

## Unit- 1

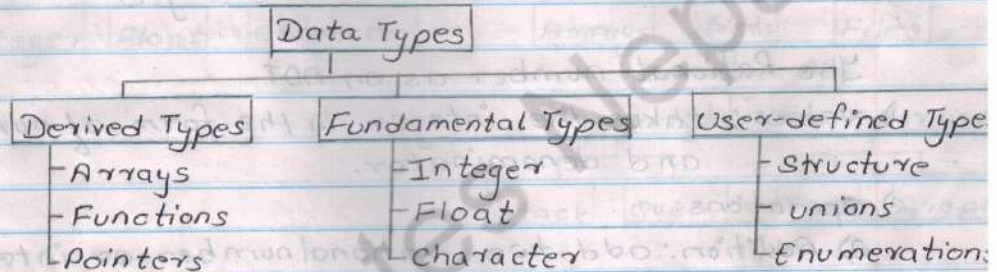
### ~~Complexity Analysis~~

#### Data:

Data may be defined as raw facts or figures or observations about physical phenomena.

#### Data Types:

The data type is a method to interpret the data. Basically data types are divided into three categories.



#### Abstract Data Types (ADT):

A useful tool for specifying the logical properties of a data type is called the abstract data types. Set of integer together with operation of union, intersect and set difference from a simple example of ADT. It may not even be possible to implement a particular on a particular software system.

An ADT have both the set of values of a give type and the set of operations that can be performed on those values.

The important principle for the definition of ADT involves two operations:

- 1) Description of the way in which the component are



related to each other.

2. Statement of the operation that can be performed on the elements of the ADT.

For eg, array can be considered as an ADT, we have the following operations:

- 1) Values: ordered list of elements.

- 2) Operations:

- a) Insertion: add the new element to an array.

- b) Deletion: delete an element from an array.

- c) Traverse: access each and every element of array.

The Rational number as an ADT

- 1) Values: takes the integer in the form of numerator and denominator.

- 2) Operations:

- a) Addition: add two rational numbers or integer.

- b) Subtraction:

- c) Multiplication:

- d) Division:

- e) Exponentiation:

Data structure:

Data can be organized in many ways; the logical or mathematical model of particular organization of data is known as the data structure. The selection of a particular data model depends upon two factors.

- a) The data model should be rich enough in structure to show the actual relationships of the data in the real world.

- b) The structure should be simplest so that one can



them for an effective data processing when required.

Example: Array, stack, Queue, list, etc.

Data structures are normally divided into two

Categories: i) Primitive

ii) Non-Primitive

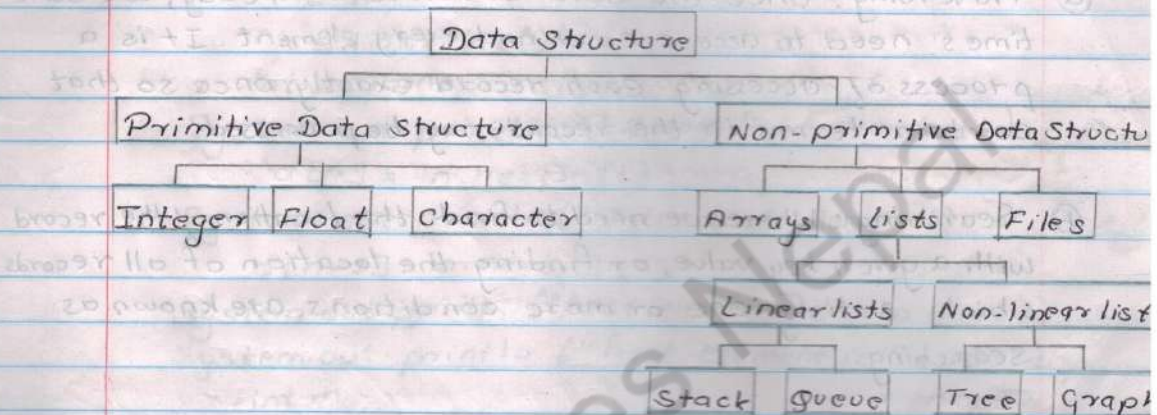


Fig: classification of data structure.

i) Primitive Data structure:

These are basic structures and are directly operated up by the machine instructions. For eg: integer, float, etc.

ii) Non-Primitive Data structure:

These are more sophisticated data structures. These are derived from the primitive data structures. The non-primitive data structures emphasize on structuring of group of homogeneous or heterogeneous data items. For eg: Array, structure, lists, etc.

### Data Structure Operations:

The data assigned to our data structure can be processed by means of certain operations. In general there are four operations that come into play when working on data structure.

- (a) Traversing: Once the data structure is ready, we sometimes need to access each and every element. It is a process of accessing each record exactly once so that certain terms in the record may be processed.
- (b) Searching: Where we need to find the location of the record with a given key value, or finding the location of all records which satisfy one or more conditions are known as Searching.
- (c) Insertion: When we add new record to the structure.
- (d) Deletion: It is the process of delete on existing record from the structure.

### Algorithm



```

import java.util.Scanner;
class NewArray {
    public static void main (String args[]) {
        Scanner in = new Scanner (System.in);
        System.out.println ("Enter size of array");
        int n;
        n = in.nextInt();
        int a[] = new int[n];
        System.out.println ("Enter first element");
        a[0] = in.nextInt();
        System.out.println ("Enter last element");
        a[n-1] = in.nextInt();
        System.out.println ("last element is " + a[n-1]);
        System.out.println ("first element is " + a[0]);
        int m;
        System.out.println ("Enter Index of array");
        m = in.nextInt();
        System.out.println ("Enter element in " + m + " index");
        a[m] = in.nextInt();
        System.out.println (m + "index has " + a[m] + "element");
    }
}

```



### Array Reverse

```
import java.util.Scanner;
class Reverse {
    public static void main (String args[])
    {
        int i, n;
        Scanner in = new Scanner (System.in);
        System.out.println ("Enter element:");
        n = in.nextInt();
        int a[] = new a[n];
        System.out.println ("Enter all element:");
        for (i = 0; i < n; i++)
        {
            a[i] = in.nextInt();
        }
        System.out.println ("Array after reversing:");
        for (i = n-1; i >= 0; i--)
        {
            System.out.println (a[i]);
        }
    }
}
```

### Insertion:

```
import java.util.Scanner;
public class Insert {
    public static void main (String args[]) {
        int n, pos, x;
        Scanner in = new Scanner (System.in);
        System.out.println ("Enter no. of element:");
        n = in.nextInt();
```



### Insert:

```
int a[] = new int [n+1];
System.out.println("Enter all element:");
for (int i=0 ; i<n ; i++){
    a[i] = in.nextInt();
}
System.out.println("Enter position to insert");
pos = in.nextInt();
System.out.println("Enter element you want");
x = in.nextInt();
for (int i = (n-1) ; i >= (pos-1) ; i--) {
    a[i+1] = a[i];
}
a[pos-1] = x;
System.out.println("After inserting");
for (int i=0 ; i<n ; i++){
    System.out.println(a[i] + " , ");
}
System.out.println(
}
}
```

### Deletion:

```
import java.util. Scanner;
public class Delete {
    public static void main(String args []){
        int n, pos;
        Scanner in = new Scanner (System.in);
        System.out.println("Enter size of array");
        n = in.nextInt();
        int a[] = new int [n];
```



```
System.out.println("Enter all element");  
for (int i = 0; i < n; i++) {  
    a[i] = in.nextInt();  
}  
System.out.println("Enter position you want to delete");  
pos = in.nextInt();  
try {  
    for (int i = (pos-1); i < n; i++) {  
        a[i] = a[i+1];  
    }  
} catch (Exception e) {  
    System.out.println("Error");  
}  
System.out.println("After deleting");  
for (int i = 0; i < n-1; i++) {  
    System.out.print(a[i] + ", ");  
}  
}
```

## Complexity Analysis

Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is.

When algorithm is developed it must be analyzed, an algorithm is analyzed in terms of its complexity. It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of time/space requirements as a function of the input size.

Thus we have the notion of:

- Time Complexity: Running time of the program as a function of the size of input.

Space Complexity: Amount of computer memory required during the program execution as a function of input size.

For example: Algorithm to find factorial of number:

fact

Step 1: Start

Step 2: Input fact = 1 and  $i = 1$

Step 3: If  $i \leq n$  go to step 4 else go to step 6.

Step 4: Multiply fact with  $i$

Step 5: Increment  $i$  and go to step 3.

Step 6: return the value of fact.

Step 7: End

```
fact=1;
```

```
for(i=1; i<=n; i++)
```

```
    fact = fact * i;
```

```
return fact
```



From the above factorial algorithm:

- First assignment statement executes 1 times.
- Loop executes  $n+1$  times
- Body inside loop executes  $n$  times.
- Return statements executes 1 times.

So, total complexity of algorithm is

$$T(n) = 1 + n + 1 + n + 1$$

$$= 2n + 3$$

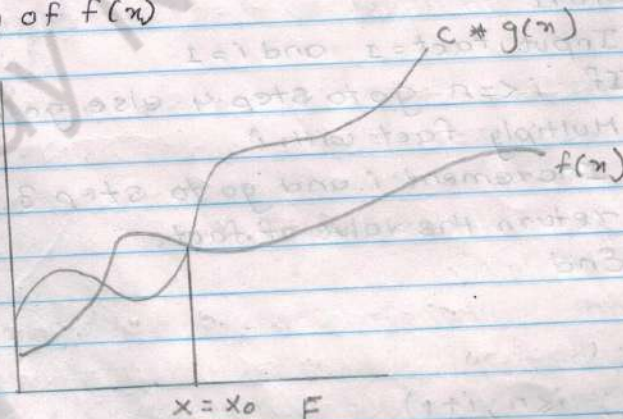
i.e. linear time

Growth of function.

i) Big oh ( $O$ ) notation:

When we have only asymptotic upper bound then we use  $O$  notation. A function  $f(x) = O(g(x))$  iff there exists two positive constants  $c$  and  $x_0$  such that for all  $x \geq x_0$ ,  $0 \leq f(x) \leq c * g(x)$ .

The above relation says that  $g(x)$  is an upper bound of  $f(x)$ .



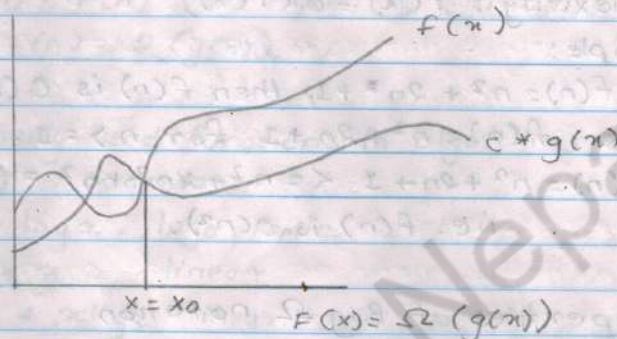
The figure above shows that for all value of  $x \geq x_0$ , the function  $f(x)$  lies below the function  $g(x)$ . So,  $g(x)$  is upper bound of  $f(x)$ .



### ii) Big omega ( $\Omega$ ) notation

Big omega notation gives asymptotic lower bound. A function  $f(x) = \Omega(g(x))$  iff there exists two positive constants  $c$  and  $x_0$  such that for all  $x \geq x_0$ ,  $0 < c * g(x) \leq f(x)$ .

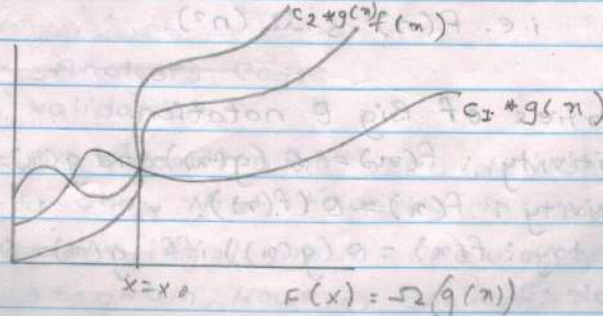
3th



The figure above shows that for all values of  $x \geq x_0$ , the function  $f(x)$  lies above the function  $g(x)$ . So  $g(x)$  is lower bound of  $f(x)$ .

### iii) Big Theta ( $\Theta$ ) notation

When we need asymptotically tight bound then we use  $\Theta$  notation. A function  $f(x) = \Theta(g(x))$  iff there exist three positive constants  $c_1$ ,  $c_2$  and  $x_0$  such that for all  $x \geq x_0$ ,  $0 < c_1 * g(x) \leq f(x) \leq c_2 * g(x)$ .



For all values of  $n \geq n_0$ , plot shows clearly that  $f(n)$  lies between  $c_1 * g(n)$  and  $c_2 * g(n)$ .



### Properties of Big O notation

1) Transitivity :  $f(x) = O(g(x))$  and

$$g(x) = O(h(x))$$

$$\Rightarrow f(x) = O(h(x)).$$

2) Reflexivity :  $f(x) = O(f(x))$

Example:

Let  $f(n) = n^2 + 2n + 1$ , then  $f(n)$  is  $O(n^2)$

$\Rightarrow$  Since,  $f(n) = n^2 + 2n + 1$  for  $n \geq 1$ .

$$f(n) = n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$$

i.e.  $f(n)$  is  $O(n^2)$ .

### Properties of Big $\Omega$ notation

1) Transitivity :  $f(x) = \Omega(g(x))$  and

$$g(x) = \Omega(h(x))$$

$$\Rightarrow f(x) = \Omega(h(x))$$

2) Reflexivity :  $f(x) = \Omega(f(x))$

Example:

Let  $f(n) = n^2 + 2n + 1$ , then  $f(n)$  is  $\Omega(n^2)$

$\Rightarrow$  Since,  $f(n) = n^2 + 2n + 1$ , for  $n \geq 1$ .

$$f(n) = n^2 + 2n + 1 \geq n^2$$

i.e.  $f(n)$  is  $\Omega(n^2)$ .

### Properties of Big $\Theta$ notation

1) Transitivity :  $f(x) = \Theta(g(x))$  and  $g(x) = \Theta(h(x)) \Rightarrow f(x) = \Theta(h(x))$

2) Reflexivity :  $f(x) = \Theta(f(x))$ .

3) Symmetry :  $f(x) = \Theta(g(x))$  iff  $g(x) = \Theta(f(x))$ .

Example:

$$\text{Let } f(n) = 3n^2 + 4n + 7,$$

$$g(n) = n^2, \text{ the prove that } f(n) = \Theta(g(n))$$



Proof: Let us choose  $C_1$  and  $C_2$  and no. values 14, 1 and 1 respectively then we can write,

$$f(n) \leq C_1 * g(n), n \geq n_0 \text{ as } 3n^2 + 4n + 7 \leq 14 * n^2 \text{ and}$$

$$f(n) \geq C_2 * g(n), n \geq n_0 \text{ as, } 3n^2 + 4n + 7 \geq 1 * n^2$$

for all  $n \geq 1$  (in both cases)

$C_2 * g(n) \leq f(n) \leq C_1 * g(n)$  is trivial.

Hence,  $f(n) = O(g(n))$ .

Some commonly used functions in algorithms complexity

$$f(x) = O(1) \quad \text{constant}$$

$$f(x) = c * \log x \quad \text{logarithmic}$$

$$f(x) = c * x \quad \text{linear}$$

$$f(x) = c * x \log x \quad \text{linearithmic}$$

$$f(x) = c * x^2 \quad \text{quadratic}$$

$$f(x) = c * x^3 \quad \text{cubic}$$

$$f(x) = c * x^k \quad \text{polynomial int.}$$

$$f(x) = c * k^x \quad \text{exponential int.}$$

Order of growth

$$O(1) < c * \log(x) < c * x < c * x \log x < c * x^2 < c * x^3 < c * x^k \quad (k > 3) < c * k^x \quad (k > 1) \text{ value of } k \text{ should be in increasing order.}$$

Complexity Analysis cases

Algorithm validation:

Once an algorithm has been devised it becomes necessary to show that it works in computer correct to all possible, legal input. One way is to code into computer program. However converting an algorithm into computer program is a time consuming process.



Hence it is essential to be reasonably sure about the effectiveness of the algorithm before it is coded. The process, at the algorithm level, is called "validation".

Algorithm:

An algorithm is a finite set of instructions executed in finite time for problem solving or performing computation. To represent an algorithm we use pseudo code that can represent an algorithm in clear manner like in English language and gives that the implementation view as in the programming language.

Properties of an Algorithm:

- 1) Input/output : An algorithm has input or set of input values from the set that has possible input values for each set of inputs an algorithm produces the solution of the problem that are in the set of output values.
- 2) Definiteness : Each step must be clear and unambiguous.
- 3) Correctness : An algorithm produced output must be correct for each set of input values.
- 4) Finiteness : The algorithm must terminate after finite amount of time for every possible set of input values.
- 5) Effectiveness : Each step must be executable in finite time.



### Complexity Analysis case:

- Best case Complexity gives lower bound on the running time of the algorithm for any instance of input(s). It indicates that the algorithm can never have lower running time than best case for particular class of problem.
- Worst case complexity gives upper bound on the running time of the algorithm for all the instances of the input(s). This insures that no input can overcome the running time limit posed by worst case complexity.
- Average case complexity gives average number of steps required on any instance of the input(s).

### \* Amortized Complexity

Here, the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation may be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

### • Complexity classes:

#### • Class P (Polynomial)

The class of decision problems that have polynomial-time deterministic algorithm.



- class NP: (Non-deterministic Polynomial)

The class of decision problems that are solvable in polynomial time on a non-deterministic algorithms.

- NP-Hard class

A problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP-problem.

- NP-complete:

In computational complexity theory, a decision problem is NP-complete when it is both in NP and NP-hard.

\* ~~Stack~~



### Unit-3

### Stacks and Queues

#### Stack

- A stack is an ordered collection of items in which new items are inserted and from which items are deleted at one end, called the top of the stack (TOS).

|  |   |                |
|--|---|----------------|
|  | E | ← Top of stack |
|  | D |                |
|  | C |                |
|  | B |                |
|  | A |                |

Fig: Stack containing stack items

A good real-life example of a stack is the pile of dinner plates that you encounter when you eat at the cafeteria. When you remove a plate from the pile, you take the plate on the top of pile. But this is exactly the plate that was added (inserted) most recently to the pile by the dishwasher. If you want the plate at the bottom of the pile, you must remove all the plates on top of it to reach it.

In stack, the elements are removed only in the inverse order of their arrival i.e. the last inserted element is removed first of all, so it has last in first out (LIFO) behavior.

#### Primitive operation on stack

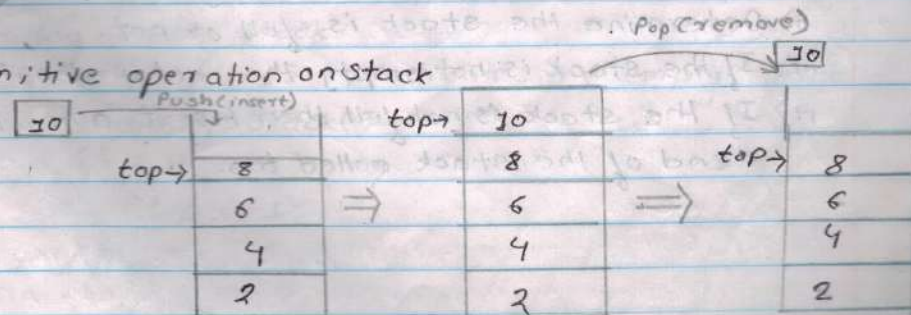


Fig: Representation of Push & Pop operation on stack



- i)  $\text{Push}(S, x)$ : Adds the item  $x$  to the top of stack
- ii)  $x = \text{pop}(s)$ : Removes the top element and return it as the function's value.

If stack contains a single item and the stack is popped, the resulting back contains no items called the empty stack. The result of an illegal attempt to Pop or access an item from an empty stack is called stack underflow.

- iii)  $\text{Empty}(s)$ : If the stack is empty,  $\text{Empty}(s)$  returns the value TRUE; otherwise FALSE.

The stack as an ADT

An ADT is a set of values, well defined set of properties of the values and set of operations needed for the processing of these values. A stack can be expressed as an ADT as:

- i) Values: Order list of items.
- a) Properties: zero or more items, items are added and deleted from the one end called the top of stack.
- 3) Operations:
- i) Determine the stack is empty or not.
  - ii) Determine the stack is full or not.
  - iii) If the stack is not empty then retrieve the item at its top.
  - iv) If the stack is not full then insert a new item at one end of the stack called top.



## Stack Implementation:

Stack can be implemented in two ways:

1) Static Implementation (Array implementation): It uses arrays to create stack. Static implementation though a very simple technique but is not a flexible way of creation, as the size of stack has to be declared during program design, after that the size cannot be varied. Moreover, static implementation is not too efficient w.r.t memory utilization.

2) Dynamic implementation (Linked list representation)

- Defining class for stack

```
class Stack
```

```
{
```

```
    final int MAXSIZE = 20;
```

```
    public int item[] = new int [MAXSIZE];
```

```
    public int top;
```

```
    public Stack()
```

```
    {
```

```
        top = -1;
```

```
    }
```

```
}
```

- Creating Empty stack

The value of  $top = -1$ , indicates the empty stack.

```
void emptyStack()
```

```
{
```

```
    top = -1;
```

```
}
```



- **Stack 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  $top = -1$ , indicates the empty stack.

The following method returns 1 if the stack is empty, 0 otherwise.

```
boolean Isempty()
```

```
{
```

```
    return top == -1;
```

```
}
```

- **Stack full or overflow**:

This is the situation when the stack becomes full and no more element can be added into the stack.

At this point the stack top is present at the highest location ( $MAX\ SIZE - 1$ ) of the stack.

The following method return 1, if the stack is full, otherwise 0.

```
boolean IsFull()
```

```
{
```

```
    return top == MAXSIZE - 1;
```

```
}
```

- **Implementing the push operation**:

Algorithm for push operation:

Step 1: If  $top = MAXSIZE - 1$ ; then

Step 2: Print the message stack overflow & exit.

Step 3: else

Step 4: [Increment top by 1] i.e. set  $top = top + 1$ ;

Step 5: [Insert item in new top] set  $stack(top) \leftarrow item$

Assumption:  $MAXSIZE = size of array$ .

top: top of stack

item: data value to be inserted



top = max-1  
unl  
else  
top = top + 1  
s.item[s.top] = ele

# Method for push operation:

We can define the push method as:

```
public void push (int element)
{
    if (!s.top++ || (!s.ISFull())) { s.top++;
        s.item[s.top] = element;
    }
    else {
        System.out.println("Stack Overflow");
    }
}
```

## Implementing the pop operation.

Algorithm for pop operation:

Step 1: If top = -1; then

Step 2: Print the message stack underflow & exit

Step 3: return s.item[top] extract item from top

Step 4: Decrement top by 1 i.e. top = top - 1;

## Method for pop operation:

```
public void pop () { int element;
{
    if (s.top == -1) {
        System.out.println("stack underflow");
    }
    else {
        int element = s.item[s.top];
        s.item[s.top] = 0;
        s.top--;
        return element;
    }
    System.out.println("Popped element is " + element);
}
```



- The Infix, Prefix, Postfix Notation:

Application of stack:

- 1) To ~~prevail~~ print characters / string in reverse order
- 2) Check the parenthesis in the expression.
- 3) To evaluate the arithmetic expressions such as infix, prefix and postfix.

Arithmetic expression

An expression is defined as a number of operands or data items combined using several operators. There are basically three types of notations for an expression

- 1) Infix notation
- 2) Prefix notation
- 3) Postfix notation

- 1) Infix notation:

It is most common notation in which, the operator is written or placed in-between the two operands.

For example: The expression to add two numbers A and B is written in infix notation as,

$$\begin{array}{c} \swarrow \quad \searrow \\ A + B \\ \uparrow \\ \text{operator} \end{array}$$

In this example, the operator is placed in-between the operands A and B. The reason why this notation is called infix.

- 2) Prefix Notation:

It is also called polish notation, named after in honor of the mathematician Jan Łukasiewicz, refe



to the notation in which the operator is placed before the operands as  $+AB$ .

As the operator '+' is placed before the operands A and B, this notation is called prefix.

### 3) Postfix notation:

In the postfix notation the operator are written after the operand, so it is called the postfix notation. It is also known as suffix notation or reverse polish notation. The given infix notation can be represented in postfix notation as  $AB+$ .

#### Operator Precedence

| Operator                | Symbol | Precedence   |
|-------------------------|--------|--------------|
| Exponential             | $\$$   | Highest      |
| Multiplication/Division | $*, /$ | Next highest |
| Addition/Subtraction    | $+, -$ | Lowest       |

### • Conversion of Infix expression to postfix expression Algorithm

Let, opstack is an operator stack, P is an expression written in postfix notation and tos is top of stack of opstack.

Step 1: Scan one character at a time of an infix expression from left to right.

Step 2: opstack = an empty stack

Step 3: Repeat till there is data in infix expression

3.1 if scanned character is 'c' then push it to opstack

3.2 if scanned character is operand then push it to P.

3.3 if scanned character is operator then,



while (precedence (top char) > precedence (scan char))  
 pop and push it into P  
 otherwise,  
 push scanned character into opstack.  
 3.4: if scanned character is ')' then pop and push  
 P until '(' is not found and ignore both.  
 Step 4: Pop and push into P until opstack is not empty  
 Step 5: Stop

### Trace of Conversion Algorithm

Example: Convert the following infix expression into postfix expression.

Soln.  $A - B / (C * D \$ E)$

| S.N. | Symbol scan | opstack    | P (Postfix expression) |
|------|-------------|------------|------------------------|
| 1.   | A           | ....       | A                      |
| 2.   | -           | -          | A                      |
| 3.   | B           | -          | AB                     |
| 4.   | /           | - /        | AB                     |
| 5.   | (           | - / (      | AB                     |
| 6.   | C           | - / (      | ABC                    |
| 7.   | *           | - / ( *    | ABC                    |
| 8.   | D           | - / ( *    | ABCD                   |
| 9.   | \$          | - / ( * \$ | ABCD                   |
| 10.  | E           | - / ( * \$ | ABCDE                  |
| 11.  | )           | - / ( *    | ABCDE\$                |
| 12.  | ...         | - / (      | ABCDE\$*               |
| 13.  | ...         | - /        | ABCDE\$*               |
| 14.  | ...         | -          | ABCDE\$*/              |
| 15.  | ...         | ...        | ABCDE\$*/-             |

∴ The final postfix expression is  $ABCDE$*/-$



Example: Convert the following infix expression into postfix expression:

$$\text{Sol}^n ((A - (B + C)) * D) \$ (E + F)$$

| S.N. | Symbol Scqn | opstack | P(Postfix expression) |
|------|-------------|---------|-----------------------|
| 1.   | (           | (       | ...                   |
| 2.   | (           | ((      | ...                   |
| 3.   | + A         | ((      | A                     |
| 4.   | -           | (( -    | A                     |
| 5.   | (           | (( - (  | A                     |
| 6.   | B           | (( - (  | AB                    |
| 7.   | +           | (( - (+ | AB                    |
| 8.   | C           | (( - (+ | ABC                   |
| 9.   | )           | (( - (  | ABC+                  |
| 10.  | )           | (( -    | ABC+                  |
| 11.  | *           | (( *    | ABC+ -                |
| 12.  | D           | (( *    | ABC+ - D              |
| 13.  | )           | ((      | ABC+ - D *            |
| 14.  | \$          | \$      | ABC+ - D *            |
| 15.  | (           | \$(     | ABC+ - D *            |
| 16.  | E           | \$(     | ABC+ - D * E          |
| 17.  | +           | \$( +   | ABC+ - D * E          |
| 18.  | F           | \$( +   | ABC+ - D * EF         |
| 19.  | )           | \$(     | ABC+ - D * EF +       |
| 20.  | ...         | \$      | ABC+ - D * EF +       |
| 21.  | ...         | ...     | ABC+ - D * EF + \$    |

∴ The final postfix expression is  $AB+C-D*EF+$



Q.  $(A+B) * (C-D)$

| S.N. | Symbol scan | opstack | P (Postfix expression) |
|------|-------------|---------|------------------------|
| 1    | (           | (       | ...                    |
| 2    | A           | (       | A                      |
| 3    | +           | (+      | AB                     |
| 4    | B           | (+      | AB+                    |
| 5    | )           | *       | AB+                    |
| 6    | *           | *(      | AB+                    |
| 7    | (           | *(      | AB+C                   |
| 8    | C           | *(      | AB+C                   |
| 9    | -           | *(-     | AB+CD                  |
| 10   | D           | *(-     | AB+CD                  |
| 11   | )           | *{      | AB+CD-                 |
| 12   | ...         | ...     | AB+CD-*                |

∴ The final postfix expression is  $AB+CD-*$

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

| S.No. | Symbol scan | opstack | P (Postfix expression) |
|-------|-------------|---------|------------------------|
| 1     | (           | (       | ...                    |
| 2     | (           | ((      | ...                    |
| 3     | A           | ((      | A                      |
| 4     | +           | ((+     | AB                     |
| 5     | B           | ((+     | AB+                    |
| 6     | )           | (       | AB+                    |
| 7     | *           | (*      | AB+C                   |
| 8     | C           | (*      | AB+C                   |
| 9     | -           | (-      | AB+C*                  |
| 10    | (           | (- (    | AB+C*                  |
| 11    | D           | (- (    | AB+C*D                 |
| 12    | -           | (- (-   | AB+C*D                 |



|     |     |      |                |
|-----|-----|------|----------------|
| 13. | E   | (-(- | AB+C*DE        |
| 14. | )   | (-   | AB+C*DE-       |
| 15. | )   |      | AB+C*DE--      |
| 16. | \$  | \$   | AB+C*DE--      |
| 17. | (   | \$(  | AB+C*DE--      |
| 18. | F   | \$(  | AB+C*DE--F     |
| 19. | +   | \$(+ | AB+C*DE--F     |
| 20. | G   | \$(+ | AB+C*DE--FG    |
| 21. | )   | \$   | AB+C*DE--FG+   |
| 22. | ... | ...  | AB+C*DE--FG+\$ |

∴ The final postfix expression is  $AB+C*DE--FG+$$

Evaluating a postfix expression:

Algorithm for evaluating postfix expression.

Let P is an expression written in postfix expression

- 1) Stack = empty stack
- 2) Scan P from left to right and repeat step 3. and 4 for each symbol in P until end of expression.
- 3) If an operand is encountered, push it into stack.
- 4) If an operator (X) is encountered then,
  - a) operand 2 = Pop (stack)
  - b) operand 1 = Pop (stack)
  - c) Value = operand 1 (X) operand 2.
  - d) Push value into stack.
- 5) Return the value at top of the stack.
- 6) Exit.

