Session-1
INTRODUCTION TO DS

Data-

- Data is a basic unit that any computing system centers around.

Data Object-

- Data object is a term that refers to set of elements. Such set may be finite or infinite.

Data Structure-

- Data Structure is an arrangement of data in computer's memory (or sometimes on a disk) so that we can retrieve it & manipulate it correctly and efficiently.
- Data structure is a set of domains D, a set of functions F and a set of axioms A.
- A triple (D, F,A) denotes the data structure d.

Data Type

- Information itself has no meaning because it is just sequence of bytes.
- It is interpretation of bit pattern that gives it’s meaning.
- For example 00100110 can be interpreted as number 38(binary), number 26(binary coded decimal) or the character ‘&’.
- A method of interpreting bit pattern is called as a data type.
- So data type is a kind of data that variable may hold in a programming language. For example in ‘C++’ int, float, char & double

Abstract Data Type

- ADT is a conceptual representation.

- ADT is a mathematical model together with various operations defined in that model.

- ADT is a way of looking at Data Structure focusing on what it does and not how it does.

- Data Structure is implementation of ADT.

Examples of Data Structure

- Arrays
- Stacks
- Queues
- Linked Lists
- Trees
- Graphs

Array

- Arrays are defined as a finite ordered set of homogeneous elements.
- Finite means there is a specific number of elements in the array.
- Ordered means the elements of the array are arranged so that so that there is zeroth, first, second and so on elements.
- Homogeneous means all the elements in the array must be of the same type.
  - E.g. An array may contain all integers or all characters but not integers and characters.
  - Declaration in C++ int a[100];

Operations on array

- The two basic operations that access an array are Extraction and Storing.
- The Extraction operation is a function that accepts an array a, and an index I, and returns an element of the array.
- In C++ this operation is denoted by the expression a[i];
- The storing operation accepts an array a, an index I and an element x.
- In C++ this operation is denoted by the expression a[I] = x;

Array

- The smallest element of an array's index is called it's lower bound. It is 0 in C++.

- The highest element is called it's upper bound.

- If Lower bound is "lower" and Upper bound is "upper" then number of elements in the array, called it's "range" is given by range=upper – lower + 1.
  - E.g. In array a, the lower bound is 0 and the upper bound is 99,so the range is 100.

- An important feature of array in C++ is that the upper bound nor the lower bound may be changed during the program execution.

- Arrays will always be stored in contiguous memory locations.

- You can have static as well as dynamic arrays, where the range of the array will be decided at run time.

Single- Dimension Array (1D Array)

```
int arr[10], i;
for( i = 0; i < 10; i++ )
{
    arr[i] = i;
}

for ( i = 0; i < 10; i++ )
{
    cout<< i ;
}
```

Two-Dimensional Array (2D Array)

- Two dimensional array can be considered as an array of 1–D array.
- Declaration int arr[2][2];
- This defines a new array containing two elements. Each of these elements is itself an array containing 2 integers.
- Each element of this array is accessed using two indices: row number and the column number.

Physical:

10	**20**	**30**	**40**

$\underbrace{10 \quad 20}_{0^{th}\ \text{1D Array}} \quad \underbrace{30 \quad 40}_{1^{st}\ \text{1D Array}}$

Logical:

	C0	**C1**
R0	**10**	**20**
R1	**30**	**40**

Using 2D Array

```
int arr[2][2], I , j;


for( I = 0; I < 2; I ++ )
    for( j = 0; j < 2; j++ )
     arr[ I ][ j ] = I + j;


for( I = 0; I < 2; I ++ )
    for( j = 0; j < 2; j++ )
     cout<< arr[ I ][ j ];
```

Advantages of Array-

- Simple and easy to understand.
- Contiguous allocation.
- Fast retrieval because of indexed nature.
- No need for the user to be worried about allocation and de-allocation of arrays.

Disadvantages of Array-

- If you need m elements out of n locations defined. n-m locations are unnecessarily wasted if n>m

- You can not have more than n elements. -
  (I.e static allocation of memory.)

- large number of data movements in case of insertion & deletion, which leads to more overheads.

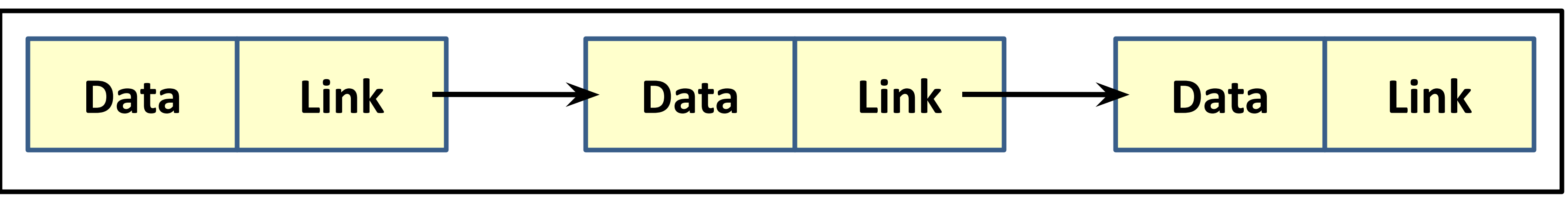
Session-2
Linked List

Drawbacks of Array

- Array does not grow dynamically (i.e length has to be known)
- Inefficient memory management.
- In ordered array insertion is slow.
- In both ordered and unordered array deletion is slow.
- Large number of data movements for insertion & deletion which is very expensive in case of array with large number of elements.
- Solution : Linked list.

Linked List

- We can overcome the drawbacks of the sequential storage.
- If the items are explicitly ordered, that is each item contained within itself the address of the next Item.
- Such an explicit ordering gives rise to a data structure called Linear Linked List.
- In Linked Lists elements are logically adjacent, need not be physically adjacent.

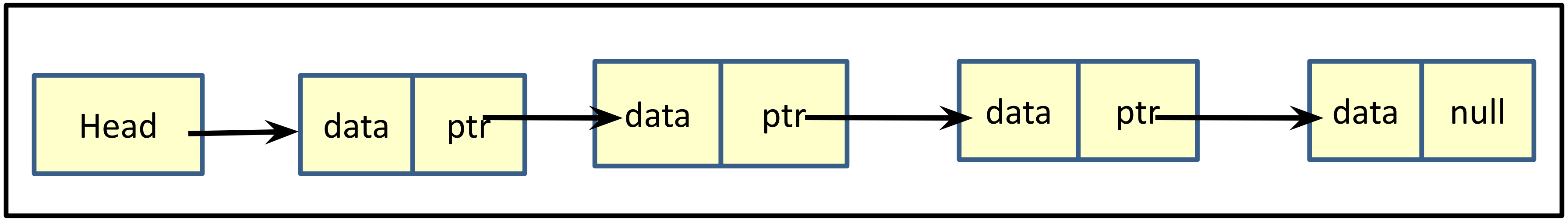


Linked List Structure

- Each Item in the list is called a node and contains two fields, an data field and next address field.
- The data field holds the actual element on the list.
- The link or the next address field contains the address of the next node in the list.
- Such an address which is used to access a particular node, is known as a pointer.
- The entire linked list is accessed from an external pointer Head, which points to the first node of the list.
- The next field of the last node in the list contains a special value, known as null, which is not a valid address.
- This null pointer is used to signal the end of the list.

Linked List

- Here the head is a pointer which is pointing to the first node of the list.
- The entire list can be accessed through head.
- So to maintain a list is nothing but maintaining a head pointer for the list.



Operations on Linked List

- Insert ( )
  - 1. Create a new node.
  - 2. Set the fields of the new node.
  - 3.If the linked list is empty insert the node as the first node.
  - 4. If the node precedes all others in the list then insert the node at the front of the list.
  - 5. Repeat through step 6 while information content of the node in the list is less than the information content of the new node.
  - 6. Obtain the next node in the list.
  - 7. Insert the new node in the list.

Operations on Linked List

- Delete ( )
  - 1. If the list is empty then write underflow and return.
  - 2. Repeat through step 3 while the end of list has not been reached and the node has not been found.
  - 3. Obtain the next node in the list and record its predecessor node.
  - 4. If the end of the list is reached and node not found then write node not found and return.
  - 5. If found delete the node from the list.
  - 6. Free the node deleted.
  - 7. Set the links of the nodes one which follows the deleted node and one which precedes the deleted node.

Operations on Linked List

- Search ( )
  - Start from the head node. Compare the key with the data item of each node. If match not found and end of list is reached then the element being searched is not present.

    If found return it's position.

- Display ()
  - Start with the first node. Print the data of the current node.
  - Get the address of the next node, and make it as current node.
  - Continue the process un till the next node is NULL.

Implementation of Linked List

```
class Node {

  int data;
  Node * next;

public:
  Node ( int data ) ;
  int getData ();
  void setData ( int data );
  Node * getNext ();
  void setNext ( Node * next );

};
```

```
class SinglyLinkedList {

  Node * head;

public:
        SinglyLinkedList();
        void insert ( int data );
        void insertByPos ( int data, int pos );
        void delByVal ( int val );
        void delByPos ( int pos );
        void display ();
        int search ( int val );
        void printReverse();

};
```

Singly Circular Linked List

- A small change is made to the linear list, so that the next field in the last node contains a pointer back to the first node rather than the NULL pointer.

- Such a linked list is called as the circular linked list.

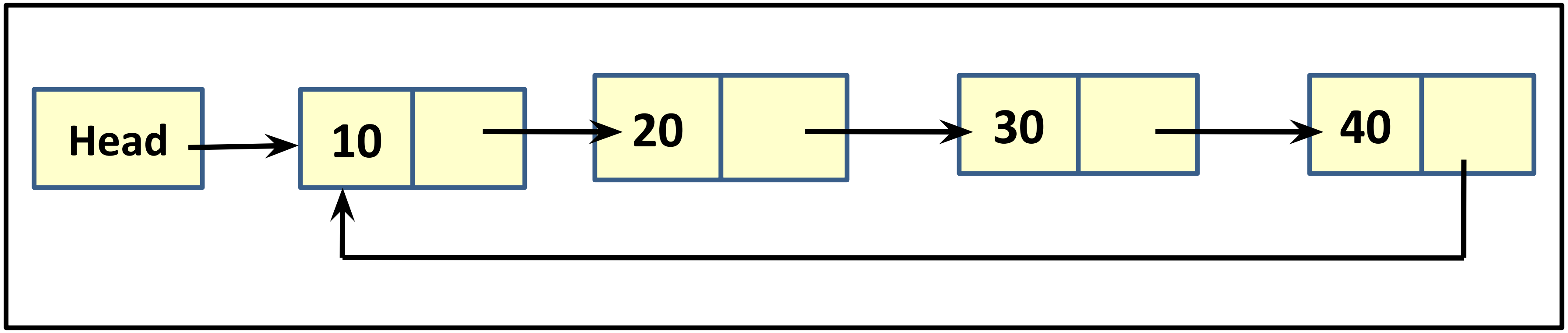
- From any point in such a list it is possible to reach any other point in the list.

- A NULL pointer represents an empty circular linked list.

Singly Circular Linked List Structure

- In This case as the list is circular there is no first or last node of the list. However some convention should be used so that a certain node is last node and the node following that is first node.

- The address of the first node is placed in the head.



Single Circular Linked List

- Advantages
  - Every node is accessible from a given node.
  - In singly linked list to do every operation we have to go sequentially starting from the head. This is not a case with circular linked list.

- Disadvantages :
  - Without some care in programming it is possible to get into an infinite loop (identify the head node).

Implementation of Circular Linked List

```
class CLinkedList {
  Node * head;

public:
  CLinkedList();
  void insert( int data );
  void insertAtBeg( int data );
  void delByPos( int pos );
  void display();
  ~CLinkedList();
};
```

Doubly Linked List

- Singly and Circular are Linked lists in which we can traverse in only one direction. Sometimes it is required that we should be able to traverse the list in both directions.

- For example to delete a node from the list we ought to have a address of previous node in the list.

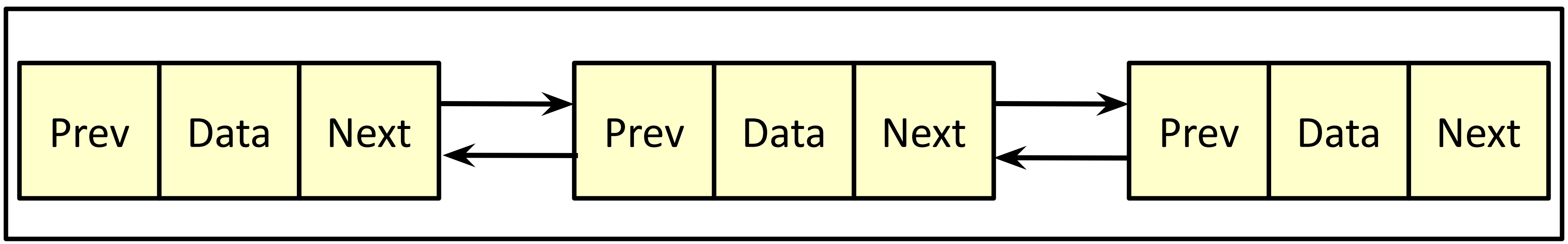
- For a linked list to have such a property implies that each node must contain two link fields instead of one.

- The links are used to denote the predecessor and successor of a node.

- A linked list containing this kind of nodes is called Doubly linked list.

Doubly linked list-Structure

- Here a node is having three parts
  - data part
  - address part to store address of previous node
  - address part to store the address of next node.



- The advantage here is that the list could be traversed in both the directions. Which removes the drawbacks of Singly and Singly circular linked list.

- But still in some situations it may be costly to put two address fields into a Node

Implementation- Double Linked List

```
class DLinkedList {

     Node * head;

public:

  DLinkedList();
  void insert( int data );
  void insert( int data, int pos );
  void deleteByVal( int data );
  void deleteByPos( int pos );
  void display();
  ~DLinkedList();
};
```

Session-3
Stack

Stack

- A Stack is an ordered collection of items into which new items may be inserted and from which items maybe deleted at one end called the top of stack.

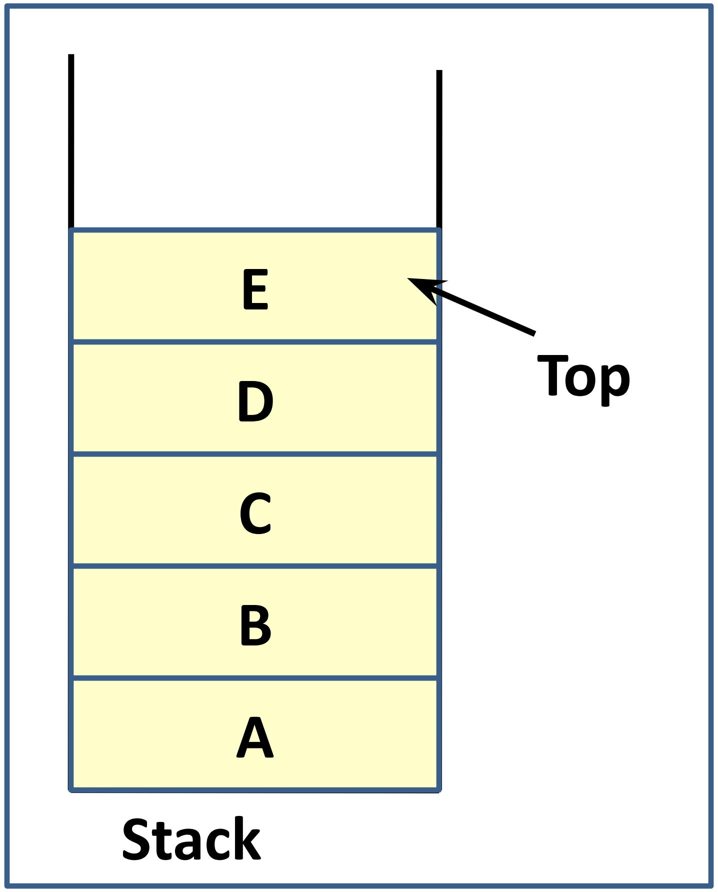
- The definition of the Stack provides insertion and deletion of items. So a stack is dynamically, constantly changing object.

- The definition specifies that a single end of the stack is designated as the stack top.

- New items may be put on top of the stack or items which are at the top of the stack may be removed.

