1 Introduction to Data Structure and Algorithms

1.1 Abstract Data Type (ADT)

Problem solving with a computer means processing data. To process data, we need to define the data type and the operations to be performed on the data. The definition of the data type and the definition of the operation to be applied to the data is part of the idea behind an abstract data type (ADT), i.e. to hide how the operation is performed on the data. In other words, the user of the ADT needs only to know that a set of operations are available for the data type, but does not need to know how they are applied.

Definition:

An abstract data type is a data type packaged with the operations that are meaningful for the data type. We then encapsulate the data and the operations on the data and hide them from the user.

A useful tool for specifying the logical properties of a data type is the ADT. Fundamentally, a data is a collection of values and a set of operations which can operate on those values and this form a mathematical construct that may be implemented. The ADT refers to the basic mathematical concept that defines the data type.

- 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.

Note:

- **Abstract Data Type**
  - **Model** of a Data Type
    - Properties of the data
    - Operation that can be performed on that data
  - **Definition**: Abstract Data Type (ADT) is a **mathematical model** with a collection of operation defined on that model.

  - Example of ADT
    - **Integer**: …., -2, -1, 0, 1, 2, …..

    - ADT for points in 2D plane

```
struct Point {
  int x, y;
};
double findDistance(struct Point a, struct Point b)
{
        double distance;
        distance = sqrt((a.x - b.x) * (a.x - b.x) + (a.y-b.y) *(a.y-b.y));
        return distance;
}
```

naresh.com.np

1.2 Introduction to Data Structures

- Data structure is a way of organizing all data items and establishing relationship among those data items.
- Data structures are the building blocks of a program.
- Data structure mainly specifies the following four things:
  - Organization of data.
  - Accessing methods
  - Degree of associativity
  - Processing alternatives for information

To develop a program of an algorithm, we should select an appropriate data structure for that algorithm. Therefore algorithm and its associated data structures form a program.

Algorithm + Data structure = Program

A **static data structure** is one whose capacity is fixed at creation. For example, array. A **dynamic data structure** is one whose capacity is variable, so it can expand or contract at any time. For example, linked list, binary tree etc.

- 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.
- 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

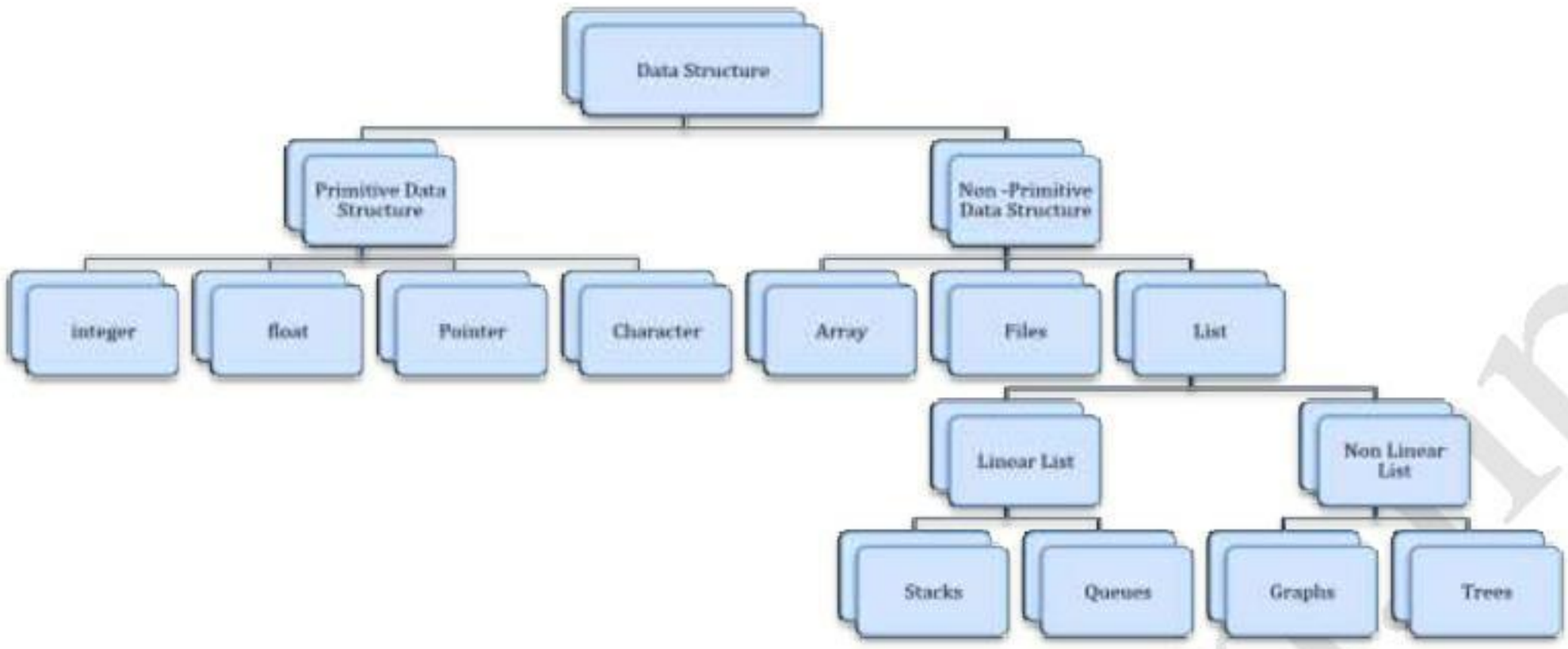
- In short Data Structure is an agreement about:
  - How to store a collection of objects in memory,
  - What operations we can perform on that data,
  - The algorithms for those operations, and
  - How time and space efficient those algorithms are

Data Structure Application Example

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

1.3 Classification of Data Structure

Data structures are normally divided into two broad categories



1.3.1 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.

1.3.2 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 are examples.

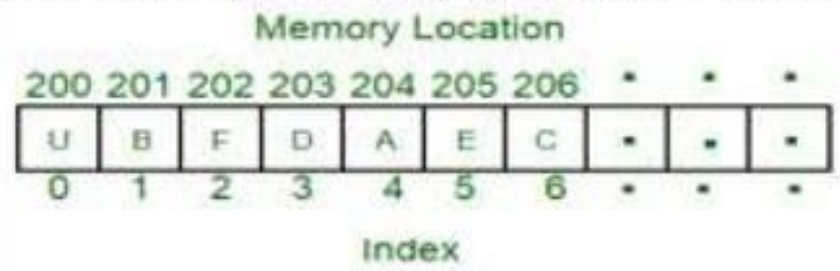
Note:

- ✓ The most commonly used operations on data structures are broadly categorized into four types
  - CREATE
  - SELECTION
  - DESTROY or DELETE
  - UPDATION
- ✓ Other operations performed on data structures include
  - Searching
  - Sorting
  - Merging

nareshpd.com

1.4 Array

- It is the collection of related data items, stored in a continuous way.
- The very common linear structure is array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data.
- An array is a list of a finite number n of homogeneous data elements (i.e., data elements of the same type) such that:
  - The elements of the array are referenced respectively by an index consisting of n consecutive numbers.
  - The elements of the array are stored respectively in successive memory locations.
- **An array is a collection of memory locations which allows storing homogeneous elements.**



- The first element of the array is referred with subscript 0(zero) & the subscript of last element is one less than size of an array i.e. (n-1).
- The elements of an array A may be denoted by the subscript notation (or bracket notation),

  A[0], A[1], A[2], … , A[N-1]
- The number K in A[K] is called a subscript or an index and A[K] is called a subscripted variable.
- An array can be single dimensional or multidimensional. The number of subscript give the dimension of an array.

An array lets you declare and work with a collection of values of the same type (homogeneous). For example, you might want to create a collection of five integers. One way to do it would be to declare five integers directly: int a, b, c, d, e;
Suppose you need to find average of 100 numbers. What will you do? You have to declare 100 variables.
For example: int a, b, c, d, e, f, g, h, i, j, k, l, m, n... etc.,
An easier way is to declare an array of 100 integers: int a[100];

The General Syntax is:
datatype array_name [size];

Example: int a[10];

Subscript

Some common operations performed in an array are (Show that an array is an ADT:

- Creating of an array
- Inserting new element at required position
- Deletion of any element
- Modification of any element
- Traversing of an array
- Merging of arrays

Insertion of new element at required position

Suppose we have an array

A[7]={10,50,8,7,30,75,85};

Suppose we want to insert 15 in array A, at location with index 4, it means the elements 30, 75 and 85 must shift 1 position downwards.

Deletion of any element from an array:

Suppose we want to delete the element A[4]=30, then the elements following it were moved upward by one location.

Traversing of an array:

Traversing means to access all the elements of the array, starting from first element upto the last element in the array one-by-one

Merging of two arrays:

Merging means combining elements of two arrays to form a new array. Simplest way of merging two arrays is the first copy all elements of one array into a third empty array, and then copy all the elements of other array into third array.

Suppose we want to merge two arrays A[7] and B[5]. The new array says C will be having (7+5) =12 elements.

1.5 Structure

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name.

An array is a data structure in which all the members are of the same data type. Structure is another data structure in which the individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. The individual structure elements are referred to as members. Defining a structure: A structure is defined as

```
struct structure_name
{
        member 1;
        member 2;
        ……..
        member n;
};
```

Structure variable declaration:

```
struct structure_name s1, s2, s3;
```

We can combine both template declaration and structure variable declaration in one statement.
Eg,

```
struct Student
{
        char name[2];
        int roll;
        char sec;
        float marks;
} s1, s2, s3;
```

Accessing members of a structure:

There are two types of operators to access members of a structure. Which are:

- Member operator (dot operator or period operator (.))
- Structure pointer operator (->).

1.6 Unions:

Both structure and unions are used to group a number of different variables together. Syntactically both structure and unions are exactly same. The main difference between them is in storage. In structures, each member has its own memory location **but all members of union use the same memory location which is equal to the greatest member's size.**

Declaration of union:

The general syntax for declaring a union is:

```
union union_name
{
        data_type member1;
        data_type member2;
        data_type member3;
        ……————————————
        data_type memberN;
};
```

union variable declaration:

```
union Student s1, s2, s3;
```

naresh.com.np

we can combine both template declaration and union variable declaration in one statement.

Eg,

```
union Student
{
    char name[2];
    int roll;
    char sec;
    float marks;
} s1, s2, s3;
```

Differences between structure and unions

The amount of memory required to store a structure variable is the sum of sizes of all the members. On the other hand, in case of a union, the amount of memory required is the same as member that occupies largest memory.

1.7 Pointer

A pointer is a variable that holds address (memory location) of another variable rather than actual value. Also, a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location. So, a pointer variable points to a memory location and we can access and change the contents of this memory location via the pointer. Pointers are used frequently in C, as they have a number of useful applications. In particular, pointers provide a way to return multiple data items from a function via function arguments.

Pointer Declaration

Pointer variables, like all other variables, must be declared before they may be used in a C program. We use asterisk (*) to do so. Its general form is:

```
data-type   *ptrvar;
```

For example,

```
int* ptr;
float *q;
char *r;
```

Reasons for using pointer:

- A pointer enables us to access a variable that is defined outside the function.
- Pointers are used in dynamic memory allocation.
- They are used to pass array to functions.
- They produce compact, efficient and powerful code with high execution speed.
- The pointers are more efficient in handling the data table.
- They use array of pointers in character strings result in saving of data storage space in memory. Sorting stings using pointer is very efficient.
- With the help of pointer, variables can be swapped without physically moving them.
- Pointers are closely associated with arrays and therefore provides an alternate way to access individual array elements.

Pointer initialization:

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement as follows:

```
int marks;
int *marks_pointer;
marks_pointer=&marks;
```

- **Memory Allocation in C**

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.

Dynamic memory allocations or Run time

In most of the real time problems, we cannot predict the memory requirements. Dynamic memory allocation does the job at run time

The process of allocating and freeing memory at run time is known as Dynamic Memory Allocation. This reserves the memory required by the program and returns this resource to the system once the use of reserved space utilized.

There are 4 library functions in C, **malloc( ), calloc( ), free( )** and **realloc( )** for memory management. These functions are defined within header file **stdlib.h** and **alloc.h**.

- **Passing (call) by Value and Passing (call) by Reference**

Arguments can generally be passed to functions in one of the two ways:

- Sending the values of the arguments (pass by value)
- Sending the addresses of the arguments (pass by reference)

Pass by value: In this method, the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.

Pass by reference: In this method, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them.

- **Pointers and Arrays**

An array name by itself is an address, or pointer. A pointer variable can take different addresses as values. In contrast, an array name is an address, or pointer, that is fixed.

Pointers and One-dimensional Arrays

In case of one dimensional array, an array name is really a pointer to the first element in the array. Therefore, if x is a one-dimensional array, then the address of the first array element can be expressed as either &x[0] or simply x. Moreover, the address of the second array element can be expressed as either &x[1] or as (x+1), and so on. In general, the address of array element (x+i) can be expressed as either &x[i] or as (x+i). Thus we have two different ways to write the address of any array element: we can write the actual array element, preceded by an ampersand; or we can write an expression in which the subscript is added to the array name. Since, &x[i] and (x+i) both represent the address of the ith element of x, it would seem reasonable that x[i] and *(x+i) both represent the contents of that address, i.e., the value of the ith element of x. The two terms are interchangeable. Hence, either term can be used in any particular situation. The choice depends upon your individual preferences.

1.8 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.
- A class is a way to bind the data and its associated functions together. It allows the data (and function) to be hidden, if necessary, from external use.
- When defining a class we are creating a new abstract data type that can be treated like any other built-in data type.
- The data inside the class are called member data and the functions are called member function.
- The binding of data and functions together into a single class type variable is called encapsulation, which is one of the benefit of object-oriented programming.
- The general form of declaring class is:

```
class class_name
{
        access-specifier1: member_data1;
                           member_data2;
                           - - - - - - - -
                           - - - - - - - -
                           member_function1;
                           - - - - - - - -
                           - - - - - - - -
        access-specifier2: member_data1;
                           - - - - - - - -
                           member_function1;
        access-specifier: member_data;
                           - - - - - - - -
                           member_function;
};
```

In above declaration, **class** is keyword. class_name is any identifier name. The number of member data and member function depends on the requirements. An object is an instance of a class i.e. variable of a class. The general form of declaring an object is

class_name object_name**;**

OR

```
class class_name
{
        private:
                variable_declaration;
                function_declaration;
        public:
                variable_declaration;
                function_declaration;
};
```

The **class** declaration is similar to a **struct** declaration. The keyword class specifies that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called class members. They are usually grouped under section, namely private and public to denote which of the members are private and which of them are public. The keyword private and public are known as visibility labels. Note that these keywords are followed by a colon.

Differentiate ADT with C++ class

A class is basically a block of code that contains anything your program needs to run whereas an ADT is an abstract way to store data.

- Classes have a slightly different terminology than ADT's and add other characteristics, like:
  - for example may live in packages
  - their members are called attributes and methods
  - attributes and methods have a certain visibility constraint
- An ADT is an abstract data structure with associated operations. A class is a programming construct associated with object oriented programming. Often a class will implement an ADT, but they are conceptually different beasts.
- In other words you don't need an Object Oriented language to do Object Oriented programming. It's convenient to work in a language that supports classes; but it's not necessary. It's not the language that makes a design Object Oriented. It's the use of ADTs.

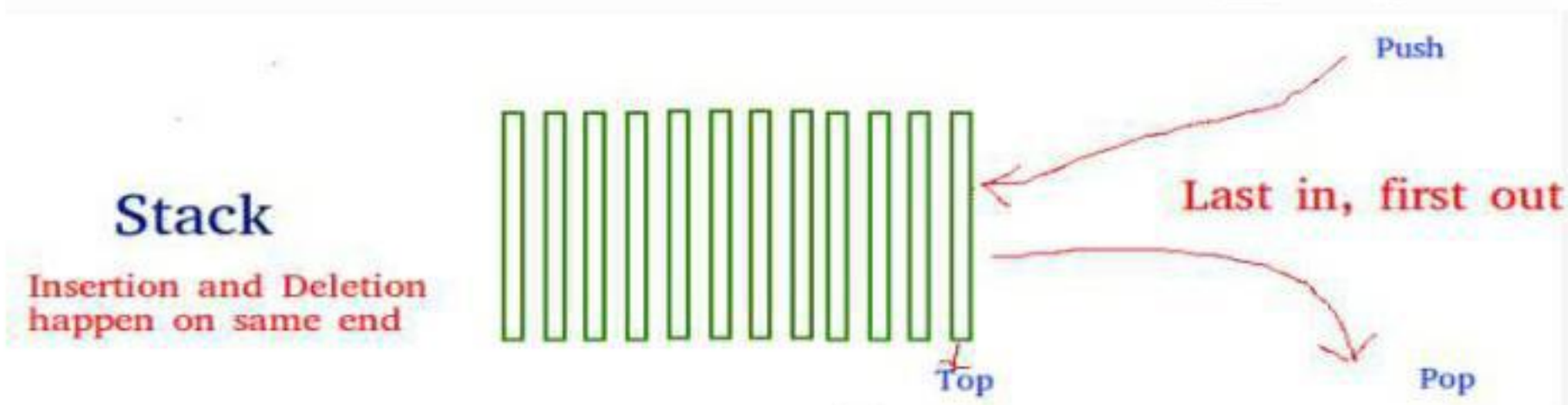
So ADTs are the concept; and Classes are one (but not the only) implementation.

Example: A Stack ADT defines the basic stack operations like push and pop (but says nothing of how these operations should be implemented), while a Stack class would use either a linked-list or an array to actually implement these operations.

naresh1.com.np

2 The Stack

- A stack is a non-primitive linear data structure.
- It is an ordered list in which addition of new data item and deletion of already existing data item is done form only one end, known as **top of stack (TOS).**
- The most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.
- 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.
- Unlike arrays, access of elements in a stack is restricted.
- It has two main function **PUSH** and **POP.**
- Insertion in a stack is done using **push** function and removal from a stack is done using **pop** function.
- Stack allows access to only the last element inserted hence, an item can be inserted or removed from the stack from one end called the top of the stack.
- It is therefore, also called **Last-In-First-Out (LIFO).**

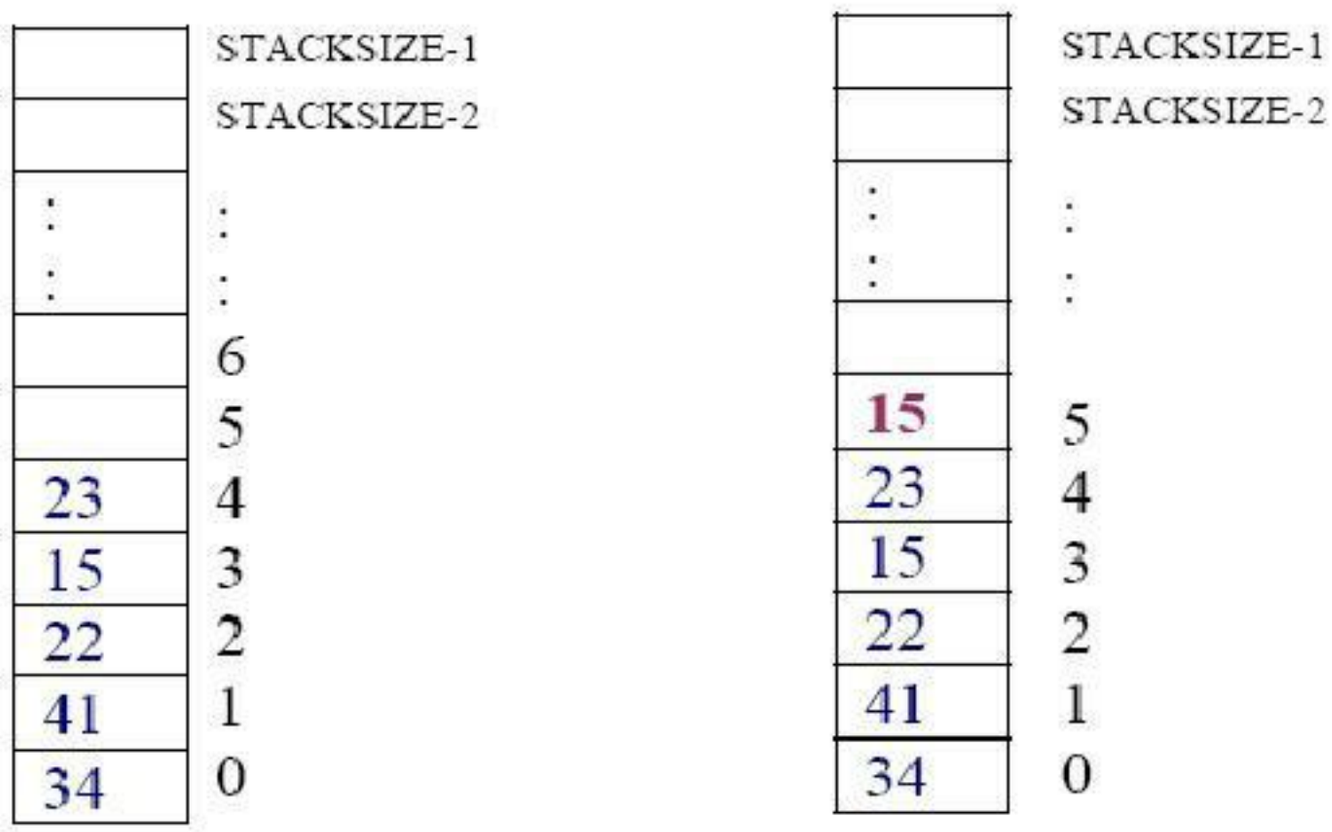


There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden. As new plates are added (pushed), each new plate becomes the top of the stack, and hides each plate below. As plates are removed (popped) from the stack, they can be used, and second plate becomes the top of the stack. Two important principles are illustrated by this metaphor, the Last-In First-Out principle is one. The second is that the contents of the stack are hidden. Only the top plate is visible, so to see what is on the third plate, the first and second plates will have to be removed.

2.1 Stack Operations

PUSH operation: The push operation is used to add (or push or insert) elements in a stack

- When we add an item to a stack, we say that we push it onto the stack
- The last item put into the stack is at the top

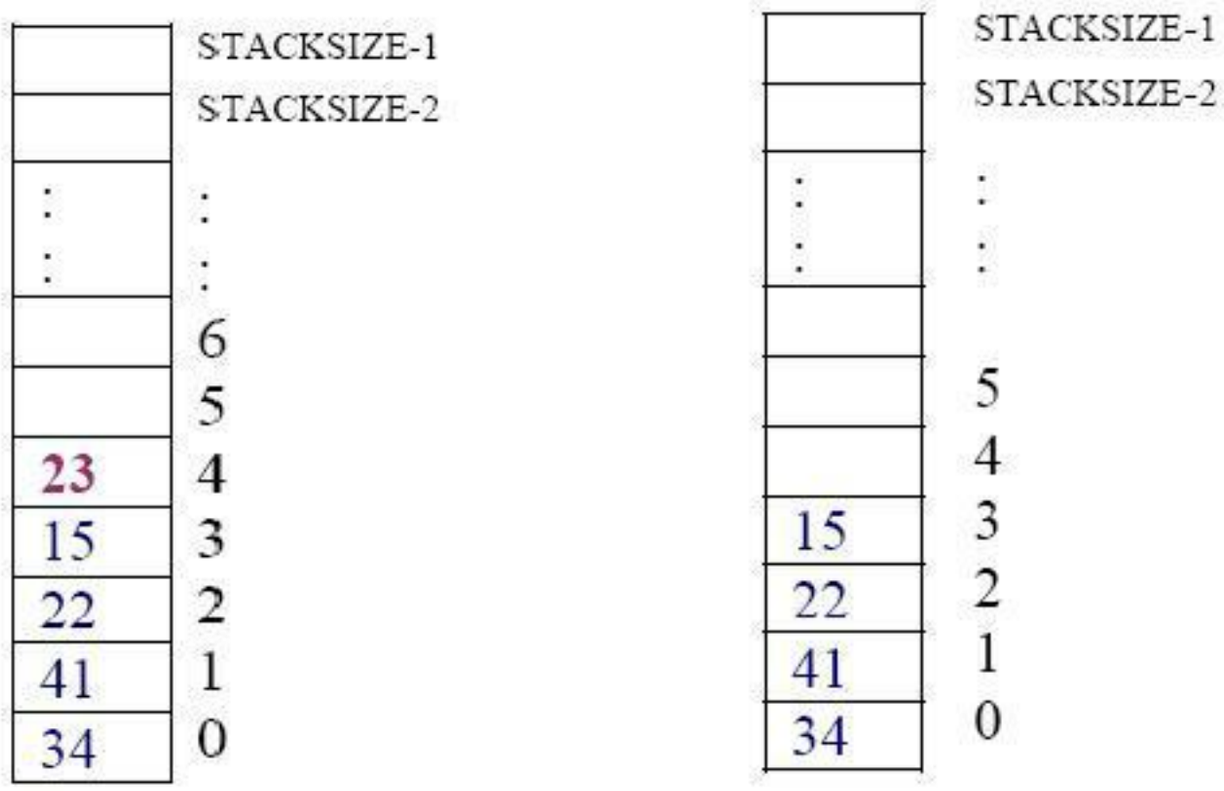


Before PUSH
(top=4, count=5)

After PUSH
(top=5, count= 6)

POP operation: The pop operation is used to remove or delete the top element from the stack.

- When we remove an item, we say that we pop it from the stack
- When an item popped, it is always the top item which is removed



Before POP
(top=4, count=5)

After POP
(top=3 count=4)

The **PUSH** and the **POP** operations are the **basic or primitive** operations on a stack. Some others operations are:

- **CreateEmptyStack operation**: This operation is used to create an empty stack.
- **IsFull operation:** The isfull operation is used to check whether the stack is full or not (i.e. stack overflow)
- **IsEmpty operation**: The isempty operation is used to check whether the stack is empty or not. (i. e. stack underflow)
- **Top operations:** This operation returns the current item at the top of the stack, **it doesn't remove it**

The Stack ADT:

A stack of elements of type T is a finite sequence of elements of T together with the operations
CreateEmptyStack(S): Create or make stack S be an empty stack
Push(S, x): Insert x at one end of the stack, called its top
Top(S): If stack S is not empty; then retrieve the element at its top
Pop(S): If stack S is not empty; then delete the element at its top
IsFull(S): Determine if S is full or not. Return true if S is full stack; return false otherwise
IsEmpty(S): Determine if S is empty or not. Return true if S is an empty stack; return false otherwise.

Stack ADT	
Definition	A list of data items that can only be accessed at one end, called the *top* of the stack.
Operations	**stack:** Creates an empty stack. **push:** Inserts an element at the top. **pop:** Deletes the top element. **empty:** Checks the status of the stack.

2.2 Stack Implementation

Stack can be implemented in two ways:

- Array Implementation of stack (or static implementation)
- Linked list implementation of stack (or dynamic implementation)

2.2.1 Array (static) implementation of a stack:

It is one of two ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value (an index of an array) that indicates the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack. By convention, in C implementation the empty stack is indicated by setting the value of top to -1(top=-1).

```
#define maxsize 10
sruct stack
{
        int items[maxsize];     //Declaring an array to store items
        int top;                //Top of a stack
};
typedef struct stack STACK;
```

Creating Empty stack :

The value of top=-1 indicates the empty stack in C implementation.

```
/*Function to create an empty stack*/
        void initStack(STACK *s)
        {
                s->top=-1;
        }
```

Stack Empty or Underflow:

This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty.

The following function return 1 if the stack is empty, 0 otherwise.

```
        int isempty(STACK *s)
        {
                if(s->top==-1)
                        return 1;
                else
                        return 0;
        }
```

Stack Full or Overflow:

This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location (maxsize-1) of the stack. The following function returns true (1) if stack is full false (0) otherwise.

```
        int isfull(STACK *s)
        {
                if(s->top==maxsize-1)
                        return 1;
                else
                        return 0;
        }
```

naresh.com.np

- **Algorithm for PUSH and POP operations on Stack**

Let **items[maxsize]** be an array to implement the stack. The variable top denotes the top of the stack

Algorithm for PUSH (inserting an item into the stack) operation:

This algorithm adds or inserts an item at the top of the stack

Step 1: [Check for stack overflow?]

if top=maxsize-1 then

print "Stack Overflow" and Exit

else

Set top=top+1 [Increase top by 1]

Set items[top] = data [Inserts data in new top position]

Step 2: Exit

Algorithm for POP (removing an item from the stack) operation

This algorithm deletes the top element of the stack and assign it to a variable item

Step 1: [Check for the stack Underflow]

If top<0 then

Print "Stack Underflow" and Exit

else

[Remove the top element]

Set data = items [top]

[Decrement top by 1]

Set top=top-1

Return the deleted item from the stack

Step 2: Exit

- **Representing Stack in C**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define maxsize 50
struct stack
{
        int top;
        int items[maxsize];
};
typedef struct stack STACK;

void initStack(STACK *s)
{
        s->top = -1;
}

int isEmpty(STACK *s)
{
        if(s->top == -1)
```

nareshpad.com.np

```
                return 1;
        else
                return 0;
}
int isFull(STACK *s)
{
        if(s->top==maxsize-1)
                return 1;
        else
                return 0;
}
void push(STACK *s, int data)
{
        if(isFull(s) == 1)
        {
                printf("Data overflow.");
                exit(1);
        }
        else
        {
                (s->top)++;
                s->items[s->top] = data;
        }
}
int pop(STACK *s)
{
        int v;
        if(isEmpty(s)==1)
        {
                printf("Data Underflow");
                exit(1);
        }
        else
        {
                 v = s->items[s->top];
                (s->top)--;
                return v;
        }
}
int stackTop(STACK *s)
{
        if(isEmpty(s)==1)
        {
                printf("Underflow");
                exit(1);
        }
        else
        {
                int v = s->items[s->top];
                return v;
        }
}
void printStack(STACK *s)
{
```

nareshpad.com.np

```
        int i;
        for(i=0;i<=s->top;i++)
        {
                printf("\n%d",s->items[i]);
        }
}

int main()
{
        int element,choice;
        int flag=1;
        STACK st;
        STACK *s;
        s=&st;
        initStack(s);       /* s->top=-1; indicates empty stack */
        do
        {
                printf("\n\n Enter your choice");
                printf(" \n\n\t 1:Push the elements");
                printf(" \n\n\t 2: To display the elements");
                printf(" \n\n\t 3: Pop the element");
                printf(" \n\n\t 4: To check top element");
                printf(" \n\n\t 5: Exit");
                printf("\n\n\n Enter of your choice:\t");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:
                                printf("\n Enter the number:");
                                scanf("%d", &element);                  /*Read an element from keyboard*/
                                push(s,element);
                                break;
                        case 2
                                printf("Data Elements in Stack are:")
                                printStack(s);
                                break;
                        case 3:
                                element = pop(s);
                                printf("\nPoped Element is %d",element);
                                break;
                        case 4:
                                element = stackTop(s);
                                printf("\nTop Element is %d",element);
                                break;
                        case 5:
                                flag=0;
                                break;
                        default:
                                printf("\n Invalid Choice");
                }
        }while(flag);
getch();
return 0;
}
```

nareshpad.com.np

2.3 Applications of Stack

Stack is used directly and indirectly in the following fields:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Conversion of an infix expression into a prefix expression
- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser
- To perform the undo sequence in a text editor
- Used in recursion
- To pass the parameters between the functions in a C program
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures

2.3.1 Expression

An expression is fined as the number of operands or data items combined with several operators. One of the applications of the stack is to evaluate the expression. We can represent the expression following three types of notation:

- **Infix notation**
- **Prefix notation**
- **Postfix notation**

Infix expression : It is an ordinary mathematical notation of expression where operator is written in between the operands. Example: **A+B**. **Here '+' is an operator and A** and **B** are called operands

Prefix notation: In prefix notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notation. Example: **+AB**

Postfix notation : In postfix notation the operators are written after the operands so it is called the postfix notation (post mean after). In this notation the operator follows the two operands. Example: **AB+**

Examples:

A + B (Infix)
+ AB (Prefix)
AB + (Postfix)

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

- You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated
- Both prefix and postfix are parenthesis free expressions. For example

(A + B) * C Infix form
*** + A B C Prefix form**
A B + C * Postfix form

Infix	Postfix	Prefix
A + B	AB +	+ AB
A + B-C	AB + C-	- + ABC
(A + B)*(C-D)	AB + CD-*	* + AB-CD

2.3.1.1 Converting an Infix Expression to Postfix

- First convert the sub-expression to postfix that is to be evaluated first and repeat this process. You substitute intermediate postfix sub-expression by any variable whenever necessary that makes it easy to convert.
- 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:
  - Parenthesize the expression starting from left to light.
  - 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.
  - The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
  - Once the expression is converted to postfix form, remove the parenthesis.

Example:

(A + B) * ((C - D) + E) / F	Infix form
(AB+) * ((C – D) + E) / F	
(AB+) * ((CD-) + E) / F	
(AB+) * (CD-E+) / F	
(AB+CD-E+*) / F	
AB+CD-E+*F/	Postfix form

Example: (Convert the infix expression listed in the table into postfix notation and verify yourself.)

Infix	*Postfix*
A + B	AB +
A + B – C	AB + C -
(A + B) * (C – D)	AB + CD - *
A $ B * C – D – E / F / (G + H)	AB $ C * D – EF / GH + / +
((A + B) * C – (D - E)) $ (F + G)	AB + C * DE - - FG + $
A – B / (C * D $ E)	ABCDE $ * / -

naresh.com.np

2.3.1.2 Converting an Infix expression to Prefix expression

The precedence rule for converting from an expression from infix to prefix are identical. Only changes from postfix conversion is that the operator is placed before the operands rather than after them. The prefix of

A+B-C (infix)
=(+AB)-C
=-+ABC (prefix)

Example Consider an example:

A $ B * C – D + E / F / (G + H) **infix form**
= A $ B * C – D + E / F /(+GH)
=$AB* C – D + E / F /(+GH)
=*$ABC-D+E/F/(+GH)
=*$ABC-D+(/EF)/(+GH)
=*$ABC-D+//EF+GH
= (-*$ABCD) + (//EF+GH)
=+-*$ABCD//EF+GH **which is in prefix form.**

2.3.1.3 Algorithm to convert Infix to Postfix notation:

POSTFIX(Q, P)

Suppose **Q** is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression **P**. Besides operands and operators; **Q** (infix notation) may also contain left and right parentheses. We assume that the operators in **Q** 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 P will be constructed from left to right using the operands from **Q** 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 **Q**. The algorithm is completed when the stack is empty.

1. **Push "(" onto** stack, and **add")" to the end of Q**.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of **Q** until the stack is empty.
3. If an operand is encountered, add it to **P**.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator ⊗ is encountered, then:
   - **a** Repeatedly pop from stack and add **P** each operator (on the top of stack), which has the same precedence as, or higher precedence than ⊗.
   - **b** Add ⊗ to stack. [End of If structure]
6. If a right parenthesis is encountered, then:
   - **a** Repeatedly pop from stack and add to **P** (on the top of stack until a left parenthesis is encountered.
   - **b** Remove the left parenthesis. [Do not add the left parenthesis to P.]

   [End of If structure]

   [End of step 2 loop]
7. Exit

Note: Special character ⊗ is used to symbolize any operator in **P**.

Example: The following tracing of the algorithm illustrates the algorithm. Consider an infix expression

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

Scan character	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+$ (postfix)

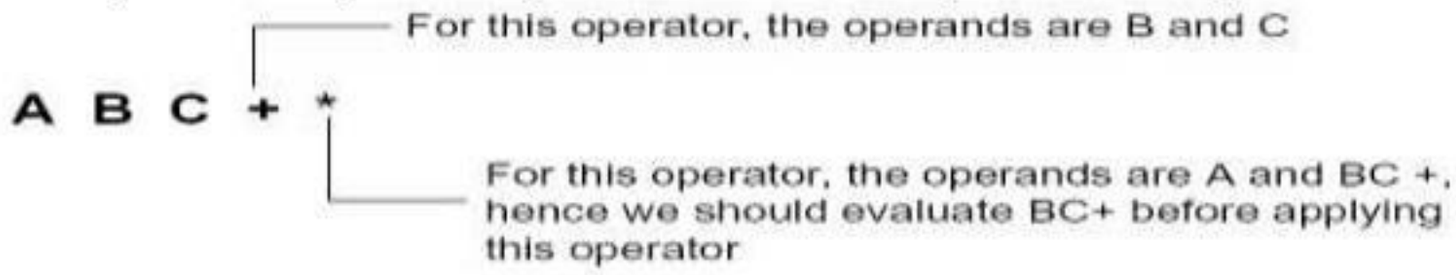
Example: Consider the Following Infix String: a+b*c+(d*e+f)*g

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 * +

nareshpd.com.np

2.3.1.4 Evaluating the Postfix expression

- Each operator in a postfix expression refers to the previous two operands in the expression.



- To evaluate the postfix expression we use the following procedure:

  Each time we read an operand we push it onto a stack. When we reach an operator, its operand s will be the top two elements on the stack. We can then pop these two elements perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.

- Consider an example

  3 4 5 * +
  =3 20 +
  =23 (answer)

2.3.1.5 Algorithm to Evaluate a 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:
   - **a** Remove the two top elements of STACK, where **A** is the top element and **B** is the next to-top element.
   - **b** Evaluate **B** ⊗ **A**.
   - **c** Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.

Trace of Evaluation:

Consider an example to evaluate the postfix expression tracing the algorithm (A = 1, B = 2, C= 3)

ABC+*CBA-+*

123+*321-+*

Scanned character	value	Op2	Op1	Result	Stack
A	1	……	…….	…….	1
B	2	……	……	…..	1 2
C	3	……	……	…..	1 2 3
+	…….	3	2	5	1 5
*	……	5	1	5	5
C	3	…….	……	……	5 3
B	2	…	…….		5 3 2
A	1	…….	…….	……	5 3 2 1
-	…….	1	2	1	5 3 1
+	…….	1	3	4	5 4
*	…….	4	5	20	20

naresh.com.np

2.3.1.6 Evaluating the Prefix Expression

To evaluate the prefix expression we use two stacks and some time it is called two stack algorithms. One stack is used to store operators and another is used to store the operands. Consider an example for this

+ 5 *3 2 prefix expression
= +5 6
=11

Illustration: Evaluate the given prefix expression

/ + 5 3 – 4 2 prefix equivalent to (5+3)/(4-2) infix notation
= / 8 – 4 2
= / 8 2
= 4

Algorithm:

```
Step 1: Accept the prefix expression
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
        (a) Scan the prefix expression
            from right, one character at a
            time.
        (b) If the scanned character is an
            operand, push it on the
            operand stack.
        (c) If the scanned character is an
            operator, then
              (i) Pop two values from the
                  operand stack
             (ii) Apply the operator on
                  the popped operands
            (iii) Push the result on the
                  operand stack
Step 3: END
```

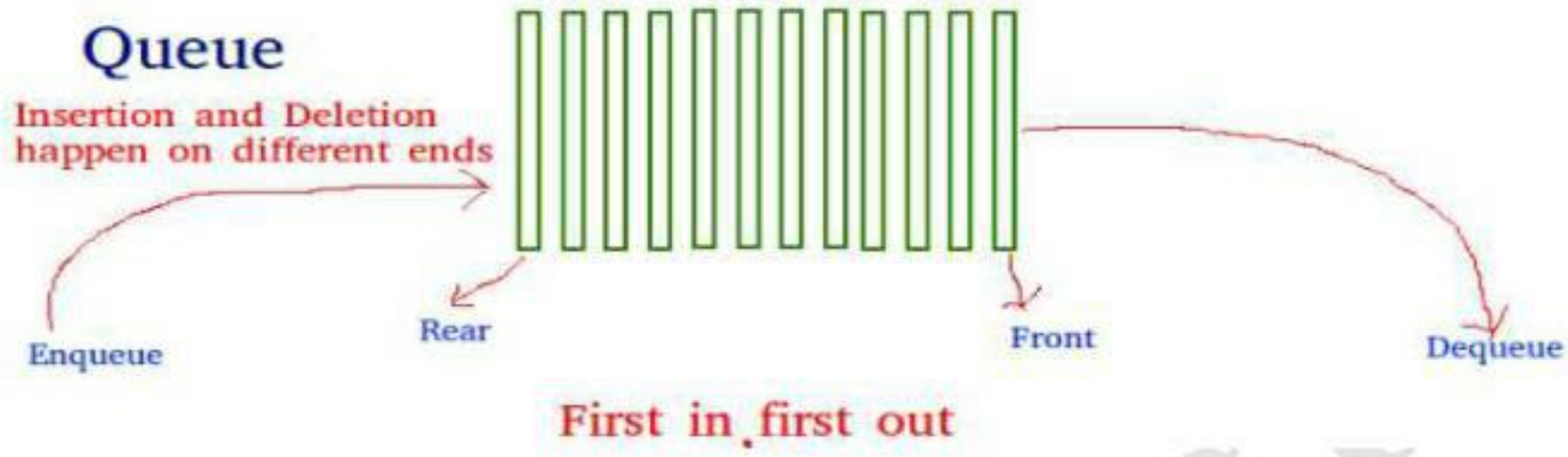
- **For example, consider the prefix expression + – 9 2 7 * 8 / 4 12. Let us now apply the algorithm to evaluate this expression**

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
–	24, 5
+	29

naresh prasad

3 Queue

- A **Queue** is an ordered collection of items from which items may be deleted at one end (called the front of the queue) and into which items may be inserted at the other end (the rear of the queue).
- The first element inserted into a queue is the first element to be removed. For this reason a queue is sometimes called **FIFO (First In First Out).**
- A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



3.1 The Queue ADT

- Queues are containers, and they hold values of some type. We must therefore speak of the ADT queue of T, where T is the type of the elements held in the queue. The carrier set of this type is the set of all queues holding elements of type T. The carrier set thus includes the empty queue, the queues with one element of type T, the queues with two elements of type T, and so forth.
- The basic operations (**Primitive Operations**) also called as Queue ADT that can be performed on queue are
  - **MakeEmpty(q):** To make q as an empty queue
  - **IsEmpty(q):** To check whether the queue q is empty. Return true if q is empty, return false otherwise.
  - **IsFull(q):** To check whether the queue q is full. Return true in q is full, return false otherwise.
  - **Enqueue(q, x) or Insert(q,x) (or add):** To insert an item x at the rear of the queue, if and only if q is not full.
  - **Dequeue(q) or Delete(q) (or remove):** To delete an item from the front of the queue q. if and only if q is not empty.
  - **Traverse (q):** To read entire queue that is display the content of the queue.

Applications of Queue:

- Task waiting for the printing
- Time sharing system for use of CPU
- For access to disk storage
- Task scheduling in operating system

Types of Queue

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

3.2 Implementation of queue:

There are two techniques for implementing the queue:

- Array implementation of queue(static memory allocation)
- Linked list implementation of queue(dynamic memory allocation)

If Queue is implemented using arrays, we must be sure about the exact number of elements we want to store in the queue, because we have to declare the size of the array at design time or before the processing starts. In this case, the beginning of the array will become the front for the queue and the last location of the array will act as rear for the queue.

3.2.1 Linear Queue (Algorithm for Queue Operations)

- **Algorithm for insertion (or Enqueue)**

QINSERT(Queue[MaxSize], data)

Step 1: if Rear = MaxSize -1
print "Overflow" and Exit
Step 2: if Front = -1 and Rear = -1
SET Front = Rear = 0
else
SET Rear = Rear + 1
[END OF IF]
Step 3: SET Queue[Rear] = data
Step 4: Exit

- **Algorithm for deletion (or Dequeue)**

QDELETE (Queue[MaxSize])

Step 1: if Front = -1 OR Front > Rear
print "Underflow" and Exit
else
SET data = Queue[Front]
SET Front = Front + 1
[END OF IF]
Step 2: EXIT

Declaration of a Queue:

```
# define MaxSize 50     /* size of the queue items*/
struct QUEUE
{
        int front;
        int rear;
        int items[MaxSize];
};
typedef struct QUEUE queue;
```

Defining the operations of linear queue:

- The **MakeEmpty** function:

```
void initQueue(queue *q)
{
        q->rear = -1;
        q->front = -1;
}
```

- The **isEmpty** function:

```
int isEmpty(queue *q)
{
        if( q->front = = -1 || q->rear < q->front)
                return 1;
        else
                return 0;
}
```

naresh.com.np

- The **isfull** function:

```
int isFull(queue *q)
{
        if(q->rear==MaxSize-1)
                return 1;
        else
                return 0;
}
```

- The **Enqueue** function:

```
void Enqueue(queue *q, int data)
{
        if(isFull(q))
        {
                printf("queue is full");
                exit(1);
        }
        if(q->rear = -1 && q->front = -1)
        {
                q->rear = 0;
                q->front = 0;
        }
        else
            q->rear++;
    q->items[q->rear] = data;

}
```

- The **Dequeue** function:

```
int Dequeue(queue *q)
{
    int data;
        if(isEmpty(q))
        {
                printf("queue is Empty");
                exit(1);
        }
        else
        {
                data = q->items[q->front];
                q->front++;
                return data;
        }
}
```

3.2.2 Problems with Linear queue implementation:

- Both rear and front indices are increased but never decreased.
- As items are removed from the queue, the storage space at the beginning of the array is discarded and never used again. Wastage of the space is the main problem with linear queue which is illustrated by the following example.

		11	22	33	44	55
0	1	2 f	3	4	5	6 r

front=2, rear=6

This queue is considered full, even though the space at beginning is vacant.

- To remove this problem, the first and an obvious solution which comes into our mind is whenever an element is deleted, shift all afterward elements to the left by one position. But if the queue is too large of say 5000 elements it will be difficult job to do and time consuming too. To remove this problem we use circular queue.

naresh pad.com.np

Representation of Queue in C

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MaxSize 50
struct QUEUE
{
        int items[MaxSize];
        int rear;
        int front;
};
typedef struct QUEUE queue;
void initQueue(queue *q)
{
        q->rear = -1;
        q->front = -1;
}
int isEmpty(queue *q)
{
        if(q->front == -1||q->rear<q->front)
                return 1;
        else
                return 0;
}
int isFull(queue *q)
{
        if(q->rear==MaxSize-1)
                return 1;
        else
                return 0;
}

void Enqueue (queue*,int);
int Dequeue (queue*);
void display(queue*);
int main()
{
        int ch,data,flag=1;
        queue lq;
        queue *q;
        q= &lq;
        initQueue(q);
        printf("Menu for program:\n");
        printf("1:insert\n2:delete\n3:display\n4:exit\n");
        do
        {
                printf("\nEnter youer choice\n");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:
                                printf("Enter data to be inserted\n");
                                scanf("%d",&data);
```

naresh pad.com.np

```
                insert(q,data);
                break;
        case 2:
                data= Dequeue(q);
                printf("Deleted item is:");
                printf("%d\n",data);
                break;
        case 3:
                display(q);
                break;
        case 4:
                flag=0;
                break;
        default:
                printf("Your choice is wrong\n");
        }
    }while(flag);
    getch();
    return 0;
}
/**********insert function*************/
void Enqueue(queue *q,int data)
{
        if(isFull(q))
        {
                printf("Queue is full\n");
                exit(1);
        }
        if(q->rear == -1 && q->front == -1)
        {
                q->rear = 0;
                q->front = 0;
        }
        else
                q->rear++;
    q->items[q->rear] = data;

}
 /***********delete function*****************/
int Dequeue(queue *q)
{
    int d;
    if(isEmpty(q))
    {
        printf("Queue is empty\n");
        exit(1);
    }
    else
    {
        d=q->items[q->front];
        q->front++;
        return d;
    }
}
```

naresh.com.np

```
/**************display function***********/
void display(queue *q)
{
        int i;
        if(isEmpty(q))
        {
                printf("Queue is empty\n");
        }
        else
        {
                printf("\nItems in queue are:");
                for(i=q->front;i<=q->rear;i++)
                {
                        printf("%d\t",q->items[i]);
                }
        }
}
```

Note:

- Wastage of the space is the main problem with linear queue,
  1. To remove this problem, the first and an obvious solution which comes into our mind is whenever an element is deleted, shift all afterward elements to the left by one position.

```
/***********Modified delete function*****************/
int Dequeue(queue *q)
{
        int d,i;
        if(isEmpty(q))
        {
                printf("Queue is empty\n");
                exit(1);
        }
        else
        {
                d=q->items[q->front];
                 for(i=q->front;i<q->rear;i++)
                        q->items[i]=q->items[i+1];
                q->rear--;
                return d;
        }
}
```

  2. To remove this problem, Another solution is don't shift elements at time of deletion do normal deletion by simply increasing front value but whenever an element need to be inserted, check if **rear** is equal to **maxsize-1** and **front** is not equal to **zero** then shift all elements so that front start from zero position.

```
/**********Modified isFull function*************/
        int isFull(queue *q)
        {
                if(q->rear==MaxSize-1 && q->front == 0)
                        return 1;
                else
                        return 0;
        }
```

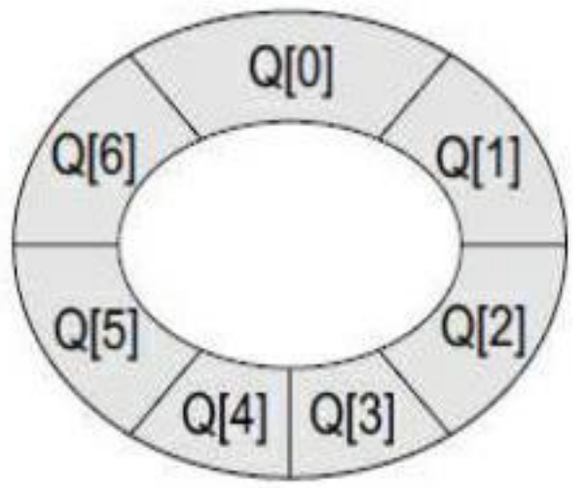
```
/**********Modified insert function*************/
void Enqueue(queue *q,int data)
{
        int i,j;
                if(isFull(q))
                {
                        printf("Queue is full\n");
                        exit(1);
                }
                if(q->rear == -1 && q->front == -1)
                {
                        q->rear = 0;
                        q->front = 0;
                }
                else if(q->rear == MaxSize-1 && q->front != 0)
                {
                        for(i=q->front, j=0; i <= q->rear; i++, j++)
                                q->items[j]=q->items[i];
                        q->rear = q->rear- q->front;
                        q->front = 0;
                }
                else
                {
                        q->rear++;
                }
        q->items[q->rear] = data;

}
```

- <u>But if the queue is too large it will be difficult job to do shifting and time consuming too (Time Complexity high). To remove this problem we use circular queue.</u>

nareshpad.com.np

3.3 Circular Queue

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- A circular queue is one in which the insertion of a new element is done at very first location of the queue if the last location of the queue is full.



- A circular queue overcomes the problem of unutilized space in linear queue implementation as array.

