13 Trees

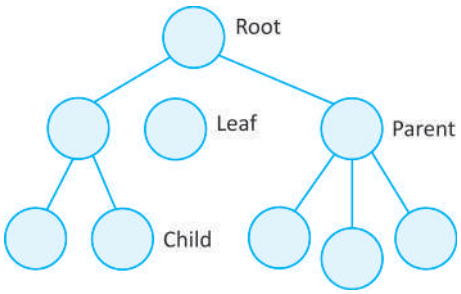
It is a non-linear data structure in which each node may be associated with more than one node.Each node in the tree points to its children. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

- The root of a tree is the node with no parent. There can be only one root node.
- A node with no children is known as a leaf.
- A node p is ancestor of node q if the node p lies on the path from q to the root of the tree. [Two type of ancestor drawing]
- If each node of a tree has only one child then it is called skew.

The most basic examples of a Tree are :

- file system inside the computer,
- HTML DOM, where there is a hierarchy of HTML tags,
- organisational structure of employees inside an organization.

The Tree data structure looks reverse of the actual physical tree, and has root on top.



Basic Terminology in Trees

- **Node :** It is the structure that contains the data about each element in the tree. Each element in a tree constitutes a node.
- **Root :** The top node in a tree.
- **Children :** A node directly connected to another node when moving away from the Root.
- **Parent :** The immediate node to which a node is connected on the path to root.
- **Siblings :** A group of nodes with same parent.
- **Ancestor :** A node reachable by repeated proceeding from child to parent.
- **Descendant :** A node reachable by repeated proceeding from parent to child.

- **Leaves :** The nodes that don't have any children.
- **Path :** The sequence of nodes along the edges of the tree.
- **Height of Tree :** The number of edges on the longest downward path from root to a leaf.
- **Depth :** The number of edges from the node to the root node.
- **Degree :** The number of edges that are being connected to the node, can be further subdivided into in-degree and out-degree.

Binary Tree

A tree is called Binary tree if it has each node has two or fewer children.

In C++ each node of a binary tree can be represented as a structure or class.

This structure contains left and right pointers, pointing to the children of the node.

Eg.

```
struct node {
      int data; //Data of the Node
      node* left;   //Pointing to the left child
      node* right; //Pointing to the right child
}
```

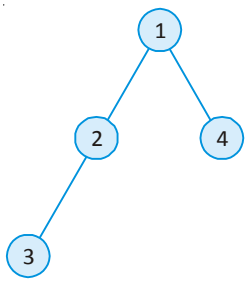
Since input of a tree is not known, we will consider -1 as the *NULL* node for the tree.

So for input,

1 2 3 -1 -1 -1 4 -1 -1

1→*left*(2)→*right*(3)→*left*(-1)→*backtrack*→*right*(-1)→*backtrackright* of 2(-1)→*backtrack*→*right of* 1(4) →*left*(-1)→*backtrack*→*right*(-1)→*backtrack*

The tree would look like,



Code for the above tree construction can be written like :

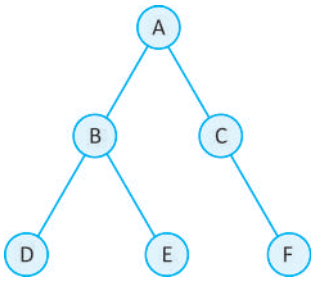
```
class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int d) {
```

```
        data  = d;
        next = NULL;
        left = NULL;
        right = NULL;
    }
};

TreeNode*  createTree()  {
    int x;
    cin >> x;
    if (x == -1) {
        return NULL;
    }

    TreeNode* root = new TreeNode(x);
    // cout << "Enter left child of " << x << " ";
    root->left = createTree();
    // cout << "Enter right child of " << x << " ";
    root->right = createTree();
    return root;
}
```

Applications

- File hierarchy structure in the computer is tree based. Assuming folder as a node can have subdirectories and files as a leaf node
- For Android Developers out there, findViewById searches the view using search in tree type layout structure
- For Web Developers out there, Document Object Model is tree type of structure and all traversal in it are tree search.