Stack- Structure

- While adding the elements to stack A is added first then B, C, D, E and along with that the top of the stack keeps growing.
- Initially the Top of the stack is -1.
- While deleting the items from the stack E is deleted first and the D, C, B, A and the stack Top keeps decrementing.



Stack Operations

- Push
  - The Items are put on the Stack using Push function.
  - If S is the Stack the items could be added to it as S.Push(Item);

- Pop
  - The items from the stack can be deleted using the Pop operation.
  - Using pop only the topmost element can be deleted.
  - e.g. S.Pop();
  - The pop function returns the item that is deleted form the stack.

- StackEmpty

- StackFull

Stack Operations

- There is one more operation that can be performed on the Stack is to determine what the top item on the stack is without removing it.
- This operation is written as
  - S.Peek();
- It returns the top element of the Stack S.
- The operation Peep is not a new operation, since it can be decomposed into Pop & a Push.
- I = S.Peek(); is equivalent to
  - I = S.Pop();
  - S.Push(I);
- Like the operation Pop, Peep is not defined for an empty stack.

Some Facts On Stack

- There is no upper limit on the number of items that may be kept in a Stack, since the definition does not specify how many items are allowed in the Stack collection.

- If the stack does not contain any element then it is called the ‘Empty Stack’.

- Although the push operation is applicable to any stack , the Pop operation can not be applied to the Empty stack.

- So it is necessary to check out whether the stack is empty before Popping the elements from the Stack.

Usage of Stack

- Stacks are mainly used to support function call mechanism.
- To support or remove recursion.
- Conversion of Infix expressions to post fix expression, pre fix expression, and their evaluation.

Implementation of Stack-

- **Stack as an Object**

- **State:**
  - The data, Stack is holding. The stack Top position.

- **Identity:**
  - Every stack will have a name & location.

- **Behavior:**
  - Push the elements on Stack.
  - Pop the elements from Stack.

- **Responsibility:**
  - Manage the data in last in first out fashion

Stack Implementation

- Stack can be implemented in two ways
  - - Using an Array
  - - Using Linked list representation

```
class Stack {

  int * arr;
  int top;

Public:

  Stack();
  Stack( int );
  void push( int data );
  int pop();
  int StackFull();
  int StackEmpty();

};
```

Stack Implementation using Linked List

- The Stack may be represented by a linear linked list.

- The operation of adding an element to the front of a linked list is quite similar to that of pushing an element onto a stack.

- In both the cases the element is added as the only immediately accessible item in a collection.

- A Stack can be accessed only through it’s top element, and a list can be accessed through the pointer to the first element.

List as a Stack

- The operation of removing the first element from a Linked List is analogous to popping a stack.
- In both cases only immediately accessible item of a collection is removed from that collection, and the next item becomes the immediately accessible.
- The first node of the list is the top of the stack.
- The advantage of implementing a stack as a linked list is that stack is able to grow and shrink to any size.
- No space has been pre allocated to any single stack and no stack is using the space that it does not need.

Implementation

```
class Stack {

   Node * top;

Public:

   Stack ();
   void push( int data );
   int pop ();
   int stackEmpty ();

};
```

Session-4
Queue

Queue

- A Queue is an ordered collection of items from which items may be deleted at one end called front of the queue, and into which items may be inserted at the other end called the rear of the queue.

- The first element inserted in the queue is first element deleted from the queue.

Operation On Queue

- Four primitive operations can be performed on the queue.
- **Insert**:
  - The operation insert inserts the element at the rear of the queue.
- **Delete**:
  - The operation delete deletes the front element from the queue.
- **QEmpty**:
  - Returns true if the queue is empty.

Queue

- Initially the front and rear is set to -1.
- The queue is empty whenever the rear < front.
- The number of elements in the queue at any time is equal to the value rear – front.
- In simple queues it is possible to reach the absurd situation where the queue is empty, yet no new element could be added.
- One solution is to modify the remove operation so that when an item is deleted the entire queue is shifted to the beginning of the array.
- The queue will no longer need a front field. Since the element at the position 0 is always the first element.
- The empty queue is represented by the condition if rear equals -1.

Queue Implementation- Array

```
class Queue {

  int * arr;
  int size, front, rear;

public:

  Queue( int size);
  void insert( int data );
  int deleteData ( );
  int queueEmpty ( );
  int queueFull ( );

};
```

Queue Implementation Using Linked List

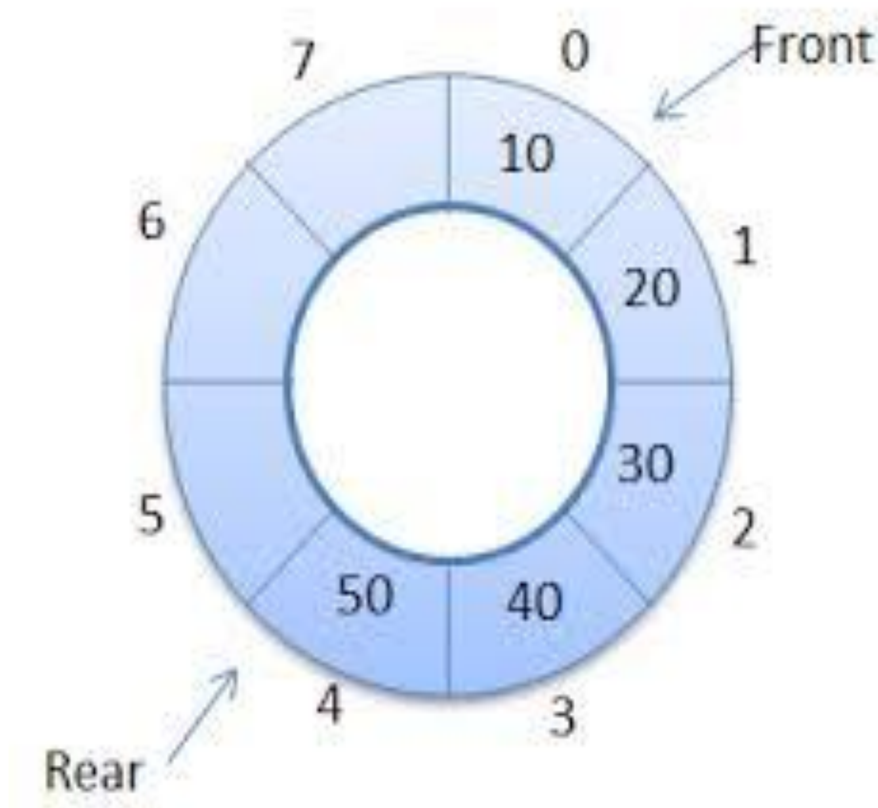
```
class Queue {

  Node * front;
  Node * rear;

public:
  Queue( );
  void insert( int data );
  int deleteData ( );
  int queueEmpty ( );
  int queueFull ( );

};
```

Circular Queue

- Static queues have a very big drawback that once the queue is FULL, even though we delete few elements from the "front“ and relieve some occupied space, we are not able to add anymore elements, as the "rear" has already reached the Queue's rear most position.
- The solution lies in a queue in which the moment "rear" reaches the Queue's last position, the "first" element will become the queue's new "rear".

Structure of Circular Queue

- Here in this queue as the rear will reach the last position, then it will be advanced to zero so that memory locations can be reused.



- The operations that can be performed on circular queue are
  - - Insert

Is Queue Empty or Full

- It is difficult under this representation to determine when the queue is empty.
- Both an empty queue and a full queue would be indicated by having the head and tail point to the same element.
- There are two ways around this: either maintain a variable with the number of items in the queue, or create the array with one

Other types of Queue

- **DQueue:**
  - It is a linear list in which elements can be added or deleted at either ends of the queue.

- **Priority Queue:**
  - It is a linear list in which elements are stored according to their priority of processing.

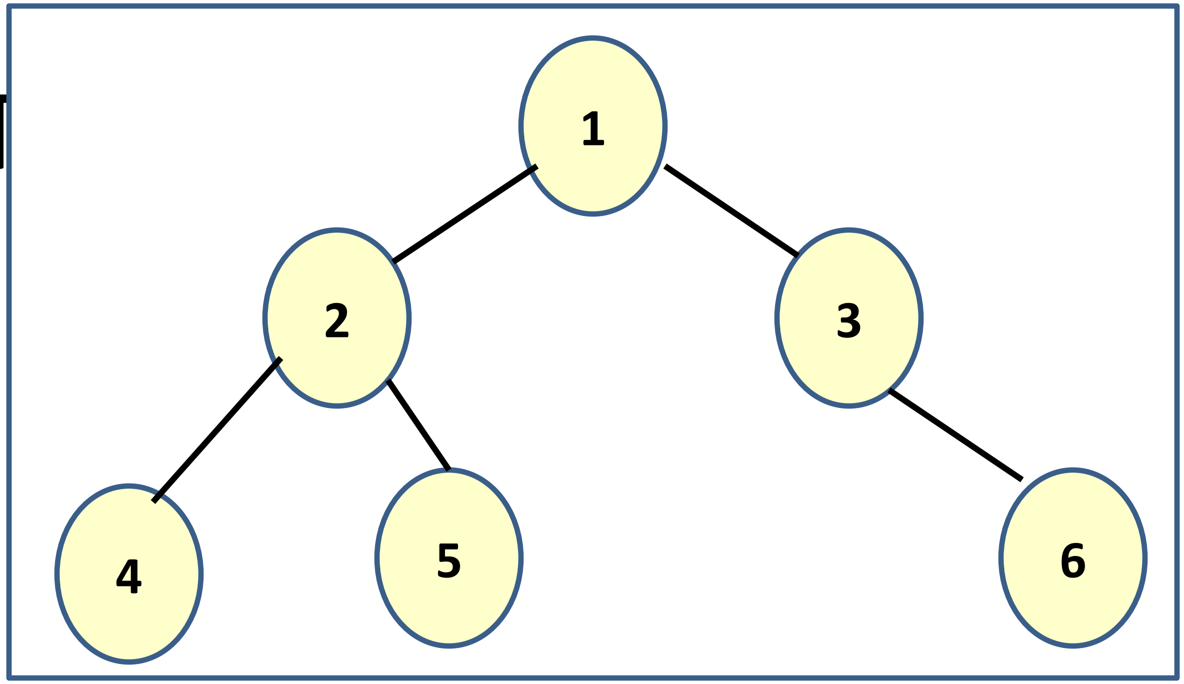
- **Bulk Move Queues:**
  - Here make the array much bigger than the maximum number of items in the queue. When you run out of

Session-4
Tree

Tree

- A tree consists of nodes connected by edges, which do not form cycle.
- For collection of nodes & edges to define as tree, there must be one & only one path from the root to any other node.
- A tree is a connected graph of N vertices with N-1 Edges.
- Tree is nonlinear data structure.

T



Recursive definition of Tree

- A tree is finite set of one or more nodes such that:
  - a) There is specially designated node called root.
  - b) The remaining nodes are partitioned into n>=0 disjoint sets T1…
- Tn where each of these sets is a tree. T1….Tn

Tree Terminology

- **Node:** A node stands for the item of information plus the branches of other items. E.g. 1,2,3,4,5,6

- **Siblings:** Children of the same parent are siblings. E.g. siblings of node 2 are 4 & 5.

- **Degree:** The number of sub trees of a node is called degree. The degree of a tree is the maximum degree of the nodes in the tree. E.g. degree of above tree is 2.

Tree Terminology

- **Ancestor:** The ancestor of a node are all the nodes along the path from the root to that node.

- **Level:** The level of a node is defined by first letting the root of the tree to be level = 1. If a node is at level x then the children are at level x+1.

- **Height/Depth:** The height or depth of the tree

Binary Tree

- If the degree of a tree is 2 then it is called binary tree.
- A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called left sub tree & right sub tree.
- The max number of nodes on level n of binary tree is 2^(n-1), n>= 1.
- The max number of nodes in binary tree of level k is (2^k)-1.
- If every non leaf node in a binary tree has non

Application of Binary Search Tree

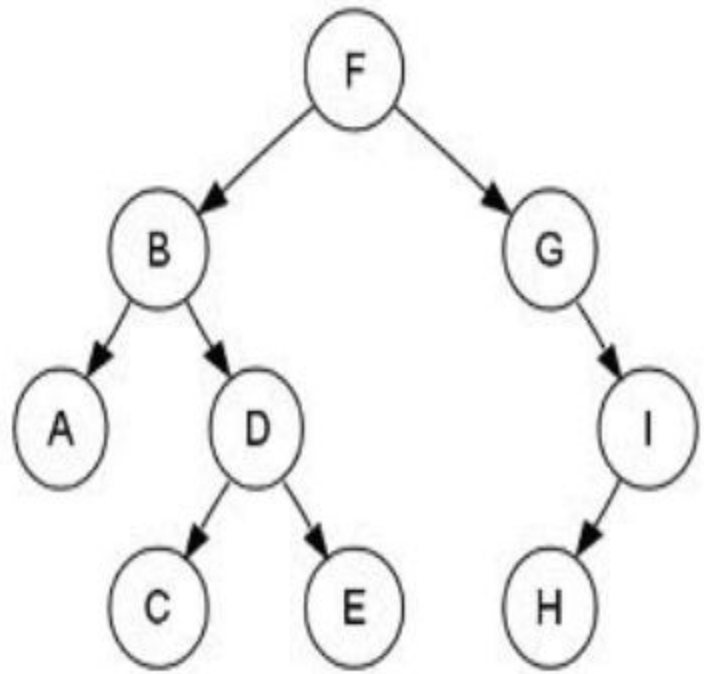
- Information retrieval is the most important use of Binary tree.
- A binary tree is useful when two way decisions must be made at each point in a process.
- e.g. Binary search tree.
- In decision trees where each node has a condition and based on the answer to the condition the tree traversal takes place.

Tree Traversals

Example-

- Pre Order –(DLR) (Node-Left-Right)

  Algorithm-
  – Visit the node first
  – Traverse left subtree in preorder
  – Traverse right subtree in preorder
  – Continue this process till all nodes have been visited



F B A D C E G I H

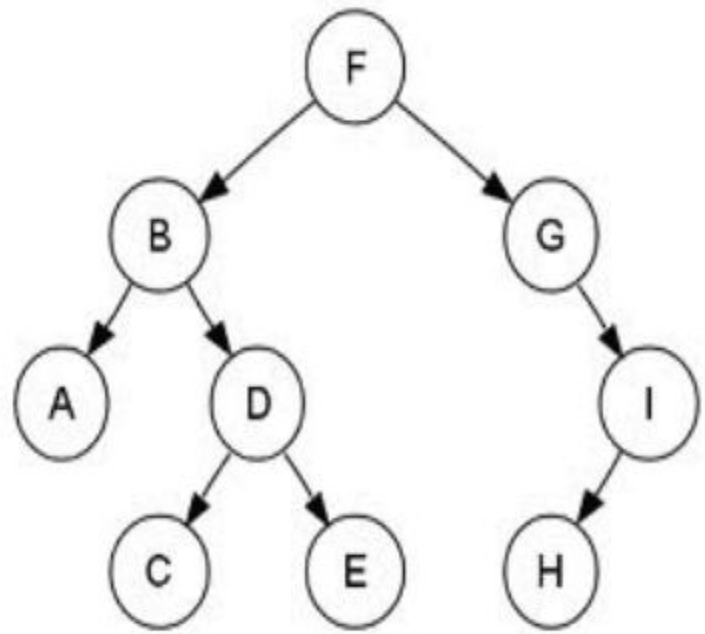
Tree Traversals

Example-

- In Order- (LDR) (Left-Node-Right)

Algorithm-

- Traverse left subtree in inorder
- Visit the node
- Traverse right subtree in inorder
- Continue this process till all nodes have been visited