3.3.1 Algorithm to insert an element in a circular queue

This algorithm is assume that rear and front are initially set to MaxSize-1.

```
1. if (front==(rear+1)% MaxSize)
        print Queue is full and exit
   else
        rear=(rear+1)% MaxSize; [increment rear by 1]
2. cqueue[rear]=data;
3. end
```

3.3.2 Algorithm to delete an element from a circular queue

This algorithm is assume that rear and front are initially set to MaxSize -1.

```
1. if (rear==front) [checking empty condition]
        print Queue is empty and exit
2. front=(front+1)% MaxSize; [increment front by 1]
3. data=cqueue[front];
4. return data;
5. end
```

Declaration of a Circular Queue

```
# define MaxSize 50              /* size of the circular queue items*/
struct circularQueue
{
        int front;
        int rear;
        int items[MaxSize];
};
typedef struct circularQueue cqueue;
```

- **Operations on Circular Queue:**

The MakeEmpty function:

```
void initQueue(cqueue *q)
{
        q->rear= MaxSize -1;
        q->front= MaxSize -1;
}
```

The IsEmpty function:

naresh.com.np

```
int isEmpty(cqueue *q)
{
        if(q->rear = = q->front)
                return 1;
        else
                return 0;
}
```

The Isfull function:

```
int isFull(cqueue *q)
{
        if(q->front==(q->rear+1)% MaxSize)
                return 1;
        else
                return 0;
}
```

The Enqueue function:

```
void Enqueue(cqueue *q, int data)
{
        if(IsFull(q))
        {
                printf("queue is full");
                exit(1);
        }
        else
        {
                q->rear=(q->rear+1)% MaxSize;
                q->items[q->rear]=data;
        }
}
```

The Dequeue function:

```
int Dequeue(cqueue *q)
{
    int data;
        if(IsEmpty(q))
        {
                printf("queue is Empty");
                exit(1);
        }
        else
        {
                q->front=(q->front+1)% MaxSize;
                data = q->items[q->front];
                return data;
        }
}
```

nareshpd.com.np

Representation of circular queue in C, with secrifying one cell (One Position Vacant)

```
#include<stdio.h>
#include<conio.h>
#define MaxSize 50
struct circularQueue
{
        int items[MaxSize];
        int rear;
        int front;
};
typedef struct circularQueue cqueue;
void initQueue(cqueue *q)
{
        q->rear= MaxSize -1;
        q->front= MaxSize -1;
}
int isEmpty(cqueue *q)
{
        if(q->rear==q->front)
                return 1;
        else
                return 0;
}
int isFull(cqueue *q)
{
        if(q->front==(q->rear+1)% MaxSize)
                return 1;
        else
                return 0;
}

void Enqueue(cqueue*);
void Dequeue(cqueue*);
void Display(cqueue*);
void main()
{
        int ch,flag=1;
        cqueue *q;
        cqueue cq;
        q = &cq;
        initQueue(q);
        printf("Menu for program:\n");
        printf("1:insert\n2:delete\n3:display\n4:exit\n");
                do
                {
                        printf("\nEnter youer choice\n");
                        scanf("%d",&ch);
                        switch(ch)
                        {
                                case 1:
                                        Enqueue(q);
                                        break;
                                case 2:
```

meshpad.com.np

```
                Dequeue(q);
                break;
        case 3:
                Display(q);
                break;
        case 4:
                flag = 0;
                break;
        default:
                printf("Your choice is wrong\n");
        }
        }while(flag);
    getch();
}
/*********insert function************/
void Enqueue(cqueue *q)
{
    int d;
        if(isFull(q))
            printf("Queue is full\n");
        else
        {
            q->rear=(q->rear+1)%MaxSize;
            printf ("Enter data to be inserted\n");
            scanf("%d",&d);
            q->items[q->rear]=d;
        }
}
/*********delete function***************/
void Dequeue(cqueue *q)
{
    int data;
    if(isEmpty(q))
        printf("Queue is empty\n");
    else
    {
        q->front=(q->front+1)%MaxSize;
        data = q->items[q->front];
        printf("Deleted item is:");
        printf("%d\n",data);

    }
}
/*************display function**********/
void Display(cqueue *q)
{
    int i;
    if(isEmpty(q))
        printf("Queue is empty\n");
    else
    {
        printf("Items of queue are:\n");
        for(i=(q->front+1)%MaxSize;i!=q->rear;i=(i+1)%MaxSize)
        {
```

naresh.com.np

```
                        printf("%d\t",q->items[i]);
                }
                printf("%d\t",q->items[q->rear]);
        }
}
```

Representation of circular queue in C, without secrifying one cell by using a count variable

```
#include<stdio.h>
#include<conio.h>
#define MaxSize 5
struct circularQueue
{
        int items[MaxSize];
        int rear;
        int front;
        int count;
};
typedef struct circularQueue cqueue;

void initQueue(cqueue *q)
{
        q->rear= MaxSize -1;
        q->front= MaxSize -1;
        q->count = 0;
}
int isEmpty(cqueue *q)
{
        if(q->count==0)
                return 1;
        else
                return 0;
}
int isFull(cqueue *q)
{
        if(q->count == MaxSize)
                return 1;
        else
                return 0;
}

void Enqueue(cqueue*);
void Dequeue(cqueue*);
void Display(cqueue*);
void main()
{
        int ch,flag=1;
        cqueue *q;
        cqueue cq;
        q = &cq;
        initQueue(q);
        printf("Menu for program:\n");
        printf("1:insert\n2:delete\n3:display\n4:exit\n");
```

nareshpad.com.np

```
do
{
        printf("\nEnter youer choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:
                        Enqueue(q);
                        break;
                case 2:
                        Dequeue(q);
                        break;
                case 3:
                        Display(q);
                        break;
                case 4:
                        flag = 0;
                        break;
                default:
                        printf("Your choice is wrong\n");
                }
        }while(flag);
    getch();
}
/*********insert function************/
void Enqueue(cqueue *q)
{
    int d;
        if(isFull(q))
                printf("Queue is full\n");
        else
        {
                q->rear=(q->rear+1)%MaxSize;
                printf ("Enter data to be inserted\n");
                scanf("%d",&d);
                q->items[q->rear]=d;
                q->count++;
        }
}
/*********delete function****************/
void Dequeue(cqueue *q)
{
    int data;
    if(isEmpty(q))
        printf("Queue is empty\n");
    else
    {
        q->front=(q->front+1)%MaxSize;
        data = q->items[q->front];
        printf("Deleted item is:");
        printf("%d\n",data);
        q->count--;
    }
}
```

nareshpad.com.np

```
/**************display function***********/
void Display(cqueue *q)
{
        int i;
        if(isEmpty(q))
                printf("Queue is empty\n");
        else
        {
                printf("Items of queue are:\n");
                for(i=(q->front+1)%MaxSize;i!=q->rear;i=(i+1)%MaxSize)
                {
                        printf("%d\t",q->items[i]);
                }
                printf("%d\t",q->items[q->rear]);
        }
}
```

- **Representation of circular queue in C**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MaxSize 10
struct CircularQueue
{
   int items[MaxSize];
   int rear,front;
};
typedef struct CircularQueue cqueue;
void initialize(cqueue *q);
int isEmpty(cqueue *q);
int isFull(cqueue *q);
void Insert(cqueue *q);
void Delete(cqueue *q);
void display(cqueue *q);
int main()
{
   int ch,flag=1;
   cqueue cq;
   cqueue *q;
   q= &cq;
   initialize(q);
   printf("\n1.Insert(Rear)\n2.Delet(Front)\n5.Print\n6.Exit\n");
   do
   {
      printf("\nEnter your choice:");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1:
               Insert(q);
               break;
         case 2:
               Delete(q);
               break;
```

freshpad.com.np

```
        case 3:
                display(q);
                break;
        case 4:
                flag = 0;
                break;
        default:
                printf("Your choice is wrong\n");
    }
}while(flag);

    getch();
    return 0;
}
void initialize(cqueue *q)
{
    q->rear=-1;
    q->front=-1;
}
int isEmpty(cqueue *q)
{
    if(q->rear== -1)
        return 1;
    else
            return 0;
}
int isFull(cqueue *q)
{
    if((q->rear+1)%MaxSize==q->front)
        return 1;
    else
            return 0;
}
void Insert(cqueue *q)
{
            int d;
    if(isFull(q))
            {
                    printf("Queue is Full");
                    return;
            }
    else if(isEmpty(q))
    {
            q->rear=0;
            q->front=0;
    }
    else
    {
        q->rear=(q->rear+1)%MaxSize;
    }
    printf ("Enter data to be inserted\n");
            scanf("%d",&d);
            q->items[q->rear]=d;
}
```

nareshpad.com.np

```
void Delete(cqueue *q)
{
   int data;
        if(isEmpty(q))
        {
                printf("Queue is Empty");
                return;
         }
   data = q->items[q->front];
        if(q->front == q->rear)                    //delete the last element
                initialize(q);
        else
                q->front=(q->front+1)%MaxSize;
   printf("Deleted item is:%d",data);
}
void display(cqueue *q)
{
        int i;
        if(isEmpty(q))
                printf("Queue is empty\n");
        else
        {
                printf("Items of queue are:\n");
                i=q->front;
                while(i!=q->rear)
                {
                        printf("\n%d",q->items[i]);
                        i=(i+1)%MaxSize;
                }
                printf("\n%d\n",q->items[q->rear]);
        }
}
```

3.3.3 Application of Circular Queue

- **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
- **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
- **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

naresh.com.np

3.4 DEQUE

A deque is a homogeneous list in which elements can be added or inserted (called Enqueue operation) and deleted or removed (called Dequeue operation) from both the ends. 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.

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

- **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.
- **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.

Declaration of a Queue:

```
# define MaxSize 50     /* size of the queue items*/
struct QUEUE
{
        int front;
        int rear;
        int items[MaxSize];
};
typedef struct QUEUE queue;
```

The possible operation (Deque ADT) performed on deque is

- Insertion of an element at the REAR end of the queue
- Deletion of an element from the FRONT end of the queue
- Insertion of an element at the FRONT end of the queue
- Deletion of an element from the REAR end of the queue

Functions to carry out these four operations using linear array are:

- **Insertion of an element at the Rear end of the queue**

```
void dqinsert_rear(queue *q, int data)
{
        if(q->rear == (MaxSize - 1))
        {
                printf("Queue is full");
                exit(1);
        }
        else
        {
                q->rear = q->rear + 1;
                q->items[q->rear] = data;
        }
}
```

- **Deletion of an element from the FRONT end of the queue**

```
int dqdelete_front(queue *q)
{
    int data;
    if(q->front == q->rear)
    {
            printf("Queue is empty");
            exit(1);
```

```
        }
        else
        {
                data = q->items[q->front];
                q->front = q->front + 1;
                return data;
        }
}
```

- **Insertion of an element at the FRONT end of the queue**

```
void dqinsert_front(queue *q, int data)
{
        if(q->front == 0)
        {
                printf("Queue is full");
                exit(1);
        }
        else if(q->front = -1 && q->rear = -1)
        {
                q->front = 0;
                q->rear = 0;
        }
        else
                q->front = q->front - 1;
    q->items[q->front] = data;
}
```

- **Deletion of an element from the REAR end of the queue**

```
int dqdelete_rear(queue *q)
{
    int data;
    if(q->front == q->rear)
    {
            printf("Queue is empty");
            exit(1);
    }
    else
    {
            data = q->items[q->rear];
            q->rear = q->rear - 1;
            return data;
    }
}
```

- **Function to display the contents (Status) of a queue**

```
void dq_display(queue *q)
{
    int i;
    if(q->front <= q->rear)
```

nareshpad.com.np

```
        {
                printf("Status of the Queue");
                for(i=q->front;i<=q->rear;i++)
                        printf("%d",q->items[i]);
        }
        else
                printf("Queue is Empty")
}
```

- **Implementation of Deque or Double Ended Queue using circular array**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MaxSize 5
struct Deque
{
   int items[MaxSize];
   int rear,front;
};
typedef struct Deque deque;
void initialize(deque *q);
int isEmpty(deque *q);
int isFull(deque *q);
void insert_rear(deque *q);
void insert_front(deque *q);
void delete_front(deque *q);
void delete_rear(deque *q);
void display(deque *q);
int main()
{
    int ch,flag=1;
   deque dq;
   deque *q;
   q= &dq;
   initialize(q);
   printf("\n1.Insert(Rear)\n2.Insert(Front)\n3.Delete(Rear)\n4.Delet(Front)\n5.Print\n6.Exit\n");
   do
   {
      printf("\nEnter your choice:");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1:
               insert_rear(q);
               break;
         case 2:
               insert_front(q);
               break;
         case 3:
               delete_rear(q);
               break;
         case 4:
               delete_front(q);
```

naresh pad.com.np

```
            break;
        case 5:
            display(q);
            break;
        case 6:
            flag = 0;
            break;
        default:
            printf("Your choice is wrong\n");
    }
}while(flag);

    getch();
    return 0;
}
void initialize(deque *q)
{
    q->rear=-1;
    q->front=-1;
}
int isEmpty(deque *q)
{
    if(q->rear== -1)
        return 1;
    else
            return 0;
}
int isFull(deque *q)
{
    if((q->rear+1)%MaxSize==q->front)
        return 1;
    else
            return 0;
}
void insert_rear(deque *q)
{
            int d;
    if(isFull(q))
    {
        printf("Queue is Full");
        return;
    }
    else if(isEmpty(q))
    {
            q->rear=0;
            q->front=0;
    }
    else
            q->rear=(q->rear+1)%MaxSize;
    printf ("Enter data to be inserted\n");
            scanf("%d",&d);
            q->items[q->rear]=d;
}
void insert_front(deque *q)
```

nareshpad.com.np

```
{
        int d;
        if(isFull(q))
        {
                printf("Queue is Full");
                return;
        }
         else if(isEmpty(q))
        {
                 q->rear=0;
                 q->front=0;
         }
        else
                q->front=(q->front-1+MaxSize)%MaxSize;
        printf ("Enter data to be inserted\n");
        scanf("%d",&d);
        q->items[q->front]=d;
}
void delete_front(deque *q)
{
   int data;
   if(isEmpty(q))
   {
      printf("Queue is Empty");
      return;
   }
   data = q->items[q->front];
   if(q->front == q->rear)                          //delete the last element
        initialize(q);
   else
        q->front=(q->front+1)%MaxSize;
   printf("Deleted item is:%d",data);
}
void delete_rear(deque *q)
{
   int data;
        if(isEmpty(q))
        {
                printf("Queue is Empty");
                return;
        }
        data=q->items[q->rear];
        if(q->front == q->rear)                     //delete the last element
                initialize(q);
        else
             q->rear=(q->rear-1+MaxSize)%MaxSize;
   printf("Deleted item is:%d",data);
}
void display(deque *q)
{
        int i;
        if(isEmpty(q))
                printf("Queue is empty\n");
        else
```

naresh.com.np

```
        {
                printf("Items of queue are:\n");
                i=q->front;
                while(i!=q->rear)
                {
                        printf("\n%d",q->items[i]);
                        i=(i+1)%MaxSize;
                }
                printf("\n%d\n",q->items[q->rear]);
        }
}
```

3.5 Priority Queues

- A priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed comes from the following rules:.
  - An element of higher priority is processed before any element of lower priority.
  - If two elements has same priority then they are processed according to the order in which they were added to the queue.
- The best application of priority queue is observed in CPU scheduling.
  - The jobs which have higher priority are processed first.
  - If the priority of two jobs is same this jobs are processed according to their position in queue.
  - A short job is given higher priority over the longer one.

Types of priority queues:

- **Ascending priority queue(min priority queue):** An ascending priority queue is a collection of items into which items can be inserted arbitrarily but from which only the smallest item can be removed.
- **Descending priority queue(max priority queue):** An descending priority queue is a collection of items into which items can be inserted arbitrarily but from which only the largest item can be removed

- 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.

3.6 APPLICATION OF QUEUES

- Round robin techniques for processor scheduling is implemented using queue.
- Printer server routines (in drivers) are designed using queues.
- All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.
- Disk Driver: maintains a queue od disk input/output requests
- Scheduler (e.g, in operating system): maintains a queue of processes awaiting a slice of machine time.
- Call center phone systems will use a queue to hold people in line until a service representative is free.
- 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.
- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- 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.

4 Static and Dynamic List

We have studied that an array is a linear collection of data elements in which the elements are stored in consecutive memory locations. While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store. For example, if we declare an array as int marks[10], then the array can store a maximum of 10 data elements but not more than that. But what if we are not sure of the number of elements in advance? Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations. So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

Linked list is a data structure that is free from the aforementioned restrictions. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it. However, unlike an array, a linked list does not allow random access of data. Elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time

- **Self-referential Structure**

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. Hence, a structure which contains a reference to itself is called self-referential structure. In general terms, this can be expressed as:

```
struct node
{
        member 1;
        member 2;
        .......
        struct node *ptr;
};
```

For example:

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

This is a structure of type node. The structure contains two members: a info integer member, and a pointer to a structure of the same type (i.e., a pointer to a structure of type node), called next. Therefore this is a self-referential structure.

- **LIST**

- A list is a sequential data structure. It differ from the stack and queue data structures in that additions and removals can be made at any position in the 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.
- **Contiguous list:** If the elements are stored such that they share common adjacent memory boundary as in case of array then the list is known as contiguous list.

  Note:

  An array stores elements in successive order in memory. This means, elements are one after another consecutive in memory.
- The position of each element is given by an index from 0 to n-1, where n is the number of elements.
- Given any index, the element with that index can be accessed in constant time, i.e. the time to access does not depend on the size of the list.

- To add an element at the end of the list, the time taken does not depend on the size of the list. However, the time taken to add an element at any other point in the list does depend on the size of the list, as all subsequent element must be shifted up. Addition near the start of the list take longer than additions near the middle or end.
- When an element is removed, subsequent elements must be shifted down, so removals near the start of the list take longer than the removals near the middle or end.

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**.

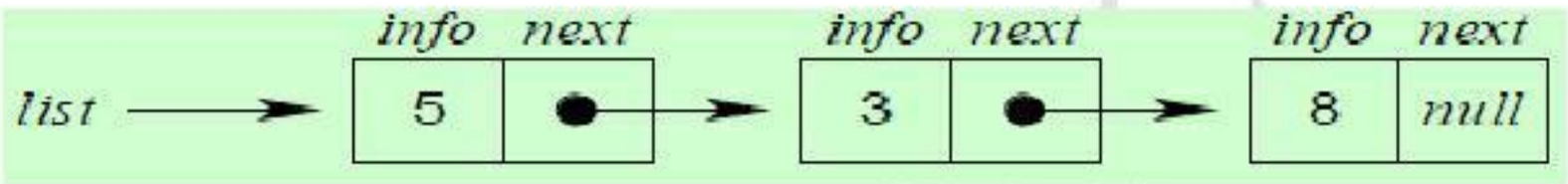
4.3 Linked List

- **Definition**:

  Linear linked list is a data structure of explicit ordering of items and each item contains two portions; one is information portion and the other one is next address portion.

  In explicit ordering of items, each item contained within itself the address of another item. Such an explicit ordering gives rise to a data structure which is known as a linear linked list.

- A linked list is a collection of nodes, where each node consists of two parts:
  - **info**: the actual element to be stored in the list. It is also called data field.
  - **link**: one or two links that points to next and previous node in the list. It is also called next or pointer field.



  - The nodes in a linked list are not stored contiguously in the memory
  - You don't have to shift any element in the list.
  - Memory for each node can be allocated dynamically whenever the need arises.
  - The size of a linked list can grow or shrink dynamically

Advantages of Linked List

- **Linked lists are dynamic data structures.** That is, they can grow or shrink during the execution of a program
- **Efficient memory utilization.** Here, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (removed) when it is no longer needed.
- **Insertion and deletions are easier and efficient:** Linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
- Many complex applications can be easily carried out with linked lists.

Disadvantages of Linked List

- More memory: If the number of fields are more, then more memory space is needed.
- Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
- Extra memory space for a pointer is required with each element of the list.

4.4 Operations on linked list

The basic operations to be performed on the linked list are as follows:

- **Creation**: This operation is used to create a linked list
- **Insertion**: This operation is used to insert a new nose in a kinked list in a specified position. A new node may be inserted
  - ✓ At the beginning of the linked list
  - ✓ At the end of the linked list
  - ✓ At the specified position in a linked list

- **Deletion**: The deletion operation is used to delete a node from the linked list. A node may be deleted from
  - ✓ The beginning of the linked list
  - ✓ the end of the linked list
  - ✓ the specified position in the linked list.
- **Traversing**: The list traversing is a process of going through all the nodes of the linked list from one end to the other end. The traversing may be either forward or backward.
- **Searching** or **find**: This operation is used to find an element in a linked list. In the desired element is found then we say operation is successful otherwise unsuccessful.
- **Concatenation**: It is the process of appending 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
- **Display**: This operation is used to print each and every node's information.

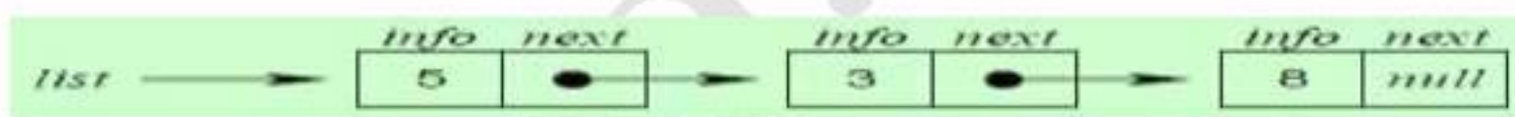
4.5 Types of Linked List

Basically we can put linked list into the following four types:

- Singly Linked List
- Doubly linked List
- Circular Linked List
  - Circular Singly Linked List
  - Circular Doubly Linked List

4.5.1 Singly Linked List

➢ A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made. In this type of linked list each node contains two fields one is info field which is used to store the data items and another is link field that is used to point the next node in the list. The last node has a NULL pointer.



➢ We can create a structure for the singly linked list the each node has two members, one is info that is used to store the data items and another is next field that store the address of next node in the list.

```
struct Node
{
        int info;
        struct Node *next;
};

typedef struct Node Lnode;
Lnode  *START;                    //START is a pointer type structure variable
```

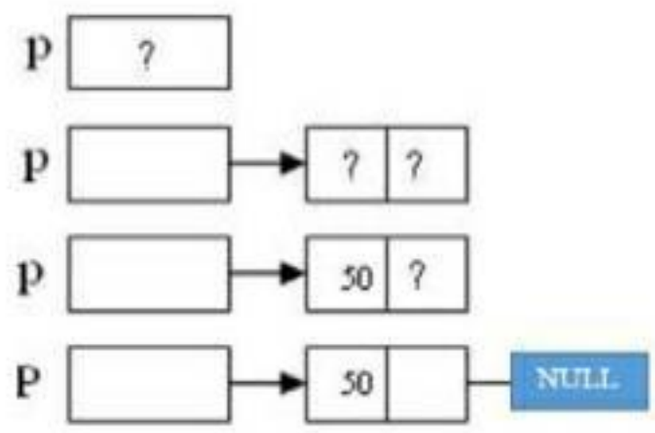
Creating a Node:

✓ To create a new node, we use the **malloc** function to dynamically allocate memory for the new node.

✓ After creating the node, we can store the new item in the node using a pointer to that node.

The following steps clearly shows the steps required to create a node and storing an item

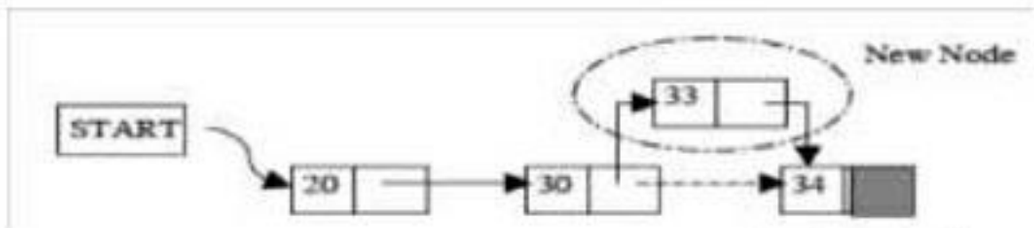
```
Lnode *p;
p = (Lnode*)malloc(sizeof(Lnode));
p->info = 50;
p->next = NULL;
```



Note that **p** is not a node; instead it is a pointer to a node.

4.5.1.1 Inserting Nodes

- To insert an element or a node in a linked list, the following three things to be done:
  - Allocating a node
  - Assigning a data to info field of the node
  - Adjusting the pointer.
- Inserting a new node into the linked list has the following three instances:
  - Insertion at the beginning of the list
  - Insertion at the end of the list
  - Insertion at the specified position within the list



Algorithm to insert a node at the beginning of the singly linked list:

Let ***START** be the pointer to first node in the current list

1. Input DATA in newItem to be inserted
2. Create a new node from memory and assign its address to **NewNode**
   NewNode=(Lnode*)malloc(sizeof(Lnode));
3. Assign data to the **info** field of new node
   NewNode->info=newItem;
4. Set **next** of new node to **START**
   NewNode->next=START;
5. Set the **START** pointer to the new node
   START=NewNode;
6. End

Algorithm to insert a node at the end of the singly linked list:

Let ***START** be the pointer to first node in the current list and **TEMP** is a temporary pointer to hold the node address

1. Input DATA in newItem to be inserted
2. Create a new node from memory and assign its address to NewNode
   NewNode=(Lnode*)malloc(sizeof(Lnode));
3. Assign data to the **info** field of new node and Set NULL to the next of new node
   NewNode->info=newItem;
   NewNode -> next = NULL
4. If (SATRT equal to NULL)
   START = NewNode
   Else
   TEMP = START
   While (TEMP -> next not equal to NULL)
   TEMP = TEMP->next
   TEMP ->next = NewNode
5. Exit

Algorithm to insert a node at the specified position in a singly linked list:

Let ***START** be the pointer to first node in the current list and **TEMP** is a temporary pointer to hold the node address

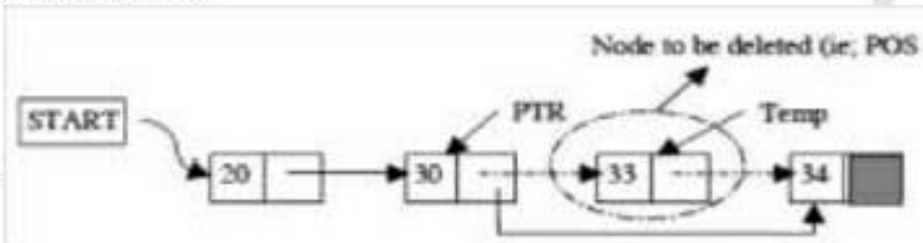
1. Input DATA and POS to be inserted
2. Initialize TEMP = START; and k = 0
3. Repeat the step 3 while (k is less than POS)
   I. TEMP = TEMP->next
   II. If (TEMP is equal to NULL)
      a) **Display "Node in the list less than the position"**
      b) Exit
   III. k = k + 1

naresh.com

4. Create a new node from memory and assign its address to **NewNode**
   NewNode=(Lnode*)malloc(sizeof(Lnode));
5. Assign data to the **info** field of new node and Set **TEMP → next** to the **next** of new node
   NewNode->info=newItem;
   NewNode -> next = TEMP->next
6. TEMP → next = NewNode
7. End

4.5.1.2 Deleting Nodes

- Deleting a node from the linked list has the following three instances
  - Deleting the first node of the linked list
  - Deleting the last node of the linked list
  - Deleting the specified node within the linked list



Algorithm to delete the first node of the singly linked list

Let ***START** be the pointer to first node in the current list and **PTR** is a pointer to hold the node address

Step 1: IF START = NULL
  Write UNDERFLOW
  Go to Step 5
  [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START->next
Step 4: Print, element deleted is PTR->info
Step 5: FREE PTR
Step 6: EXIT

Algorithm to delete the last node of the singly linked list

Let ***START** be the pointer to first node in the current list, **PTR** is a pointer to hold the node address and another pointer variable **PREPTR** such that it always points to one node before the **PTR**

Step 1: IF START = NULL
  Write UNDERFLOW
  Go to Step 10
  [END OF IF]
Step 2: SET PTR = START
Step 3: If PTR->next = NULL
    SET START = NULL
    Go to Step 8
  [END OF IF]
Step 4: Repeat Steps 5 and 6 while PTR ->next != NULL
Step 5: SET PREPTR = PTR
Step 6: SET PTR = PTR->next
  [END OF LOOP]
Step 7: SET PREPTR->next = NULL
Step 8: Print, element deleted is PTR->info
Step 9: free PTR
Step 10: EXIT

Algorithm to delete the node at the specified position of the singly linked list

Let ***START** be the pointer to first node in the current list, **PTR** is a pointer to hold the node address and another pointer variable **PREPTR** such that it always points to one node before the **PTR**

1. IF START = NULL
   print UNDERFLOW and exit
2. Enter position of a node at which you want to delete a new node. Let this position is POS.

3. Set PTR=START and k = 0
4. Repeat the step 4 while (k is less than POS)
    I. Set PREPTR = PTR
    II. PTR = PTR->next
    III. If (PTR is equal to NULL)
        c) **Display "Node in the list less than the position"**
        d) Exit
    IV. k = k + 1

5. Set PREPTR->next = PTR->next
6. free PTR
7. Exit

- **Algorithm to print the number of nodes in a linked list**

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:         SET COUNT = COUNT +1
Step 5:         SET PTR = PTR->next
                [END OF LOOP]
Step 6: Print COUNT
Step 7: EXIT
```

- **Algorithm for traversing a linked list**

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:         Apply Process to PTR->info
Step 4:         SET PTR = PTR->next
        [END OF LOOP]
Step 5: EXIT
```

- **Algorithm to search a linked list**

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3: IF VAL = PTR->info
                SET POS = PTR
                Go To Step 5
        ELSE
                SET PTR = PTR->next
                [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node. In Step 2, a while loop is executed which will compare every **node's DATA(info) with VAL for which the search is being made.** If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

naresh.com.np

Representation of Linked list in C

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
   int info;
   struct node *next;
};
typedef struct node Lnode;
Lnode *start = NULL;
void create_linkedlist();
void display();
void insert_beg();
void insert_end();
void insert_position();
void delete_beg();
void delete_end();
void delete_position();
void delete_linkedlist();
void search_linkedlist();
int main()
{
    int option,flag=1;
    printf("\n\n *****MAIN MENU *****");
   printf("\n 1: Create a list");
   printf("\n 2: Display the list");
   printf("\n 3: Add a node at the beginning");
   printf("\n 4: Add a node at the end");
   printf("\n 5: Add a node at nth position");
   printf("\n 7: Delete a node from the beginning");
   printf("\n 8: Delete a node from the end");
   printf("\n 9: Delete nth position node");
   printf("\n 10: Delete the entire list");
   printf("\n 11: Search in the linked list");
   printf("\n 12: EXIT");
   do
   {
            printf("\n\n Enter your option : ");
            scanf("%d", &option);
            switch(option)
            {
   case 1:
                        create_linkedlist();
                        printf("\n LINKED LIST CREATED");
                        break;
   case 2:
                        display();
                        break;
   case 3:
                        insert_beg();
                        break;
   case 4:
                        insert_end();
```

naresh.com.np

```
                    break;
case 5:
                    insert_position();
                    break;
case 7:
                    delete_beg(start);
                    break;
case 8:
                    delete_end();
                    break;
case 9:
                    delete_position();
                    break;
          case 10:
                    delete_linkedlist();
                    printf("\n LINKED LIST DELETED");
                    break;
          case 11:
                    search_linkedlist();
                    break;
          case 12:
                    flag = 0;
                    break;
          default:
                    printf("Invalide Choice");
     }
  }while(flag);
  getch();
  return 0;
}
void create_linkedlist()
{
  Lnode *new_node, *ptr;
  int data,ch;
  do
  {
          printf("\n Enter the data : ");
          scanf("%d",&data);
          new_node = (Lnode*)malloc(sizeof(Lnode));
     new_node->info=data;
          if(start==NULL)
          {
                    new_node->next = NULL;
                    start=new_node;
          }
          else
          {
                    ptr=start;
                    while(ptr->next!=NULL)
                              ptr=ptr->next;
                    ptr->next = new_node;
                    new_node->next=NULL;
          }
  printf("\n Do you want to add more data then press 1:");
```

naresh.com.np

```
    scanf("%d",&ch);
    }while(ch == 1);

}
void display()
{
    Lnode *ptr;
    if(start == NULL)
    {
            printf("\nList is empty\n");
            return;
    }
    ptr = start;
    printf("\nLinked List Data are:\n");
    while(ptr!=NULL)
    {
            printf("\t %d", ptr->info);
            ptr=ptr->next;
    }
}
void insert_beg()
{
    Lnode *new_node;
    int data;
    printf("\n Enter the data : ");
    scanf("%d", &data);
    new_node=(Lnode *)malloc(sizeof(Lnode));
    new_node->info=data;
    new_node->next=start;
    start=new_node;
}
void insert_end()
{
    Lnode *ptr, *new_node;
    int data;
    printf("\n Enter the data : ");
    scanf("%d", &data);
    new_node = (Lnode *)malloc(sizeof(Lnode));
    new_node->info = data;
    new_node->next = NULL;
    if(start == NULL)
            start = new_node;
    else
    {
            ptr = start;
            while(ptr->next != NULL)
                    ptr = ptr -> next;
            ptr->next = new_node;
    }
}
void insert_position()
{
    Lnode *new_node,*ptr,*preptr;
    int data,pos,k;
```

nepalesh pad.com.np

```
printf("\n Enter the data :");
scanf("%d", &data);
printf("\n Enter the position at which the data has to be inserted : ");
scanf("%d", &pos);
ptr = start;
k=0;
while(k<pos)
{
        ptr=ptr->next;
        if(ptr==NULL)
        {
                printf("Node in the list less than the position");
                return;
        }
        k++;
  }
  new_node = (Lnode *)malloc(sizeof(Lnode));
  new_node->info = data;
  new_node->next = ptr->next;
  ptr->next = new_node;
}

void delete_beg()
{
  Lnode *ptr;
  if(start == NULL)
  {
        printf("\nList is empty\n");
        return;
  }
  ptr = start;
  start = start -> next;
  printf("\nDleted data from begining is:%d\n",ptr->info);
  free(ptr);
}
void delete_end()
{
  Lnode *ptr,*preptr;
  ptr = start;
  if(start == NULL)
  {
        printf("\nList is empty\n");
        return;
  }
  while(ptr -> next != NULL)
  {
        preptr = ptr;
        ptr = ptr -> next;
  }
  preptr -> next = NULL;
  printf("\nDleted data from end is:%d\n",ptr->info);
  free(ptr);
}
void delete_position()
```

nareshpad.com.np

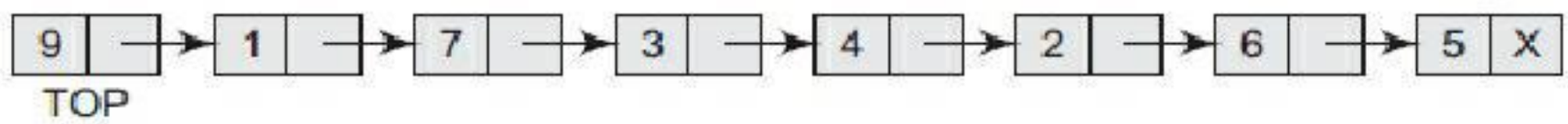
```
{
   Lnode *ptr,*preptr;
   int pos,k;
   if(start == NULL)
   {
            printf("\nList is empty\n");
            return;
   }
   printf("\n Enter the index position of the node which has to be deleted : ");
   scanf("%d",&pos);
   ptr = start;
   for(k=0;k<pos;k++)
   {
            preptr = ptr;
            ptr = ptr->next;
            if(ptr==NULL)
            {
                     printf("Node in the list is less than entered position");
                     return;
            }
   }
   preptr->next = ptr->next;
   printf("\n Deleted data at position %d is %d",pos,ptr->info);
   free(ptr);
}
void delete_linkedlist()                    //Delete complete linked list
{
   Lnode *ptr;
   if(start == NULL)
   {
            printf("\nList is empty\n");
            return;
   }
   while(start != NULL)
   {
            ptr=start;
            start = start->next;
            printf("\nDeleted data is %d\n", ptr->info);
            free(ptr);
   }
}
void search_linkedlist()
{
   Lnode *ptr,pos;
   int data;
   printf("\nEnter value you want to search:");
   scanf("%d",&data);
   ptr=start;
   while(ptr != NULL)
   {
            if(ptr->info == data)
            {
                     printf("\n%d found at address %u\n",ptr->info,ptr);
                     return;
```

naresh pad.com.np

```
            }
            ptr = ptr->next;
        }
        printf("\nEntered value is not found in the linked list\n");
    }
```

4.5.2 Linked List Implementation of Stack

- We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.
- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.



4.5.2.1 OPERATIONS ON A LINKED STACK

- **Push Operation**

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

In Step 1, memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. This is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

- **Pop Operation**

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

In Step 1, we first check for the UNDERFLOW condition. In Step 2, we use a pointer PTR that points to TOP. In Step 3, TOP is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool

Representation Of Linked Stack in C

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct STACK
{
    int data;
    struct STACK *next;
};
typedef struct STACK stack;
stack *top = NULL;
void push(int);
void display();
void pop();
int main()
{
    int val,option,flag=1;
            printf("\n *****MAIN MENU*****");
            printf("\n 1. PUSH");
            printf("\n 2. POP");
            printf("\n 3. DISPLAY");
            printf("\n 4. EXIT");
    do
    {
            printf("\n Enter your option: ");
            scanf("%d", &option);
            switch(option)
            {
                case 1:
                        printf("\n Enter the number to be pushed on stack: ");
                        scanf("%d", &val);
                        push(val);
                        break;
                case 2:
                        pop();
                        break;
                case 3:
                        display();
                        break;
                case 4:
                        flag=0;
                        break;
                default:
                        printf("Invalid Choice");
            }
    }while(flag);
    getch();
    return 0;
}
void push(int val)
{
    stack *ptr;
    ptr = (stack*)malloc(sizeof(stack));
```

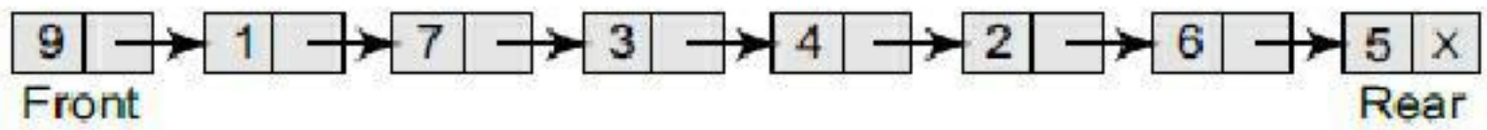
nareshpad.com.np

```
ptr->data = val;
if(top == NULL)
{
        ptr->next = NULL;
        top=ptr;
}
else
{
        ptr->next=top;
        top=ptr;
}
}
void display()
{
    stack *ptr;
    ptr = top;
    if(top == NULL)
            printf("\n STACK IS EMPTY");
    else
    {
            printf("\nData in stack are:\n");
            while(ptr != NULL)
            {
                    printf("\t %d", ptr->data);
                    ptr = ptr->next;
            }
    }
}
void pop()
{
    stack *ptr;
    ptr = top;
    if(top == NULL)
            printf("\n STACK UNDERFLOW");
    else
    {
            top = top->next;
            printf("\n Data being deleted is: %d", ptr->data);
            free(ptr);
    }
}
```

naresh.com.np

4.5.3 Linked list implementation of queue

- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty.



Algorithm to insert an element in a linked queue

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

In Step 1, the memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked queue. This is done by checking if FRONT = NULL. If this is the case, then the new node is tagged as FRONT as well as REAR. Also NULL is stored in the NEXT part of the node (which is also the FRONT and the REAR node). However, if the new node is not the first node in the list, then it is added at the REAR end of the linked queue (or the last node of the queue).

Algorithm to delete an element from a linked queue

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

In Step 1, we first check for the underflow condition. If the condition is true, then an appropriate message is displayed, otherwise in Step 2, we use a pointer PTR that points to FRONT. In Step 3, FRONT is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

Representation of Linked Queue in C

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct Node
{
        int data;
        struct Node *next;
};
typedef struct Node node;
struct Queue
{
        node *front;
        node *rear;
};
typedef struct Queue queue;
queue *q;
```

```
void create_queue();
void insert(int);
void delete_element();
void display();
int main()
{
        int val,option,flag=1;
        create_queue();
                printf("\n *****MAIN MENU*****");
                printf("\n 1. INSERT");
                printf("\n 2. DELETE");
                printf("\n 3. DISPLAY");
                printf("\n 4. EXIT");
        do
        {
                printf("\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                        case 1:
                                printf("\n Enter the number to insert in the queue:");
                                scanf("%d", &val);
                                insert(val);
                                break;
                        case 2:
                                delete_element();
                                break;
                        case 3:
                                display();
                                break;
                        case 4:
                                flag=0;
                                break;
                        default:
                                        printf("\nInvalid Choice\n");
                }
        }while(flag);
        getch();
        return 0;
}
void create_queue()
{
        q = (queue*)malloc(sizeof(queue));
        q ->rear = NULL;
        q ->front = NULL;
}
void insert(int val)
{
        node *ptr;
        ptr = (node*)malloc(sizeof(node));
        ptr->data = val;
        if(q->front == NULL)
        {
                q->front = ptr;
```

naresh.com.np

```
                q->rear = ptr;
                q->front->next=q->rear->next = NULL;
        }
        else
        {
                q->rear->next=ptr;
                q->rear=ptr;
                q->rear->next=NULL;
        }
}
void display()
{
        node *ptr;
        ptr=q->front;
        if(ptr == NULL)
                printf("\n QUEUE IS EMPTY");
        else
        {
                printf("\nData in queue are:\n");
                while(ptr!=q->rear)
                {
                        printf("%d\t", ptr -> data);
                        ptr=ptr->next;
                }
                printf("%d\t", ptr->data);
        }
}
void delete_element()
{
        node *ptr;
        ptr= q->front;
        if(q->front == NULL)
                printf("\n UNDERFLOW");
        else
        {
                q->front= q->front->next;
                printf("\n The value being deleted is : %d", ptr->data);
                free(ptr);
        }
}
```

4.5.3.1 Advantages of Singly Linked List

- Singly linked list can store data in non-contiguous locations. Thus there is no need of compaction of memory, when some large related data is to be stored into the memory.
- Insertion and deletion of values is easier as compared to array, as no shifting of values is involved.

4.5.3.2 Disadvantages of Singly Linked List

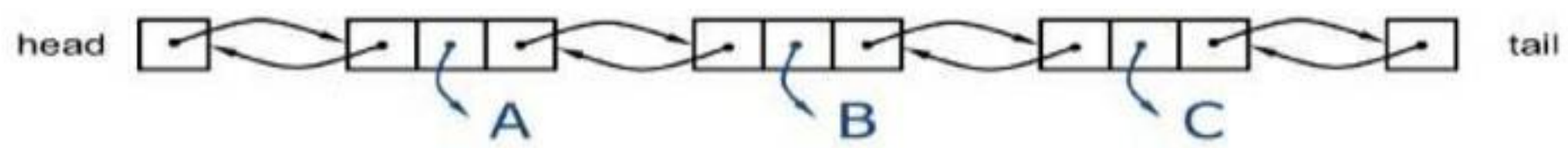
- Nodes can only be accessed sequentially. That means, we cannot jump to a particular node directly.
- Because of the above disadvantage, binary search algorithm cannot be implemented on the singly linked list.
- There is no way to go back from one node to previous one. Only forward traversal is possible.

nareshpad.com.np

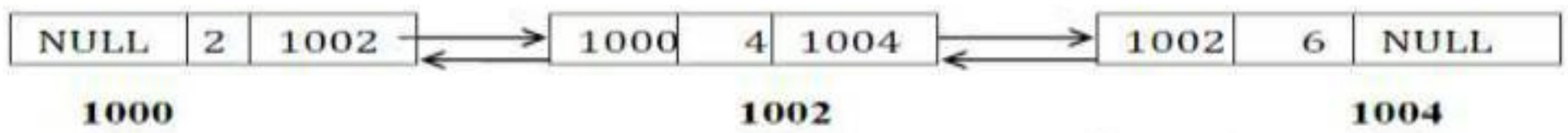
4.5.4 DOUBLY LINKED LIST

- A doubly-linked list is a linked data structure that consists of a set of data records, each having two special link fields that contain references to the previous and to the next record in the sequence. It can be viewed as two singly-linked lists formed from the same data items, in two opposite orders.
- A doubly-linked list whose nodes contain three fields: a data value, the link to the next node, and the link to the previous node.
- The two links allow walking along the list in either direction with equal ease. Compared to a singly-linked list, modifying a doubly-linked list usually requires changing more pointers, but is simpler because there is no need to keep track of the address of the previous node.
- In simpler terms, Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.
  - Usually two pointers are maintained to keep track of the list, head and tail.

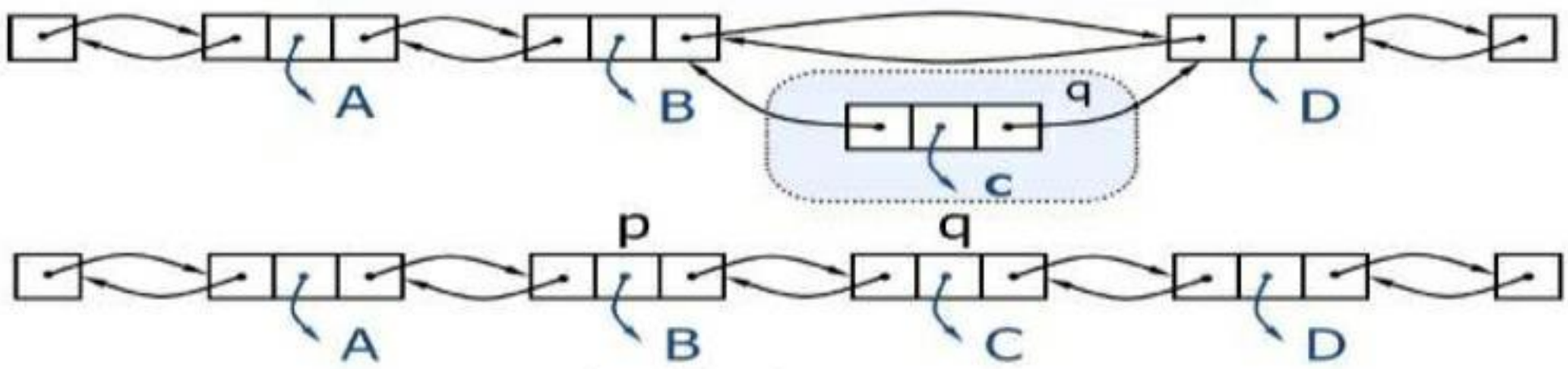
Representation of a doubly linked list



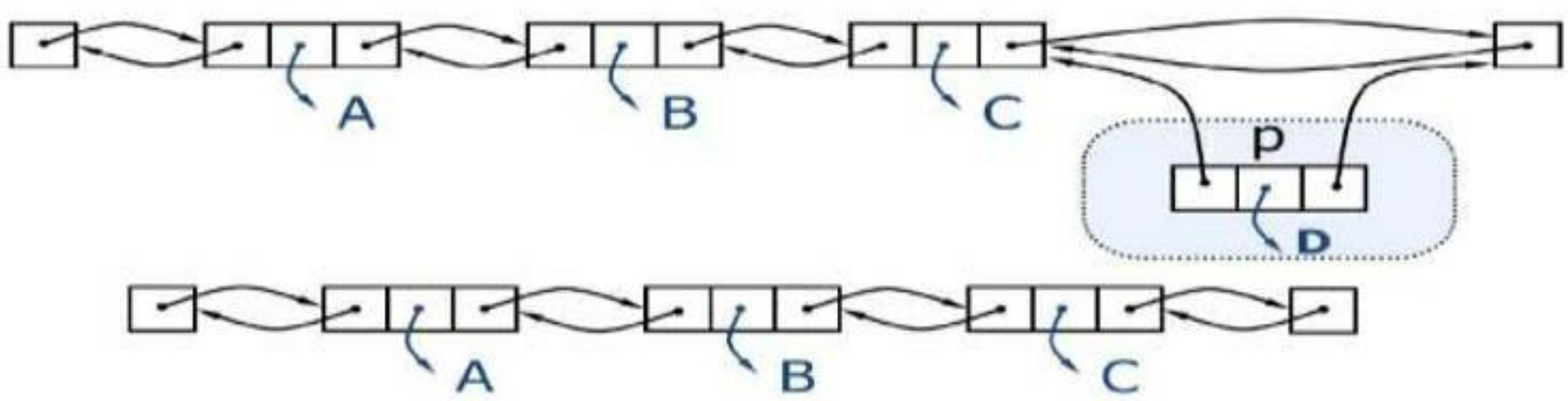
Example



Insertion of node in Doubly Linked List



Deletion of node in Doubly Linked List



- In C, the structure of a doubly linked list can be given as,

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
typedef struct node NodeType;
NodeType *head=NULL:
```

4.5.4.1 Inserting a New Node in a Doubly Linked List

Let * **head** be the pointer to first node in the current list, **PTR** is a pointer to hold the node address.

- **Algorithm to insert a node at the beginning of a doubly linked list :**
  1. Allocate memory for the new node as,
     newnode=(NodeType*)malloc(sizeof(NodeType))
  2. Assign value to data field of a new node
     set newnode->data=item
  3. set newnode->prev=newnode->next=NULL
  4. set newnode->next=head
  5. set head->prev=newnode
  6. set head=newnode
  7. End
- **Algorithm to insert a node at the end of a doubly linked list :**
  1. Allocate memory for the new node as,
     newnode=(NodeType*)malloc(sizeof(NodeType))
  2. Assign value to data field of a new node
     set newnode->data=item
  3. set newnode->next=NULL
  4. if head==NULL
     set newnode->prev=NULL;
     set head=newnode;
  5. if head!=NULL
     set PTR=head
     while(PTR->next!=NULL)
     PTR=PTR->next;
     end while
     set PTR->next=newnode;
     set newnode->prev=PTR
  6. End
- **Algorithm to insert a node at the specified position in a doubly linked list :**
  1. Input DATA and POS to be inserted
  2. Initialize PTR = head; and k = 0
  3. Repeat the step 3 while (k is less than POS)
     I. PTR = PTR->next
     II. If (PTR is equal to NULL)
        a) **Display "Node in the list less than the position"**
        b) Exit
     III. k = k + 1
  4. Create a new node from memory and assign its address to **NewNode**
     newnode =(NodeType *)malloc(sizeof(NodeType));
  5. Assign DATA to the **data** field of new node and Set **PTR → next** to the **next** of new node, **PTR** to the **prev** of newnode
     newnode ->data=newItem;
     newnode -> next = PTR->next
     newnode -> prev = PTR
  6. PTR → **next** = **NewNode**
  7. End

naresh.com.np

4.5.4.2 Deleting a Node from a Doubly Linked List

Let ***head** be the pointer to first node in the current list, **PTR** is a pointer to hold the node address

- **Algorithm to delete a node from beginning of a doubly linked list:**

```
1. if head==NULL then
            print "empty list" and exit
   else
            set PTR=head
            set head=head->next
            set head->prev=NULL;
            free(PTR)
2. End
```

- **Algorithm to delete a node from end of a doubly linked list:**

```
1. if head==NULL then
            print "empty list" and exit
   else if(head->next==NULL) then
            set PTR=head
            set head=NULL
            free(PTR)
   else
            set PTR=head;
            while(PTR->next!=NULL)
                    PTR=PTR->next
            end while
            set PTR->prev->next=NULL
            free(PTR)
2. End
```

- **Algorithm to delete the node at the specified position of the doubly linked list:**

1. if head==NULL then
   print "empty list" and exit
2. Enter position of a node at which you want to delete a new node. Let this position is POS.
3. Set PTR=head and k = 0
4. Repeat the step 4 while (k is less than POS)
   I. PTR = PTR->next
   II. If (PTR is equal to NULL)
      a) **Display "Node in the list less than the position"**
      b) Exit
   III. k = k + 1
5. Set PTR->prev->next = PTR->next
6. Set PTR->next->prev = PTR->prev
7. free PTR
8. Exit

- **Representation of Doubly Linked List in C**

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct Node
{
        int info;
        struct Node *next;
        struct Node *prev;
};
```

naresapd.com.np

```
typedef struct Node node;
node *head = NULL;
void create_linkedlist();
void display();
void insert_beg();
void insert_end();
void insert_position();
void delete_beg();
void delete_end();
void delete_position();
int main()
{
   int option,flag=1;
   printf("\n\n *****MAIN MENU *****");
         printf("\n 1: Create a list");
         printf("\n 2: Display the list");
         printf("\n 3: Add a node at the beginning");
         printf("\n 4: Add a node at the end");
         printf("\n 5: Add a node at nth position");
         printf("\n 7: Delete a node from the beginning");
         printf("\n 8: Delete a node from the end");
         printf("\n 9: Delete nth position node");
         printf("\n 10: EXIT");
         do
         {
                  printf("\n\n Enter your option : ");
                  scanf("%d", &option);
                  switch(option)
                  {
                           case 1:
                                    create_linkedlist();
                                    printf("\n LINKED LIST CREATED");
                                    break;
                           case 2:
                                    display();
                                    break;
                           case 3:
                                    insert_beg();
                                    break;
                           case 4:
                                    insert_end();
                                    break;
                           case 5:
                                    insert_position();
                                    break;
                           case 7:
                                    delete_beg();
                                    break;
                           case 8:
                                    delete_end();
                                    break;
                           case 9:
                                    delete_position();
                                    break;
```

nareshpad.com.np

```
                        case 10:
                                flag = 0;
                                break;
                        default:
                                printf("Invalide Choice");
                }
        }while(flag);
        getch();
        return 0;
}
void create_linkedlist()
{
        node *new_node, *ptr;
        int data,ch;
        do
        {
                printf("\n Enter the data : ");
                scanf("%d",&data);
                new_node = (node*)malloc(sizeof(node));
            new_node->info=data;
            new_node->next = NULL;
                if(head==NULL)
                {
                        new_node->prev = NULL;
                        head=new_node;
                }
                else
                {
                        ptr=head;
                        while(ptr->next!=NULL)
                                ptr=ptr->next;
                        ptr->next = new_node;
                        new_node->prev=ptr;
                }
        printf("\n Do you want to add more data then press 1:");
        scanf("%d",&ch);
        }while(ch == 1);

}
void display()
{
        node *ptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        ptr = head;
        printf("\nLinked List Data are:\n");
        while(ptr!=NULL)
        {
                printf("\t %d", ptr->info);
                ptr=ptr->next;
        }
```

naresh.com.np

```
}
void insert_beg()
{
        node *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node=(node *)malloc(sizeof(node));
        new_node->info=data;
        new_node->next=head;
        head->prev=new_node;
        new_node->prev=NULL;
        head=new_node;
}
void insert_end()
{
        node *ptr, *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node = (node *)malloc(sizeof(node));
        new_node->info = data;
        new_node->next = NULL;
        if(head == NULL)
        {
                head->prev = NULL;
                head = new_node;
        }
        else
        {
                ptr=head;
                while(ptr->next!=NULL)
                        ptr=ptr->next;
                ptr->next = new_node;
                new_node->prev=ptr;
        }
}
void insert_position()
{
        node *new_node,*ptr,*preptr;
        int data,pos,k;
        printf("\n Enter the data :");
        scanf("%d", &data);
        printf("\n Enter the index position at which the data has to be inserted : ");
        scanf("%d", &pos);
        ptr = head;
        k=0;
        while(k<pos)
        {
                ptr=ptr->next;
                if(ptr==NULL)
                {
                        printf("Node in the list less than the position");
                        return;
```

naresh.com.np

```
                }
                k++;
        }
        new_node = (node *)malloc(sizeof(node));
        new_node->info = data;
        new_node->next = ptr->next;
        new_node->prev = ptr;
        ptr->next = new_node;
}

void delete_beg()
{
        node *ptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        ptr = head;
        head = head -> next;
        head->prev=NULL;
        printf("\nDleted data from begining is:%d\n",ptr->info);
        free(ptr);
}
void delete_end()
{
        node *ptr,*preptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        else if(head->next == NULL)
        {
                ptr = head;
                head=NULL;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
        }
        else
        {
                ptr = head;
                while(ptr->next != NULL)
                {
                        ptr = ptr->next;
                }
                (ptr->prev)-> next = NULL;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
        }
}
void delete_position()
{
        node *ptr,*preptr;
```

naresh.com.np

```
int pos,k;
if(head == NULL)
{
        printf("\nList is empty\n");
        return;
}
printf("\n Enter the position of the node which has to be deleted : ");
scanf("%d",&pos);
ptr = head;
for(k=0;k<pos;k++)
{
        ptr = ptr->next;
        if(ptr==NULL)
        {
                printf("Node in the list is less than entered position");
                return;
        }
}
(ptr->prev)->next = ptr->next;
(ptr->next)->prev = ptr->prev;
printf("\n Deleted data at position %d is %d",pos,ptr->info);
free(ptr);
}
```

4.5.4.3 Application of Doubly Linked List

- Applications that have an MRU list (a linked list of file names)
- The cache in your browser that allows you to hit the BACK button (a linked list of URLs)
- Undo functionality in Photoshop or Word (a linked list of state)
- A stack, hash table, and binary tree can be implemented using a doubly linked list.
- A great way to represent a deck of cards in a game

4.5.4.4 Advantages of Doubly Linked List

- The primary advantage of a doubly linked list is that given a node in the list, one can navigate easily in either direction.
- This can be very useful, for example, if the list is storing strings, where the strings are lines in a text file (e.g., a text editor).
- One might store the ``current line'' that the user is on with a pointer to the appropriate node; if the user moves the cursor to the next or previous line, a single pointer operation can restore the current line to its proper value.
- Or, if the user moves back 10 lines, for example, one can perform 10 pointer operations (follow the chain) to get to the right line.
- For either of these operations, if the list is singly linked, one must start at the head of the list and traverse until the proper point is reached. This can be very inefficient for large lists.

4.5.4.5 Disadvantages of Doubly Linked List

- Each node requires an extra pointer, requiring more space
- The insertion or deletion of a node takes a bit longer (more pointer operations).

4.5.5 CIRCULAR SINGLY LINKED LIST

- Singly Linked List has a major drawback. From a specified node, it is not possible to reach any of the preceding nodes in the list. To overcome the drawback, a small change is made to the SLL so that the next field of the last node is pointing to the first node rather than NULL. Such a linked list is called a circular linked list.
    - Because it is a circular linked list, it is possible to reach any node in the list from a particular node.
    - There is no natural first node or last node because by virtue of the list is circular.
    - Therefore, one convention is to let the external pointer of the circular linked list, tail, point to the last node and to allow the following node to be the first node.
    - If the tail pointer refers to NULL, means the circular linked list is empty.

- We can declare the structure for the circular linked list in the same way as declared it for the linear linked list.