[Note : (X) = +, -, \*, /, \$]



Tracing of algorithm.

$$P = 623 + - 382 / + * 2 \$ 3 +$$

| S.N. | Symbol scan | operand1 | operand2 | value | Stack |
|------|-------------|----------|----------|-------|-------|
|      |             |          |          |       | 6     |
| 1.   | 6           | ...      | ...      | ...   | 62    |
| 2.   | 2           | ...      | ...      | ...   | 623   |
| 3.   | 3           | ...      | ...      | ...   | 65    |
| 4.   | +           | 2        | 3        | 5     | 1     |
| 5.   | -           | 6        | 5        | 1     | 13    |
| 6.   | 3           |          |          |       | 138   |
| 7.   | 8           |          |          |       | 1382  |
| 8.   | 2           |          |          |       | 134   |
| 9.   | /           | 8        | 2        | 4     | 17    |
| 10.  | +           | 3        | 4        | 7     | 7     |
| 11.  | *           | 1        | 7        | 7     | 72    |
| 12.  | 2           |          |          |       | 149   |
| 13.  | \$          | 7        | 2        |       | 49.3  |
| 14.  | 3           |          |          |       | 52    |
| 15.  | +           | 49       | 3        | 52    |       |

Q. Evaluate the following postfix expressions:

1)  $AB + C - BA + C \$ -$

2)  $ABC + * CBA - + *$

where  $A=1, B=2, C=3$ .



i)  $12+3-21+3\$-$

| S.N. | Symbol scan | operand 1 | operand 2 | value | Stack |
|------|-------------|-----------|-----------|-------|-------|
| 1.   | 1           |           |           |       | 1     |
| 2.   | 2           |           |           |       | 1 2   |
| 3.   | +           | 1         | 2         | 3     | 3     |
| 4.   | 3           |           |           |       | 3 3   |
| 5.   | -           | 3         | 3         | 0     | 0     |
| 6.   | 2           |           |           |       | 0 2   |
| 7.   | 1           |           |           |       | 0 2 1 |
| 8.   | +           | 2         | 1         | 3     | 0 3   |
| 9.   | 3           |           |           |       | 0 3 3 |
| 10.  | \$          | 3         | 3         | 927   | 0 927 |
| 11.  | -           | 0         | 927       | -27   | -27   |

ii)  $123+*321-+*$

| S.N. | Symbol scan | operand 1 | operand 2 | value | Stack   |
|------|-------------|-----------|-----------|-------|---------|
| 1.   | 1           |           |           |       | 1       |
| 2.   | 2           |           |           |       | 1 2     |
| 3.   | 3           |           |           |       | 1 2 3   |
| 4.   | +           | 2         | 3         | 5     | 1 5     |
| 5.   | *           | 1         | 5         | 5     | 5       |
| 6.   | 3           |           |           |       | 5 3     |
| 7.   | 2           |           |           |       | 5 3 2   |
| 8.   | 1           |           |           |       | 5 3 2 1 |
| 9.   | -           | 2         | 1         | 1     | 5 3 1   |
| 10.  | +           | 3         | 1         | 4     | 5 4     |
| 11.  | *           | 5         | 4         | 20    | 20      |



$$-28 + 12 - 2 + 21$$

## Conversion from Infix to Prefix

### Algorithm:

Step 1: Scan one character at a time from right to left.

Step 2: Repeat until there is data.

a) If ')' Push into opstack

b) If operand, push into prestack

c) If operator - if stack is empty - push it into opstack

Else

- repeat while (preced (top char) > preced (scan char))  
and push to prestack.

- Pop and push to prestack.

- push scanchar to opstack.

d) if '(' found pop and push to prestack until the  
matching ')' is found and ignore (cancel) both.

Step 3: Pop and push to prestack until stack is empty.

Step 4: Pop and display from prestack until stack is empty.

### Tracing of algorithm:

Example:  $(A - (B/C)) * (D * E) - F$

soln.

| S.No. | Scan symbol | Prefix stack | opstack. |
|-------|-------------|--------------|----------|
| 1.    | )           | ....         | )        |
| 2.    | F           | F            | )        |
| 3.    | -           | F            | ) -      |
| 4.    | )           | F            | ) - )    |
| 5.    | E           | FE           | ) - )    |
| 6.    | *           | FE           | ) - ) *  |
| 7.    | D           | FED          | ) - ) *  |
| 8.    | (           | FED *        | ) -      |
| 9.    | (           | FED * -      | ...      |



|     |     |             |     |
|-----|-----|-------------|-----|
| 10. | *   | FED*-       | *   |
| 11. | )   | FED*-       | *)  |
| 12. | )   | FED*-       | *)  |
| 13. | c   | FED*-c      | *)  |
| 14. | /   | FED*-c      | *)/ |
| 15. | B   | FED*-cB     | *)/ |
| 16. | (   | FED*-cB/    | *)  |
| 17. | -   | FED*-cB/    | *)- |
| 18. | A   | FED*-cB/A   | *)- |
| 19. | (   | FED*-cB/A-  | *)  |
| 20. | ... | FED*-cB/A-* | ... |

∴ The final expression is  $*-A/Bc-*DEF$ .

9.  $A\$B*c-D+E/F/(G+H)$

| S.N. | Scan symbol | Prefix stack | op stack |
|------|-------------|--------------|----------|
| 1.   | )           | ...          | )        |
| 2.   | H           | H            | )        |
| 3.   | +           | H            | )+       |
| 4.   | G           | HG           | )+       |
| 5.   | (           | HG+          | ...      |
| 6.   | /           | HG+          | /        |
| 7.   | F           | HG+F         | /        |
| 8.   | /           | HG+F/        | /        |
| 9.   | E           | HG+F/E       | /        |
| 10.  | +           | HG+F/E/      | +/       |
| 11.  | D           | HG+F/E/D     | +/       |
| 12.  | -           | HG+F/E/D+    | +/-      |
| 13.  | C           | HG+F/E/D+C   | +/-      |
| 14.  | *           | HG+F/E/D+C   | -*       |
| 15.  | B           | HG+F/E/D+C   | -*       |



|     |     |                   |      |
|-----|-----|-------------------|------|
| 16. | \$  | HG+F/E/D+CB       | -*\$ |
| 17. | A   | HG+F/E/D+CB A     | -*\$ |
| 18. | ... | HG+F/E/D+CB A\$   | -*   |
| 19. | ... | HG+F/E/D+CB A\$*  | -    |
| 20. | ... | HG+F/E/D+CB A\$*- |      |

The final eq expression is  $- * \$ A B C + D / E / F + G H$ .

Q. Trace <sup>conversion</sup> ~~evaluation~~ algorithm for the given prefix expression.  
 $((A+B) * C / D - E) + F \$ G$

| S.No. | Scansymbol | Prefix stack               | opstack     |
|-------|------------|----------------------------|-------------|
| 1.    | G          | G                          | ...         |
| 2.    | \$         | G                          | \$          |
| 3.    | F          | G F                        | \$          |
| 4.    | +          | G F \$                     | +           |
| 5.    | )          | G F \$                     | +) )        |
| 6.    | E          | G F \$ E                   | +) )        |
| 7.    | -          | G F \$ E                   | +) ) -      |
| 8.    | D          | G F \$ E D                 | +) ) -      |
| 9.    | /          | G F \$ E D                 | +) ) - /    |
| 10.   | C          | G F \$ E D C               | +) ) - /    |
| 11.   | *          | G F \$ E D C /             | +) ) - *    |
| 12.   | )          | G F \$ E D C /             | +) ) - *)   |
| 13.   | B          | G F \$ E D C / B           | +) ) - *)   |
| 14.   | +          | G F \$ E D C / B           | +) ) - *) + |
| 15.   | A          | G F \$ E D C / B A         | +) ) - *) + |
| 16.   | (          | G F \$ E D C / B A +       | +) ) - * (  |
| 17.   | -          | G F \$ E D C / B A + * -   | +           |
| 18.   | ...        | G F \$ E D C / B A + * - + | ...         |

The final expression is  $+ - * + A B / C D E \$ F G$



## Evaluation of Prefix expression Algorithm

- 1) Read prefix expression from right to left until there is data.
- 2) Repeat :
  - 2.1: If char is operand <sup>(push)</sup> add to prestack.
  - 2.2: If char operator
    - 2.2.1: Operand 1 = pop prestack
    - 2.2.2: operand 2 = pop prestack
    - 2.2.3: result = value after applying operator bet<sup>n</sup> operand 1 & operand 2.
    - 2.2.4: Push the result into prestack.
- 3) Pop prestack get required value.

Example: Trace evaluation algorithm for the given prefix expression:

$+ - * + AB / CDE \$ FG$  where,

$A=1, B=2, C=3, D=2, E=1, F=4, G=2$

Sol<sup>n</sup>: The given prefix expression can be written as:

$+ - * + 12 / 32 1 \$ 42$

| S.N. | ScanSymbol | operand1 | operand2 | value | Prestack         |
|------|------------|----------|----------|-------|------------------|
| 1.   | 2          |          |          |       | 2                |
| 2.   | 4          |          |          |       | 2, 4             |
| 3.   | \$         | 2, 4     | 2        | 16    | 16               |
| 4.   | 1          |          |          |       | 16, 1            |
| 5.   | 2          |          |          |       | 16, 1, 2         |
| 6.   | 3          |          |          |       | 16, 1, 2, 3      |
| 7.   | /          | 3        | 2        | 1.5   | 16, 1, 1.5       |
| 8.   | 2          |          |          |       | 16, 1, 1.5, 2    |
| 9.   | 1          |          |          |       | 16, 1, 1.5, 2, 1 |
| 10.  | +          | 1        | 2        | 3     | 16, 1, 1.5, 3    |
| 11.  | *          | 3        | 1.5      | 4.5   | 16, 1, 4.5       |
| 12.  | -          | 4.5      | 1        | 3.5   | 16, 3.5          |
| 13.  | +          | 3.5      | 16       | 19.5  | 19.5             |



Q.1)  $-1A \$ BCD$

W)  $* - A / B C - * D E F$

Where  $A = 2, B = 1, C = 4, D = 3, E = 5, F = 1$

Soln:  $-2 \$ 143$

| S.No. | Scan Symbol | operand 1 | operand 2 | Value   | prestack. |
|-------|-------------|-----------|-----------|---------|-----------|
| 1.    | 3           |           |           | 3       | 3         |
| 2.    | 4           |           |           | 3, 4    | 3, 4      |
| 3.    | 1           |           |           | 3, 4, 1 | 3, 4, 1   |
| 4.    | \$          | 1         | 4         | 1       | 3, 1      |
| 5.    | 2           |           |           |         | 3, 1, 2   |
| 6.    | /           | 2         | 1         | 2       | 3, 2      |
| 7.    | -           | 2         | 3         | -1      | -1        |

Soln:  $2 / 14 - * 351$

| S.No. | Scan Symbol | operand 1 | operand 2 | Value | Prestack.   |
|-------|-------------|-----------|-----------|-------|-------------|
| 1.    | 1           |           |           |       | 1           |
| 2.    | 5           |           |           |       | 1, 5        |
| 3.    | 3           |           |           |       | 1, 5, 3     |
| 4.    | *           | 3         | 5         | 15    | 1, 15       |
| 6.    | -           | 15        | 1         | 14    | 14          |
| 7.    | 4           |           |           |       | 14, 4       |
| 8.    | 1           |           |           |       | 14, 4, 1    |
| 9.    | /           | 1         | 4         | 0.25  | 14, 0.25    |
| 10.   | 2           |           |           |       | 14, 0.25, 2 |
| 11.   | -           | 2         | 0.25      | 1.75  | 14, 1.75    |
| 12.   | *           | 1.75      | 14        | 24.5  | 24.5        |



## Unit - 4

## Recursion:

Defining a function / methods in terms of itself is called recursion. We call a method that calls itself is a recursive method.

Recurrence is derived from Latin,

re = back + currence = to run or happen again.

Each recursive call has its own distinct set of parameters and local variables. A recursive call is a separate entry on the execution stack.

The recursion process has two parts:

## 1) The Base case:

- In order for recursion to work correctly, every recursive method must have a basis case.
- The basis case is an input for which the method does not make a recursive call. The basis case prevents infinite recursion.

## 2) Recursive case that works to the base case.

The structure of a recursive method may be describe as:

method rec\_meth (int count) → Recursive method

{

-----

rec\_meth(count); → Recursive call

-----

}

## Example:

int power (int base, int exp)

{

if (exp == 0) // Base case



{ - Factorial  
- Fibonacci

```

        return 1;
    else
        return (base * power (base, exp-1)); // recursive call.
    }

```

### Designing a Recursive Algorithm :

- Every recursive call must either solve a part of the problem or reduce the size.
- Reduction must lead to a solution.
- Steps :
  - \* First determine the base case.
  - \* Determine the general case.
  - \* Combine these two to produce the algorithm.

### \* Methods of factorial using recursion :

```

int fact (int n)
{
    if (n==0)
        return 1;
    else
        return (n * fact (n-1));
}

```

### \* Methods of fibonacci using recursion :

```

int fib (int n)
{
    if (n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return fib (n-1) + fib (n-2);
}

```



- The Multiplication of Natural Numbers

The production  $a * b$ , where  $a$  and  $b$  are positive integers may be defined as  $a$  added to itself  $b$  times. This is an iterative definition. An equivalent recursive definition can be defined as:

Basis case:

$$a * b = a \text{ if } b = 1$$

$$a * b = 0, \text{ if } a = 0 \text{ or } b = 0$$

Recursive case:  $(b > 1)$

$$a * b = a + \text{mult}(a, b-1)$$

method:

```
int mult(int a, int b)
```

```
{
```

```
    if (b == 1)
```

```
        return a;
```

```
    else if (a == 0 || b == 0)
```

```
        return 0;
```

```
    else
```

```
        return a + mult(a, b-1);
```

```
}
```

- The Towers of Hanoi Problem (TOH)

It is a game problem, developed by Edouard Lucas in 1883. There will be different sized disks. Each disk has a hole in its centre so that it can be stacked on any of the pegs. Call the three pegs  $A, B$  and  $C$ . At the beginning of the game, the disks are stacked on the peg  $A$ , that is the largest sized disk on the bottom and the smallest sized disk on top.



Rule of the game:

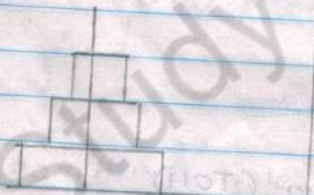
- 1) Transferring the disks from the source peg to the destination peg such that at any point of transformation, no larger size disk is placed on the smaller one.
- 2) Only one disk may be moved at a time.
- 3) Each disk must be stacked on anyone of the pegs.

Recursive Definition for TOH problems.

To solve the TOH problem for  $n$  disks; i.e. to move  $n$  disks from first peg A (source peg) to Third peg (destination peg) using second intermediate peg B as auxiliary:

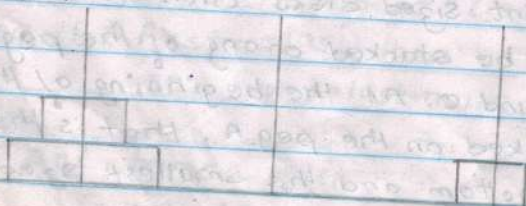
- 1) if  $n=1$ , move single disk from A to C and stop. (Base case).
- 2) Move top  $n-1$  disks from A to B, using C as auxiliary.
- 3) Move remaining disks from A to C.
- 4) Move top  $n-1$  disks from B to C, using A as auxiliary.

Trace the TOH algorithm for  $n=3$  (disks).



Initial Step

Step: 2





Step: 2

A<sup>c</sup>  
A<sup>b</sup> B<sup>c</sup>

n=1  
A-X  
Method: A → B  
B → C

Step: 3

Step: 4

A<sup>c</sup>  
A<sup>b</sup>  
A<sup>c</sup> B<sup>c</sup>  
A<sup>b</sup> C<sup>c</sup>

Step: 5

Step: 6

Step: 7



- Types of Recursion;

- 1) Direct Recursion: A method is called direct recursive if it calls directly to itself until base condition is not satisfied.

For example:

```
int fact (int num)
{
    if (num == 0)
        return 1;
    else
        return (num * fact (num-1));
}
```

- 2) Indirect Recursion: A method is said to be indirect recursive if it calls to another method and again this new method calls to its calling method.

For example:

```
int m1(int x)
{
    if (x <= 0)
        return 1;
    else
        return m2(x);
}

int m2(int y)
{
    return m1(y-1);
}
```



### 3) Tail Recursion :

A method that returns the value of its recursive call is said to be tail recursive or a method is said to be tail recursive if there are no any calculations occur in the recursive stage and it only returns the value.

For example:

```
int factorial (int n) {
```

```
    if (n == 0) {
```

```
        return 1;
```

```
    }
```

```
    factorial = f (n, 1);
```

```
}
```

```
int factorial_f (int cn ; int sum) {
```

```
    if (cn == 1) {
```

```
        return sum;
```

```
    }
```

```
    else {
```

```
        return factorial_f (n-1, sum*n);
```

```
    }
```

### 4) Nested Recursion:

The process of nested recursion is such a recursion in which a method is not only defined in terms of itself but also used as one of the parameters. The following definition is an example of such nesting:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ f(2 + f(2n)) & \text{if } n \leq 4 \end{cases}$$



## \* Differences between Iteration and recursion

| Iteration                                                                  | Recursion                                                                                  |
|----------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| 1) Iteration means repetition of process until the Condition fails.        | 1) In recursion, function call itself until the base condition is reached.                 |
| 2) It involves four steps, initialization, condition, increment/decrement. | 2) In recursive function, only base condition is specified. (terminate cond <sup>n</sup> ) |
| 3) It makes code longer.                                                   | 3) It keeps code short & simple.                                                           |
| 4) It is faster.                                                           | 4) It is slower than iteration due to overhead of maintaining stack.                       |
| 5) It takes less memory.                                                   | 5) It takes more memory.                                                                   |

## \* Backtracking:

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called backtracking. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in AI, and one of the problems in which backtracking is very useful is eight queens problem.



## • Queue

A queue is an ordered collection of the items in which a addition of items are made at one end called the rear, and all deletion are made at the other end called the front of the queue. A queue is logically a first in first out (FIFO) type of list. Queue means a line, eg. people waiting for getting tickets of cinema hall, passenger bus etc.

In computer application, there may be queue of tasks waiting for the line printer, for the access of disks, storage, or even in timesharing system, for use of CPU.

## • Queue as an ADT

An ADT is a set of values, well defined set of properties of the values and set of operation needed for the processing of these values. A queue can be expressed as an ADT as

- 1) Values: Order list of items.
- 2) Properties: Zero or more items, items are added at one end called rear and deleted from another end called front of the queue.
- 3) Operations:
  - a) Determine the queue is empty or not.
  - b) Determine the queue is full or not.
  - c) If the queue is not empty then delete the item from front end.
  - d) If the queue is not full then insert a new item at one end of the queue called rear.

## • Operation on Queue:

Given a queue  $Q$  and an element  $x$  of its data type typical operations to be applied on a queue are:



- Insert ( $Q, x$ ): Add  $x$  to the rear of  $Q$ , the new element  $x$  becomes the rear, size of Queue is increment by 1.
- $x = \text{Delete}(Q)$ : Return in  $x$ , the front element of  $Q$ , the front element is deleted, element next to it becomes front, elements size of  $Q$  decreased by 1.
- Empty ( $Q$ ): Return's true if  $Q$  is empty, otherwise false.
- Full ( $Q$ ): Return's true if  $Q$  is full, otherwise false.

#### \* Declaration of Queue

```
Class queue
```

```
{
```

```
    final int SIZE = 20;
```

```
    public int qu[] = new qu[SIZE];
```

```
    int rear, front;
```

```
    public queue()
```

```
    {
```

```
        rear = -1;
```

```
        front = 0;
```

```
    }
```

```
}
```

#### \* Implementation of Queue

There are two techniques for implementing the queue.

- 1) Array implementation
- 2) Linked list implementation.



## \* Array Implementation of Queue.

In array implementation of queue, an array is used to store the data elements. Array implementation is also further divided into two types:

- 1) Linear array implementation (linear queue)
- 2) Circular array implementation (circular queue)

### • Linear array

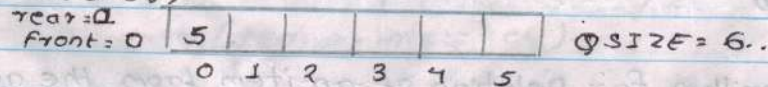
All elements of a queue are of same type. An array is a suitable choice for the member data part of the queue. We can use an array data [QSIZE] of required data type. Two variables front and rear are used to point the elements in queue. For example:



Initially rear is set to -1 and front is set to 0 indicate queue is empty.



Insert(Q, 5);



Insert(Q, 10);



Insert(Q, 15);

Similarly insert: 15, 20, 25, 30.



Insert(Q, 15);

|           |   |    |    |    |    |    |           |
|-----------|---|----|----|----|----|----|-----------|
| rear = 3  | 5 | 10 | 15 | 20 | 25 | 30 | QSIZE = 6 |
| front = 0 | 0 | 1  | 2  | 3  | 4  | 5  |           |

Delete two items

|           |   |   |    |    |    |    |
|-----------|---|---|----|----|----|----|
| Rear = 5  |   |   | 15 | 20 | 25 | 30 |
| Front = 2 | 0 | 1 | 2  | 3  | 4  | 5  |

Here,  $\text{rear} = \text{QSIZE} - 1$ , queue is actually not full. This is an absurd situation.

Linear queue implementation

\* Algorithm for insertion on item in queue (enqueue):

1) Initialize  $\text{front} = 0$  and  $\text{rear} = -1$ .

If  $\text{rear} == \text{QSIZE} - 1$

Print "Queue is full" and return

else

Set  $\text{rear} = \text{rear} + 1$ ;

$\text{queue}[\text{rear}] = \text{item}$ ;

2) Stop

\* Algorithm for Deletion of an item from the queue (Dequeue)

1) If  $\text{rear} < \text{front}$

Print "Queue is empty" and return.

else

$\text{item} = \text{queue}[\text{front}]$  //  $\text{item} = \text{queue}[\text{front}]$   
 $\text{front}++$ ;

Print "item" as a deleted element

2) Stop



\* Method for insertion

```
public void insert (int item)
{
    if ( rear == QSIZE-1)
    {
        System.out.println("Queue is full");
    }
    else
    {
        rear = rear + 1;
        queue[rear] = item;
        System.out.println("The inserted element is: " + item);
    }
}
```

\* Method for deletion

```
public void delete()
{
    if (rear < front)
    {
        System.out.println("Queue is empty");
    }
    else
    {
        item = queue[front];
        front++;
        System.out.println("The deleted element is: " + item);
    }
}
```



### Stack implementation

```
import java.util.Scanner;
class Stack {
    final int size = 10;
    int item[] = new int[size];
    int top;
    Stack() {
        top = -1;
    }
    void push() {
        System.out.println("Enter an item to push");
        Scanner in = new Scanner(System.in);
        if (top == size - 1) {
            System.out.println("Stack is Full");
        }
        else {
            top++;
            item[top] = in.nextInt();
        }
    }
    void pop() {
        int element;
        if (top == -1) {
            System.out.println("Stack is empty");
        }
        else {
            element = item[top];
            top--;
            System.out.println("Popped element: " + element);
        }
    }
}
```



```
class PushPop {
    public static void main (String args[]) {
        Scanner in = new Scanner (System.in);
        Stck s = new Stck();
        int x;
        System.out.println ("Enter 1 to push, 2 to
            pop, 3 to display and 4 to exit");
        do {
            System.out.println ("Enter your choice:");
            x = in.nextInt();
            Switch (x) {
                case 1:
                    s.push();
                    break;
                case 2:
                    s.pop();
                    break;
                case 3:
                    for (int a = s.top; a >= 0; a--) {
                        System.out.println (s.item[a]);
                    }
                    break;
                case 4:
                    System.exit(0);
            }
        } while (x < 4);
    }
}
```



### Queue implementation

```
import java.util.Scanner;
class Queue {
    final int size = 10;
    int qu[] = new int[size];
    int rear, front;
    public Queue()
    {
        rear = -1;
        front = 0;
    }
    void enqueue() {
        Scanner in = new Scanner(System.in);
        if (rear == size - 1)
            System.out.println("Queue is full");
        else {
            rear++;
            System.out.println("Enter item to insert");
            qu[rear] = in.nextInt();
        }
    }
    void dequeue() {
        if (rear < front) {
            System.out.println("Queue is Empty");
        }
        else {
            int element = qu[front];
            for (int i = 0; i <= rear; i++) {
                qu[i] = qu[i + 1];
            }
            rear--;
        }
    }
}
```



```

        System.out.println("Deleted item:" + telement);
    }
}

class QueueDemo {
    public static void main (String args[]) {
        Scanner in = new Scanner (System.in);
        Queue q = new Queue();
        int x;
        System.out.println ("Enter 1 to insert, 2 to delete,
                             3 to display and 4 to exit:");
        do {
            System.out.println ("Enter your choice:");
            x = in.nextInt();
            Switch (x) {
                Case 1:
                    q.enqueue();
                    break;
                Case 2:
                    q.dequeue();
                    break;
                Case 3:
                    for (int a=0; a<=q.rear; a++) {
                        System.out.println (q.queue[a]);
                    }
                    break;
                Case 4:
                    System.exit(0);
            }
        } while (x < 4);
    }
}

```



## Circular Queue

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. When the rear part reaches the last index of linear queue but front part is not at first index, of the queue then in this case if we try to insert some elements, then according to the logic when  $\text{rear} = \text{QSIZE} - 1$  then it encounters an overflow situation. But there are some elements left blank at the beginning part of queue. To utilize those left over spaces more efficiently, a circular fashion is implemented in a queue representation. The circular fashion of queue reassigns the rear with 0 if it reaches  $\text{QSIZE} - 1$  and beginning elements are free and the process is continued for deletion also. Such queues are called circular queue.

A circular queue overcomes the problem of unutilized space in linear queue implementation as array. In circular queue we sacrifice one element of the array thus to insert  $n$  elements in circular queue we need an array of size  $n+1$  (or we can insert one less element than the size of an array in circular queue).

|      |  |
|------|--|
| a[5] |  |
| a[4] |  |
| a[3] |  |
| a[2] |  |
| a[1] |  |
| a[0] |  |

Fig: linear representation of queue.



Fig: circular representation of queue.



### Initialization of circular queue

rear = front = QSIZE - 1

#### \* Algorithms for inserting an element in a circular queue

This algorithm assumed that rear and front are initially set to QSIZE - 1.

Step 1: Check queue full condition as,

if (front == (rear + 1) % QSIZE)

Print Queue is full and exit.

else

rear = (rear + 1) % QSIZE

Step 2: Cqueue [rear] = data;

Step 3: Stop

#### \* Algorithm for deleting an element from a circular queue.

This algorithm assumed that rear and front are initially set to QSIZE - 1.

Step 1: Check queue empty condition as

if (rear == front)

Print Queue is empty and exit.

else

front = (front + 1) % QSIZE

Step 2: data = cqueue [front];

Step 3: return the value of data.

Step 4: Stop.

#### • Declaration of circular queue

class Cqueue {

final int QSIZE = 5;

public int arr[] = new int [QSIZE];

public int rear, front;



```
public Queue()
{
```

```
    rear = QSIZE - 1;
```

```
    front = QSIZE - 1;
```

```
}
```

```
}
```

\* The isEmpty method:

```
boolean isEmpty()
```

```
{
```

```
    return rear == front;
```

```
}
```

\* The IsFull method :

```
boolean IsFull()
```

```
{
```

```
    return (front == (rear + 1) % QSIZE);
```

```
}
```

\* Method for inserting element in Circular queue

```
public void insert(int item){
```

```
    if(front == (rear + 1) % QSIZE)
```

```
        System.out.println("Queue is full");
```

```
    else
```

```
    {
```

```
        rear = (rear + 1) % QSIZE;
```

```
        Queue[rear] = item;
```

```
    }
```

```
}
```



\* Method for deleting element from circular queue.

```
public void delete ()  
{  
    if (rear == front){  
        System.out.println("Queue is empty");  
    }  
    else{  
        front = (front + 1) % QSIZE;  
        item = queue[front];  
        System.out.println("Deleted item is: " + item);  
    }  
}
```

\* Priority queue

A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed comes from the following rules:

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

\* Types of Priority queue

- 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 deleted.

- Descending Priority queue (Max Priority queue)

A descending priority queue is a collection of items



into which items can be inserted arbitrarily but from which only the largest item can be deleted.

### Priority Queue Operations:

- 1) Insertion: The insertion in priority queues is the same as in non-priority queue.
- 2) Deletion: Deletion requires a search for the element of highest priority and delete the element with highest priority.
- 3) Display: The display in priority queue is the same as in non-priority queues.

### Declaration of Priority queue:

```
class Pqueue
```

```
{
```

```
    final int PQSIZE=5;
```

```
    public int arr[] = new arr[PQSIZE];
```

```
    public int front, rear;
```

```
    public Pqueue()
```

```
    {
```

```
        rear = -1;
```

```
        front = 0;
```

```
    }
```

```
    boolean IsFull() {
```

```
        return (rear == PQSIZE-1);
```

```
    }
```

```
    boolean IsEmpty() {
```

```
    }
```

```
        return rear < front;
```

```
    }
```



- Priority queue as an ADT

An ADT is a set of well defined values and set of properties of the values and operation needed for the processing of those values.

A priority queue can be expressed as an ADT as:

- 1) Values: Ordered list of items.
- 2) Properties: Zero or more items are added at one end called rear and deleted from another end called front of the queue.
- 3) Operations:
  - a) Determine the priority queue is empty or not.
  - b) Determine the queue is full or not.
  - c) If the queue is not empty, then delete the item from front end.
  - d) If the priority queue is not full then insert a new item at one end of queue called rear.

- method for inserting

```
void insert (int data) {
    if (front == (rear + 1) % QSIZE)
        System.out.println ("Queue is full");
    else
        rear = (rear + 1) % QSIZE;
        Pqueue [rear] = data;
}
```



## Linked list

Linked list are special type of list of data in which data elements are linked to one another.

A Linked list consists of nodes of data which are connected with each other. Every node consists of two fields data (info) and the link (next). The nodes are created dynamically.

1) Info : The actual element to be stored in the list. It is also called data field.

2) Link : One or two links that points to next and previous node in the list. It is also called next field.

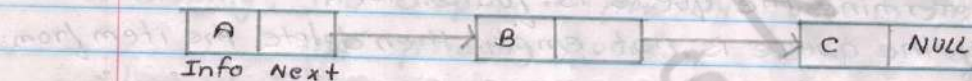


Fig : Singly linked list containing three nodes.

### Header Nodes:

A header node is an extra node in the linked list that holds no data but serves to satisfy the requirement that every node containing data have a previous node in the list. By using the header node, we can simplify the code with negligible space penalty.

### Types of Linked list :-

- \* Single Linked list
- \* Double Linked list
- \* Circular singly linked list
- \* Circular doubly linked list



### Operations on linked list:-

- 1) Inserting elements to linked list:
  - \* Inserting an elements at the first position.
  - \* Inserting an elements at the end
  - \* Inserting an elements at specified position.
- 2) Deleting elements from linked list
  - \* Deleting an element at first position.
  - \* Deleting an element at end
  - \* Deleting an element at specified position.
- 3) Searching a node.
- 4) Merging linked list.

### Singly Linked list

A singly linked list is a dynamic data structure which may grow or shrink, 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 value and another is link field that is used to point the next node in the list. For eg:



Fig: Singly linked list



### Structure of a Node of Singly-Linked List

```
class SLLNode
{
    public int info;
    public SLLNode next;
    SLLNode() {
        info = 0;
        next = null;
    }
    SLLNode first, last;
    first = null;
    last = null;
}
```

- \* An algorithm to insert a node at the beginning of the Singly linked list.

Let 'first' and 'last' are reference to first node and last node in the current linked list respectively.

Step 1: Create a new node using class name as class\_name

```
Newnode = new class_name();
```

Step 2: Read data item to be inserted say 'data'.

Step 3: Assign data to info field of Newnode,

```
Newnode.info = data;
```

Step 4: Set the next of Newnode to first,

```
Newnode.next = first
```

Step 5: Set the first reference to Newnode,

```
first = Newnode.
```

- \* The Java method to insert a node at the beginning of single linked list

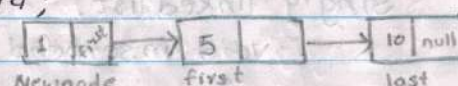
```
void inserting (int data)
{
```



```

SLLNode Newnode = new SLLNode();
Newnode.info = data;
empty // if (first == null)
{
    Newnode.next = null;
    first = Newnode;
    last = Newnode;
}
else {
    Newnode.next = first;
    first = Newnode;
}
}

```



\* An algorithm to insert a node at the end of the linked list.