A B C D E F G H I

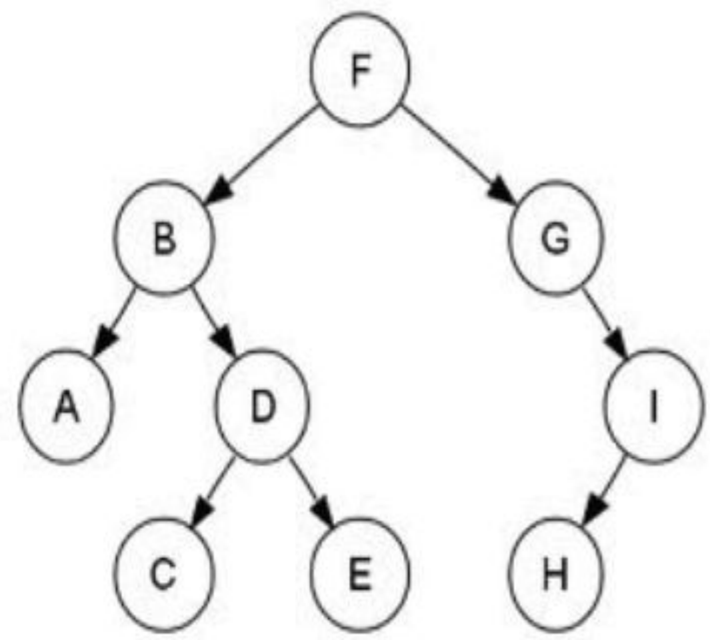
Tree Traversals

Example-

- Post Order- (LRD) (Left- Right- Node)

Algorithm-

- Traverse left subtree in postorder
- Traverse right subtree in postorder
- Visit the node
- Continue this process till all nodes have been visited



A C E D B H I G F

Level Wise Printing of Tree -BFS

Algorithm-

- Insert the root node into queue
- While the queue is not empty do the following
- Delete a node from the queue and print it
- Insert a node corresponding to it’s left subtree into the queue if it is not NULL
- Insert a node corresponding to it’s right subtree into the queue if it is not NULL
- Go to the step 2
- Stop

Implementation Structure For Classes

- Class Node will have the following structure

```
class Node {
        int data;
        Node * right;
        Node * left;
  public:
  Node( int data );
  int  getData();
  void setData( int );
  void setRight( Node * );
  Node * getRight();
  void setLeft( Node * );
  Node * getLeft();

};
```

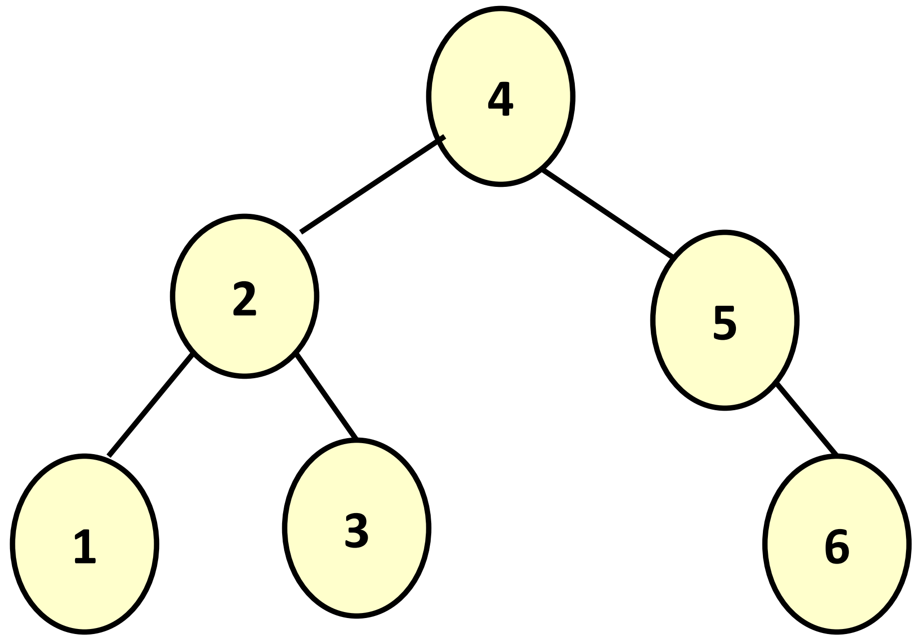
Class BinaryTree will have the following structure

```
class BinaryTree {
        Node * root;
public:
        TBinaryTree();
        void insert( int );
        void inOrder();
        void preOrder();
        void postOrder();
        void deleteData( int );
        ~BinaryTree();
};
```

Binary Search Tree

- It is a binary tree that is either empty or in which each node contains a key that satisfies the conditions:
- 1. All keys (if any) in the left sub tree of the root precede the key in the root.
- 2. The key in the root precedes all keys (if any) in the right sub

Put following numbers in a linked list into a binary search tree 4,2,5,1,3,6

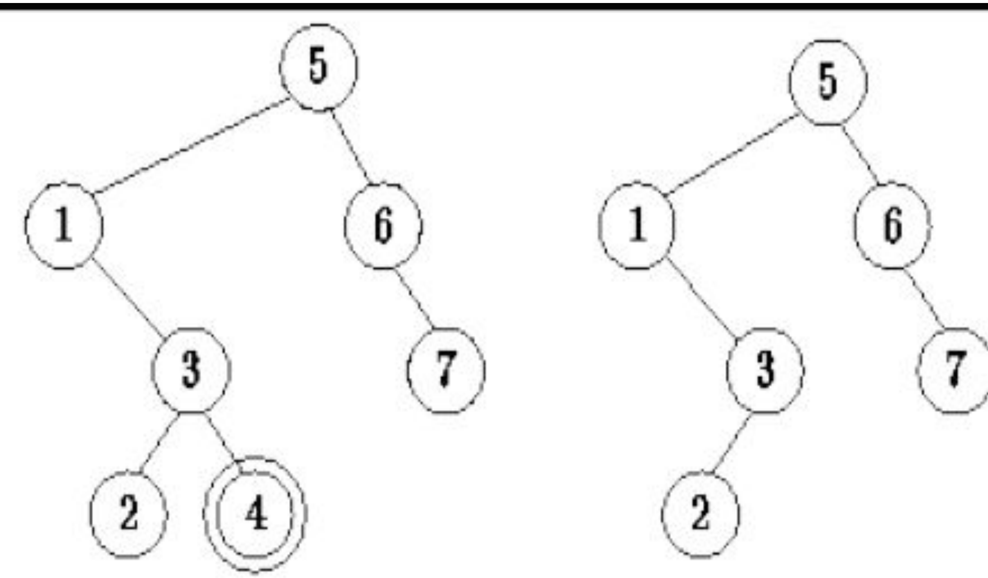


Removing Element from Binary Search tree

- When removing an item from a search tree, it is imperative that the tree satisfies the data ordering criterion.

- If the item to be removed is in a leaf node, then it is fairly easy to remove that item

For example, consider the binary search tree shown in Figure (a). Suppose we wish to remove the **node labeled 4.** Since node 4 is a leaf, its subtrees are empty. When we remove it from the tree, the tree remains a valid search tree as shown in Figure (b).



Removing Non-leaf Node From Binary Search Tree

- To remove a non-leaf node, we move it down in the tree until it becomes a leaf node since a leaf node is easily deleted.

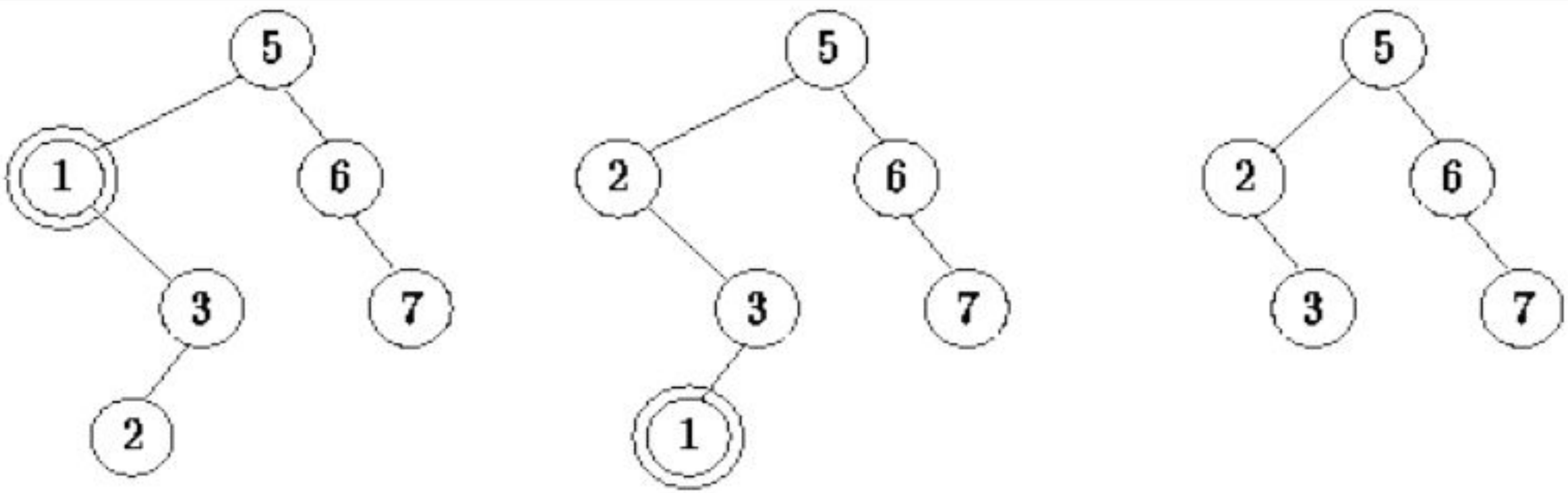
- To move a node down we swap it with another node which is further down in the tree.

- For example, consider the search tree shown in Figure (a).

- Node 1 is not a leaf since it has an empty left sub tree but a nonempty right sub tree.

- To remove node 1, we swap it with the smallest key in its right sub tree, which in this case is node 2.

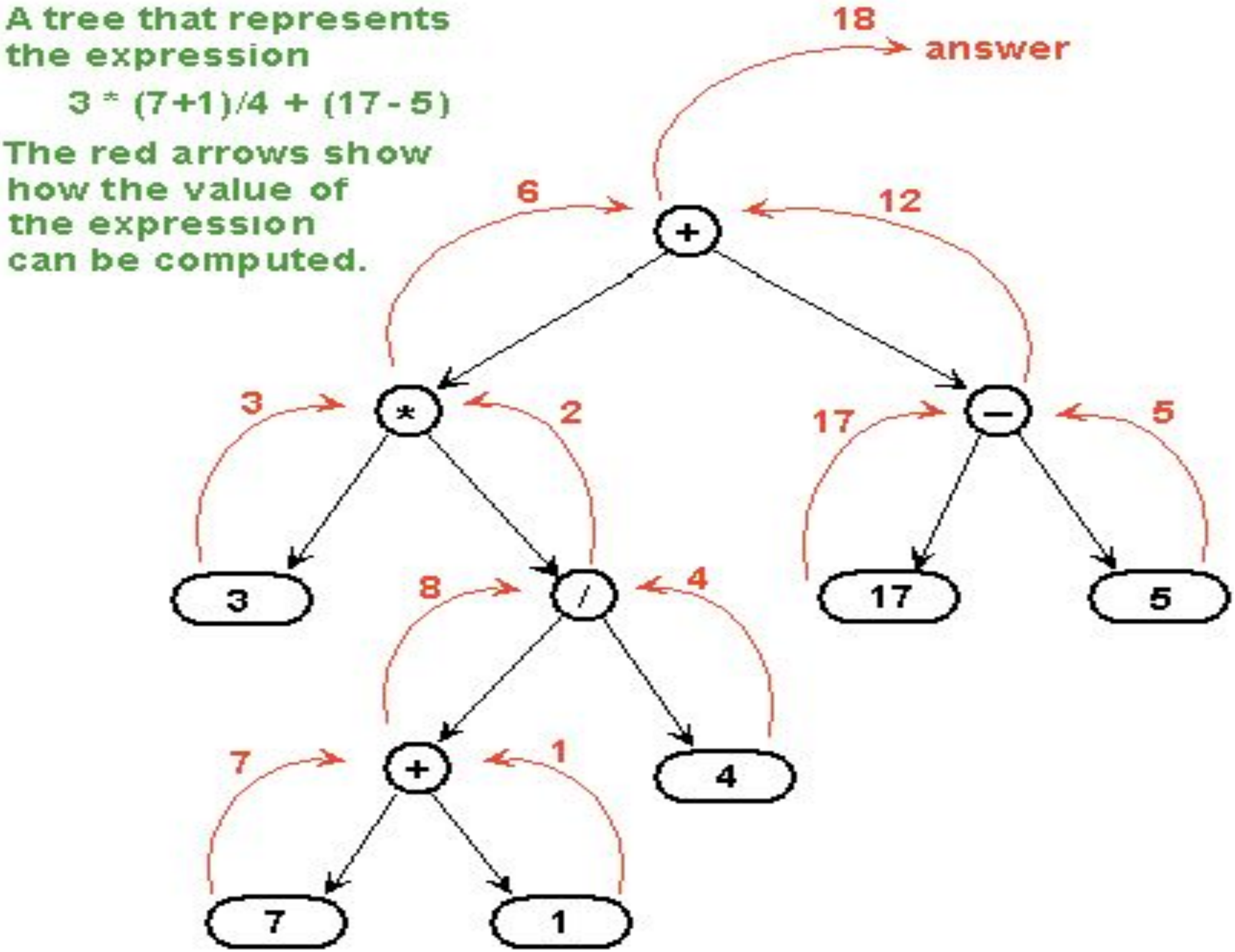
- To move a non-leaf node down in the tree, we either swap it with the smallest key in the right subtree or with the largest one in the left subtree.
- At least one such swap is always possible, since the node is a non-leaf and therefore at least one of its subtrees is nonempty.
- If after the swap, the node to be deleted is not a leaf, then we push it further down the tree with yet



(a) (b) (c)

Expression tree

- When an expression is represented through a tree, it is known as expression tree.
- The leaves of an expression tree are operands, such as constant variable names and all internal nodes contain operators.
- Preorder traversal of an expression tree gives prefix equivalent of the expression.
- Postorder traversal of an expression gives the postfix equivalent of the expression.


A tree that represents the expression
3 * (7+1)/4 + (17 - 5)
The red arrows show how the value of the expression can be computed.
18
answer
6
12
+
3
2
*
17
5
–
3
8
4
/
17
5
7
1
+
4
7
1

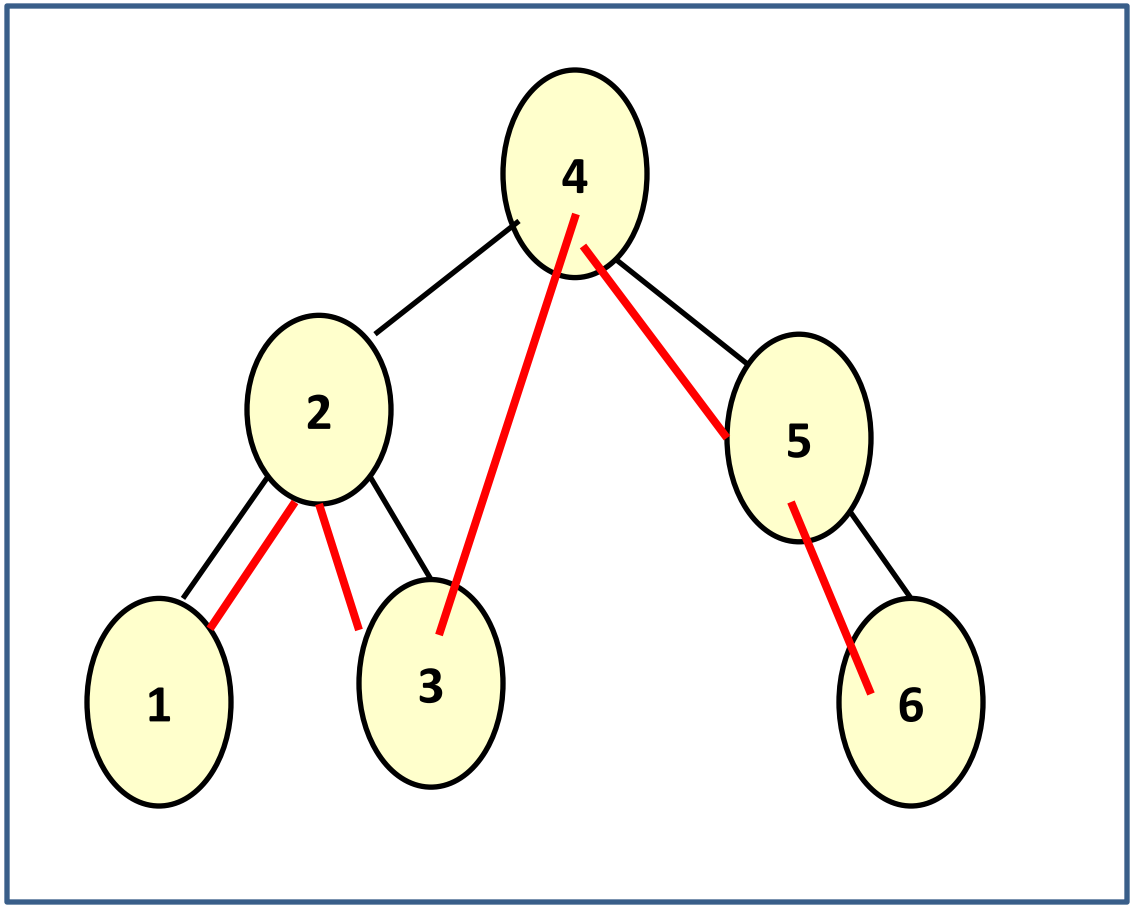
Threaded Binary tree

A **threaded binary search tree may be defined as follows:**

- A binary tree is *threaded by making all right child pointers that* would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.

