Prasanna Wadekar 9970248407

19:00

Data Structure:

- Structure of Data
  - The organization of data in memory.
  - Operations performed on that data.
- Types of data structures:
  - Linear - Array, Stack, Queue, Linked List.
  - Non-Linear - Tree, Graph
  - Associative:- data is stored in key value pair e.g. HashTable
- Usually data structures are represented as "Abstract Data Types".
- Example Data Structures:
  - Array, Stack, Queue, Linked List, Tree, Graph, Hashtable, etc.

Stack (ADT):

- Operations:
  - Push()
  - Pop()
  - Peek()
  - IsEmpty()
  - IsFull()
- Data Organization In Memory:
  - Using Arrays:
    - int arr[MAX];
    - int top;
  - Using Linked List;
    - node *head;
    - Push() ==> AddFirst();
    - Pop() ==> DelFirst();
    - Peek() ==> return head->data;
    - IsEmpty() ==> return head==NULL;

PRASANNA

Array (ADT)

- Operations:
  - GetAt(index);
  - SetAt(index, value);
  - Sort();

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

- Search(key);
  - Linear Search
  - Binary Search

Time Complexity:

- Approximate measure of time required for completing any algo/operation.
- Example: Fatorial:

```
res = 1;
for(i=1; i <= num; i++)
    res = res * i;
printf("factorial : %d\n", res);
```

- Time complexity of any algo depends on number of iterations in that algo.
- In above program, time is proportional to number whose factorial is to be calculated.
- T α n . Hence time complexity is O(n) -> Order of n.
- If time required for any algo is constant (not dependent on any factor) i.e. T = k. Then time complexity is represented as O(1).
- Time complexity is represented in "Big O" notation.

Selection Sort:

```
for(i=0; i<n-1; i++)
{
    for(j=i+1; j<n; j++)
    {
        if(a[i] > a[j])
            swap(a[i], a[j]);
    }
}
```

- Number of iterations = 1 + 2 + 3 + ... + n-1 = n (n-1) / 2
- T α n (n-1) / 2
- T α n^2 - n

- If n >> 1, then n^2 >>>> n => Hence all lower order terms can be ignored.
- T α n^2
- Time complexity = $O(n^2)$

Bubble Sort:

```
for(i=0; i<n-1; i++)
{
    for(j=0; j<n-1; j++)
    {
        if(a[j] > a[j+1])
            swap(a[j], a[j+1]);
    }
}
```

- Number of iterations = (n-1)(n-1)
- T α n^2 - 2n + 1
- If n >> 1, then n^2 >>>> n => Hence all lower order terms can be ignored.
- T α n^2
- Time complexity = $O(n^2)$

Linear Search:

```
for(i=0; i<n; i++)
{
    if(key==a[i])
        return i; // element found at index i
}
return -1; // ele not found
```

- Best case : "key" is found at index 0.
  - Number of iterations = 1
  - T = k
  - O(1)
- Worst case : "key" is found at last index (n-1).
  - Number of iterations = n
  - T α n
  - O(n)
- Average case : "key" is found in middle.
  - Number of iterations (average) = n / 2
  - T α n / 2
  - T α n

- O(n)

Binary Search:

```
while(left <= right)
{
    mid = (left+right) / 2;
    if(key == arr[mid])
       return mid; // ele found at index "mid"
    if(key < arr[mid])
       right = mid - 1;
    else
       left = mid + 1;
}
return -1;
```

- Array must be sorted to use binary search.
- $2^i = n$ -> where 'n' is number of elements, 'i' is number of iterations
- i log 2 = log n
- i = log n / log 2
- T α log n / log 2
- T α log n
- Time complexity : O (log n)

Most common time complexities:

- O(1)
- O(log n)
- O(n)
- O(n log n)
- $O(n^2)$

Time complexities of different algorithms:

1. Stack push & pop: T = k. O(1)
2. Queue push & pop: T = k. O(1)
3. Singly Linked list with head pointer:
   a. Add first -> T = k. O(1)
   b. Add last -> T α n O(n)
   c. Del first -> T = k. O(1)
   d. Search -> T α n / 2 O(n)
4. Singly Linked list with head & tail pointer:
   a. Add first -> T = k. O(1)

b. Add last -> T = k. O(1)
c. Del first -> T = k. O(1)
d. Search -> T α n / 2 O(n)

Insertion Sort:

```
for(i=1; i<n; i++)
{
    temp = arr[i];
    for(j=i-1; j >= 0 && arr[j] > temp; j--)
       arr[j+1] = arr[j];
    arr[j+1] = temp;
}
```

- Time complexity:
  - Best case: O(n)
  - Average/worst case: $O(n^2)$

Recursion:

- Function calling itself is called as "recursive function".
- From programming perspective two things are important:
  - Explain process in terms of itself.
  - Terminating condition.
- Advantages:
  - Simplifies / reduce the code.
- Disadvantages:
  - Increased space complexity (function activation records are created per call).
  - Increased time (time required to create activation record and call the function).

Quick Sort:

```
void quick_sort(int arr[], int left, int right)
{
    // 0. if part contains single ele (left==right) or part is invalid
(left>right), stop.
    if(left>=right)
        return;
    // arr[left] is pivot
    int i = left, j = right;
    while(i < j)
```

```
    {
        // 1. from left find number greater than pivot - i index
        while(i <= right && arr[i] <= arr[left])
            i++;
        // 2. from right find number less than or equal to pivot - j index
        while(arr[j] > arr[left])
            j--;
        // 3. if i < j, swap arr[i] & arr[j]
        if(i < j)
            swap(arr[i], arr[j]);
    } // 4. repeat steps 1-3, till i < j

    // 5. swap pivot (arr[left]) and arr[j]
    swap(arr[left], arr[j]);
    // 6. quick_sort for left part -> 'left' to 'j-1'
    quick_sort(arr, left, j-1);
    // 7. quick_sort for right part -> 'j+1' to 'right'
    quick_sort(arr, j+1, right);
}
```

- Time Complexity :
  - Avg case : O(n log n)
  - Worst case : $O(n^2)$
- Pivot selection:
  - leftmost or rightmost element
  - middle element
  - average of all values
  - median of all values

