Chapter 1: Introduction to Data Structure

1. Review of C and C++

1.1. Arrays

- Arrays are most frequently used in programming.
- Mathematical problems like matrix, algebra and etc can be easily handled by arrays.
- An array is a collection of homogeneous data elements described by a single name.
- Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis.
- If an element of an array is referenced by single subscript, then the array is known as one dimensional array or linear array and if two subscripts are required to reference an element, the array is known as two dimensional array and so on.
- Analogously the arrays whose elements are referenced by two or more subscripts are called multi dimensional arrays.

1.1.1. One Dimensional Array

One-dimensional array (or linear array) is a set of '**n**' finite numbers of homogenous data elements such as:

1. The elements of the array are referenced respectively by an index set consisting of '**n**' consecutive numbers.
2. The elements of the array are stored respectively in successive memory locations.

'n' number of elements is called the length or size of an array. The elements of an array 'A' may be denoted in C as A [0], A [1], A [2], ... A [n –1].
The number 'n' in A [n] is called a subscript or an index and A[n] is called a subscripted variable.
If 'n' is 10, then the array elements A [0], A [1]......A[9] are stored in sequential memory locations as follows :

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

In C, array can always be read or written through loop. To read a one-dimensional array, it requires one loop for reading and writing the array, for example:

```
For reading an array of 'n' elements
for (i = 0; i < n; i ++)
        scanf ("%d",&a[i]);
For writing an array
for (i = 0; i < n; i ++)
        printf ("%d", a[i]);
```

1.1.1.1. Insertion in One-Dimensional Array

Insertion of a new element in an array can be done in two ways:

1. Insertion at the end of an array
   Providing the memory space allocated for the array enough to accommodate the additional element can easily do insertion at the end of an array.
2. Insertion at the required position
   For inserting the element at required position, element must be moved downwards to new locations. To accommodate the new element and keep the order of the elements. For inserting an element into a linear arraet **insert(a,len,pos,num)** where **a** is a linear array,**len** be total numbers of elements with an array, **pos** is the position at whicnumber **num** will be inserted.

ALGORITHM

```
1. [Initialize the value of i] set i=len
2. repeat for i=len down to pos
   [Shift the elements down by 1 position]
   Set a[i+1] = a[i]
   [End of Loop]
3. [Insert the element at required position]
   Set a[pos] = num
4. [Reset len] set len = len +1
5. Display the new list of arrays
6. End
```

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
void insertat(int num,int pos, int len,int a[]);
int main()
{
    int a[10]={0},len, pos,i,num;
    printf("Enter the length of an array\n");
    scanf("%d",&len);
    for(i=0;i<len;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Length = %d\n",len);
    printf("Enter number and position to be insert\n");
    scanf("%d%d",&num,&pos);
    pos--;
    printf("Array before insertion\n");
    for(i=0;i<=len;i++)
        printf("a[%d] = %d\n ",i,a[i]);

    insertat(num,pos,len,a);

    printf("Array after insertion\n");
    for(i=0;i<=len;i++)
        printf("a[%d] = %d\n ",i,a[i]);
    return 0;
}
void insertat(int num,int pos, int len,int a[])
{
    int i;
    for(i=len;i>=pos;i--)
        a[i+1]=a[i];
    a[pos] = num;
    if(pos > len)
        printf("Insertion outside the array");
    len = len++; }
```

1.1.2. Multi Dimensional Array

If we are reading or writing two-dimensional array, two loops are required. Similarly the array of '**n**' dimensions would require '**n**' loops. The structure of the two dimensional array is illustrated in the following figure:

int A[3][3];

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]

For reading an array of 'r' rows and 'c'

```
for (i = 0; i < r; i ++)
      for( j = 0;j<c;j++)
            scanf ("%d",&a[i][j]);
For writing an array
for (i = 0; i < r; i ++)
      for( j = 0;j<c;j++)
            printf ("%d", a[i][j]);
```

1.1.3. Sparse Array

- A sparse array is an array where nearly all of the elements have the same value (usually zero) and this value is a constant.
- One-dimensional sparse array is called sparse vectors and two-dimensional sparse arrays are called sparse matrix. Following is an example of sparse matrix where only 7 elements are non-zero among 35 elements where 28 elements are zeros.

$$\begin{matrix} 0 & 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 31 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 \end{matrix}$$

We will store only non-zero elements in the above sparse matrix because storing all the elements of the sparse array will be consisting of memory sparse. The non-zero elements are stored in an array of the form.

A[0......**n**][1......3] , Where '**n**' is the number of non-zero elements in the array. In above array, 'n = 7'.The sparse array given above may be represented in the array A[0......7][1.....3].

A[0][1]   A[0][2]

	1	**2**	**3**
0	**5**	**7**	**7**
1	**1**	**3**	**8**
2	**2**	**2**	**1**
3	**2**	**6**	**9**
4	**3**	**4**	**3**
5	**4**	**2**	**31**
6	**4**	**6**	**4**
7	**5**	**5**	**7**

Table: Sparse array representation

- The element A[0][1] and A[0][2] contain the number of rows and columns of the sparse array.
- A[0][3] contains the total number of nonzero elements in the sparse array.
- A[1][1] contains the number of the row where the first nonzero element is present in the sparse array.
- A[1][2] contains the number of the column of the corresponding nonzero element.
- A[1][3] contains the value of the nonzero element.
- In above sparse array, the first nonzero element can be found at 1st row in 3rd column.

1.1.4. Vectors

- A vector is a one-dimensional ordered collection of numbers. Normally, a number of contiguous memory locations are sequentially allocated to the vector. A vector size is fixed and, therefore, requires a fixed number of memory locations. A vector can be a column vector which represents a '**n**' by 1 ordered collections, or a row vector which represents a1 by '**n**' ordered collections.

1.2. Structure

In nearly every programming language, it becomes necessary to group data together to form larger constructs. We do this for many reasons, but mainly because it's easier for us to think of a group of information as part of a larger item. For example, when you think of someone's date of birth, we are really thinking of at least 3 numbers, the day of the month, the month, and the year. But encapsulating all this data into a single place, we can start thinking of it as a single unit, and that's very advantageous over keeping track of variables that are related by yourself.

So, now that we have some belief that it's a good idea to put data members together somehow, how do we go about accomplishing this task? Easy! 'C' has a built-in method for working with sets of data, and they are called **Structures.**

Structure, as name implies, are containers for varying kinds of data. You can put any kind of data within structure: Integer, character, pointers, array, even other structure, if you like. Let's take a look at a small example to see how structure work.

```
struct DOB {
      int month;
      int day;
      int year;
};
struct DOB birthday;
birthday.month =6;
birthday.day = 2;
birthday.year = 1980;
```

Now, Let's take a look at our various parts. All structure has 2 parts, the definition and the declaration. Once they are defined and declared, we can use them. We'll start with the definition, since we must first define a structure before we can declare it or use it.

Structures are defined using the **struct** keyword. That keyword is followed by the name of the structure. The name of a structure is just like a variable type name, like int or char, so it must be unique. The name of the structure we defined is called 'DOB', short for the date of birth. Once we have the name of the structure, we need to open a brace { to enclose the member of the structure. We use the same procedure for declaring variables inside a structure as we do for declaring variables in a function. Except we don't initialize any of the values. This is because these variables do not actually exist yet, because we haven't created any structure yet, we are only telling the compiler how to create structure so that when we tell it to build one, it will know how. Finally, we are finished declaring the members of the structure, we close the brace } and signify the end with a semi-colon.

To actually declare a structure, like we declare a variable, we must use the **struct** keyword again with the name of the structure (so the compiler knows which structure to create) and then give it a variable name, So we can used struct DOB to tell the compiler we want it to create a DOB structure, and then give it a variable name we can use to access the structure, which we called birthday.

Finally, how do we access the members of a structure? Well, that's easy. 'C' has built-in operators, called **dot operator.** The dot operator is just that, a dot, or more properly, a period. So, we use the name of the structure (the variable name, not the structure name), followed by the dot operator, and then the name of the variable structure member we want to access. This is just like using any other variable, so you can do anything with a structure member that you can do with a variable. An another example is

```
struct DOB
{
      int month, day, year;
};
struct Person
{
      char name[20];
      int height;
      struct DOB birthday;
};
void printPerson(struct Person per)
{
        scanf("%s %d",&per.name,&per.height);
       per.birthday.month=6;
       per.birthday.day = 2;
      per.birthday.year = 1980;
      //print the structure information
       printf("Name : %s\n", per.name);
       printf("Height : %d inches\n",per.height);
      printf("Birthday:      %d      -%d-%d\n",per.birthday.month,
      per.birthday.day, per.birthday.year);
}
void main()
{
      //Clear the screen
      clrscr();
      struct Person per;
      printPerson(per);
      getch();
}
```

1.3. Union

Well, structure is the first and probably most popular way to group data members together in 'C'. But it is not the way of grouping data together. Suppose we want to use either an integer, or a floating-point number to store a number, but we don't want to use both. Well, instead of declaring two variables, we could put them together in what is called a union in 'C'. Union are like structure in that they combine data members together in a single block. But the difference is that we can only use the variable in the union at a time. This is because the variable space for all the variables are defined on the top of one another.

Why would anyone want to create a structure where you could only use one variable? The answer is easy: Memory Management. If we need to store two different kinds of data, there is no reason to reserve space for both kinds. We can just reserve space for the largest one, and save the space used by the smaller one.

As far as using union, they are used in exactly the same manner as structure. The dot operator accesses their variables or in this case, their member variable, singular. Members of a union can only be accessed one at a time.

```
#include<stdio.h>

     typedef union myunion
     {
      double PI;
      int B;
     }MYUNION;

     int main()
     {
      MYUNION numbers;
      numbers.PI = 3.14;
      numbers.B = 50;

         return 0;
      }
```

1.4. Pointers

Pointers are core to the 'C' programming language. They allow us to access memory directly, which lets us to write very efficient and thus very fast programs. Basically, Pointer is an address in memory somewhere. We don't know where, and we don't need to know where. That's why we have pointers. Pointers are one of the most useful concepts in 'C'. The reason is that pointers allow us to keep a single copy of a variable (i.e. the memory we are using) and change that memory from anywhere in the program.

A pointer, in general, is a number representing a memory address. We call it a pointer because it is pointing to a place in memory we want to manipulate. Declaration of pointer can be done as following:

*int *ptr; //declare a pointer to a integer.*
*void *ptr; //declares a pointer of unknown type.*
*char *ptr; //pointer to a character, often used for string literals.*

A pointer is declared by putting an asterisk * in front of variable name. We can also have void pointer, which are just pointers of unknown type. Other kinds of pointers (char, float, int, double etc) are just the memory addresses of variables.

Why would we need to know the memory address of a variable?

- If we want to change the value of a variable without copying the variable, and without assigning it to the return value of a function. Simple, if we supply the address of the variable, we can change it from any place in our program. This makes our programs very efficient.

1.5. Memory Allocation in C

There are two types of memory allocations in C.

A. Static memory allocations or Compile time

In static or compile time memory allocations, the required memory is allocated to the variables at the beginning of the program. Here the memory to be allocated is fixed and is determined by the compiler at the compile time itself. For example

int i, j;   //Two bytes per (total 2) integer variables
float a[5], f; //Four bytes per (total 6) floating point variables

When the first statement is compiled, two bytes for both the variable 'i' and 'j' will be allocated. Second statement will allocate 20 bytes to the array A [5 elements of floating point type, i.e., 5*4] and four bytes for the variable 'f '.

But static memory allocation has following drawbacks.

- If you try to read 15 elements of an array whose size is declared as 10, then first 10 values and other five consecutive unknown random memory values will be read. Again if you try to assign values to 15 elements of an array whose size is declared as 10, then first 10 elements can be assigned and the other 5 elements cannot be assigned/accessed.
- The second problem with static memory allocation is that if you store less number of elements than the number of elements for which you have declared memory, and then the rest of the memory will be wasted.

B. Dynamic memory allocations or Run time

The dynamic or run time memory allocation helps us to overcome this problem. It makes efficient use of memory by allocating the required amount of memory whenever is needed. In most of the real time problems, we cannot predict the memory requirements. Dynamic memory allocation does the job at run time. C provides the following dynamic allocation and de-allocation functions:

malloc( ): The malloc( ) function is used to allocate a block of memory in bytes. The malloc() function returns a pointer of any specified data type after allocating a block of memory of specified size. It is of the form

*ptr = (int_type *) malloc (block_size)*

'ptr' is a pointer of any type 'int_type' byte size is the allocated area of memory block.

For example

*ptr = (int *) malloc (10 * sizeof (int));*

On execution of this statement, 10 times memory space equivalent to size of an 'int' byte is allocated and the address of the first byte is assigned to the pointer variable 'ptr' of type 'int'.

malloc() function allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it fails, it returns a **NULL** pointer.

Similarly, memory can be allocated to structure variables. For example

```
struct Employee
{
int Emp_Code;
char Emp_Name[50];
float Emp_Salary;
};
```

Here the structure is been defined with three variables.

```
struct Employee *str_ptr;
str_ptr = (struct Employee *) malloc(sizeof (struct Employee));
```

When this statement is executed, a contiguous block of memory of size 56 bytes (2 bytes for integer employee code, 50 bytes for character type Employee Name and 4 bytes for floating point type Employee Salary) will be allocated.

calloc( ): The calloc() function works exactly similar to malloc() function except for the fact that it needs two arguments as against one argument required by malloc() function. While malloc() function allocates a single block of memory space, calloc() function allocates multiple blocks of memory, each of the same size, and then sets all bytes to zero. If there is no sufficient memory space, calloc() returns a NULL pointer.

```
ptr = (int_type*) calloc(n, sizeof (block_size));
ptr = (int_type*) malloc(n* (sizeof (block_size));
```

The above statement allocates contiguous space for '**n**' blocks, each of size of block_size bytes.

```
ptr = (int *) calloc(25, 4);
ptr = (int *) calloc(25,sizeof (float));
```

Here, in the first statement the size of data type in byte for which allocation is to be made (4 bytes for a floating point numbers) is specified and 25 specifies the number of elements for which allocation is to be made.

- The memory allocated using malloc() function contains garbage values, thememory allocated by calloc() function contains the value zero.

realloc( ): if we need to change an already allocated memory block then we can use realloc() function. The syntax of this function is

```
ptr = realloc(ptr, New_Size)
```

Where 'ptr' is a pointer holding the starting address of the allocated memory block. And New_Size is the size in bytes that the system is going to reallocate. Following example will elaborate the concept of reallocation of memory.

```
ptr = (int *) malloc(sizeof (int));
ptr = (int *) realloc(ptr, sizeof (int));
ptr = (int *) realloc(ptr, 2);
```

free( ): Dynamic memory allocation allocates block(s) of memory when it is required and deallocates or releases when it is not in use. The free() function is used to deallocate the previously allocated memory using malloc() or calloc() function. The syntax of this function is

free(ptr);

'ptr' is a pointer to a memory block which has already been allocated by malloc() or calloc() functions. Trying to release an invalid pointer may create problems and cause
system crash.

1.6. Class:

- In C++ programming it is possible to separate program specific data types through the use of classes.
- Classes define types of data structures and the functions that operate on those data structures.
- A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package.
- Instances of these data types are known as objects and can contain member variables, constants, member functions, and operators defined by the programmer.
- Syntactically, classes are extensions of the C struct, which cannot contain functions or overloaded operators. Following example shows the definition of class.

```
class Box
{
public:
     double length;    // Length of a box
     double breadth;   // Breadth of a box
     double height;    // Height of a box
};
```

Class member functions: A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
Class access modifiers: A class member can be defined as public, private or protected. By default members would be assumed as private.
Constructor & destructor: A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.
Pointer to C++ classes: A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.

Pointer to structure
You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now you can store the address of a structure variable in the above defined pointer variable

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, we have to use the -> operator. The following program shows that the sample program in C++ using pointer to structure.

```
#include <iostream>

using namespace std;
void printBook( struct Books *book );

struct Books
{
    char  title[50];
    char  author[50];
    char  subject[100];
    int   book_id;
};

int main( )
{
    struct Books Book1;        // Declare Book1 of type Book
    struct Books Book2;        // Declare Book2 of type Book

    // Book 1 specification
    strcpy( Book1.title, "Learn C++ Programming");
    strcpy( Book1.author, "Chand Miyan");
    strcpy( Book1.subject, "C++ Programming");
    Book1.book_id = 6495407;

    // Book 2 specification
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Yakit Singha");
    strcpy( Book2.subject, "Telecom");
    Book2.book_id = 6495700;

    // Print Book1 info, passing address of structure
    printBook( &Book1 );

    // Print Book1 info, passing address of structure
    printBook( &Book2 );

    return 0;
}
// This function accept pointer to structure as parameter.
void printBook( struct Books *book )
{
    cout << "Book title : " << book->title <<endl;
    cout << "Book author : " << book->author <<endl;
    cout << "Book subject : " << book->subject <<endl;
    cout << "Book id : " << book->book_id <<endl;
}
```

2. Abstract Data Type (ADT)

- A useful tool for specifying the logical properties of a data type is the abstract data type or ADT.
- Fundamentally, a data type is a collection of values and set of operation on those values.

- That collection and those operations form a mathematical construct that may be implemented using a particular hardware or software data structure.
- The term "Abstract Data type" refers to the basic mathematical concept that defines the data type.
- Formally, An abstract data type is a data declaration packaged together with the operations that are meaningful on the data type.
- In other words, we can encapsulate the data and the operation on data and we hide them from user.

Examples of ADT

1. Linear ADTs:
   Stack (last-in, first-out) ADT
   Queue (first-in, first-out) ADT
   a. Lists ADTs
      Arrays
      Linked List
      Circular List
      Doubly Linked List
2. Non-Linear ADTs
   a. Trees
      Binary Search Tree ADT
   b. Heaps
   c. Graphs
      Undirected
      Directed
   d. Hash Tables

We can perform following basic operations on ADTs insert(),delete(), search(), findMin(), findMax(),findNext(), findPrevious(), enqueue(), dequeue() etc.

3. An Introduction to Data Structure

- Data are simply value or set of values. A data item refers to a single unit of values.
- Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other.
- Data structure can also be defined as the logical or mathematical model of a particular organization of data is called data structure.
- Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity. Data structure affects the design of both the structural and functional aspects of a program.
  Algorithm + Data Structure = Program
- Data structures are the building blocks of a program. Hence proper selection of data structure increases the productivity of programmers due to the proper designing and the use of efficient algorithms.
- If the problem is analyzed and divided into sub problems, the task will be much easier i.e., divide, conquer and combine.
- A complex problem usually cannot be divided and programmed by set of modules unless its solution is structured or organized.
- This is because when we divide the big problems into sub problems, different programmers or group of programmers will program these sub problems.
- By choosing a particular structure (or data structure) for the data items, certain data items

become friends while others loses its relations.

- The representation of a particular data structure in the memory of a computer is called a storage structure. That is, a date structure should be represented in such a way that it utilizes maximum efficiency.
- The data structure can be represented in both main and auxiliary memory of the computer.
- A storage structure representation in auxiliary memory is often called a file structure.
- The data structure and the operation on organized data items can integrally solve the problem using computer

Data structure = Organized data + Operations

In short Data Structure is an agreement about:
• How to store a collection of objects in memory,
• What operations we can perform on that data,
• The algorithms for those operations, and
• How time and space efficient those algorithms are

3.1. Data Structure Application Example

1. How does Google quickly find web pages that contain a search term?
2. What's the fastest way to broadcast a message to a network of computers?
3. How can a subsequence of DNA be quickly found within the genome?
4. How does your operating system track which memory (disk or RAM) is free?

3.2. Classification of Data structure

Data structure can be classified into 2 categories

1. Primitive data structures: These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level. They are integers, floating point numbers, characters, string constants, pointers etc.
2. Non-primitive data structures: It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. Array, list, files, linked list, trees and graphs fall in this category.

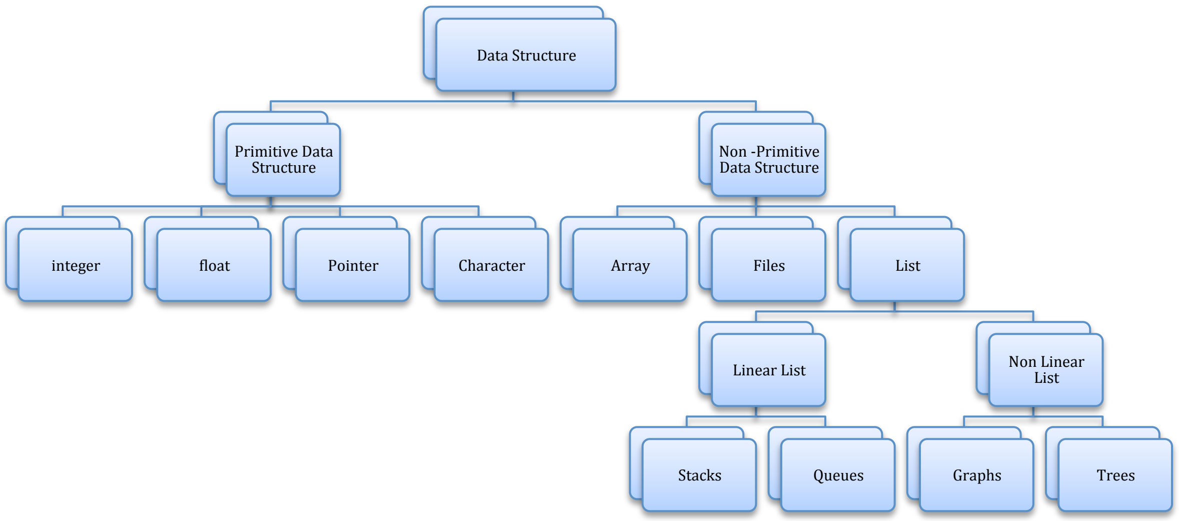


Fig: Classification of Data Structure

- Basic operations on data structure are to create a (non-primitive) data structure.

Chapter 2: THE STACK

- A stack is one of the most important and useful non-primitive linear data structure in computer science.
- It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the **top** of the stack.
- As all the addition and deletion in a stack is done from the **top** of the stack, the last added element will be first removed from the stack.
- That is why the stack is also called **Last-in-First-out** (LIFO).
- The most frequently accessible element in the stack is the **top** most elements, whereas the least accessible element is the **bottom** of the stack.
- The insertion (or addition) operation is referred to as **push**, and the deletion (or remove) operation as **pop**.
- A stack is said to be empty or underflow, if the stack contains no elements.
- At this point the **top of the stack is present at the bottom of the stack**. And it is overflow when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point the **top pointer is at the highest location of the stack**.

2.1. OPERATION PERFORMED ON STACK

The primitive operations (ADT on the stack family) performed on the stack are as follows:

1. **PUSH (X,S):** The process of adding (or inserting) a new element to the top of the stack is

called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the **top is incremented by one**. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

2. **POP(S):** The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the **stack is decremented by one**. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.
3. **MAKENULL (S):** Make stack S is an empty stack.
4. **TOP (S):** Return the elements at the top of the stack.
5. **EMPTY (S):** Return true if stack S is empty; return false otherwise.

Stack Implementation

1. Static Implementation (Using Arrays)
2. Dynamic Implementation (Using Pointer)

Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (i.e., increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (i.e., memory utilization). The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

Algorithm for PUSH

Suppose STACK [SIZE] is a one-dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

```
1. If TOP = SIZE – 1, then:
        (a) Display "The stack is in overflow condition"
        (b) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit
```

Algorithm for POP

Suppose STACK [SIZE] is a one-dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

```
1. If TOP < 0, then
        (a) Display "The Stack is empty"
        (b) Exit
2. Else remove the Top most element
    a. DATA = STACK[TOP]
    b. TOP = TOP – 1
5. Exit
```

Program to demonstrate the operation performed on stack and its implementation-using array.

```
#include<stdio.h>
//Defining the maximum size of the stack
#define MAXSIZE 100
//Declaring the stack array and top variables in a structure
struct stack
{
    int stack[MAXSIZE];
    int Top;
};
//type definition allows the user to define an identifier that would
//represent an existing data type. The user-defined data type
//identifier can later be used to declare variables.
typedef struct stack NODE;
//This function will add/insert an element to Top of the stack
void push(NODE *pu)
{
    int item;
    //if the top pointer already reached the maximum allowed size then
    //we can say that the stack is full or overflow
    if(pu->Top == MAXSIZE-1)
    {
        printf("\nThe Stack Is Full");
        exit(0);
    }
    //Otherwise an element can be added or inserted by
    //incrementing the stack pointer Top as follows
    else
    {
        printf("\nEnter The Element To Be Inserted = ");
        scanf("%d",&item);
        pu->stack[++pu->Top]=item;
    }
}
//This function will delete an element from the Top of the stack
void pop(NODE *po)
{
    int item;
    //If the Top pointer points to NULL, then the stack is empty
    //That is NO element is there to delete or pop
    if (po->Top == -1)
        printf("\nThe Stack Is Empty");
         exit(0);
    //Otherwise the top most element in the stack is popped or
    //deleted by decrementing the Top pointer
    else
    {
        item=po->stack[po->Top--];
        printf("\nThe Deleted Element Is = %d",item);
    }
}
//This function to print all the existing elements in the stack
void traverse(NODE *pt)
{
    int i;
    //If the Top pointer points to NULL, then the stack is empty
    //That is NO element is there to delete or pop
    if (pt->Top == -1)
        printf("\nThe Stack is Empty");
    //Otherwise all the elements in the stack is printed
    else
    {
        printf("\n\nThe Element(s) In The Stack(s) is/are...");
        for(i=pt->Top; i>=0; i--)
            printf ("\n %d",pt->stack[i]);
    }
}
int main( )
{
    int choice;

    int check =1;
    //Declaring an pointer variable to the structure
    NODE *ps;
    ps = (NODE*)malloc(100* sizeof(NODE));
    //Initializing the Top pointer to NULL
    ps->Top=-1;
    do
    {

        //A menu for the stack operations
        printf("\n1. PUSH");
        printf("\n2. POP");
        printf("\n3. TRAVERSE");
        printf("\nEnter Your Choice = ");
        scanf ("%d", &choice);
        switch(choice)
        {
            case 1://Calling push() function by passing
                //the structure pointer to the function
                push(ps);
                break;
            case 2://calling pop() function
                pop(ps);
                break;
            case 3://calling traverse() function
                traverse(ps);
                break;
            default:
                printf("\nYou Entered Wrong Choice") ;
        }
        printf("\n\nPress 1 for continue and 2 To Exit = ");
        scanf("%d",&check);
    }while(check != 2);
    return 0;
}
```

2.2. APPLICATION OF STACK

There are a number of applications of stacks. Compiler internally uses stack when we implement (or execute) any recursive function. If we want to implement a recursive function non-recursively, stack is programmed explicitly. Stack is also used to evaluate a mathematical expression and to check the parentheses in an expression.

2.2.1. Recursion (Will Discussed in Chapter 5)

2.2.2. Expression

An expression is fined as the number of operands or data items combined with several operators. An application of stack is calculation of postfix expression. There are basically three types of notation for an expression (Mathematical expression).

1. Infix notation
2. Prefix notation
3. Postfix notation

The **infix** notation is what we come across in our general mathematics, where the operator is written in-between the operands. For example: The expression to add two numbers A and B is written in infix notation as:

A + B ; here the operator '+' is written in between the operands A and B.

The **prefix** notation is a notation in which the operator(s) is written before the operands, it is also called polish notation in the honor of the polish mathematician Jan Lukasiewicz who developed this notation. The same expression when written in prefix notation looks like:

+ A B ;As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

In the **postfix** notation the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as suffix notation or reverse polish notation. The above expression if written in postfix expression looks like:

A B +

The prefix and postfix notations are not really as awkward to use as they might look. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction:

add(A, B)

Note that the operator add (name of the function) precedes the operands A and B. Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated.

In a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

Operator Precedence

Exponential operator	^	Highest precedence (**Note : ^ = $**)
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

<u>2.2.2.1. Conversion of Infix to Postfix Expression</u>

The method of converting infix expression A + B * C to postfix form is:

A + B * C	Infix Form
A + (B * C)	Parenthesized expression
A + (B C *)	Convert the multiplication
A (B C *) +	Convert the addition
A B C * +	Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression B * C is parenthesized first before A + B.

3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Examples: Give postfix form for A + [(B + C) + (D + E) * F] / G

Solution: Evaluation order is
A + { [(BC +) + (DE +) * F] / G}
A + { [(BC +) + (DE + F *] / G}
A + { [(BC + (DE + F * +] / G} .
A + [BC + DE + F *+ G /]
ABC + DE + F * + G / + Postfix Form

Give postfix form for (A + B) * C / D + E ^ A / B

Solution: Evaluation order is
[(AB +) * C / D] + [(EA ^) / B]
[(AB +) * C / D] + [(EA ^) B /]
[(AB +) C * D /] + [(EA ^) B /]
(AB +) C * D / (EA ^) B / +
AB + C * D / EA ^ B / + Postfix Form

Algorithm: Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators; P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-). The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. The algorithm is completed when the stack is empty.

```
1. Push "(" onto stack, and add")" to the end of P.
2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is
   empty.
3. If an operand is encountered, add it to Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator ⊗ is encountered, then:
      i. Repeatedly pop from stack and add P each operator (on the top of stack), which has
         the same precedence as, or higher precedence than ⊗.
     ii. Add ⊗ to stack.
6. If a right parenthesis is encountered, then:
      i. Repeatedly pop from stack and add to P (on the top of stack until a left parenthesis
         is encountered.
     ii. Remove the left parenthesis. [Do not add the left parenthesis to P.]
7. Exit
   Note: Special character ⊗ is used to symbolize any operator in P.
```

Consider the following arithmetic infix expression P

P = A + (B / C - (D * E $ F) + G) * H

Following table shows the character (operator, operand or parenthesis) scanned, status of the stack and postfix expression Q of the infix expression P.

Character Scanned	Stack	Postfix String
A	(	A
+	(+	A
(	(+ (	A

B	(+ (	AB
/	(+ (/	AB
C	(+ (/	ABC
-	(+ (-	ABC/
(	(+ (- (	ABC/
D	(+ (- (	ABC/D
*	(+ (- (*	ABC/D
E	(+ (- (*	ABC/DE
^	(+ (- (* ^	ABC/DE
F	(+ (- (* ^	ABC/DEF
)	(+ (-	ABC/DEF$*
+	(+ (+	ABC/DEF$* -
G	(+ (+	ABC/DEF$* - G
)	(+	ABC/DEF$* - G+
*	(+ *	ABC/DEF$* - G+
H	(+ *	ABC/DEF$* - G + H
)		ABC/DEF$*- G + H * +

Input prefix string:
a+b*c+(d*e+f)*g

Character scanned	Operator Stack	Postfix String
a		a
+	+	a
b	+	ab
*	+ *	ab
c	+ *	abc
+	+	abc * +
(	+ (	abc * +
d	+ (	abc * + d
*	+ (*	abc * + d
e	+ (*	abc * + de
+	+ (+	abc * + de *
f	+ (+	abc * + de * f
)	+	abc * + de * f +
*	+ *	abc * + de * f +
g	+ *	abc * + de * f + g
		abc * + de * f + g * +

Another Example: Input string ((A - (B+C)) * D) $ (E + F)

Character scanned	Operator stack	Postfix string
(	(	
(	((	
A	((	A

-	((-	A
(	((- (	A
B	((- (	AB
+	((- (+	AB
C	((- (+	ABC
)	((-	ABC+
)	(	ABC+ -
+	(*	ABC+ -
D	(*	ABC+ - D
)		ABC+ - D*
$	$	ABC+ - D*
(	$ (	ABC+ - D*
E	$ (	ABC+ - D*E
+	$ (+	ABC+ - D*E
F	$ (+	ABC+ - D*EF
)	$	ABC+ - D*EF+
		ABC+ - D*EF+$

2.2.2.2. Evaluations of postfix Expression

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator ⊗ is encountered, then:
   i. Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   ii. Evaluate B ⊗ A.
   iii. Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.

Let us assume that the following input string is valid postfix string.

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Char Scanned	Operand1	Operand2	Value	Operator stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	7	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3

+	49	3	52	52

Another Example: 6 5 2 3 + 8 * + 3 + *

Char Scanned	Operand 1	Operand 2	Value	Operator stack
6				6
5				6,5
2				6,5,2
3				6,5,2,3
+	2	3	5	6,5,5
8	2	3	5	6,5,5,8
*	5	8	40	6,5,40
+	5	40	45	6,45
3	5	40	45	6,45,3
+	45	3	48	6,48
*	6	48	288	288

2.2.3. Backtracking

For game playing, finding paths and exhaustive searching**.**

Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal.

- Find your way through a maze.
- Find a path from one point in a graph (roadmap) to another point.
- Play a game in which there are moves to be made (checkers, chess).

In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative Consider the maze. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative. Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

2.2.4. Sorting

a. Quick sort (Will cover in chapter 7)
b. Merge sort (Will cover in chapter 7)
c. Tree Traversal (Will cover in chapter 6)

2.2.5. Runtime memory management

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack.

Forth uses two stacks, one for argument passing and one for subroutine return addresses. The use of a return stack is extremely commonplace, but the somewhat unusual use of an argument stack for a human-readable programming language is the reason Forth is referred to as a *stack-based* language.

Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine.

Chapter 3: THE QUEUE

QUEUE

A queue is logically a first in first out (**FIFO** or first come first serve) linear data structure. It is a homogeneous collection of elements in which new elements are added at one end called **rear**, and the existing elements are deleted from other end called **front**.

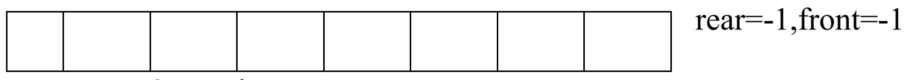
The basic operations (Primitive Operations) also called as **Queue ADT** that can be performed on queue are

1. **Enqueue() or Insert()** (or add) an element to the queue (push) from the **rear** end.
2. **Dequeue() or Delete()** (or remove) an element from a queue (pop) from **front** end.
3. **Front**: return the object that is at the front of the queue without removing it.
4. **Empty**: return true if the queue is empty otherwise return false.
5. **Size**: returns the number of items in the queue.

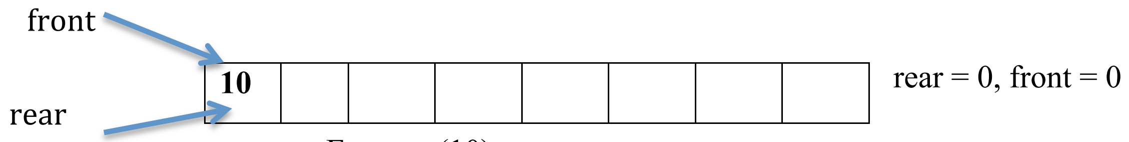
TYPES OF QUEUE

i) Linear Queue or Simple queue
ii) Circular queue
iii) Double ended queue (de-queue)
iv) Priority queue: Priority queue is generally implemented using linked list

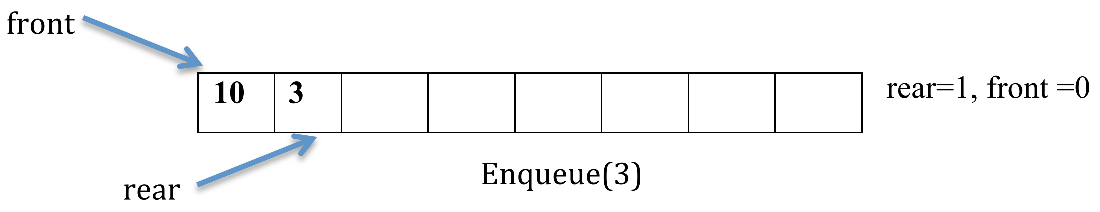
Enqueue operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. **Pop** operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Initially rear is set to -1 and front is set to -1. The queue is empty whenever **rear < front** or both the *rear and front is equal to -1*. Total number of elements in the queue at any time is equal to **rear-front+1**, when implemented using arrays. Below are the few operations in queue.



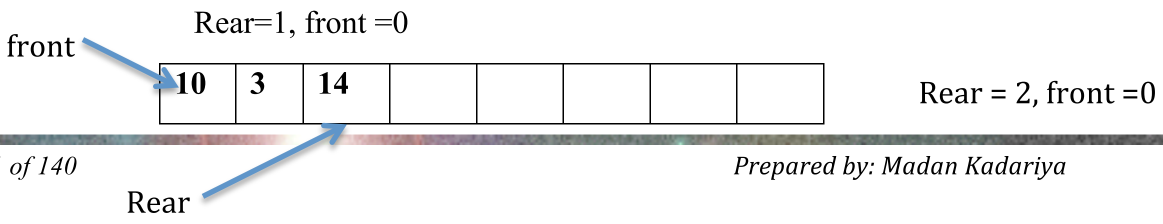
Queue is empty

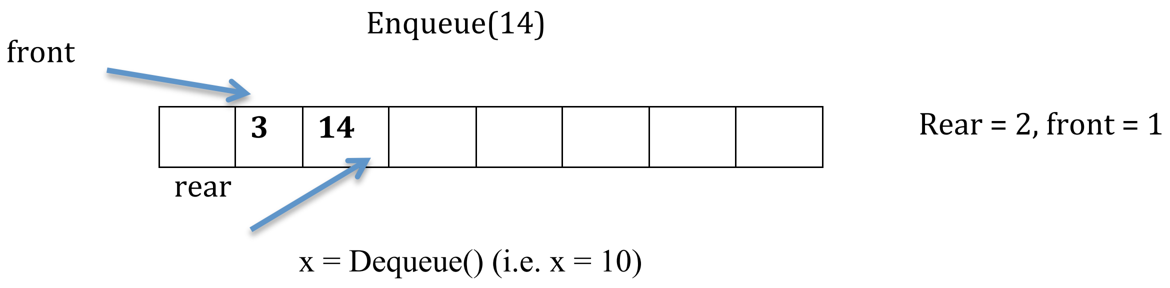


Enqueue(10)



Enqueue(3)





Note. During the insertion of first element in the queue we always increment the front by one.

Queue can be implemented in two ways:
1. Using arrays (static)
2. Using pointers (dynamic)

If we try to dequeue (or delete or remove) an element from queue when it is empty, **underflow** occurs. It is not possible to delete (or take out) any element when there is no element in the queue. Suppose maximum size of the queue (when it is implemented using arrays) is 50. If we try to Enqueue (or insert or add) an element to queue, overflow occurs. When queue is full it is naturally not possible to insert any more elements.

3.1. ALGORITHM FOR QUEUE OPERATIONS (Linear Queue)

Let Q be the arrays of some specified size say SIZE.

3.1.1. Inserting An Element Into The Queue

```
1.  Initialize front=-1 rear = -1;  [Create an Empty Queue]
2.  Input the value to be inserted and assign to variable "data"
3.  If (rear >= SIZE)
      i.  Display "Queue overflow"
     ii.  Exit
4.  Else
      i.  Rear = rear +1
5.  Q [rear] = data
6.  Exit
```

3.1.2. Deleting An Element From Queue

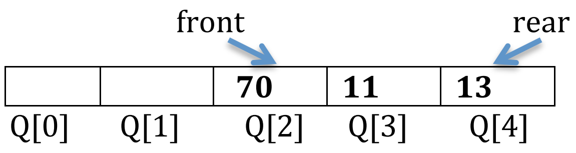
```
1.  If (rear< front) or if(front ==-1 && rear == -1);[ checking for queue is empty]
      i.  front = -1, rear = -1;(optional)
     ii.  Display "The queue is empty"
    iii.  Exit
2.  Else
    i.    Data = Q [front]
3.  Front = front +1
4.  Exit
```

3.1.3. Program to Implement Queue Using Array

```
//PROGRAM TO IMPLEMENT QUEUE USING ARRAYS
#include<stdio.h>
#include <stdlib.h>
#define MAX 50
int queue_arr[MAX];
int rear = -1;
int front =-1;
//This function will insert an element to the queue
void Enqueue ()
{
    int added_item;
    if (rear==MAX-1)
    {
        printf("\nQueue Overflow\n");
        return;
    }
    else
    {
        if (isEmpty() == 1) /*If queue is initially empty */
         {
            printf("Queue  is  Empty,  Your  are  going  to  create  a
queue\n");
            front = 0;
         }
        printf("\nInput the element for adding in queue:");
        scanf("%d", &added_item);
        rear=rear+1;
        //Inserting the element
        queue_arr[rear] = added_item ;
    }
}/*End of insert()*/
//This function will delete (or pop) an element from the queue
void Dequeue()
{
    if (isEmpty() == 1)
    {
        printf ("\nQueue Underflow or Queue is Empty\n");
        return;
    }
    else
    {
        //deleteing the element
        printf ("\nElement deleted from queue is : %d\n",
                queue_arr[front]);
        front=front+1;
    }
}/*End of del()*/
//Displaying all the elements of the queue
void Traverse()
{
    int i;
    //Checking whether the queue is empty or not
    if (isEmpty() ==1)
    {
        printf ("\nQueue is empty\n");
        return;
    }
    else
    {
        printf("\nFront is %d and Queue is :\n",front);
        for(i=front;i<= rear;i++)
            printf("%d ",queue_arr[i]);
        printf("\n");
    }
}/*End of display() */