```
struct node
{
        int info;
        struct node *next;
};
typedef struct node NodeType;
NodeType *start=NULL:
NodeType *last=NULL:
```

- **Algorithm to insert a node at the beginning of a circular linked list:**

```
1. Create a new node as
        newnode=(NodeType*)malloc(sizeof(NodeType));
2. if start==NULL then
        set newnode->info=item
        set newnode->next=newnode
        set start=newnode
        set last = newnode
   else
        set newnode->info=item
        set newnode->next=start
        set start=newnode
        set last->next=newnode
3. Exit
```

- **Algorithm to insert a node at the end of a circular linked list:**

```
1. Create a new node as
        newnode=(NodeType*)malloc(sizeof(NodeType));
2. if start==NULL then
        set newnode->info=item
        set newnode->next=newnode
        set start=newnode
        set last = newnode
   else
        set newnode->info=item
        set last->next=newnode
        set last=newnode
        set last->next=start
3. Exit
```

- **Algorithm to delete a node from the beginning of a circular linked list:**

```
1. if start==NULL then
        "empty list" and exit
   else if start==last
        set PTR = start;
        print the deleted element=PTR->info
        free(PTR)
        set start = last = NULL;
   else
        set PTR=start
        set start=start->next
        set last->next=start;
        print the deleted element=PTR->info
        free(PTR)
2. Exit
```

naresh.com.np

Algorithm to delete a node from the end of a circular linked list:

1. if start==NULL then
   "**empty list" and exit**
2. if start==last
   set PTR=start
   print deleted element=PTR->info
   free(PTR)
   start=last=NULL
   else
   set temp=start
   while(temp->next!=last)
   set temp=temp->next
   end while
   set PTR=last
   set last=temp
   set last->next=start
   print the deleted element=PTR->info
   free(PTR)
3. Exit

Representation of Circular Singly Linked List in C

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
        int info;
        struct node *next;
};
typedef struct node Lnode;
Lnode *start = NULL;
Lnode *last = NULL;
void create_linkedlist();
void display();
void insert_beg();
void insert_end();
void delete_beg();
void delete_end();
int main()
{
  int option,flag=1;
  printf("\n\n *****MAIN MENU *****");
  printf("\n 1: Create a list");
  printf("\n 2: Display the list");
  printf("\n 3: Add a node at the beginning");
  printf("\n 4: Add a node at the end");
  printf("\n 5: Delete a node from the beginning");
  printf("\n 6: Delete a node from the end");
  printf("\n 7: EXIT");
  do
  {
        printf("\n\n Enter your option : ");
        scanf("%d", &option);
        switch(option)
```

naresh pad.com.np

```
{
        case 1:
                create_linkedlist();
                printf("\n LINKED LIST CREATED");
                break;
        case 2:
                display();
                break;
        case 3:
                insert_beg();
                break;
        case 4:
                insert_end();
                break;
        case 5:
                delete_beg();
                break;
        case 6:
                delete_end();
                break;
        case 7:
                flag = 0;
                break;
        default:
                printf("Invalide Choice");
}
}while(flag);
getch();
return 0;
}
void create_linkedlist()
{
        Lnode *new_node, *ptr;
        int data,ch;
        do
        {
                printf("\n Enter the data : ");
                scanf("%d",&data);
                new_node = (Lnode*)malloc(sizeof(Lnode));
                new_node->info=data;
                if(start==NULL)
                {
                        new_node->next = new_node;
                        start=new_node;
                        last=new_node;
                }
                else
                {
                        last->next = new_node;
                        last = new_node;
                        last->next=start;
                }
        printf("\n Do you want to add more data then press 1:");
        scanf("%d",&ch);
```

nareshpad.com.np

```
        }while(ch == 1);

}
void display()
{
        Lnode *ptr;
        if(start == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        ptr = start;
        printf("\nLinked List Data are:\n");
        while(ptr!=last)
        {
                printf("\t %d", ptr->info);
                ptr=ptr->next;
        }
        printf("\t %d", ptr->info);
}
void insert_beg()
{
        Lnode *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node=(Lnode *)malloc(sizeof(Lnode));
        new_node->info=data;
        if(start==NULL)
        {
                new_node->next=new_node;
                start=new_node;
                last=new_node;
        }
        else
        {
                new_node->next=start;
                start = new_node;
                last->next=new_node;
        }

}
void insert_end()
{
        Lnode *ptr, *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node = (Lnode *)malloc(sizeof(Lnode));
        new_node->info = data;
        if(start==NULL)
        {
                new_node->next=new_node;
                start=new_node;
```

naresh.com.np

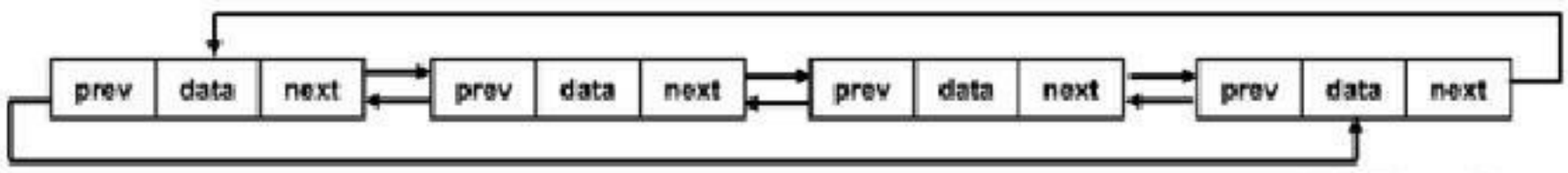
```
                last=new_node;
        }
        else
        {
                last->next = new_node;
                last = new_node;
                last->next=start;
        }
}
void delete_beg()
{
        Lnode *ptr;
        if(start == NULL)
        {
                printf("\nList is empty\n");
        }
        else if(start==last)
        {
                ptr = start;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
                start = last = NULL;
        }
        else
        {
                ptr = start;
                start = start -> next;
                last->next = start;
                printf("\nDleted data from begining is:%d\n",ptr->info);
                free(ptr);
        }
}
void delete_end()
{
        Lnode *ptr,*temp;
        if(start == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        else if(start==last)
        {
                ptr = start;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
                start = last = NULL;
        }
        else
        {
                temp = start;
                while(temp -> next != last)
                {
                        temp = temp -> next;
                }
```

nareshpad.com.np

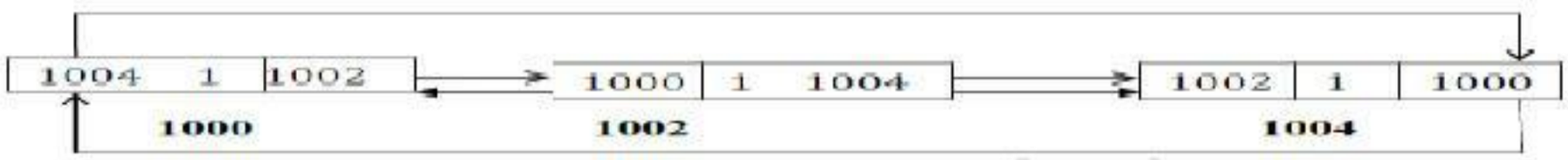
```
                ptr=last;
                last=temp;
                last->next=start;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
            }
        }
```

4.5.6 Circular Doubly Linked List

- A Circular Doubly Linked List (CDL) is a doubly linked list with first node linked to last node and vice-versa.
- **The 'prev' link of first node contains the address of last node and 'next'** link of last node contains the address of first node.
- Traversal through Circular Singly Linked List is possible only in one direction.
- The main advantage of Circular Doubly Linked List (CDL) is that, a node can be inserted into list without searching the complete list for finding the address of previous node.
- We can also traversed through CDL in both directions, from first node to last node and vice-versa.



Example:

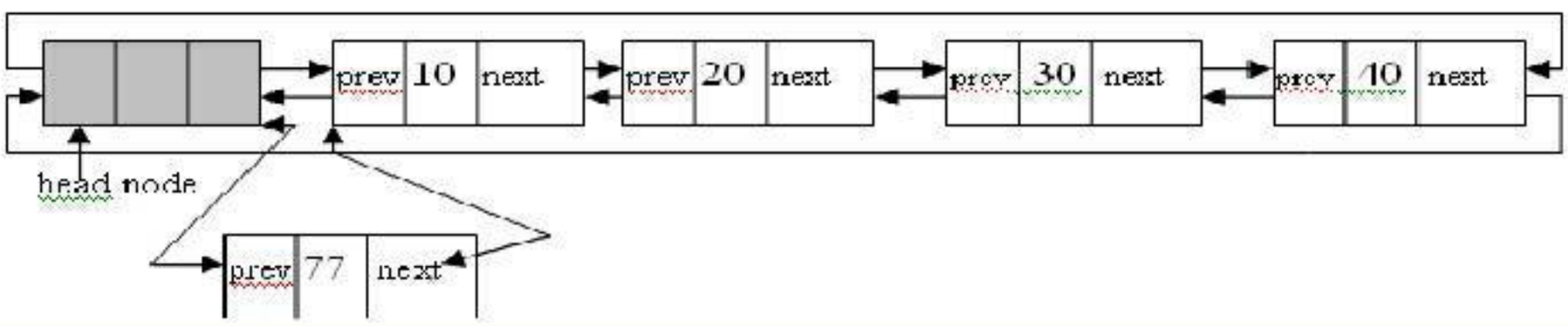


- We can declare the structure for the circular doubly linked list in the same way as declared it for the double linked list

```
struct node
{
  int info;
  struct node *prev;
  struct node *next;
};
typedef struct node NodeType;
NodeType *head=NULL:
```

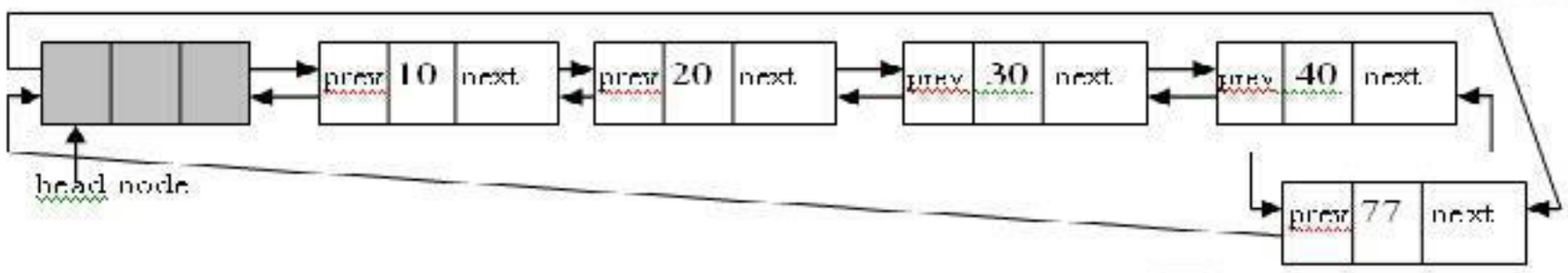
<u>Algorithm to insert a node at the beginning of a circular doubly linked list</u>

1. Allocate memory for the new node as,
   newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node
   set newnode->info=item
3. If head == NULL
   head = newnode
   head->next = head->prev = newnode
   else
   set newnode->next=head
   set newnode->prev=head->prev
   set (head->prev)->next = newnode
   set head = newnode
4. Exit



Algorithm to insert a node at the end of a circular doubly linked list :

1. Allocate memory for the new node as,
   newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node
   set newnode->info=item
5. If head == NULL
   head = newnode
   head->next = head->prev = newnode
   else
   set (head->prev)->next=newnode
   set newnode->prev=(head->prev)
   set newnode->next=head
   set head->prev=newnode
3. Exit



Algorithm to delete a node from the beginning of a circular doubly linked list

1. if head==NULL then
   print "empty list" and exit
   else if(head->next==head)
   PTR = head;
   head=NULL;
   print "Deleted data from end= PTR ->info";
   free(PTR);
   else
   set PTR=head;
   set (PTR->next)->prev = (PTR->prev)
   set (PTR->prev)->next = PTR->next
   set head = PTR->next
   free(PTR)
2. Exit

Algorithm to delete a node from the end of a circular doubly linked list:

1. if head ==NULL then
   print "empty list" and exit
   else if(head->next==head)
   PTR = head;
   head=NULL;
   print "Deleted data from end= PTR ->info";
   free(PTR);
   else
   set PTR=head->prev
   set (PTR->prev)->next = head
   set head->prev = PTR->prev
   free(PTR)
2. Exit

nareshpdas.com

Representation of circular doubly linked list in C

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct Node
{
        int info;
        struct Node *next;
        struct Node *prev;
};
typedef struct Node node;
node *head = NULL;
void create_linkedlist();
void display();
void insert_beg();
void insert_end();
void delete_beg();
void delete_end();
int main()
{
   int option,flag=1;
   printf("\n\n *****MAIN MENU *****");
        printf("\n 1: Create a list");
        printf("\n 2: Display the list");
        printf("\n 3: Add a node at the beginning");
        printf("\n 4: Add a node at the end");
        printf("\n 5: Delete a node from the beginning");
        printf("\n 6: Delete a node from the end");
        printf("\n 7: EXIT");
        do
        {
                printf("\n\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                        case 1:
                                create_linkedlist();
                                printf("\n LINKED LIST CREATED");
                                break;
                        case 2:
                                display();
                                break;
                        case 3:
                                insert_beg();
                                break;
                        case 4:
                                insert_end();
                                break;
                        case 5:
                                delete_beg();
                                break;
                        case 6:
                                delete_end();
                                break;
```

nareshpd.com.np

```
                        case 7:
                                flag = 0;
                                break;
                        default:
                                printf("Invalide Choice");
                }
        }while(flag);
        getch();
        return 0;
}
void create_linkedlist()
{
        node *new_node, *ptr;
        int data,ch;
        do
        {
                printf("\n Enter the data : ");
                scanf("%d",&data);
                new_node = (node*)malloc(sizeof(node));
                new_node->info=data;
                if(head==NULL)
                {
                        head=new_node;
                        head->next=head->prev = new_node;
                }
                else
                {
                        (head->prev )->next=new_node;
                        new_node->prev=(head->prev);
                        new_node->next=head;
                        head->prev=new_node;
                }
        printf("\n Do you want to add more data then press 1:");
        scanf("%d",&ch);
        }while(ch == 1);

}
void display()
{
        node *ptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        ptr = head;
        printf("\nLinked List Data are:\n");
        while(ptr!=head->prev)
        {
                printf("\t %d", ptr->info);
                ptr=ptr->next;
        }
        printf("\t %d", ptr->info);
}
```

naareshpad.com.np

```
void insert_beg()
{
        node *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node=(node *)malloc(sizeof(node));
        new_node->info=data;
                if(head==NULL)
                {
                        head=new_node;
                        head->next=head->prev = new_node;
                }

                else
                {
                        (head->prev )->next=new_node;
                        new_node->prev=(head->prev);
                        new_node->next=head;
                        head->prev=new_node;
                }
}
void insert_end()
{
        node *ptr, *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node = (node *)malloc(sizeof(node));
        new_node->info = data;
                if(head==NULL)
                {
                        head=new_node;
                        head->next=head->prev = new_node;
                }
                else
                {
                        (head->prev )->next=new_node;
                        new_node->prev=(head->prev);
                        new_node->next=head;
                        head->prev=new_node;
                }
}
void delete_beg()
{
        node *ptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
        }
        else if(head->next==head)
        {
                ptr = head;
                head=NULL;
```

naresh.com.np

```
            printf("\nDleted data from end is:%d\n",ptr->info);
            free(ptr);
        }
        else
        {
            ptr = head;
            (ptr->next)->prev = (ptr->prev);
            (ptr->prev)->next = ptr->next;
            head = ptr->next;
            printf("\nDleted data from begining is:%d\n",ptr->info);
            free(ptr);
        }
    }

    void delete_end()
    {
        node *ptr,*preptr;
        if(head == NULL)
        {
            printf("\nList is empty\n");
        }
        else if(head->next==head)
        {
            ptr = head;
            head=NULL;
            printf("\nDeleted data from end is:%d\n",ptr->info);
            free(ptr);
        }
        else
        {
            ptr=head->prev;
            (ptr->prev)->next = head;
            head->prev = ptr->prev;
            printf("\nDleted data from end is:%d\n",ptr->info);
            free(ptr);
        }
    }
```

4.5.6.1 Advantages of Circular Linked Lists:

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- **Useful for implementation of queue. Unlike this implementation, we don't need to maintain** two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

nareshpad.com.np

4.5.7 Polynomials using singly linked list (Application of Linked List)

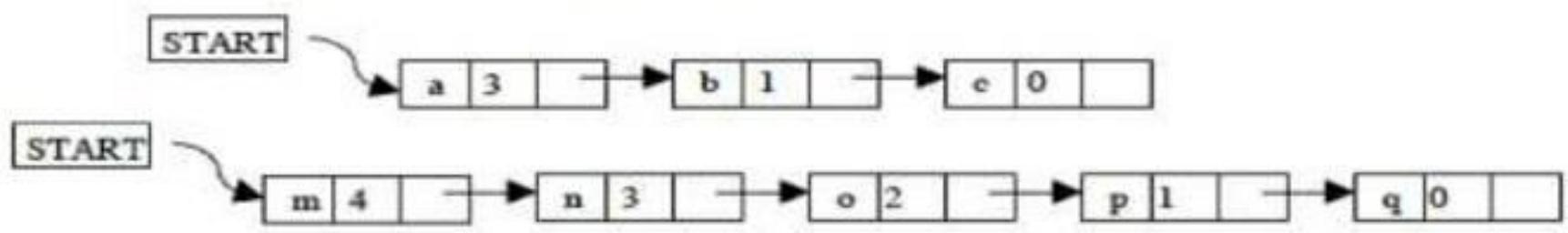
Different operations, such as addition, subtraction, division and multiplication of polynomials can be performed using linked list. Following example shows that the polynomial addition using linked list.

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.

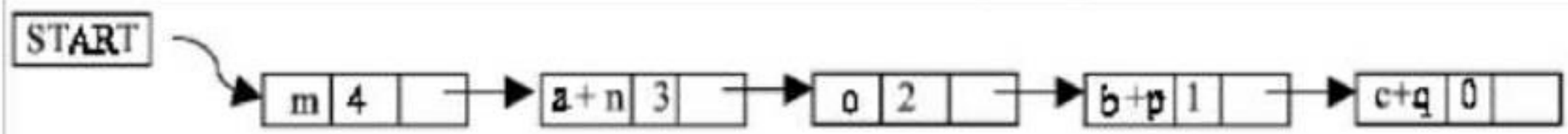
$$f(x) = ax^3 + bx + c$$
$$g(x) = mx^4 + nx^3 + ox^2 + px + q$$



two polynomials can be added by

$$h(x) = f(x) + g(x) = mx4 + (a + n)\ x3 + ox2 + (b + p)x + (c + q)$$

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



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.

- **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')

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))

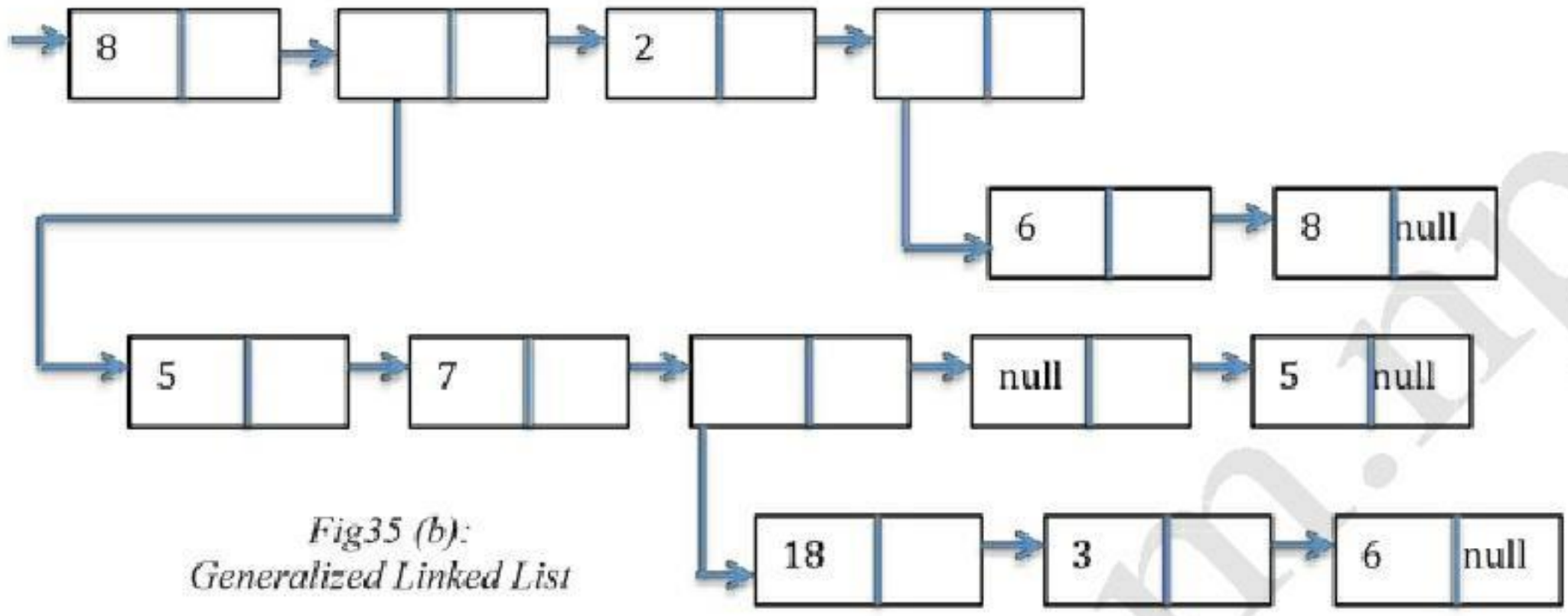


Fig35 (b): Generalized Linked List

nareshpad.com.np

5 Recursion

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are
  - **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - **Recursive case**, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts
- Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.
- To understand recursive functions, let us take an example of calculating factorial of a number. To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.

  In other words, n! = n * (n–1)!

PROBLEM	SOLUTION
5!	5 × 4 × 3 × 2 × 1!
= 5 × 4!	= 5 × 4 × 3 × 2 × 1
= 5 × 4 × 3!	= 5 × 4 × 3 × 2
= 5 × 4 × 3 × 2!	= 5 × 4 × 6
= 5 × 4 × 3 × 2 × 1!	= 5 × 24
	= 120

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. Every recursive function must have a base case and a recursive case. For the factorial function

Base case is when n = 1, because if n = 1, the result will be 1 as

1! = 1.

Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as

factorial(n) = n × factorial (n–1)

Program to calculate the factorial of a given number

```
#include <stdio.h>
#include<conio.h>
long int Fact(int);                    // FUNCTION DECLARATION
int main()
{
        int num;
        long int val;
        printf("\n Enter the number: ");
        scanf("%d", &num);
        val = Fact(num);
        printf("\n Factorial of %d = %ld", num, val);
        getch();
        return 0;
}
long int Fact(int n)
{
        if(n==1)
                return 1;
        else
                return (n * Fact(n–1));
}
```

- **Steps of a recursive program**

Step 1: Specify the base case which will stop the function from making a call to itself.
Step 2: Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.
Step 3: Divide the problem into smaller or simpler sub-problems.
Step 4: Call the function from each sub-problem.
Step 5: Combine the results of the sub-problems.
Step 6: Return the result of the entire problem.

- **The advantages of using a recursive program include the following:**
  - Recursive solutions often tend to be shorter and simpler than non-recursive ones.
  - Code is clearer and easier to use.
  - Recursion works similar to the original formula to solve a problem.
  - Recursion follows a divide and conquer technique to solve problems.
  - In some (limited) instances, recursion may be more efficient.
- **The drawbacks/disadvantages of using a recursive program include the following:**
  - For some programmers and readers, recursion is a difficult concept.
  - Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
  - Aborting a recursive program in midstream can be a very slow process.
  - Using a recursive function takes more memory and time to execute as compared to its no recursive counterpart.
  - It is difficult to find bugs, particularly while using global variables.

The advantages of recursion pay off for the extra overhead involved in terms of time and space required.

5.1 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:

- It requires stack implementation.
- 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.
- 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 set of statements repeatedly, until some specific condition is specified.
There must be an exclusive if statement inside the recursive function, specifying stopping condition.	Iteration involves four clear cut steps initialization, condition, execution and updation.
Not all problems have recursive solution.	Any recursive problem can be solved iteratively.
Recursion is generally a worse option to go for simple problem or problems not recursive in nature.	Iterative counterpart of a problem is more efficient in terms of memory utilization and execution speed.

naresh.com.np

5.2 Types of Recursion

Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem. Any recursive function can be characterized based on:

- whether the function calls itself directly or indirectly (direct or indirect recursion),
- whether any operation is pending at each recursive call (tail recursive or not), and
- the structure of the calling pattern (linear or tree-recursive).

```
int Func (int n)
{
    if (n == 0)
        return n;
    else
        return (Func (n-1));
}
```

Direct Recursion

```
int Func1 (int n)
{
    if (n == 0)
        return n;
    else
        return Func2(n);
}
int Func2(int x)
{
    return Func1(x-1);
}
```

Indirect Recursion

```
int Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return (n * Fact(n-1));
}
```

Non-tail recursion

```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
      return Fact1(n-1, n*res);
}
```

Tail recursion

- In simple words, a recursive function is said to be linearly recursive when the pending operation (if any) does not make another recursive call to the function. For example, observe the last line of recursive factorial function. The factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another recursive call to Fact. On the contrary, a recursive function is said to be tree recursive (or non-linearly recursive) if the pending operation makes another recursive call to the function. For example, the Fibonacci function in which the pending operations recursively call the Fibonacci function.

```
int Fibonacci(int num)
{
  if(num == 0)
    return 0;
  else if (num == 1)
    return 1;
    else
    return (Fibonacci(num - 1) + Fibonacci(num - 2));
}
```

Observe the series of function calls. When the function returns, the pending operations in turn calls the function

Fibonacci(7) = Fibonacci(6) + Fibonacci(5)
Fibonacci(6) = Fibonacci(5) + Fibonacci(4)
Fibonacci(5) = Fibonacci(4) + Fibonacci(3)
Fibonacci(4) = Fibonacci(3) + Fibonacci(2)
Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
Fibonacci(2) = Fibonacci(1) + Fibonacci(0)

Now we have, Fibonacci(2) = 1 + 0 = 1

Fibonacci(3) = 1 + 1 = 2
Fibonacci(4) = 2 + 1 = 3
Fibonacci(5) = 3 + 2 = 5
Fibonacci(6) = 3 + 5 = 8
Fibonacci(7) = 5 + 8 = 13

Tree recursion

5.3 The Fibonacci Series

The Fibonacci series can be given as

0 1 1 2 3 5 8 13 21 34 55 ……

That is, the third term of the series is the sum of the first and second terms. Similarly, fourth term is the sum of second and third terms, and so on. Now we will design a recursive solution to find the nth term of the Fibonacci series. The general formula to do so can be given as As per the formula, FIB(0) =0 and FIB(1) = 1. So we have two base cases. This is necessary because every problem is divided into two smaller problems.

$$FIB(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ FIB(n-1) + FIB(n-2), & \text{otherwise} \end{cases}$$

5.4 Tower of Hanoi

- **The tower of Hanoi is one of the main applications of recursion. It says, 'if you can solve n−1 cases, then you can easily solve the nth case'.**

Initial state:

- There are three poles named as origin, intermediate and destination.
- n number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.
- The **disks are numbered as 1, 2, 3, 4, ……,n.**

Objective:

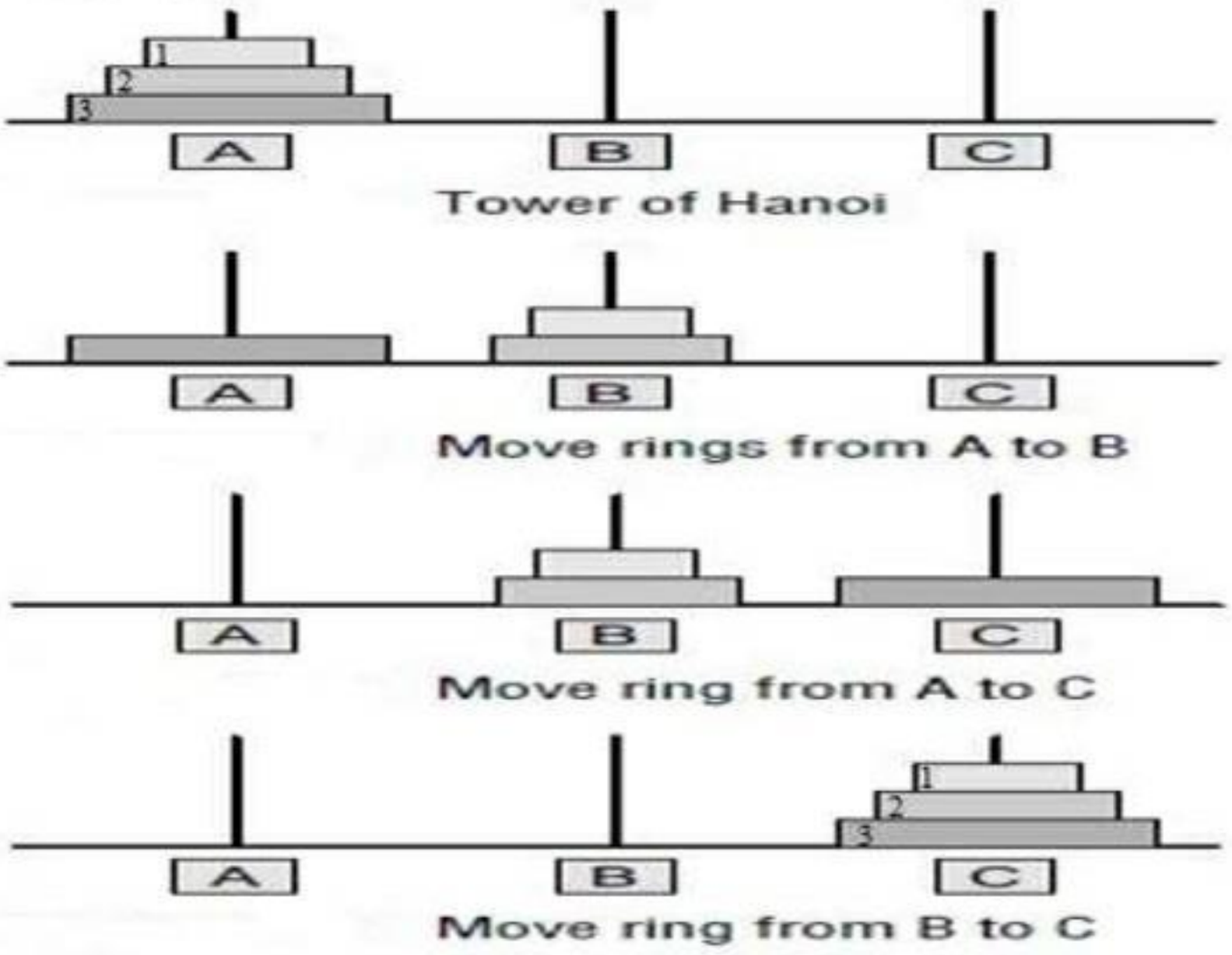
- Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

Conditions:

- Move only one disk at a time.
- Each disk must always be placed around one of the pole.
- Never place larger disk on top of smaller disk.

Algorithm: - To move a tower of n disks from source to destination (where n is positive integer):

1. If n ==1
   Move a single disk from source to dest.
2. If n > 1
   I. Let temp be the remaining pole other than source and dest.
   II. Move a tower of (n – 1) disks form source to temp.
   III. Move a single disk from source to dest.
   IV. Move a tower of (n – 1) disks form temp to dest.
3. Terminate.



naresh.com.np

Recursive solution of tower of Hanoi:

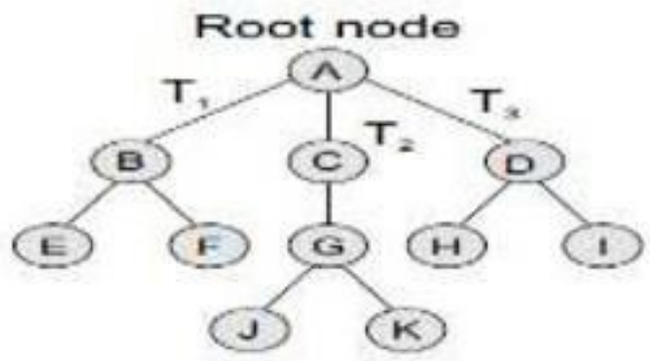
```
#include<stdio.h>
#include<conio.h>
int count =0;
// move n discs from Source (A) to Destination (C) using B as temporary
void TOH(int n,char source,char temp,char destination)
{
        if(n==1)
        {
                printf("Move Disc 1 from %c to %c\n",source,destination);
                count++;
                return;
        }
        //move n-1 discs from A to B using C as temporary
                TOH(n-1,source,destination,temp);
                printf("Move Disc %d from %c to %c\n",n,source,destination);
                count++;
        // move n-1 discs from B to C using A as temporary
                TOH(n-1,temp,source,destination);
}
int main()
{
        int n;
        printf("Enter the number of discs:");
        scanf("%d",&n);
        TOH(n,'A','B','C');
        printf("Total number of disc moves = %d\n",count);
        getch();
        return 0;
}
```

```
Enter the number of discs:3
Move Disc 1 from A to C
Move Disc 2 from A to B
Move Disc 1 from C to B
Move Disc 3 from A to C
Move Disc 1 from B to A
Move Disc 2 from B to C
Move Disc 1 from A to C
Total number of disc moves = 7
```

naresh.com.np

6 Trees

- Tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.
  - Tree is a sequence of nodes.
  - There is a starting node known as root node.
  - Every node other than the root has a parent node.
  - Nodes may have any number of children.
- Figure shows a tree where node A is the root node; nodes B, C, and D are children of the root node and form sub-trees of the tree rooted at node A.



- A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.
- **Definition** (recursively): A tree is a finite set of one or more nodes such that
  - There is a specially designated node called root.
  - **The remaining nodes are partitioned into n>=0 disjoint set T1,…,Tn, where each of these sets is a tree. T1,…,Tn are called the subtrees of the root.**

Characteristics of trees:

- Non-linear data structure
- Combines advantages of an ordered array
- searching as fast as in ordered array
- insertion and deletion as fast as in linked list

Application:

- Manipulate hierarchical data.
- Make information easy to search.
- Manipulate sorted lists of data.
- As a workflow for compositing digital images for visual effects.
- Router algorithms
- Form of a multi-stage decision-making (see business chess).

6.1 Basic Terminology

Root node: The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

Sub-trees: If the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R.

Leaf node: A node that has no children is called the leaf node or the terminal node.

Path: A sequence of consecutive edges is called a path. For example, in above figure the path from the root node A to node I is given as: A, D, and I. There is a single unique path from the root to any node.

Ancestor node: An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in above figure nodes A, C, and G are the ancestors of node K.

Descendant node: A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in above figure nodes C, G, J, and K are the descendants of node A.

Level number: Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes **have a level number given by parent's level number + 1.**

Degree of node: Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.

Degree of a Tree: The degree of a tree is the maximum degree of nodes in a given tree.

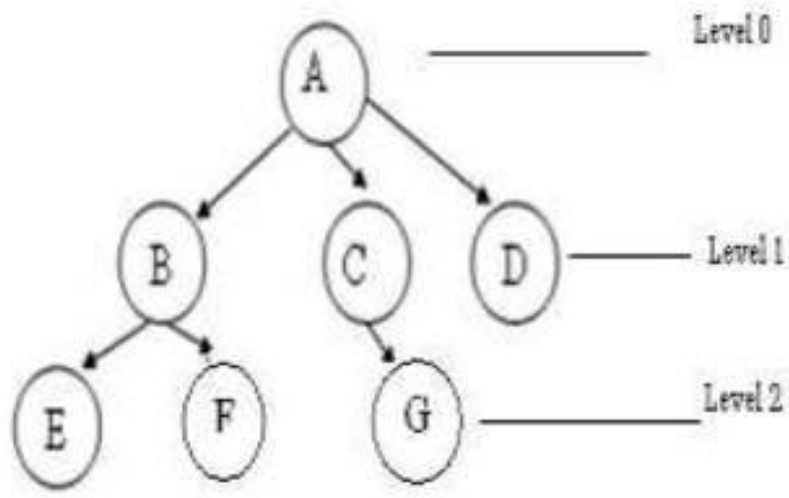
Height of a node: The height of a node is the maximum path length from that node to a leaf node. A leaf_node has a height of 0.

naresh.com.np

Height of a tree: The height of a tree is the height of the root.

Depth of a node: Depth of a node is the path length from the root to that node. The root node has a depth of 0.

Depth of a tree: Depth of a tree is the maximum level of any leaf in the tree. This is equal to the longest path from the root to any leaf.



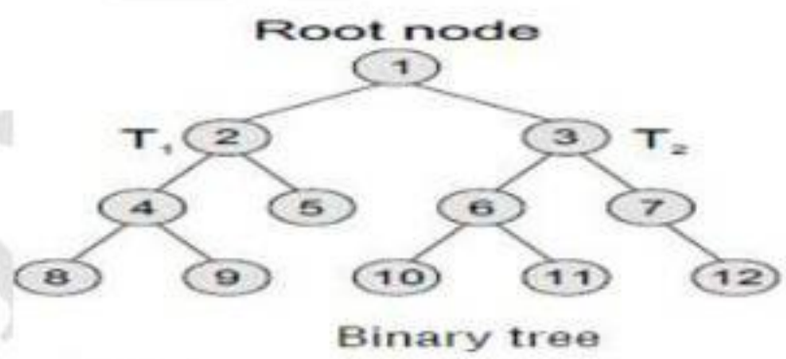
- ✓ A is the root node
- ✓ B is the parent of E and F
- ✓ D is the sibling of B and C
- ✓ E and F are children of B
- ✓ E, F, G, D are external nodes or leaves
- ✓ A, B, C are internal nodes
- ✓ Depth of F is 2
- ✓ the height of tree is 2
- ✓ the degree of node A is 3
- ✓ The degree of tree is 3

6.2 Binary Trees

- The simplest form of tree is a binary tree. A binary tree consists of
  - a node (called the root node) and
  - left and right sub-trees. Both the sub-trees are themselves binary trees.

  A binary tree is a hierarchical (tree) data structure in which each node has at most two children. Typically the child nodes are called left and right. One common use of binary trees is binary search trees; another is binary heaps.
- **Definition:**

  A binary tree is a finite set of elements that is either empty or it consists of a node called root and two other disjoint subsets. And these two subsets are themselves binary trees called left sub tree and the right sub tree. Each element of a binary tree is called a node of the tree.



- Figure shows a binary tree. In the figure, R is the root node and the two trees T1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R.
- Note that the left sub-tree of the root node consists of the nodes: 2, 4, 5, 8, and 9. similarly, the right sub-tree of the root node consists of nodes: 3, 6, 7, 10, 11, and 12.

6.3 Types of binary tree

- Strictly binary tree
- Complete binary tree
- Almost complete binary tree

6.3.1 Strictly binary tree

- A tree is said to be strictly binary tree if every non-leaf node has non empty left and right sub trees.

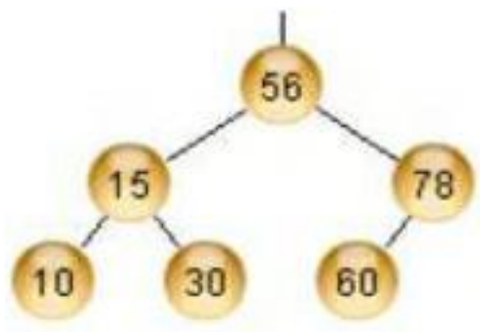


Figure 1

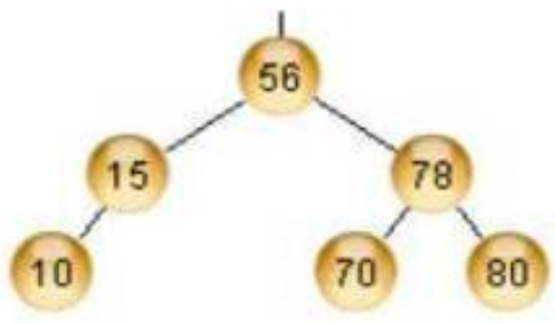


Figure 2

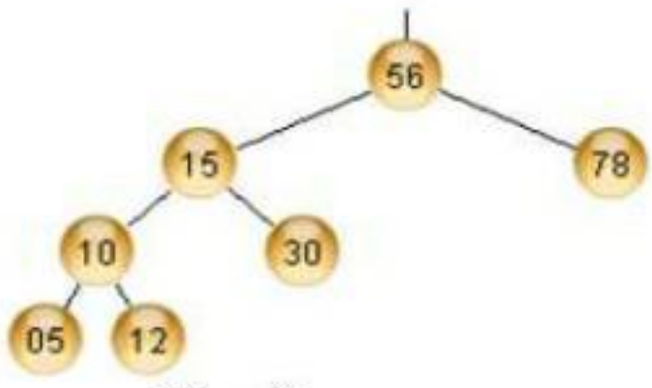


Figure 3

In the above diagram, **figure 1** is not a strictly binary tree because node **78** does not contain a right sub tree though it contains left sub tree. Similarly, **figure 2** is not a strictly binary tree because node **15** does not have right sub tree. But **figure 3** is a strictly binary tree because each non leaf node has right and left sub trees.

6.3.2 Complete binary tree

- A complete binary tree of depth d is the strictly binary tree whose leaves are at level d.

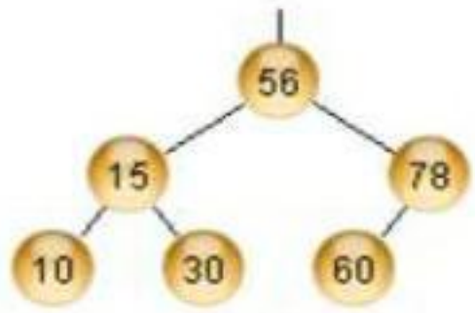


Figure 1

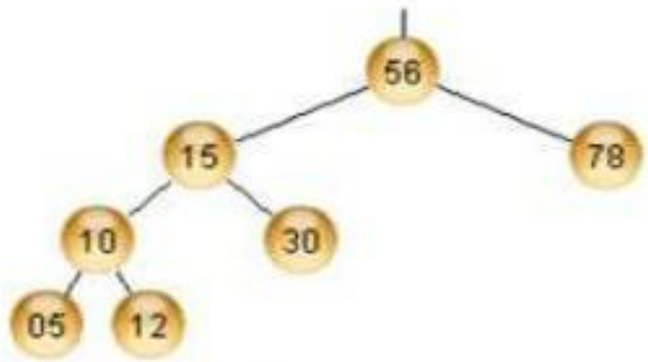


Figure 2

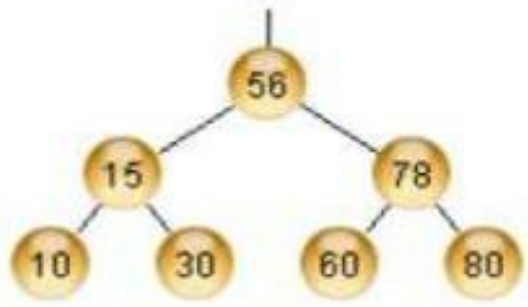


Figure 3

In the above diagram, **figure 1** is not a complete binary tree because it is not a strictly binary tree. **Figure 2** is a strictly binary tree but still it is not a complete binary tree because every leaf is not at the same level. Leaf nodes 78 and 30 from **figure 2** are not at the depth level of a binary tree. **Figure 3** is a complete binary tree because it is strictly binary **tree and every leaf node is at level d. So, "every complete binary tree is a strictly binary tree but every strictly binary tree is not a complete binary tree". If d is the depth of a complete binary** tree, what is the total number of nodes there?

6.3.3 Almost complete binary tree

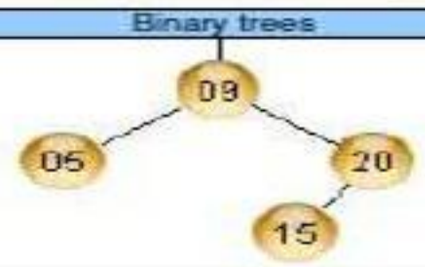
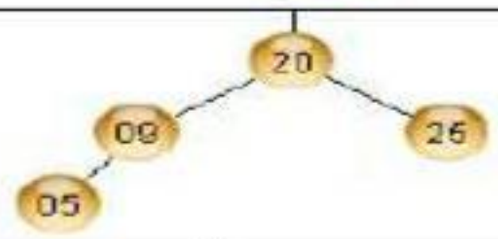
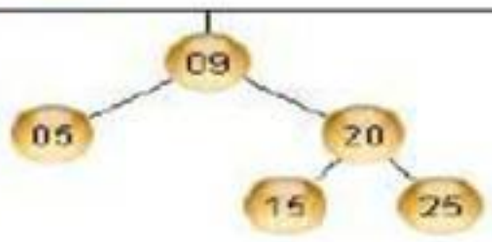
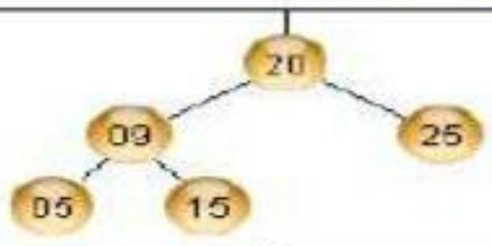
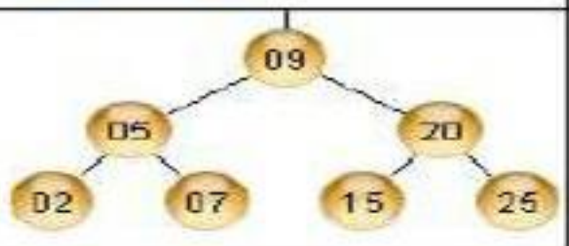
- A binary tree of depth d is an almost complete binary tree if:
  - The tree is complete Binary tree (All nodes) till level d-1.
  - At level d (i.e. the last level) if a node is present, then all the nodes to the left of that node should also be present.

OR

- For any node n in the tree with a right descendent at level d, n must have a left son and every left descendent of n is either a leaf at level d or has two sons.
- Any node n at level less than d-1 has two sons.

Explanation of point 1:

If any non-leaf node has right descendent that it must have left descendent. If right descendent goes to level d (depth) that each left descendent must have two sons or is a leaf at level d.

Binary trees	Strictly Binary Tree	Complete Binary Tree	Almost Complete Binary Tree
	No	No	No
	No	No	Yes
	Yes	No	No
	Yes	No	Yes
	Yes	Yes	Yes

Note:

If a binary tree has only left sub trees, then it is called left skewed binary tree. Figure 1 is a left skewed binary tree. If a binary tree has only right sub trees, then it is called right skewed binary tree. Figure 2 is a right skewed binary tree.

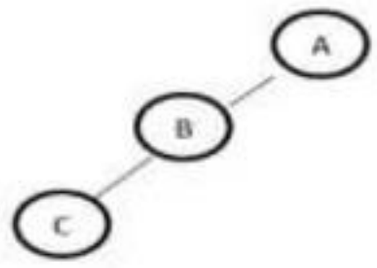


Figure 1

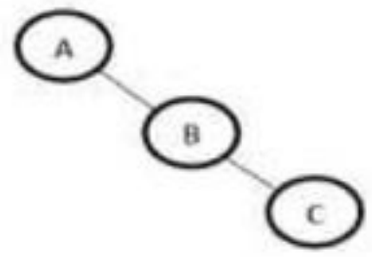


Figure 2

6.4 BINARY TREE REPRESENTATION

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

6.4.1 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}-1 nodes can be there in T. (i.e., SIZE = 2^{d+1}– 1).Consider a binary tree in Figure 1 of depth 2. Then SIZE = $2^{2+1} - 1 = 7$.

Then the array A [7] is declared to hold the nodes.

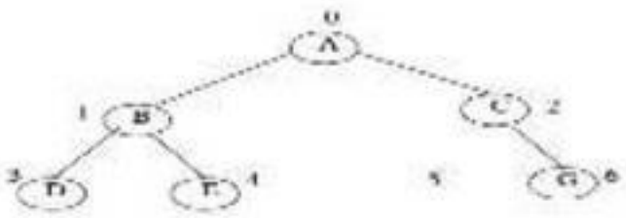


Figure 1: Binary Tree of Depth 2

A[ ]	A	B	C	D	E		F
	[0]	[1]	[2]	[3]	[4]	[5]	[6]

Figure 2: Array Representation of Binary Tree of Figure 1

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).

nareshpd.com.np

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.

- Sequential representation of trees is done using single or one-dimensional arrays. Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space. A sequential binary tree follows the following rules:
  - A one-dimensional array, called TREE, is used to store the elements of tree.
  - The root of the tree will be stored in the first location. That is, TREE[0] will store the data of the root element.
  - The children of a node stored in location K will be stored in locations (2 × K+1) and (2 × K+2).
  - The maximum size of the array TREE is given as ($2^{h+1}-1$), where h is the height of the tree.
  - An empty tree or sub-tree is specified using NULL. If TREE[0] = NULL, then the tree is empty.

6.4.2 LINKED LIST REPRESENTATION

- In the linked representation of a binary tree, every node will have three parts: the data element, a pointer to the left node, and a pointer to the right node.
- So in C, the binary tree is built with a node type given below.