1) Create new node using class name as,

class-name Newnode = new class-name();

2) Read data item to be inserted say 'data'.

3) Assign data to info field of Newnode,

Newnode.info = data;

4) Set the next of Newnode to null,

Newnode.next = null;

5) if (first == null) then

Set first = last = Newnode then exit.

6) else

Set, last.next = Newnode;

Set, last = Newnode;



## Circular Queue

- The Java method to insert a node at the end of the singly linked list.

```
void insertend (int data)
{
```

```
    SLLNode Newnode = new SLLNode();
```

```
    Newnode.info = data; Newnode.next = null;
```

```
    if (first == null)
```

```
    {
```

```
        first = Newnode;
```

```
        last = Newnode;
```

```
    }
```

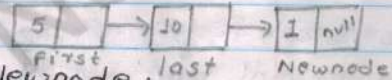
```
    else {
```

```
        last.next = Newnode; last = Newnode;
```

```
        last = Newnode;
```

```
    }
```

```
}
```



- \* An algorithm to insert a node at the specified position in the singly linked list.

Let first and last are the reference to first node and last node in linked list respectively.

- 1) Create a newnode using class name as,

```
Class-name Newnode = new Class-name ();
```

- 2) Assign data to the info field of Newnode

```
Newnode.info = data;
```

- 3) Enter the position of a node at which you want to insert a Newnode. Let this position is 'pos'.

- 4) Set temp = first

- 5) if (first == null) then

print "No insertion" and exit.



6) else

for( $i=1$  ;  $i < pos-1$  ;  $i++$ )

temp = temp.next;

7) Set Newnode.next = temp.next;

8) Set temp.next = Newnode;

- Method to insert node at the specified position in a singly linked list.

void insertpos (int data)

{

SLLNode Newnode = new SLLNode();

SLLNode temp = new SLLNode();

Newnode.info = data;

Scanner sc = new Scanner (System.in);

System.out.println ("Enter position of a node at which you want to insert a new node");

pos = sc.nextInt();

if (first == null)?

first = Newnode;

last = Newnode; temp = first;

}

else {

temp = first;

for ( $i=1$  ;  $i < pos-1$  ;  $i++$ ) {

temp = temp.next;

}

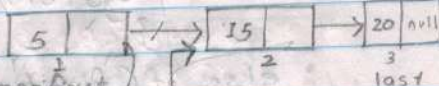
Newnode.next = temp.next;

temp.next = Newnode;

}

}

pos = 2



Newnode



### Deletion process

- i) An algorithm to deleting the first node of the singly linked list.

Let first and last are reference variable to first and last node in the linked list respectively,

1. If (first == null) then

print "No deletion process" and exit.

2. Store the address of first node in a temporary variable temp.

// Set, temp = first;

3. Set first to next of first.

i.e. set, first = first.next

Java method to deleting first node of the singly linked list.

```
void Deletefirst(){
```

```
    if (first == null)
```

```
    {
```

```
        System.out.println("Linked list is empty");
```

```
    }
```

```
    else if (first == last){
```

```
        first = null;
```

```
        last = null;
```

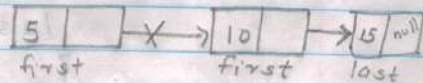
```
    }
```

```
    else{
```

```
        first = first.next;
```

```
    }
```

```
}
```





2) An algorithm for deleting the last node of the Singly linked list.

(let first and last are reference variable to first and last node in the linked list respectively,

1. If (first == null) then  
print "No deletion process" and exit.

2. else if (first == last) then  
print deleted item as, first-info.  
set = first = last = null;

3. else  
temp = first;  
while (temp.next != last)  
set, temp = temp.next;  
set temp.next = null;  
Set last = temp;

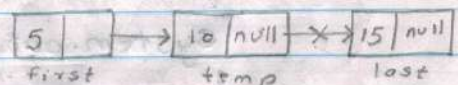
• Method :

```
void Deletelast() {
    if (first == null) {
        System.out.println("Linked list is empty");
    }
```

```
    else if (first == last) {
        first = null;
        last = null;
    }
```

```
    else {
        SLLNode temp = new SLLNode();
        temp = first;
```

```
        while (temp.next != last) {
            temp = temp.next;
        }
        temp.next = null;
        last = temp;
    }
```





## Stack Implementation

3) An algorithm to delete a node at the specified position in a Singly Linked List.

Let 'first' and 'last' are the reference variable to first node and last node in the linked list respectively.

1) Read the position of a node which to be deleted let it be 'pos'.

2) If (first == null)

print "No deletion" and exit.

otherwise,

3) Enter position of node.

4) Set temp = first

5) for (i = 1; i < pos - 1; i++)

set temp = temp.next;

6) Display deleted item is temp.next.info

7) Set loc = temp.next;

8) Set temp.next = loc.next;

\* Method

pos = 2 →



void deleteAtSpecifiedPos()

{ if (first == null)

{

System.out.println("linked list is empty");

}

else { int pos, i;

SLLNode temp = new SLLNode();

temp = first; Scanner sc = new Scanner(System.in);

System.out.println("Enter position of a node which you want to delete");

pos = sc.nextInt();

for (i = 1; i < pos - 1; i++) {

temp = temp.next;

}

66



```

SLLNode loc = new SLLNode();
loc = temp.next;
temp.next = loc.next;
}
}

```

- Searching a data value in a Single Linked list.

Method :

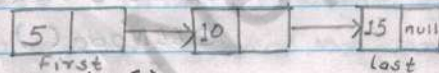
let 'first' and 'last' are the reference variable to first node and last node in the linked list respectively

void SearchData ()

```

{
    SLLNode temp = new SLLNode();
    int key;
    if (first == null) {
        System.out.println ("Linked list is empty");
    }
    else if Scanner Sc = new Scanner (System.in);
    System.out.println ("Enter the value of item to be
        searched");
    key = Sc.nextInt();
    temp = first;
    while (temp != null) {
        if (temp.info == key)
        {
            System.out.println ("Search is Successful"
                + temp.info);
        }
        temp = temp.next;
    }
    else {
        System.out.println ("Search unsuccessful");
    }
}

```





## Circular linked list

A circular linked list is a list where the link field of last node points to the first node of the list. A circular linked list is very similar to the linear linked list where in the CLL the pointer of the last node points not null but the first node.

### Representation of circular linked list

```
class CLLNode {
    int info;
    CLLNode next;
    CLLNode()
    {
        info = 0;
        next = null;
    }
}
```

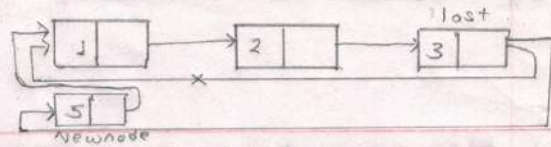


Fig: representation of circular linked list.

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

- 1) Create a new node by using class name as:  
`class_name Newnode = new class_name();`
- 2) Read data item to be inserted say it be 'data'.
- 3) Set `Newnode.info = data`
- 4) If `start == null` then,  
`set, Newnode.next = newnode.`  
`set, (first)start = Newnode`  
`set, last = Newnode`
- 5) else, `set, Newnode.next = start;`  
`set, (first)start = Newnode`  
`set, last.next = Newnode`





Method:

```
void insert(int data)
{
```

```
    CLL Newnode = new CLL();
```

```
    Newnode.info = data;
```

```
    if (first == null)
```

```
    {
```

```
        Newnode.next = newnode;
```

```
        first = Newnode;
```

```
        last = Newnode;
```

```
    }
```

```
    else
```

```
    {
```

```
        Newnode.next = first;
```

```
        first = Newnode;
```

```
        last->next = Newnode;
```

```
    }
```

```
}
```

\* Algorithm to insert node at the end of circular linked list

1. Create a newnode by using the class\_name as:

```
class_name Newnode = new class_Name();
```

2. Read data item to be inserted say it be 'data'.

3. Set Newnode.info = data;

4. if first == null then,

```
Set Newnode.next = newnode;
```

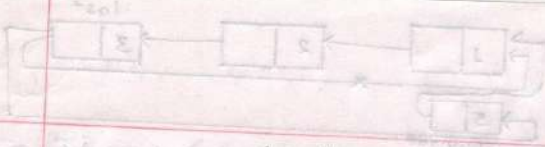
```
first = Newnode;
```

```
last = Newnode;
```

else,

```
Set last->next = Newnode;
```

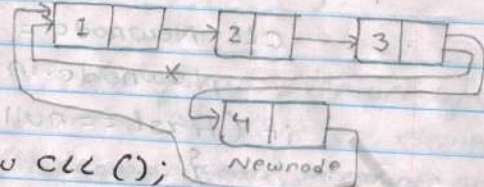




last = Newnode;  
Newnode.next = first;

\* Method:

```
void Einsert (int data)
{
```



```
    CLL Newnode = new CLL ();
    Newnode.info = data;
```

```
    if (first == null)
    {
```

```
        Newnode.next = Newnode;
        first = Newnode;
        last = Newnode;
    }
```

```
    else
    {
```

```
        last.next = Newnode;
        last = Newnode;
        Newnode.next = first;
    }
```

```
}
```

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

1) If start == null then,

Print "Linked list is empty" and exit.

2) else

Print the deleted element = start.info

Set start = start.next

Set last.next = start.



Method:

```
void Bdelete( )
```

```
{
```

```
if (start == null)
```

```
{
```

```
System.out.println("Linked list is empty");
```

```
}
```

```
else if (start == last)
```

```
{
```

```
start = null;
```

```
last = null;
```

```
}
```

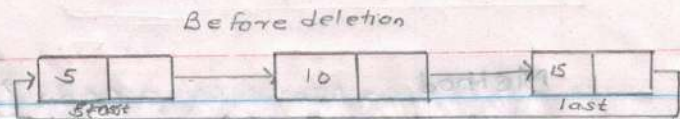
```
else {
```

```
start = start.next;
```

```
last.next = start;
```

```
}
```

```
}
```



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

1) If start == null then,

Print "Linked list is empty" and exit.

2) else if (start == last)

- Print deleted element = start.info

- Set, start = last = null

3) else,

Set, temp = start

while (temp.next != last)

Set, temp = temp.next

Print the deleted element = last.info

Set last = temp

Set last.next = start.



Method

```
void deleteEnd()
```

```
{
```

```
    CLL temp = new CLL();
```

```
    if (start == null)
```

```
    {
```

```
        System.out.println("Linked list is empty");
```

```
    }
```

```
    else if (start == last)
```

```
    {
```

```
        start = null;
```

```
        last = null;
```

```
    }
```

```
    else {
```

```
        temp = start;
```

```
        while (temp.next != last)
```

```
        {
```

```
            temp = temp.next;
```

```
        }
```

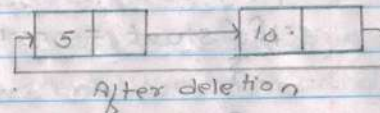
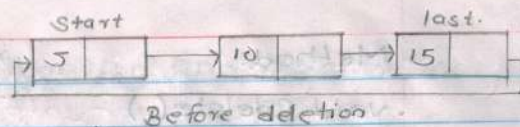
```
        last = temp;
```

```
        last.next = start;
```

```
        System.out.println("Deleted element: " + last.info);
```

```
    }
```

```
}
```



### \* Doubly Linked List (DDL)

A doubly linked list is similar to singly linked list except that it links nodes in both directions. i.e. each node has links (pointers) to its next node as well as its previous node in the list. It is also called two-way



list. Following figure shows the structure of this type :

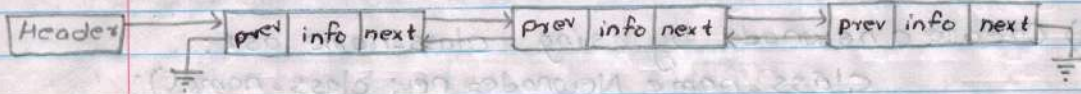


Fig: A doubly linked list

The following properties characterized the doubly linked list.

- 1) Node are linked on both forward and backward direction i.e. each node has one address to point to the next node and another address to point the previous.
- 2) Last node, next pointer as well as first node, previous pointer are null.
- 3) Searching for a node can be done in either way, thus it is faster than SLL.

Representation of doubly linked list

Class DLLNode

```
{
    int info;
    DLLNode prev;
    DLLNode next;
public DLLNode()
{
    info = 0;
    prev = null;
    next = null;
}
}
```



\* Algorithm to insert a node at the beginning of a doubly linked list.

- 1) Create a newnode by using class-name as:  
`class_name Newnode = new class_name();`
- 2) Read the data item to be inserted say it be 'data';
- 3) Set, `Newnode.info = data`
- 4) Set, `Newnode.prev = Newnode.next = null`.
- 5) If `first == null` then  
`Set, first = last = Newnode;`
- 6) else, set `Newnode.next = first;`  
`Set first.prev = Newnode;`  
`Set first = Newnode;`

Method:

`void insertBeg(int data)`  
`{`

`DLLNode Newnode = new DLLNode();`

`Newnode.info = data;`

`Newnode.prev = null;`

`Newnode.next = null;`

`if (first == null)`

`{`

`first = Newnode;`

`last = Newnode;`

`}`

`else {`

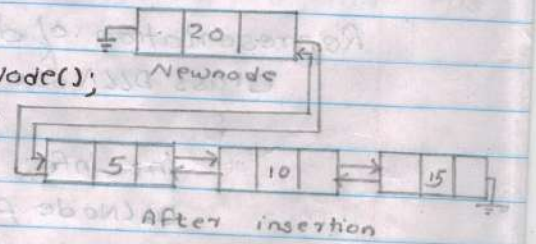
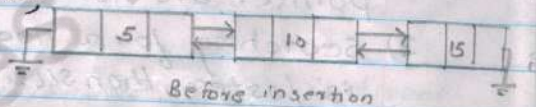
`Newnode.next = first;`

`first.prev = Newnode;`

`first = Newnode;`

`}`

`}`





\* Algorithm to insert a node at the end of a doubly linked list.

1. Create a newnode by using class\_name as:  
`class_name Newnode = new class_name();`
2. Read the data item to be inserted say it be 'data'.
3. Set, `Newnode.info = data`
4. Set, `Newnode.prev = Newnode.next = null`
5. If `first == null` then  
`set, first = last = Newnode;`
6. else, set `last.next = Newnode` ~~uprev~~;  
`Set Newnode.prev = last;`  
`set last = Newnode;`

Method:

`void insertEnd(int data)`

`{`  
`DLLNode Newnode = new DLLNode();`

`Newnode.info = data;`

`Newnode.prev = null;`

`Newnode.next = null;`

`if (first == null) {`

`first = Newnode;`

`last = Newnode;`

`}`

`else {`

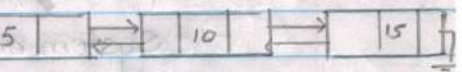
`last.next = Newnode;`

`Newnode.prev = last;`

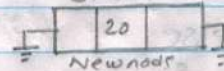
`last = Newnode;`

`}`

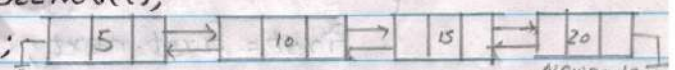
`}`



Before insertion



Newnode



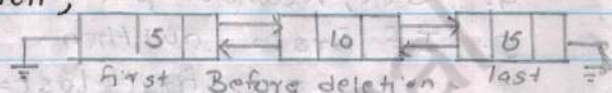
After insertion

Newnode



\* Algorithm to delete a node from beginning of a doubly linked list.

1. If  $first == null$  then  
print "Empty list" and exit.
2. else  
set,  $first = first.next$ ;  
set,  $first.prev = null$ ;



Method:

```
void deleteBeg()
```

```
{
```

```
    if (first == null)
```

```
    {
```

```
        System.out.println("Empty link list");
```

```
    }
```

```
    else
```

```
    {
```

```
        first = first.next;
```

```
        first.prev = null;
```

```
    }
```

```
}
```



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

- 1) If  $(first == null)$  then,  
Display the message "List is empty" and exit.
- 2) else if  $(first == last)$  then  
set  $first = last = null$
- 3) else  
set,  $temp = first$ ;  
while  $(temp.next != last)$   
temp = temp.next;

7



set, temp.next = null;

set, last = temp;

Method:

void deleteEnd()

{

if (first == null)

{

System.out.println("List is empty");

}

elseif (first == last)

{

first = null;

last = null;

}

else { DLLNode temp = new DLLNode();

temp = first;

while (temp.next != last)

{

temp = temp.next;

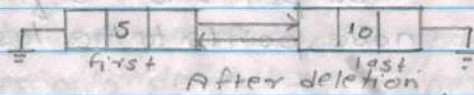
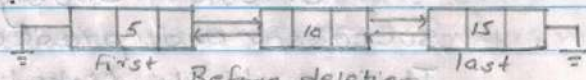
}

temp.next = null;

last = temp;

}

}





## Circular Doubly Linked List (CDLL)

A circular doubly linked list is one which has successor and predecessor pointer in circular manner. It is a doubly linked list where the next link of last node points to the first node and previous link of first node points to the last node of the list.

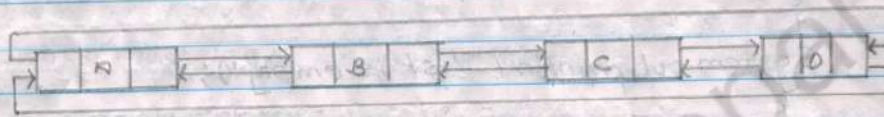


Fig: Circular Doubly linked list.

Representation of doubly circular linked list

```
class CDLLNode
```

```
{
```

```
    int info;
```

```
    public CDLLNode prev;
```

```
    public CDLLNode next;
```

```
    public CDLLNode()
```

```
    {
```

```
        info = 0;
```

```
        prev = null;
```

```
        next = null;
```

```
    }
```

```
}
```

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

- i) Create a Newnode by using class\_name as,  
class\_name Newnode = new class\_name();



2) Read the data value to be inserted say 'data'.

3) Set, Newnode.info = data

4) IF (first == null) then,

Set, first = last = Newnode.

5) set, Newnode.next = first

set, Newnode.prev = last

6) else

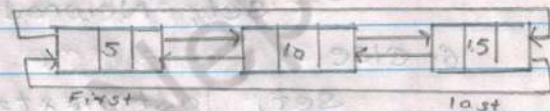
Set, Newnode.next = first

Set, first.prev = Newnode

Set, first = Newnode

Set, last.next = first

Set, first.prev = last



Before insertion

Method:

void insertBeg(int data)

{

CDLLNode Newnode = new CDLL();

Newnode.info = data;

if (first == null) {

first = Newnode;

last = Newnode;

Newnode.next = Newnode;

Newnode.prev = Newnode;

}

else {

Newnode.next = first;

first.prev = Newnode;

first = Newnode;

last.next = first;

first.prev = last;

}

}

72

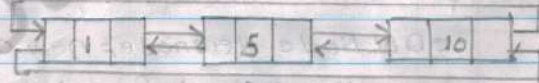


After insertion



\* Algorithm to insert the node at the ending of a circular doubly linked list.

1. Create a Newnode by using class-name,  
 $\text{class-name Newnode} = \text{new class-name}();$
2. Read the data value to be inserted say 'data'.
3. Set,  $\text{Newnode.info} = \text{data};$
4. If ( $\text{first} == \text{null}$ ) then,  
 $\text{Set, first} = \text{last} = \text{Newnode}$   
 $\text{Set, Newnode.next} = \text{Newnode}$   
 $\text{Set, Newnode.prev} = \text{Newnode}$
6. else  
 $\text{Set, last.next} = \text{Newnode};$   
 $\text{Newnode.prev} = \text{last};$   
 $\text{Newnode.next} = \text{first};$   
 $\text{first.prev} = \text{Newnode};$   
 $\text{last} = \text{Newnode};$



Method

void insertEnd(int data)

{

DCLLNode Newnode = new DCLLNode();

Newnode.info = data;

if (first == null) {

first = Newnode;

last = Newnode;

Newnode.next = Newnode;

Newnode.prev = Newnode;

}

else {

last.next = Newnode;





```

Newnode.prev = last;
Newnode.next = first;
first.prev = Newnode;
last = Newnode;
}
}

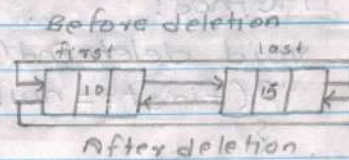
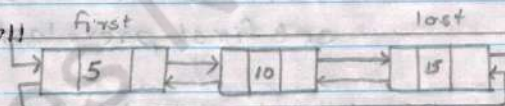
```

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

1. IF (first == null)  
print "linked list is empty".

2. else if (first == last)  
set first = last = null;

3. else  
set first = first.next;  
set last.next = first;  
set first.prev = last;



Method:

```

void deletefirst() {
    if (first == null) {
        System.out.println("linked list is empty");
    }
    else if (first == last) {
        first = null;
        last = null;
    }
    else {
        first = first.next;
        last.next = first;
        first.prev = last;
    }
}

```



\* Algorithm to delete a node from the end of a circular <sup>doubly</sup> linked list.

1) if (first == null) then  
print "Linked list is empty" and exit.

2) ~~if~~ else if (first == last)  
set, first = last = null

3) else

set, temp = first;

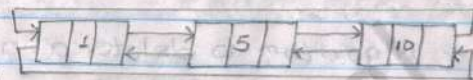
while (temp.next != last)

set temp = temp.next

set last = temp

set last.next = first

set first.prev = last



Method:

void deleteEnd()

if (first == null)

{

System.out.println("Empty DLL");

}

else if (first == last)

{

first = null;

last = null;

}

else {

CDLLNode temp = new CDLLNode();

temp = first;

while (temp.next != last)

{

temp = temp.next;

}

82



skip list

```
last = temp;
last.next = first;
first.prev = last;
}
```

\* Display() for CDLL

```
void display()
{
```

```
    CDLLNode temp = new CDLLNode();
```

```
    if (First == null) {
```

```
        System.out.println("CDLL is empty");
```

```
    }
```

```
    else {
```

```
        temp = first;
```

```
        while (temp != last) {
```

```
            {
```

```
                System.out.println(temp.info);
```

```
                temp = temp.next;
```

```
            }
```

```
        System.out.println(last.info);
```

```
    }
```

```
}
```



\* Stack as Linked List (Linked list implementation of stack)  
(Dynamic implementation of stack)

The dynamic representation of a stack, commonly termed as linked stack, is more suitable if an accurate estimate of the stack's size can not be made in advance and the data elements are large records. In the linked list representation, the stack is represented as a linked list where each node contains info and next part.

The top pointer of the stack is represented by Start pointer of the Linked list. If the Start pointer is null, then it shows that the stack is empty. If the last node link field contains null then it shows the bottom of the stack.

The following figure shows the linked list or dynamic representation of stack.

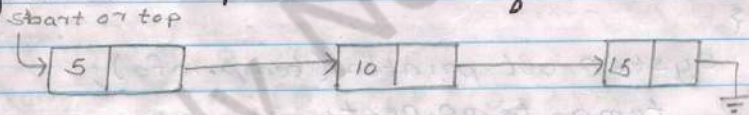
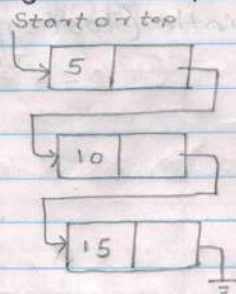


Fig: Dynamic representation of stack with three elements



• Method for Push operation:

```
void push (int data)
{
```

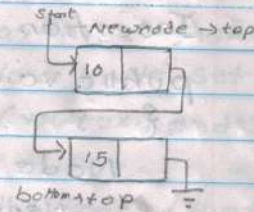
```
Node Newnode = new Node();
```



```

Newnode.info = data;
if (top == null)
{
    Newnode.next = null;
    top = Newnode;
}
else
{
    Newnode.next = top;
    top = Newnode;
}
}

```



• Method for Pop operation:

```

void pop()
{
    if (top == null)
    {
        System.out.println("Stack is empty");
    }
    else {
        Node temp = new Node ();
        temp = top;
        top = top.next;
        System.out.println("The popped item is: " + temp.info);
    }
}

```



```

newnode.info = data;
newnode.next = null;
}

```

## \* Queue as a linked list

### • Insertion operation:

```

public void enqueue(int data)
{
    Node newnode = new Node();
    newnode.info = data;
    if (rear == null && front == null)
    {
        rear = newnode;
        front = newnode;
    }
    else {
        rear.next = newnode;
        rear = newnode;
    }
}

```

### • Deletion operation:

```

public void dequeue()
{
    if (front == null)
    {
        System.out.println("Empty queue");
    }
    else {
        Node temp = new Node();
        temp = front;
        front = front.next;
        System.out.println("The deleted item is " + temp.info);
    }
}

```



## Trees

### Trees

A tree is a non-linear data structure in which data items are arranged in a hierarchical fashion. Formally, tree can be defined as it is a finite set of one or more data items called node (vertex) and set of edges which links nodes such that:

- i) There is a special data item called the root of the tree
- ii) And its remaining data items are partitioned into number of mutually exclusive (i.e. disjoint) subsets, each of which is itself a tree, and they are called subtrees.

For example:

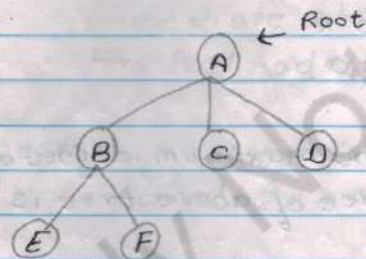


Fig: Tree representation.

Here, A is called root (parent) node and others are called child nodes of A.

### • Tree terminology

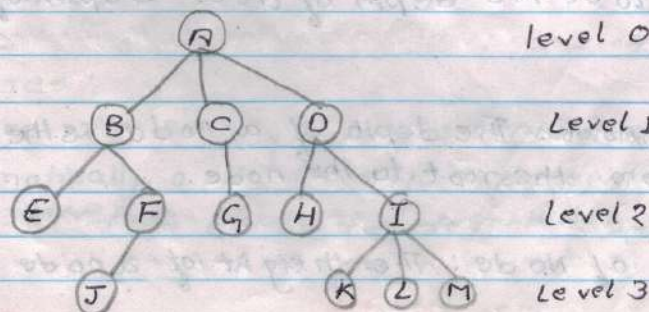


Fig: A tree



\* **Root**: It is specially designed data item in tree. It is the first in the hierarchical arrangement of data items. A.

\* **Node**: Each data item in a tree is called a node. It specifies the data value and links (branches) to other data items. There are 13 nodes in the above tree.

\* **Degree of node**: It is the number of subtrees of a node in a given tree. In the above tree:

→ The degree of node A is 3

→ The degree of node C is 1.

→ The degree of node B is 2

→ The degree of node E is 0.

\* **Degree of tree**: It is the maximum degree of nodes in a given tree. So the degree of above tree is 3.

\* **Leaf node**: A node with degree zero is called a leaf node. In the above tree there are 7 leaf nodes. They are E, J, G, H, K, L, M.

\* **Depth**: The length of largest path from root to terminals is said to be the depth of tree. Depth of above tree is 3.

\* **Depth of node**: The depth of a node is the length of the path from the root to the node.

\* **Height of Node**: The height of a node is the length of



the path from the node to the deepest leaf node.

For example of Height of A is 3.

Height of D is 2.

Height of H is 0

Height of C is 1

- \* **Edge**: It is a connecting line of two nodes.
- \* **path**: It is a sequence of consecutive node edges from the source node to the destination node.
- \* **Level**: The entire tree structure is leveled in such a way that the root node is always at level 0, then its immediate children are at level 1, and their immediate children are at level 2 and so on upto the leaf node.
- \* **Ancestor**: If there is a path from node A to node B then A is ancestor of B.
- \* **Descendant**: If A is ancestor of B, then B is descendant of A.
- \* **Siblings**: The child nodes of a given parent node are called Siblings. In the above tree E and F are Siblings of parent node B.

**Null tree**:

A tree with no any nodes is termed as null tree. Mathematically, a null tree is represented as  $\epsilon$  and in data structure it is represented by a null pointer.



**Forest:** It's a set of disjoint trees. In any tree if we remove its root node then it becomes a forest.

### Binary Trees:

A binary tree is a finite collection of nodes which is either empty or it consists of node called root together with two disjoint binary trees called left-subtree and right-subtree of root.

The left-subtree, right-subtree may also empty. The general representation of binary tree is given as:

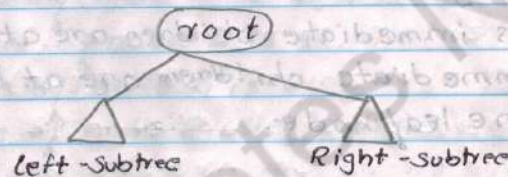


Fig: Representation of binary tree.

Example:

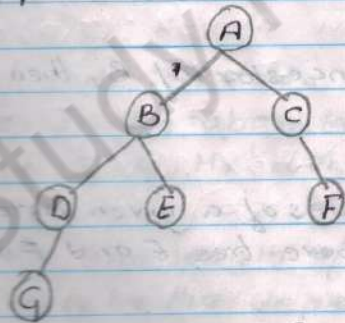


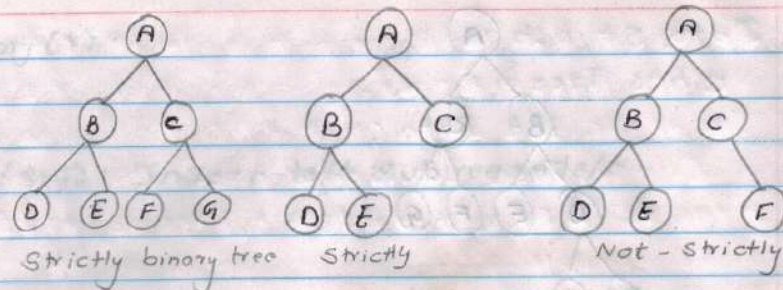
Fig: Binary tree with 7 nodes rooted at node A

### Types of binary tree

#### i) Strictly binary tree:

If every non-leaf node of binary tree has empty left and right sub-tree, then it is strictly binary tree.





## 2) Complete binary tree:

A complete binary tree with  $n$  nodes and of depth  $d$  is a strictly binary tree, all of whose terminal nodes are at level  $d$ .

For eg:

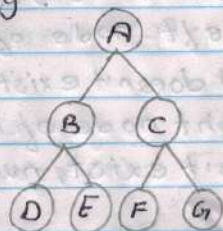


Fig: Complete binary tree

A complete binary tree with depth  $d$  has  $2^d$  leaf nodes and  $2^d - 1$ , non-leaf nodes (non-terminal nodes).

## 3) Almost complete binary tree:

A binary tree of depth  $d$  is an almost <sup>complete</sup> binary tree if:

- i) Any node  $n_d$  at level less than  $d-1$  has two child.
- ii) For any node  $n_d$  in the tree with right descendent at level  $d$ ,  $n_d$  must have a left child and every left descendent of  $n_d$  is either a leaf at level  $d$  or has two child.

For eg:



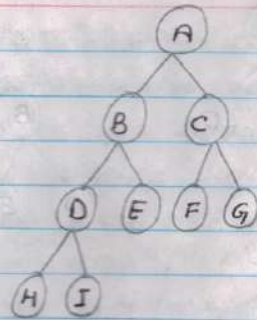


Fig: Almost complete binary tree.