- A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal.

Threaded Binary tree - Example



In-order Traversal : 1 2 3 4 5 6

Implementation –Threaded Binary tree

```
class Node {

  int data;
  Node * left, * right;
  char lflag,rflag;

public:

  Node( int data );
  int getData();
  void setRight( Node * );
  Node * getRight();
  void setLeft( Node * );
  Node * getLeft();
  void setlflag( char );
  char getlflag();
  void setrflag( char );
  char getrflag();

};
```

```
class TBinaryTree {
        Node * root;
public:
        TBinaryTree();
        void insert( int );
        void inOrder();
        void postOrder();
        void preOrder();
        void deleteData( int );
        ~TBinaryTree();
};
```

AVL tree

- An **AVL tree is a self-balancing binary search tree .**

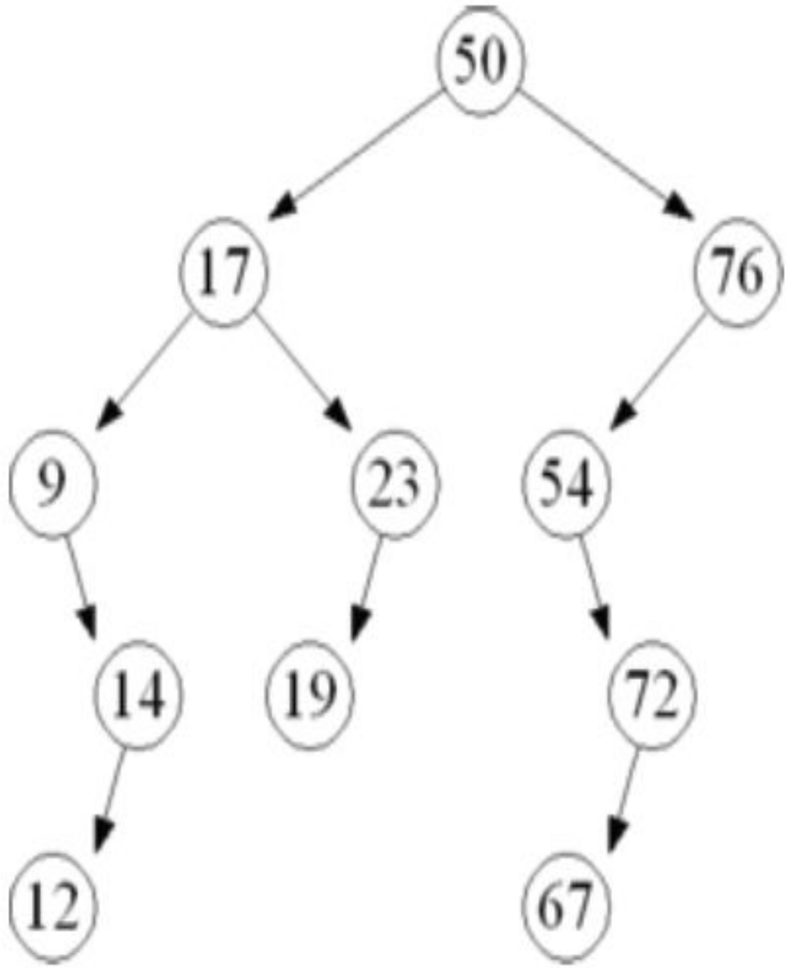
- In an AVL tree the heights of the two child sub trees of any node differ by at most one, therefore it is also called height-balanced tree.

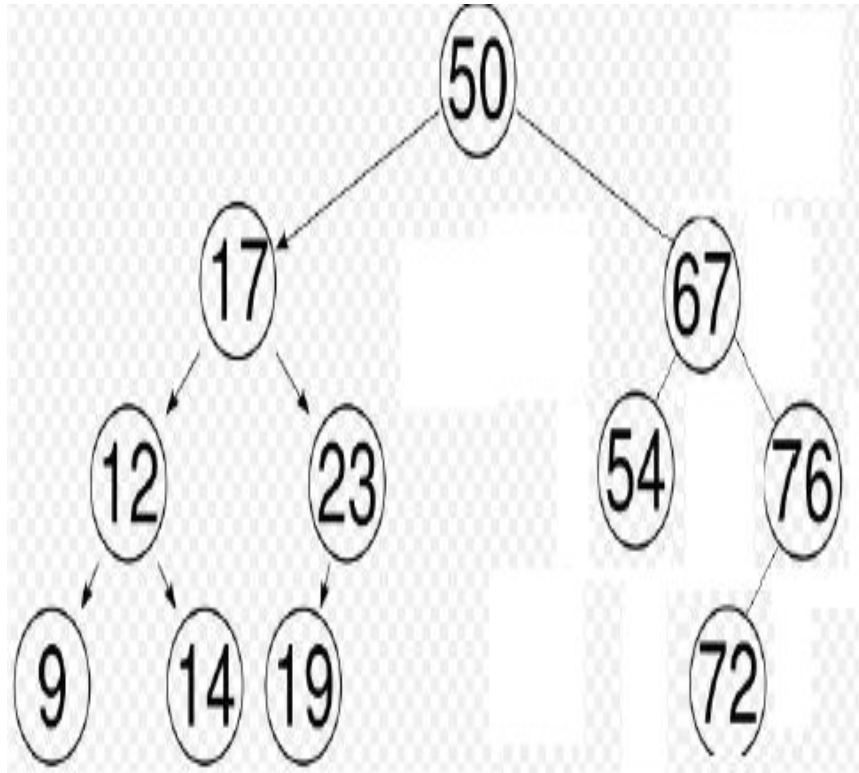
- Additions and deletions may require the tree to be rebalanced by one or more tree rotations .

- The **balance factor of a node is the height of its right sub tree** minus the height of its left sub tree.

- A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree.

Example of AVL Tree





e.g: Non AVL Tree

Insertion in Avl tree

- Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

- Retracing is stopped when a node's balance factor becomes 0,1, or -1.

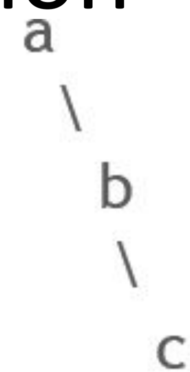
- If the balance factor becomes 0 then the

Deletion from AVL Tree

- If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node.
- After deletion retrace the path back up the tree to the root, adjusting the balance factors as needed.

Avl tree Rotation –(Left Rotation)LL Rotation

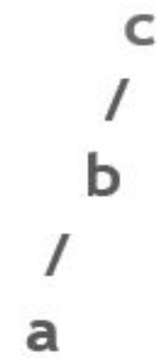
Imagine we have this situation:



- To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:
- b becomes the new root.
- a takes ownership of b's left child as its right child, or in this case, null.
- b takes ownership of a as its left chi
- The tree now looks like this:

Avl tree Rotation –(Right Rotation)RR Rotation

A right rotation is a mirror of the left rotation operation described above.
Imagine we have this situation:



- To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:
- b becomes the new root.
- c takes ownership of b's right child, as its left child. In this case, that value is null.
- b takes ownership of c, as it's righ
- The resulting tree:

The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left"**

Sometimes a single left rotation is not sufficient to balance an unbalanced tree. Take this situation:

```
a
 \
  c
```

It's balanced. Let's insert 'b'.

```
a
 \
  c
 /
b
```

Our initial reaction here is to do a single left rotation. Let's try that.

```
  c
 /
a
 \
  b
```

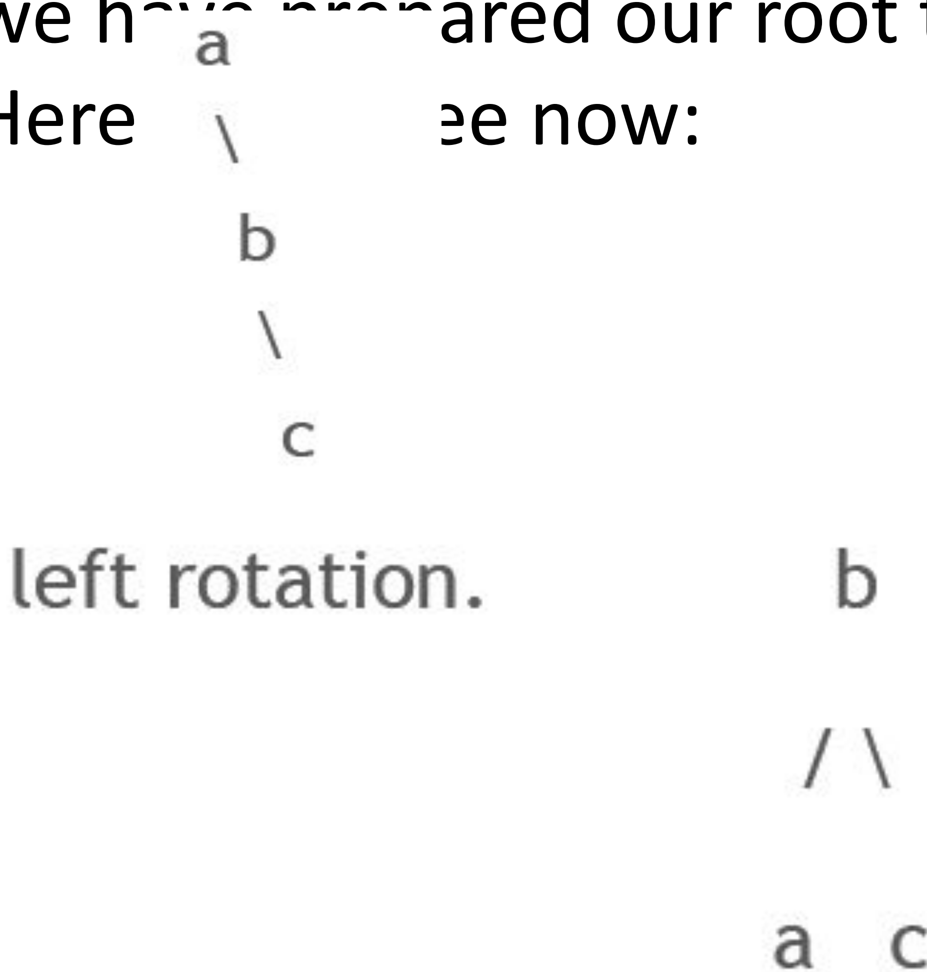
The AVL Tree Rotations **Left-Right Rotation (LR) or “Double left”**

- This is a result of the right subtree having a negative balance.
- Because the right subtree was left heavy, our rotation was not sufficient.
- The answer is to perform a right rotation on the right subtree. We are not rotating on our current root. We are rotating on our right child.
- Think of our right subtree, isolated from our main tree, and perform a right rotation on it:

```
Before:      c
            /
           b

After:       b
              \
               c
```

Rotation
(LR) or “Double left”

- After performing a rotation on our right subtree, we have prepared our root to be rotated left. Here [illegible]ee now:



Right-Left Rotation or “Double Right”

- A double right rotation, or right-left rotation, is a rotation that must be performed when attempting to balance a tree which has a left subtree, that is right heavy.
- This is a mirror operation of what was illustrated in the section on Left-Right Rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.

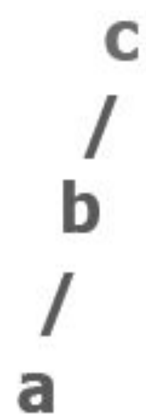
```
  c
 /
a
 \
  b
```

The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2. Some kind of right rotation is clearly necessary, but a single right rotation will not solve our problem.

```
a
 \
  c
 /
b
```

Right-Left Rotation or “Double Right”

- The reason our right rotation did not work, is because the left subtree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left subtree that is right heavy will result in the problem .
- The answer is to make our left subtree left-heavy. We do this by performing a left rotation our left subtree. Doing so leaves us with this situation:



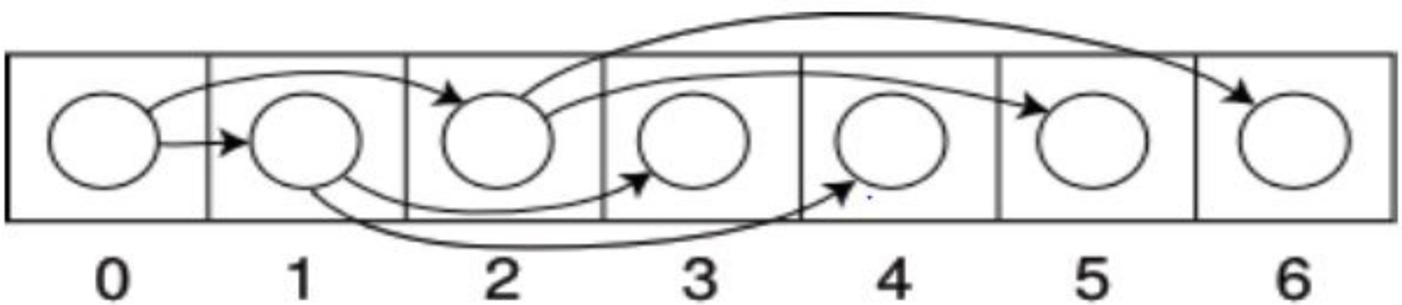
This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C
The result:

When to Rotate Tree

```
IF tree is right heavy {
    IF tree's right subtree is left heavy
      Perform Double Left rotation
    }
    else {
      Perform Single Left rotation
    }
}

Else IF tree is left heavy {
    IF tree's left subtree is right heavy {
      Perform Double Right rotation
    }
    ELSE {
      Perform Single Right rotation
```

Binary tree Representation using Array

- Binary trees can also be stored as an implicit data structure in arrays, and if the tree is a complete binary tree, this method wastes no space.
- In this compact arrangement, if a node has an index *i, its* children are found at indices 2*i + 1 and 2i + 2, while its* parent (if any) is found at index (i-1)/2 (assuming the root has index zero).
- ge
ing



However, it is expensive to grow and wastes space.

B -Tree

- In computer science, a **B-tree is a tree data structure** that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. It is most commonly used in databases and file systems.

- Each node of a b-tree may have a variable number of keys and children.

- Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceeding key.

B-Tree

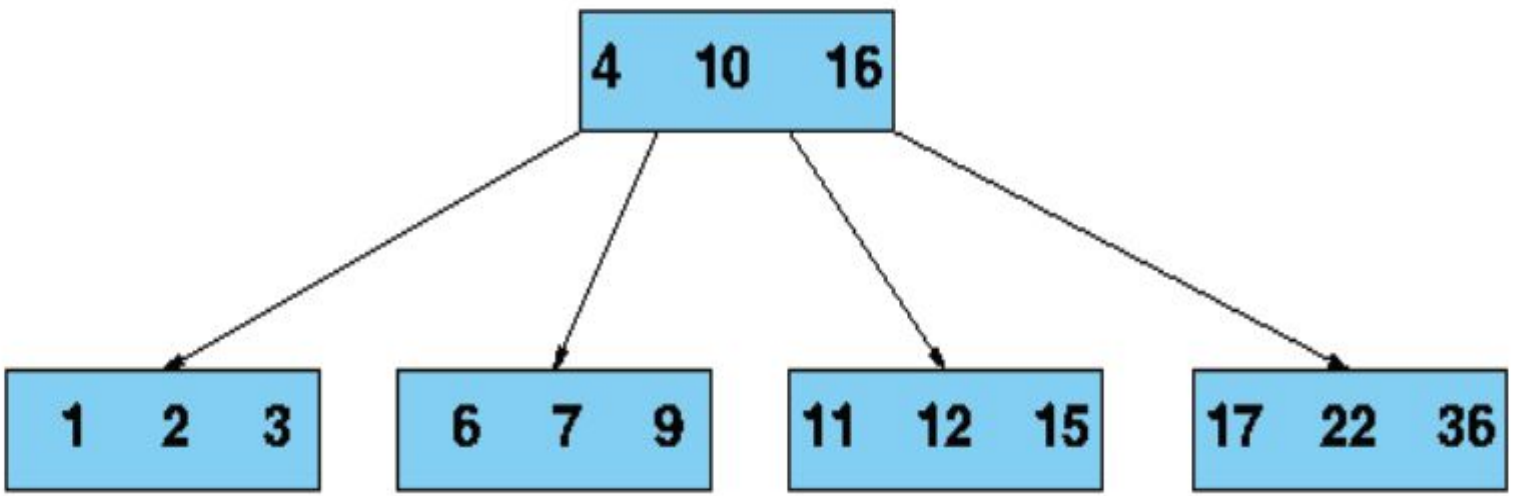
- A b-tree has a minimum number of allowable children for each node known as the *minimization factor. If t is this minimization factor, every node must have at least t – 1* keys.