```
struct bnode {
        struct bnode *left;
        int data;
        struct bnode *right;
};
```

- Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree.
- If ROOT = NULL, then the tree is empty. Consider the binary tree given in Figure 1. The schematic diagram of the linked representation of the binary tree is shown in Figure 2.
- In Figure 2, the left position is used to point to the left child of the node or to store the address of the left child of the node. The middle position is used to store the data. Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node. Empty sub-trees are represented using X (meaning NULL).

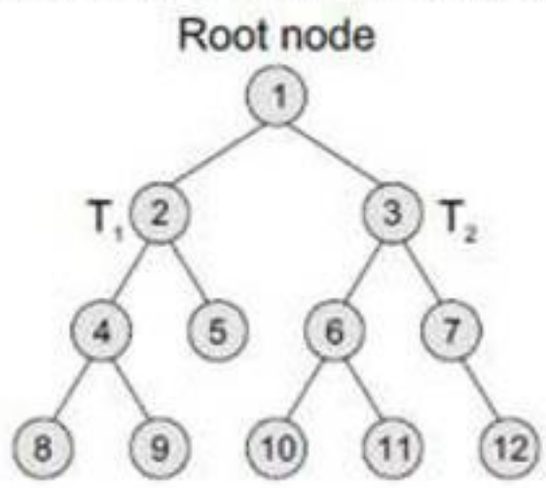


Figure 1: Binary Tree

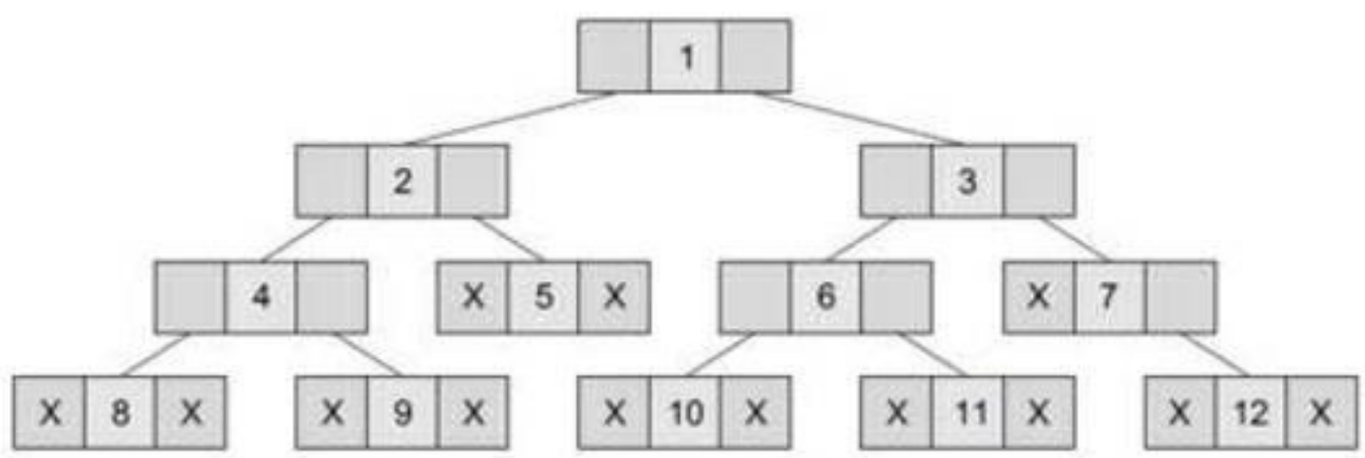


Figure 2: Linked representation of a binary tree

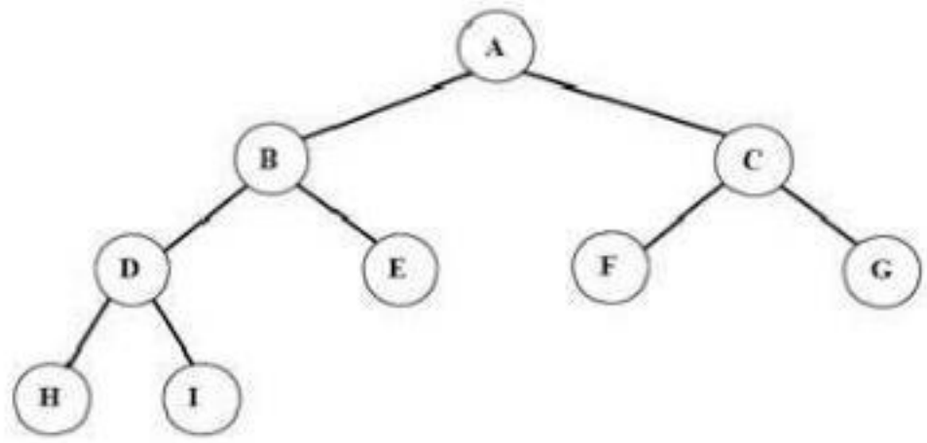
6.5 Operations on Binary tree:

- ✓ **father(n,T):** Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- ✓ **LeftChild(n,T):** Return the left child of node n in tree T. Return NULL if n does not have a left child.
- ✓ **RightChild(n,T):** Return the right child of node n in tree T. Return NULL if n does not have a right child.
- ✓ **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- ✓ **Sibling(n,T):** return the sibling node of node n in tree T. Return NULL if n has no sibling.
- ✓ **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- ✓ **Size(T):** Return the number of nodes in tree T
- ✓ **MakeEmpty(T):** Create an empty tree T
- ✓ **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- ✓ **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- ✓ **Preorder(T):** Traverses all the nodes of tree T in preorder.
- ✓ **postorder(T):** Traverses all the nodes of tree T in postorder
- ✓ **Inorder(T):** Traverses all the nodes of tree T in inorder.

6.6 Binary Tree Traversal

The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner. There are basically three different ways in which we traverse a binary tree.

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



6.6.1 Pre-order traversal

- 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

- The preorder traversal output of the given tree is: A B D H I E C F G
  The preorder is also known as depth first order.
- C implementation of preorder traversal technique is as follows:

```
void preorder(struct bnode *tree)
{
        if(tree!=NULL)
        {
                printf("%d\n", tree->data);
                preorder(tree->left);
                preorder(tree->right);
        }
}
```

6.6.2 In-order traversal

- 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.
- The inorder traversal output of the given tree is: H D I B E A F C G
- C implementation of inorder traversal technique is as follows:

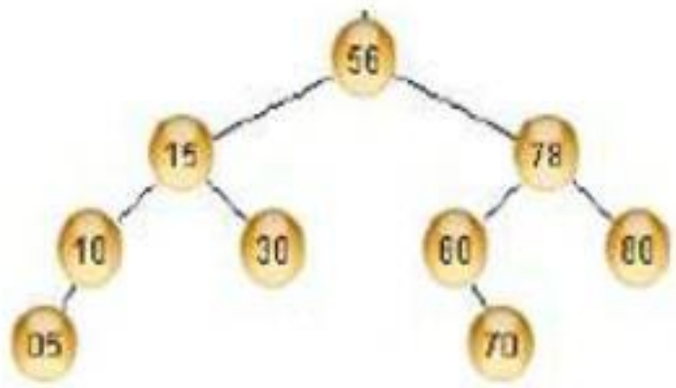
```
void inorder(struct bnode *tree)
{
        if(tree!=NULL)
        {
                inorder(tree->left);
                printf("%d\n", tree->data);
                inorder(tree->right);
        }
}
```

naresh.com.np

6.6.3 Post-order traversal

- 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.
- The post-order traversal output of the given tree is: H I D E B F G C A
- C implementation of post order traversal technique is as follows:

```
void postorder(struct bnode *tree)
{
        if(tree!=NULL)
        {
                postorder(tree->left);
                postorder(tree->right);
                printf("%d\n", tree->data);
        }
}
```




```
Result of preorder traversal  :  56 15 10 05 30 78 60 70 80
Result of inorder traversal  :  05 10 15 30 56 60 70 78 80
Result of postorder traversal  :  05 10 30 15 70 60 80 78 56
```

Examples of binary tree traversals

Rebuild a binary tree from Inorder and Preorder/Postorder traversals

This is well known problem where given any two traversals of a tree such as inorder & preorder or inorder & postorder we need to rebuild the tree.

Example: Given preorder and inorder traversal construct a binary tree
Preorder Traversal: 56, 15, 10, 05, 30, 78, 60, 70, 80
Inorder Traversal: 05, 10, 15, 30, 56, 60, 70, 78, 80
[Since we know that the first node in preorder is its root, we can easily locate the root node in the inorder traversal and hence we can obtain left subtree and right subtree from the inorder]

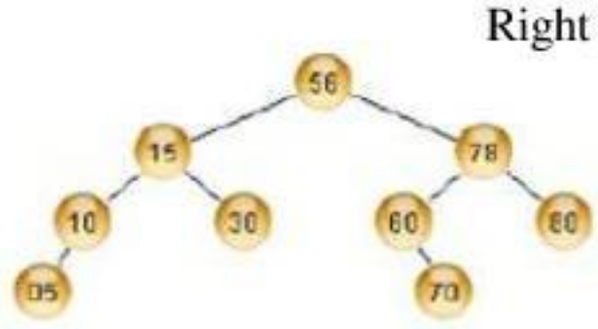
Iteration 1:
Root- {56}
Left subtree- {05, 10, 15, 30}
Right subtree- {60, 70, 78, 80}

Iteration 2:
Root- {15}
Left subtree- {05, 10}
Right subtree- {30}

Root- {78}
Left subtree- {60, 70}
Right subtree- {80}

Iteration 3:
Root- {10}
Left subtree- {05}
Right subtree-

Root- {60}
Left subtree-
Right subtree- {70}

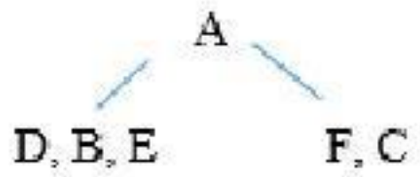


Example: Given inorder and postorder traversal construct a binary tree

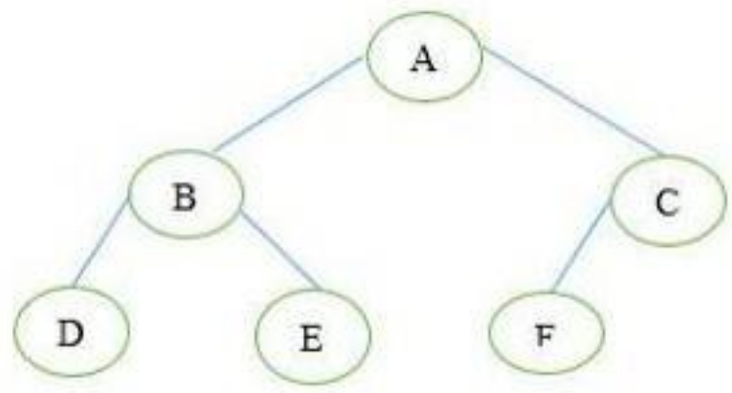
Inorder sequence: **D B E A F C**
Postorder sequence: **D E B F C A**

[In a postorder sequence, rightmost element is the root of the tree. So we know **A** is root.

Search for A in Inorder sequence. Once we know position of A (or index of A) in inorder sequence, we also know that all elements on left side of A are in left subtree and elements on right are in right subtree]



Recursively follow above step and we get the complete tree



6.7 Binary Search Tree (BST)

- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.
- **Definition**:

  A binary search tree is a binary tree that is either empty or in which each node contains a key that satisfies the following conditions:
  - All keys (if any) in left sub-tree of the root are smaller than that of a root.
  - The key in the root is always less than all the keys in the right sub-tree.
  - The left and right sub-trees of the root are again binary search trees.

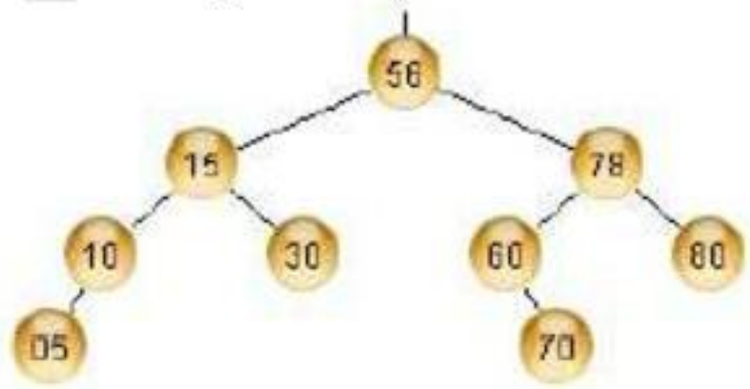


Figure: A Binary Search Tree

- The major advantage of binary search trees is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Operations on Binary search tree (BST):

Following operations can be done in BST:

- **Search(k, T):** Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- **Insert(k, T):** Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
- **Delete(k, T):**Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- **FindMin(T), FindMax(T)**: Find minimum and maximum element from the given nonempty BST.

Naresh Prasad.com.np

6.7.1 Searching through the BST

- Problem: Search for a given target value in a BST.
- Idea: Compare the target value with the element in the root node.
  - ✓ If the target value is equal, the search is successful.
  - ✓ If target value is less, search the left subtree.
  - ✓ If target value is greater, search the right subtree.
  - ✓ If the subtree is empty, the search is unsuccessful.

BST search algorithm

To find which if any node of a BST contains an element equal to target. *curr* is pointer to hold the node address.

1. Set *curr* **to the BST's root.**
2. Repeat:
   2.1. If *curr* is null:
      2.1.1. Terminate with answer none.
   2.2. Otherwise, if target is equal to *curr's* element:
      2.2.1. Terminate with answer *curr*.
   2.3. Otherwise, if target is less than *curr's* element:
      2.3.1. Set *curr* to *curr's* left child.
   2.4. Otherwise, if target is greater than *curr's* element:
      2.4.1. Set *curr* to *curr's* right child.
3. End

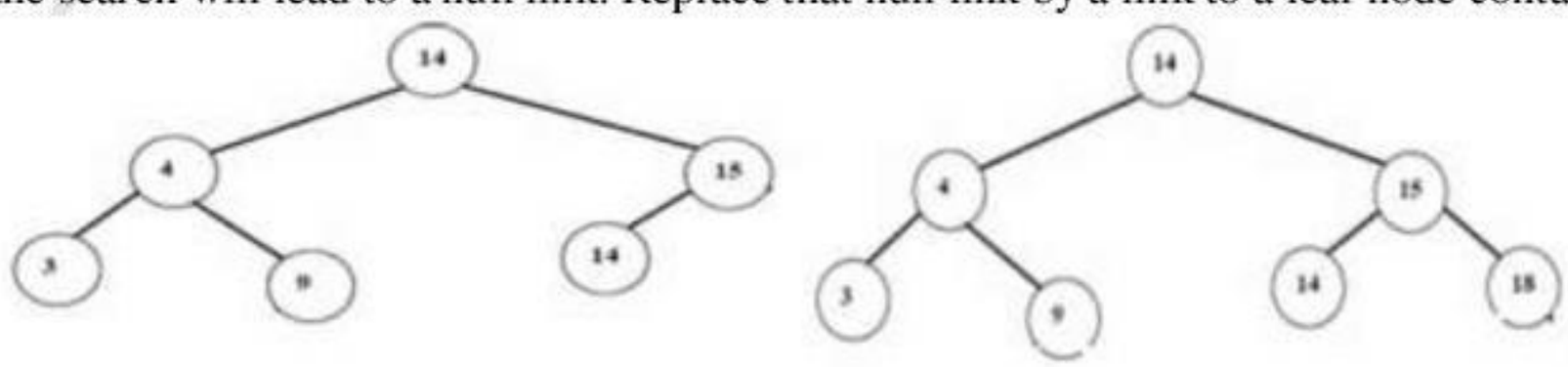
C function for BST searching:

```
void BinSearch(struct bnode *root , int key)
{
        if(root == NULL)
        {
                printf("The number does not exist");
                exit(1);
        }
        else if (key == root->info)
        {
                printf("The searched item is found"):
        }
        else if(key < root->info)
        {
                return BinSearch(root->left, key);
        }
        else
                return BinSearch(root->right, key);
}
```

6.7.2 Insertion of a node in BST

To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this a search is carried out. If the search is unsuccessful, then item is inserted.

- **Idea**: To insert a new element into a BST, proceed as if searching for that element. If the element is not already present, the search will lead to a null link. Replace that null link by a link to a leaf node containing the new element.



insert(18)

nareshpad.com.np

BST insertion algorithm

To insert the element *elem* into a BST:

1. Set *parent* to null, and set *curr* to the BST's root.
2. Repeat:
   - 2.1. If *curr* is null:
     - 2.1.1. Replace the null link from which *curr* was taken (either the BST's root or *parent*'s left child or *parent*'s right child) by a link to a newly-created leaf node with element *elem*.
     - 2.1.2. Terminate.
   - 2.2. Otherwise, if *elem* is equal to *curr*'s element:
     - 2.2.1. Terminate.
   - 2.3. Otherwise, if *elem* is less than *curr*'s element:
     - 2.3.1. Set *parent* to *curr*, and set *curr* to *curr*'s left child.
   - 2.4. Otherwise, if *elem* is greater than *curr*'s element:
     - 2.4.1. Set *parent* to *curr*, and set *curr* to *curr*'s right child.
3. End

C function for BST insertion

```
void insert(struct bnode *root, int item)
{
        if(root=NULL)
        {
                root=(struct bnode*)malloc (sizeof(struct bnode));
                root->left=root->right=NULL;
                root->info=item;
        }
        else
        {
                if(item<root->info)
                        root->left=insert(root->left, item);
                else
                        root->right=insert(root->right, item);
        }
}
```

6.7.3 Deleting a node from the BST

While deleting a node from BST, there may be three cases:

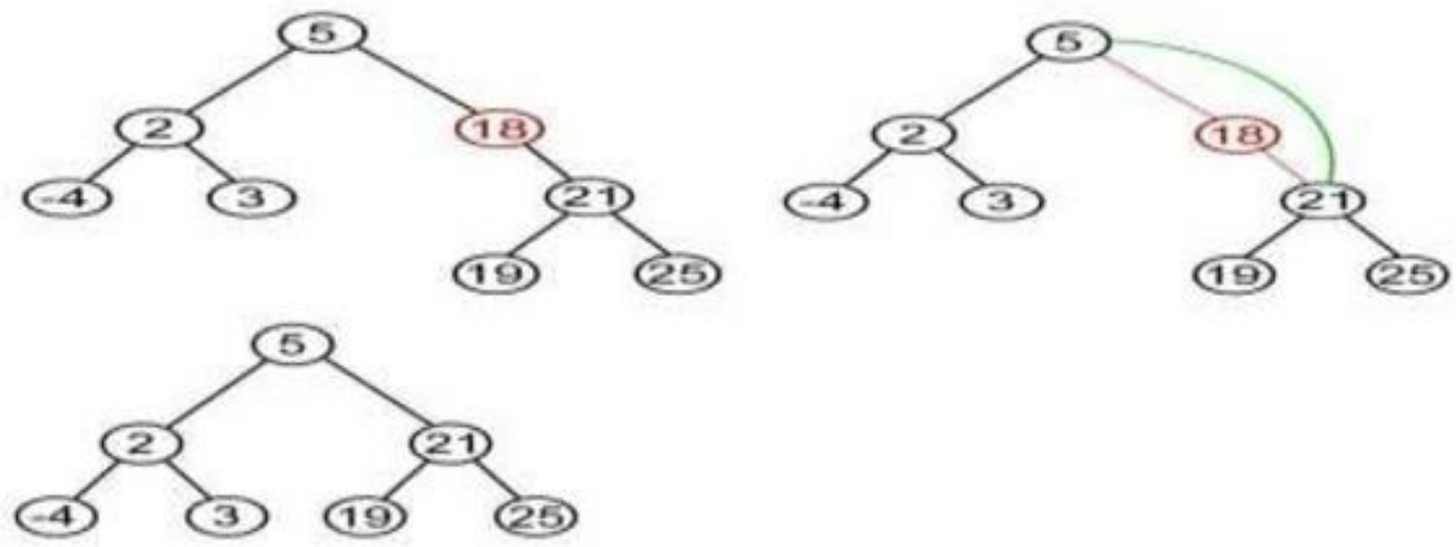
1. The node to be deleted may be a leaf node:

   In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.

Suppose node to be deleted is -4

2. The node to be deleted has one child

   In this case the child of the node to be deleted is appended to its parent node. Suppose node to be deleted is 18

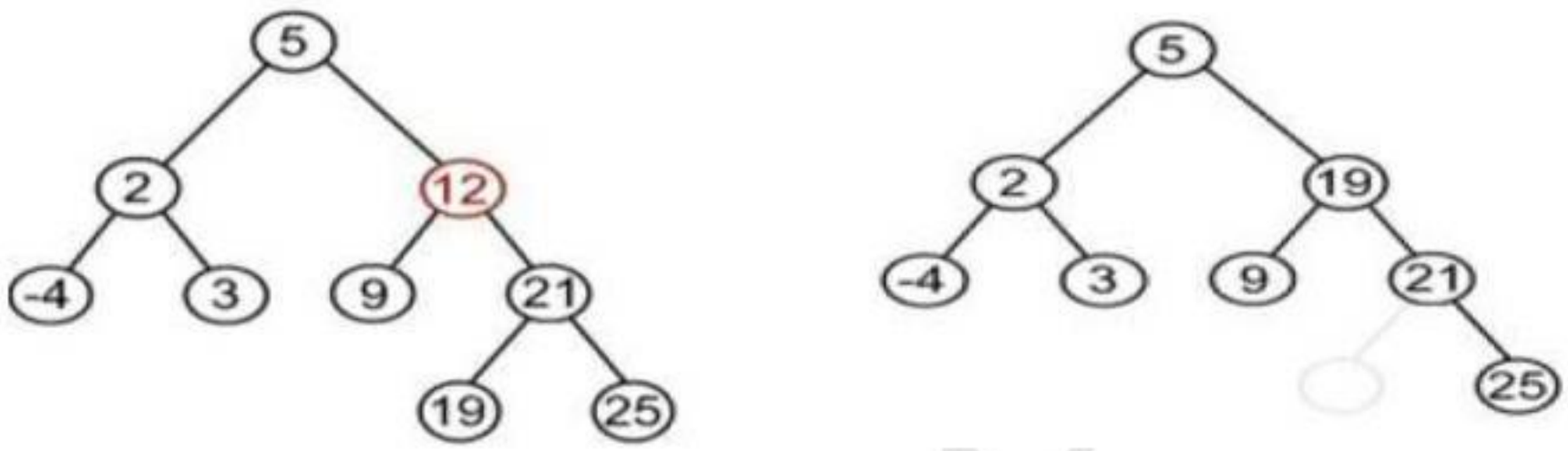


3. the node to be deleted has two children

In this case node to be deleted is replaced by its in-order successor node.

OR

If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.



Suppose node to deleted is 12

Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.

General algorithm to delete a node from a BST

1. Start
2. if a node to be deleted is a leaf nod at left side then simply delete and set null pointer to it's parent's left pointer.
3. if a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer
4. if a node to be deleted has on child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
   a. right most node of it's left sub-tree or
   b. left most node of it's right sub-tree.
6. End

C function to delete a node from a BST

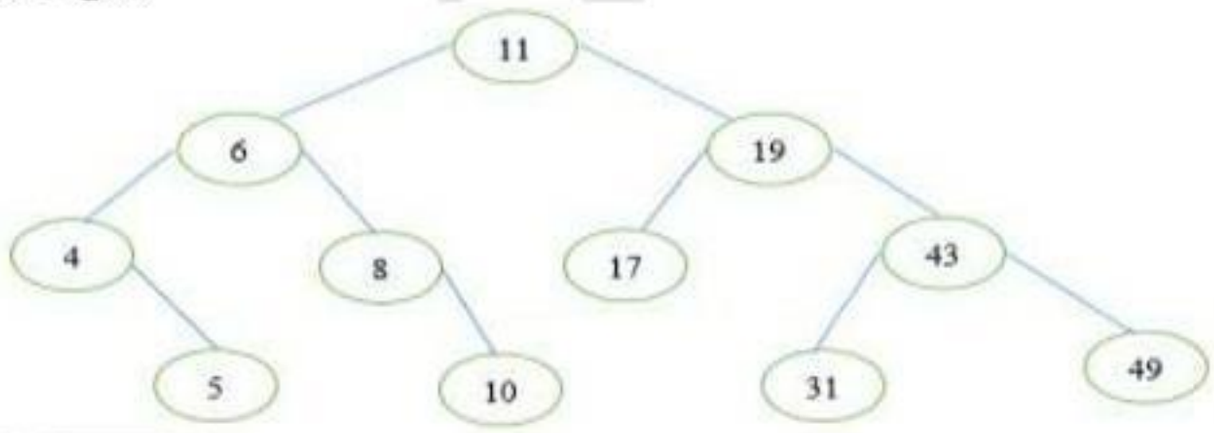
```
struct bnode *Delete(struct bnode *root, int item)
{
        struct bnode *temp;
        if(root==NULL)
        {
                printf("Empty tree");
                return;
        } else if(item<root->info)
                root->left=Delete(root->left, item);
        else if(item>root->info)
                root->right=Delete(root->right, item);
        else if(root->left!=NULL &&root->right!=NULL) //node has two child
        {
                temp=find_min(root->right);
```

```
            root->info=temp->info;
            root->right=Delete(root->right, root->info);
        } else
        {
            temp=root;
            if(root->left==NULL)
                root=root->right;
            else if(root->right==NULL)
                root=root->left;
            free(temp);
        }
        return(temp);
    }
```

- **C function for find minimum element**

```
struct bnode *find_min(struct bnode *root)
{
    if(root==NULL)
        return 0;
    else if(root->left==NULL)
        return root;
    else
        return(find_min(root->left));
}
```

- **Given a sequence of numbers: 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31, Draw a binary search tree by inserting the given numbers from left to right.**



- **Construct BST from given preorder traversal**

10, 5, 1, 7, 40, 50

In given data, 10 is the first element (The first element of preorder traversal is always root), so we make it root. Now we look for the first element greater than 10, we find 40. So we know the structure of BST is as following.

```
        10
       /  \
{5, 1, 7}   {40, 50}
```

We recursively follow above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree.