### Operations on Binary Tree

- 1) Parent( $n, T$ ): Return the parent node of the node  $n$  in tree  $T$ . If  $n$  is the root, null is returned.
- 2) Leftchild( $n, T$ ): Return the left node of the node  $n$  in tree  $T$ . If  $n$  is the leftchild doesn't exist, null is returned.
- 3) Rightchild( $n, T$ ): Return the right node of the node  $n$  in tree  $T$ . If rightchild doesn't exist, null is returned.
- 4) info( $n, T$ ): Return the info part of the node  $n$  and tree  $T$ .
- 5) Siblings( $n, T$ ): Return the sibling of the node  $n$  in tree  $T$ . If sibling doesnot exist, null is returned.
- 6) Root( $T$ ): Return the root of the tree.
- 7) Size( $T$ ): return the total number of nodes present in a tree.



- 8) MakeEmpty (T) :
- 9) SetLeft (S, T) : Insert subtree in left
- 10) SetRight (S, T) : Insert subtree in right
- 11) Preorder (T) : Traverse the node in VLR form.
- 12) Postorder (T) : Traverse the node in LRV form.
- 13) Inorder (T) : Traverse the node in LVR form.

Representation of binary tree (linked list representation)  
 The basic component to be represented in binary tree is a node. The node consists of three fields such as :

- 1) Data
- 2) left child
- 3) Right child.

The structure of node is given as.

|        |      |        |
|--------|------|--------|
| Lchild | Data | Rchild |
|--------|------|--------|



Data Structure.

class Tree {

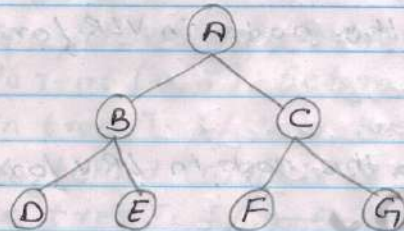
int Data;

Tree Lchild;

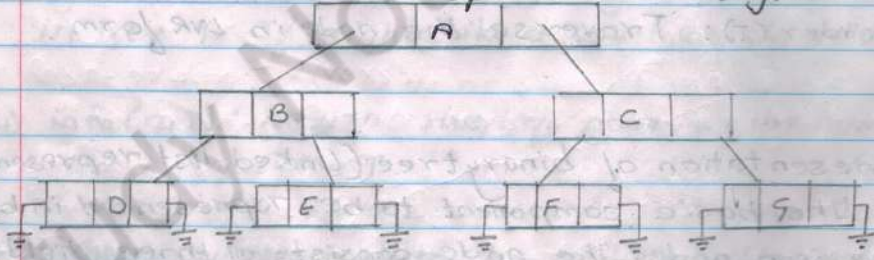
Tree Rchild;

}

Consider the following binary tree.



Then its linked list representation is given as :



Binary Tree Traversal

The tree traversal is a way in which each node in the tree is visited exactly once. There are three methods of tree traversal:

- \* Pre-order traversal
- \* In-order traversal
- \* Post-order traversal



### \* Pre-order Traversal:

The pre-order traversal of non empty binary tree is defined as,

- Visit the root node
- Traverse the left sub-tree in preorder.
- Traverse the right sub-tree in preorder.

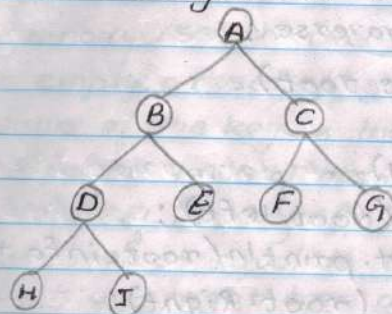


Fig: Binary Tree

The preorder traversal output of given tree is  
A B D H I E C F G

Method for preorder traversal

```
void preorder(Tree root)
```

```
{
```

```
    if (root != null)
```

```
    {
```

```
        System.out.println(root.info + " ");
```

```
        preorder(root.left);
```

```
        preorder(root.right);
```

```
    }
```

```
}
```

### \* In-order Traversal

The in-order traversal of non-empty binary tree is defined as,





- Traverse the left sub-tree in in-order.
- Visit the root node.

- Traverse the right sub-tree in in-order.

Considering the above tree, the pre-order traversal output of given tree is **HIDEBFGCA**.  
**HDIBEAFCG**.

Method for in-order traverse:

```
void inorder(Tree root)
{
    if (root != null)
    {
        inorder(root.left);
        System.out.println(root.info + " ");
        inorder(root.right);
    }
}
```

#### \* Post-order traversal.

The post-order traversal of nonempty binary tree is defined as,

- Traverse the left sub-tree in post-order.
- Traverse the right sub-tree in post-order.
- Visit the root node.

Considering the above tree, the post-order traversal output of given tree is **HIDEBFGCA**.

Method for post-order traverse:

```
void postorder(Tree root)
{
```

```
    if (root != null)
    {
```

```
        postorder(root.left);
```



```

postorder(root, Right);
System.out.println (root.info + " ");
}
}

```

### Binary Search Tree

A binary search tree is a binary tree which is either empty or satisfies the following conditions:

- 1) The value of the key in the left child or left subtree is less than the value of the root.
- 2) The value of the key in the right child or right subtree is more than or equal to the value of root.
- 3) Eg: left and right tree again must be binary search tree.

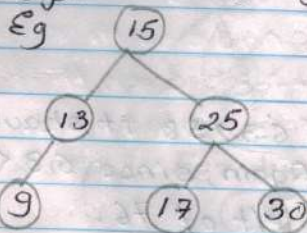


Fig: BST

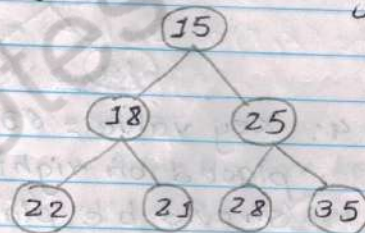


Fig: Not BST

### Application:

- Sorting, Searching
- Networking

### Example:

Construct the BST of following data value.

58, 76, 14, 63, 78, 43, 6, 11, 61.

Soln.

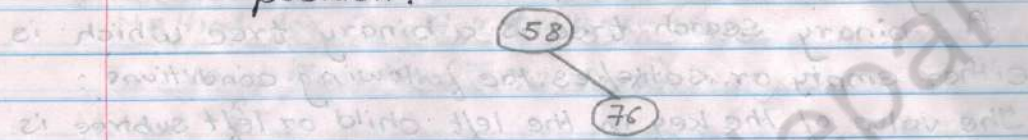
Step 1: Initially, the tree is empty so place the first data value at the root.

(58)

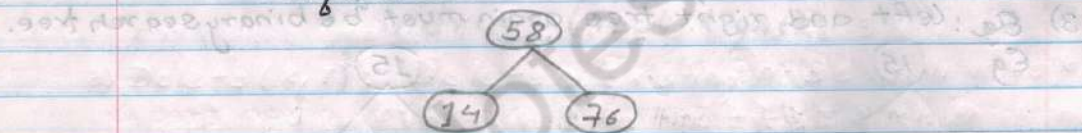


Step 2 : Compare the next data value 76 with root. If the data value is greater than or equal to the root then place it in the right child position, otherwise place it into the left child position.

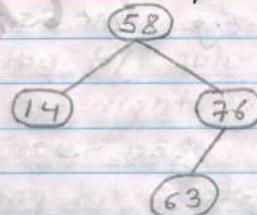
Since,  $76 > 58$ , so place it into right child position.



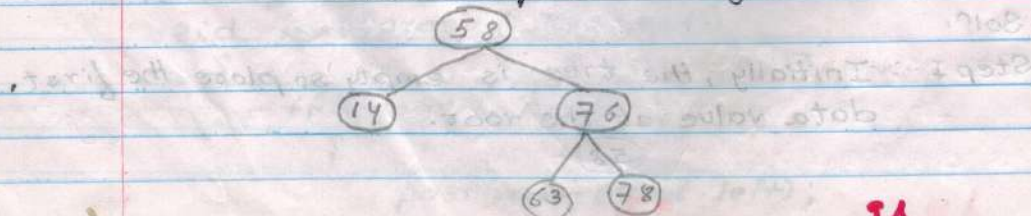
Step 3: Compare the next data value 14 with root, which is less than root so it is placed in left side.



Step 4: Key value = 63. Since  $63 > 58$  it should be placed on right side. Again, since  $63 < 76$  it should be placed on left of 76.

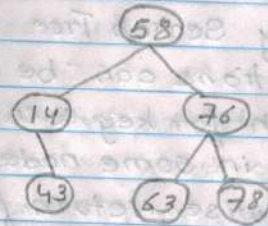


Step 5: key value = 78. Since,  $78 > 58$  so it should be placed on right side. Again, ~~78~~<sup>78 > 76</sup> ~~< 76~~, it should be placed on right side.

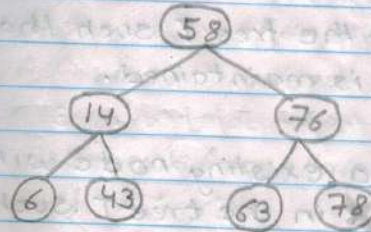




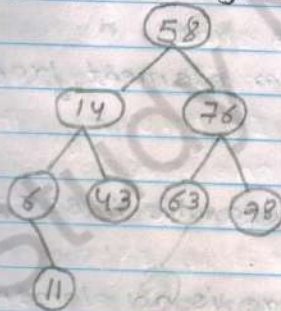
Step 6 : key value = 43 . Since  $43 < 58$ , So it should be placed on left side .  $43 > 14$  it should be placed on right .



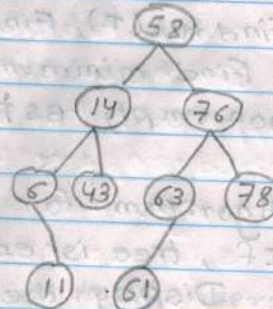
Step 7 : For key value = 6



Step 8 : For key value = 11



Step 9 : For key value = 61



BST representation

```
class BST {
```

```
    int info;
```

```
    BST left;
```

```
    BST right;
```

```
}
```



Step 1: BST root = new BST ();  
Step 2: root = null;

### Operations on Binary Search Tree

Following operations can be done in BST:

1) Search (k, T): Search for key k in the tree T. If k is found in some node of the tree return true otherwise return false.

2) 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.

3) Delete (k, T): Delete a existing node with value k in the info field in the tree T such that the property of BST is maintained.

4) Find Min (T), Find Max (T):  
Find minimum and maximum element from the given non empty BST.

Algorithm for searching a Data value in BST:

1) If, tree is empty then,  
Display the message "There is no elements for searching"  
otherwise,

2) If (Search item is greater than root) then  
Search (root.right);

3) else if (Search item is <sup>less than</sup> equal to root) then  
Search (root.left);

Diagram showing a node structure:  
k < root < 10



Method:

```
void Search (int data)
{
    if (root == null)
    {
        System.out.println("There is no element for search");
    }
    else if (root == data)
    {
        return root;
    }
    else if (data > root)
    {
        Search (root.right);
    }
    else
    {
        Search (root.left);
    }
}
```

Insertion of node in BST

Algorithm:

Let 'k' be the key value of the node to be inserted into the BST.

- 1) If tree is empty, then create a new node 'nd', and set data value to this node and make this node as a root node.
- 2) Otherwise, compare k with the value stored into the root node.
- 3) If  $(k < \text{root.data})$  then,  
Insert k into the left subtree of the root node recursively.



4) else

- Insert  $k$  into the right subtree of the root node recursively.

Deleting a Node from the BST.

While deleting a node from BST, first we need to perform a search for deleting element. Once we have found the node to be deleted, there may be three cases as:

1. The node to be deleted, there may be leaf node.

The node is a leaf, if it has no children. This is the easiest case. The appropriate reference of its parent is set to null.

For eg: Suppose Node to be deleted is 3

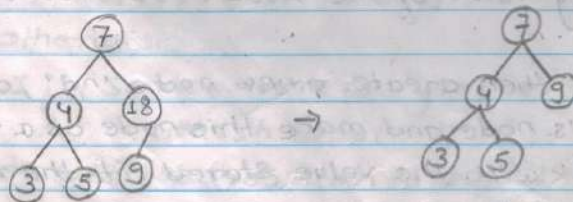


2. The node to be deleted has one child

In this case the child of the node to be deleted is appended to its parent node. In this way, the node's children are lifted up by one level.

For eg:

Suppose Node to be deleted is 18



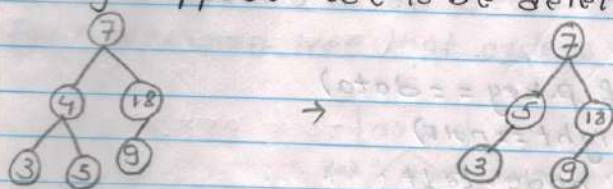


3) The node to be deleted has two children:

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

The node to be deleted is either replaced by its right sub-tree's leftmost node or its left sub-tree's rightmost node.

For eg: Suppose node to be deleted is 4,



Algorithm to delete a node from a BST

1. If a node to be deleted is a leaf node at left side, then simply delete and set null pointer to its parent's left pointer.
2. If a node to be deleted is a leaf node at right side, then simply delete and set null pointer to its parent's right pointer.
3. If a node is to be deleted has one child, then connect its child pointer with its parent pointer and delete it from the tree.
4. If a node to be deleted has two children, then replace the node being deleted either by a right most node of its left sub-tree or left most node of its right sub-tree (replace their in-order successor node).

Java function for deleting element from the BST.

Here, after deleting node from BST, we have two sub-trees. Thus we need to merge these two nodes properly. This type of deleting element from BST is called deletion by merging operation.

```
void deleteBST (int data)?
```

```
BST temp, node, p = root, prev = null;
```



```

while (p != null && p.key != data)
{
    prev = p;
    if (p.key < data)
        p = p.right;
    else
        p = p.left;
}
node = p;
if (p != null && p.key == data)
{
    if (node.right == null)
        node = node.left;
    else if (node.left == null)
        node = node.right;
    else {
        temp = node.left;
        while (temp.right != null)
            temp = temp.right;
        temp.right = node.right;
        node = node.left;
    }
    if (p == root)
        root = node;
    else if (prev.left == p)
        prev.left = node;
    else
        prev.right = node;
}
else if (root != null)
    System.out.println("key" + data + "is not in tree");
else
    System.out.println("The tree is empty");
}

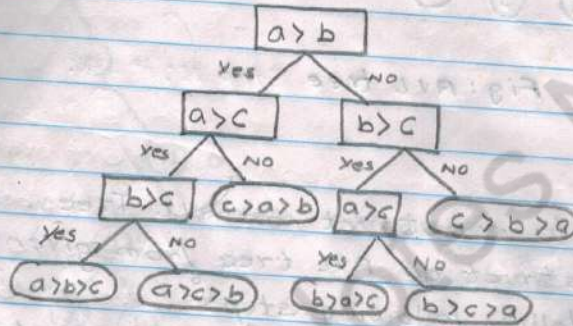
```



## Decision Trees

Binary trees can be used to locate item based on comparisons, in this situation each comparison tells us to visit either left sub-tree or right sub-tree. A rooted tree where internal vertices correspond to a decision and sub-tree at these vertices gives possible outcomes of the decision made is called decision trees.

Eg: A decision tree that orders the elements of the list  $a, b, c$ .



## AVL Tree (Height balanced tree)

An AVL Tree, after the Russian mathematician G.M. Adelson-Velskil and E.M. Landis, is a BST, in which the height of left and right sub-trees of the root differ by at most 1 and in which left and right subtrees are again AVL trees. It is also called height balanced BST.

With each node of AVL tree, a balance factor, is associated i.e. the left sub-tree has greater than equal to or less than the height of the right sub-tree. But the height difference bet<sup>n</sup> left sub-trees is either 1 or 0 or -1. So the balance factor of each node of AVL tree must be 1, 0 or -1.



Example:

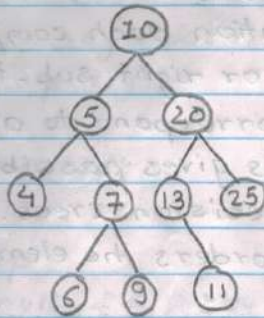


Fig: AVL tree

Procedure to construct an AVL Tree

To construct an AVL tree from given set of data points we follow following steps:

- 1) Take a first element of given array of elements and make it as first node of AVL tree.
- 2) Set, next node either left side or right side of AVL tree.
- 3) Continue this process until all the elements are not included in resulting AVL tree and maintain balance factor for each node either -1 or 0 or 1.
- 4) If balance factor of a particular node is not in given range (-1, 0 or 1) then rotate either single or double.
- 5) If straight path take place, then perform single rotation.
- 6) Otherwise perform double rotation.

Example: Construct AVL tree from following data:

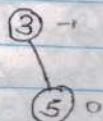
3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10



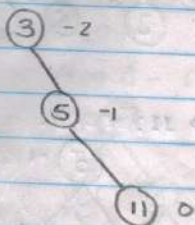
Step 1: Let us consider the root be 3.



Step 2: Then insert 5.

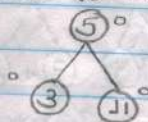


Step 3:

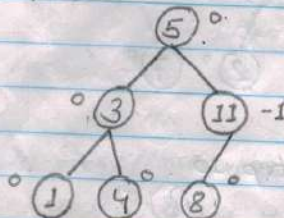


Reconstruct to maintain AVL property.

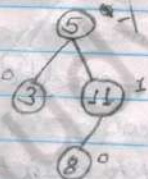
Step 4:



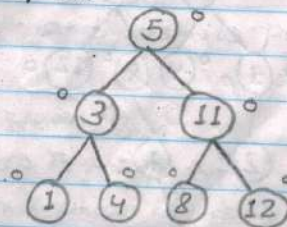
Step 6:



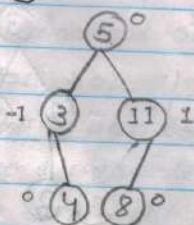
Step 5:



Step 7:

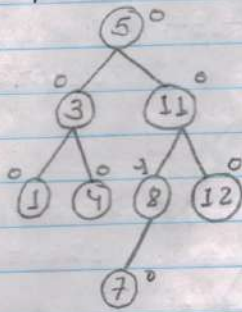


Step 5:

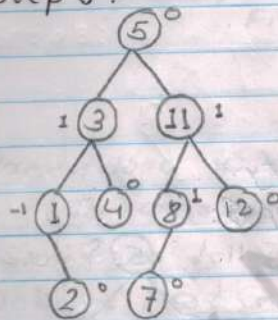




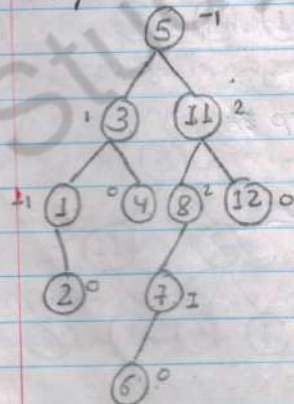
Step 8:



Step 9:

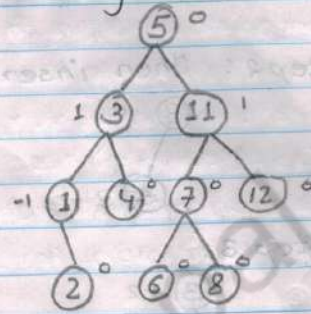


Step 10:

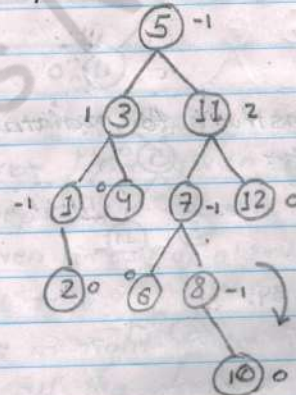


8

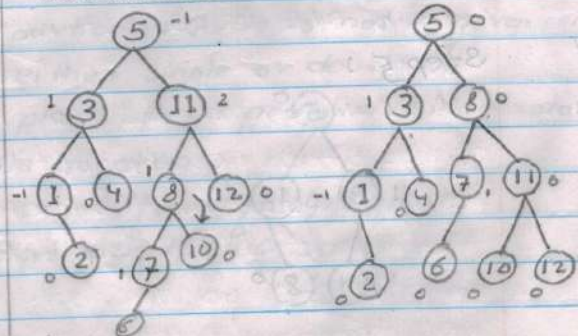
Reconstructing to maintain all proper



Step 11:



Reconstruct to maintain AVL (double rot



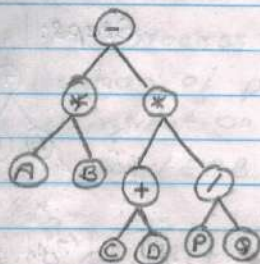


## Expression Tree

A binary tree with each leaf node contains the operands and internal node contains the operators of given expression is called expression tree. During construction of expression tree from given expression we take an operator with lowest precedence as root and set left subtree and right subtree around the root node.

Example: Construct an expression tree of given infix expression:

$$A * B - (C + D) * (P / Q)$$



Infix traverse:  $A * B - C + D * P / Q$

Constructing an expression tree from postfix expression

Algorithm:

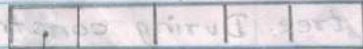
- 1) Scan given postfix expression from left to right.
- 2) If scanned symbol is operand, then
  - Make a tree with only one node and push its pointer to the top of stack.
- 3) If scanned symbol is operator, then
  - Pop top element and place it to  $T_2$
  - Again pop top of the element and place it to  $T_1$ .
- 4) Make a new tree with root node as given operator and its left and right subtree as  $T_1$  and  $T_2$  respectively.
- 5) Continue this process until scanning pointer do not reach at last location of given postfix expression.



Example: Construct an expression tree of the postfix exp

$AB + CDE + * \$$

Step 1:



A

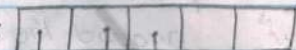
Step 2:



A

B

Step 7:



A

B

C

D

Step 3:



A

B

Step 8:



A

B

C

D

Step 4:



A

B

C

Step 9:



A

B

C

D

Step 5:



A

B

C

A

B

C

D

Step 6:



A

B

C

D

E

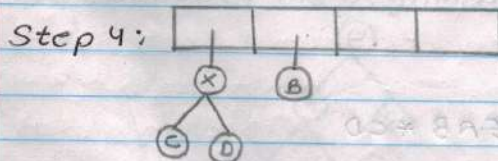
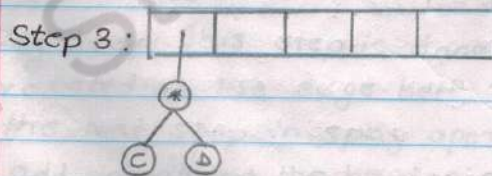
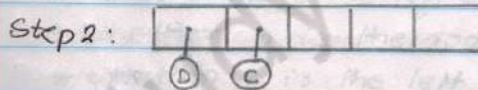


Constructing an expression tree from the prefix expression :

- 1) Scan given prefix expression from right to left.
- 2) If scanned symbol is operand, then
  - \* Make a tree with only one node and push its pointer to the top of stack.
- 3) If scanned symbol is operator, then
  - \* Pop top element and place it to  $T_1$ .
  - \* Pop top element and place it to  $T_2$ .
- 4) Make a new tree with root node as given operator and its left and right subtree as  $T_1$  and  $T_2$  respectively.
- 5) Continue this process until scanning pointer do not reach at last location of prefix expression.

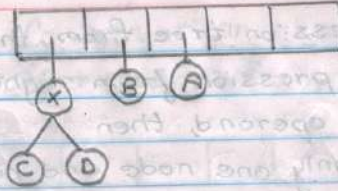
Example: Construct an expression tree of the prefix expn.

$\$ + / E A B * C D$

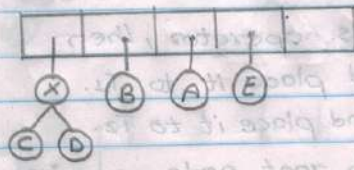




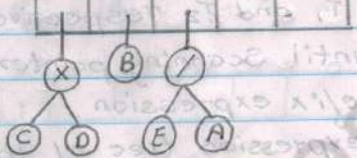
Step 5:



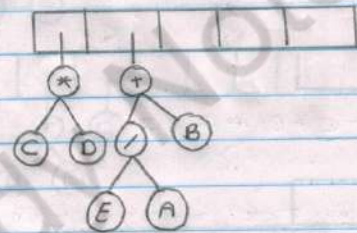
Step 6:



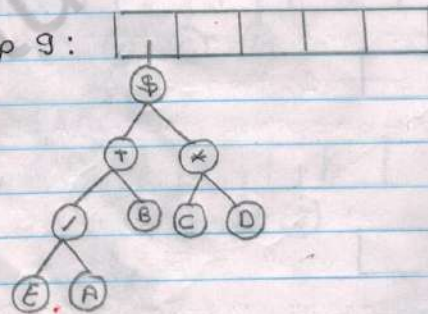
Step 7:



Step 8:



Step 9:



Prefix traverse = \$+/EAB\*CD



## Splay Tree

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. <sup>splaying property</sup>

All normal operations on binary search tree are combined with one basic operation, called splaying. Splaying the tree for a certain element rearrange the tree so that the element is placed at the root of the tree.

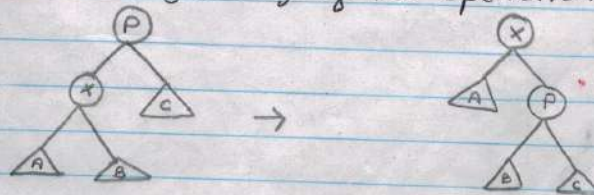
### Splaying:

When a node  $x$  is accessed, a splay operation is performed on  $x$  to move it to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves  $x$  closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root.

Each particular step depends on three factors:

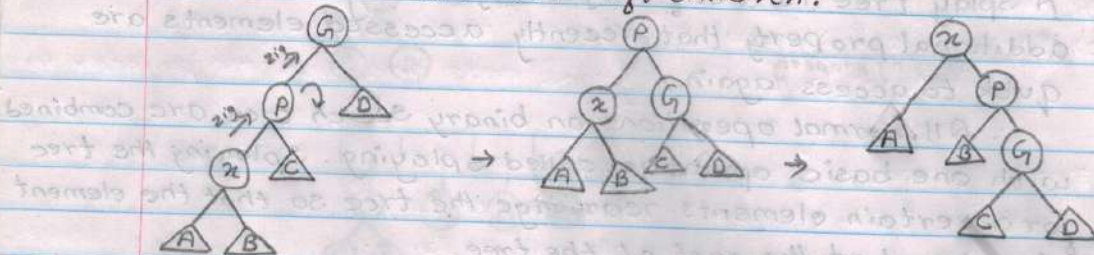
- \* whether  $x$  is the left or right child of its parent node,  $p$ .
- \* whether  $p$  is the root or not, and if not,
- \* whether  $p$  is the left or right child of its parent,  $g$ .

1) Zig step: This step is done when  $p$  is the root. The tree is rotated on the edge between  $x$  and  $p$ . Zig steps exist only as the last step in splay operation and used only when  $x$  has odd depth at the beginning of the operation.

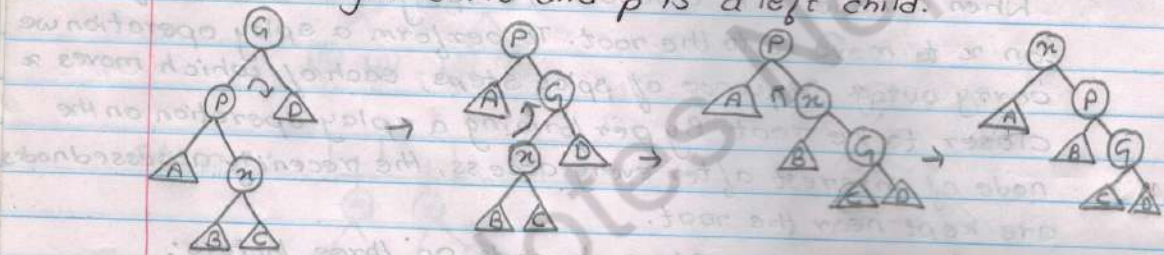




2) Zig-Zig Step: This step is done when  $p$  is not the root and  $x$  and  $p$  are both left children.

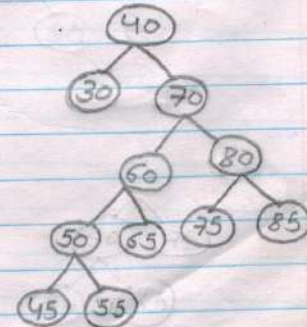
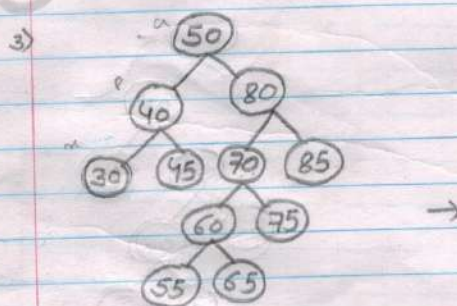
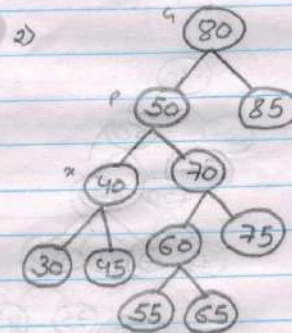
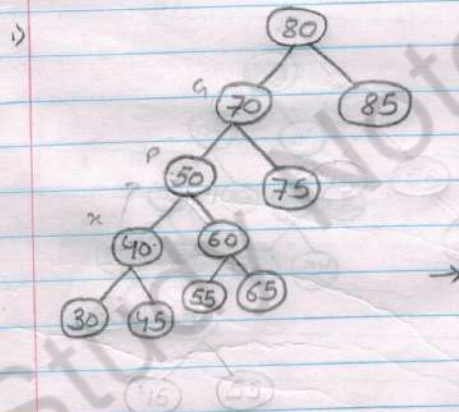
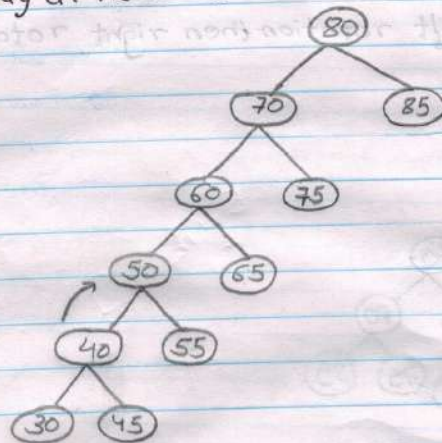


3) Zig-zag step: This step is done when  $p$  is not the root and  $x$  is a right child and  $p$  is a left child.





Q. Consider the following splay tree.  
Splay at node 40

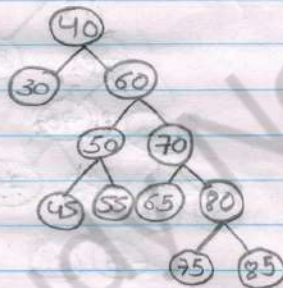
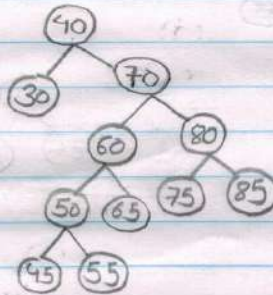




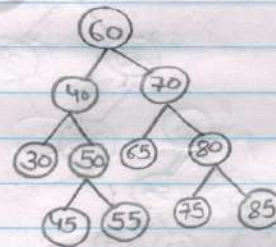
Zig-Zig: Parent - grand parent is rotated first.

Zig-zag: first left rotation then right rotation.