- Since each node tends to have a large branching factor (a large number of children), it is typically necessary to traverse relatively few nodes before locating the desired key.

- If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required.

B-Tree

- This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).



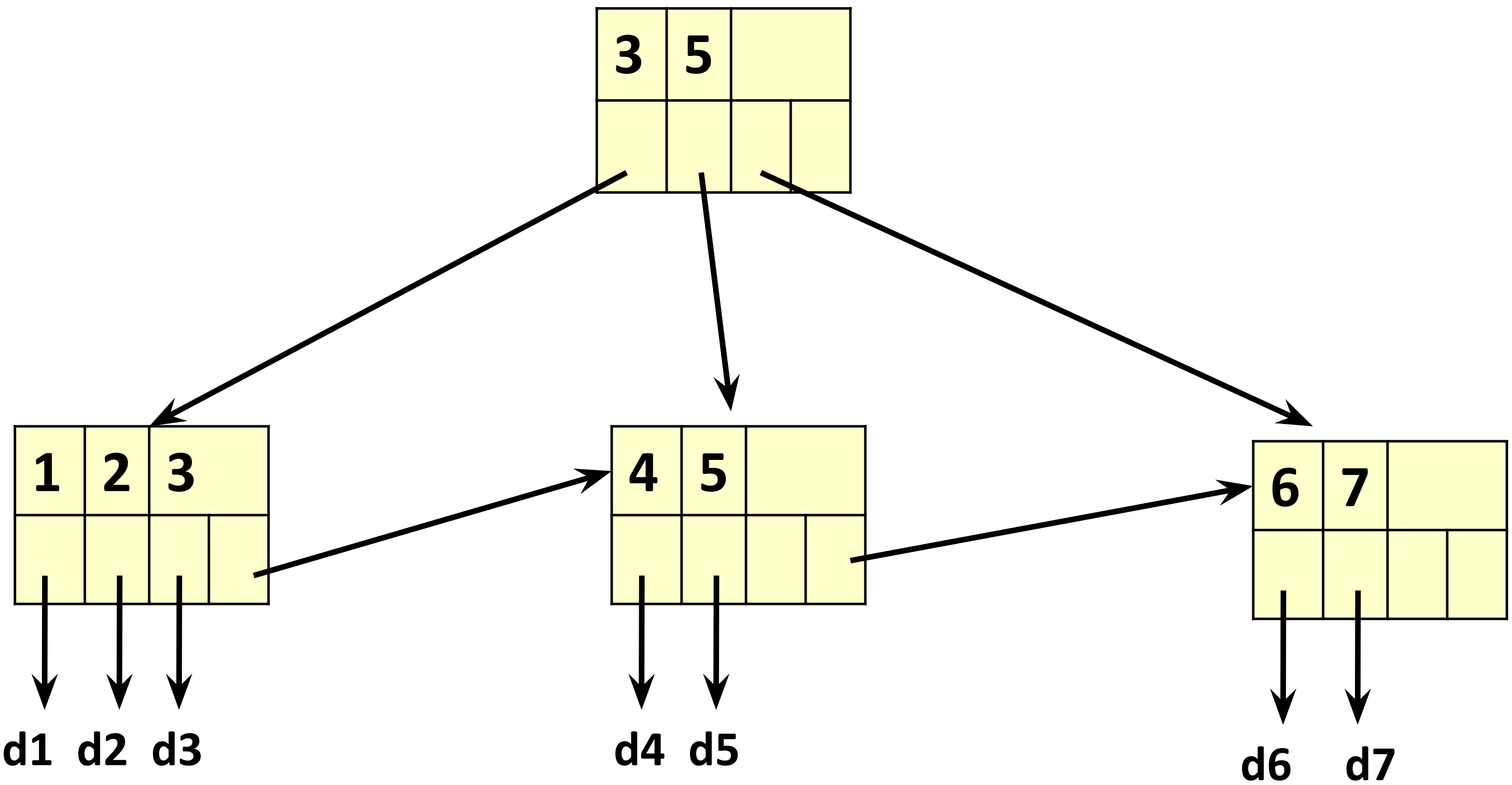
B+ Tree

- a **B+ tree (also known as a Quaternary Tree) is a type of tree,** which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a *key.*

- In a B+ tree, in contrast to a B-tree, all records are stored at the lowest level of the tree; only keys are stored in interior blocks.

- The ReiserFS filesystem (for Unix and Linux), XFS filesystem (for IRIX and Linux), JFS2 filesystem (for AIX, OS/2 and Linux) and NTFS filesystem (for Microsoft Windows) all use this type of tree for block indexing. Relational databases also often use this type of tree for table indices.

B+ Tree



Session-6
Graph

Graph

- A Graph is a collection of nodes, which are called vertices ‘V’, connected in pairs by line segments, called Edges E.

- Sets of vertices are represented as V(G) and sets of edges are represented as E(G).

- So a graph is represented as G = (V,E).

- There are two types of Graphs
  - \- Undirected Graph
  - \- Directed Graph

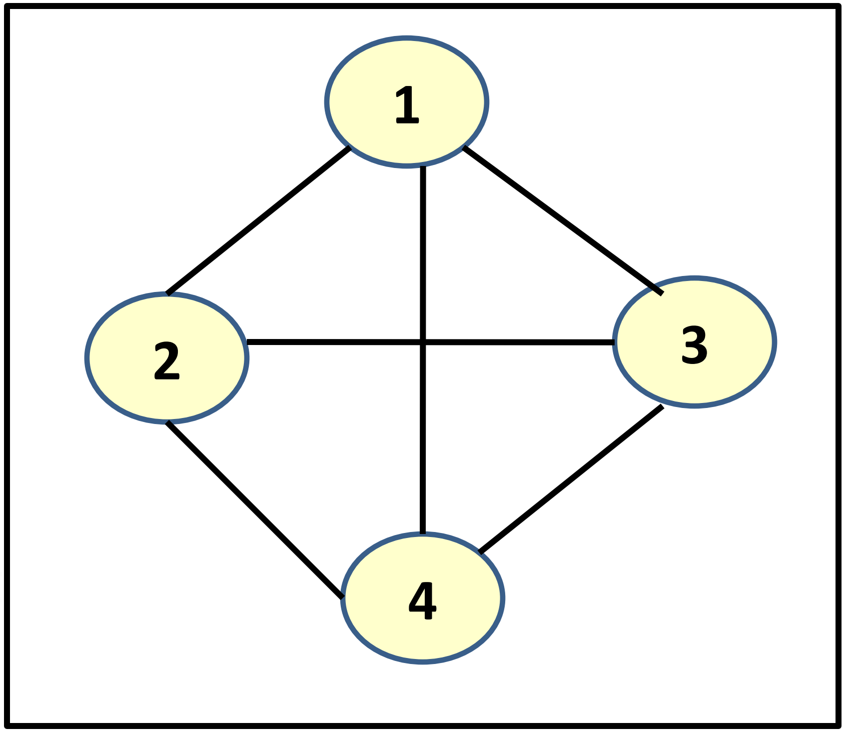
Undirected Graph

- An undirected Graph is one, where each edge E is an unordered pair of vertices. Thus pairs (v1, v2) & (v2, v1) represents the same edge.

G1

V(G1) = { 1, 2, 3, 4 };

E(G1) = { (1,2), (1,3), (1,4),
(2, 3), (2, 4), (3, 4) };



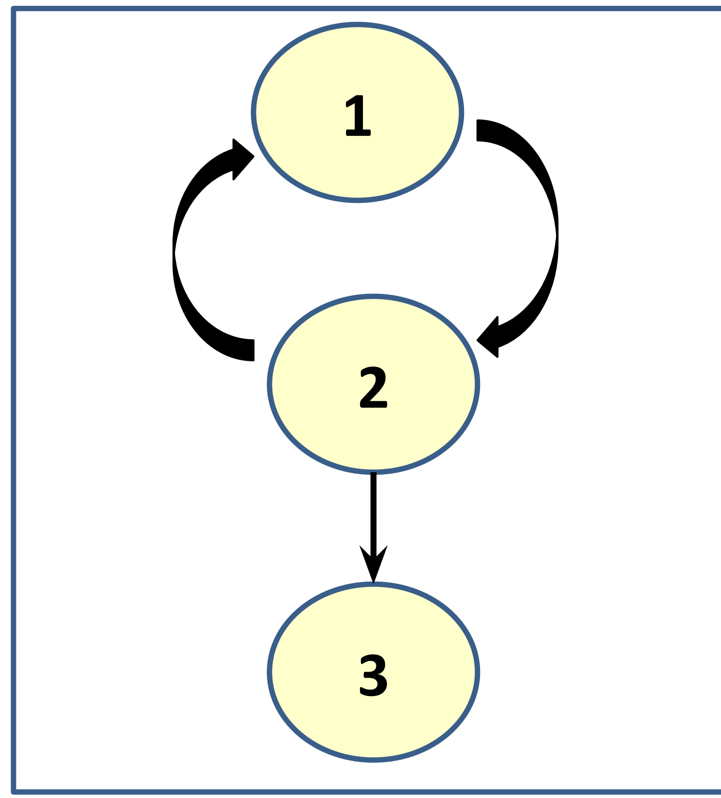
Directed Graph

- Directed Graph are usually referred as Digraph for Simplicity.
- A directed Graph is one, where each edge is represented by a specific direction or by a directed pair <v1, v2>.
- Hence <v1, v2> & <v2, v1> represents two different edges.

G2

V(G2) = { 1, 2, 3 };

E(G2) = { < 1, 2 >, < 2 , 1> , < 2, 3 > };



Graph Terminology

- **Out – degree:** The number of Arc exiting from the node is called the out degree of the node.
- **In- Degree:** The number of Arcs entering the node is the In degree of the node.
- **Sink node:** A node whose out-degree is zero is called Sink node.
- **Path:** path is sequence of edges directly or indirectly connected between two nodes.
- **Cycle:** A directed path of length at least L which originates and terminates at the same node in the graph is a cycle.

Graph Terminology

- **Adjacent:** Two vertices in an undirected Graph are called adjacent if there is an edge from the first to the second.

- **Incident:** In an Undirected graph if e=(v, w) is an edge with vertices v and w, then v and W are said to lie on e, and e is said to be incident with v and w.

- **Sub-Graph:** A sub-graph of G is G1 if
  V(G1) is subset of V(G) .
  and E(G1) is subset of E(G)

Implementing Graph – Array Method

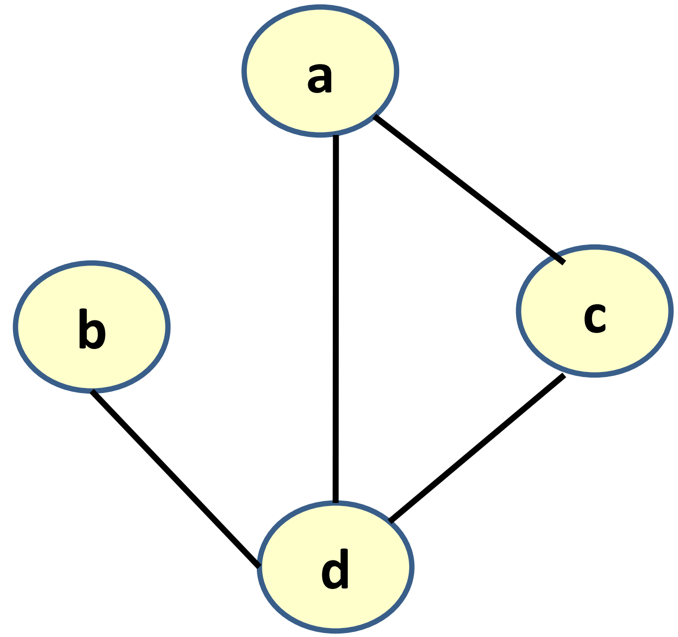
- In this an array is used to store the values of nodes.
- e.g. V[] = { a, b, c, d };
- It means that node 1 has value a, 2 has value b, 3 has value c and 4 has value d.
- To show the connectivity between nodes a two dimensional array is used.
- This matrix will have an entry ‘1’ if there is an edge between the two nodes, otherwise ‘0’.
- This matrix is known as “adjacency matrix”.
- In undirected graph this matrix will be symmetric.

Example- Graph Using Array

- Char v[] = { a, b, c, d}

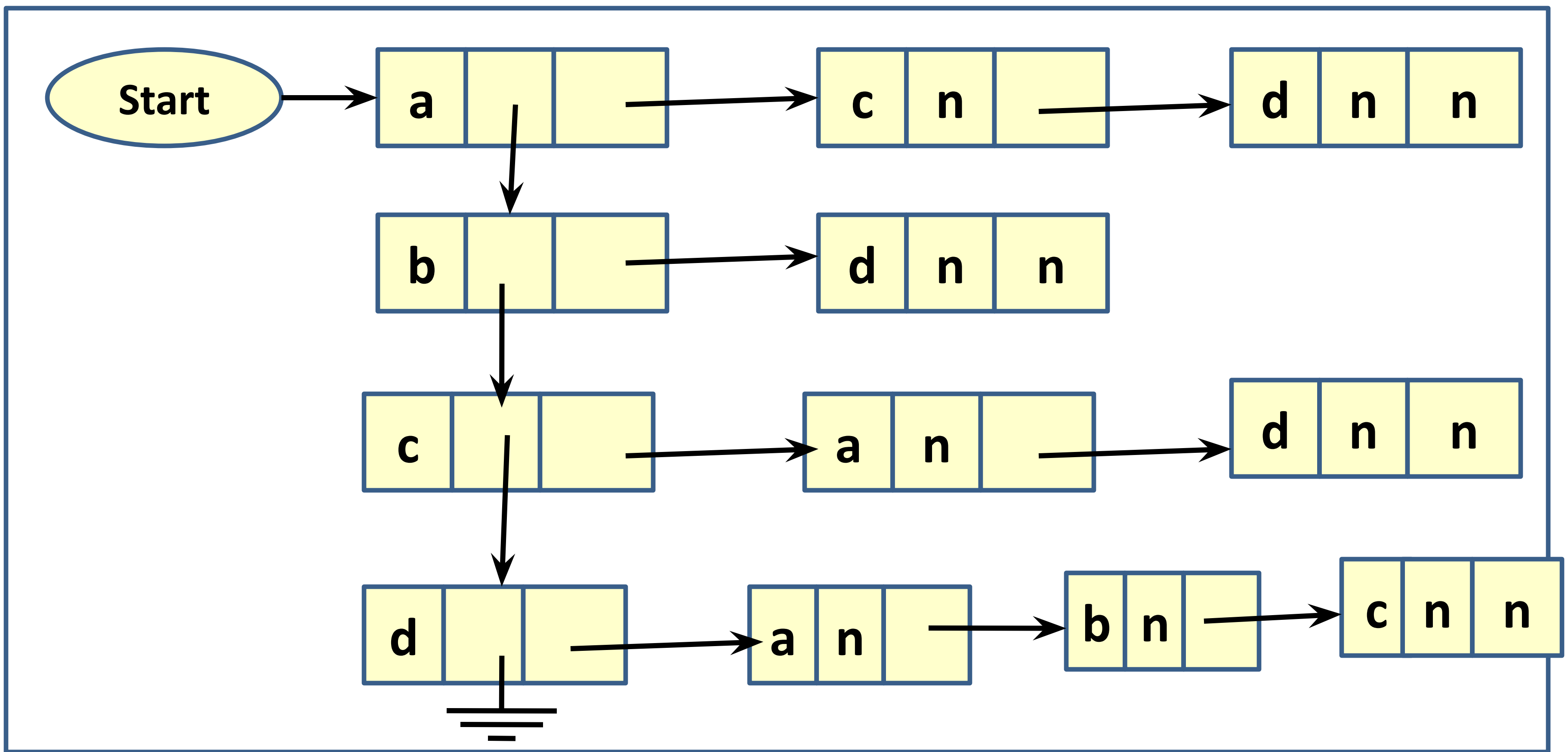
- Adj – Matrix adj[][4] =

$$
\begin{array}{c c}
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \left(\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array}\right)
\end{array}
$$