/*Checks whether the Queue is Empty or not */
int isEmpty()
{
    if (front > rear) // if(front == -1 && rear == -1)
    {
        return 1; // True
    }
    else
        return 0; //False

}
/*End if isEmpty() */


int main()
{
    int choice;
    while (1)
    {
        //clrscr();
        //Menu options
        printf("\n1.Enqueue\n");
        printf("2.Dequeue\n");
        printf("3.Traverse\n");
        printf("4.Size\n");
        printf("5.Quit");
        printf("\nEnter your choice:");
        scanf("%d", & choice);
        switch(choice)
        {
            case 1 :
                Enqueue();
                break;
            case 2:
                Dequeue();

                break;
            case 3:
                Traverse();

                break;
            case 4:
                printf("The size of the queue is %d\n",rear-front+1);
                break;
            case 5:
                exit(1);

            default:
                printf ("\n Wrong choice\n");

        }/*End of switch*/
    }/*End of while*/
    return 0;
}/*End of main()*/
```

3.2. Circular Queue

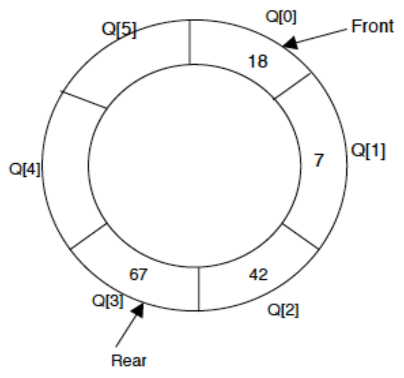
Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.



Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the rear end and hence rear points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcome if we use circular queue.

In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.

Suppose Q is a queue array of 6 elements. Enqueue() and Dequeue() operation can be performed on circular. The following figures will illustrate the same.



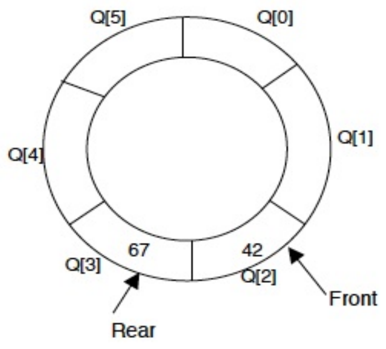


Fig1: A Circular Queue After inserting 18,7,42,67. **Fig2**: Circular Queue after popping 18,7.

After inserting an element at last location Q[5], the next element will be inserted at the very first location (i.e., Q[0]) that is circular queue is one in which the first element comes just after the last element.

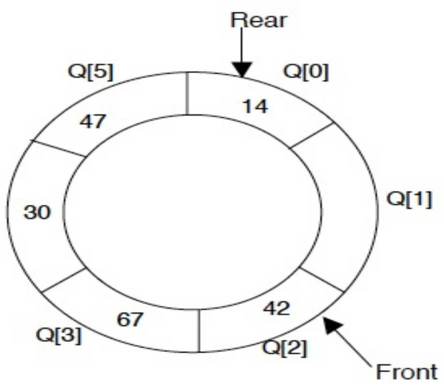


Fig3: Circular Queue after pushing 30, 47, 14

At any time the relation will calculate the position of the element to be inserted
Rear = (Rear + 1) % SIZE

After deleting an element from circular queue the position of the front end is calculated by the relation

Front= (Front + 1) % SIZE

After locating the position of the new element to be inserted, **rear**, compare it with **front**. If (**rear = front**), the queue has only one element.

3.2.1. ALGORITHMS

Let Q be the arrays of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue. DATA is the element to be inserted.

Initialize FRONT = -1 and REAR = -1

3.2.1.1. Inserting an element to circular Queue

```
1. If ((FRONT is equal to 0 && rear = MAX-1) OR front = rear+1) //check for queue full
      i. Display "Queue is full"
     ii. Exit
2. If (FRONT is equal to – 1 && rear == -1) //Empty Queue (inserting first time)
      i. FRONT = 0
     ii. REAR = 0
3. Else
      i) If (REAR == MAX-1) //Rear is at last position
              REAR = 0;
      ii) Else
              REAR = (REAR + 1) % SIZE
4. Input the value to be inserted and assign to variable "DATA"
5. Q [REAR] = DATA
6. Repeat steps 2 to 7 if we want to insert more elements
7. Exit
```

3.2.1.2. Deleting an element from a circular queue

```
1. If (FRONT is equal to – 1 and rear == -1)
     i. Display "Queue is empty"
    ii. Exit
2. Else
     i. DATA = Q [FRONT]
3. If (REAR is equal to FRONT) //Queue has only one element
     i. FRONT = -1
    ii. REAR = -1
4. Else
     i. FRONT = (FRONT +1) % SIZE
5. Repeat the steps 1 to 4, if we want to delete more elements
6. Exit
```

3.2.1.3. Program to Implement Circular Queue Using Array

```
//PROGRAM TO IMPLEMENT CIRCULAR QUEUE USING ARRAY
#include<stdio.h>
#define MAX 50
//Global Variables

     int cqueue_arr[MAX];
     int front,rear;

     //initialize the variables
     void circular_queue()
```

```
    {
        front = -1;
        rear = -1;
    }


//Function to insert an element to the circular queue
void Enqueue()
{
    int added_item;
    //Checking for overflow condition
    if ((front == 0 && rear == MAX-1) || (front == rear +1))
    {
        printf("\nQueue Overflow \n");

        return;
    }
    if (front == -1 && rear == -1) /*If queue is empty */
    {
        front = 0;
        rear = 0;
    }
    else
        if (rear == MAX-1)/*rear is at last position of queue */
            rear = 0;
        else
            rear = (rear + 1)%MAX;
    printf("\nInput the element for insertion in queue:");
    scanf("%d",&added_item);
    cqueue_arr[rear] = added_item;
}/*End of insert()*/

//This function will delete an element from the queue
void Dequeue()
{
    //Checking for queue underflow
    if (front == -1 && rear == -1)
    {
        printf("\nQueue Underflow\n");
        return;
    }
    printf("\nElement deleted from queue is: %d,\n",cqueue_arr[front]);
    if (front == rear) /* queue has only one element */
    {
         front = -1;
        rear = -1;
    }
    else
        if(front == MAX-1)
            front = 0;
        else
            front = (front + 1)%MAX;
}/*End of del()*/
//Function to display the elements in the queue
void Traverse()
{
    int front_pos = front,rear_pos = rear;
    //Checking whether the circular queue is empty or not
    if (front == -1 && rear == -1)
    {
        printf("\nQueue is empty\n");
        return;
    }
    //Displaying the queue elements
    printf("\nQueue elements:\n");
    if(front_pos <= rear_pos )
        while(front_pos <= rear_pos)
        {
            printf("%d,",cqueue_arr[front_pos]);
            front_pos++;
        }
    else
    {
        while(front_pos <= MAX-1)
        {
            printf("%d,", cqueue_arr[front_pos]);
            front_pos++;
        }
        front_pos = 0;
        while(front_pos <= rear_pos)
        {
            printf("%d,",cqueue_arr[front_pos]);
            front_pos++;
        }
    }/*End of else*/
    printf("\n");
}/*End of display() */
int main()
{
    int choice;

    //Creating the objects for the class
    circular_queue();
    while(1)
    {
        //Menu options
        printf("\n1.Enqueue\n");
        printf("2.Dequeue\n");
        printf("3.Traverse\n");
        printf("4.Quit\n");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                Enqueue();
                break;
            case 2 :
               Dequeue();
                break;
            case 3:
                Traverse();
                break;
            case 4:
                exit(1);
            default:
                printf("\nWrong choice\n");
        }/*End of switch*/
    }/*End of while*/
    return 0;
}/*End of main()*/
```

3.3. DEQUES

A deque is a homogeneous list in which elements can be added or inserted (called Enqueue operation) and deleted or removed from both the ends (which is called Dequeue operation). ie; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.

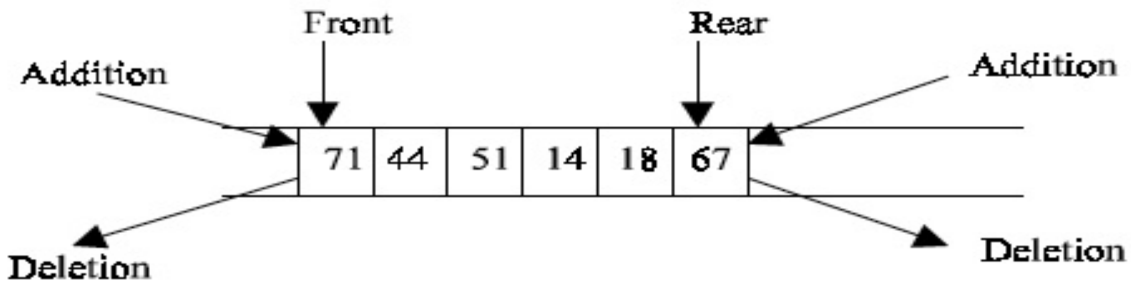


Fig4: A Deque

There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

1. **Input restricted deque:** An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.
2. **Output restricted deque:** An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

The possible operation (Deque ADT) performed on deque is

1. Add an element at the rear end (insert_rear)
2. Add an element at the front end (insert_front)
3. Delete an element from the front end (delete_front)
4. Delete an element from the rear end (delete_rear)

Only 1st, 3rd and 4th operations are performed by input-restricted deque and 1st, 2nd and 3rd operations are performed by output-restricted deque.

3.3.1. Algorithms For Inserting An Element

Let Q be the array of MAX elements. front (or left) and rear (or right) are two array index (pointers), where the addition and deletion of elements occurred. Let DATA be the element to be inserted. Before inserting any element to the queue left and right pointer will point to the -1.

<u>Insert an element at the *right side (rear end)* of the de-queue</u>

```
1.  Input the DATA to be inserted
2.  If ((front == 0 && rear == MAX-1) | | (front == rear + 1))
      i.  Display "Queue Overflow"
     ii.  Exit
3.  If (front == -1 && rear == -1)
      i.  front = 0
     ii.  rear = 0
    iii.  Q[rear] = DATA
4.  Else
```

```
    i.  if (rear != MAX -1) //Deque already contain more than one element
            a) rear = rear+1
            b) Q[rear] = DATA
   ii.  Else //rear has reach at the end of array
        [Shift all the elements from position front until rear back by 1 position]
            for(i = front; i<=rear;i++)
            Q[i-1] = Q[i]
        Q [rear] = DATA
5.  Exit
```

<u>Insert An Element At The Left Side (Front) Of The De-Queue</u>

```
1.  Input the DATA to be inserted
2.  If ((front == 0 && rear == MAX-1) | | (front == rear+1))
     i.  Display "Queue Overflow"
    ii.  Exit
3.  If (front == – 1 && rear == -1)
     i.  front = 0
    ii.  rear = 0
   iii.  Q [front] = DATA
4.  Else
     i.  if (front != 0) // Deque already contain more than one element
            a) front = front– 1
            b) Q [ front] = DATA
    ii.  Else
          a.  Shift all the elements from position front until rear ahead by 1 position
                for( i=rear; i>=front;i--)
                    Q[ i+1] = Q [i]
          b.  Q[front] = DATA
5.  Exit
```

3.3.2. ALGORITHMS FOR DELETING AN ELEMENT

Let Q be the array of MAX elements. front (or left) and rear (or right) are two array index (pointers), where the addition and deletion of elements occurred. DATA will contain the element just deleted.

<u>Delete An Element From The *Right Side (Rear side)* Of The De-Queue</u>

```
  1.  If (front == – 1 && rear == -1)
        i.  Display "Queue Underflow"
       ii.  Exit
  2.  If (front == rear) //Deque has only one element (last element)
          i)   DATA = Q [rear]
               a) front = – 1
               b) rear = – 1
  3.  Else //deque has more than one element
         i.    DATA = Q [rear]
               a) rear = rear -1
  4.  Exit
```

<u>Delete An Element From The *Left Side (Front Side)* Of The De-Queue</u>

```
1.  If (front == – 1 && rear == -1)
     i.  Display "Queue Underflow"
    ii.  Exit
2.  If(front == rear)
      i) DATA = Q [front]
```

10	14	12	60	13				
9	10	17	30	46				

```
            a)  front = - 1
            b)  rear = - 1
3.  Else // Dequeue has more than one element
        i) DATA = Q [front]
            a) Front = front +1
4.  Exit
```

3.4. PRIORITY QUEUES

Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were inserted to the queue.

For example, Consider a manager who is in a process of checking and approving files in a first come first serve basis. In between, if any urgent file (with a high priority) comes, he will process the urgent file next and continue with the other low urgent files.

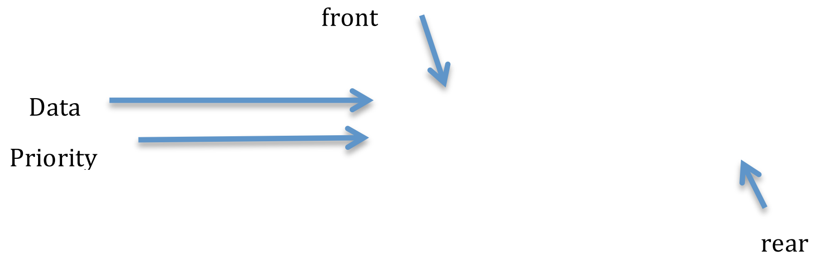


Fig5: Priority Queue representation using Array

Above Fig5 gives a pictorial representation of priority queue using arrays after adding 5 elements (10,14,12,60,13) with its corresponding priorities (9,10,17,30,46). Here the priorities of the data(s) are in ascending order. Always we may not be pushing the data in an ascending order. From the mixed priority list it is difficult to find the highest priority element if the priority queue is implemented using arrays. Moreover, the implementation of priority queue using array will yield n comparisons (in liner search), so the time complexity is much higher than the other queue for inserting an element. So it is always better to implement the priority queue using linked list - where a node can be inserted at anywhere in the list.

3.5. APPLICATION OF QUEUES

1. Round robin techniques for processor scheduling is implemented using queue.
2. Printer server routines (in drivers) are designed using queues.
3. All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.
4. Disk Driver: maintains a queue od disk input/output requests
5. Scheduler (e.g, in operating system): maintains a queue of processes awaiting a slice of machine time.
6. Call center phone systems will use a queue to hold people in line until a service representative is free.
7. Buffers on MP3 players and portable CD players, iPod playlist. Playlist for jukebox - add songs to the end, play from the front of the list.

8. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
9. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Reference
1. V.V. Das; "*Principles of Data Structures Using C and C++";* New Age International Publishers, New Delhi, India.
2. Y Langsam, MJ, Augenstein and A.M, Tanenbaum; "*Data Structure Using C and C++*"; Prentice Hall India.

CHAPTER 4: STATIC AND DYNAMIC LIST

4.1. STATIC IMPLEMENTATION OF LIST

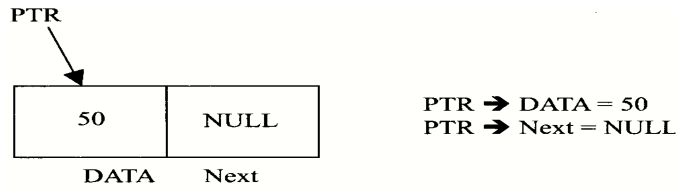
Static implementation can be implemented using arrays. It is very simple method but it has Static implementation. Once a size is declared, it cannot be change during the program execution. It is also *not efficient for memory utilization*. When array is declared, memory allocated is equal to the size of the array. The vacant space of array also occupiers the memory space. In both cases, if we store fewer arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded. It is suitable only when exact numbers of elements are to be stored.

4.2. DYNAMIC IMPLEMENTATION OF LIST

In static implementation of memory allocation, we cannot alter (increase or decrease) the size of an array and the memory allocation is fixed. So we have to adopt an alternative strategy to allocate memory only when it is required. There is a special data structure called linked list that provides a more flexible storage system and it does not required the use of array. *The advantage of a list over an array occurs when it is necessary to insert or delete an element in the middle of a group of other elements.*

LINKED LIST

A linked list is a linear collection of specially designed data structure, called **nodes**, linked to one another by means of **pointer**. Each node is divided into 2 parts: the first part contains information of the element and the second part contains address of the next node in the link list.



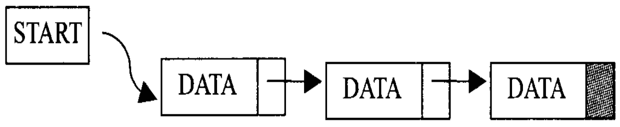


Fig1: Node Fig2: Linked List

Above Fig2 shows that the schematic diagram of a linked list with 3 nodes. Each node is pictured with two parts. The left part of each node contains the data items and the right part represents the address of the next node. The next pointer of the last node contains a special value, called the **NULL** pointer, which does not point to any address of the node. That is NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START =NULL if there is no list (i.e.; NULL list or empty list).

Representation of Linked List

Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as

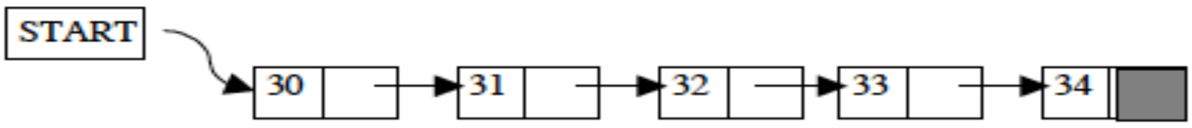


Fig3: Linked list representation

We can declare linear linked list as follows

```
struct Node{
   int data; //instead of data we also use info
   struct Node *Next; //instead of Next we also use link
};
typedef struct Node *NODE;
```

Advantages and Disadvantages of Linked List

Linked list have many advantages and some of them are:

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (or removed) when it is not needed.
3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked list.

Linked list has following disadvantages

1. More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

Operations on Linked List (Primitive Operations)

The primitive operations performed on Linked list is as follows

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching
6. Find Previous
7. Concatenation

Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

i. At the beginning of the linked list
ii. At the end of the linked list
iii. At any specified position in between in a linked list

Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the

i. Beginning of a linked list
ii. End of a linked list
iii. Specified location of the linked list

Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.

Concatenation is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1) th node in A. After concatenation A will contain (n+m) nodes

<u>Types of Linked List</u>

Following are the types of Linked list depending upon the arrangements of the nodes.

1. Singly Linked List
2. Doubly linked List
3. Circular Linked List
    a. Circular Singly Linked List
    b. Circular Doubly Linked List

4.2.1. SINGLY LINKED LIST

All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure. The following figure explains the different operations on Singly Linked List.

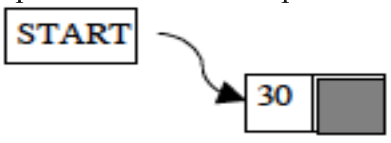


Fig4: Create a node with Data (30)

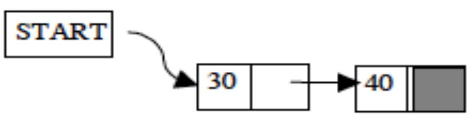
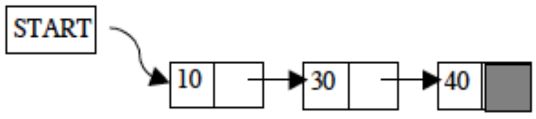


Fig5: Insert a node with Data (40) at the end



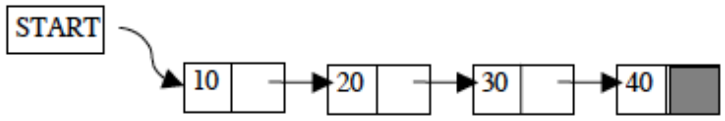
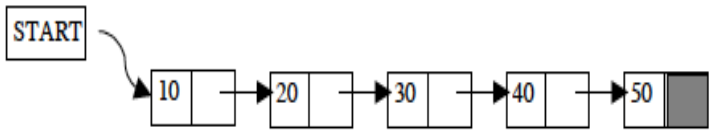


Fig6: Insert a node with Data (10) at the beginning Fig7: Insert a node with Data (20) at 2nd position



Output → 10, 20, 30, 40, 50

Fig8: Insert a node with Data (50) at the end Fig9: Traversing the nodes from left to right

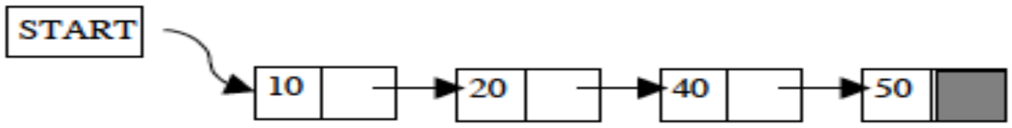


Fig10: Delete 3rd node from the

4.2.1.1. Algorithm For Inserting A Node

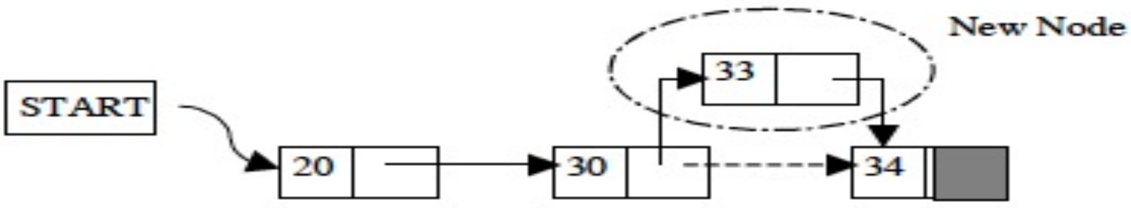


Fig11: Insertion of New Node at specific position

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the new node is to be inserted. TEMP is a temporary pointer to hold the node address.

1) Insert a Node at the beginning of Linked List

```
1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. If (SATRT equal to NULL)
       (a) NewNode → next = NULL
5. Else
       (a) NewNode → next = START
6. START = NewNode
7. Exit
```

2) Insert a Node at the end of Linked List

```
1. Input DATA to be inserted
```

```
2. Create a NewNode
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
8. If (SATRT equal to NULL)
      (a) START = NewNode
9. Else
      (a) TEMP = START
      (b) While (TEMP → Next not equal to NULL)
      (i)      TEMP = TEMP → Next
10. TEMP → Next = NewNode
11. Exit
```

3) Insert a Node at any specified position of Linked List

```
1. Input DATA and POS to be inserted
2. Initialize TEMP = START; and j = 0
3. Repeat the step 3 while (k is less than POS)
      (a) TEMP = TEMP->Next
      (b) If (TEMP is equal to NULL)
            i. Display "Node in the list less than the position"
           ii. Exit
      (c) k = k + 1
4. Create a New Node
5. NewNode → DATA = DATA
6. NewNode → Next = TEMP → Next
7. TEMP → Next = NewNode
8. Exit
```

<u>4.2.1.2 Algorithm For Display All Nodes</u>

Suppose START is the address of the first node in the linked list. Following algorithm will visit all nodes from the START node to the end.

```
1. If (START is equal to NULL)
      (a) Display "The list is Empty"
      (b) Exit
2. Initialize TEMP = START
3. Repeat the step 4 and 5 until (TEMP == NULL)
4. Display "TEMP → DATA"
5. TEMP = TEMP → Next
6. Exit
```

Program (Single Linked List Insertion Operation)

```c
#include<stdio.h>
#include <stdlib.h>
struct node
{
    int data; //instead of data we can use info
    struct node *next;
};
typedef struct node NODE;
NODE *start;
void createmptylist(NODE *start)
{
    start = (NODE*)malloc(sizeof(NODE)); //start = NULL;

}
```

```
void insert_at_begin(int item)
{
    NODE *ptr;
    ptr=(NODE *)malloc(sizeof(NODE));
    ptr->data=item;
    if(start==(NODE *)NULL)
        ptr->next=(NODE *)NULL;
    else
        ptr->next=start;
    start=ptr;
}

void insert_at_end(int item)
{
    NODE *ptr,*loc;
    ptr=(NODE *)malloc(sizeof(NODE));
    ptr->data=item;
    ptr->next=NULL;
    if(start==NULL)
        start=ptr;
    else
    {
        loc=start;
        while(loc->next!=(NODE *)NULL)
            loc=loc->next;
        loc->next=ptr;
    }
}
void insert_spe(int item,int pos)
{
    struct node *tmp,*q;
    int i;
    q=start;
    //Finding the position to add new element to the linked list
    for(i=0;i<pos-1;i++)
    {
        q=q->next;
        if(q==NULL)
        {
            printf ("\n\n There are less than %d elements",pos);
            return;
        }
    }/*End of for*/
    tmp=(struct node*)malloc(sizeof (struct node));
    tmp->next=q->next;
    tmp->data=item;
    q->next=tmp;
}
//This function will display all the element(s) in the linked list
void traverse(NODE *start)
{
    NODE *tmp;
     if(start == NULL)
     {
           printf("List is empty\n");
           return;
    }
    tmp = start;
    while(tmp != NULL)
    {
        printf("%d\n",tmp->info);
        tmp=tmp->next;
    }
}

int main()
{
    int choice,item,pos;
        createmptylist(start);
    while(1)
    {
        printf("1. Insert element at begin \n");
        printf("2. Insert element at end positon\n");
        printf("3. Insert at specific position\n");
        printf("4. Travers the list in order\n");
        printf("5. Exit\n");
        printf(" Enter your choice\n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: printf("Enter the item\n");
                scanf("%d",&item);
                insert_at_begin(item);
                break;
            case 2: printf("Enter the item\n");
                scanf("%d",&item);
                insert_at_end(item);
                break;
            case 3: printf("Enter the item and pos\n");
                scanf("%d%d",&item,&pos);
                insert_spe(item,pos);
                break;
            case 4: printf("\ntravers the list\n");
                traverse(start);
                break;
            case 5:
                exit(0);
        }
    }
   } //End of main()
```

4.2.1.3 Algorithm For Deleting A Node

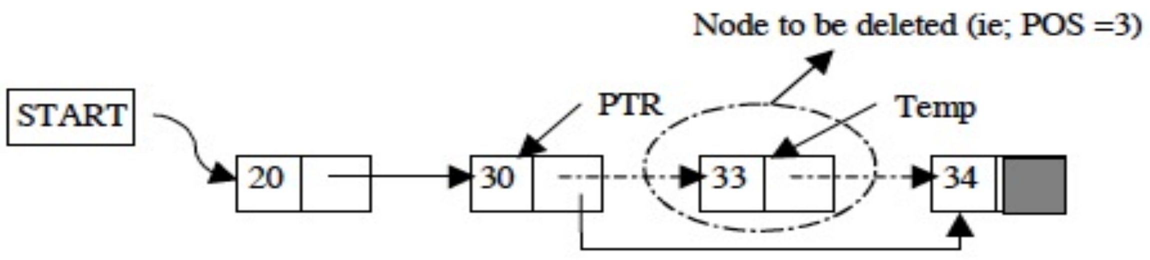


Fig12: Deletion of Node

Suppose START is the first position in linked list. Let DATA be the element to be deleted. TEMP, HOLD is a temporary pointer to hold the node address.

```
1. Input the DATA to be deleted
2. If (START → DATA is equal to DATA) // Data at first node
        (a) TEMP = START
        (b) START = START → Next
        (c) Set free the node TEMP, which is deleted
        (d) Exit
3. HOLD = START
4. while ((HOLD → Next → Next) not equal to NULL))
        (a) if ((HOLD → NEXT → DATA) equal to DATA)
            (i)   TEMP = HOLD → Next
            (ii)  HOLD → Next = TEMP → Next
            (iii) Set free the node TEMP, which is deleted
            (iv)  Exit
        (b) HOLD = HOLD → Next
5. if ((HOLD → next → DATA) == DATA)
        (a) TEMP = HOLD → Next
        (b) Set free the node TEMP, which is deleted
        (c) HOLD → Next = NULL
        (d) Exit
6. Disply "DATA not found"
7. Exit
```

Program: Deleting a Node from single linked list

```
//Delete any element from the linked list
void Delete(int item)
{
    struct node *tmp,*q;
    if (start->info == data)
    {
        tmp=start;
        start=start->next; /*First element deleted*/
        free(tmp);
        return;
    }
    q=start;
    while(q->next->next ! = NULL)
    {
        if(q->next->data == item) /*Element deleted in between*/
        {
            tmp=q->next;
            q->next=tmp->next;
            free(tmp);
            return;
        }
        q=q->next;
    }/*End of while */
    if(q->next->data==info) /*Last element deleted*/
    {
        tmp=q->next;
        free(tmp);
        q->next=NULL;
        return;
    }
    printf ("\n\nElement %d not found",data);
}/*End of del()*/
```

4.2.1.4. Algorithm For Searching A Node

Suppose START is the address of the first node in the linked list and DATA is the information to be searched. After searching, if the DATA is found, POS will contain the corresponding position in the list.

```
1. Input the DATA to be searched
2. Initialize TEMP = START; POS =1;
3. Repeat the step 4, 5 and 6 until (TEMP is equal to NULL)
4. If (TEMP → DATA is equal to DATA)
       (a) Display "The data is found at POS"
       (b) Exit
5. TEMP = TEMP → Next
6. POS = POS+1
7. If (TEMP is equal to NULL)
       (a) Display "The data is not found in the list"
8. Exit
```

Code: Searching an element

```
//Function to search an element from the linked list
void Search(int item)
{
    struct node *ptr = start;
    int pos = 1;
    //searching for an element in the linked list
    while(ptr!=NULL)
    {
        if (ptr->data==item)
        {
            printf ("\n\nItem %d found at position %d", data, pos);
            getch();
            return;
        }
        ptr = ptr->next;
        pos++;
    }
    if (ptr == NULL)
        printf ("\n\nItem %d not found in list",data);
}
```

4.2.2. Stack Using Linked List

The following figure shows that the implementation of stack using linked list.

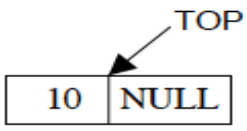


Fig13: Push (10)

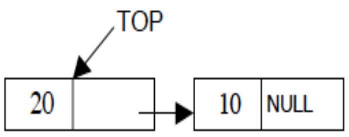


Fig14: Push (20)

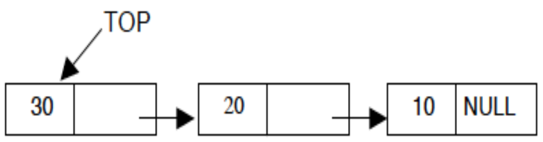
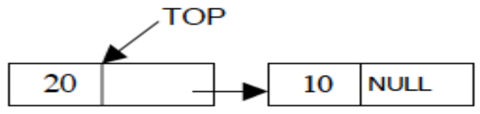


Fig15: Push (30)



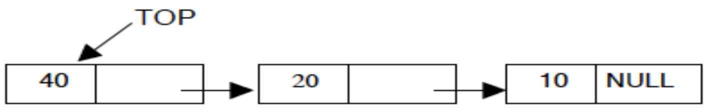


Fig16: X = Pop (i.e. X=30) Fig17: Push (40)

4.2.2.1 Algorithm For Push Operation

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

```
1. Input the DATA to be pushed
2. Creat a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = TOP
5. TOP = NewNode
6. Exit
```

4.2.2.2. Algorithm For POP Operation

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

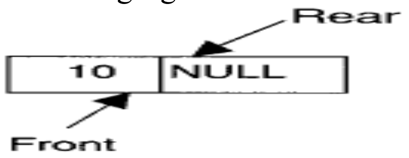
```
1. if (TOP is equal to NULL)
      (a) Display "The stack is empty"
2. Else
      (a) TEMP = TOP
      (b) Display "The popped element TOP → DATA"
      (c) TOP = TOP → Next
      (d) TEMP → Next = NULL
      (e) Free the TEMP node
3. Exit
```

Program: Push, Pop and Display using Linked list on Stack

```
#include<stdio.h>
#include<stdlib.h>
//Structure is created a node
struct node
{
    int data;
    struct node *next;//A link to the next node
};
//A variable named NODE is been defined for the structure
typedef struct node *NODE;
//This function is to perform the push operation
NODE push(NODE top)
{
    NODE NewNode;
    int pushed_item;
    //A new node is created dynamically
    NewNode = (NODE)malloc(sizeof(NODE));
    printf("\nInput the new value to be pushed on the stack:");
    scanf("%d",&pushed_item);
    NewNode->data=pushed_item;//Data is pushed to the stack
    NewNode->next=top;//Link pointer is set to the next node
    top=NewNode;//Top pointer is set
    return(top);
}/*End of push()*/
//Following function will implement the pop operation
NODE pop(NODE top)
{
    NODE tmp;
    if(top == NULL)//checking whether the stack is empty or not
        printf ("\nStack is empty\n");
    else
    {
        tmp=top;//popping the element
        printf("\nPopped item is %d\n",tmp->data);
        top=top->next;//resetting the top pointer
        tmp->next=NULL;
        free(tmp);//freeing the popped node
    }
    return(top);
}/*End of pop()*/
//This is to display the entire element in the stack
void display(NODE top)
{
    NODE tmp;
    if(top==NULL)
        printf("\nStack is empty\n");
    else
    {   tmp = top;
        printf("\nStack elements:\n");
        while(tmp != NULL)
        {
            printf("%d\n",tmp->data);
            tmp = tmp->next;
        }/*End of while */
    }/*End of else*/
}/*End of display()*/
int main()
{
    char opt;
    int choice;
    NODE Top=NULL;
    while(1){
        printf("\n1.PUSH\n");
        printf("2.POP\n");
        printf("3.DISPLAY\n");
        printf("4.EXIT\n");
        printf("\nEnter your choice:");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                Top=push(Top);
                break;
            case 2:
                Top=pop(Top);
                break;
            case 3:
                display(Top);
                break;
            case 4:
                exit(1);
            default:
                printf("\nWrong choice\n");
        }/*End of switch*/
    }
}/*End of main() */
```

4.2.3. Queue Using Linked List

The following figure shows that the implementation issues of Queue using linked list.



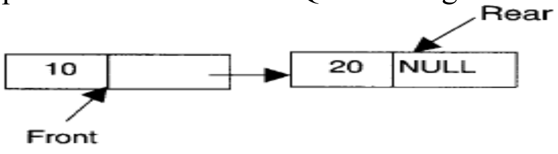
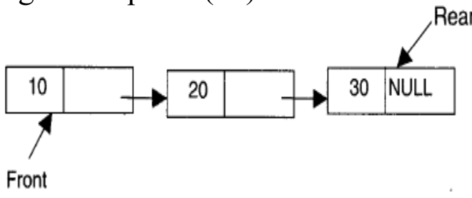


Fig18: Enqueue (10) Fig19: Enqueue (20)



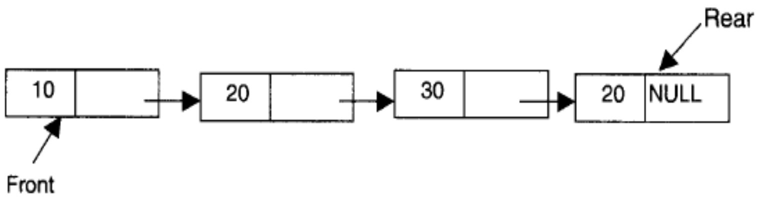


Fig20: Enqueue (30) Fig21: Enqueue (20)

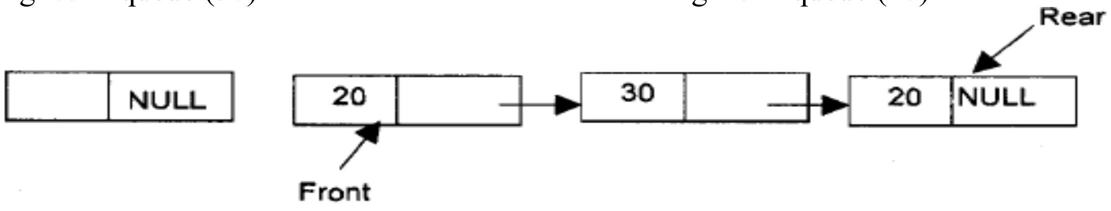


Fig22: X = Dequeue (i.e. X = 10)

4.2.3.1. Algorithm for pushing an element into a Queue

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

```
1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If(REAR not equal to NULL)
       (a) REAR → next = NewNode;
6. REAR =NewNode;
7. Exit
```

4.2.3.2. Algorithms for popping an element from the queue

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

```
1. If (FRONT is equal to NULL)
        (a) Display "The Queue is empty"
2. Else
        (a) Display "The popped element is FRONT → DATA"
        (b) If (FRONT is not equal to REAR)
            (i) FRONT = FRONT → Next
        (c) Else
        (d) FRONT = NULL;
3. Exit
```

Code: Queue using Linked List (Enqueue, Dequeue and Traverse)

```
//A structure is created for the node in queue
#include <stdio.h>
#include <stdlib.h>
struct queue
{
    int data;
    struct queue *next;//Next node address
};
typedef struct queue *NODE;
//This function will Enqueue an element into the queue
NODE Enqueue(NODE rear)
{
    NODE NewNode;
    //New node is created to push the data
    NewNode=(NODE)malloc(sizeof(struct queue));
    printf ("\nEnter the no to be pushed = ");
    scanf ("%d",&NewNode->data);
    NewNode->next=NULL;
    //setting the rear pointer
    if (rear != NULL)
        rear->next=NewNode;
    rear=NewNode;
    return(rear);
}
//This function will Dequeue the element from the queue
NODE Dequeue(NODE f,NODE r)
{
    //The Queue is empty when the front pointer is NULL
    if(f==NULL)
        printf ("\nThe Queue is empty");
    else
    {
        printf ("\nThe poped element is = %d\n",f->data);
        if(f != r)
            f=f->next;
        else
            f=NULL;
    }
    return(f);
}
//Function to display the element of the queue
void Traverse(NODE fr,NODE re)
{
    //The queue is empty when the front pointer is NULL
    if (fr==NULL)
```

```
        printf ("\nThe Queue is empty");
    else
    {
        printf ("\nThe element(s) is/are = ");
        while(fr != re)
        {
            printf("%d ",fr->data);
            fr=fr->next;
        };
        printf ("%d",fr->data);
        printf("\n");
    }
}

int main()
{
    int choice;
    //declaring the front and rear pointer
    NODE front, rear;
    //Initializing the front and rear pointer to NULL
    front = rear = NULL;
    while(1)
    {

        printf ("1. Enqueue\n");
        printf ("2. Dequeue\n");
        printf ("3. Traverse\n");
        printf ("4. Exit\n");
        printf ("\n\nEnter your choice = ");
        scanf ("%d",&choice);
        switch(choice)
        {   case 1:
                //calling the Enqueue function
                rear = Enqueue(rear);
                if (front==NULL)
                {
                    front=rear;
                }
                break;
            case 2:
                //calling the Dequeue function by passing
                //front and rear pointers
                front = Dequeue(front,rear);
                if (front == NULL)
                    rear = NULL;
                break;
            case 3:
                Traverse(front,rear);
                break;
            case 4:
                exit(0);
            default:
                printf("Try Again\n");
        }
    }
}
```

Advantages of Singly Linked List

1. Accessibility of a node in the forward direction is easier.
2. Insertion and deletion of node are easier.

Disadvantages of Singly Linked List

1. Can insert only after a referenced node
2. Removing node requires pointer to previous node
3. Can traverse list only in the forward direction

4.2.4. DOUBLY LINKED LIST

A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every nodes in the doubly linked list has three fields: LeftPointer (Prev), RightPointer (Next) and DATA.

Prev	DATA	Next

Fig23: Typical Doubly Linked list node

Prev will point to the node in the left side (or previous node) i.e. **Prev** will hold the address of the previous node. **Next** will point to the node in the right side (or next node), i.e. **Next** will hold the address of next node. DATA will store the information of the node.

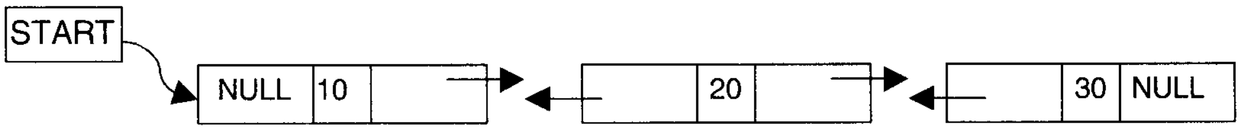


Fig24: Doubly Linked List

Representation of Doubly Linked List

Following declaration can represent doubly linked list

```
struct Node
{
    int data;
    struct Node *Next;
    struct Node *Prev;
};
typedef struct Node *NODE;
```

All the primitive operations performed on singly linked list can also be performed on doubly linked list. The following figures shows that the insertion and deletion of nodes.

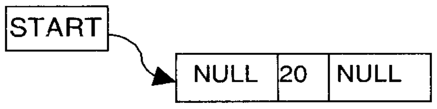


Fig25: Add (20)

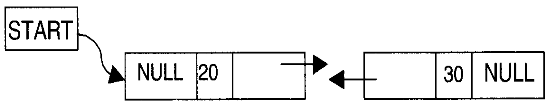


Fig26: Insert (30) at the end

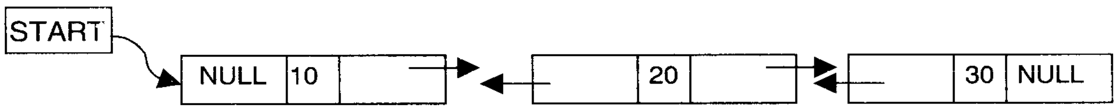


Fig27: Insert (10) at the Beginning

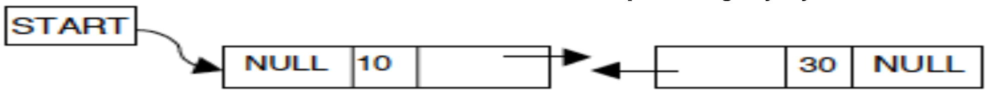


Fig28: Delete a node at the 2nd position (Delete 20 at 2nd position)

4.2.4.1. Algorithm for Inserting a Node

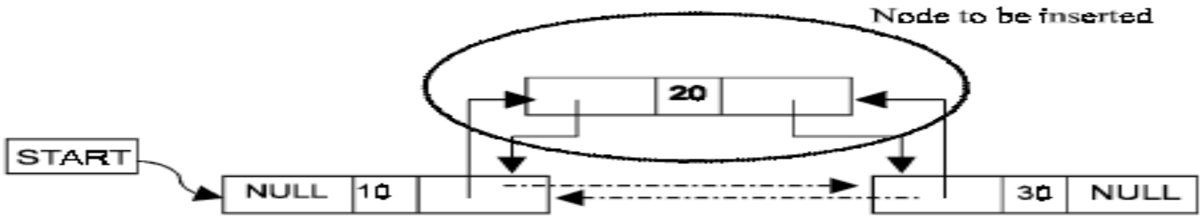


Fig29: Insert a node at the 2nd position

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the NewNode is to be inserted. TEMP is a temporary pointer to hold the node address.

```
1.  Input the DATA and POS
2.  Initialize TEMP = START; i = 0
3.  Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4.  TEMP = TEMP → Next; i = i +1
5.  If (TEMP not equal to NULL) and (i equal to POS)
        a) Create a New Node
        b) NewNode → DATA = DATA
        c) NewNode → Next = TEMP → Next
        d) NewNode → Prev = TEMP
        e) (TEMP → Next) → Prev = NewNode
        f) TEMP → Next = New Node
6.  Else
        a) Display "Position NOT found"
7.  Exit
```

4.2.4.2. Algorithm for Deleting a Node

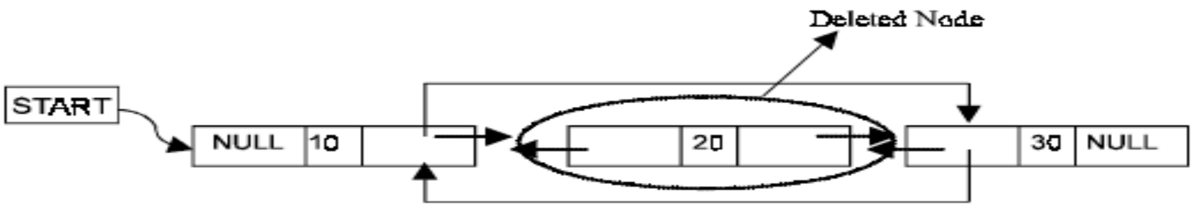


Fig30: Delete a node at the 2nd position

Suppose START is the address of the first node in the linked list. Let POS is the position of the node to be deleted. TEMP is the temporary pointer to hold the address of the node. After deletion, DATA will contain the information on the deleted node.

```
1. Input the POS
2. Initialize TEMP = START; i = 0
3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
```

```
4. TEMP = TEMP → Next; i = i +1
5. If (TEMP not equal to NULL) and (i equal to POS)
    a. Create a New Node
    b. NewNode → DATA = DATA
    c. NewNode → Next = TEMP → Next
    d. NewNode → Prev = TEMP
    e. (TEMP → Next) → Prev = NewNode
    f. TEMP → Next = New Node
6. Else
    a. Display “Position NOT found”
7. Exit
```

Source code to Implement the primitives operations on Doubly Linked List

```
//Structure is created for the node
struct node
{
    struct node *prev;
    int data;
    struct node *next;
};
typedef struct node *NODE;
NODE *start;
//fucntion to create a doubly linked list
void create_list(
{

    //a new node is created
    start=(NODE)malloc(sizeof(struct node));
    start->next=NULL;
    start->prev=NULL;
}/*End of create_list()*/
//Function to add new node at the beginning
void addatbeg(int num)
{
    NODE tmp;
    //a new node is created for inserting the data
    tmp=(NODE)malloc(sizeof(struct node));
    tmp->prev=NULL;
    tmp->data=num;
    tmp->next=start;
    start->prev=tmp;
    start=tmp;
}/*End of addatbeg()*/
//This fucntion will insert a node in any specific position
void addafter(int num,int pos)
{
    NODE tmp,q;
    int i;
    q=start;
    //Finding the position to be inserted
    for(i=0;i<pos-1;i++)
    {
        q=q->next;
        if(q==NULL)
        {
            printf ("\nThere are less than %d elements\n",pos);
            return;
        }
    }
    //a new node is created
    tmp=(NODE)malloc(sizeof(struct node) );
    tmp->data=num;
    q->next->prev=tmp;
    tmp->next=q->next;
    tmp->prev=q;
    q->next=tmp;
}/*End of addafter() */
```

//Function to delete a node

```
void del(int num)
{
    NODE tmp,q;
    if(start->data==num)
    {
        tmp=start;
        start=start->next; /*first element deleted*/
        start->prev = NULL;
        free(tmp);//Freeing the deleted node
        return;
    }
    q=start;
    while(q->next->next!=NULL)
    {
        if(q->next->data==num) /*Element deleted in between*/
        {
            tmp=q->next;
            q->next=tmp->next;
            tmp->next->prev=q;
            free(tmp);
            return;
        }
        q=q->next;
    }
    if (q->next->data==num) /*last element deleted*/
    { tmp=q->next;
        free(tmp);
        q->next=NULL;
        return;
    }
    printf("\nElement %d not found\n",num);
}/*End of del()*/
```

//Displaying all data(s) in the node

```
void display()
{
    NODE q;
    if(start==NULL)
    {
        printf("\nList is empty\n");
        return;
    }
    q=start;
    printf("\nList is :\n");
    while(q!=NULL)
    {
        printf("%d ", q->data);
        q=q->next;
    }
    printf("\n");
}/*End of display() */
```

4.2.5. CIRCULAR LINKED LIST

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node. Circular linked lists also make our implementation easier, because they eliminate the boundary conditions associated with the beginning and end of the list, thus eliminating the special case code required to handle these boundary conditions.

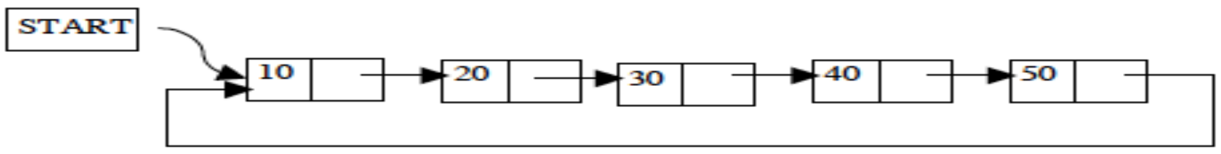


Fig31: Circular Linked List

A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner as shown in the following Fig.

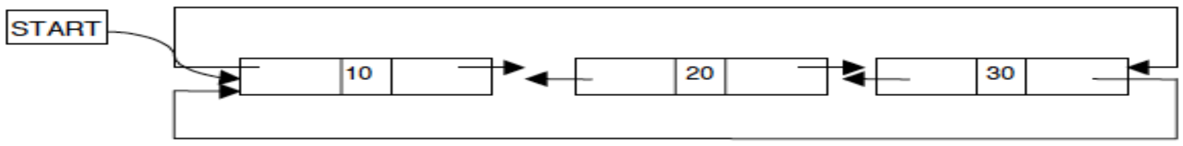


Fig32: Circular Doubly Linked List

4.2.5.1. Algorithm

1. Inserting a node at the beginning (*Insert_First(START,Item*)

```
1. Create a Newnode
2. IF START is equal to NULL then
        a. Set Newnode->data = item
        b. Set NewNode->next = Newnode
        c. Set START = Newnode
        d. Set Last = Newnode
3. Set Newnode ->data = Item
4. Set Newnode->next = Start
5. Set START = Newnode
6. Set last->next = Newnode
```

2. Inserting a node at the End (*Insert_End(START,Item*)

```
1. Create a Newnode
2. If START is equal to NULL, then
        a. Newnode->data = Item
        b. Newnide->next = Newnode
        c. Last = Newnode
        d. START = Newnode
3. Newnode->data = Item
4. Last->next = Newnode
5. Last = Newnode
6. Last->next = START
```

3. Deleting a node from beginning (*Delete_First(START*)

```
1. Declare a temporary node, ptr
2. If START is equal to NULL, then
      a. Display empty Circular queue
      b. Exit
3. Ptr = START
4. START = START->next
5. Print, Element deleted is , ptr->data
6. Last->next = START
7. Free (ptr)
```

4. Deleting a Node from End (*Delete_End(START*)

```
1. Declare a temporary node , ptr
2. If START is equal to NULL , then
      a. Print Circular list empty
      b. Exit
3. Ptr = START
4. Repeat step 5 and 6 until ptr !=Last
5. Ptr1 = ptr
6. Ptr = ptr->next
7. Print element deleted is ptr->data
8. Ptr1->next = ptr->next
9. Last = ptr1
10. Return START
```

Program:

```
/*Program to demonstrate circular singly linked list. This program will
add an element at the beginning of Linked list. */
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};
struct node *head = NULL, *tail = NULL;

struct node * createNode(int item) {
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof (struct node));
    newnode->data = item;
    newnode->next = NULL;
    return newnode;
}

/* create dummy head and tail to make insertion and deletion operation
simple.While processing data in our circular linked list. */

void createDummies() {
    head = createNode(0);
    tail = createNode(0);
    head->next = tail;
    tail->next = head;
}
```

```
/* insert data next to dummy head */
void circularListInsertion(int data) {
    struct node *newnode, *temp;
    newnode = createNode(data);
    temp = head->next;
    head->next = newnode;
    newnode->next = temp;
}

/* Delete the node that has the given data */
void DeleteInCircularList(int data) {
    struct node *temp0, *temp;
    if (head->next == tail && tail->next == head) {
        printf("Circular Queue/list is empty\n");
    }
    temp0 = head;
    temp = head->next;
    while (temp != tail) {
        if (temp->data == data) {
            temp0->next = temp->next;
            free(temp);
            break;
        }
        temp0 = temp;
        temp = temp->next;
    }
    return;
}

/* travese the circular linked list. */
void traverse() {
    int n = 0;
    struct node *tmp = head->next;
    if (head->next == tail && tail->next == head) {
        printf("Circular linked list is empty\n");
        return;
    }
    while (tmp !=tail){
            printf("%d->", tmp->data);
        tmp = tmp->next;
    }
    return;
}


int main() {
    int data, ch;
    createDummies();
    while (1) {
        printf(" \n1.Insertion\t2. Deletion\n");
        printf("3. Display\t4. Exit\n");
        printf("Enter ur choice:");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter the data to insert:");
                scanf("%d", &data);
                circularListInsertion(data);
                break;

            case 2:
                printf("Enter the data to delete:");
                scanf("%d", &data);
                DeleteInCircularList(data);
                break;

            case 3:
                traverse();
                break;

            case 4:
                exit(0);

            default:
                printf("Pls. enter correct option\n");
                break;
        }
    }
    return 0;
}
```

4.2.6. Polynomials using singly linked list (Application of Linked List)

Different operations, such as addition, subtraction, division and multiplication of polynomials can be performed using linked list. Following example shows that the polynomial addition using linked list.

In the linked representation of polynomials, each term is considered as a node. And such a node contains three fields: *coefficient field, exponent field and Link field.*

```
Struct polynode{
      int coeff;
      int expo;
      struct polynode *next;
      };
```

Consider two polynomials **f**(**x**) and **g**(**x**); it can be represented using linked list as follows.

$$f(x) = ax^3 + bx + c$$
$$g(x) = mx^4 + nx^3 + ox^2 + px + q$$

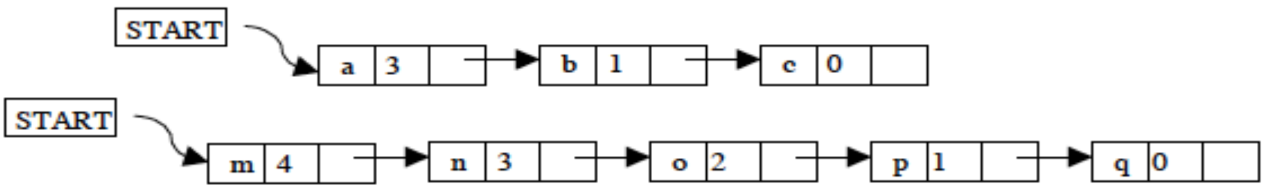


Fig33: Polynomial Representation in Linked List

These two polynomials can be added by

$h(x) = f(x) + g(x) = mx^4 + (a + n) x^3 + ox^2 + (b + p)x + (c + q)$

i.e.; adding the constants of the corresponding polynomials of the same exponentials. **h(x)** can be represented as

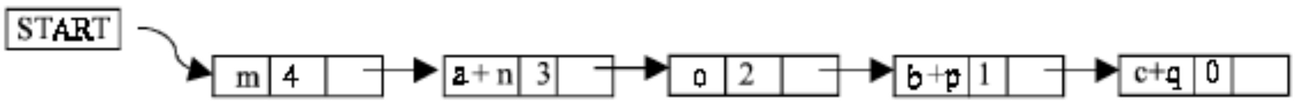


Fig34: Addition of polynomials

Steps involved in adding two polynomials

1. Read the number of terms in the first polynomial, f(x).
2. Read the coefficient and exponents of the first polynomial.
3. Read the number of terms in the second polynomial g(x).
4. Read the coefficients and exponents of the second polynomial.
5. Set the temporary pointers p and q to traverse the two polynomial respectively.
6. Compare the exponents of two polynomials starting from the first nodes.
   a. If both exponents are equal then add the coefficients and store it in the resultant linked list.
   b. If the exponent of the current term in the first polynomial p is less than the exponent of the current term of the second polynomial is added to the resultant linked list. And move the pointer q to point to the next node in the second polynomial Q.
   c. If the exponent of the current term in the first polynomial p is greater than the exponent of the current term in the second polynomial Q then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.
   d. Append the remaining nodes of either of the polynomials to the resultant linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
//structure is created for the node
struct node
{
    int coef;
    int expo;
    struct node *link;
};
typedef struct node *NODE;
//Function to add any node to the linked list
NODE insert(NODE start,float co,int ex)
{
    NODE ptr,tmp;
    //a new node is created
    tmp= (NODE)malloc(sizeof(struct node));
    tmp->coef=co;
    tmp->expo=ex;
    /*list empty or exp greater than first one */
    if(start==NULL || ex>start->expo)
    {
        tmp->link=start;//setting the start
        start=tmp;
    }
    else
    {
        ptr=start;
        while(ptr->link!=NULL && ptr->link->expo>ex)
            ptr=ptr->link;
        tmp->link=ptr->link;
        ptr->link=tmp;
        if(ptr->link==NULL) /*item to be added in the end */
            tmp->link=NULL;
    }
    return start;
}/*End of insert()*/
//This function is to add two polynomials
NODE poly_add(NODE p1,NODE p2)
{
    NODE p3_start,p3,tmp;
    p3_start=NULL;
    if(p1==NULL && p2==NULL)
        return p3_start;
    while(p1!=NULL && p2!=NULL )
    {
        //New node is created
        tmp=(NODE)malloc(sizeof(struct node));
        if(p3_start==NULL)
        {
            p3_start=tmp;
            p3=p3_start;
        }
        else
        {
            p3->link=tmp;
            p3=p3->link;
        }
        if(p1->expo > p2->expo)
        {
            tmp->coef=p1->coef;
            tmp->expo=p1->expo;
            p1=p1->link;
        }
        else
            if(p2->expo > p1->expo){
                tmp->coef=p2->coef;
                tmp->expo=p2->expo;
                p2=p2->link;
            }
            else
                if(p1->expo == p2->expo)
                {
                    tmp->coef=p1->coef + p2->coef;
                    tmp->expo=p1->expo;
                    p1=p1->link;
                    p2=p2->link;
                }
    }/*End of while*/
    while(p1!=NULL)
    {
        tmp=(NODE)malloc(sizeof(struct node));
        tmp->coef=p1->coef;
        tmp->expo=p1->expo;
        if (p3_start==NULL) /*poly 2 is empty*/
        {
            p3_start=tmp;
            p3=p3_start;
        }
        else
        {
            p3->link=tmp;
            p3=p3->link;
        }
        p1=p1->link;
    }/*End of while */
    while(p2!=NULL)
    {
        tmp=(NODE)malloc(sizeof(struct node));
        tmp->coef=p2->coef;
        tmp->expo=p2->expo;
        if (p3_start==NULL) /*poly 1 is empty*/
        {p3_start=tmp;
            p3=p3_start;
        }
        else{
            p3->link=tmp;
            p3=p3->link;
        }
        p2=p2->link;
    }/*End of while*/
    p3->link=NULL;
    return p3_start;
}/*End of poly_add() */
//Inputting the two polynomials
NODE enter(NODE start)
{
    int i,n,ex;
    int co;
    printf("\nHow many terms u want to enter:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        printf("\nEnter coeficient for term %d:",i);
        scanf("%d",&co);
        printf("Enter exponent for term %d:",i);
        scanf("%d",&ex);
        start=insert(start,co,ex);
    }
    return start;
}/*End of enter()*/
//This function will display the two polynomials and its
//added polynomials
void display(NODE ptr)
{
    if (ptr==NULL)
    {
        printf ("\nEmpty\n");
        return;
    }
    while(ptr!=NULL)
    {
        printf ("%dx^%d ", ptr->coef,ptr->expo);
        ptr=ptr->link;
        if(ptr != NULL)
            printf(" +");
    }
}/*End of display()*/
int main()
{
    NODE p1_start,p2_start,p3_start;
    p1_start=NULL;
    p2_start=NULL;
    p3_start=NULL;
    printf("\nPolynomial 1 :\n");
    p1_start=enter(p1_start);
    printf("\nPolynomial 2 :\n");
    p2_start=enter(p2_start);
    //polynomial addition function is called
    p3_start=poly_add(p1_start,p2_start);

    printf("\nPolynomial 1 is:");
    display(p1_start);
    printf ("\nPolynomial 2 is:");
    display(p2_start);
    printf ("\nAdded polynomial is:");
    display(p3_start);
}/*End of main()*/
```

Advantages

1. Polynomials of any degree can be represented.
2. Memory allocated to the nodes only when it is required and deallocated when the nodes are no more required.

Generalized Linked List

Linked list data structure can be viewed as abstract data type which consists of sequence of objects called elements. Associated with each list element is value. We can make distinction between an element, which is an object as part of a list and the elements value, which is the object considered individually. For example, the number 8 may appear on a list twice. Each appearance is a distinct element of the list, but the value of the two elements the number 8 is same. An element may be viewed as corresponding to a node in the linked list representation, where a value corresponds to the node's contents.
There is a convenient notation for specifying abstract general lists. A list may be denoted by a parenthesized enumeration of its elements separated by commas. For example, the abstract list represented by following figure may be represented as: list = (8,16,'g', 99,'b')

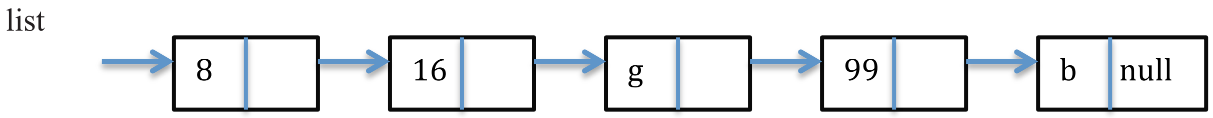


Fig35 (a): Generalized Linked List
The null list is denoted by an empty parenthesis pair such as ( ). Thus the list of the following figure is represented as: list = (8,(5,7,(18,3,6),( ),5),2,(6,8))

list

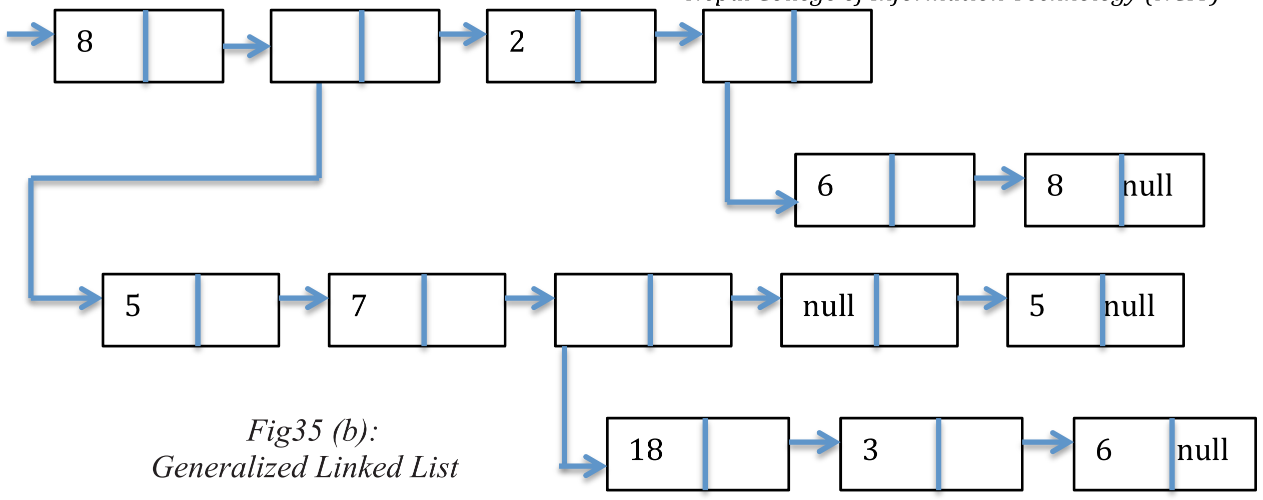


Fig35 (b):
Generalized Linked List

REFERENCES

3. V.V. Das; "*Principles of Data Structures Using C and C++";* New Age International Publishers, New Delhi, India.
4. Y Langsam, MJ, Augenstein and A.M, Tanenbaum; "*Data Structure Using C and C++*"; Prentice Hall India.
5. G.S. Baluja; "*Data Structure Through C (A Practical Approach)*"; Dhanpat Rai & Co, New Delhi, India

Chapter 5: Recursion

5.1. Recursion

Recursion occurs when a function is called by itself repeatedly. The general algorithm model for any recursive function contains the following steps:

1. *Prologue:* Save the parameters, local variables, and return address.
2. *Body:* If the base criterion has been reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1 (initiate a recursive call).
3. *Epilogue:* Restore the most recently saved parameters, local variables, and return address.

The following **figure1** shows that the flow-chart model for a recursive algorithm. The key box in the flowchart contained in the body of the function is the one, which invokes a call to itself. Each time a function call to itself is executed, the prologue of the function saves necessary information required for its proper functioning. The Last-in-First-Out characteristics of a recursive function points that the stack is the most obvious data structure to implement the recursive function. Programs compiled in modern high-level languages (even C) make use of a stack for the procedure or function invocation in memory. When any procedure or function is called, a number of words (such as variables, return address and other arguments and its data(s) for future use) are pushed onto the program stack. When the procedure or function returns, this frame of data is popped off the stack.

As a function calls a function, its arguments, return address and local variables are pushed onto the stack. Since each function runs in its own environment or context, it becomes possible for a function to call itself-a technique known as recursion. The stack is a region of main memory within which programs temporarily store data as they execute. For example, when a program sends parameters to a function, the parameters are placed on the stack. When the function completes its execution these parameters are popped off from the stack. When a function calls other function the current contents (ie.variables) of the caller function are pushed onto the stack with the address of the instruction just next to the call instruction, this is done so after the execution of called function, the compiler can backtrack (or remember) the path from where it is sent to the called function.

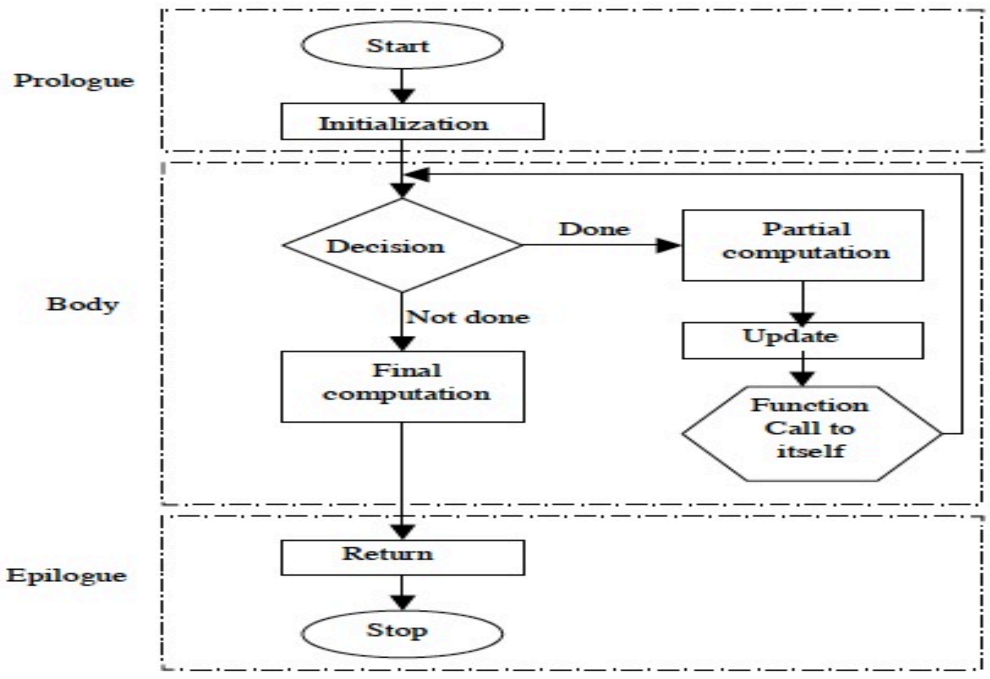


Figure1: Flow chart Model for a recursive algorithm

5.2. Factorial of a number

Following program demonstrate the recursive method for finding a factorial of a number.

```
#include<stdio.h>

void fact(int no, int facto)
{
    if (no <= 1)
    {
        //Final computation and returning and restoring address
        printf("\nThe factorial is: %d ",facto);
        return;
    }
    else
    {
        //Partiial computation of the program
        facto=facto*no;
        //Function call to itself, that is recursion
        fact(--no,facto);
    }
}
int main()
{
    int number,factorial;
    //Initialization of formal parameters, local variables and etc.
    factorial=1;
    printf("\n Enter Number ");
    scanf("%d",&number);
    //Starting point of the function, which calls itself
    fact(number,factorial);
    return 0;
}
```

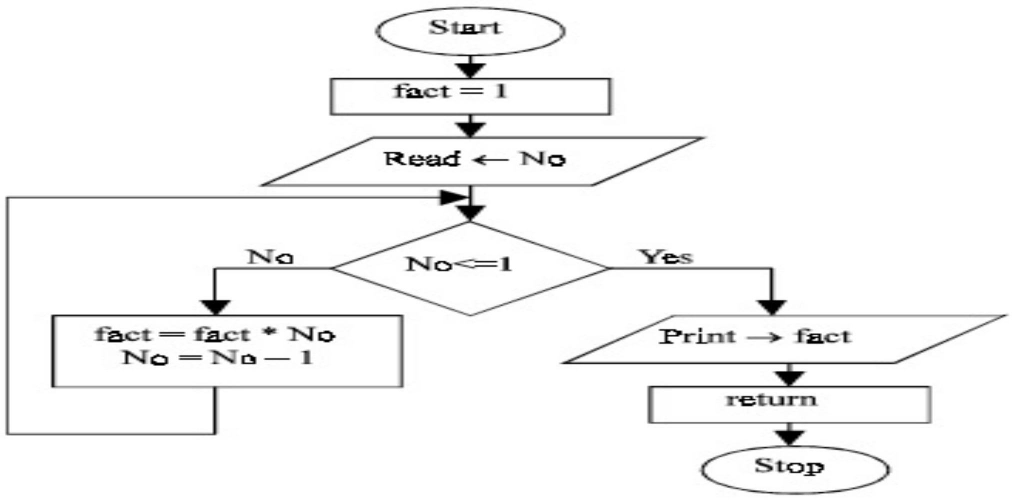


Figure 2: Flowchart for finding factorial recursively.

5.3. Recursion vs Iteration

Recursion of course is an elegant programming technique, but not the best way to solve a problem, even if it is recursive in nature. This is due to the following reasons:
1. It requires stack implementation.
2. It makes inefficient utilization of memory, as every time a new recursive call is made a new set of local variables is allocated to function.
3. Moreover it also slows down execution speed, as function calls require jumps, and saving the current state of program onto stack before jump.

Although recursion provides a programmer with certain pitfalls, and quite sharp concepts about programming. Moreover recursive functions are often easier to implement and maintain, particularly in case of data structures which are by nature recursive. Such data structures are queues, trees, and linked lists. Below are the few key difference point between recursion and iteration

Recursion	**Iteration**
Recursion is the technique of defining anything in terms of itself.	It is a process of executing a statement or a set of statements repeatedly, until some specified condition is satisfied.
There must be an exclusive if statement inside	Iteration involves four clear-cut Steps like

the recursive function, specifying stopping condition.	initialization, condition, execution, and updating.
Not all problems have recursive solution.	Any recursive problem can be solved iteratively.
Recursion is generally a worse option to go for simple problems, or problems not recursive in nature.	Iterative counterpart of a problem is more efficient in terms of memory, utilization and execution speed.

Disadvantages of Recursion

1. It consumes more storage space because the recursive calls along with automatic variables are stored on the stack.
2. The computer may run out of memory if the recursive calls are not checked.
3. It is not more efficient in terms of speed and execution time.
4. According to some computer professionals, recursion does not offer any concrete advantage over non-recursive procedures/functions.
5. If proper precautions are not taken, recursion may result in non-terminating iterations.
6. Recursion is not advocated when the problem can be through iteration. Recursion may be treated as a software tool to be applied carefully and selectively.

5.4. Tower of Hanoi

Let us look at the Tower of Hanoi problem and see how we can use recursive technique to produce a logical and elegant solution. Let us suppose that we have three towers A, B, C and three different sizes disks let say 1, 2 and 3. Each disk has a hole in the center so that it can be stacked on any of the pegs. At the beginning, the disks are stacked on the A peg, that is the largest sized disk on the bottom and the smallest sized disk on top. Here we have to transfer all the disks from source peg A to the destination peg C by using an intermediate peg B. Following are the rules to be followed during transfer:
1. Transferring the disks from the source peg to the destination peg such that at any point of transformation no large size disk is placed on the smaller one.
2. Only one disk may be moved at a time.
3. Each disk must be stacked on any one of the pegs.
Now following could be the steps to solve the tower of Hanoi problem for 3 disks.

```
    Move disk 1 from peg A to peg C
    Move disk 2 from peg A to peg B
    Move disk 1 from peg C to peg B
    Move disk 3 from peg A to peg C
    Move disk 1 from peg B to peg A
    Move disk 2 from peg B to peg C
    Move disk 1 from peg A to peg C
```

We can generalize the solution to the tower of Hanoi problem recursively as follows:
To move n disks from peg A to peg C, Using B as auxiliary peg.
1. If $n = 1$, move the single disk from A to C and stop.
2. Move the *top(n – 1)* disks from the peg A to the peg B, using C as auxiliary.
3. Move *nth* disk to peg C.
4. Now move $n – 1$ disk from B to C, using A as auxiliary.

Below Program simulate the tower of Hanoi problem.

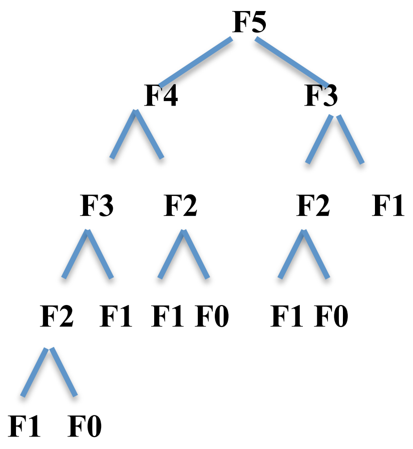
```
#include<stdio.h>
/* Function Prototype */

void towers(int,char,char,char);
void towers(int n,char frompeg,char topeg,char auxpeg)
{ /* If only 1 disk, make the move and return */
    if(n==1)
    {
     printf("\nMove disk 1 from peg %c to peg %c",frompeg,topeg);
        return;
    }
    /* Move top n-1 disks from A to B, using C as auxiliary */
    towers(n-1,frompeg,auxpeg,topeg);
    /* Move remaining disks from A to C */
    printf("\nMove disk %d from peg %c to peg %c",n,frompeg,topeg);
    /* Move n-1 disks from B to C using A as auxiliary */
    towers(n-1,auxpeg,topeg,frompeg);
}
int main()
{
     int n;
    printf("Enter the number of disks : ");
    scanf("%d",&n);
    printf("The Tower of Hanoi involves the moves :\n\n");
    towers(n,'A','C','B');
    return 0;
}
```

5.5. Fibonacci Series

Another problem is illustrated by an attempt to calculate the Fibonacci numbers recursively. Lets assume that we have written a recursive algorithm to calculate the Fib numbers.

```
long fib(int n)
{
    if(n =1)
        return 1;
    else
        return fib(n-1) +fib(n-2);
}
```

This routine works but has a serious problem? It performs quite badly. (fib(40) takes 4 minutes to compute). Problem: Redundant calculations. To compute fib(n) we recursively compute fib(n-1), when this call returns we compute fib(n-2). But we have already computed f(n-2) in the process of computing f(n-1), so the call to fib(n-2) is wasted. It is a redundant calculation. Note that: The redundancy increases for each recursive call.



Chapter 6: TREES

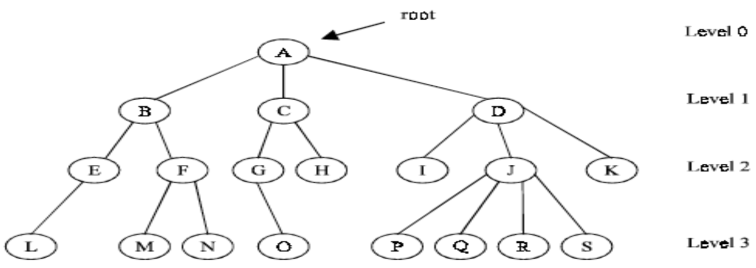


Fig 1: A Tree

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

1. There is a special node called the **root** of the tree.
2. Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., disjoined) subsets each of which is itself a tree, are called sub tree.

6.1. Basic Terminology

Root: Root is the first node in the hierarchical arrangement of the data items. 'A' is a root node in the Fig1. Each data item in a tree is called a **node**. It specifies the data information and links (branches) to other data items.

Degree of a node is the number of sub trees of a node in a given tree. In an above figure, The degree of node A, B, C and D are 3,2,2 and 3 respectively.

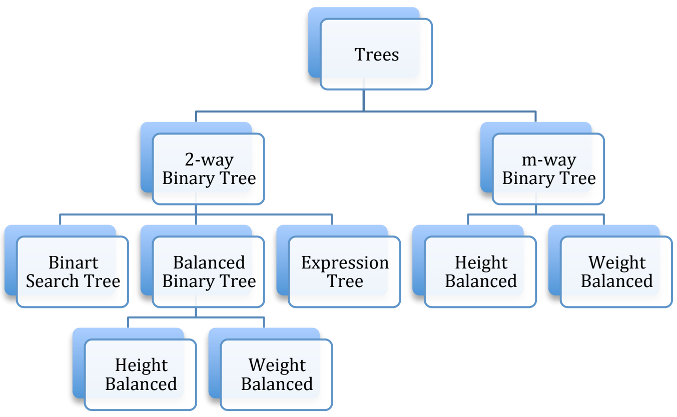
The **degree of a tree** is the maximum degree of node in a given tree. In above tree, degree of a node J is 4. All the other nodes have less or equal degree, so, the degree of the above tree is 4.

A node with degree 0 is called a **terminal node** or a **leaf**. In above tree M, N, I, O etc. are leaf node (terminal node).

Any Node whose degree is not zero is called **non-terminal** node. There are intermediate nodes while traversing the given tree from its root node to the terminal nodes.

A tree is structured in different **levels**. The entire tree is leveled in such a way that the root node is always of level 0. Then, its immediate children are level 1 and their immediate children are at level 2 and so on up to the terminal nodes. That is, if a node is at level n, then its children will be at level n+1.

Depth of a tree is the maximum level of any node in a given tree. The term height is also used to denote the depth. Trees can be divided in different categories as follows:



6.2. BINARY TREES

- A binary tree is a tree in which no node can have more than two children. Typically these children are described as "left child" and "right child" of the parent's node.
- A binary tree T is defined as a finite set of elements, called nodes, such that:
  - T is empty (i.e. if T has no node called null tree or empty tree).
  - T contains a special node R, called root node of T, and the remaining nodes of T from an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub tree of R. If T1 is non empty then its root is called the left successor of R, Similarly if T2 is non empty then its root is called the right successor of R.

Consider a binary tree T in Fig 2. Here 'A' is the root node of the binary tree T. Then 'B' is the left child of 'A' and 'C' is the right child of 'A' i.e., 'A' is a father of 'B' and 'C'. The node 'B' and 'C' are called brothers. If a node has no child then it is called a leaf node. Nodes H, I, F, J are leaf node in Fig 2.

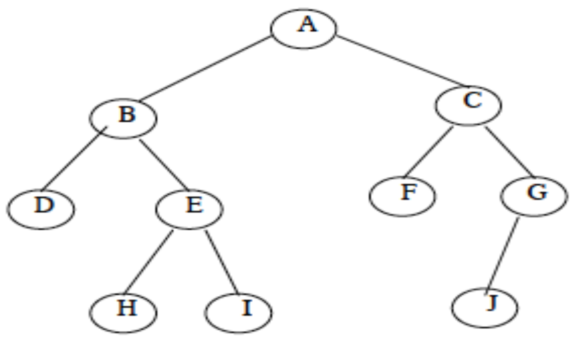


Fig 2: Binary Tree

6.2.1. Strictly Binary Tree: The tree is said to be **strictly binary tree**, if every non-leaf node in a binary tree has non-empty left and right sub trees. A strictly binary tree with *n* leafs always contains *2n–1* node. The tree in Fig 3 is strictly binary tree; where as the tree in Fig 2 is not. That is every node in the strictly binary tree can have either no children or two children. They are also called *2-tree or extended binary tree*.

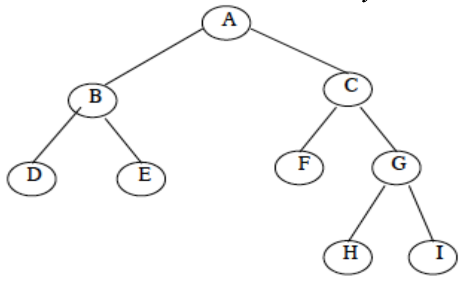


Fig 3: Strictly Binary Tree

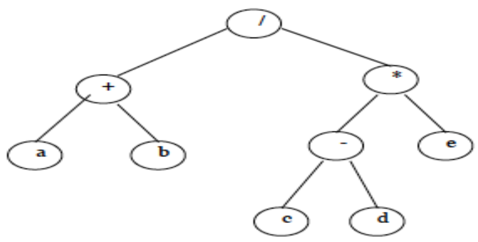


Fig 4: Expression Tree

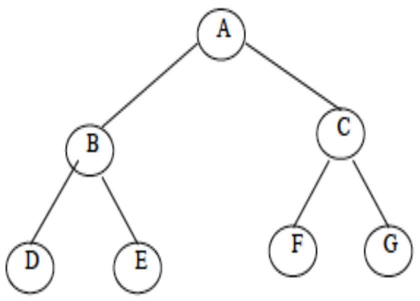


Fig5: Complete Binary Tree

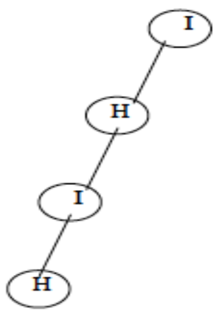


Fig6: Left Skewed

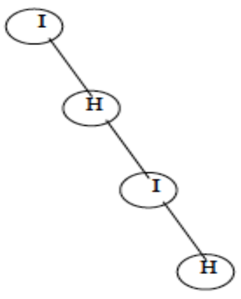


Fig7: Right Skewed

The main application of a 2-tree (Strictly binary Tree) is to represent and compute any algebraic expression using binary operation. For example, consider an algebraic expression E.
E = (a + b)/((c – d)*e) ; E can be represented by means of the extended binary tree T as shown in Fig 4. Each variable or constant in E appears as an internal node in T whose left and right sub tree corresponds to the operands of the operation.

