = T - 1;
 for (int j = N; j >= i + 1; j--)
  C[j + 1] = C[j];
 C[i + 1] = z;
 for (int j = N - 1; j >= i; j--)
  Keys[j + 1] = Keys[j];
 Keys[i] = y->Keys[T - 1];
 N = N + 1;
}
int main() {
 BTree T(12);
 T.insert(23);
 T.insert(56);
 T.insert(44);
 T.insert(61);
 T.insert(80);
 T.insert(76);
 T.insert(89);
 T.insert(49);
 T.insert(87);
 T.insert(54);
 cout << "The B-tree is: ";
 T.traverse();
 int x = 76;
 (T.search(x) != NULL) ? cout << endl
         << x << " is found"
       : cout << endl
         << x << " is not Found";
 x = 49;
 (T.search(x) != NULL) ? cout << endl
         << x << " is found"
       : cout << endl
         << x << " is not Found\n";
}
```

The output of given C++ code is as follows:

```
/tmp/QQIgPOAw1Y.o
The B-tree is: 23 44 49 54 56 61 76 80 87 89
76 is found
49 is found
```

9.6 CONCLUSION

- Tree is one of the most significant data structures which is used for efficiently executing operations.
- AVL tree is defined as height balanced binary search tree (BST) in which each individual node is connected with a balance factor.
- The worst case execution of a binary search tree is similar to the linear search program execution, O(n).
- AVL tree was exposed by Adelson, Velshi and Landis provides the short name as AVL tree or balanced binary tree.
- B tree is referred as self-balanced search tree.
- The operation of finding elements in a B Tree resembles the search process of BST.
- B+ Tree are utilised to hold the huge quantity of information for which the main memory is insufficient.
- Red black tree is referred to as a self-balanced tree, where the individual nodes possess the colour - black or red.

9.7 GLOSSARY

- Tree: It is one of the most significant data structures which is used executing operations.
- AVL trees: It is a special binary search tree which auto balances it.
- B trees: It is a special tree which auto stabilizes itself-preserves the arranged information.
- B+ trees: It is a tree having a variable with greater count of children for every node.
- Red black tree: It is referred to as a self-balanced tree, where the each node has black or red colour.

9.8 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. You will find in some cases where the NULL link to a binary tree to distinct links is called as threads. What is the concept of trees in data structure?
2. The method of balancing in AVL tree was exposed by Adelson, Velshi and Landis provide the short name as AVL tree or balanced binary tree. Describe AVL tree in brief.
3. The data retrieval time span of disk is greater than main memory. The technique is to use B trees to minimize the disk visits. Describe the importance of B tree in data structure.
4. In B+ tree, the information is held at the leaf nodes and the key values are held in the interior nodes. Explain the significance of B+ tree in data structure.
5. We must underline that every node needs single bit of memory to hold the data of colour. Determine the concept of colours of red black tree and how its colour makes it different from other trees.

9.9 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. Tree is one of the most significant data structures which is used for efficiently executing operations such as insertion, deletion and searching of values. Refer to Section Introduction
2. In a data structure AVL tree is defined as height balanced binary search tree (BST) in which each individual node is connected with a balance factor. Refer to Section AVL Tree
3. B tree is referred as self-balanced search tree. In order to analyse the mechanism of B trees, let us assume was quantity of information which cannot be held in the main memory. Refer to Section B Tree
4. B+ Tree augments B Tree permitting much organised data addition, removal and search functionalities. In B Tree, both the interior and leaf nodes can hold the key values and information. Refer to Section B+ Tree
5. Red black tree is referred to as a self-balanced tree, where the individual nodes possess an additional bit indicating the colour - black or red. Refer to Section Red Black Tree

9.10 POST-UNIT READING MATERIAL

- https://gurukpo.com/Content/BCA/Data_structure_and_Algorithm.pdf
- https://www.upgrad.com/blog/trees-in-data-structure/

9.11 TOPICS FOR DISCUSSION FORUMS

- Discuss the concept of advanced trees in data structure and its importance with your friends and classmates. Also, discuss the various trees, such as AVL tree, B tree and B+ tree, with examples.

UNIT

10

Advanced and Complex Data Structures

Names of Sub-Units

Introduction to Advanced and Complex Data Structures, Disjoint Sets, Self-Balancing Trees, Segment Trees, Tries, Suffix Array and Tree

Overview

This unit begins by discussing the concept of advanced and complex data structures. Next, the unit discusses the disjoint sets and self-balancing trees. Further, the unit explains the segment tress and tries. Towards the end, the unit discusses the suffix array and tree.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of advanced and complex data structures
- Explain the concept of disjoint sets
- Describe the importance of self-balancing trees and segment trees
- Explain the significance of tries
- Discuss the concept of suffix array and tree

	Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of advanced and complex data structures
- Assess the concept of disjoint sets
- Evaluate the importance of self-balancing trees and segment trees
- Determine the significance of tries
- Explore the concept of suffix array and tree

	Pre-Unit Preparatory Material

- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec16.pdf

10.1 INTRODUCTION

Complex data structures persist as one of the indispensable offshoot of data science which is used for vault, logistics and governance of data and information for efficient, easy accessibility and modification of data. They act as the fundamental component for generating coherent and constructive software design and algorithms. The awareness of building and devising a sterling data structure is crucial for being a creditable programmer. Its purview is expanding with the advent of innovative approaches of execution in information technology.

10.2 DISJOINT SETS

The productivity of an algorithm occasionally relies on underlying data structure. An eminent data structure, say for example, disjoint-set-union, minimises the running timespan of an algorithm.

Imagine that, you operate on a group of N elements that are divided in different subsets where it is required chase the correlation among items of a particular subset or correlation among the subsets. One can utilise the union-find algorithm (disjoint set union) to attain this.

Assume 5 persons, Q, R, S, and T

P is Q's chum, Q is R's chum, S and E are bezzies, hence, the facts presented below are correct are as follows:

- P, Q, and R are associated.
- S and T are interlinked as well.

We can utilise the union-find data structure to verify the associativity among individual friends either straight away obliquely. We can be assuring of the dual distinct detached subsets, in this case. The dual distinct subsets are {A, B, C} and {D, E}.

We need to carry out dual operations are as follows:

- Union(P, Q): Link the items P and Q
- Find(P, Q): search if the items P and Q are interlinked.

For instance, consider a group of items A = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Since ten data items are present, (N = 10).

P and Q data items are interlinked only if arr[P] = arr[Q].

10.2.1 Implementation of Disjoint Sets

For performing the union and find processes, the steps mentioned below are to be carried out:

- Find(P, Q): Verify if arr[P] = arr[Q]
- Union(P, Q): link P to Q and combine the constituents which includes P and Q by restoring items present in arr[P] with the value in arr[Q].

In the beginning, we have 10 subsets and individual subsets has single data item as shown in Figure 1 (a):

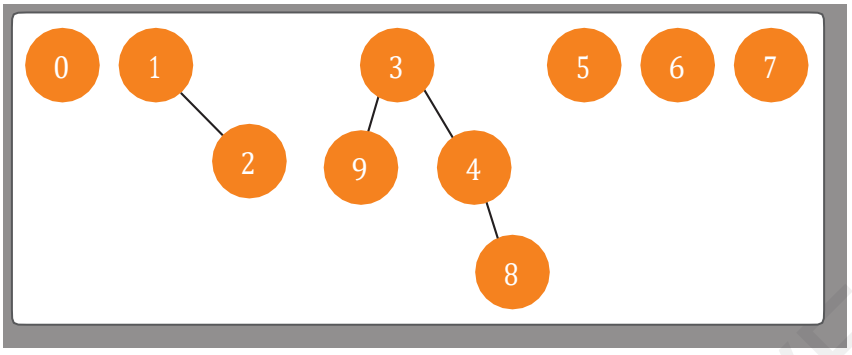
Figure 1(a): 10 Subsets of Single Data

The corresponding array of 10 subsets as shown in Figure 1(b):

Let us carry out few operations as shown in Figure 1(c):

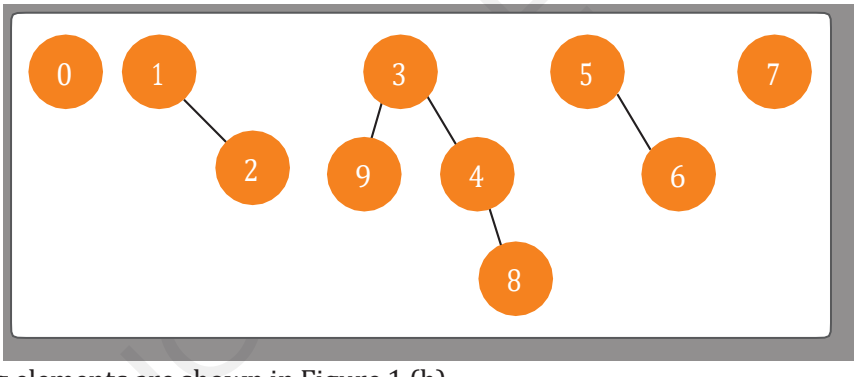
The respective array is shown in Figure 1(d):

The union of (4,3), (8,4) and (9,3) is shown in Figure 1(e):



The corresponding array elements are shown in Figure 1 (f):

The union of (6,5) is shown in Figure 2(g):



The corresponding elements are shown in Figure 1 (h):

Post execution of Union (P, Q), we have 5 subsets:

- Subset1 has the data items {3, 4, 8, 9}
- Subset 2 has the data items {1, 2}
- Subset3 has the data items {5, 6}
- Subset4 has the data items{0}
- Subset6 has the data items {7}

The constituents of a subset, interlinked with one another either straight away obliquely, could be viewed as the vertices or nodes of specific graph. Hence, such formulated subsets are known as *connected* components.

In graphs, the union-find data structure is used to perform many tasks such as interconnecting nodes, searching connected components, and so on.

Let's carry out few find (P, Q) operations:

- Find (0, 7): 0 and 7 are detached, hence, false will be displayed as output.
- Find (8, 9): since 8 and 9 are interlinked obliquely, the output will display true value.

There are multiple approaches to achieve this. But the ideal is weighted union operation as described below:

- We have to keep a record of capacity of all subsets and during interlinking two constituents link the root of the subsets which possess minimal count of data items to the root of the subsets which possess a greater count of data items.
- If we wish to link 1 and 5, then interlink the root of subset A (the subset comprising 1) with the root of subset B (the subset comprising 5) as subset. A consists of minimal count of data items when compared to subset B as shown in Figure 2:

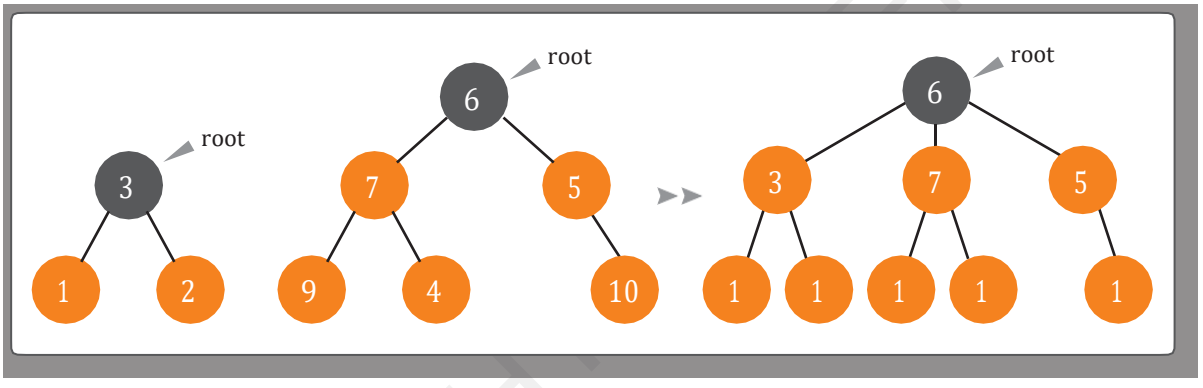


Figure 2: Subset A contain minimal data items then Subset B

The process mentioned above will poise the tree created by carrying out the operations explained priory. The process is termed as weighted-union operation.

To begin with, the capacity of individual subset is 1 as individual subsets consist of single data item. You can set it in the initialize function mentioned below. The capacity [ ] array will record of the capacity of individual subsets.

The following algorithm is to initialize the union functions are as follows:

```
//updated initialize procedure:
     void initialize( int Arr[ ], int N)
          {
               for(int count = 0;count<N;count++)
               {
          Arr[ count ] = i ;
          capacity[ count ] = 1;
          }
          }
```

We have to update the union function since the dual subsets will be interlinked depending upon count of data items in individual subsets.

The following algorithm is to update the union functions are as follows:

```
//Updated union function
void weighted-union(int Arr[ ],int capacity[ ],int P,int Q)
     {
          int root_P = root(P);
          int root_Q = root(Q);
          if(capacity[root_P] < capacity[root_Q ])
          {
     Arr[ root_P ] = Arr[root_Q];
     capacity[root_Q] += capacity[root_P];
     }
          else
          {
     Arr[ root_Q ] = Arr[root_P];
     capacity[root_P] += capacity[root_Q];
     }
}
```

10.3 SELF-BALANCING TREES

Self-Balancing BSTs are height-balanced binary search trees that maintain height on their own. as minimal during data addition and removal processes carried out on tree. The height is preserved approximately as Log n for executing operations in time span of O(Log n) on the whole.

10.3.1 Red Black Tree

Red black tree is referred to as a self-balanced tree, where the individual nodes possess an additional bit indicating the colour -black or red. The colours are utilised for assuring that the tree is kept balanced during data addition and removal. The time to search an element is O(log n), where n refers to the overall count of data values in the tree. Rudolf Bayer devised the tree in 1972.

We must underline that every node needs single bit of memory to hold the data of colour. Such categories of trees display alike memory traces contrast to the vintage (colourless) BST.

Canons to be complied by Red-Black tree are as follows:

- All the individual nodes must possess colour either red or black.
- The root node is certainly black.
- We cannot have two adjoining or adjacent red nodes, which mean a red node can never possess either a red parent or red child).
- All the ways from an individual node (involving root nodes) to each of its successor NULL nodes have the identical count of black nodes.

10.3.2 AVL Tree

In a data structure AVL tree is defined as height balanced binary search tree (BST) in which each individual node is connected with a balance factor and the difference of height of right and left subtree is less than or equal to one by deducting the height of its right from that of its left subtree. This method of balancing in AVL tree was exposed by Adelson, Velshi, and Landis provide the short name as AVL

tree or balanced binary tree. When the balance factor of each node of right and left subtree of a tree is between -1 to 1, it said to be a balanced tree if not it is considered as unbalanced.

The balanced and unbalanced tree is shown in Figure 3:

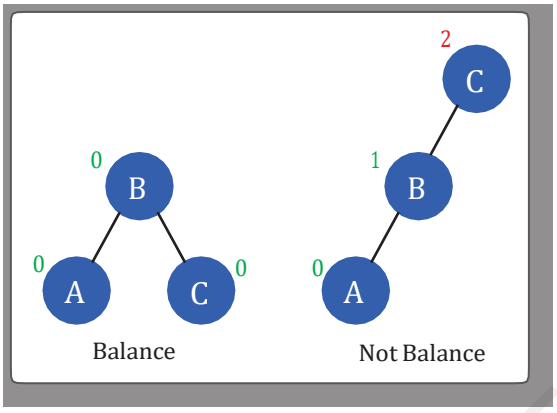


Figure 3: Balanced and Unbalanced Tree

From the figure 3, considering the second tree, the height is 2, the balance factor is 2. To make the tree AVL tree a balance factor of 1 should be maintained as:

Balance Factor = Height of (left-subtree) – Height of (right-subtree)

If the balance factor becomes greater than 1, the tree is stabilized through the rotation processes. For auto balancing, an AVL tree might carry out four types of rotations are as follows:

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

10.3.3 Usage of Self Balancing Tree in Programming Languages

Presence of set and map options in the Standard Template Library (STL) in C++. We have Tree Set and Treemap options in Java. Majority of the data structures to implement library functionality utilise Red Black Tree. One can utilise the Python modules such as Pypi,rbtree and pyavl implement the red black tree and AVL trees.

The time complexity of self-balancing trees are shown in Table 1:

Table 1: Time Complexity of Self-Balancing Trees

Criteria	RB Tree	AVL Tree	Splay Tree
Data addition in worst case	O(1)	O(logn)	Amortized O(logn)
Greatest height of tree	2*log(n)	1.44*log(n)	O(n)
Locating elements in worst case	O(logn), Modest	O(logn), Rapid	Amortized O(logn), Lower

Criteria	RB Tree	AVL Tree	Splay Tree
Productive performance needs	Three pointers having colour bits for individual nodes.	Two pointers having balance factor for individual node	Possess dual pointers lacking additional data.
Removal in worst case	O(logn)	O(logn)	Amortised O(logn)
Widely utilised	As ubiquitous data structure	When recurrent data quest are needed.	When identical data item is accessed multiple times
Concrete application	Database transactions	Multiset, multimap, map, set, etc.	Implementing cache, garbage collection algorithms

10.4 SEGMENT TREES

It is, in essence, a binary tree. A segment tree is a binary tree that is used to store intervals or segments. In the segment tree, each node represents an interval.

We make use of the segment trees in situations which involves various range queries on arrays,and updations of data items of array. For instance, calculating the sum of all data items in an array ranging with indices denoting intervals(L) to (R), lying in between 0 to N-1, in the range of searching the lowest (popularly addressed as Range Minimum Query nodus) of entire data items of an array with indices ranging from L to R. Such problems can be comfortably resolved through one among the adaptable data structures, segment tree.

10.4.1 Visualisation of Segment Trees

We can view the Segment Trees primarily as a binary tree utilised for holding the intervals or segments. The individual nodes of the Segment Tree denote an interval. Assume an array P with capacity N and the respective Segment Tree as T:

- Root node of T indicates the entire array P[0:N−1].
- The leaves of T indicates a solitary data item P[i] such that $0 \le i < N$.
- The interior nodes of T indicates the union of rudimentary intervals A[i:j] where $0 \le i < j < N$.

Entire array P [0:N−1] is denoted as root of segment tree. it is split in two semi segments which acts as descendants of the root denoted by P[0:(N−1)/2] and P[(N−1)/2+1:(N−1)]. Height of the segment tree is given by log2N. The N leaves denote the N data items of the array. The count of interior nodes is N−1. The overall count of nodes is calculated as 2×N−1.

The following functionalities are offered by segment trees:

- Update: To modify the data item of the array P and display the respective updates in the segment tree.
- Query: This is utilized to query an interval and retain the solution of a problem.

10.4.2 Implementation of Segment Tree

A Segment Tree is represented through, a linear array and created by making use of recursion (following a bottom-up mechanism). Begin with the leaf nodes, move towards the root and modify the nodes lying in the way from leaf nodes to root. Leaf nodes denote a solitary data item. At every stage, the values of two descendants are utilised to create an interior antecedent node. The interior nodes denote union or

merge of segments of descendants. For update(), find the leaf nodes which consists of the data item to be modified, which is achieved by moving towards either the left descendant or the right descendant, based on the segment consisting the data item. When the leaf is located, it is modified and using the bottom-up mechanism the modifications are made in the way from that leaf node to the root.

To query() a Segment Tree, choose a range from L to R (present in the question). Perform recursion on the tree initiating with the root and verify whether the segment denoted by the node lies in the range of L to R. In case it falls in the range, retain the value of the node.

The corresponding Segment Tree for array A with capacity 7 resembles as per Figure 4:

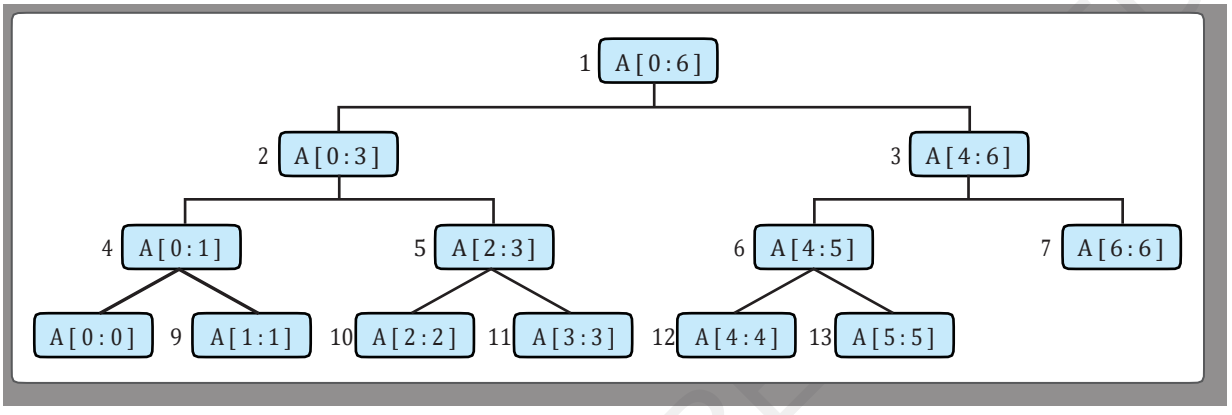


Figure 4: Segment Tree

Figure 5 depicts the segment tree represented as a linear array as per above figure:

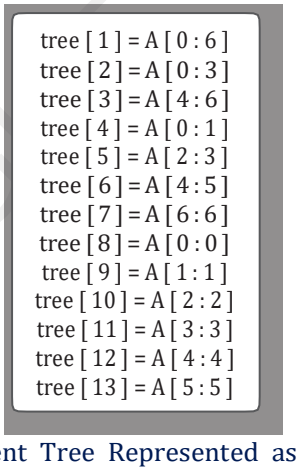


Figure 5: Segment Tree Represented as Linear Array

Consider an array A with capacity N and few queries. The queries are divided in two classes:

- Update: provided index and val, modify array item A[index] as A[index]=A[index]+val.
- Query: provided l and r retain the value of A[l]+A[l+1]+A[l+2]+.....+A[r−1]+A[r] such that 0≤l≤r<N

We can have different orders for Queries and Updates are as follows:

- Naive Algorithm: This is the most fundamental method. Run a loop from l to r and calculate the total of all the elements for the query. As a result, the query will take O. (N). The value of the element will be updated if A[idx] += val is used. It will take O to complete the update (1). This technique works well when there are a lot of update actions and few query operations.

- Another Naive Algorithm: In this approach, the cumulative total of the array's elements is pre-processed and stored in an array sum. Simply return for each query (sum[r] - sum[l-1]). The query operation will now take O minutes (1). To update, however, we must execute a loop and alter the value of the entire sum[i] so that l = I = r. As a result, the update operation will take O. (N). This technique works well when there are a lot of query operations and few update operations.
- Utilising segment tree: In this problem, we'll see how to use segment trees and what we'll store in them. Each node of the segment tree will represent an interval or segment, as we know. We must find the sum of all the elements in the provided range in this problem. As a result, the total of all the elements of the interval represented by the node will be stored in each node. How do we go about doing that? As previously stated, we will construct a segment tree using recursion (bottom-up approach). A single element will be present on each leaf. The total of both children will be present in all internal nodes.

