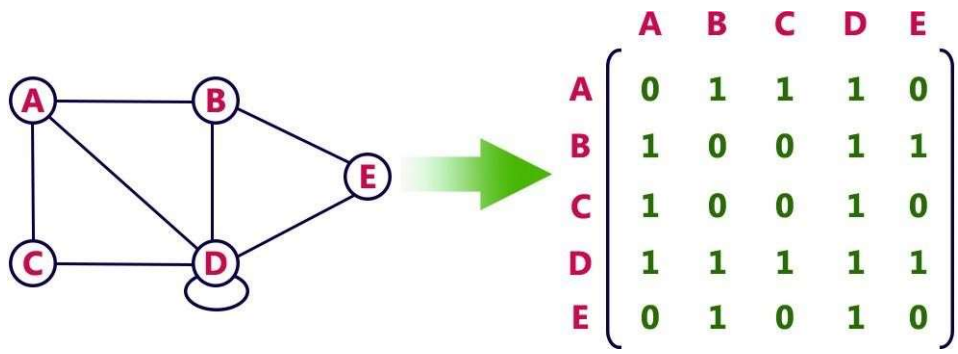
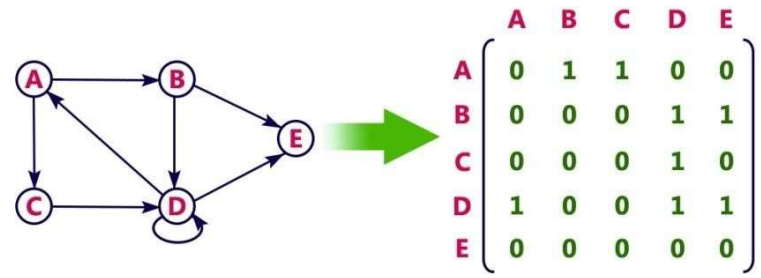
vertex to column vertex.

For example, consider the following undirected graph representation...



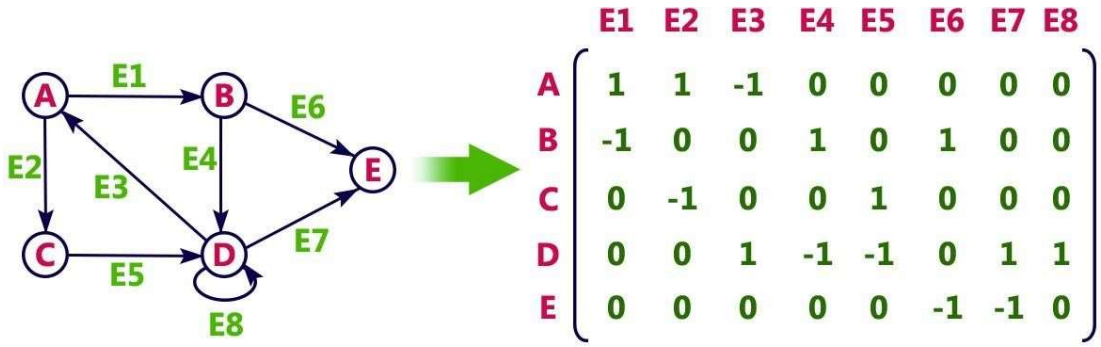
Directed graph representation...



Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

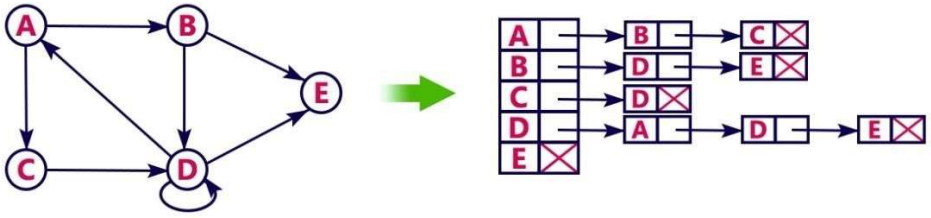
For example, consider the following directed graph representation...



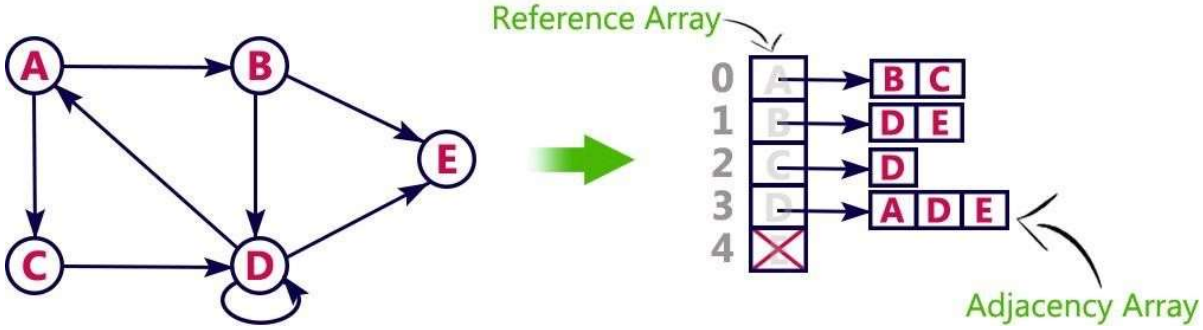
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Depth First Search (DFS)

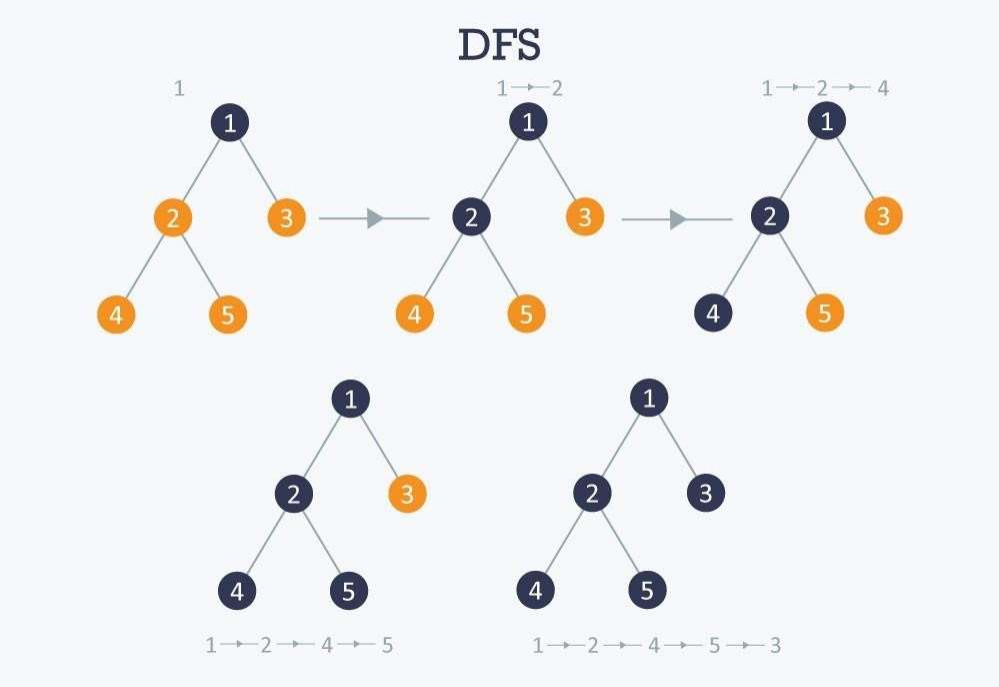
The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will beselected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:
Pick a starting node and push all its adjacent nodes into a stack.
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

```
DFS-iterative (G, s):                          //Where G is graph and s is source vertex
    let S be stack
    S.push( s )          //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
        //Pop a vertex from stack to visit next
        v = S.top( )
        S.pop( )
        //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                    S.push( w )
                    mark w as visited
```

```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

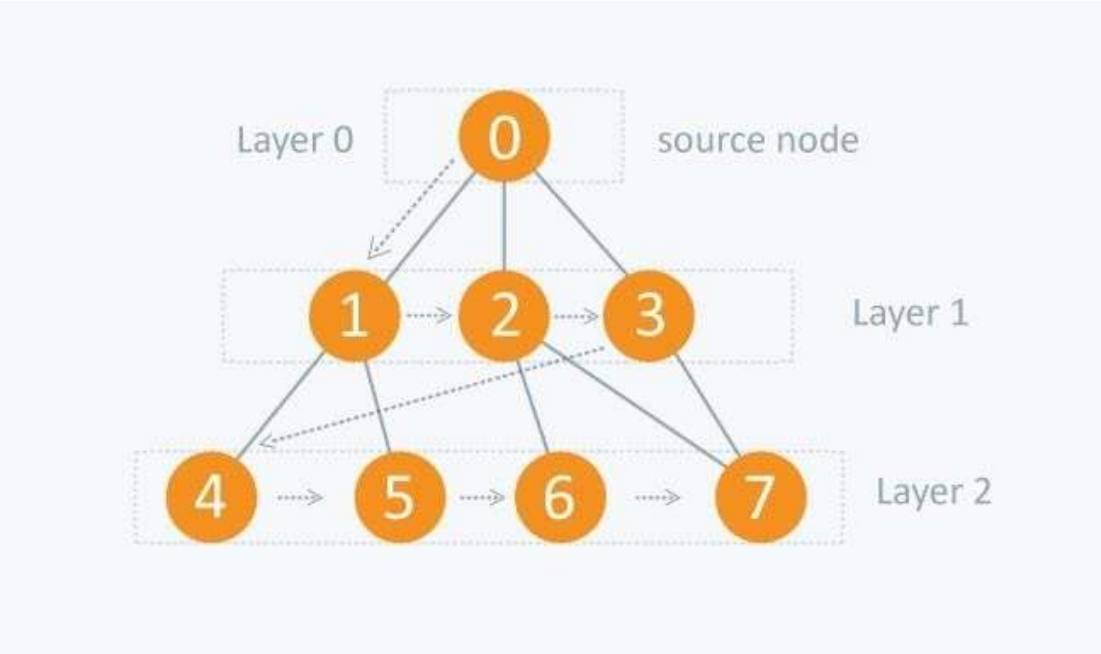


Breadth First Search (BFS);

There are many ways to traverse graphs. BFS is the most commonly used approach.BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1.First move horizontally and visit all the nodes of the current layer

2.Move to the next layer



Dictionaries:- linear list representation, skip list representation, operations insertion, deletion and searching, hash table representation, hash functions, collision resolution-separate chaining, open addressing- linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

DICTIONARIES:

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key.
Basic operations that can be performed on dictionary are:
1. Insertion of value in the dictionary
2. Deletion of particular value from dictionary
3. Searching of a specific value with the help of key

Linear List Representation

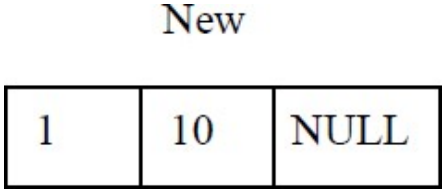
The dictionary can be represented as a linear list. The linear list is a collection of pair and value.
There are two method of representing linear list.
1. Sorted Array- An array data structure is used to implement the dictionary.
2. Sorted Chain- A linked list data structure is used to implement the dictionary

Structure of linear list for dictionary:

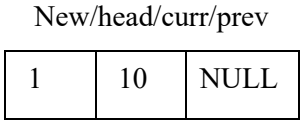
```
class dictionary
{

private:
        int k,data;
        struct node
        {
        public: int key;
        int value;
        struct node *next;
        } *head;

public:
        dictionary();
        void insert_d( );
        void delete_d( );
        void display_d( );
        void length();
};
```

Insertion of new node in the dictionary:

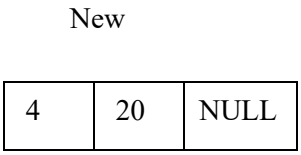
Consider that initially dictionary is empty then
head = NULL
We will create a new node with some key and value contained in it.



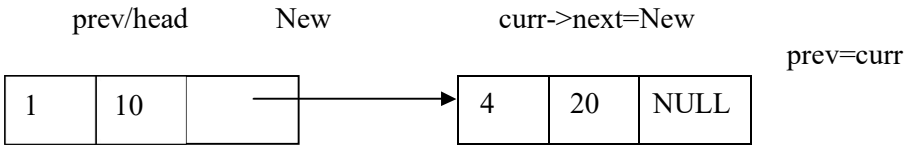
Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be 'curr' and 'prev' as well. The 'cuur' node will always point to current visiting node and 'prev' will always point to the node previous to 'curr' node. As now there is only one node in the list mark as 'curr' node as 'prev' node.



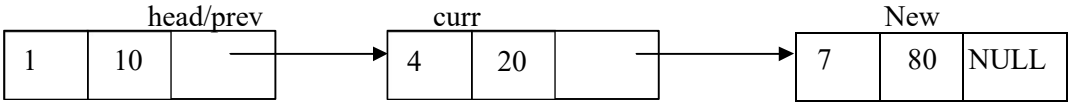
Insert a record, key=4 and value=20,



Compare the key value of 'curr' and 'New' node. If New->key > Curr->key then attach New node to 'curr' node.

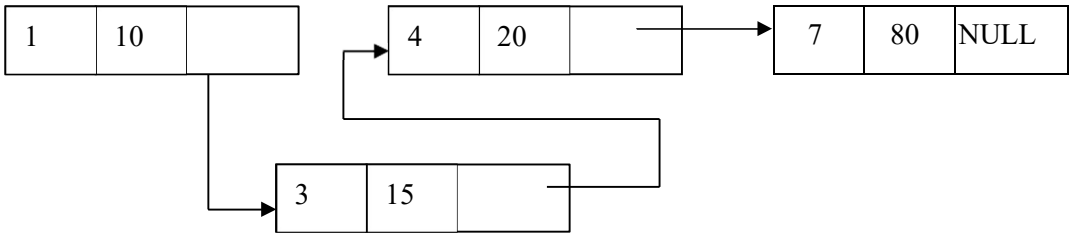


Add a new node <7,80> then



If we insert <3,15> then we have to search for it proper position by comparing key value.

(curr->key < New->key) is false. Hence else part will get executed.




```
void dictionary::insert_d( )
{
node *p,*curr,*prev;
            cout<<"Enter an key and value to be inserted:";
            cin>>k;
           cin>>data;
```

```
            p=new node;
            p->key=k;
           p->value=data;
           p->next=NULL;
            if(head==NULL)
                    head=p;
           else
           {
                  curr=head;
                   while((curr->key<p->key)&&(curr->next!=NULL))
                   {
                           prev=curr;
                           curr=curr->next;
                     }
                   if(curr->next==NULL)
                   {
                           if(curr->key<p->key)
                             {
                                   curr->next=p;
                                    prev=curr;
                           }
                           else
                           {
                                  p->next=prev->next;
                                    prev->next=p;
                           }
                    }
                    else
                   {

                           p->next=prev->next;
                           prev->next=p;
                   }
                   cout<<"\nInserted  into dictionary Sucesfully .... \n";
           }
}
```

The delete operation:

Case 1: Initially assign 'head' node as 'curr' node.Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node's key value. We will get node which is to be deleted in variable 'curr'. The node given by variable 'prev' keeps track of previous node of 'cuu' node. For eg, delete node with key value 4 then