Merge Sort

```
void merge_sort(int arr[], int left, int right)
{
    // 0. if part contains single ele (left==right) or part is invalid
(left>right), stop.
    if(left>=right)
        return;
    // 1. divide array in two equal partitions.
    int mid = (left + right) / 2;
    // 2. sort left part -> left to mid
    merge_sort(arr, left, mid);
    // 3. sort right part -> mid+1 to right
    merge_sort(arr, mid+1, right);
    // 4. allocate a temp array : num of elements (right - left + 1)
```

```
    int cnt = right - left + 1;
    int *temp = new int[cnt];
    // 5. merge two sorted partitions into this temp array
    int i=left, j=mid+1, k=0;
    while(i<=mid && j<=right)
    {
        if(arr[i] < arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }
    while(i<=mid)
        temp[k++] = arr[i++];
    while(j<=right)
        temp[k++] = arr[j++];
    // 6. copy temp array back to original array partition
    for(k=0; k<cnt; k++)
        arr[left+k] = temp[k];
    // 7. release temp array
    delete[] temp;
}
```

- Time Complexity :
  - Avg/Worst case : O(n log n)

Heap Sort:

- Array implementation of almost complete binary tree is called as "Heap".
- Time Complexity :
  - Avg/Worst case : O(n log n)

Queue (ADT)

- Operations:
  - Push()
  - Pop()
  - Peek()
  - IsEmpty()
  - IsFull()
- Data orgranization in memory:
  - Using linked list (with head & tail pointer):
    - front => node *head;
    - rear => node *tail;
    - Push() => AddLast()

- Pop() => DelFirst()
- Peek() => return head->data;
- IsEmpty() => return head==NULL;

o Using arrays:

- int arr[MAX];
- int front;
- int rear;

Linear Queue:

```
1. Init()
   front = rear = -1;
2. Push()
   rear ++;
   arr[rear] = ele;
3. Pop()
   front ++;
4. Peek()
   return arr[front+1];
5. IsEmpty()
   front == rear
6. IsFull()
   rear==MAX-1
```

Circular Queue:

- In linear queue, when rear reaches MAX-1, even if few locations are vacant, it shows queue is full. In other words, in linear queue there is no proper utilization of memory.
- To overcome this limitation, circular queue is implemented.
- In circular queue, front & rear are incremented in circular fashion i.e. 0, 1, 2, ..., MAX-1, 0, 1, ...

Implementation 1

```
1. Init()
   front = rear = -1;
2. Push()
   rear = (rear + 1) % MAX;
   arr[rear] = ele;
3. Pop()
   front = (front + 1) % MAX;
   if(front==rear)
        front = rear = -1;
4. Peek()
   temp = (front + 1) % MAX;
   return arr[temp];
5. IsEmpty()
```

```
   front == rear && front == -1
6. IsFull()
   (front==-1 && rear==MAX-1) || (front==rear && front!=-1)
```

Implementation 2 – Using count

```
1. Init()
   front = rear = -1;
   count = 0;
2. Push()
   rear = (rear + 1) % MAX;
   arr[rear] = ele;
   count++;
3. Pop()
   front = (front + 1) % MAX;
   count--;
4. Peek()
   temp = (front + 1) % MAX;
   return arr[temp];
5. IsEmpty()
   count == 0
6. IsFull()
   count == MAX
```

Types of Queues:

1. Linear Queue:
   a. No proper utilization of memory.
2. Circular Queue:
   a. Proper utilization of memory.
   b. Increments rear and front in circular fashion i.e. 0, 1, 2, 3, ..., MAX-1, 0, 1, ...
3. Priority Queue:
   a. Each element is associated with some priority.
   b. Element with highest priority is popped out first. (No FIFO behavior).
   c. Typically it is implemented as sorted linked list. While insertion element is added at appropriate position as per its priority. So insertion time complexity will be O(n). While deleting first (head) element is deleted (as it is element with highest priority).
4. Double Ended Queue (deque):
   a. Can push/pop elements from both sides i.e. front as well as rear.
   b. Usually implemented as doubly linked list with head & tail. because time complexity at both end complexity is O(1)

Stack Vs Queue:

	Stack	Queue
1	Stack is LIFO utility data structure.	Queue is FIFO utility data structure.
2	Push and Pop operations are done from the same end i.e. "top".	Push and Pop operations are done from different ends i.e. "rear" and "front" respectively.
3	There are no types of stack. However in single array you can implement two or more stacks (called as multi-stack approach).	There are four types of queues i.e. linear, circular, priority and deque.
4	Applications: • Function activation records are created on the stack for each function call. • To solve infix expression by converting to prefix or postfix. • Parenthesis balancing • To implement algos like Depth First Search.	Applications: • Printer maintains queue of documents to be printed. • OS uses queues for many functionalities: Ready queue, Waiting queue, Message queue. • To implement algos like Breadth First Search.

STL: Standard Template Library:

```
#include <stack>
stack<char> s;
s.push('A');
s.pop();
ele = s.top(); // s.peek()
s.empty();     // s.isempty()
```

```
#include <queue>
queue<char> q;
q.push('A');
q.pop();
ele = q.front(); // q.peek()
q.empty();       // q.isempty()
```

Infix, Prefix, Postfix:

- These are notations to represent math equations.
  - Infix: A + B -> For human
  - Prefix: + A B -> For computers
  - Postfix: A B + -> For computers
- Infix expression:
  - 4 + 3 - (9 - 5) / 2 * 8 - 6
- To convert Infix to Prefix/Postfix considers the priorities:
  1. ( )
  2. $ or ^ i.e. Exponential Operator
  3. * / % (left to right)
  4. + - (left to right)

Infix to Postfix using stack:

- Traverse Infix string element by element from left to right.

Infix to
1
2
3
4
5
6
7
8
Postfix
1

- If operand is found, then append to the postfix string.
- If operator is found, then push it on the stack.
- If priority of topmost of stack is greater or equal to priority of current operator, pop it from the stack and append to postfix string. Repeat this step if required.
- When infix string is completed, pop all operators from the stack and append to postfix string (one by one).
- If opening '(' is found, then push it on the stack.
- If closing ')' is found, pop all operators from the stack and append to postfix string (one by one) until '(' is found on stack. Also pop and discard that '('.
- Example: 4 + 3 - (9 - 5) / 2 * 8 - 6

Symbol	Postfix	Stack
4	4	
+	4	+
3	4 3	+
-	4 3 +	-
(	4 3 +	- (
9	4 3 + 9	- (
-	4 3 + 9	- (-
5	4 3 + 9 5	- (-
)	4 3 + 9 5 -	-
/	4 3 + 9 5 -	- /
2	4 3 + 9 5 - 2	- /
*	4 3 + 9 5 - 2 /	- *
8	4 3 + 9 5 - 2 / 8	- *
-	4 3 + 9 5 - 2 / 8 * -	-
6	4 3 + 9 5 - 2 / 8 * - 6	-
	4 3 + 9 5 - 2 / 8 * - 6 -	

Infix to Prefix using stack:

1. Traverse Infix string element by element from right to left.
2. If operand is found, then append to the prefix string.
3. If operator is found, then push it on the stack.
4. If priority of topmost of stack is greater than priority of current operator, pop it from the stack and append to prefix string. Repeat this step if required.
5. When infix string is completed, pop all operators from the stack and append to prefix string (one by one).
6. If closing ')' is found, then push it on the stack.
7. If opening '(' is found, pop all operators from the stack and append to prefix string (one by one) until ')' is found on stack. Also pop and discard that ')'.
8. Reverse prefix string.