Splay at 60

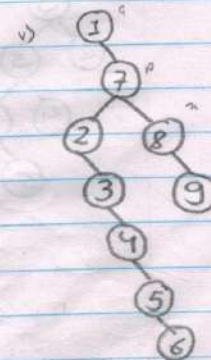
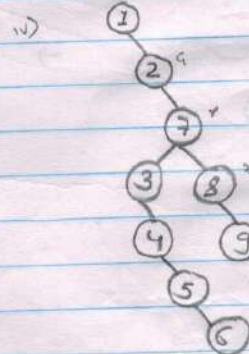
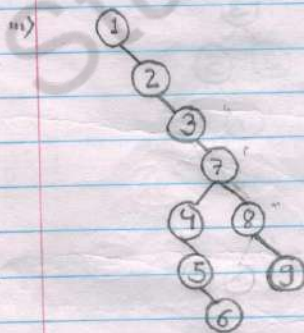
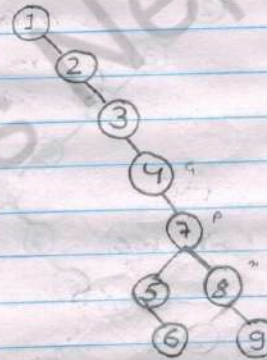
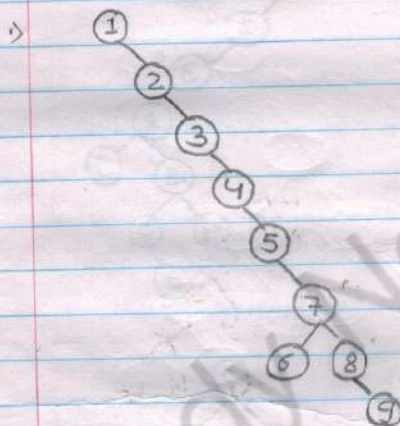
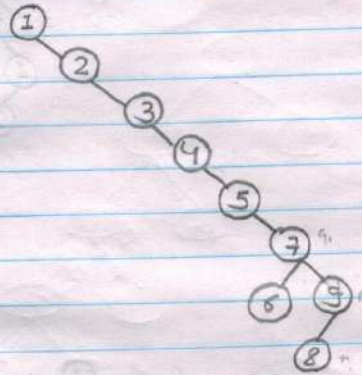


→

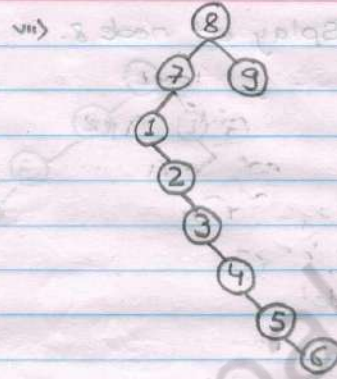
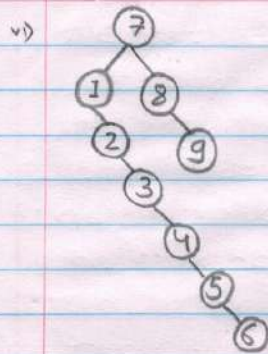




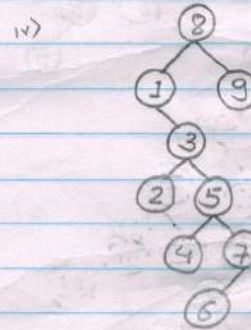
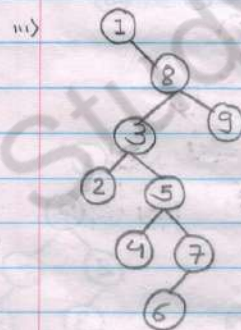
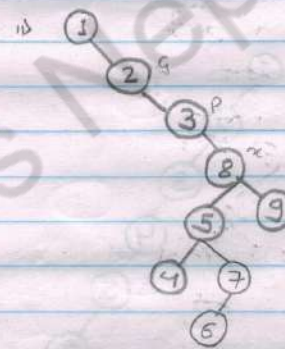
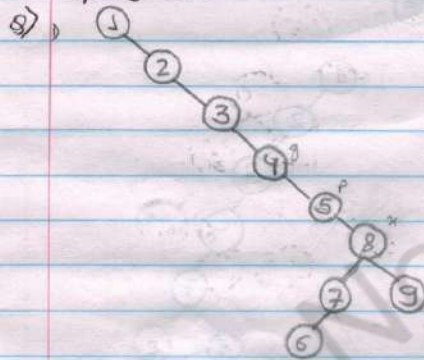
Q. Splay at node 8.





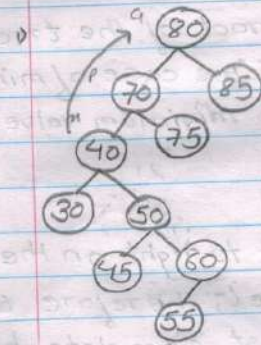
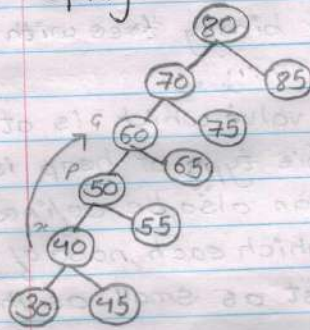


Splay at 8.

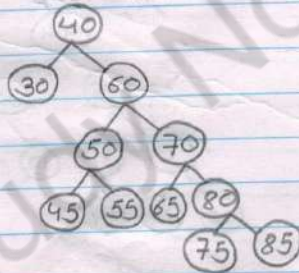




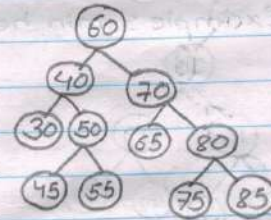
Splay at 40



Splay at 60



→



\* Heaps

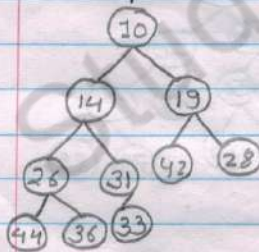


### Heaps:

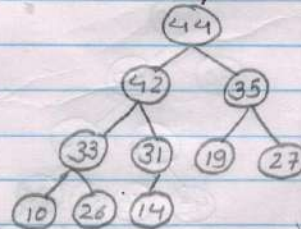
A heap is a almost complete binary tree with the following properties.

- ① Each node of the tree has the value which is at least as the value of its children. This type of heap is called the max heap. The min heap can also be defined in the similar manner as the tree in which each node of the tree has a value that is at least as small as the value of its children.
- ② In the case of max heap the root of the tree contains the maximum value and in the case of min heap the root of the tree contains the minimum value if all the values are distinct.
- ③ The elements are filled from left to right on the same level before moving to the next level, therefore the heap is usually defined as almost complete binary tree. Hence every leaf node is of depth  $d$  or  $d-1$ .

Example: min heap



max heap



A max or min heap can be implemented using an array where if  $x[i]$  is the element stored at the root then its left child will be at  $x[2i+1]$  and right child will be at  $x[2i+2]$ .

Here we must remember that the array index starts from 0 to  $\text{max} - 1$ . The parent of a node present



at index  $i$  can be found at  $(i-1)/2$ .

Operations on heaps:

- 1) Maintain the max-heap property
  - Max heapify
- 2) Create a max-heap from an unordered array.
  - Build max-heap
- 3) Sort an array in place.
  - Heap sort.

Heapify property:

If any node violates the heap property then swap this node with its larger children to maintain the heap property, this operation is called heapify.

Max-heapify operation:

- Find location of largest value of:  
 $A[i], A[\text{left}(i)], A[\text{right}(i)]$
- If not  $A[i]$ , max-heap property doesn't hold.
- Exchange  $A[i]$  with the larger of the two children to preserve max-heap property.
- Continue this process / exchange down the heap until sub-tree rooted at  $i$  is a max heap.
- At a leaf, the sub-tree rooted at leaf is trivially a max heap.

Building a Heap:

The process of converting a given binary tree into a heap by performing heap operation on each of the non-leaf node of the tree is known as building heap operation.

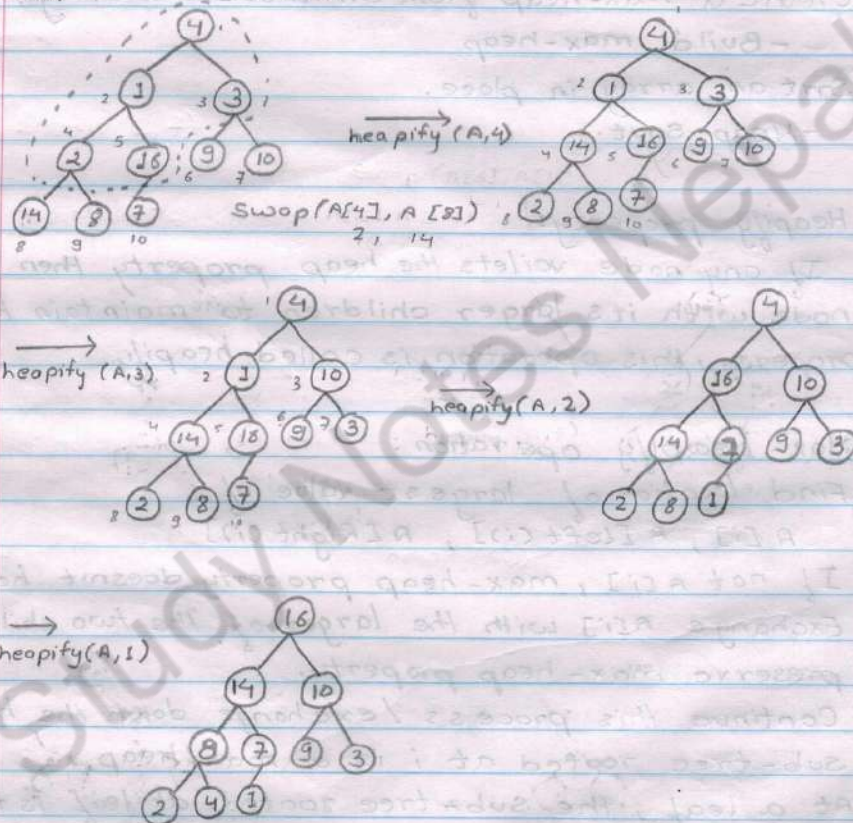


Example:

Construct binary tree of following data values and then build heap by performing heapify operation on every non-leaf nodes.

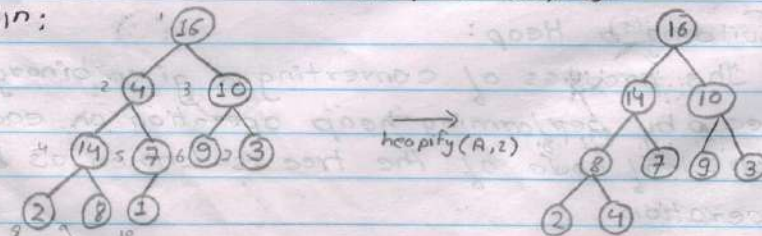
Q.  $A[] = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$

Soln: Scramble tree



Q.  $A[] = \{16, 4, 10, 14, 7, 9, 3, 2, 8, 1\}$

Soln:





## Heap sort

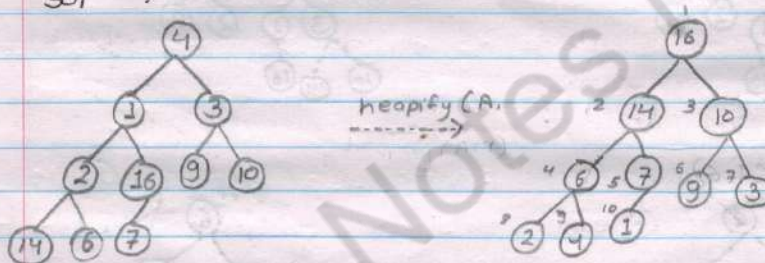
To sort the given set of  $n$  elements by using heap sort, it follows the following steps.

- ① Build a max-heap from the array.
- ② Swap the root (the maximum element) with the last element in the array.
- ③ "Discard" this last node by decreasing the heap size.
- ④ Perform Max-heapify operation on the new root node.
- ⑤ Repeat this process until only one node remains.

Example: Sort following elements by using heap sort.

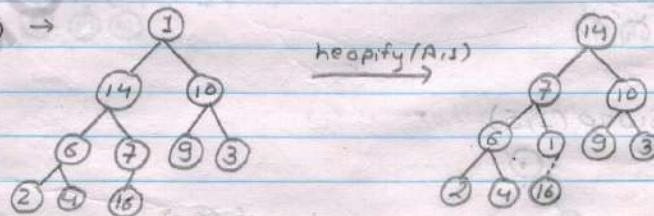
$A[] = \{4, 1, 3, 2, 16, 9, 10, 14, 6, 7\}$

Sol<sup>n</sup> :

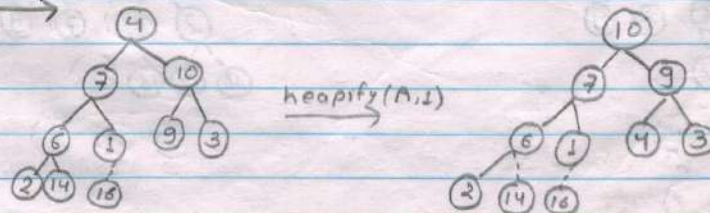


Heap sort

Swap(1, 10) →

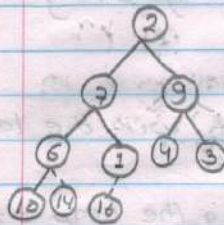


Swap(1, 9)

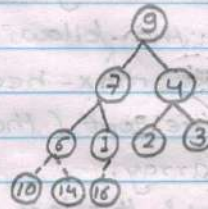




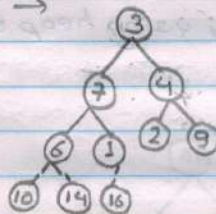
Swap(1,8)



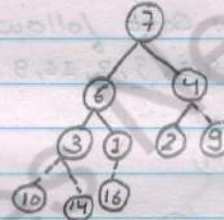
heapify(A,1)



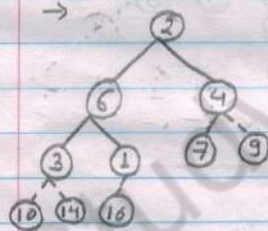
Swap(1,7)



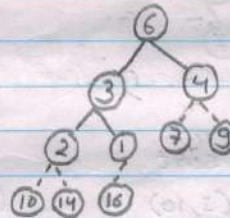
heapify(A,1)



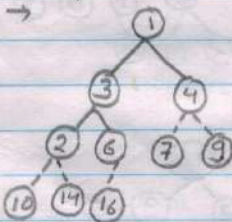
Swap(1,6)



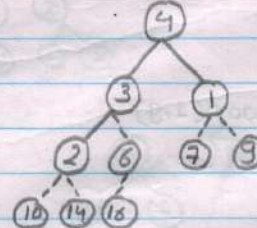
heapify(A,1)



Swap(1,5)

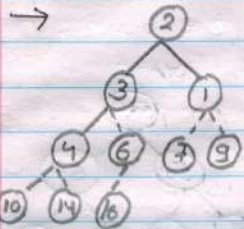


heapify(A,1)

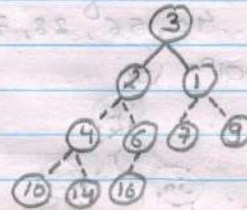




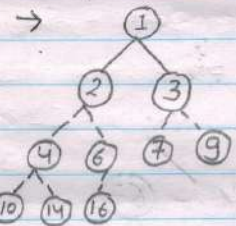
Swap(1, 4)



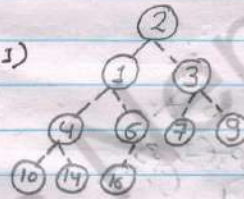
heapify(A, 1)



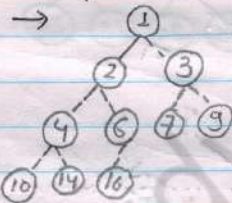
Swap(1, 3)



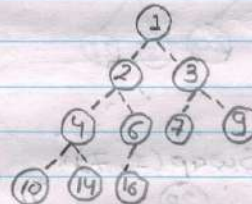
heapify(A, 1)



Swap(1, 2)



heapify(A, 1)



Algorithm

Heapsort(A)

{

    Buildheap(A);

    n = length[A];

    for (i = n; i >= 2; i--)

    {

        swap(A[i], A[n]);

        n = n - 1;

        Heapify(A, i);

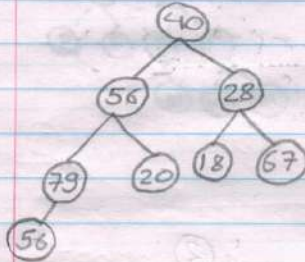
    }

}

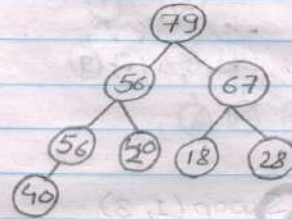


Q. Sort the following datas using heap sort.  
40, 56, 28, 79, 20, 18, 67, 56.

Soln,

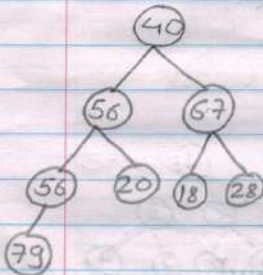


heapify(A, ...)

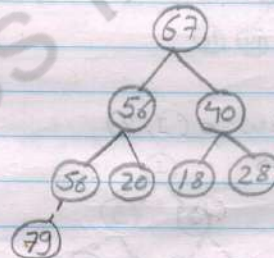


Heap sort:

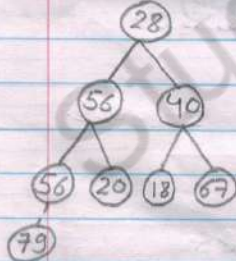
Swap (1, 8)



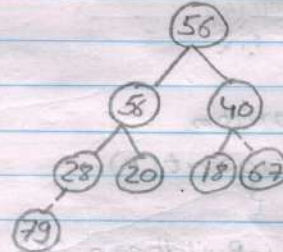
heapify(A, 1)



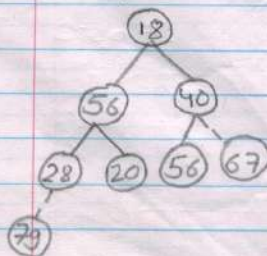
Swap (1, 7)



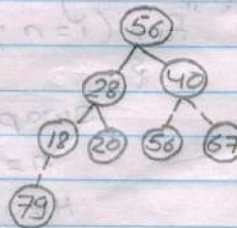
heapify(A, 1)



Swap (1, 6)

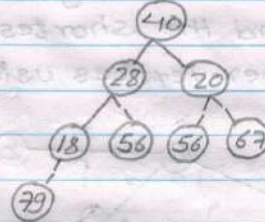
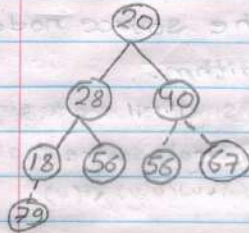


heapify(A, 1)

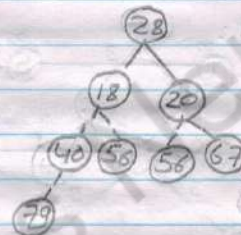
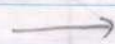
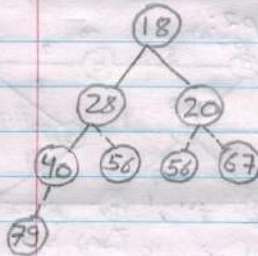




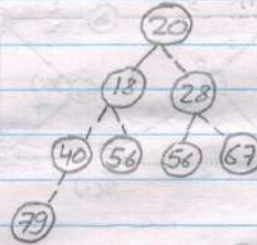
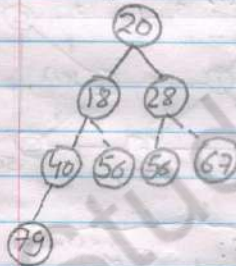
Swap (1,5)



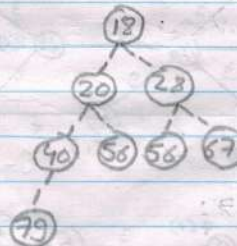
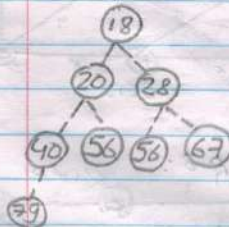
Swap (1,4)



Swap (1,3)



Swap (1,2)

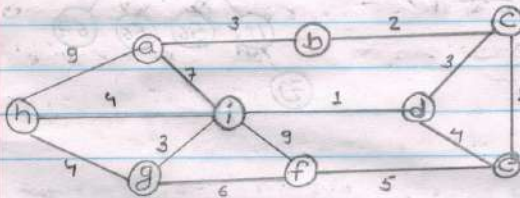




# Graphs

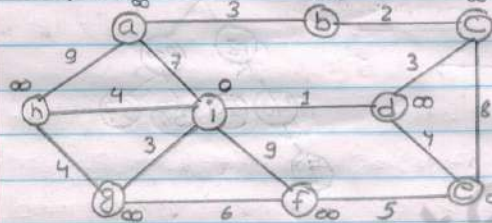
## Dijkstra's Algorithm Tracing

Find the shortest paths from the source node i to all other vertices using Dijkstra's algorithm.

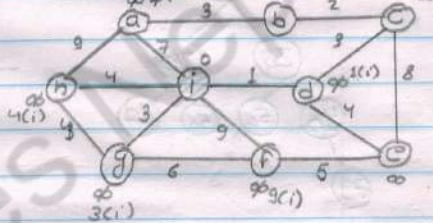


- ① Single source to single destination
  - ② Single source to all destination
- $L(u) + w(u, v) < L(v)$

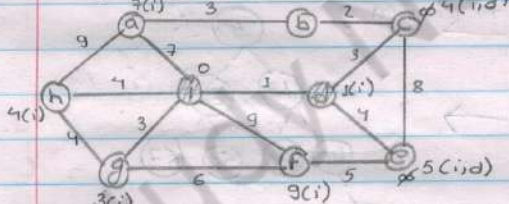
Step 1:



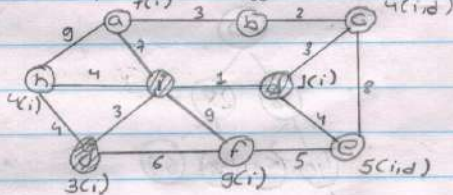
Step 2:



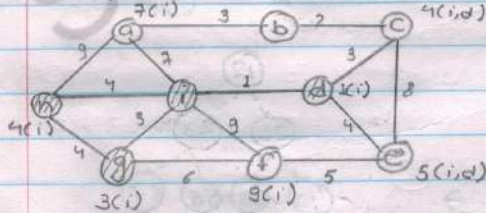
Step 3:



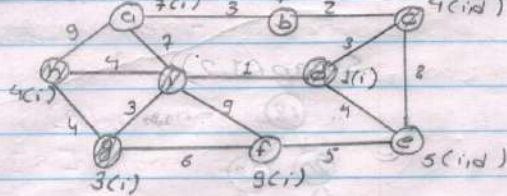
Step 4:



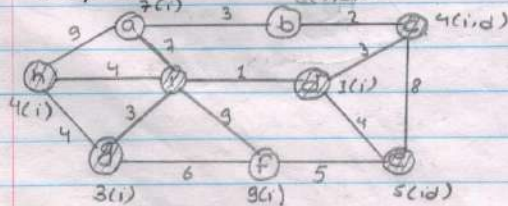
Step 5:



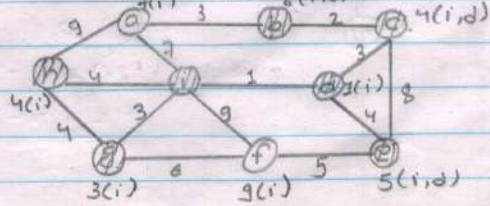
Step 6:



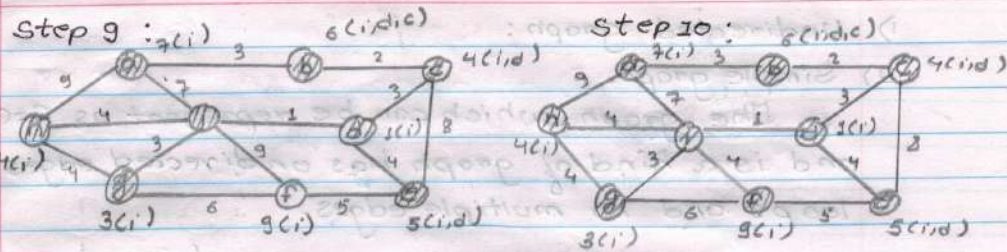
Step 7:



Step 8:







The shortest path from  $i$  to  $a$  is 7

$i$  to  $b$  is 6 ( $i,d,c$ )

$i$  to  $c$  is 4 ( $i,d$ )

$i$  to  $d$  is 1 ( $i$ )

$i$  to  $e$  is 5 ( $i,d$ )

$i$  to  $f$  is 9 ( $i$ )

$i$  to  $g$  is 3 ( $i$ )

$i$  to  $h$  is 4 ( $i$ )

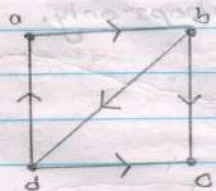
Graph:

Graph is a discrete structure consisting of vertices and edges connecting the vertices. A graph  $G=(V,E)$  is a mathematical model consist of two non empty sets:  $V=\{v_1, v_2, v_3, \dots, v_n\}$  called set of vertices and  $E=\{e_1, e_2, e_3, \dots, e_n\}$  of ordered or unordered pairs of distinct vertices called edges.

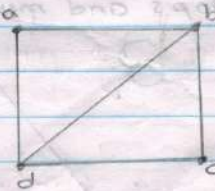
There are two types of graph. They are:

i) Directed graph

ii) Undirected graph.



Directed graph



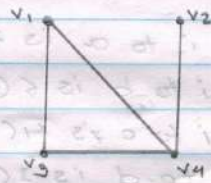
Undirected graph



i) Undirected graph:

a) Simple graph.

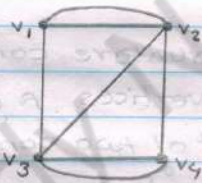
The graph which can be represent as  $G=(V,E)$  is, and is a kind of graph has un directed edges , no loops and no multiple edges.



Simple graph.

b) Multigraph.

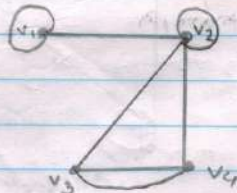
A multigraph  $G=(V,E)$  is a type of graph that can have multiple edges between the same two vertices and has undirected edges and no loops.



Multigraph

c) Pseudo graph:

A graph  $G=(V,E)$  consists of a set of vertices  $V$ , a set of edges  $E$  is said to be pseudo graph if  $G$  has both loops and multiple edges or loops only.



Pseudo graph.

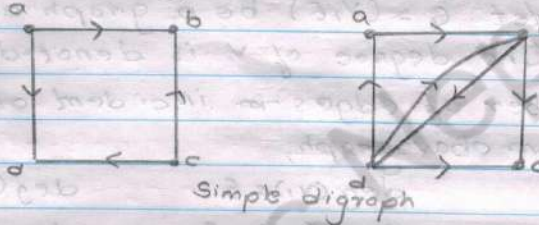


1) Directed graph (Di graph)

a) ~~Simple digraph~~: A directed graph or digraph  $D=(V,E)$  consists a set  $V=V(D)$  whose elements are called vertices of  $D$  and a set  $E=E(D)$  of ordered pairs  $(u,v)$  of vertices called directed edges or simply edges.

a) Simple digraph:

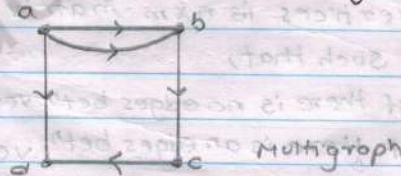
A directed graph  $D=(V,E)$  which has neither loops nor multiple edges is called simple directed graph. Eg:



b) Multidigraph

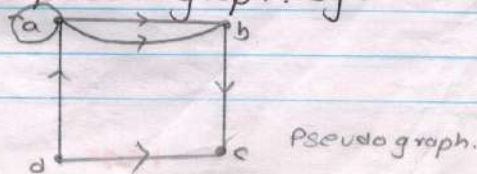
In any digraph, two or more edges are said to be parallel if they have same initial and terminal vertex with same direction.

A digraph  $D=(V,E)$  which contains parallel edges is called a multi directed graph. Eg:



c) Pseudograph

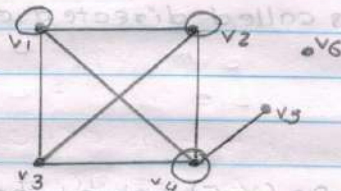
A digraph  $D=(V,E)$  which contain self/loop is called a pseudo graph. Eg:





Order and size of graph:

If  $G = (V, E)$  be a graph then the number of vertices in graph  $G$  is called order of graph  $G$  and the number of edges in  $G$  is called size of  $G$ .



Order of  $G = 6$

Size of  $G = 10$

Degree of vertex:

Let  $G = (V, E)$  be a graph and  $v$  be a vertex of  $G$ . The degree of  $v$  is denoted by  $\deg(v)$  is the number of edges incident on  $v$ .

From above graph,

$$\deg(v_1) = 5$$

$$\deg(v_4) = 6$$

$$\deg(v_2) = 5$$

$$\deg(v_5) = 1$$

$$\deg(v_3) = 3$$

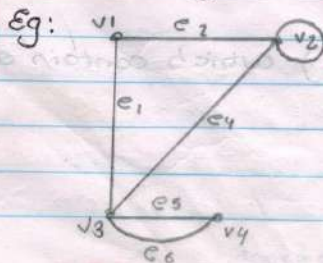
$$\deg(v_6) = 0$$

Representation of Graph:

i) Adjacency matrix:

Let  $G = (V, E)$  be a graph with  $n$  vertices,  $v_1, v_2, \dots, v_n$ . The adjacency matrix of  $G$  with respect to given ordered list of vertices is  $n \times n$  matrix denoted by  $A(G) = (a_{ij})_{n \times n}$  such that

$$a_{ij} = \begin{cases} 0 & \text{if there is no edges betn vertices } v_i \text{ and } v_j \\ 1 & \text{if there is an edges betn vertices } v_i \text{ and } v_j \\ k & \text{if there are } k( \geq 2 ) \text{ edges betn vertices } v_i \text{ and } v_j \end{cases}$$





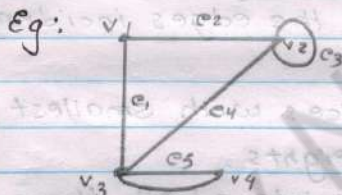
Sol<sup>n</sup>: Since  $G$  has 4 vertices, adjacency matrix  $A(G)$  will be a  $4 \times 4$  matrix as:

$$A(G) = \begin{matrix} & \begin{matrix} v_1 & v_2 & v_3 & v_4 \end{matrix} \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 2 \\ 0 & 0 & 2 & 0 \end{bmatrix} \end{matrix}$$

ii) Incidence matrix:

Let  $G$  be a graph with vertices  $v_1, v_2, \dots, v_m$  and edges  $e_1, e_2, \dots, e_n$ . The incidence matrix  $I(G)$  of graph  $G$  is  $m \times n$  matrix with  $I(G) = (m_{ij})_{m \times n}$  where,

$$m_{ij} = \begin{cases} 0 & \text{if } e_j \text{ is not incident with } v_i \\ 1 & \text{if } e_j \text{ is incidence with } v_i \\ 2 & \text{if } v_i \text{ is the end of the loop.} \end{cases}$$



Sol<sup>n</sup>: Since  $G$  has 6 edges, incidence matrix  $I(G)$  will be  $4 \times 6$  matrix as:

$$I(G) = \begin{matrix} & \begin{matrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{matrix} \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} & \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{matrix}$$

iii) Adjacency list

One of the way of representing a graph without multiple edges is by listing its edge. This type of representation is suitable for the undirected graphs, without multiple edges, and directed graphs. This



representation looks as in the table below:

| vertex | Adjacent vertices |
|--------|-------------------|
| a      | b, c              |
| b      | a, d              |
| c      | a, d              |
| d      | b, c              |



**Dijkstra's Algorithm:**

- Step 1: Label the initial vertex of the graph with weight zero and all other vertices are labelled with  $\infty$ .
- Step 2: Calculate the weights of all vertices adjacent to the weight of the edges incident on the initial vertex.
- Step 3: Label these vertices with smallest possible value of their weights.
- Step 4: Calculate the weights of all these vertices which are adjacent to the vertices with minimum weight determined in Step 3.
- Step 5: Label these vertices with minimum weight.
- Step 6: Continue this process until all the vertices of weighted graph are labelled.
- Step 7: Trace the path of cumulative minimum weight from the initial vertex to desired vertex.