Traversals in a Tree



- **Pre-order Traversal :** Each node is processed before(pre) any nodes in its subtrees.

  root left right

  A (B D E) (C F)

```
void preOrderPrint(TreeNode* root) {
  if (root == NULL) {
      return;
  }

  cout << root->data << " ";
   preOrderPrint(root->left);
   preOrderPrint(root->right);
}
```

- **Post-order Traversal** : Each node is processed after(post) all nodes in its subtrees.

  Left right root

  (D E B) (F C) A

```
void postOrderPrint(TreeNode* root) {
    if (root == NULL) {
        return;
    }

     preOrderPrint(root->left);
     preOrderPrint(root->right);
    cout << root->data << " ";
}
```

- **Inorder Traversal :** Each node is processed after all nodes *in between* its left subtree, and its right subtree.

  Left root right

  (D B E) A (C F)

```
void postOrderPrint(TreeNode* root) {
    if (root == NULL) {
        return;
    }
     preOrderPrint(root->left);
    cout << root->data << " ";
     preOrderPrint(root->right);
}
```

- **Level-order Traversal**: Each node is processed level by level from left to right before any nodes in their subtrees.

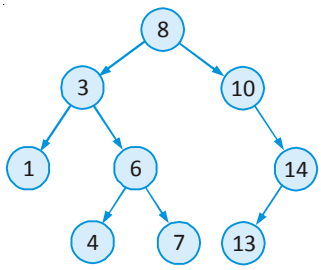
A (B C) (D E F)

```
void printLevelOrder2(TreeNode* root) {
    queue<TreeNode*> q;
    q.push(root);

    while (q.empty() == false) {
        TreeNode* cur = q.front();
        q.pop();
        cout << cur->data << " ";
       if (cur->left) {
            q.push(cur->left);
        }

        if (cur->right) {
            q.push(cur->right);
        }
    }
}
```

14 Binary Search Trees

It's a binary tree with special property that is satisfied is by each node, which is:

- *Each node in the left subtree is less than the value of the root node,*
- *Each node in the right subtree is greater than the root node's value.*



Inorder traversal (left root right) of above tree : 1 , 3 , 4 , 6 , 7 , 8 , 10 , 13 , 14

The inorder traversal of any Binary Search Tree will give elements in sorted order.

Creating a BST

```
class Node {
public:
    int data;
    Node * left;
    Node * right;
    Node(int d) {
        data = d;
        left = NULL;
        right = NULL;
    }
};
Node * buildBST() {
    int x;
    Node * root = NULL;
    Node * insertBST(Node *, int);
    while (true) {
        cin >> x;
        if (x == -1) break;
        root = insertBST(root, x);
```

```
    }
    return root;
}

Node * insertBST(Node * root, int d) {
    if (root == NULL) {
        Node * newNode = new Node(d);
        return newNode;
    }

    if (d < root->data) {
        root->left = insertBST(root->left, d);
    }
    else {
        root->right = insertBST(root->right, d);
    }
    return root;
}
```

Searching in a Binary Search Tree :

The main application of a Binary search tree is efficient searching.

Due to the Binary Search Tree property, one Subtree of the BST can be discarded, as the value to be searched can be either greater or smaller or equal to the root of the tree, but can't be multiple of these.

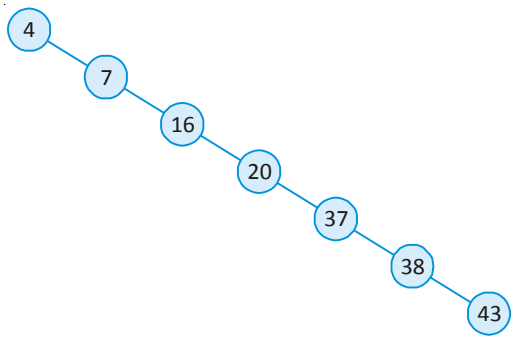
```
Node* search(Node* root,int val){
     if(root is empty)
          // Not found
     else{
          if(current node's data is same as val)
               // return this node
          else if(current node's data is > val)
               // return answer found in left subtree
          else if(current node's data is < val)
               // return answer found in right subtree
     }
}
```

This type of searching algorithm is similar to that of the binary search, where we reduce the search space and thereby optimise the search process.