```
        10
       /  \
      5    40
     / \     \
    1   7     50
```

6.8 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

6.8.1 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

6.8.2 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.

6.9 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. Figure shows some example for AVL trees.

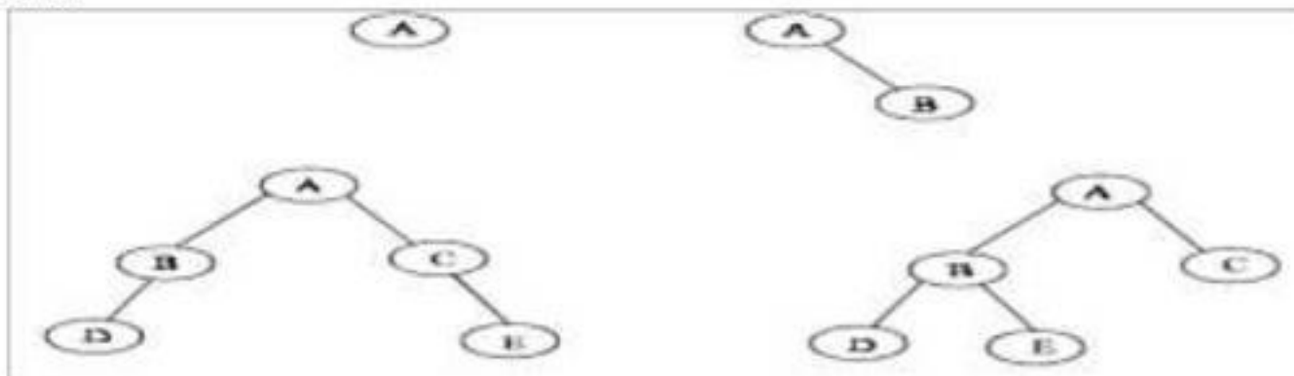


Figure: AVL Trees

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.

- **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 Figure A, 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.

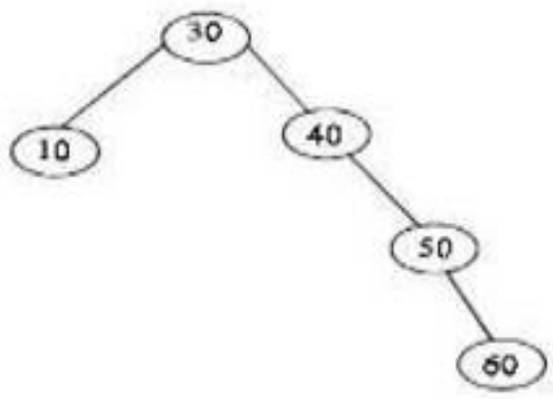


Figure A

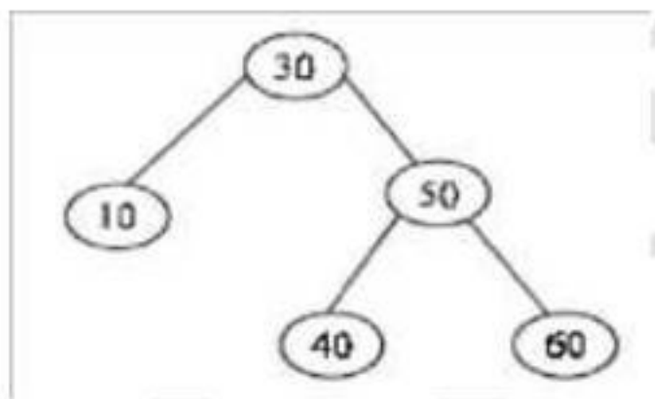


Figure B

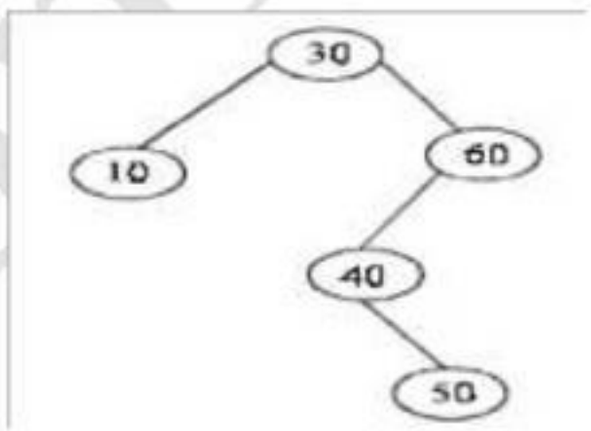


Figure C

Figure B is a balance tree of the unbalanced tree in Figure A. Consider a tree in Figure C 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 Figure D.

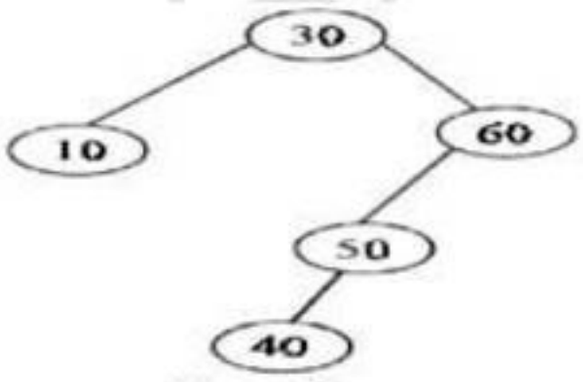


Figure D

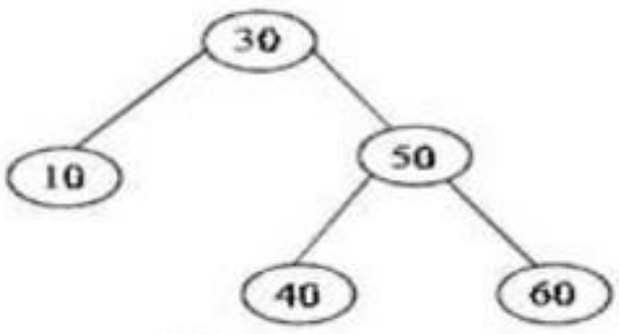
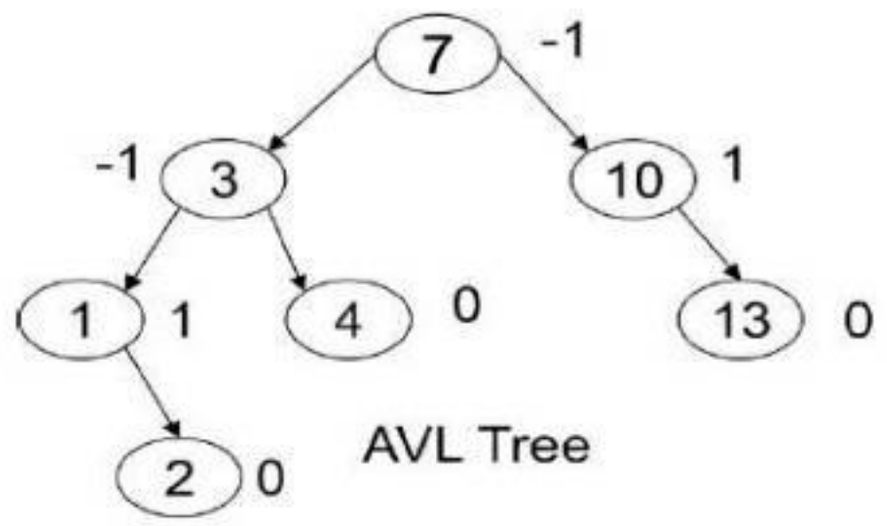
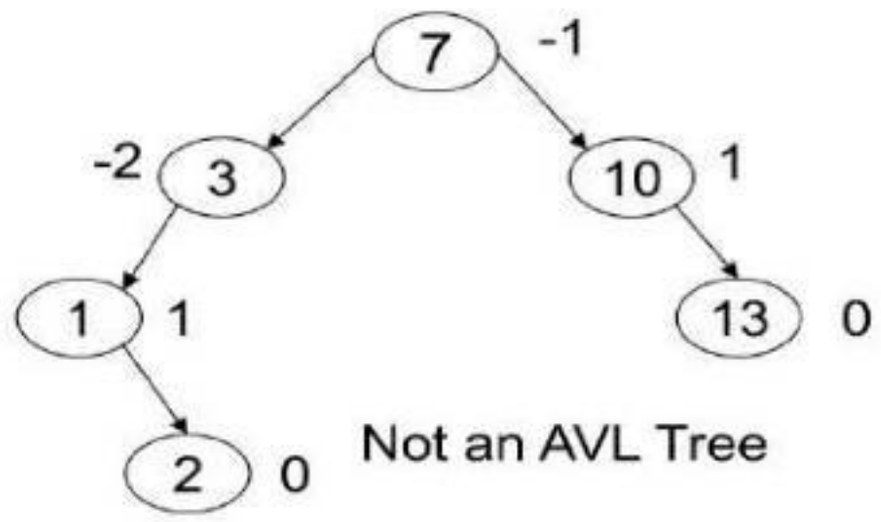


Figure E

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 Figure E. Balanced binary tree is a very useful data structure for searching the element with less time.

- **Need for AVL tree**
  - The disadvantage of a binary search tree is that its height can be as large as N-1
  - This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
  - We want a tree with small height
  - A binary tree with N node has height at least Q(log N)
  - Thus, our goal is to keep the height of a binary search tree O(log N)
  - Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.





HEIGHTS OF AVL TREE

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" which is the difference between the heights of subtrees of a root in the tree. The "height" of tree is the "number of levels" in the tree. The height of a tree is defined as follows:

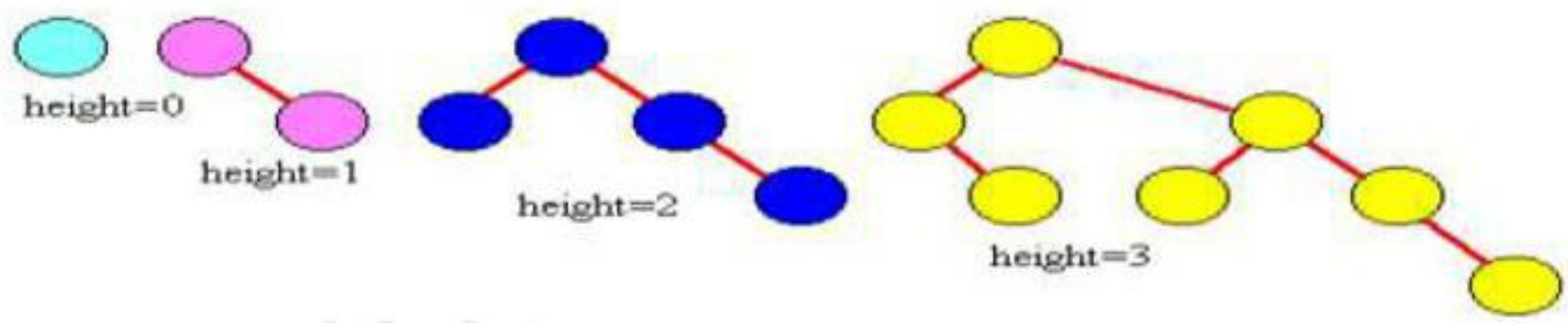
1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.
4. The height of a leaf is 1. The height of a null pointer is zero.

The height of an internal node is the maximum height of its children plus 1.

FINDING THE HEIGHT OF AVL TREE

AVL trees are identical to standard binary search trees except that for every node in an AVL tree, the height of the left and right subtrees can differ by at most 1 . AVL trees are HB-k trees (height balanced trees of order k) of order HB-1. The following is the height differential formula: |Height (Tl)-Height(Tr)|<=k

When storing an AVL tree, a field must be added to each node with one of three values: 1, 0, or -1. A value of 1 in this field means that the left subtree has a height one more than the right subtree. A value of -1 denotes the opposite. A value of 0 indicates that the heights of both subtrees are the same.



An AVL tree is a binary search tree with a balanced condition.

Balance Factor(BF) = Hl --- Hr.

Hl => Height of the left subtree.

Hr => Height of the right subtree.

If BF={ -1,0,1 } is satisfied, only then the tree is balanced.

AVL tree is a Height Balanced Tree.

If the calculated value of BF goes out of the range, then balancing has to be done.

Rotation :

Modification to the tree. i.e. , If the AVL tree is Imbalanced, proper rotations has to be done.

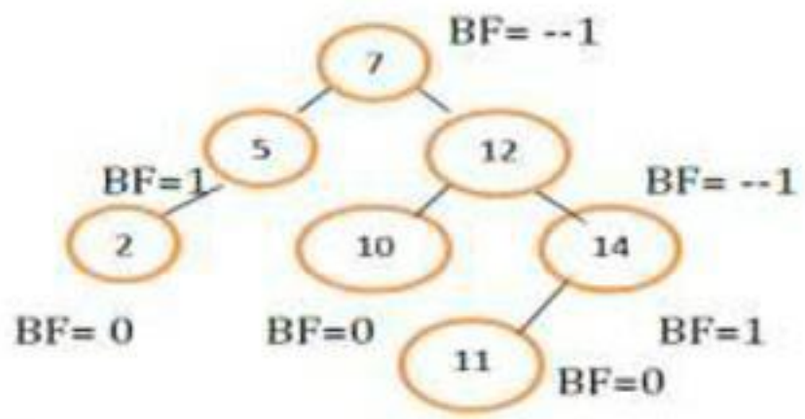
A rotation is a process of switching children and parents among two or three adjacent nodes to restore balance to a tree.

- There are two kinds of single rotation:

An insertion or deletion may cause an imbalance in an AVL tree.

The deepest node, which is an ancestor of a deleted or an inserted node, and whose balance factor has changed to -2 or +2 requires rotation to rebalance the tree.

Balance Factor :



This Tree is an AVL Tree and a height balanced tree.

- **An AVL tree causes imbalance when any of following condition occurs:**
  - An insertion into Right child's right subtree.
  - An insertion into Left child's left subtree.
  - An insertion into Right child's left subtree.
  - An insertion into Left child's right subtree.

These imbalances can be overcome by,

LL rotation: The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
RR rotation: The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
LR rotation: The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
RL rotation: The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

1. Single Rotation

- LL (Left -- Left rotation)
- RR (Right -- Right rotation)

2. Double Rotation

- RL (Right -- Left rotation) --- Do single Right, then single Left.
- LR (Left -- Right rotation) --- Do single Left, then single Right.

- **General Representation of Single Rotation**

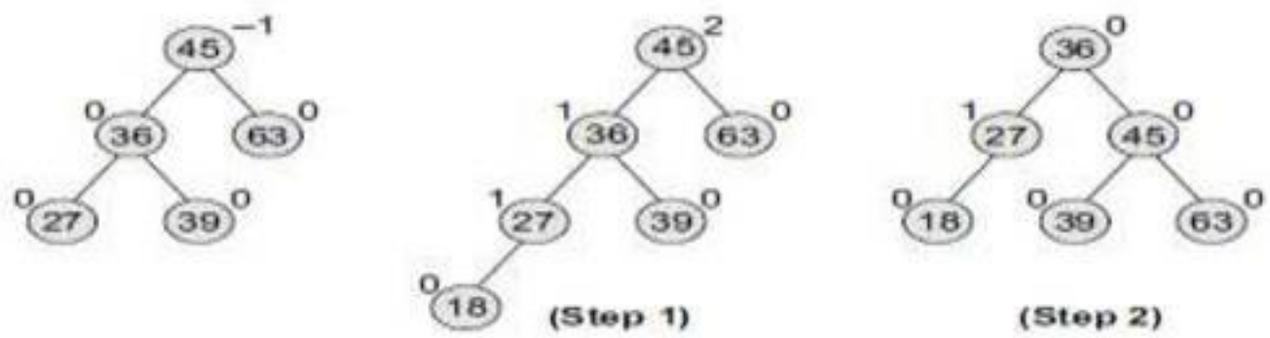
1. LL Rotation:

- The left child x of a node y becomes y's parent.
- y becomes the right child of x.
- The right child T2 of x, if any, becomes the left child of y.

Note: The pivot of the rotation is the deepest unbalanced node

nareshpad.com

Example: Consider the AVL tree given in Figure and insert 18 into it.

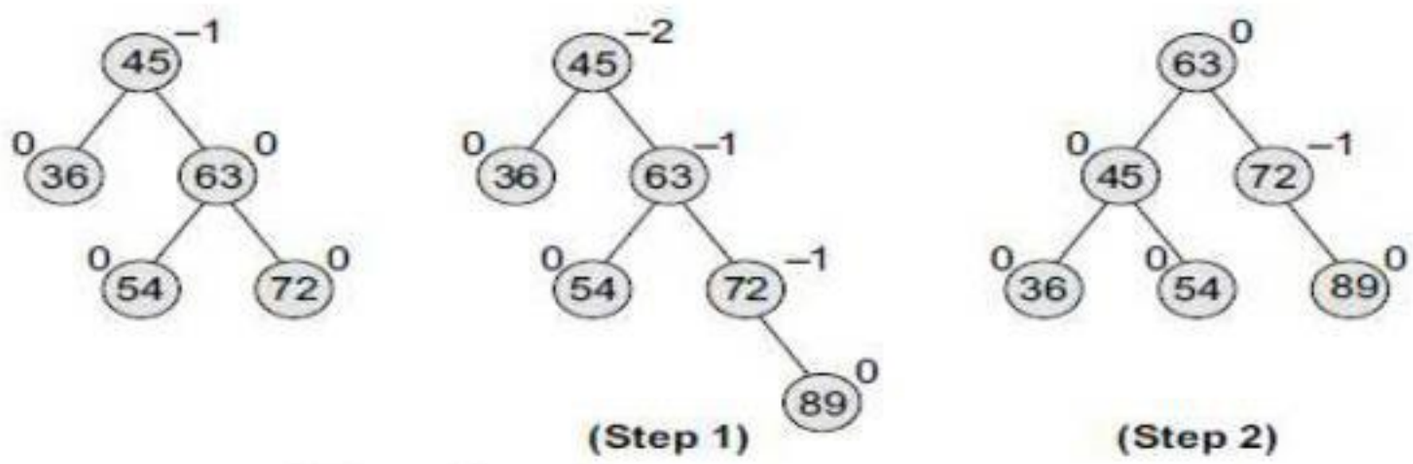


2. RR Rotation :

- The right child y of a node x becomes x's parent.
- x becomes the left child of y.
- The left child T2 of y, if any, becomes the right child of x.

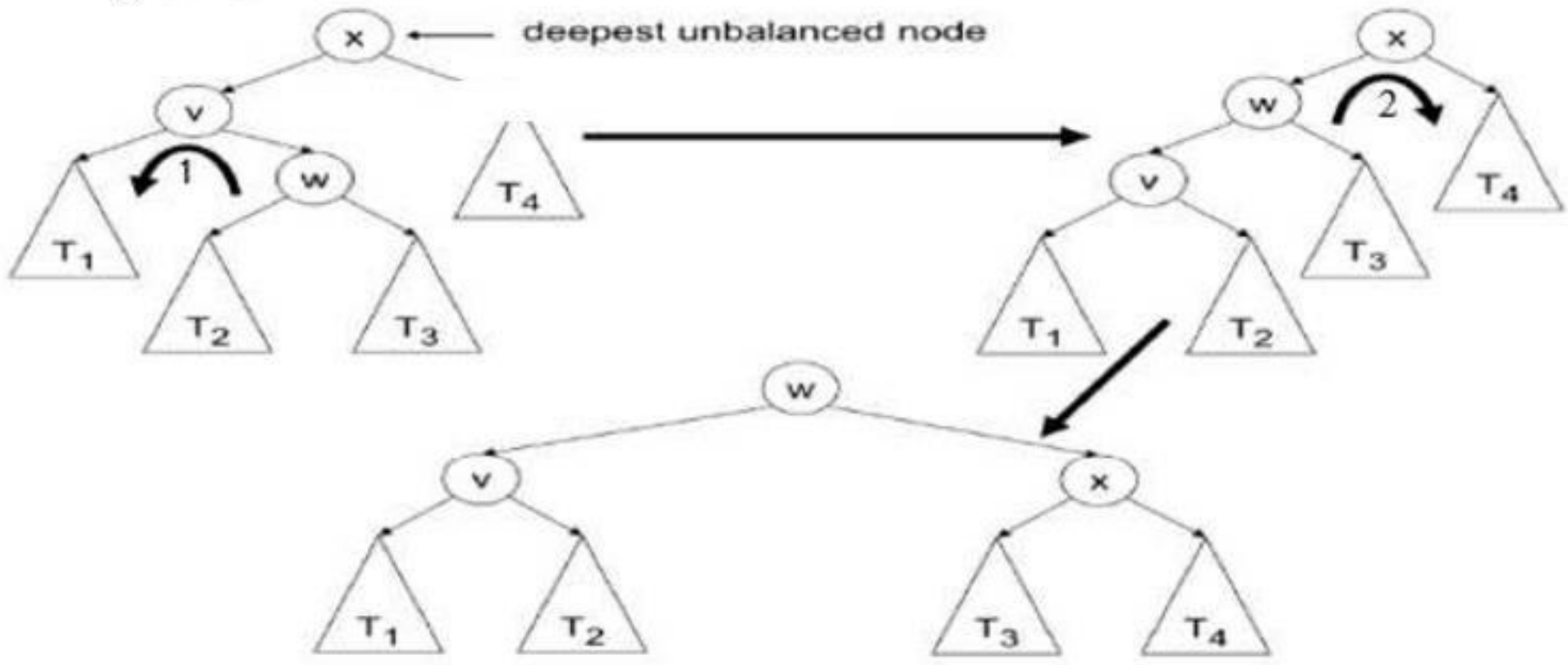
Note: The pivot of the rotation is the deepest unbalanced node

Example: Consider the AVL tree given in Figure and insert 89 into it.



General Representation of Double Rotation

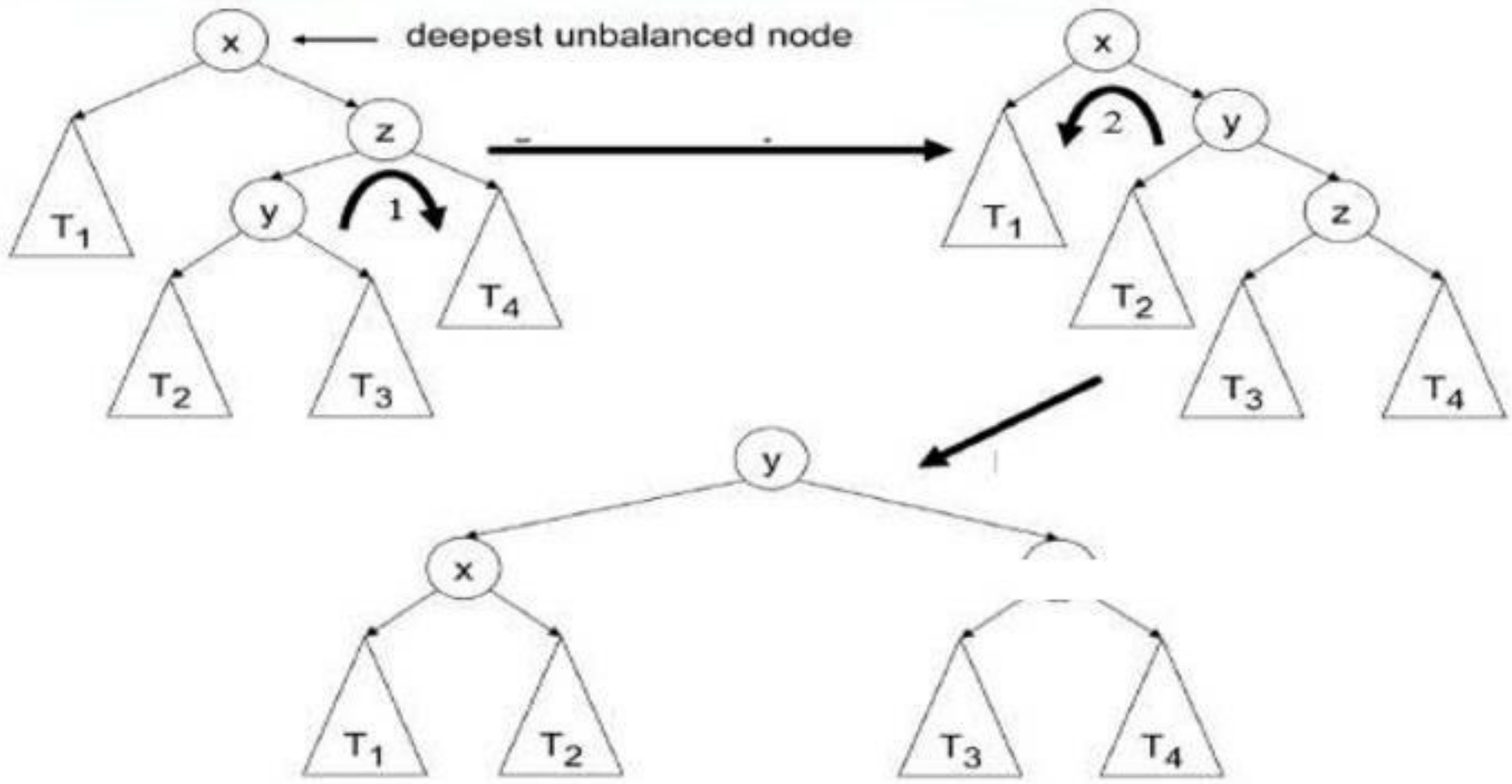
1. Left –Right Rotation



Note: First pivot is the left child of the deepest unbalanced node; second pivot is the deepest unbalanced node

2. Right-Left Rotation

Note: First pivot is the right child of the deepest unbalanced node; second pivot is the deepest unbalanced node

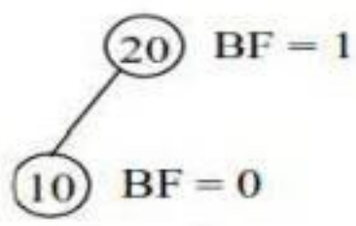


EXAMPLE:

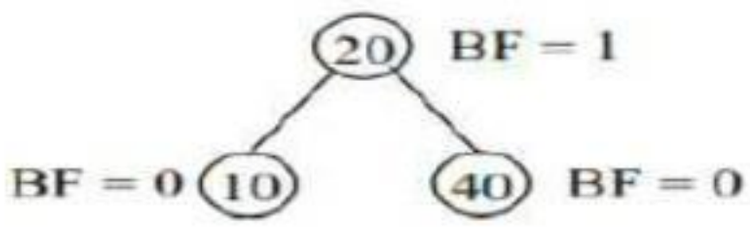
LET US CONSIDER INSERTING OF NODES 20,10,40,50,90,30,60,70 in an AVL TREE

Step 1:(Insert the value 20)

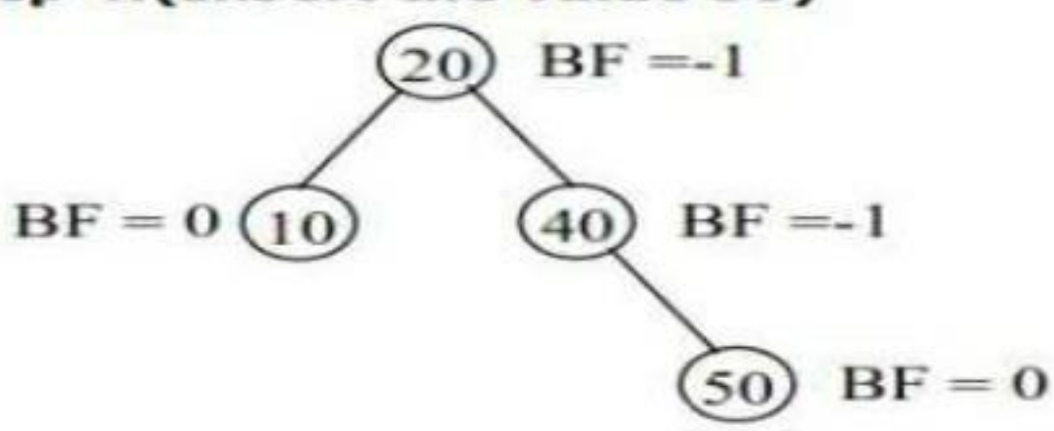
Step 2: (Insert the value 10)



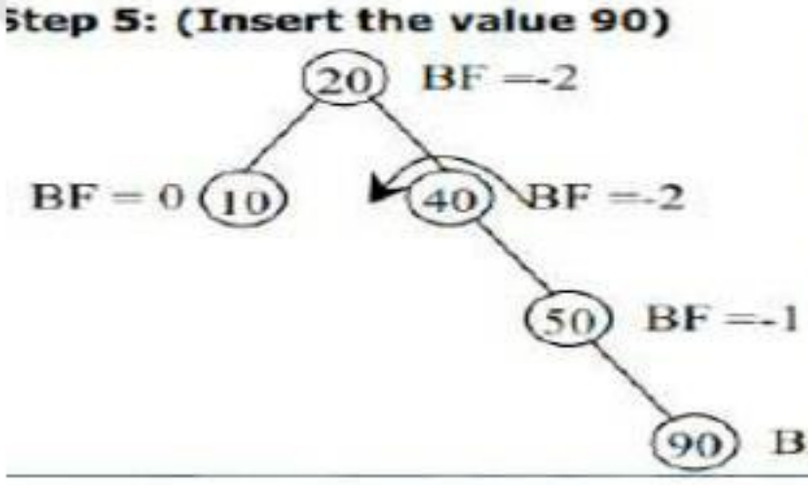
Step 3: (Insert the value 40)



Step 4:(Insert the value 50)

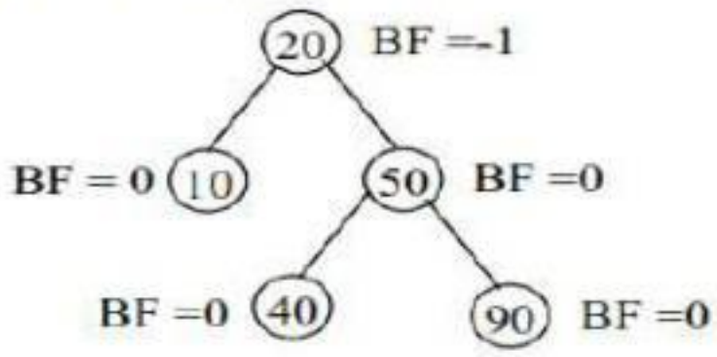


Step 5: (Insert the value 90)



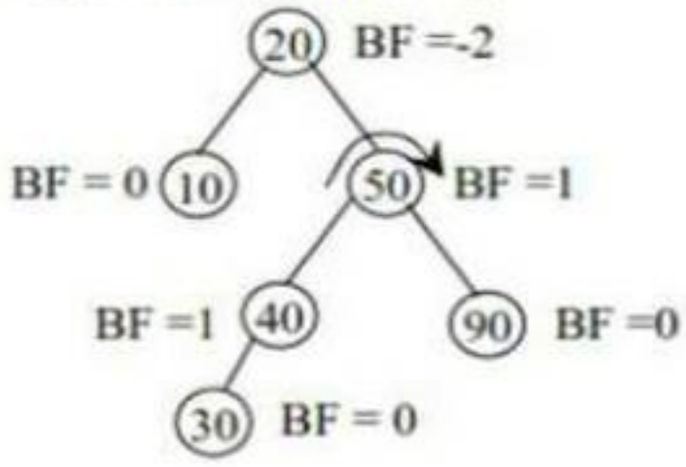
Now the tree is unbalanced due to the insertion of node in the Right subtree of the Right subtree. So we have to make single Right rotation in the node '40'.

After the Right Rotation



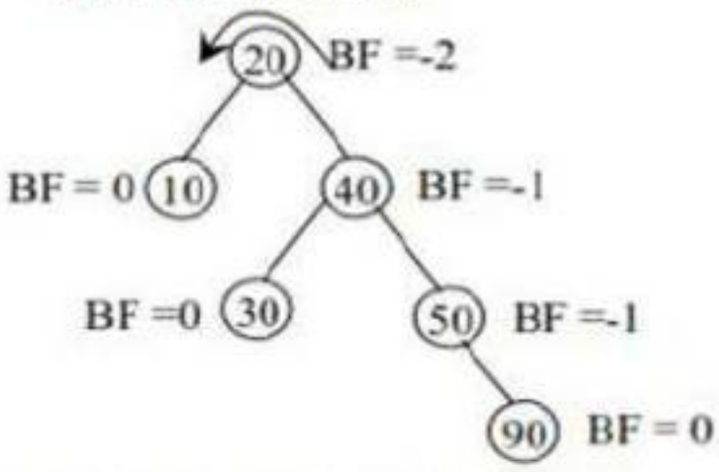
Now the tree is Balanced

Step 6: (Insert the value 30)

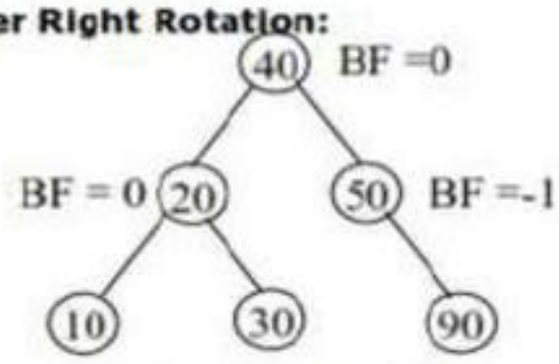


Now the tree is unbalanced due to the insertion of node'30' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '50' and then with Right on the node '20'.

After Left Rotation:

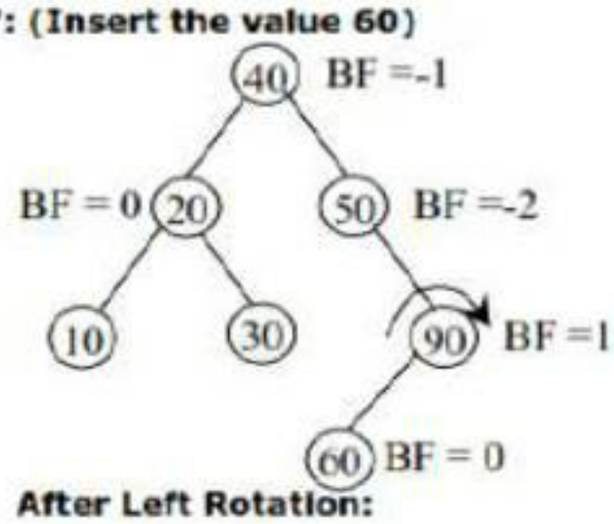


After Right Rotation:



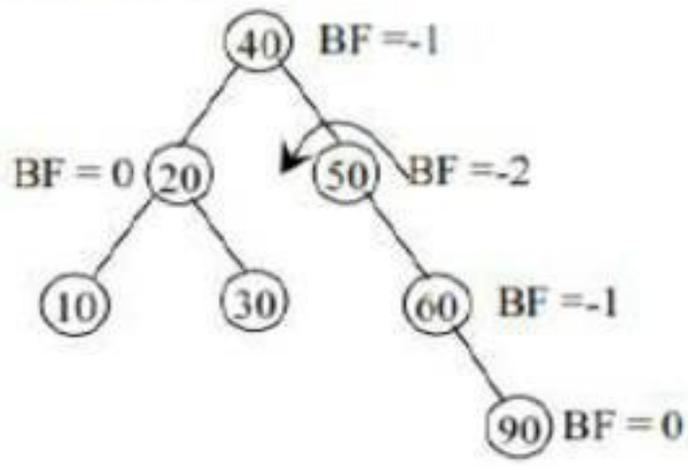
Now the tree is Balanced

Step 7: (Insert the value 60)

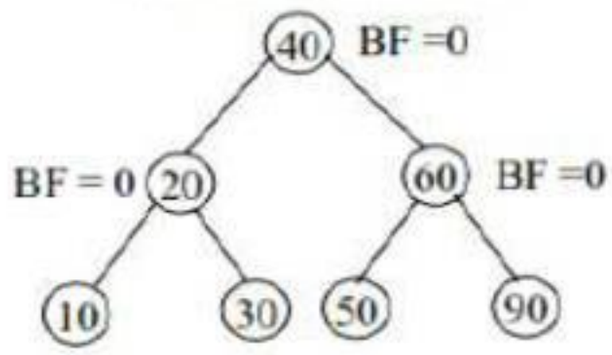


Now the tree at the node '50'is unbalanced due to the insertion of node '60' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '90' and then with Right on the node '50'.

After Left Rotation:

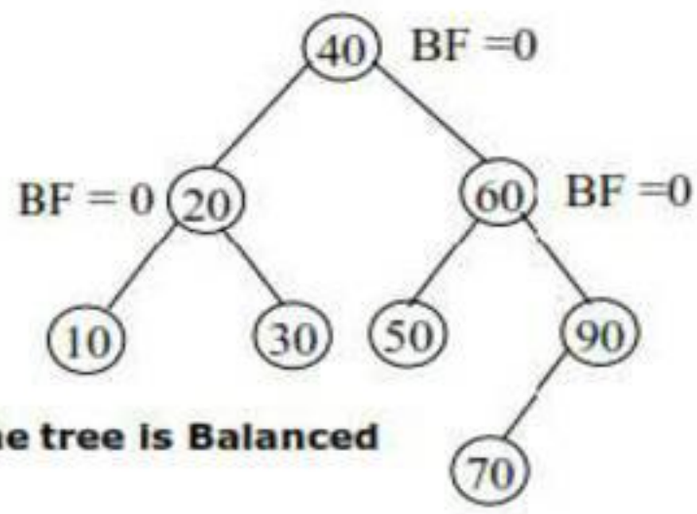


After Right Rotation:



Now the tree is Balanced

Step 8: (Insert the value 70)



The tree is Balanced

APPLICATIONS

AVL trees play an important role in most computer related applications. The need and use of AVL trees are increasing day by day. Their efficiency and less complexity add value to their reputation. Some of the applications are

- Contour extraction algorithm
- Parallel dictionaries
- Compression of computer files
- Translation from source language to target language
- Spell checker

ADVANTAGES OF AVL TREE

- AVL trees guarantee that the difference in height of any two subtrees rooted at the same node will be at most one. This guarantees an asymptotic running time of O(log(n)) as opposed to O(n) in the case of a standard BST.
- Height of an AVL tree with **n** nodes is always very close to the theoretical minimum.
- Since the AVL tree is height balanced the operation like insertion and deletion have low time complexity.
- Since tree is always height balanced. Recursive implementation is possible.
- The height of left and the right sub-trees should differ by atmost 1. Rotations are possible.

DISADVANTAGES OF AVL TREE

- One limitation is that the tree might be spread across memory
- As you need to travel down the tree, you take a performance hit at every level down
- One solution: store more information on the path
- Difficult to program & debug; more space for balance factor.
- Asymptotically faster but rebalancing costs time.
- Mioost larger searches are done in database systems on disk and use other structures

6.10 Multi-way Search Trees

- We have discussed that every node in a binary search tree contains one value and two pointers, left and right, which **point to the node's left and right sub-trees**, respectively. The structure of a binary search tree node is shown in Figure **A**.

Pointer to left sub-tree	Value or Key of the node	Pointer to right sub-tree

Figure A: Structure of a binary search tree node

- The same concept is used in an M-way search tree which has M – 1 values per node and M subtrees. In such a tree, M is called the degree of the tree. Note that in a binary search tree M = 2, so it has one value and two sub-trees. In other words, every internal node of an M-way search tree consists of pointers to M sub-trees and contains M – 1 keys, where M > 2. The structure of an M-way search tree node is shown in Figure **B**.

P_0	K_0	P_1	K_1	P_2	K_2	.....	P_{n-1}	K_{n-1}	P_n

Figure B: Structure of an M-way search tree node

In the structure shown, $P_0, P_1, P_2, \ldots, P_n$ are pointers to the node's sub-trees and $K_0, K_1, K_2, \ldots, K_{n-1}$ are the key values of the node. All the key values are stored in ascending order. That is, $K_i < K_{i+1}$ for $0 \le i \le n-2$.

- In an M-way search tree, it is not compulsory that every node has exactly M–1 values and M subtrees. Rather, the node can have anywhere from 1 to M–1 values, and the number of sub-trees can vary from 0 (for a leaf node) to i + 1, where i is the number of key values in the node. M is thus a fixed upper limit that defines how many key values can be stored in the node.

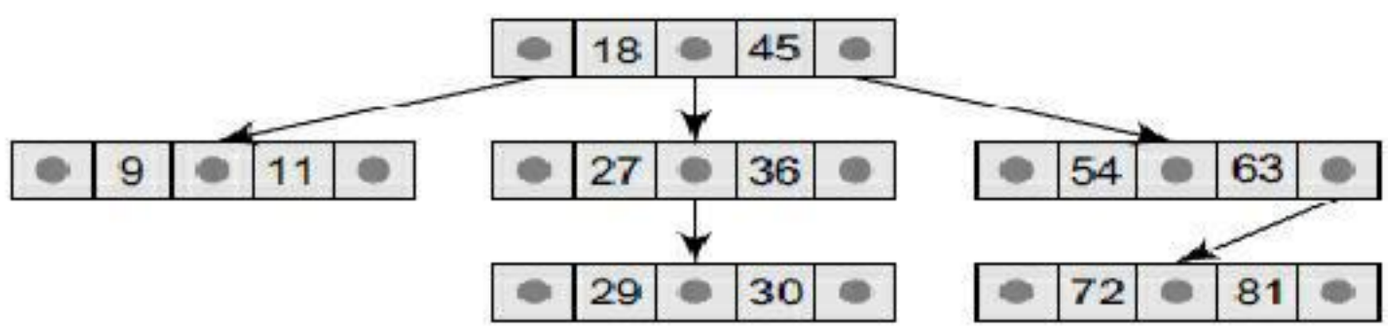


Figure C: M-way search tree of order 3

- Consider the M-way search tree shown in Figure C. Here M = 3. So a node can store a maximum of two key values and can contain pointers to three sub-trees.
- In our example, we have taken a very small value of M so that the concept becomes easier, but in practice, M is usually very large. Using a 3-way search tree, let us lay down some of the basic properties of an M-way search tree.
  - Note that the key values in the sub-tree pointed by P_0 are less than the key value K_0. Similarly, all the key values in the sub-tree pointed by P_1 are less than K_1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are less than K_i, where $0 \le i \le n-1$.
  - Note that the key values in the sub-tree pointed by P_1 are greater than the key value K_0. Similarly, all the key values in the sub-tree pointed by P_2 are greater than K_1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are greater than K_{i-1}, where $0 \le i \le n-1$.

  In an M-way search tree, every sub-tree is also an M-way search tree and follows the same rules.

6.11 B – Tree

A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access. A B tree of order m can have a maximum of m-1 keys and m pointers to its sub-trees. A B tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.

A B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic amortized time. A B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree. In addition it has the following properties:

1. Every node in the B tree has at most (maximum) m children.
2. Every node in the B tree except the root node and leaf nodes has at least (minimum) m/2 children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data.
3. The root node has at least two children if it is not a terminal (leaf) node.
4. All leaf nodes are at the same level.

An internal node in the B tree can have n number of children, where $0 \le n \le m$. It is not necessary that every node has the same number of children, but the only restriction is that the node should have at least m/2 children. As B tree of order 4 is given in Fig. 11.4.

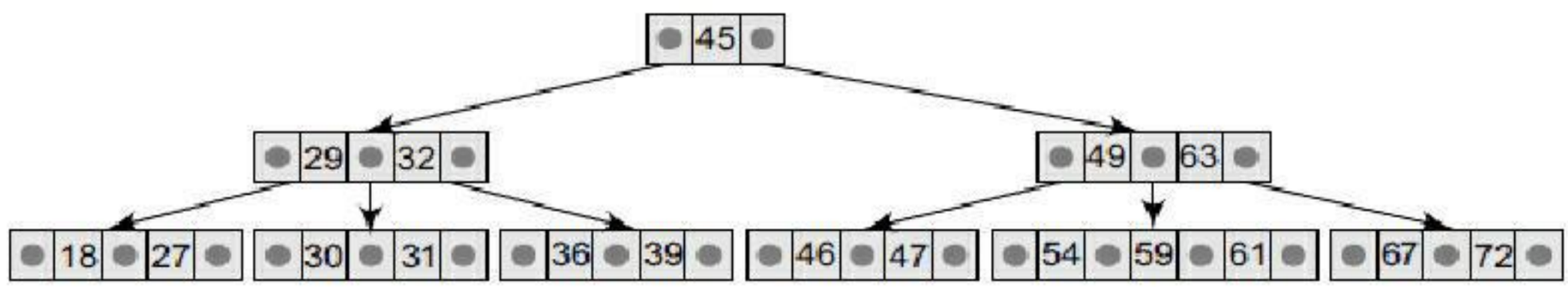


Figure 11.4 B tree of order 4

While performing insertion and deletion operations in a B tree, the number of child nodes may change. So, in order to maintain a minimum number of children, the internal nodes may be joined or split.

6.11.1 Searching for an Element in a B Tree

Searching for an element in a B tree is similar to that in binary search trees. Consider the B tree given in Fig. 11.4. To search for 59, we begin at the root node. The root node has a value 45 which is less than 59. So, we traverse in the right sub-tree. The right sub-tree of the root node has two key values, 49 and 63. Since $49 \le 59 \le 63$, we traverse the right sub-tree of 49, that is, the left sub-tree of 63. This sub-tree has three values, 54, 59, and 61. On finding the value 59, the search is successful.

Take another example. If you want to search for 9, then we traverse the left sub-tree of the root node. The left sub-tree has two key values, 29 and 32. Again, we traverse the left sub-tree of 29. We find that it has two key values, 18 and 27. There is no left sub-tree of 18, hence the value 9 is not stored in the tree.

Since the running time of the search operation depends upon the height of the tree, the algorithm to search for an element in a B tree takes $O(\log_t n)$ time to execute.

6.11.2 Inserting a New Element in a B Tree

In a B tree, all insertions are done at the leaf node level. A new value is inserted in the B tree using the algorithm given below.

1. Search the B tree to find the leaf node where the new key value should be inserted.
2. If the leaf node is not full, that is, it contains less than m-1 key values, then insert the new element in the node keeping the node's elements ordered.
3. If the leaf node is full, that is, the leaf node already contains m-1 key values, then
   (a) insert the new value in order into the existing set of keys,
   (b) split the node at its median into two nodes (note that the split nodes are half full), and
   (c) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

Example 11.1 Look at the B tree of order 5 given below and insert 8, 9, 39, and 4 into it.

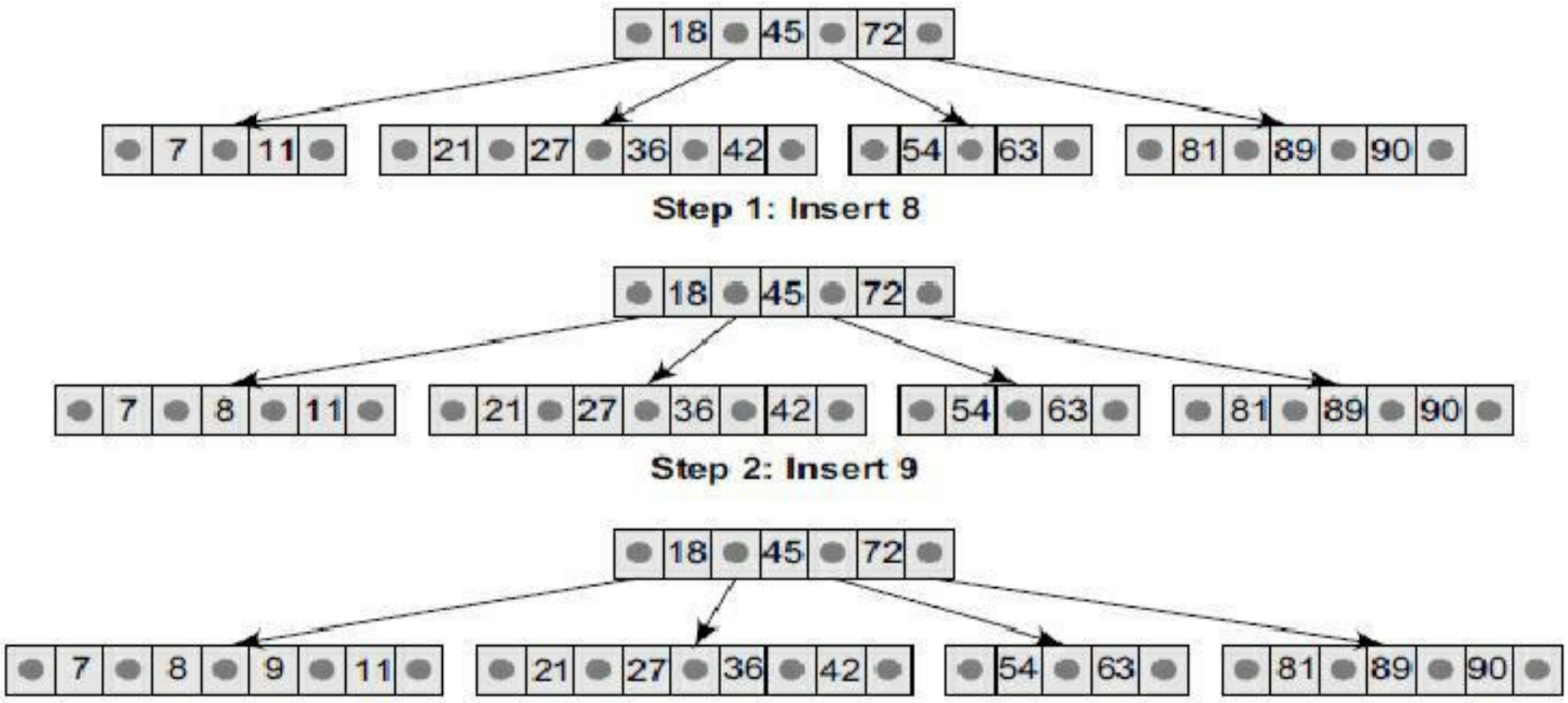


Figure 11.5(a)

Till now, we have easily inserted 8 and 9 in the tree because the leaf nodes were not full. But now, the node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value). The ordered set of values is given as 21, 27, 36, 39, and 42. The median value is 36, so push 36 into its parent's node and split the leaf nodes.

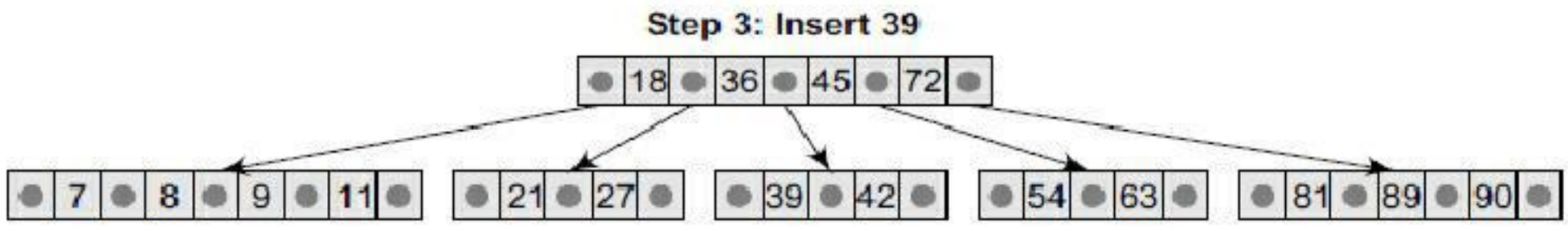


Figure 11.5(b)

Now the node in which 4 should be inserted is already full as it contains four key values. Here we split the nodes to form two separate nodes. But before splitting, we arrange the key values in order (including the new value). The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure.

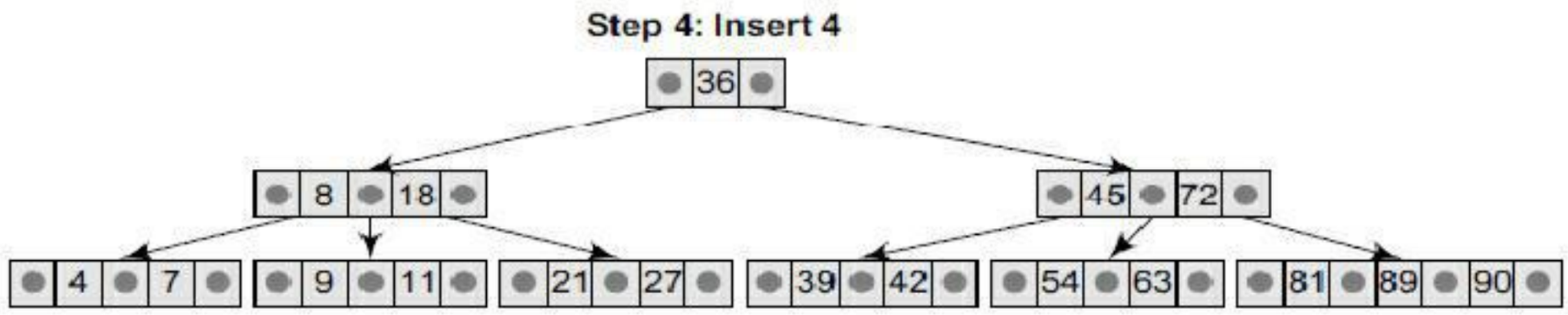


Figure 11.5(c) B tree

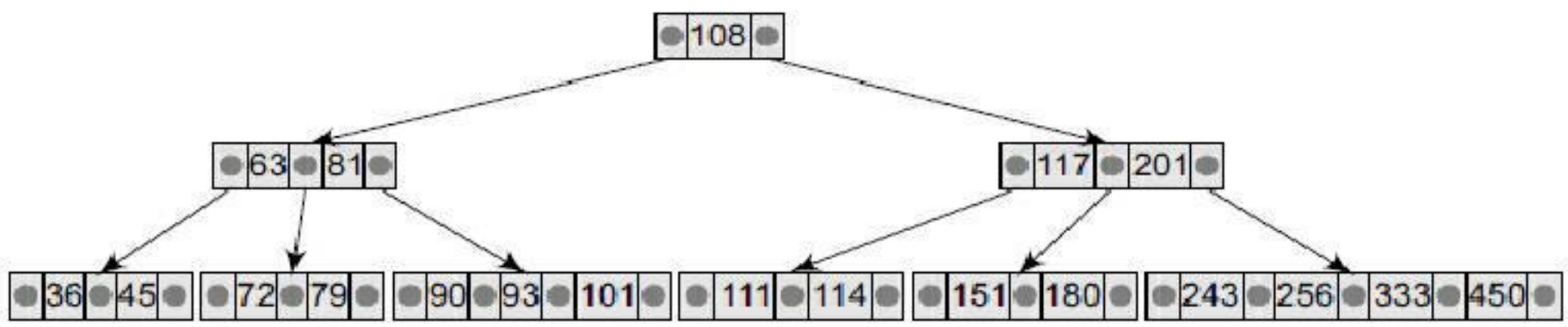
6.11.3 Deleting an Element from a B Tree

Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

1. Locate the leaf node which has to be deleted.
2. If the leaf node contains more than the minimum number of key values (more than m/2 elements), then delete the value.
3. Else if the leaf node does not contain m/2 elements, then fill the node by taking an element either from the left or from the right sibling.
   (a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
   (b) Else, if the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
4. Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements does not exceed the maximum number of elements a node can have, that is, m). If pulling the intervening element from the parent node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, thereby reducing the height of the B tree.

To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So the processing will be done as if a value from the leaf node has been deleted.

Example 11.2 Consider the following B tree of order 5 and delete values 93, 201, 180, and 72 from it (Fig. 11.6(a)).



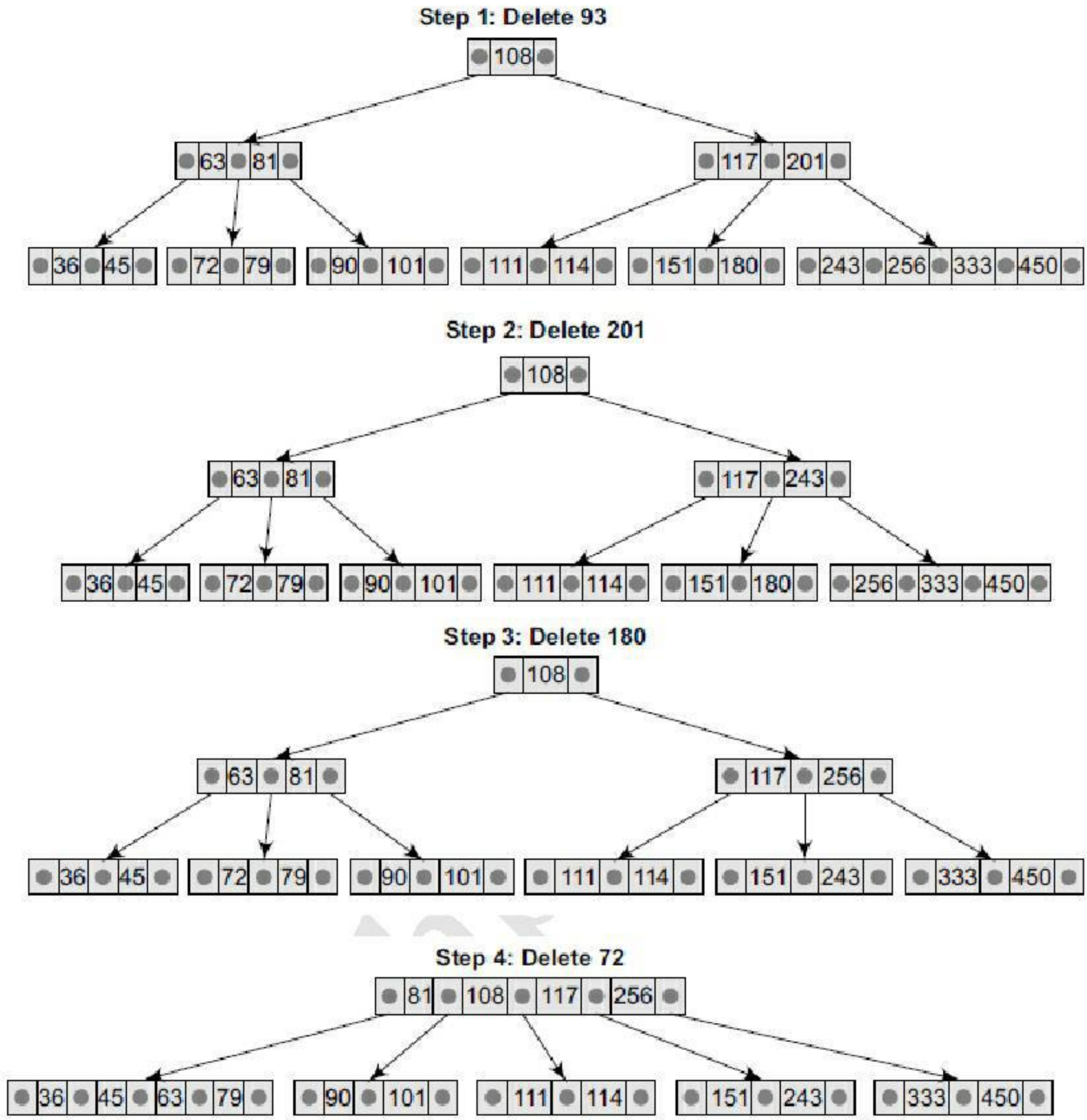
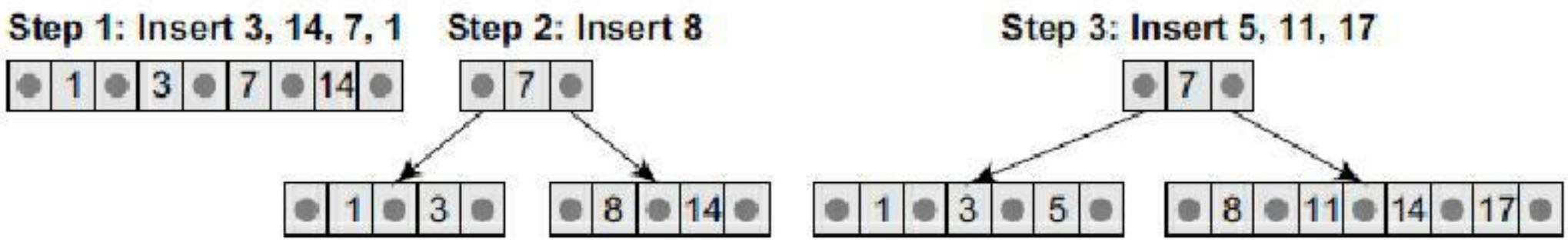


Figure 11.6 B tree

Example 11.4 Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.



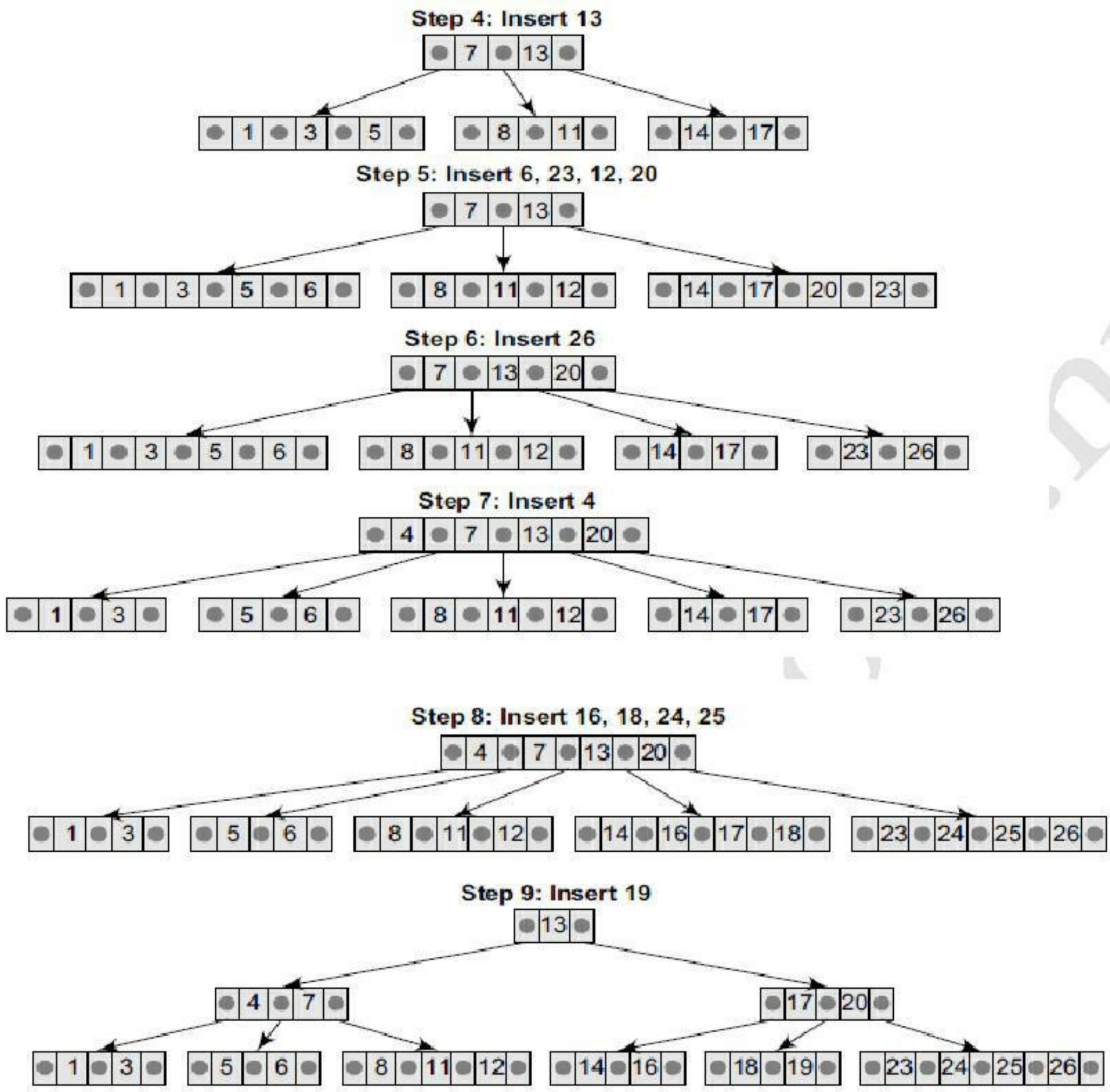


Figure 11.8 B tree

6.11.4 Applications of B Trees

A database is a collection of related data. The prime reason for using a database is that it stores organized data to facilitate its users to update, retrieve, and manage the data. The data stored in the database may include names, addresses, pictures, and numbers. For example, a teacher may wish to maintain a database of all the students that includes the names, roll numbers, date of birth, and marks obtained by every student.

Nowadays, databases are used in every industry to store hundreds of millions of records. In the real world, it is not uncommon for a database to store gigabytes and terabytes of data. For example, a telecommunication company maintains a customer billing database with more than 50 billion rows that contains terabytes of data. We know that primary memory is very expensive and is capable of storing very little data as compared to secondary memory devices like magnetic disks. Also, RAM is volatile in nature and we cannot store all the data in primary memory. We have no other option but to store data on secondary storage devices. But accessing data from magnetic disks is 10,000 to 1,000,000 times slower than accessing it from the main memory. So, B trees are often used to index the data and provide fast access.

Consider a situation in which we have to search an un-indexed and unsorted database that contains n key values. The worst case running time to perform this operation would be O(n). In contrast, if the data in the database is indexed with a B tree, the same search operation will run in O(log n). For example, searching for a single key on a set of one million keys will at most require 1,000,000 comparisons. But if the same data is indexed with a B tree of order 10, then only 114 comparisons will be

required in the worst case. Hence, we see that indexing large amounts of data can provide significant boost to the performance of search operations. When we use B trees or generalized M-way search trees, the value of m or the order of B trees is often very large. Typically, it varies from 128–512. This means that a single node in the tree can contain 127–511 keys and 128–512 pointers to child nodes.

We take a large value of m mainly because of three reasons:

- Disk access is very slow. We should be able to fetch a large amount of data in one disk access.
- Disk is a block-oriented device. That is, data is organized and retrieved in terms of blocks. So while using a B tree (generalized M-way search tree), a large value of m is used so that one single node of the tree can occupy the entire block. In other words, m represents the maximum number of data items that can be stored in a single block. m is maximized to speed up processing. More the data stored in a block, lesser the time needed to move it into the main memory.
- A large value minimizes the height of the tree. So, search operation becomes really fast.

6.12 B+ TREES

A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a key. While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; only keys are stored in the interior nodes. The leaf nodes of a B+ tree are often linked to one another in a linked list. This has an added advantage of making the queries simpler and more efficient. Typically, B+ trees are used to store large amounts of data that cannot be stored in the main memory. With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory. B+ trees store data only in the leaf nodes. All other nodes (internal nodes) are called index nodes or i-nodes and store index values. This allows us to traverse the tree from the root down to the leaf node that stores the desired data item. Figure 11.9 shows a B+ tree of order 3.

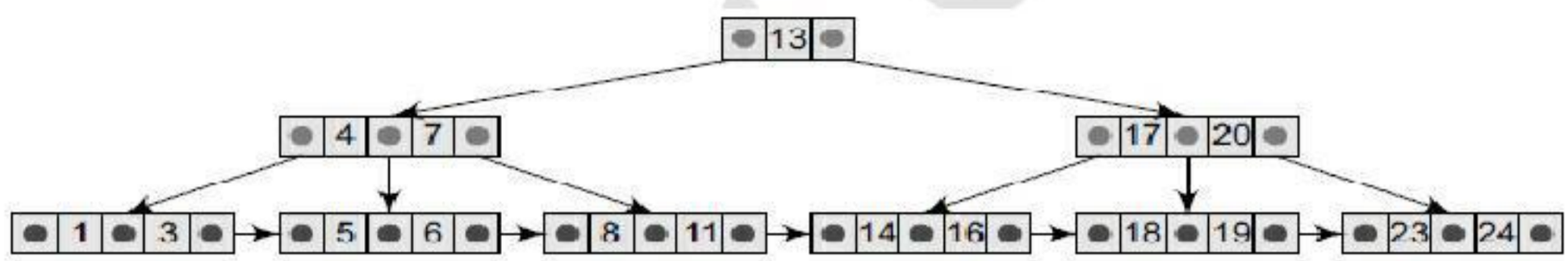


Figure 11.9 B+ tree of order 3

Many database systems are implemented using B+ tree structure because of its simplicity. Since all the data appear in the leaf nodes and are ordered, the tree is always balanced and makes searching for data efficient.

A B+ tree can be thought of as a multi-level index in which the leaves make up a dense index and the non-leaf nodes make up a sparse index. The advantages of B+ trees can be given as follows:

1. Records can be fetched in equal number of disk accesses
2. It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level
3. Height of the tree is less and balanced
4. Supports both random and sequential access to records
5. Keys are used for indexing

Table 11.1 Comparison between B trees and to B+ trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

6.12.1 Inserting a New Element in a B+ Tree

A new element is simply added in the leaf node if there is space for it. But if the data node in the tree where insertion has to be done is full, then that node is split into two nodes. This calls for adding a new index value in the parent index node so that future queries can arbitrate between the two new nodes.

However, adding the new index value in the parent node may cause it, in turn, to split. In fact, all the nodes on the path from a leaf to the root may split when a new value is added to a leaf node. If the root node splits, a new leaf node is created and the tree grows by one level.
The steps to insert a new node in a B+ Tree are summarized in Fig. 11.10.

Step 1:	Insert the new node as the leaf node.
Step 2:	If the leaf node overflows, split the node and copy the middle element to next index node.
Step 3:	If the index node overflows, split that node and move the middle element to next index page.

Figure 11.10 Algorithm for inserting a new node in a B+ tree

Example 11.5 Consider the B+ tree of order 4 given and insert 33 in it.

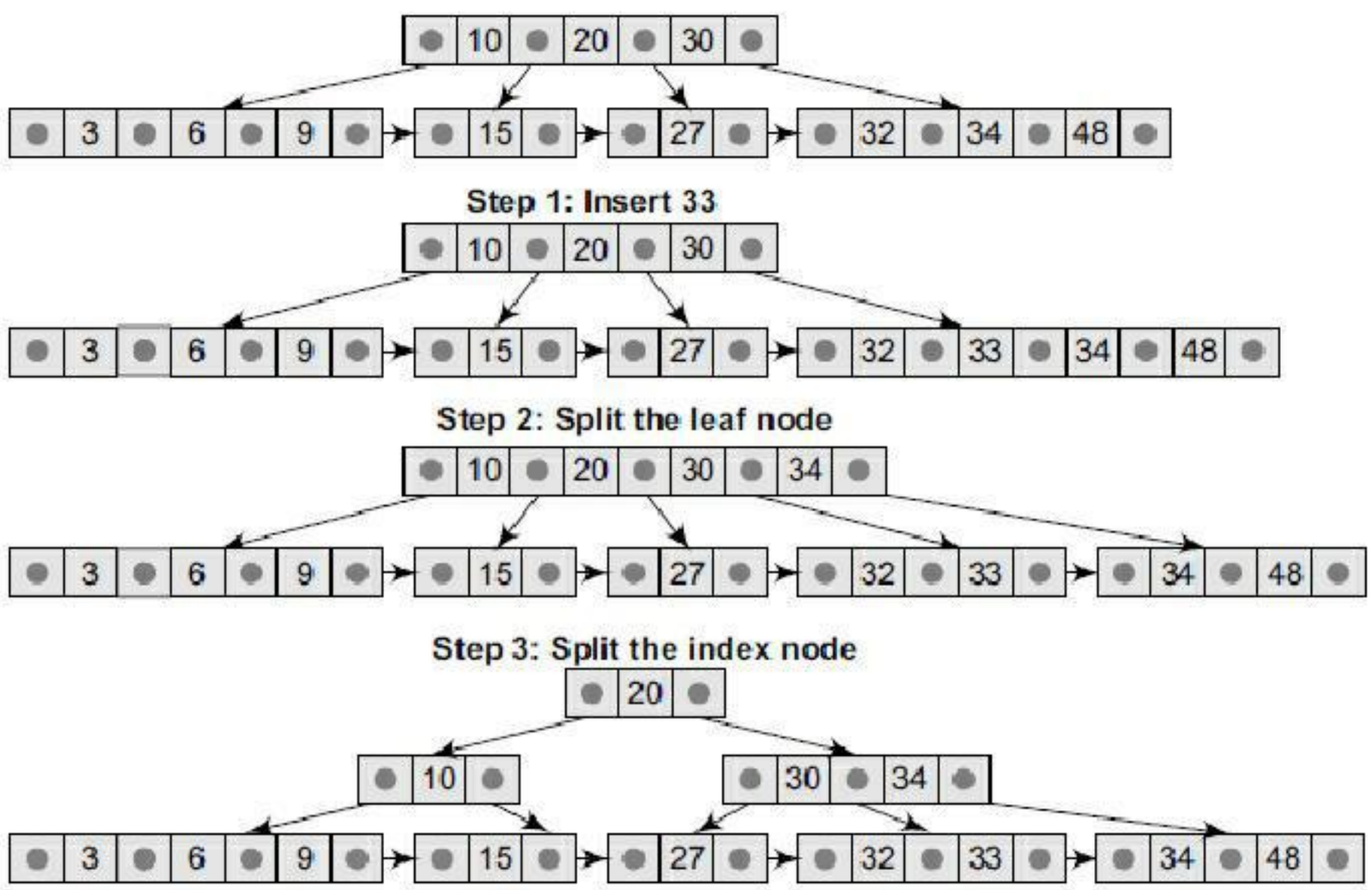


Figure 11.11 Inserting node 33 in the given B+ Tree

6.12.2 Deleting an Element from a B+ Tree

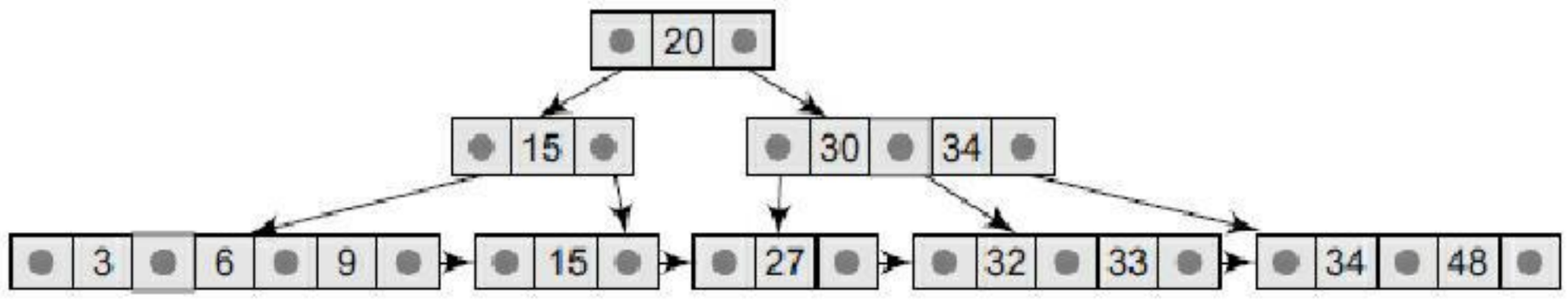
As in B trees, deletion is always done from a leaf node. If deleting a data element leaves that node empty, then the neighbouring nodes are examined and merged with the *underfull* node.

This process calls for the deletion of an index value from the parent index node which, in turn, may cause it to become empty. Similar to the insertion process, deletion may cause a merge-delete wave to run from a leaf node all the way up to the root. This leads to shrinking of the tree by one level. The steps to delete a node from a B+ tree are summarized in Fig. 11.12.

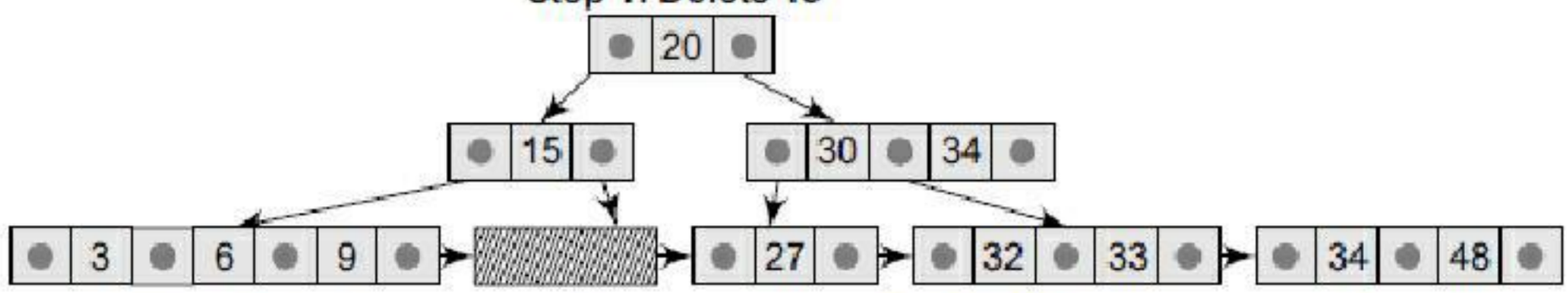
Step 1:	Delete the key and data from the leaves.
Step 2:	If the leaf node underflows, merge that node with the sibling and delete the key in between them.
Step 3:	If the index node underflows, merge that node with the sibling and move down the key in between them.

Figure 11.12 Algorithm for deleting a node from a B+ Tree

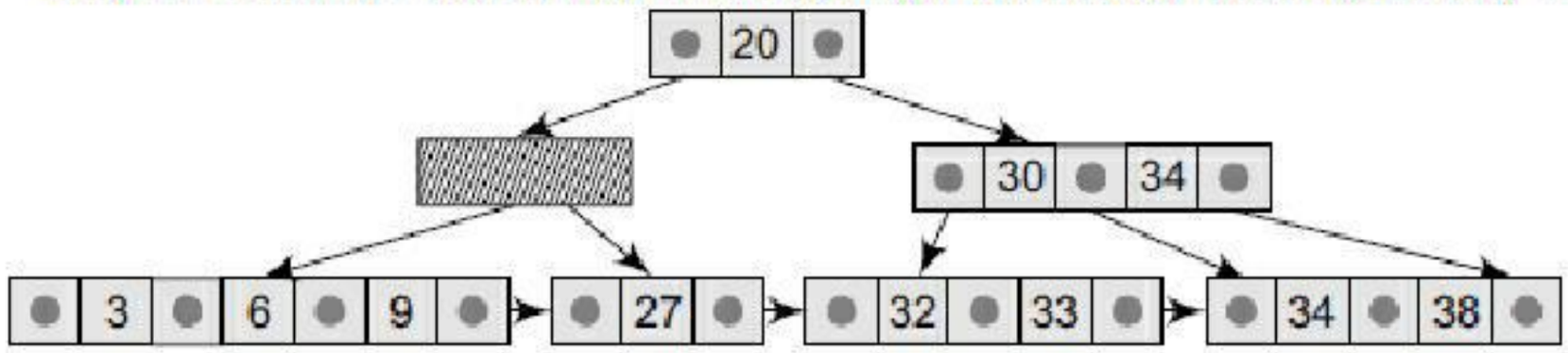
Example 11.6 Consider the B+ tree of order 4 given below and delete node 15 from it.



Step 1: Delete 15



Step 2: Leaf node underflows so merge with left sibling and remove key 15



Step 3: Now index node underflows, so merge with sibling and delete the node

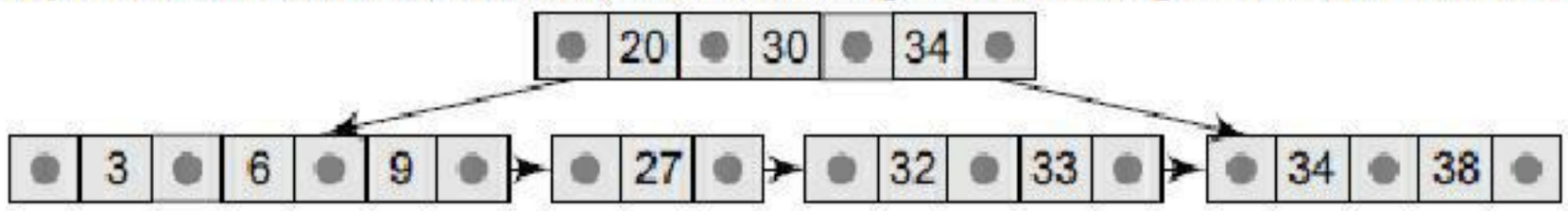


Figure 11.13 Deleting node 15 from the given B+ Tree

nareshp

6.13 HUFFMAN ALGORITHM

- Huffman coding is an entropy encoding algorithm developed by David A. Huffman that is widely used as a lossless data compression technique. The Huffman coding algorithm uses a variable length code table to encode a source character where the variable-length code table is derived on the basis of the estimated probability of occurrence of the source character.
- The key idea behind Huffman algorithm is that it encodes the most common characters using shorter strings of bits than those used for less common source characters.
- The algorithm works by creating a binary tree of nodes that are stored in an array. A node can be either a leaf node or an internal node. Initially, all the nodes in the tree are at the leaf level and store the source character and its frequency of occurrence (also known as weight).
  - Suppose there are 'n' weights w1, w2, --- , wn.
  - Take 2 minimum weights among the 'n' given inputs. Suppose w1, w2 are first two minimum, then the subtree will be
  - Now the remaining weights will be w3, w1 + w2, ---, wn.
  - Create all subtree till the last weight. Thus Huffman algorithms construct the tree from the bottom up approach rather than top down approach.
- **Technique**:

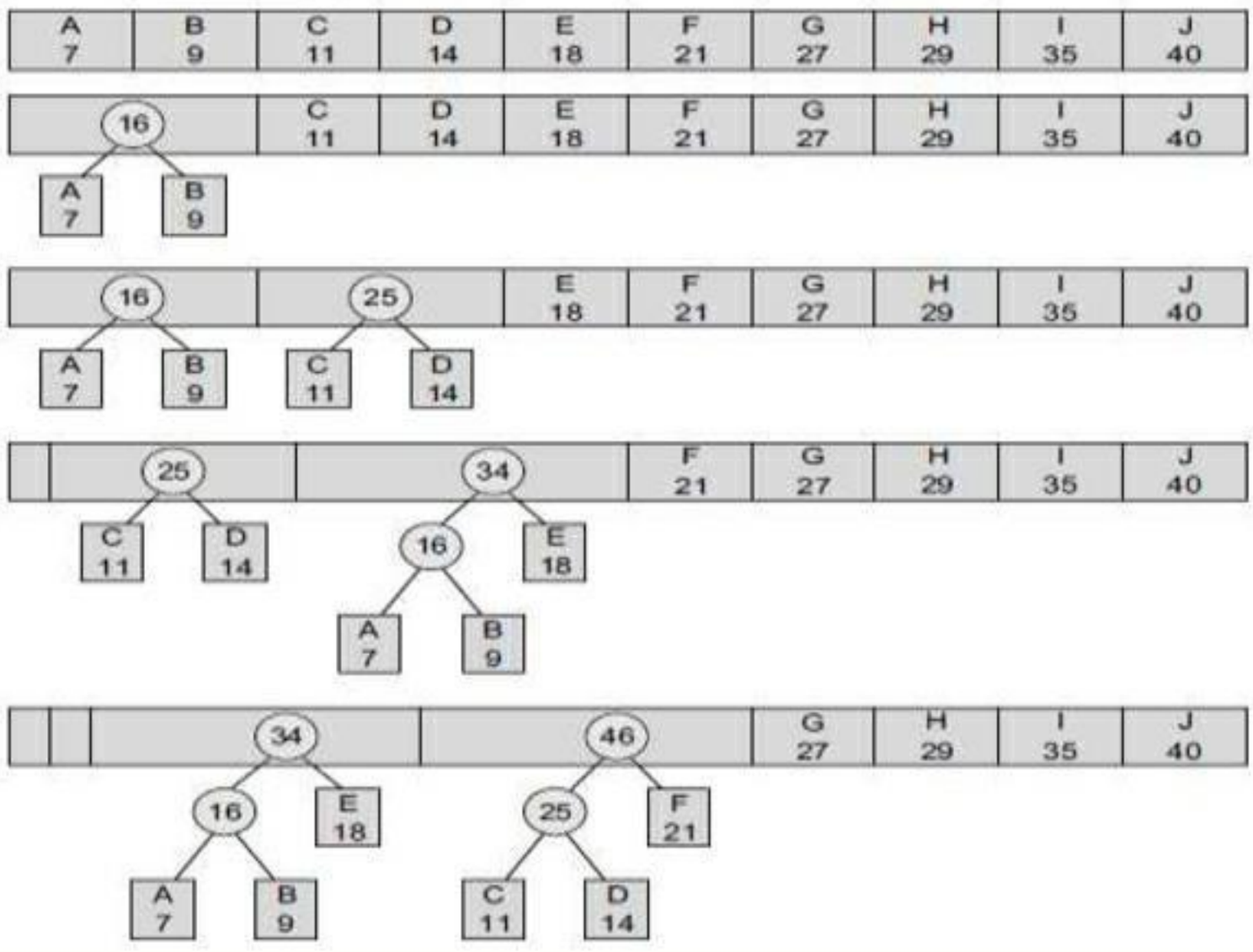
  Given n nodes and their weights, the Huffman algorithm is used to find a tree with a minimum weighted path length. The process essentially begins by creating a new node whose children are the two nodes with the smallest weight, such **that the new node's weight is equal to the sum of the children's weight. That is, the two nodes are merged into one** node. This process is repeated until the tree has only one node. Such a tree with only one node is known as the Huffman tree.

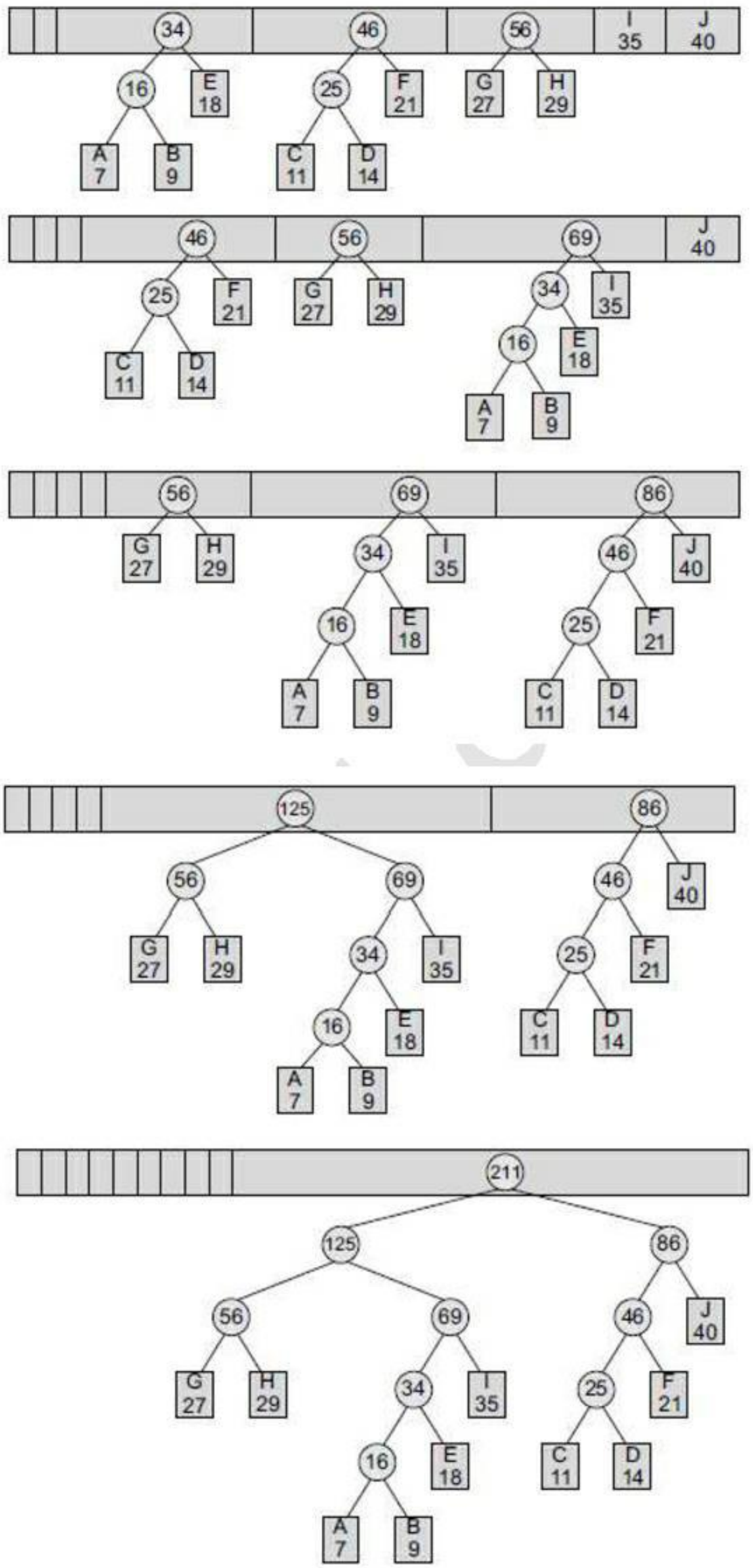
  The Huffman algorithm can be implemented using a priority queue in which all the nodes are placed in such a way that the node with the lowest weight is given the highest priority. The algorithm is shown in Figure A.