**Pseudo code (Dijkstra's Algorithm)**

Dijkstra ( $G$ : weighted connected simple graph, with all vertices have positive weight).

$G$  has vertices  $a = v_0, v_1, \dots, v_n = z$  and weights  $w(v_i, v_j)$



where  $w(v_i, v_j) = \infty$  if  $(v_i, v_j)$  is not an edge in  $G$ .

- for  $i = 1$  to  $n$

-  $L(v) = \infty$

-  $L(a) = 0$

-  $S = \emptyset$

{ the labels are now initialized so that the label of  $a$  is 0 and all others label are  $\infty$ , and  $S$  is the empty set }

- while  $z \notin S$

$u =$  a vertex not in  $S$  with  $L(u)$  minimum.

$S = S \cup \{u\}$

for all vertices  $v$  not in  $S$

if  $L(u) + w(u, v) < L(v)$  then

$L(v) = L(u) + w(u, v)$

(this adds a vertex to  $S$  with minimum label and updates the labels of vertices not in  $S$ )

( $L(z) =$  length of the shortest path from  $a$  to  $z$ )

All pairs Shortest path:

This type of problem asks us to find the shortest path from the all vertices to all other vertices in a connected graph.

Problem: Find the shortest path between every pair of vertices of a graph.

Graph: may contain negative edges but no negative cycles.

Representation: A weight matrix where

$w(i, j) = 0$  if  $i = j$  (loop)

$w(i, j) = \infty$  if there is no edges between  $i$  and  $j$

$w(i, j) =$  "weight of edge".



### [Floyd - Warshall]

Floyd's Algorithm to find the shortest path for all pairs of vertices

( $G$ : weighted simple graph)

$\{ G \text{ has vertices } v_1, v_2, \dots, v_n \text{ and weights } w(v_i, v_j) \text{ with } w(v_i, v_j) = \infty \text{ if } (v_i, v_j) \text{ is not edge, and } w(v_i, v_j) = 0 \text{ if } i = j. \}$

for  $i = 1$  to  $n$

for  $j = 1$  to  $n$

$D^0(v_i, v_j) = w(v_i, v_j)$  // initialize array  $D$

for  $k = 1$  to  $n$  // use intermediate  $\{1, \dots, k\}$

for  $i = 1$  to  $n$  // from  $i$

for  $j = 1$  to  $n$  // to  $j$

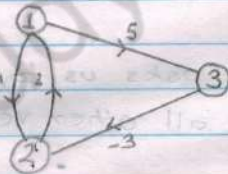
if ( $D^{k-1}[i, j] > D^{k-1}[i, k] + D^{k-1}[k, j]$ )

then  $D^k[i, j] \leftarrow D^{k-1}[i, k] + D^{k-1}[k, j]$

else

$D^k[i, j] \leftarrow D^{k-1}[i, j]$

Example:



Soln

|           | 1        | 2  | 3        |
|-----------|----------|----|----------|
| $D^0 = 1$ | 0        | 4  | 5        |
| 2         | 2        | 0  | $\infty$ |
| 3         | $\infty$ | -3 | 0        |

for  $k = 1$

|           | 1        | 2  | 3        |
|-----------|----------|----|----------|
| 1         | 0        | 4  | 5        |
| $D^0 = 2$ | 2        | 0  | $\infty$ |
| 3         | $\infty$ | -3 | 0        |

|           | 1        | 2  | 3 |
|-----------|----------|----|---|
| 1         | 0        | 4  | 5 |
| $D^1 = 2$ | 2        | 0  | 7 |
| 3         | $\infty$ | -3 | 0 |

$$D^1[2, 3] = \min(D^0[2, 3], D^0[2, 1] + D^0[1, 3])$$

$$= \min(\infty, 2 + 5)$$

$$D^1[2, 3] = 7$$

$$D^1[3, 2] = \min(D^0[3, 2], D^0[3, 1] + D^0[1, 2])$$

$$= \min(-3, \infty + 4)$$

$$= -3$$

131



$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

for  $k=2$

|   | 1  | 2  | 3 |
|---|----|----|---|
| 1 | 0  | 4  | 5 |
| 2 | 2  | 0  | 7 |
| 3 | -1 | -3 | 0 |

$$D^2[1,3] = \min(D^1[1,3], D^1[1,2] + D^1[2,3])$$

$$= \min(5, 4+7)$$

$$= 5$$

$$D^2[3,1] = \min(D^1[3,1], D^1[3,2] + D^1[2,1])$$

$$= \min(\infty, -3+2)$$

$$= \min(\infty, -1)$$

for  $k=3$

|   | 1  | 2  | 3 |
|---|----|----|---|
| 1 | 0  | 2  | 5 |
| 2 | 2  | 0  | 7 |
| 3 | -1 | -3 | 0 |

$$D^3[1,2] = \min(D^2[1,2], D^2[1,3] + D^2[3,2])$$

$$= \min(4, 5-3)$$

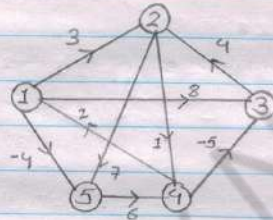
$$= 2$$

$$D^3[2,2] = \min(D^2[2,2], D^2[2,3] + D^2[3,2])$$

$$= \min(7, 7+(-1))$$

$$= 2$$

Q.



Soln,

|   | 1        | 2 | 3  | 4        | 5  |
|---|----------|---|----|----------|----|
| 1 | 0        | 3 | 8  | $\infty$ | -4 |
| 2 | $\infty$ | 0 | 0  | 1        | 7  |
| 3 | 0        | 4 | 0  | 0        | 0  |
| 4 | 2        | 0 | -5 | 0        | 0  |
| 5 | 0        | 0 | 0  | 6        | 0  |

for  $k=1$

|   | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 |   |   |   |   |
| 2 |   | 0 |   |   |   |
| 3 |   |   | 0 |   |   |
| 4 |   |   |   | 0 |   |
| 5 |   |   |   |   | 0 |



### Spanning Tree:

A spanning tree of a graph  $G=(V,E)$  is its sub-graph which:

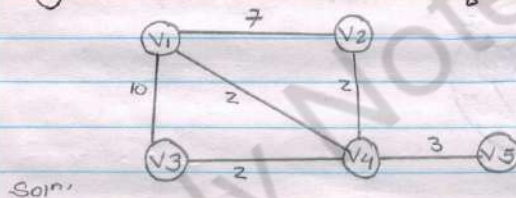
- ① is a tree
- ② Contains all vertices of  $G$ .

### Minimum spanning Tree (MST):

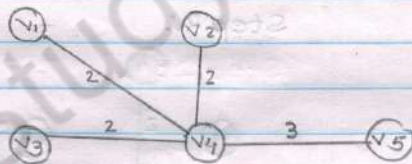
A minimum spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected components.

A minimum spanning tree of a graph is a tree which contains all vertices of the graph and the cost (weight) of the tree is smallest among all the trees.

Eg: Construct the MST of the following graph.



Soln:



### Kruskal's algorithm for MST

There are several algorithms to solve the minimum spanning tree of graph. One of such algorithm is Kruskal's algorithm.

The basic idea behind this algorithm is:

- It selects initially  $n$  nodes as  $n$  trees i.e. forest with  $n$  trees.
- It combines two trees by connecting them by a lowest cost edges that does not form cycle.



Algorithm:- Kruskal

Step 1:  $T = n$  nodes.

Step 2: while  $T$  contains fewer than  $n-1$  nodes and

$E \neq \emptyset$

do

Step 3: choose an edge  $(v, w)$  from  $E$  of lowest cost.

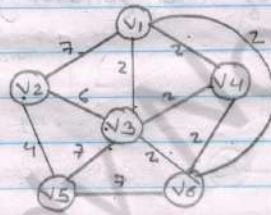
Step 4: Delete  $(v, w)$  from  $E$ .

Step 5: if adding edge  $(v, w)$  to  $T$  does not form cycle  
then add edge  $(v, w)$  to  $T$ .

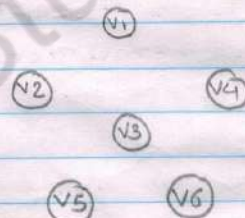
else

discard  $(v, w)$ .

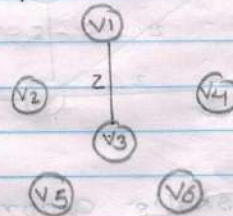
Eg: Construct the MST from the following graph.



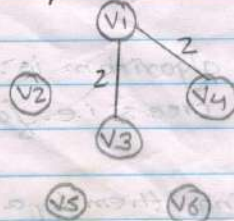
Step 1:



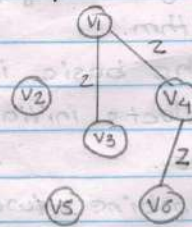
Step 2:



Step 3:

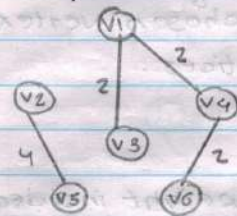


Step 4:

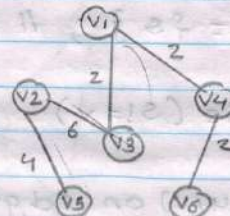




Step 5:



Step 6:



minimum weight =  $2+2+2+6+4 = 16$

| edge         | weight | selected edge |
|--------------|--------|---------------|
| $(v_1, v_2)$ | 7      |               |
| $(v_1, v_3)$ | 2      | ✓             |
| $(v_1, v_4)$ | 2      | ✓             |
| $(v_2, v_3)$ | 6      | ✓             |
| $(v_2, v_5)$ | 4      | ✓             |
| $(v_3, v_4)$ | 2      |               |
| $(v_3, v_6)$ | 2      |               |
| $(v_4, v_6)$ | 2      | ✓             |
| $(v_5, v_3)$ | 7      |               |
| $(v_5, v_6)$ | 7      |               |
| $(v_1, v_6)$ | 2      |               |

### Prim's Algorithm

This is another algorithm for finding MST. The idea behind this algorithm is just take any arbitrary vertex and choose the edge with minimum weight incident on the chosen vertex. Add the vertices as are not added to the list. Remember that the cycle must be avoided while adding the vertices.

Algorithm:

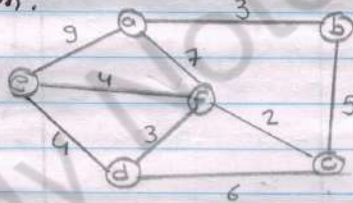
Primm st ( $G$ )

{



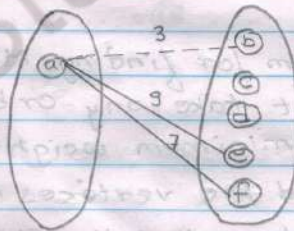
$T = \emptyset$  //  $T$  is a set of edges of MST  
 $S = \{s\}$ ; //  $s$  is randomly chosen vertex and  $S$  is a set of vertices.  
 while ( $S \neq V$ )  
 {  
    $e(u,v)$  on edge of minimum weight incident to vertices in  $T$  and forming simple circuit in  $T$  if added to  $T$  i.e.  $u \in S$  and  $v \in V - S$   
    $T = T \cup \{e(u,v)\}$ ;  
    $S = S \cup \{v\}$ ;  
 }  
 }

Q. Find the MST from following graph using Prim's algorithm.

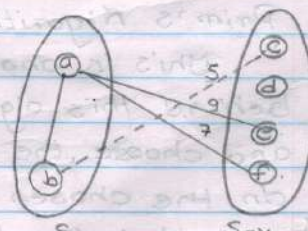


Soln,

Step 1:

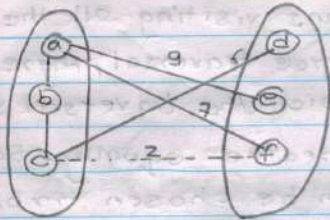


Step 2:

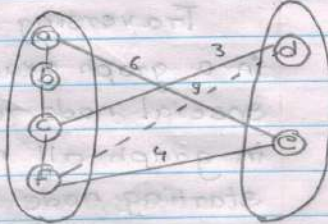




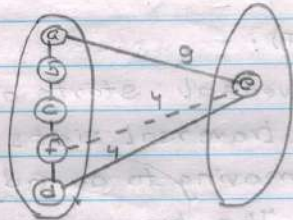
Step 3:



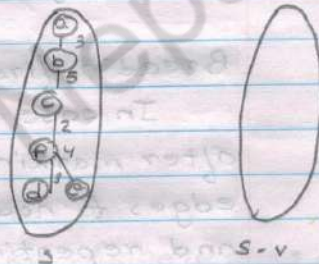
Step 4:



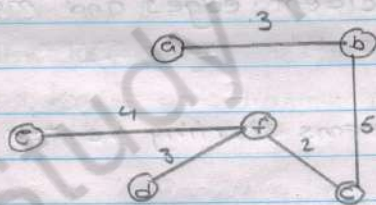
Step 5:



Step 6:



Thus, final MST given by Prim's algorithm is:



The minimum cost is  $3 + 5 + 2 + 3 + 4 = 17$



## Graph Traversal

Traversing a graph means visiting all the vertices in a graph exactly once. In tree traversal, there is a special node root, from which the traversal starts, but in graph all nodes are treated equally. So, the starting node in graph can be chosen arbitrarily.

Two common approaches for traversal are:

- \* Breadth first traversal
- \* Depth first traversal

### Breadth first traversal (BFT):

In case of BFT, the traversal starts at a node  $v$ , after marking the node the traversal visits all incident edges to node  $v$  and then moving to an adjacent node and repeating the process. The traversal continues until all unmarked nodes in the graph have been visited. A queue is maintained in the technique to maintain the list of incident edges and marked nodes.

### Algorithm:

BFT ( $G, s$ )      //  $s$  means starting vertex

{

$T = \{s\}$ ;      // edge list

$Q = \emptyset$ ;

    Enqueue ( $Q, s$ );

    while ( $Q \neq \emptyset$ )

    {

$v = \text{Dequeue}(Q)$ ;

        for each neighbour  $w$  to  $v$

        if ( $w \notin Q$  & &  $w \notin T$ )

        {

            Enqueue ( $Q, w$ );

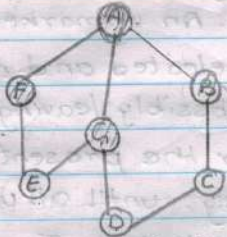
$T = T \cup \{w\}$ ;

        }



Example:

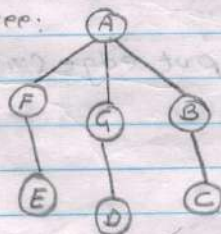
Tracing of BFS algorithm:



Suppose that Initial vertex is A

| Queue(Q) | Dequeue item (v) | Set of vertices(L)  | Adjacent vertex(w) | Edge list(T)           |
|----------|------------------|---------------------|--------------------|------------------------|
| A        | ---              | ---                 | ---                | ---                    |
| ---      | A                | A                   | B                  | AB                     |
| B        | ---              | A, B                | F                  | AB, AF                 |
| BF       | ---              | A, B, F             | G                  | AB, AF, AG             |
| BFG      | B                | A, B, F, G          | C                  | AB, AF, AG, BC         |
| BFGC     | F                | A, B, F, G, C       | E                  | AB, AF, AG, BC, FE     |
| BFGCE    | G                | A, B, F, G, C, E    | D                  | AB, AF, AG, BC, FE, GD |
| CEDE     | C                | A, B, F, G, C, E, D | ---                | AB, AF, AG, BC, FE, GD |
| ED       | E                | A, B, F, G, C, E, D | ---                | AB, AF, AG, BC, FE, GD |
| D        | D                | A, B, F, G, C, E, D | ---                | AB, AF, AG, BC, FE, GD |
| ---      | ---              | A, B, F, G, C, E, D | ---                | AB, AF, AG, BC, FE, GD |

Resulting BFS tree:



(L = A, B, F, G, C, E, D and the resulting BFS

spanning tree returned in the set

$T = \{AB, AF, AG, BC, FE, GD\}$



### Depth first traversal (DFT)

A depth first traversal of an graph can be used to perform a traversal of general graph. The technique picks up a node and marks it. An unmarked adjacent node to previous node is then selected and marked, becomes the new start node, possibly leaving the previous node with unexplored edges for the present. The traversal continued recursively, until all unmarked nodes of the current path are visited. The process is continued for all the path of the graph. If this cannot be done, move back to another vertex and repeat the process. The whole process is continued until all the vertices are visited. This method is also called backtracking. A Stack data structure is maintained in this traversing, to maintain the list of incident edges and visited vertices.

#### Algorithm:

DFT ( $G, s$ ) //  $s$  is starting vertex.

{

$T = \{s\}$  ;

Traverse ( $s$ ) ;

}

Traverse ( $v$ )

{

for each  $w$  adjacent to  $v$  and not yet in  $T$

{

$T = T \cup \{w\}$  ; // put edge ( $v, w$ )

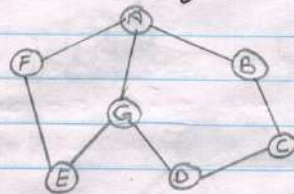
Traverse ( $w$ ) ;

}

}



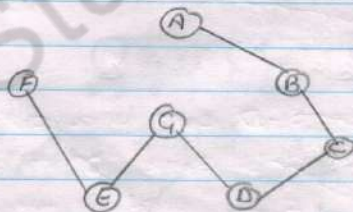
Tracing of DFT  
Consider the following graph



Starting vertex = A

| Set of vertices (L) | Stack (S) | Popped element of stack (v) | Adjacent vertex | edge list (T)          |
|---------------------|-----------|-----------------------------|-----------------|------------------------|
| A                   | A         | ---                         | ---             | ---                    |
| ---                 | ---       | A                           | B               | AB                     |
| A, B                | B         | B                           | C               | AB, BC                 |
| A, B, C             | C         | C                           | D               | AB, BC, CD             |
| A, B, C, D          | D         | D                           | G               | AB, BC, CD, DG         |
| A, B, C, D, G       | G         | G                           | E               | AB, BC, CD, DG, GE     |
| A, B, C, D, G, E    | E         | E                           | F               | AB, BC, CD, DG, GE, EF |
| A, B, C, D, G, E, F | F         | F                           | ---             | AB, BC, CD, DG, GE, EF |

$L = A, B, C, D, G, E, F$  and the resulting DFT spanning tree returned in the set  $T = \{AB, BC, CD, DG, GE, EF\}$



\* Connectivity in Graph



## Network Flow Problems

### Flow Network

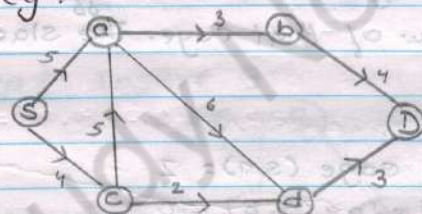
A flow network or transport network is a connected directed graph  $G = (V, E)$  that does not contain any loops such that,

- There are exactly two distinguished vertices  $S$  and  $D$ , called source and destination (sink) of graph  $G$  respectively.

- There is a non-negative real values function  $k$  defined on edge  $E$  called the capacity function of  $G$ .

Then,  $(G, k)$  is called transport network and function  $k$  is called capacity function of  $G$ . The vertices distinct from  $S$  and  $D$  are called intermediate vertices. If  $e \in E$  then  $k(e)$  is called capacity of  $e$ .

For eg:

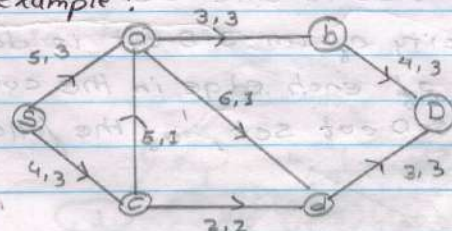


Flow: let  $(G, k)$  be a transport network, then a flow in  $G$  is a non-negative function  $f$  defined on edge  $E$  such that,

$$0 \leq f(e) \leq k(e) \text{ for each edge } e \in E$$

where  $f(e)$  = flow to the edge  $e$ .

For example:





From the above figure, the flow-in of vertex 'a' is  
 $F(s, a) + F(c, a) = 3 + 1 = 4$

The flow-out of vertex a is  
 $F(a, b) + F(a, d) = 3 + 1 = 4$  and so on.

Saturated and unsaturated edges

The edges for which the flow and capacity are equal called saturated edges. Otherwise they are called unsaturated edges.

From the above graph,

edges (a, b), (c, d) and (d, D) are saturated and remaining others are called unsaturated edges.

Slack of edge:

The slack of edge 'e' is the difference of capacity and maximum flow of that edge. The slack of saturated edge is 0.

For example,

The slack of edge (s, a) = 2

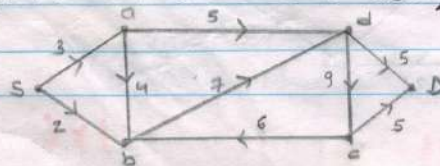
Slack of edge (a, b) = 0

S-D cut:

In a flow network, an S-D cut is a cut, that requires the source 's' and sink 'D' to be in different subsets and it consist of edges going from the source side to sink side (destination)

The capacity of an S-D cut is defined by the sum of capacity of each edge in the cut set.

For eg: Find the S-D cut set, of the following graph.



147



Soln.

| $x$           | $x'$                | Capacity( $x, x'$ ) |
|---------------|---------------------|---------------------|
| $\{s\}$       | $\{a, b, c, d, 0\}$ | $2+3=5$             |
| $\{s, a\}$    | $\{b, c, d, 0\}$    | $5+4+2=11$          |
| $\{s, b\}$    | $\{a, c, d, 0\}$    | $3+7=10$            |
| $\{s, c\}$    | $\{a, b, d, 0\}$    | $6+3+2+5=16$        |
| $\{s, d\}$    | $\{a, b, c, 0\}$    | $3+2+5+9=19$        |
| $\{s, a, b\}$ | $\{c, d, 0\}$       | $5+7=12$            |
| $\{s, a, c\}$ | $\{b, d, 0\}$       | $5+5+2+5=17$        |
| $\{s, a, d\}$ | $\{b, c, 0\}$       | $9+5+4+2=20$        |
| $\{s, b, c\}$ | $\{a, d, 0\}$       | $3+7+5+5=20$        |
| $\{s, b, d\}$ | $\{a, c, 0\}$       | $3+7+9+5=24$        |



### Flow augmenting path

A flow-augmenting path from source  $s$  to sink  $D$  is a sequence of edges from  $s$  to  $D$  such that, for each edge in this path, the flow  $F(e) < k(e)$  on forward edges and  $F(e) > 0$  on backward edges.  
slack add (+) slack minus (-)

It means that such a path is not optimally used yet, and it can transfer more units than it is currently transferring. If the flow for at least one edge of the path reaches its capacity, then obviously the flow cannot be augmented.

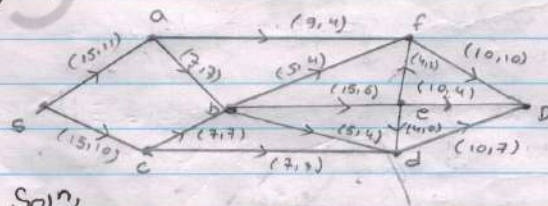
### Max-flow min cut:

In any network, the value of the max flow is equal to the value of min cut.

This theorem asserts the following:

- i) The existence of a minimal cut  $(x, x')$
- ii) The existence of maximal flow  $F$ .
- iii) The equality of  $F$  and  $c(x, x')$  for any maximal flow  $F$  and any minimal cut  $(x, x')$

Q. Find the maximum flow for the following Network graph.



Soln,

At first we see for edges that are unsaturated and has forward paths only.

Here, unsaturated forward path

$$P_1 = s-c-d-D$$

Now,



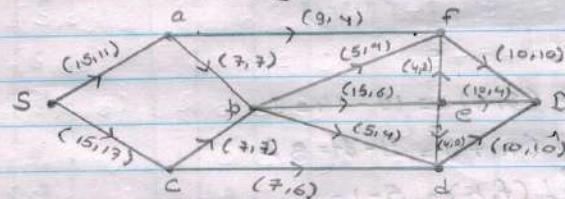
$$\text{Slack of } (s, c) = 15 - 10 = 5$$

$$\text{Slack of } (c, d) = 7 - 3 = 4$$

$$\text{Slack of } (d, D) = 10 - 7 = 3$$

Minimum slack value is 3, so, add 3 to each edges,

So, above graph becomes,



Now, look for edges which are unsaturated and has backward edges.

$$P_2 = S-a-f-e-D$$

$$P_3 = S-c-d-e-D$$

$$P_4 = S-c-d-b-e-D$$

$$P_5 = S-a-f-b-e-D$$

For  $P_5$ :  $\text{Slack of } (s, a) = 15 - 11 = 4$

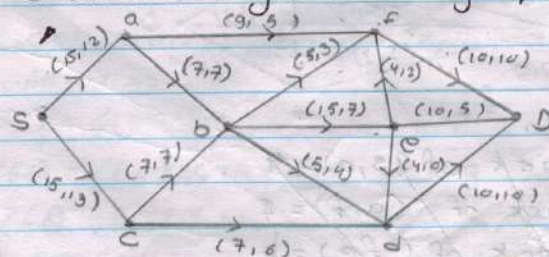
$$\text{Slack of } (a, f) = 9 - 4 = 5$$

$$\text{Slack of } (f, b) = 5 - 4 = 1$$

$$\text{Slack of } (b, e) = 15 - 6 = 9$$

$$\text{Slack of } (e, D) = 10 - 4 = 6$$

Now, add 1 to forward edges and subtract to backward edges. The graph becomes,



Again,  $\text{Slack of } (s, a) = 3$

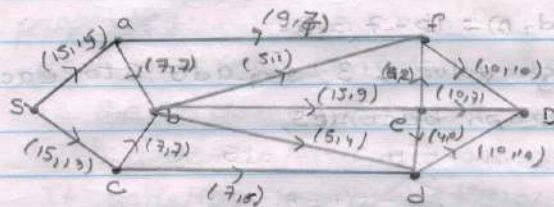
$$\text{Slack of } (a, f) = 4$$

$$\text{Slack of } (f, b) = 2$$



Slack of  $(b,e) = 8$

Slack of  $(e,d) = 5$



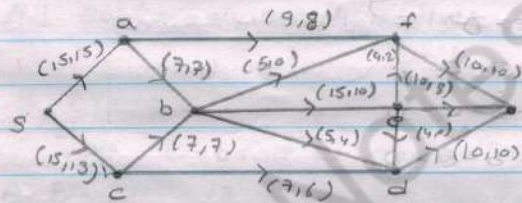
Again, Slack of  $(s,a) = 15 - 14 = 1$

Slack of  $(a,f) = 9 - 7 = 2$

Slack of  $(f,b) = 5 - 1 = 4$

Slack of  $(b,e) = 15 - 9 = 6$

Slack of  $(e,d) = 10 - 7 = 3$



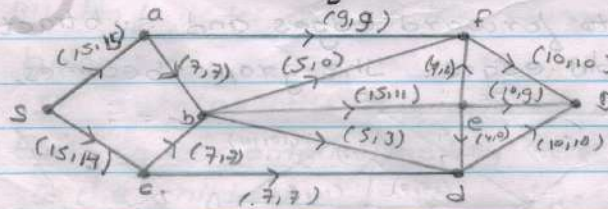
For  $P_4$ : Slack of  $(s,c) = 15 - 13 = 2$

Slack of  $(c,d) = 7 - 6 = 1$

Slack of  $(d,b) = 5 - 4 = 1$

Slack of  $(b,e) = 15 - 10 = 5$

Slack of  $(e,d) = 10 - 8 = 2$



For  $P_2$ : Slack of  $(s,a) = 4$

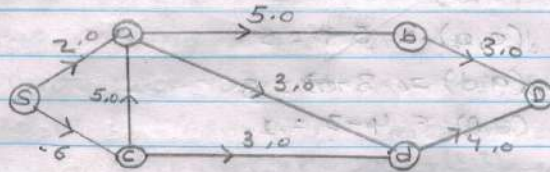
Slack of  $(a,f) = 3$

Slack of  $(f,e) = 4$

Slack of  $(e,d) = 10$



Q. Find max flow of the following network flow graph



Soln:

At first we see for edges that are unsaturated and has forward paths only.

Here, unsaturated forward path

$$P_1 = S-a-b-D$$

$$P_2 = S-c-d-D$$

$$P_3 = S-c-a-d-D$$

$$P_4 = S-a-d-D$$

$$P_5 = S-c-a-b-D$$

Now, for  $P_1$ :

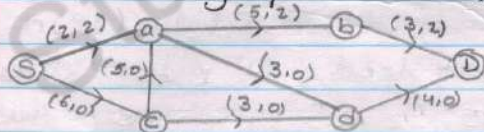
$$\text{Slack of } (S, a) = 5 - 2 = 3$$

$$\text{Slack of } (a, b) = 5 - 5 = 0$$

$$\text{Slack of } (b, D) = 4 - 3 = 1$$

Minimum slack value is 1. So add 1 to each edges.

So, above graph becomes,

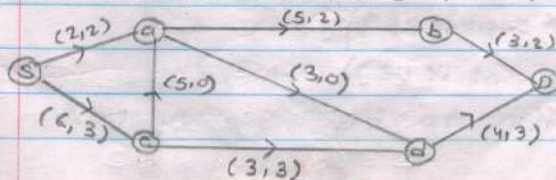


For  $P_2$ :

$$\text{Slack of } (S, c) = 6 - 6 = 0$$

$$\text{Slack of } (c, d) = 4 - 3 = 1$$

$$\text{Slack of } (d, D) = 4 - 4 = 0$$





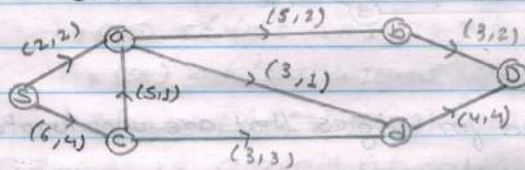
For P3:

$$\text{Slack of } (s, c) = 6 - 3 = 3$$

$$\text{Slack of } (c, a) = 5 - 0 = 5$$

$$\text{Slack of } (a, d) = 3 - 0 = 3$$

$$\text{Slack of } (d, D) = 4 - 3 = 1$$



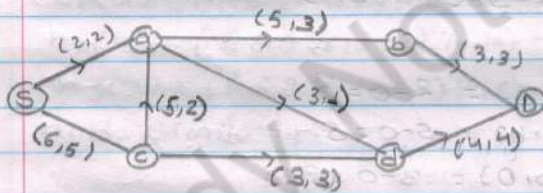
For P6:

$$\text{Slack of } (s, c) = 6 - 4 = 2$$

$$\text{Slack of } (c, a) = 5 - 1 = 4$$

$$\text{Slack of } (a, b) = 5 - 2 = 3$$

$$\text{Slack of } (b, d) = 3 - 2 = 1$$





## Sorting

Sorting is the process of ordering elements in an array in specific order i.e. either ascending or descending on the basis of value, chronological ordering of records.