```
void create(int node, int begin, int last)
{
 if(begin == last)
 {
       // Leaf node possesses single data item
       tree[node] = A[begin];
 }
 else
 {
       int mdl = (begin + last) / 2;
       // apply recursion on the left child
       create(2*node, begin, mdl);
       // apply recursion on the right child
       create(2*node+1, mdl+1, last);
       // Interior node possess the sum of its child nodes.
       tree[node] = tree[2*node] + tree[2*node+1];
 }
}
```

Here 2 × node indicates the left node and 2×node+1 indicates the right node, begin and last indicates the interval denoted by the node. Complexity of create() is O(N). Figure 6 depicts segment tree of A (1,3,5,7,9,11):

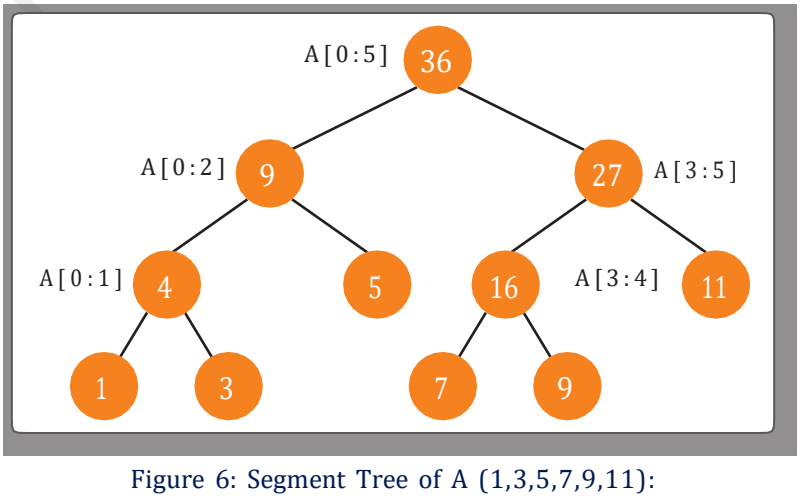


Figure 6: Segment Tree of A (1,3,5,7,9,11):

The time Complexity of update operation is O(logN).

To query a provided range, verify 3 criterias:

1. Range denoted by a node lies within the provided range
2. Range denoted by a node lies external to the provided range
3. A node's range is partially inside and partly outside of the supplied range.

Time Complexity of query is O(logN). The following algorithm describe the time complexity of query is O(logN) are as follows:

```
int query(int node, int begin, int last, int l, int r)
{
     if(r < begin or last < l)
     {
          // range denoted by a node is entirely external to the provided range
          return 0;
     }
     if(l <= begin and last <= r)
     {
     // range denoted by a node is entirely within the provided range
     return tree[node];
     }
     // range denoted by a node is partly interior and partly exterior to the
     provided range
     int mdl = (begin + last) / 2;
     int q1 = query(2*node, begin, mdl, l, r);
     int q2 = query(2*node+1, mdl+1, last, l, r);
     return (q1 + q2);
}
```

10.5 TRIES

The word "Trie" is an abbreviation for "retrieval." The set of strings is stored in a sorted tree-based data structure called a trie. Each node has the same number of pointers as the number of characters in the alphabet. It may use the prefix of a word to look up that word in the dictionary. If we suppose that all strings are made up of the letters a through z from the English alphabet, each trie node can have a maximum of 26 points.

Tries are exceptional data-structures dependent on the prefix of a string. They denote the access to information and are termed as Trie.

We can reckon strings as a significant and general theme of multiple of programming snags. Processing strings has multiple of concrete applications, some are displayed below:

- Search Engines
- Genome Analysis
- Data Analytics

Entire textual data accessible to us can be interpreted as strings.

10.5.1 Prefix of String

The prefix of a string S represents the initial n letters n≤|S| from which the given string commences. For instance, the word “abacaba” possess the prefixes mentioned below:

```
a
ab
aba
abac
abaca
abacab
```

Trie refers to a remarkable data structure utilised to reserve strings which could be viewed as nodes of graph. Every node contains of atmost 26 successors and edges link each antecedent node to its descendants. The 26 child nodes represent the 26 letters of the English alphabet series.

Strings are held in a top down approach on the based on the prefixes of a Trie. Entire prefixes having size 1 are reserved till level 1, the prefixes having size 2 are reserved till level 2 and Sequence continues. For instance, as shown in Figure 7:

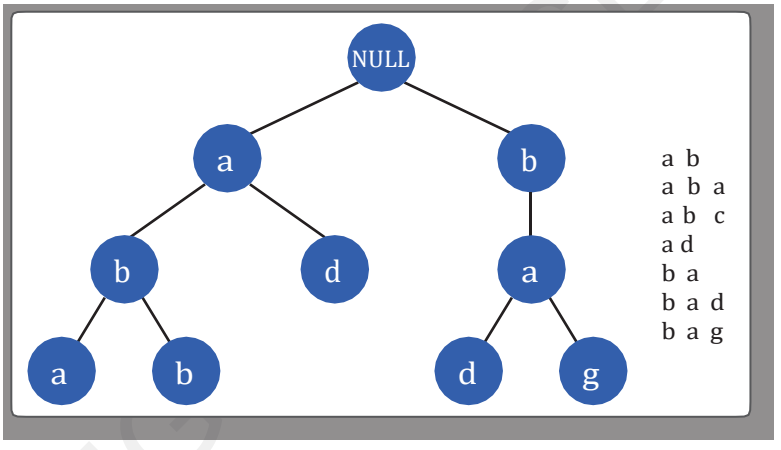


Figure 7: Prefix of String

Now, it is obvious to wonder why we prefer a Trie for working with a single string. In general, Tries are applicable on cluster of strings, instead of a solo string. With various strings, we could resolve a multiple problems relying on them. For instance, let us think of an English dictionary and a solo strings, search the prefix of greatest size from the dictionary coinciding with string involving tries could resolve this problem in a more organised way.

A Trie consists of 26 outgoing edges representing the individual letters of the alphabet.

The pseudo code for adding a string into a Trie is as follows:

```
void insert(String s)
{
     for(each char in string s)
     {
          if(successor of the current char is null)
          {
               child node=new Node();
          }
          current_node=child_node;
```

```
    }
}
```

The pseudo code for verifying whether a specific word is present in a dictionary or not is as follows:

```
boolean Verify(String s)
{
    for(every char in String s)
    {
        if(successor is null)
        {
            return false;
        }
    }
    return true;
}
```

10.6 SUFFIX ARRAYS AND TREES

A suffix array is an organized collection of all the suffixes in a string. The description is alike to that of the Suffix Tree, which is a compressed trie of all suffixes in a text. Any suffix tree-based technique can be substituted with a suffix array enriched with additional information that solves the similar problem in the same amount of time (Source Wiki).

By performing a DFS traverse of the suffix tree, a suffix array can be generated. In fact, both the suffix array and the suffix tree can be built in linear time from each other.

Suffix array refers to the structure based upon arrays. It is a lexical arranged array holding suffixes of a string s. Consider the string s = abakan, which has six suffixes: abakan, bakan, akan, kan, an, n, and the related suffix tree as shown in Figure 8:

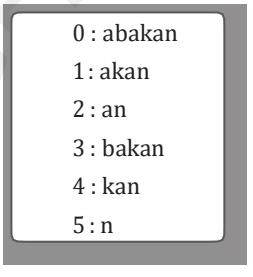


Figure 8: 6 suffix of word abakan

For minimising the memory space, we do not reserve the suffixes explicitly. It is adequate to reserve their indices alone.

Suffix arrays, integrated with LCP table (indicates longest common prefix of neighbouring suffixes table), are extremely worthy for resolving multiple problems. Suffix arrays are created in $O(n * \log^2 n)$ time, where n indicates the length of s. The time complexity could be enhanced to O(n * log n) through linear time sorting algorithm.

Correlation among suffix tree and suffix array:

It is noteworthy to put forth, that it is possible to build Suffix Array straight away through a suffix tree in linear time by making use of DFS traversal.

We view suffix tree as a natural refinement over Tries utilised in pattern recognition cases. Such Tries consists of lengthier ways excluding branches. If we minimise such lengthy pathways in single leap, the length of Trie shrinks considerably. Such a contracted trie described on the A suffix tree of s is a subset of a string's suffixes.

Take, for example, the suffix tree for the string s = abakan. The word possess 6 suffixes {abakan , bakan, akan, kan, an, n} and corresponding videos suffix tree resembles as shown in Figure 9:

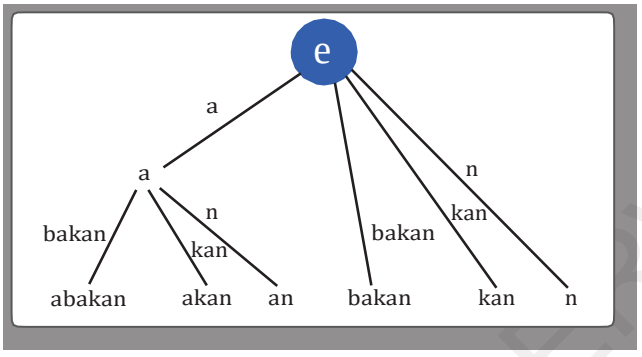


Figure 9: Suffix tree of a String S

Suffix trees are used to resolve various intricate problems, as it consists of detailed data regarding the string. A popular application is searching the count of discrete substrings of s, which is comfortably resolved through suffix tree.

10.7 CONCLUSION

- Complex data structures persist as one of the indispensable offshoot of data science which is used for vault, logistics and governance of data.
- Self-balancing BSTs are height-balanced binary search trees that maintain height on their own.
- Red black tree is referred to as a self-balanced tree, where the individual nodes possess an additional bit indicating the colour -black or red.
- In a data structure AVL tree is defined as height balanced binary search tree (BST) in which each individual node is connected with a balance factor.
- It is, in essence, a binary tree. A segment tree is a binary tree that is used to store intervals or segments.
- A Segment Tree is represented through, a linear array and created by making use of recursion.
- The set of strings is stored in a sorted tree-based data structure called a trie.
- A suffix array is an organized collection of all the suffixes in a string.
- Suffix trees are used to resolve various intricate problems, as it consists of detailed data regarding the string.

10.8 GLOSSARY

- Disjoint sets: It refers to a data structure which records a group of data items divided into a multiple disjoint (non-imbricating) subsets, which is utilised for estimating whether two constituents belong to the identical subset

- Self balancing trees: Refers to a BST which tries to maintain its height, or the count of levels of nodes underneath the root, as minimal every time, in an automated way
- Tries: data structure utilized for holding strings which is viewed as a graph. It contains nodes and edges
- Red black tree: It is referred to as a self-balanced tree, where the individual nodes possess an additional bit indicating the colour-black or red
- AVL tree: It is defined as height balanced binary search tree (BST) in which each individual node is connected with a balance factor
- Segment tree: It is a binary tree that is used to store intervals or segments
- Trie: It is the set of strings is stored in a sorted tree-based data structure called a trie
- Suffix array: It is an organised collection of all the suffixes in a string
- Suffix trees: It is used to resolve various intricate problems, as it consists of detailed data regarding the string

10.9 SELF ASSESSMENT QUESTIONS

A. Answers to Essay Type Questions

1. The awareness of building and devising a sterling data structure is crucial for being a creditable programmer. What is the concept of complex data structure?
2. Explain the concept of disjoint sets.
3. The colours are utilised for assuring that the tree is kept balanced during data addition and removal. Describe the colors concept of red-black tree.
4. A Segment Tree is represented through, a linear array and created by making use of recursion (following a bottom-up mechanism). What is a segment tree?
5. Tries are exceptional data-structures dependent on the prefix of a string. Describe the significance of tries.

10.10 ANSWERS AND HINTS FOR SELF ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. Complex data structures persist as one of the indispensable offshoot of data science which is used for vault, logistics and governance of data and information for efficient, easy accessibility and modification of data. Refer to Section Introduction
2. The productivity of an algorithm occasionally relies on underlying data structure. An eminent data structure, say for example, disjoint-set-union, minimises the running timespan of an algorithm. Refer to Section Disjoint Sets
3. Red black tree is referred to as a self-balanced tree, where the individual nodes possess an additional bit indicating the colour -black or red. Refer to Section Self-Balancing Trees
4. It is, in essence, a binary tree. A segment tree is a binary tree that is used to store intervals or

segments. In the segment tree, each node represents an interval. Refer to Section Segment Trees

5. The word "Trie" is an abbreviation for "retrieval." The set of strings is stored in a sorted tree-based data structure called a trie. Refer to Section Tries

10.11 POST-UNIT READING MATERIAL

- https://jntukucev.ac.in/wp-content/uploads/2020/03/ADS-lecture1.pdf
- https://jntukucev.ac.in/wp-content/uploads/2020/03/ADS-lecture1.pdf

10.12 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates the concept of advanced and complexed data structure. Also, discuss on the self-balancing trees and practice the various programming codes and assess their behaviours.

UNIT

11

Heaps

Names of Sub-Units

Applications of Heap, Definitions of Max-Heaps and Min-Heaps, Implementing a Heap, Using a Heap to Implement Heapsort

Overview

This unit begins by discussing the concept of heaps. Next, the unit describes the application of heap and definitions of max-heaps and min-heaps. Further, the unit explains the implemention a heap. Towards the end, the unit highlights the process of using a heap to implement heapsort.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of heaps
- Explain the concept of applications of heap
- Describe the definitions of max-heaps and min-heaps
- Explain the significance of implementing a heap
- Understanding the using of heap to implement heapsort

	Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of heaps
- Assess the concept of applications of heap
- Clarify the definitions of max-heaps and min-heaps
- Determine the significance of implementing a heap
- Explore the using of heap to implement heapsort

	Pre-Unit Preparatory Material

- https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture4.pdf

11.1 INTRODUCTION

A data structure heap is a specialised tree that fulfills all the traits: if P is an antecedent of C, then P has a value either larger or equal to in case of Max-Heaps or lower or equal to in case of min-heap the than the value of C [1]. The topmost node of the heap without antecedents is referred to as the root.

The heap is an effective contrivance of an abstract data type known as priority queue which is usually known as “heaps”, irrespective of the background process of their implementation. Heap is generally implemented as a binary heap which has the binary tree. A heap is an unorganised structure and can be thought of as incompletely arranged. Heaps lack the presence of a specific correlation between nodes at any level, even siblings exist without any association. If the heap is a complete binary tree, it possessess a minimal height—a heap having N nodes with node-wise branches has an invariable height of log N. A heap proves to be an efficient data structure during cases involving element deletion with the Maximum (or Minimum) preferences or priority.

11.1.1 Operations on Heap

The following are the general operations on the heap:

- Find-max (or find-min): Searches the greatest element in a max-heap or the smallest element in a min-heap
- Insert: Pushes a new element into the heap
- Extract-max (or extract-min): retains the node with the greatest value in a max-heap [or smallest element in a min-heap] post deleting it from the heap
- Delete-max (or delete-min): deletes the root node of a max-heap (or min-heap)
- Replace: Removes existing root value push a new element
- Create-heap: Builds a vacant heap
- Heapify: Builds a heap from provided elements
- Merge (union): Attaches two heaps to create a viable new heap having entire elements, conserving the actual heaps

- Meld: Attaches two heaps to create a viable new heap consisting of entire elements of both, demolishing the actual heaps
- Inspection: Examines the elements of the heaps
- Size: Retains the overall count of elements in the heap
- Is-empty: Retains true when the heap is vacant, else false is returned
- Increase-key or decrease-key: Modifies a data item present in a max- or min-heap, respectively.
- Delete: Removes the relevant nodes (followed by relocating the final node to sustain heap)
- Sift-up: Transfers a node to the upper levels of the tree, for a required period, reinstates heap after data addition. Known as "sift" since node travels up in the tree till it arrives at the accurate level
- Sift-down: Transfers a node at the bottom levels in the tree, analogous to sift-up; reinstates heap after data removal

11.2 APPLICATIONS OF HEAP

We can find the application of heaps in popular algorithms like Dijkstra's algorithm for searching the minimal path. The heapsort algorithm makes use of priority queues. Heaps finds usage in the situations where we wish to retrieve the largest or smallest data items faster.

Heap Data Structure accompanies Heapsort. Heapsort algorithm is rarely used.

The applications below utilise Heapsort is as follows:

- Priority Queues: They are executed through Binary Heaps. It completes the functionalities like data Addition(), Removal() and Find Greatest(), ReduceKey() in O(logn) time. Binomial Heap and Fibonacci Heap are variants of Binary Heap. They carry out union in O (logn) time which requires O(n) operations in Binary Heap. The priority queues using heaps are applicable in Graph algorithms like prim's Algorithm and Dijkstra's algorithm.
- Order statistics: The Heaps are used effectively to search the nth lowest (or greatest) data item in an array.

  Heapsort is used to subdue the Worst-Case Complexity of Quick Sort procedure from O (n^2) to O(nlog(n)).

  Security Systems and embedded systems like Linux Kernel utilises Heapsort due to time complexity of O(nlog(n).

11.3 DEFINITIONS OF MAX-HEAPS AND MIN-HEAPS

A heap is a complete binary tree, which can be either a max-heap or min-heap. The max-heap has the property that the key value of any node must be greater than or equal to the key values of its children. In min-heap, the key value of any node must be lower than or equal to the values of its children.

11.3.1 Min-Heap

In a Min-Heap the data item existing at the root node must be lower than or equal to the data values existing in the descendants. The identical principle is recursively applicable for entire sub-trees representing a binary trees. Here the lowest data item exists at the root.

Figure 1 depicts a binary tree which fulfills the criteria of Min-Heap:

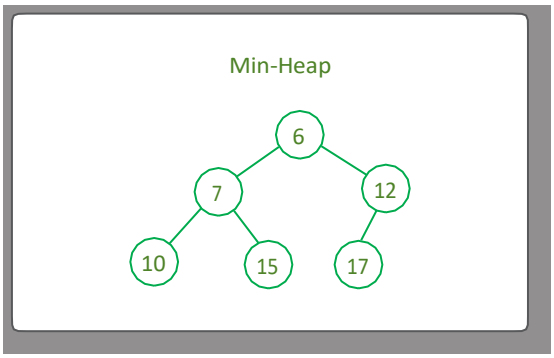


Figure 1: Min-Heap

The following C++ program is used to implement min-heap is as follows:

```
#include <iostream>
using namespace std;
void min_heap(int *x, int y, int z){
   int i, k;
   k=  x[y];
   i = 2 * y;
   while (i <= z) {
      if (i < z && x[i+1] < x[i])
         i = i + 1;
      if (k < x[i])
         break;
      else if (k >= x[i]) {
         x[i/2] = x[i];
         i = 2 * i;
      }
   }
   x[i/2] = k;
   return;
}
void build_minheap(int *x, int z) {
   int k;
   for(k = z/2; k >= 1; k--) {
      min_heap(x,k,z);
   }
}
int main() {
   int z, i;
   cout<<"Enter no of Elements of Array\n";
   cin>>z;
   int x[30];
   for (i = 1; i <= z; i++) {
      cout<<"Enter element"<<" "<<(i)<<endl;
      cin>>x[i];
   }
   build_minheap(x, z);
   cout<<"Min Heap is\n";
```

```
    for (i = 1; i <= z; i++) {
       cout<<x[i]<<endl;
    }
}
```

The output of given C++ code is as follows:

```
/tmp/GiYHVftMnK.o
Enter no of Elements of Array
7
Enter element 1
1
Enter element 2
2
Enter element 3
3
Enter element 4
4
Enter element 5
5
Enter element 6
6
Enter element 7
7
Min Heap is
1
2
3
4
5
6
7
```

11.3.2 Max-Heap

In a Max-Heap the data items existing at the root node must be larger than or equal to the data items existing in the descendants. This identical principle holds recursively accurate for entire sub-trees in respective Binary Tree. Here the largest data item exists at the root. Figure 2 depicts a binary tree that fulfills the relevant criteria's for Max-heap:

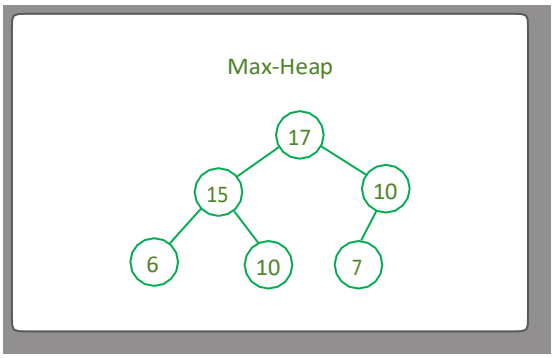


Figure 2: Max-Heap

The following C++ program is used to implement max-heap is as follows:

```
#include <iostream>
using namespace std;
void max_heap(int *b, int c, int d) {
   int x, y;
   y = b[c];
   x = 2 * c;
   while (x <= d) {
      if (x < d && b[x+1] > b[x])
         x = x + 1;
      if (y > b[x])
         break;
      else if (y <= b[x]) {
         b[x / 2] = b[x];
         x = 2 * x;
      }
   }
   b[x/2] = y;
   return;
}
void build_maxheap(int *b,int d) {
   int k;
   for(k = d/2; k >= 1; k--) {
      max_heap(b,k,d);
   }
}
int main() {
   int n, i;
   cout<<"Enter no of Elements of Array:\n";
   cin>>n;
   int a[30];
   for (i = 1; i <= n; i++) {
      cout<<"Enter Elements"<<" "<<(i)<<endl;
      cin>>a[i];
   }
   build_maxheap(a,n);
   cout<<"Max Heap is:\n";
   for (i = 1; i <= n; i++) {
      cout<<a[i]<<endl;
   }
}
```

The output of given C++ code is as follows:

```
/tmp/GiYHVftMnK.o
Enter no of Elements of Array:
4
Enter Elements 1
11
```

```
Enter Elements 2
12
Enter Elements 3
13
Enter Elements 4
14
Max Heap is:
14
12
13
11
```

11.3.3 Min-Heap vs Max-heap

Min-Heap	Max-Heap
The data value of min-heap at the root node must be lower than or equal to values of its descendants.	The data item in max-heap existing at the root must be larger than or equal to the data items existing in entire child nodes.
Here the least data value exists at the root.	Here the largest data item exists at the root.
It works on the increasing priority or preference.	It works on decreasing priority or preference.
During its construction, the lowest data item is given higher preference.	During its, construction the largest data item has highest preference.
Here, the lowest data item is initially removed from the heap.	Here, the greatest data item is initially removed from the heap.

11.4 IMPLEMENTING A HEAP

We contrive the Heaps through Arrays. We can reserve a binary tree in an Array. Since a binary heap is inevitably a complete binary tree, occupies minimal memory space. Additional space is not needed for pointers; rather, the antecedent and descendant of each node are computed by performing an arithmetic operation on array indices. These traits present the heap implementation as an illustration of an implicit data structure or Ahnentafel list. The features rely upon the location of the root, which is dependent on the limitations of the underlying programming language. Occasionally, the root is set at index 1, to untangle arithmetic calculations.

Assume n as the overall data items in the heap and i as a random logical index of an array holding the heap. When the root is present at index 0, having indices 0 to $n - 1$, then every data item a at index i possess the following points:

- Descendants at indices $2i + 1$ and $2i + 2$
- Antecedents at index $floor((i - 1) / 2)$.

Conversely, when the root is at index 1, having indices 1 to n, then every data item a at index i possess the following points:

- Successors at indices $2i$ and $2i + 1$
- Antecedents at index $floor(i / 2)$.

This contrivance is utilised in the process of heapsort, where it permits the reutilisation of space in the input array to hold the heap. The utilise heaps to implement a Priority queue which uses dynamic array permitting data addition without restrictions, as shown in Figure 3:

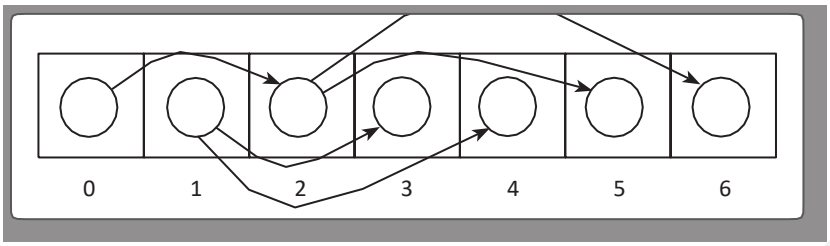


Figure 3: Implementation of Heap through Array

Furthermore, we can implement a binary heap with a classic binary tree data structure, but it is troublesome to search the adjoining data item at the final level of the binary heap during insertion. This data item is evaluated algorithmically or by inserting additional data to the nodes, known as “threading” the tree—rather than reserving references to the child nodes, we reserve the in order descendant of the node.

11.5 USING A HEAP TO IMPLEMENTING HEAPSORT

Heapsort is a contrast-dependent data arrangement mechanism using Binary Heap data structure. It resembles selection sort in which we initially search the low valued element and place it at the position. We iterate the technique again for the leftover data items.

11.5.1 Understanding Binary Heap

A binary heap is a complete binary tree. All layers until the last are filled in a complete binary tree. The keys are as far to the left as possible in the final level which satisfied the heap property. Depending on the heap property it satisfies, the binary heap can be either max or min-heap.

A Binary Heap refers to the binary tree having the charareristics are as follows:

- It refers to a complete tree (where all levels are fully occupied excluding the final level which has entire keys inclined to the left). This trait of Binary Heap suits it to be reserved in an array.
- A Binary Heap could be classified into Min-Heap or Max-Heap. In a Min Binary Heap, the element value at the root must be lowest amidst of entire elements existing in the Binary Heap. The identical principle holds recursively true for entire nodes in Binary Tree. Max Binary Heap resembles Min-Heap.

To comprehend the Binary heaps, one must have an in-depth understanding of a Complete Binary tree. A complete binary tree refers to the binary tree where all the levels, excluding the final level, are fully occupied and entire nodes are inclined to the left. A Binary Heap refers to a Complete Binary Tree in which data is held in a specific manner, such that the element in a parent node is larger for Max-heap (or lowest for Min-Heap) as compared to the values of the child nodes. The heap can be denoted using a binary tree or array.

11.5.2 Array Representation of Binary Heap

A Binary Heap can be smoothly denoted through an array and such representation is space-efficient. If we reserve the parent node at index I, its left child is computed as 2 * I + 1 and right child as 2 * I + 2 (Supposing the index begins at 0).

A Binary Heap refers to a Complete Binary Tree. We make use of arrays to represent Binary Heap.

The data item of Root is placed at Arr[0].

The following points displays the indexes of different nodes,Arr[I] indicates i th node.

- Arr[(i-1)/2]: denotes the parent node
- Arr[(2*i)+1]: denotes the left child node
- Arr[(2*i)+2]: denotes the right child node

Figure 4 depicts array representation of binary heap:

The traversal technique to attain

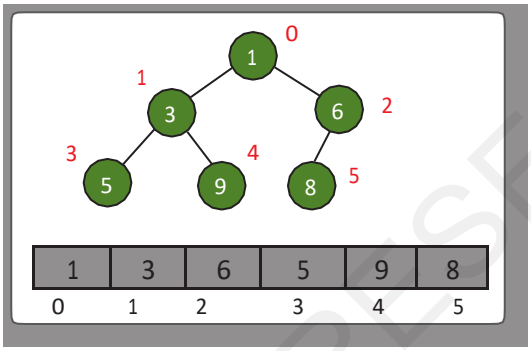


Figure 4: Array Representation of Binary Heap

Array representation is Level Order.

An algorithm of Heapsort in ascending order is as follows:

- Create a max-heap from the scanned data.
- Initially the greatest data value is held at the root. Reinstate it with the final data value of the heap and decrease the heap size by 1. Ultimately, heapify the root node.
- Iterate step 2 as long as the heap size is larger than 1.

11.5.2 Construction of Heap

The technique Heapify applies to a node only if its child nodes are heapified. Hence heapification must be carried out following a bottom-up approach.

Consider the following example is as follows:

Scanned values: 4, 10, 3, 5, 1

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

The numbers in braces denote the array indices.

Implementing heapify technique at index 1:

```
      4(0)
     /  \
  10(1)  3(2)
  / \
5(3)  1(4)
```

Implementing heapify technique to index 0:

```
      10(0)
      /  \
   5(1) 3(2)
   / \
4(3)  1(4)
```

The technique heapify calls itself recursively to create heap.

11.6 CONCLUSION

- A heap is an unorganised structure and can be thought as incompletely arranged.
- We can find the application of heaps in popular algorithms like Dijkstra's algorithm for searching the minimal path.
- In a Min-Heap the data item existing at the root node must be lower than or equal to the data values existing in the descendants.
- In a Max-Heap the data items existing at the root node must be larger than or equal to the data items existing in the descendants.
- A binary heap is a complete binary tree.
- Heapsort is a contrast dependent data arrangement mechanism using Binary Heap data structure.

11.7 GLOSSARY

- Heap: It is a specialised Data Structure belonging to the class of trees where the tree is complete binary tree.
- Priority queue: it is an abstract data type resembling the queue data structure where the individual elements possesses a "Priority" or preference.
- Binary heap: it is a class of heap data structure which exists as binary tree.
- Dijkstra's algorithm: It is an algorithm for searching the minimal pathways amongst the nodes in a graph.
- Max-heap: It is a class of heap, where the data value in the interior nodes is larger or equal to the values of the children of the specified node.

- Min-heap: It is the data item existing at the root node must be lower than or equal to the data values existing in the descendants.
- Binary heap: It is a complete binary tree.
- Heapsort: It is a contrast dependent data arrangement mechanism using Binary Heap data structure.

11.8 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. A data structure heap is a specialised tree which fulfills all the traits. State the concept of Heap.
2. A heap is a complete binary tree, which can be either max-heap or min-heap. Explain the significance of min and max heaps.
3. Describe the concept of binary heap.
4. We contrive the Heaps through Arrays. We can reserve a binary tree in an Array. Clarify how a heap can be implemented.
5. We can find the application of heaps in popular algorithms like Dijkstra's algorithm for searching the minimal path. Describe the applications of heaps.

11.9 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. The heap is an effective contrivance of an abstract data type known as priority queue which are usually known as "heaps", irrespective of the background process of their implementation. Refer to Section Introduction
2. The max-heap has the property that the key value of any node must be greater than or equal to the key values of its children. In min-heap, the key value of any node must be lower than or equal to the values of its children. Refer to Section Definitions of Max-Heaps and Min Heaps
3. A binary heap is a complete binary tree. All layers until the last are completely filled in a complete binary tree. Refer to Section Using a Heap to Implementing Heapsort
4. Assume n as the overall data items in the heap and i as random logical index of an array holding the heap. When the root is present at index 0, having indices 0 to $n - 1$, then every data item a at index i possess:
   - Descendants at indices $2i + 1$ and $2i + 2$
   - Antecedents at index $floor((i - 1) / 2)$.

   Refer to Section Implementing a Heap
5. The applications below utilise Heapsort is as follows:
   - Priority Queues: They are executed through Binary Heaps. It completes the functionalities like data Addition(), Removal() and Find Greatest(), ReduceKey() in O(logn) time. Binomoial Heap and Fibonacci Heap are variants of Binary Heap.

Refer to Section Applications of Heap

11.10 POST-UNIT READING MATERIAL

- https://www.techgeekbuzz.com/dsa-binary-heap-tree/
- https://www.interviewkickstart.com/learn/heap-sort

11.11 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates about the concept of heaps, implementations of heaps and its various operations. Also, discuss about the applications of heap with real world examples.

UNIT

12 Graphs

Names of Sub-Units

Graph Terminology, Directed and Undirected Graph, Adjacency List and Adjacency Matrix Representation of Graphs, Elementary Graph Operations, Traversal Methods

Overview

This unit begins by discussing about the concept of graphs and graph terminologies. Next, the unit describes the directed and undirected graph. Further, the unit explains the matrix and adjacency list representation of graphs and elementary graph operations. Towards the end, the unit analyses the traversal methods.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of graphs
- Explain the concept of graph terminologies and directed and undirected graph
- Describe the matrix and adjacency list representation of graphs
- Outline the significance of elementary graph operations
- Elucidate the importance of traversal methods

Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of graphs
- Assess the concept of graph terminologies and directed and undirected graph
- Evaluate the importance of matrix and adjacency list representation of graphs
- Determine the significance of elementary graph operations
- Understand the importance of traversal methods

Pre-Unit Preparatory Material

- https://www.pvpsiddhartha.ac.in/dep_it/lecture%20notes/CDS/unit5.pdf

12.1 INTRODUCTION

A graph is a sort of a tree (with or without cycles), in which each node is called a vertex and is connected with lines, called edges. The set of vertices is represented by V and the set of edges is represented by E. Hence, a graph is designated as: G = (V, E). For example, the graph shown in Figure 1 has a set of vertices:

V={d,a,b,c} and a set of edges: E={(d,a), (a,b),(a,c)}.

The degree of a vertex is the number of edges that are joined to that vertex, i.e., the number of neighbors it has. If the vertex a has three neighbors: b, c, d, then the degree of vertex a is said to be 3, as shown in Figure 1:

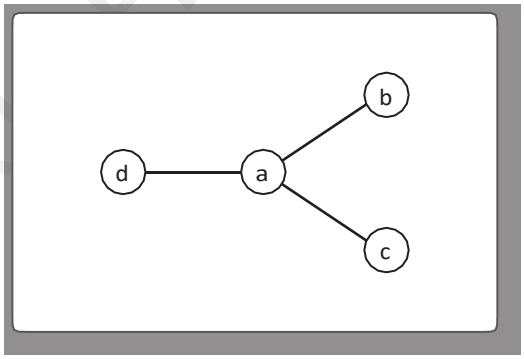


Figure 1: Graph

A graph is said to be a simple graph if:

- It has no cycles. A cycle is an edge of a node that connects to itself)
- No more than one edge joins any pair of nodes

In the graph shown in Figure 1, the edges do not have any direction, i.e., there is no starting or ending vertices. The edge (d, a) can also be written as (a, d) which depicts that there is an edge connecting the vertices, d and a. Since there is no direction in the edges, this graph can also be called “Undirected Graph”.

We can assign direction to the edges of this graph, as shown in the Figure 2:

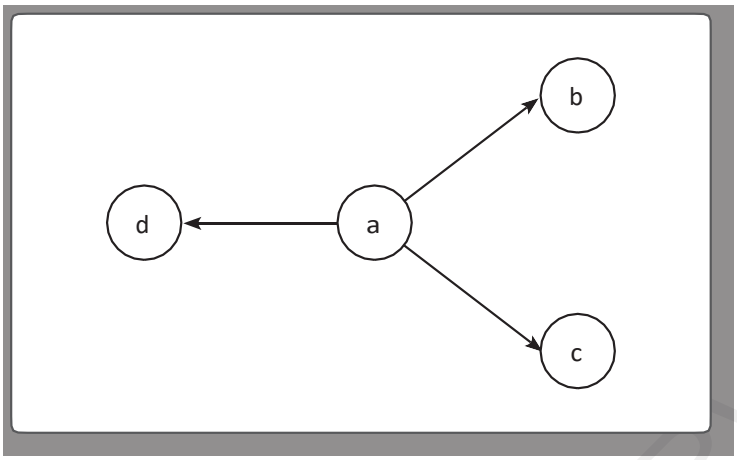


Figure 2: A Directed Graph has directions to its edges

The Graph shown in Figure 2 has directions attached to its edges and is hence called a Directed Graph. Since the edges connecting the two vertices have a direction, we term one of the edges as a starting vertex and the other as the ending vertex. For example, let's observe an edge in the preceding graph: (a,d), it represents that there is an edge from vertex a to vertex d (vertex a can be called as starting vertex and vertex d can be called as ending vertex. The set of edges: E of the graph shown above can be written as {(a,d), (a,b), (a,c)}.

We learn about graphs, their different types and about where they are applied. You also learn about the methods, Adjacency Matrix and Adjacency List, which are used to represent graphs on computer. You also learn about how to traverse a graph using the Breadth First Traversal and Depth First Traversal methods.

12.1.1 Graph Data Structure

Statistical graphs could be depicted using the data structure. A graph is represented through an array holding vertices and a single-dimensional array consisting of edges. Before moving ahead, let's get acquainted with certain significant terms are as follows:

- Vertex: Every node in a graph is called a vertex. In the image below, the marked circle denotes vertices. The circles from, A to G denotes vertices. They are represented through an array as displayed in the figure 3. A is recognised through index 0. B is recognised through index 1 and the pattern continues.
- Edge: Edge indicates a way or a line among two vertices. In the figure 3, the lines from A to B, B to C, etc. denotes edges. A two-dimensional array is utilised to depict an array as displayed in the figure 3. Here AB is denoted as 1 at row 0, column 1, BC as 1 at row 1, column 2, etc., making remaining coalescence as 0.
- Adjacency: Dual nodes or vertices become adjacent when they are linked among themselves by an edge. In figure 3, B and A are adjacent nodes, C and B are adjacent nodes, etc.
- Path: Path denotes a series of edges among the two vertices. In the figure 3, ABCD indicates a path from A to D.

Figure 3 depicts a graph according to the above points:

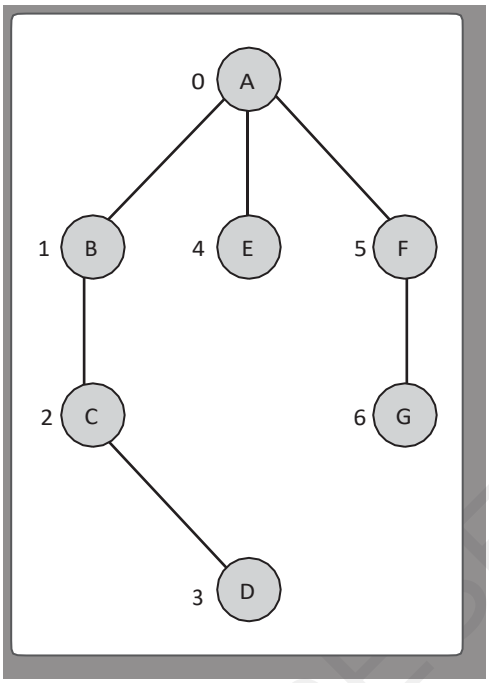


Figure 3: A Graph Data Structure

The fundamental operations of the graph are as follows:

- Add vertex: Inserting a vertex to the graph.
- Add edge: Inserting an edge amidst two vertices in a graph.
- Display vertex: Showing a vertex of a graph.

12.2 GRAPH TERMINOLOGY

The graph data structure has multiple terminologies which are as follows:-

- Vertex: Individual data element of a graph is called Vertex. It is also known as a node. In the above example graph, A, B, C, D & E are known as vertices.
- Edge: An edge refers to interconnected linkage among two vertices. Alternatively, an edge is termed as an Arc. An edge has a (Starting Vertex and an Ending Vertex). For instance, the graph above has the link among vertices A and B denoted as (A,B). On the whole, 7 edges (i.e., (A, B), (A, C), (A, D), (B, D), (B, E), (C, D) and (D, E) exists. Edges are divided into three classes:
  - Undirected edge: refers to a bidirectional edge. When an undirected edge exist among vertices A and B then edge (A, B) and edge (B, A) are identical.
  - Directed edge: refers to the unidirectional edge. When a directed edge exists among vertices A and B then edge (A, B) is unequal to edge (B, A).
  - Weighted edge: refers to an edge having assigned value (price) to it.
- Undirected graph: A graph having undirected edges alone is termed as undirected graph.
- Directed graph: A graph having directed edges alone is called as directed graph.
- Mixed graph: A graph having the undirected as well as the directed edges is called as mixed graph.

- End vertices or endpoints: The dual vertices connected through an edge are termed as end vertices (or End points) of the specific edge.
- Origin: In a directed edge, the initial terminating point is called as its origin.
- Destination: In a directed edge, its initial terminating point is called its origin and another terminating point is called as the destination of the specific edge.
- Adjacent: If an edge exists amidst of vertices A and B then they are termed as adjacent.
- Incident: An Edge is an incident on a vertex when the vertex becomes one of the terminating points of the specific edge.
- Outgoing edge: A directed edge is called as the outgoing edge of the source vertex.
- Incoming edge: A directed edge is called as the incoming edge on the target vertex.
- Degree: Overall count of edges linked to a vertex is termed as the degree of the vertex.
- Indegree: Overall count of incoming edges linked to a vertex is termed as indegree of the vertex.
- Outdegree: Overall count of outgoing edges linked to a vertex is termed as outdegree of the vertex.
- Parallel edges or multiple edges: When two directionless edges have common terminating vertices and two directed edges having common source and destination, exists, we term such edges as parallel edges or multiple edges.
- Self-loop: Edge (undirected or directed) becomes self-loop when its two terminating points concur.
- Simple graph: A graph is called simple when no parallel and auto-loop edges exist.
- Path: A path refers to a series of intermittent vertices and edges which initiates at one vertex and terminates at another vertex where each edge connects the antecedent and descendant vertices.

12.3 DIRECTED AND UNDIRECTED GRAPHS

A collection or a set of vertices of directed edges that links with an individual ordered pair of vertices is known as a directed graph or it is also termed as a digraph.

It possesses directed edges, which denote a one-way relationship, where individual edges could be traversed in a unidirectional way alone. The image displays a directed graph having three nodes and two edges. The accurate location, size or alignment of the edges in a graph depiction is meaningless as a graph can be interpreted in various forms by reshuffling the nodes and/or deforming the edges, provided the primary structure stays unchanged. Figure 4 depicts a directed graph:

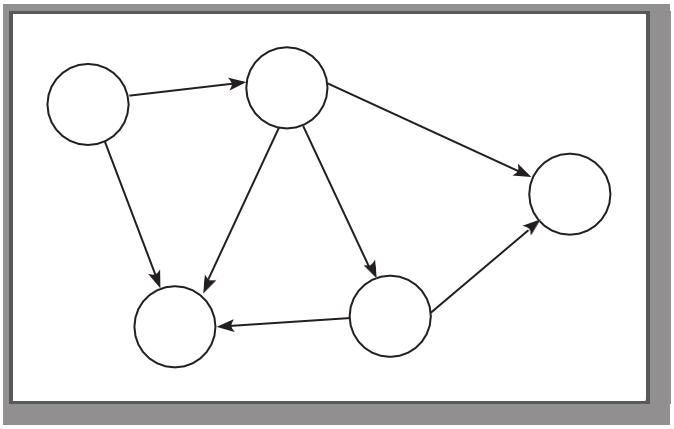


Figure 4: Directed Graph

12.3.1 General Application for Directed Graphs

Directed graphs are essential for the designing of specific classes of concrete structures. A general directed graph is a genealogical or phylogenetic tree that plots the relationship among successors and their antecedents.

12.3.2 Undirected Graphs

It is a set of nodes and of links among the nodes are known as an undirected graph. An individual node is termed as vertex and an individual link is termed as edge, which connects two vertices. An undirected graph contains a fixed set of vertices and a fixed set of edges together.

It possesses directionless edges. The edges denote a two-way correspondence, in which every edge could be travelled in both directions. Figure 5 displays an undirected graph:

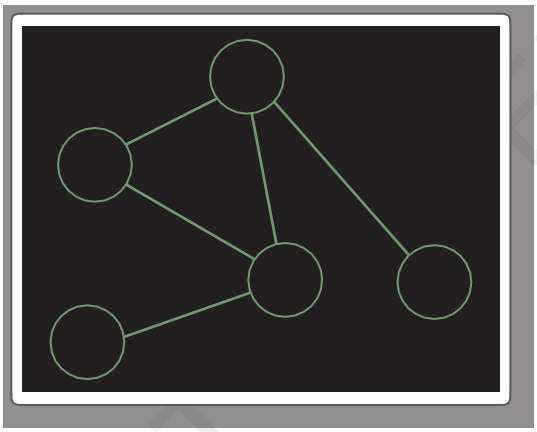


Figure 5: Undirected Graph

12.3.3 General Applications of Undirected Graphs

One of the famous undirected graphs in the world of computers is the topology of links with respect to a computer network. The graph is undirected to manifest the interconnection persisting among nodes. Other recognised instances of undirected graphs are the topology of social networking platforms.

12.4 ADJACENCY LIST AND ADJACENCY MATRIX REPRESENTATION OF GRAPHS

Diagrammatically, it is easy to represent graphs by making vertices and lines connecting them in the form of edges. However, representing them in the form of a computer program is a bit critical. Therefore, we need to develop some notations to represent vertices and edges. There are mainly two methods of representing graphs:

- Adjacency matrix
- Adjacency list

12.4.1 Adjacency Matrix

Adjacency Matrix is a method by which we represent a graph in the form of a matrix. The first row and first column of a matrix represent the vertices and the rest of the elements represent the edges. The elements are represented as m and jij, where i is the row number and j is the column number, and it

represents the edge from vertex i to vertex j. The value of the element mij is set to 1 if there is an edge from vertex i to vertex j; otherwise, its value is set to 0. Consider the graph shown in Figure 6:

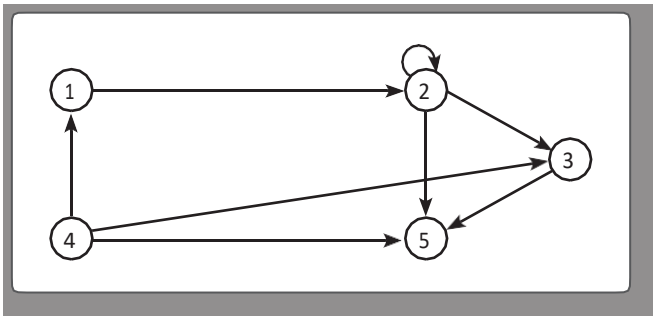


Figure 6: A Graph

Now, its adjacency matrix representation is:

5, 5 1 2 3 4 5

1 0 1 0 0 0

2 0 1 1 0 1

3 0 0 0 0 1

4 1 0 1 0 1

5 0 0 0 0 0

The first row and first column represent the vertices. If there is an edge between the vertices, i and j, then it will be the one at the intersection of the ith row and jth columns. However, the absence of an edge between i and j will be represented by 0. The number of non-zero elements of an adjacency matrix indicates the number of edges in a directed graph.

Following are the advantages of representing an adjacency matrix:

- The degree of any vertex "i" of an undirected graph can be easily computed from the adjacency matrix of that graph, just by counting the number of 1s in the i-th row. In the case of a directed graph, the in-degree (the number of edges having the direction towards the vertex) of a vertex ''i" is the number of 1s in the i-th column and the out-degree (the number of edges with direction away from the vertex) of "i" is the number of 1s in the ith row of the adjacency matrix.
- An adjacency matrix is also suitable to represent a weighted graph. Instead of assigning "1" to the element mij (when there is an edge from vertex i to vertex j), the weight of the edge is assigned to the element. This means that if the weight of the edge from vertex I to vertex j is 5, then the value of the element mij is set to 5. If there is no edge in between the vertices i and j, then infinity is assigned to the element mij.

Following are the drawbacks of representing an adjacency matrix:

- This representation requires n2 elements to represent a graph having 'n' vertices (as the first row and first column of the matrix represent the vertices). If a directed graph has "e" edges, then the number of elements in the matrix having the 0 value is n2-e. So, there is a lot of memory wastage if the number of edges is less in a graph.
- Parallel edges cannot be represented by an adjacency matrix.

The following C++ program is used to implement adjacency matrix are as follows:

```
#include<iostream>
using namespace std;
int vertArr[10][10]; //the adjacency matrix initially 0
int count = 0;
void DisplayMatrix(int V) {
   int i, j;
   for(i = 0; i < V; i++) {
      for(j = 0; j < V; j++) {
         cout << vertArr[i][j] << " ";
      }
      cout << endl;
   }
}
void add_edge(int U, int V) {
   vertArr[U][V] = 1;
   vertArr[V][U] = 1;
}
main(int argc, char* argv[]) {
   int V = 6;    //six vertices in graph
   add_edge(0, 4);
   add_edge(0, 3);
   add_edge(1, 3);
   add_edge(2, 4);
   add_edge(1, 4);
   add_edge(3, 2);
   add_edge(5, 4);
   add_edge(5, 2);
   add_edge(3, 4);
   DisplayMatrix(V);
}
```

The output of given C++ code is as follows:

```
/tmp/06m2U4mQ5U.o
0 0 0 1 1 0
0 0 0 1 1 0
0 0 0 1 1 1
1 1 1 0 1 0
1 1 1 1 0 1
0 0 1 0 1 0
```

12.4.2 Adjacency List

Adjacency list is an efficient means of representing a graph G= (V, E) where V is a set of all the vertices and E is a set of all the edges in the graph G. It uses linked lists of adjacent vertices for all the vertices in 'V'. This approach creates separate linked lists for each vertex in 'V'.

Let us consider the graph shown in Figure 7:

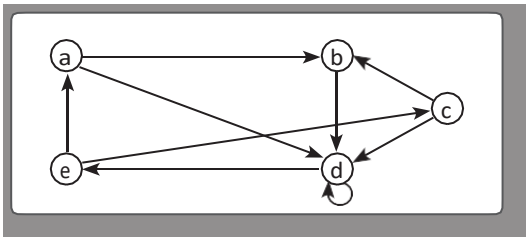


Figure 7: A Graph

Now, to view its Adjacency List representation, you can have a look at Figure 8:

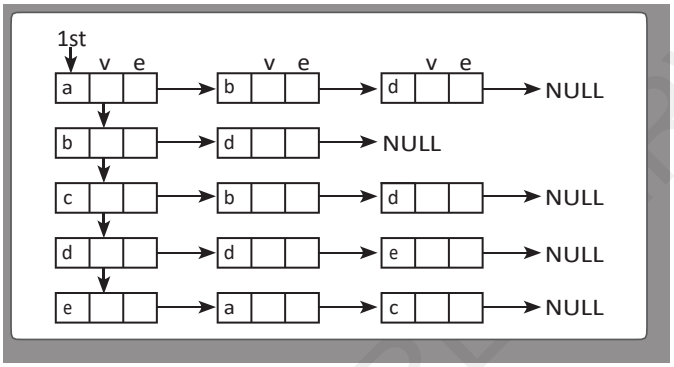


Figure 8: Adjacency List Representation

It generally uses less memory when a large number of vertices and a small number of edges are contained in a graph. The total number of nodes in an adjacency list for an undirected graph with “n” vertices and “e” edges are (n+2e); and the same for a directed graph is (n+ e).

If “e” is very large, then due to the overhead of maintaining pointers, the adjacency list representation may become more wasteful as compared to the adjacency matrix representation.

The degree of a vertex in an undirected graph is the length of the list pointed to by that vertex, but the computation of in-degree of a vertex in a directed graph may require O(e): number of comparisons, as all the lists pointed to by all the vertices may have to be examined.

In ’C++’ language, the adjacency list of a graph may be represented by an array of pointers. Each pointer points to a linked list of the vertices adjacent to a particular vertex. The number of elements of the array would be equal to the number of vertices present in the graph.

The following C++ program is used to implement adjacency list are as follows:

```
#include<iostream>
#include<list>
#include<iterator>
using namespace std;
void DisplayAdjList(list<int> Adj_List[], int V) {
   for(int i = 0; i<V; i++) {
      cout << i << "--->";
      list<int> :: iterator it;
      for(it = Adj_List[i].begin(); it != Adj_List[i].end(); ++it) {
         cout << *it << " ";
      }
```

```
      cout << endl;
   }
}
void add_edge(list<int> adj_list[], int x, int y) {
   adj_list[x].push_back(y);
   adj_list[y].push_back(x);
}
main(int argc, char* argv[]) {
   int v = 6;    // total six vertices are there in the graph
   list<int> adj_list[v];
   add_edge(adj_list, 0, 2);
   add_edge(adj_list, 0, 4);
   add_edge(adj_list, 0, 5);
   add_edge(adj_list, 1, 5);
   add_edge(adj_list, 1, 4);
   add_edge(adj_list, 2, 3);
   add_edge(adj_list, 2, 2);
   add_edge(adj_list, 5, 3);
   add_edge(adj_list, 5, 4);
   DisplayAdjList(adj_list, v);
}
```

The output of given C++ code is as follows:

```
/tmp/06m2U4mQ5U.o
0--->2 4 5
1--->5 4
2--->0 3 2 2
3--->2 5
4--->0 1 5
5--->0 1 3 4
```

12.5 ELEMENTARY GRAPH OPERATIONS

Provided a graph G = (V E) and a vertex v in V (G). To view entire vertices in G which are traceable from v (i.e., complete vertices which are linked to v). We have two methods of achieving this: depth-first search and breadth-first search. They function on the directed as well as on undirected graphs. The upcoming discourse considers directionless graphs.

12.6 TRAVERSAL

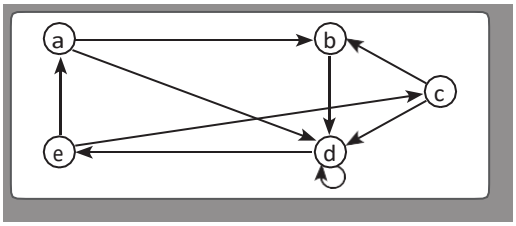
Traversal of a graph means visiting the vertices of the graph and displaying them on the screen. However, the approach that is used in deciding which adjacent vertex has to be visited next (from the current vertex) decides the type of traversal. There are two types of traversals for a graph:

- Depth first traversal
- Breadth first traversal

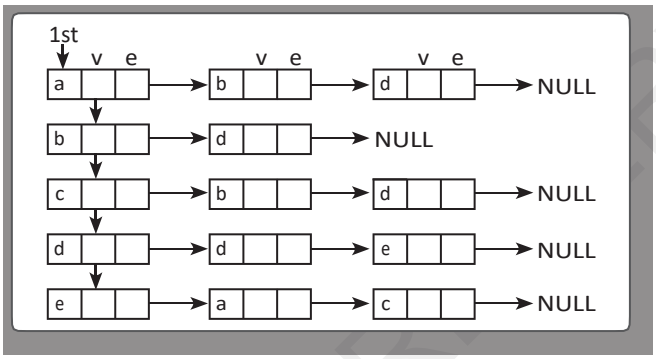
12.6.1 Depth First Traversal

Depth First Traversal tends to create very long and narrow trees. In this traversal, we start at some vertex v, process it (display it) and then recursively traverse all the adjacent vertices.

This traversal creates a spanning forest that can be used to determine if an undirected graph is connected and then to identify the connected components of that undirected graph. We take the help of a stack structure for Breadth First Traversal of a graph. Now, let us consider the graph whose depth first traversal is required, as shown in Figure 9:



The Adjacency List representation of this graph is shown in Figure 9:



We begin the traversal from vertex a. We first store vertex a into a stack, as shown in Figure 10:

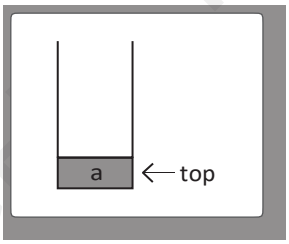


Figure 10: Vertex a Stored into a Stack

Then, we pop it out from the stack and check whether it is marked Y or N. If it is marked N, it means that it is not yet visited; so we visit it, i.e., we display the vertex on the screen and set its marked attribute m to Y (so that is should not be visited again in future). Then, all the nodes connected to the adjacency list of vertex a, i.e., node b and node d are pushed into the stack, as shown in Figure 11:

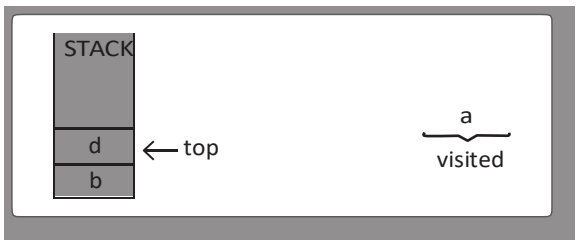


Figure 11: Node b and d are Pushed into the Stack

Now, node d is popped out from the stack and again its marked field, m is checked. If it is N, it is visited, i.e., printed or displayed. After that, its marked field m is set to Y and its adjacency list nodes, i.e., d and e are checked for their m field. If any of them is N, it is pushed into the stack.

Now, since d is already Y, it is not pushed into the stack but node e is pushed into the stack, as shown in Figure 12:

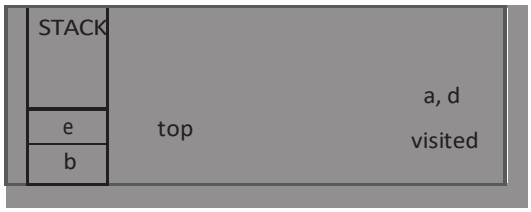


Figure 12: Node e is Pushed into the Stack

Now, e is popped out from the stack, since its field m is marked N, i.e., it is visited. After visiting node I, its m field is set to "Y" and its adjacency list nodes, i.e., a and c are checked, and the nodes having m field equal to N are pushed into the stack. Now, since a is marked Y, we do not enter it into the stack again, but node c is pushed into the stack, as shown in Figure 13:

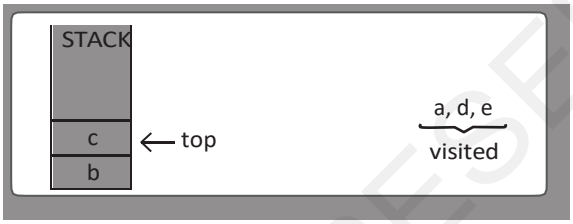


Figure 13: Node c is Pushed In

Now, node c is popped out from the stack, and its marked field m is checked. Since it is N, it is visited and its marked field is set to Y and then its adjacency list nodes, b and d are checked. If any of these nodes have their m field set to N, they are pushed into the stack.

Now since d is marked Y, it is not pushed into the stack, but b is pushed, as shown in Figure 14:

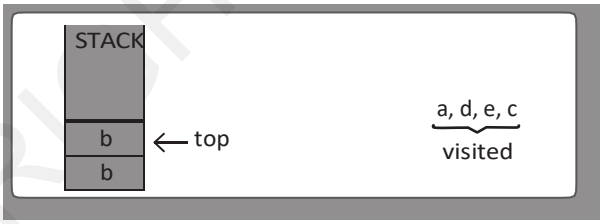


Figure 14: Node b is Pushed into the Stack

Now, the next element b is popped out from the stack. Its marked field m is checked and since it's set to N, it is visited. After that, its m field is set to Y and its adjacency list node d is checked. However, since its m field is set to N, it is not pushed in. Then, the last node b in the stack is popped out, and since its m field is already set to Y, it is not visited. Hence, the depth first traversal of the graph can be shown as: a,d,e,c,b

The following C++ program is used to implement Depth first search are as follows:

```
#include <iostream>
#include <list>
using namespace std;
class DFSGraph
{
int v;    // No. of vertices
list<int> *adjList;
void DFS_util(int v, bool visited[]);
```

```
public:
    // Constructor of a class
DFSGraph(int V)
    {
 this->v = V;
 adjList = new list<int>[V];
    }
void addEdge(int x, int y){
adjList[x].push_back(y);
    }

void DFS();
};
void DFSGraph::DFS_util(int v, bool visited[])
{
visited[v] = true;
cout << v << " ";

    //Process all the adjacent vertices of node recursively
list<int>::iterator i;
for(i = adjList[v].begin(); i != adjList[v].end(); ++i)
if(!visited[*i])
DFS_util(*i, visited);
}
void DFSGraph::DFS()
{
bool *visited = new bool[v];
for (int i = 0; i < v; i++)
visited[i] = false;

    // Analyse the vertices one by one by recursively
for (int i = 0; i < v; i++)
if (visited[i] == false)
DFS_util(i, visited);
}

int main()
{
    // Create a graph
DFSGraph gdfs(5);
gdfs.addEdge(0, 2);
gdfs.addEdge(0, 3);
gdfs.addEdge(0, 4);
gdfs.addEdge(1,3);
gdfs.addEdge(2, 1);
gdfs.addEdge(3, 3);
gdfs.addEdge(3, 4);
cout << "Depth-First Traversal for the Given Graph is:"<<endl;
gdfs.DFS();
return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/O6tCuX54rt.o
Depth-First Traversal for the Given Graph is:
0 2 1 3 4
```

12.6.2 Breadth First Traversal

Traversal means visiting each of the vertices of a graph, exactly once. Breadth First Traversal is the traversal that tends to create very wide short trees. It operates by vertices in layers, i.e., the vertices closest to the start are evaluated first and the most distant vertices are evaluated last. Hence, we can say that Breadth First Traversal is the level-by-level traversal of a tree.

Breadth First Traversal is used to determine whether a graph is cyclic or not. For a directed graph, this is detected when a back edge is found. For an undirected graph, it is detected when a cross edge within the same tree is found. This traversal can also be used to find the shortest path from one node to another. We take the help of a Queue structure for the Breadth First Traversal of a Graph.

In the preceding code, we are considering a graph whose Breadth First Traversal is required. You can find such a graph in Figure 15:

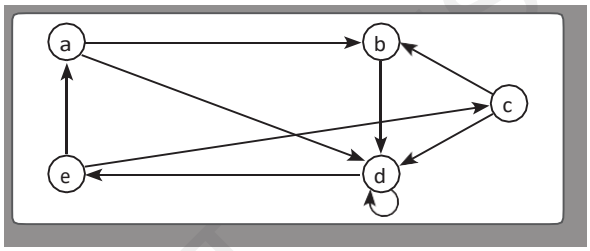


Figure 15: A Graph

The Adjacency List representation of this graph is shown in Figure 16:

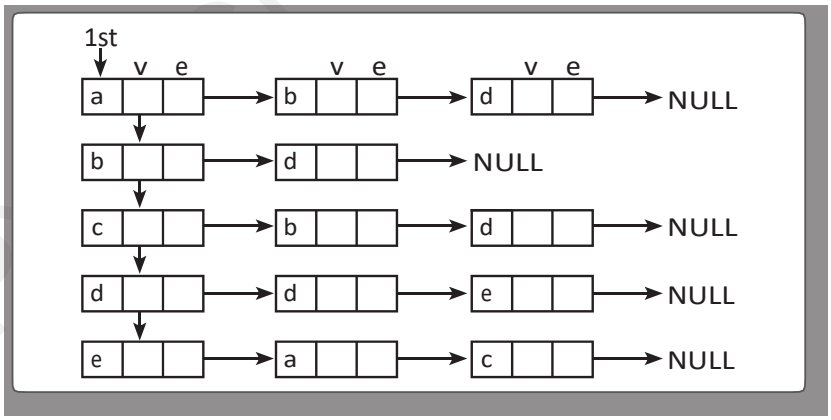


Figure 16: Adjacency List Representation

In this List representation, we begin our traversal from vertex a. We first store vertex into a queue, as shown in Figure 17:

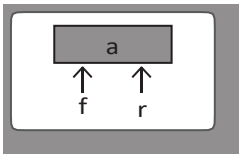


Figure 17: Vertex a

Then, we remove it from the queue and check whether it is marked Y or N. If it is marked N, it means that it is not yet visited; so we visit it, i.e., we display the vertex on the screen and set its marked attribute m to Y (so that is should not be visited again in future). Then, all the nodes connected to the adjacency list of vertex a, i.e., node b and node d, are stored in the queue, as shown in Figure 18:

Figure 18: Node b and d are Stored in the Queue

Now, when we remove node b from the queue, we again check its marked field m. If the marked field is N, the node is visited, i.e., printed. After this, its marked attribute m is set to Y and its adjacency list node, i.e., d is inserted into the queue, as shown in Figure 19:

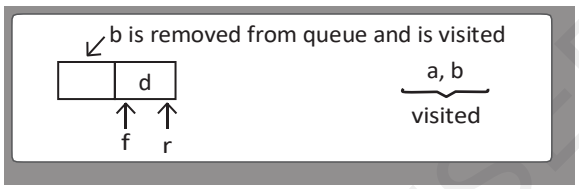


Figure 19: Node d is Inserted into the Queue

Now, d is removed from the queue, since its field m is marked N, i.e., it is visited. Then, its m field is set to Y and its adjacency list nodes, i.e., d and e are checked, and the nodes which have m field equal to N are inserted into the queue. Since d is marked Y, it is not entered into the queue again, but node e is added to the queue, as shown in Figure 20:

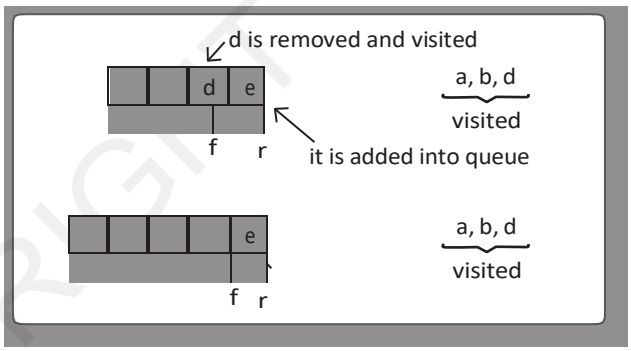


Figure 20: Node e is Added to the Queue

Now, the next element e is removed from the queue. Its marked field m is checked and since it is set to N, node e is visited. Then, its m field is set to Y and its adjacency list nodes, a and c are checked, and if any of the nodes have their m field set to N, they are added to the queue.

Now, since node a is marked Y, it is not added to the queue, but node c is added to the queue, as shown in Figure 21:

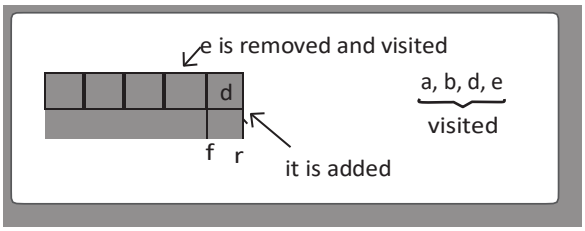


Figure 21: Node c is Added to the Queue

Now, node c is removed from the queue and since its marked field m is N, it is visited and is set to Y and its adjacency elements, b and d are checked. If any of their marked fields represented by m are N, they are added to the queue. However, since both the nodes have been visited earlier, none is added to the queue. So, the Breadth First Traversal of the graph can be presented in this order: a,b,d,e,c.

The following C++ program is used to implement breadth first search are as follows:

```
#include<iostream>
#include <list>
using namespace std;
class Graph
{
     int v;
     list<int> *adj;
public:
     Graph(int v); // Constructor
     void addEdge(int a, int b);
     // Display BFS traversal from a given source S
     void BFS(int S);
};
Graph::Graph(int v)
{
     this->v = v;
     adj = new list<int>[v];
}
void Graph::addEdge(int a, int b)
{
     adj[a].push_back(b);
}
void Graph::BFS(int S)
{
     bool *visited = new bool[v];
     for(int i = 0; i < v; i++)
          visited[i] = false;
     // Create a queue for Breadth First Search
     list<int> Queue;
     // Mark the existing node
     visited[S] = true;
     Queue.push_back(S);
     list<int>::iterator i;
     while(!Queue.empty())
     {
          S = Queue.front();
          cout << S << " ";
          Queue.pop_front();
          for (i = adj[S].begin(); i != adj[S].end(); ++i)
          {
               if (!visited[*i])
               {
                    visited[*i] = true;
                    Queue.push_back(*i);
               }
```

```
            }
      }
}
//    Test the methods of graph class
int main()
{
      // Generating a graph
      Graph g(4);
      g.addEdge(0, 1);
      g.addEdge(0, 2);
      g.addEdge(1, 2);
      g.addEdge(2, 3);
      g.addEdge(2, 0);
      g.addEdge(3, 1);
      cout << " Breadth First Traversal "
            << "(starting from vertex 2) is: \n";
      g.BFS(2);
      return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/06m2U4mQ5U.o
```

Breadth First Traversal (starting from vertex 2) is:

```
2 3 0 1
```

12.7 LAB EXERCISE

12(a): Write a Program in C++ to implement Graph (G) using nodes and vertices.

The following C++ program is used to implement a graph using nodes and vertices are as follows:

```
// A simple representation of graph using STL
#include <bits/stdc++.h>
using namespace std;
// undirected graph.
void addEdge(vector<int> adj[], int a, int b)
{
      adj[a].push_back(b);
      adj[b].push_back(a);
}
void printGraph(vector<int> adj[], int V)
{
      for (int v = 0; v < V; ++v) {
            cout << "\n Adjacency list of vertex " << v
                  << "\n head ";
            for (auto x : adj[v])
                  cout << "-> " << x;
            printf("\n");
      }
}
int main()
```

```
{
     int V = 5;
     vector<int> adj[V];
     addEdge(adj, 0, 1);
     addEdge(adj, 0, 4);
     addEdge(adj, 1, 4);
     addEdge(adj, 1, 2);
     addEdge(adj, 1, 3);
     addEdge(adj, 2, 1);
     addEdge(adj, 3, 4);
     printGraph(adj, V);
     return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/U0yXDoXAIp.o
Adjacency list of vertex 0
 head -> 1-> 4
 Adjacency list of vertex 1
 head -> 0-> 4-> 2-> 3-> 2
 Adjacency list of vertex 2
 head -> 1-> 1
 Adjacency list of vertex 3
 head -> 1-> 4
 Adjacency list of vertex 4
 head -> 0-> 1-> 3
```

12(b): Write a Program in C++ to check whether a given graph is connected or not.

The following C++ program is used to check a given graph is connected or not are as follows:

```
#include<iostream>
#define node 5
using namespace std;
int graph[node][node] = {{0, 1, 0, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 0, 1, 1},
   {1, 0, 0, 0, 0},
   {0, 1, 0, 0, 0}
};
void Traverse(int U, bool visited[]){
   visited[U] = true;
   for(int v = 0; v<node; v++){
      if(graph[U][v]){
         if(!visited[v])
         Traverse(v, visited);
      }
   }
}
bool isConnected(){
   bool *vis = new bool[node];
```

```
   for(int U; U < node; U++){
      for(int i = 0; i<node; i++)
      vis[i] = false; //initialize as no node is visited
      Traverse(U, vis);
      for(int i = 0; i<node; i++){
         if(!vis[i]) //Graph is not connected if node is not visited to
the traversal
         return false;
      }
   }
   return true;
}
int main(){
   if(isConnected())
   cout << "The Graph is connected.";
   else
   cout << "The Graph is not connected.";
}
```

The output of given C++ code is as follows:

```
/tmp/06m2U4mQ5U.o
```

The Graph is connected.

12.8 CONCLUSION

- A graph is a sort of a tree (with or without cycles), in which each node is called a vertex and is connected with lines, called edges.
- A graph is called as simple when no parallel and auto-loop edges exist.
- A graph having the undirected as well as the directed edges is called as mixed graph.
- An edge refers to interconnected linkage among two vertices.
- A collection or a set of vertices of directed edges that links with an individual ordered pair of vertices is known as directed graph.
- It is a set of nodes and links among the nodes is known as an undirected graph.
- Adjacency Matrix is a method by which we represent a graph in the form of a matrix.
- Traversal of a graph means visiting the vertices of the graph and displaying them on the screen.
- Depth First Traversal tends to create very long and narrow trees.
- Breadth First Traversal is the traversal that tends to create very wide short trees.

12.9 GLOSSARY

- Graph: It is a sort of a tree (with or without cycles), in which each node is called a vertex and is connected with lines, called edges.
- Simple graph: It is a graph called as simple when no parallel and auto-loop edges exist.
- Mixed graph: It is a graph having the undirected as well as the directed edges is called mixed graph.

- Edge: It refers to interconnected linkage among two vertices.
- Directed graph: It is a collection or a set of vertices of directed edges that links with an individual ordered pair of vertices is known as a directed graph.
- Undirected graph: It is a set of nodes and links among the nodes is known as an undirected graph.
- Adjacency matrix: It is a method by which we represent a graph in the form of a matrix.
- Traversal: It means visiting the vertices of the graph and displaying them on the screen.
- Depth first traversal: It tends to create very long and narrow trees.
- Breadth first traversal: It is the traversal that tends to create very wide short trees.

12.10 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. Define the term Graph.
2. It possesses directed edges, which denote a one-way relationship, where individual edge could be traversed in a unidirectional way alone. Outline the concept of directed graph.
3. The approach that is used in deciding which adjacent vertex has to be visited next (from the current vertex) decides the type of traversal. Explain the term traversal and also, discuss its types.
4. Describe the different graph terminologies.
5. It possesses directionless edges. The edges denote a two-way correspondence, in which every edge could be travelled in both directions. Elucidate the concept of undirected graph.

12.11 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. A graph is a sort of a tree (with or without cycles), in which each node is called a vertex and is connected with lines, called edges. Refer to Section Introduction
2. A collection or a set of vertices of directed edges that links with an individual ordered pair of vertices is known as directed graph or it is also termed as digraph. Refer to Section Directed and Undirected Graphs
3. Traversal of a graph means visiting the vertices of the graph and displaying them on the screen.

   There are two types of traversals for a graph:

   - Depth first traversal
   - Breadth first traversal

   Refer to Section Traversal
4. The graph data structure has multiple terminologies which are as follows:-
   - Vertex: Individual data element of a graph is called as Vertex. It is also known as node. In above example graph, A, B, C, D & E are known as vertices.
   - Edge: An edge refers to interconnected linkage among two vertices. Alternatively an edge is termed as an Arc. An edge has a (Starting Vertex and an Ending Vertex). For instance, graph

above has the link among vertices A and B denotes as (A, B). On the whole, 7 edges (i.e., (A, B), (A, C), (A, D), (B, D), (B, E), (C, D) and (D, E) exists.

Refer to Section Graph Terminology

5. It is a set of nodes and of links among the nodes is known as undirected graph. An individual node is termed as vertex and an individual link is termed as edge, which connects two vertices. Refer to Section Directed and Undirected Graphs

12.12 POST-UNIT READING MATERIAL

- https://www.google.co.in/books/edition/Data_Structures_and_Algorithms/1lCHYj5eV-EC?hl=en&gbpv=1&dq=importance+of+graphs+in+DSA&pg=PA291&printsec=frontcover
- https://www.differencebetween.com/difference-between-directed-and-vs-undirected-graph/

12.13 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates about the concept of graphs and its terminologies. Also, discuss about the traversal of graphs and its effectiveness.

UNIT

13

Sorting and Searching

Names of Sub-Units

Implementation of Sorting Technique, Insertion Sort, Radix Sort, Address Calculation Sort using Hashing, Searching Algorithms, Linear Search, Binary Search, Jump Search, Interpolation Search

Overview

This unit begins by discussing about the concept of sorting and searching. Next, the unit outlines the insertion sort, radix sort and address calculation sort. Further, the unit explains the searching algorithms, sequential search and binary search. Towards the end, the unit covers the jump search and interpolation search.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of sorting and searching
- Explain the concept of insertion sort, radix sort and address calculation sort
- Describe the searching algorithms and sequential search
- Outline the significance of binary search
- Discuss the concept jump search and interpolation search

Learning Outcomes

At the end of this unit, you would:

- Assess the concept of insertion sort, radix sort and address calculation sort
- Evaluate the importance of searching algorithms and sequential search
- Determine the significance of binary search
- Understand the concept jump search and interpolation search

	Pre-Unit Preparatory Material
⌘ https://www.srividyaengg.ac.in/coursematerial/ECE/106325.pdf	

13.1 INTRODUCTION

Searching and sorting are the two most important tasks applied to data structures. Searching is the process by which we try to find whether the required data exists in the given structure or not. On the other hand, sorting is a task in which we arrange the given structure in a particular order, i.e., ascending order or descending order.

Since searching in a sorted structure can be quite faster than searching in a non-sorted structure, the two tasks of searching and sorting are usually collectively applied and due to this, we refer to the two tasks together. Both sorting and searching processes are done by iterating through the elements of a structure with the help of loops. So, before using any algorithm for sorting and searching, we need to find out its efficiency, i.e., how much time and memory space it would require for sorting or searching a structure.

13.2 IMPLEMENTATION OF SORTING TECHNIQUE

Sorting is the process of putting data in a predetermined order. The sorting algorithm specifies how data should be organised in a particular order. Numerical and lexicographical ordering are the two most common types of order.

Sorting is important because it is thought that if data is organised logically data searching would be substantially enhanced. Data can also be sorted to make it more readable.

13.2.1 Insertion Sort

In insertion sort, an element is placed at its correct position by moving all the elements greater than this element to its right. In every pass, the element p[i], is compared with p[i-1], p[i-2]…elements until either an element smaller than p[i] is found or the beginning of the array is reached. All the elements greater than p[i] are moved to their right position.

Let's consider the array shown in Figure 1:

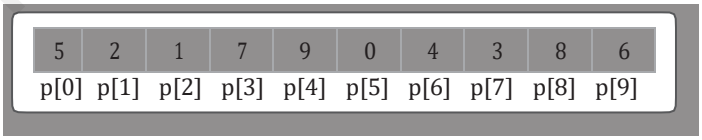


Figure 1: An Array p

As you can see in this array, there are 10 elements and therefore, there will be 9 iterations. During the first iteration, p[1] is compared with p[0]. If p[1] > p[0], then it is fine; otherwise the interchanging takes place.

During the second iteration, p[2] is compared with p[1] and p[1] with p[0]. Again, if p[2]>p[1], we consider it to be fine; otherwise, the interchanging of p[2] and p[1] takes place. Then, p[1] and p[0] are compared. Now, if p[1] >p[0], nothing changes; otherwise, interchanging takes place. Now, let the elements of our array are:

5 2 1 7 9 0 4 3 8 6

Hence, in the space below, we move to the iterations related to this array:

- Iteration 1: p[1] > p[0] is not true: so interchanging take place : 2 5 1 7 9 0 4 3 8 6
- Iteration 2: p[2] > p[1] is not true, so interchanging of p[2] and p[1] takes place : 2

  1 5 7

  9 0 4 3 8 6

  p[1] > p[0] is not true, so interchanging of p[1] and p[0] takes place : 1 2

  5 7

  9 0 4 3 8 6
- Iteration 3: In iteration 3, p[3] is compared with p[2], p[2] is compared with p[1] and

  p[1] is compared with p[0]

  p[3] >p[2] is true, so no interchanging takes place : 1 2 5 7 9 0 4 3 8 6

  p[2] > p[1] is true and p[1] >p[0] is also true, so no more interchanging takes place
- Iteration 4: During iteration 4, p[4] is compared with p[3]: p[3] is compared with p[2]; p[2] is compared with p[1] and p[1] is compared with p[0].

  1 2 5 7 9 0 4 3 8 6----- Output of iteration 3

  p[4] > p[3] is true, so no interchanging takes place Similarly, the conditions p[3] >p[2], p[2]>p[1] and p[1] >p[0] are also true: therefore no interchanging is required.
- Iteration 5: During iteration 5, p[5] is compared with p[4], p[4] is compared with p[3], p[3] is compared with p[2], p[2] is compared with p[1] and p[1] is compared with p[0].

  p[5]>p[4] is not true, so the interchanging of p[5] and p[4] takes place :

  1 2

  5 7 0 9 4 3 8 6

  p[4] >p[3] is not true, so the interchanging of p[4] and p[3] takes place :

  1 2

  5 0 7 9 4 3 8 6

  p[3] >p[2] is not true, so the interchanging of p[3] and p[2] takes place :

  1 2

  0 5 7 9 4 3 8 6

  p[2] >p[1] is not true, so the interchanging of p[2] and p[1] takes place :

  1 0

  2 5 7 9 4 3 8 6

  p[1] >p[0] is not true, so the interchanging of p[1] and p[0] takes place :

  0 1

  2 5 7 9 4 3 8 6
- Iteration 6: During this iteration also, p[6] is compared with p[5], p[5] with p[4].....p[1] with p[0]. In other words, p[j] is compared with p[j-1] and if in any p[j]> p[j-1], we consider the condition to be true;

and therefore, interchanging takes place, as shown in Figures 2 to 5:

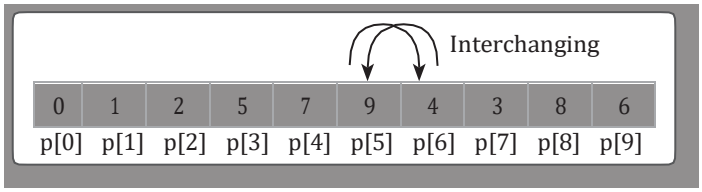


Figure 2: Interchanging Takes Place

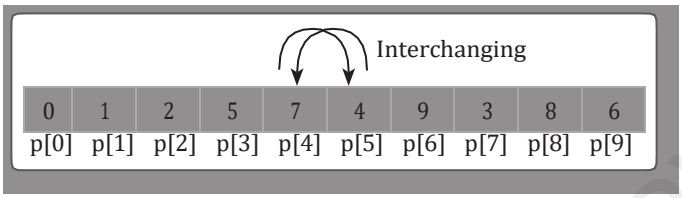


Figure 3: Interchanging Takes Place

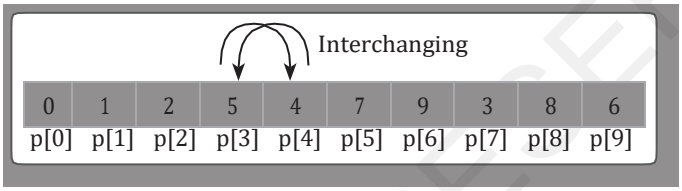


Figure 4: Interchanging Takes Place

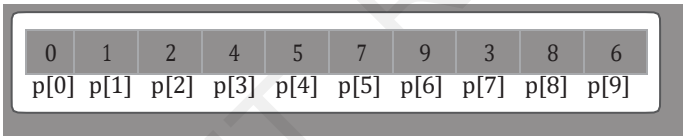


Figure 5: No Interchanging

Now, since p[3]>p[2], p[2]>p[1], p[1]>p[0] conditions are true, no more interchanging is desired. Similarly after iteration 1, our array is like the one shown in Figure 6:

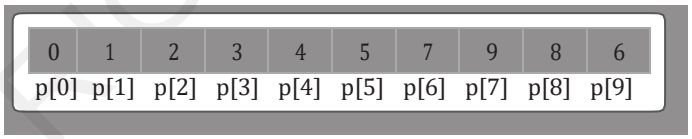


Figure 6: The Array After Iteration 1

After iteration 2, the array becomes like the one shown in Figure 7:

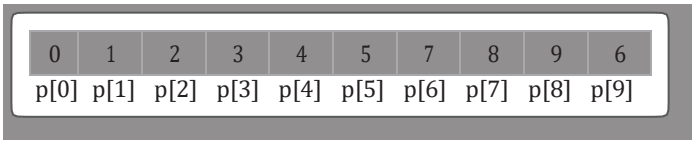


Figure 7: The Array After Iteration 2

After the 3rd iteration, our array is sorted as shown in Figure 8:

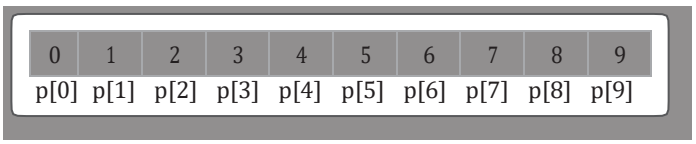


Figure 8: The Array After Iteration 3

If the initial file is sorted, then only one comparison is made on each pass; so the number of comparisons is O(n). If the file is initially sorted in the reverse order, the number of comparisons is O(n2).

The simple insertion sort is considered better than the bubble sort. The closer the file is to the sorted order, the more efficient the insertion sort becomes. The average number of comparisons in the insertion sort is also O(n2) and the space requirements for the sort consist of only one temporary variable.

The mechanism of the Insertion sort algorithm resembles the technique when we arrange playing cards in our hands. We divide the array implicitly into an ordered and an unordered part. Elements are collected from the unordered portion and are assigned to a suitable location in the ordered portion.

Algorithm of Insertion Sort

To understand this algorithm, let's consider a function Insertion Sort (p,n). This function states that p is the array of n elements. Let's now come to the algorithm for the insertion sort method:

Step 1: Initialise a variable, say i to 1

i=1

Step 2: Increment the value of i by 1

The value of i is incremented by 1 after every iteration, i.e., i=1, 2, 3....

n-1

Repeat the steps 2 to 5 for n-1 times, while i>=n-1

Step 3: Initialise a variable j to the value of i

j=i

Step 4: If p[j] <p[j-1], then interchange their values

temp=p[j]

p[j]=p[j-1]

p[j-1]=k

Step 5: Repeat step 4 for j >=0

The value of j is decremented by 1 after every iteration i.e. j=i, i-1, i-2,

....0

Step 6: Exit

The following C++ program is used to implement the insertion sort are as follows:

```
#include<iostream>
using namespace std;
void dis(int *arr, int size) {
   for(int i = 0; i<size; i++)
      cout << arr[i] << " ";
   cout << endl;
}
void insertionSort(int *arr, int size) {
   int key, j;
   for(int i = 1; i<size; i++) {
      key = arr[i];
```

```
        j = i;
        while(j > 0 && arr[j-1]>key) {
           arr[j] = arr[j-1];
           j--;
        }
        arr[j] = key;
     }
  }
  int main() {
     int n;
     cout << "Enter the number of elements: ";
     cin >> n;
     int arr[n];
     cout << "Enter elements:" << endl;
     for(int i = 0; i<n; i++) {
        cin >> arr[i];
     }
     cout << "Array before Sorting: ";
     dis(arr, n);
     insertionSort(arr, n);
     cout << "Array after Sorting: ";
     dis(arr, n);
  }
```

The output of given C++ code is as follows:

```
/tmp/1hKnRk2uPM.o
Enter the number of elements: 4
Enter elements:
33 45 66 56
Array before Sorting: 33 45 66 56
Array after Sorting: 33 45 56 66
```

13.2.2 Radix Sort

In Radix sort, the numerical data is sorted in n passes, where n is the maximum number of digits in the numerical data. For example, if the numerical data being sorted consists of 3 digits, the Radix sort requires 3 passes to sort them. Sorting begins by arranging the numerical data on the basis of LSD (Least Significant Digit) and with every pass, we move towards MSD (Most Significant Digit). We begin with the LSD and store the numerical data in a two dimensional array in the ascending order of LSD. In the second pass, the numerical data (that was sorted on LSD) is again placed in the two dimensional array; but this time in the ascending order of the next significant digit. In this way, with every pass, we move towards the MSD. The last pass consists of storing the numerical data in ascending order of MSD in the two dimensional array.

For example, let the numerical data to be sorted is:

```
167, 201, 590, 110, 325, 429, 831, 908, 746, 674
```

The important thing to note here is that the size of all the elements is same, i.e., all elements are 3 digit numbers. If any numerical has few digits, it is padded with 0's. For example, if a numerical is 23, it is

written as 023 to make it of 3 digits. Now, here, since the data is of 3 digits, there are 3 passes to sort this data.

In the first pass, we store the data in a two dimensional array on the basis of LSD. This means that all the numerical with LSD as 0, are placed in the 0'th row and all the numerical with LSD as 1 are placed in the 1st row and so on, as shown in Figure 9:

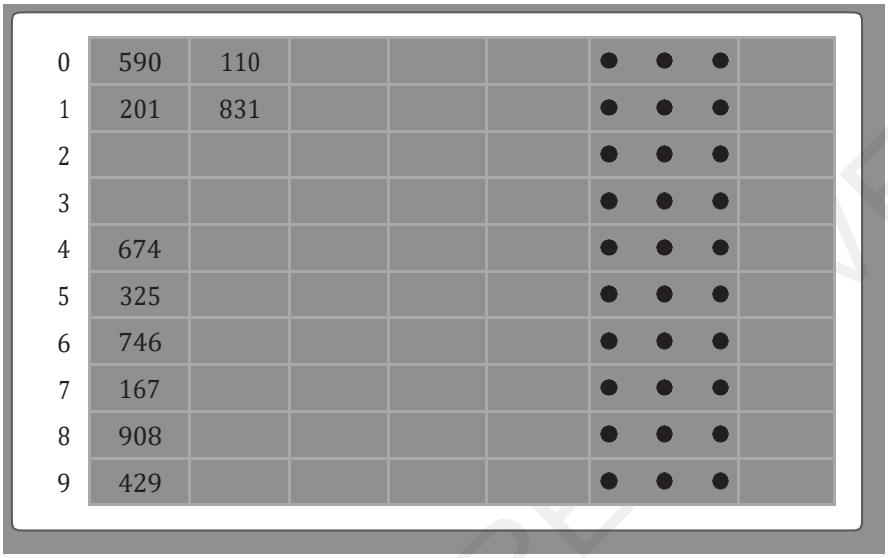


Figure 9: The numerical placed in a two dimensional array on the basis of LSD

The numerical data arranged on the basis of LSD is taken out of the two dimensional array. The sequence of data is: 590, 110, 201, 831, 674, 325, 746, 167, 908 and 429. This data is again placed in the two dimensional array on the basis of the middle digit. That is, all the numericals with the middle digit as 0 are placed in the 0'th row and all the numerical with the middle digit as 1 are placed in the 1st row and so on, as shown in Figure 10:

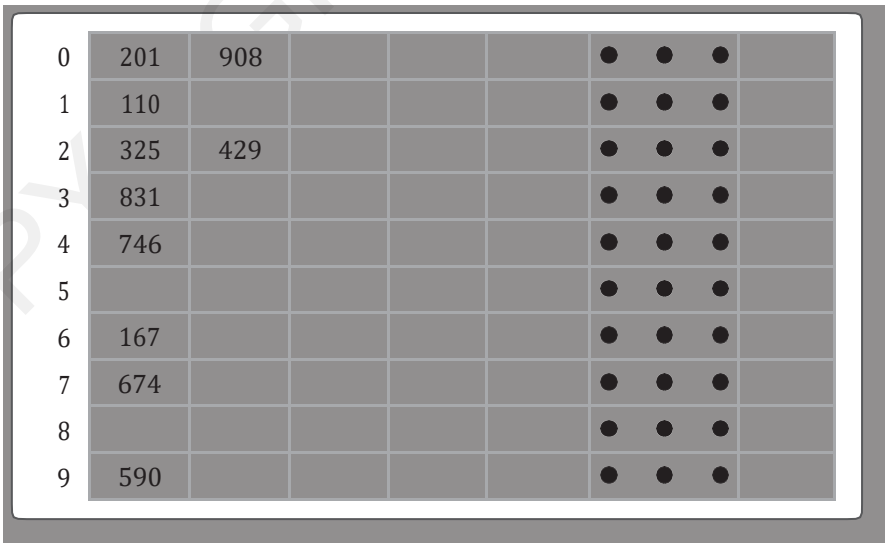


Figure 10: The numerical placed in two dimensional array on the basis of middle digit

The numerical data arranged on the basis of the middle digit is taken out of the two dimensional array. The sequence of data will be: 201, 908, 110, 325, 429, 831, 746, 167, 674 and 590. This data is again placed in the two dimensional array on the basis of MSD.

That is, all the numericals with MSD as 0 are placed in the 0'th row and all the numerical with MSD as 1 are placed in the 1st row and so on, as shown in Figure 11:

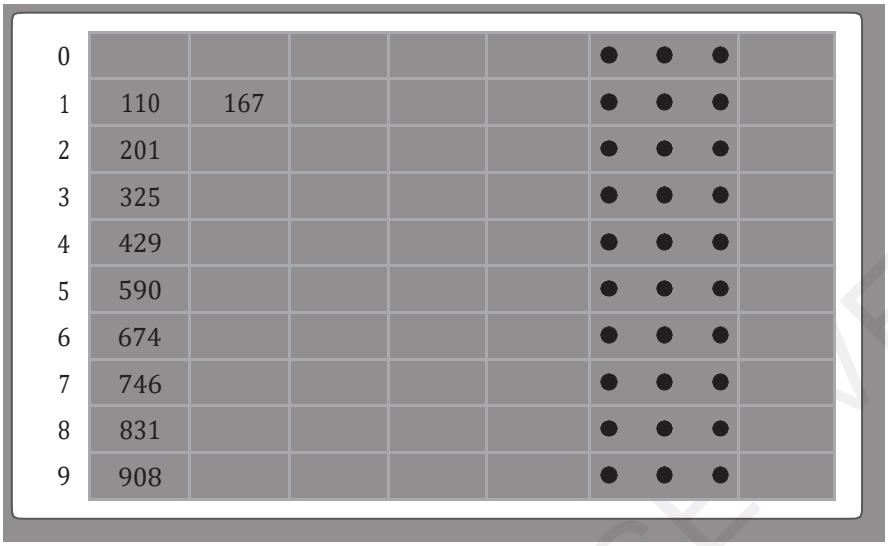


Figure 11: The numerical placed in two dimensional array on the basis of MSD

We get the numerical sorted as shown in Figure 11. The sorted sequence is 110, 167, 201, 325, 429, 590, 674, 746, 831, and 908. The following C++ program is used to show radix sort is as follows:

```
#include<iostream>
#include<list>
#include<cmath>
using namespace std;
void Dis(int *Arr, int size) {
   for(int i = 0; i<size; i++)
      cout << Arr[i] << " ";
   cout << endl;
}
void radixSort(int *arr, int n, int max) {
   int a,b , c, d = 1, index, temperature, count = 0;
   list<int> pocket[10];
   for(a = 0; a< max; a++) {
      c = pow(10, a+1);
      d = pow(10, a);
      for(b = 0; b<n; b++) {
         temperature = arr[b]%c;
         index = temperature/d;
         pocket[index].push_back(arr[b]);
      }
      count = 0;
      for(b = 0; b<10; b++) {
         while(!pocket[b].empty()) {
            arr[count] = *(pocket[b].begin());
            pocket[b].erase(pocket[b].begin());
            count++;
         }
```

```
      }
   }
}
int main() {
   int n, max;
   cout << "Enter the number of elements: ";
   cin >> n;
   cout << "Enter the maximum digit of elements: ";
   cin >> max;
   int arr[n]; //create an array with given number of elements
   cout << "Enter elements:" << endl;
   for(int i = 0; i<n; i++) {
      cin >> arr[i];
   }
   cout << "Data before Sorting: ";
   Dis(arr, n);
   radixSort(arr, n, max);
   cout << "Data after Sorting: ";
   Dis(arr, n);
}
```

The output of given C++ code is as follows:

```
/tmp/bItJRqfPru.o
Enter the number of elements: 6
Enter the maximum digit of elements: 4
Enter elements:
66 57 99 78 34 44
Data before Sorting: 66 57 99 78 34 44
Data after Sorting: 34 44 57 66 78 99
```

Applications of Radix Sort

In a classic computer, radix sort is utilised for entering the data with various fields. For instance, we wish to apply sort on 3 fields, i.e., day, month and year. In such a case, Radix sort is applied initially on the date, followed by month, and lastly on year..

It was applied in card arranging devices that possessed 80 columns, and in every column, the device punched a hole in 12 different positions. The device then arranged the cards, based upon the position where the card was punched, which was utilised by the operator to pile up the cards for which the initial row was punched, ensued by the 2nd row, the pattern continues till the end.

13.2.3 Address Calculation Sort using Hashing

Hashing is a technique used for a quick retrieval of the desired data from a large volume of data. This scheme is used when a record is stored at a particular address and this address is to be computed by applying a formula: hash function on the key, the primary key of the record. The hash function ( ) is used in the following manner:

```
a=h(k)
```

In this equation, a is the address computed at the time of the application of the hash function on the k key of the record. The hash function should be selected in such a way that it results in a unique

address, every time it is used. However, it is not practically feasible because there are frequent chances of "collision" i.e. we may get the address of the record with the k1 key, where already a record with the k2 key is stored. These collided records are called synonyms and we apply certain collision resolution techniques to resolve the conflict.

The domain of the hashing function is the interval of the key. If the keys are of 3 digits, we say the domain of the hash function is (0,999). If the keys are of 5 digits, we say that the domain of the hash function is (0, 99999).

The range of the hash function is the capacity of the storage, where the records are stored. If the array where the records are stored consists of 1000 addresses, we say that the range of the hash function is (0,999).

The hash function should be chosen so that it distributes the keys uniformly over the range of the storage. If the total capacity of the storage is n, then the good function should distribute the keys over the range (0, n-1).

This algorithm makes use of a Hash Function f with the trait of Order Preserving Function. The algorithm utilises an address table to hold the items which persists as a list (or an array) of Linked lists. The Hash function is computed upon the individual value of the array to evaluate the respective address in the address table. Following this, the elements are entered at their respective locations in an arranged way by contrasting them with the elements existing priory in that address.

Post of entering the elements, the elements at every memory location in the address table are arranged. Therefore we repeat through each memory location individually and enter the data items at that location in the processed array.

13.3 SEARCHING ALGORITHMS

The process of searching for a particular data item in a data structure consists of comparison and advancement operations. On the basis of the technique used for locating the required data in a structure, the searching process is divided into different types. It basically differs in the number of comparisons done to get the desired data. All the searching processes have one thing in common, which is a result, which can be either successful or unsuccessful. When the result of a searching process is successful, it means that the required data is found in the structure and its location is returned. In case of an unsuccessful result, usually, a boolean value "false" is returned. Common Search types are:

- Linear search
- Binary search
- Jump search
- Interpolation search

13.3.1 Linear Search

In Linear Search, each record of a file is searched, one at a time, until the desired data is found. The time required to execute the algorithm is proportional to the number of comparisons. The desired target can be found at the first iteration or else all the elements may be traversed. So, the average behaviour of searching in an element array can be stated as:

```
(1+2+3.....+n)/n = (n+1)/2=O(n)
```

Hence, for a linear search, the average number of comparisons for a file with n records is n/2 and the time taken in both the average and worst cases is O(n). In other words, the time complexity of a

sequential search of an element in an n element array is O(n). On the same basis, we can say that the time requirement for computing 1+2+3+. +n without using the summation formula is O(n).

To get the maximum value or minimum value in an n element array, we require n-1 comparisons. Hence, a function to evaluate both the maximum and minimum elements of an array requires 2 (n-1) numbers of comparisons in the best, worst and average cases.

Function for Linear Search for an Element in an Array:

```
int search ( int p[], int n, int x)
{
int i;
for(i=0;i<n;i++)
{
if(p[i]==x)
break;
}
if (i==n)
return -1;
else
return i;
}
```

In this code, the function returns an integer i, if x is found in array p (i is the index of the array, where x is found); otherwise, it returns -1. The comparison starts from the 0th index value of the array and continues until the value x is found in the array or the array is over. Even in an unsuccessful search, the number of comparisons required to check whether x exists in the array or not is denoted by n. When the search is unsuccessful, its time complexity is O(n).

If the search is successful, the number of comparisons required is one more than the index of the array, where x is found. Therefore, the number of comparisons can be 1,2,3,….. or 'n', depending on the location where x is found in the array.

The average behavior of the algorithm for linear search can be written as:

```
(1+2+3+……+n)n= (n+1)/2 = O(n).
```

Hence, both the successful and unsuccessful searches have O(n) complexity in the average case. Even in the worst case, the search has O(n) complexity.

13.3.2 Binary Search

Binary Search is a search that uses the divide-and-conquer approach. Here, we compare the data to be searched with the data available in the middle of the file (array): If the data to be searched is smaller than the middle of the array, it means we have to concentrate on the first half of the array and if the data to be searched is larger than the middle of the array, it means the data will be found in the second half of the array. Then, we compare the data to be searched with the data in the middle of the selected half, i.e., we determine which quarter of the list contains data. This process is continued until we get the desired data.

The major drawback of this algorithm is that it assumes that we can directly access the middle value in the list. This means that the list must be stored in an array. Moreover, the insertion and deletion in an array is a typical task and requires lots of shuffling of the elements. The complexity of the binary search algorithm is given by log2 n.

The maximum number of comparisons required for a successful search is:

```
2j >= (n+1), where 'j' is the smallest integer
```

The maximum number of elements that are left after the first comparison is 2(j-1)-1 and the maximum number of elements left after 'k' comparisons are 2(j-k)-1. This means that the desired number of elements can be found after 'j' comparisons. Therefore, the maximum number of comparisons for a successful search is given by j, where 'j' is the smallest integer satisfying the equation 2j. Alternatively, we can say that the maximum number of comparisons for a successful search can be written as: log2 (n+1).

The numbers of comparisons for a successful as well as an unsuccessful search are approximately given by O(log2 n).

Though the binary search requires fewer comparisons, each comparison involves more computation. The worst case of time complexity in the binary search algorithm is O(log2 n). We can conclude that the number of comparisons in the best case is 1 and the number of comparisons in the worst case is j where 2j >=n+1. The average number of comparisons is represented by log2n.

Algorithm of Binary Search

To understand the algorithm for binary search, let us consider array p. Its length is n and the value to search is k. After that we can initialise the two variables: lower and upper, as shown in Figure 12:

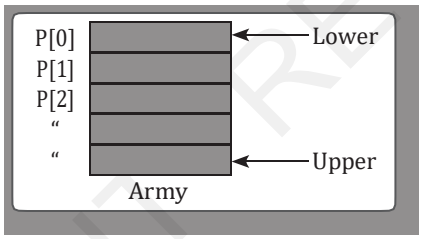


Figure 12: Array p

Lower=0

Upper=n (length of array)

The steps used in the algorithm for the binary search are:

- Step 1: If upper >=lower, then repeat the steps from 2 to 4; else go to step 5
- Step 2: Calculate the middle location of the array by using the following formula:

```
mid=(lower+upper)/2
```

- Step 3: If the value to be searched is found at the location p[mid], then display “Value found” and exit; else go to the next step:

```
if p[mid]=k
print "Value found" and exit
```

- Step 4: If the value to be searched is larger than the middle value of the array, then the search is confined to the lower half of the array. So, the lower limit of the array is set to the middle location of the array:

```
if k >p[mid]
lower=mid+1
```

If the value to be searched is smaller than the middle value of the array, then the search is confined to the upper half of the array. So, the upper limit of the array is set to the middle location of the array:

```
else
upper=mid-1,
```

- Step 5: The control appears at this step if the value is not found. Then, you can display "Value not found" and exit.

13.3.3 Jump Search

The Jump search approach is a new approach for searching a sorted array for a single element.

The primary premise of this searching technique, when compared to a linear search algorithm, is to search a lower number of elements (which scans every element in the array to check if it matches with the element being searched or not). This can be performed by skipping a specified number of array elements or leaping forward a fixed number of steps in each iteration.

Consider a sorted array A[] of size n, with indexing ranging from 0 to n-1, and a missing element x within the array A[]. This approach also necessitates the use of an p-sized block, which can be as follows:

- Iteration 1: if (x==A[0]), then success and if (x > A[0]), then jump to the next block.
- Iteration 2: if (x==A[p]), then success and if (x > A[p]), then jump to the next block.
- Iteration 3: if (x==A[2p]), then success and if (x > A[2p]), then jump to the next block.

  In time at any point, if (x < A[kp]), then a linear search is executed from index A[(k-1)p] to A[kp].

Figure 13 depicts the Jump Search technique:

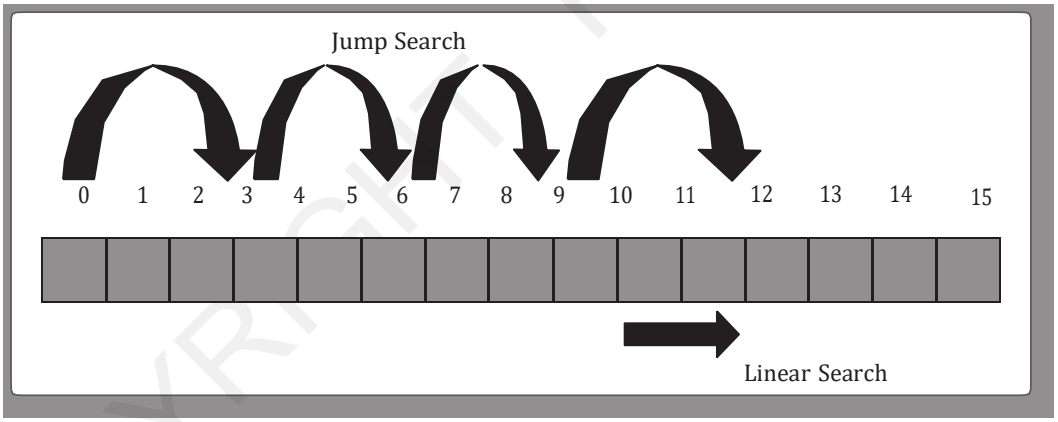


Figure 13: Jump Search Technique

Algorithm of Jump Search

The following steps describe the algorithm of jump Search is as follows:

Step 1: Fix j=0 and p = √n.

Step 2: Relate A[j] with the item. If A[j] != item and A[j] < item, then jump to the next block. Also, follow the following points:

- Set j = p
- Increase p by √n

Step 3: Replicate the step 2 till p < n-1

Step 4: If A[j] > item, then move to the starting of the present block and do a linear search. Also, follow the following points:

- Set x = j
- Equate A[x] with item. If A[x]== item, then print x as the valid location else set x++
- Replicate Step 4.1 and 4.2 till x < p

Step 5: Exit

The following C++ program is used to implement Jump Search algorithm is as follows:

```
#include <bits/stdc++.h>
using namespace std;

int JumpSearch(int array[], int a, int b)
{
     int Step = sqrt(b);
     int previous = 0;
     while (array[min(Step, b)-1] < a)
     {
          previous = Step;
          Step += sqrt(b);
          if (previous >= b)
               return -1;
     }
     while (array[previous] < a)
     {
          previous++;
          if (previous == min(Step, b))
               return -1;
     }
     // If the element is found
     if (array[previous] == a)
          return previous;
     return -1;
}
int main()
{
     int array[] = { 0, 33, 11, 52, 33, 15, 68, 63, 91,
                     32, 50, 99, 14, 23, 47, 60 };
     int a = 52;
     int b = sizeof(array) / sizeof(array[0]);

     // Find the index of 'a'
     int index = JumpSearch(array, a, b);

     // Print the index where 'a' is located
     cout << "\nNumber " << a << " is at index " << index;
     return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/bItJRqfPru.o
Number 52 is at index 3
```

Some of the salient features of jump search are as follows:

- Implemented in ordered arrays.
- The ideal size of a block to be leaped is ($\sqrt{n}$). Hence time complexity of Jump Search O($\sqrt{n}$).
- Jump search is preferred in situations when the binary search is expensive. For instance in case, when the value to be found is minimal.

13.3.4 Interpolation Search

Interpolation search enhancement over Binary Search in cases, when the elements in an ordered array are consistently distributed. This search goes to multiple positions with respect to the search key.

To locate the relevant position, the formula being used is as follows:

//We need to retain larger value of pos.

// if the value to be found is nearer to ary[high],and

// lower value if nearer to ary[low]

pos = low + [(x-ary[low])*(high - low) / (ary[high]-ary[low])]

ary[] ==> Array in which we need to search the value

x ==> data item to be found

low ==> Initial index in ary[]

high ==> Final index in ary[]

The derivation of the formula for Position is as follows:

Consider a linearly distributed or array.

The General equation of line : y = m*x + c.

Y refers to the element in the array and x is respective index.

Substituting the values of low, high and x in the equation:

ary[high] = m*high+c---- (1)

ary[low] = m*low+c --- (2)

X = m*pos + c ------ (3)

M = (ary[high] – ary[low])/ (high – low)

Subtracting equation (2) from (3):

X – ary[low] = m * (pos – low)

low + (x – ary[low])/m = pos

Pos = low + (x – ary[low]) *(high – low)/(ary[high] – ary[low])

The following C++ program is used to show implementation of interpolation search technique is as follows:

```
#include<iostream>
using namespace std;
int interpolationSearch(int arr[], int Start, int end, int key) {
   int dist, valRange, indexRange, estimate;
   float fraction;
```

```
   while(Start <= end && key >= arr[Start] && key <= arr[end]) {
      dist = key - arr[Start];
      valRange = arr[end] - arr[Start];
      fraction = dist / valRange;
      indexRange = end - Start;
      estimate = Start + (fraction * indexRange);
      if(arr[estimate] == key)
         return estimate;
      if(arr[estimate] < key)
         Start = estimate +1;
      else
         end = estimate - 1;
   }
   return -1;
}
int main() {
   int x, searchKey, location;
   cout << "Enter number of items: ";
   cin >> x;
   int arr[x];
   cout << "Enter items: " << endl;
   for(int i = 0; i< x; i++) {
      cin >> arr[i];
   }
   cout << "Enter search key to search in the list: ";
   cin >> searchKey;
   if((location = interpolationSearch(arr, 0, x-1, searchKey)) >= 0)
      cout << "Item found at location: " << location << endl;
   else
      cout << "Item is not found in the list." << endl;
}
```

The output of given C++ code is as follows:

```
/tmp/bItJRqfPru.o
Enter number of items: 5
Enter items:
23 44 56 76 88
Enter search key to search in the list: 56
Item found at location: 2
```

13.4 LAB EXERCISE

13(a): Write a Program in C++ to implement Linear Search.

The following C++ program is used to show implementation of Linear Search algorithm is as follows:

```
#include<iostream>
using namespace std;
int main()
{
```

```
    int array[7], i, num, index;
    cout<<"Enter 7 Numbers: ";
    for(i=0; i<7; i++)
        cin>>array[i];
    cout<<"\nEnter a Number to Search: ";
    cin>>num;
    for(i=0; i<7; i++)
    {
        if(array[i]==num)
        {
            index = i;
            break;
        }
    }
    cout<<"\nFound at Index No."<<index;
    cout<<endl;
    return 0;
}
```

The output of given C++ code is as follows:

```
/tmp/bItJRqfPru.o
Enter 7 Numbers: 1 2 3 4 5 6 7
Enter a Number to Search: 4
Found at Index No.3
```

13(b): Write a Program in C++ to implement binary search technique.

The following C++ program is used to show the implementation of the Binary Search Technique is as follows:

```
#include <iostream>
using namespace std;
int BinarySearch(int[], int, int, int);
int main()
{
   int num[6] = {20, 52, 77, 30, 82, 108};
   int search_num, location=-1;
   cout<<"Enter the number that you want to search: ";
   cin>>search_num;
   location = BinarySearch(num, 0, 6, search_num);
   if(location != -1)
   {
     cout<<search_num<<" found in the array at the location: "<<location;
   }
   else
   {
      cout<<"Element not found";
   }
   return 0;
}
int BinarySearch(int a[], int First, int Last, int search_num)
```

```
{
   int Middle;
   if(Last >= First)
   {
      Middle = (First + Last)/2;
      // to check whether the element exit in the middle location or not
      if(a[Middle] == search_num)
      {
         return Middle+1;
      }
      else if(a[Middle] < search_num)
      {
         return BinarySearch(a,Middle+1,Last,search_num);
      }
      //Checking if the search element is exit in lower half
      else
      {
         return BinarySearch(a,First,Middle-1,search_num);
      }
   }
   return -1;
}
```

The output of given C++ code is as follows:

```
/tmp/bItJRqfPru.o
Enter the number that you want to search: 30
30 found in the array at the location: 4
```

13.5 CONCLUSION

- Searching is the process by which we try to find whether the required data exists in the given structure or not.
- Sorting is a task in which we arrange the given structure in a particular order: ascending order or descending order.
- The Jump search approach is a new approach for searching a sorted array for a single element.
- In insertion sort, an element is placed at its correct position by moving all the elements greater than this element to its right.
- In Radix sort, the numerical data is sorted in n passes, where n is the maximum number of digits in the numerical data.
- In Linear Search, each record of a file is searched, one at a time, until the desired data is found.
- Binary Search is a search that uses the divide-and-conquer approach.
- Jump Search is a searching algorithm applicable for ordered arrays.
- Interpolation Search enhancement over Binary Search in cases, when the elements in an ordered array are consistently distributed.

13.6 GLOSSARY

- Searching: the process by which we try to find whether the required data exists in the given structure or not
- Sorting: a task in which we arrange the given structure in a particular order, i.e., ascending order or descending order
- Jump search: an approach for searching a sorted array for a single element
- Insertion sort: An element is placed at its correct position by moving all the elements greater than this element to its right
- Radix sort: The numerical data is sorted in n passes, where n is the maximum number of digits in the numerical data is a radix sort
- Linear search: In this search, each record of a file is searched, one at a time, until the desired data is found
- Binary search: a search which uses the divide-and-conquer approach
- Jump search: a searching algorithm applicable for ordered arrays
- Interpolation search: an enhancement over Binary Search in cases, when the elements in an ordered array are consistently distributed

13.7 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. Outline the concept of sorting and searching.
2. Explain the implementation of insertion sort.
3. On the basis of the technique used for locating the required data in a structure, the searching process is divided into different types. Describe the term searching algorithm with its types.
4. Interpret the binary search in detail.
5. Define Radix Sort.

13.8 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. Searching and sorting are the two most important tasks applied to data structures. Searching is the process by which we try to find whether the required data exists in the given structure or not. Refer to Section Introduction
2. In insertion sort, an element is placed at its correct position by moving all the elements greater than this element to its right. Refer to Section Implementation of Sorting Technique
3. The process of searching for a particular data item in a data structure consists of comparison and advancement operations. Refer to Section Searching Algorithms
4. Binary Search is a search that uses the divide-and-conquer approach. Refer to Section Searching Algorithms

5. In Radix sort, the numerical data is sorted in n passes, where n is the maximum number of digits in the numerical data. Refer to Section Implementation of Sorting Technique

13.9 POST-UNIT READING MATERIAL

- https://www.google.co.in/books/edition/Searching_Sorting_for_Coding_Interviews/UGI9DwAAQBAJ?hl=en&gbpv=1&dq=sorting+and+searching&printsec=frontcover
- https://www.quora.com/What-are-the-real-life-applications-of-searching-and-sorting-algorithm

13.10 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates about the concept of searching and sorting and try to explore the process of sorting and searching utilised in real world situations.

UNIT

14 Hashing

Names of Sub-Units

Hashing Table Organizations, Hashing: The Symbol Table, Hashing Functions, Static and Dynamic Hashing, Collision-Resolution Techniques

Overview

This unit begins by discussing about the concept of hashing, hashing table organization. Next the unit discusses the Hashing: the symbol table. Further the unit explains the hashing functions and static and dynamic hashing. Towards the end, the unit discusses the collision-resolution techniques.

Learning Objectives

In this unit, you will learn to:

- Explain the concept of hashing
- Describe the significance of hashing table organizations and hashing: the symbol table
- Discuss the hashing functions
- Assess the significance of static and dynamic hashing
- Explain the concept of collision-resolution techniques

Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of hashing
- Determine the significance of hashing table organizations and hashing: the symbol table
- Explore the concept of hashing functions
- Determine the significance of static and dynamic hashing
- Evaluate the importance of collision-resolution techniques

Pre-Unit Preparatory Material

- https://www.kdkce.edu.in/pdf/YDC-4IT-ADS-Hashing%20Techniques.pdf

14.1 INTRODUCTION

Hashing is a technique used for a quick retrieval of the desired data from a large volume of data. This scheme is used when a record is stored at a particular address and this address is to be computed by applying a formula: hash function on the key, the primary key of the record. The hash function ( ) is used in the following manner:

```
a=h(k)
```

In this equation, a is the address computed at the time of the application of the hash function on the k key of the record. The hash function should be selected in such a way that it results in a unique address, every time it is used. However, it is not practically feasible because there are frequent chances of "collision" i.e. we may get the address of the record with the k1 key, where already a record with the k2 key is stored. These collided records are called synonyms and we apply certain collision resolution techniques to resolve the conflict.

The domain of the hashing function is the interval of the key. If the keys are of 3 digits, we say the domain of the hash function is (0,999). If the keys are of 5 digits, we say that domain of the hash function is (0, 99999).

The range of the hash function is the capacity of the storage, where the records are stored. If the array where the records are stored consists of 1000 addresses, we say that the range of the hash function is (0,999). The hash function should be chosen so that it distributes the keys uniformly over the range of the storage. If the total capacity of the storage is n, then the good function should distribute the keys over the range (0, n-1).

14.2 HASH TABLE ORGANIZATION

Hash Table or Hash Map is a two-dimensional structure where the data (associated with some key) is mapped or hashed to some value. Hash function is used to determine the value of a key. In addition, it is used to transform the key into the address or the slot (bucket address) where the corresponding value is to be sought.

A hash table is a data structure that stores data in multiple places at the same time. The information is kept in an array with unique indexes for each entry. Data retrieval can be quick after we understand the index values of the various data fields. As the size of the data grows larger, the search and insertion operations in data structures become exceedingly fast. Hash tables keep data in arrays and utilise the hash technique to create an index that may be used to find or insert elements.

14.2.1 Linear Probing

As you know, we use a hashing technique to index an array that is already in use. You can use this to find subsequent free addresses in the array by navigating to subsequent slots until you search for a free slot. This process is called linear probing.

Some of the fundamental operations of linear probing are as follows:

- Search: Find a data entry in a hash table
- Insert: Insert data item into hash table
- Delete: Hash table data item delete item

14.2.2 Hashing Method

There are several methods of hashing and each method has its own way of computing the hashing address. The method which uniformly distributes the hashing addresses and leads to minimum collisions is highly preferred. Following are the different methods of hashing:

- Division method
- Mid square method
- Folding method

Division Method

This method is also called the divide and remainder method. Here, the key is divided by any number n and the remainder is taken to be the address.

```
Hence, the hash function is: h(k)=k mod n
```

This method produces the addresses ranging from 0 to n-1. The value of n should be chosen carefully. If it is taken as the power of 10, say 100, then all the keys having identical last two digits certainly hash into the same address. If n is chosen as an even integer, then all the even keys hash into odd addresses. A good choice of n is any number, which is not divisible by 2, 3, 5, 10 or which is a prime number.

Mid Square Method

In this method, the key value is squared and a few digits are extracted from the middle of the squared value. For example, if the hash address to be selected is of 3 digits and the key is 1234, then the square of 1234 is 1522756 and the middle value of 3 digits can be chosen as 227. This method produces the addresses that are uniformly distributed over the range of the hashing function and if the keys are large enough, then a selected part of the key can be squared.

Folding Method

Suppose the hash address that needs to be generated is of d digits. Now, when we use this method, the key is first divided into the groups of d digits, starting from the right. Then, these groups are added to compute their sum. The last d digit of the sum is considered as the hash address. For example, suppose the key is 12345678 and the hash address desired is of 3 digits. Then, from the right, the key 12345678 is divided into the groups of 3 digits, beginning from the right. Thereafter, the three groups of digits are added to make the sum as 1035, as shown in the following example:

```
12/345/678
Sum of 12+345+678 is 1035
```

The last three digits of the sum are considered as hash address i.e. 035 or 35 as are the hash address. Hence, this method is considered to be flexible and can be modified as per our need.

14.2.3 Search Functionality

To find an element, we must compute the hash code of the key supplied and then store the element in the array using the hash code as an index. If the element cannot be recognised using the computed hash code, we can use linear probing to find it.

14.2.4 Insert Operation

To insert an element, we must first compute the hash code of the key provided, and then use the hash code as an index to position the element in the array. Using linear probing for vacant regions, we can discover the resulting hash code.

14.2.5 Delete Operation

In order to delete an element, we need to compute ciphered hash of key value entered and place it through ciphered hash as array index. By using linear probing, we can obtain the data item if it is not identified by calculated ciphered hash. If it is located, reserve a counter fort value for managing execution of hash table.

14.3 HASHING: THE SYMBOL TABLE

A substantial information store generated and handled by a compiler is referred to as a symbol table. It stores information on the binding and scope of names and other items such as function names, classes, variables, and objects.

During the syntactic and lexical analysis phases, a symbol table can be created.

Compilers take data during the analysis phase and generate code during the synthesis phase. It is used to achieve time efficiency and is used in numerous phases of the compiler, as described below:

- Lexical analysis: It creates a symbol table from the most recent table records.
- Syntax analysis: Add details about the scope, attitude type, line of reference, and dimension in the table.
- Semantic analysis: Search semantics using the information in the table.
- Intermediate code generation: Uses a symbol table to determine how and what type of run time is allotted, as well as to add temporary variable data.

- Code optimization: For machine-dependent optimization, use the information in the symbol table.
- Target code generation: Uses the identifier's address information from the table to generate code.

14.3.1 Implementation of Symbol Table

Generally, two tables are managed in hashing scheme. Symbol table and hash table are the significant technique to deploy symbol tables.

Hash table refers to the array having index spanning from 0 to Size Of Table-1. There cords directs to the identities of symbol table. For looking up identities (names), one must utilize hash procedure which has been resulted in integer between 0 to tablesize-1. Lookup and insertion rapid-O(1). Quick search can be applied but hashing becomes complex to deploy.

14.4 HASHING FUNCTIONS

In cryptography, hash functions are the most often employed mathematical functions for establishing security. A hash function converts an arbitrary-size input value to a fixed-size value. As a result, the input can be any length, but the output is always the same length. Hash values or hashes are the outputs generated.

Another important thing to remember is that Hash Functions and Cryptography are independent. Encryption is a two-way function, which means that encrypted data can only be decrypted via a private key, making it reversible. Hashing is a one-way function that is also known as hash functions (i.e., hashes cannot be reversed). Hashing, as a result, outperforms cryptography.

Password verification is the most common application of hashing. When the user enters the password, the hash is created and compared to the hash in the database. The user can log in if the hashes are the same; otherwise, the user must re-enter the password.

The following are some of the most commonly used hashing functions:

- MD: It is referred to as Message Digest. MD2, MD4, MD5, and MD6 are all possibilities. MD is a Hash function with a 128-bit value.
- (SHA): It stands for Secure Hash Algorithm. It can be one of the following: SHA-0, SHA-1, SHA-2, or SHA-3. The SHA-2 family includes versions such as SHA-224, SHA-256, SHA-384, and SHA-512.
- RIPEMD: It stands for Race Integrity Primitives Evaluation Message Digest is the acronym for RACE Integrity Primitives Evaluation Message Digest. Many people utilise RIPEMD, RIPEMD-128, and RIPEMD-160. This method is also available in 256 and 320-bit versions.
- Whirlpool: It is a modified variant of AES that uses a 512-bit hash function. Whirlpool comes in three different versions: WHIRLPOOL-0, WHIRLPOOL-T, and WHIRLPOOL.

14.4.1 Properties of Hash Functions

To be efficient against numerous attacks from attackers, an ideal hash function should have the following features. They are as follows:

- Resistance to Pre-Image
  - The hash algorithm could not be reversed due to pre-image resistance.

- In other words, if any hash function "a" returns a hash value "c," finding an input value "b" that hashes to "c" should be extremely difficult.
- An attacker using a hash value seeking to find the input will be unable to do so because of this property.

- Resistance to second pre –image
  - Pre-image second to resistance, it should be extremely difficult to discover a different input that generates the same hash value for any input and its hash value.
  - To put it another way, if any hash function given an input "a" returns the hash value h(a), it should be difficult to identify any other input value "b" for which h(b) = h(a) (a).
- Resistance to Collision
  - Collision resistance implies that finding two different inputs of any length that create the same hash should be extremely difficult. Collision-free hash function is another name for this property. This attribute protects against the well-known hash collision attack.
  - Simply put, finding any two inputs x and y for a given hash function h is extremely difficult, hence h(x) = h (y).
  - This collision-free property ensures that collisions for a given hash function should be difficult to find.
  - This attribute also makes it difficult for an attacker to locate two input values that produce the same hash.

Some of the application of hash function is commonly used in the following fields:

- Authentication using Cryptocurrency Password Verification
- Check for data and file integrity
- Signature on a computer

14.5 STATIC AND DYNAMIC HASHING

Static hashing is a hashing technique that allows users to do lookups on a finished dictionary set (all objects in the dictionary are final and not changing). Dynamic hashing, on the other hand, is a hashing technique that creates and removes data buckets on demand.

14.5.1 Static Hashing

Static hashing is a method of shortening a string of letters in computer programming in which the set of shorter characters remains the same length to make data access quicker.

Address of output data bucket remains identical in static hashing. If you create address for stud_Id=13 through hashing procedure modulo(5), the output address is 3. It doesn't modify the bucket location. The count of information buckets remains same if static hashing is used.

Static hashing function is classified into two methods are as follows:

- Open hashing: Rather than replacing the former one, the subsequent data cluster focused on entering new record in the open hashing method. It is also called linear probing. For instance, A2 be latest record that we need to add. The hash procedure creates address 222. It can be allocated at an

alternative slot. The program focused on subsequent data bucket 501 followed by allocation of A2 to the bucket, as shown in Figure 1:

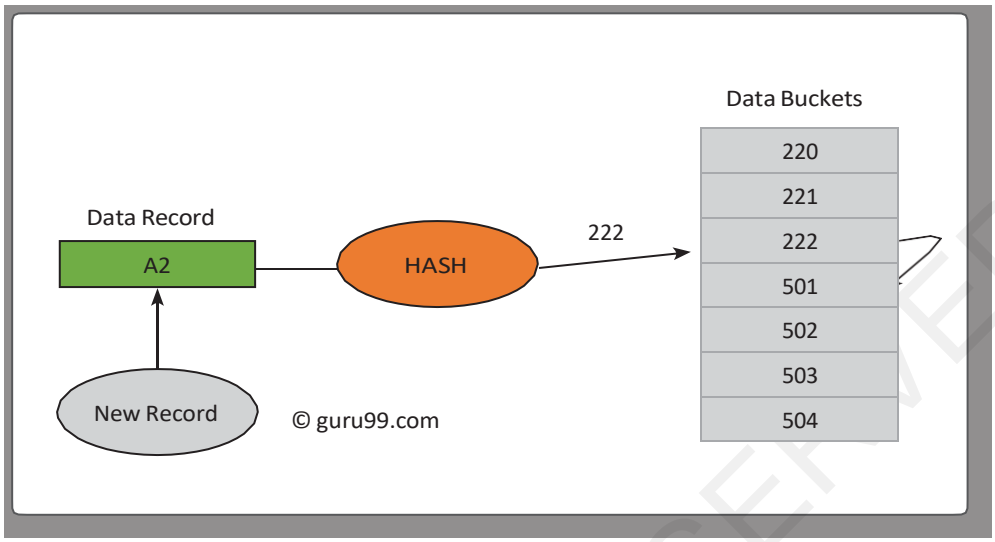


Figure 1: Working of Open Hashing

- Close hashing: Here, if we have occupied buckets, vacant bucket is given to identical hash and results will be associated after previous one.

Some of the commonly used static hash functions are as follows:

- Insert: This command adds a new record to the hash table. The hash key will be used to construct an address for that record.
- Delete: This action fetches the record to be destroyed before deleting the record's address from memory.
- Update: The hash function first locates the record before updating it with new data.
- Query: This method, also known as a search, uses the hash function to find entries that match specified criteria.

14.5.2 Dynamic Hashing

Dynamic hashing is a method of hashing, or decreasing a string of characters, in which the set of shorter characters grows, shrinks, and reorganises to fit the way data is retrieved in computer programming. All things listed in an object dictionary become dynamic and may change when dynamic hashing is employed.

This facilitates the process of removing and adding data buckets on the fly. The hash technique aids in the development of a huge number of elements. Static hashing has the drawback of shrinking dynamically as the database expands or shrinks. Extended hashing is another name for it. The hash function produces a larger number of values.

Some of the commonly used dynamic hash functions are as follows:

- Insertion: Calculates the bucket's address. If the bucket is already full, new buckets can be added. Furthermore, the hash function can be re-computed by adding additional bits to the hash value. It is possible to add data to buckets that are not yet full.

- Querying: Examines the hash index's depth value and uses those bits to calculate the bucket address.
- Update: This command run a query and updates the data.
- Delete: Executes a query to locate the data to be deleted.

14.6 COLLISION-RESOLUTION TECHNIQUES

Collision is said to occur when the hashing function produces the hash address (for a key), which is already used by another key i.e. a key that already exists on that hash address. Such a situation is not desirable and several methods are used to resolve collisions in this situation. There are two types of collision resolution methods that resolve the problem which occurs on getting the same addresses (synonyms) on the application of hash function on two different keys. These collision resolution methods are:

- Open addressing
- Chaining

14.6.1 Open Addressing

When the collision occurs, the synonym is placed at some other adjacent empty location in the hash table. That is, no location outside the hash table is allocated for storing the key values. For example, consider our hash table, which has the hash addresses in the range 0 to 100 and in this hash table, we want to store the records of a few students. On applying the hash function on some roll number, say 312456, we get the hash address as 36. However, we find that the location is already occupied by a record of another student with roll number, say 302446 (so the situation of collision has occurred); and we cannot store our record at this hash address. So, we look for an adjacent hash address in the Open Addressing scheme to see if it is vacant to store the record of the student with the roll number 312456. This implies that the hash address 37 is searched and if it is found vacant, the record is stored at this location; and if it is also occupied with some student record, the next adjacent location (i.e. hash address 38) is searched and so on. The Open Addressing method can be further divided into the following methods:

- Linear probing method: The simplest method to resolve a collision is to start with the hash address (the location where the collision occurred) and do a sequential search for an empty location. Hence, this method searches in a straight line, and it is therefore called linear probing. While applying this method, the array should be considered as circular so that when the last location is reached, the search proceeds to the first location of the array.

  The major drawback of linear probing is that, as the table becomes about half full, there is a tendency towards clustering; that is, the records start to appear at continuous positions with small gaps between them. Therefore, the sequential search needed to find an empty position, becomes longer and longer. Hence, the performance of the hash table starts to degenerate.

  The reason for clustering is the simple fact that on finding a collision, the increment added to the hash address, to find the next vacant location, in the case of linear probing is '1'. To avoid this problem of clustering, we should have different increment values every time to find the next empty location. The solution is a method called rehashing ()where we use a second hash function. This function generates an increment value, which is added to the hash address generated by the first hash function (which generated the collision). This second hash address location is checked and if it is vacant, the record is stored here; else again the second hash function is applied to generate some other increment value until we get an empty location.

- Quadratic probing method: If there is a collision at the hash address h, then this method probes the table at locations h+1, h+4, h+9,....., i.e., at locations h+i2 (%HASHSIZE) for i=1, 2, ..., that is, the increment function is i2. This method reduces clustering and if the HASHSIZE is a power of 2, then relatively few positions are probed.

14.6.2 Chaining

Here, the synonym is linked with the help of pointers i.e. extra space outside the hash table is allocated and connected in the form of a linked list.

Chaining method is used for the linked storage and hence, in this method, each slot of the hash table has a pointer to the linked list and all the elements hashed to a slot are placed in the linked list attached to that slot. For example, we can see in Figure 2 that all the elements hashed to slot 3 are placed in the linked list attached to that slot. Similarly, the element hashed to slot 0 is linked to that slot with the help of a pointer:

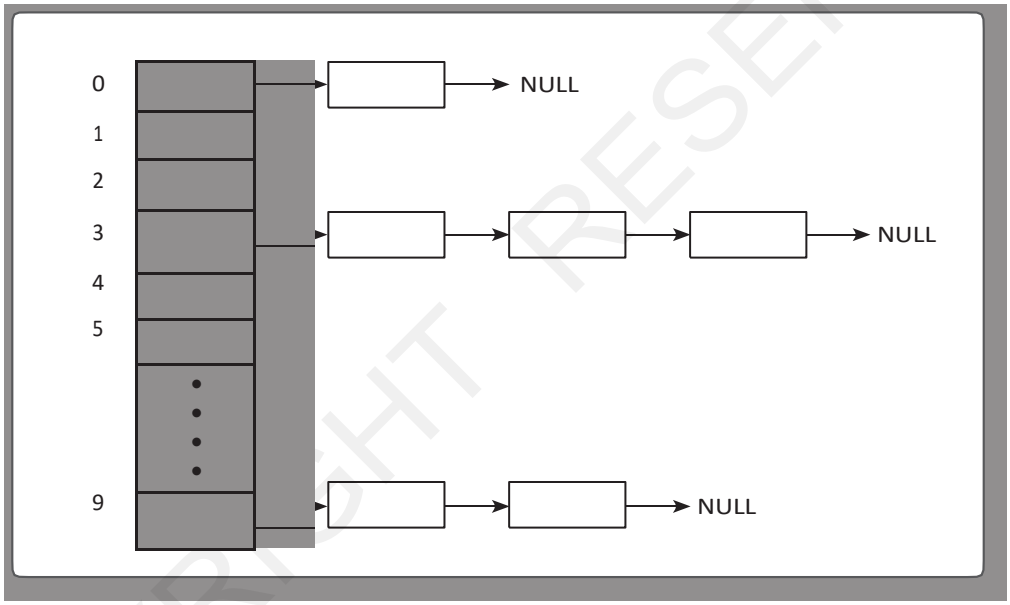


Figure 2: Hash Table with pointers to the linked list

Linked storage has the following advantages:

- Space saving: When the hash table is maintained as a contiguous array, enough space has to be set aside at the compilation time to avoid overflow. If the records themselves are in the hash table, the empty positions (if there are many such positions) consume considerable space that might be needed elsewhere. If, on the other hand, the hash table contains only pointers to the records, i.e. the pointers that require only one word each, then the size of the hash table may be reduced to a large extent.
- Collision resolution: It allows a simple and efficient way of collision handling. In this resolution, we need to add only a link field to each record. Clustering is no problem at all, because the keys with distinct hash addresses always go to distinct linked lists attached to distinct slots. In other words, there are 10 linked lists and one linked list is attached to each slot of the hash table. The elements are hashed to their respective slots and hence linked (in the form of a node) to the linked list attached to that slot.

- Overflow: A third advantage is that it is no longer necessary that the size of the hash table exceeds the number of records. If there are more records than the entries in the table, then it means that only some of the linked lists are now sure to contain more than one record. Even if the size of the records is several times more than the size of the table, then the average length of the linked list remains small and the sequential search on the appropriate list remains efficient.
- Deletion: Deletion becomes a quick and easy task in a chained hash table. For example, deletion of a node from a linked list just requires the adjustment of address pointers.

The disadvantage of linked storage, however, is as follows:

- The use of space: All the links require space. If the record is large, then this space is negligible in comparison with the space needed for the records themselves; but if the records are small, then it is not so. For instance, if we use the chaining method and make the hash table quite small with the number of entries (n entries) equal to the number of items, then we use 3n words of storage altogether: n for the hash table, n for the keys, and n for the links to find the next node on each chain. Now, since the hash table is nearly full, the result is many collisions and several items in some of the chains. Hence, searching gets a bit slow. On the other hand, suppose we use open addressing, then putting the same 3n words of storage entirely into the hash table means that it is only one-third full; and therefore, there are relatively few collisions and the search for any given item gets faster.

14.7 CONCLUSION

- Hashing is a technique used for a quick retrieval of the desired data from a large volume of data.
- Hash Table or Hash Map is a two-dimensional structure where the data (associated with some key) is mapped or hashed to some value.
- A substantial information store generated and handled by a compiler is referred to as a symbol table.
- A hash function converts an arbitrary-size input value to a fixed-size value.
- Static hashing is a hashing technique that allows users to do lookups on a finished dictionary set (all objects in the dictionary are final and not changing).
- Dynamic hashing is a hashing technique that creates and removes data buckets on demand.
- In open hashing the subsequent data cluster focused on entering new record in the open hashing method.
- If we have occupied buckets, vacant bucket is given to identical hash and results will be associated after previous one in close hashing.
- Chaining method is used for the linked storage.

14.8 GLOSSARY

- Hashing: This technique used for a quick retrieval of the desired data from a large volume of data
- Hash table: It is also known as hash map and it is a two-dimensional structure where the data is mapped or hashed to some value
- Symbol table: A substantial information store generated and handled by a compiler is referred to as a symbol table

- Hash function: It converts an arbitrary-size input value to a fixed-size value
- Static hashing: It is a hashing technique that allows users to do lookups on a finished dictionary set (all objects in the dictionary are final and not changing)
- Dynamic hashing: This hashing technique that creates and removes data buckets on demand
- Opening hashing: In open hashing the subsequent data cluster focused on entering new record in the open hashing method
- Close hashing: If we have occupied buckets, vacant bucket is given to identical hash and results will be associated after previous one
- Chaining: This method is used for the linked storage

14.9 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. Explain the concept of hashing.
2. During the syntactic and lexical analysis phases, a symbol table can be created. Discuss
3. Hashing is a one-way function that is also known as hash functions. Describe the significance of hashing functions.
4. Explain the concept of static and dynamic hashing.
5. When the collision occurs, the synonym is placed at some other adjacent empty location in the hash table. Discuss

14.10 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. Hashing is a technique used for a quick retrieval of the desired data from a large volume of data. Refer to Section Introduction
2. A substantial information store generated and handled by a compiler is referred to as a symbol table. Refer to Section Hashing: The Symbol Table
3. In cryptography, hash functions are the most often employed mathematical functions for establishing security. Refer to Section Hashing Functions
4. Static hashing is a hashing technique that allows users to do lookups on a finished dictionary set (all objects in the dictionary are final and not changing). Dynamic hashing, on the other hand, is a hashing technique that creates and removes data buckets on demand. Refer to Section Static and Dynamic Hashing
5. Collision is said to occur when the hashing function produces the hash address (for a key), which is already used by another key i.e. a key that already exists on that hash address. Refer to Section Collision Resolution Techniques

14.11 POST-UNIT READING MATERIAL

- https://levelup.gitconnected.com/the-3-applications-of-hash-functions-fab1a75f4d3d
- https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1208/lectures/hashing/Lecture27_Slides.pdf

14.12 TOPICS FOR DISCUSSION FORUMS

- Discuss with your friends and classmates about the concept of hashing and their applications. Also, discuss about the real world examples of hashing.

UNIT

15

Files and Files Organisation

Names of Sub-Units

Introduction, Data Hierarchy, File Attributes, Text Files, Binary Files, Basic File Operations, File Organizations and Indexing

Overview

This unit begins by discussing about the concept of files and files organization and data hierarchy. Next, the unit discusses the files attributes, text files and binary files. Further the unit explains the basic file operations. Towards the end, the unit discusses the file organizations and indexing.

Learning Objectives

In this unit, you will learn to:

- Discuss the concept of files and files organization
- Explain the concept of data hierarchy and files attributes
- Describe the text files and binary files
- Explain the significance of basic file operations
- Discuss the importance of file organizations and indexing

Learning Outcomes

At the end of this unit, you would:

- Evaluate the concept of files and files organization
- Assess the concept of data hierarchy and files attributes
- Evaluate the importance of text files and binary files
- Determine the significance of basic file operations
- Assess the importance of file organizations and indexing

Pre-Unit Preparatory Material

- http://cre8te.co.uk/wp-content/uploads/2014/07/Files-And-Folders-Updated-June-2015.pdf

15.1 INTRODUCTION

File is a collection of record. It is allocated for storing large amount of information stored on devices excluding internal memory of the computer. It meant that records are stored in the secondary storage. The file should be organized so that operations will be determined effectively based on features of secondary storage devices to deploy the file. Operations on file are to insert and delete records, update or process records and search for records. These operations are applicable for lists, trees, arrays, list structures and complex lists. Time used in managing information has been resulted in fast operations and the efficiency should be determined against care needed to manage the organized data structure.

At the time of executing high level language programs, the operations manage the files as data. Operations are traversing and processing all records and also individuals chosen in random order. Primary function of file system is to offer storage facilities and enable files to be searched conveniently so that records may be sequentially retrieved.