Postfix Evaluation:

1. Traverse postfix string from left to right.

2. If operand is found, push it on the stack.
3. If operator is found, pop two operands from the stack; calculate result and push it on stack.
   a. First popped is second operand; while second popped is first operand.
4. Repeat steps 1-2, until postfix string is completed.
5. At the end, pop final result from the stack.
6. Example: 4 3 + 9 5 - 2 / 8 * - 6 -

Symbol	Stack	Operation
4	4	
3	4 3	
+	7	4 + 3 = 7
9	7 9	
5	7 9 5	
-	7 4	9 - 5 = 4
2	7 4 2	
/	7 2	4 / 2 = 2
8	7 2 8	
*	7 16	2 * 8 = 16
-	-9	7 - 16 = -9
6	-9 6	
-	-15	-9 - 6 = -15

Prefix Evaluation:

1. Traverse prefix string from right to left.
2. If operand is found, push it on the stack.
3. If operator is found, pop two operands from the stack; calculate result and push it on stack.
4. First popped is first operand; while second popped is second operand.
5. Repeat steps 1-2, until prefix string is completed.
6. At the end, pop final result from the stack.

Linked List:

- **Terminologies:**
  - Linked list is a linear data structure that contains multiple records linked to each other.
  - Each item in the list is called as "Node".
  - Each node contains data and pointer (address) to the next node.
  - Typically linked lists are implemented as self-referential structure/class. The structure/class contains pointer of the same type to hold address of next node.
- **LinkedList (ADT):**
  - The commonly supported operations in a linked list is called as:
    A. AddFirst()

B. AddLast()
C. InsertAtPos()
D. Search()
E. DeleteFirst()
F. DeleteLast()
G. DeleteAtPos()
H. Clear()
I. Sort()
J. Reverse()

- **Linked list in C++:**

```
class list;
class node
{
  private:
      int data;
      node *next;
  public:
      node(int val=0);
      friend class list;
};
class list
{
   private:
      node *next;
   public:
      list();
      ~list();
      addfirst(int val);
      addlast(int val);
      insert(int val, int pos);
      void delfirst();
      void dellast();
      void del(int pos);
      node* search(int key);
      void sort();
      void reverse();
      void clear();
};
```

- **Types of Linked Lists:**
  - **Singly Linear Linked List**
    - **A. Singly Linked List – only head pointer:**
      1. **Add First: O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      // newnode's next --> head
      newnode->next = head;
      // head --> newnode
      head = newnode;
}
```

2. Add Last: O(n)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      // traverse till last node
      trav = head;
      while(trav->next!=NULL)
            trav = trav->next;
      // trav's next --> newnode
      trav->next = newnode;
}
```

3. Insert at position: O(n)

```
// if list is empty or first pos, newnode is first node.
if(head==NULL || pos==1)
      addfirst(val);
else
{
      // create and initialize node
      newnode = new node(val);
      // traverse till pos-1 node
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // newnode's next --> trav's next
      newnode->next = trav->next;
      // trav's next --> newnode
      trav->next = newnode;
```

```
}
```

4. Delete First: O(1)

```
if(head!=NULL)
{
      // keep address of first node in temp pointer
      temp = head;
      // take head to next node.
      head = head->next;
      // delete temp node.
      delete temp;
}
```

5. Delete At Position: O(n)

```
// if node to be deleted is first or list is empty
if(pos==1 || head==NULL)
      delfirst();
else
{
      // traverse till pos-1
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // temp should point to the node to be deleted.
      temp = trav->next;
      // trav's next --> temp's next
      trav->next = temp->next;
      // delete temp node
      delete temp;
}
```

6. Delete All: O(n)

```
// delete all nodes one by one, until list become empty
while(head!=NULL)
      delfirst();
```

7. Traverse Linked List:

```
// start traversing from first node
trav = head;
while(trav!=NULL)
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the next node
```

```
        trav = trav->next;
} // repeat until end of list is reached
```

B. Singly Linked List –head and tail pointer:

1. Add First: O(1)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
        head = tail = newnode;
else
{
        // newnode's next --> head
        newnode->next = head;
        // head --> newnode
        head = newnode;
}
```

2. Add Last: O(1)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
        head = tail = newnode;
else
{
        // tail's next --> newnode
        tail->next = newnode;
        // tail --> newnode
        tail = newnode;
}
```

3. Delete First: O(1)

```
if(head!=NULL)
{
        // keep address of first node in temp pointer
        temp = head;
        // take head to next node.
        head = head->next;
        // if deleted node was last node, then
        if(head==NULL)
                tail = NULL;
        // delete temp node.
        delete temp;
```

```
}
```

- **Singly Circular Linked List:**

  Last node's next pointer keeps address of first (head) node.

1. **Add First: O(n)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
      head = newnode;
      newnode->next = newnode;
}
else
{
      // traverse till last node
      trav = head;
      while(trav->next!=head)
            trav = trav->next;
      // trav's next --> newnode
      trav->next = newnode;
      // newnode's next --> head
      newnode->next = head;
      // head --> newnode
      head = newnode;
}
```

2. **Add Last: O(n)**

```
// if list is empty, newnode is first node.
if(head==NULL)
{
      head = newnode;
      newnode->next = newnode;
}
else
{
      // traverse till last node
      trav = head;
      while(trav->next!=head)
            trav = trav->next;
      // trav's next --> newnode
      trav->next = newnode;
      // newnode's next --> head
      newnode->next = head;
```

```
}
```

3. Insert at position: O(n)

```
// if list is empty or first pos, newnode is first node.
if(head==NULL || pos==1)
      addfirst(val);
else
{
      // create and initialize node
      newnode = new node(val);
      // traverse till pos-1 node
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // newnode's next --> trav's next
      newnode->next = trav->next;
      // trav's next --> newnode
      trav->next = newnode;
}
```

4. Delete First: O(n)

```
// if head is null, return
if(head==NULL)
      return;
// if there is a single node, delete it
else if(head==head->next)
{
      delete head;
      head = NULL;
}
// if there are more than one node
else
{
      // keep address of head node in temp pointer
      temp = head;
      // traverse till last node
      trav = head;
      while(trav->next!=head)
            trav = trav->next;
      // take head to next node.
      head = head->next;
      // update last node's next --> new head
      trav->next = head;
      // delete temp node.
      delete temp;
}
```

5. Delete At Position: O(n)