```
Step 1: Create a leaf node for each character. Add the character and its weight or frequency
        of occurrence to the priority queue.
Step 2: Repeat Steps 3 to 5 while the total number of nodes in the queue is greater than 1.
Step 3: Remove two nodes that have the lowest weight (or highest priority).
Step 4: Create a new internal node by merging these two nodes as children and with weight
        equal to the sum of the two nodes' weights.
Step 5: Add the newly created node to the queue.
```

Figure A: Huffman algorithm

<u>Example:</u> Create a Huffman tree with the following nodes arranged in a priority queue.




34
46
56
I 35
J 40
16
E 18
25
F 21
G 27
H 29
A 7
B 9
C 11
D 14
46
56
69
J 40
25
F 21
G 27
H 29
34
I 35
C 11
D 14
16
E 18
A 7
B 9
56
69
86
G 27
H 29
34
I 35
46
J 40
16
E 18
25
F 21
A 7
B 9
C 11
D 14
125
86
56
69
46
J 40
G 27
H 29
34
I 35
25
F 21
16
E 18
C 11
D 14
A 7
B 9
211
125
86
56
69
46
J 40
G 27
H 29
34
I 35
25
F 21
16
E 18
C 11
D 14
A 7
B 9

naresh.imp

Huffman Coding

In the Huffman tree, circles contain the cumulative weights of their child nodes. Every left branch is coded with 0 and every right branch is coded with 1. So, the characters A, E, R, W, X, Y, and Z are coded as shown in Table

Table Characters with their codes

Character	Code
A	00
E	01
R	11
W	1010
X	1000
Y	1001
Z	1011

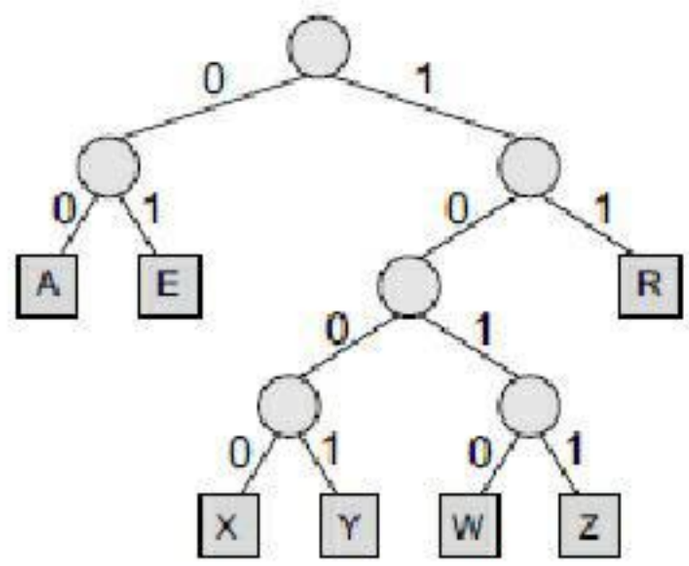


Figure Huffman tree

Thus, we see that frequent characters have a shorter code and infrequent characters have a longer code.

Example:

Huffman's Algorithm with Example

Algorithm: Suppose w_1 and w_2 are two minimum weights among the n given weights $w_1, w_2, \ldots, w_n$

Find a tree T' which gives a solution for the (n-1) weights

$w_1+w_2, w_3, w_4, \ldots, w_n$

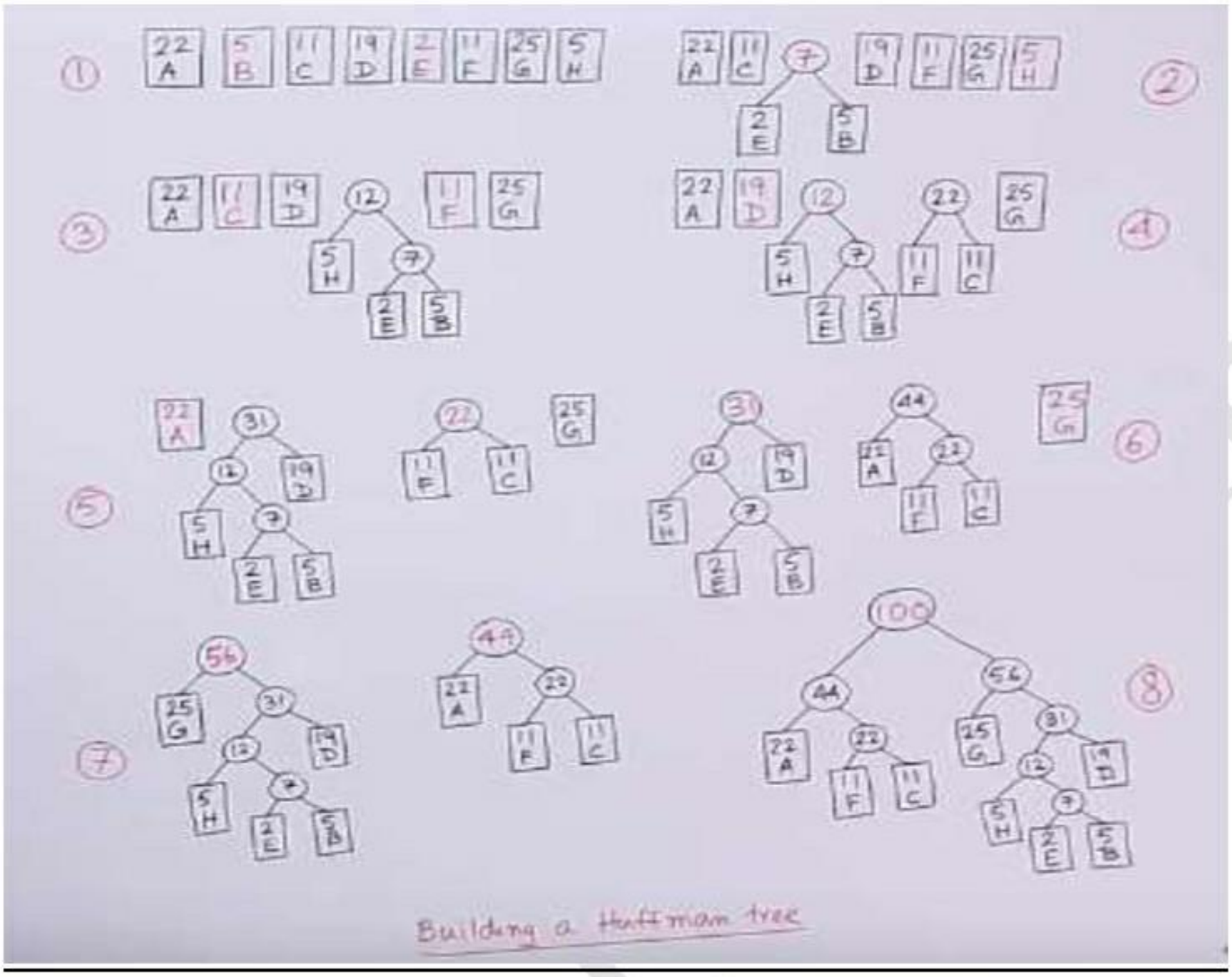
Then, in the tree T', replace the external node $\boxed{w_1+w_2}$ by the subtree

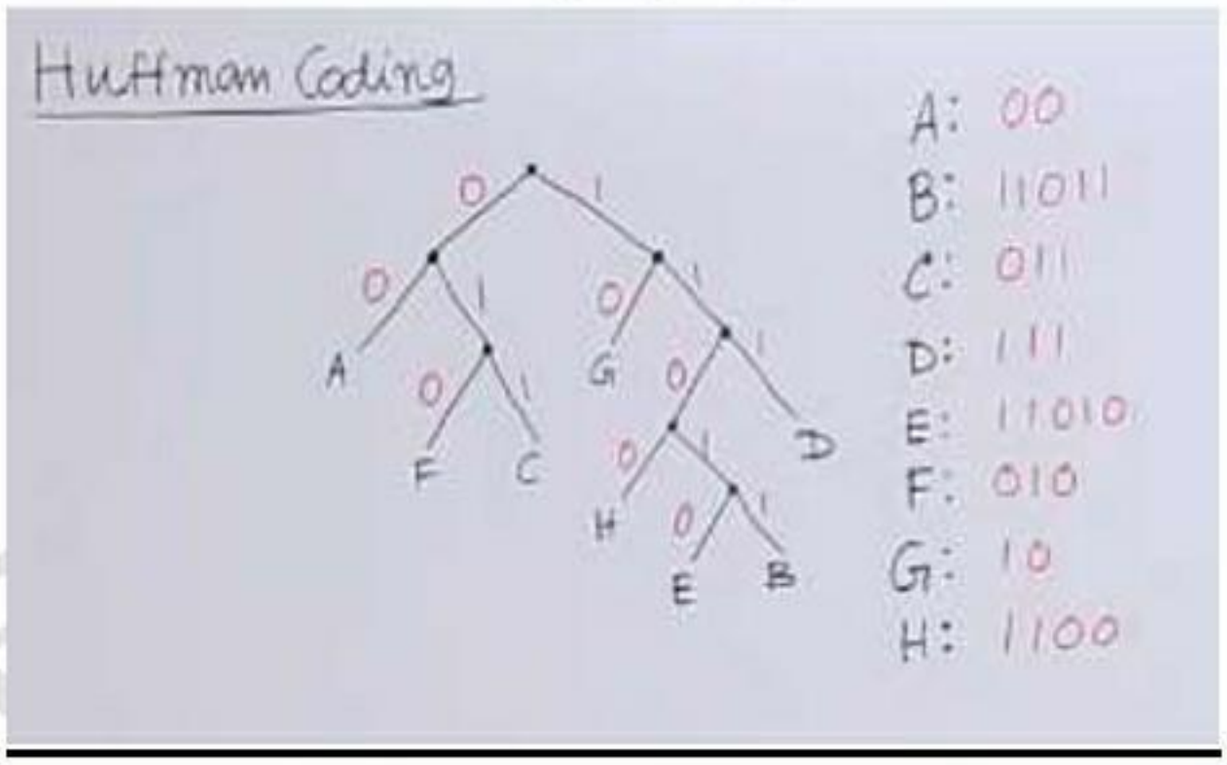
The new 2-tree T is the desired solution

Example: Suppose A, B, C, D, E, F, G and H are 8 data items, and they are assigned weights as follows:

Data items:	A	B	C	D	E	F	G	H
Weight :	22	5	11	19	2	11	25	5

Construct tree T with minimum-weighted path length using the above data and Huffman's algorithm


Building a Huffman tree


Huffman Coding
A: 00
B: 11011
C: 011
D: 111
E: 11010
F: 010
G: 10
H: 1100

naresh

6.14 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.

Figure A: Simple game of rock with 2 piles

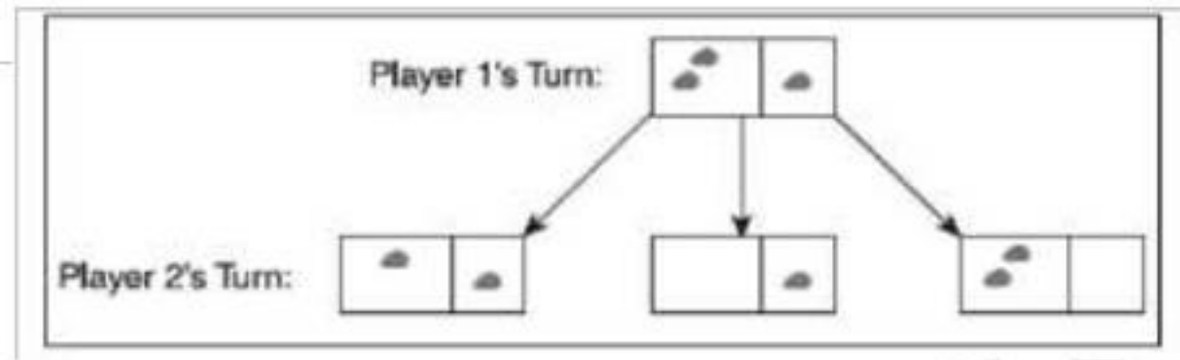


Figure B: First 2 levels of game tree

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 Figure B. **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 Figure A, 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.

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

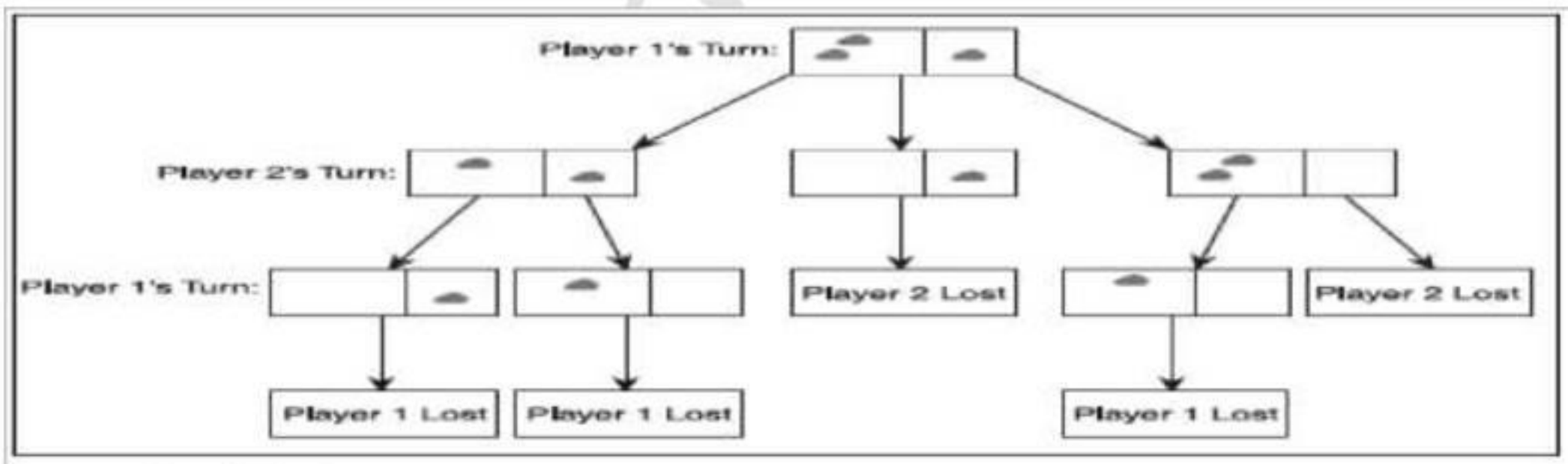


Figure C: A Complete Game tree

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.

6.15 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). Selfbalancing 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

nareshpd.com.np

7 Sorting

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. Sorting algorithms often have additional properties that are of interest, depending on the application.

Sorting algorithm: An algorithm that rearranges records in lists so that they follow some well-defined ordering relation on values of keys in each record.

Let a[n] be an array of n elements a0,a1,a2,a3.....,an-1 in memory. The sorting of the array a[n] means arranging the content of a[n] in either increasing or decreasing order.

i.e. a0<=a1<=a2<=a3<.=.....<=an-1

consider a list of values: 2 ,4 ,6 ,8 ,9 ,1 ,22 ,4 ,77 ,8 ,9

After sorting the values: 1, 2, 4, 4, 6, 8, 8,9 , 9 , 22, 77

In-place: The algorithm uses no additional array storage, and hence (other than perhaps the system's recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

- 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).

7.1 Bubble Sort

- Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts. This procedure of sorting is called **bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element** in the list will be placed at its proper position (i.e., at the end of the list).
- If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.
- Consider Following Example

Initially:

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

After Pass 1:

25	48	37	12	57	86	33	92
0	1	2	3	4	5	6	7

After Pass 2:

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

After Pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

After Pass 4:

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 5:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 6:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 7:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

Here, we notice that after each pass, an element is placed in its proper order and is not considered in succeeding passes. Furthermore, we need n – 1 passes to sort n elements.

Algorithm

Let A be a linear array of n numbers. **temp** 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. temp = A[j]
   ii. A[j] = A[j + 1]
   iii. A[j + 1] = temp
5. Display the sorted numbers of array A
6. Exit

Source Code

```
Void BubbleSort(int A[],int n)
{
        int i, j, temp;
        for(i = 0; i <n-1; i++)
        {
                for(j = 0; j < n-i-1; j++)
                {
                        if(A[j] > A[j+1])
                        {
                                temp = A[j];
                                A[j] = A[j+1];
                                A[j+1] = temp;
                        }
                }
        }
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + …….. +2 +1

$= O(n^2)$

There is no best-case linear time complexity for this algorithm.

Space Complexity:

Since no extra space besides 3 variables is needed for sorting

Space complexity = O(n)

Bubble Sort Optimization

Consider a case when the array is already sorted. In this situation no swapping is done but we still have to continue with all n–1 passes. We may even have an array that will be sorted in 2 or 3 passes but we still have to continue with rest of the passes. So once we have detected that the array is sorted, the algorithm must not be executed further. This is the optimization over the original bubble sort algorithm. In order to stop the execution of further passes after the array is sorted, we can have a variable flag which is set to FALSE before each pass and is made TRUE when a swapping is performed. The code for the optimized bubble sort can be given as:

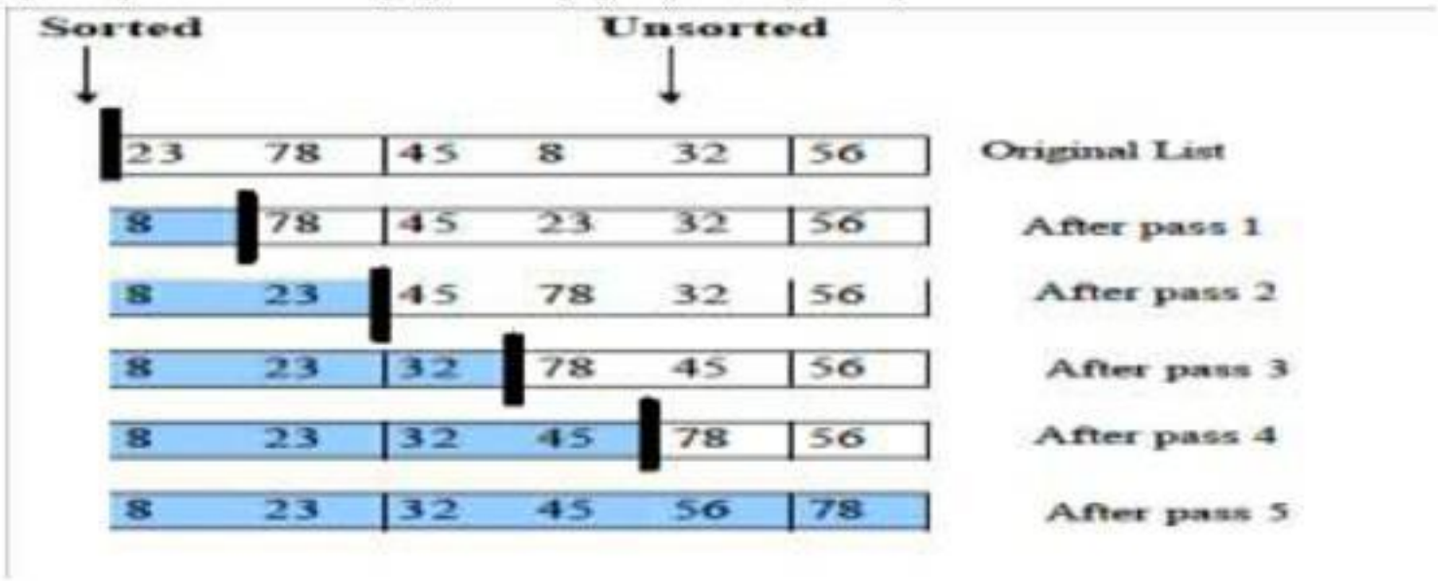
```
void BubbleSort(int A[],int n)
{
        int i, j, temp,flag = 0;
        for(i = 0; i <n-1; i++)
        {
                for(j = 0; j < n-i-1; j++)
                {
                        if(A[j] > A[j+1])
                        {
                                flag = 1;
                                temp = A[j];
                                A[j] = A[j+1];
                                A[j+1] = temp;
                        }
                }
                if(flag == 0)           // array is sorted
                        return;
        }
}
```

Complexity of Optimized Bubble Sort Algorithm

In the best case, when the array is already sorted, the optimized bubble sort will take O(n) time. In the worst case, when all the passes are performed, the algorithm will perform slower than the original algorithm. In average case also, the performance will see an improvement. Compare it with the complexity of original bubble sort algorithm which takes O(n2) in all the cases.

7.2 Selection Sort

- **Idea:** Find the least (or greatest) value in the array, swap it into the leftmost (or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly. Let a[n] be a linear array of n elements.
- First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,
  - In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
  - In Pass 2, find the position POS of the smallest value in sub-array of N–1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.
  - In Pass N–1, find the position POS of the smaller of the elements ARR[N–2] and ARR[N–1]. Swap ARR[POS] and ARR[N–2] so that ARR[0], ARR[1], ..., ARR[N–1] is sorted.



naresh.com.np

Algorithm

Let A be a linear array of n numbers. **temp** be a temporary variable for swapping (or interchanging). **pos** 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)
   pos = i
3. Initialize j = i + 1 and repeat through step 4 if (j < n)
4. if (A[pos] > A [j])
   pos = j
5.
   i. temp = A[pos]
   ii. A[pos] = A [i]
   iii. A [i] = temp
6. Display "the sorted numbers of array A"
7. Exit

Source Code

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

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + …….. +2 +1

= $O(n^2)$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

Space complexity = O(n)

7.3 Insertion Sort

- Insertion sort works by repeatedly taking an element from the unsorted portion of a list and inserting it into the sorted portion of the list until every element has been inserted.
- We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.
- The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

Technique

Insertion sort works as follows:

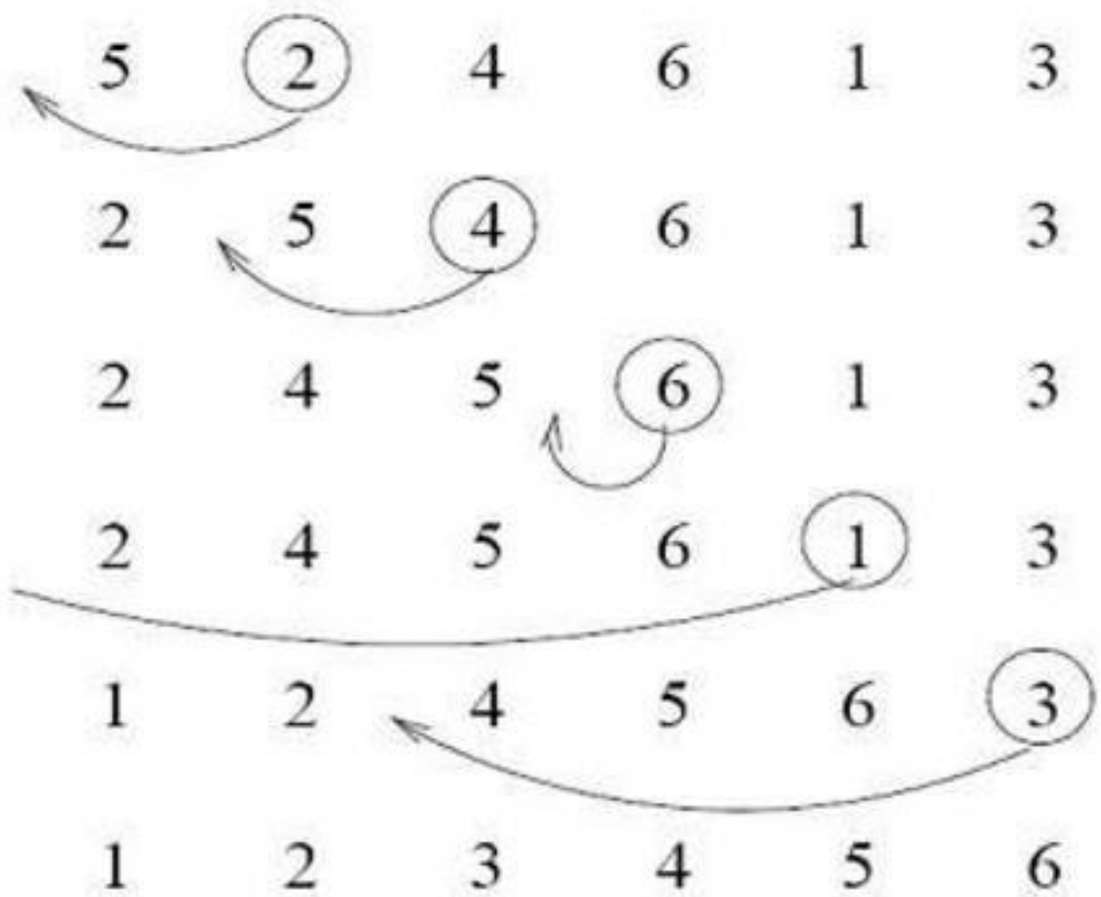
- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.

naresh.com.np

- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Suppose an array a[n] with n elements. The insertion sort works as follows:

Pass 1: a[0] by itself is trivially sorted.
Pass 2: a[1] is inserted either before or after a[0] so that a[0], a[1] is sorted.
Pass 3: a[2] is inserted into its proper place in a[0],a[1] that is before a[0], between a[0] and a[1], or after a[1] so that a[0],a[1],a[2] is sorted.
……
Pass N: a[n-1] is inserted into its proper place in a[0],a[1],a[2],.....,a[n-2] so that a[0],a[1],a[2],.....,a[n-1] is sorted with n elements.



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 Repeat Steps 3 to 5 for i = 1 to N – 1
3. temp = A [i],
   Pos = i – 1
4. Repeat the step 3 while(temp < A[Pos] and (Pos >= 0))
   A [Pos+1] = A [Pos]
   Pos = Pos-1
5. A [Pos +1] = temp
6. Exit

Source Code

```
void insertion_sort(int A[], int n)
{
        int i,pos,temp;
        for(i=1;i<n;i++)
        {
                temp = A[i];                    /*temp is to be inserted at proper place*/
                pos = i-1;
                while(temp<A[pos] && pos >= 0)
                {
```

```
                    A[pos+1]= A[pos];
                    pos = pos -1;
                }
                A[pos+1]= temp;
            }
        }
```

- **Time Complexity**

Worst Case Analysis:
Array elements are in reverse sorted order
Inner loop executes for 1 times when i=1, 2 times when i=2... and n-1 times when i=n-1:
Time complexity = 1 + 2 + 3 + ... +(n-2) +(n-1)
= $O(n^2)$

Best case Analysis:
Array elements are already sorted
Inner loop executes for 1 times when i=1, 1 times when i=2... and 1 times when i=n-1:
Time complexity = 1 + 2 + 3 + ... +1 +1
= O(n)

- **Space Complexity**

Since no extra space besides 5 variables is needed for sorting
Space Complexity = O(n)

- **Advantages of Insertion Sort**

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- it requires less memory space (only O(1) of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

7.4 SHELL SORT

Shell sort, invented by Donald Shell in 1959, is a sorting algorithm that is a generalization of insertion sort. While discussing insertion sort, we have observed two things:

- **First, insertion sort works well when the input data is 'almost sorted'.**
- Second, insertion sort is quite inefficient to use as it moves the values just one position at a time.

Shell sort is considered an improvement over insertion sort as it compares elements separated by a gap of several positions. This enables the element to take bigger steps towards its expected position. In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes. However, the last step of shell sort is a plain insertion sort. But by the time we reach the last step, the elements are already **'almost sorted', and hence it provides good performance. If we** take a scenario in which the smallest element is stored in the other end of the array, then sorting such an array with either bubble sort or insertion sort will execute in $O(n^2)$ time and take roughly n comparisons and exchanges to move this value all the way to its correct position. On the other hand, Shell sort first moves small values using giant step sizes, so a small value will move a long way towards its final position, with just a few comparisons and exchanges.

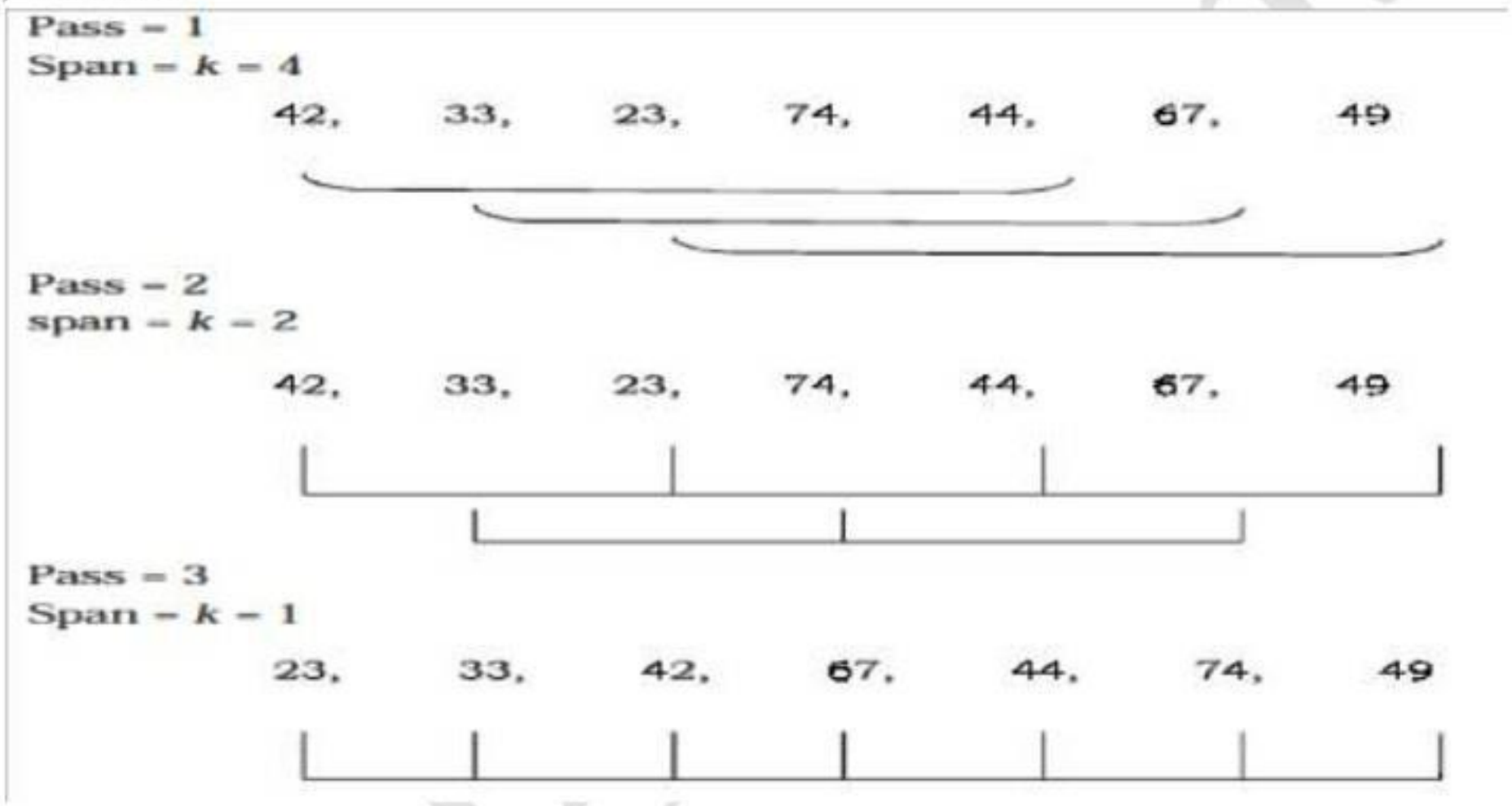
Technique

- To visualize the way in which shell sort works, perform the following steps:
  Step 1: Arrange the elements of the array in the form of a table and sort the columns (using insertion sort).
  Step 2: Repeat Step 1, each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.
  Note that we are only visualizing the elements being arranged in a table, the algorithm does its sorting in-place.

- 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 for shell short

```
Shell_Sort(Arr, n)
    Step 1: SET FLAG = 1, GAP_SIZE = N
    Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
    Step 3: SET FLAG = 0
    Step 4: SET GAP_SIZE = (GAP_SIZE + 1) / 2
    Step 5: Repeat Step 6 for I = to I < (N - GAP_SIZE)
    Step 6: IF Arr[I + GAP_SIZE] > Arr[I]
                SWAP Arr[I + GAP_SIZE], Arr[I]
                SET FLAG = 0
    Step 7: END
```

In the algorithm, we sort the elements of the array Arr in multiple passes. In each pass, we reduce the gap_size (visualize it as the number of columns) by a factor of half as done in Step 4. In each iteration of the for loop in Step 5, we compare the values of the array and interchange them if we have a larger value preceding the smaller one.

Example: Sort the elements given below using shell sort.

63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33

Solution

Arrange the elements of the array in the form of a table and sort the columns.

63	19	7	90	81	36	54	45
72	27	22	9	41	59	33	

Result:

63	19	7	9	41	36	33	45
72	27	22	90	81	59	54	

The elements of the array can be given as:

63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54

Repeat Step 1 with smaller number of long columns.

63	19	7	9	41
36	33	45	72	27
22	90	81	59	54

Result:

22	19	7	9	27
36	33	45	59	41
63	90	81	72	54

The elements of the array can be given as:

22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54

Repeat Step 1 with smaller number of long columns.

22	19	7
9	27	36
33	45	59
41	63	90
81	72	54

Result:

9	19	7
22	27	36
33	45	54
41	63	59
81	72	90

The elements of the array can be given as:

9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90

Finally, arrange the elements of the array in a single column and sort the column.

	Result:
9	7
19	9
7	19
22	22
27	27
36	33
33	36
45	41
54	45
41	54
63	59
59	63
81	72
72	81
90	90

Finally, the elements of the array can be given as:

7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90

Program to implement Shell Sort

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int arr[10];
        int i, j, n, flag = 1, gap_size, temp;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter %d numbers: ",n); // n was added
        for(i=0;i<n;i++)
                scanf("%d", &arr[i]);
        gap_size = n;
        while(flag == 1 || gap_size > 1)
        {
                flag = 0;
                gap_size = (gap_size + 1) / 2;
```

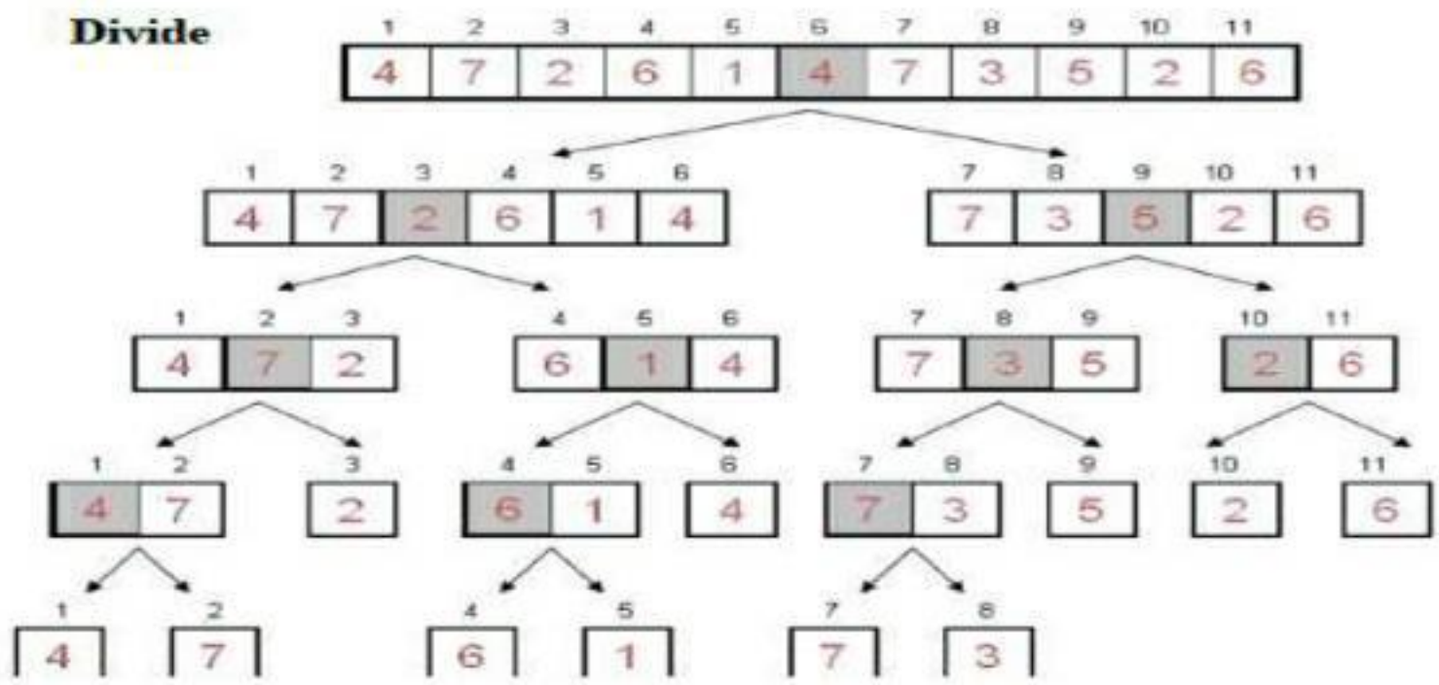
```
        for(i=0; i< (n - gap_size); i++)
        {
                if( arr[i+gap_size] < arr[i])
                {
                        temp = arr[i+gap_size];
                        arr[i+gap_size] = arr[i];
                        arr[i] = temp;
                        flag = 0;
                }
        }
    }
    printf("\n The sorted array is: \n");
    for(i=0;i<n;i++)
            printf(" %d\t", arr[i]);
    getch();
    return 0;
}
```

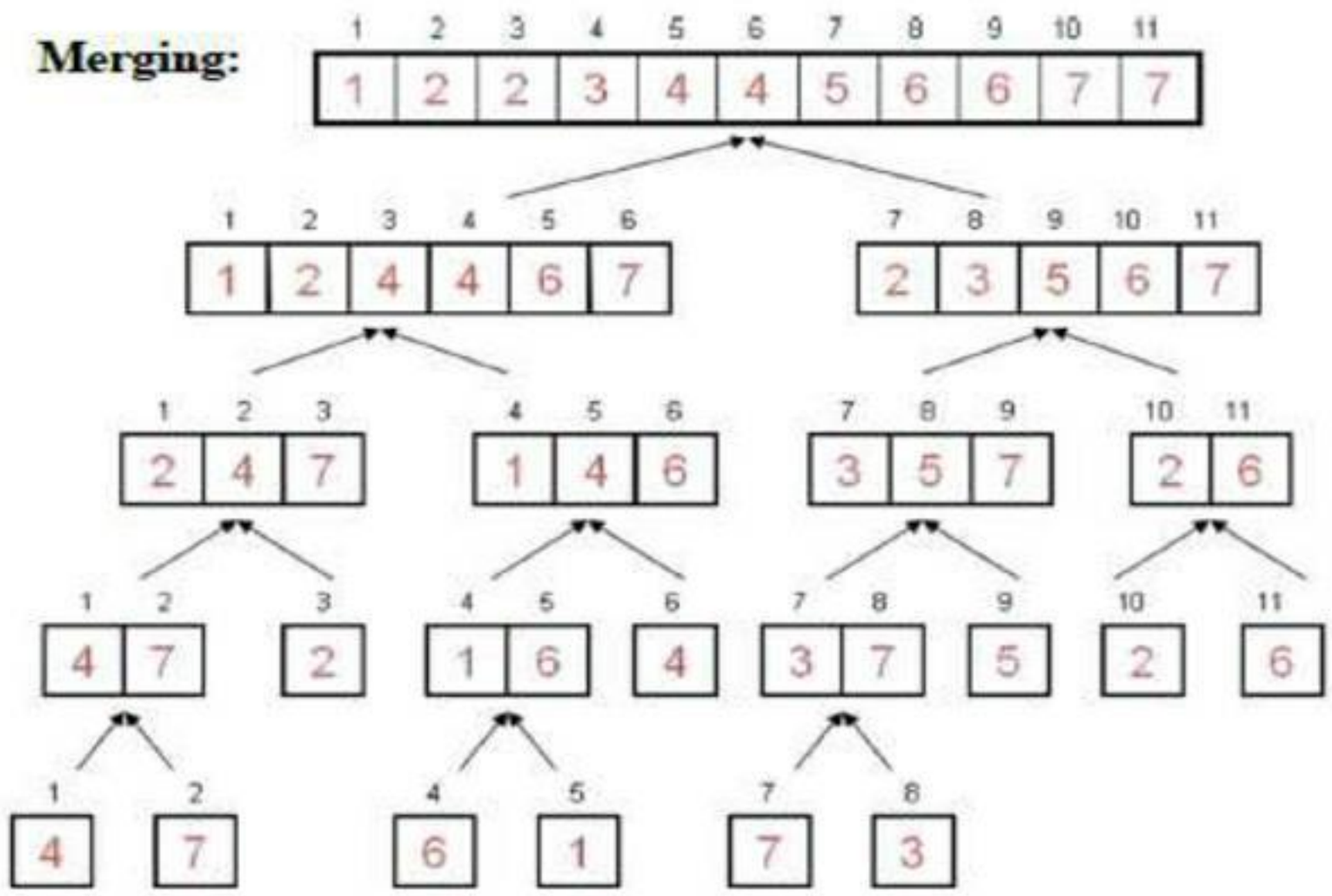
7.5 MERGE SORT

- Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.
  Divide means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.
  Conquer means sorting the two sub-arrays recursively using merge sort.
  Combine means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.
- Merge sort algorithm focuses on two main concepts to improve its performance (running time):
  - A smaller list takes fewer steps and thus less time to sort than a large list.
  - As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.
- The basic steps of a merge sort algorithm are as follows:
  - If the array is of length 0 or 1, then it is already sorted.
  - Otherwise, divide the unsorted array into two sub-arrays of about half the size.
  - Use merge sort algorithm recursively to sort each sub-array.
  - Merge the two sub-arrays to form a single sorted list.

nareshpd.com.np

Example: a[]={4, 7, 2, 6, 1, 4, 7, 3, 5, 2, 6}





Algorithm for merge sort

```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                SET TEMP[INDEX] = ARR[I]
                SET I = I + 1
            ELSE
                SET TEMP[INDEX] = ARR[J]
                SET J = J + 1
            [END OF IF]
            SET INDEX = INDEX + 1
        [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IF I > MID
                Repeat while J <= END
                    SET TEMP[INDEX] = ARR[J]
                    SET INDEX = INDEX + 1, SET J = J + 1
                [END OF LOOP]
        [Copy the remaining elements of left sub-array, if any]
            ELSE
                Repeat while I <= MID
                    SET TEMP[INDEX] = ARR[I]
                    SET INDEX = INDEX + 1, SET I = I + 1
                [END OF LOOP]
            [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
                SET ARR[K] = TEMP[K]
                SET K = K + 1
        [END OF LOOP]
Step 6: END
```

```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)
            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 2: END
```

naresh

Program to implement merge sort

```
#include <stdio.h>
#include <conio.h>
#define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
                scanf("%d", &arr[i]);
        merge_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
                printf(" %d\t", arr[i]);
        getch();
}
void merge(int arr[], int beg, int mid, int end)
{
        int i=beg, j=mid+1, index=beg, temp[size], k;
        while((i<=mid) && (j<=end))
        {
                if(arr[i] < arr[j])
                {
                        temp[index] = arr[i];
                        i++;
                }
                else
                {
                        temp[index] = arr[j];
                        j++;
                }
                index++;
        }
        if(i>mid)
        {
                while(j<=end)
                {
                        temp[index] = arr[j];
                        j++;
                        index++;
                }
        }
        else
        {
                while(i<=mid)
                {
                        temp[index] = arr[i];
                        i++;
                        index++;
                }
```

nareshpad.com.np

```
        }
        for(k=beg;k<index;k++)
                arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
        int mid;
        if(beg<end)
        {
                mid = (beg+end)/2;
                merge_sort(arr, beg, mid);
                merge_sort(arr, mid+1, end);
                merge(arr, beg, mid, end);
        }
}
```

- **Time Complexity:**

  Recurrence Relation for Merge sort:

  $T(n) = 1$ if $n=1$

  $T(n) = 2\,T(n/2) + O(n)$ if $n>1$

  Solving this recurrence we get

  $T(n) = O(n\log n)$

- **Space Complexity:**

  It uses one extra array and some extra variables during sorting, therefore

  Space Complexity= $2n + c = O(n)$

7.6 QUICK SORT

- Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes O(n log n) comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as O(n2). Basically, the quick sort algorithm is faster than other O(nlog n) algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.
- Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.
- The quick sort algorithm works as follows:
  1. Select an element pivot from the array elements.
  2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the partition operation.
  3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)
- Like merge sort, the base case of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.
- Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

- **Technique**

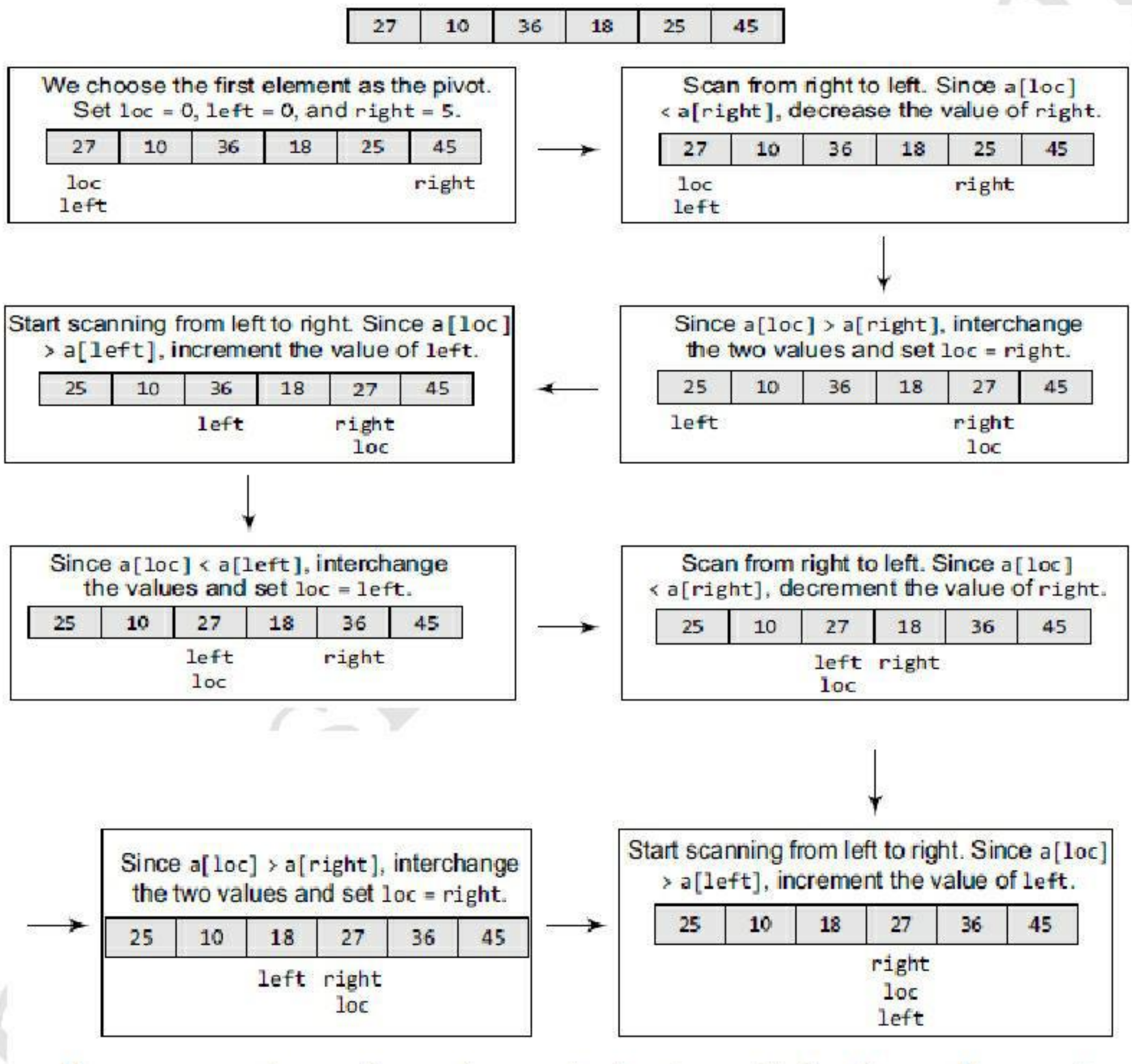
Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable. That is, loc = 0, left = 0, and right = n−1 (where n in the number of elements in the array)
2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, a[loc] should be less than a[right].

a) If that is the case, then simply continue comparing until right becomes equal to loc. Once right = loc, it means the pivot has been placed in its correct position.
b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.
c) Set loc = right

3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc. That is, a[loc] should be greater than a[left].
   a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position.
   b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.
   c) Set loc = left.

<u>Example:</u> Sort the elements given in the following array using quick sort algorithm



Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

Algorithm for Quick sort

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
                SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
                SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
                SWAP ARR[LOC] with  ARR[RIGHT]
                SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
                Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
                SET LEFT = LEFT + 1
                [END OF LOOP]
Step 6:         IF LOC = LEFT
                        SET FLAG = 1
                ELSE IF ARR[LOC] < ARR[LEFT]
                        SWAP ARR[LOC] with  ARR[LEFT]
                        SET LOC = LEFT
                [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Time Complexity:

Best Case:

Divides the array into two partitions of equal size, therefore
$T(n) = 2T(n/2) + O(n)$, Solving this recurrence we get,
$T(n)=O(n\log n)$

Worst case:

when one partition contains the n-1 elements and another partition contains only one element.
Therefore its recurrence relation is:
$T(n) = T(n-1) + O(n)$, Solving this recurrence we get
$T(n)=O(n2)$

Average case:

Good and bad splits are randomly distributed across throughout the tree
$T1(n)= 2T'(n/2) + O(n)$ Balanced
$T'(n)= T(n-1) + O(n)$ Unbalanced
Solving:
$B(n)= 2(B(n/2-1) + \Theta(n/2)) + \Theta(n)$
$= 2B(n/2-1) + \Theta(n)$
$= O(n\log n)$
$\Rightarrow T(n)=O(n\log n)$

Pros and Cons of Quick Sort

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.

Program to implement quick sort algorithm

```
#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
int main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
                scanf("%d", &arr[i]);
        quick_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
                printf(" %d\t", arr[i]);
        getch();
        return 0;
}

int partition(int a[], int beg, int end)
{
        int left, right, temp, loc, flag;
        loc = left = beg;
        right = end;
        flag = 0;
        while(flag != 1)
        {
                while((a[loc] <= a[right]) && (loc!=right))
                        right--;
                if(loc==right)
                        flag =1;
                else if(a[loc]>a[right])
                {
                        temp = a[loc];
                        a[loc] = a[right];
                        a[right] = temp;
                        loc = right;
                }
                if(flag!=1)
                {
                        while((a[loc] >= a[left]) && (loc!=left))
                                left++;
                        if(loc==left)
                                flag =1;
                        else if(a[loc] <a[left])
                        {
                                temp = a[loc];
                                a[loc] = a[left];
                                a[left] = temp;
                                loc = left;
                        }
```

naresh.com.np

```
                }
        }
        return loc;
}
void quick_sort(int a[], int beg, int end)
{
        int loc;
        if(beg<end)
        {
                loc = partition(a, beg, end);
                quick_sort(a, beg, loc-1);
                quick_sort(a, loc+1, end);
        }
}
```

7.7 RADIX SORT

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on. During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the nth pass, where n is the length of the name with maximum number of letters. After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on. When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least **significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9)** and the number of passes will depend on the length of the number having maximum number of digits.

Algorithm for Radix Sort

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:            SET I = 0 and INITIALIZE buckets
Step 6:            Repeat Steps 7 to 9 while I<N-1
Step 7:                  SET DIGIT  = digit at PASSth place in A[I]
Step 8:                  Add A[I] to the bucket numbered DIGIT
Step 9:                  INCEREMENT bucket count for bucket numbered DIGIT
                   [END OF LOOP]
Step 10:           Collect the numbers in the bucket
        [END OF LOOP]
Step 11: END
```

Example: Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654 924					654 924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result.

After the third pass, the list can be given as

123, 345, 472, 555, 567, 654, 808, 911, 924.

Complexity of Radix Sort

To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes O(kn) time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time.

Pros and Cons of Radix Sort

Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters. But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters. Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task. Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.

Program to implement radix sort algorithm

```
#include <stdio.h>
#include <conio.h>
#define size 10
int largest(int arr[], int n);
void radix_sort(int arr[], int n);
int main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
                scanf("%d", &arr[i]);
        radix_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
                printf(" %d\t", arr[i]);
        getch();
}
int largest(int arr[], int n)
{
        int large=arr[0],i;
        for(i=1;i<n;i++)
        {
                if(arr[i]>large)
                        large = arr[i];
        }
        return large;
}
void radix_sort(int arr[], int n)
{
        int bucket[size][size], bucket_count[size];
        int i, j, k, remainder, NOP=0, divisor=1, large, pass;
        large = largest(arr, n);
        while(large>0)
        {
                NOP++;
                large/=size;
        }
        for(pass=0;pass<NOP;pass++) // Initialize the buckets
        {
                for(i=0;i<size;i++)
                        bucket_count[i]=0;
                for(i=0;i<n;i++)
                {
                        // sort the numbers according to the digit at passth place
                        remainder = (arr[i]/divisor)%size;
                        bucket[remainder][bucket_count[remainder]] = arr[i];
                        bucket_count[remainder] += 1;
                }
                // collect the numbers after PASS pass
                i=0;
                for(k=0;k<size;k++)
```

nareshpad.com.np

```
                {
                        for(j=0;j<bucket_count[k];j++)
                        {
                                arr[i] = bucket[k][j];
                                i++;
                        }
                }
                divisor *= size;
        }
}
```

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]

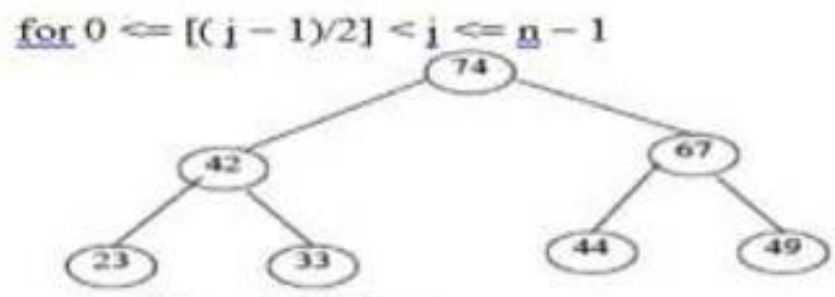


Figure 1: Heap

74	42	67	23	33	44	49

Figure 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**.

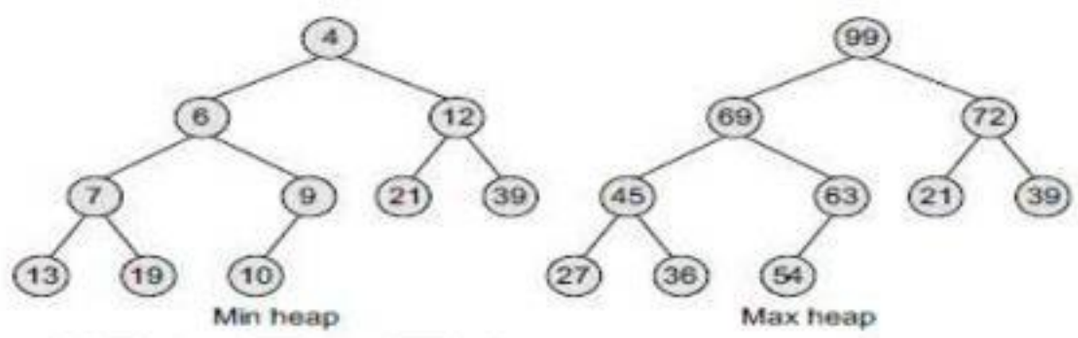


Figure A

7.9 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.