Implementation Graph- Linked List Method

- The information and edges must be stored in the node structure.
- Consider the same graph in last example.
- The linked list representation is:

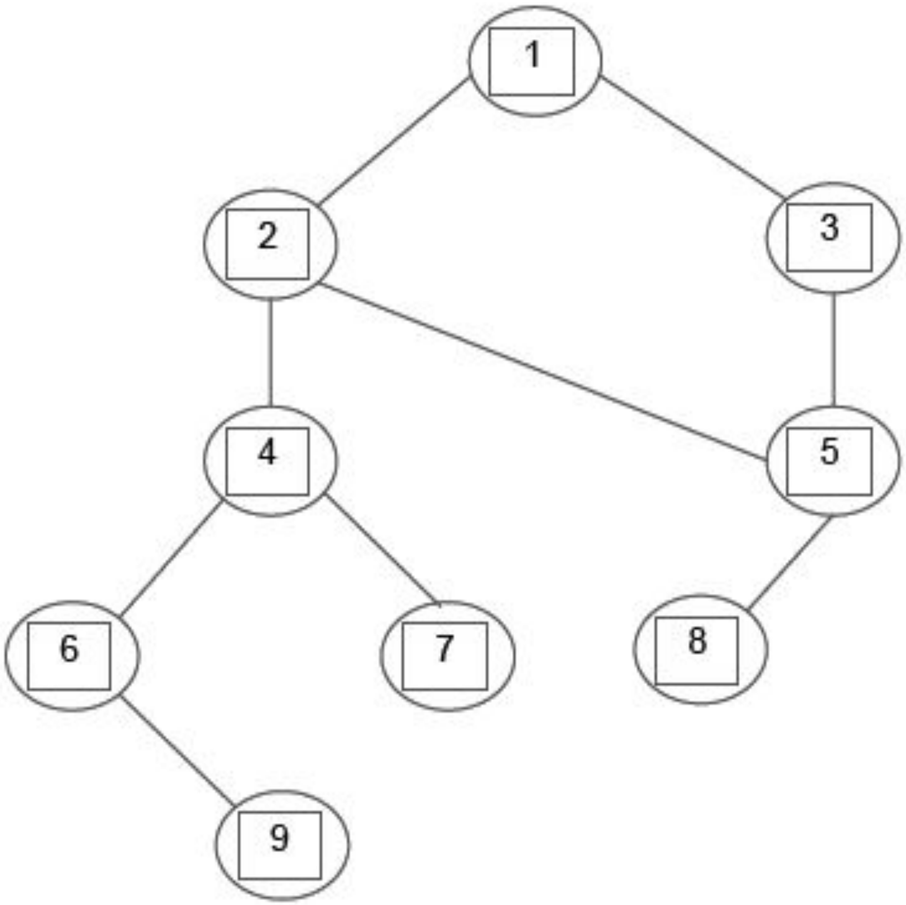


Graph traversal- Depth First Search

- Step 1 : Select any node in the graph. Visit that node. Mark this node as visited and push this node onto the stack.
- Step 2: Find the adjacent node to the node on top of the stack, and which is not yet visited. Visit this new node. Make this node as visited and push it onto the stack.
- Step 3: Repeat the step 2 until no adjacent node to the top of stack node can be found. When no new adjacent node can be found, pop the top of stack.

Depth First Search Example

- Find out DFS for following graph.



Output-

1 2 4 6 9 7 5 8 3

Graph Traversal- Breadth First Search

- Step 1 : Start with any node and mark it as visited & place it into the queue.

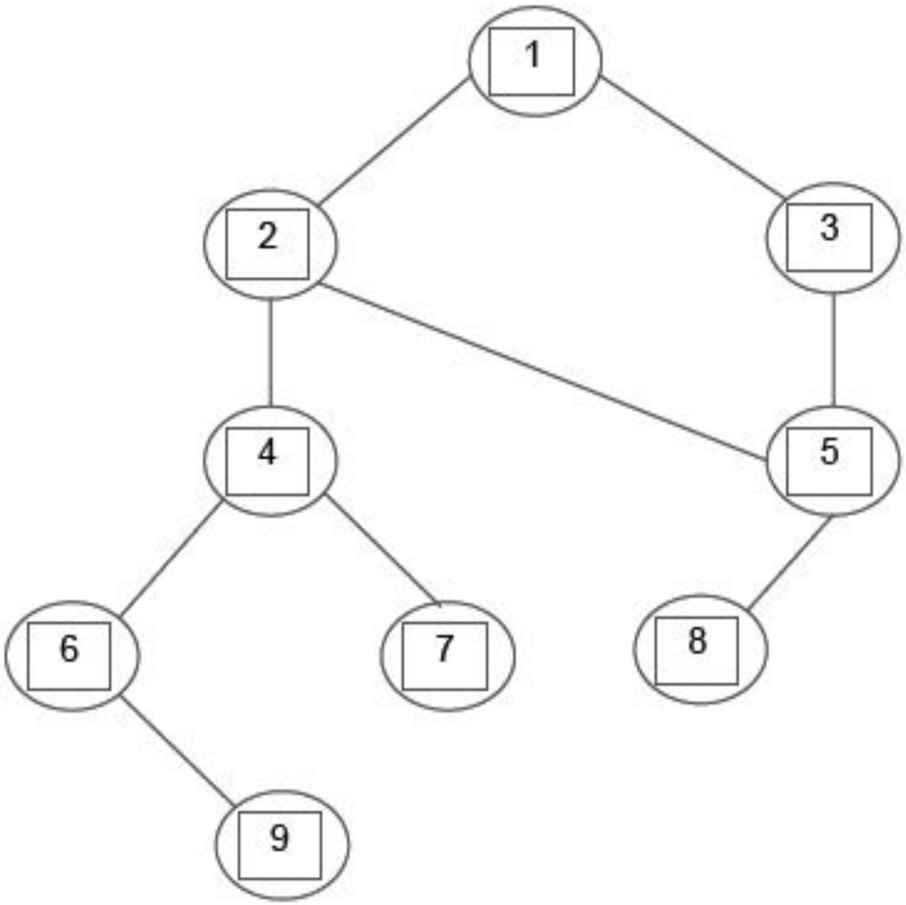
- Step 2: Find the adjacent nodes to the node marked in step 1 and place it in the queue.

- Step 3: Visit the node at the front of the queue. Delete from the queue, place it's adjacent nodes in the queue.

- Step 4 : Repeat step 3 till the queue is not empty.

Breadth First Search Example

- Find out BFS for following graph.



Output-

1 2 3 4 5 6 7 8 9

Applications of Graph

- Game theory
- Telephone networking
- Scheduling of interrelated tasks for job
- Routing from one location to another

Spanning Tree

- A Sub-graph of a graph ‘g’ is a tree containing all the nodes of ‘g’ but less number of links with minimum cost.

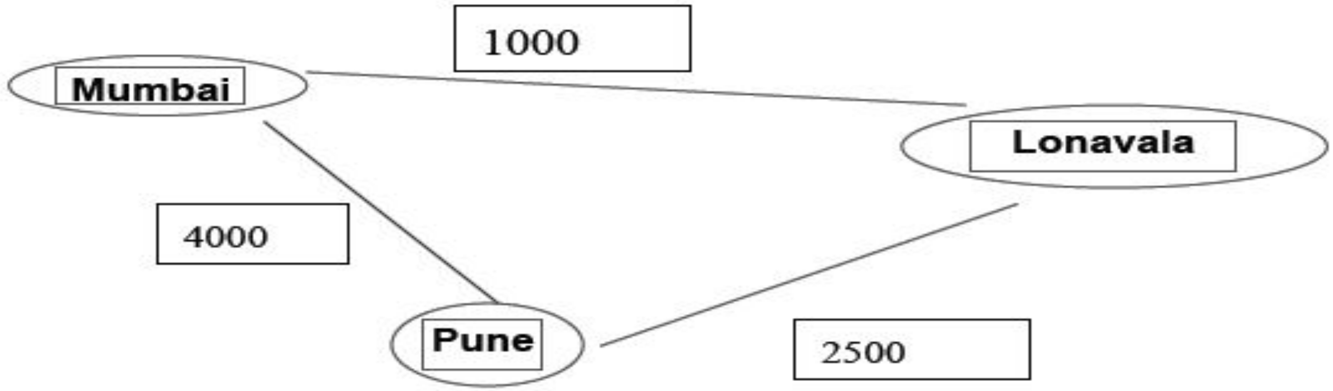
- Minimal spanning tree is a spanning tree with smallest possible weight values.

Prim’s Algorithm

- This method builds minimum spanning tree edge by edge
- Select an edge from the graph whose cost is minimum among all the edges.
- Add the next edge (i,j) to the tree such that vertex i is already in the tree & vertex j is new one & cost of the edge (i,j) is minimum among all the edges (k.l) such that k is in the tree & l is not in the tree.
- While including any edge ensure that it

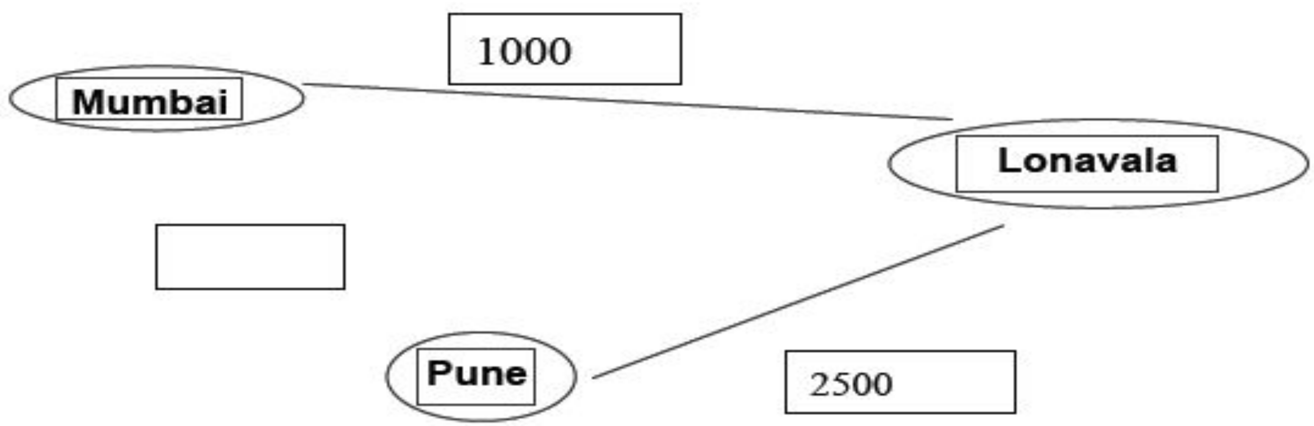
Prim’s Algorithm- Example

Find out minimum cost & spanning tree for following graph by applying Prim’s algorithm



Spanning tree using Prim’s algorithm is-

Total weight = 3500 This is also minimal spanning tree



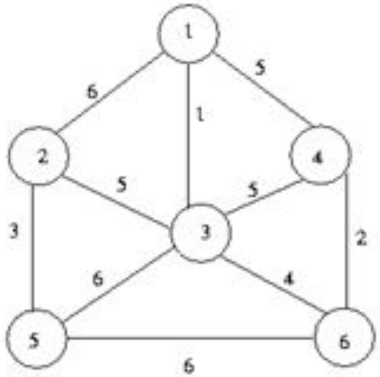
Kruskal’s Algorithm

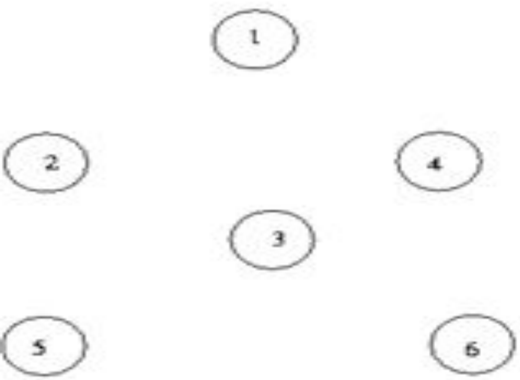
- It is another method for finding minimum cost spanning tree.
- Edges are added to the spanning tree in increasing order of cost.
- If the edge to added forms a cycle then it is discarded.
- Time complexity of algorithm:
  - n□ O(e log e) + O(e log n)

Kruskal's Algorithm

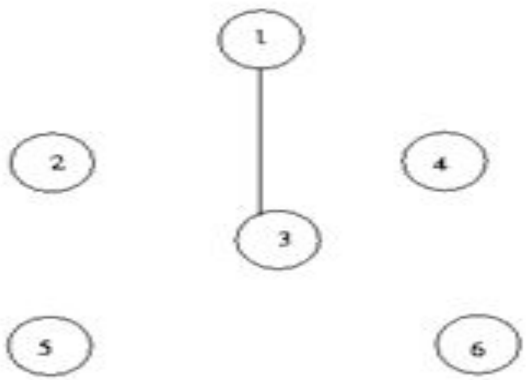
- create a forest *F (a set of trees), where each vertex in the* graph is a separate tree
- create a set *S containing all the edges in the graph*
- while *S is nonempty*
  - remove an edge with minimum weight from *S*
  - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
  - otherwise discard that edge
- At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

EXAMPLE :KRUSKAL ALGORITHM

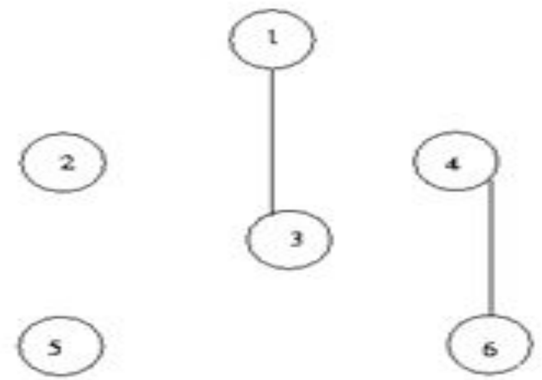




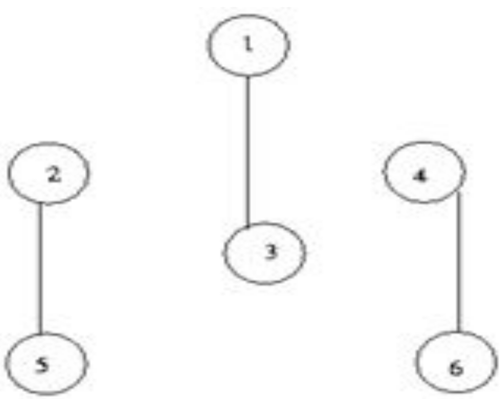
Initial Configuration



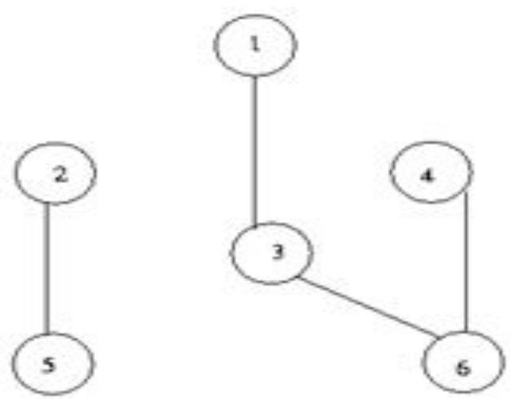
step1. choose (1,3)



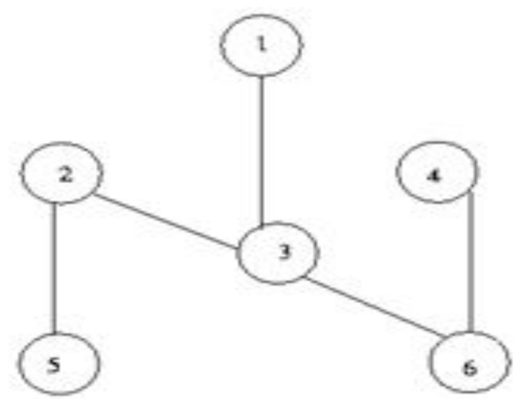
step2. choose (4,6)



step3. choose (2,5)



step4. choose (3,6)



step5. choose (2,3)

Session-4
Sorting Techniques

Sorting

- Sorting is the operation of arranging the records/elements/keys in some sequential order according to an ordering criterion.
- Ordering criterion
  - Ascending order
  - Descending order
- Sorting data is preliminary step to searching.