```
// if node to be deleted is first or list is empty
if(pos==1 || head==NULL)
      delfirst();
else
{
      // traverse till pos-1
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // temp should point to the node to be deleted.
      temp = trav->next;
      // trav's next --> temp's next
      trav->next = temp->next;
      // delete temp node
      delete temp;
}
```

6. Traverse Linked List:

```
// start traversing from first node
trav = head;
do
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the next node
      trav = trav->next;
} while(trav!=head);
// repeat until end of list is reached
```

- **Doubly Linear Linked List**

  Each node contains data, address of next node and address of previous node.

1. Add First: O(1)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      // newnode's next --> head
      newnode->next = head;
      // cur head's prev --> newnode
      head->prev = newnode;
```

```
        // head --> newnode
        head = newnode;
}
```

2. Add Last: O(n)

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
        head = newnode;
else
{
        // traverse till last node
        trav = head;
        while(trav->next!=NULL)
                trav = trav->next;
        // trav's next --> newnode
        trav->next = newnode;
        // newnode's prev --> last node (trav)
        newnode->prev = trav;
}
```

3. Insert at position: O(n)

```
// if list is empty or first pos, newnode is first node.
if(head==NULL || pos==1)
        addfirst(val);
else
{
        // create and initialize node
        newnode = new node(val);
        // traverse till pos-1 node
        trav = head;
        for(i=1; i<pos-1; i++)
                trav = trav->next;
        // newnode's next --> trav's next
        newnode->next = trav->next;
        // newnode's prev --> trav
        newnode->prev = trav;
        // next node's prev --> newnode
        if(trav->next!=NULL)
                trav->next->prev = newnode;
        // trav's next --> newnode
        trav->next = newnode;
}
```

4. **Delete First: O(1)**

```
if(head!=NULL)
{
      // keep address of first node in temp pointer
      temp = head;
      // take head to next node.
      head = head->next;
      // delete temp node.
      delete temp;
      // if node deleted is not last, make its prev NULL
      if(head!=NULL)
            head->prev = NULL;
}
```

5. **Delete At Position: O(n)**

```
// if node to be deleted is first or list is empty
if(pos==1 || head==NULL)
      delfirst();
else
{
      // traverse till pos
      trav = head;
      for(i=1; i<pos; i++)
            trav = trav->next;
      // trav's prev node's next --> trav's next
      trav->prev->next = trav->next;
      // trav's next node's prev --> trav's prev
      if(trav->next!=NULL)
            trav->next->prev = trav->prev;
      // delete trav node
      delete trav;
}
```

6. **Delete All: O(n)**

```
// delete all nodes one by one, until list become empty
while(head!=NULL)
      delfirst();
```

7. **Traverse Linked List:**

```
// start traversing from first node
trav = head;
while(trav!=NULL)
{
      // visit current node
```

```
        printf("%d, ", trav->data);
        // go to the next node
        trav = trav->next;
} // repeat until end of list is reached
```

8. **Traverse Linked List – In Reverse Direction:**

```
// traverse till last node
trav = head;
while(trav->next!=NULL)
        trav = trav->next;
// start traversing from last node
while(trav!=NULL)
{
        // visit current node
        printf("%d, ", trav->data);
        // go to the prev node
        trav = trav->prev;
} // repeat until beginning of list is reached
```

- **Doubly Circular Linked List**

  Each node contains data, address of next node and address of previous node. Last node's next contains address of first node, while first node's prev contains address of last node.

1. **Add First: O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
        head = newnode;
        newnode->next = newnode;
        newnode->prev = newnode;
}
else
{
        // jump to the last node
        tail = head->prev;
        // newnode's next --> head
        newnode->next = head;
        // newnode's prev --> tail
        newnode->prev = tail;
        // tail's next --> newnode
        tail->next = newnode;
        // cur head's prev --> newnode
        head->prev = newnode;
```

```
        // head --> newnode
        head = newnode;
}
```