15.2 DATA HIERARCHY

Data hierarchy is the systematic organisation of data, which is typically done in a hierarchical method. In data organisation, characters, fields, records, files, and so on are all used. This concept is a useful place to start when trying to understand what makes up data and whether it has a structure. How can someone decipher facts like 'employee,' 'name,' 'department,' 'Marcy Smith,' 'Sales Department,' and so on, assuming they're all connected? To help you understand these notions, think of them as smaller or larger components in a hierarchy. Marcy Smith could be seen as a Sales Department employee or an example of a Sales Department employee.

All data has its own hierarchy in data hierarchy, starting at a comprehensive top level and continuing down to a definite bottom level. Someone, for example, is looking for a video game title in a database.

The video game console type is first, followed by the game creator, the genre, the first letter of the game's name, and lastly the game itself. This method of cataloguing data makes it easier to locate. It also makes it easier for the database to process new data by ensuring that datum is only recorded in the appropriate category.

15.2.1 Components of Data Hierarchy

The following are the components of data hierarchy:

- Bits: In a computer, the smallest data item can have the value 0 or 1. A bit (short for "binary digit"—a digit that can take one of two values) is a type of data item. Simple bit manipulations are performed by computer circuitry, such as evaluating the value of a bit, altering the value of a bit, and reversing the value of a bit (from 1 to 0 or from 0 to 1).
- Characters: Working with data in the low-level form of bits is inconvenient for programmers. Instead, they prefer to operate with numbers (0–9), letters (A–Z and a–z), and special symbols (e.g., $, @, percent, &, *, (,), –, +, ",:,?, and /). Characters are made up of digits, letters, and special symbols. The character set of a computer is the collection of all the characters that can be used to develop programmes and represent data. Because computers can only handle 1s and 0s, the character set of a computer depicts each character as a sequence of 1s and 0s. Unicode characters in Java are made up of two bytes, each of which is eight bits long. The data type byte is available in Java and can be used to represent byte data.
- Fields: Fields are made up of characters or bytes. A field is a group of characters or bytes that transmit data. A person's name, for example, can be represented by a field of uppercase and lowercase letters.
- Recordsandfiles: A record is acollection of interconnected fields. A file is acollection of interconnected records. Any type of data in any format can be stored in a file. In different operating systems, a file is viewed as a collection of bytes.
- Record keys: At least one field in each record is chosen as a record key in order to retrieve specific records from a file. A record key is a unique identifier for each record that identifies it as belonging to a specific person or entity.
- Sequential files: A file can be organised in a number of different ways. A sequential file is the most popular, as it stores records in the order defined by the record-key field.
- Database: A collection of related files is known as database and the collection of designed program to create and manage database is known as database management system (DBMS).

15.3 FILE ATTRIBUTES

A file can be a "free formed," "indexed" or "organised" collection of linked bytes that is only understood by the person who generated it. In other terms, a file is an item in a directory. Name, creator, date, type, permissions and other information may be present in the file.

A file is a data structure that contains a series of records in a logical order. Files are kept in a file system, which might be located on a drive or in main memory. Simple (plain text) or complicated files are both possible (specially-formatted). The term "directory" refers to a group of files.

The file system is a collection of directories organised at various levels as shown in Figure 1:

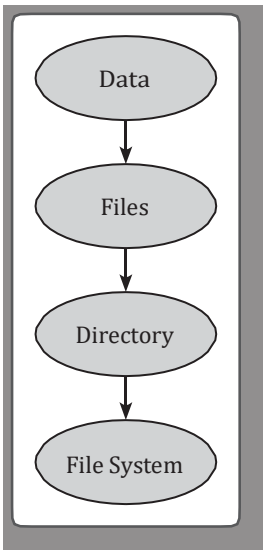


Figure 1: Levels in File System

Some of the attributes of file system are as follows:

- Name: Every file has a name that is used to identify it in the file system. Two files with the same name cannot exist in the same directory.
- Identifier: Each file has its own extension that specifies the file's kind in addition to its name. A text file, for example, has the extension.txt, whereas a video file has the extension.mp4.
- Type: Files are categorised into several sorts in a File System, such as video files, audio files, text files and executable files.
- Location: There are various areas in the File System where files can be stored. The location of each file is stored as an attribute.
- Size: One of the most essential characteristics of a file is its size. The number of bytes obtained by the file in memory is referred to as the file's size.
- Protection: Distinct safeguards for different files may be desired by the computer's administrator. As a result, each file has its unique set of rights for each User group.
- Time and date: Every file has a time stamp that includes the time and date when it was last changed.

15.3.1 Advanced File Attributes

The permissions you can provide to folders and files are determined by how they are accessed. Because they show in the advanced security settings dialogue box, these rights are referred to as "advanced" permissions. To access them, go to the security tab of the Properties dialogue box and select the advanced option.

The following is a list of advanced permissions for files and folders, along with a brief description of each are as follows:

- Traverse folder: This allows or disallows going through a restricted folder in the folder hierarchy to access files and folders beneath the restricted folder. Only when the group or user is not granted the "Bypass traverse checking user" right in the Group Policy snap-in does the traverse folder take effect. This permission does not automatically grant access to programme files that can be run.

- Execute file: This allows or disallows the execution of executable files.
- List folder: This allows or disallows viewing of the folder's file names and subfolder names. List Folder affects just the contents of the folder; it has no bearing on whether or not the folder you are setting the permission on will be listed.
- Read data: This allows or disallows viewing of data in files.
- Read attributes: This allows or disallows viewing of a file's or folder's properties, such as "read-only" and "hidden"
- Read extended attributes: This allows or denies viewing the extended attributes of a file or folder. Extended attributes are defined by programs and may vary by program.
- Create files: This allows or denies creating files within the folder
- Write data: This allows or denies making changes to a file and overwriting existing content.
- Create folders: This allows or denies creating subfolders within the folder.
- Append data: It may consent or reject while creating changes to the end of the file but does not change, deleting and overwriting when data is existing.
- Write attributes: It allows or denies changing the attributes of a file or folder, for example, "readonly" or "hidden".
- Write extended attributes: It allows or disallows modifying a file's or folder's extended attributes. Programs define extended characteristics, which might differ from one programme to the next. The Write Extended Attributes permission does not grant the ability to create or delete files or folders; rather, it grants the ability to modify the extended attributes of an existing file or folder.
- Delete subfolders and files: It allows or disallows the deletion of subfolders and files, even if the delete permission on the subfolder or file has not been given.
- Delete: It allows or prevents the deletion of a file or folder. Even if you do not have Remove permission on a file or folder, if you have Delete Subfolders and Files permission on the parent folder, you may still delete it.
- Read permissions: This allows or denies reading permissions of a file or folder.
- Change permissions: This allows or denies changing permissions of the file or folder.
- Take ownership: This allows or disallows the user to take ownership of a file or folder. Regardless of any current rights that protect the file or folder, the owner of the file or folder can always alter its permissions.
- Synchronise: This allows or disallows separate threads to synchronise with another thread that may signal the handle for the file or folder. Only multithreaded, multiprocessing programs are allowed to use this permission.

15.4 TEXT FILES

A text file is a form of digital file that is non-executable and only contains text. It can comprise numbers, characters, symbols, and/or a combination of these, but not special formatting such as italic text, bold text, underlined text, graphics, and so on. The.txt file extension is used to identify text files on a Microsoft Windows computer. A given image is an example of a text file.

A text file, often known as an ASCII file or a flat file, is a type of file that is used to contain structured and standard textual data or information that humans can read. The text file can be defined in a number of

different formats, including ANSI for Windows-based operating systems and ASCII for cross-platform use.

In a Windows OS, a text editor such as Word or Notepad is used to create a text file with the extension.txt (operating system). Nearly all computer languages, including PHP and Java, employ text files to write and store source code. By changing the file extension from.txt to.php or.cpp, the generated file can be converted into a similar programming language.

15.5 BINARY FILES

A binary file includes all files that aren't used to store textual material. A binary file can be used to construct any custom file type as long as the essential information for reading the file is stored in the file. Multiple types of data, such as images, video, and audio, are stored in the same file. The only stipulation they make is that you have an application that can read this type of data from the system. The PNG file format is an excellent example of the above-mentioned scenario. Most image viewers can read PNG files, which contain graphical data. When you open a PNG file in a text editor, you'll notice that the majority of the file is made up of unrecognisable characters. However, there are readable text fragments strewn over the file. This is due to the fact that the PNG file includes small portions for storing textual data in addition to the graphical data. Other file formats may also enable this, which is possible due to binary files' dynamic nature.

A header appears at the top of binary files. The file's key is this header. It's used to keep track of the data that identifies the file's content. An example of binary file is shown in Figure 2:

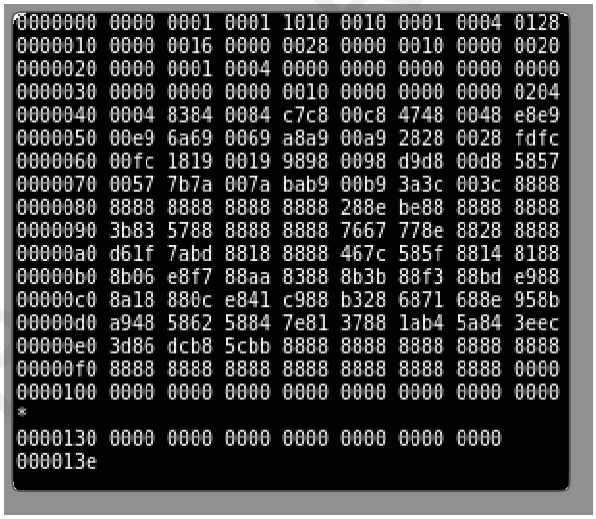


Figure 2: Binary File

The first column has starting address of line when * represent repetition. Binary files has sequence of bytes that represents the binary digits in eights. Binary files has bytes which are interpreted as symbol or other characters. Example: compile computer programs which are sometimes referred as binaries. These binary files contain sounds, images, compressed version of other files and any type of file content. In computer program, binary files composed of blocks of metadata and headers to interpret data in file. Header has magic numbers or signatures that will determine the format. GIF file has multiple images and headers are used to define block of image data. If binary file do not have headers, it can be called as flat binary file.

In order to binary files over some systems which do not enable all data values , they will be converted into plain text. Encoding the data has demerits of enhancing the file size at the time of transfer and need

translation into binary. An enhanced size will be determined by low level link compression and text data have less entropy as it has enhanced size.

The standard libraries and Microsoft windows enables the programmer to determine parameter if file is focused on binary or plain text while opening a file. In Unix, the standard libraries enables the programmer to determine whether a file is expected to be binary or text.

Viewing

Hex viewer is used to view file data as sequence of hexadecimal values of binary file. If the binary file is viewed in text editor, each group will be translated as a character and user shows textual characters. If the file is opened in other applications, then it has own use for each byte. The application considers each byte as output stream of numbers between 0 and 255. It replaces the unprintable characters with spaces indicating human readable text. It can be helpful for monitoring binary file to identify password in games and hidden text and retain corrupted document. It can be used to explore the suspicious files for unwanted effects. If the file is considered as run and executable, the operating system will interpret the file as sequence of instructions in machine language.

Interpretation

Standards are significant to the binary files. ASCII character will be displayed in text. Byte may be pixel or sound or entire word. Binary is meaningless until the executed algorithm describes what needs to be done with each byte, word or bit. Evaluating the binary to map against the known formats will cause wrong conclusion. It can be used in steganography where the binary file exhibits the hidden content.

15.6 BASIC FILE OPERATIONS

A file is a collection of logically related data stored on secondary storage in the form of a sequence of operations. The creator of the file determines the content of the file. The many actions that can be performed on a file, such as read, write, open, and close, are referred to as file operations. The user performs these tasks with the assistance of the operating system's commands. Some examples of common operations are as follows:

- Create operation: This action is performed to create a file in the file system. On the file system, it is the most widely used operation. The linked application programme uses the file system to create a new file of a certain type. This file system allocates space to the file. This new file gets placed in the correct directory because the file system recognises the directory structure's format.
- Open operation: This is the most typical operation that is done on the file. Before executing any file processing actions, the file must first be opened. When a user wants to open a file, he or she specifies a file name that will open that file in the file system. It instructs the operating system to use the open system function and provides the file system with the file name.
- Write operation: Using this method, the information is written into a file. A system call write specifies the name of the file and the length of data to be written to it. After the last byte is written, the file length is increased by a certain amount and the file pointer is shifted.
- Read operation: The contents of a file are read with this operation. The OS keeps a Read pointer those points to the spot where the data has been read up to.
- Re-position or seek operation: The seek system call advances the file pointers forward or backward in the file, depending on the user's demands. This technique is typically done out with the help of file management systems that provide direct access to files.

- Delete operation: Not only will removing the file delete all of the data it contains, but it will also free up disc space. To delete the selected file, the directory is searched. Once the directory entry is found, all associated file space and the directory entry are released.
- Close operation: When the file has been processed, it should be closed so that all of the changes are permanent and all of the resources used are released. When you close the file, it deallocates all of the internal descriptors that were created when you opened it.

15.7 FILE ORGANIZATION AND INDEXING

A file organization guarantees that records are prepared for processing. It's used to discover out how to efficiently organise each base relation's files.

For example, let's say we wish to sort employee information alphabetically by name. Sorting files by employee name is a good way to organise them. A file organised by employee name, on the other hand, isn't the ideal way to find all employees having grades in a certain range.

The file organization can be classified into three types:

- Sequential access file organization
- Direct access file organization
- Indexed sequential access file organization

15.7.1 Indexing

Indexing is a data structure technique that helps to speed up data retrieval. As we can quickly locate and access the data in the database, it is a must-know data structure that will be needed for database optimizing. Indexing minimizes the number of disk accesses required when a query is processed. Indexes are created as a combination of the two columns.

Data retrieval is aided by indexing, which is a data structure approach. Because it lets us to quickly identify and access data in the database, it is a must-know data structure for database optimization. Indexing lowers the number of disc accesses necessary when a query is run. The two columns are mixed together in indexes:

- First column: The Search key is in the first column. It has a copy of the table's primary key or candidate key. This column's values can be sorted or not. However, if the values are sorted, the related data is easily accessible.
- Second column: The Data reference or Pointer is the second column. It contains the disc block address where the relevant key value can be found. Figure 3 depicts the structure of index:

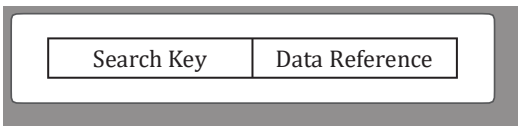


Figure 3: Structure of Index

15.7.2 Types of Indexing

Indexing is classified into four types are as follows:

- Primary indexing
- Secondary indexing

- Clustered indexing
- Multilevel indexing

Primary Indexing

There are only two columns in primary indexing. The main key values, which are the search keys, are in the first column. The pointers in the second column contain the address to the search key value's matching data block. The table should be sorted, and the records in the index file and the data blocks should have a one-to-one relationship. This is a slower but more traditional mechanism. Primary indexing is further classified into two types are as follows:

- Dense index: For each search key value in the data file, there is an index record that contains a search key and a pointer. Despite the fact that the dense index is a quick solution, it requires more memory to store index records for each key value. Figure 4 depicts dense index:

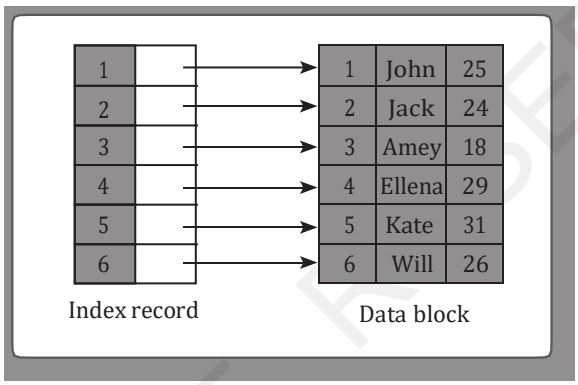


Figure 4: Dense Index

- Sparse index: There are only a few index records that point to the search key value. First, the index record starts searching sequentially by pointing to a location of a value in the data file until it finds the actual location of the search key value. Though sparse indexing is time-consuming, it requires less memory to store index records as it has less of them. Figure 5 depicts sparse index:

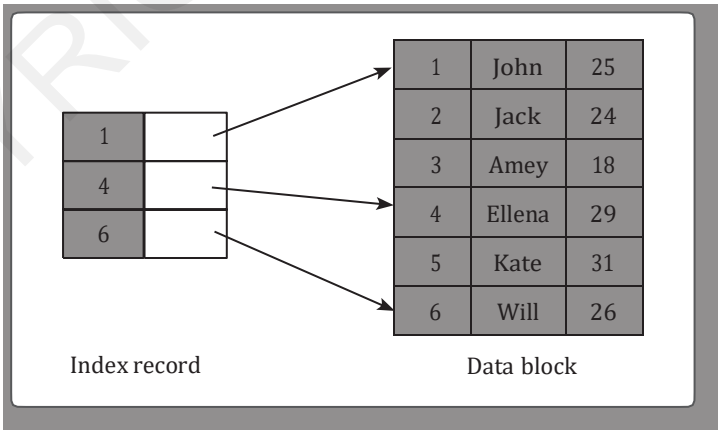


Figure 5: Sparse Index

Secondary Indexing (Non-Clustered Indexing)

In the secondary indexing the columns of the candidate key hold the values with the respective pointer that has the values to the location of an address.

An intermediate node is a communication medium between index and data, as shown in Figure 6:

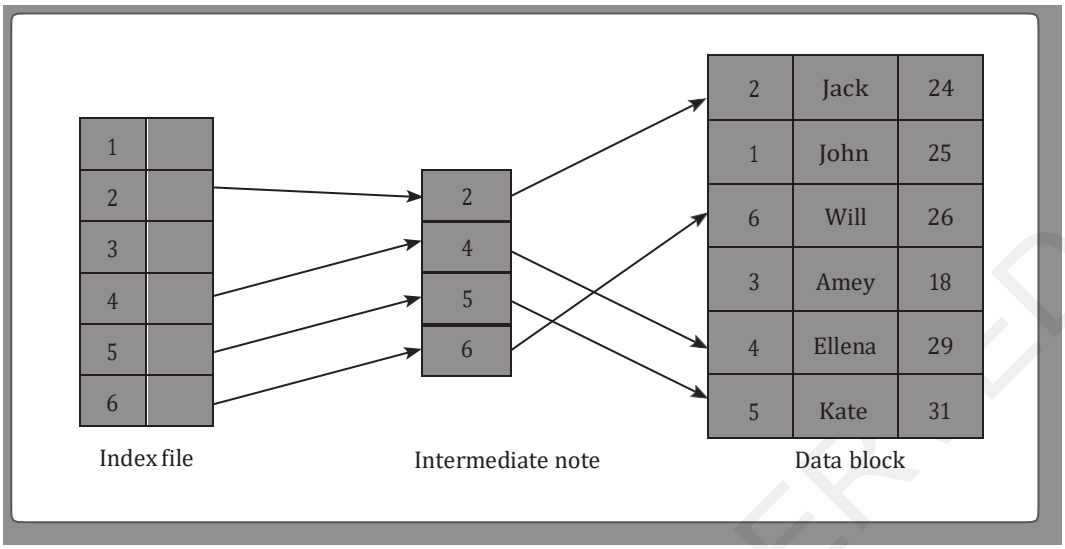


Figure 6: Secondary Indexing

Clustered Indexing

In clustered indexing the table is well-organized. When the indexes are created with the help of non-primary key at that time, to get the unique values we associate more than two columns together to identify data uniquely to create the index, as shown in Figure 7:

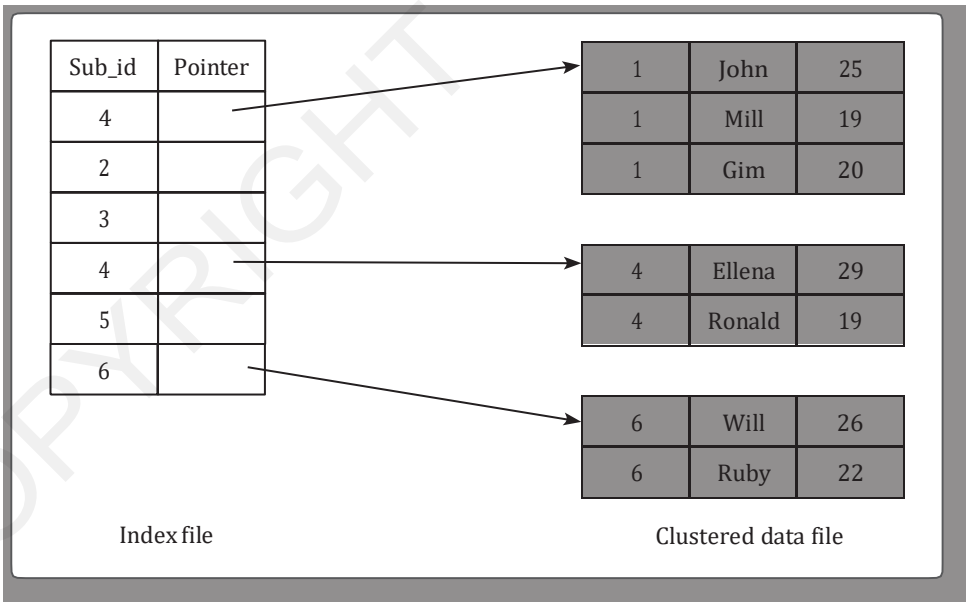


Figure 7: Clustered Indexing

Multilevel Indexing

Multilevel indexing is used when the primary index does not fit in the memory. The indices are increased when the size of the database is increased. In fact only a single-level index can be too huge to accumulate in the main memory. The data block gets breaken down into the smaller blocks to be stored in the main memory in multilevel indexing.

The multilevel indexing is further classified into two methods:

- B+ tree indexing
- B- tree indexing

15.8 CONCLUSION

- Data hierarchy is the systematic organisation of data, which is typically done in a hierarchical method.
- A file can be a "free formed," "indexed" or "organised" collection of linked bytes that is only understood by the person who generated it.
- A text file is a form of digital file that is non-executable and only contains text.
- A file organization guarantees that records are prepared for processing.
- Indexing is a data structure technique that helps to speed up data retrieval.
- In the secondary indexing the columns of the candidate key hold the values with the respective pointer that has the values to the location of an address.
- There are only two columns in primary indexing.
- In clustered indexing the table is well-organized.
- Multilevel indexing is used when the primary index does not fit in the memory.

15.9 GLOSSARY

- Data hierarchy: The systematic organisation of data, which is typically done in a hierarchical method.
- File: It can be a "free formed," "indexed" or "organised" collection of linked bytes that is only understood by the person who generated it.
- Text file: It is a form of digital file that is non-executable and only contains text.
- File organization: The guarantees that records are prepared for processing in file organization.
- Indexing: It is a data structure technique that helps to speed up data retrieval.
- Secondary indexing: The columns of the candidate key hold the values with the respective pointer that has the values to the location of an address.
- Clustered indexing: The table is well-organized in clustered indexing.
- Multilevel indexing: It is used when the primary index does not fit in the memory.

15.10 SELF-ASSESSMENT QUESTIONS

A. Essay Type Questions

1. Describe the concept of file.
2. All data has its own hierarchy in data hierarchy. Discuss
3. A text file, often known as an ASCII file or a flat file. Explain the significance of text file.
4. Describe the significance of binary file.

5. Indexing minimizes the number of disk accesses required when a query is processed. What is the concept of indexing?

15.11 ANSWERS AND HINTS FOR SELF-ASSESSMENT QUESTIONS

A. Hints for Essay Type Questions

1. File is a collection of record. It is allocated for storing large amount of information stored on devices excluding internal memory of the computer. Refer to Section Introduction
2. Data hierarchy is the systematic organisation of data, which is typically done in a hierarchical method. Refer to Section Data Hierarchy
3. A text file is a form of digital file that is non-executable and only contains text. It can comprise numbers, characters, symbols, and/or a combination of these, but not special formatting such as italic text, bold text, underlined text, graphics, and so on. Refer to Section Text Files
4. A binary file includes all files that aren't used to store textual material. A binary file can be used to construct any custom file type as long as the essential information for reading the file is stored in the file. Refer to Section Binary Files
5. Indexing is a data structure technique that helps to speed up data retrieval. As we can quickly locate and access the data in the database, it is a must-know data structure that will be needed for database optimizing. Refer to Section File Organization and Indexing

15.12 POST-UNIT READING MATERIAL

- https://limbd.org/objectives-factors-to-be-consider-of-file-organization/
- https://www.indeed.com/career-advice/career-development/types-of-files

15.13 TOPICS FOR DISCUSSION FORUMS

- You can discuss about the concept of files and files organization with your friends. Also, discuss about the concept of data hierarchy in real life.