There are two types of sorting

- ① Internal sorting: Internal sorting means we are arranging the numbers within the array only which is in Computer primary memory.
- ② External sorting: External sorting is the sorting of numbers from the external file by reading it from Secondary memory.

Let  $P$  be a list of  $n$  elements  $P_1, P_2, \dots, P_n$  in memory. Sorting  $P$  means arranging the contents of  $P$  in either increasing or decreasing order i.e.,  
 $P_1 \leq P_2 \leq P_3 \leq \dots \leq P_n$  and  $(x-n)$

### 1) Bubble sort:

In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one then the position of the elements are interchanged, otherwise, it is not changed. Then next element is compared with its adjacent element and same process is repeated for all the elements in the array. During the pass, the second largest element occupied the second last position. The same process is repeated until no more elements are left for comparison. Finally the array is sorted.

### Algorithm:

This algorithm sorts the array 'a' with 'n' elements



- ① set  $i = 0$
- ② Repeat Step 3 to 7 until  $i < n$ .
- ③ Set  $j = 0$ ;
- ④ Repeat Step 5 to 6 until  $j < n - i - 1$
- ⑤ If  $a[j] > a[j+1]$   
     set  $temp = a[j]$   
     set  $a[j] = a[j+1]$   
     set  $a[j+1] = temp$
- ⑥  $j = j + 1$
- ⑦  $i = i + 1$
- ⑧ exit

### Complexity analysis of Bubble Sort

In bubble sort, the first pass requires  $(n-1)$  comparisons to fix the highest element to its location, the second pass requires  $(n-2)$ , ...,  $k$ th pass requires  $(n-k)$  and the last pass requires only one comparison to be fixed at its proper position.

Therefore, the total comparisons are:

$$f(n) = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

$$= \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Hence, time complexity of bubble sort is  $O(n^2)$ .

Example:

Trace the bubble sort algorithm for following data value:

$A[] = 32, 51, 27, 85, 66, 23, 13, 57$

Soln: The given array is,

$A[] = 32, 51, 27, 85, 66, 23, 13, 57$

Now applying bubble sort algorithm,



Pass 1: Compare  $A[0]$  with  $A[1]$ , since  $32 < 51$ , do not interchange.

- ~~Pass 1~~ Compare  $A[1]$  with  $A[2]$ , since  $51 > 27$ , interchange 51 and 27.

32, 27, 51, 85, 66, 23, 13, 57

- Compare  $A[2]$  with  $A[3]$ , since  $51 < 85$ , do not interchange.

- Compare  $A[3]$  with  $A[4]$ , since  $85 > 66$ , interchange 85 and 66

32, 27, 51, 66, 85, 23, 13, 57

- Compare  $A[4]$  with  $A[5]$ , since  $85 > 23$ , interchange 85 and 23.

32, 27, 51, 66, 23, 85, 13, 57

- Compare  $A[5]$  with  $A[6]$ , since,  $85 > 13$ , interchange 85 and 13.

32, 27, 51, 66, 23, 13, 85, 57

- Compare  $A[6]$  with  $A[7]$ , since  $85 > 57$ , interchange 85 and 57.

32, 27, 51, 66, 23, 13, 57, 85

At the end of Pass 1, the largest element 85 is in its position

Pass 2: Compare  $A[0]$  with  $A[1]$ , since  $32 > 27$ , interchange 32 and 27.

27, 32, 51, 66, 23, 13, 57, 85

- Compare  $A[1]$  with  $A[2]$ , since  $32 < 51$ , no interchange.

- Compare  $A[2]$  with  $A[3]$ , since  $51 < 66$ , no interchange.

- Compare  $A[3]$  with  $A[4]$ , since  $66 > 23$ , interchange.

27, 32, 51, 23, 66, 13, 57, 85

- Compare  $A[4]$  with  $A[5]$ , since  $66 > 13$ , interchange.

27, 32, 51, 23, 13, 66, 57, 85

- Compare  $A[5]$  with  $A[6]$ , since  $66 > 57$ , interchange.

27, 32, 51, 23, 13, 57, 66, 85

At the end of Pass 2, the two largest element is in its proper position.



Pass 8:

- Compare  $A[0]$  with  $A[1]$ , since  $27 < 32$ , no interchange.
- Compare  $A[1]$  with  $A[2]$ , since  $32 < 51$ , no interchange.
- Compare  $A[2]$  with  $A[3]$ , since  $51 > 23$ , interchange.

27, 32, 23, 51, 13, 57, 66, 85

- Compare  $A[3]$  with  $A[4]$ , since  $51 > 13$ , interchange.

27, 32, 23, 13, 51, 57, 66, 85

- Compare  $A[4]$  with  $A[5]$ , since  $51 < 57$ , no interchange.

Pass 4:

- Compare  $A[0]$  with  $A[1]$ , since  $27 < 32$ , no interchange.
- Compare  $A[1]$  with  $A[2]$ , since  $32 > 23$ , interchange,

27, 23, 32, 13, 51, 57, 66, 85

- Compare  $A[2]$  with  $A[3]$ , since  $32 > 13$ , interchange

27, 23, 13, 32, 51, 57, 66, 85

- Compare  $A[3]$  with  $A[4]$ , since  $32 < 51$ , no interchange.

Pass 5:

- Compare  $A[0]$  with  $A[1]$ , since  $27 > 23$ , interchange.

23, 27, 13, 32, 51, 57, 66, 85

- Compare  $A[1]$  with  $A[2]$ , since  $27 > 13$ , interchange.

23, 13, 27, 32, 51, 57, 66, 85

- Compare  $A[2]$  with  $A[3]$ , since  $27 < 32$ , no interchange.

Pass 6:

- Compare  $A[0]$  with  $A[1]$ , since  $23 > 13$ , interchange.

13, 23, 27, 32, 51, 57, 66, 85

- Compare  $A[1]$  with  $A[2]$ , since  $23 < 27$ , no interchange.

The final sorted data are:

13, 23, 27, 32, 51, 57, 66, 85



## Insertion sort;

An insertion sort is one that sorts a set of elements by inserting into an existing sorted file. Suppose an array  $A$  with  $N$  elements  $A[0], A[1], A[2], \dots, A[N-1]$ . The insertion sort algorithm scans  $A$  from  $A[0]$  to  $A[N-1]$ , inserting each element into its proper position in the previously sorted sub array.

For example:

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 position in  $A[0], A[1]$  i.e. before  $A[0]$ , between  $A[0]$  and  $A[1]$  or after  $A[1]$  so  $A[0], A[1], A[2]$  is sorted.

⋮

Pass  $N$ :  $A[N-1]$  is inserted into its proper position in  $A[0], A[1], A[2], \dots, A[N-2]$ , so that whole array is sorted.

Algorithm:-

Insertion ( $A, n$ )

{

For  $i=1$  to  $n-1$

{

key =  $a[i]$ ;

$j = i-1$ ;

while  $((j > 0) \text{ and } (a[j] > \text{key}))$

{

$A[j+1] = A[j]$

$j = j-1$

}

$A[j+1] = \text{key}$ ;

}



Example:- Sort the following data value using insertion

Sort :

34, 8, 64, 51, 32, 21

Soln:

$A[6] = \{34, 8, 64, 51, 32, 21\}$

Pass 1:  $i = 1$   $n = 6$

$key = a[1] = 8$

$j = 1 - 1 = 0$

while  $((0 > 0) \&\& (a[0] > key \text{ i.e. } 34 > 8))$

$A[1] = A[0] = 34$

$j = -1$

$A[-1+1] = key$

$A[0] = 8$

$A[6] = \{8, 34, 64, 51, 32, 21\}$

Pass 2:  $i = 2$

$key = a[2] = 64$

$j = 2 - 1 = 1$

while  $((1 > 0) \&\& (a[1] > key \text{ i.e. } 34 > 64))$

$A[1+1] = key$

$A[2] = key \text{ i.e. } 64$

$A[6] = \{8, 34, 64, 51, 32, 21\}$

Pass 3:  $i = 3$

$key = a[3] = 51$

$j = 3 - 1 = 2$

while  $((2 > 0) \&\& (a[2] > key \text{ i.e. } 64 > 51))$

$A[3] = A[2] = 64$

$j = 2 - 1 = 1$

$A[1+1] = key$

$A[2] = 51$

$A[6] = \{8, 34, 51, 64, 32, 21\}$



```
while (i > 0 && (a[i] > key i.e. 34 > 51))  
    A[i+1] = key  
    A[2] = 51  
A[6] = {8, 34, 51, 64, 32, 21}
```

```
Pass 4: i = 4  
key = a[4] = 32  
j = 4 - 1 = 3  
while (j > 0 && a[j] > key i.e. 64 > 32)  
    A[j+1] = A[j]  
    j = j - 1 = 2  
A[2+1] = key  
A[3] = 32
```

```
while (j > 0 && a[j] > key i.e. 51 > 32)  
    A[j+1] = A[j]  
    j = j - 1 = 1  
A[1+1] = key  
A[2] = 32
```

```
while (j > 0 && a[j] > key i.e. 34 > 32)  
    A[j+1] = A[j]  
    j = j - 1 = 0  
A[0+1] = key  
A[1] = 32
```

```
while (0 > 0 && a[0] > key i.e. 8 > 32)  
    A[0+1] = A[0]  
    A[1] = key  
8, 32, 34, 51, 64, 21
```



## Selection sort

Consider an array  $A$  with  $n$  elements  $A[0]$  to  $A[n-1]$ . The selection sort algorithm for sorting  $A[]$  works as follows: First find the smallest elements in the array and put it in the first position. Then find the second smallest element and put it in the second position and so on. i.e.

In first pass: find the location  $L$  of the smallest element in array of  $n$  elements and then interchange  $A[L]$  with  $A[0]$ . So,  $A[0]$  is sorted.

In second pass: find the location of the smallest element in the array of  $n-1$  elements and interchange it with  $A[1]$ . So,  $A[0], A[1]$  is sorted. i.e.  $A[0] \leq A[1]$  and so on.

Finally, in pass  $n-1$ : find the location  $L$  of the smallest element in the array of  $A[n-2], A[n-1]$  then interchange  $A[L]$  with  $A[n-2]$ . Thus  $A[0], A[1], \dots, A[n-1]$  is sorted.

## Algorithm for selection sort

Selectionsort ( $A, n$ )

Step 1: [loop for pass] Repeat Step 2 to 4 for pass = 0 to  $n-1$

Step 2: [initialize]

min\_index = pass;

min =  $A[\text{pass}]$

Step 3: for  $i = \text{pass} + 1$  to  $n-1$  do

if  $A[i] < A[\text{min\_index}]$  then

min\_index =  $i$ ;

Step 4: [Exchange]

if ( $\text{min\_index} \neq \text{pass}$ ) then

$A[\text{pass}] = A[\text{min\_index}]$

$A[\text{min\_index}] = \text{min};$

Step 5: Exit



Method:

```

SelectionSort (int A[], int n)
{
    int pass, min_index, i, min;
    for (pass = 0; pass < n-1; pass++)
    {
        min_index = pass;
        min = A[pass];
        for (i = pass+1; i < n; i++)
        {
            if (A[i] < A[min_index])
                min_index = i;
        }
        if (min_index != pass)
        {
            A[pass] = A[min_index];
            A[min_index] = min;
        }
    }
}

```

Example: Trace the Selection sort for

$A[] = \{40, 66, 93, 8, 12, 44, 6, 88, 60\}$

soln:  $n = 9$

Pass = 0

$\text{min\_index} = 0$

$\text{min} = A[0] \text{ i.e. } 40$

$i = 0 + 1 = 1$

if ( $A[1] < A[0]$ ) i.e.  $66 < 40$

$i = 2$  ~~40 < 66~~

if ( $A[2] < A[0]$ ) i.e.  $93 < 40$

$i = 3$

if ( $A[3] < A[0]$ ) i.e.  $8 < 40$

$\text{min\_index} = 3$

112



$i=4$

if ( $A[4] < A[3]$ ) i.e.  $12 < 8$

$i=5$

if ( $A[5] < A[3]$ ) i.e.  $44 < 8$

$i=6$

if ( $A[6] < A[3]$ ) i.e.  $6 < 8$

min\_index = 6

$i=7$

if ( $A[7] < A[6]$ ) i.e.  $88 < 6$

$i=8$

if ( $A[8] < A[6]$ ) i.e.  $60 < 6$

if ( $6 \neq 0$ )

{

$A[0] = A[6]$  i.e. 6

$A[6] = 40$ ;

}

$A[] = \{ 6, 66, 93, 8, 12, 44, 40, 88, 60 \}$

Pass = 1

min\_index = 1

min =  $A[1]$  i.e. 66

$i = 1 + 1 = 2$

if ( $A[2] < A[1]$ ) i.e.  $93 < 66$

$i=3$

if ( $A[3] < A[1]$ ) i.e.  $8 < 66$

min\_index = 3

$i=4$

if ( $A[4] < A[3]$ ) i.e.  $12 < 8$

$i=5$

if ( $A[5] < A[3]$ ) i.e.  $44 < 8$

$i=6$

if ( $A[6] < A[3]$ ) i.e.  $40 < 8$



Smaller or equal to the pivot element are kept in one sublist which are placed before the pivot element, and all the elements greater than or equal to pivot elements are kept in another sublists which is placed after the pivot element. So after choosing the key element, partitioning is performed.

Let  $x$  be an array, and  $n$  be the number of elements in the array to be sorted. Choose an element  $k$  from a specific position within an array (for example,  $k$  can be chosen as the first element so that  $k = x[0]$ ). Suppose that the elements in  $x$  are partitioned so that  $k$  is placed into position  $j$  and the following conditions hold:

- ① Each of the elements in position 0 through  $j-1$  is less than or equal to  $k$ .
- ② Each of the elements in position  $j+1$  through  $n-1$  is greater than or equal to  $k$ .

It is also called partition exchange sort.

Example:

Trace the quick sort algorithm for the following data value.

25, 57, 48, 37, 12, 92, 86, 33

General idea for partitioning:

- Take first element as pivot.
- Set left pointer at beginning and right pointer at end
- Increment (move right) left pointer until the element at pointer location is not greater than pivot or end of list.
- Decrease (move left) the right pointer until the element at pointer location is not less than pivot.



32 32

→ If  $L < R$ , interchange elements at  $L$  and  $R$  position.  
 → If  $L > R$ , interchange pivot with  $a[R]$ .  
 where,  $L \rightarrow$  left pointer  
 $R \rightarrow$  Right pointer  
 $a[] \rightarrow$  array.

\* Soln, pivot = 25  
 25 57 48 37 12 92 86 33  
 $L$   $R$   
 25 57 48 37 12 92 86 33  
 $L$   $R$   
 $a[L] \leftrightarrow a[R]$   
 25 12 48 37 57 92 86 33  
 $L$   $R$   
 25 12 48 37 57 92 86 33  
 $R$   $L$

Pivot  $\leftrightarrow a[R]$   
 12 25 48 37 57 92 86 33  
 low array high array (table)

Pivot = 48  
 12 25 48 37 57 92 86 33  
 $L$   $R$   
 12 25 48 37 57 92 86 33  
 $L$   $R$

12 25 48 37 33 92 86 57  
 $L$   $R$   
 12 25 48 37 33 92 86 57  
 $R$   $L$

12 25 33 37 48 92 86 57  
 $L$   $R$   
 Pivot = 92  
 12 25 33 37 48 92 86 57  
 $L$   $R$

1 1 1



12 25 33 37 48 92 86 57  
LR

Pivot = 86

12 25 33 37 48 92 86 57  
LR

Pivot = 57

12 25 33 37 48 92 86 57  
R L

12 25 33 37 48 57 86 92

9. Trace:

21 17 18 28 11 9 34 76

Soln:

pivot = 21

21 17 18 28 11 9 34 76

L

21 17 18 28 11 9 34 76

L

R

21 17 18 9 11 28 34 76

L

R

21 17 18 9 11 28 34 76

R

L

11 17 18 9 21 28 34 76

Pivot = 11

11 17 18 9 21 28 34 76

L

R

11 17 18 9 21 28 34 76

L

R

11 9 18 17 21 28 34 76

L

R

11 9 18 17 21 28 34 76

R

L

162



9 11 18 17 21 28 34 76

Pivot = 18

9 11 18 17 21 28 34 76  
L R

9 11 17 18 21 28 34 76

Pivot = 28

9 11 17 18 21 28 34 76  
L R

9 11 17 18 21 28 34 76  
L R

Pivot = 34

9 11 17 18 21 28 34 76  
L R

9 11 17 18 21 28 34 76  
R L

The sorted data: 9 11 17 18 21 28 34 76

Function:

```
void quicksort(int a[], int l, int r)
{
```

```
    if (l < r)
```

```
    {
```

```
        int j = partition(a, l, r);
```

```
        quicksort(a, l, j-1);
```

```
        quicksort(a, j+1, r);
```

```
    }
```

```
int partition(int a[], int l, int r)
```

```
{
```



```

int temp, p = a[i], i = 1, j = r+1;
do {
    {
        do {
            i++;
        }
        while (a[i] < p && i < r);
        do {
            j--;
        }
        while (a[j] > p);
        if (i < j) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    while (i < j);
    a[i] = a[j];
    a[j] = p;
    return j;
}

```

Trace:

40 6 12 8 93 44 66 88 60



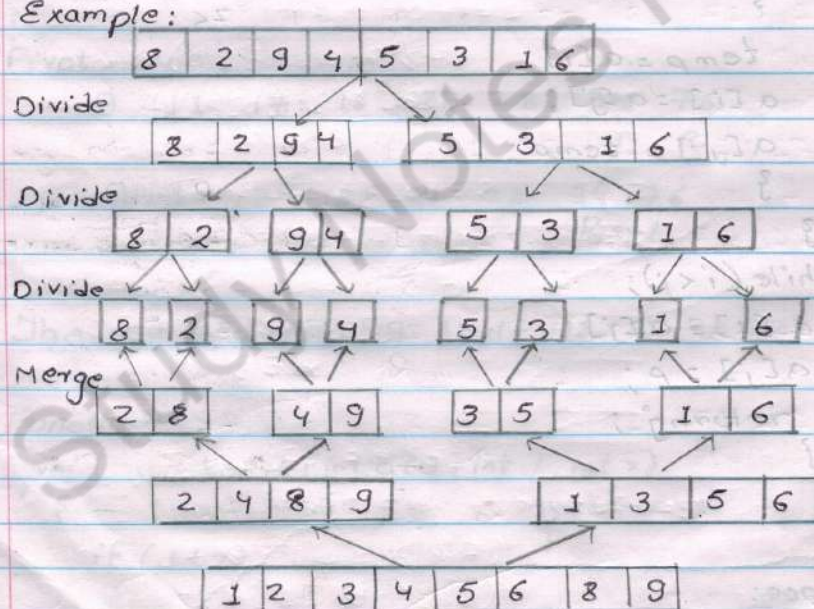
Merge sort - Complexity Analysis =  $O(n \log n)$

Merge sort is an efficient sorting algorithm which involves merging two or more sorted files into a third sorted file. Merging is the process of combining two or more sorted files into a third sorted file. The merge sort algorithm is based on divide and conquer method.

The process of merge sort can be formalised into three basic operations:

- ① Divide the array into two sub arrays.
- ② Recursively sort the two sub arrays.
- ③ Merge the newly sorted sub arrays.

Example:



Algorithm:

```
mergesort (A[], Temp[], left, right)
{
    if (left < right)
    {
```



```

mid = (left + right) / 2
mergesort(A, Temp, left, mid)
mergesort(A, Temp, mid+1, right)
mergesort(A, Temp, left, mid, right)
}
merge (A[], Temp[], left, mid, right)
{
    int i, j, k, L, target;
    i = left;
    j = mid + 1;
    target = left;
    while (i <= mid && j <= right)
    {
        if (A[i] <= A[j])
            Temp[target] = A[i];
            i++;
            target++;
        else
            Temp[target] = A[j];
            j++;
            target++;
    }
    if (i > mid)
        for (k = left to target-1)
            A[k] = Temp[k];
    if (j > right)
        k = mid
        l = right
        while (k > i)
            A[l--] = A[k--];
    for (k = left to target-1)
        A[k] = Temp[k];
}

```



$A[] = \{8, 2, 9, 4, 5, 3, 1, 6\}$

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 4 | 8 | 9 | 1 | 3 | 5 | 6 |

left = 0, right = 7, i = 0, j = 4, target = 0, mid(0+7/2) = 3.5  
= 3

① while (0 ≤ 3 && 4 ≤ 7)

if (A[0] ≤ A[4])

2 ≤ 1 x

Temp[] = 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 |
|---|---|---|---|---|---|---|---|

else

Temp[0] = A[4] = 1

j = j + 1 = 4 + 1 = 5

target = 0 + 1 = 1

② while (0 ≤ 3 && 5 ≤ 7)

if (A[0] ≤ A[5])

2 ≤ 3

Temp[1] = A[0] = 2

i = 0 + 1 = 1, target = 1 + 1 = 2

③ while (1 ≤ 3 && 5 ≤ 7)

if (A[1] ≤ A[5])

4 ≤ 3 x

else

Temp[2] = A[5]

j = 5 + 1 = 6, target = 2 + 1 = 3

④ while (1 ≤ 3 && 6 ≤ 7)

if (A[1] ≤ A[6])

4 ≤ 5

Temp[3] = A[1] = 4

i = 1 + 1 = 2, target = 3 + 1 = 4

⑤ while (2 ≤ 3 && 6 ≤ 7)

if (A[2] ≤ A[6])

8 ≤ 5 x

172



```

else
    Temp[4] = A[6] = 5
    j = 6+1 = 7, target = 4+1 = 5
⑥ while (2 ≤ 3 & 7 ≤ 7)
    if (A[2] ≤ A[7])
        8 ≤ 6 x
    else
        Temp[5] = A[7] = 6
        j = 7+1 = 8, target = 5+1 = 6
⑦ while (2 ≤ 3 & 8 ≤ 7) x
    if (2 > 3) x
    if (8 > 7)
        k = 3, l = 7
        while (3 > 2)
            A[7] = A[3]
            A[3] = 9
            l = 6, k = 2
        while (2 > 2)
            A[6] = A[2] = 8
            l = 5, k = 1
        while (1 > 2) x
    
```

Radix sort: Complexity analysis:  $O(n \log n)$

If the radix sort is used to sort the list of decimal numbers then 10 buckets are required and if we want to arrange the names in their alphabetical order then 26 buckets are required.

In radix sort the list of elements to be sorted is broken down into several buckets and the elements are assigned to the bucket according to the position



of the element in the digit. For example, the digits are assigned first according to the least significant element, then to the next least significant element and so on.

Example : Sort the list 27, 38, 29, 31, 45, 87, 26, 25 using the radix sort.

Soln.

Pass 1 : Assign the elements to each bucket according to unit position (least significant element) of the digits.

Input file : 27, 38, 29, 31, 45, 87, 26, 25

|           |       |
|-----------|-------|
| Bucket[0] |       |
| Bucket[1] | 31    |
| Bucket[2] |       |
| Bucket[3] |       |
| Bucket[4] |       |
| Bucket[5] | 25 45 |
| Bucket[6] | 26    |
| Bucket[7] | 27 87 |
| Bucket[8] | 38    |
| Bucket[9] | 29    |

Pass 2 : Read the elements from each bucket starting from bucket 0 to bucket 9. Therefore the input to next pass is 31, 25, 45, 26, 27, 87, 38, 29

Now assign the elements to each bucket according to the ten's position (i.e. second least significant digit) of the element.

|           |             |
|-----------|-------------|
| Bucket[0] |             |
| Bucket[1] |             |
| Bucket[2] | 25 26 27 29 |



|            |       |
|------------|-------|
| Bucket [3] | 31 38 |
| Bucket [4] | 45    |
| Bucket [5] |       |
| Bucket [6] |       |
| Bucket [7] |       |
| Bucket [8] | 87    |
| Bucket [9] |       |

Pass 3 : Read the element from each bucket starting from bucket 0 to bucket 9. The list now is  
25, 26, 27, 29, 31, 38, 45, 87

The list is sorted.

Searching



## Searching

Searching is a process of finding an element within the list of elements stored in any order or randomly. Searching is divided into two categories: Sequential and binary search.

### - Sequential search :

Sequential search technique is simpler searching technique. This searching technique is equally applied to a table organized in either as array or linked list.

In this search technique the record to be searched is compared to each record in the storage table until it matched. No matter the records are arranged or not in any sorted order.

Let  $A[]$  is an array of  $n$  key values,  $A[0]$  to  $A[n-1]$ ,  $r$  is record to be search within array  $A[]$ .

The algorithm for sequential search is :

Steps:

```
int SequentialSearch (int r, A[])
{
    for (int i = 0; i < (n-1); i++)
    {
        if (A[i] == r)
        {
            return i;
        }
        else
        {
            return -1;
        }
    }
}
```

This algorithm (method) for search returns the index position of searched element in searching array (i.e.  $A[]$ ) if search is successful, otherwise it returns -1 to indicate unsuccessful.



### Efficiency of Sequential Search (Time complexity)

Base case :  $O(1)$

worst case :  $O(n)$

### - Binary Search: Time Complexity : $O(\log n)$

It is the efficient searching technique for a data item from an sorted array. This technique compares data item to be search with middle item in the ordered list. If not matched it checks either the data item to be searched is greater or smaller than middle data item in the list. If it is smaller, search is shifted to the left half otherwise shifted to right half recursively.

### Algorithm for binary search

BinSearch(A[], Beg, End, key)

[Initialize]

Beg = 0, End = n-1;

while (Beg ≤ End)

{

Set mid =  $\lfloor (Beg + End) / 2 \rfloor$

if (key == A[mid])

{

Display SearchSuccessful message

return (mid);

}

else if (key < A[mid])

Set End = mid - 1

else

Set Beg = mid + 1

}

Display Search unsuccessful message

return (-1);



```

Function:
int BinSearch(A[], Beg, End, key)
{
    int Beg = 0, int End = n-1;
    while (Beg <= End)
    {
        int mid = (Beg + End) / 2;
        if (int key == A[mid])
        {
            System.out.println("Search Successful");
            return mid;
        }
        else if (key < A[mid])
        {
            End = mid - 1;
        }
        else
        {
            Beg = mid + 1;
        }
    }
    System.out.println("Search Unsuccessful");
    return -1;
}
    
```



Example: Let us consider the input data:

9, 12, 24, 30, 36, 45, 70

Search 45 and 40 using binary search.

Soln:

① Beg = 0, End = 6, A[] = 9, 12, 24, 30, 36, 45, 70.

while (0 ≤ 6)

mid =  $\lfloor (0+6)/2 \rfloor = 3$

if (45 == A[3])

45 == 30 x

else if (45 < A[3])

45 < 30 x

else

Beg = 3+1 = 4

while (4 ≤ 6)

mid =  $\lfloor (4+6)/2 \rfloor = 5$

if (45 == A[5])

45 == 45

Search is successful

return 5

Search for 40:

② Beg = 0, End = 6, A[] = 9, 12, 24, 30, 36, 45, 70.

while (0 ≤ 6)

mid =  $\lfloor (0+6)/2 \rfloor = 3$

if (40 == A[3])

40 == 30 x

else if (40 < A[3])

40 < 30 x

else ~~(40 > A[3])~~

Beg = 3+1 = 4

[∵ 40 > A[3]]

while (4 ≤ 6)

mid =  $\lfloor (4+6)/2 \rfloor = 5$

if (40 == A[5])

40 == 45 x

else if (40 < A[5])

End = 5-1 = 4

(7)



```

- while( 4 ≤ 4)
    mid = ⌊(4+4)/2⌋ = 4
    if (40 == A[4])
        40 == 36 x
    else if (40 < 36) x
    else
        Beg = 4+1 = 5
- while( 5 ≤ 4) x
    Search is unsuccessful
    return -1.
    
```

### Hashing

Hashing is an efficient searching technique in which the key of the records to be searched is obtained before finding the desired record. A key is converted into an integer within a limited range using hash function.

A hash function is one transforms a key into a <sup>hash table</sup> table index. If  $h$  is a hash function and  $k$  is the key,  $h(k)$  is called the hash of the key and is called the hash of index at which record with the key  $k$  should be placed.

Clearly two records can not occupy the same position. In direct addressing, it requires the large amount of space. Since the key may be extremely large but the items representing by keys may be limited. So using hashing technique, we can transform the key into limited table index.

The factors that control transformation in hashing:

1. The no. of address to be generated.



- ② The direct space required.
- ③ The direct access to starting address.

Collision:

Sometimes two or more records key are transformed into same addresses, which is called collision (synonyms).

Hashing Algorithms

- ① Folding: In this method, a key value is two folded, i.e. divided from approximately center and the addition of two parts is address used for storage.

For example: 621545

$$\begin{array}{r} 621 \\ + 545 \\ \hline 1166 \end{array} \leftarrow \text{Storage location.}$$

Folding is a simple method which may cause many collisions.

- ② Extraction: In this method, the digits at alternate position are extracted and some arithmetic operations are performed. The resultant value is used for address.

For example:

$$\begin{array}{r} \text{key } 821455 \\ \text{pick 1st, 3rd, 5th } 815 \\ \text{pick 2nd, 4th, 6th } - 245 \\ \hline 570 \end{array} \leftarrow \text{Storage location}$$

- ③ End Folding: In this method, a key is folded from both end to fit appropriate location.

For example: 10 digit key as

$$\begin{array}{r} 1234567890 \\ \hline 34567 \\ + 12890 \\ \hline 47457 \end{array} \leftarrow \text{Storage location}$$



Reduce to fit 50,000 location storage,  
if result > 50,000 subtract 50,000

④ Mid Square hashing: In mid square hashing the key is squared and a part is selected for address.

For example:

| key  | Square   | address |
|------|----------|---------|
| 1456 | 2119936  | 19      |
| 1882 | 3579664  | 79      |
| 9467 | 89624039 | 24      |
| 2434 | 5924356  | 24      |

} collision

Collision Removed in Hashing i.e. resolving Hash-clash

In hashing, we can use any algorithm to find the storage location of a record. For each algorithm we have a hash function that transforms key into table index. Clearly, in table, two records cannot occupy the same location. By applying hash function to the key there may be many duplicate index generated. If more than one key has same index, it is known as collision or hash-clash.

For example let us consider hash function is

$$h(k) = k \% 100$$

Then for keys 3892 and 3992,  $h(k)$  gives the same value 92. So, a collision occurs, so we have to resolve it. There are several methods for resolving such a hash-clash. The simplest methods for resolving such collisions are:

- ① Separate chaining.
- ② Open addressing.



### Separate Chaining :-

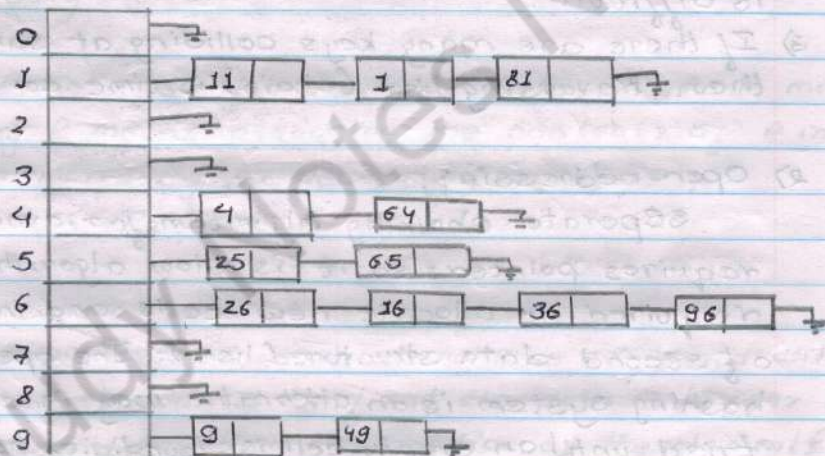
In separate chaining method, a separate linked list of elements is maintained that have the same index.