Types of Sorting

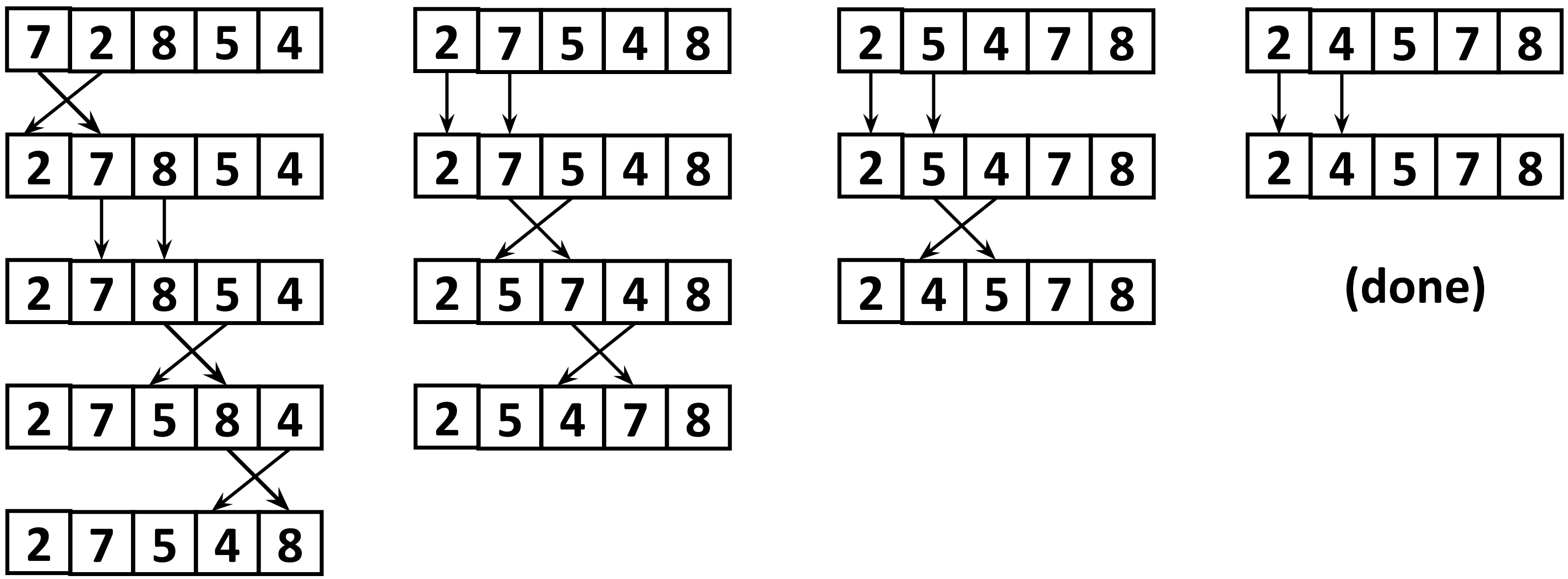
- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Radix sort
- Quick sort
- Heap Sort

Bubble Sort

- Bubble sort examines the array from start to finish, comparing elements as it goes.
- Any time it finds a larger element before a smaller element, it swaps the two.
- In this way, the larger elements are passed towards the end.
- The largest element of the array therefore "bubbles" to the end of the array.
- Then it repeats the process for the unsorted portion of the array until the whole array is sorted.

- Algorithm-
- two records are compared & interchanged immediately upon discovering that they are out of order
- This method will cause records with small keys to move or "bubble up“
- After the first pass, the record with the largest key will be in the nth position

Example of bubble sort



Code for bubble sort

```
○ public static void bubbleSort(int[] a) {
    int outer, inner;
    for (outer = a.length - 1; outer > 0; outer--) {  // counting down
      for (inner = 0; inner < outer; inner++) {    // bubbling up
        if (a[inner] > a[inner + 1]) {  // if out of order...
          int temp = a[inner];         // ...then swap
          a[inner] = a[inner + 1];
          a[inner + 1] = temp;
        }
      }
    }
  }
```

Selection Sort

- Select the highest one and put it where it belongs.
- Select the second highest and place it in order.
- Do this for all the keys to be sorted.

Algorithm-

1. Define the entire array as the unsorted portion of the array
2. While the unsorted portion of the array has more than one element:
   $\Rightarrow$ Find its largest element.
   $\Rightarrow$ Swap with last element (assuming their values are different).
   $\Rightarrow$ Reduce the size of the unsorted portion of the array by 1.

Example of Selection Sort:

Before sorting	14	2	10	5	1	3	17	7
After pass 1	14	2	10	5	1	3	7	17
After pass 2	7	2	10	5	1	3	14	17
After pass 3	7	2	3	5	1	10	14	17
After pass 4	1	2	3	5	7	10	14	17

Code for Selection Sort

```
public static void selectionSort(int[] a) {
      int outer, inner, min;
      for (outer = 0; outer < a.length - 1; outer++) {
         min = outer;
         for (inner = outer + 1; inner < a.length; inner++) {
            if (a[inner] < a[min]) {
               min = inner;
            }
            // Invariant: for all i, if outer <= i <= inner, then
  a[min] <= a[i]
         }
         // a[min] is least among a[outer]..a[a.length - 1]
         int temp = a[outer];
         a[outer] = a[min];
         a[min] = temp;
         // Invariant: for all i <= outer, if i < j then a[i] <= a[j]
      }
  }
```

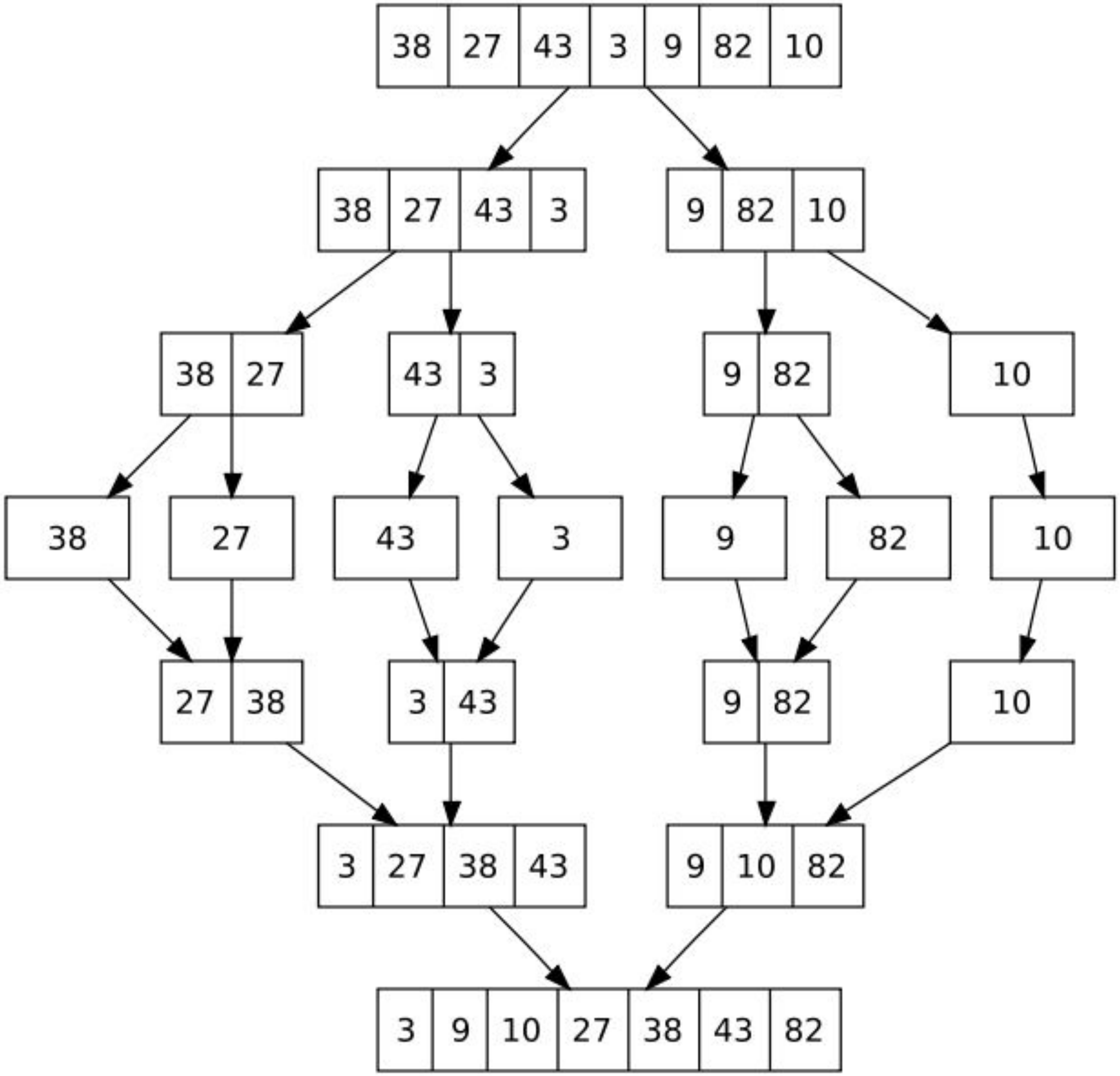
Insertion Sort

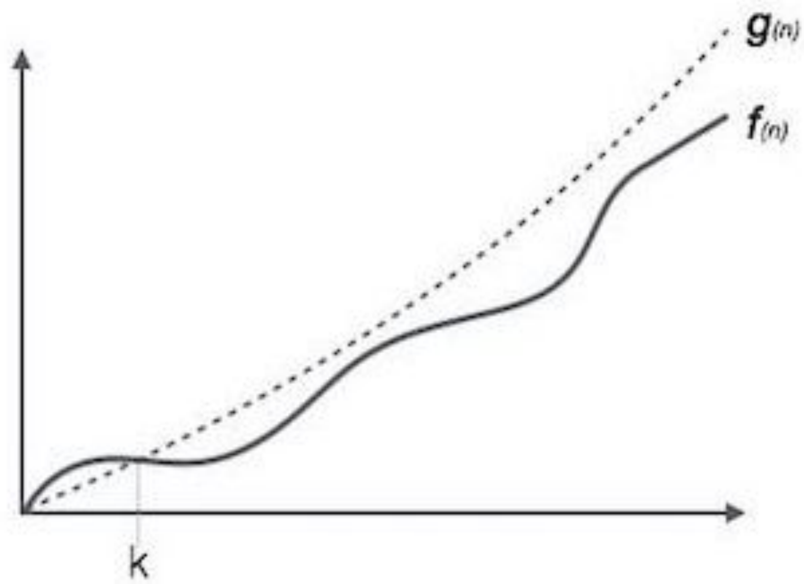
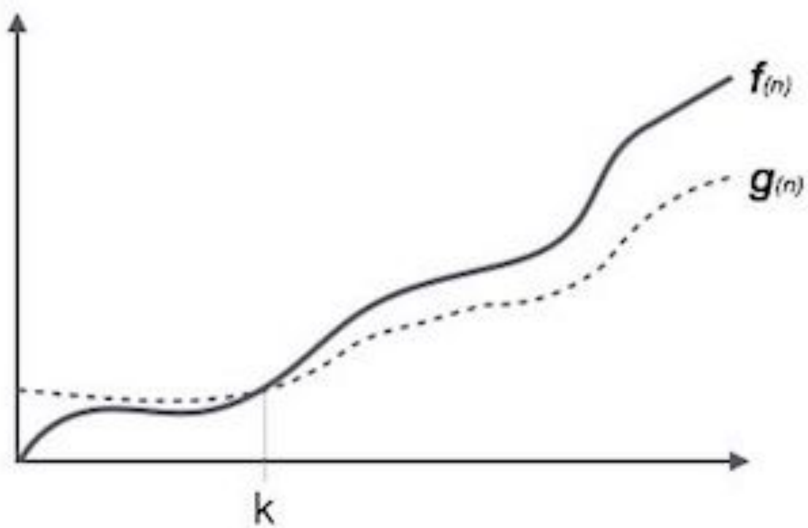
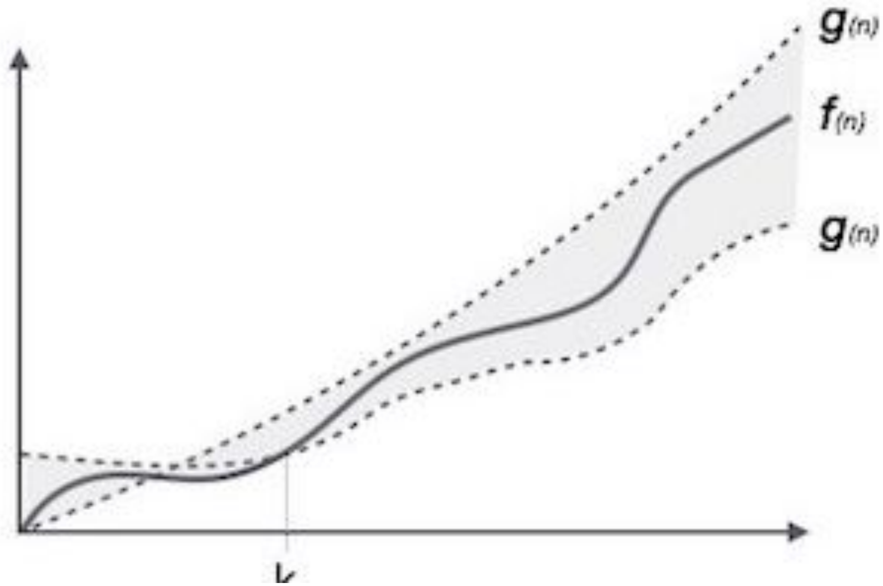
- This makes use of the fact the keys are partly ordered in the entire set of keys.
- One key from the unordered array is selected and compared with the keys which are ordered.
- The correct location of this key would be when a key greater than this particular key is found.
- The keys from that key onwards in the ordered array are shifted one position to the right.
- The hole developed is then filled in by the key from the unordered array.
- This is repeated for all keys in the unordered array

Example of Insertion Sort-

- Original 34 8 64 51 32 21 Positions
- Moved
- After p = 2 8 34 64 51 32 21 1
- After p = 3 8 34 64 51 32 21 0
- After p = 4 8 34 51 64 32 21 1
- After p = 5 8 32 34 51 64 21 3
- After p = 6 8 21 32 34 51 64 4

Code for Insertion Sort

- ```
  void insertion(int arr[], int size)
  { int i,j,tmp;
   for(i=0; i<size; i++)
   {
     for(j=i-1; j>=0; j--)
     {
      if(arr[j]>arr[j+1])
      {
        tmp=arr[j];
        arr[j]=arr[j+1];
        arr[j+1]=tmp;
      }
      else
        break;
     }
   }
  }
  ```
  ```

Merge Sort

- Merging is a process of combining two or more sorted arrays/files into third sorted array/file
- In merge sort the first elements in both tables are compared & the smallest key is then stored in a third table.
- This process is repeated till end of the both arrays
- In this way two sorted tables are merged to form third sorted table

Example of Merge Sort



Code for Merge Sort-

```
void part(int arr[],int min,int max)
    { int mid;
     if(min<max)
     {
       mid=(min+max)/2;
       part(arr,min,mid);
       part(arr,mid+1,max);
       merge(arr,min,mid,max);
     }
    }
```

