MODULE - 4

TREES(Cont..): Binary Search trees, Selection Trees, Forests, Representation of Disjoint sets, Counting Binary Trees,
GRAPHS: The Graph Abstract Data Types, Elementary Graph Operations

Binary Search Trees

ADT Dictionary
objects: a collection of n > 0 pairs, each pair has a key and an associated item
functions: for all d ∈ Dictionary, item ∈ Item, k ∈ Key, n ∈ integer

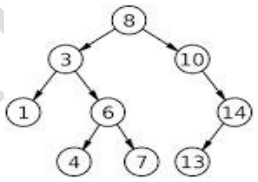
DictionaryCreate(max_size) ::=	create an empty Dictionary
Boolean IsEmpty(d, n) ::=	**if** (n > 0) **return** TRUE **else return** FALSE
Element Search(d, k) ::=	**return** item with key k, **return** NULL if no such element.
Element Delete(d, k) ::=	delete and return item (if any) with key k;
void Insert(d,item,k) ::=	insert item with key k into d.

Definition Binary search tree

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1) Each node has exactly one key and the keys in the tree are distinct.
2) The keys (if any) in the left subtree are smaller than the key in the root.
3) The keys (if any) in the right subtree are larger than the key in the root.
4) The left and right subtrees are also binary search trees.
5) The root has a key.

Example:



Binary search Tree

Searching a Binary Search Tree

To search for a node whose key is *k*. We begin at the root of the binary search tree.

- If the root is *NULL*, the search tree contains no nodes and the search is unsuccessful.
- we compare *k* with the key in root. If *k* equals the root's key, then the search terminates successfully.
- If *k* is less than root's key, then, we search the left subtree of the root.
- If *k* is larger than root's key value, we search the right subtree of the root.

Structure of the node can be defined as follows

```
struct node
{
Struct node *lchild;
struct
{
int  item;                /* Itype represents the data type of the element*/
int key;
}data;

Struct node *rchild;
};

Typedef struct node TreeNode;
```

Recursive search of a binary search tree: Return a pointer to the element whose key is k, if there is no such element, return NULL. We assume that the data field of the element is of type elemenet and it has two components key and item.

```
Treenode * search(TreeNode * tree, int k)
{
        if (tree==NULL) return NULL;
        if (k == tree→data.key)
        return (tree);
        if (k < tree→data.key)
            return search(tree→leftChild, k);
        return search(tree→rightChild, k);
}
```

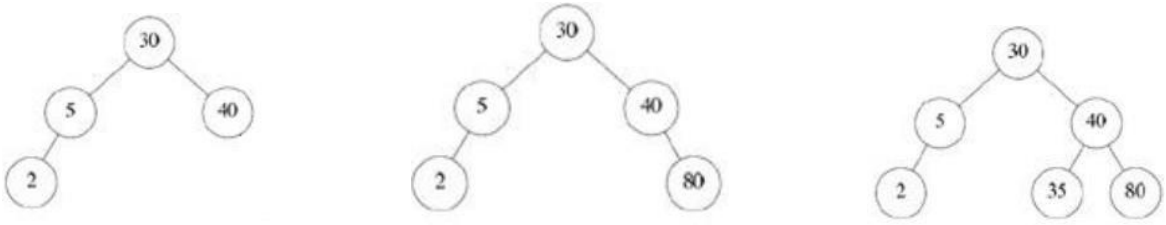
Iterative search of a Binary Search tree

```
Treenode* iterSearch(TreeNode * tree, int k)

{
while (tree!=null)
{
        if (k == tree→data.key)
         return (tree);
        if (k < tree→data.key)
        tree = tree→leftChild;
        else
        tree = tree→rightChild;
}
return NULL;
}
```

Analysis of Search(Both iterative and recursive): If h is the height of the binary search tree, then we can perform the search using either search in O(h). However, recursive search has an additional stack space requirement which is O(h).

Inserting in to a Binary Search Tree: Binary search tree has distinct values, first we searchthe tree for the key and if the search is unsuccessful the key is inserted at the point the search terminated

Example : consider the tree given below



BST Tree **Insert 80** **Insert 35**

Inserting a dictionary pair into a binary search tree

If k is in the tree pointed at by node do nothing. Otherwise add a new node with data = (k, item)

```
TreeNode*  insert(TreeNode *root, int k, int Item)
{
TreeNode * ptr, *lastnode;

lastnode=Modifiedsearch(root,k);
Ptr=(TreeNode*)malloc(sizeof(TreeNode));
ptr→data.key = k;
ptr→data.item = Item;
ptr→leftChild = ptr→rightChild = NULL;


if (root==NULL)
{
 root=ptr;
return(root);
}

if(lastnode!=NULL)
{
        if (k < lastnode→data.key)
        lastnode→leftChild = ptr;
        else
        lastnode→rightChild = ptr;
        return (root);
}

}
```

If the element is present or if the tree is empty the function Modifiedsearch returns NULL. If the element is not present it retrun a pointer to the last node searched.

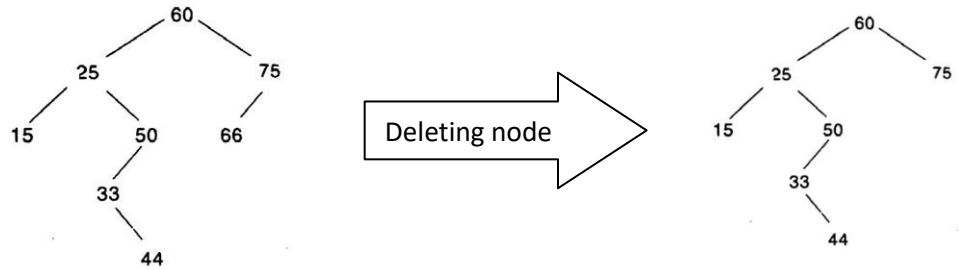
```
Modifiedsearch(Treenode *root,int k)
```

```
{

TreeNode *temp,*prev;
temp==node;
prev=NULL;
if(temp==NULL)
return(NULL);
while(temp!=NULL)
{
        if(temp->data.key==k)
        {
                printf("element already found");
                return(NULL);
        }
        if(key<temp->data.key)
        {
        Prev=temp;
        temp=temp->rcchild;
        }
        else
        {
        Prev=temp;
        Temp=temp->rchild;
        }
}
retrun(prev);

}
```

Deletion from a binary search tree: Suppose T is a a binary search tree. The function to delete an item from tree T first search the tree to find the location of the node with the item and the location of the parent of N and the deletion of the node N depends on three cases:

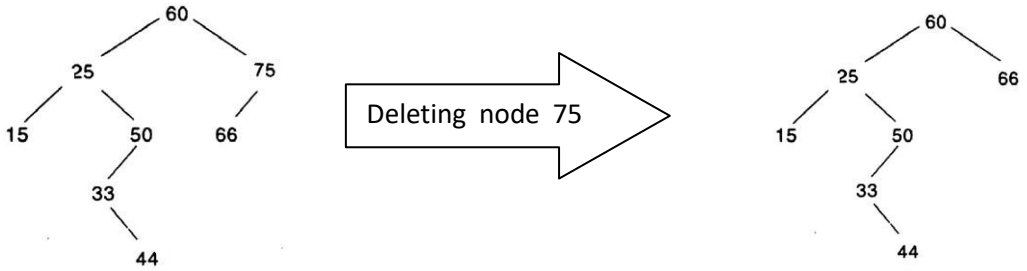
Case 1: N has no children. Then N is deleted from T by replacing the location of the node N in the parent(N) by the NULL pointer

Example: Deleting Node 66 with NO children



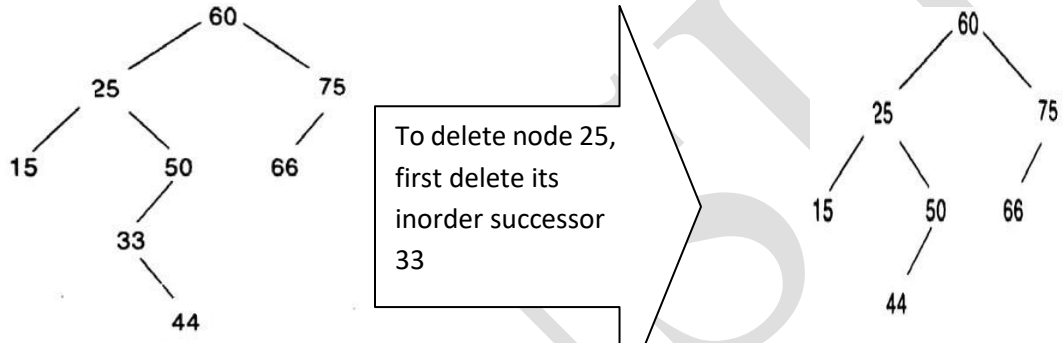
Case 2: If N has exactly one child. Then N is deleted from T by replacing the location of N in Parent (N) by the location of the only child of N.

Example: Deleting Node 75 with exactly one children

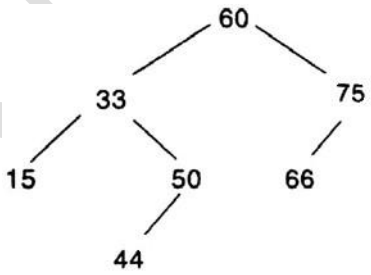


Case 3: N has Two children. Let S(N) denote the inorder successor of N(S(N) does not have a left child).Then N is deleted from T by first deleting S(N) from T (by using case1 or cae 2) and then replacing node N in T by the node S(N).

Example: Deleting Node 25 with two children



Now replace the node 25 with its inorder successor 33



Recursive function to delete a node in a BST

```
TreeNode *delete_element(TreeNode *node, int key)
{
TreeNode * temp;

   if (node == NULL)
      return node;


   if (key < node->data.key)
      node->lchild = delete_element(node->lchild, key);
   else if (key > node->data.key)
     node->rchild = delete_element(node->rchild, key);
```

```
else
{
   // node with only one child
   if (node->lchild == NULL)
   {
      temp = node->rchild;
      free(node);
      return temp;
   }
   else if (node->rchild == NULL)
   {
      temp = node->lchild;
      free(node);
      return temp;
   }
   // node with two children

   else
   {
      temp = node->rchild;
      while(temp->lchild!=NULL)      //Get the inorder successor
               temp=temp->lchild;
       node->data.item = temp->data.item;
      node->data.key=temp->data.key;
       node->rlink = delete_element(node->rchild, temp->data.key);
       return node;

   }
}
}
```

GRAPHS

Introduction

The first recorded evidence of the use of graph dates back to 1736. When Leonhard Euler used them to solve the classical Konigsberg bridge problem.

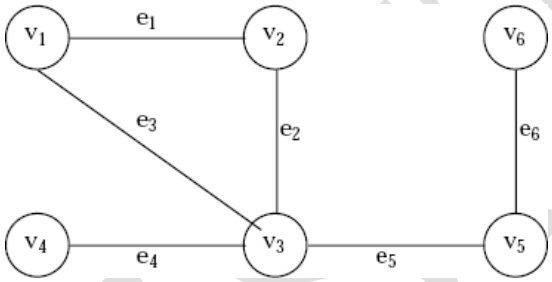
Definitions

Graph: A graph G consist of two sets V and E

1. V is a finite nonempty set of vetices and
2. E is a set of pairs of vertices these pairs are called edges

A graph can be represents as G = (V, E). V(G) will represent the set of vertices and E(G) will represent the set of edges of the graph G

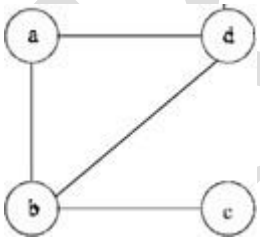
Example:



V(G)= {v1, v2, v3, v4, v5, v6}
E(G) = {e1, e2, e3, e4, e5, e6} E(G) = {(v1, v2) (v2, v3) (v1, v3) (v3, v4), (v3, v5) (v5, v6)}.
There are six edges and six vertex in the graph

Undirected Graph: In a undirected graph the pair of vertices representing an edge is unordered. thus the pairs (u,v) and (v,u) represent the same edge.

Example:

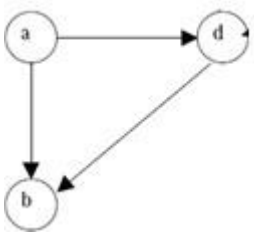


V(G)={a,b,c,d}

E(G)={(a,b),(a,d),(b,d),(b,c)

Directed Graph (digraph): In a directed graph each edge is represented by a directed pair (u,v), v is the head and u is the tail of the edge. Therefore (v,u) and(<u,v) represent two different edges

Example:

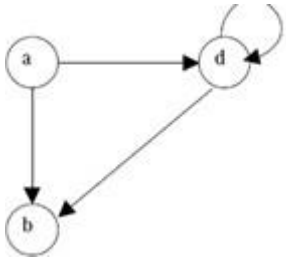


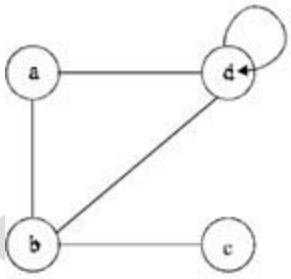
V(G)={a,b,d}

E(G)={(a,d), (a,b), (d,b)}

Self Edges/Self Loops: Edges of the form(v,v) are called self edges or self loops . It is an edge which starts and ends at the same vertex.

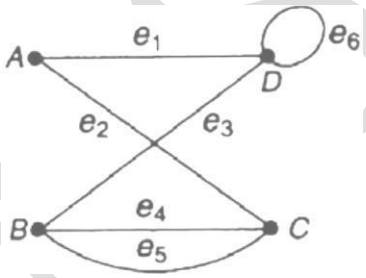
Example:





Mutigraph: A graph with multiple occurrences of the same edge is called a multigraph

Example:



Complete Graph: An undirected graph with n vertices and exactly n(n-1)/2 edges is said to be a complete graph. In a graph all pairs of vertices are connected by an edge.

Example : A complete graph with n=3 vertices

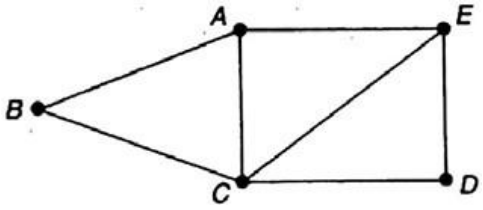
Adjacent Vertex

If (u,v) is an edge in E(G), then we say that the vertices u and v are adjacent and the edge(u,v) is incident on vertices u and v.

Path: A path from vertex u to v in graph g is a sequence of vertices u,i1,i2,…….ik,v such that (u,i1),(i1,i2)……(ik,v) are edges in E(G). if G' is directed then the path consists of <u,i1>,<i1,i2>……<ik,v> edges in E(G').

The length of the path is the number of edges in it.

Example:



(B,C),(C,D) is a path from B to D the length of the path is 2

A simple path is a path in which all the vertices are distinct.

Cycle: A cycle is a simple path in which all the vertices except the first and last vertices are distinct. The first and the last vertices are same.

Example :

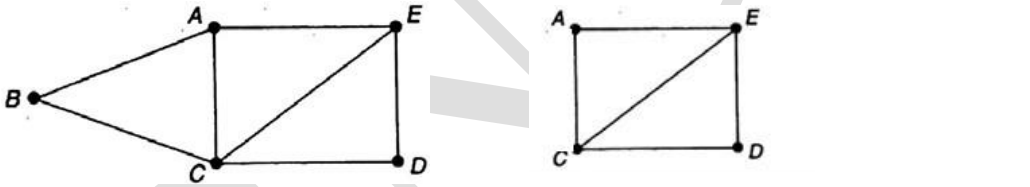
(B,C),(C,D)(D,E)(E,A)(A,B) is a cycle

Degree of a vertex : In **a undirected graph** degree of a vertex is the number of edges incident on a vertex.

In a **directed graph** the **in-degree** if a vertex v is the number of edges for which v is the head i.e. the number of edges that are coming into a vertex. The **out degree** is defined as the number of edges for which v is the tail i.e. the number of edges that are going out of a vertex

Subgraph: A subgraph of G is a graph G' such that V(G') $\subseteq$ V(G) and E(G') $\subseteq$ E(G)

Example :



Graph(G) Subgraph(G')

Connected Graph: An undirected graph G is said to be connected if for every pair of distinct vertices u and v in V(G) there is a path from u to v in G.

Connected Component is a maximal connected subgraph

Strongly connected graph : A directed graph G is said to be strongly connected if for every pair of distinct vertices u an v in V(G), there is a directed path from u to v and from v to u.

Tree: A tree is a connected acyclic connected graph.

ADT Graph

Objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.

Functions: for all graph$\in$ Graph, v,v1,v2 $\in$ vertices

Example: Create():=	**return** an empty graph
Graph InsertVertex(graph,v):=	**return** a graph with v inserted. v has no incident edges
Graph InsertEdge(graph,v1,v2) :=	**retrun** a graph with a new edge between v1 and v2
Graph DeleteVertex(graph,v) :=	**return** a graph in which v and all edges incident to it is removed
Graph DeleteEdge(graph,v1,v2):=	**retrun** a graph in which the edge (v1,v2) is removed, leave the incident nodes in the graph
Boolean IsEmpty:=	If (graph == empty graph) retrun TRUE else Retrun FALSE
List Adjacent(graph,v) :=	**retrun** a list of all vertices that are adjacent to v

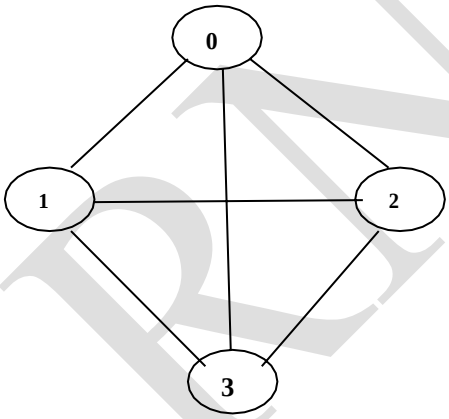
Graph Representation

The three most commonly used representations are

- Adjacency Matrix
- Adjacency List
- Adjacency Multilist

Adjacency Matrix: Let G=(V,E) be a graph with n vertices, n>=1. The adjacency matrix of G is a two dimensional n*n array for example a, with the property that a[i][j]=1 if there exist ane edge (i,j) (for a directed graph edge <i,j> is in E(G).a[i][j]=0 if no such edge in G.

Example:



Adjacency Matrix

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

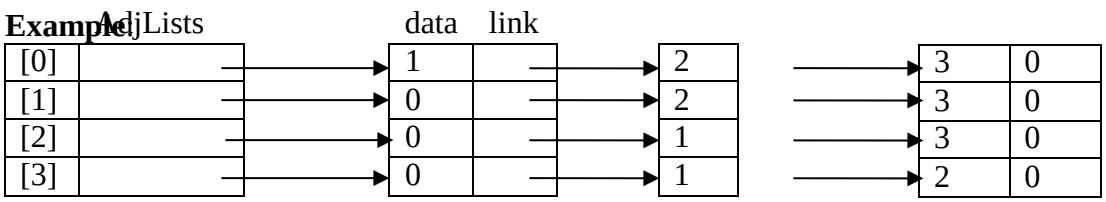
Figure 5.1 Graph G1

- The space requirement to store an adjacency matrix is n^2 bits.
- The adjacency matrix for a undirected graph is symmetric .About half the space can be saved in an undirected graph by storing only the upper or lower triangle of the matrix.
- For an undirected graph the degree of any vertex i is its row sum. For a directed graph the row sum is the out-degree and the column sum is the in-degree.

Adjacency list: In adjacency matrix the n rows of the adjacency matrix are represented as n chains. There is one chain for each vertex in G. The nodes in chain i represent the vertices that are adjacent from vertex i. The data field of a chain node stores the index of an adjacent vertex.

Example: the adjacency list of graph G1 in figure 5.1 is shown below

Example:



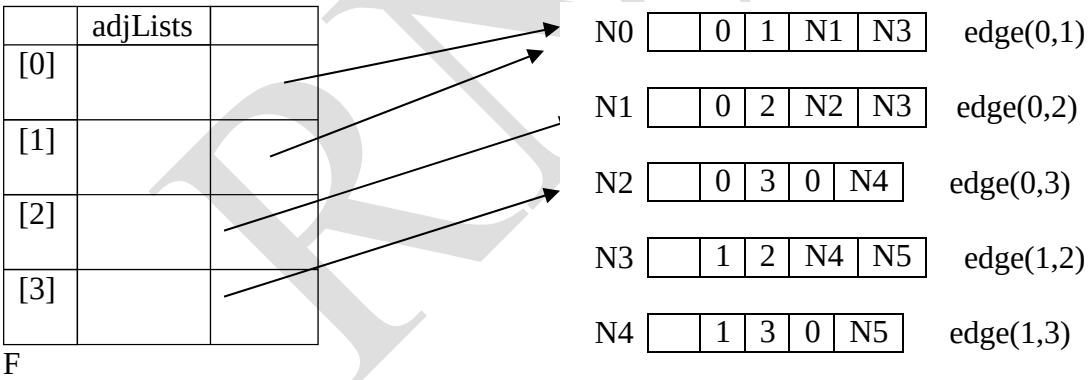
- For an undirected graph with n vertices and e edges. The linked adjacency lists representation requires an array of size n and 2e chain nodes.
- The degree of any vertex in an undirected graph may be determined by counting the number of nodes in the adjacency list.
- For a digraph the number of list nodes is only e.

Adjacency Multi lists: For each edge there will be exactly one node, but this node will be in two list(i.e., the adjacency list for each of the two nodes to which it is incident). A new field is necessary to determine if the edge is determined and mark it as examined.

The new node structure is

m	Vertex1	Vertex2	Link1	Link2

Example: The adjacency multilist for graph G1 is shown below

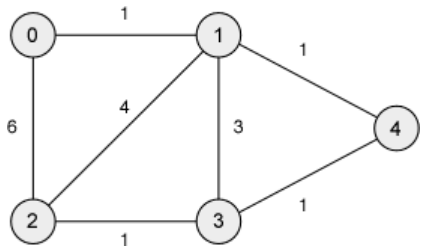


F

The Lists are Vertex 0: N0->N1->N2
Vertex 1: N0->N3->N4
Vertex 2: N1->N3->N5
Vertex 3: N2->N4->N5

Weighted Edges: In many applications the edges of a graph have weight assigned to them. These weights may represent the distance from one vertex t o another or the cost for going from one vertex to an adjacent vertex. The adjacency matrix and list maintains the weight information also. A graph with weighted edges are also called network.

Example:



Elementary Graph Operations

Given an undirected graph G=(V,E) and a vertex v in V(G) ,there are two ways to find all the vertices that are reachable from v or are connected to v .

- Depth First Search and
- Breadth First Search

Depth First Search

1. Visit the starting vertex v. (visiting consist of printing node's vertex)
2. Select an unvisited vertex w from v's adjacency and carry a depth first search on w.
3. A stack is maintained to preserve the current position in v's adjacency list.
4. When we reach a vertex u that has no unvisited vertices on adjacency list, remove a vertex from the stack and continue processing its adjacency list. Previously visited vertices are discarded and unvisited vertices are placed on stack
5. The search terminates when the stack is empty.

A recursive implementation of depth first search is shown below.

A global array visited is maintained , it is initialized to false, when we visit a vertex i we change the visited[i] to true.

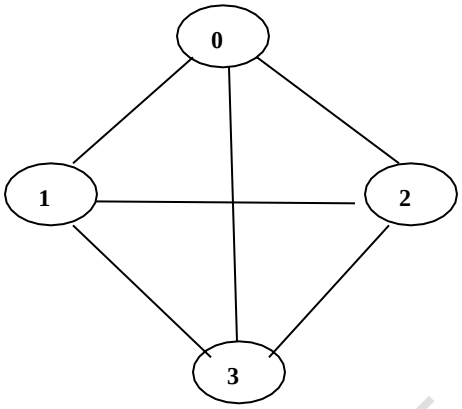
Global Declaraions

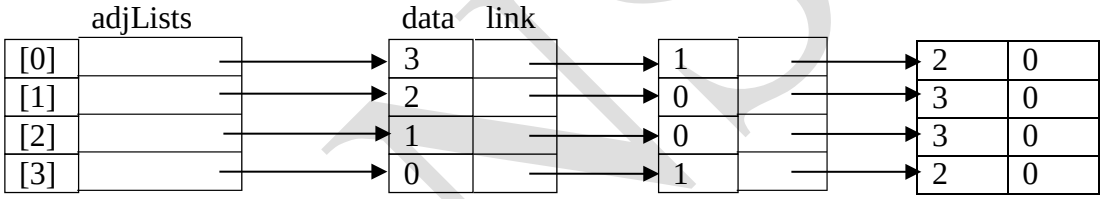
```
# define FALSE 0
# define true 1
Short int visited[max_vertices];

void dfs(int v)
{
        visited[v]=TRUE;
        printf("%d",v);
        w=graph[v]
        while(w!=NULL)
        {
                if(visited[w->vertex]==FALSE)
                        dfs(w->vertex);
                w=w->link;

        }}
```

Analysis:

- If we represent G by its adjacency list then we can determine the vertices adjacent to v by following a chain of links. Since dfs examines each node in the adjacency list at most once then the time to complete the search is O(e).
- If we represent G by its adjacency matrix then determining all vertices adjacent o v requires O(n) time. Since we visit at most n vertices the total time is $O(n^2)$.

Example: For the graph given below if the search is initiated from vertex 0 then the vertices are visited in the order vertex 3, 1, 2





Breadth first Search

1. Search starts at vertex v marks it as visited.
2. It then visits each of the vertices on v's adjacency list.
3. As we visit each vertex it is placed on a queue.
4. When all the vertices in the adjacency list is visited we remove a vertex from the queue and proceed by examining each of the vertices in its adjacency list.
5. Visited vertices are ignored and unvisited vertices are placed on the queue
6. The search terminates when the queue is empty.

The queue definition and the function prototypes

```
 struct node
{
        int vertex;
        struct node * link;
};
typedef struct node queue;
```

```
queue * front,*rear;
int visied[max_vertics];

void addq(int);
int delete();

void bfs(int v)
{
front=rear=NULL;
printf("%d",v);
visisted[v]= TRUE;
addq(v);
while(front)
{
v=deleteq();
while(w!=NULL)
        {
                if(visited[w->vertex]==FALSE)
                {
                        printf("%d",w->vertex);
                        addq(w->vertex);
                        visited[w->vertex]=TRUE;
                }
                        w=w->link;
        }
}
}
```

Analysis of BFS:

- For each vertex is placed on the queue exactly once, the while loop is iterated at most n times.
- For the adjacency list representation the loop has a total cost of O(e). For the adjacency matrix representation the loop takes O(n) times
- Therefore the total time is O (n^2).