To search an element, we first obtain the index using the applied hash function then corresponding list is traversed to get the require element.

For example, let the hash function  $h(x)$  is defined as:

$$h(x) = x \% 10$$

We have the key values 11, 4, 25, 26, 16, 64, 36, 9, 49, 1, 81, 96, 65, then using separate chaining, the storage location and collision lists can be maintained as,



When  $x = 11$

$$h(11) = 11 \% 10 = 1$$

Insert key 11 in hash table in location 1.

when  $x = 4$ ,  $h(4) = 4 \% 10 = 4$

Insert key 4 in hash table in location 4

when  $x = 25$ ,  $h(25) = 25 \% 10 = 5$

Insert key 25 in hash table in location 5

when  $x = 26$ ,  $h(26) = 26 \% 10 = 6$

Insert key 26 in hash table in location 6.

When  $x = 16$ ,  $h(16) = 16 \% 10 = 6$

Insert key 16 in hash table in location 6 and similarly



### Advantages :

- 1) It resolves the hash clash even for large number of keys.
- 2) It has fixed table and dynamic data structures for list can be used, so, unlimited.
- 3) Better for number of records growing beyond the table size.

### Disadvantages :

- 1) Need extra space for hash table and pointer.
- 2) Implementation of second data structure (linked list) is difficult.
- 3) If there are many keys colliding at same location, then traversing list (chain) is time consuming.

### 2) Open addressing:

separate chaining algorithm for avoiding hash-clash requires pointers so it is slow algorithm due to time required to allocate new cells and implementation of second data structure (list). The open addressing hashing system is an alternative way in which cells are tried until an empty cell is found. i.e.  $h_0(x)$ ,  $h_1(x)$ ,  $h_2(x)$ , ... are tried in succession.

where,

$$h_i(x) = (h(x) + F(i)) \text{ with } F(0) = 0$$

The function  $F$  is for collision resolution. Because all data go inside the table, a bigger table is needed for open addressing than separate chaining.

The most common method used for this collision resolution with open addressing are:

- 1) Linear probing
- 2) Quadratic probing



i) Linear probing: If the function  $F$  for collision resolution is linear, then it is known as linear probing. In linear probing  $F(i) = i$ .

Example: Consider the keys 49, 28, 39, 68, 69, 79, 99 that are to be inserted into hash table using some hash function say  $h(x)$ ,  $h(x) = x \% 10$ .

Sol<sup>n</sup>:

Initial

Empty table:

|   | 49 | 28 | 39 | 68 | 69 | 79 | 99 |
|---|----|----|----|----|----|----|----|
| 0 |    |    | 39 | 39 | 39 | 39 | 39 |
| 1 |    |    |    | 68 | 68 | 68 | 68 |
| 2 |    |    |    |    | 69 | 69 | 69 |
| 3 |    |    |    |    |    | 79 | 79 |
| 4 |    |    |    |    |    |    | 99 |
| 5 |    |    |    |    |    |    |    |
| 6 |    |    |    |    |    |    |    |
| 7 |    |    |    |    |    |    |    |
| 8 |    | 28 | 28 | 28 | 28 | 28 | 28 |
| 9 | 49 | 49 | 49 | 49 | 49 | 49 | 49 |

When  $x = 49$ ,

$$h(49) = 49 \% 10 = 9$$

Insert key 49 in hash table in location 9.

When  $x = 28$

$$h(28) = 28 \% 10 = 8, \text{ Insert key 28 in hash table in location 8.}$$

When  $x = 39$

$$h(39) = 39 \% 10 = 9 \text{ (Collision occur)}$$

Since, collision occur at index 9. So, next position can be calculated as,

$$h_0(x) = ((39 \% 10 + 0) \% 10) = 9 \text{ not empty (49)}$$

$$h_1(x) = ((39 \% 10 + 1) \% 10) = 0 \text{ empty.}$$

So, 39 is placed at index 0.



when  $x = 68$

$$h(68) = 68 \% 10 = 8 \text{ (collision occur)}$$

Since, collision occur at index 8. So next position can be calculated as,

$$h_0(x) = ((68 \% 10 + 0) \% 10) = 8 \text{ not empty (28)}$$

$$h_1(x) = ((68 \% 10 + 1) \% 10) = 9 \text{ not empty (49)}$$

$$h_2(x) = ((68 \% 10 + 2) \% 10) = 10 \% 10 = 0 \text{ not empty}$$

$$h_3(x) = ((68 \% 10 + 3) \% 10) = 11 \% 10 = 1 \text{ empty.}$$

So, 68 is placed at index 1.

when  $x = 69$

$$h(69) = 69 \% 10 = 9 \text{ (collision occur)}$$

Since, collision occur at index 9. So next position can be calculated as,

$$h_0(x) = ((69 \% 10 + 0) \% 10) = 9 \text{ not empty}$$

$$h_1(x) = ((69 \% 10 + 1) \% 10) = 10 \% 10 = 0 \text{ not empty}$$

$$h_2(x) = ((69 \% 10 + 2) \% 10) = 11 \% 10 = 1 \text{ not empty.}$$

$$h_3(x) = ((69 \% 10 + 3) \% 10) = 12 \% 10 = 2 \text{ empty.}$$

and similarly.

ii) Quadratic probing: The linear probing method for collision resolution creates clustering problems since the collision is removed using linear displacement. In quadratic probing the resolution function  $F$  is quadratic function if collision occurs.

$$\text{i.e. } h_i(x) = (h(x) + F(i)) \% 10$$

$$\text{where, } F(i) = i^2$$

Example: Consider the above example with quadratic probing: 49, 28, 39, 68, 69, 79, 99

Soln,



|   | 49 | 28 | 39 | 63 | 69 | 79 | 99 |
|---|----|----|----|----|----|----|----|
| 0 |    |    | 39 | 39 | 39 | 39 | 39 |
| 1 |    |    |    |    |    |    |    |
| 2 |    |    |    | 68 | 68 | 68 | 68 |
| 3 |    |    |    |    | 69 | 69 | 69 |
| 4 |    |    |    |    |    |    | 99 |
| 5 |    |    |    |    |    | 79 | 79 |
| 6 |    |    |    |    |    |    |    |
| 7 |    |    |    |    |    |    |    |
| 8 |    | 28 | 28 | 28 | 28 | 28 | 28 |
| 9 | 49 | 49 | 49 | 49 | 49 | 49 | 49 |

When  $x = 49$

$$h(49) = 49 \% 10 = 9$$

Insert key ~~29~~ 49 in hash table in location 9.

when  $x = 28$

$$h(28) = 28 \% 10 = 8$$

Insert key 28 in hash table in location 8.

when  $x = 39$

$h(39) = 39 \% 10 = 9$  collision occur.

Since, collision occur at index 9. so next position can be calculated as,

$$h_0(x) = ((39 \% 10 + 0) \% 10) = 9 \text{ not empty}$$

$$h_1(39) = ((39 \% 10 + 1^2) \% 10) = 0 \text{ not empty}$$

when  $n = 68$

$$h(68) = 68 \% 10 = 8 \quad \text{collision occurs}$$

Since, collision occur at index  $x$ . So next position can be calculated as,

$$ho(68) = ((68 \% 10 + 0) \% 10) = 8 \quad \text{not empty}$$

$$h_1(68) = ((68 \% 10 + 1^2) \% 10) = 9 \text{ not empty}$$

$$h_2(68) = ((68 \% 10 + 2) \% 10) - 12 \% 10 = 2 \text{ empty.}$$

and similarly.



## Multi-way Trees

Multi-way search tree of order  $m$  (or an  $m$ -way search tree) is a search tree with the following properties:

- ① Each node can contain  $m-1$  key elements, where  $m$  is the order of the tree.
- ② Each node can have  $m$  branches (subtrees).
- ③ It satisfies the property of BST, the elements with smaller than the key are positioned into the left subtree and the elements with greater or equal values are positioned into the right subtree of the key.
- ④ All leaf nodes are on the same level so the tree is balanced.
- ⑤ All subtrees are themselves  $m$ -way trees.

### B-tree:-

The B-tree of order  $m$  can be defined as follows:

- \* A B-tree is a  $m$ -way search tree.
- \* The root is either a leaf or it has 2 to  $m$  subtrees, where  $m$  is the order of the B-tree. minimum:  $\lceil m/2 \rceil - 1$
- \* All nodes can contain maximum  $m-1$  ~~max~~ elements.
- \* All nodes except the root and the leaves can have minimum  $\lceil m/2 \rceil$  and maximum  $m$  branches (subtrees).
- \* All leaf nodes are on the same level so the tree is balanced.
- \* It retains the general property of BST.
- \* Each node contains the key element and the pointers to the children.
- \* When a new key is inserted into full node (with  $m-1$  elements), then the node is split into two new nodes at the same level and the key with the median value is inserted in the parent node in case the parent node is the root, a new root will be created.



\* All the key elements are arranged in a defined order (ascending order) within the node.

For example:

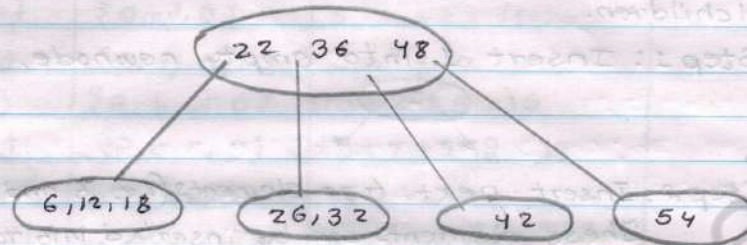


Fig: A B-tree of order 4

Insertion into the B-tree :

- ① First, find the position in the tree where the element can be inserted. If the node does not violate the B-tree condition, that is, if the position is found at non-full node then insert the element into that position in proper order.
- ② If the node is a full node then the node splits at the median element into two nodes on the same level, each with the minimum number of elements. This process continues recursively up to the tree until the root is reached; if the root is split then a new root is created.
- ③ The median key is not put into either of the two new nodes, but is instead sent up to the tree to become a new root and to be inserted into the parent node.
- ④ If the order of B-tree is an odd number, then the median is found by using  $m/2 + 1$ . And if the order is an even number then median is found by using  $m/2$ .

Example:

Construct the B-tree of order 4 for the following elements:

1, 6, 8, 2, 9, 12, 15, 7, 18, 3, 4, 20



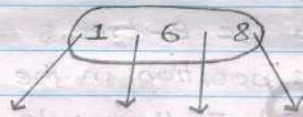
Soln: 1, 6, 8, 2, 9, 12, 15, 7, 18, 3, 4, 20

Since the order of the B-tree is 4, so each node can contain maximum 3 elements and can have maximum 4 children.

Step 1: Insert 1 into empty new node,



Step 2: Insert next two elements (i.e. 6 and 8) into root node. These elements can be inserted into root node because each node can contain maximum 3 elements as shown below:



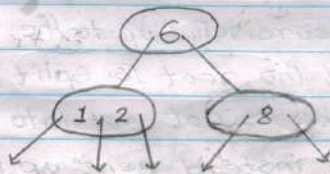
Step 3: Now, insert next elements (2) into the tree (root node)

Since root node is full, so it splits into two new nodes at the same level at the mid element

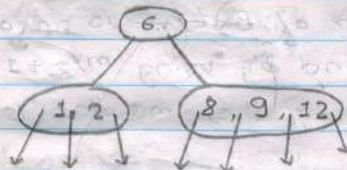
( $m/2 = 2$ ; the element at position 2 is 6) and

the median is sent upward, so create it as

root node.

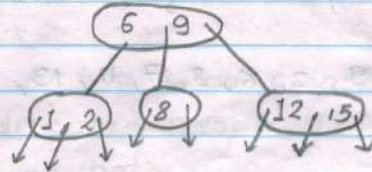


Step 4: Now, insert next two elements i.e. 9, 12

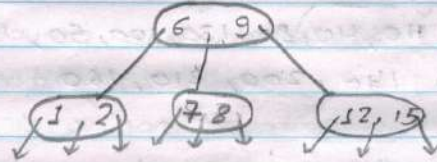




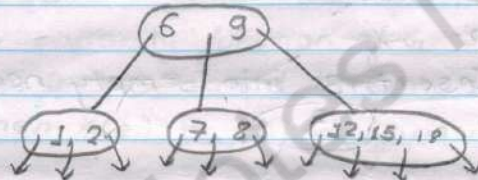
Step 5: Insert 15



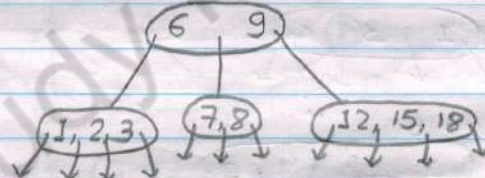
Step 6: Insert 7



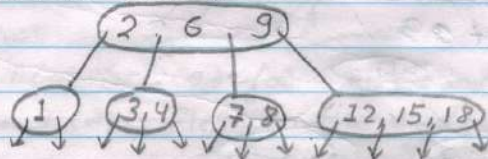
Step 7: Insert 18



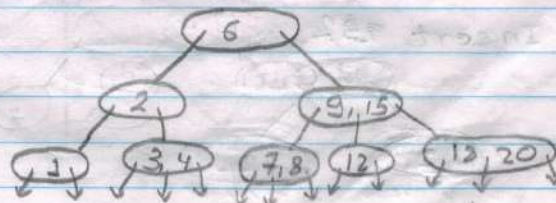
Step 8: Insert 3



Step 9: Insert 4



Step 10: Insert 20





Q<sub>1</sub>. Illustrate the growth of B-tree of order 5 inserting the keys

3, 1, 4, 5, 9, 2, 6, 8, 7, 11, 13, 19, 2, 23

Q<sub>2</sub> Order 5:

10, 70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90, 180, 240, 30, 120, 140, 200, 210, 160

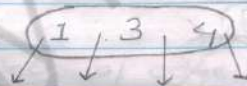
Q<sub>1</sub>. Soln.

Since the order of B-tree is 5, so each node can contain maximum 5 elements and can have maximum 5 children.

Step 1: Insert 3 into empty new node



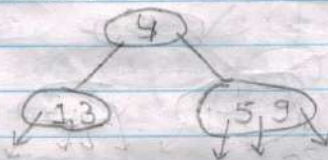
Step 2: Insert 1, 4



Step 3: Insert 5



Step 4: Insert 9

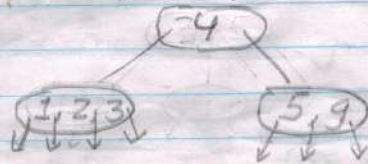


Step 5: Insert 2

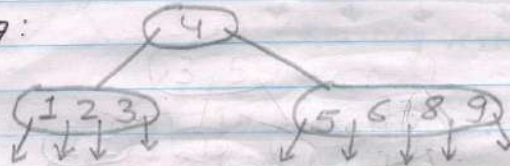




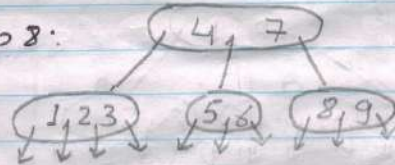
Step 6: Insert ~~4~~ 3



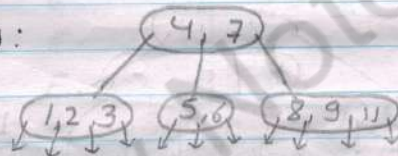
Step 7:



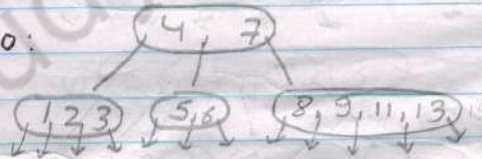
Step 8:



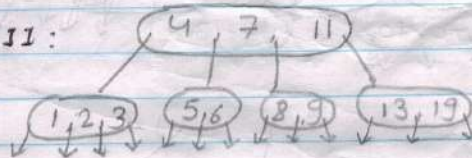
Step 9:



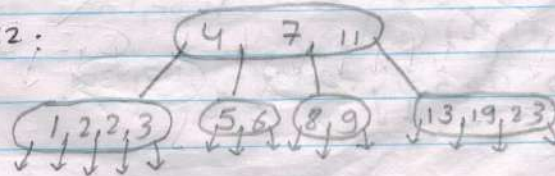
Step 10:



Step 11:



Step 12:



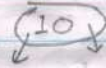


# overflow is a situation in which m-1 condn is fulfilled  
where we can't insert more element in tree.

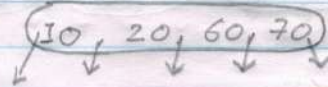
www.studynotesnepal.com

Q. 2.

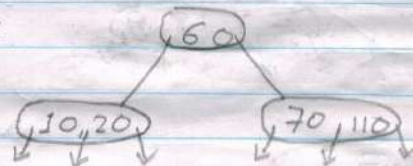
Step 1:



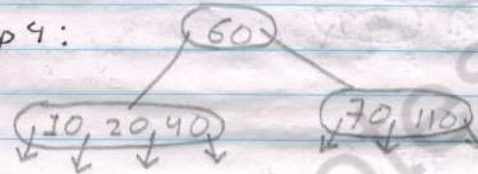
Step 2:



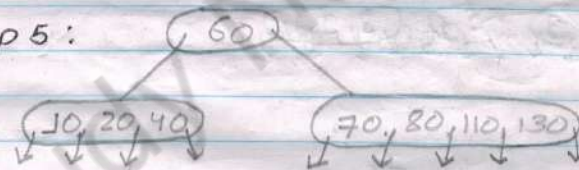
Step 3:



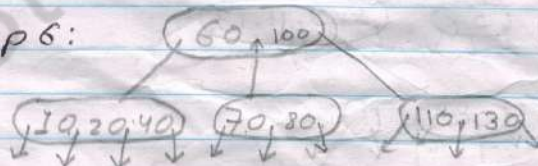
Step 4:



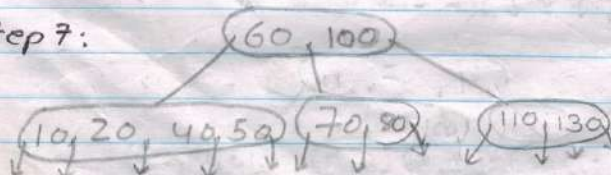
Step 5:



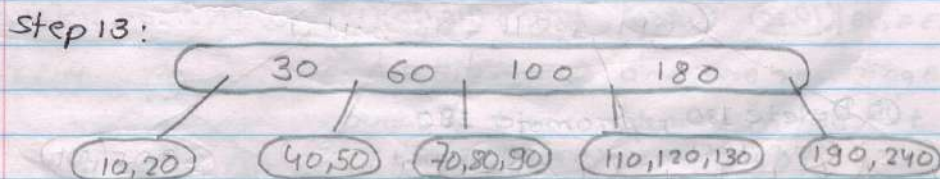
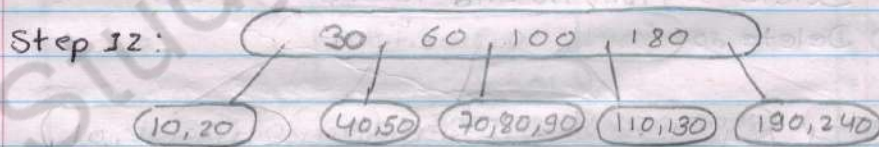
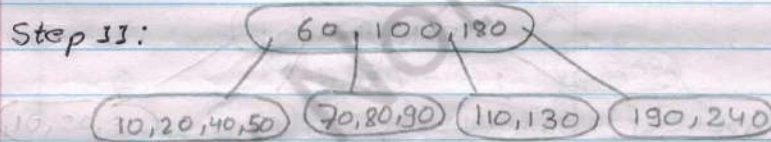
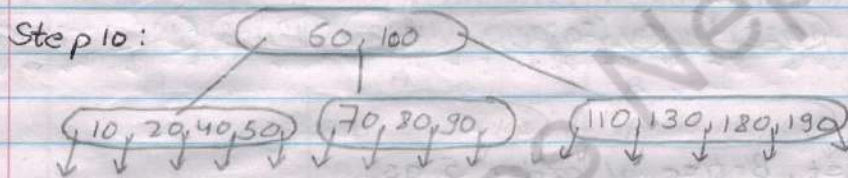
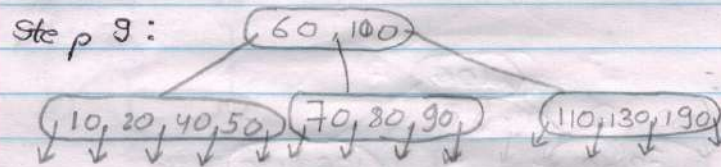
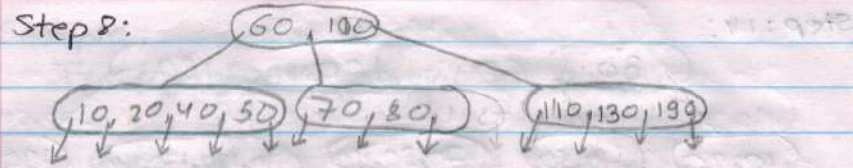
Step 6:



Step 7:

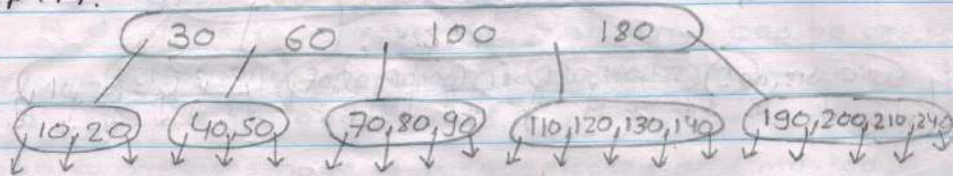




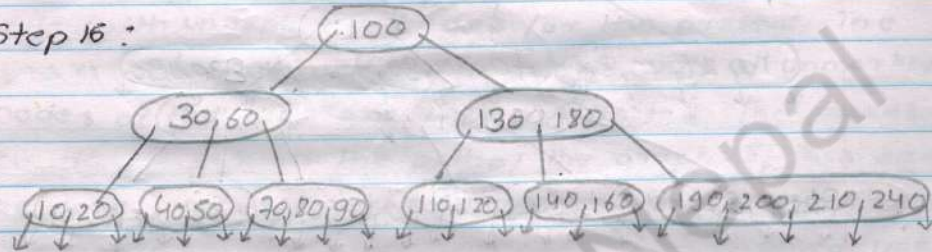




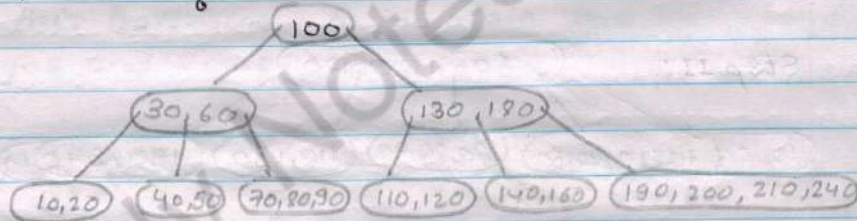
Step 14:



Step 16:

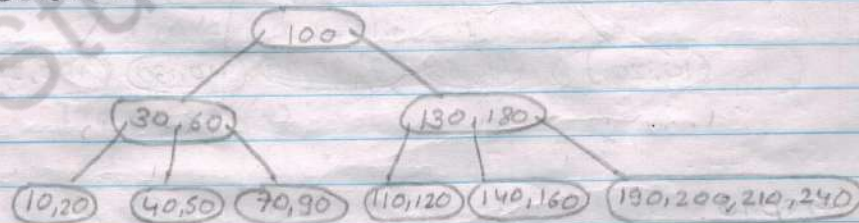


Let, B-tree of order 5 as,

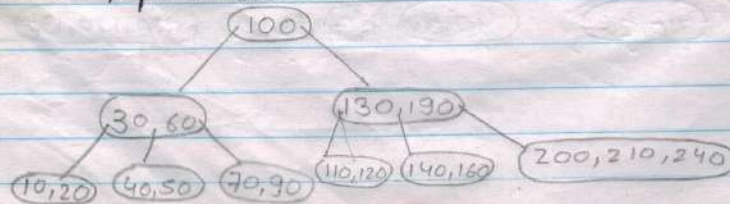


Delete 80, 180, 160 and 90 in order

i Delete 80

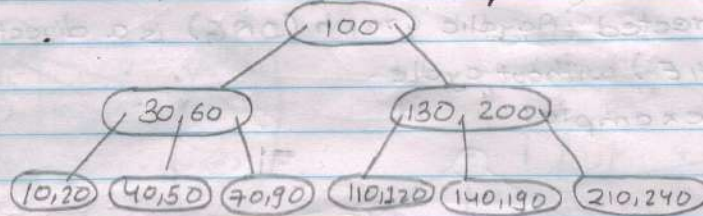


ii Delete 180, promote 190

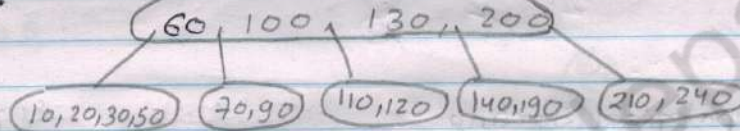




③ Delete 160, demote 190 and promote 200



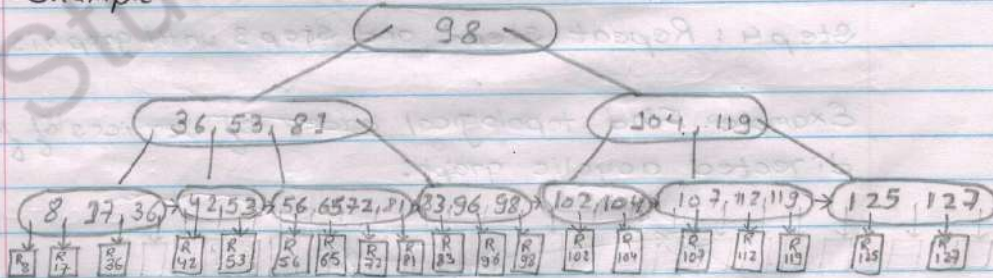
④ Delete 40, demote 60 and promote



### • B<sup>+</sup> Trees

- \* At most  $n$  sub-trees and  $n-1$  keys (where  $n$  is order of B<sup>+</sup> trees).
- \* At least  $\lceil n/2 \rceil$  sub-trees and  $\lceil n/2 \rceil - 1$  keys.
- \* Root: at least 2 sub-trees and 1 key.
- \* The key can be repeated in non-leaf nodes.
- \* The leafs are linked together with pointers.

Example:



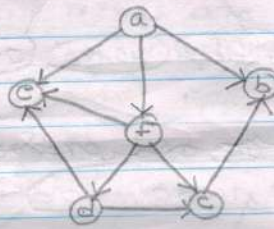
- \* Only the leafs point to data pages.



### Directed Acyclic Graph (DAG)

Directed Acyclic Graph (DAG) is a directed graph  $G = (V, E)$  without cycle.

For example:



### Topological Sorting

A topological sort orders vertices in a directed acyclic graph such that if there is a path from 'u' to 'v', then 'v' appears after 'u' in the ordering.

Topological sort Algorithm:

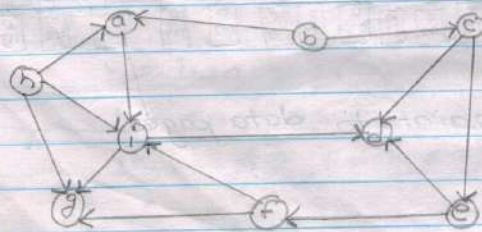
Step 1: Calculate the indegree of each vertex in DAG.

Step 2: Identify vertices that have no incoming edge i.e. in-degree zero.

Step 3: Delete the vertex of in-degree 0 and all its outgoing edges from the graph place it in the output.

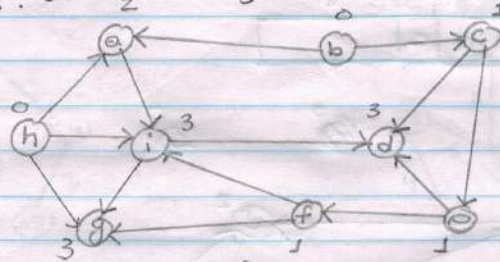
Step 4: Repeat step 2 and step 3 until graph is empty.

Example: Find topological order of vertices of following directed acyclic graph:

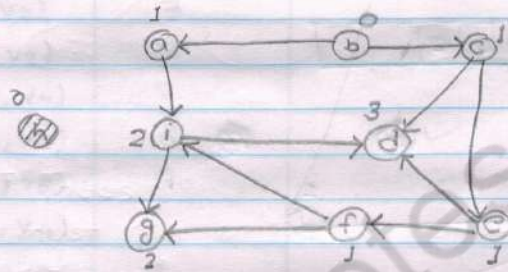




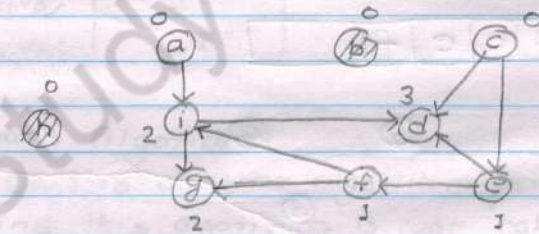
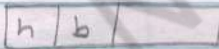
Step 1: Calculate indegree:



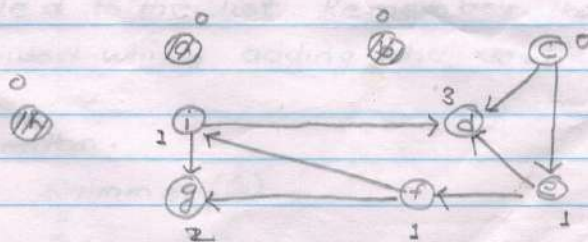
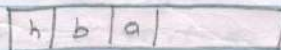
Step 2:



Step 3:



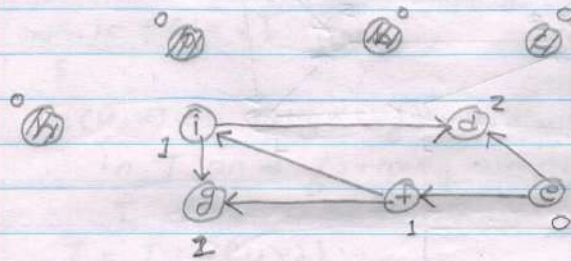
Step 4:





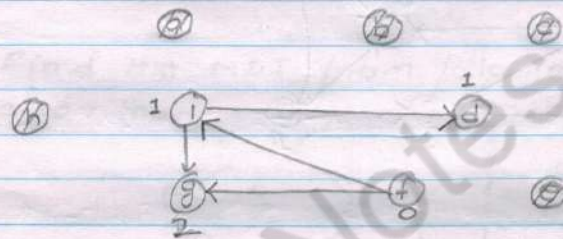
Step 5:

|   |   |   |   |  |
|---|---|---|---|--|
| h | b | a | c |  |
|---|---|---|---|--|



Step 6:

|   |   |   |   |   |  |
|---|---|---|---|---|--|
| h | b | a | c | e |  |
|---|---|---|---|---|--|



Step 7:

|   |   |   |   |   |   |  |
|---|---|---|---|---|---|--|
| h | b | a | c | e | f |  |
|---|---|---|---|---|---|--|