<u>Inserting an element in to a heap</u>

Consider the heap H in Figure 1. 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 Figure 3. Now we have to find the appropriate place for 55 in the heap by rearranging it.

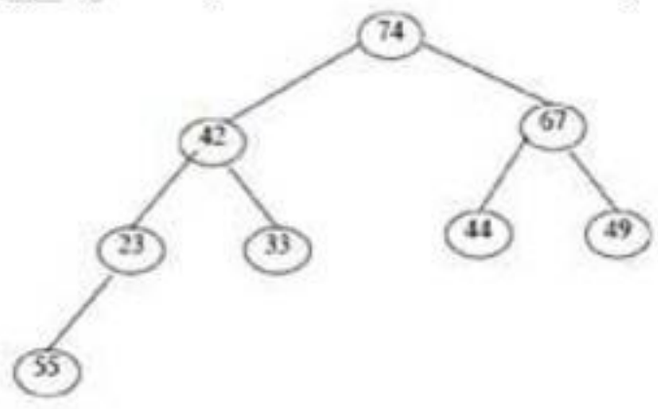


Figure 3

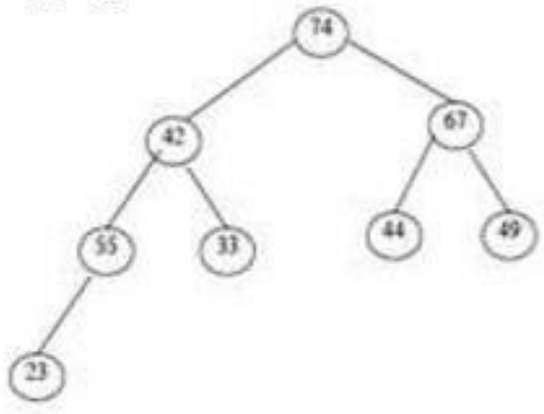


Figure 4

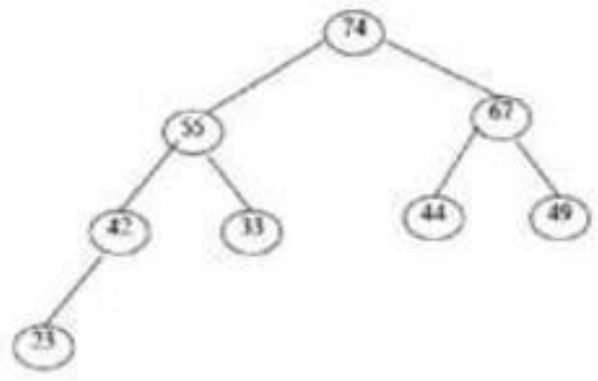


Figure 5

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 Figure 4.

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 Figure 5.

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 Figure 5 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 Figure 6.

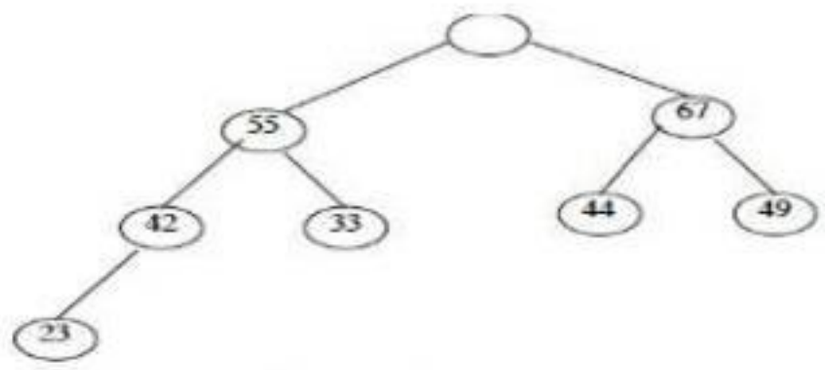


Figure 6

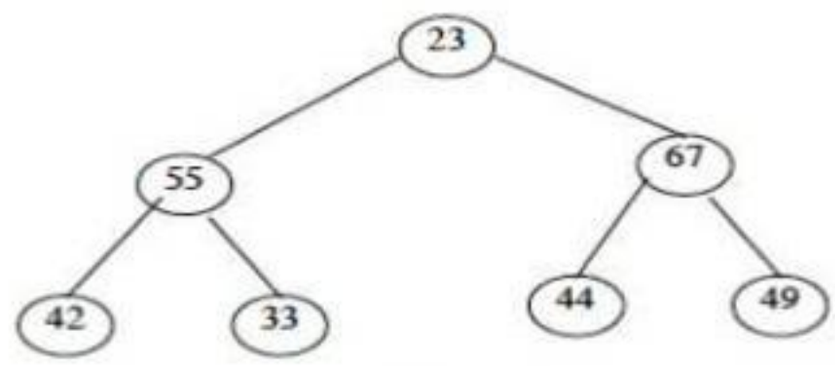


Figure 7

Replace the deleted root node R by the last node L as shown in the Figure 7.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 Figure 8. Again compare 23 with its new two children, 44 and 49. Since 23 is less than the largest child 49, interchange 23 and 49 as shown in Figure 9.

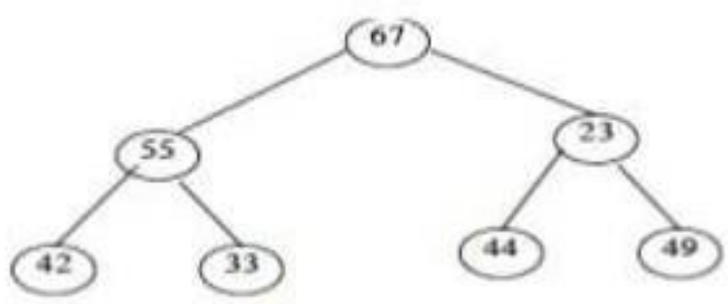


Figure 8

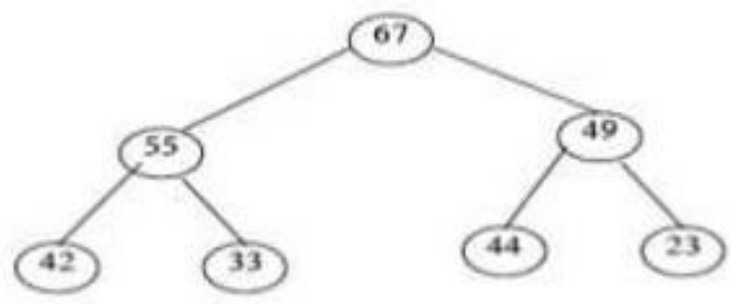


Figure 9

The Figure 9 is the required heap H without its original root R.

Algorithm for heap sort

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

Complexity of Heap Sort

Heap sort uses two heap operations: insertion and root deletion. Each element extracted from the root is placed in the last empty location of the array. In phase 1, when we build a heap, the number of comparisons to find the right location of the
new element in H cannot exceed the depth of H. Since H is a complete tree, its depth cannot exceed m, where m is the number of elements in heap H. Thus, the total number of comparisons g(n) to insert n elements of ARR in H is bounded as:

$$g(n) <= n \log n$$

Hence, the running time of the first phase of the heap sort algorithm is O(n log n). In phase 2, we have H which is a complete tree with m elements having left and right sub-trees as heaps. Assuming L to be the root of the tree, reheaping the tree would need 4 comparisons to move L one step down the tree H. Since the depth of H cannot exceed O(log m), reheaping the tree
will require a maximum of 4 log m comparisons to find the right location of L in H.
Since n elements will be deleted from heap H, reheaping will be done n times. Therefore, the number of comparisons to delete n elements is bounded as:

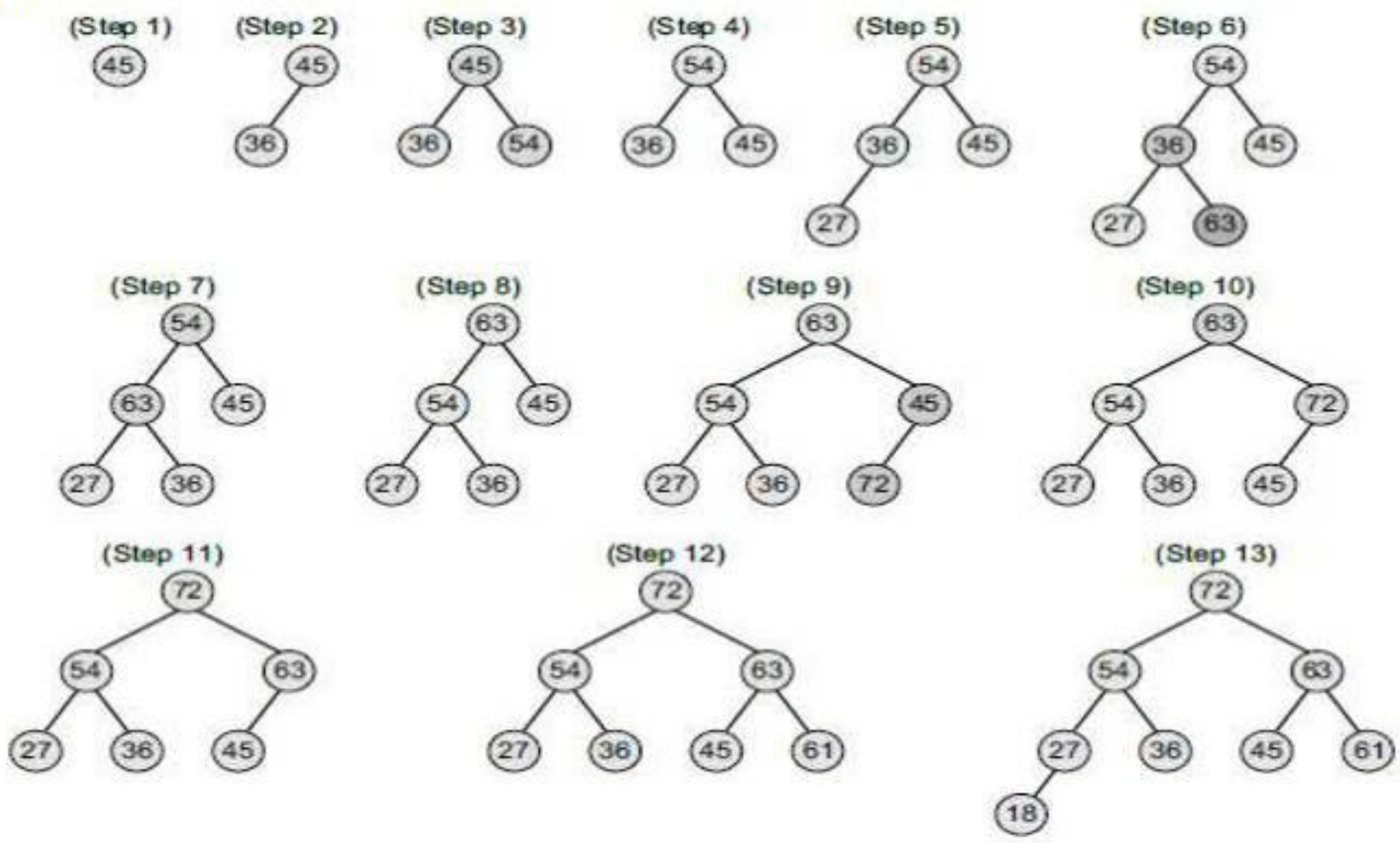
$$h(n) <= 4n \log n$$

Hence, the running time of the second phase of the heap sort algorithm is O(n log n). Each phase requires time proportional to O(n log n). Therefore, the running time to sort an array of n elements in the worst case is proportional to O(n log n).

Therefore, we can conclude that heap sort is a simple, fast, and stable sorting algorithm that can be used to sort large sets of data efficiently.

Example: Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18.

Solution



Program to implement heap sort algorithm

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
void RestoreHeapUp(int *,int);
void RestoreHeapDown(int*,int,int);
```

```
int main()
{
        int Heap[MAX],n,i,j;
        printf("\n Enter the number of elements : ");
        scanf("%d",&n);
        printf("\n Enter the elements : ");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&Heap[i]);
                RestoreHeapUp(Heap, i); // Heapify
        }
        // Delete the root element and heapify the heap
        j=n;
        for(i=1;i<=j;i++)
        {
                int temp;
                temp=Heap[1];
                Heap[1]= Heap[n];
                Heap[n]=temp;
                n = n-1; // The element Heap[n] is supposed to be deleted
                RestoreHeapDown(Heap,1,n); // Heapify
        }
        n=j;
        printf("\n The sorted elements are: ");
        for(i=1;i<=n;i++)
                printf("%4d",Heap[i]);
        return 0;
}
void RestoreHeapUp(int *Heap,int index)
{
        int val = Heap[index];
        while( (index>1) && (Heap[index/2] < val) ) // Check parent's value
        {
                Heap[index]=Heap[index/2];
                index /= 2;
        }
        Heap[index]=val;
}
void RestoreHeapDown(int *Heap,int index,int n)
{
        int val = Heap[index];
        int j=index*2;
        while(j<=n)
        {
                if( (j<n) && (Heap[j] < Heap[j+1]))// Check sibling's value
                        j++;
                if(Heap[j] < Heap[j/2]) // Check parent's value
                        break;
                Heap[j/2]=Heap[j];
                j=j*2;
        }
        Heap[j/2]=val;
}
```

naresh.com.np

7.10 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 counterpart 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

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

- **Program to implement Exchange sort**

```
#include<stdio.h>
#include<conio.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");
}
```

7.11 COMPARISON OF Sorting ALGORITHMS

Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	Requires Memory
Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Large constants
Quick Sort	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Small constants

Algorithm	Average Case	Worst Case
Bubble sort	$O(n^2)$	$O(n^2)$
Bucket sort	$O(n.k)$	$O(n^2.k)$
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$
Shell sort	–	$O(n \log^2 n)$
Merge sort	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n^2)$

naresh.np

7.12 External Sort

External sorting is a sorting technique that can handle massive amounts of data. It is usually applied when the data being sorted does not fit into the main memory (RAM) and, therefore, a slower memory (usually a magnetic disk or even a magnetic tape) needs to be used.

Example: Let us consider we need to sort 700 MB of data using only 100 MB of RAM.

The steps for sorting are given below.

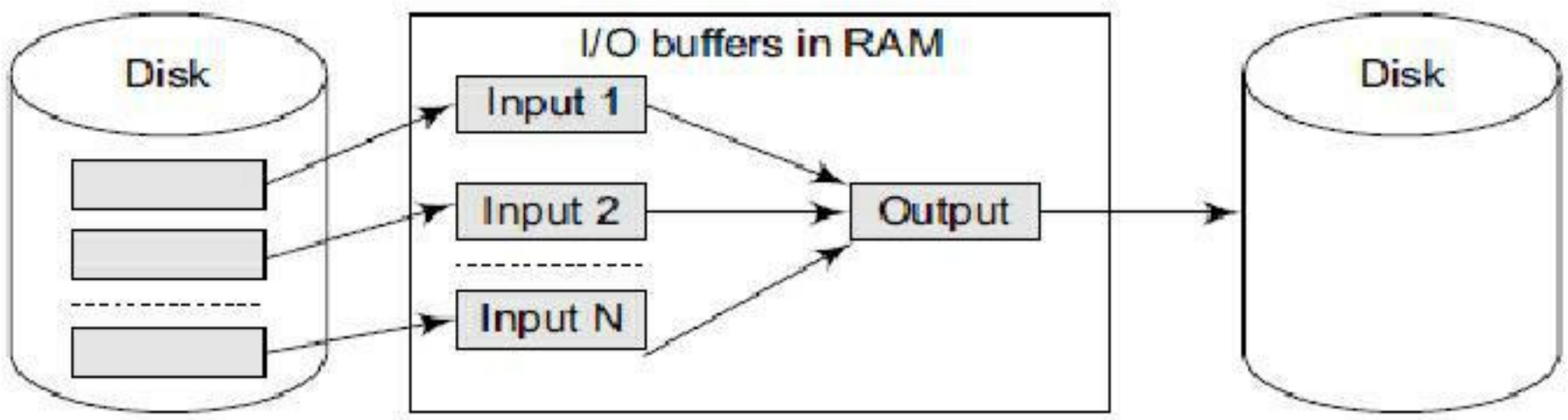
Step 1: Read 100 MB of the data in RAM and sort this data using any conventional sorting algorithm like quick sort.

Step 2: Write the sorted data back to the magnetic disk.

Step 3: Repeat Steps 1 and 2 until all the data (in 100 MB chunks) is sorted. All these seven chunks that are sorted need to be merged into one single output file.

Step 4: Read the first 10 MB of each of the sorted chunks and call them input buffers. So, now we have 70 MB of data in the RAM. Allocate the remaining RAM for output buffer.

Step 5: Perform seven-way merging and store the result in the output buffer. If at any point of time, the output buffer becomes full, then write its contents to the final sorted file. However, if any of the 7 input buffers gets empty, fill it with the next 10 MB of its associated 100 MB sorted chunk or else mark the input buffer (sorted chunk) as exhausted if it does not has any more left with it. Make sure that this chunk is not used for further merging of data. The external merge sorting can be visualized as given in Figure.



Generalized External Merge Sort Algorithm

From the example above, we can now present a generalized merge sort algorithm for external sorting. If the amount of data to be sorted exceeds the available memory by a factor of K, then K chunks (also known as K run lists) of data are created. These K chunks are sorted and then a K-way merge is performed. If the amount of RAM available is given as X, then there will be K input buffers and 1 output buffer.

In the above example, a single-pass merge was used. But if the ratio of data to be sorted and available RAM is particularly large, a multi-pass sorting is used. We can first merge only the first half of the sorted chunks, then the other half, and finally merge the two sorted chunks. The exact number of passes depends on the following factors:

- Size of the data to be sorted when compared with the available RAM
- Physical characteristics of the magnetic disk such as transfer rate, seek time, etc.

Applications of External Sorting

External sorting is used to update a master file from a transaction file. For example, updating the EMPLOYEES file based on new hires, promotions, appraisals, and dismissals.

It is also used in database applications for performing operations like Projection and Join. Projection means selecting a subset of fields and join means joining two files on a common field to create a new file whose fields are the union of the fields of the two files. External sorting is also used to remove duplicate records.

8 Searching

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful. The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.

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

8.1 Linear or Sequential Search

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

For **example**, if an array A[] is declared and initialized as, int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5}; and the value to be searched is VAL = 7, **then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position** of its occurrence. Here, POS = 3 (index starting from 0).

Algorithm for Linear Search

```
LINEAR_SEARCH(A, n, val)
        Step 1: Initialize SET pos = -1 and I = 0
        Step 2: Repeat Step 3 while I < n
        Step 3: If  A[I] = val
                        SET pos = I
                        Print pos
                        Go to Step 5
                [End of if]
                I = I + 1
        [End of Loop]
        Step 4: if pos = -1
                Print "Value is not present in the array"
        Step 5: Exit
```

Source Code

```
#include <stdio.h>
#include <conio.h>
#define size 20
void Linear_search(int [],int, int);
int main()
{
        int A[size],num,i,n;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
                scanf("%d", &A[i]);
```

```
        printf("\n Enter the number that has to be searched : ");
        scanf("%d", &num);
        Linear_search(A,n,num);
        getch();
        return 0;
}
void Linear_search(int A[],int n, int val)
{
        int i,pos=-1;
        for(i=0;i<n;i++)
        {
                if(A[i] == val)
                {
                        pos=i;
                        printf("\n %d is found in the array at position= %d", val,i+1);
                }
        }
        if (pos == -1)
        printf("\n %d does not exist in the array", val);
}
```

- **Complexity of Linear Search Algorithm**

  Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array.

8.2 Binary Search

Binary search is an extremely efficient algorithm. This search technique searches the given item in minimum possible comparisons. To do this binary search, first we need to sort the array elements. The logic behind this technique is given below:

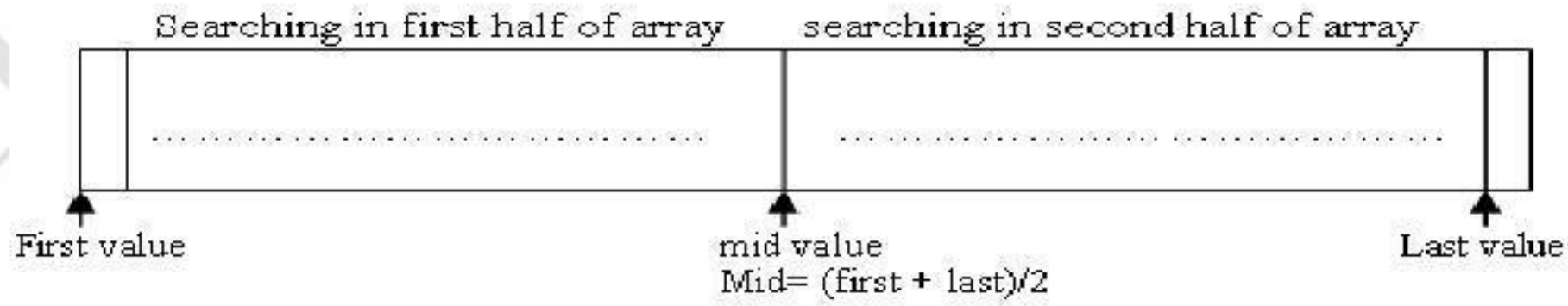
✓ First find the middle element of the array

✓ Compare the middle element with an item.

✓ There are three cases:

- If it is a desired element then search is successful
- If it is less than desired item then search only the first half of the array.
- If it is greater than the desired element, search in the second half of the array.

Repeat the same process until element is found or exhausts in the search area. In this algorithm every time we are reducing the search area.



Running example:

Take input array a[ ] = {2 , 5 , 7, 9 ,18, 45 ,53, 59, 67, 72, 88, 95, 101, 104}

nareshpad.com.np

For key = 2

low	high	mid	
0	13	6	key < A[6]
0	5	2	key < A[2]
0	1	0	

Terminating condition, since A[mid] = = 2, return 1(successful).

For key = 103

low	high	mid	
0	13	6	key > A[6]
7	13	10	key > A[10]
11	13	12	key > A[12]
13	13	-	

Terminating condition high = = low, since A[0] != 103, return 0(unsuccessful).

For key = 67

low	high	mid	
0	13	6	key > A[6]
7	13	10	key < A[10]
7	9	8	

Terminating condition, since A[mid] = 67, return 9(successful).

Algorithm for Binary Search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] = VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Efficiency:

From the above algorithm we can say that the running time of the algorithm is

$$T(n) = T(n/2) + O(1)$$

$$= O(\log n) \text{ (verify).}$$

In the best case output is obtained at one run i.e. O(1) time if the key is at middle. In the worst case the output is at the end of the array so running time is O(logn) time.ith In the average case also running time is O(logn). For unsuccessful search best, worst and average time complexity is O(logn).

Program to search an element in an array using binary search.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10
void selection_sort(int [], int);                    // Added to sort array
void Binary_search(int [],int,int,int);
int main()
{
        int arr[size],num,i,n,lower_bound,upper_bound;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        selection_sort(arr, n);              // Added to sort the array
                printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
                printf(" %d\t", arr[i]);
        printf("\n\n Enter the number that has to be searched: ");
        scanf("%d", &num);
        lower_bound = 0;
        upper_bound = n-1;
        Binary_search(arr,lower_bound,upper_bound,num);
        getch();
        return 0;
}
void Binary_search(int arr[],int lower_bound,int upper_bound,int val)
{
        int beg,end,mid,pos;
        beg = lower_bound;
        end = upper_bound;
        pos=-1;
        while(beg<=end)
        {
                mid = (beg + end)/2;
                if (arr[mid] == val)
                {
                        pos = mid;
                        printf("\n %d is present in the array at index position %d", val,pos);
                        break;
                }
                else if (arr[mid]>val)
                        end = mid-1;
                else
                        beg = mid+1;
        }
        if (pos==-1)
                printf("\n %d does not exist in the array", val);
}
```

nareshpad.com.np

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

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

Enter the elements:
Enter 1 element:10
Enter 2 element:20
Enter 3 element:5
Enter 4 element:30
Enter 5 element:15

The sorted array is:
5       10      15      20      30

Enter the number that has to be searched: 15

15 is present in the array at index position 2
```

8.3 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.
- It is an efficient searching technique in which key is placed in direct accessible address for rapid search.
- Hashing provides the direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. This technique uses a hashing function say **H** which maps the key with the corresponding key address or location.
- **Collision**: suppose that two elements k1 and k2 are such that H(k1) = H(k2). Then when element k1 is entered into the table, it is inserted at position H(k1). But when k2 is hashed, an attempt may be made to insert the element into the same position where the element k1 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.

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 greater than 90. Suppose m = 93. Then for the following employee code (or key k) :

H(k) = H(2103) = 2103 mod 93 = 57

H(k) = H(6147) = 6147 mod 93 = 9

H(k) = H(3750) = 3750 mod 93 = 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.

Mid Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

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

17197609
14062500
4422609

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

Folding Method

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.
Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any.
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

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

8.3.1 Hash Collision and Handling of Hash Collision

It is possible that two non-identical keys K1, K2 are hashed into the same hash address. This situation is called Hash Collision.

Let us consider a hash table having 10 locations as in table.

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

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

$$H(500) = 500(\text{mod } 10) = 0.$$

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 techniques:

- Open addressing
- Chaining
- Bucket addressing

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 Table 1. 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.

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,.....

Double Hashing

To start with, double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name double hashing. In double hashing, we use two hash functions rather than a single function. The hash function in the case of double hashing can be given as:

$$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$$

where m is the size of the hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as $h_1(k) = k \bmod m$, $h_2(k) = k \bmod m'$, i is the probe number that varies from 0 to $m-1$, and m' is chosen to be less than m. We can choose $m' = m-1$ or $m-2$.

When we have to insert a key k in the hash table, we first probe the location given by applying $[h_1(k) \bmod m]$ because during the first probe, $i = 0$. If the location is vacant, the key is inserted into it, else subsequent probes generate locations that are at an offset of $[h_2(k) \bmod m]$ from the previous location. Since the offset may vary with every probe depending on the value generated by the second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing.

Rehashing

When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table. All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table. Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently.

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 can represented using linked list as in following figure.

Location	Keys	Records
0	210	30
1 2	111	12
3	883	14
4	344	18
5		
6	546	32
7		
8 9	488	31

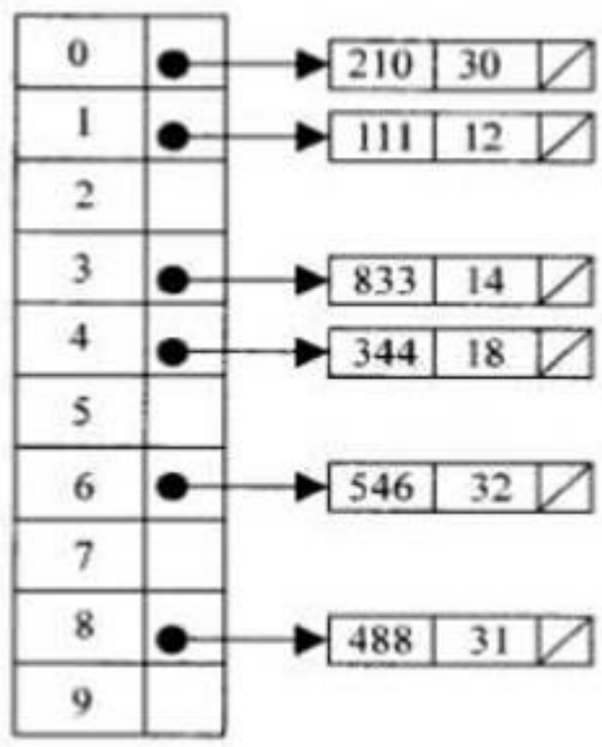


Figure: Chaining

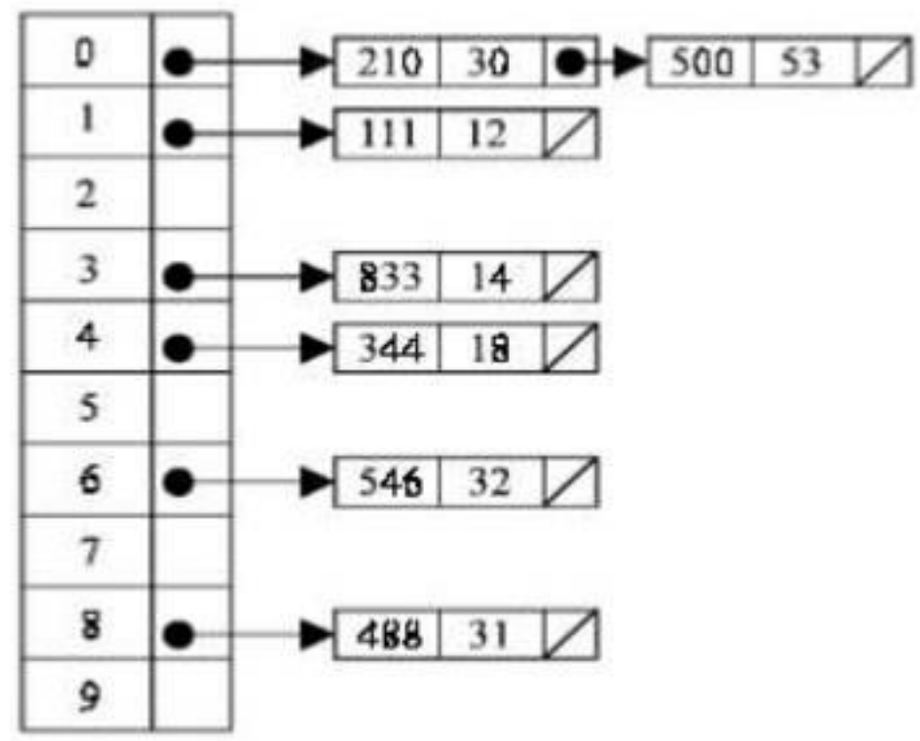


Figure: Linked List at 0 index

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.

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.

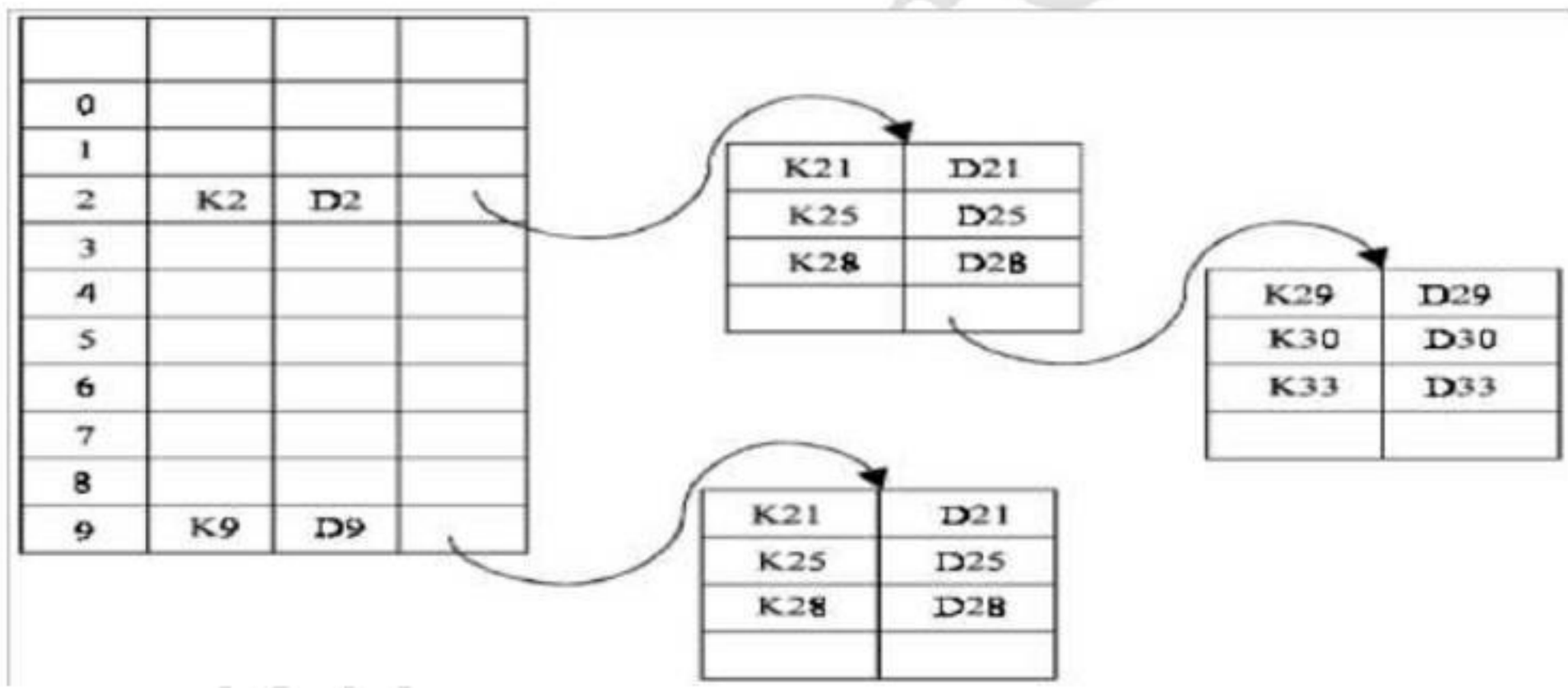


Figure: Avoid Collision Using bucket

Figure 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.

9 Graphs

- A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.
- **Why are Graphs Useful?**
  Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:
  - Family trees in which the member nodes have an edge from parent to each of their children.
  - Transportation networks in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.
- Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc.
- **Definition**
  A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.
- Figure 9.1 shows a graph with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}. Note that there are five vertices or nodes and six edges in the graph.

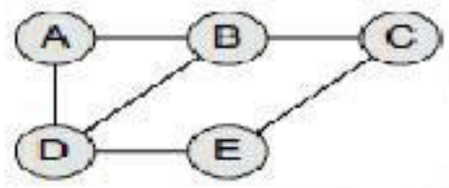


Figure 9-1: Undirected Graph

- A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. Figure 9.1 shows an undirected graph because it does not give any information about the direction of the edges.

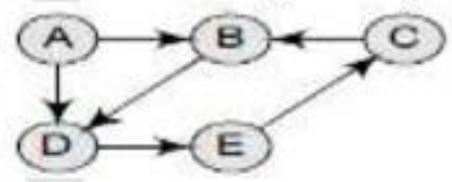


Figure 9-2: Directed Graph

- Look at Figure 9.2 which shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

Graph Terminology

- **Adjacent nodes** or neighbours For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end-points and are said to be the adjacent nodes or neighbours.
- **Degree of a node** Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.
- **Regular graph** It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a k–regular graph or a regular graph of degree k. Figure 9.3 shows regular graphs.

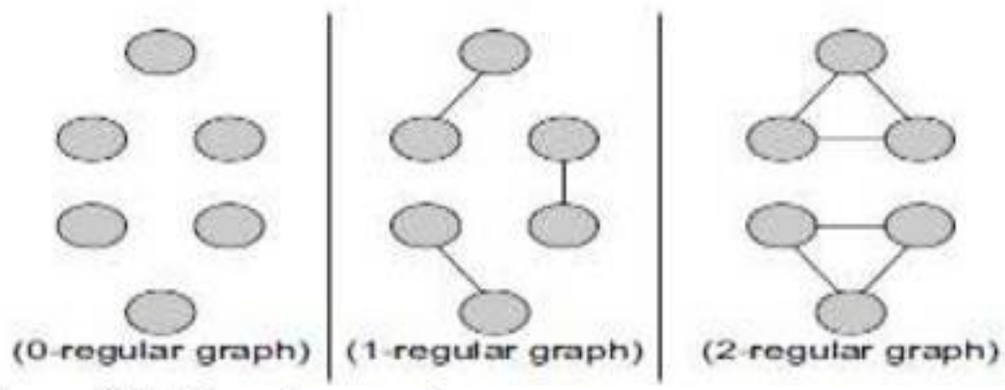


Figure 9-3: Regular graphs

- **Path** A path P written as P = {v0, v1, v2, ..., vn), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v0, v = vn and vi–1 is adjacent to vi for i = 1, 2, 3,..., n.
- **Closed path** A path P is known as a closed path if the edge has the same end-points. That is, if v0 = vn.
- **Simple path** A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0 = vn, then the path is called a closed simple path.
- **Cycle** A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).
- **Connected graph** A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a **tree**. Therefore, a tree is treated as a special graph (Refer Fig. 9.4(b)).

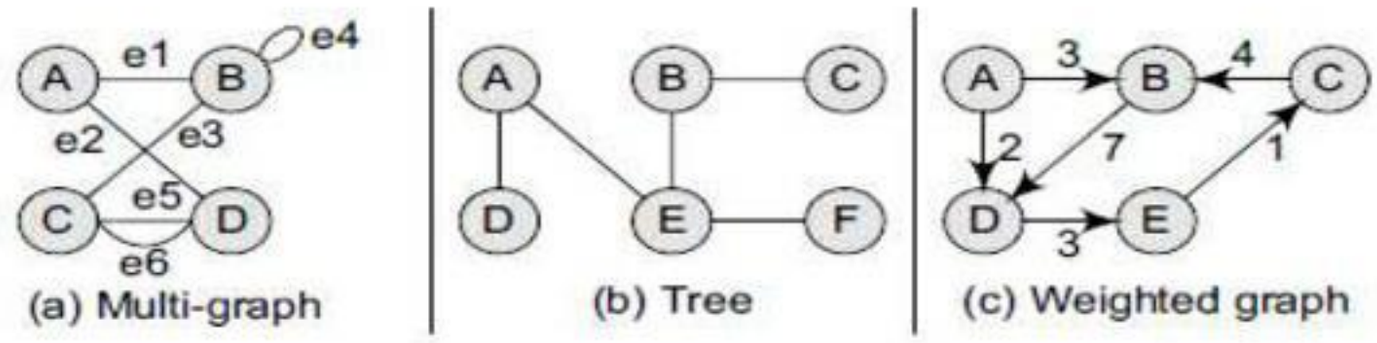


Figure 9-4: Multi-graph, tree, and weighted graph

- **Complete graph** A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.
- **Clique** In an undirected graph G = (V, E), clique is a subset of the vertex set C ⊆V, such that for every two vertices in C, there is an edge that connects two vertices.
- **Labelled graph** or **weighted graph** A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge. Figure 9.4(c) shows a weighted graph.
- **Multiple edges** Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.
- **Loop** An edge that has identical end-points is called a loop. That is, e = (u, u).
- **Multi-graph** A graph with multiple edges and/or loops is called a multi-graph. Figure 9.4(a) shows a multi-graph.
- **Size of a graph** The size of a graph is the total number of edges in it.

9.1 Representations of Graph

Graph is a mathematical structure and finds its application in many areas of interest in which problem need to be solved using computer. Thus, this mathematical structure must be represented in some kind of data structure. There are three common ways of storing graphs in **the computer's memory. They are**:

- Sequential representation by using an **adjacency matrix**.
- Linked representation by using an **adjacency list** that stores the neighbours of a node using a linked list.
- **Adjacency multi-list** which is an extension of linked representation.

9.1.1 Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n x n.

$$a_{ij} \begin{cases} 1 & \text{[if } v_i \text{ is adjacent to } v_j\text{, that is there is an edge } (v_i, v_j)\text{]A} \\ 0 & \text{[otherwise]} \end{cases}$$

Figure 9-5: Adjacency matrix entry

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero. It is summarized in Figure 9.5 Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix. Figure 9.6 shows some graphs and their corresponding adjacency matrices.

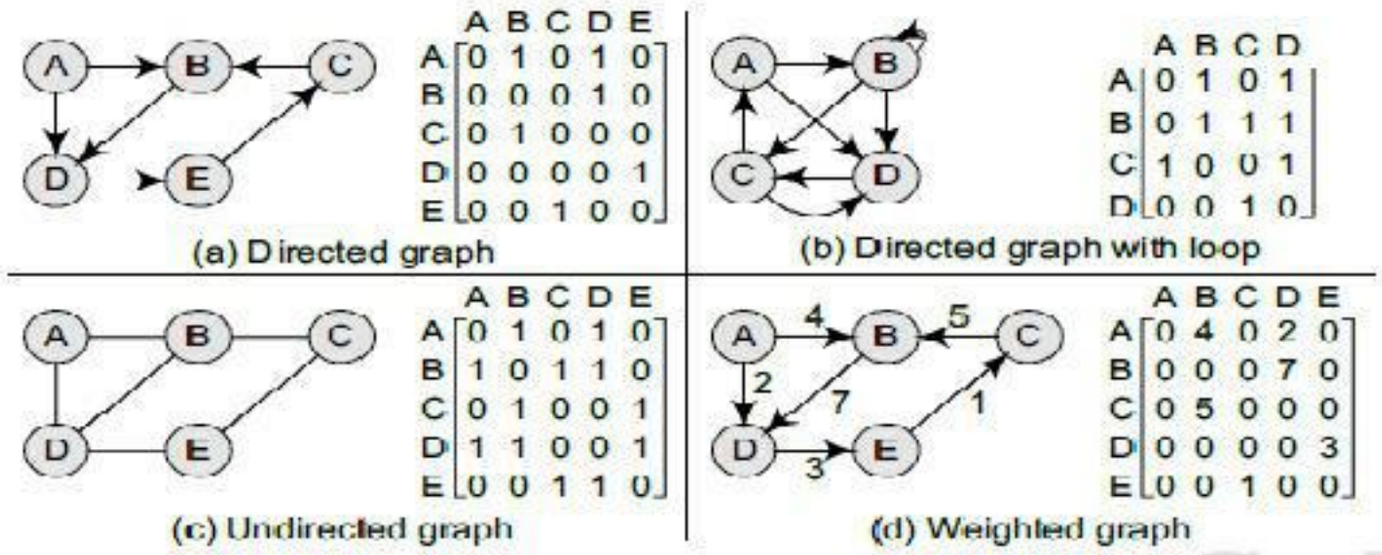


Figure 9-6: Graphs and their corresponding adjacency matrices

From the above examples, we can draw the following conclusions:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is O(n2), where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Now let us discuss the powers of an adjacency matrix. From adjacency matrix A^1, we can conclude that an entry 1 in the ith row and jth column means that there exists a path of length 1 from v_i to v_j. Now consider, A^2, A^3, and A^4.

$$(a_{ij})^2 = \sum a_{ik}\, a_{kj}$$

Any entry $a_{ij} = 1$ if $a_{ik} = a_{kj} = 1$. That is, if there is an edge (v_i, v_k) and (v_k, v_j), then there is a path from v_i to v_j of length 2.

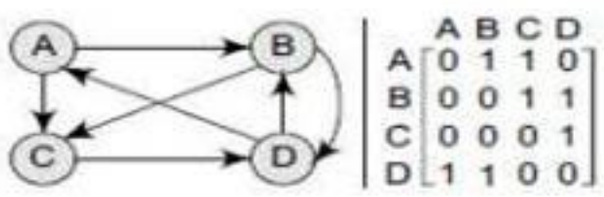


Figure 9.7 Directed graph with its adjacency

Similarly, every entry in the ith row and jth column of A^3 gives the number of paths of length 3 from node v_i to v_j.

In general terms, we can conclude that every entry in the ith row and jth column of A^n (where n is the number of nodes in the graph) gives the number of paths of length n from node v_i to v_j. Consider a directed graph given in Figure 9.7 Given its adjacency matrix A, let us calculate A^2, A^3, and A^4.

$$A^2 = A^1 \times A^1$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

Now, based on the above calculations, we define matrix B as:

$B^r = A^1 + A^2 + A^3 + \ldots + A^r$

An entry in the ith row and jth column of matrix B^r gives the number of paths of length r or less than r from vertex v_i to v_j. The main goal to define matrix B is to obtain the path matrix P. The path matrix P can be calculated from B by setting an entry $P_{ij} = 1$, if B_{ij} is non-zero and $P_{ij} = 0$, if otherwise. The path matrix is used to show whether there exists a simple path from node v_i to v_j or not. This is shown in Figure 9.8

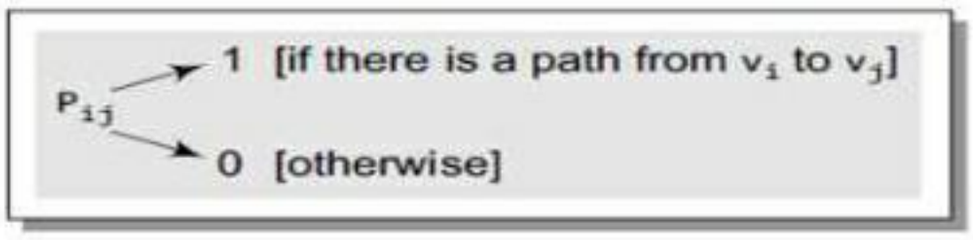


Figure 9.8: Path matrix entry

Let us now calculate matrix B and matrix P using the above discussion.

$$\begin{bmatrix}0&1&1&0\\0&0&1&1\\0&0&0&1\\1&1&0&0\end{bmatrix}+\begin{bmatrix}0&0&1&2\\1&1&0&1\\1&1&0&0\\1&1&2&1\end{bmatrix}+\begin{bmatrix}2&2&0&1\\1&2&2&1\\0&1&2&1\\1&2&2&3\end{bmatrix}+\begin{bmatrix}1&3&4&2\\1&2&3&4\\1&1&1&3\\3&4&3&4\end{bmatrix}=\begin{bmatrix}3&6&6&5\\3&5&6&7\\2&3&3&5\\6&8&7&8\end{bmatrix}$$

B =

Now the path matrix P can be given as:

$$P=\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}$$

In this representation, n2 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.

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

9.1.2 Adjacency List Representation

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred **for representing sparse graphs in the computer's** memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered. Consider the graph given in Figure 9.9 and see how its adjacency list is stored in the memory. For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G. However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u. Adjacency lists can also be modified to store weighted graphs. Let us now see an adjacency list for an undirected graph as well as a weighted graph. This is shown in Figure 9.10.

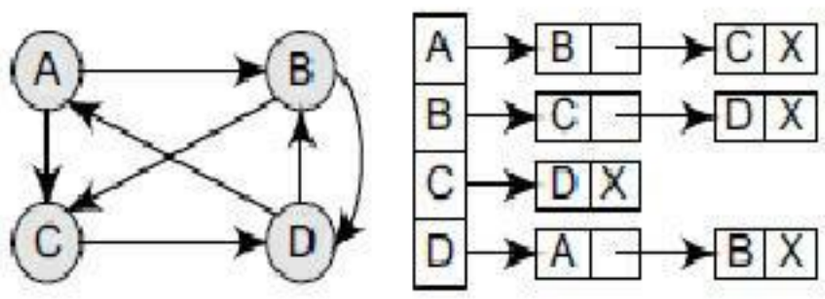


Figure 9-9: Graph G and its adjacency list

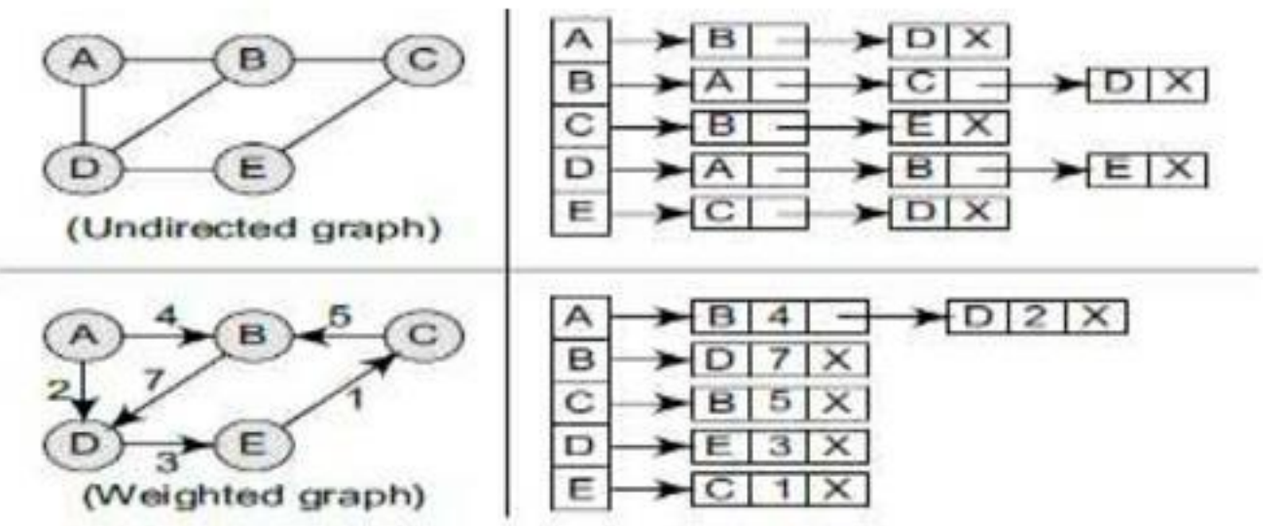


Figure 9-10: Adjacency list for an undirected graph and a weighted graph

9.1.3 Adjacency Multi-list Representation

Graphs can also be represented using multi-lists which can be said to be modified version of adjacency lists. Adjacency multi-list is an edge-based rather than a vertex-based representation of graphs. A multi-list representation basically consists of two parts—**a directory of nodes' information** and a set of linked lists storing information about edges. While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge). For example, the directory entry for node i points to the adjacency list for node i. This means that the nodes are shared among several lists.

In a multi-list representation, the information about an edge (v_i, v_j) of an undirected graph can be stored using the following attributes:

M: A single bit field to indicate whether the edge has been examined or not.

v_i: A vertex in the graph that is connected to vertex v_j by an edge.

v_j: A vertex in the graph that is connected to vertex v_i by an edge.

Link i for v_i: A link that points to another node that has an edge incident on v_i.

Link j for v_i: A link that points to another node that has an edge incident on v_j.

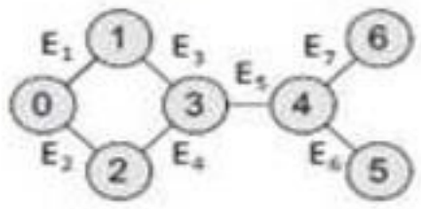


Figure 9.11: Undirected graph

Consider the undirected graph given in Figure 9.11.

The adjacency multi-list for the graph can be given as:

Edge 1		0	1	Edge 2	Edge 3
Edge 2		0	2	NULL	Edge 4
Edge 3		1	3	NULL	Edge 4
Edge 4		2	3	NULL	Edge 5
Edge 5		3	4	NULL	Edge 6
Edge 6		4	5	Edge 7	NULL
Edge 7		4	6	NULL	NULL

Using the adjacency multi-list given above, the adjacency list for vertices can be constructed as shown below:

VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7

9.2 Implementation of graph in C

// Write a program to create a graph of **n** vertices using an **adjacency list**. Also write the code to read and print its information and finally to delete the graph.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct node
{
        char vertex;
        struct node *next;
};
struct node *gnode;
void createEmptyGraph(struct node *Adj[], int no_of_nodes);
void displayGraph(struct node *adj[], int no_of_nodes);
void deleteGraph(struct node *adj[], int no_of_nodes);
void inputGraph(struct node *adj[], int no_of_nodes);
int main()
{
        struct node *Adj[10];
        int i, no_of_nodes;
        printf("\n Enter the number of nodes in G: ");
        scanf("%d", &no_of_nodes);
        createEmptyGraph(Adj, no_of_nodes);
        inputGraph(Adj, no_of_nodes);
        printf("\n The graph is: ");
        displayGraph(Adj, no_of_nodes);
        deleteGraph(Adj, no_of_nodes);
        getch();
        return 0;
}
void createEmptyGraph(struct node *Adj[], int no_of_nodes)
{
        int i;
        for(i = 0; i < no_of_nodes; i++)
                        Adj[i] = NULL;
}
void inputGraph(struct node *Adj[], int no_of_nodes)
{
        struct node *new_node, *last;
        int i, j, n, val;
        for(i = 0; i < no_of_nodes; i++)
        {
                last = NULL;
                printf("\n Enter the number of neighbours of %d: ", i);
                scanf("%d",&n);
                for(j = 0; j < n; j++)
                {
                        printf("\n Enter the neighbour %d of %d: ", j+1, i);
                        scanf("%d", &val);
                        new_node = (struct node *)malloc(sizeof(struct node));
                        new_node->vertex = val;
                        new_node->next = NULL;
                        if (Adj[i] == NULL)
```

naresh.com.np

```
                        Adj[i] = last=new_node;
                    else
                    {
                        last -> next = new_node;
                        last = new_node;
                    }
                }
            }
}
void displayGraph (struct node *Adj[], int no_of_nodes)
{
        struct node *ptr;
        int i;
        for(i = 0; i < no_of_nodes; i++)
        {
                ptr = Adj[i];
                printf("\n The neighbours of node %d are:", i);
                while(ptr != NULL)
                {
                        printf("\t%d", ptr -> vertex);
                        ptr = ptr -> next;
                }
        }
}
void deleteGraph (struct node *Adj[], int no_of_nodes)
{
        int i;
        struct node *temp, *ptr;
        for(i = 0; i< no_of_nodes;i++)
        {
                ptr = Adj[i];
                while(ptr != NULL)
                {
                        temp = ptr;
                        ptr = ptr -> next;
                        free(temp);
                }
                Adj[i] = NULL;
        }
}
```

```
Enter the number of nodes in G: 3

Enter the number of neighbours of 0: 1

Enter the neighbour 1 of 0: 1

Enter the number of neighbours of 1: 2

Enter the neighbour 1 of 1: 0

Enter the neighbour 2 of 1: 2

Enter the number of neighbours of 2: 1

Enter the neighbour 1 of 2: 1

The graph is:
The neighbours of node 0 are:  1
The neighbours of node 1 are:  0        2
The neighbours of node 2 are:  1
```

naresh.com.np

9.3 Graph Traversals

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

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal. That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Figure: Algorithm for breadth-first search

Analysis

From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjacent vertices are looked for and this can be done in O(n) time(for the worst case the graph is complete). This computation for all the possible vertices that may be in the queue i.e. n, produce complexity of an algorithm as $O(n^2)$.

Example: Consider the graph G given in Figure below. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.