Time complexity for searching in a Binary Search Tree

Though the search algorithm is similar to that of a binary search, the time complexity is not Log(n) every time. Instead it depends on the height of the tree.

Consider this binary search tree:



This is an example of Skew tree.

In this case the Time Complexity to search an element would be O(n).

The Time Complexity for searching an element in a Binary Search Tree is O(Height of the BST).

The Height of the tree is Log(N) only in the case of complete/balanced binary tree, only for these cases the time complexity would be O(log(n)).

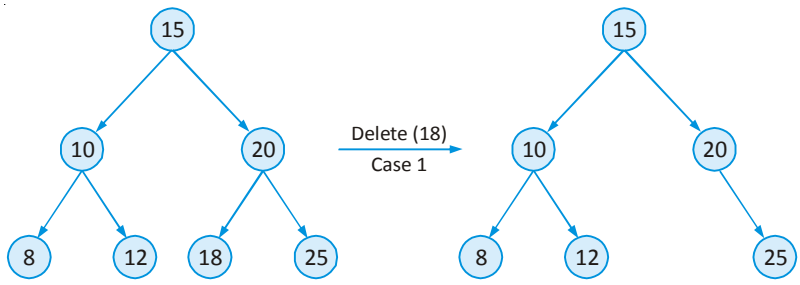
Balanced BST

A height balanced BST is a BST where the absolute difference between heights of left and right subtrees is 1 for all nodes in the tree.

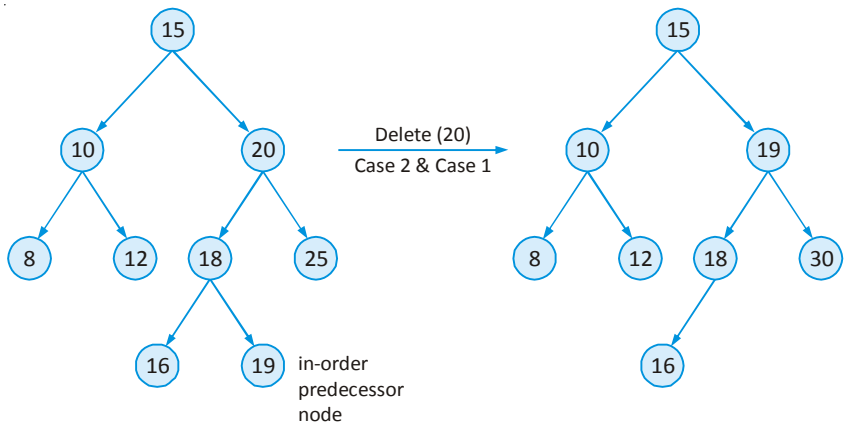
Deletion in a BST

Given a BST, write an efficient function to delete a given key in it.
To delete a node from BST, there are three possible cases to consider:

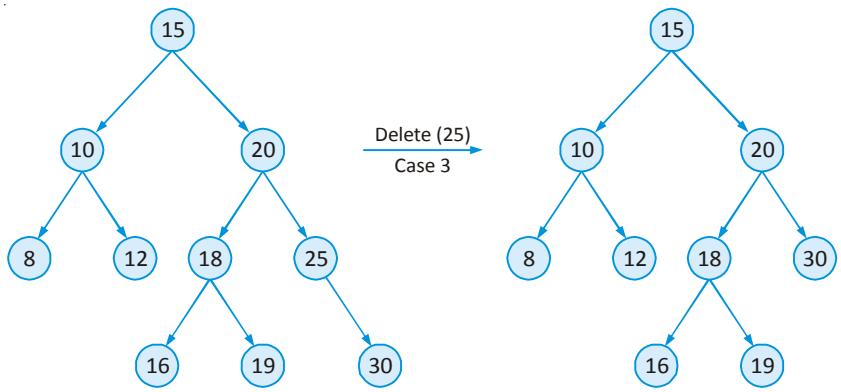
Case 1: Deleting a node with no children: simply remove the node from the tree.