2. **Add Last: O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
        head = newnode;
        newnode->next = newnode;
        newnode->prev = newnode;
}
else
{
        // jump to the last node
        tail = head->prev;
        // newnode's next --> head
        newnode->next = head;
        // newnode's prev --> tail
        newnode->prev = tail;
        // tail's next --> newnode
        tail->next = newnode;
        // cur head's prev --> newnode
        head->prev = newnode;
}
```

	Array	Linked List
.	Array cannot grow or shrink dynamically (realloc() is not efficient).	Linked list can grow/shrink dynamically.
.	Array has contiguous memory. Hence random access is possible.	Nodes are not in contiguous memory. Hence only sequential access is allowed.
.	Arrays do not have memory overheads.	Linked lists have memory overheads e.g. each node contains address of 'next' node.
.	Insert/Delete at middle position need to shift remaining elements. Hence will be slower.	Insert/Delete at middle position need to modify links (next/prev pointers). Hence will be faster.
.	To deal to fixed number of elements and frequent random access, arrays are better.	To deal to dynamic number of elements and frequent insertion/deletion operators, linked lists are better.

	Singly Linked List with Head	Singly Linked List with Head & Tail
.	Add last operation is slower i.e. O(n)	Add last operation is faster i.e. O(1)
.	No overload.	Extra overhead i.e. tail pointer.

3.	Simplified list manipulation operations.	List manipulation need to consider tail pointer whereve appropriate.

	Singly Linked List	Doubly Linked List
1.	Unidirectional access.	Bi-directional access.
2.	For search and delete operation two pointers are required (Need to maintain prev pointer) during traversing.	For search and delete operation only one pointer is required (No need to maintain prev pointer) during traversing.
3.	Simplified list manipulation operations.	List manipulation need to consider prev pointer of each node in all operations.

	Doubly Linked List	Doubly Circular Linked List
1.	Traversing till last node is slower (add last operation) i.e. O(n)	Traversing till last node is faster (add last operations) i.e O(1)

Trees

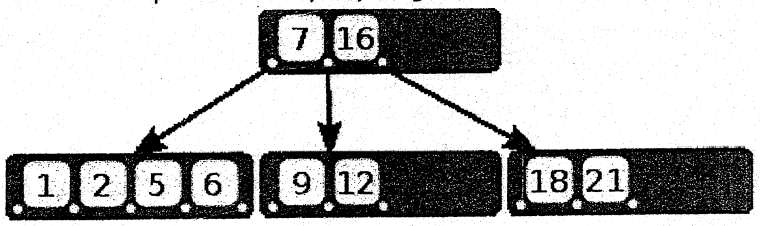
- **Terminologies:**
  - Tree is a non-linear data structure in which one specially designated node is called as "root" (i.e. startin point of the tree) and remaining elements can be partitioned into "m" disjoint subsets so that each subset is a tree itself.
  - Number of child nodes for any node is called as "degree of a node".
  - Maximum degree of a child in tree is called as "degree of the tree".
  - All the nodes in the path from the root to that node, are called "ancestors" of that node.
  - All the nodes accessible from the given node, are called as "descendants" of that node.
  - Child nodes of the same parent are called as "Siblings".
  - Terminal node of the tree is called as "leaf node". Leaf node does not have child nodes.
  - Level of node:
    - Indicates position of the node in tree hierarchy.
    - Level of child = Level of parent + 1
    - Level of root = 0
  - The maximum level of a node is "Height of tree".
  - Tree with zero nodes (i.e. empty tree) is called as "Null Tree". Height of Null tree is -1.
- **Types of Trees:**
  - **Binary Tree:**
    - Tree in which each node has maximum two child nodes, is called as "binary tree".
    - Binary tree has degree 2. Hence it is also called as 2-tree.
  - **Binary Search Tree:**

- The binary tree in which left child node is always smaller and right child node is always greater or equal to the (parent) node; is called as "Binary Search Tree".
- Searching is faster. Time complexity: O(h). Where "h" is height of the tree.

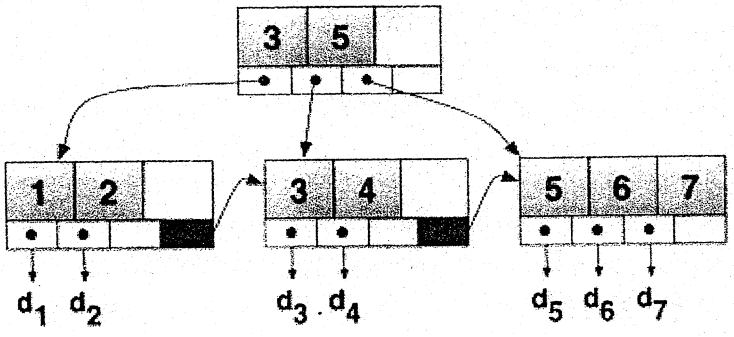
- **Complete Binary Tree:**
  - Binary tree in which all leaf nodes are at the same level.
- **Strictly Binary Tree:**
  - Binary tree in which each non-leaf node has exact two child nodes.
- **Full Binary Tree:**
  - Binary tree with its full capacity for the given height.
  - In other words, adding one more node will increase height of the tree.
  - It is always complete as well as strictly binary tree.
  - Number of elements = $2^{(h+1)} - 1$
- **Almost complete binary tree:**
  - The binary tree which follows two conditions:
    - All leaf nodes are at level h or h-1.
    - All leaf nodes at last level (h), are aligned to left as much as possible.
- **Threaded Binary Tree:**
  - These trees will allow very fast in-order traversal (ascending or descending).
  - Binary tree in which right null link of each node is replaced by address of its in-order successor; such tree is called as "right threaded binary tree".
  - Binary tree in which left null link of each node is replaced by address of its in-order predecessor; such tree is called as "left threaded binary tree".
  - If right as well as left null links of each node is replaced by its in-order successor and in-order predecessor respectively; then such tree is called as "in threaded binary tree".
- **Skewed Binary Tree:**
  - The binary tree in which only left link is used to keep address of child node (right link of each node is kept null), is called as "left skewed binary tree".
  - The binary tree in which only right link is used to keep address of child node (left link of each node is kept null), is called as "right skewed binary tree".
  - Height of skewed binary tree is maximum for given number of nodes. Hence searching will be too slow i.e. O(n)
- **Balanced binary tree:**
  - In binary search trees, searching is faster if height of the tree is kept as minimum as possible.
  - The binary search tree with minimum possible height (for given number of nodes) is called as "balanced binary search tree" or "height balanced binary search tree".
  - Balance factor of node = height of left sub tree - height of right sub tree.
  - The binary search tree, in which balance factor of each node is in range of -1 to 1, is called as "balanced binary search tree".

- On any binary search tree, series of appropriate left rotations and right rotations on appropriate nodes can convert it into "balanced binary search tree".
  - If balance factor of a node < -1, apply left rotation on that node.
  - If balance factor of a node > +1, apply right rotation on that node.

- **AVL Tree:**
  - Invented by "Greogy Adelson-Velsky" and "E.M. Landis".
  - Self-balancing binary search tree.
  - After each insert or delete operation, appropriate rotations are applied to make tree balanced again.
- **B Tree:**
  - The B-tree is a generalization of a binary search tree in that a node can have more than two children.
  - In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range.
  - In order to maintain the pre-defined range, internal nodes may be joined or split.
  - Each internal node of a B-tree will contain a number of keys. The keys act as separation values which divide its sub-trees.
  - A B-tree is kept balanced by requiring that all leaf nodes be at the same depth (level).



- **B+ Tree:**
  - Like B trees but with few changes as follows:
  - In the B+ tree, copies of the keys are stored in the internal nodes.
  - The keys and records are stored in leaves.
  - A leaf node may include a pointer to the next leaf node to speed sequential access.



Binary Search Tree:

- Implementation (Design) of binary search tree:

```
class tree;
class node
{
private:
   int data;
   node *left, *right;
public:
   node(int val=0);
   friend class tree;
};
class tree
{
private:
   node *root;
public:
   tree();
   void add(int val); // add node
   node* search(int key); // search node
   node* search(int key, node **parent); // search node with parent
   void preorder(node *trav); // preorder traversal
   void preorder(); // wrapper for preorder(root);
   void inorder(node *trav); // inorder traversal
   void inorder(); // wrapper for inorder(root);
   void postorder(node *trav); // postorder traversal
   void postorder();     // wrapper for postorder(root);
   void height(node *trav); // height of tree
   void height();  // wrapper for height(root);
   void clear(node *trav); // delete node and its descendants
   void clear();    // wrapper for clear(root);
   void preorder_nonrec(); // preorder traversal non-recursive
   void inorder_nonrec(); // inorder traversal non-recursive
   void postorder_nonrec(); // postorder traversal non-recursive
   void del(int key); // search and delete node
};
```

- Add a node in tree:

```
void add(int val)
{
        // create and initialize node
        node *newnode = new node(val);
        // if tree is empty
        if (root == NULL)
```

```
                root = newnode;
        else
        {
                node *trav = root;
                while (true)
                {
                        if (val < trav->data)
                        {
                                if (trav->left == NULL)
                                {
                                        trav->left = newnode;
                                        break;
                                }
                                trav = trav->left;
                        }
                        else
                        {
                                if (trav->right == NULL)
                                {
                                        trav->right = newnode;
                                        break;
                                }
                                trav = trav->right;
                        }
                }
        }
}
```

- **Tree Traversal:**
  - Pre-order: Parent , Left, Right
  - In-order: Left, Parent, Right
  - Post-order: Left, Right, Parent

```
void preorder(node *trav)
{
      if (trav == NULL)
            return;
      printf("%d, ", trav->data);
      preorder(trav->left);
      preorder(trav->right);
}
void inorder(node *trav)
{
      if (trav == NULL)
```

```
            return;
      inorder(trav->left);
      printf("%d, ", trav->data);
      inorder(trav->right);
}
void postorder(node *trav)
{
      if (trav == NULL)
            return;
      postorder(trav->left);
      postorder(trav->right);
      printf("%d, ", trav->data);
}
```

- **Delete All Nodes in Tree:**

```
void clear(node *trav)
{
      if (trav == NULL)
            return;
      clear(trav->left);
      clear(trav->right);
      delete trav;
}
void clear()
{
      clear(root);
      root = NULL;
}
```

- **Height of tree:**

```
int height(node *trav)
{
      if (trav == NULL)
            return -1;
      int hl = height(trav->left);
      int hr = height(trav->right);
      int max = (hl > hr ? hl : hr);
      return max + 1;
}
int height()
{
      return height(root);
}
```

- **Search a node with its parent:**

```
node* search(int key, node **parent)
{
	node *trav = root;
	*parent = NULL;
	while (trav != NULL)
	{
		if (key == trav->data)
			return trav;
		*parent = trav;
		if (key < trav->data)
			trav = trav->left;
		else
			trav = trav->right;
	}
	*parent = NULL;
	return NULL;
}
```

- **Delete a node:**
  - if node's left is null:

```
if(temp->left==NULL)
{
	if(temp==root)
		root = temp->right;
	else if(temp==parent->left)
		parent->left = temp->right;
	else
		parent->right = temp->right;
}
```

  - if node's right is null:

```
if(temp->right==NULL)
{
	if(temp==root)
		root = temp->left;
	else if(temp==parent->left)
		parent->left = temp->left;
	else
		parent->right = temp->left;
}
```

  - if node's left as well as right is NOT null:

```
if(temp->left!=NULL && temp->right!=NULL)
{
```

```
        // find the succ of temp along with its (succ) parent
        parent = temp;
        succ = temp->right;
        while(succ->left!=NULL)
        {
              parent = succ;
              succ = succ->left;
        }
        // replace temp data by succ data
                  temp->data = succ->data;
        // mark succ for deletion
        temp = succ;
}
```

- deleting node (full) - steps:
  - Search node to be deleted along with its parent.
  - If node's left as well as right is NOT null, deletion code for that.
  - If node's left is null, deletion code for that.
  - Else if node's right is null, deletion code for that.
  - delete memory of temp node.

- **Pre-order non-recursive:**

```
void preorder_nonrec()
{
      printf("NONREC PRE : ");
      node *trav = root;
      stack<node*> s;
      while (trav != NULL || !s.empty())
      {
            while (trav != NULL)
            {
                  printf("%d, ", trav->data);
                  if (trav->right != NULL)
                        s.push(trav->right);
                  trav = trav->left;
            }
            if (!s.empty())
            {
                  trav = s.top();
                  s.pop();
            }
      }
      printf("\n");
}
```

- **In-order non-recursive:**

```
void inorder_nonrec()
{
	printf("NONREC IN  : ");
	node *trav = root;
	stack<node*> s;
	while (trav != NULL || !s.empty())
	{
		while (trav != NULL)
		{
			s.push(trav);
			trav = trav->left;
		}
		if (!s.empty())
		{
			trav = s.top();   s.pop();
			printf("%d, ", trav->data);
			trav = trav->right;
		}
	}
	printf("\n");
}
```

- **Post-order non-recursive:**

```
void postorder_nonrec()
{
	printf("NONREC POST: ");
	node *trav = root;
	stack<node*> s;
	while (trav != NULL || !s.empty())
	{
		while (trav != NULL)
		{
			s.push(trav);
			trav = trav->left;
		}
		if (!s.empty())
		{
			trav = s.top(); s.pop();
			if (trav->right == NULL || trav->right->visited == true)
			{
				printf("%d, ", trav->data);
				trav->visited = true;
				trav = NULL;
			}
			else
			{
```

```
                        s.push(trav);
                        trav = trav->right;
                    }
                }
            }
            printf("\n");
}
```

- Depth First Search:

```
node* bfs(int key)
{
      node *trav;
      queue<node*> q;
      //push root on queue
      q.push(root);
      //repeat until queue is empty
      while (!q.empty())
      {
            // pop an ele from queue
            trav = q.front();     q.pop();
            // compare with ele to search
            if (key == trav->data)
                  return trav;
            // if trav has left child, push on queue
            if (trav->left != NULL)
                  q.push(trav->left);
            // if trav has right child, push on queue
            if (trav->right != NULL)
                  q.push(trav->right);
      }
      return NULL;
}
```

- Breadth First Search:

```
node* dfs(int key)
{
      node *trav;
      stack<node*> s;
      //push root on stack
      s.push(root);
      //repeat until stack is empty
      while (!s.empty())
      {
            // pop an ele from stack
            trav = s.top(); s.pop();
```

```
            // compare with ele to search
            if (key == trav->data)
                return trav;
            // if trav has right child, push on stack
            if (trav->right != NULL)
                s.push(trav->right);
            // if trav has left child, push on stack
            if (trav->left != NULL)
                s.push(trav->left);
        }
        return NULL;
}
```

Array Implementation of tree:

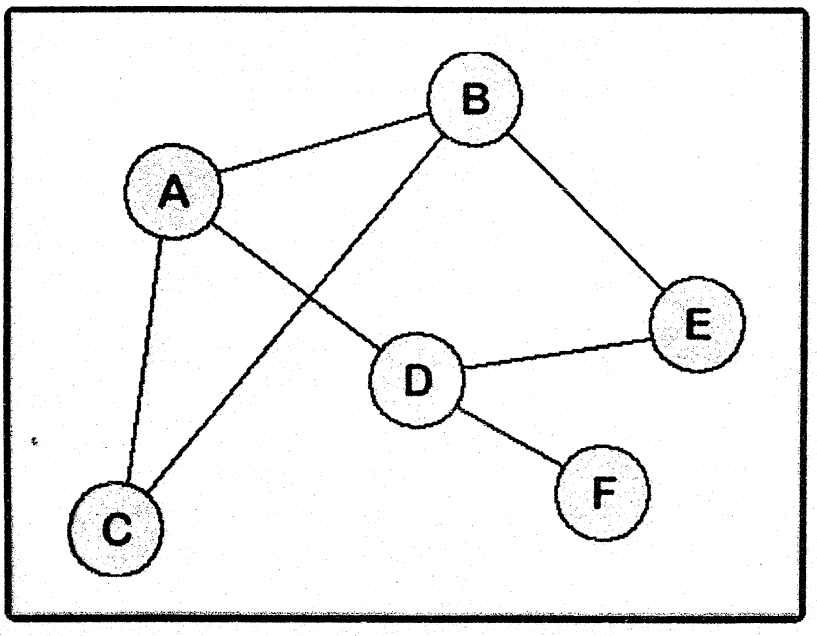
- A binary tree can be simulated in an array i.e. parent-child relationship can be maintained.
- Typically array begin with index 1. Then
  - parent index = child index / 2
  - left child index = parent index * 2
  - right child index = parent index * 2 + 1
- Array implementation of almost complete binary tree, is called as "Heap" data structure.
- If each parent element in heap is greater than both of its child elements, then it is called as "Max Heap".
- If each parent element in heap is less than both of its child elements, then it is called as "Min Heap".
- Heap sort algorithm uses Max heap (ascending sort) or Min heap (descending sort).

Hash-table

- Hash-table is data structure used for fastest searching mechanism.
- In hash-table, data is stored in key value pair, so that for a given key value can be searched in fastest possible time.
- Ideal time complexity of searching/inserting in hash-table = O(1).
- Since data is stored as key-value pair, it is an "associative" data structure.
- Typically hash table is implemented as array and element is stored at appropriate index (calculated using hash function).
- Hash function:
  - The mathematical function of "key", which decides the slot (index) of the table (array) to be used for storing key-value pair.
  - e.g. Key is "roll" and value is "Student" object; simplest hash function can be $h(r) = r - 1$
  - In this case, Student with roll number 51 will be stored at index 50.
- Ideal hash function gives different slot for different keys.

- However, many times different key values can compete for the same slot of the table. This situation is called as "collision".
- e.g. Key is "roll" and value is "Student" object and table has 10 slots. So hash function can be h(r) = r % 10
- In this case, student with roll 52, 32, 72 will result in the same slot number i.e. slot 2.
- Load Factor = Number of Entries / Number of slots
- Collision can be handled by one of the following mechanisms:
    - Open Addressing:
        - If slot computed by hash function is already in use; then another mathematical function can be used for next possible empty slot. This function is called as "rehashing" function.
        - In above example: h(r) = r % 10
        - Then, Rehashing function can be : f(i) = (i + 1) % 10 i.e. Immediately next slot.
        - The rehashing function can be a linear function (as example). In that case, open addressing is also called as "linear probing".
        - If required, rehashing function can be a quadratic or any other polynomial function.
        - This mechanism can be used if and only if, number of entries is less than number of slots in hash-table. In other words, open addressing can be used only if load factor <= 1.
    - Chaining:
        - Each slot in the hash table maintains a linked list of key-value pairs, which will compete for the same slot. This linked list is called as "buckets".
        - This mechanism can be used irrespective of load factor.
        - If number of collisions is more, time required for searching/insertion will increase. In other words, time complexity will be deviated from O(1).

Graph

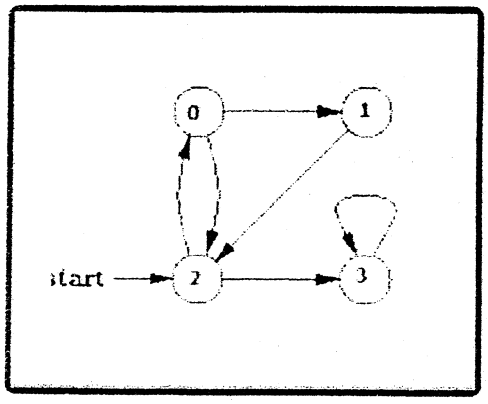


- **Terminologies:**
  - Graph is a non-linear data structure having set of vertices (nodes) and set of edges (arcs).
    - G={V, E} Where V is set of vertices and E is set of edges.
  - Vertex (node) is an element in graph.
    - V = {A, B, C, D, E, F}
  - Edge (arc) is a line connecting two vertices (nodes).
    - E = {(A,B), (A,C), (A,D), (B,C), (B,E), (D,E), (D,F)}
  - Vertex A is said to be adjacent to B, if and only if there is an edge from A to B.
  - Number of vertices adjacent to given node, is called as "degree of node".
  - Set of edges connecting any two vertices is called as "Path" between those two vertices.
    - Path between A to D = { (A,C), (A,B), (B,E), (E,D) }
  - Set of edges connecting a node to itself is called as "Cycle".
    - {(A,B), (B,E), (E,D), (D,A)}
  - An edge connecting a node to itself is called as "Loop". Loop is smallest cycle.

- **Types of Graph:**
  - **Simple graph:**
    - If graph do not have multiple edges between adjacent nodes and no loops present, then it is called as "Simple graph".
  - **Complete graph:**

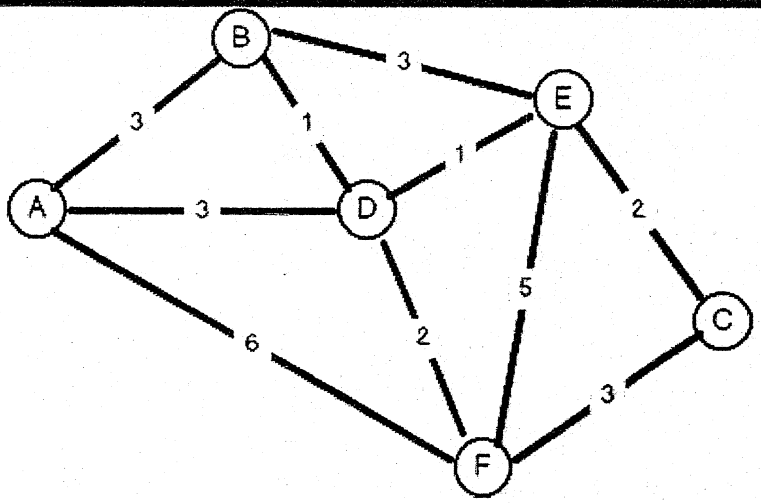
- A simple graph in which each node is adjacent with every other node, then it is called as "Complete Graph".
- Number of edges = n (n-1) / 2

- **Connected graph:**
  - A simple graph in which there is some path exists between any two vertices, is called as "Connected graph".
- **Multi-graph:**
  - If in a graph, there are multiple edges between two adjacent nodes or loops, then it is called as "multi-graph" (See above diagram).

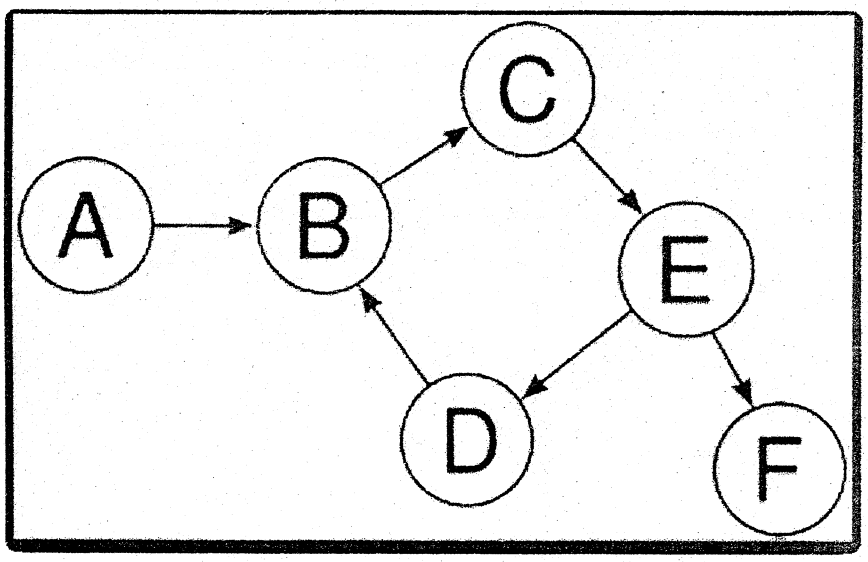


Prasanna

- **Weighted graph:**
  - A graph in which each edge is associated with a number (i.e. weight) is called as "weighted graph".
  - Examples:
    - If two vertices represents cities, then weight will represent distance between those two cities.
    - If two vertices represents connecting points in an electronic circuit, then weight will represent resistance between those two points.
  - The sum of weights of all edges in a path is called as "weight of the path".
  - Dijkstra's algorithm is used to find shortest path in two vertices.



- **Directed Graph:**
  - A graph in which each edge has some edge is called as "Directed Graph".
  - In below example:
    - E = {<A,B>, <B,C>, <C,E>, <D,B>, <E,D>, <E,F>}
  - Note that there is an edge from A to B; so A is adjacent to B. In this case B is not adjacent to A.
  - For node E,
    - Number of edges originated from E = 2 called as "out-degree".
    - Number of edges terminated on E = 1 called as "in-degree".
  - Directed graphs are also called as "di-graphs".



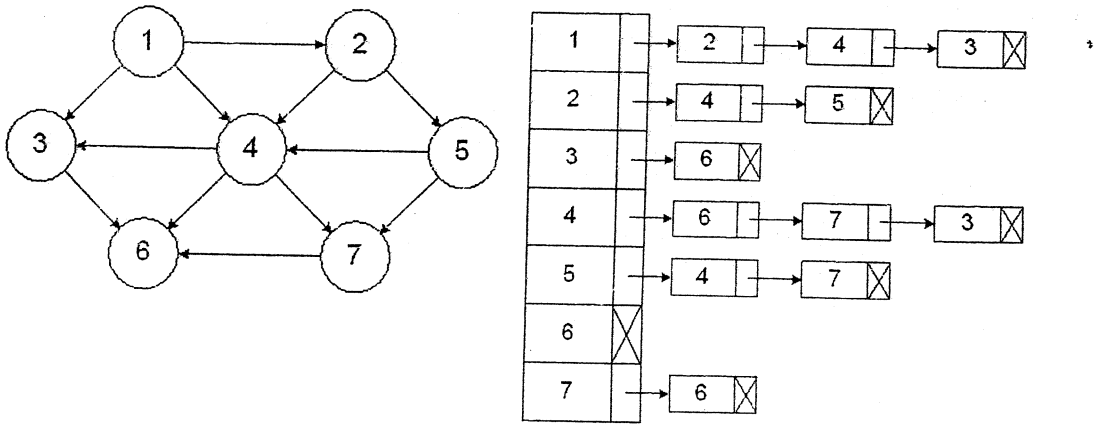
- **Directed Weighted Graph:**

- The graph in which each edge has some weight as well as direction, it is called as "directed weighted graph".
- It is called as "Network".

- **Spanning Tree:**
  - A tree can be defined as a graph without any cycle.
  - Any connected graph can be converted into a tree by removing few edges.
  - A tree having all vertices of the graph but eliminated one or more edges is called as "Spanning tree" of the graph.
  - One graph can have multiple spanning trees (by removing different edges).
  - Sum of weights of all edges in spanning tree is weight of the spanning tree.
  - Algorithms used to calculate minimum weight spanning tree are:
    - Prim's algorithm
    - Kruskal's algorithm
- **Forest:**
  - A forest is an undirected graph, all of whose connected components are trees.
  - In other words, the graph consists of a disjoint union of trees. Equivalently, a forest is an undirected cycle-free graph.

Graph Presentation/Creation

1. **Adjacency List [Linked list]**
   a. In this method, array is used to represent vertices. Also each vertex holds a linked list for all vertices that are adjacent to that vertex.

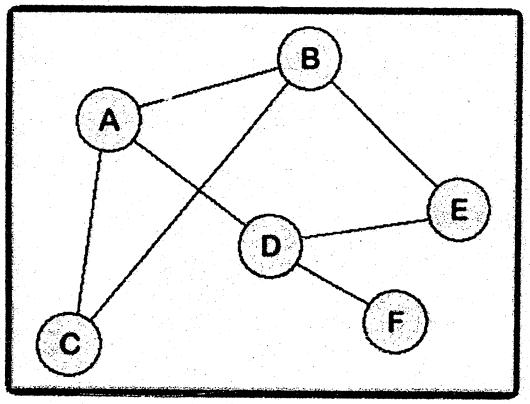


   b. Total space required is V + E.

2. **Adjacency Matrix [2-D array]**
   a. A graph with "n" vertices can be represented by n x n matrix.

b. If there is an edge between two vertices '1' & '2', then it is represented by 1 in the matrix i.e. mat[1][2] = 1. If there is no edge, corresponding element in matrix will be zero.

c. For undirected graphs, matrix will be symmetric across the diagonal.



d. For above graph matrix will be:

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

e. In case of weighted graph, matrix element corresponding to an edge stores weight (number). If there is no edge connected, matrix element will be ∞.

f. Total space required is N^2.