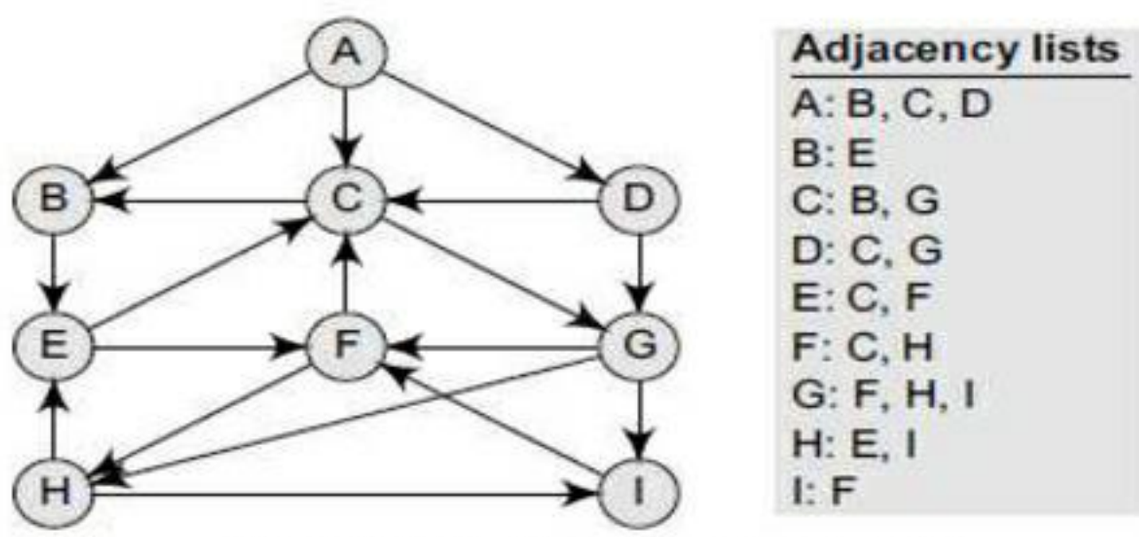


Figure: Graph G and its adjacency list

Solution

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays: QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1. The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A –> C –> G –> I.

Features of Breadth-First Search Algorithm

Space complexity: In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$. If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time complexity In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches $O(b^d)$. However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

Completeness Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

Optimality Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

9.3.2 Depth First Search

- The depth-first search algorithm (Fig. 13.22) progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
- The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the deadend, we backtrack to find another path P¢. The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges.
- Observe that this algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue. Again, we use a variable STATUS to represent the current state of the node.

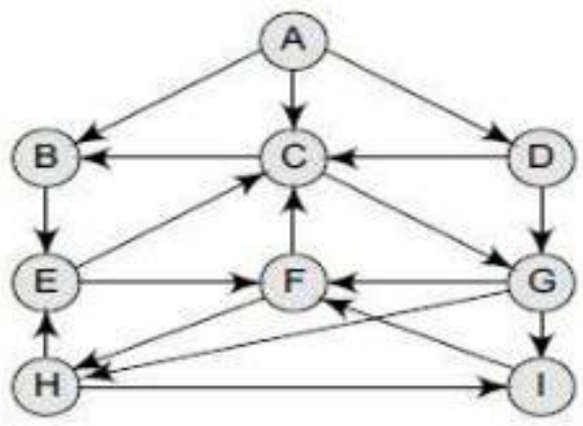
```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Figure: Algorithm for depth-first search

Analysis:

The complexity of the algorithm is greatly affected by Traverse function we can write its running time in terms of the relation $T(n) = T(n-1) + O(n)$, here $O(n)$ is for each vertex at most all the vertices are checked (for loop). At each recursive call a vertex is decreased. Solving this we can find that the complexity of an algorithm is $O(n^2)$.

Example: Consider the graph G given in Figure The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Figure: Graph G and its adjacency list

Solution

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

- **Features of Depth-First Search Algorithm**

Space complexity The space complexity of a depth-first search is lower than that of a breadth first search.

Time complexity The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)).

Completeness Depth-first search is said to be a complete algorithm. If there is a solution, depth first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

- **Applications of Depth-First Search Algorithm**

Depth-first search is useful for:

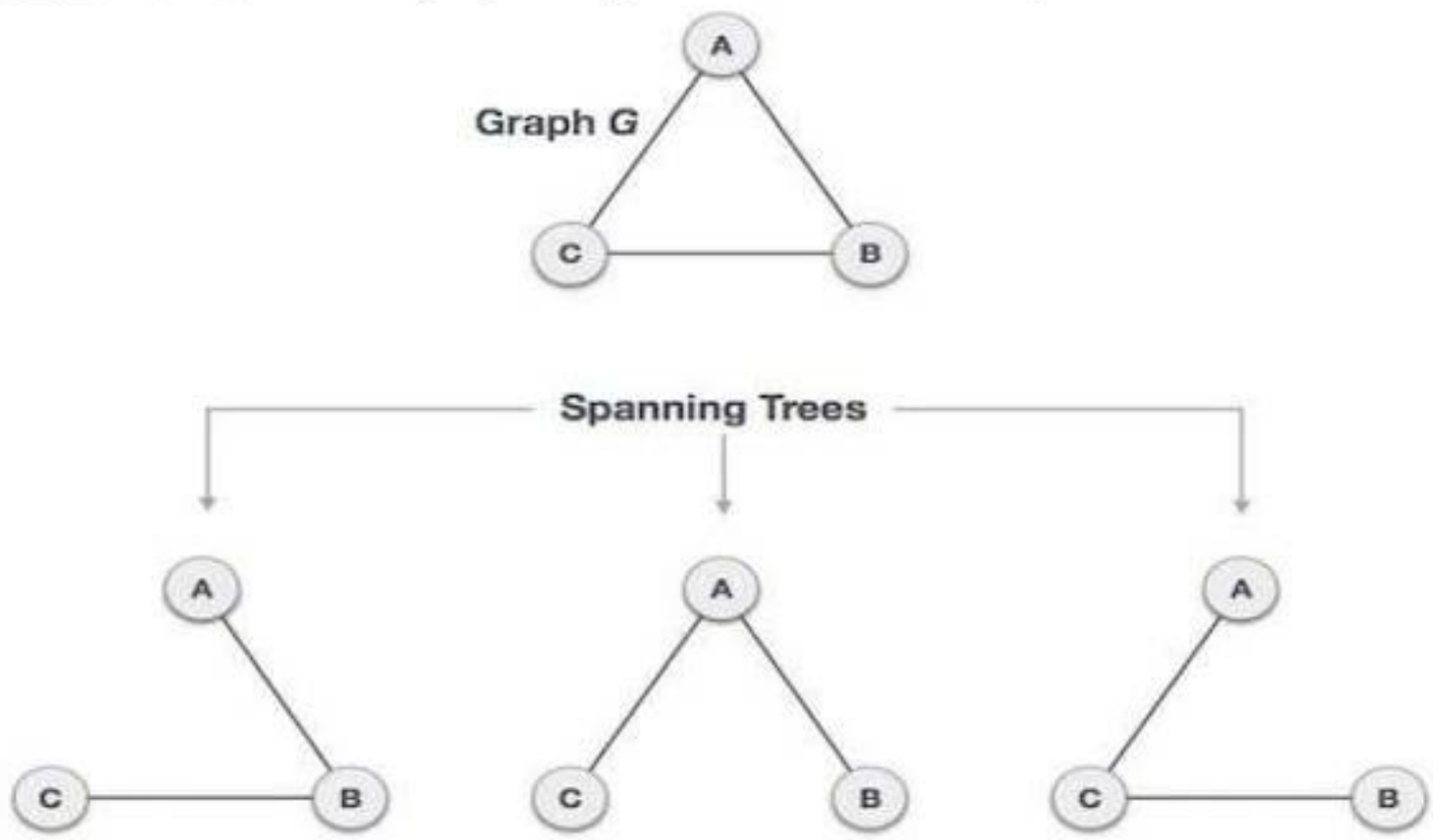
- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

❖ **Application of Graph Structures**

- Graph is an important data structure whose extensive applications are known in almost all application areas. Nowadays, many applications related with computation can be managed efficiently with graph structures.
- It is not possible to discuss all the problems where graph structures are involved because this domain is very large; however, we limit discussions with a few important of them which play significant roles in various applications.
- In order to maintain the generality, we will discuss them as graph theoretic problems. The major graph theoretic problems are:
  - Shortest path problem
  - Topological sorting of a graph
  - Spanning trees
  - Connectivity of a graph
  - Euler's path and Hamiltonian path
  - Binary decision diagram

9.4 Spanning Tree

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.
- By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



- We found three spanning trees off one complete graph. A complete undirected graph can have maximum nn-2 number of spanning trees, where n is the number of nodes. In the **above addressed example, n is 3, hence 33−2 = 3 spanning** trees are possible.

General Properties of Spanning Tree

- We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G.
- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Mathematical Properties of Spanning Tree

- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum nn-2 number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

- Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are:
  - Civil Network Planning
  - Computer Network Routing Protocol
  - Cluster Analysis

  Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

9.5 Minimum Spanning Tree

- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.
- A minimum spanning tree (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.
- A minimum spanning tree (MST) for a graph G = (V, E) is a sub graph G1 = (V1, E1) of G contains all the vertices of G.
  - The vertex set V1 is same as that at graph G.
  - The edge set E1 is a subset of G.
  - 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. Figure B shows the minimum spanning tree of the weighted graph in Figure A.

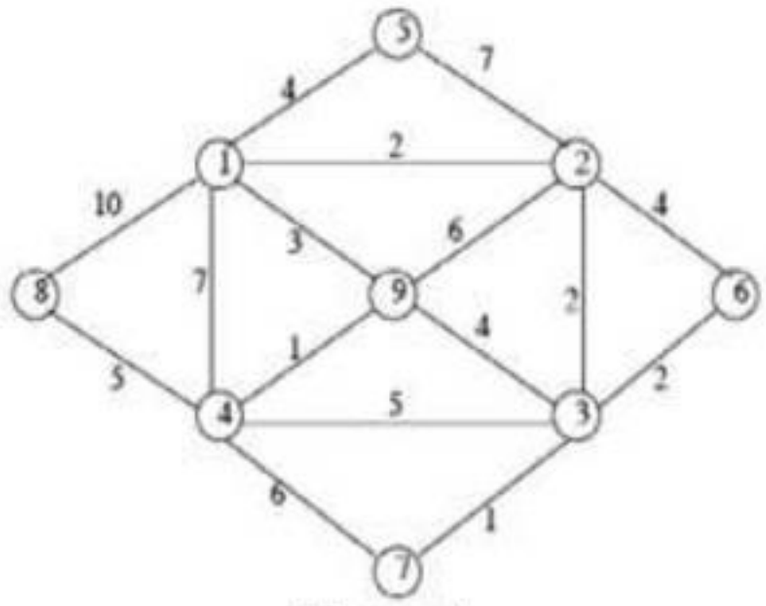


Figure A

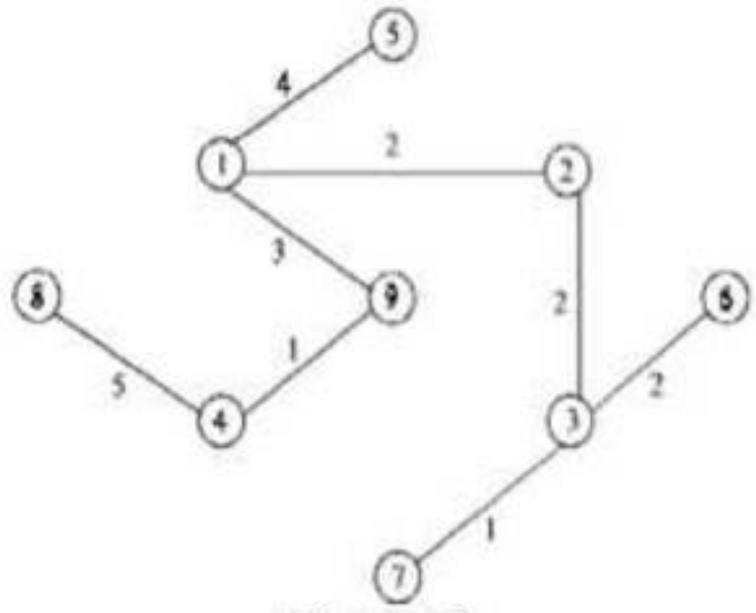


Figure B

9.5.1 Kruskal's Algorithm

- **Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph.** The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if the graph is not connected, then it finds a minimum spanning forest. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees.
- 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.

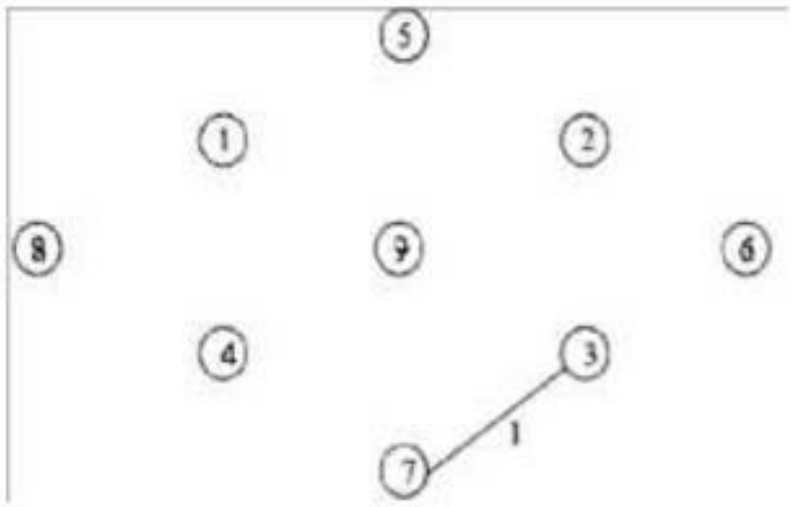


Figure A

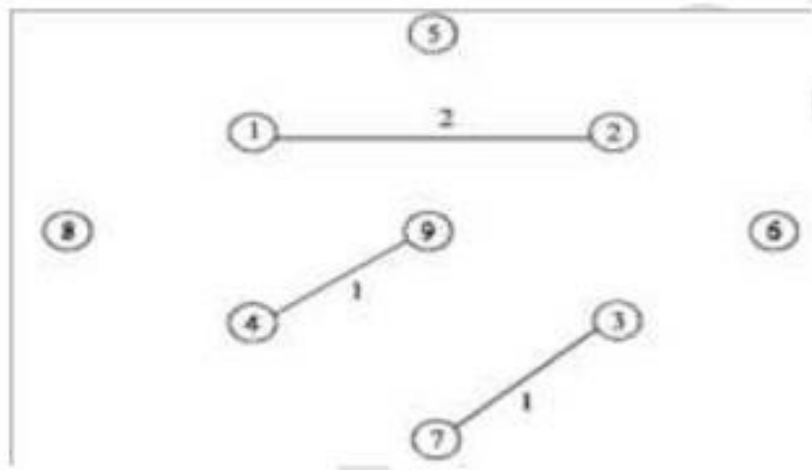


Figure B

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 Figure A. 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 Figure B.

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 Figure C

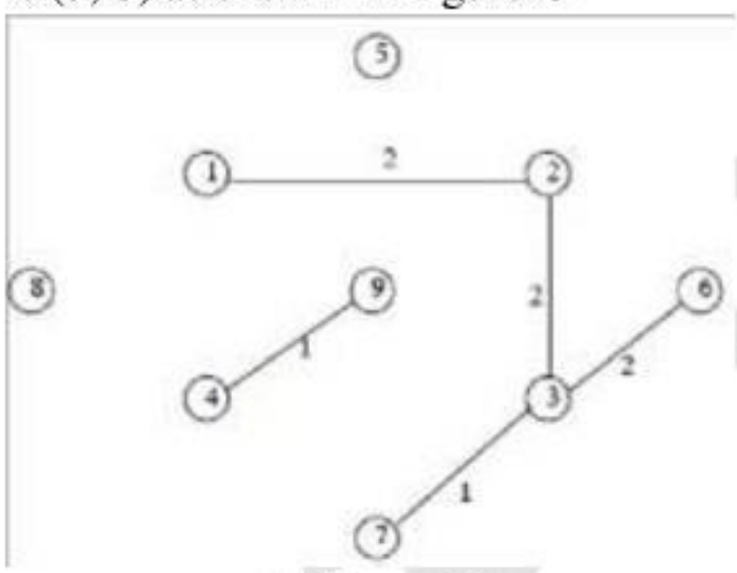


Figure C

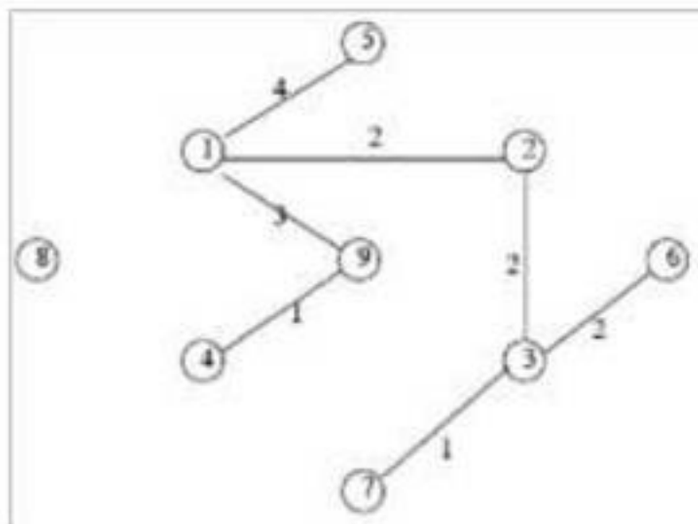


Figure D

Next minimum cost edge is (1, 9) with cost 3. Add the minimum cost edge to the minimumspanning 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 Figure D. Next minimum cost edge is (4, 8) with cost 5. Add the minimum cost edge to the minimum-spanning tree as shown in Figure E which is the minimum spanning tree with minimum cost of edges.

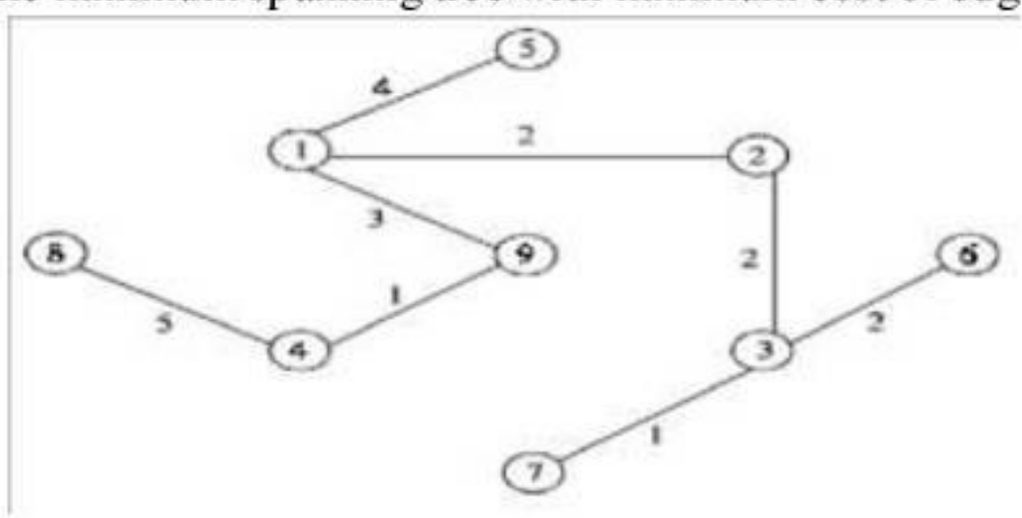


Figure E

Algorithm

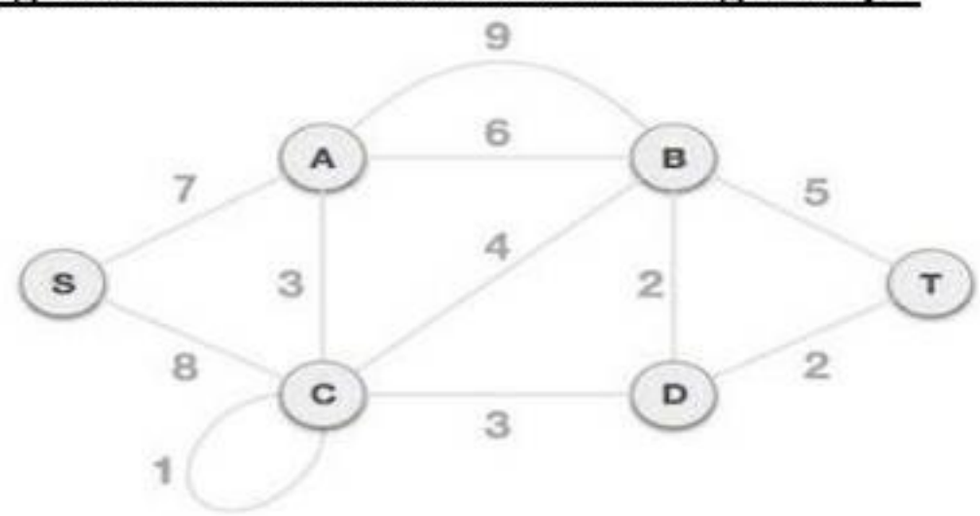
Suppose G = (V, E) is a graph, and T is a minimum spanning tree of graph G.

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

Analysis:

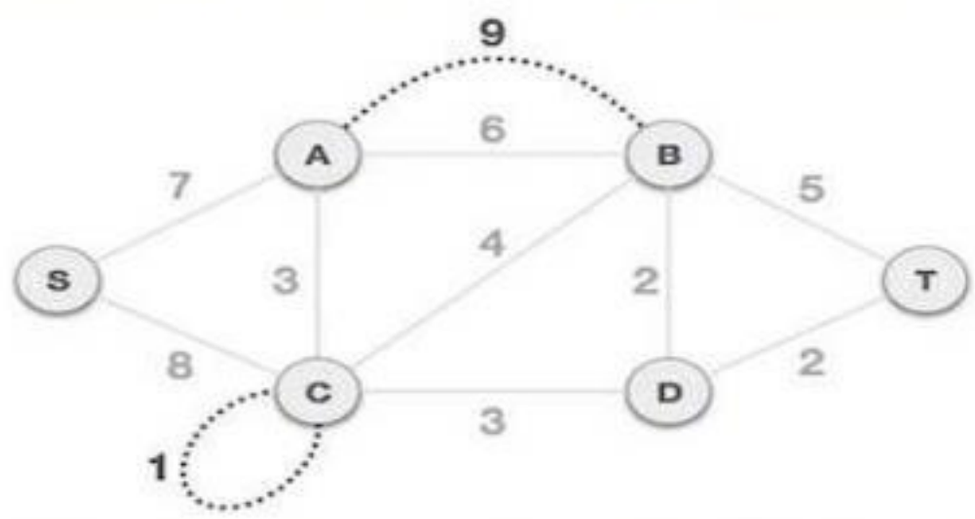
In the above algorithm the n tree forest at the beginning takes (V) time, the creation of set S takes O(ElogE) time and while loop execute O(n) times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is O(ElogE)

To understand Kruskal's algorithm let us consider the following example

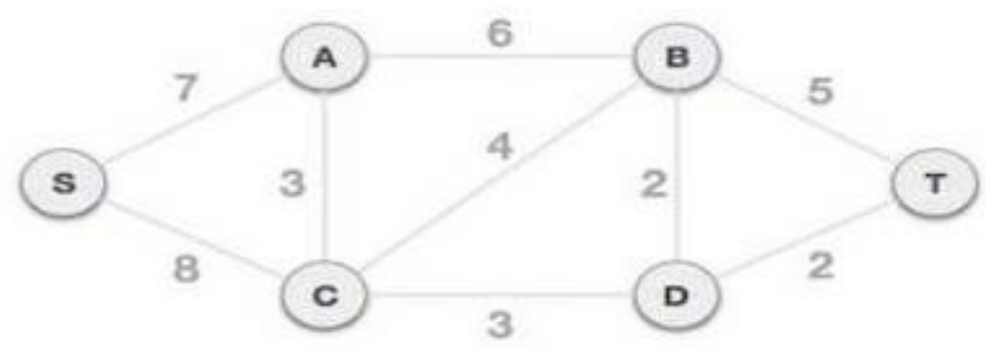


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



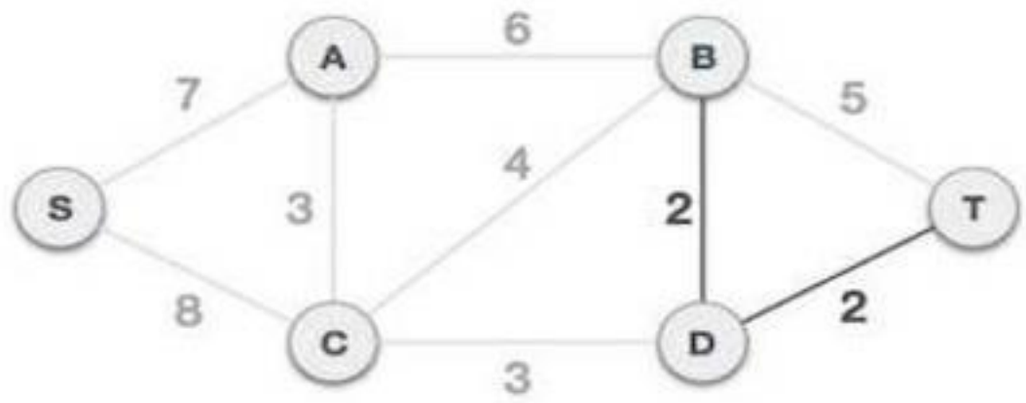
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

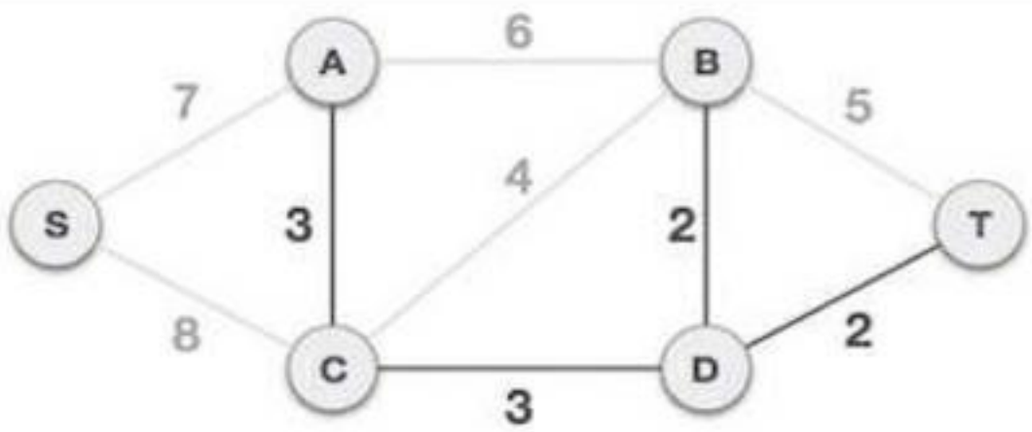
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

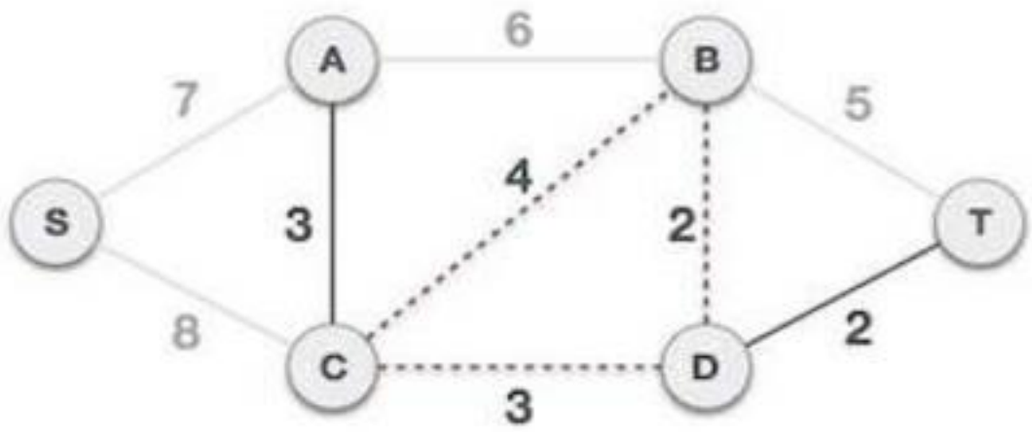


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

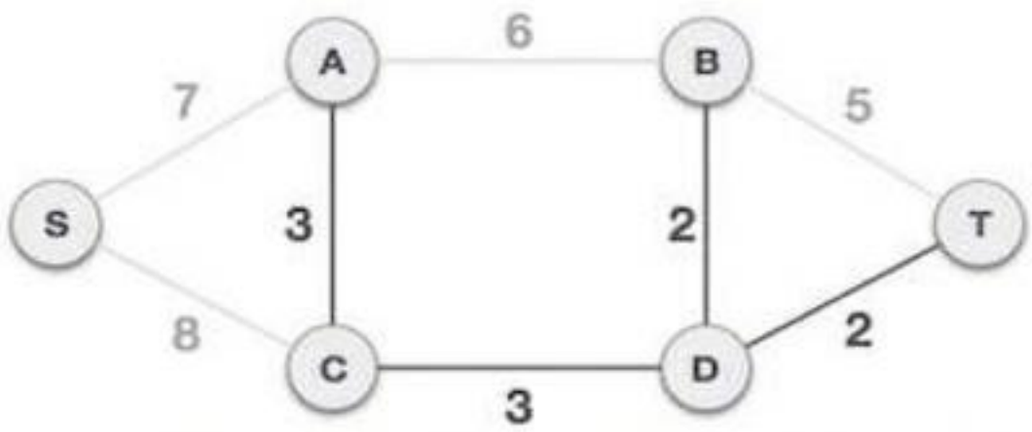
Next cost is 3, and associated edges are A,C and C,D. We add them again –



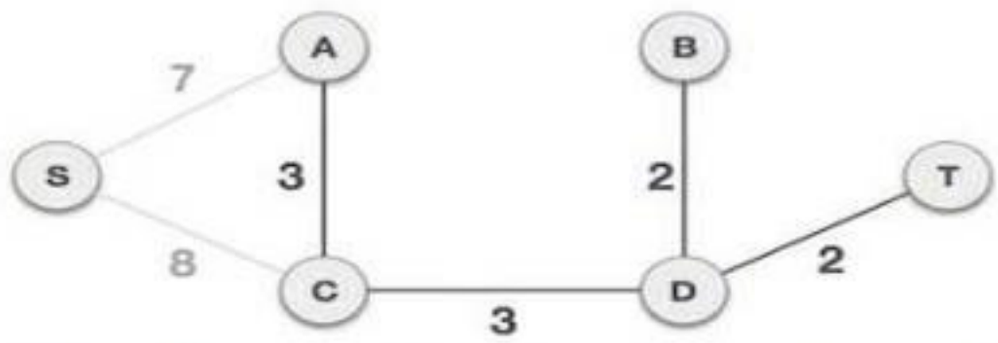
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



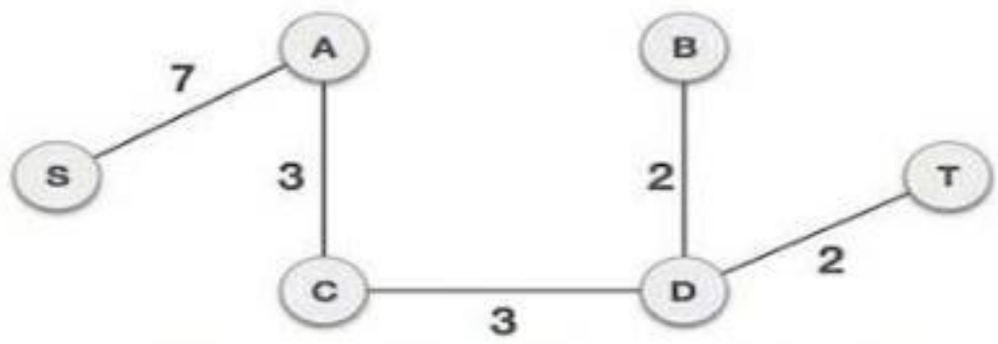
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

9.5.2 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.

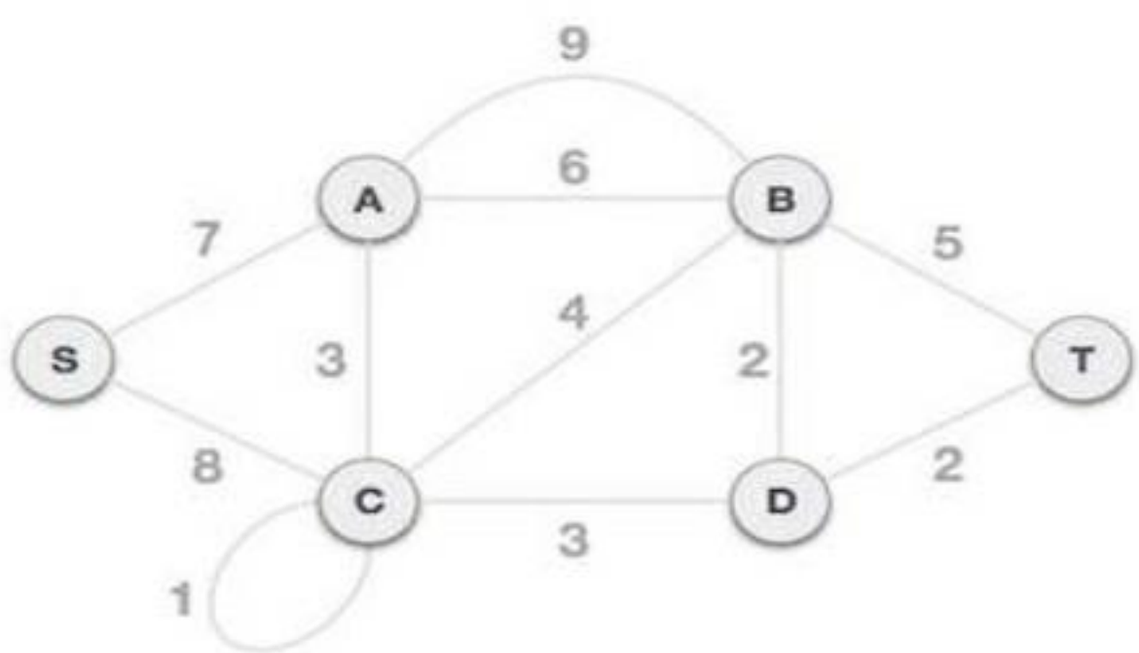
All partial trees are maintained in a queue, Q. Associated with each partial tree, T, is a priority queue, P(T), of all arc with **exactly one incident node in a tree, ordered by the weights of the arcs. Initially, as in Krusal's algorithm**, each node is a particular tree. A priority queue of the entire 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.

9.5.3 Prim's Algorithm

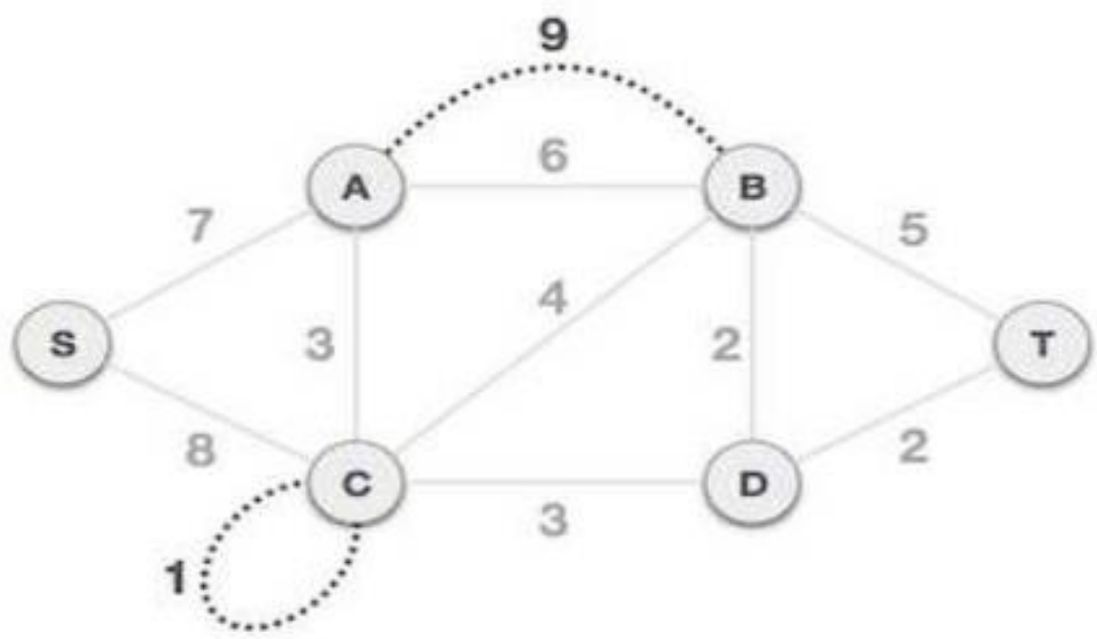
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

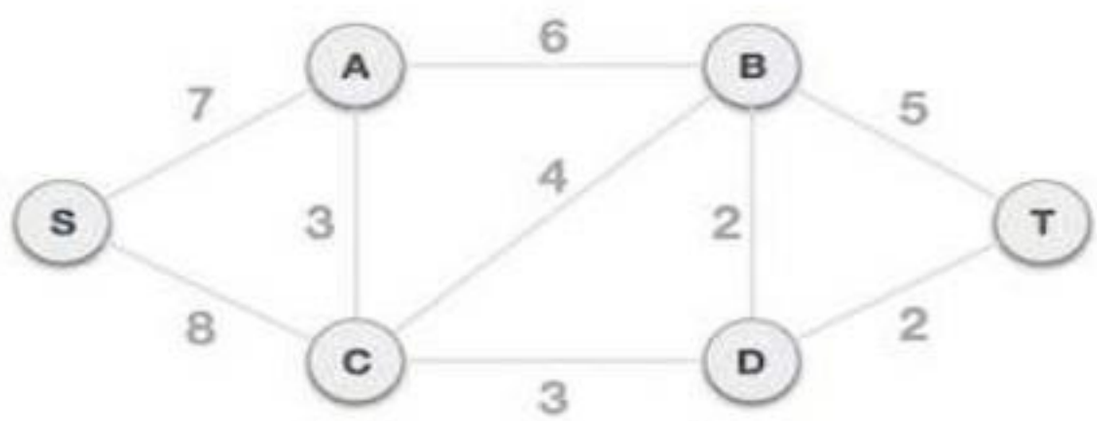
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

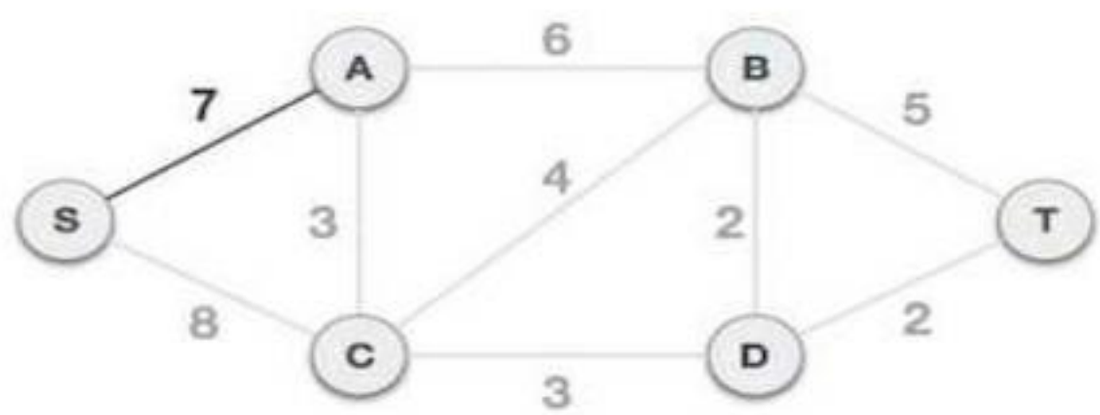


Step 2 - Choose any arbitrary node as root node

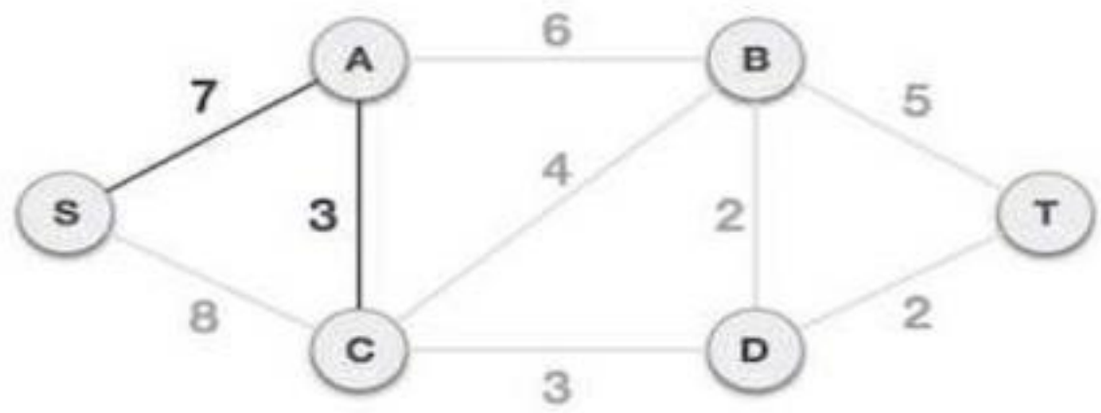
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

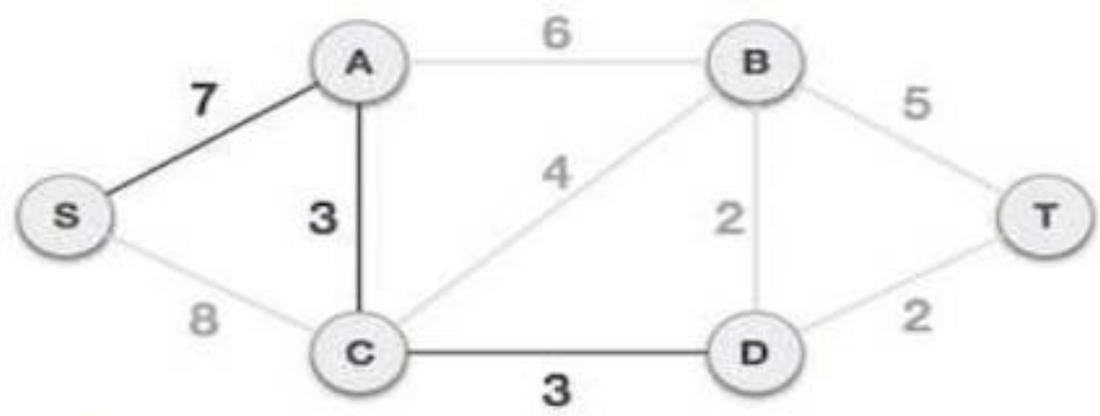
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



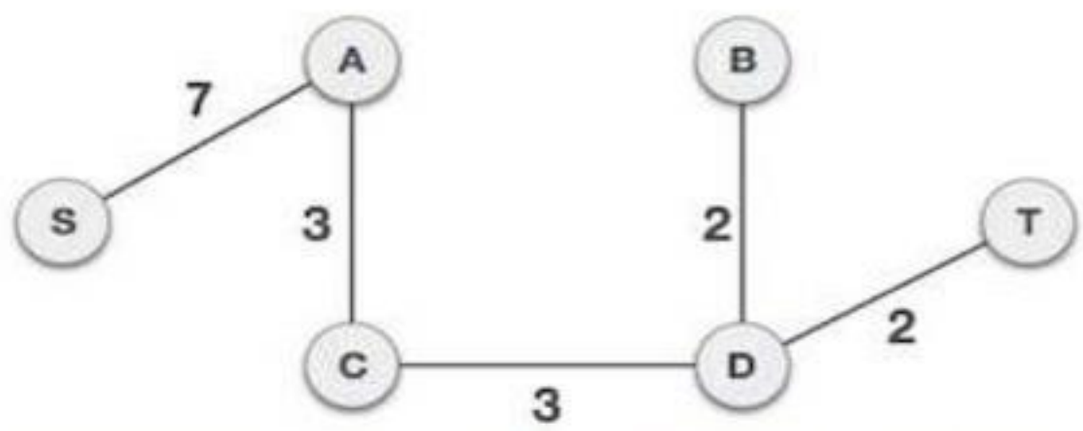
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

9.6 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.

9.7 Dijkstra's Algorithm

Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree. This algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First). Given a graph G and a source node A, the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every **other node. Moreover, Dijkstra's algorithm** is also used for finding the costs of the shortest paths from a source node to a destination node. For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving **distances between pairs of cities connected by a direct road, then Dijkstra's** algorithm when applied gives the shortest route between one city and all other cities.

- **Algorithm**

  Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph. The term optimal can anything, shortest, cheapest, or fastest. If we start the algorithm with an initial node, then the distance of a node Y can be given as the distance from the initial node to that node.

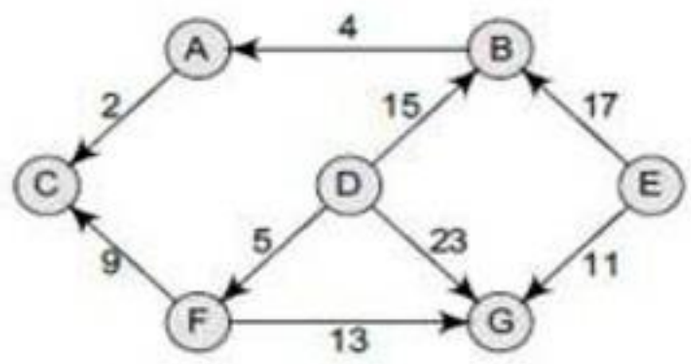
```
1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest
   path has been found.
3. Label the initial node with 0, and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more
   labelled nodes in N.
5. Consider each node that is not in N and is connected by an edge from
   the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the
       node = the label of the newly inserted node + the length of the edge.
   (b) Else if the node that is not in N was already labelled, then SET its new
       label = minimum (label of newly inserted vertex + length of edge, old
       label)
7. Pick a node not in N that has the smallest label assigned to it and add it
   to N.
```

Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node. There are two kinds of labels: temporary and permanent. Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.

The execution of this algorithm will produce either of the following two results:

- If the destination node is labelled, then the label will in turn represent the distance from the source node to the destination node.
- If the destination node is not labelled, then there is no path from the source to the destination node.

Example: Consider the graph G given in Figure Taking D as the initial node, execute **the Dijkstra's algorithm on it.**



Figu Graph G

Step 1: Set the label of D = 0 and N = {D}.
Step 2: Label of D = 0, B = 15, G = 23, and F = 5. Therefore, N = {D, F}.
Step 3: Label of D = 0, B = 15, G has been re-labelled 18 because minimum (5 + 13, 23) = 18, C has been re-labelled 14 (5 + 9). Therefore, N = {D, F, C}.
Step 4: Label of D = 0, B = 15, G = 18. Therefore, N = {D, F, C, B}.
Step 5: Label of D = 0, B = 15, G = 18 and A = 19 (15 + 4). Therefore, N = {D, F, C, B, G}.
Step 6: Label of D = 0 and A = 19. Therefore, N = {D, F, C, B, G, A}

Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that are in N are reachable from B.

The running time of Dijkstra's algorithm can be given as $O(|V|^2+|E|)=O(|V|^2)$ where V is the set of vertices and E in the graph.

9.8 Warshall's Algorithm

If a graph G is given as G=(V, E), where V is the set of vertices and E is the set of edges, the path matrix of G can be found as, $P = A + A^2 + A^3 + \ldots + A^n$. This is a lengthy process, so Warshall has given a very efficient algorithm to calculate the path matrix. Warshall's algorithm defines matrices $P_0, P_1, P_2, \ldots, P_n$ as given in Figure A

$P_k[i][j]$ → 1 [if there is a path from v_i to v_j. The path should not use any other nodes except $v_1, v_2, \ldots, v_k$]

$P_k[i][j]$ → 0 [otherwise]

Figure A Path matrix entry

This means that if $P_0[i][j] = 1$, then there exists an edge from node v_i to v_j.

If $P_1[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1.

If $P_2[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1 and v_2.

Note that P_0 is equal to the adjacency matrix of G. If G contains n nodes, then $P_n = P$ which is the path matrix of the graph G.

From the above discussion, we can conclude that $P_k[i][j]$ is equal to 1 only when either of the two following cases occur:

- There is a path from v_i to v_j that does not use any other node except $v_1, v_2, \ldots, v_{k-1}$. Therefore, $P_{k-1}[i][j] = 1$.
- There is a path from v_i to v_k and a path from v_k to v_j where all the nodes use $v_1, v_2, \ldots, v_{k-1}$. Therefore,

$$P_{k-1}[i][k] = 1 \text{ AND } P_{k-1}[k][j] = 1$$

Hence, the path matrix P_n can be calculated with the formula given as:

$$P_k[i][j] = P_{k-1}[i][j] \vee (P_{k-1}[i][k] \wedge P_{k-1}[k][j])$$

where V indicates logical OR operation and Λ indicates logical AND operation.

Figure 13.38 shows the Warshall's algorithm to find the path matrix P using the adjacency matrix A.

```
Step 1: [INITIALIZE the Path Matrix] Repeat Step 2 for I = 0 to n-1,
        where n is the number of nodes in the graph
Step 2:     Repeat Step 3 for J = 0 to n-1
Step 3:           IF A[I][J] = 0, then SET P[I][J] = 0
                  ELSE P[I][J] = 1
            [END OF LOOP]
      [END OF LOOP]
Step 4: [Calculate the path matrix P] Repeat Step 5 for K = 0 to n-1
Step 5:     Repeat Step 6 for I = 0 to n-1
Step 6:           Repeat Step 7 for J=0 to n-1
Step 7:                       SET P_K[I][J] = P_K-1[I][J] V (P_K-1[I][K]
                              Λ P_K-1[K][J])
Step 8: EXIT
```

Example: Consider the graph in Figure B and its adjacency matrix A. We can straightaway calculate the path matrix P using the Warshall's algorithm.

The path matrix P can be given in a single step as:

$$P = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

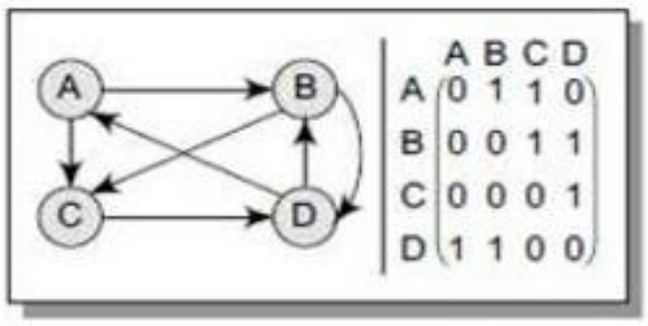


Figure B Graph G and its path matrix P

Thus, we see that calculating A, A2, A3, A4, . . . , A5 to calculate P is a very slow and inefficient technique as compared to the Warshall's technique.

9.9 Modified Warshall's Algorithm

Warshall's algorithm can be modified to obtain a matrix that gives the shortest paths between the nodes in a graph G. As an input to the algorithm, we take the adjacency matrix A of G and replace all the values of A which are zero by infinity (∞). Infinity (∞) denotes a very large number and indicates that there is no path between the vertices. In Warshall's modified algorithm, we obtain a set of matrices $Q_0, Q_1, Q_2, \ldots, Q_n$ using the formula given below.

$$Q_k[i][j] = \text{Minimum}(\ M_{k-1}[i][j],\ M_{k-1}[i][k] + M_{k-1}[k][j])$$

Q_0 is exactly the same as A with a little difference that every element having a zero value in A is replaced by (∞) in Q_0. Using the given formula, the matrix Q_n will give the path matrix that has the shortest path between the vertices of the graph.

```
Step 1: [Initialize the Shortest Path Matrix, Q] Repeat Step 2 for I = 0
        to n-1, where n is the number of nodes in the graph
Step 2:     Repeat Step 3 for J = 0 to n-1
Step 3:           IF A[I][J] = 0, then SET Q[I][J] = Infinity (or 9999)
                  ELSE Q[I][J] = A[I][j]
            [END OF LOOP]
      [END OF LOOP]
Step 4: [Calculate the shortest path matrix Q] Repeat Step 5 for K = 0
        to n-1
Step 5:     Repeat Step 6 for I = 0 to n-1
Step 6:           Repeat Step 7 for J=0 to n-1
Step 7:                     IF Q[I][J] <= Q[I][K] + Q[K][J]
                            SET Q[I][J] = Q[I][J]
                      ELSE SET Q[I][J] = Q[I][K] + Q[K][J]
                      [END OF IF]
            [END OF LOOP]
      [END OF LOOP]
      [END OF LOOP]
Step 8: EXIT
```

9.10 Applications of Graphs

Graphs are constructed for various types of applications such as:

- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.
- In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- Graphs are also used to draw activity network diagrams. These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

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 di!cult 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 maximumsize 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 di!cult 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.

10.6 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(nc) Where c >1
5. Exponential time O(cn) 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 N2 . Eg. Most sorting algorithms have growth rates of Nlog(N) or N2 . 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

10.6.1 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
        }
}
```

$$\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}1=\sum_{i=0}^{N-1}N=N\sum_{i=0}^{N-1}1=N^2$$

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(N2).

```
for (i=1; i<=N; i++)
{
        for (j=0; j< i; j++)
        {
                sequence of simple statement
        }
}
```

$$\sum_{i=1}^{N}\sum_{j=0}^{i-1}1=\sum_{i=1}^{N}i=\frac{N(N+1)}{2}=\frac{N^2}{2}+\frac{N}{2}$$

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+……(N-1)+N = N(N+1) / 2 = ((N2) +N)/2 Therefore the performance is O(N2).

Generalization: A structure with k nested counting loops where the counter is just incremented or decrement by one has performance O(Nk) if the trip counts depends on the problem size only.

Note:

- **Extra:**

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm.

O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements)
{
    return elements[0] == null;
}
```

O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(IList<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```

O(N^2)

O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N3), O(N4) etc.

```
bool ContainsDuplicates(IList<string> elements)
{
    for (var outer = 0; outer < elements.Count; outer++)
    {
        for (var inner = 0; inner < elements.Count; inner++)
        {
            // Don't compare with self
            if (outer == inner) continue;

            if (elements[outer] == elements[inner]) return true;
        }
    }

    return false;
}
```

nareshpad.com

$O(2^N)$

$O(2^N)$ denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an O(2N) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2N) function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number)
{
   if (number <= 1) return number;

   return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

Logarithms

Logarithms are slightly trickier to explain so I'll use a common example:

Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

This type of algorithm is described as O(log N). The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

HEURISTIC AND APPROXIMATE ALGORITHMS

A heuristic is a much broader term than approximation algorithm.
An absolute approximation algorithm as the name suggests has an absolute error with respect to the objective value of an optimal solution. The important other property is that it terminates in polynomial time with respect to the input size.
A heuristic does not need these two properties. A heuristic is a vague term that can capture a ``sometimes right, sometimes wrong" approach of approximating a solution (may not even be an algorithm) or an approach of designing algorithms that may not have any properties other than giving you a feasible solution . There are lots of types of heuristic algorithms, even for approximation algorithms.

SERIES AND PARALLEL ALGORITHM

In computer science, a sequential algorithm or serial algorithm is an algorithm that is executed sequentially – once through, from start to finish, without other processing executing

In computer science, a parallel algorithm, as opposed to a traditional 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

The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cy-cles) that they take. Parallel algorithms need to optimize one more resource, the communication between differ-ent processors. There are two ways parallel processors communicate, shared memory or message passing.

DIVIDE AND CONQUER ALGORITHM

In computer science, divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.
This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort)