Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

AVL TREES

Abstract

Balanced tree, Need for Balanced tree, definition, AVL Trees, Rotation.

Dr.Sandesh and Saritha

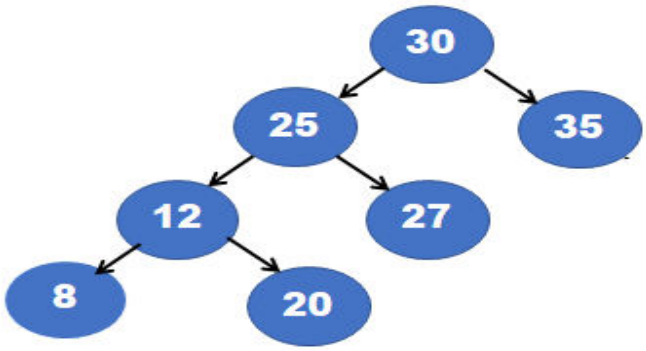
Sandesh_bj@pes.edu

Saritha.k@pes.edu

Balanced tree

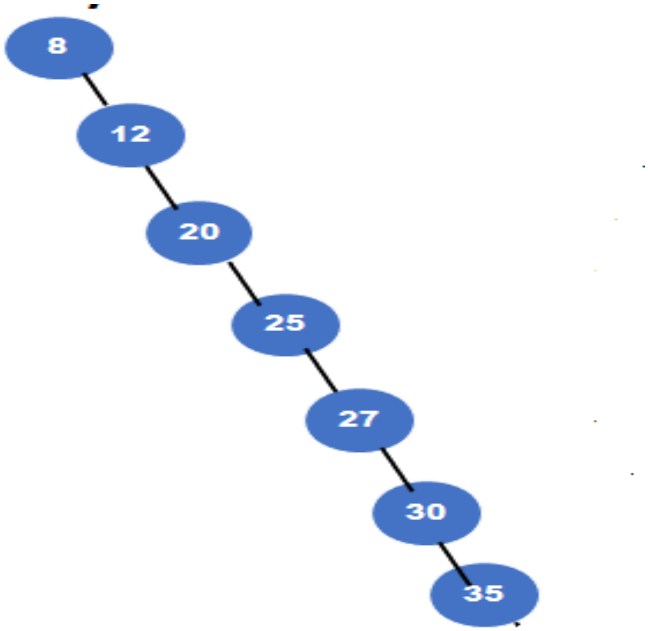
Most operations on a Binary search tree take time directly proportional to the height of the tree, so it is important to keep the height of the tree small.

A balanced binary search tree is a tree that naturally keeps its height small for a sequence of insertion and deletion operation.



Balanced Binary search tree

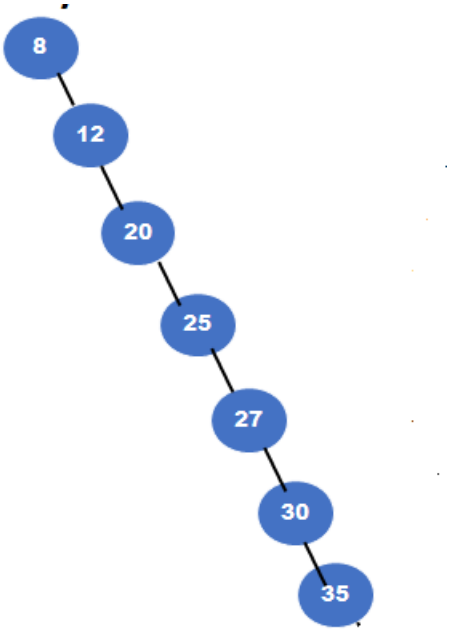
The tree shown above is balanced because the difference between the heights of left subtree and right subtree is not more than one.



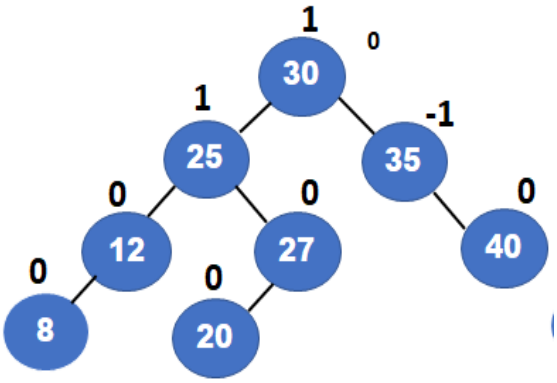
The tree shown above is unbalanced because the tree right side is 6 leaves taller than the left hand side.

Why do Binary search trees have to be balanced?

Balanced search tree decreases the number of comparisons required to find a data element. For example consider a below unbalanced tree, to search for an element 35 in the below tree seven comparisons is required. Whereas in balanced tree it requires only 2 comparisons which means the search performance increased by 50% in a balanced tree.



(i)Unbalanced tree



(ii) Balanced tree

There are different algorithms used to balance the binary search tree such as

1. AVL trees

2. B-tree

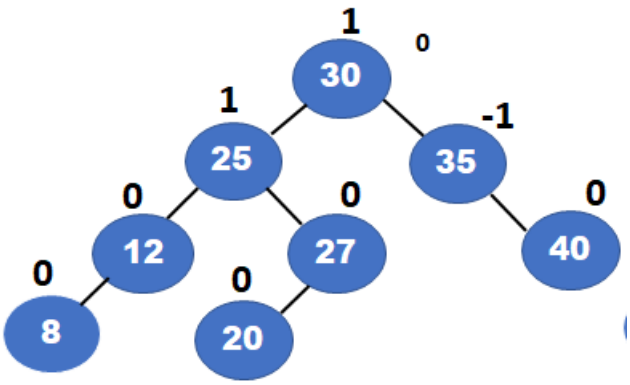
3. Red Black trees

All these data structures force the tree to remain balanced and therefore guarantee performance.

AVL Tree

An AVL tree is height balanced Binary search tree invented by Adelson-Velskii and Landis. In AVL tree the heights of left and right sub-trees of a root differ by at-most one. Every node in the AVL tree is associated with balance factor that is left higher, equal-height or right-higher accordingly ,respectively as the left subtree has height greater than, equal to, or less than that of the right subtree.

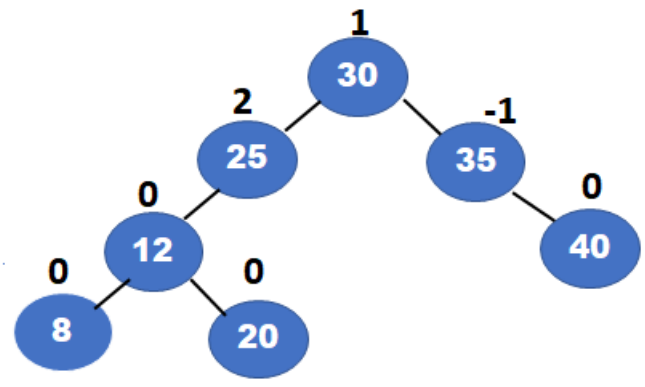
Example of AVL tree:



AVL tree

The tree shown above is an AVL tree as the difference between height of left subtree and right subtree of every node is at most one.

Example for not an AVL tree



Not an AVL tree

The above figure is not an AVL tree as the difference between height of left subtree and right subtree of root node is 2

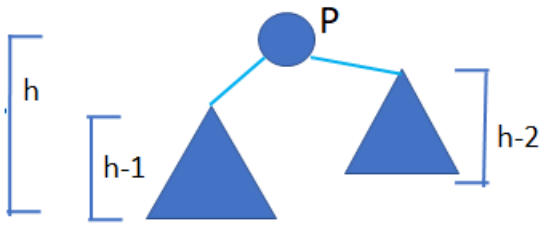
An AVL tree has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.
3. Height of AVL tree is always logarthmic in n.

An AVL tree has balance factor calculated at every node. For every node, height of left and right sub-trees can differ by no more than 1.

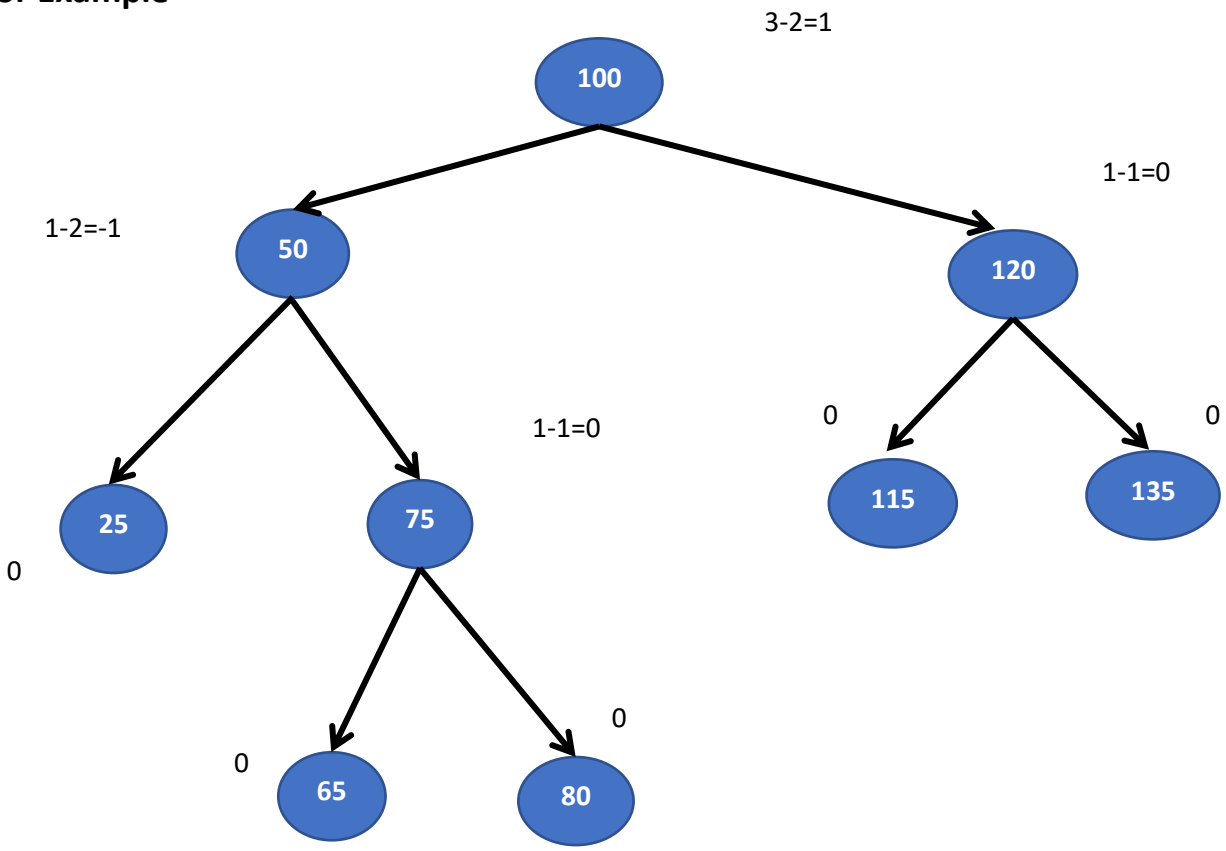
The balance factor of any node in an AVL tree is given by:

Balance factor=Height (left subtree)-Height (right subtree)



Balance factor = (h-1)-(h-2)

For Example



AVL tree rotations:

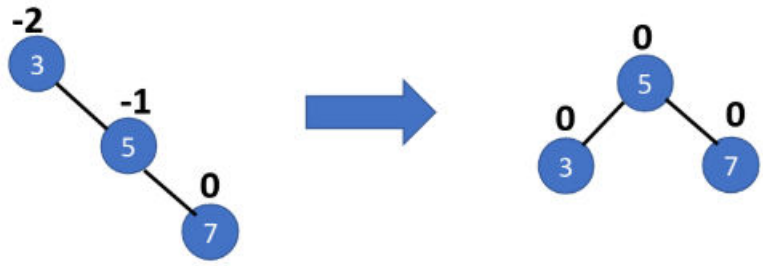
To balance the AVL tree after the insertions and deletion operations the four kinds of rotations are used. Rotations are an adjustment to the tree, around a node, that maintains the required ordering in the binary search tree.

1. Single Rotation
   a) Left rotation
   b) Right rotation
2. Double Rotation
   a) Left-Right rotation
   b) Right –Left rotation

1. Left Rotation

The tree is rotated left side to rebalance the tree, when a node is inserted into the right subtree of the right subtree. For example if the insertion order is 3, 5 and 7 the tree becomes imbalanced after the insertion of node 7. So we perform Left rotation (rotate in anti- clockwise) to balance the tree.

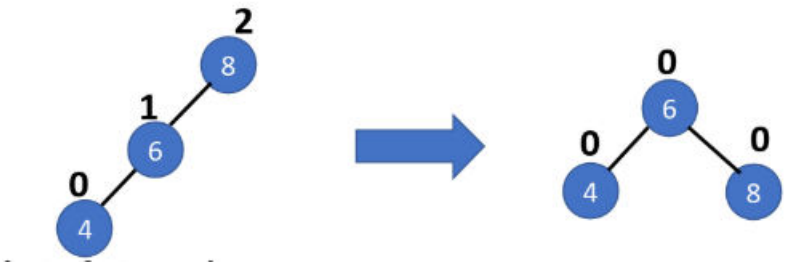
For example



Right unbalanced tree Left Rotation Balanced tree

2. Right Rotation

The tree is rotated right side to rebalance the tree, when a node is inserted in the left subtree of the left subtree. For Example if the insertion order is 8,6,4 the tree becomes imbalanced after the insertion of node 4 so to balance the tree we perform Right rotation(rotate clockwise).



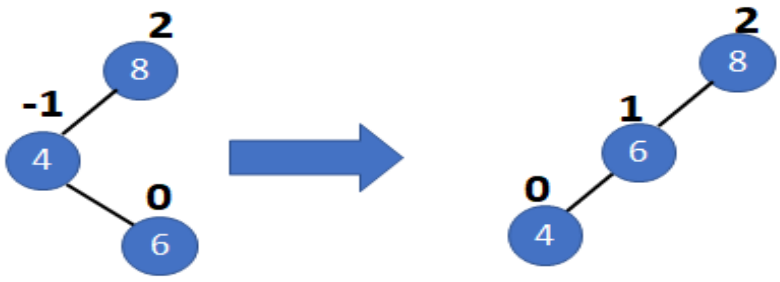
Left unbalanced tree right rotation Balanced tree

3. Left-Right Rotation

In Left-Right Rotation two rotations are performed to balance the tree. The tree is first rotated on left side and then on right side.

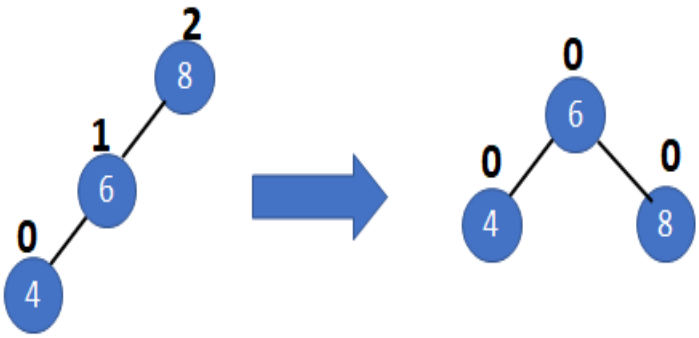
For example in the below unbalanced tree 8 is the root node, 4 is the left child of 8 and 6 is the Right child of 4. After left rotation on 4 and 6 ,the tree

structure looks like as shown below 8 is the root node,6 is the left child of 8 and 4 is the left child of 6



Unbalanced tree Left rotation unbalanced tree

The tree obtained after left rotation is unbalanced so again the tree is rotated right side inorder to balance the AVL tree. So a balanced tree with 6 as the root node,4 will become the left child of 6 and 8 is the right child of 8 is obtained.

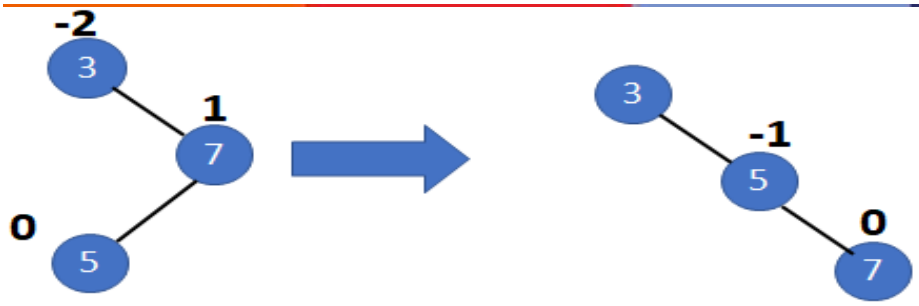


Unbalanced tree Right rotation Balanced Tree

4. Right-Left Rotation

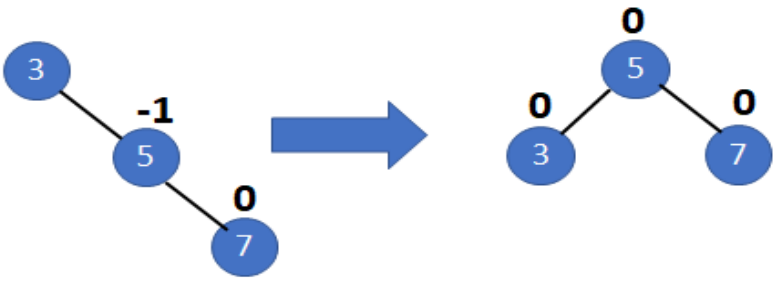
In Right-Left Rotation, first the tree is rotated right side and then to the left side.

For Example in the below unbalanced graph, 3 is the root node 7 is the right child of 3 and 5 is the left child of 7.After applying first right rotation on 7 and 5, then we get a unbalanced tree with 3 as the root node, 5 is the right child of 3 and 7 is the right child of 5.



Unbalanced tree Right rotation unbalanced tree

Then the tree is rotated left side inorder to obtain the balanced tree where 5 is the root node, 3 will be left child of 5 and 7 will be right child of 5.



Unbalanced tree Left rotation Balanced tree

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

AVL TREES

Abstract

Representation of AVL tree, operations performed on AVL, insertion, Deletion, Search

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Representation of AVL Trees

```
struct NODE
{
    int info;
    struct NODE *left,*right;
    int balancefactor;
};
```

AVL operations

The Different operations performed on AVL tree are

1. Insert
2. Delete
3. Search

Insertion in AVL tree:

For example construct AVL Tree for the following sequence of numbers-

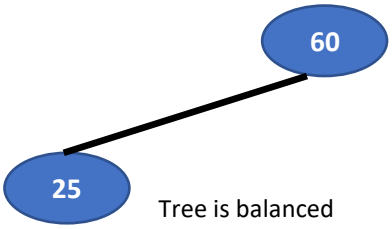
60, 25, 70, 10, 5

Solution-
Step-01: Insert 60

Tree is balanced

Step-02: Insert 25

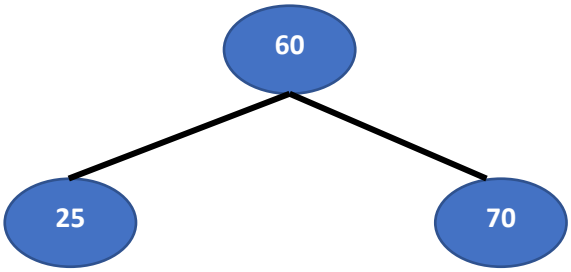
As 25 < 60, so insert 25 in 60's left sub tree.



Tree is balanced

Step-03: Insert 70

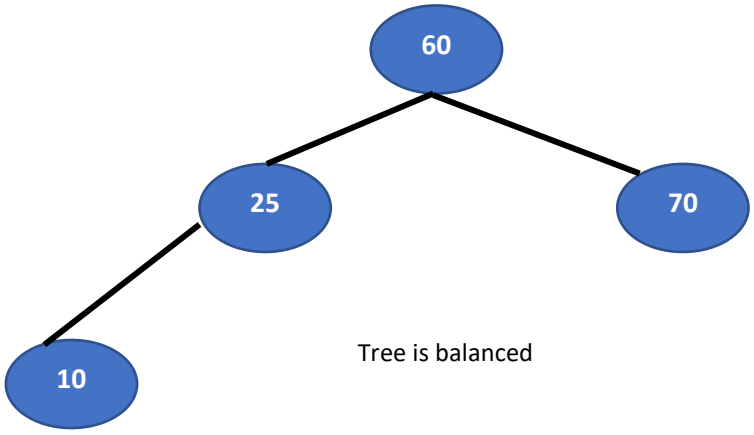
As 70 > 60, so insert 70 in 60's right sub tree.



Tree is balanced

Step-04: Insert 10

As 10 < 60, so insert 10 in 60's left sub tree.
As 10 < 25, so insert 10 in 25's left sub tree.



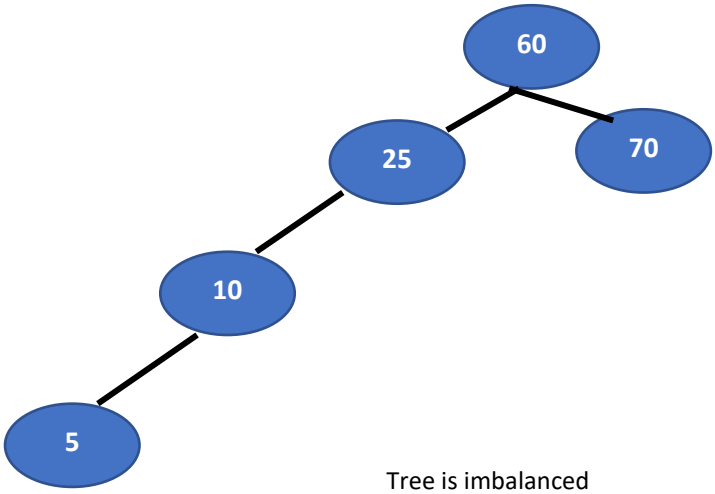
Tree is balanced

Step-05: Insert 5

As 5< 60, so insert 5 in 60's left sub tree.

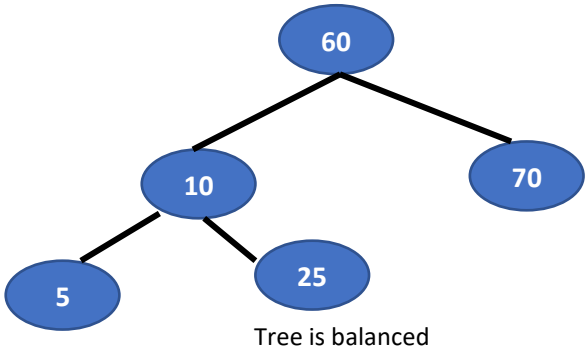
As 5 < 25, so insert 5 in 25's left sub tree.

As 5 < 10, so insert 5 in 10's left sub tree.



Tree is imbalanced

To balance the tree, find the first imbalanced node on the path from the newly inserted node (node 5) to the root node. The first imbalanced node is node 25. Now, count three nodes from node 25 in the direction of leaf node. Then, use AVL tree rotation to balance the tree. After applying RR Rotation



Tree is balanced

Algorithm for insertion:

Step1: insert an element into the tree using Binary search tree method.

Step2: check the balance factor after inserting an element.

Step3: if the balance factor of every node is less than or equal to one then proceed to next operation. Otherwise tree is imbalanced, perform the suitable rotation to make it balanced and then proceed to next operation.

2. Deletion operation

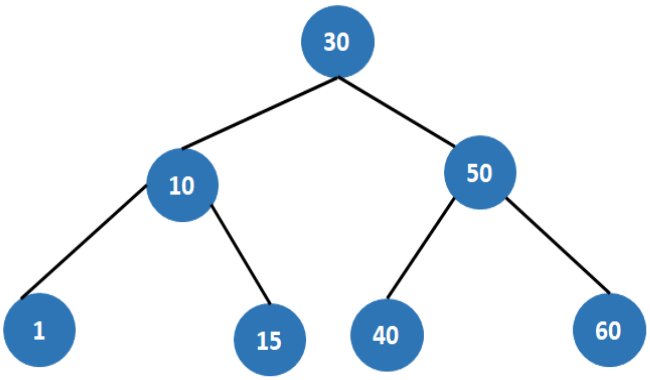
Deletion of a node from a tree decreases the height of the tree, which might lead to unbalanced tree.

The steps of deletion of a node in AVL are –

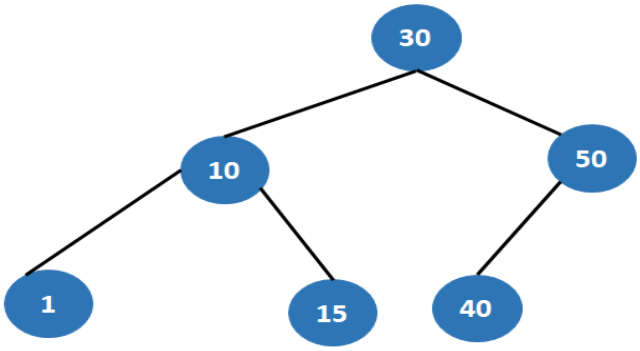
1. Binary Search Tree deletion algorithm is used to delete a given key from the AVL tree.
   2. Update the height of the current node of the AVL tree after the deletion.
   3. Calculate the Balance factor at the current node by finding the difference between height of left sub-tree-height of right sub-tree.
   3a. If the Balance factor > 1 which means the height of the left sub-tree is greater than the height of the right sub-tree. This indicates left-left or left-right case. If the Balance factor of left subtree is greater than 0 then that confirms left-left case, if it less than 0 then that confirms left-right case. If it is equal to 0, then this we can either consider this as a left-left case or as a left-right case. For left-left case, do a right rotation at the current node. For the left-right case, do a left rotation at left child of current node followed by a right rotation at the current node itself.

   3b. If Balance Factor < -1 then that means the height of the right sub-tree is greater than the height of the left sub-tree. This indicates right-right or right-left case. In this case, if balance factor of right sub-tree of current node is less than 0 then this confirms right-right case, if it is greater than 0 then this confirms

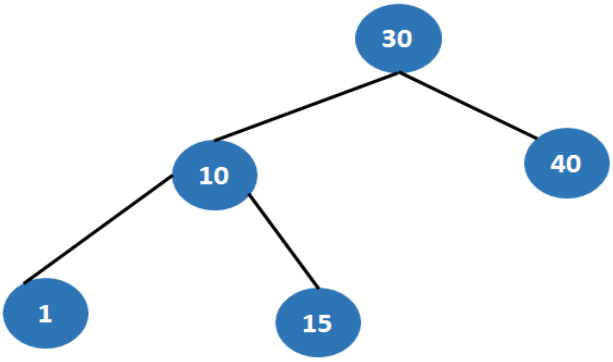
right-left case. If it is equal to 0, then we can either consider this as a right-right case or a right-left case. In this implementation, we will consider this as a right-right case. In right-right case, do a left rotation at the current node. In right-left case, do a right rotation at the right child of current node followed by left rotation at the current node itself.



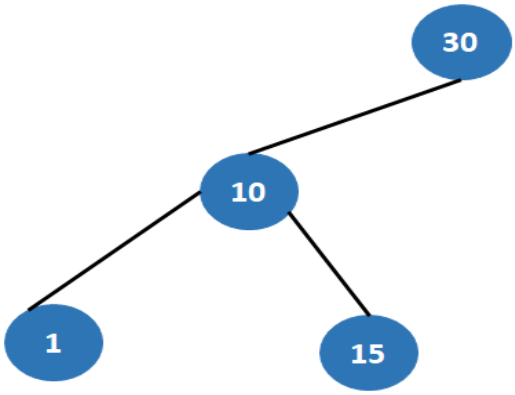
Delete 60



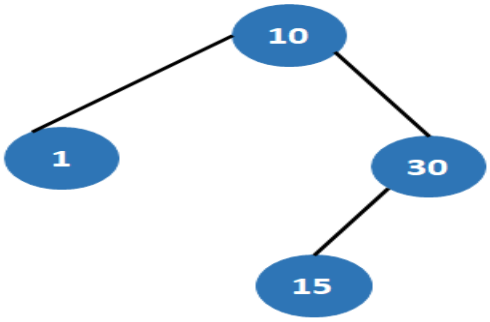
Delete 50



Delete 40



Height is not balanced at node 3, left-left case; perform the right rotation to balance the tree.



The above tree is balanced.

3. Search

Search operations returns a node if the element to be searched is present otherwise does not returns any value

Advantages of AVL Tree

1. AVL trees are height balanced trees, so the operations like insertion and deletion have low complexity.
2. provides faster search options

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPHS

Abstract

Introduction, Properties, Representation of graphs: Adjacency Matrix, Adjacency List

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Overview:

Graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of edges. We will often denote n = |V|, e = |E|. It consists of set of nodes and arcs. Each arch in a graph is specified by a pair of nodes.

There are three types of graph

1. Directed graph

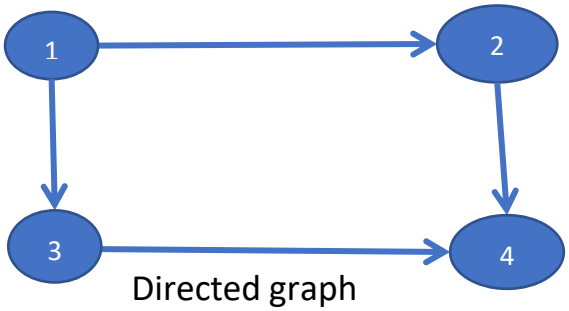
2. Undirected graph

3. Weighted graph

1. Directed graph: A graph G= (V, E) in which every edge is directed is called directed graph. In directed graph pair of vertices representing any edge is ordered. For example in the below fig < v_1, v_2 > and < v_2, v_1> represents the different edge. Directed graph is also known as digraph.

Examples: 1

2.



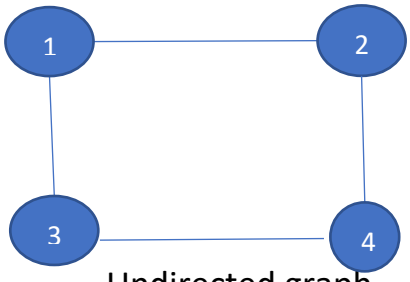
Directed graph

In the above Example.2 V={1,2,3,4} is set of vertices and E={<1,2>,<2,4>,<3,4>,<1,3>} is set of edges. Since all the edges are directed it is a directed graph

2. Undirected graph: The graph G= (V, E) in which every edge is undirected .In undirected graph the pair of vertices representing any edge is unordered.
Example1: The < v_1, v_2 > and < v_2, v_1> represents the same edge.

Example.1

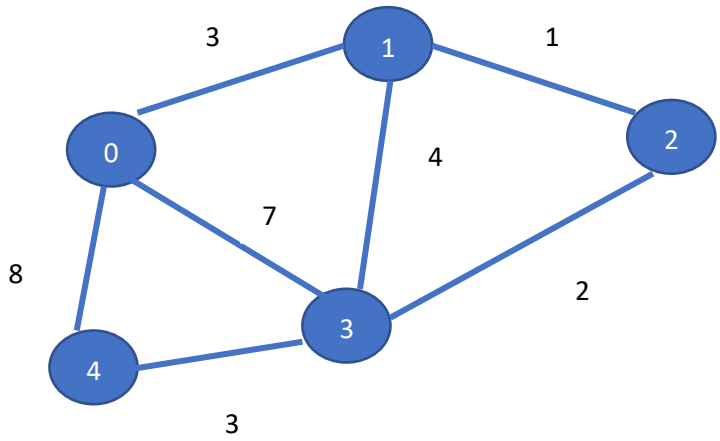
Example.2



Undirected graph

In the above example.2 V={1,2,3,4} is the set of vertices and E={(1,2),(1,3),(2,1),(2,4),(4,2),(3,4),(4,3),(3,1)} is the set of edges. Since all the edges are undirected the above graph is undirected graph

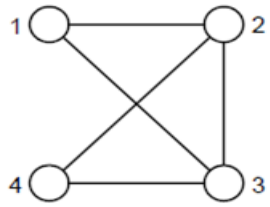
3. Weighted graph or Network: A weighted graph is a graph where each edge has a numerical value called weight. Given below is an example of a weighted graph. Weighted graph can be directed and undirected.



Weighted graph

Graph terminologies:

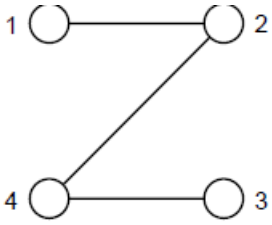
Adjacent Nodes: When there is an edge from one node to another then these nodes are called adjacent nodes.



Adjacent Node

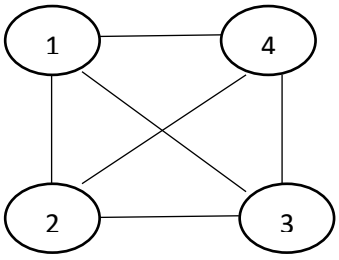
In the above graph 2 is adjacent to 3 and 4 is adjacent to 3 and so on

Path: A path from edges u_0 to a node u_n is a sequence of nodes u_0, u_1, u_2.....u_{n-1},u_n. Here u_0 is adjacent to u_1, u1 is adjacent to u_2 and u_{n-1} is adjacent to u_{n-1}. In the below graph, the path from vertex 1 to 3 is denoted by (1,2,4,3) which can also be written as (1, 2), (2, 4), (4, 3).



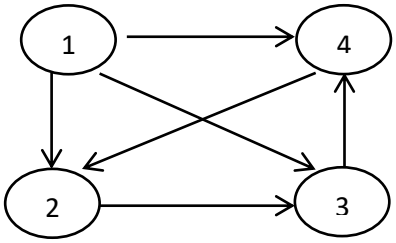
Path

Length of the path: Length of the path is the total number of edges included in the path from the source node to destination node.



Undirected graph

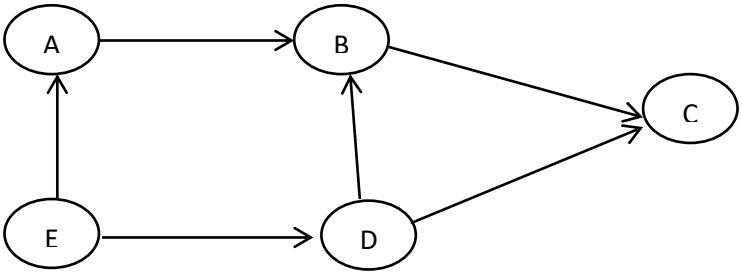
For example in the above undirected graph ,the path(1,2,3,4) has length 3 since there are three edges(1,2),(2,3),(3,4). The path 1, 2, 3 has length 2 since there are 2 edges (1, 2), (2, 3).



Directed graph

Similarly in the directed graph, the path <1,2,3,4> has length 3 since there are 3 edges <1,2>,<2,3>,<3,4> and the path <1,2,3> has length 2 since there are 2 edges <1,2>,<2,3>.

Degree: In an undirected graph, the total number of edges linked to a node is called degree of the node. In digraph there are 2 degrees for every node called indegree and outdegree.

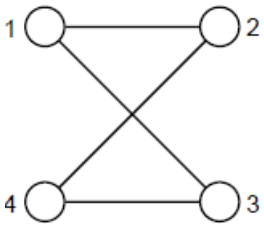


Degree

In-degree: The in-degree of a node n is the total number of arcs that have n as the head. For example C is receiving 2 edges so indegree of C is 2.

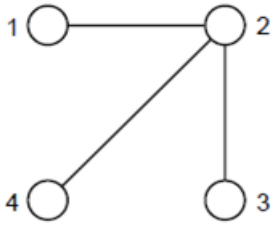
Outdegree: The outdegree of a node n is the total number of arcs that have n as the tail. For example outdegree of D is 1.

Cycle graph: A path from node to itself be called a cycle or cycle is path in which first and last vertices are same. A graph with at-least one cycle is called cyclic graph. A directed acyclic graph is called dag. For example the path<1, 2, 4, 3, and 1> is cycle since the first node and the last node are same. It can be represented as <1,2>,<2,4>,<4,3>,<3,1>.



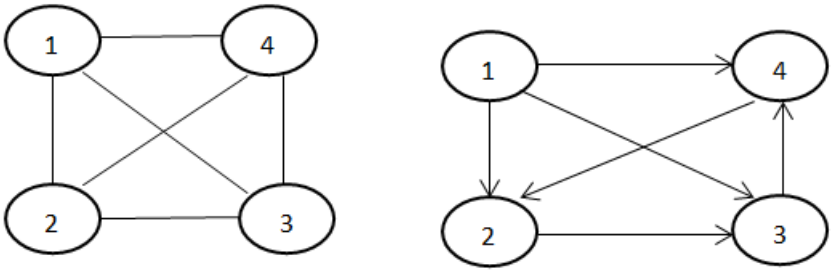
Cycle

Acyclic graph: A graph with no cycle is called acyclic graph. Tree is an acyclic graph.



Acyclic graph

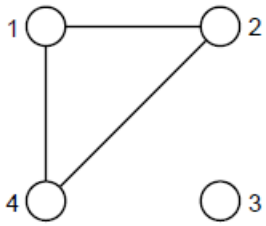
Connected: A graph is called connected if there is a path from any vertex to any other vertex. A tree is defined as connected undirected graph with no cycles. Example



Connected graph

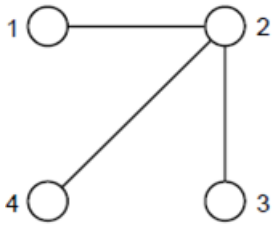
Disconnected graph: if there exist atleast one vertex in a graph that cannot be reached from the other vertices in the graph,then such graph is aclled

disconnected graph.In the below example vertex 3 cannot be reached by any other vertices in the graph.



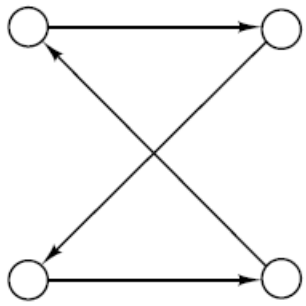
Disconnected graph

Tree: A tree is defined as connected undirected graph with no cycles.



Tree

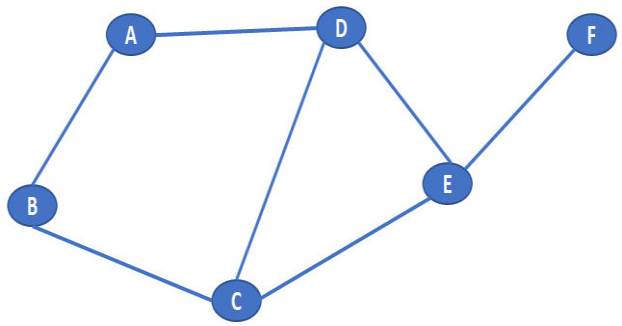
Directed cycle: In directed graph all the edges in a path or cycle have same direction



Directed cycle

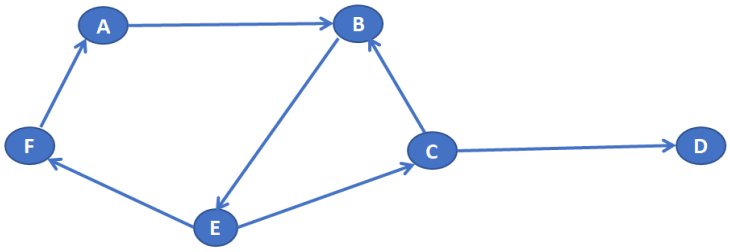
Properties of graph (referred from geeks for geeks):

1. **Undirected graph**
   a. The number of possible pairs in an m vertex graph is m*(m-1).
   b. The number of edges in an undirected graph is m*(m-1)/2 since the edge (u, v) is same as the edge(v, u).



Undirected graph

2. **Directed graph**
   a. The number of possible pairs in an m vertex graph is m*(m-1)
   b. The number of edges in an directed graph is m*(m-1) since the edge(u, v) is not the same as the edge(v, u)
   c. The number of edges in an directed graph is<=m*(m-1)



Directed graph

Representation of graph

Graph representation is the method to store the graphs in computer memory. Graph is represented using a set of vertices, and for each vertex, the vertices are

directly connected to it by an edge. For a weighted graph weight will be associated with each edge. The choice of graph representation depends on the application and the types of operation performed and ease of use.

There are 2 ways of representing the graph

1. Adjacency matrix.
2. Adjacency List.

1. Adjacency Matrix:

Let G= (V, E) be a graph where V is the set of vertices and E is the set of edges. Let N be the number of vertices in the graph G. The adjacency matrix A of a graph G is defines as

A[i][j] = 1 if there is an edge from vertex i to j

0 if there is no edge from vertex i to j

The adjacency matrix of a graph is a Boolean square matrix or bit matrix with n rows and n columns with entries zeros and ones.

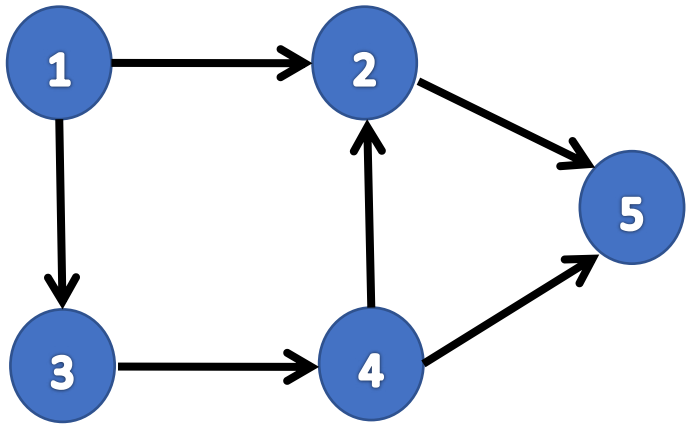
In an undirected graph, if there exist an edge (i, j),then A[i][j] and A[j][i] is made one since (i, j) is similar to (j, i).

In the directed graph if there exist an edge <i,j>, then A[i][j]=1

If there exist no edge between vertex i and j then A[i][j]=0

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

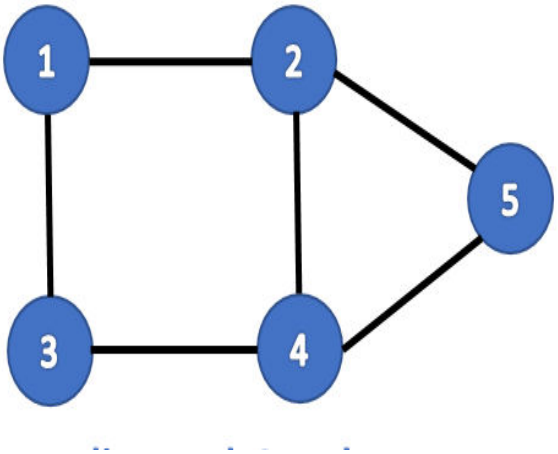
1. Directed graph



Directed graph

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	0	1	0	0	1
5	0	0	0	0	0

2. Undirected graph

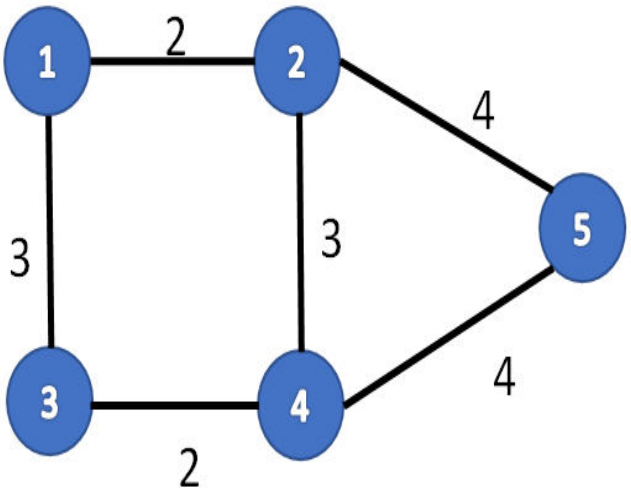


Undirected graph

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	1
3	1	0	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

3. Weighted graph:

In the weighted graph, distance or cost between the nodes are represented on the edge cost/distance value specified on the edge between adjacent nodes are used for representation in the adjacency matrix.



Weighted graph

	1	**2**	**3**	**4**	**5**
1	0	2	3	0	0
2	2	0	0	3	4
3	3	0	0	2	0
4	0	3	2	0	4
5	0	4	0	4	0

C representation of graph:

The number of nodes in the graph is constant, that is arcs may be added or deleted but the nodes may not. A graph with 25 nodes could be declared as

A graph with 25 nodes can be declared as

```
#define MAX 25

struct node

{

                    //information associated with each node

};

struct arc

{

                    int adj;//information associated with each arc

};

struct graph

{

                    struct node nodes[MAX];

                    struct arc arcs[MAX][MAX];

};

struct graph g;
```

Each node in a graph is represented by an integer number from zero to MAX-1, and the array field nodes represent the appropriate information assigned to each node. The array field an arcs is a two dimensional array representing every possible ordered pair of nodes. The value of g.arcs[i][j].adj is either one or zero depending on whether a node i is adjacent to j.

A weighted graph with fixed number of nodes can be declared as

```
struct arc
{
                    int adj;
                    int weight;
};
struct arc g[MAX][MAX];
```

Drawback of adjacency matrix representation of graph is it requires advance knowledge of the number of nodes. A new matrix must be created for each addition and deletion of a node. Adjacency matrix representation is inefficient in real world situation where a graph may have hundreds of nodes. An alternate solution to this is to use adjacency list.

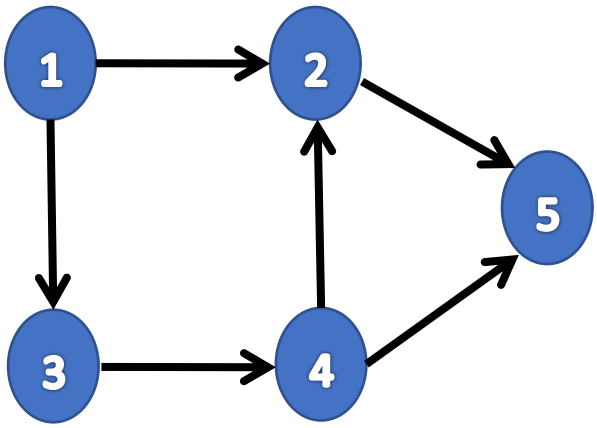
Function to read adjacency matrix

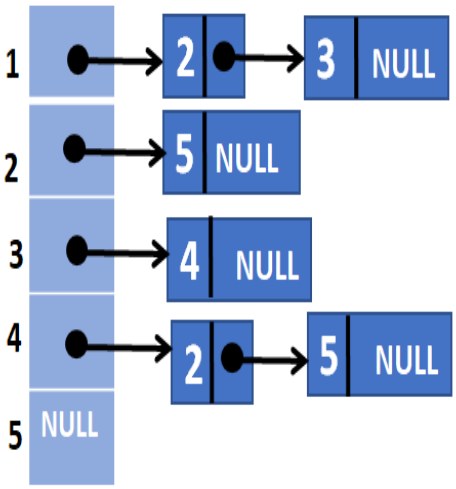
```
void read_admat(int A[10][10], int n)
{
    int I,j;
     for(i=0; i<n; i++)
    {
          for(j=0; j<n; j++)
           {
              scanf("%d", &A[i][j]);
            }
     }
}
```

2. Adjacency Linked List:

Adjacency linked list is an array of n linked list in which every vertex of a graph contains the list of adjacent vertices

1. Directed graph





Nodes adjacent to 1 are 2 and 3

Nodes adjacent to 2 is 5

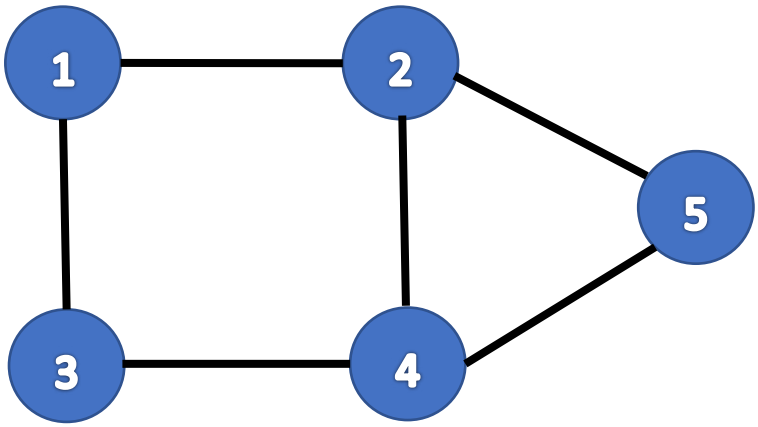
Nodes adjacent to 3 is 4

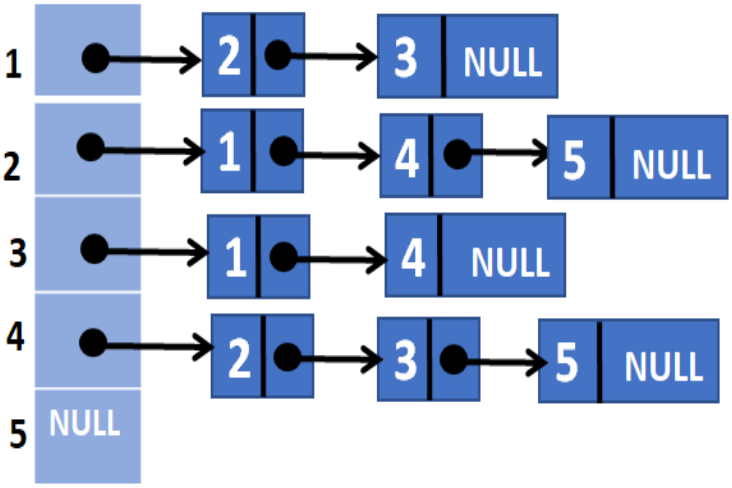
Nodes adjacent to 4 is 2 and 5

No Nodes adjacent to 5

Directed graph

2. Undirected graph





Nodes adjacent to 1 are 2 and 3

Nodes adjacent to 2 are 1, 4 and 5

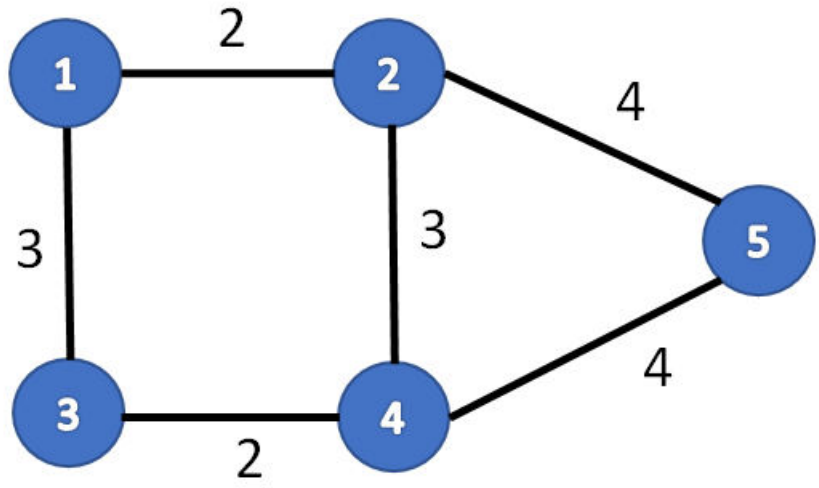
Nodes adjacent to 3 are 1 and 4

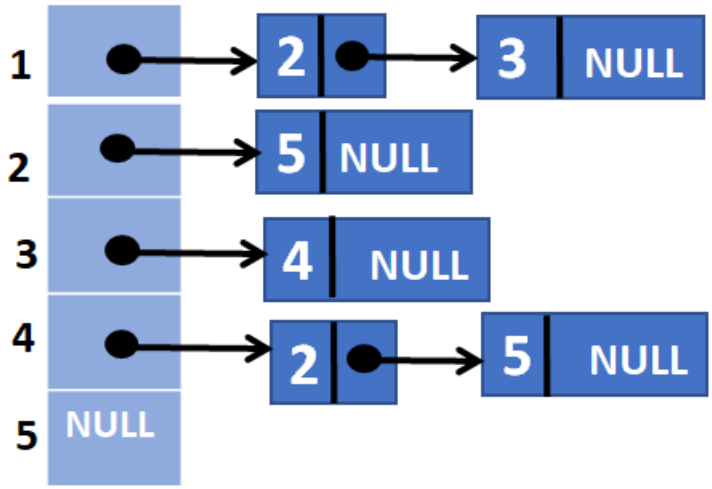
Nodes adjacent to 4 are 2, 3 and 5

No Nodes adjacent 5

Undirected graph

3. Weighted graph





Weighted graph

C representation of adjacency list

Two types of allocated nodes are needed since each graph carries some information.(arcs does not carry any info).

1. header nodes

arcptr	info	nextnode

A Sample header node

Each header node contains an info field and two pointers. The first of these to is to adjacency list of arcs emanating from the graph node, and the second is to next header node in the graph.

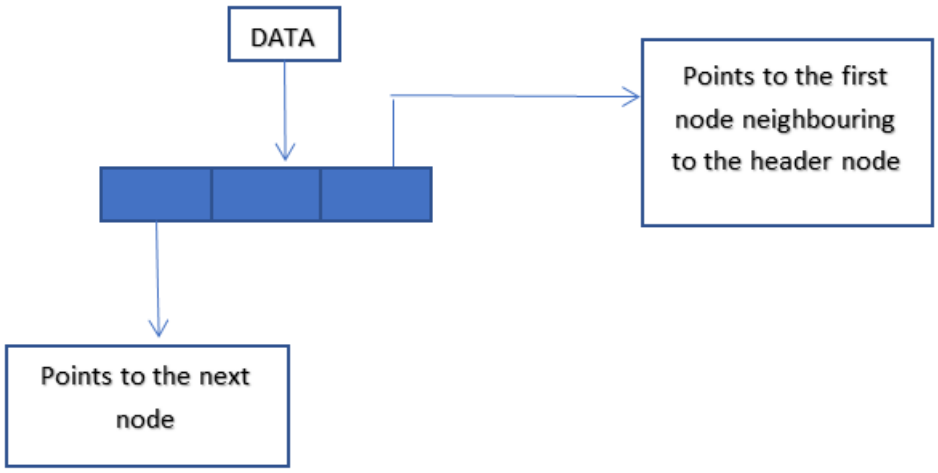
2. Adjacency list nodes

Ndptr	nextarc

A sample list node representing an arc

Each arc node contains two pointers, one for nextarc node in adjacency list and the other to the header node representing the graph that terminates the arc. Refer to the diagram from textbook T1 8.3.1 page no 543.

Header nodes and list nodes have different formats and must be represented by different structures. However for simplicity we make assumption that both header and the list nodes have same format and contain two pointers and a single integer information field.



1. Using array implementation the nodes are declared as

```
#define MAX 50

struct node

{

    int info;

     int point;

      int next;

};

struct node NODE[MAX];
```

In the case of header node, node[p] represents a graph node A, node[p].info represents the information associated with the graph node, node[p].next points to the next graph node and node[p].point points to the first list node representing an arc emanating from A. In the case of a list node, node[p] represents an arc <A, B>, node[p].info represents the weight of arc, node[p].point points to the header node representing the graph node B.

2. Using dynamic implementation

```
struct node

{

    int info;

    strut node *pointer;

    struct node *next;

};

struct node *nodeptr;
```

Function to read the adjacency list

```
void read_adlist(nodeptr a[],int n)
{
    int I,j,m ele;
    for(i=0;i<n;i++)
     {
        printf("enter the number of nodes adjacent to %d:",i);
         scanf("%d",&m);
         if(m==0) continue;
         printf("Enter the nodes adjacent to %d",i);
        for(j=0;j<m;j++)
        {
         scanf("%d",&ele);
        a[i]=insert_rear(ele,a[i]); //Function to insert an element at the rear of the list
         }
    }
}
```

Advantages of using Adjacency list:

1. Adding a new vertex is easy

2. Graphs can be stored in more compact form

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPHS

Abstract

Implementation of Graph using Adjacency Matrix

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Implementation of graph using adjacency matrix

C representation of graph:

A graph with 25 nodes can be declared as

```
#define MAX 25
struct node
{
                    //information associated with each node
};
struct arc
{
                    int adj;//information associated with each arc
};
struct graph
{
    struct node nodes[MAX];
    struct arc arcs[MAX][MAX];
};
struct graph g;
```

Each node in a graph is represented by an integer number from zero to MAX-1, and the array field nodes represent the appropriate information assigned to each node. The array field an arc is a two dimensional array representing every possible ordered pair of nodes.

The different operations performed on adjacency matrix are

1. To add an arc from node1 to node2

```
void join(int adj[]MAX],int node1,int node2)
 {
    adj[node1][node2]=TRUE;
}
```

2. To delete an arc from node1 to node2 if it exists

```
void remv(int adj[][MAX],int node1,int node2)
{
  adj[node1][node2]=FALSE;
}
```

3. To check whether arc exists between node1 and node2

```
int adjacent(int adj[][MAX],int node1,int node2)
{
   return((adj[node1][node2]==TRUE)?TRUE:FALSE);
}
```

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPHS

Abstract

Implementation of Graph using Adjacency List.

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Implementation of adjacency List:

C representation of adjacency list

1.using array implementation

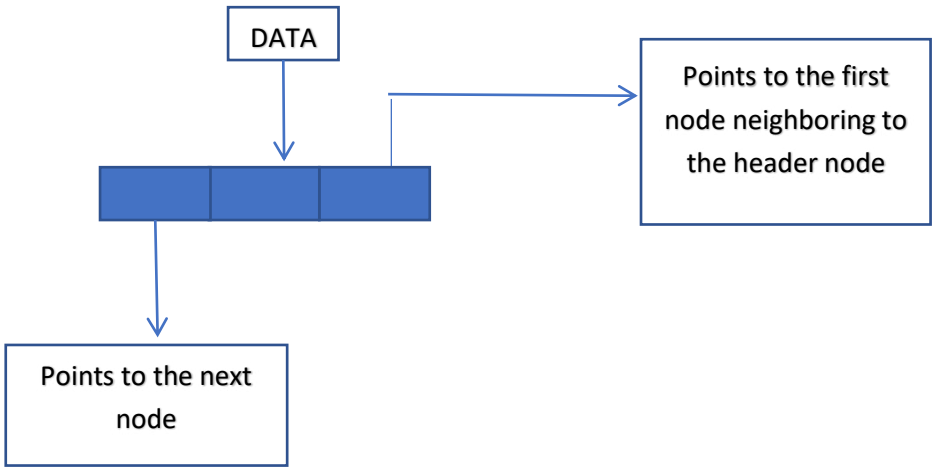
```
#define MAX 50
struct node
{
  int info;
  int point;
  int next;
};
struct node NODE[MAX];
```

2.Using dynamic implementation

```
struct node
{
   int info;
  strut node *pointer;
   struct node *next;
};
struct node *nodeptr;
```

Two lists are maintained for adjacency of list. The first list is used to store information of all the nodes. The second list is used to store the information of adjacent nodes for each node of a graph.

Header node is used to represent each list, which is assumed to be the first node of a list as shown below.



The different operations performed on adjacency list are

1. To create an arc between two nodes

```
void jointwt(int p,int q,int wt)
    {
        int r,r2;
        r2=-1;
       r=node[p].point;
       While(r>=0 && node[r].point!=q)
        {
            r2=r;
            r=node[r].next;
         }
       If(r>=0)
        {
            node[r].info=wt;
```

```
                    return;
                }
            R=getnode();
            node[r].point=q;
            node[r].next=-1;
             node[r].info=wt;
             (R2<0)?(node[p].point=r):(node[r2].next=r);
}
```

2. To remove an arc between two nodes if it exists:

```
Void remv(int p,int q)
{
    int r,r2;
   r2=-1;
   r=node[p].point;
  while(r>=0 ** node[r].point !=q)
 {
     r2=r;
    r=node[r].next;
}
If(r>=0)
{
   (r2<0)?(node[p].point=node[r].next):(node[r2].next=node[r].next);
   freenode(r);
   return;
}
```

```
}
```

3. Checks whether a node is adjacent to another node

```
int adjacent(int p,int q)
{
   int r;
    r=node[p].point;
   while(r>=0)
        If(node[r].point==q)
            return(TRUE)
         else
            r=node[r].next;
              return(FALSE);
}
```

4. Find a node with a specific information in a graph:

```
int findnode(int graph,int x)
{
        int p;
        p=graph;
        while(p>=0)
                if (node[p].info==x)
                        return(p);
                else
                        p=node[p].next;
```

```
        return(-1);
}
```

5. Add a node with specific information to a graph.

```
int addnode(int *graph,int x)
{
    int p;
    p=getnode();
    node[p].info=x;
    node[p].point=-1;
    node[p].next=*graph;
    *graph=p;
    return(p);
}
```

PES UNIVERSITY

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPH TRAVERSAL

Abstract

Traversals, Breadth first search, Depth first search, Implementation, Advantages, Disadvantages and Applications

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Graph traversal:

Many graph algorithms requires a systematic way to examine all the nodes and edges of a graph. Traversing is a process of visiting each and every node of a graph.

Defining a traversal that relates to the structure of a graph is more complex than for a list or tree due to following reasons

1. A graph has no first node or root node. Therefore the graph can be traversed from any node as starting node and once the starting node is selected there may be few nodes which are not reachable from the starting node. Traversal algorithm faces the problem of selecting another starting point.

2. In a graph to reach a particular node multiple paths may be available.

3. There is no natural order among the successors of particular node. Thus there is no prior order in which the successors of a particular node should be visited.

4. A graph may have more than one predecessor and therefore there is a possibility for a node to be visited before one of its predecessors. For example if node A is a successor of nodes B and C, A may be visited after B but before C.

There are two standard algorithms for traversal of graphs. They are:

• Breadth first search (BFS): The BFS will use a queue as an auxiliary structure to hold nodes for future processing

• Depth first Search (DFS): DFS algorithm traverses a graph in a depth-ward motion and uses a stack to remember to get the next vertex to start a search, when dead end occurs in any iteration.

Depth First Search:

In Depth first search method, an arbitrary node of a graph is taken as starting node and is marked as visited. On each iteration, the algorithm proceeds to an unvisited vertex which is adjacent to the one which is currently in. Process continues until a vertex with no adjacent unvisted vertex is found. When the dead end is reached the control returns to the previous node and continues to visit all the unvisited vertices. The algorithm ends after backing up to the starting vertex, when the latter being a dead end. The algorithm has to restart at an arbitrary vertex, if there still remain unvisited vertices.

This process is implemented using stack. A vertex is inserted into the stack when the vertex is visited for the first time and deletes a vertex when the visit of the vertex ends.

Iterative procedure to traverse a graph is shown below:

1. Select a node u as a start vertex, push u onto the stack and mark it as visited.

2. While stack is not empty

    For vertex u on top of the stack, find the next immediate adjacent vertex

    If v is adjacent

        If a vertex v is not visited then

            Push it on to stack and mark it as visited

        else

            Ignore the vertex

        End if

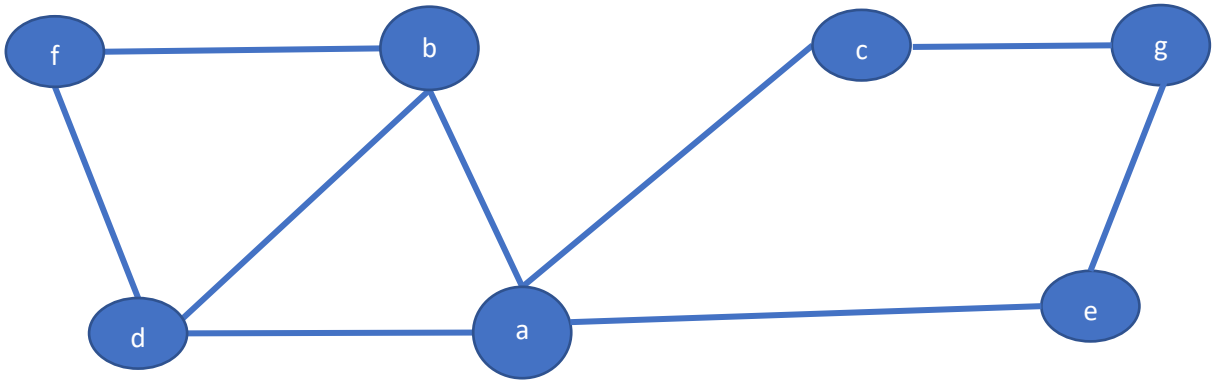
    Else

        Remove the vertex from the stack

    End if

   End while

3. Repeat step1 and step2 until all the vertices in the graph are visited.



	Stack	**V(adjacent vertex)**	**Nodes visited(S)**	**Pop(stack)**
Initial step	a	-	a	
	a	b	a,b	-
	a,b	d	a,b,d	-
	a,b,d	f	a,b,d,f	-
	a,b,d,f	-	a,b,d,f	f
	a,b,d	-	a,b,d,f	d
	a,b	-	a,b,d,f	b
	a	c	a,b,d,f,c	-
	a,c	g	a,b,d,f,c,g	-
	a,c,g	e	a,b,d,f,c,g,e	-
	a,c,g,e	-	a,b,d,f,c,g,e	e
	a,c,g	-	a,b,d,f,c,g,e	g
	a,c	-	a,b,d,f,c,g,e	c
	a	-	a,b,d,f,c,g,e	a

Step 1: Insert a source vertex *a* onto the stack and add *a* to node visited(S)
Step 2: Push the node adjacent to *a* that is *b* onto the stack (alphabetical order) and add to S if not present in S (Also note that in DFS only one adjacent vertex which is not visited is considered).
Step 3: push the node adjacent to *b*, that is *d* onto stack and add to S if not present.
Step 4: push the node adjacent to *d* that is *f* onto the stack and add to S if not visited.
Step 5: Since the node adjacent to *f* is already visited. Pop *f* from the stack.
Step 6: Since the node adjacent to *d* is already visited, pop *d* from the stack.
Step 7: Pop *b* from the stack.
Step 8: Since there is adjacent node from *a* to *c* we add c to stack and S
Step 9: Push the node adjacent to *c* that is *g* to stack and add to S.
Step 10: push the node adjacent to *g* that is e to stack and add to S.
Step 11: Since the node adjacent to *e* is visited, pop *e* from the stack.
Step 12: Since the node adjacent to *g* is visited, pop *g* from stack
Step 13: Since the node adjacent to *c* is visited, pop *c* from stack.
Step 14. Pop *a* from stack
Thus nodes reachable from *a* are *a, b, c, d, e, f, g.*

Advantages of Depth First Search:

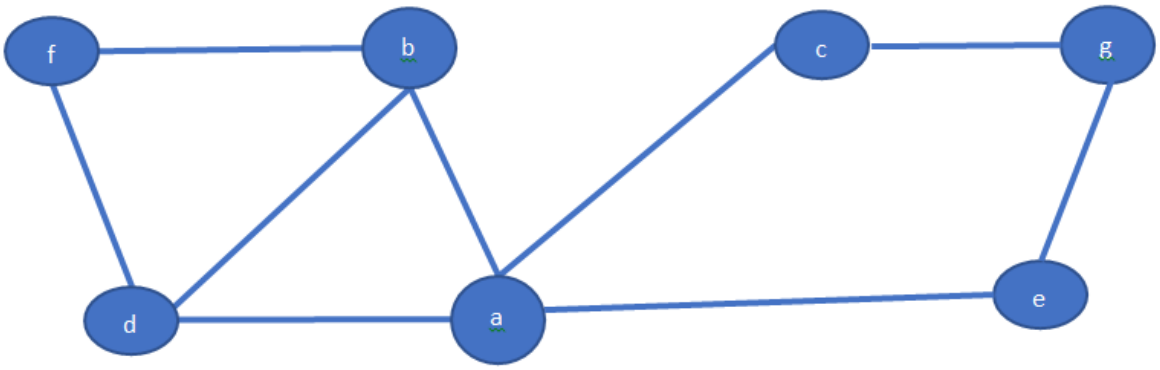
1. Consumes less memory
2. Finds the larger distant element from source vertex in less time.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever. To eliminate this we keep a list of states previously visited, and never permit search to return to any of them.
2. We cannot come up with shortest solution to the problem.

Breadth First Search: Breadth first search uses the queue data structure for traversing vertices of the graph. Any node of the graph can act as the starting node. Using the starting node, all other nodes of the graph are traversed. To prevent repeated visit to the same node, an array is maintained which keeps track of visited node.

In breadth first search algorithm the queue is initialized with the starting vertex and is marked as visited. On each iteration, the algorithm identifies all the unvisited vertices adjacent to the front vertex, marks them as visited and adds them to the queue; after that ,the front vertex is removed from the queue.



	U(deleted from queue)	V(adjacent vertex)	Nodes visited(s)	Queue
Initial node			a	a
	a	b,c,d,e	a,b,c,d,e	b,c,d,e
	b	a,d,f	a,b,c,d,e,f	c,d,e,f
	c	a,g	a,b,c,d,e,f,g	d,e,f,g
	d	a,b,f	a,b,c,d,e,f,g	e,f,,g,
	e	a,g	a,b,c,d,e,f,g	f,g
	f	b,d	a,b,c,d,e,f,g	g
	g	c,e	a,b,c,d,e,f,g	empty

Step 1: Insert a source vertex *a* into the queue and add *a* to node visited(S).

Step 2: Delete an element a from queue and find all the adjacent nodes of .The nodes adjacent to *a* are *b, c, d* and *e*. Add these nodes to S only if it is not present in S. So we add *b, c, d* and *e* to S

Step 3: Delete b from queue and find all the adjacent nodes to *b* i.e. *a, d, f* and add to S only if it not present in S, so only *f* is added to S.

Step 4: Delete *c* from queue, Find all the adjacent nodes to *c*, and add those nodes to S if it is not present in S. So add only *g* to S.

Step 5: Delete *d* from queue, Find all the adjacent nodes of *d* and add those to S if not present. All the adjacent nodes of *d* is already in S.

Step 6: Delete *e* from queue, find all the adjacent nodes of *e*. i.e *a* and *g* .Since *g* is not in S we add *g* to S.

Step7: Delete *f* from queue; find all the nodes adjacent to *f.* add to S if not present in S

Step 8: Delete *g* from queue, find all the adjacent nodes to *g*. Since all the adjacent nodes of *g* are already present in S we don't add to S.

And the queue is empty so the nodes reachable from Source *a*: *a, b, c, d, e, f, g*

Pseudo code:

```
Insert source u to queue
Mark u as visited
While queue is not empty
        Delete a vertex u from queue
        Find the vertices v adjacent to u
        If v is not visited
            Print v
            Mark v as visited
            Insert v to queue
        End if
```

End while

Advantages of Breadth First Search:

1. BFS algorithm is used find the shortest path between vertices
2. Always finds optimal solutions.

Disadvantages of BFS:

All of the connected vertices must be stored in memory. So consumes more memory

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

APPLICATION

Abstract

Graph representation: Representation of Computer Network Topology, Different types of Topology

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

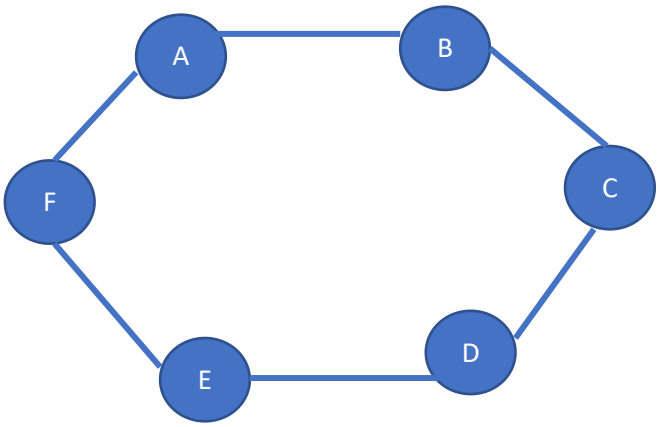
Graph representation: Representation of Network Topology:

Graph data structure is mainly in telecommunication and Computer Networks. Networking uses the Notation G(N,L) instead of G(V,E) for a graph were N is the set of nodes and L is the set of links.

Topology is the order in which nodes and edges are arranged in the network.

The different types of Network topology are

1. Ring topology (cycle): Ring topology is also called as cycle graph. A simple graph with two degrees of vertices is called cycle graph. In ring Topology each vertex is connected to other two vertices, So that they form a network which looks like a circle or ring.

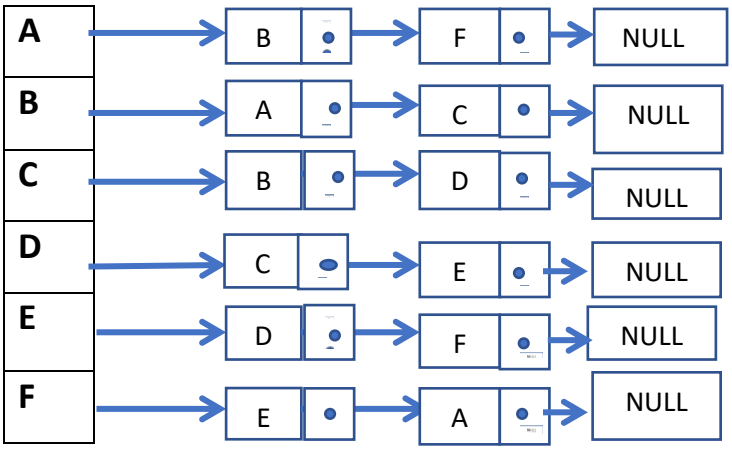


Ring topology

Adjacency matrix for the above graph

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

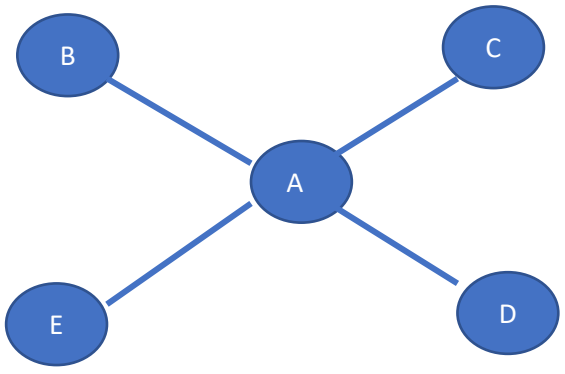
Adjacency list representation for ring topology



2. Star topology: Star topology is a network topology in the form of merging from the central vertex to each vertex.

A graph of V vertices represents a star topology if it satisfies the following three conditions:

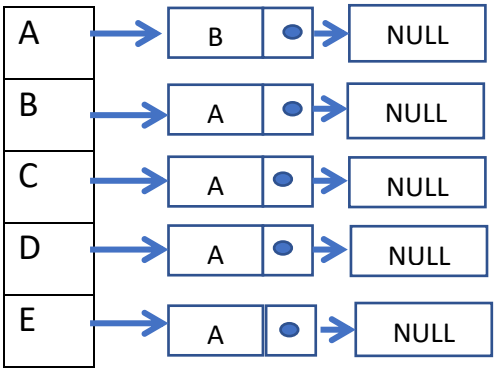
1. The centre node has degree n– 1.
2. All nodes except the central node have degree 1.
3. No of edges = No of Vertices – 1.



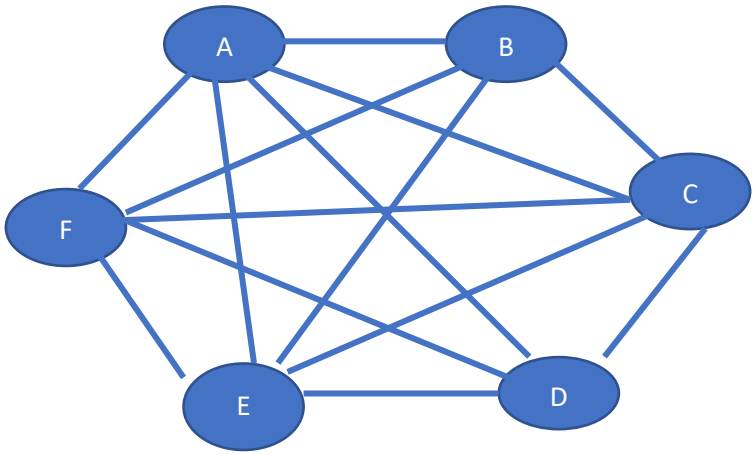
Star topology

Adjacency matrix representation for Star topology

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



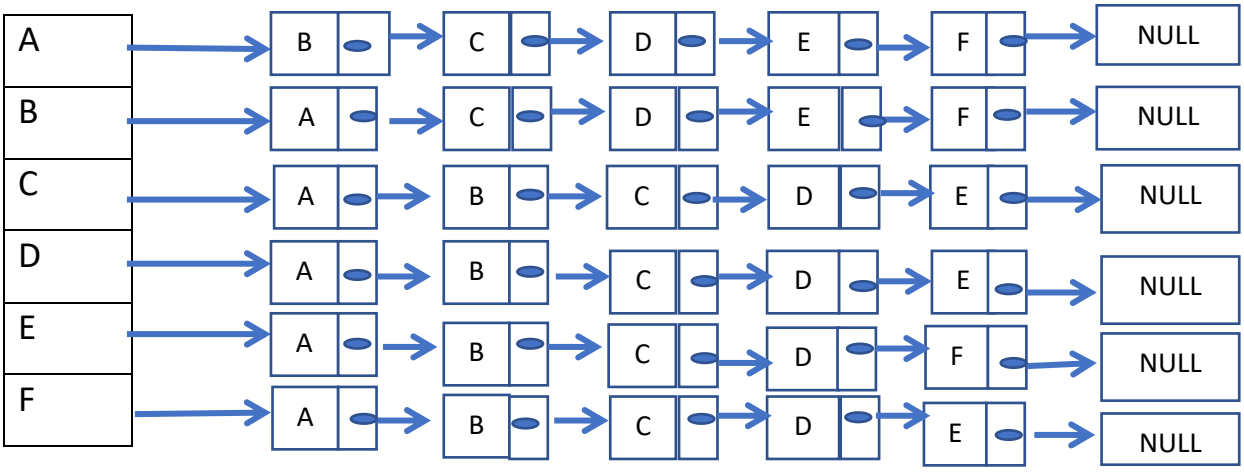
3. **Mesh topology**: Mesh topology uses a complete graph form. All the nodes in mesh topology are interconnected. Every node has degree n-1



Mesh Topology

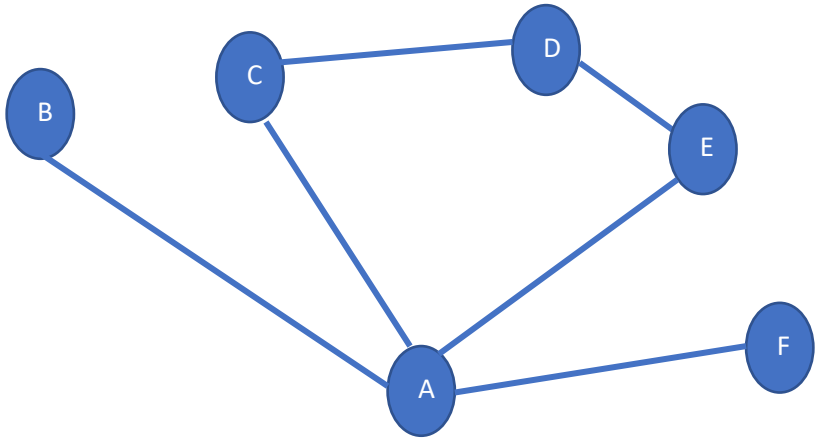
Adjacency matrix for the above graph

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



4. Hybrid Topology

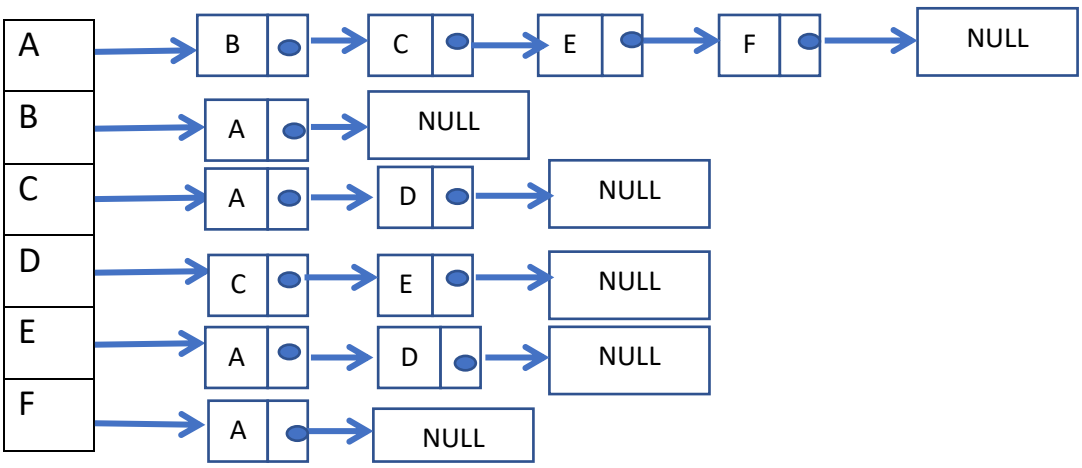
Hybrid topology is the combination of two or more topologies.



Hybrid Topology

Adjacency matrix for the above graph

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


A
B
C
D
E
F
B
C
E
F
NULL
A
NULL
A
D
NULL
C
E
NULL
A
D
NULL
A
NULL

Department of Computer science and Engineering
PES UNIVERSITY
UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Applications of BFS and DFS

Abstract

Finding the path in a network using dfs and bfs traversal method.

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Applications of BFS and DFS:

- The different applications of DFS are
- Detecting whether a cycle exist in graph.
- Finding a path in a network
- Topological Sorting: Used for job scheduling
- To check whether a graph is strongly connected or not: A directed graph is said to be strongly connected if there exist a path between every pair of vertex.

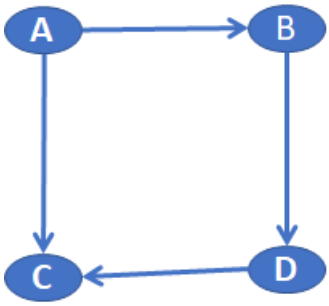
Applications of BFS are

- Finding the shortest path
- Social Networking websites like twitter, Facebook etc.
- GPS Navigation system
- Web crawlers
- Finding a path in network
- In Networking to broadcast the packets.

Application of DFS and BFS

Finding the path in a Network

Given an graph with N vertices and E edges and two vertices(x,y) from the graph, we need to print the path between these two vertices if the path exists and no otherwise .



The path from A to D=1.A→B→D

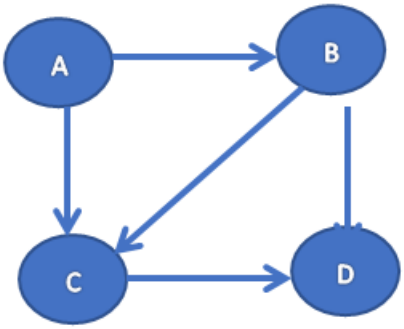
There exists no path from C to A

This can be achieved in 2 ways
1.DFS (Depth First Search)
2.BFS (Breadth First Search)

Function to traverse the graph using bfs

Finding all path in a network

- Path is defined as route between any two nodes. For Example given a directed graph, a source vertex s and a destination vertex d, print all the paths from s to d.
  For Example



Directed Graph

	A	B	C	D
A	0	1	1	0
B	0	0	1	1
C	0	0	0	1
D	0	0	0	0

Output:

Path from A to D: A->B->D

A->C->D

A->B->C->D

/*Program to print all the paths from a given source to vertex using dfs- matrix representation*/

```
#include<stdio.h>
#include<stdlib.h>
int a[10][10],n,p=0;
void read_ad_mat()
{
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
```

```
    }
}


void printall(int u,int d,int visited[10],int path[10])
{
    visited[u]=1;
    path[p]=u;
    p++;
    if(u==d)
    {

        for(int i=0;i<p;i++)
        {
            printf("%d ",path[i]);
        }
        printf("\n");
    }
    else{
        for(int v=0;v<n;v++)
            if(a[u][v]==1 && visited[v]==0)
            {
                printall(v,d,visited,path);
            }
    }
```

```
        p--;
        visited[u]=0;
    }

    void printpath(int s,int d)
    {
        int visited[10];
        int path[10];
        int p=0;
        for(int i=0;i<n;i++)
                visited[i]=0;
        printall(s,d,visited,path);
    }


    int main()
    {
        int i,so,de;
        printf("enter the number of nodes\n");
        scanf("%d",&n);
        printf("enter the adjacency list\n");
        read_ad_mat();
        printf("enter the source and destination\n");
        scanf("%d%d",&so,&de);
        printpath(so,de);
```

```
    }
```

/*program to find all the paths from a given source to destination using dfs list representation*/

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

struct node
{
        int info;
        struct node *link;
};
typedef struct node *NODE;
NODE a[10];
int p=0;

NODE getnode()
{
        NODE x;
        x=(NODE) malloc(sizeof(struct node));
        if(x==NULL)
        {
                printf("out of memory\n");
                exit(0);
        }
        return(x);
```

```
}
NODE insert_rear(int ele,NODE first)
{
        NODE temp;
        NODE cur;
        temp=getnode();
        temp->info=ele;
        temp->link=NULL;
        if(first==NULL)
                return temp;
        cur=first;
        while(cur->link!=NULL)
                cur=cur->link;
        cur->link=temp;
        return first;
}

void read_ad_list(NODE a[],int n)
{
        int i,j,m,ele;
        for(i=0;i<n;i++)
        {
                printf("enter the number of nodes adjacent to %d\n",i);
                scanf("%d",&m);
                if(m==0)
```

```
                continue;
            printf("enter the nodes adjacent to %d\n",i);
            for(j=0;j<m;j++)
            {
                scanf("%d",&ele);
                a[i]=insert_rear(ele,a[i]);
            }
        }
}

void printall(int u,int d,int visited[10],int path[10])
{
        visited[u]=1;
        path[p]=u;
        p++;
        if(u==d)
        {

            for(int i=0;i<p;i++)
            {
                printf("%d ",path[i]);
            }
            printf("\n");
        }
        else{
```

```
                for(NODE temp=a[u];temp!=NULL;temp=temp->link)
                        if(!visited[temp->info])
                        {
                                printall(temp->info,d,visited,path);
                        }
        }
        p--;
        visited[u]=0;
}

void printpath(int s,int d,int n)
{
        int visited[10];
        int path[10];
        int p=0;
        for(int i=0;i<n;i++)
                visited[i]=0;
        printall(s,d,visited,path);
}


int main()
{
        int i,so,de;
        printf("enter the number of nodes\n");
```

```
        scanf("%d",&n);
        printf("enter the adjacency list\n");
        read_ad_list(a,n);
        printf("enter the source and destination\n");
        scanf("%d%d",&so,&de);
        printpath(so,de,n);
}
```

PES UNIVERSITY

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Applications of BFS and DFS

Abstract

Applications of BFS, Applications of DFS, Implementation of Connectivity of graph in directed and undirected graphs.

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

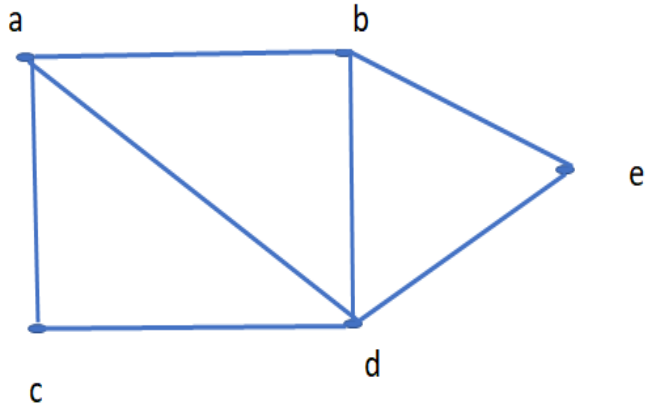
Applications of BFS and DFS:

Connectivity of graph

A graph is said to be connected if there is a path between every pair of vertex.. A graph with multiple disconnected vertices and edges is said to be disconnected. To check connectivity of a graph, we traverse all nodes using graph traversal algorithm like BFS and DFS. On completion of traversal, if there is any node, which is not visited, then the graph is disconnected.

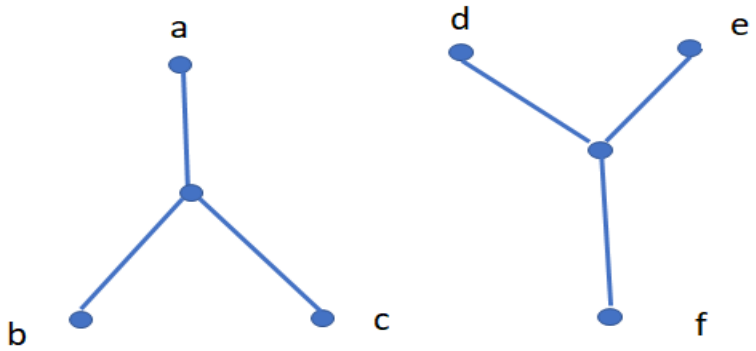
Connected graph:

Below graph is an example for connected graph since it is possible to traverse from one vertex to any other vertex. For example, one can traverse from vertex c to vertex d using the path c-d-e. Similarly we can traverse from vertex a to e using the path a-c-d-e



Disconnected

The graph shown below is an example for disconnected graph because there exist no path from b to e.



- To check whether a graph is connected or not, we traverse the graph using either bfs traversal or dfs traversal method
- After the traversal if there is at-least one node which is not marked as visited then that graph is disconnected graph

Procedure to check the whether a graph is connected or not using adjacency matrix

- Read the adjacency matrix .
- Create a visited [] array. Start DFS/BFS traversal method from any arbitrary vertex and mark the visited vertices in the visited [] array.
- Once DFS/BFS is completed check the visited [] array. If there is at-least one vertex which is marked as unvisited then the graph is disconnected otherwise it is connected.

Connectivity of undirected graph using DFS

We choose one vertex as an arbitrary node and traverse from that node.

Algorithm:

connected (G)

Input – undirected graph.

Output – Returns True if the graph is connected otherwise False.

```
Begin
  define visited array
    for all nodes u in the G, do
      mark all nodes unvisited
      traversal (u, visited)
      if unvisited , then
```

```
        return false
    done
  return true
End
```

traversal(u, visited)

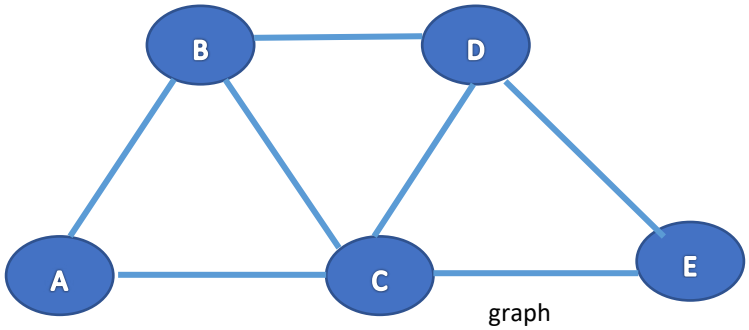
Input – Any arbitrary node u as the start node and the visited node to mark which node is visited

Output: Traverse all connected vertices.

```
Begin
  mark u as visited
  for all nodes v, if it is adjacent with u, do
    if v is not visited, then
      traversal(v, visited)
  done
End
```

Output: For the below graph is connected

To check whether the below graph is connected we give the adjacency matrix for the below graph as input.



graph

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

Connectivity of directed graph using DFS

We choose one vertex as an arbitrary node and traverse from that node.

Algorithm:

connected (G)

Input – directed graph.

Output – Returns True if the graph is connected otherwise False.

Begin

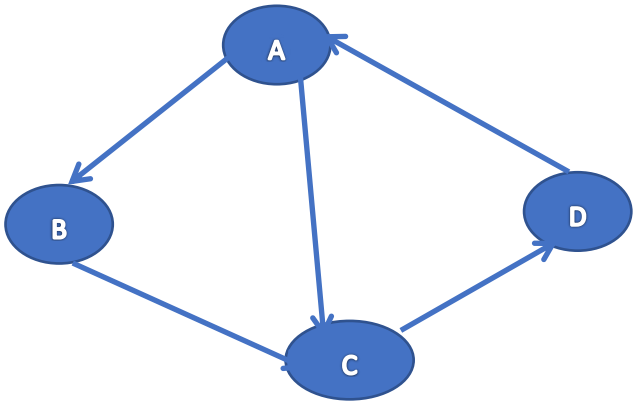
```
  define visited array
    for all nodes u in the G, do
      mark all nodes unvisited
      traversal (u, visited)
      if unvisited , then
        return false
    done
  return true
End
```

traversal(u, visited)

Input – Any arbitrary node u as the start node and the visited node to mark which node is visited

Output: Traverse all connected vertices.

```
Begin
  mark u as visited
  for all nodes v, if it is adjacent with u, do
    if v is not visited, then
      traversal(v, visited)
  done
End
```



graph

Adjacency matrix

	A	**B**	**C**	**D**
A	0	1	1	0
B	0	0	1	0
C	0	0	0	1
D	1	0	0	0

Output for the above directed graph is connected graph

Connectivity of directed graph using BFS

Algorithm:

traversal(x, visited)

Input: the arbitrary node x as start node

Output: Traverse all connected vertices.

Begin

```
  make x as visited
  insert x into a queue Que
  until the Que is not empty, do
  u = node taken out from queue
  for each vertex v of the graph G, do
    if the u and v are connected, then
      if u is not visited, then
        make u as visited
      insert u into the queue Que.
    done
  done
End
```

connected(G)

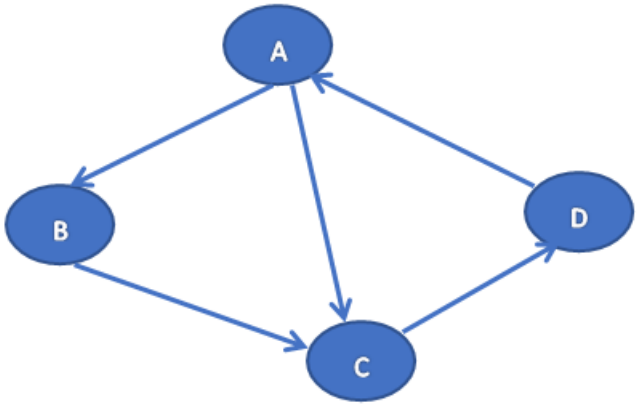
Input – The directed graph.

Output – True if the graph is connected otherwise returns False.

```
Begin
  define visited array
  for all nodes u in the G, do
    mark all nodes as unvisited
  traversal(u, visited)
  if  unvisited , then
    return false
  done
  return true
```

End

We input the adjacency matrix for the below graph as and the The output for the above directed graph is connected.



	A	B	C	D
A	0	1	1	0
B	0	0	1	0
C	0	0	0	1
D	1	0	0	0

Connectivity of undirected graph using BFS

Algorithm:

traversal(x, visited)

Input: The arbitrary node x as start node and the visited node to mark which node is visited.

Output: Traverse all connected vertices.

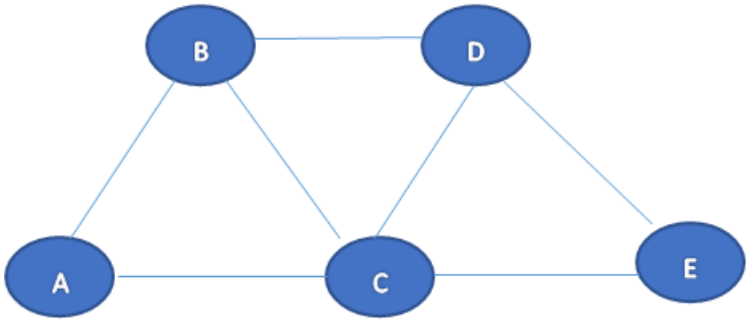
```
Begin
  make x as visited
  insert x into a queue Que
  until the Que is not empty, do
  u = node that is taken out from the queue
  for each vertex v of the graph G, do
    if the u and v are connected, then
      if u is not visited, then
        make u as visited
      insert u into the queue Que.
    done
  done
End
```

Connected(G)

Input – The directed graph.

Output – True if the graph is connected otherwise returns False.

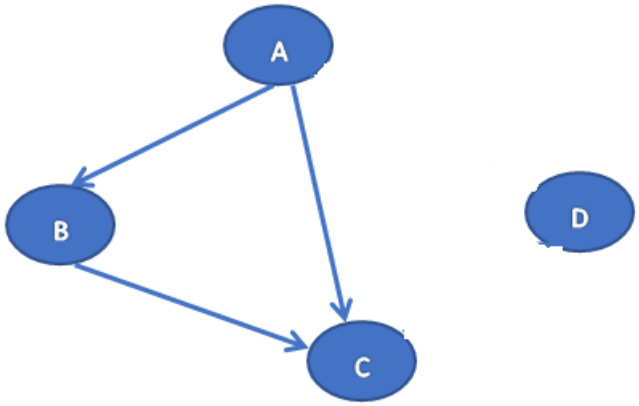
```
Begin
  define visited array
  for all nodes u in the G, do
    mark all nodes as unvisited
  traversal(u, visited)
  if  unvisited , then
    return false
  done
  return true
End
```



graph

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

The output for the below undirected graph using BFS is connected.



graph

	A	B	C	D
A	0	1	1	0
B	0	0	1	0
C	0	0	0	0
D	0	0	0	0

The output for the above graph is disconnected.

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPHS

Abstract

Case study-Indexing in databases (B-tree: K way tree)

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

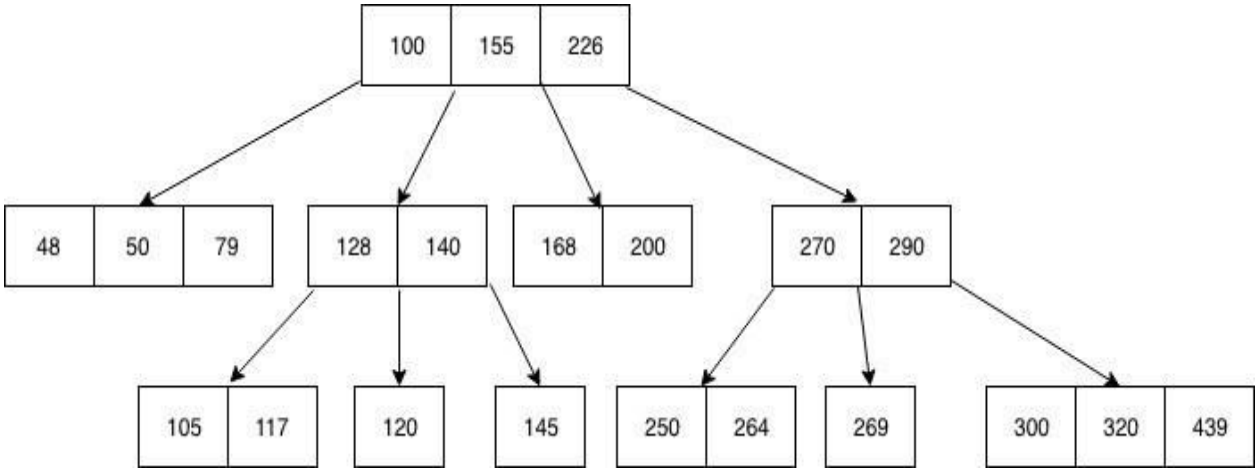
Case study-Indexing in databases (B-tree: K way tree)

Indexing:

Indexing is a technique used in data structure to access the records quickly from the database file. An Index is a small table containing two columns one for primary key and the second column contains a set of pointers for holding the address of disk block where the specific key value is stored.

What Is B-tree?

B-tree is a data structure that store data in its node in sorted order. A Simple B-tree is represented as shown below



Sample B-tree

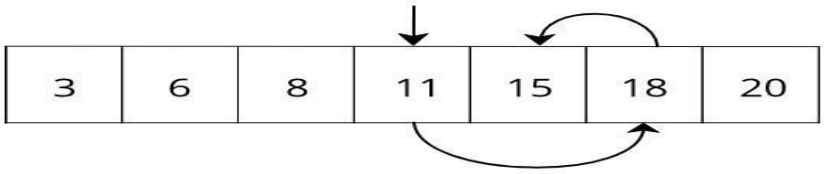
B-tree stores data such that each node contains keys in ascending order. Each of these keys has two references to another two child nodes. The left side child node keys are less than the current keys and the right side child node keys are more than the current keys. If a single node has “n” number of keys, then it can have maximum “n+1” child nodes.

Why Is Indexing Used in the Database?

Imagine a user needs to store a list of numbers in a file and search a given number on that list. The simplest solution is to store data in an array and append values when new values come. But to check if a given value exists in the array, then user need to search through the entire array elements one by one and check whether the given value exists. If the user finds the given value

in the first element then it is the best case. In the worst case, the value can be the last element in the array.

How could you improve this time? The solution is to sort the array and use binary search to find the value. Whenever you insert a value into the array, it should maintain order. Search starts by selecting a value from the middle of the array. Then compare the selected value with the search value. If the selected value is greater than search value, ignore the left side of the array and search the value on the right side and vice versa



Binary search

Here, we search key 15 from the array 3,6,8,11,15, and 18, which is already in sorted order. If you do a normal search, then it will take five units of time to search since the element is in the fifth position. But in the binary search, it will take only three searches.

Properties of B-tree:

- All the leaves created are at same level.
- B-Tree is determined by a number of degree m. The value of m depends upon the block size on the disk .
- The left subtree of the node has lesser values than the right side of the subtree.
- The maximum number of child nodes(a root node as well as its child nodes can contain) is calculated by :m – 1
- Every node except the root node must contain minimum keys of[m/2]-1
- The maximum number of child nodes a node can have is equal to its degree that is m.
- The minimum children a node can have is half of the order that is m/2

Why use B-Tree

- B-tree reduces the number of reads made on the disk
- B Trees can be easily optimized to adjust its size according to the disk size
- It is a specially designed technique for handling a bulky amount of data.
- It is a useful algorithm for databases and file systems.
- B-tree is efficient for reading and writing from large blocks of data

Insertion

- B-tree insertion takes place at leaf node.
- Locate the leaf node for the data being inserted.
- If the node is not full that is fewer than m-1 entries, the new data is simply inserted in the sequence of node.
- When the leaf node is full, we say overflow condition.
- Overflow requires that the leaf node be split into 2 nodes, each containing half of the data.
- To split the node, create a new node and copy the data from the end of the full node to the new node.
- After the data has been split, the new entry is inserted into either the original or the new node depending on its key value.
- Then the median data entry is inserted into parent node.

Insertion Algorithm

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.

2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.

3. Else, if the leaf node contains m-1 keys, then follow the following steps.

a) Insert the new element in the increasing order of elements.

b) Split the node into the two nodes at the median.

c) Push the median element upto its parent node.

d) If the parent node also contain m-1 number of keys, then split it too by following the same steps.

B-tree Deletion

- When deleting a node from B-tree , there are three considerations
- 1. data to be deleted are actually present in the tree.
- 2. if the node does not have enough entries after the deletion, then correct the structural deficiency.
- A deletion that results in a node with fewer than minimum number of entries is an underflow
- 3. Deletion should takes place only from leaf node.
- If the data to be deleted are in internal node, find a data entry to take their place.

Deletion Algorithm

1.Locate the leaf node.

2.If there are more than m/2 keys in the leaf node then delete key

3.If the leaf node doesn't contain m/2 keys then

a)If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.

b)If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

4.If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.

5.If parent is left with less than m/2 nodes then, apply the above process on the parent too.