6.2.2. Complete Binary Tree: A **complete binary tree** at depth '*d*' is the strictly binary tree, where all the leafs are at level *d*. Fig 5 illustres the complete binary tree of depth 2.
A binary tree with n nodes, n > 0, has exactly n – 1 edges. If a binary tree contains *n* nodes at level *I*, then it contains at most 2n nodes at level *I* + 1. A complete binary tree of depth *d* is the binary tree of depth *d* contains exactly 2^I nodes at each level *I* between 0 and *d*.

The main difference between a binary tree and ordinary tree is:
1. A binary tree can be empty where as a tree cannot.
2. Each element in binary tree has exactly two sub trees (one or both of these sub trees may be empty). Each element in a tree can have any number of sub trees.
3. The sub tree of each element in a binary tree is ordered, left and right sub trees. The sub trees in a tree are unordered.

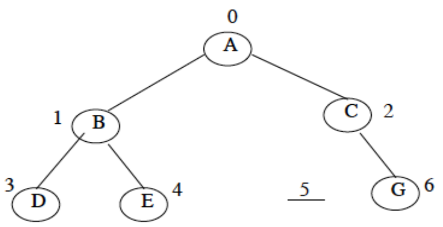
If a binary tree has only left sub trees, then it is called **left skewed** binary tree. Fig 6 is a left skewed binary tree. If a binary tree has only right sub trees, then it is called **right skewed** binary tree. Fig 7 is a right skewed binary tree.

6.3. BINARY TREE REPRESENTATION

There are two ways of representing binary tree T in memory:
1. Sequential representation using arrays
2. Linked list representation

6.3.1. ARRAY REPRESENTATION

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially. Suppose a binary tree T of depth d. Then at most 2^d-1 nodes can be there in T. (i.e.,SIZE = 2^d–1).Consider a binary tree in Fig8 of depth 3. Then SIZE = $2^3 – 1 = 7$. Then the array A [7] is declared to hold the nodes.



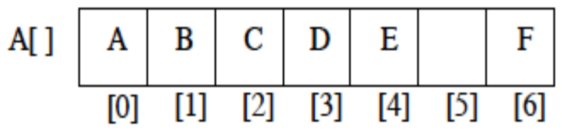


Fig8: Binary Tree of Depth 3   Fig9: Array Representation of Binary Tree of Fig8

To perform any operation on Binary tree we have to identify the father, the left child and right child of node.

1. The father of a node having index **n** can be obtained by **(n – 1)/2**. For example to find the father of D, where array index n = 3. Then the father nodes index can be obtained
   = (n – 1)/2
   = 3 – 1/2
   = 2/2
   = 1
   i.e., A[1] is the father D, which is B.
2. The left child of a node having index n can be obtained by (2n+1). For example to find the

left child of C, where array index n = 2. Then it can be obtained by
= (2n +1)
= 2*2 + 1
= 4 + 1
= 5
i.e., A[5] is the left child of C, which is NULL. So no left child for C.

3. The right child of a node having array index n can be obtained by the formula (2n+ 2). For example to find the right child of B, where the array index n = 1. Then
   = (2n + 2)
   = 2*1 + 2
   = 4
   i.e., A[4] is the right child of B, which is E.
4. If the left child is at array index n, then its right brother is at (n + 1). Similarly, if the right child is at index n, then its left brother is at (n – 1).

The array representation is more ideal for the complete binary tree. The Fig8 is not a complete binary tree. Since there is no left child for node C, i.e., A[5] is vacant. Even though memory is allocated for A [5] it is not used, so wasted unnecessarily.

6.3.2. LINKED LIST REPRESENTATION

The most popular and practical way of representing a binary tree is using linked list (or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as:

(a) Left Child (LChild)
(b) Information of the Node (Info)
(c) Right Child (RChild)

The LChild links to the left child of the parent's node, info holds the information of every node and RChild holds the address of right child node of the parent node. The following figures shows that the Linked List representation of the binary tree of Fig8.

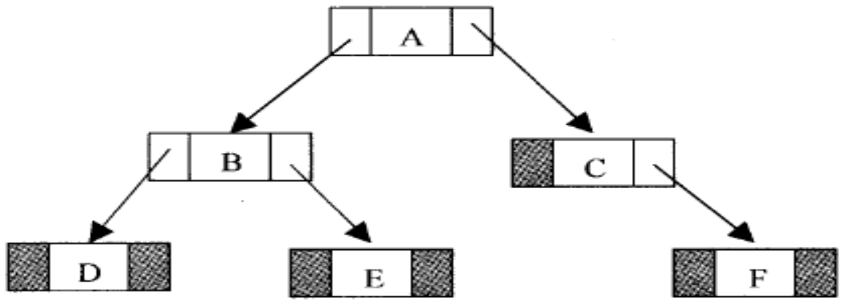


Fig9: Linked List representations of Binary Tree

If a node has no left or/and right node. Corresponding LChild and RChild is assigned to NULL. The node structure in C can be represented as:

```
struct Node {
      int info;
      struct Node *LChild;
      struct Node *RChild;
};
typedef struct Node *NODE;
```

6.4. BASIC OPERATIONS IN BINARY TREE (Primitive Operations)

The basic operations that are commonly performed in binary tree are:

1. Create an empty binary tree

2. Traversing a binary tree
3. Inserting a new node
4. Deleting a Node
5. Searching for a Node
6. Copying the mirror image of a tree
7. Determine the total no: of Nodes
8. Determine the total no: leaf Nodes
9. Determine the total no: non-leaf Nodes
10. Find the smallest element in a Node
11. Finding the largest element
12. Find the Height of the tree
13. Finding the Father/Left Child/Right Child/Brother of an arbitrary node.

6.5. TRAVERSING BINARY TREE

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are three standard ways of traversing a binary tree. They are:

1. Pre Order Traversal (Node-left-right)
2. In order Traversal (Left-node-right)
3. Post Order Traversal (Left-right-node)

6.5.1. Pre Order Traversal (Root-Left-Right)

A systematic way of visiting all nodes in a binary tress that visits a node, then visits the nodes in the left sub tree of the node, and then visits the nodes in the right sub tree of the node.

1. Visit the root node
2. Traverse the left sub tree in preorder
3. Traverse the right sub tree in preorder

That is in preorder traversal, the root node is visited (or processed) first, before traveling through left and right sub trees recursively. C/C++ implementation of preorder traversal technique is as follows:

```
void preorder (Node * Root)
{
      If (Root != NULL)
      {
            printf (“%d\n”,Root → Info);
            preorder(Root → Lchild);
            preorder(Root → Rchild);
      }
}
```

The preorder traversal of the following Binary tree is: A,B,D,E,H,I,C,F,G,J

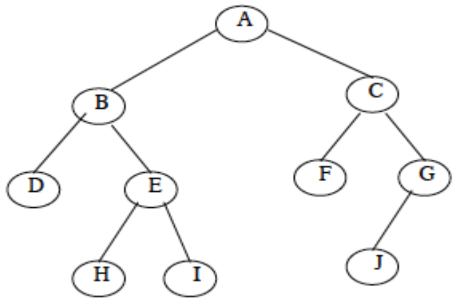


Fig10: Binary Tree

6.5.2. In Order Traversal (Left-Root-Right)

A systematic way of visiting all nodes in a binary tress that visits the nodes in the left sub tree of a node, then visits the node, and then visits the nodes in the right sub tree of the node.

1. Traverse the left sub tree in order
2. Visit the root node
3. Traverse the right sub tree in order

In order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root the right sub tree is traversed recursively. C/C++ implementation of inorder traversal technique is as follows:

```
void inorder (NODE *Root)
{
      If (Root != NULL){
            inorder(Root → Lchild);
            printf ("%d\n",Root → info);
            inorder(Root → Rchild);
      }
}
```

The in order traversal of a binary tree in Fig10 is: D, B, H, E, I, A, F, C, J, G.

6.5.3. Post Order Traversal (Left-Right-Root)

A systematic way of visiting all nodes in a binary tress that visits the node in the left sub tree of the node, then visits the node in the right sub tree in the node, and then visits the node.

1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node

In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root. C/C++ implementation of post order traversal technique is as follows:

```
void postorder (NODE *Root)
{
      If (Root != NULL)
      {
            postorder(Root → Lchild);
            postorder(Root → Rchild);
            printf ("%d\n", Root → info);
      }
}
```

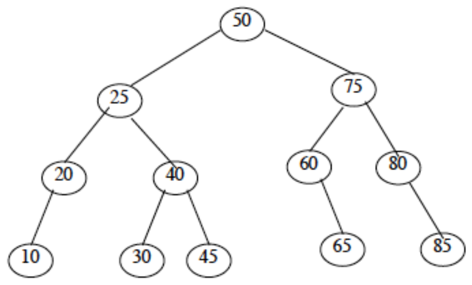
The post order traversal of a binary tree in Fig10 is: D, H, I, E, B, F, J, G, C, A.

6.6. BINARY SEARCH TREE (BST)

A Binary Search Tree is a binary tree, which is either empty or satisfies the following properties:

1. Every node has a value and no two nodes have the same value (i.e., all the values are unique).
2. If there exists a left child or left sub tree then its value is less than the value of the root.
3. The value(s) in the right child or right sub tree is larger than the value of the root node.

The Fig11 shows a typical binary search tree. Here the root node information is 50. Note that the right sub tree node's value is greater than 50, and the left sub tree nodes value is less than 50. Again right child node of 25 has large values than 25 and left child node has small values than 25. Similarly right child node of 75 has large values than 75 and left child node has small values that 75 and so on.



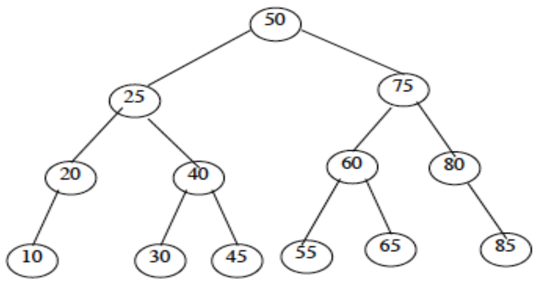


Fig11: Binary Search Tree Fig12: 55 is Inserted

All the primitives operation performed on Binary tree can also performed in Binary Search Tree (BST).

6.6.1. Inserting A Node In Binary Search Tree

A BST is constructed by the repeated insertion of new nodes to the tree structure. Inserting a node in to a tree is achieved by performing two separate operations.

1. The tree must be searched to determine where the node is to be inserted.
2. Then the node is inserted into the tree.

Suppose a “DATA” is the information to be inserted in a BST.

```
Step 1: Compare DATA with root node information of the tree
        (i) If (DATA < ROOT → Info)
                Proceed to the left child of ROOT
        (ii) If (DATA > ROOT → Info)
                Proceed to the right child of ROOT
Step 2: Repeat the Step 1 until we meet an empty sub tree, where we can insert the DATA in
        place of the empty sub tree by creating a new node.
Step 3: Exit
```

For example, consider a binary search tree in Fig11. Suppose we want to insert a DATA = 55 in to the tree, then following steps one obtained:

1. Compare 55 with root node info (i.e., 50) since 55 > 50 proceed to the right sub tree of 50.
2. The root node of the right sub tree contains 75. Compare 55 with 75. Since 55 < 75 proceed to
   the left sub tree of 75.
3. The root node of the left sub tree contains 60. Compare 55 with 60. Since 55 < 60 proceed to the right sub tree of 60.
4. Since left sub tree is NULL place 55 as the left child of 60 as shown in above Fig12.

6.6.2. Algorithm For Inserting An Element Into BST

NEWNODE is a pointer variable to hold the address of the newly created node. DATA is the information to be pushed and TEMP is the Temporary pointer variable of type NODE

```
1. Input the DATA to be pushed and ROOT node of the tree.
2. NEWNODE = Create a New Node.
3. If (ROOT == NULL)
```

```
        ROOT=NEWNODE
4. ELSE
    i.   TEMP = ROOT
    ii.  Repeat until true (while (1))
            a. IF (DATA < TEMP →info)
                i.  IF (TEMP →LChild not equal to NULL)
                        TEMP = TEMP→LChild
                ii. ELSE
                        TEMP→LChild = NewNode
                        Return
            b. ELSE IF (DATA > TEMP →info)
                i.  IF (TEMP →RChild not equal to NULL)
                        TEMP = TEMP→RChild
                ii. ELSE
                        TEMP→RChild = NewNode
                        Return
            c. ELSE
                i.  Display Duplicate node
                ii. Return
5. Exit
```

Source Code:

```
void insert(int Data, NODE Root)
{
    NODE NewNode = (NODE)malloc(sizeof(NODE));
    NODE temp;
    NewNode->info = Data;
    if(Root == NULL)
        Root = NewNode;
    else
    {
        temp = Root;
        while(1)
        {
            if(Data < temp->info)
            {
                if(temp->LChild != NULL)
                    temp = temp->LChild;
                else
                {
                    temp->LChild = NewNode;
                    return;
                }
            }
            else if(Data > temp->RChild)
            {
                if(temp->RChild != NULL)
                    temp = temp->RChild;
                else
                {
                    temp->RChild = NewNode;
                    return;
                }
            }
            else
            {
                printf("Duplicate node \n");
                return;
            }
        }
    }
}
```

6.6.3. Algorithm for Searching A Node from BST

```
1.  Input the DATA to be searched and assign the address of the root node to ROOT.
2.  TEMP = ROOT
3.  Repeat Until DATA is not Equal to TEMP → Info)
        (a) IF (DATA == TEMP → Info )
                i. Display “The DATA exist in the tree”
                ii. Return
        (b) If (TEMP == NULL)
                i. Display “The DATA does not exist”
                ii. Return
        (c) If(DATA > TEMP→Info)
                i. TEMP = TEMP→RChild
        (d) Else If(DATA < TEMP→Info)
                i. TEMP = TEMP→LChild
4.  Exit
```

Source Code:

```
void findBST(int data, NODE Root, NODE *node)
{
    NODE TEMP = Root;
    while(data != TEMP->info)
    {
        if(data == TEMP->info)
        {
            printf("Data Found in tree \n");
            *node = TEMP;
            return;
        }
        if(TEMP == NULL)
        {
            printf("Data Not found in tree");
            return;
        }
        if(data > TEMP->info)
            TEMP = TEMP->RChild;
        else if(data < TEMP->info)
            TEMP = TEMP->LChild;
    }

}
```

6.6.4. Deleting A Node From BST

First search and locate the node to be deleted. Then any one of the following conditions arises:

1. The node to be deleted has no children
2. The node has exactly one child (or sub tress, left or right sub tree)
3. The node has two children (or two sub tress, left and right sub tree)

Suppose the node to be deleted is N.

- If N has no children then simply delete the node and place its parent node by the NULL pointer.

- If N has one child, check whether it is a right or left child.
- If it is a right child, then find the smallest element from the corresponding right sub tree. Then replace the smallest node information with the deleted node.
- If N has a left child, find the largest element from the corresponding left sub tree. Then replace the largest node information with the deleted node.

The same process is repeated if N has two children, i.e., left and right child. Randomly select a child and find the small/large node and replace it with deleted node. NOTE that the tree that we get after deleting a node should also be a binary search tree.

Lets look an example of deleting a node from BST. Consider the BST of Fig12 and if we want to delete 75 from tree, following steps are obtained.

Step 1: Assign the data to be deleted in DATA and NODE = ROOT.

Step 2: Compare the DATA with ROOT node, i.e., NODE, information of the tree. Since (50 < 75)
NODE = NODE → RChild

Step 3: Compare DATA with NODE. Since (75 = 75) searching successful. Now we have located the
data to be deleted, and delete the DATA.(Fig13).

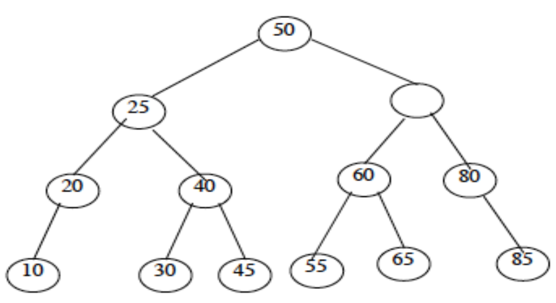


Fig 13: Delete 75

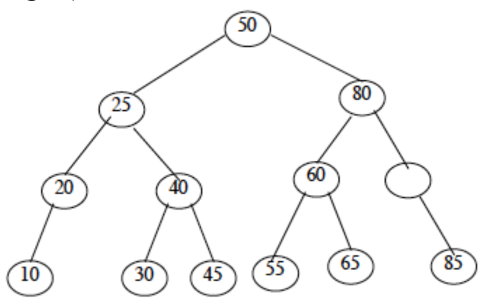


Fig14: Deleted node replaced

Step 4: Since NODE (i.e., node where value was 75) has both left and right child choose one. (Say
Right Sub Tree) - If right sub tree is opted then we have to find the smallest node. But if
left sub tree is opted then we have to find the largest node.

Step 5: Find the smallest element from the right sub tree (i.e., 80) and replace the node with deleted node (Fig 14).

Step 6: Again the (NODE → RChild is not equal to NULL) find the smallest element from the right
sub tree (Which is 85) and replace it with empty node (Fig15).

Step 7: Since (NODE→RChild = NODE→LChild = NULL) delete the NODE and place NULL in the parent node (Fig16).

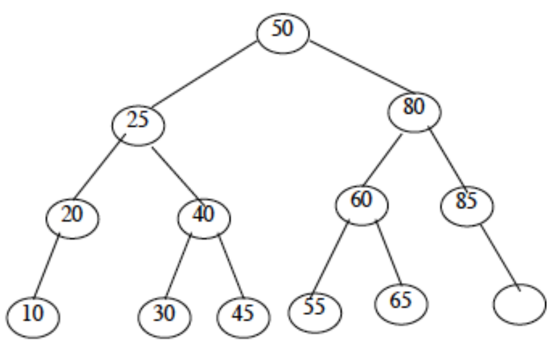


Fig15

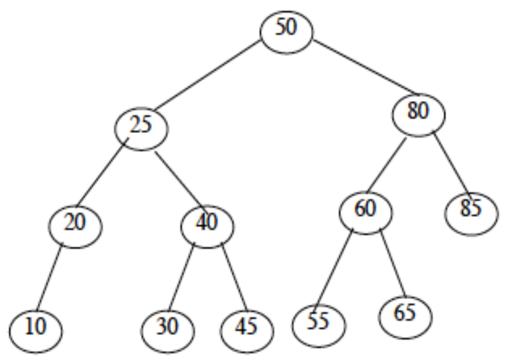


Fig16

Step 8: Exit.

6.6.5. Algorithm for deleting a node from BST

NODE is the current position of the tree, which is in under consideration. LOC is the place where node is to be replaced and TEMP is the temporary node. DATA is the information of node to be deleted.

```
1. Find the location NODE of the DATA to be deleted.
2. If (NODE = NULL)
       (a) Display “DATA is not in tree”
       (b) Exit
3. If(NODE → LChild == NULL)  (Node has no left child)
       (a) NODE = NODE → RChild
4. ELSE If(NODE → RChild == NULL) (Node has no right child)
       (a) NODE = NODE → LChild
5. Else If((NODE → LChild not equal to NULL) && (NODE → RChild not equal to NULL))
       (a) TEMP = NODE →LChild
       (b) LOC = NODE
       (c) While(TEMP → RChild not equal to NULL)
               i.  LOC = TEMP
               ii. TEMP = TEMP→RChild
       (d) NODE→info = TEMP→info
       (e) IF(LOC == NODE)
               i.  LOC→LChild =TEMP→LChild
       (f) ELSE
               i.  LOC → RChild = TEMP→LChild
6. Free TEMP
7. Exit
```

Source Code:

```
void deleteBST(int data, NODE Root)
{
    NODE node, tmp, loc;
    find(data,Root,&node); // find the node to be delete
    if(node == NULL)
    {
        printf("NO data in Tree\n");
        return;
    }
    if(node->RChild == NULL) //node has no right child
        node=node->LChild;
    else if(node->LChild == NULL) //node has no left child
        node = node->RChild;
    else //node has both child
    {
        tmp = node->LChild;             //tmp = node->RChild;
        loc = node;
        while(tmp->RChild != NULL)      //while(tmp->LChild != NULL)
        {
            loc = tmp;
            tmp = tmp->RChild;          //tmp=tmp->LChild;
        }
        node->info = tmp->info;
        if(loc == node)
            loc->LChild = tmp->LChild;  //loc->RChild = tmp->RChild;
        else
```

```
                loc->RChild = tmp->LChild; //loc->LChild = tmp->RChild;
        }
        free(tmp);
    }
```

6.7. BALANCED BINARY TREE

A balanced binary tree is one in which the **largest path** through the left sub tree is the same length as the **largest path** of the right sub tree, i.e., from root to leaf. Searching time is very less in balanced binary trees compared to unbalanced binary tree. i.e., balanced trees are used to maximize the efficiency of the operations on the tree. There are two types of balanced trees:

1. Height Balanced Trees
2. Weight Balanced Trees

6.7.1. Height Balanced Tree

In height balanced trees balancing the height is the important factor. There are two main approaches, which may be used to balance (or decrease) the depth of a binary tree:

a. Insert a number of elements into a binary tree in the usual way, using the algorithm of binary search Tree insertion. After inserting the elements, copy the tree into another binary tree in such a way that the tree is balanced. This method is efficient if the data(s) are continually added to the tree.
b. Another popular algorithm for constructing a height balanced binary tree is the AVL tree.

6.7.2. Weight Balanced Tree

A weight-balanced tree is a balanced binary tree in which additional weight field is also there. The nodes of a weight-balanced tree contain four fields:

a. Data Element
b. Left Pointer
c. Right Pointer
d. A probability or weight field

- The data element, left and right pointer fields are same as that in any other tree node.
- The probability field is a specially added field for a weight-balanced tree. This field holds the probability of the node being accessed again, that is the number of times the node has been previously searched for.
- When the tree is created, the nodes with the highest probability of access are placed at the top. That is the nodes that are most likely to be accessed have the lowest search time.
- And the tree is balanced if the weights in the right and left sub trees are as equal as possible.
- The average length of search in a weighted tree is equal to the sum of the probability and the depth for every node in the tree.
- The root node contain highest weighted node of the tree or sub tree.
- The left sub tree contains nodes where data values are less than the current root node, and the right sub tree contain the nodes that have data values greater than the current root node.

6.7.3. AVL TREES

Two Russian Mathematicians, G.M. Adel'son Vel'sky and E.M. Landis developed algorithm in 1962; here the tree is called AVL Tree.

An AVL tree is a binary tree in which the left and right sub tree of any node may differ in height by at most 1, and in which both the sub trees are themselves AVL Trees. Each node in the AVL Tree possesses any one of the following properties:

a. A node is called **left heavy**, if the largest path in its left sub tree is one level larger than the largest path of its right sub tree.
b. A node is called **right heavy**, if the largest path in its right sub tree is one level larger than

the largest path of its left sub tree.

c. The node is called **balanced**, if the largest paths in both the right and left sub trees are equal. Fig17 shows some example for AVL trees.

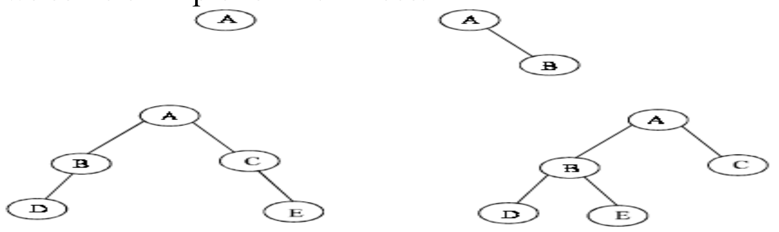


Fig17: AVL Trees

The construction of an AVL Tree is same as that of an ordinary binary tree except that after the addition of each new node, a check must be made to ensure that the AVL balance conditions have not been violated. If the new node causes an imbalance in the tree, some rearrangement of the tree's nodes must be done.

6.7.3.1. Algorithm for inserting a node in an AVL Tree

1. Insert the node in the same way as in an ordinary binary tree.
2. Trace a path from the new nodes, back towards the root for checking the height difference of the two sub trees of each node along the way.
3. Consider the node with the imbalance and the two nodes on the layers immediately below.
4. If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.
5. If these three nodes lie in a dogleg pattern (i.e., there is a bend in the path) apply a double rotation to correct the imbalance.
6. Exit.

The above algorithm will be illustrated with an example shown in Fig18, which is an unbalance tree. We have to apply the rotation to the nodes 40, 50 and 60 so that a balance tree is generated. Since the three nodes are lying in a straight line, single rotation is applied to restore the balance.

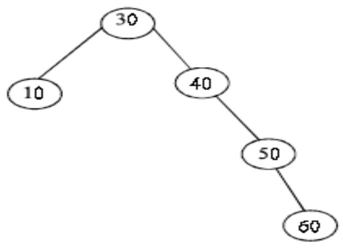


Fig 18

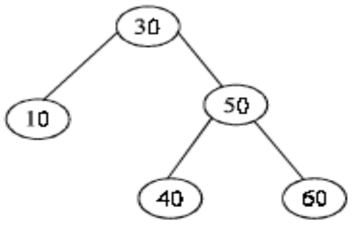


Fig19

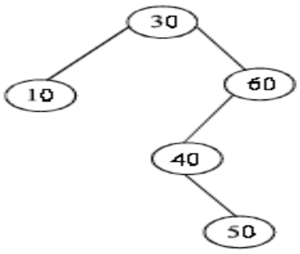


Fig20

Fig19 is a balance tree of the unbalanced tree in Fig18. Consider a tree in Fig20 to explain the double rotation.

While tracing the path, first imbalance is detected at node 60. We restrict our attention to this node and the two nodes immediately below it (40 and 50). These three nodes form a ***dogleg pattern***. That is there is bend in the path. Therefore we apply double rotation to correct the balance. A double rotation, as its name implies, consists of two single rotations, which are in opposite direction. The first rotation occurs on the two layers (or levels) below the node where the imbalance is found (i.e., 40 and 50). Rotate the node 50 up by replacing 40, and now 40 become the child of 50 as shown in Fig21.

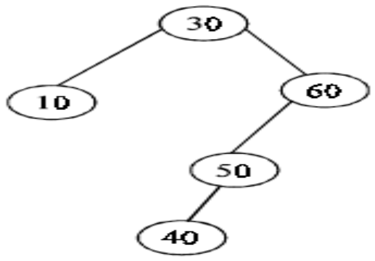


Fig21

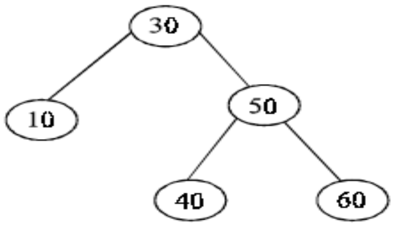
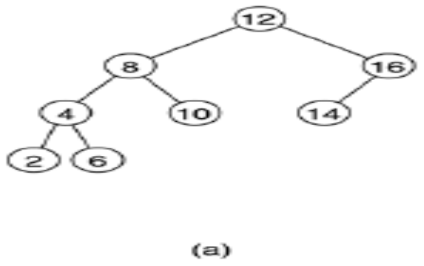


Fig22

Apply the second rotation, which involves the nodes 60, 50 and 40. Since these three nodes are lying in a straight line, apply the single rotation to restore the balance, by replacing 60 by 50 and placing 60 as the right child of 50 as shown in Fig22. Balanced binary tree is a very useful data structure for searching the element with less time.

More Examples on AVL Balanced Tree

An AVL Tree is defined to be a binary search tree with this balance property. In the work that follows, the height of an empty sub tree is defined to be -1.



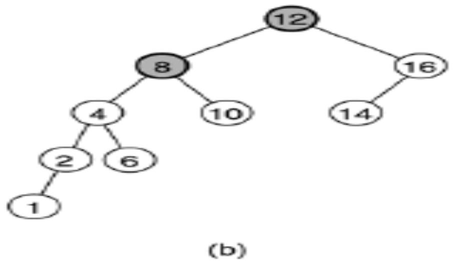
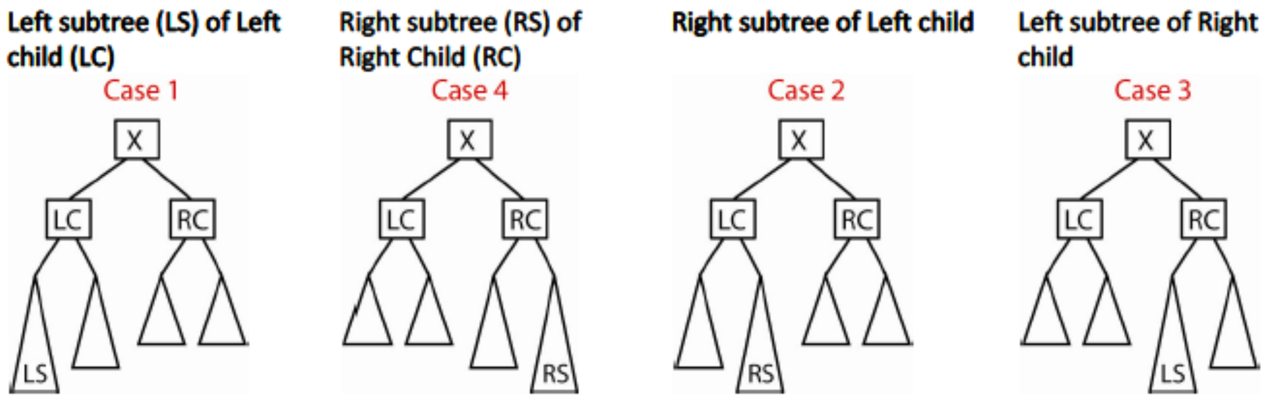


Fig 22(a): AVL Tree Fig22 (b): Not AVL tree (Unbalanced)

Note in figure 22(a), that insert (1), using the BST algorithm will destroy the AVL property as shown in figure 22(b). Thus, we will have to modify the insert (and remove) algorithms so that they don't destroy the AVL property. The following algorithm, called single rotation, finds the deepest node that violates the property (node 8 in figure 22(b) above) and rebalances the tree from there. It can be proven that this rebalancing guarantees that the entire tree satisfies the AVL property.

Consider inserting a node into an AVL Tree. Suppose a height imbalance of 2 results at some deepest node, X. Thus, X needs to be rebalanced. Assuming an AVL tree (e.g. balance condition met) existed before the insertion, relative to X, we can define these 4 places where the insertion took place:


Left subtree (LS) of Left child (LC)
Case 1
X
LC
RC
LS
Right subtree (RS) of Right Child (RC)
Case 4
X
LC
RC
RS
Right subtree of Left child
Case 2
X
LC
RC
RS
Left subtree of Right child
Case 3
X
LC
RC
LS

Case 1: In figure 23 (a) below, node k_2 plays the role of X in preceding figure, the deepest node where the imbalance is observed, and k_1 is the left child of k_2. The idea is to rotate k_1 and k_2

clockwise making k_2 the right sub tree of k_1 and making B the left sub tree of k_2. It is easy to verify that this approach works. First, k_2 is larger than k_1, thus k_2 can be the right child of k_1. Second, all nodes in B are between k_1 and k_2 which is still true when B becomes k_2's left child.

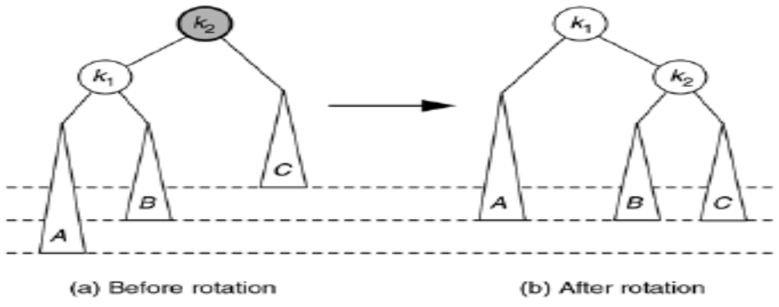
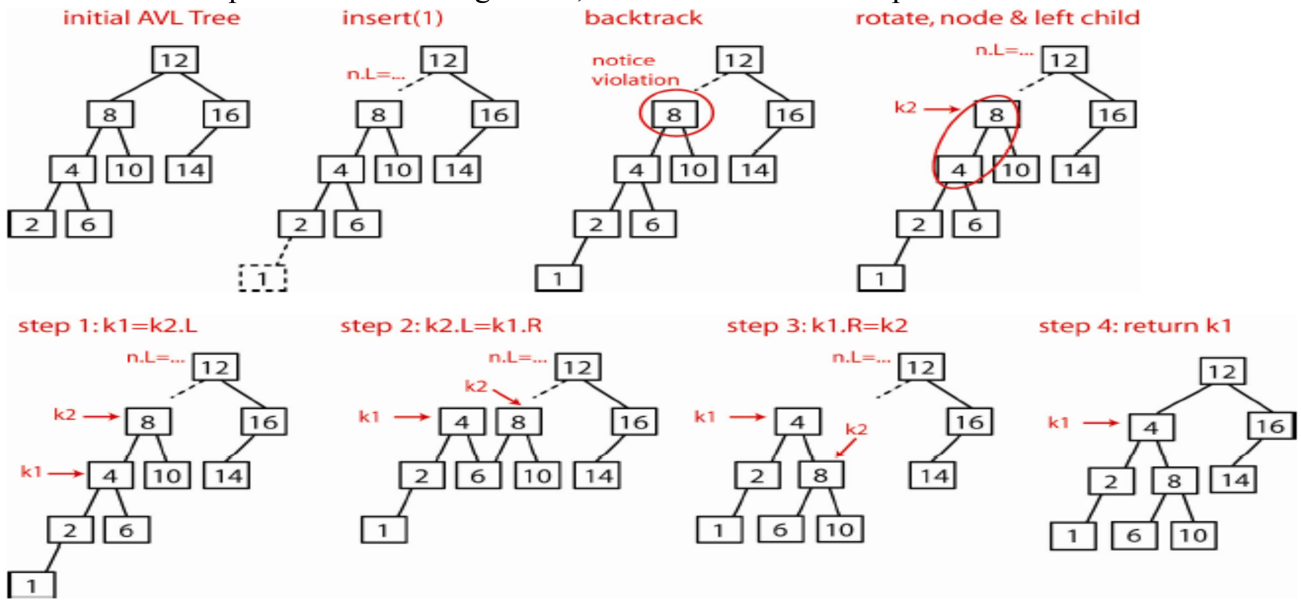


Fig 23: Single rotation to fix case 1

Consider an example of the Case 1 algorithm, which is shown in 4 steps below:



Implementation Algorithm

```
NODE K1, K2;
K1 = K2->left;
K2->Left = K1->Right;
K1->Right = K2;
Return K1;
```

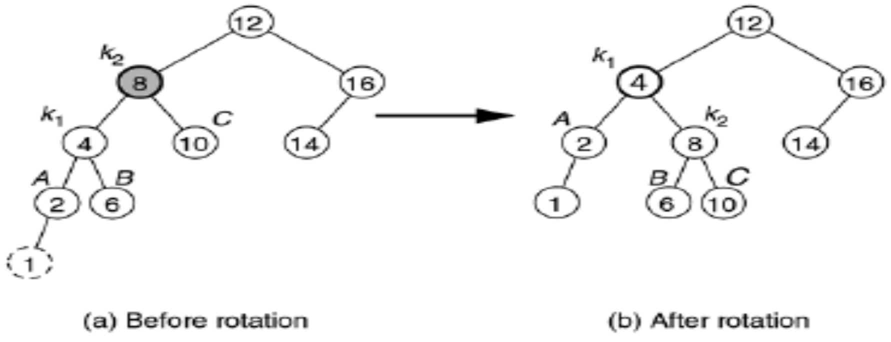


Fig 24: Single rotation fixes an AVL tree after insertion of 1.

Case 4: Consider Case 4, which is a mirror image of Case 1. In Fig 25(b) below, node k_1 plays the role of X, the deepest node where the imbalance is observed, and k_2 is the right child. The idea is to rotate k_1 and k_2 counterclockwise k_1 the left child of k_2 and making B the right child of k_1. It is easy to verify that this approach works. First, k_1 is smaller than k_2, thus it can be the left child of k_2. Second, all nodes in B are between k_1 and k_2 which is still true when B becomes k_1's right child.

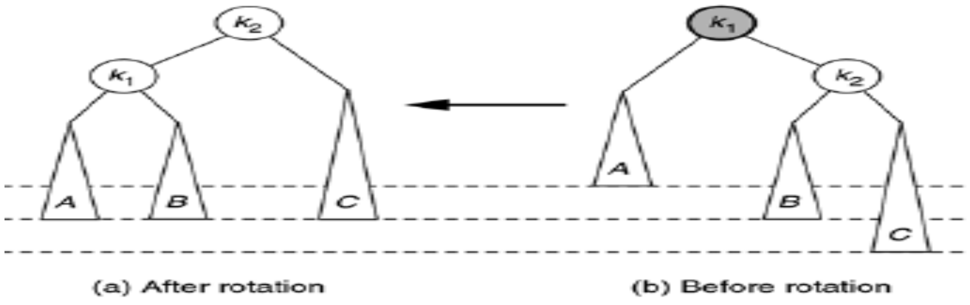
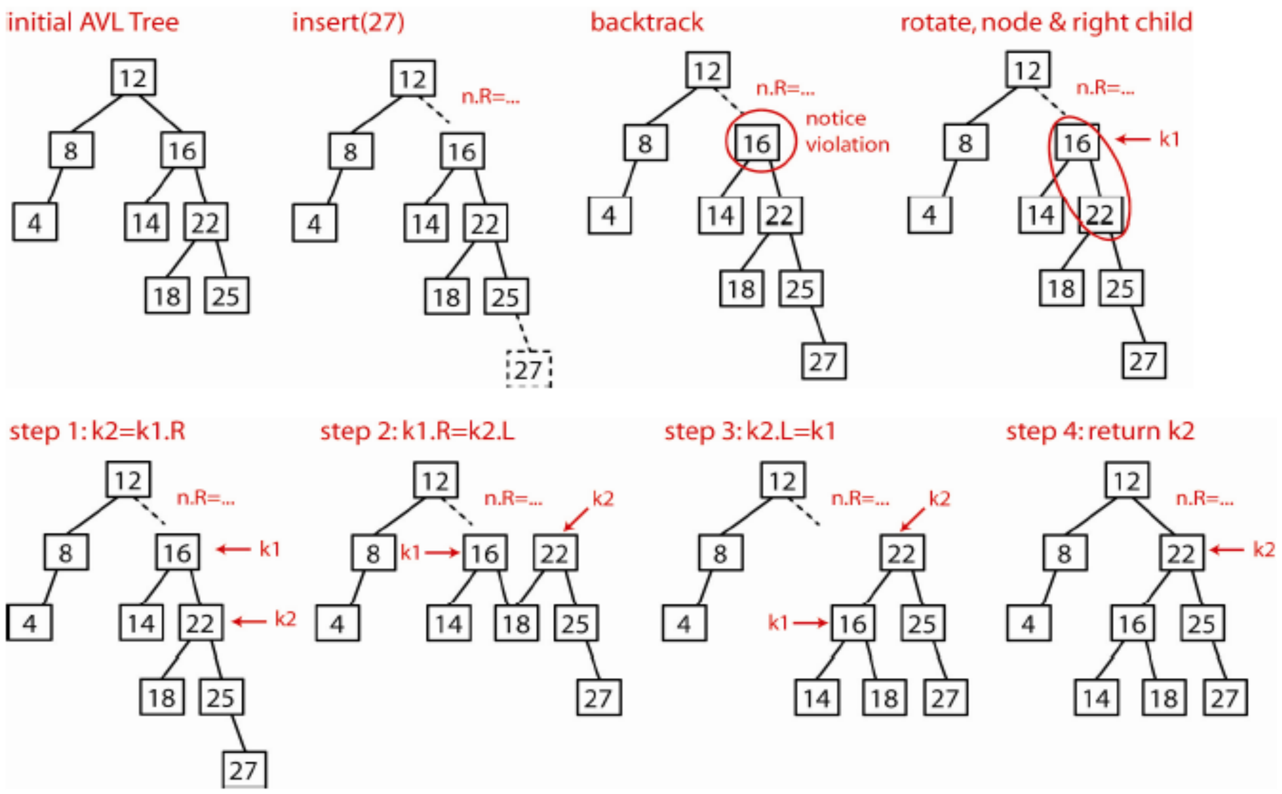


Fig25: Symmetric single rotation to fix case 4.

Consider an example of the Case 4 algorithm that is shown in 4 steps below:



Implementation Algorithm

```
NODE K1, K2;
K2=K1->right;
K1->right= K2->left;
K2->left = K1;
Return K2;
```

Case 2: Consider Case 2. A rotation as described above, does not work.

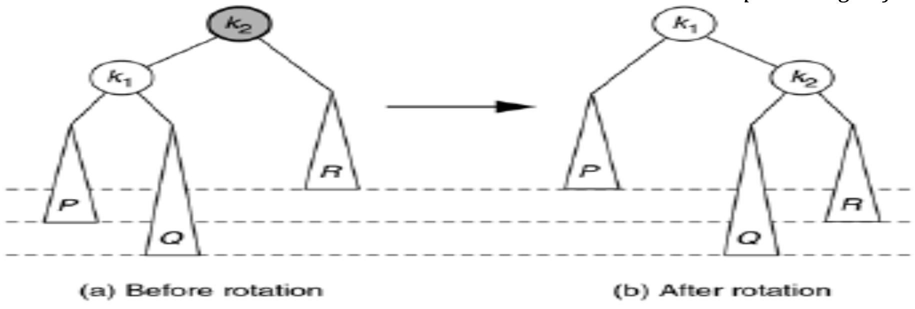


Fig26: Single Rotation does not fix case2

Consider Case 2 again. A double rotation does work. Since $k_1<k_2<k_3$, the nodes can be rearranged so that k_1 and k_3 are the left and right, respectively, sub trees of k_2. Since all elements in B are between k_1 and k_2 they remain that way when we make k_1's right child be B. Similarly, since all elements in C are between k_2 and k_3, we can make C k_3's left child.

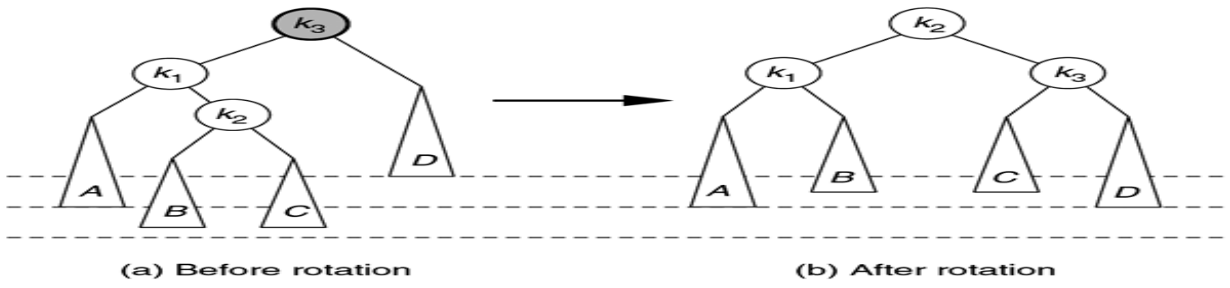
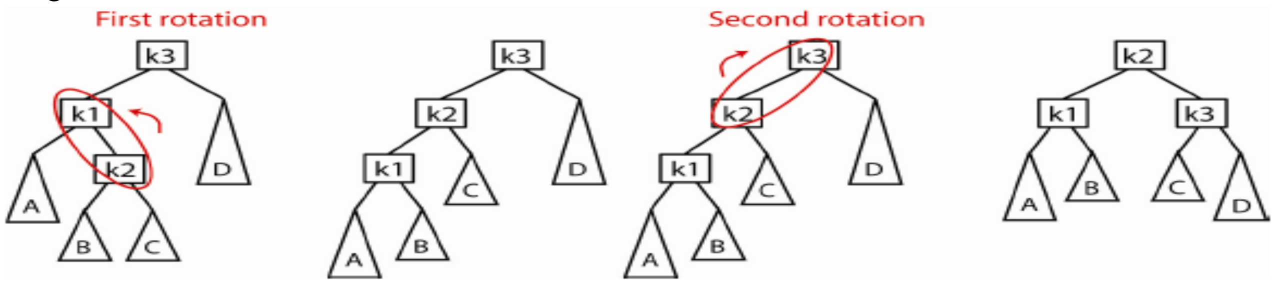


Fig27: Left-Right Double rotation to fix Case2.

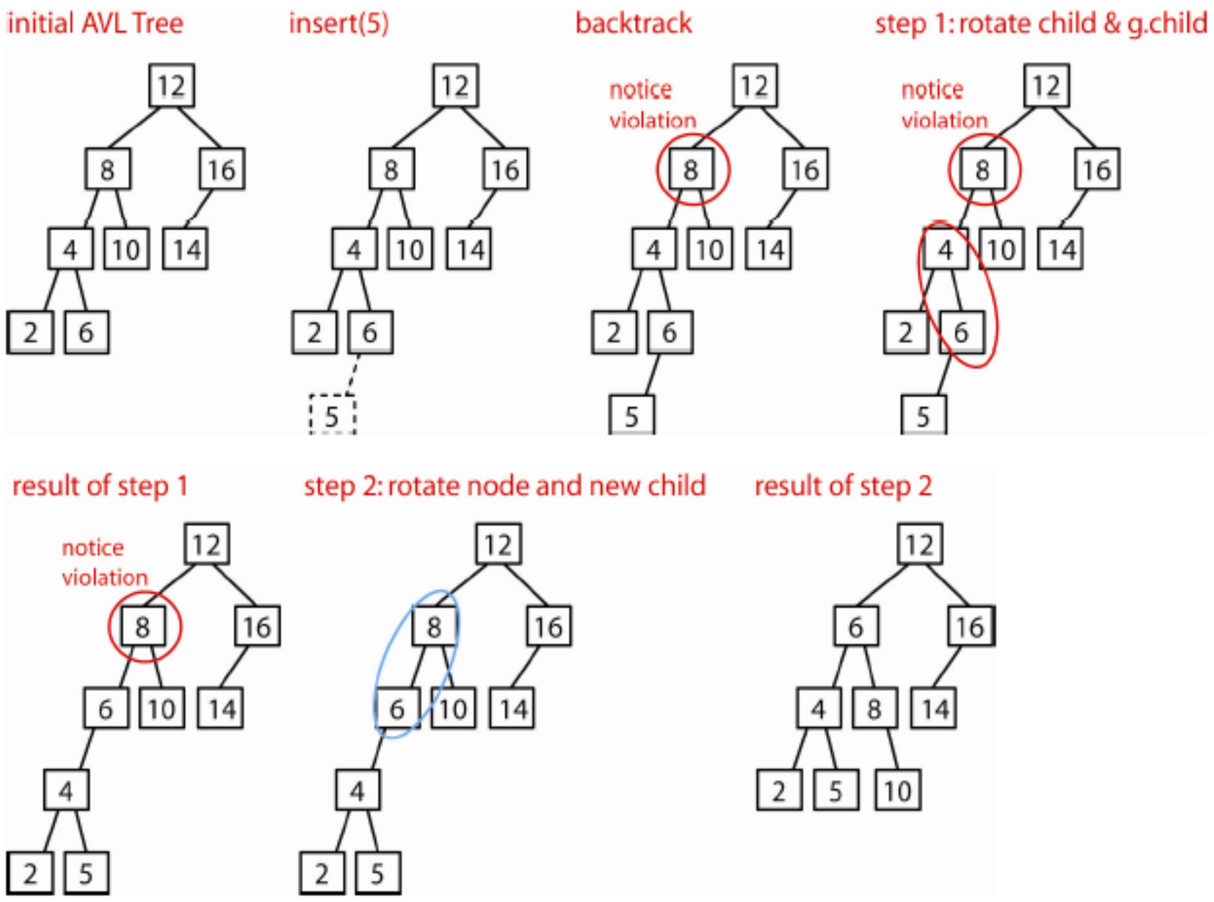
The double rotation can be seen as two single rotations as described for cases 1 and 4:

1. Rotate X's child and grandchild
2. Rotate X and its new child

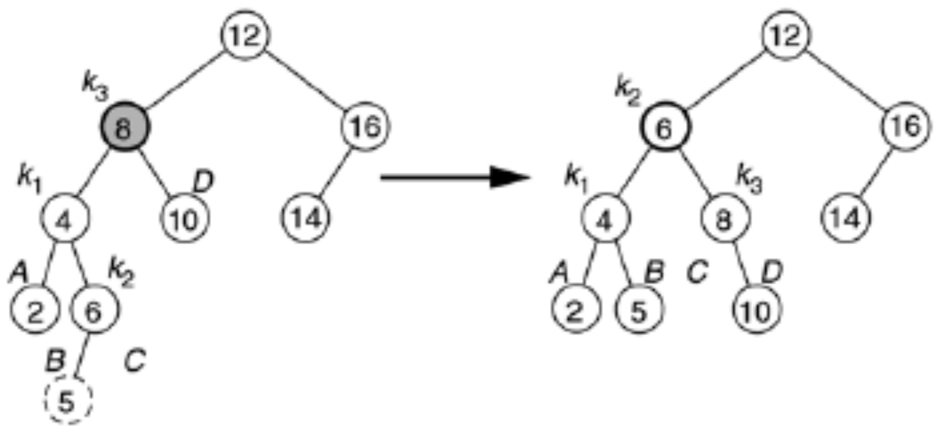
In figure 27 above, first rotate k_1 and k_2 counter-clockwise, then rotate k_2 and k_3 clockwise:



Example



Example



(a) Before rotation (b) After rotation

6.7.4. B-TREE

B-Trees are tree data structure that is most commonly found in database and file system. B-Trees have substantial advantage over alternative implementations when node access times far exceed access times within nodes. This usually occurs when most nodes are in secondary storage such as hard drive. By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often, and efficiency increases.

The idea behind B-Trees is that inner nodes can have a variable number of child nodes within some pre-defined range. Hence, B-trees do not need re-balancing as frequently as other self-balancing binary trees. The lower and upper bounds on the number of child nodes are fixed for a particular implementation.

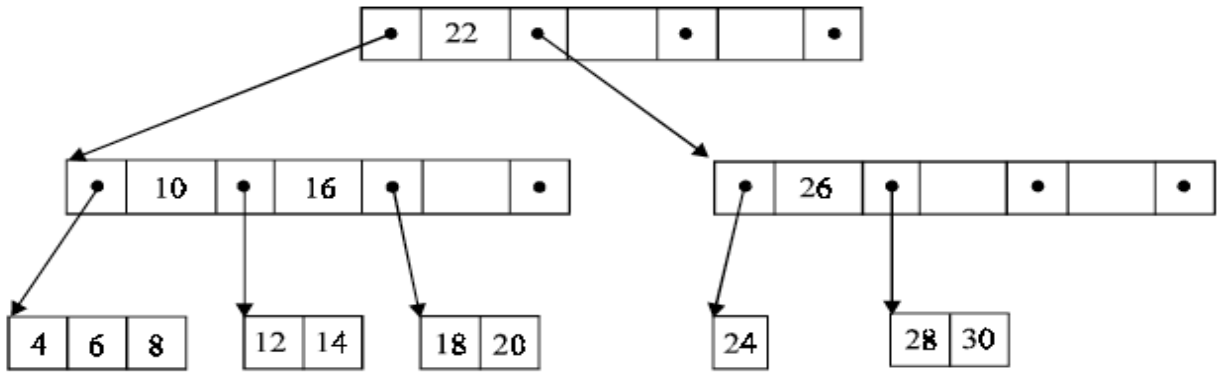


Fig23: B-Tree

Nodes in a B-Tree are usually represented as an ordered set of elements and child pointers. Every node but the root contains a minimum of m elements, a maximum of n elements, and a maximum of n + 1 child pointers, for some arbitrary m and n. For all internal nodes, the number of child pointers is always one more than the number of elements. Since all leaf nodes are at the same height, nodes do not generally contain a way of determining whether they are leaf or internal.
Each inner node's elements act as separation values which divide its sub trees. For example, if an inner node has three child nodes (or sub trees) then it must have two separation values or elements a_1 and a_2. All values in the leftmost sub tree will be less than a_1,all values in the middle sub tree will be between a_1 and a_2, and all values in the rightmost sub tree will be greater than a_2.

6.7.4.1. Algorithms in B-Tree

Search

Search is performed in the typical manner, analogous to that in a binary search tree. Starting at the root, the tree is traversed top to bottom, choosing the child pointer whose separation values are on either side of the value that is being searched. Binary search is typically used within nodes to determine this location.

Insertion

For a node to be in an illegal state, it must contain a number of elements, which is outside of the acceptable range.

1. First, search for the position into which the node should be inserted. Then, insert the value into that node.
2. If no node is in an illegal state then the process is finished.
3. If some node has too many elements, it is split into two nodes, each with the minimum amount of elements. This process continues recursively up the tree until the root is reached. If the root is split, a new root is created. Typically the minimum and maximum number of elements must be selected such that the minimum is no more than one half the maximum in order for this to work.

Deletion

1. First, search for the value, which will be deleted. Then, remove the value from the node, which contains it.
2. If no node is in an illegal state then the process is finished.
3. If some node is in an illegal state then there are two possible cases:
    a. Its sibling node, a child of the same parent node, can transfer one or more of its child nodes to the current node and return it to a legal state. If so, after updating the separation values of the parent and the two siblings the process is finished.
    b. Its sibling does not have an extra child because it is on the lower bound. In that

case both siblings are merged into a single node and we recurse onto the parent node, since it has had a child node removed. This continues until the current node is in a legal state or the root node is reached, upon which the root's children are merged and the merged node becomes the new root.

6.7.5. M-WAY SEARCH TREES

Trees having (m–1) keys and m children are called m-way search trees. A binary tree is a 2-way tree. It means that it has m – 1 = 2 – 1 = 1 key (here m = 2) in every node and it can have maximum of two children. A binary tree is also called an m-way tree of order 2. Similarly an m-way tree of order 3 is a tree in which key values could be either 1 or 2 (i.e., inside every node and it can have maximum of two children). For example an m-way tree at order (degree) 4 is shown in following figure.

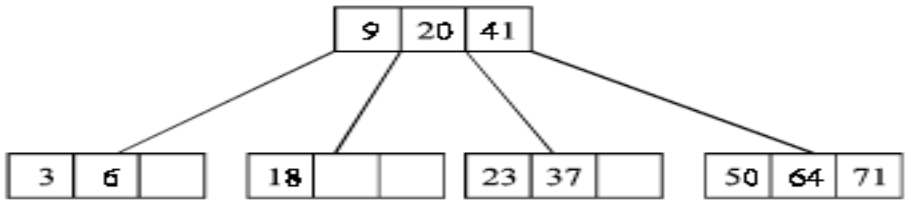
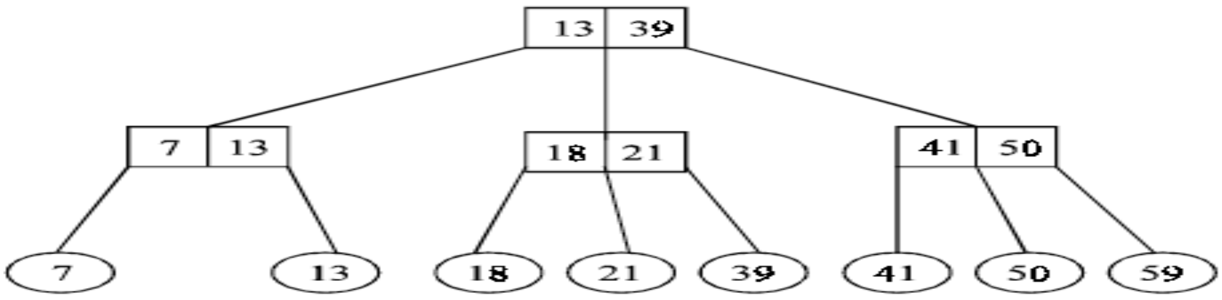


Fig. M-way tree of order 4

6.7.6. 2-3 TREES

Every node in the 2-3 trees has two or three children. A 2-3 tree is a tree in which leaf nodes are the only nodes that contains data values. All non-leaf nodes contain two values of the sub trees. All the leaf nodes have the same path length from the root node. Fig below shows a typical 2-3 tree.

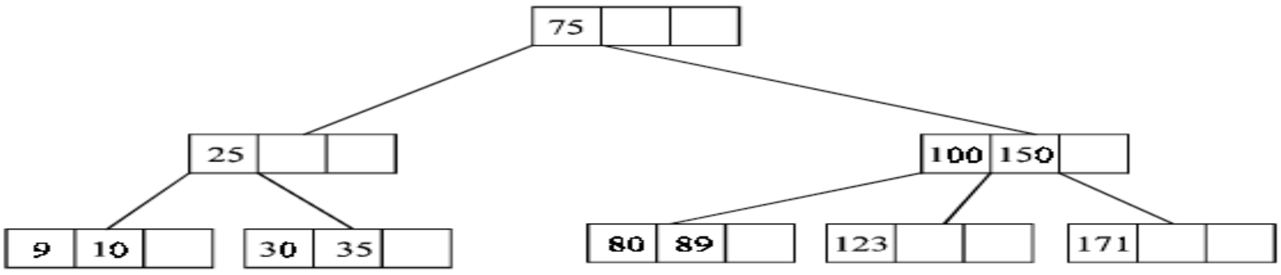


The first value in the non-leaf root node is the maximum of all the leaf values in the left sub tree. The second value is the maximum value of the middle sub tree. With this root node information following conclusion can be obtained:

1. Every leaf node in the left sub tree of any non-leaf node is equal to or less than the first value.
2. Every leaf node in the middle sub tree is less than or equal to the second value and greater than the first value.
4. Every leaf node in the right sub tree is greater than the second value

6.7.7. 2-3-4 TREES

A 2-3-4 tree is an extension of a 2-3 tree. Every node in the 2-3-4 trees can have maximum of 4 children. A typical 2-3-4 tree is shown in following figure.



A 2-3-4 tree is a search tree that is either empty or satisfies the following properties.

1. Every internal node is a 2, 3, or 4 node. A 2 node has one element, a 3 node has two elements and a 4 node has three elements.
2. Let LeftChild and RightChild denote the children of a 2 node and Data be the element in this node. All the elements in LeftChild have the elements less than the data, and all elements in the RightChild have the elements greater than the Data.
3. Let LeftChild, MidChild and RightChild denote the children of a 3 node and LeftData and RightData be the element in this node. All the elements in LeftChild have the elements less than the LeftData, all the elements in the MidChild have the element greater than LeftData but less than RightData, and all the elements in the RightChild have the elements greater than Right Data.
4. Let LeftChild, LeftMidChild, RightMidChild, and RightChild denote the children of a 4 node. Let LeftData, MidData and RightData be the three elements in this node. Then LeftData is less than MidData and it is less than RightData. All the elements in the LeftChild is less than LeftData, all the elements in the Left MidChild is less than MidData but greater than LeftData, all the elements in RightMidChild is less than RightData but greater than MidData, and all the elements in RightChild is greater than RightData.
5. All external nodes are at the same level.

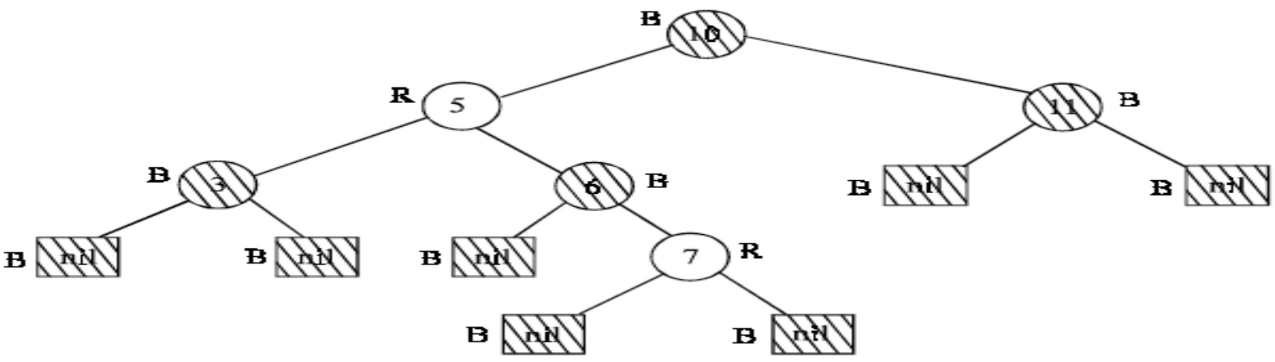
6.7.8. RED-BLACK TREE

A red-black tree is a balanced binary search tree with the following properties:

1. Every node is colored red or black.
2. Every leaf is a NULL node, and is colored black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

The red-black tree algorithm is a method for balancing trees. The name derives from the fact that each node is colored red or black, and the color of the node is instrumental in determining the balance of the tree. During insert and delete operations, nodes may be rotated to maintain tree balance.

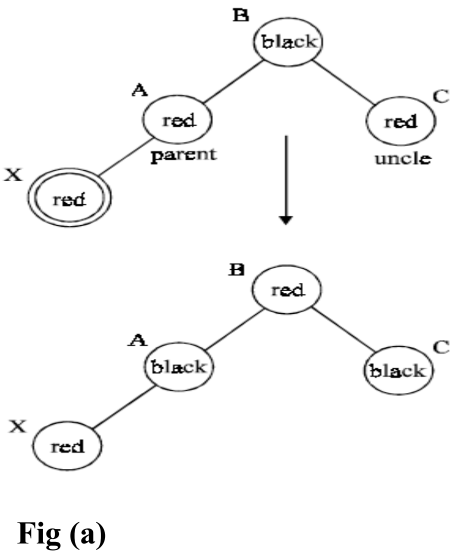
We classify red-black trees according to the order **n**, the number of internal nodes. In the following Figure, **n** = 6. Among all red-black trees of height 4, this is one with the minimum number of nodes.



To insert a node, we search the tree for an insertion point, and add the node to the tree. A new node replaces an existing NULL node at the bottom of the tree, and has two NULL nodes as children. In the implementation, a NULL node is simply a pointer to a common sentinel node that is colored black. After insertion, the new node is colored red. Then the parent of the node is examined to determine if the red-black tree properties have been violated. If necessary, we recolor the node and do rotations to balance the tree.

By inserting a red node with two NULL children, we have preserved black-height property (property 4). However, property 3 may be violated. This property states that both children of a red node must be black. Although both children of the new node are black (they're NULL), consider the case where the parent of the new node is red. Inserting a red node under a red parent would violate this property. There are two cases to consider:

- Red parent, red uncle: Fig (a) illustrates a red-red violation. Node X is the newly inserted node, with both parent and uncle colored red. A simple recoloring removes the red-red violation. After recoloring, the grandparent (node B) must be checked for validity, as its parent may be red. Note that this has the effect of propagating a red node up the tree. On completion, the root of the tree is marked black. If it was originally red, then this has the effect of increasing the black height of the tree.
- Red parent, black uncle: Fig (b) illustrates a red-red violation, where the uncle is colored black. Here the nodes may be rotated, with the subtrees adjusted as shown. At this point the algorithm may terminate as there are no red-red conflicts and the top of the sub tree (node A) is colored black. Note that if node X were originally a right child, a left rotation would be done first, making the node a left child.



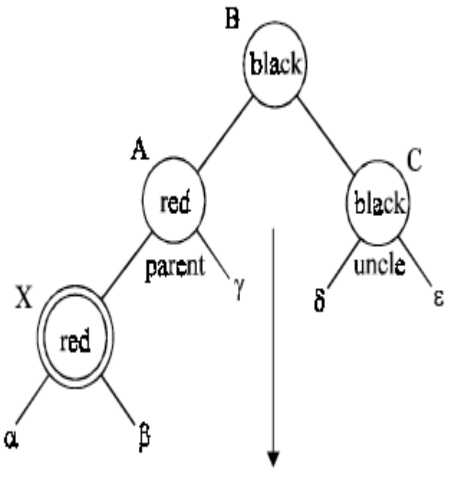


Fig (a) **Fig (b)**

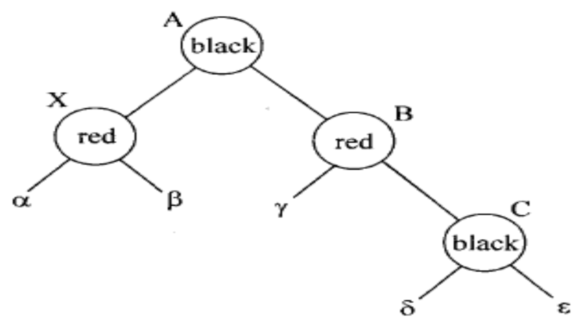


Fig (b)

6.7.9. SPLAY TREES

A splay tree is a self-balancing binary search tree with the additional unusual property that recently accessed elements are quick to access again. It performs basic operations such as insertion, search and removal in O(log(**n**)) amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. Daniel Sleator and Robert Tarjan invented the splay tree.

All normal operations on a splay tree are combined with one basic operation, called *splaying*, also called rotations. That is the efficiency of splay trees comes not from an explicit structural constraint, as with balanced trees, but from applying a simple restructuring heuristic, called splaying, whenever the tree is accessed. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.

One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a bottom-up algorithm can combine the search and the tree reorganization. There are several ways in which splaying can be done. It always involves interchanging the root with the node in operation. One or more other nodes might change position as well. The purpose of splaying is to minimize the number of (access) operations required to recover desired data records over a period of time.

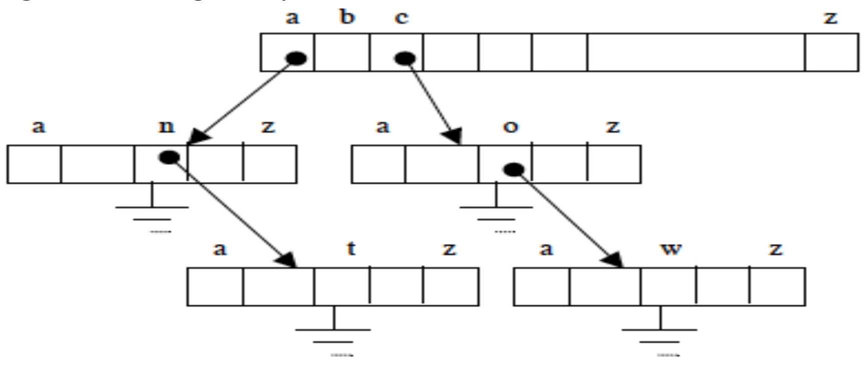
6.7.10. TRIES

The *trie* is a data structure that can be used to do a fast search in a large text. And a trie (from retrieval), is a multi-way tree structure useful for storing strings over an alphabet and it is introduced by Fredkin in 1960's. It has been used to store large dictionaries of English (say) words in spelling-checking programs and in natural-language "understanding" programs.

A trie is an ordered tree data structure that is used to store an associative array where the datas (or key or information) are strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest.

That is no node in the trie contains full characters (or information) of the word (which is a key). But each node will contain associate characters of the word, which may ultimately lead to the full word at the end of a path. For example consider the data: an, ant, cow, the corresponding trie would look like be in following Figure, which is a non-compact trie.
The idea is that all strings sharing a common stem (or character or prefix) hang off a common node. When the strings are words over {a, z}, a node has at most 27 children —one for each letter

plus a terminator.
The elements in a string can be recovered in a scan from the root to the leaf that ends a string. All strings in the trie can be recovered by a depth-first scan of the tree. The height of a trie is the length of the longest key in the trie.



ADVANTAGES AND DISADVANTAGES

There are three main advantages of tries over binary search trees (BSTs):

- Searching a data is faster in tries. Searching a key (or data) of length m takes worst case O(m) = O(1) time; where BST takes O(log n) time, because initial characters are examined repeatedly during multiple comparisons. Also, the simple operations tries use during search, such as array indexing using a character, are fast on real machines.
- Tries require less space. Because the keys are not stored explicitly, only an amortized constant amount of space is needed to store each key.
- Tries make efficient longest-prefix matching, where we wish to find the key sharing the longest possible prefix with a given key. They also allow one to associate a value with an entire group of keys that have a common prefix.

Although it seems restrictive to say a trie's key type must be a string, many common data types can be seen as strings; for example, an integer can be seen as a string of bits. Integers with common bit prefixes occur as map keys in many applications such as routing tables and address translation tables.

6.8. HUFFMAN ALGORITHM

Huffman code is a technique for compressing data. Huffman's greedy algorithm looks at the occurrence of each character and it as a binary string in an optimal way. Suppose we have a data consists of 100,000 characters that we want to compress. The characters in the data occur with following frequencies.

	A	B	C	D	E	F
Frequency	45,000	13,000	12,000	16,000	9,000	5,000

A binary code encodes each character as a binary string or codeword. We would like to find a binary
code that encodes the file using as few bits as possible, ie., compresses it as much as possible. In a fixed-length code each codeword has the same length. In a variable-length code codewords may have different lengths. Here are examples of fixed and variable length codes for our problem (note that a fixed length code must have at least 3 bits per codeword).

	A	B	C	D	E	F
Frequency	45,000	13,000	12,000	16,000	9,000	5,000
Fixed Length code	000	001	010	011	100	101
Variable Length code	0	101	100	111	1101	1100

This method requires 3000,000 bits to code the entire file.
Total number of characters are 45,000 + 13,000 + 12,000 + 16,000 + 9,000 + 5,000 = 1000,000.
Each character is assigned 3-bit codeword => 3 * 1000,000 = 3000,000 bits.
A variable-length code can do better by giving frequent characters short codewords and infrequent characters long codewords.
Character 'A' are 45,000. Each character 'A' assigned 1 bit codeword.
1 * 45,000 = 45,000 bits.
Characters (B, C, D) are 13,000 + 12,000 + 16,000 = 41,000. Each character assigned 3 bit codeword
3 * 41,000 = 123,000 bits
Characters (E, F) are 9,000 + 5,000 = 14,000. Each character assigned 4 bit codeword.
4 * 14,000 = 56,000 bits.
Implies that the total bits are: 45,000 + 123,000 + 56,000 = 224,000 bits.
Variable length code save approximately 25% than fixed length code word. Fixed-length code requires more memory while variable code requires les memory for storage.

Encoding
Given a code (corresponding to some alphabet Z) and a message it is easy to encode the message. Just replace the characters by the codewords.
Example: Z = {a, b, c, d} and if the code is C_1 = {a=00,b=01,c=10,d=11} then **bad** is encoded into 010011

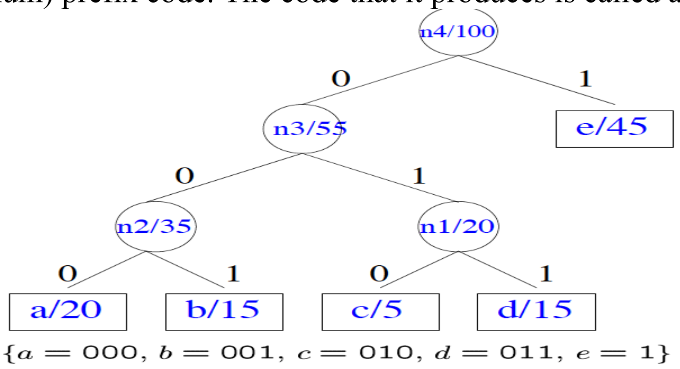
Decoding:
C_1= {a= 00,b=01,c=10,d=11}
C_2= {a= 0,b=110,c=10,d=111}
C_3 ={a= 1, b=110,c=10,d=111}
Given an encoded message, decoding is the process of turning it back into the original message.
A message is uniquely decodable if it can only be decoded in one way. For example relative to C_1, 010011 is uniquely decodable to **bad**. But, relative to C_2, 1101111 is not uniquely decipherable since it could have encoded either **bad** or **acad**.
In fact, one can show that every message encoded Using C_1 and C_2 are uniquely decipherable. The unique decipherability property is needed in order for a code to be useful.

Prefix Codes
Fixed-length codes are always uniquely decipherable. We saw before that these do not always give the best compression so we prefer to use variable length codes.
Prefix Code: A code is called a prefix (free) code if no codeword is a prefix of another one.
Example: {a = 0, b=10, c=10, d=111} is a prefix code.
Important Fact: Every message encoded by a prefix free code is uniquely decipherable. Since no codeword is a prefix of any other. We can always find the first codeword in a message, peel it off, and continue decoding. Example:
01101100 = 01101100 = abba

6.8.1. Optimum Source Coding Problem
The problem: Given an alphabet $A= \{a_1, \ldots\ldots\ldots, a_n\}$ with frequency distribution $f(a_i)$ find a binary prefix code C for A that minimizes the number of bits.

$B(C) = \sum_{a=1}^{n} f(a_i)\, L(C(a_i))$ needed to encode a message of $B(C) = \sum_{a=1}^{n} f(a_i)$ characters, where $C(a_i)$ is the codeword for encoding a_i and $L(C(a_i))$ is the length of the codeword of $C(a_i)$.

Remark: Huffman developed a nice greedy algorithm for solving this problem and producing a minimum cost (optimum) prefix code. The code that it produces is called a Huffman code.



$\{a = 000, b = 001, c = 010, d = 011, e = 1\}$

Correspondence between Binary Trees and prefix codes. 1-1 correspondence between leaves and characters. Label of leaf is frequency of character. Left edge is labeled 0; right edge is labeled 1. Path from root to leaf is code word associated with character.

Note that $d(a_i)$ the depth of leaf a_i in tree T is equal to the length, $L(C(a_i)$ of the codeword in code C associated with that leaf. So $\sum_{a=1}^{n} f(a_i)\, L(C(a_i)) = \sum_{a=1}^{n} f(a_i)\, d(a_i)$

The sum $\sum_{a=1}^{n} f(a_i)\, d(a_i)$ is the weighted external path length of tree T.

6.8.2. Huffman coding

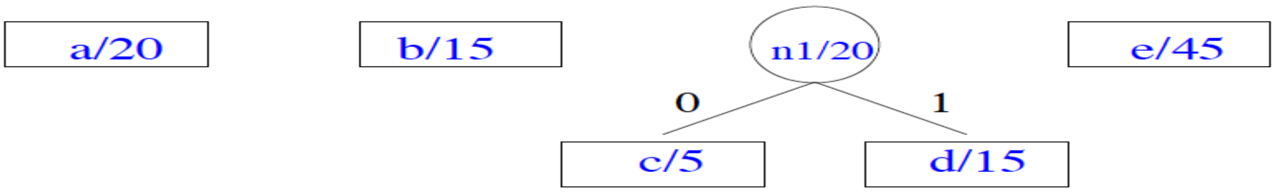
Step 1: Pick two letters x and *y* from Alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea) Label the root of this subtree as *z.*

Step 2: Set frequency $f(z) = f(x) + f(y)$:Remove *x,y* and add *z* creating new alphabet . $A' = A \cup \{Z\} - \{x,y\}$

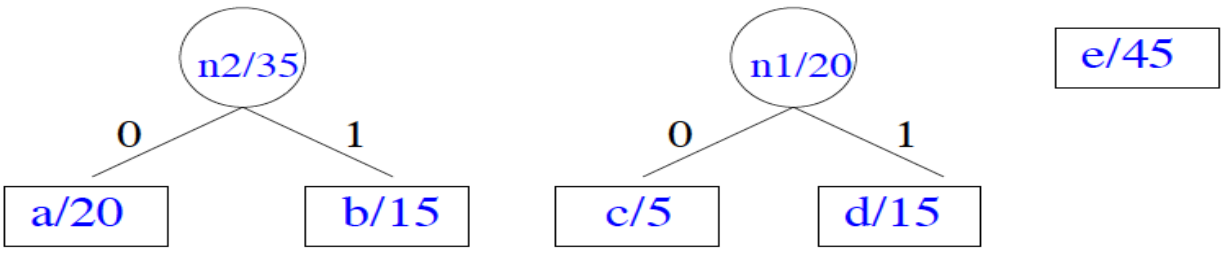
Note that mod$|A'| = |A|$ -1. Repeat this procedure, called merge, with new alphabet A′ until an alphabet with only one symbol is left. Then the resulting tree is the Huffman.

Example of Huffman Coding

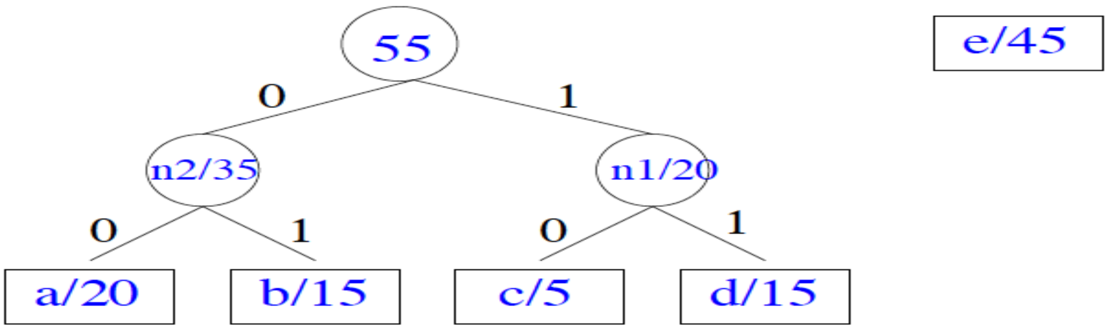
Let A = {a/20, b/15, c/5, d/15, e/45} be the alphabet and its frequency distribution.
In the first step Huffman coding **merge c and d.**



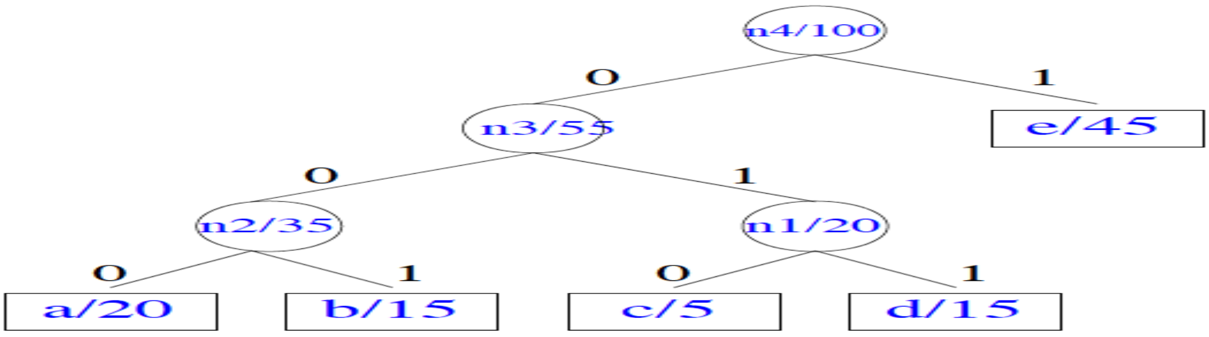
Alphabet is now A1 = {a/20, b/15, n1/20,e/45}. Algorithm **merges a and b**



Alphabet is now A2 = { n2/35,n1/20, e/45}. Algorithm **merge n1 and n2**.



Alphabet is now A3 = {n3/55, e/45}.
Algorithm merges n3 and e and finished.



Huffman code a=000, b=001, c = 010, d= 011, e=1

6.8.3. Huffman Algorithm

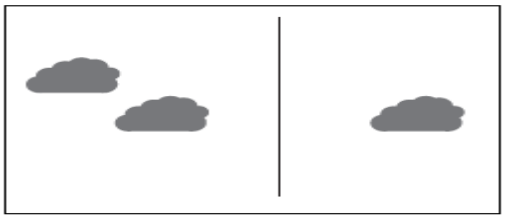
Given an alphabet A with frequency distribution $\{f(a) : a \in A\}$. The binary Huffman tree is constructed using a priority queue, *Q*, of nodes, with labels (frequencies) as keys.

```
Huffman (A)
{
    n=|A|;
    Q = A; (Future leafs)
    For i = 1 to n-1
    {
        z = new node;
        left[z] = Extract-Min(Q);
        right[z] = Extract-Min(Q);
        f[z] = f[left[z]] + f[right[z]]
        Insert(Q, z);
    }
    return Extract-Min(Q);
    }
```

6.9. Game Tree

A game tree isn't a special new data structure—it's a name for any regular tree that maps how a discrete game is played.

Lets take an example of game called Rocks. In this game, we have different piles of rocks, with one or more rocks in each pile. The game has two players, who take turns taking one or more rocks from a single pile until one pile is left. When one pile is left, our goal is to force our opponent to remove the last rock. The person who removes the last rock loses.



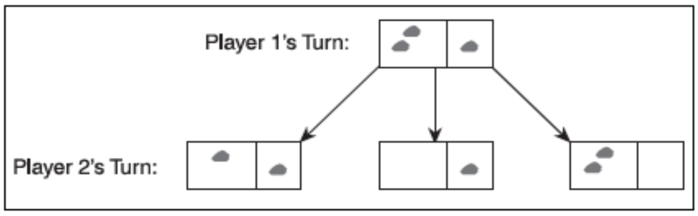


Fig24: Simple game of rock with 2 piles Fig25: First 2 levels of game tree

In the figure, there are two piles. The first pile has two rocks, and the second pile has only one rock. The first player has three choices:

- Remove one rock from pile 1.
- Remove two rocks from pile 1.
- Remove one rock from pile 2.

We can start the game off with one of those three moves. We can create a simple game tree to represent these moves, as shown in Fig25. After Player 1 has moved, it is now Player 2's turn. Player2's choice of a move is limited to the current state of the game, however. In the leftmost state of Fig24, Player 2 has two choices: He can remove one rock from pile 1 or one rock from pile 2. His choice for the middle state is even less useful He can only remove one rock from pile 2. Of course, because this is the last rock, Player 2 has lost the game. On the right state, Player 2 has two options again: He can remove one or two rocks from pile 2.

Fig26 shows the game tree for all five of these moves and goes down one more level to show the complete game tree.

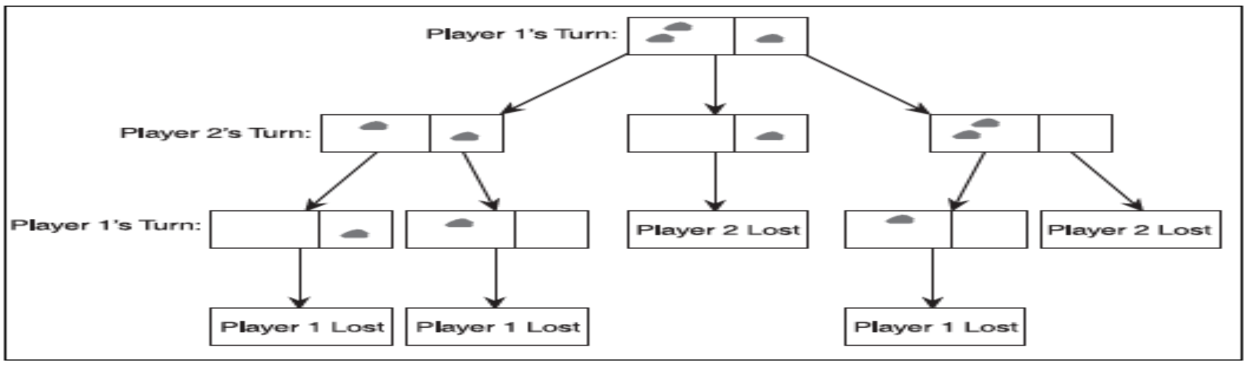


Fig26: A Complete Game tree

The game is entirely complete by the time the fourth level is reached. The game can have up to three moves because there were only three rocks. We can also tell from the tree that there are five total outcomes from the game because there are five leaf nodes. The game always ends on a leaf node because there are no more moves that can be made. So what can we tell about the game tree that we couldn't easily tell about the initial game setup?

For Player1, the obvious first move is the second one, removing the two stones from pile 1. By doing that, Player 2 forced to be lost the game, because he has no other option and cannot possibly win. Another thing that we would notice if Player 1 plays the leftmost move, removing one rock from pile 1, is a death sentence. If Player 1 makes that move, then he have given Player 2 a free win, because no matter what move he makes, there is no chance for Player 1 to win in that branch. If Player 1 takes the third route on the opening move, then Player 2 decides the outcome of the game. If Player 2 removes both rocks in pile 1 (a very stupid move), he loses. If he only removes one rock, then he forces Player 1 to remove the last one, and he loses.

6.10. Applications of Tree Data Structure

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1. One reason to use trees might be because we want to store information that naturally forms a hierarchy. For example, the file system on a computer:
2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for search.
3. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for insertion/deletion.
4. Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

In Short, following are the common uses of tree data structure.

1. Manipulate hierarchical data.
2. Make information easy to search.
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

Other Applications

1. *Binary Search Tree*: Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
2. *Hash Trees* - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
3. *Heaps* - Used in heap-sort; fast implementations of Dijkstra's algorithm; and in implementing efficient priority-queues, which are used in scheduling processes in many operating systems, Quality-of-Service in routers, and A* (path-finding algorithm used in AI applications, including video games).
4. *Huffman Coding Tree* - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
5. Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.
6. Treap - Randomized data structure used in wireless networking and memory allocation.
7. T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so

Reference

6. V.V. Das; "*Principles of Data Structures Using C and C++";* New Age International Publishers, New Delhi, India.
7. Y Langsam, MJ, Augenstein and A.M, Tanenbaum; "*Data Structure Using C and C++*"; Prentice Hall India.
8. P. Ron; "*Data Structure for Game Programmers*"; Premier Press, Game development. 2003, United States of America, Ohio;
9. J. R., Alexander; L. K. Robert; "*Data Structures and Program Design in C++*"; Prentice Hall, Upper Saddle River, New Jersey 07458

Chapter 7: Sorting

Sorting is used to arrange names and numbers in meaningful ways. Let A be a list of n elements A1, A2, ... An in memory. Sorting of list A refers to the operation of rearranging the contents of A so that they are in increasing (or decreasing) order (numerically or lexicographically); A1 < A2 < A3 < ...... < An.
There are many kinds of sorting techniques are available in computer science. Some of them are as
Bubble sort, Selection sort, Quick sort, Heap sort, Merge sort, Binary sort, Shell sort, Radix sort, Exchange sort, Internal sort, Insertion sort etc.
It is very difficult to select a sorting algorithm over another. And there is no sorting algorithm better than all others in all circumstances. Some sorting algorithm will perform well in some situations, so it is important to have a selection of sorting algorithms. Some factors that play an important role in selection processes are the time complexity of the algorithm (use of computer time), the size of the data structures (for Eg: an array) to be sorted (use of storage space), and the time it takes for a programmer to implement the algorithms (programming effort).

Complexities of sorting algorithms
The complexity of sorting algorithm measures the running time of n items to be sorted. The operations in the sorting algorithm, where A1, A2 ….. An contains the items to be sorted and B is an auxiliary location, can be generalized as:

(a) Comparisons- which tests whether A**i** < A**j** or test whether A**i** < B
(b) Interchange- which switches the contents of A**i** and A**j** or of A**i** and B
(c) Assignments- which set B = A and then set A**j** = B or A**j** = A**i**

Normally, the complexity functions measure only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

7.1. BUBBLE SORT

In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed. Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array until we get a sorted array.

Algorithm

Let A be a linear array of n numbers. tmp is a temporary variable for swapping (or interchange) the position of the numbers.

```
1. Input n numbers of an array A
2. Initialize i = 0 and repeat through step 4 if (i < n)
3. Initialize j = 0 and repeat through step 4 if (j < n - i - 1)
4. If (A[j] > A[j + 1])
     i.   tmp = A[j]
     ii.  A[j] = A[j + 1]
     iii. A[j + 1] = tmp
5. Display the sorted numbers of array A
6. Exit.
```

The elements of an array A to be sorted are: 42, 33, 23, 74, 44

First Pass

33 swapped	33	33	33
42	23 swapped	23	23
23	42	42 no swapping	42
74	74	74	44 swapped
44	44	44	74

Second Pass

23 swapped	23	23
33	33 no swapping	33
42	42	42 no swapping
44	44	44
74	74	74

Third Pass

23 no swapping	23
33	33 no swapping
42	42
44	44
74	74

Fourth Pass

23 no swapping
33
42
44
74

Thus the sorted array is 23, 33, 42, 44, 74.

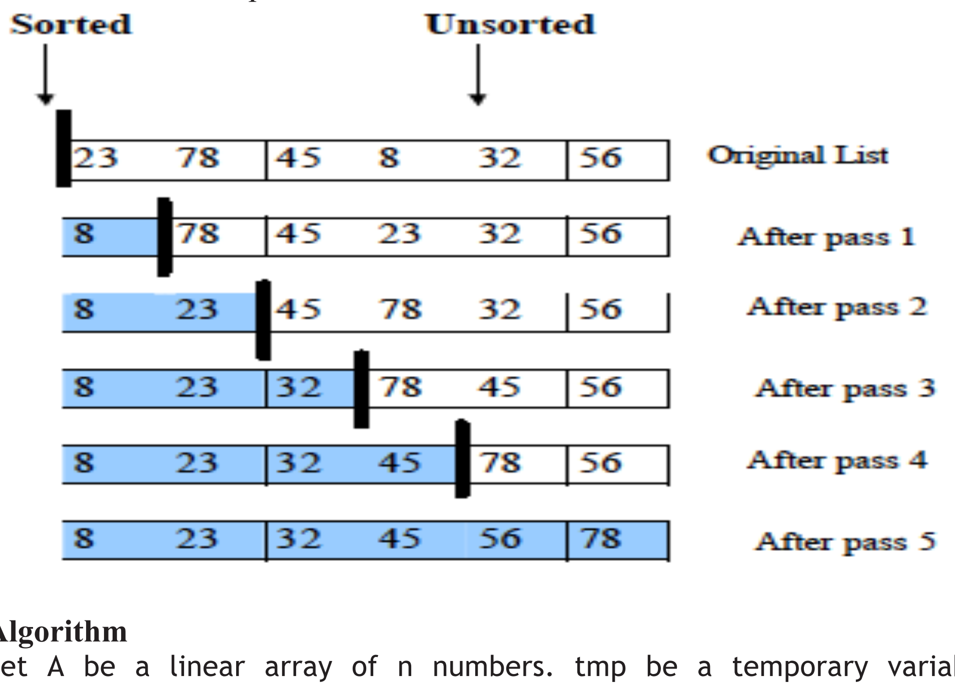
Source Code:

```
/* Bubble sort*/
void bubble_sort(int a[],int n)
{
    int i,j,tmp;
    for (i = 0; i <n-1; i++)
    {
        for (j = 0; j <n-1-i; j++)
        {
            if (arr[j] > arr[j+1])
            {
                tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }//End of if
        }//End of inner for loop
    } //End of outer for loop
} //End of function
```

7.2. SELECTION SORT

- The list is divided into two subsists, sorted and unsorted, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sub list and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sub lists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sub list to the sorted sub list, we say that we have completed a sort pass.
- A list of n elements requires n-1 passes to completely rearrange the data.

Selection Sort Example



Algorithm

Let A be a linear array of n numbers. tmp be a temporary variable for swapping (or interchanging). min is the variable to store the location of smallest number

```
1.  Input n numbers of an array A
2.  Initialize i = 0 and repeat through step5 if (i < n - 1)
        i. min = i
3.  Initialize j = i + 1 and repeat through step 4 if (j < n)
4.  if (min > a[j])
```

```
          i.  min = j
5.
          i.  tmp = a[min]
         ii.  a[min] = a[i]
        iii.  a[i] = tmp
6.    Display "the sorted numbers of array A"
7.    Exit
```

Source Code:

```
void selectionsort(int arr[],int n)
{
    int i,j,temp;
    int min;
    for(i=0;i<n-1;i++)
    {
        min = i;
        for(j=i+1;j<n;j++)
        {
            if(arr[min] > arr[j])
                min = j;
        }
            temp= arr[min];
            arr[min]=arr[i];
            arr[i]=temp;
    }
  }
```

7.3. INSERTION SORT

Insertion sort algorithm sorts a set of values by inserting values into an existing sorted file. Compare the second element with first, if the first element is greater than second, place it before the first one. Otherwise place is just after the first one. Compare the third value with second. If the third value is greater than the second value then place it just after the second. Otherwise place the second value to the third place. And compare third value with the first value. If the third value is greater than the first value place the third value to second place, otherwise place the first value to second place. And place the third value to first place and so on.

Let A be a linear array of n numbers A [1], A [2], A [3], ... A[n]. The algorithm scans the array A from A [1] to A [n], by inserting each element A [k], into the proper position of the previously sorted sub list. A [1], A [2], A [3], ... A [**k** – 1]

Step 1: As the single element A [1] by itself is sorted array.

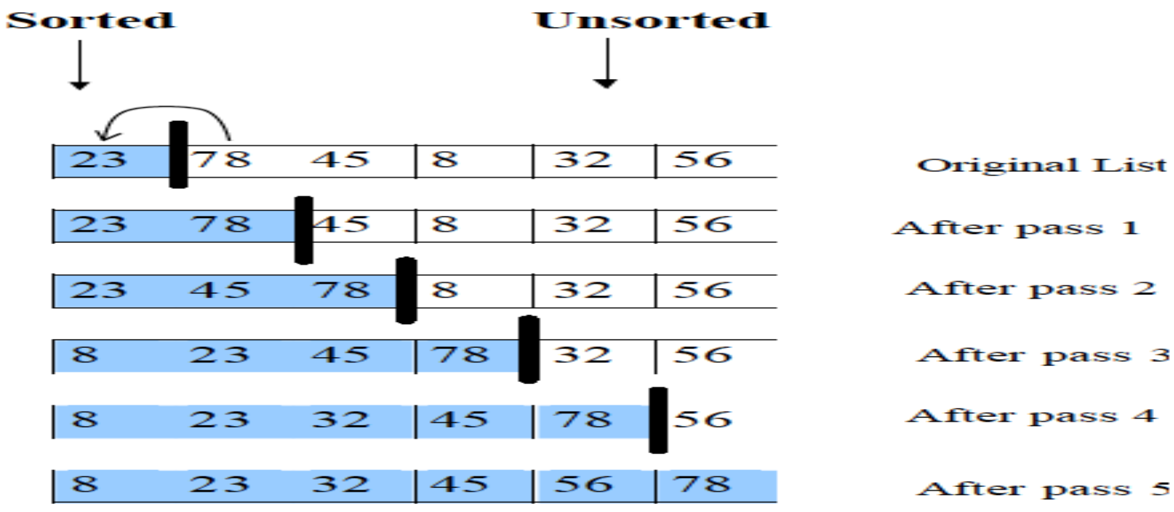
Step 2: A [2] is inserted either before or after A [1] by comparing it so that A[1], A[2] is sorted array.

Step 3: A [3] is inserted into the proper place in A [1], A [2], that is A [3] will be compared with A
[1] and A [2] and placed before A [1], between A [1] and A [2], or after A [2] so that A [1], A
[2], A [3] is a sorted array.

Step 4: A [4] is inserted in to a proper place in A [1], A [2], A [3] by comparing it; so that A [1], A
[2], A [3], A [4] is a sorted array.

Step 5: Repeat the process by inserting the element in the proper place in array

Step n :A [**n**] is inserted into its proper place in an array A [1], A [2], A [3], ... A [**n** –1] so that A
[1], A [2], A [3], ... ,A [**n**] is a sorted array.



Algorithm

Let A be a linear array of n numbers A [1], A [2], A [3], ... ,A [n]......Swap be a temporary variable to interchange the two values. Pos is the control variable to hold the position of each pass.

```
1. Input an array A of n numbers
2. Initialize i = 1 and repeat through steps 4 by incrementing i by one.
     i. If (i < = n - 1)
    ii. tmp = A [i],
   iii. Pos = i - 1
3. Repeat the step 3 if (tmp < A[Pos] and (Pos >= 0))
    iv. A [Pos+1] = A [Pos]
     v. Pos = Pos-1
4. A [Pos +1] = tmp
5. Exit
```

Source Code:

```
/*Insertion sort*/
    void insertion_sort(int arr[], int n)
    {
        int i,j,k;
        for(i=1;i<n;i++)
        {
            k=arr[i]; /*k is to be inserted at proper place*/
            for(j=i-1;j>=0 && k<arr[j];j--)
                arr[j+1]=arr[j];
            arr[j+1]=k;
        }
    }
```

7.4. SHELL SORT

Shell Sort is introduced to improve the efficiency of simple insertion sort. Shell Sort is also called diminishing increment sort. In this method, sub-array, contain kth element of the original array, are sorted. Let A be a linear array of n numbers A [1], A [2], A [3], ... A [n].

Step 1: The array is divided into k sub-arrays consisting of every kth element. Say k= 5, then five sub-
array, each containing one fifth of the elements of the original array.

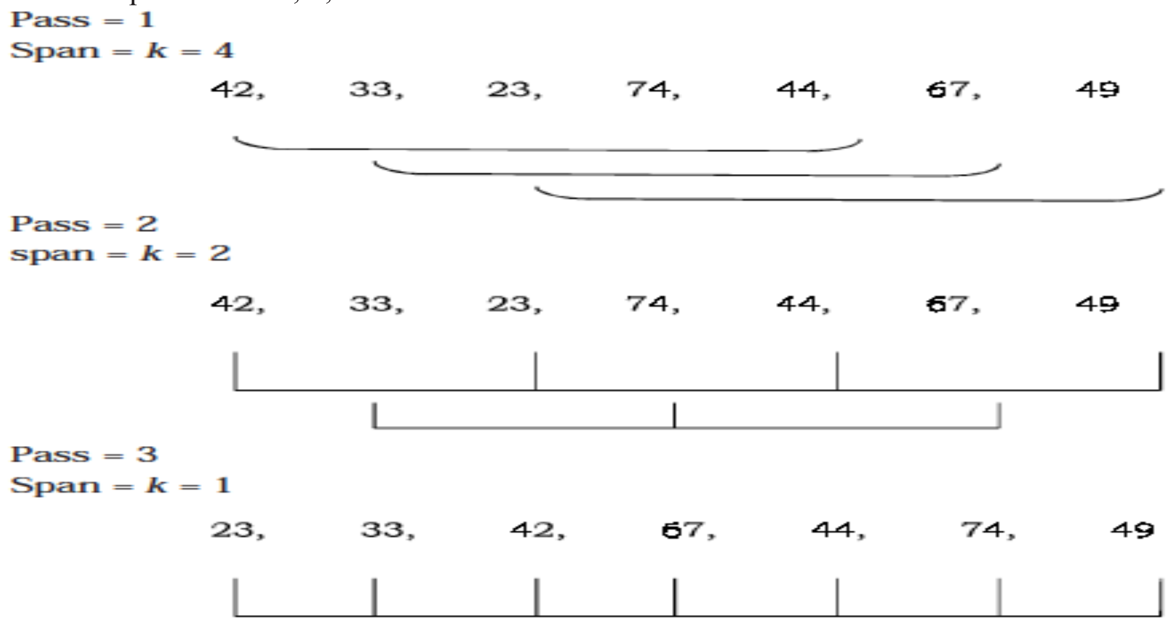
Sub array 1 → A[0] A[5] A[10]
Sub array 2 → A[1] A[6] A[11]
Sub array 3 → A[2] A[7] A[12]
Sub array 4 → A[3] A[8] A[13]
Sub array 5 → A[4] A[9] A[14]

Note : The ith element of the jth sub array is located as A [(i–1) * k+j–1]

Step 2: After the first k sub array are sorted (usually by insertion sort) , a new smaller value of k is
chosen and the array is again partitioned into a new set of sub arrays.

Step 3: And the process is repeated with an even smaller value of k, so that A [1], A [2], A [3], ... A
[n] is sorted.

To illustrate the shell sort, consider the following array with 7 elements 42, 33, 23,74, 44, 67, 49 and the sequence K = 4, 2, 1 is chosen.



ALGORITHM

Let A be a linear array of n elements, A [1], A [2], A [3], ... A[n] and Incr be an array of sequence of span.

```
1. Input n numbers of an array A
2. Input incr and repeat through step 5 until (incr >=1)
3. Initialize j = incr and repeat through step 5 until ( j < n)
        (a) k = A [ j ]
4. Initialize i = j-incr and repeat through step 5 if (i > = 0) and (k < A [i ])
        (a) A [i + incr] = A [i]
5. A [i + incr] = k
6. Exit
```

Source Code:

```
void shell_sort(int arr[],int n)
{
    int incr, i,j,k,
    scanf("%d",&incr);
```

```
        while(incr>=1)
        {
            for (j=incr;j<n;j++)
            {
                k=arr[j];
                for(i=j-incr; i >= 0 && k < arr[i]; i = i-incr)
                    arr[i+incr]=arr[i];
                arr[i+incr]=k;
            }
            incr=incr-2; /*Decrease the increment*/
        }/*End of while*/
      }
```

7.5. Quick Sort

It is one of the widely used sorting techniques and it is also called the partition exchange sort. Quick sort is an efficient algorithm and it passes a very good time complexity in average case. It is an algorithm of the **divide-and conquers** type. The quick sort algorithm works by partitioning the array to be sorted. And each partition is internally sorted recursively. This key value can be the first element of an array. That is, if A is an array then key = A[0], and rest of the elements are grouped into two portions such that,

a) One partition contains elements smaller than key value.
b) Another partition contains elements larger than key value.

Two pointers, Up and Low, are initialized to the upper and lower bounds of the sub array. During execution, at any point each element in a position above up is greater than or equal to key value and each element in a position below low pointer is less than or equal to the key. Up pointer will move in a decrement and low in an increment fashion.

Let A be an array A[1], A[2], A[3],….A[n] of n numbers to be sorted. Then

Step1: Choose the first elements of the array(or sub-array) as the key i.e. key = A[1].
Step2: Place the low pointer in second position of the array and up pointer in the last position of the
array i.e. low = 2 and up = n.
Step3: Repeatedly increase the pointer low by one position until A[low] > key.
Step4: Repeatedly decrease the pointer up by one position until A[up] <=key.
Step5: If up >low, interchange A[low] with A[up].
Step6: Repeat the step 3,4,5 until the step 5 fails.(i,e. up <=low)
Note: The given array is partition into two sub arrays. The sub array A[1] A[2]…..A[k-1] is
less than A[k] i.e. key and the second sub array A[k+1] A[k+2]……A[n] which is greater than
a key value A[k].

Lets take an example of the following data and i and j as the low and up pointer.

```
40    20    10    80    60    50    7    30    100
```

We took first element as a key element that is 40.

```
40    20    10    80    60    50    7    30    100
Key   i                                       j
```

Now i stops at 80 since A[i] > key and j stops at 30 since A[j] <= key
As i<j, swap A[i] and A[j]

```
40    20    10    30    60    50    7    80    100
                  i                        j
```

again i stops at 60 since A[i] > key and j stops at 7 since A[j] <=key.
As i<j, swap A[i] and A[j]

```
40   20   10   30   7    50   60   80   100
                    i         j
```

Now i stops at 50 and j stops at 7, since A[i] >key and j stops at 7 since A[j] <=key
As i>j, no swap between A[i] and A[j] but swap A[j] with key element.

```
7    20   10   30   40   50   60   80   100
     data < key     key       data >= key
```

Now 2 Sub-array will be sort recursively.

Algorithm

Let A be a linear array of n elements A[1], A[2], A[3]......A[n], low represents the lower bound pointer and up represents the upper bound pointer. Key represents the first element of the array, which is going to become the middle element of the sub-arrays.

```
1. Input n number of elements in an array A
2. Initialize low = 1, up = n , key = A[0]
3. Repeat through step 8 while (low < up)
4. Repeat step 5 while(A [low] > key)
5. low = low + 1
6. Repeat step 7 while(A [up] <= key)
7. up = up-1
8. If (low <= up)
      (a) Swap = A [low]
      (b) A [low] = A [up]
      (c) A [up] = swap
      (d) low=low+1
      (e) up=up-1
9.  Quick sort (A, low, key-1)
10. Quick sort (A, key+1,up)
11. Exit
```

Source Code:

```
void quick_sort(int arr[],int low,int up)
{
    int piv,temp,left,right;
    enum bool pivot_placed=FALSE;
    //setting the pointers
    left=low;
    right=up;
    piv=low; /*Take the first element of sublist as piv */
    if (low>=up)
        return;
    /*Loop till pivot is placed at proper place in the sublist*/
    while(low < up)
    {
        /*Compare from right to left */
        while(arr[piv]<=arr[right])
            right=right-1;
        if (arr[piv] > arr[right] )
        {
            temp=arr[piv];
            arr[piv]=arr[right];
            arr[right]=temp;
            piv=right;
        }
        /*Compare from left to right */
        while( arr[piv]>arr[left])
            left=left+1;
        if ( arr[piv] < arr[left] )
        {
            temp=arr[piv];
            arr[piv]=arr[left];
            arr[left]=temp;
            piv=left;
        }
    }/*End of while */
    quick(arr,low,piv-1);
    quick(arr,piv+1,up);
}/*End of quick()*/
```

7.6. MERGE SORT

Merging is the process of combining two or more sorted array into a third sorted array. Divide the array into approximately n/2 sub-arrays of size two and set the element in each sub array. Since at any time the two arrays being merged are both sub-arrays of A, lower and upper bounds are required to indicate the sub-arrays of a being merged. l1 and u1 represents the lower and upper bands of the first sub-array and l2 and u2 represents the lower and upper bands of the second sub-array respectively. Let A be an array of n number of elements to be sorted A[1], A[2] ...... A[n].

Step 1: Divide the array A into approximately n/2 sorted sub-array of size 2. i.e., the elements in the

(A [1], A [2]), (A [3], A [4]), (A [k], A [k + 1]), (A [n – 1], A [n]) sub-arrays are in sorted order.

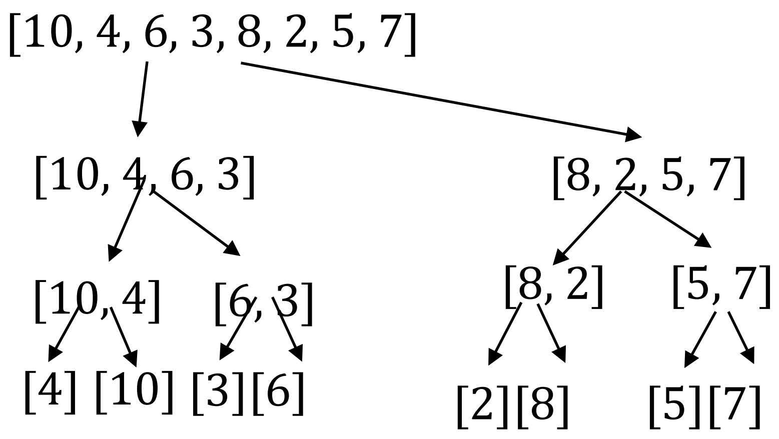
Step 2: Merge each pair of pairs to obtain the following list of sorted sub-array of size 4; the elements

in the sub-array are also in the sorted order.(A [1], A [2], A [3], A [4)),...... (A [k – 1], A [k], A

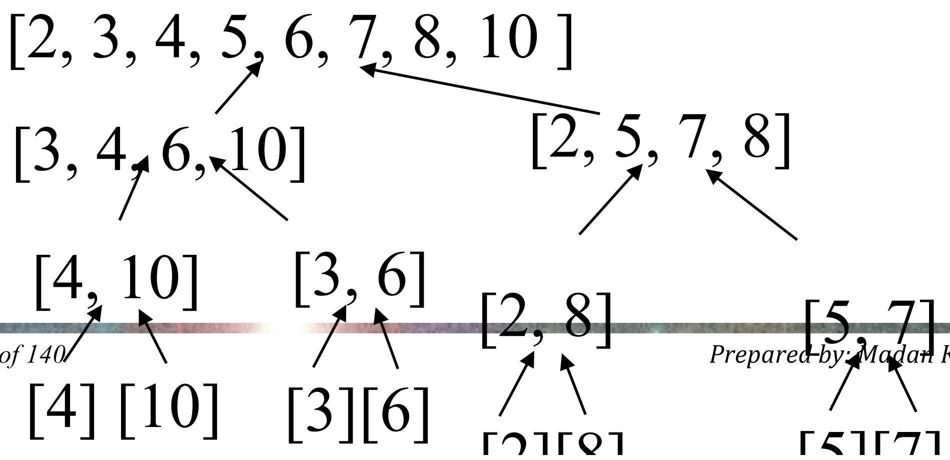
[k + 1], A [k + 2]),...... (A [n – 3], A [n – 2], A [n – 1], A [n].

Step 3: Repeat the step 2 recursively until there is only one sorted array of size.

Partition of list of size n/2



Merging of n/2 partition.



Algorithm of Merge sort

Input: Array A[1…N], indices p, q,r(p ≤ q <r). A[p…r]is the array to be divided. A[p] is the beginning element and A[r] is the ending element
Output: Array A[p…r] in ascending order

```
MERGE-SORT(A,p,q,r)

   1.  if p <r
   2.  then q ← (r + p)/2
   3.  MERGE-SORT(A, p, q )
   4.  MERGE-SORT(A,q+1,r)
   5.  MERGE(A, p, q, r)

MERGE(A, p, q, r)
   6.  n1←q-p+1
   7.  n2←r-q
   8.  create arrays L[1...N1+1] and R[1...N2+1]
   9.  for i←1 to N1
   10.          do L[i] ← A[p+i-1]
   11. for j ← 1 to n2
   12.          do R[j] ← A[q+j]
   13.          L[N1+1] ← ∞
   14.          R[N2+1] ← ∞
   15. i ← 1
   16. j ← 1
   17. for k ← p to r
   18.     do if L[i] ≤ R[j]
   19.           then A[k] ← L[i]
   20.                 i ← i+1
   21.           else A[k] ← R[j]
   22.                 j ← j+1
```

7.7. Radix Sort

Radix sort or bucket sort is a method that can be used to sort a list of numbers by its base. It is also called as clever sort. To sort an array of decimal numbers, where the radix or base is 10, we need 10 buckets and can be numbered as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Number of passes required to have a sorted array depends upon the number of digits in the largest element. To illustrate the radix sort, consider the following array with 7 elements:

42, 133, 7, 23, 74, 670, 49

In this array the biggest element is 670 and the number of digit is 3. So 3 passes are required to sort the array. Read the element(s) and compare the first position (2 is in first position of 42) digit with the digit of the bucket and place it.

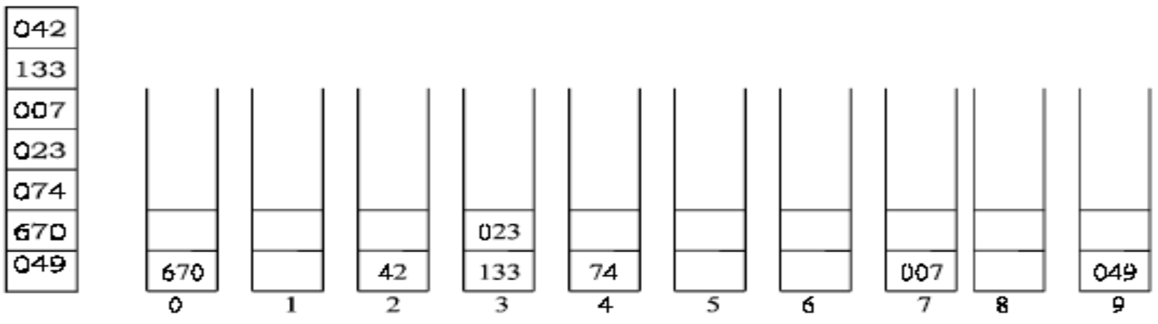


Fig: Radix sort Pass 1

Now read the elements from left to right and bottom to top of the buckets and place it in array for the next pass. Read the array element and compare the second position (4 is in second position of the element 042) digit with the number of the bucket and place it.

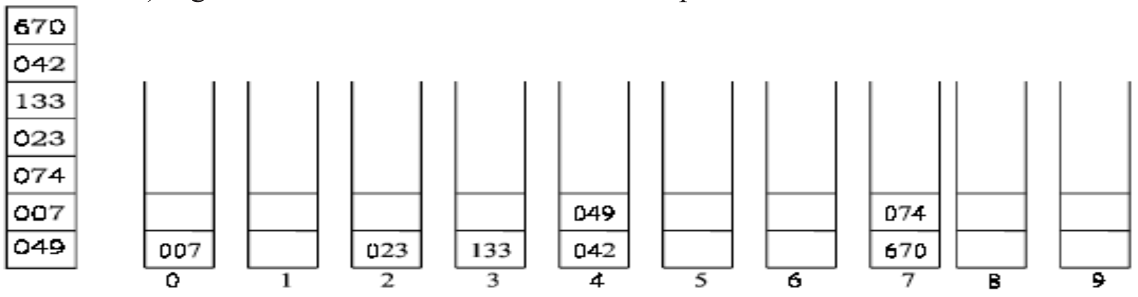


Fig: Radix Sort Pass 2

Again read the element from left to right and from bottom to top to get an array for the third pass. (0 is in the third position of 042) Compare the third position digit in each element with the bucket digit and place it wherever it matches.

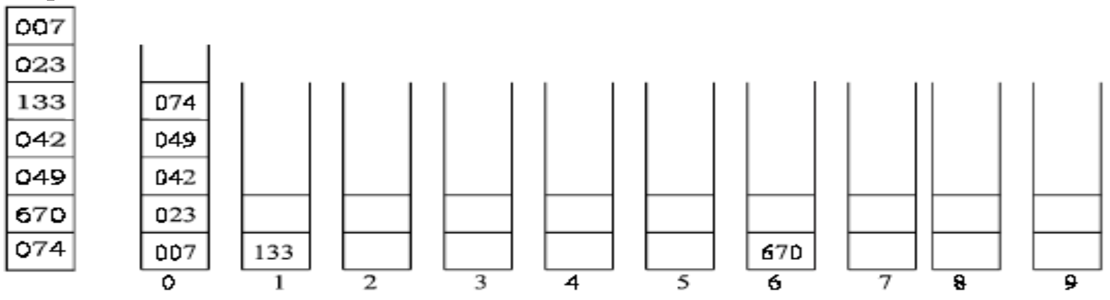


Fig: Radix Sort Pass 3

Read the element from the bucket from left to right and from bottom to top for the sorted array. i.e., 7 23, 42, 49, 74, 133, 670.

Algorithm (Radix Sort)

Let A be a linear array of n elements A[1],A [2],A [3],...... A[n]. Digit is the total number of digits in the largest element in array A.

```
1. Input n number of elements in an array A.
2. Find the total number of Digits in the largest element in the array.
3. Initialize i = 1 and repeat the steps 4 and 5 until (i <= Digit).
4. Initialize the buckets j = 0 and repeat the steps (a) until (j < n)
       (a) Compare ith position of each element of the array with bucket number and
           place it in the corresponding bucket.
5. Read the element(s) of the bucket from 0th bucket to 9th bucket and from first position
   to higher one to generate new array A.
6. Display the sorted array A.
7. Exit.
```

Source Code: Using Linked List

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

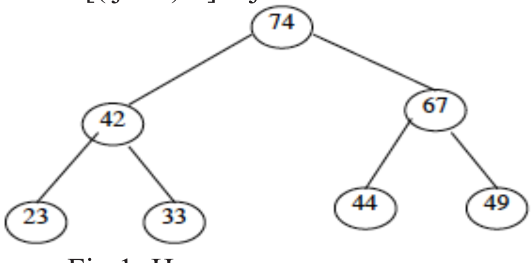
```
#include "stdlib.h"
struct node
{
    int info ;
    struct node *link;
}*start=NULL;
//Display the array elements
void display()
{
    struct node *p=start;
    while( p !=NULL)
    {
        printf ("%d ", p->info);
        p= p->link;
    }
    printf ("\n");
}/*End of display()*/
/* This function finds number of digits in the largest element of
the list */
int large_dig(struct node *p)
{
    int large = 0,ndig = 0 ;
    while (p != NULL)
    {
        if (p ->info > large)
            large = p->info;
        p = p->link ;
    }
    printf ("\nLargest Element is %d ,",large);
    while (large != 0)
    {
        ndig++;
        large = large/10 ;
    }
    printf ("\nNumber of digits in it are %d\n",ndig);
    return(ndig);
} /*End of large_dig()*/
/*This function returns kth digit of a number*/
int digit(int number, int k)
{
    int digit, i ;
    for (i = 1 ; i <=k ; i++)
    {
        digit = number % 10 ;
        number = number /10 ;
    }
    return(digit);
}/*End of digit()*/
//Function to implement the radix sort algorithm
void radix_sort()
{
    int i,k,dig,maxdig,mindig,least_sig,most_sig;
    struct node *p, *rear[10], *front[10];
    least_sig=1;
    most_sig=large_dig(start);
    for (k = least_sig; k <= most_sig ; k++)
    {
        printf ("\nPASS %d : Examining %dth digit from right",k,k);
        for(i = 0 ; i <= 9 ; i++)
        {
            rear[i] = NULL;
            front[i] = NULL ;
        }
        maxdig=0;
        mindig=9;
        p = start ;
        while( p != NULL)
        {
            /*Find kth digit in the number*/
            dig = digit(p->info, k);
            if (dig>maxdig)
                maxdig=dig;
            if (dig<mindig)
                mindig=dig;
            /*Add the number to queue of dig*/
            if (front[dig] == NULL)
                front[dig] = p ;
            else
                rear[dig]->link = p ;
            rear[dig]= p ;
            p=p->link;/*Go to next number in the list*/
        }/*End while */
        /* maxdig and mindig are the maximum amd minimum
         digits of the kth digits of all the numbers*/
        printf ("\nmindig=%d maxdig=%d\n",mindig,maxdig);
        /*Join all the queues to form the new linked list*/
        start=front[mindig];
        for (i=mindig;i<maxdig;i++)
        {
            if (rear[i+1]!=NULL)
                rear[i]->link=front[i+1];
            else
                rear[i+1]=rear[i];
        }
        rear[maxdig]->link=NULL;
        printf ("\nNew list :");
        display();
    }/* End for */
}/*End of radix_sort*/
int main()
{
    struct node *tmp,*q;
    int i,n,item;
    printf ("\nEnter the number of elements in the list : ");
    scanf ("%d", &n);
    for (i=0;i<n;i++)
    {
        printf ("\nEnter element %d :",i+1);
        scanf ("%d",&item);
        /* Inserting elements in the linked list */
        tmp=(struct node*)malloc(sizeof(struct node));
        tmp->info=item;
        tmp->link=NULL;
        if (start==NULL) /* Inserting first element */
```

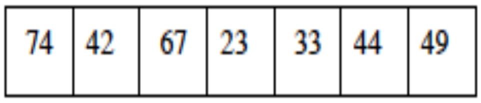
```
            start=tmp;
        else
        {
            q=start;
            while(q->link!=NULL)
                q=q->link;
            q->link=tmp;
        }
    }/*End of for*/
    printf ("\nUnsorted list is :\n");
    display();
    radix_sort();
    printf ("\nSorted list is :\n");
    display ();
}/*End of main()*/
```

7.8. Heap

A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of the father. It can be sequentially represented as

A[j] <= A[(j – 1)/2]
for 0 <= [(j – 1)/2] < j <= n – 1





74	42	67	23	33	44	49

Fig 1: Heap Fig 2: Array representation

The root of the binary tree (i.e., the first array element) holds the largest key in the heap. This type of heap is usually called descending heap or **max heap**. We can also define an ascending heap as an almost complete binary tree in which the value of each node is greater than or equal to the value of its father. This root node has the smallest element of the heap. This type of heap is also called **min heap**.

Priority Queue: A priority queue is an ADT with the following operations:
- Insert a new element into the queue
- Find the element with the largest key
- Delete the element with the largest key

Other common operations:
- Heaps provide an efficient implementation of priority queues:
- get the maximum → take the root

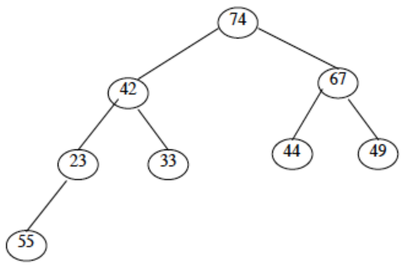
- delete the maximum → move the last element to the root and heapify.
- insert a new element → put it at the end and raise it until it's in place

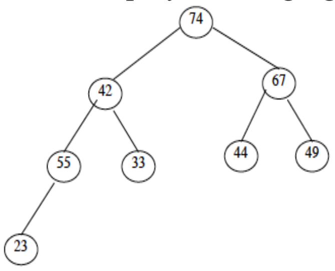
Heap Sort

A heap can be used to sort a set of elements. Let H be a heap with n elements and it can be sequentially represented by an array A. Inset an element data into the heap H as follows:
1. First place data at the end of H so that H is still a complete tree, but not necessarily a heap.
2. Then the data be raised to its appropriate place in H so that H is finally a heap.

Inserting an element in to a heap

Consider the heap H in Fig1. Say we want to add a data = 55 to H.
Step 1: First we adjoin 55 as the next element in the complete tree as shown in Fig3. Now we have to find the appropriate place for 55 in the heap by rearranging it.





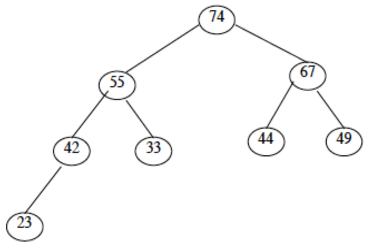


Fig3 Fig4

Fig5

Step 2: Compare 55 with its parent 23. Since 55 is greater than 23, interchange 23 and 55. Now the heap will look like as in Fig4.
Step 3: Compare 55 with its parent 42. Since 55 is greater than 42, interchange 55 and 42. Now the heap will look like as in Fig5.
Step 4: Compare 55 with its new parent 74. Since 55 is less than 74, it is the appropriate place of node 55 in the heap H.

Deleting the Root of a Heap

Let H be a heap with n elements. The root R of H can be deleted as follows:
Step1: Assign the root R to some variable data.
Step2: Replace the deleted node R by the last node (or recently added node) L of H so that H is still a complete tree, but not necessarily a heap.
Step3: Now rearrange H in such a way by placing L (new root) at the appropriate place, so that H is finally a heap.

Consider the heap H in Fig5 where R = 74 is the root and L = 23 is the last node (or recently added node) of the tree. Suppose we want to delete the root node R = 74. Apply the above rules to delete the root. Delete the root node R and assign it to data (i.e., data =74) as shown in Fig6.

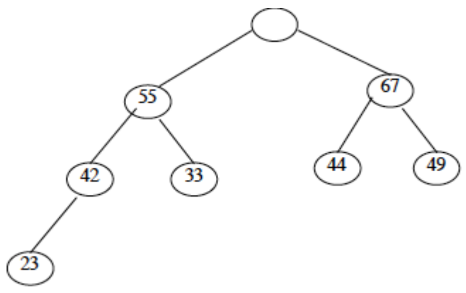


Fig6

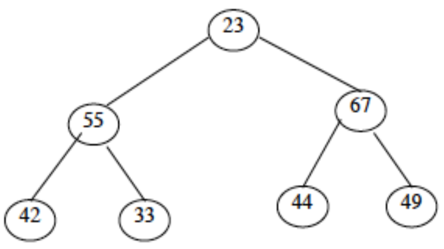


Fig7

Replace the deleted root node R by the last node L as shown in the Fig7.Compare 23 with its new two children 55 and 67. Since 23 is less than the largest child 67, interchange 23 and 67. The new tree looks like as in Fig8. Again compare 23 with its new two children, 44 and 49. Since 23 is less than the largest child 49, interchange 23 and 49as shown in Fig9.

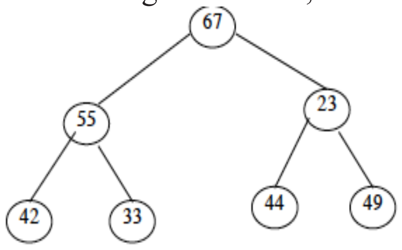


Fig8

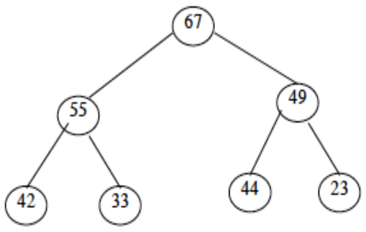


Fig9

The Fig9 is the required heap H without its original root R.

Algorithm

Let H be a heap with n elements stored in the array HA. Data is the item of the node to be removed. Last gives the information about last node of H. The LOC, left, right gives the location of Last and its left and right children as the Last rearranges in the appropriate place in the tree.

```
1. Input n elements in the heap H
2. Data = HA[0]; last = HA[n-1] and n = n - 1
3. LOC = 0, left = 2*LOC+1 and right = 2*LOC+2
4. Repeat the steps 5, 6 and 7 while (right <= n)
5. If (last >= HA[left]) and (last >= HA[right])
        (a) HA[LOC] = last
        (b) Exit
6.      a)If (HA[right] <= HA[left])
                (i) HA[LOC] = HA[left]
                (ii) LOC = left
        (b) Else
                (i) HA[LOC] = HA[right]
                (ii) LOC = right
7. left = 2 * LOC; right = left +1
8. If (left = n ) and (last < HA[left])
        (a) LOC = left
9. HA [LOC] = last
10. Exit
```

Source Code:

```
#include<stdio.h>
int arr[20],n;
//Function to display the elements in the array
void display()
{ int i;
    for(i=0;i<n;i++)
        printf ("%d ",arr[i]);
    printf ("\n");
}/*End of display()*/
//Function to insert an element to the heap
void insert(int num,int loc)
{int par;
    while(loc>0)
    {
        par=(loc-1)/2;
        if (num<=arr[par])
        {
            arr[loc]=num;
            return;
        }
        arr[loc]=arr[par];
        loc=par;
    }/*End of while*/
    arr[0]=num;
}/*End of insert()*/
//This function to create a heap
void create_heap()
{
    int i;
    for(i=0;i<n;i++)
        insert(arr[i],i);
}/*End of create_heap()*/
//Function to delete the root node of the tree
void del_root(int last)
{
    int left,right,i,temp;
    i=0; /*Since every time we have to replace root with last*/
    /*Exchange last element with the root */
    temp=arr[i];
    arr[i]=arr[last];
    arr[last]=temp;
    left=2*i+1; /*left child of root*/
    right=2*i+2;/*right child of root*/
    while( right < last)
    {
        if ( arr[i]>=arr[left] && arr[i]>=arr[right] )
            return;
        if ( arr[right]<=arr[left] )
        {
            temp=arr[i];
            arr[i]=arr[left];
            arr[left]=temp;
            i=left;
        }
        else
        {
            temp=arr[i];
            arr[i]=arr[right];
            arr[right]=temp;
            i=right;
        }
        left=2*i+1;
        right=2*i+2;
    }/*End of while*/
    if ( left==last-1 && arr[i]<arr[left] )/*right==last*/
    {
        temp=arr[i];
        arr[i]=arr[left];
        arr[left]=temp;
    }
}/*End of del_root*/
//Function to sort an element in the heap
void heap_sort()
{
    int last;
    for(last=n-1; last>0; last--)
        del_root(last);
}/*End of del_root*/
int main()
{
    int i;
    printf ("\nEnter number of elements : ");
    scanf ("%d",&n);
    for(i=0;i<n;i++)
    {
        printf ("\nEnter element %d : ",i+1);
        scanf ("%d",&arr[i]);
    }
    printf ("\nEntered list is :\n");
    display();
    create_heap();
    printf ("\nHeap is :\n");
    display();
    heap_sort();
    printf ("\nSorted list is :\n");
    display();
}/*End of main()*/
```

7.9. Exchange Sort

The exchange sort is almost similar as the bubble sort. In fact some people refer to the exchange sort as just a different bubble sort. The exchange sort compares each element of an array and swap those elements that are not in their proper position, just like a bubble sort does. The only difference between the two sorting algorithms is the manner in which they compare the elements.

The exchange sort compares the first element with each element of the array, making a swap where is necessary. In some situations the exchange sort is slightly more efficient than its counter part the bubble sort. The bubble sort needs a final pass to determine that it is finished, thus is slightly less efficient than the exchange sort, because the exchange sort doesn't need a final pass.

Assume the following sequence of number: **45 67 78 12 9**

```
45    67    78    12    9
12    67    78    45    9
9     67    78    45    12
9     67    78    45    12
9     45    78    67    12
9     12    78    67    45
9     12    67    78    45
9     12    45    78    67
```

```
9          12   45   67   78
```

Algorithm

1. Compare the first pair of numbers (positions 0 and 1) and reverse them if they are not in the correct order.
2. Repeat for the next pair (positions 1 and 2).
3. Continue the process until all pairs have been checked.

4. Repeat steps 1 through 3 for positions 0 through n - 1 to i (for i = 1, 2, 3, ...) until no pairs remain to be checked.
5. The list is now sorted.
6. Display the sorted list
7. Exit

Source Code:

```
#include<stdio.h>
void display(int a[],int n);
int main(void)
{
      int array[50];        // An array of integers.
      int i, j;
      int temp;
    int n;
    printf("Enter n");
    scanf("%d",&n);
    //Some input
      for (i = 0; i < n; i++)
      {
            printf("Enter a number: ");
            scanf("%d", &array[i]);
      }
      //Algorithm
      for(i = 0; i < (n -1); i++)
      {
            for (j=(i + 1); j < n; j++)
            {
                  if (array[i] > array[j])
                  {
                        temp = array[i];
                        array[i] = array[j];
                        array[j] = temp;
                  }
             display(array,n);
            }
      }
    printf("\nSorted Array is: \n");
    display(array,n);
}
void display(int array[],int n)
{
    int i;
      for (i = 0; i < n; i++)
      {
            printf("%d\t",array[i]);

    }
```

```
        printf("\n");
    }
```

7.10 External Sort

These are methods employed to sort elements (or items), which are too large to fit in the main memory of the computer.
That is any sorting algorithm that uses external memory, such as tape or disk, during the sort is called external sort. Since most common sort algorithms assume high-speed random access to all intermediate memory, they are unsuitable if the values to be sorted do not fit in main memory. Internal sorting refers to the sorting of an array of data which is in RAM. The main concern with external sorting is to minimize external disk access since reading a disk block takes about a million times longer than accessing an item in RAM.
The involvement of external storage device makes sorting algorithms more complex because of the following reasons:
1. The cost of accessing an item is much higher than any computational cost.
2. Different procedures and methods have to be implemented and executed for different external storage devices.
External storage devices can be categorized into two types based on the access method. They are:
• Sequential Access Devices (e.g., Magnetic tapes)
• Random Access Devices (e.g., Disks)
MAGNETIC TAPES : Magnetic tape is wound on a spool. Tracks run across the length of the tape. Usually there are 7 to
9 tracks across the tape width and the data is recorded on the tape in a sequence of bits. The number that can be written per inch of track is called the tape density — measured in bits per inch.

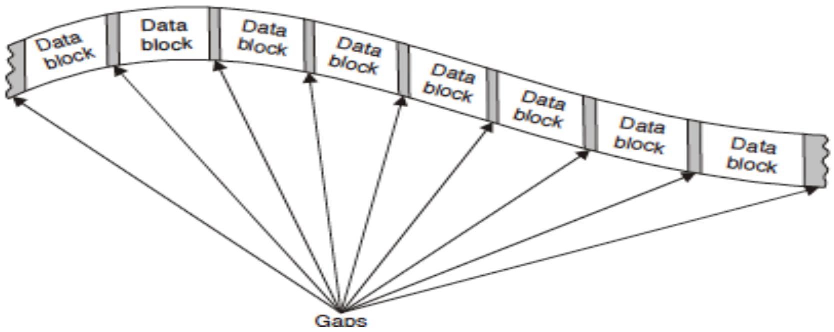


Fig: Magnetic Tape

Information on tapes is usually grouped into blocks, which may be of fixed or variable size. Blocks are separated by an inter-block gap. Because request to read or write blocks do not arrive at a tape drive at constant rate, there must be a gap between each pair of blocks forming a space to be passed over as the tape accelerates to read/write speed. The medium is not strong enough to withstand the stress that it would sustain with instantaneous starts and stops. Because the tape is not moving at a constant speed, a gap cannot contain user data. Data is usually read/written from tapes in terms of blocks.

In order to read or write data to a tape the block length and the address in memory to/from, which the data is to be transferred, must be specified. These areas in memory from/to which data is transferred will be called buffers. The block size (or length) will respond to its buffer size. And it is a crucial factor in tape access. A large block size is preferred because of the following reasons:

1. Consider a tape with a tape density of 600 bdp and an inter block gap of 3/4 inch; generally this gap is enough to write 450 characters. With a small block size, the number of blocks per tape length will increase. This means a larger number of inter block gaps, i.e., bits of data, which cannot be utilized for data storage, and thus tape utilization decreased. Thus the larger

the block size, fewer the number of blocks, fewer the number of inter block gaps and better the tape utilization.

2. Larger block size reduces the input/output time. The delay time in tape access is the time needed to cross the inter block gap. This delay time is larger when a tape starts from rest than when the tape is already moving. With a small block size the number of halts in a read are considerable causing the delay time to be incurred each time.

DISKS: Disks are an example of direct access storage devices. Data on disks are recorded on disk platter in concentric tracks. A disk has two surfaces on which data can be recorded. Disk packs have several such disks or platters rigidly mounted on a common spinder. Data is read/written to the disk by a read/write head. A disk pack would have one such head per surface. Each disk surface has a number of concentric circles called tracks. In a disk pack, the set of parallel tracks on each surface is called a cylinder. Tracks are further divided into sectors. A sector is the smallest addressable segment of a track.

Data is stored along the tracks in blocks. Therefore to access a disk, the track or cylinder number and the sector number of the starting block must be specified. For disk packs, the surface must also be specified. The read/write head moves horizontally to position itself over the correct track for accessing disk data. This introduces three time components into disk access:
Seek time: The time taken to position the read/write head over the correct cylinder.
Latency time: The time taken to position the correct sector under head.
Transfer time: The time taken to actually transfer the block between main memory and the disk.

The general method for external sorting is the merge sort. In this, file segments are sorted using a good internal sort method. These sorted file segments, called runs, are written out onto the device. Then all the generated runs are merged into one run.

External Sorting Algorithm

Perhaps the simplest form of external sorting is to use a fast internal sort with good locality of reference (which means that it tends to reference nearby items, not widely scattered items). Quicksort is one sort algorithm that is generally very fast and has good locality of reference. If the file is too huge, even virtual memory might be unable to fit it. Also, the performance may not be too great due to the large amount of time it takes to access data on disk.

Merge sort is an ideal candidate for external sorting because it satisfies the two criteria for developing an external sorting algorithm. Merge sort can be implemented either top-down or bottom-up. The top-down strategy is typically used for internal sorting, whereas the bottom-up strategy is typically used for external sorting.

Merge sort typically break a large data file into a number of shorter, sorted runs. These can be produced by repeatedly reading a section of the data file into RAM, sorting it with ordinary quicksort, and writing the sorted data to disk. After the sorted runs have been generated, a merge algorithm is used to combine sorted files into longer sorted files. The simplest scheme is to use a 2-way merge: merge 2 sorted files into one sorted file, and then merge 2 more, and so on until there is just one large sorted file.
One example of external sorting is the external mergesort algorithm. Let us assume that 900 megabyte of data needs to be sorted using only 100 megabytes of RAM.

1. Read 100 MB of the data in main memory and sort by some conventional method (usually quicksort).
2. Write the sorted data to disk.
3. Repeat steps 1 and 2 until all of the data is sorted in chunks of 100 MB. Now you need to

merge them into one single sorted output file.

4. Read the first 10 MB of each sorted chunk (call them input buffers) in main memory (90 MB total) and allocate the remaining 10 MB for output buffer.
5. Perform a 9-way merging and store the result in the output buffer. If the output buffer is full, write it to the final sorted file. If any of the 9 input buffers gets empty, fill it with the next 10 MB of its associated 100 MB sorted chunk or otherwise mark it as exhausted if there is no more data in the sorted chunk, do not use it for merging.

Let us analyze how the merge sort algorithm responds when it is practically applied to run using slow tape drives as input and output devices. It requires very little memory, and the memory required does not change with the number of data elements. If you have four tape drives, it works as follows:

1. Divide the data to be sorted in half and put half on each of two tapes.
2. Merge individual pairs of records from the two tapes; write two-record chunks alternately to each of the two output tapes.
3. Merge the two-record chunks from the two output tapes into four-record chunks; write these alternately to the original two input tapes.
4. Merge the four-record chunks into eight-record chunks; write these alternately to the original two output tapes.
5. Repeat until you have one chunk containing all the data, sorted that is, for logn passes, where n is the number of records.

On tape drives that can run both backwards and forwards, you can run merge passes in both directions, avoiding rewind time. For the same reason it is also very useful for sorting data on disk that is too large to fit entirely into primary memory. The above described algorithm can be generalized by assuming that the amount of data to be sorted exceeds the available memory by a factor of K. Then, K chunks of data need to be sorted and a K-way merge has to be completed. If X is the amount of main memory available, there will be K input buffers and 1 output buffer of size X/(K+1) each.
Depending on various factors (how fast the hard drive is, what is the value of K) better performance can be achieved if the output buffer is made larger (for example twice as large as one input buffer).
Note that you do not want to jump back and forth between 2 or more files in trying to merge them (while writing to a third file). This would likely produce a lot of time-consuming disk seeks. Instead, on a single-user PC, it is better to read a block of each of the 2 (or more) files into RAM and carry out the merge algorithm there, with the output also kept in a buffer in RAM until the buffer is filled (or we are out of data) and only then writing it out to disk. When the merge algorithm exhausts one of the blocks of data, refill it by reading from disk another block of the associated file. This is called buffering. On a larger machine where the disk drive is being shared among many users, it may not make sense to worry about this as the read/write head is going to be seeking all over the place
anyway.

```
mergesort(int a[], int left, int right)
{
      int i, j, k, mid;
      if (right > left)
      {
            mid = (right + left) / 2;
            mergesort(a, left, mid);
            mergesort(a, mid+1, right);
            /* copy the first run into array b */
            for (i = left, j = 0; i <= mid; i++, j++)
            b[ j] = a[i];
            b[j] = MAX_INT;
            /* copy the second run into array c */
            for (i = mid+1, j = 0; i <=right; i++, j++)
            c[ j] = a[i];
            c[ j] = MAX_INT;
            /* merge the two runs */
            i = 0;
            j = 0;
            for (k = left; k <= right; k++)
            a[k] = (b[i] < c[ j]) ? b[i++] : c[ j++];
      }
}
```

Reference

10. V.V. Das; "*Principles of Data Structures Using C and C++";* New Age International Publishers, New Delhi, India.
11. Y Langsam, MJ, Augenstein and A.M, Tanenbaum; "*Data Structure Using C and C++*"; Prentice Hall India.

Chapter 8: Searching

Searching is a process of checking and finding an element from a list of elements. Let A be a collection of data elements, i.e., A is a linear array of say n elements. If we want to find the presence of an element “data” in A, then we have to search for it. The search is successful if data does appear in A and unsuccessful if otherwise. There are several types of searching techniques; one has some advantage(s) over other. Following are the three important searching techniques:
1. Linear or Sequential Searching
2. Binary Searching
3. Hashing

The records that are stored in a list being searched must conform to the following minimal standards:

- Every record is associated to a key.
- Keys can be compared for equality or relative ordering.
- Records can be compared to each other or to keys by first converting records to their associated keys.

1. Linear or Sequential Search

In linear search, each element of an array is read one by one sequentially and it is compared with the desired element. A search will be unsuccessful if all the elements are read and the desired element is not found.

Algorithm for Linear Search

Let A be an array of n elements, A [1], A[2],A[3], ... A[n]. “data” is the element to be searched. Then this algorithm will find the data and display the location if present otherwise display data not found.

```
1.  Input an array A of n elements and “data” to be searched and initialize flag =0.
2.  Initialize i = 0; and repeat through step 3 if (i < n) by incrementing i by one .
3.  If (data == A[i])
      i.   Display data is found at location i
      ii.  Flag = 1
      iii. Return
4.  If (flag == 0)
      i.   Display “data is not found and searching is unsuccessful”
5.  Exit
```

Source Code:

```
void search(int arr[], int n)
{
    int flag = 0;
    int i,data;
    printf("Enter Data to be search \n");
    scanf("%d",&data);
    for(i=0;i<n;i++)
    {
        if(arr[i] == data)
        {
            printf("\n %d data is found at location %d\n",data,i);
            flag = 1;
            break;
        }
    }
    if(flag == 0)
```

```
            printf("\n %d data is not found in an array",data);
        }
```

2. Binary Search

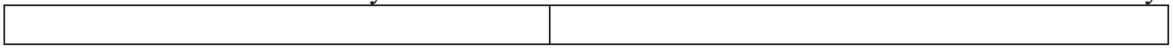
Binary search is an extremely efficient algorithm when it is compared to linear search. Binary search technique searches "data" in minimum possible comparisons. Suppose the given array is a sorted one, otherwise first we have to sort the array elements.
Then apply the following conditions to search a "data".

1. Find the middle element of the array (i.e., n/2 is the middle element if the array or the sub-array contains n elements).
2. Compare the middle element with the data to be searched, then there are following three cases.
   i. If it is a desired element, then search is successful.
   ii. If it is less than desired data, then search only the first half of the array, i.e., the elements which come to the left side of the middle element.
   iii. If it is greater than the desired data, then search only the second half of the array, i.e., the elements which come to the right side of the middle element.
1. Repeat the same steps until an element is found or exhaust the search area.

Algorithm for Binary Search

Let A be an array of n elements."Data" is an element to be searched. "mid" denotes the middle location of a segment (or array or sub-array) of the element of A. LB and UB is the lower and upper bound of the array which is under consideration.

Search in the 1st half of the array   mid value   Search in the 2nd half of the array



First Value   (first + last) /2   Last Value

```
1.  Input an array A of n elements and "data" to be sorted.
2.  LB = 0, UB = n; mid = int ((LB+UB)/2)
3.  Repeat step 4 and 5 while (LB <= UB) and (A[mid] ! = data)
4.  If (data < A[mid])
      i.  UB = mid-1
5.  Else
      i.  LB = mid + 1
6.  Mid = int ((LB + UB)/2)
7.  If (A[mid]== data)
      i.  Display "the data found"
8.  Else
      i.  Display "the data is not found"
9.  Exit
```

Suppose we have an array of 7 elements. Following steps are generated if we binary search a data = 45 from the above array.

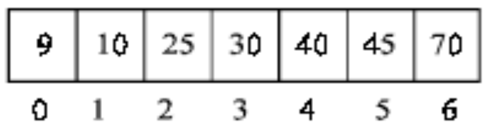


Fig: Step 1

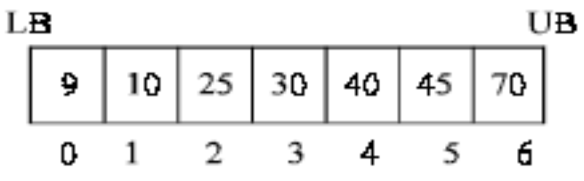
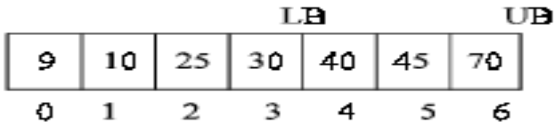


Fig: Step 2

Step 1:
LB = 0; UB = 6
mid = (0 + 6)/2 = 3
A[mid] = A[3] = 30
Step 2:
Since (A[3] < data) - i.e., 30 < 45 - reinitialise the variable LB, UB and mid.
LB = 3 UB = 6

mid = (3 + 6)/2 = 4
A[mid] = A[4] = 40
Step 3:
Since (A[4] < data) - i.e., 40 < 45 - reinitialize the variable LB, UB and mid (Fig Below).
LB = 4 UB = 6
mid = (4 + 6)/2 = 5
A[mid] = A[5] = 45



Step 4:
Since (A [5] == data) - i.e., 45 == 45 - searching is successful.
Source Code:

```
void binary_search(int arr[],int n,int data)
{
    int start = 0;
    int end = n-1;
    int middle = (start + end)/2;
    while(data != arr[middle] && start <= end)
    {
        if (data > arr[middle])
            start = middle+1;
        else
            end = middle-1;
        middle = (start+end)/2;
    }
    if (data == arr[middle])
        printf("\n%d found at position %d\n",data,middle + 1);
    if (start>end)
        printf ("\n%d not found in array\n",data);
}
```

3. Hashing

Hashing is a technique where we can compute the location of the desired record in order to retrieve it in a single access (or comparison). Let there is a table of n employee records and each employee record is defined by a unique employee code, which is a key to the record and employee name. If the key (or employee code) is used as the array index, then the record can be accessed by the key directly. If L is the memory location where each record is related with the key. If we can locate the memory address of a record from the key then the desired record can be retrieved in a single access. A function that transforms an element into an array index is called hash function. If h is the hash function and key is the elements, then h (key) is called as hash of key and is the index at which the element should be placed.

Collision: suppose that two elements k_1 and k_2 are such that $h(k_1) = h(k_2)$. Then when element k_1 is entered into the table, it is inserted at position $h(k_1)$. But when k_2 is hashed, an attempt may be made to insert the element into the same position where the element k_1 is stored. Clearly two elements cannot occupy the same position. Such a situation is known as a hash collision or simply collision.

Hash Function: The basic idea of hash function is the transformation of the key into the corresponding location in the hash table. A Hash function H can be defined as a function that takes key as input and transforms it into a hash table index.

Some of the hash functions are as follows
Division method, mid square method and folding method.

3.1. Division Method:

Choose a number m, which is larger than the number of keys k. i.e., m is greater than the total number of records in the TABLE. The hash function H is defined by H(k) = k (mod m) Where H(k) is the hash address (or index of the array) and here k (mod m) means the remainder when k is divided by m.

Let a company has 90 employees and 00, 01, 02, ... 99 be the two digit 100 memory address (or index or hash address) to store the records. We have employee code as the key. Choose m in such a way that it is grater than 90. Suppose m = 93. Then for the following employee code (or key k) :

H(k) = H(2103) = 2103 (mod93) = 57
H(k) = H(6147) = 6147 (mod 93) = 9
H(k) = H(3750) = 3750 (mod93) = 30

Then a typical employee hash table will look like as below table.

Hash Address	Employee Code (keys)	Employee Name and other Details
0		
1		
..		
..		
..		
9	6147	Anish
..		
..		
30	3750	Saju
..		
..		
57	2103	Rarish
..		
..		
99		

Hash Table

So if you enter the employee code to the hash function, we can directly retrieve TABLE[H(**k**)] details directly. Note that if the memory address begins with 01-**m** instead of 00-**m**, then we have to choose the hash function H(**k**) = **k** (mod **m**)+1.

3.2. Mid Square Method

The key k is squared. Then the hash function H is defined by H(k) = k^2 = l Where l is obtained by digits from both the end of k^2 starting from left. Same number of digits must be used for all of the keys. For example consider following keys in the table and its hash index :

K	4147	3750	2103
K^2	17197609	14062500	4422609
H(*k*)	97	62	22

$\cancel{1}\cancel{7}\cancel{1}97\cancel{6}\cancel{0}\cancel{9}$

~~140~~62~~500~~

$\cancel{4}\cancel{4}22\cancel{6}\cancel{0}\cancel{9}$

Hash Address	Employee Code (keys)	Employee Name and other Details
0		
1		
..		
..		
..		
22	2103	Giri
..		
..		
62	3750	Suni
..		
..		
..		
97	4147	Renjith
..		
99		

Hash Table with Mid Square Division

3.3. Folding Method

The key K, K_1, K_2,...... K_r is partitioned into number of parts. The parts have same number of digits as the required hash address, except possibly for the last part. Then the parts are added together, ignoring the last carry. That is $H(K) = K_1 + K_2 + ... + K_r$

Here we are dealing with a hash table with index form 00 to 99, i.e**,** two-digit hash table. So we divide the K numbers of two digits.

K	2103	7148	12345
$k_1\ k_2\ k_3$	21, 03	71, 46	12, 34, 5
H(k) $= k_1 + k_2 + k_3$	H(2103) = 21+03 = 24	H(7148) = 71+46 = 19	H(12345) = 12+34+5 = 51

K	2103	7148	12345
k_1, k_2, k_3	21, 03	71, 46	12, 34, 5
Reversing k_2, k_4	21, 30	71, 64	12, 43, 5
H(k) $= k_1 + k_2 + k_3$	H(2103) = 21+30 = 51	H(7148) = 71+64 = 55	H(12345) = 12+43+5 = 60

Extra milling can also be applied to even numbered parts, K_2, K_4, ... are each reversed before the addition(second table in above). H(7148) = 71 + 64 = 155, here we will eliminate the leading carry (i.e., 1). So H(7148) = 71 + 64 = 55.

3.4. Hash Collision and Handling of Hash Collision

It is possible that two non-identical keys K_1, K_2 are hashed into the same hash address. This situation is called Hash Collision.

Location	Keys	Records
0	210	
1	111	
2		
3	883	
4	344	
5		
6		
7		
8	488	
9		

Let us consider a hash table having 10 locations as in table. Division
Method is used to hash the key.
H(K) = K (mod) m ; Here m is chosen as 10. The Hash function produces any integer between 0 and 9 inclusions, depending on the value of the key. If we want to insert a new record with key 500 then
H(500) = 500(mod 10) = 0.
The location 0 in the table is already filled (i.e., not empty). Thus collision occurred.
Collisions are almost impossible to avoid but it can be minimized considerably by introducing any one of the following three techniques:

Table1
1. Open addressing
2. Chaining
3. Bucket addressing

3.4.1. Open Addressing (Linear Probing)
In open addressing method, when a key is colliding with another key, the collision is resolved by finding a nearest empty space by probing the cells. Suppose a record R with key K has a hash address H(**k**) = **h**. then we will linearly search h + i (where i = 0, 1, 2, ... m) locations for free space (i.e., h, h + 1, h + 2, h + 3 ...... hash address).
To understand the concept, let us consider a hash collision which is in the hash table Table1. If we try to insert a new record with a key 500 then
H(500) = 500(mod 10) = 0.
The array index 0 is already occupied by H(210). With open addressing we resolve the hash collision by inserting the record in the next available free or empty location in the table. Here the key 111 also occupies next location, i.e., array hash index 1. Next available free location in the table is array index 2 and we place the record in this free location.

The position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found. This type of probing is called Linear Probing.

The main disadvantage of Linear Probing is that substantial amount of time will take to find the free cell by sequential or linear searching the table.

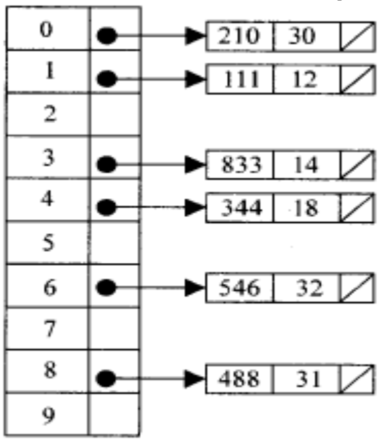
i. **Quadratic Probing**
   Suppose a record with R with key k has the hash address H(K) = h. Then instead of searching the location with address h, h + 1, h + 2,...... h + i ......, we search for free hash address h, h + 1, h + 4, h + 9, h + 16, ... h + i^2,......

ii. **Double Hashing**
   Second hash function H1 is used to resolve the collision. Suppose a record R with key k has the hash address H(K) = h and H1(K) = h1, which is not equal to m. Then we linearly search for the location with addresses h, h + h^1, h + 2h^1, h + 3h^1, ... h + i $(h^1)^2$ (where i = 0, 1, 2, ...).
   Note: The main drawback of implementing any open addressing procedure is the implementation of deletion.

3.4.2. Chaining
In chaining technique the entries in the hash table are dynamically allocated and entered into a linked list associated with each hash key. The hash table in Table2 can represented using linked list as in following figure.

Location	Keys	Records
0	210	30
1	111	12
2		
3	883	14
4	344	18
5		
6	546	32

7		
8	488	31
9		



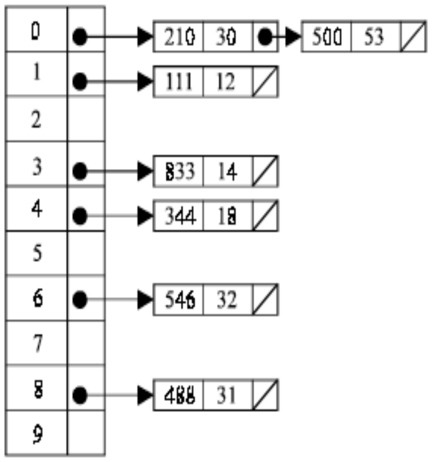


Table2 fig: Chaining Fig: Linked List at 0 index

If we try to insert a new record with a key 500 then H(500) = 500(mod 10) = 0. Then the collision occurs in normal way because there exists a record in the 0th position. But in chaining corresponding linked list can be extended to accommodate the new record with the key.

3.4.3. Bucket Addressing

Another solution to the hash collision problem is to store colliding elements in the same position in table by introducing a bucket with each hash address. A bucket is a block of memory space, which is large enough to store multiple items.

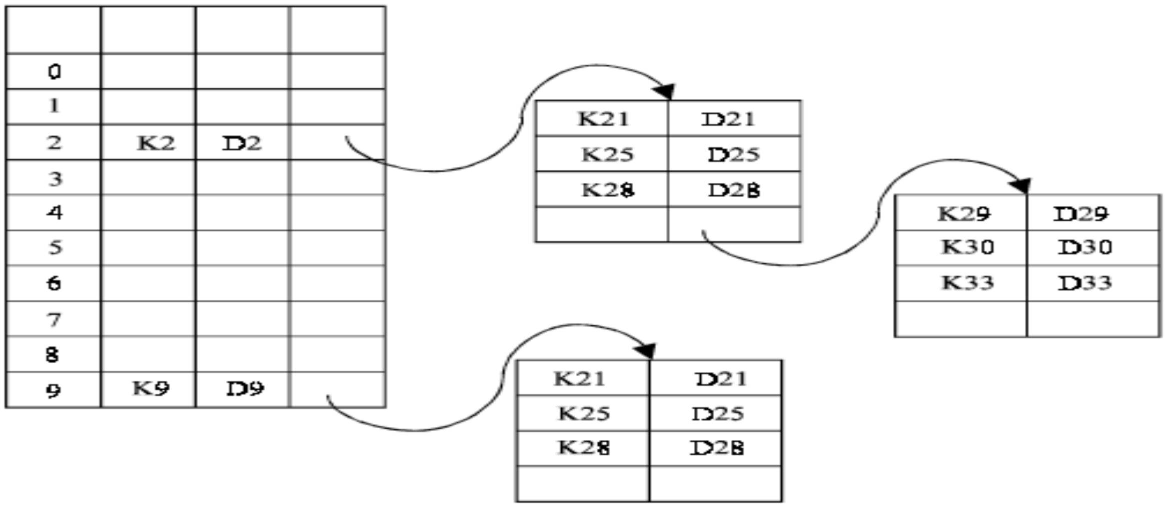


Fig: Avoid Collision Using bucket

Fig above shows how hash collision can be avoided using buckets. If a bucket is full, then the colliding item can be stored in the new bucket by incorporating its link to previous bucket.

Hash Deletion

A data can be deleted from a hash table. In chaining method, deleting an element leads to the deletion of a node from a linked list. But in linear probing, when a data is deleted with its key the position of the array index is made free. The situation is same for other open addressing methods.

Chapter 9: Graphs

A graph G consist of

1. Set of vertices V (called nodes), (V = {v_1, v_2, v_3, v_4......}) and
2. Set of edges E (E {e_1, e_2, e_3......e_m}

A graph can be represents as G = (V, E), where V is a finite and non empty set at vertices and E is a set of pairs of vertices called edges. Each edge 'e' in E is identified with a unique pair (a, b) of nodes in V, denoted by e = [a, b].

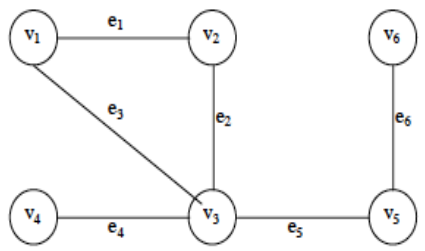


Fig1: Graph

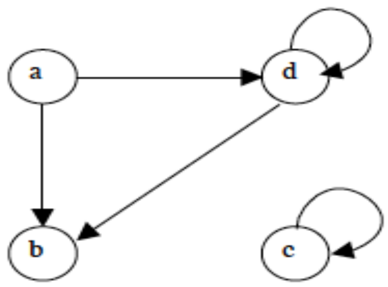


Fig2: Directed Graph

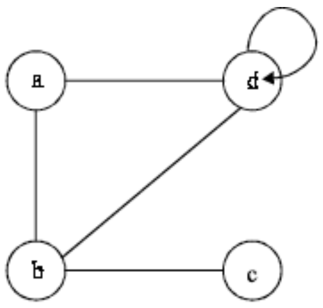


Fig3: Undirected Graph

Consider a graph, G in Fig1. Then the vertex V and edge E can be represented as: V = {v_1, v_2, v_3, v_4, v_5, v_6} and E = {e_1, e_2, e_3, e_4, e_5, e_6} E = {(v_1, v_2) (v_2, v_3) (v_1, v_3) (v_3, v_4), (v_3, v_5) (v_5, v_6)}. There are six edges and vertex in the graph.

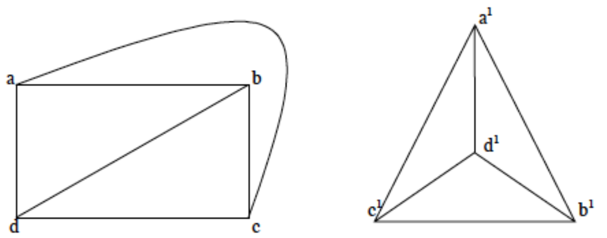
A ***directed graph*** G is defined as an ordered pair (V, E) where, V is a set of vertices and the ordered pairs in E are called edges on V. A directed graph can be represented geometrically as a set of marked points (called vertices) V with a set of arrows (called edges) E between pairs of points (or vertex or nodes) so that there is at most one arrow from one vertex to another vertex. Above Fig2 shows a directed graph, where G = {a, b, c, d }, {(a, b), (a, d), (d, b), (d, d), (c, c)}.

The edge (a, b) is ***incident from*** *a to b*. The vertex a is called the initial vertex and the vertex b is called the terminal vertex of the edge (a, b). If an edge that is incident from and into the same vertex, say (d, d) and (c, c) in Fig2, is called a loop.

Two vertices are said to be ***adjacent*** if they are joined by an edge. Consider edge (a, b), the vertex a is said to be adjacent to the vertex b, and the vertex b is said to be adjacent from the vertex a. A vertex is said to be an ***isolated vertex*** if there is no edge incident with it. In Fig2 vertex C is an isolated vertex.

An ***undirected graph*** G is defined abstractly as an ordered pair (V, E), where V is a set of vertices and the E is a set at edges. An undirected graph can be represented geometrically as a set of marked points (called vertices) V with a set at lines (called edges) E between the points. An undirected graph G is shown in Fig3.

Two graphs are said to be ***isomorphic*** if there is a one-to-one correspondence between their vertices and between their edges such that incidence are prevented. Fig4 shows an isomorphic undirected graph.



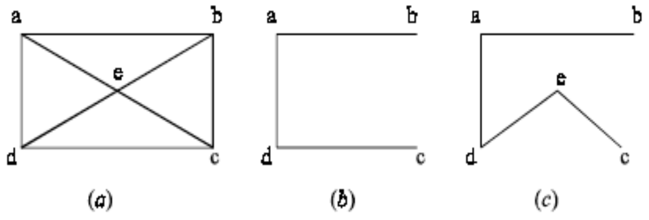
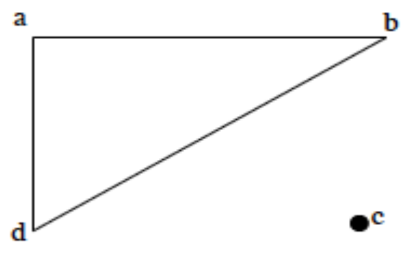


Fig4: Isomorphic Undirected Graph Fig5: Sub Graphs

Let G = (V, E) be a graph. A graph $G^1 = (V^1, E^1)$ is said to be a *sub-graph* of G if E^1 is a subset at E and V^1 is a subset at V such that the edges in E^1 are incident only with the vertices in V^1. For example Fig5 (b) is a sub-graph at Fig5 (a). A sub-graph of G is said to be a *spanning sub-graph* if it contains all the vertices of G. For example Fig5(c) shows a spanning sub-graph at Fig9 (a).
The number of edges incident on a vertex is its degree. The degree of vertex a, is written as degree (a). If the degree of vertex is zero, then vertex is called *isolated vertex*. For example the degree of the vertex a in Fig9 (a) is 3.
A graph G is said to be ***weighted graph*** if every edge and/or vertices in the graph is assigned with some weight or value.
An undirected graph is said to be ***connected graph*** if there exist a path from any vertex to any other vertex. Otherwise it is said to be disconnected.



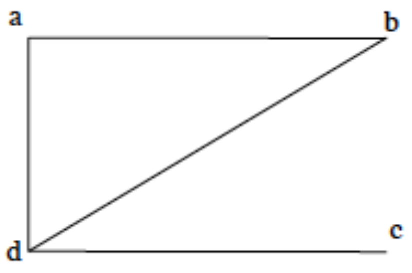
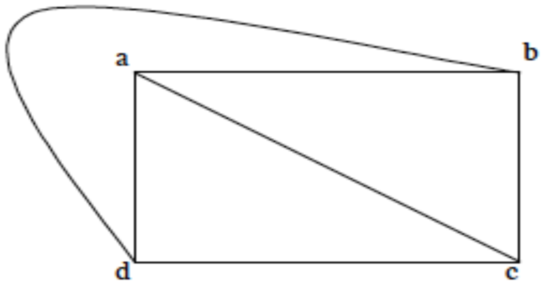


Fig6 (a): Disconnected Graph Fig6 (b): Connected Graph

A graph G is said to complete *(strongly connected)* if there is a path from every vertex to every other vertex. Let a and b are two vertices in the directed graph, then it is a complete graph if there is a path from a to b as well as a path from b to a. A complete graph with n vertices will have n (n – 1)/2 edges. Fig7 (a) illustrates the complete undirected graph and Fig7(b) shows the complete directed graph.



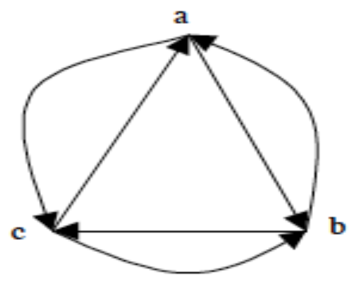


Fig7 (a): Complete undirected Graph Fig7 (b): Complete Directed Graph

In a directed graph, a path is a sequence of edges (e_1, e_2, e_3, ... en) such that the edges are connected with each other (i.e., terminal vertex en coincides with the initial vertex e_1). A path is said to be *elementary* if it does not meet the same vertex twice. A path is said to be simple if it does not meet the same edges twice. Consider a graph in Fig8.

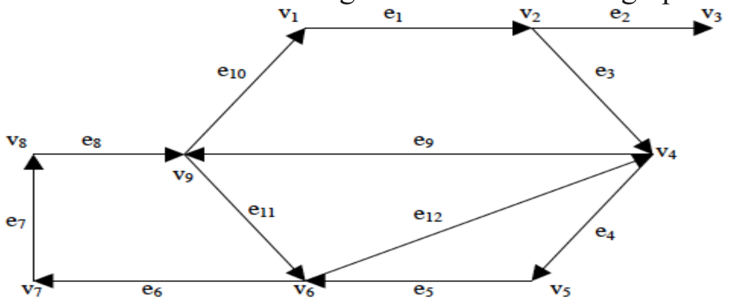


Fig8

Where (e_1, e_2, e_3, e_4, e_5) is a path; (e_1, e_3, e_4, e_5, e_{12}, e_9, e_{11}, e_6, e_7, e_8, e_{11}) is a path but not a simple one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8, $e_{11,}$ e_{12}) is a simple path but not elementary one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8) is an elementary path.

A circuit is a path (e_1, e_2, …... e_n) in which terminal vertex of e_n coincides with initial vertex of e_1. A circuit is said to be simple if it does not include (or visit) the same edge twice. A circuit is said to be elementary if it does not visit the same vertex twice. In Fig8 (e_1, e_3, e_4, e_5, e_{12}, e_9, e_{10}) are a simple circuit but not an elementary one; (e_1, e_3, e_4, e_5, e_6, e_7, e_8, e_{10}) is an elementary circuit.

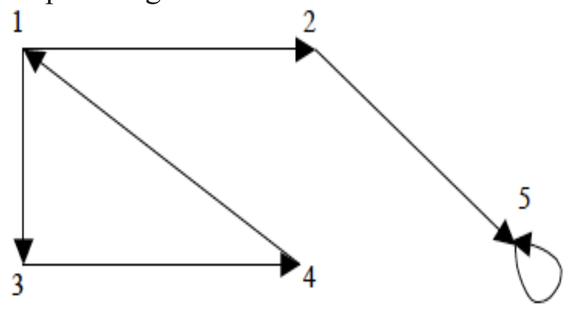
9.1. Representations of Graph

There are two standard ways of maintaining a graph G in the memory of a computer.

1. Adjacency matrix representation
2. Linked representation of a graph using linked list

9.1.1. Adjacency matrix representation of graph

The Adjacency matrix of a graph G = (V, E) with n vertices, is an n*n matrix. Considered a directed
Graph in Fig9 where all the vertices are numbered, (1, 2, 3, 4...... etc.)



i \ j	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	1	0	0	0	0
5	0	0	0	0	1

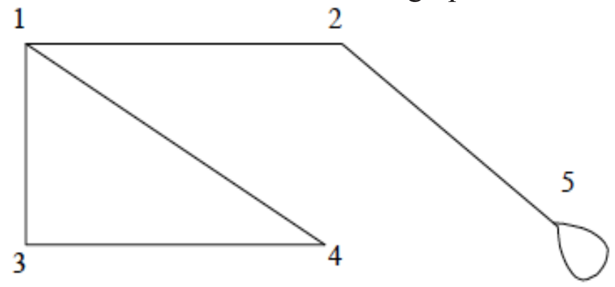
Fig9: Directed Graph Fig10: Matrix representation of directed graph of Fig9

The adjacency matrix A of a directed graph G = (V, E) can be represented (in Fig10) with the following conditions

Aij = 1 {if there is an edge from V_i to V_j or if the edge (i, j) is member of E.}

Aij = 0 {if there is no edge from V_i to V_j}

Let us considered an undirected graph of following Fig11.



i \ j	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	1	0
4	1	0	1	0	0
5	0	1	0	0	1

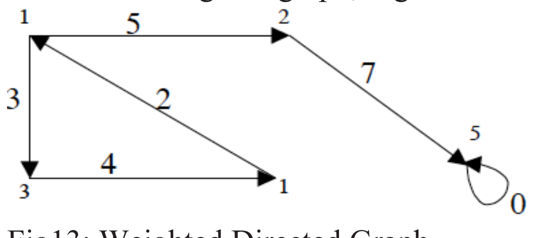
Fig11: Undirected Graph Fig12: Matrix representation of undirected graph of Fig11

The adjacency matrix A of a undirected graph G = (V, E) can be represented (in Fig12) with the following conditions

Aij = 1 {if there is an edge from V_i to V_j or if the edge (i, j) is member of E}

Aij = 0 {if there is no edge from V_i to V_j or if the edge (i, j) is not a member of E}

To represent a weighted graph using adjacency matrix, weight of the edge (i, j) is simply stored as the entry in *i* th row and *j* th column of the adjacency matrix. There are some cases where zero can also be the possible weight of the edge, then we have to store some sentinel value for non-existent edge, which can be a negative value; since the weight of the edge is always a positive number. Consider a weighted graph, Fig13.



i \ j	1	2	3	4	5
1	– 1	5	3	– 1	– 1
2	– 1	– 1	– 1	– 1	7
3	– 1	– 1	– 1	4	– 1
4	2	– 1	– 1	– 1	– 1
5	– 1	– 1	– 1	– 1	0

Fig13: Weighted Directed Graph Fig14: Matrix Representation

The adjacency matrix A for a directed weighted graph G = (V, E, W_e) can be represented (in Fig14) as
Aij = Wij { if there is an edge from Vi to Vj then represent its weight Wij.}
Aij = – 1 { if there is no edge from Vi to Vj}

In this representation, n^2 memory location is required to represent a graph with n vertices. The adjacency matrix is a simple way to represent a graph, but it has two disadvantages.

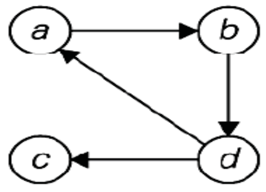
1. It takes n^2 space to represent a graph with n vertices, even for a spars graph and
2. It takes $O(n^2)$ time to solve the graph problem.

Transitive Closure and Warshall's Algorithm:
Transitive closure: The adjacency matrix, A, of a graph, G, is the matrix with elements a_{ij} such that $a_{ij} = 1$ implies there is an edge from *i* to *j*. If there is no edge then $a_{ij} = 0$.

Let A be the adjacency matrix of R and T be the adjacency matrix of transitive closer of R. T is called the reachability matrix of digraph (directed graph) D due to the property that $T_{i,j}$ =1 if and only if v_j can be reached from v_i in D by a sequence of arcs(edges). In simple word the definition is as:
A path exists between two vertices *i, j*, if

- there is an edge from *i* to *j*; or
- there is a path from *i* to *j* going through vertex 1; or
- there is a path from *i* to *j* going through vertex 1 and/or 2; or
- there is a path from *i* to *j* going through vertex 1, 2, and/or 3; or
- there is a path from *i* to *j* going through any of the other vertices

The following figure shows that the digraph(directed graph, its adjacency matrix and its transitive closure).



$$A = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array} \qquad T = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

(a) **(b)** **(c)**

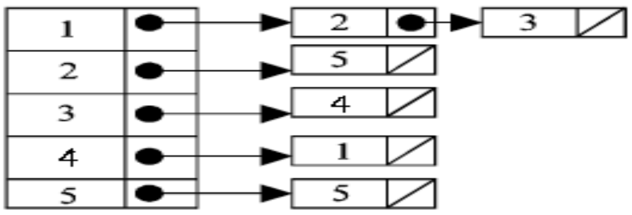
Digraph Adjacency matrix Transitive closure

If a,v_1,v_2,…..v_n,b is a path in a digraph D, a ≠ v_1, b≠ vn , n>2 then v_1,v_2,….and v_n are the interior vertices of this path.

Warshall's algorithm for finding transitive closure from diagraph
In wars hall's algorithm we construct a sequence of Boolean matrices A = $W^{[0]}$, $W^{[1]}$, $W^{[2]}$,….$W^{[n]}$ = T, where A is adjacency matrix and T is its transitive closure. This can be done from digraph D as follows.
$[W^{[1]}]_{i,j}$ = 1 if and only if there is a path from v_i to v_j with elements of a subset of {v_1} as interior vertices.
$[W^{[2]}]_{i,j}$ =1 if and only if there is a path from v_i to v_j with elements of a subset of {v_1 ,v_2} as interior vertices.
continuing this process, we generalized to
$[W^{[k]}]_{i,j}$ = 1 if and only if there is a path from v_i to v_j with elements of a subset of {v_1,v_2,….v_k} as interior vertices.

9.1.2. Linked List Representation of graph

In this representation (also called adjacency list representation), we store a graph as a linked structure. First we store all the vertices of the graph in a list and then each adjacent vertices will be represented using linked list node. Here terminal vertex of an edge is stored in a structure node and linked to a corresponding initial vertex in the list. Consider a directed graph in Fig9, it can be represented using linked list as Fig15.



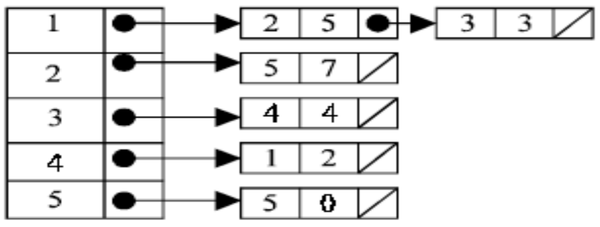


Fig15: Linked List representation of Fig9 Fig16:Linked representation of Fig13.

Weighted graph can be represented using linked list by storing the corresponding weight along with the terminal vertex of the edge. Consider a weighted graph in Fig13, it can be represented using linked list as in Fig16.

9.2. Operation on Graph

The primitive operations that can perform on graph are Creating a graph, searching, deleting a vertices and edges.

9.2.1. Creating a graph

To create a graph, first adjacency list array is created to store the vertices name, dynamically at the run time. Then the node is created and linked to the list array if an edge is there to the vertex.

Step 1: Input the total number of vertices in the graph, say n.
Step 2: Allocate the memory dynamically for the vertices to store in list array.
Step 3: Input the first vertex and the vertices through which it has edge(s) by linking the node from list array through nodes.
Step 4: Repeat the process by incrementing the list array to add other vertices and edges.
Step 5: Exit.

9.2.2. Searching and deleting from graph

Suppose an edge (1, 2) is to be deleted from the graph G. First we will search through the list array whether the initial vertex of the edge is in list array or not by incrementing the list array index. Once the initial vertex is found in the list array, the corresponding link list will be search for the terminal vertex.

Step 1: Input an edge to be searched
Step 2: Search for an initial vertex of edge in list arrays by incrementing the array index.
Step 3: Once it is found, search through the link list for the terminal vertex of the edge.
Step 4: If found display "the edge is present in the graph".
Step 5: Then delete the node where the terminal vertex is found and rearrange the link list.
Step 6: Exit

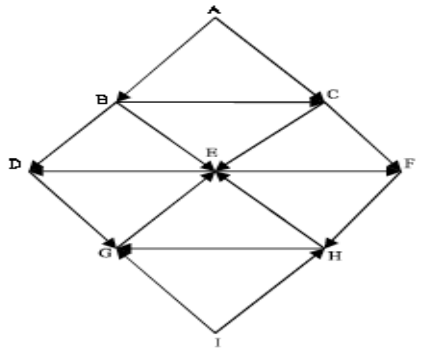
9.3. Traversing a Graph

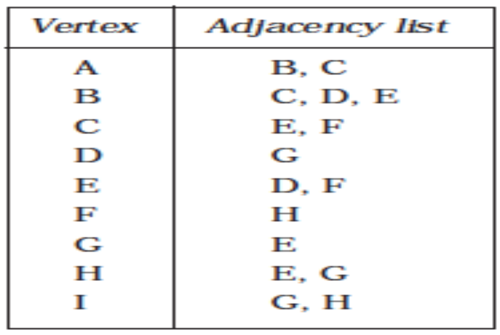
Many application of graph requires a structured system to examine the vertices and edges of a graph G. That is a graph traversal, which means visiting all the nodes of the graph. There are two graph traversal methods.

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

9.3.1. Breadth First Search

Given an input graph G = (V, E) and a source vertex S, from where the searching starts. The breadth first search systematically traverses the edges of G to explore every vertex that is reachable from S. Then we examine the entire vertices neighbor to source vertex S. Then we traverse all the neighbors of the neighbors of source vertex S and so on. A queue is used to keep track of the progress of traversing the neighbor nodes. Considered the following Fig17 for breadth first search traversing.



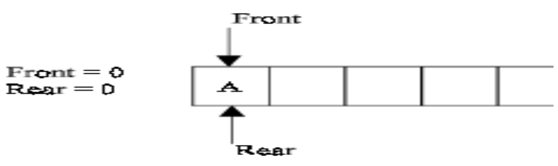


Vertex	Adjacency list
A	B, C
B	C, D, E
C	E, F
D	G
E	D, F
F	H
G	E
H	E, G
I	G, H

Fig17 a) Digraph Fig17 b) Linked List representation

Suppose the source vertex is A. Then following steps will illustrate the BFS.

Step 1: Initially push A (the source vertex) to the queue.



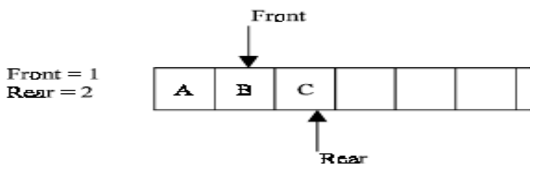
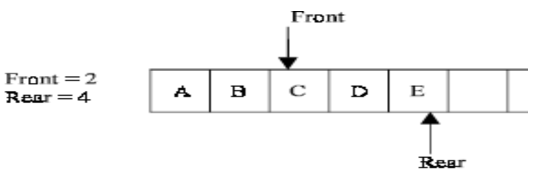


Fig: Step1 Fig: Step2

Step 2: Pop (or remove) the front element A from the queue (by incrementing front = front +1) and display it. Then push (or add) the neighboring vertices of A to the queue, (by incrementing Rear = Rear +1) if it is not in queue.

Step 3: Pop the front element B from the queue and display it. Then add the neighboring vertices of B to the queue, if it is not in queue. One of the neighboring elements C of B is preset in the queue, So C is not added to queue.



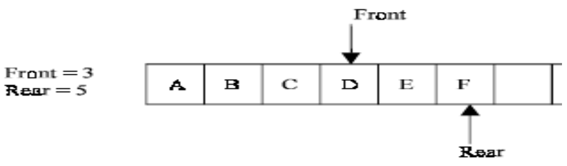
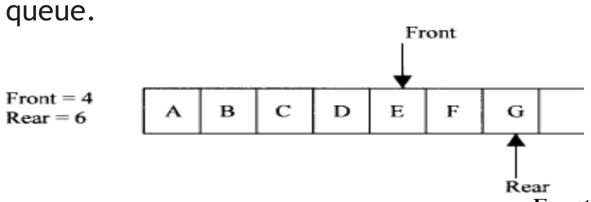


Fig: Step3 Fig: Step4

Step 4: Remove the front element C and display it. Add the neighboring vertices of C, if it is not present in queue. One of the neighboring elements E of C is present in the queue. So E is not added.

Step 5: Remove the front element D, and add the neighboring vertex if it is not present in queue.



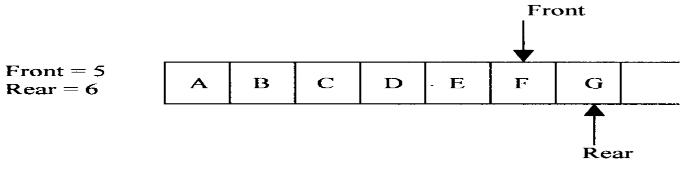
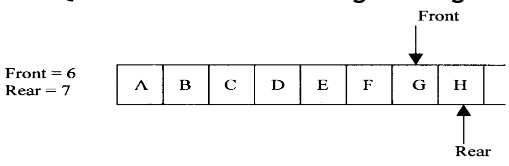
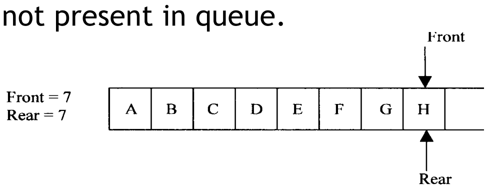
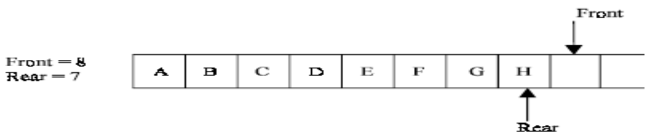


Fig: Step 5 Fig: Step6

Step6: Again the process is repeated (Until front > rear). That is remove the front element E of the Queue and add the neighboring vertex if it is not present in queue.







So A, B, C, D, E, F, G, H is the BFS traversal of the graph in Fig17.

Algorithm

1. Input the vertices of the graph and its edges G = (V, E)
2. Input the source vertex and assign it to the variable S.
3. Add or push the source vertex to the queue.
4. Repeat the steps 5 and 6 until the queue is empty (i.e., front > rear)
5. Pop the front element of the queue and display it as visited.
6. Push the vertices, which is neighbor to just, popped element, if it is not in the queue and displayed (i.e., not visited).
7. Exit

<u>9.3.2. Depth First Search (DFS)</u>

The depth first search (DFS), as its name suggest, is to search deeper in the graph, when ever possible. Given an input graph G = (V, E) and a source vertex S, from where the searching starts. First we visit the starting node. Then we travel through each node along a path, which begins at S. That is we visit a neighbor vertex of S and again a neighbor of a neighbor of S, and so on. The implementation of BFS is almost same except a stack is used instead of the queue. Consider the graph in Fig 17(a) and its linked list representation 17(b). Suppose the source vertex is I.
The following steps will illustrate the DFS

Step 1: Initially push I on to the stack.
STACK: I
DISPLAY:
Step 2: Pop and display the top element, and then push all the neighbors of popped element (i.e., I) onto the stack, if it is not visited (or displayed or not in the stack).
STACK: G, H
DISPLAY: I
Step 3: Pop and display the top element and then push all the neighbors of popped the element (i.e., H) onto top of the stack, if it is not visited.
STACK: G, E
DISPLAY: I, H
The popped element H has two neighbors E and G. G is already visited, means G is either in the stack or displayed. Here G is in the stack. So only E is pushed onto the top of the stack.
Step 4: Pop and display the top element of the stack. Push all the neighbors of the popped element on to the stack, if it is not visited.
STACK: G, D, F
DISPLAY: I, H, E
Step 5: Pop and display the top element of the stack. Push all the neighbors of the popped element onto the stack, if it is not visited.

STACK: G, D
DISPLAY: I, H, E, F
The popped element (or vertex) F has neighbor(s) H, which is already visited. Then H is displayed, and will not be pushed again on to the stack.
Step 6: The process is repeated as follows.
STACK: G
DISPLAY: I, H, E, F, D
STACK: //now the stack is empty
DISPLAY: I, H, E, F, D, G
Step 7: Repeat the same process until all vertexes are visited.
So I, H, E, F, D, G B, C, A is the DFS traversal of graph Fig17 from the source vertex I.

Algorithm

1. Input the vertices and edges of the graph G = (V, E).
2. Input the source vertex and assign it to the variable S.
3. Push the source vertex to the stack.
4. Repeat the steps 5 and 6 until the stack is empty and all note visited.
5. Pop the top element of the stack and display it.
6. Push the vertices, which is neighbor to just, popped element, if it is not in the stack and displayed (ie; not visited).
7. Exit.

9.4. Minimum Spanning Tree

A minimum spanning tree (MST) for a graph G = (V, E) is a sub graph $G^1 = (V^1, E^1)$ of G contains all the vertices of G.

1. The vertex set V^1 is same as that at graph G.
2. The edge set E^1 is a subset of G.
3. And there is no cycle.

If a graph G is not a connected graph, then it cannot have any spanning tree. In this case, it will have a *spanning forest*. Suppose a graph G with n vertices then the MST will have (n – 1) edges, assuming that the graph is connected. A minimum spanning tree (MST) for a weighted graph is a spanning tree with minimum weight. That is all the vertices in the weighted graph will be connected with minimum
edge with minimum weights. Fig19 shows the minimum spanning tree of the weighted graph in Fig. 18.

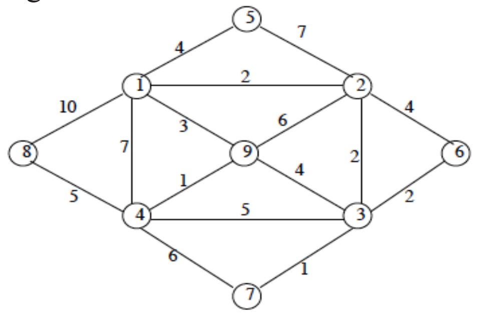


Fig18

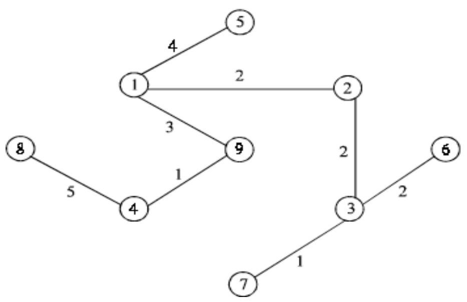


Fig19

Three different famous algorithms can be used to obtain a minimum spanning tree of a connected weighted and undirected graph.

1. Kruskal's Algorithm
2. Prim's Algorithm
3. Sollin's Algorithm
4. Round robin Algorithm

9.4.1. Kruskal's Algorithm

This is a one of the popular algorithm and was developed by Joseph Kruskal. To create a minimum cost spanning trees, using Kruskalls, we begin by choosing the edge with the minimum cost (if there are several edges with the same minimum cost, select any one of them) and add it to the spanning tree. In the next step, select the edge with next lowest cost, and so on, until we have selected (n – 1) edges to form the complete spanning tree. The only thing of which beware is that we don't form any cycles as we add edges to the spanning tree. Consider the graph of fig18.

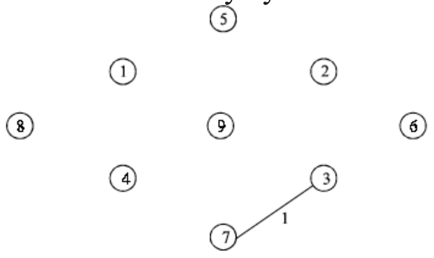


Fig20

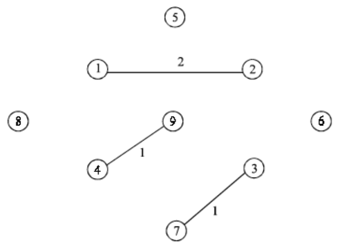


Fig21

The minimum cost edge in the graph G in Fig18 is 1. If we further analyze closely there are two edges (i.e., (7, 3), (4, 9)) with the minimum cost 1. As the algorithm says select any one of them. Here we select the edge (7, 3) as shown in Fig20. Again we select minimum cost edge (i.e., 1), which is (4, 9). Next we select minimum cost edge (i.e., 2). If we analyze closely there are two edges (i.e., (1, 2), (2, 3), (3, 6)) with the minimum cost 2. As the algorithm says select any one of them. Here we select the edge (1, 2) as shown in the Fig21.
Again we select minimum cost edge (i.e., 2), which is (2, 3). Similarly we select minimum cost edge (i.e., 2), which is (3, 6) as shown in Fig22.

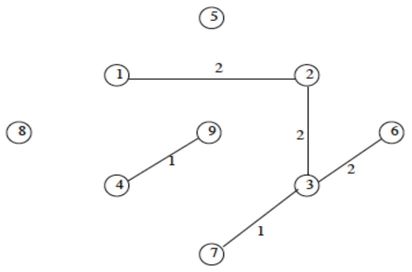


Fig22.

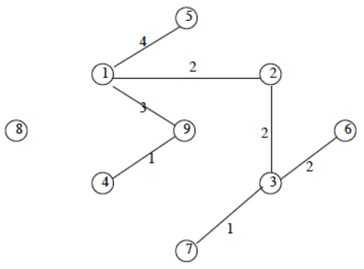


Fig23

Next minimum cost edge is (1, 9) with cost 3. Add the minimum cost edge to the minimum-spanning tree. Again, the next minimum cost edge is (1, 5) with cost 4. Add the minimum cost edge to the minimum-spanning tree as shown in Fig23. Next minimum cost edge is (4, 8) with cost 5. Add the minimum cost edge to the minimum-spanning tree as shown in Fig24 which is the minimum spanning tree with minimum cost of edges.

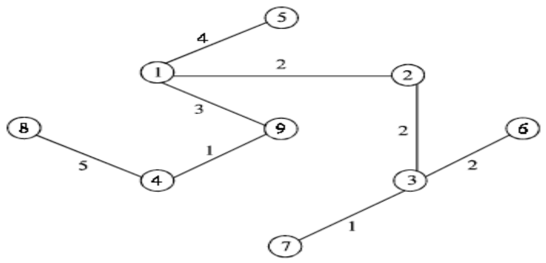


Fig24

Algorithm

Suppose G = (V, E) is a graph, and T is a minimum spanning tree of graph G.

1. Initialize the spanning tree T to contain all the vertices in the graph G but no edges.
2. Choose the edge e with lowest weight from graph G.
3. Check if both vertices from e are within the same set in the tree T, for all such sets of T.

If it is not present, add the edge e to the tree T, and replace the two sets that this edge connects.

4. Delete the edge e from the graph G and repeat the step 2 and 3 until there is no more edge to add or until the spanning tree T contains (n-1) edges.
5. Exit

9.4.2. Round Robin Algorithm

Round Robin algorithm, which provides even better performance when the number of edges is low. The algorithm is similar to Kruskal's except that there is a priority queue of arcs associated with each partial tree, rather than one global priority queue of all unexamined arcs.

All partial trees are maintained in a queue, Q. Associated with each partial tree, T, is a priority queue, P(T), of all arc with exactly one incident node in a tree, ordered by the weights of the arcs. Initially, as in Krusal's algorithm, each node is a particular tree. A priority queue of the entire arcs incident to *nd* is created for each node *nd,* and the single node trees are inserted into Q in arbitrary order. The algorithm proceeds by removing a partial tree, T1, from the front of Q, finding the minimum weight are a in P (T1), deleting from Q the tree, T2, at the other end of arc a; combining T1 and T2 into a single new tree T3[and at the same time combining P(T1) and P(T2), with a deleted, into P(T3)]; and adding T3 to the rear of Q. This continues until Q contains a single tree; the minimum spanning tree.

Shortest Path

A path from a source vertex a to b is said to be shortest path if there is no other path from a to b with lower weights. There are many instances, to find the shortest path for traveling from one place to another. That is to find which route can reach as quick as possible or a route for which the traveling cost in minimum. Dijkstra's Algorithm is used find shortest path.

Greedy Algorithm

An optimization problem is one in which we want to find, not just *a* solution, but the *best* solution. A greedy algorithm sometimes works well for optimization problems. A greedy algorithm works in phases. At each phase: we take the best we can get right now, without regard for future consequences and we hope that by choosing a *local* optimum at each step, we will end up at a *global* optimum.

Dijkstra's Algorithm

Let G be a directed graph with n vertices V_1, V_2, V_3 ...... V_n. Suppose G = (V, E, We) is weighted graph. i.e., each edge e in G is assigned a non- negative number, we called the weight or length of the edge e. Consider a starting vertices. Dijstra's algorithm will find the weight or length to each vertex from the source vertex.

Algorithm

Set V = {V_1, V_2, V_3 ...... V_n} contains the vertices and the edges E = {e_1, e_2, ... e_m} of the graph G. W(e) is the weight of an edge e, which contains the vertices V_1 and V_2. Q is a set of vertices, which are not visited. *m* is the vertex in Q for which weight W(m) is minimum, i.e., minimum cost edge. S is a source vertex.

```
1. Input the source vertices and assigns it to S
        (a) Set W(s) = 0 and
        (b) Set W (v) = ___ for all vertices V is not equal to S
2. Set Q = V which is a set of vertices in the graph
3. Suppose m be a vertices in Q for which W(m) is minimum
```

4. Make the vertices *m* as visited and delete it from the set Q
5. Find the vertices I which are incident with *m* and member of Q (That is the vertices which are not visited)
6. Update the weight of vertices I = {i_1, i_2 ...... i_k} by
   (a) $W(i_1) = \min [W(i_1), W(m) + W(m, i_1)]$
7. If any changes is made in W(v), store the vertices to corresponding vertices i, using the array, for tracing the shortest path.
8. Repeat the process from step 3 to 7 until the set Q is empty
9. Exit

The above algorithm is illustrated with a graph in Fig25.

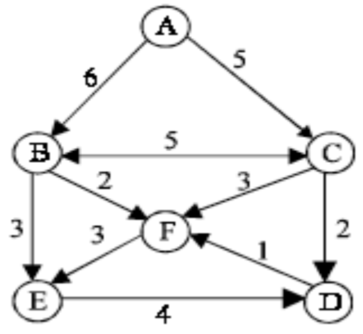


Fig25

Source vertices is = A W(A) = 0
V = {A, B, C, D, E, F) = Q

V	A	B	C	D	E	F
W (V)	0					
Q	A	B	C	D	E	F

A	
B	
C	
D	
E	
F	

ITERATION 1
m = A
W(A, A) = 0 (Distance from A to A) Now the Q = { B, C, D, E, F}. Two edges are incident with m i.e., I = { B, C}
W(B) = min [W(B), W (A) + W (A, B)]
 = Min (_, 0 + 6)
 = 6
W(C) = min [W(C), W (A) + W (A,C)]
 = min (–, 0 + 5) = 5

V	A	B	C	D	E	F
W (V)	0	6	5			
Q		B	C	D	E	F

A	
B	A
C	A
D	
E	
F	

ITERATION 2:
m = C (Because W(v) in minimum vertex and is also a member of Q)
Now the Q become (B, D, E, F)
Two edge are incident with C = {D, F} = I
W(D) = min (W(D), [W(C) + W(C, D)])
 = min (_, [5+2]) =7
W(F) = min (W(F), [W(C) + W (C, F)])
 = min (_, [5+3]) =8

V	A	B	C	D	E	F

W (V)	0	6	5	7		8
Q		B		D	E	F

A	
B	A
C	A
D	C
E	
F	C

ITERATION 3:
m = B (Because w (V) in minimum in vertices B and is also a member of Q)
Now the Q become (D, E, F)
Three edge are incident with B = { C, E, F}
Since C is not a member of Q so I = {E, F}
W(E) = min (_ , 6 + 3) =9
W(F) = min (8, 6 + 2) =8

V	A	B	C	D	E	F
W (V)	0	6	5	7	9	8
Q				D	E	F

A	
B	A
C	A
D	C
E	B
F	C

ITERATION 4:
m = D
Q = {E, F)
Incident vertices of D = { F } = I
W(F) = min (W(F) , [W(D) + W(D,F))
W(F) = min (8 , 7 + 1) =8

V	A	B	C	D	E	F
W (V)	0	6	5	7	9	8
Q					E	F

A	
B	A
C	A
D	C
E	B
F	C

ITERATION 5:
s = F
Q = { F)
Incident vertices of F = { E }
W(E) = min (W(F) , [W(E) + W(F,E))
W(E) = min (9 , 9 + 3) = 9

V	A	B	C	D	E	F
W (V)	0	6	5	7	9	8
Q					E	

A	
B	A
C	A
D	C
E	B
F	C

now E is the only chain, hence we stop the iteration and the final table is

V	A	B	C	D	E	F
W (V)	0	6	5	7	9	8

A	
B	A
C	A
D	C
E	B
F	C

If the source vertex is A and the destination vertex is D then the weight is 7 and the shortest path can be traced from table at the right side as follows. Start finding the shortest path from destination vertex to its shortest vertex. For example we want to find the shortest path from A to

D. Then find the shortest vertex from D, which is C. Check the shortest vertex, is equal to source vertex. Otherwise assign the shortest vertex as new destination vertex to find its shortest vertex as new destination vertex to find its shortest vertex. This process continued until we reach to source vertex from destination vertex.
D → C
C → A
A, C, D is the shortest path

Chapter 10: Algorithms

10.1. Deterministic and Non-deterministic algorithm

Deterministic algorithms can be defined in terms of a state machine: a *state* describes what a machine is doing at a particular instant in time. State machines pass in a discrete manner from one state to another. Just after we enter the input, the machine is in its *initial state* or *start state*. If the machine is deterministic, this means that from this point onwards, its current state determines what its next state will be; its course through the set of states is predetermined. Note that a machine can be deterministic and still never stop or finish, and therefore fail to deliver a result.

A nondeterministic algorithm is one in which for a given input instance each intermediate step has one or more possibilities. This means that there may be more than one path from which the algorithm may arbitrarily choose one. Not all paths terminate successfully to give the desired output. The nondeterministic algorithm works in such a way so as to always choose a path that terminates successfully, thus always giving the correct result.

What makes algorithms non-deterministic?

- If it uses external state other than the input, such as user input, a global variable, a hardware timer value, a random value, or stored disk data.

- If it operates in a way that is timing-sensitive, for example if it has multiple processors writing to the same data at the same time. In this case, the precise order in which each processor writes its data will affect the result.
- If a hardware error causes its state to change in an unexpected way.

Procedures of a Nondeterministic Algorithm:
The nondeterministic algorithm uses three basic procedures as follows:
1. CHOICE(1,n) or CHOICE(S) : This procedure chooses and returns an arbitrary element, in favour of the algorithm, from the closed interval [1,n] or from the set S.
2. SUCCESS : This procedure declares a successful completion of the algorithm.
3. FAILURE : This procedure declares an unsuccessful termination of the algorithm.

10.2. Divide and conquer algorithm

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub problems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.
The divide-and-conquer paradigm involves three steps at each level of the recursion:

- Divide the problem into a number of sub problems.
- Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.
- Combine the solutions to the sub problems into the solution for the original problem.

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two-sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r are indices numbering elements of the array such that $p \leq q < r$. The procedure assumes that the subarrays A[p □ q] and A[q + 1 □ r] are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray A[p □ r].

10.3. Series and Parallel Algorithm

In computer science, a parallel algorithm or concurrent algorithm, as opposed to a traditional sequential (or serial) algorithm, is an algorithm which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result. Some algorithms are easy to divide up into pieces in this way. For example, splitting up the job of checking all of the numbers from one to a hundred thousand to see which are primes could be done by assigning a subset of the numbers to each available processor, and then putting the list of positive results back together. Algorithms are also used for things such as rubik's cubing and for hash decryption.
Most of the available algorithms to compute pi (π), on the other hand, cannot be easily split up into parallel portions. They require the results from a preceding step to effectively carry on with the next step. Such problems are called inherently serial problems. Some problems are very

difficult to parallelize, although they are recursive. One such example is the depth-first search of graphs.
Parallel algorithms are valuable because of substantial improvements in multiprocessing systems and the rise of multi-core processors. In general, it is easier to construct a computer with a single fast processor than one with many slow processors with the same throughput. But processor speed is increased primarily by shrinking the circuitry, and modern processors are pushing physical size and heat limits. These twin barriers have flipped the equation, making multiprocessing practical even for small systems.
The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cycles) that they take. Parallel algorithms need to optimize one more resource, the communication between different processors. There are two ways parallel processors communicate, shared memory or message passing.
Shared memory processing needs additional locking for the data, imposes the overhead of additional processor and bus cycles, and also serializes some portion of the algorithm.
Message passing processing uses channels and message boxes but this communication adds transfer overhead on the bus, additional memory need for queues and message boxes and latency in the messages.

10.4. Heuristic and Approximate Algorithm

Many important computational problems are difficult to solve optimally. In fact, many of those problems are NP-hard1,which means that no polynomial-time algorithm exists that solves the problem optimally unless P=NP. A well-known example is the Euclidean traveling salesman problem (Euclidean TSP): given a set of points in the plane, find a shortest tour that visits all the points.

Another famous NP-hard problem is independent set: given a graph G = (V, E), find a maximum-size independent set $V * \subset V$. (A subset is independent if no two vertices in the subset are connected by an edge.) What can we do when faced with such difficult problems, for which we cannot expect to find polynomial-time algorithms? Unless the input size is really small, an algorithm with exponential running time is not useful. We therefore have to give up on the requirement that we always solve the problem optimally, and settle for a solution close to optimal. Ideally, we would like to have a guarantee on how close to optimal the solution is. For example, we would like to have an algorithm for Euclidean TSP that always produces a tour whose length is at most a factor ρ times the minimum length of a tour, for a (hopefully small) value of ρ. We call an algorithm producing a solution that is guaranteed to be within some factor of the optimum an approximation algorithm. This is in contrast to heuristics, which may produce good solutions but do not come with a guarantee on the quality of their solution.

The objective of a heuristic is to produce quickly enough a solution that is good enough for solving the problem at hand. This solution may not be the best of all the actual solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time. Heuristics may produce results by themselves, or they may be used in conjunction with optimization algorithms to improve their efficiency (e.g., they may be used to generate good seed values).
It is difficult to imagine the variety of existing computational tasks and the number of algorithms developed to solve them. Algorithms that either give nearly the right answer or provide a solution not for all instances of the problem are called heuristic algorithms . This group includes a plentiful spectrum of methods based on traditional techniques as well as specific ones.

10.5. Big OH Notation (Algorithm Analysis)

Big Oh is a characteristic scheme that measures properties of algorithm complexity performance and/or memory requirements. The algorithm complexity can be determined by eliminating constant factors in the analysis of the algorithm. Clearly, the complexity function f(n) of an algorithm increases as 'n' increases. Let us find out the algorithm complexity by analyzing the sequential searching algorithm. In the sequential search algorithm we simply try to match the target value against each value in the memory. This process will continue until we find a match or finish scanning the whole elements in the array. If the array contains 'n' elements, the maximum possible number of comparisons with the target value will be 'n' i.e., the worst case. That is the target value will be found at the nth position of the array. f (n) = n . The best case is when the number of steps is less as possible. If the target value is found in a sequential search array of the first position (i.e., we need to compare the target value with only one element from the array) we have found the element by executing only one iteration f (n) = 1
Average case falls between these two extremes (i.e., best and worst). If the target value is found at the n/2 position, on an average we need to compare the target value with only half of the elements in the array, so f (n) = n/2 The complexity function f(n) of an algorithm increases as 'n' increases. The function f (n)= O(n) can be read as "f of n is big Oh of n" or as "f (n) is of the order of n". The total running time (or time complexity) includes the initializations and several other iterative statements through the loop.

10.5.1. Growth Rate

We try to estimate the approximate computation time by formulating it in terms of the problem size N. If we consider that the system dependent factor (such as the compiler, language, computer) is constant; not varying with the problem size we can factor it out from the growth rate. The growth rate is the part of the formula that varies with the problem size. We use a notation called O-notation ("growth rate", "big-O"). The most common growth rates in data structure are:

Based on the time complexity representation of the big Oh notation, the algorithm can be categorized as :
1. Constant time O(1)
2. Logarithmic time Olog(n)
3. Linear time O(n)
4. Polynomial time $O(n^c)$ Where c >1
5. Exponential time $O(c^n)$ Where c > 1

If we calculate these values we will see that as N grows log(N) remains quite small and Nlog(N) grow fairly large but not as large as N^2 . Eg. Most sorting algorithms have growth rates of Nlog(N) or N^2 . Following table shows the growth rates for a given N.

N	Constant	log(N)	Nlog(N)	N^2
1	1	0	0	1
2	1	1	2	4
4	1	2	8	16
8	1	3	24	64
32	1	5	160	1024
256	1	8	2048	65536
2048	1	11	22528	4194304

10.5.2. Estimating the growth Rate

Algorithms are developed in a structured way; they combine simple statements into complex blocks in four ways:

- Sequence, writing one statement below another
- Decision, if-then or if-then-else
- Loops
- Subprogram call

Let us estimate the big-O of some algorithm structures.

Simple statements: We assume that statement does not contain a function call. It takes a fixed amount to execute. We denote the performance by O(1), if we factor out the constant execution time we are left with 1.

Sequence of simple statements: It takes an amount of execution time equal to the sum of execution times of individual statements. If the performance of individual statements are O(1), so is their sum.

Decision: For estimating the performance, then and else parts of the algorithm are considered independently. The performance estimate of the decision is taken to be the largest of the two individual big Os. For the case structure, we take the largest big O of all the case alternatives.

Simple counting loop: This is the type of loop in which the counter is incremented or decrement each time the loop is executed (for loop). If the loop contains simple statements and the number of times the loop executes is a constant; in other words, independent of the problem size then the performance of the whole loop is O(1). On the other hand if the loop is like

Eg:

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

The number of trips depends on N; the input size, so the performance is O(N).

Nested loops: The performance depends on the counters at each nested loop. For ex:

```
for (i=0; i< N; i++) {
        for (j=0; j< N; i++) {
                sequence of simple statements
        }
}
```

$$\boxed{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}1=\sum_{i=0}^{N-1}N=N\sum_{i=0}^{N-1}1=N^2}$$

The outer loop count is N but the inner loop executes N times for each time. So the body of the inner loop will execute N*N and the entire performance will be O(N^2).

```
for (i=1; i<=N; i++) {
        for (j=0; j< i; j++) {
                sequence of simple statement} }
```

$$\boxed{\sum_{i=1}^{N}\sum_{j=0}^{i-1}1=\sum_{i=1}^{N}i=\frac{N(N+1)}{2}=\frac{N^2}{2}+\frac{N}{2}}$$

In this case outer count trip is N, but the trip count of the inner loop depends not only N, but the value of the outer loop counter as well. If outer counter is 1, the inner loop has a trip count 1 and so on. If outer counter is N the inner loop trip count is N.

How many times the body will be executed?

$1+2+3+\ldots\ldots(N-1)+N = N(N+1) / 2 = ((N^2) +N)/2$
Therefore the performance is $O(N^2)$.
Generalization: A structure with k nested counting loops where the counter is just incremented or decrement by one has performance $O(N^k)$ if the trip counts depends on the problem size only.