```
void merge(int arr[],int min,int mid,int max)
{  int tmp[30]; int i,j,k,m;
  j=min;
  m=mid+1;
  for(i=min; j<=mid && m<=max ; i++)
  {    if(arr[j]<=arr[m])
     {   tmp[i]=arr[j];
         j++;}
     else
     {   tmp[i]=arr[m];
         m++;
     }  }
  if(j>mid)
  {   for(k=m; k<=max; k++)
     {   tmp[i]=arr[k];
         i++;
     }   }
  else
  {     for(k=j; k<=mid; k++)
     { tmp[i]=arr[k];
        i++;
}  }
  for(k=min; k<=max; k++)
     arr[k]=tmp[k];
}
```

Quick Sort Terminology

- Pivot :The key whose exact location is to be found in the sorted array.
- Keys to the left of pivot are smaller than the pivot and keys to the right of pivot are greater than the pivot.
- This is known as partitioning.
- Algorithm-

Quick Sort(A, lb,up)
If (lb >= ub) then return
If lb < ub then pivot_loc = partition (A, lb,ub)
Quick_sort(A, lb, pivot_loc)
Quick_sort(A, pivot_loc+1, ub)

Quick Sort – Partition Algorithm

- Partition algorithm
- Let down = lower bound of array and up = upper bound of array, pivot = A[lb]
- Repeatedly increase the pointer down by one position while A[down] < pivot
- Repeatedly decrease pointer up by one position while A[up] >= pivot
- If up > down then interchange A[down] with A[up]
- If down < up goto 2
- Else interchange pivot, A[up]
- Pivot_loc = up
- Return pivot_loc

Example – Quick Sort

- **Example:** 8, 3, 25, 6, 10, 17, 1, 2, 18, 5
- The first element is 8, the middle - 10, the last - 5. The median of [8, 10, 5] is **8**

- **STEP 2. Partitioning**
- Partitioning is illustrated on the above example.
- 1. The first action is to get the pivot out of the way - swap it with the last element **5**, 3, 25, 6, 10, 17, 1, 2, 18, **8**
- 2. We want larger elements to go to the right and smaller elements to go to the left.

Two "fingers" are used to scan the elements from left to right and from right to left: [5, 3, 25, 6, 10, 17, 1, 2, 18, 8] ^ ^ i j While **i** is to the left of **j**, we move **i** right, skipping all the elements less than the pivot. If an element is found greater then the pivot, **i** stops.

- While **j** is to the right of **i**, we move **j** left, skipping all the elements greater than the pivot. If an element is found less then the pivot, **j** stops
- When both **i** and **j** have stopped, the elements are swapped.
- When **i** and **j** have crossed, no swap is performed, scanning stops, and the element pointed to by **i** is swapped with the pivot .
- In the example the first swapping will be between 25 and 2, the second between 10 and 1.
- 3. Restore the pivot.

After restoring the pivot we obtain the following partitioning into three groups: [5, 3, 2, 6, 1] [8] [10, 25, 18, 17]

- **STEP 3. Recursively quicksort the left and the right parts**

Code for Quick Sort-

```
void quicksort(int x[10],int first,int last){
    int pivot,j,temp,i;
     if(first<last){
         pivot=first;
         i=first;
         j=last;
         while(i<j){
             while(x[i]<=x[pivot]&&i<last)
                 i++;
             while(x[j]>x[pivot])
                 j--;
             if(i<j){
                 temp=x[i];
                  x[i]=x[j];
                  x[j]=temp;
             }          }
         temp=x[pivot];
         x[pivot]=x[j];
         x[j]=temp;
         quicksort(x,first,j-1);
         quicksort(x,j+1,last);
    }
}
```

Radix Sort

- In radix sort method the given set of unsorted numbers are compared on the basis of columns (digit place like units tens hundreds etc).
- Each comparison through the entire set is termed as Pass.
- The number of passes required to sort the given set of numbers depends on the number of digits of the largest number.
- After each pass the number are placed in the respective pockets.

Radix Sort-

- 42, 23, 74, 11, 65, 57, 94, 36, 99, 87, 70, 81, 61
- After the first pass on the unit digit position of each number we have:

		61								
		81			94			87		
	70	11	42	23	74	65	36	57		99
poc	0	1	2	3	4	5	6	7	8	9

Radix Sort

- combining the contents of the pockets so that the contents of the "0" pocket are on the bottom and the contents of the "9“ pocket are on the top, we obtain:
- 70, 11, 81, 61, 42, 23, 74, 94, 65, 36, 57, 87, 99

							61	70	81	94
		11	23	36	42	57	65	74	87	99
Poc	0	1	2	3	4	5	6	7	8	9

Comparison of Algorithm

	Time				
Sort	Average	Best	Worst	Space	Stability
Bubble sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable
Modified Bubble sort	O(n^2)	O(n)	O(n^2)	Constant	Stable
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Constant	Instable
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Depends	Stable
Quicksort	O(n*log(n))	O(n*log(n))	O(n^2)	Constant	Stable

Complexity of Algorithm

- In general the *complexity of an algorithm is* the amount of time and space (memory use) required to execute it.
- Since the actual time required to execute an algorithm depends on the details of the program implementing the algorithm and the speed and other characteristics of the machine executing it, it is in general impossible to make an estimation in actual physical time
- however it is possible to measure the length of the computation in other ways, say by the number of operations performed.

Complexity of Algorithm

- the following loop performs the statement x := x + 1 exactly *n times,*

for i := 1 to n do

 x := x + 1

- The following double loop performs it *n2 times:*

for i := 1 to n do

 for j := 1 to n do

  x := x + 1

- The following one performs it 1 + 2 + 3 + *· · · + n =n(n + 1)/2 times:*

for i := 1 to n do

 for j := 1 to i do

  x := x + 1

Complexity of Algorithm

- Since the time that takes to execute an algorithm usually depends on the input, its complexity must be expressed as a function of the input, or more generally as a function of the *size of the input.*
- Since the execution time may be different for inputs of the same size, we define the following kinds of times:

- **Best Case Time:** minimum time needed to execute the algorithm among all inputs of a given size *n.*

- **Worst Case Time:** maximum time needed to execute the algorithm among all inputs of a given size *n.*

- **Average Case Time:** average time needed to execute the algorithm among all inputs of a given size *n.*

Session-1
Algorithm

Algorithm

- An *algorithm* specifies a series of steps that perform a particular computation or task.

- **“a *finite* sequence of *unambiguous*, *executable* steps or instructions, which, if followed would ultimately *terminate* and give the solution of the problem”.**

- Properties of algorithm-
  - An algorithm is an unambiguous description that makes clear what has to be implemented.
  - An algorithm expects a defined set of inputs.
  - An algorithm produces a defined set of outputs.
  - Most algorithms are guaranteed to produce the correct result.
  - If an algorithm imposes a requirement on its inputs (called a *precondition*), that requirement must be met.

Express An Algorithm

- You can express an algorithm –
  - English
  - Flow Chart (Schematic Diagrams)
  - Pseudo-Code
  - Program

Characteristic of Algorithm

An algorithm should have the below mentioned characteristics –

- Unambiguous
- Input
- Output
- Finiteness
- Feasibility
- Independent

Analysis of Algorithm

- We only analyze *correct* algorithms

- An algorithm is correct
  - If, for every input instance, it halts with the correct output

- Incorrect algorithms
  - Might not halt at all on some input instances
  - Might halt with other than the desired answer

- Analyzing an algorithm
  - **Predicting** the resources that the algorithm requires
  - Resources include
    - Memory
    - Communication bandwidth
    - Computational time (usually most important)

Analysis of Algorithm

- Factors affecting the running time
  - computer
  - compiler
  - algorithm used
  - input to the algorithm
    - The content of the input affects the running time
    - typically, the *input size* (number of items in the input) is the main consideration
      - E.g. sorting problem ⇒ the number of items to be sorted
      - E.g. multiply two matrices together ⇒ the total number of elements in the two matrices

- Machine model assumed
  - Instructions are executed one after another, with no concurrent operations ⇒ Not parallel computers

Example Of Algorithm Analysis

- Calculate

$$\sum_{i=1}^{N} i^3$$

```
int sum(int n)
{
    int partialSum;

    partialSum=0;                  1
    for (int i=1;i<=n;i++)         2N+2
        partialSum += i*i*i;       4N
    return partialSum;             1
}
```

- Lines 1 and 4 count for one unit each
- Line 3: executed N times, each time four units
- Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2
- total cost: 6N + 4 ⇒ O(N)

Algorithm Complexity

- Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor** − The time is measured by counting the number of key operations such as comparisons in sorting algorithm

- **Space Factor** − The space is measured by counting the maximum memory space required by the algorithm.

- The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

- Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components −

  – A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.

  – A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.

- Space complexity **S(P)** of any algorithm **P** is **S(P) = C + SP(I)** Where **C** is the fixed part and **S(I)** is the variable part of the algorithm which depends on instance characteristic **I**.

Time Complexity

- Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function **T(n)**, where **T(n)** can be measured as the number of steps, provided each step consumes constant time.

Asymptotic Analysis of Algorithm

- Asymptotic analysis of an algorithm, refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.

- Asymptotic analysis are input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.

- Usually, time required by an algorithm falls under three types −
  - **Best Case** − Minimum time required for program execution.
  - **Average Case** − Average time required for program execution.
  - **Worst Case** − Maximum time required for program execution.

Asymptotic Notations-**Big Oh Notation, O**

- The O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.



For example, for a function f(n)

O(f(n)) = { g(n) : there exists c > 0 and n_0 such that g(n) ≤ c.f(n) for all n > n_0. }

Asymptotic Notations-**Omega Notation, Ω**

- The Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.



For example, for a function f(n)

$\Omega(f(n)) \geq \{\, g(n) :$ there exists $c > 0$ and n_0 such that $g(n) \leq c.f(n)$ for all $n > n_0$. $\}$

- The θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following −



$\theta(f(n))$ = { $g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0$. }

Algorithm Design Techniques

- General approaches to the construction of efficient solutions to problems.

- Such methods are of interest because:
  - They provide templates suited to solving a broad range of diverse problems.
  - They can be translated into common control and data structures provided by most high-level languages.
  - The temporal and spatial requirements of the algorithms which result can be precisely analyzed.
  - Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.

Recursive Algorithm

- A simple recursive algorithm:
  - Solves the base cases directly
  - Recurs with a simpler subproblem
  - Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem

Example-

- To count the number of elements in a list:
  - If the list is empty, return zero; otherwise,
  - Step past the first element, and count the remaining elements in the list
  - Add one to the result

- To test if a value occurs in a list:
  - If the list is empty, return false; otherwise,
  - If the first thing in the list is the given value, return true; otherwise
  - Step past the first element, and test whether the value occurs in the remainder of the list

Brute Force

- **Brute force** is a straightforward approach to solve a problem based on the problem’s statement and definitions of the concepts involved.

- It is considered as one of the easiest approach to apply and is useful for solving small – size instances of a problem.

Example

- Computing a^n ($a > 0$, n a nonnegative integer) by multiplying a*a*…*a
- Computing n!
- Selection sort
- Bubble sort
- Sequential search
- Exhaustive search: Traveling Salesman Problem,
- Knapsack problem.

Greedy

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A “greedy algorithm” sometimes works well for optimization problems
- A greedy algorithm works in phases: At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example

- Minimal spanning tree
  - Prim's
  - Kruskal's
  - Dijkstra's
- Shortest distance in graphs

- Greedy algorithm for the Knapsack problem

- The coin exchange problem

- Huffman trees for optimal encoding

Divide-&-Conquer, Decrease-&-Conquer

- A divide and conquer algorithm consists of two parts:
  - Divide the problem into smaller sub-problems of the same type, and solve these sub-problems recursively
  - Combine the solutions to the sub-problems into a solution to the original problem
- Traditionally, an algorithm is only called divide and conquer if it contains two or more recursive calls

Example

- **Examples of divide-and-conquer algorithms**:
- Computing a^n ($a > 0$, n a nonnegative integer) by recursion
- Binary search in a sorted array (recursion)
- Mergesort algorithm,
- Quicksort algorithm (recursion)
- The algorithm for solving the fake coin problem (recursion)

- **Examples of decrease-and-conquer algorithms**:
- Insertion sort
- Topological sorting
- Binary Tree traversals: inorder, preorder and postorder (recursion)
- Computing the length of the longest path in a binary tree (recursion)
- Computing Fibonacci numbers (recursion)
- Reversing a queue (recursion)
- Warshall’s algorithm (recursion)

Back-Tracking

- Backtracking algorithms are based on a depth-first recursive search
- A backtracking algorithm:
  - Tests to see if a solution has been found, and if so, returns it; otherwise
  - For each choice that can be made at this point,
    - Make that choice
    - Recur
    - If the recursion returns a solution, return it
  - If no choices remain, return failure

Example

- The following problems can be solved using state-space search techniques:
- A farmer has to move a goat, a cabbage and a wolf from one side of a river to the other side using a small boat. The boat can carry only the farmer and one more object (either the goat, or the cabbage, or the wolf). If the farmer leaves the goat with the wolf alone, the wolf would kill the goat. If the goat is alone with the cabbage, it will eat the cabbage. How can the farmer move all his property safely to the other side of the river?"
- You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?
- DFS in Tree

Dynamic Programming

- A dynamic programming algorithm remembers past results and uses them to find new results
- Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the “best” one
  - Requires “optimal substructure” and “overlapping subproblems”
    - Optimal substructure: Optimal solution contains optimal solutions to subproblems
    - Overlapping subproblems: Solutions to subproblems can be stored and reused in a bottom-up fashion
- This differs from Divide and Conquer, where subproblems generally need not overlap

Example

- Fibonacci numbers computed by iteration.

- Warshall’s algorithm implemented by iterations

Branch and Bound

- Branch and bound algorithms are generally used for optimization problems
  - As the algorithm progresses, a tree of subproblems is formed
  - The original problem is considered the "root problem"
  - A method is used to construct an upper and lower bound for a given problem
  - At each node, apply the bounding methods
    - If the bounds match, it is deemed a feasible solution to that particular subproblem
    - If bounds do *not* match, partition the problem represented by that node, and make the two subproblems into children nodes
  - Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed

Example

- Travelling salesman problem

- the 8-puzzle problem. The cost function is the number of moves. The utility function evaluates how close is a given state of the puzzle to the goal state, e.g. counting how many tiles are not in place.

Randomized

- A randomized algorithm uses a random number at least once during the computation to make a decision
  - Example: In Quicksort, using a random number to choose a pivot
  - Example: Trying to factor a large prime by choosing random numbers as possible divisors

Session-4
Searching Algorithm

Searching technique

- Sequential Search
- Binary Search

Sequential Search

- **sequential search, also known as linear search is** suitable for searching a set of data for a particular value.
- It operates by checking every element of a list one at a time in sequence until a match is found.
- If the data are distributed randomly, on average N/2 comparisons will be needed.
- The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed.
- The worst case is that the value is not in the list (or is the last item in the list), in which case N comparisons are needed.
- The simplicity of the linear search means that if just a few elements are to be searched it is less trouble than more complex methods that require preparation such as sorting the list to be searched or more complex data structures, especially when entries may be subject to frequent revision.
- Another possibility is when certain values are much more likely to be searched for than others and it can be arranged that such values will be amongst the first considered in the list.

Implementation Snippet

```
for( int I = 0; I < n; i++ )
{
    if( arr[ I ] == key )
     break;
}

if( I == n )
    return not found;
else
    return found;
```

Binary Search

- A **binary search algorithm is a technique for finding a** particular value in a linear array, by ruling out half of the data at each step.
- A binary search finds the median, makes a comparison to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner.
- A binary search is an example of a divide and conquer algorithm.
- The entries in the table are sorted in increasing order.
- The approximate middle entry of the array is located, and it's key value is examined.
- If it's value is high, then the key value is compared with the middle half of the first half and the procedure is repeated on the first half until the required item is found.
- If the value is too low, then the key value is compared with the middle entry of second half and the procedure is repeated until the required item is found.
- The process continues until the desired key is found or the search interval becomes empty.

Implementation Algo

- ```
  BinarySearch( A, value ) {
  low = 0 ;high = N – 1;
  p = low+((high-low)/2) //Initial probe position
  while ( low <= high) {
    if ( A[p] > value )
          high = p – 1 ;
    else if ( A[p] < value )
          low = p + 1 ;
    else
          return p;
    p = low+((high-low)/2); //Next probe position.
  }
  return not_found;
  ```
  ```