Case 2: Deleting a node with two children: call the node to be deleted N . Do not delete N . Instead, choose either its in-order successor node or its in-order predecessor node R . Swap R with N, then recursively call delete on until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NULL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of the first 2 cases.



Case 3: Deleting a node with one child: remove the node and replace it with its child.




```
TreeNode* successor(TreeNode* root) {
// detaches the inorder successor it is exists
    if (root == NULL) return NULL;
    TreeNode* parent = root;
    TreeNode* cur = root->right;
    while (cur && cur->left) {
        parent = cur;
        cur = cur->left;
    }
    if (cur) parent->left = cur->right;
    return cur;
}
TreeNode* deleteFromBST(TreeNode* root, int x) {
```

```
    if (root == NULL) {
        return NULL;
    }
    if (root->data == x) {
        // if root is a leaf
        if (!root->left && !root->right) {
            delete root;
            return NULL;
        }
        // if root has one child
        if (!root->left) {
            TreeNode* tmp = root->right;
            delete root;
            return tmp;
        }
        if (!root->right) {
            TreeNode* tmp = root->left;
            delete root;
            return tmp;
        }
        // if root has 2 children
        TreeNode* inOrderSuccessor = successor(root);
        inOrderSuccessor->left = root->left;
        inOrderSuccessor->right = root->right;
        delete root;
        return inOrderSuccessor;
    }
    if (root->data > x) {
        root->left = deleteFromBST(root->left, x);
        return root;
    } else {
        root->right = deleteFromBST(root->right, x);
        return root;
    }
}
```

15 Heaps

Consider an array in which we repeatedly have to find maximum or minimum element.

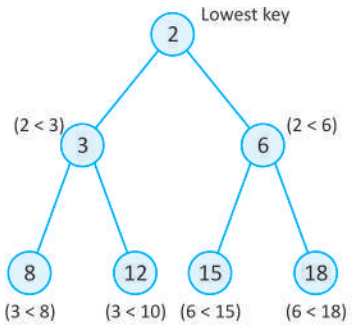
Naive method would be searching for maximum/minimum in each query, that would take O(qn) time if q queries were there. Another way could to be to keep a sorted array and give min/max element in constant time but in that case, insertion and deletion would cost us linear time.

To solve this problem we needed a data structure called heap. In heap, time complexity of operations are:

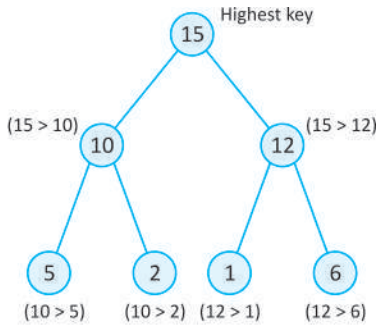
(a) Insertion : log(n) **(b)** Deletion : log(n) **(c)** Minimum Element : O(1)

Binary Heap is a tree based data structure, each node having at most 2 children. The constraint on a binary heap are:

- the value of the node is greater than both children in case max heap and smaller than both children in case of min heap.
- it is a complete binary tree that is new level will begin only when all the previous levels are completely full.



Min Heap
(Parent key is less than or equal to (≤) the child key)

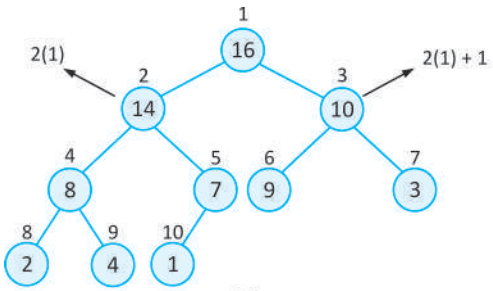


Max Heap
(Parent key is less than or equal to (≥) the child key)

Priority queues are implemented using heaps, as the top element of heap is prioritized as per the value.

What is the use of heap being complete Binary tree?

Being a complete binary tree, heap can be stored in form of an array, making access to each value easier. Also as we can see in the image below, children of each node are available at location 2i or 2i + 1, so complete can be stored in random access linear DS in effective manner.



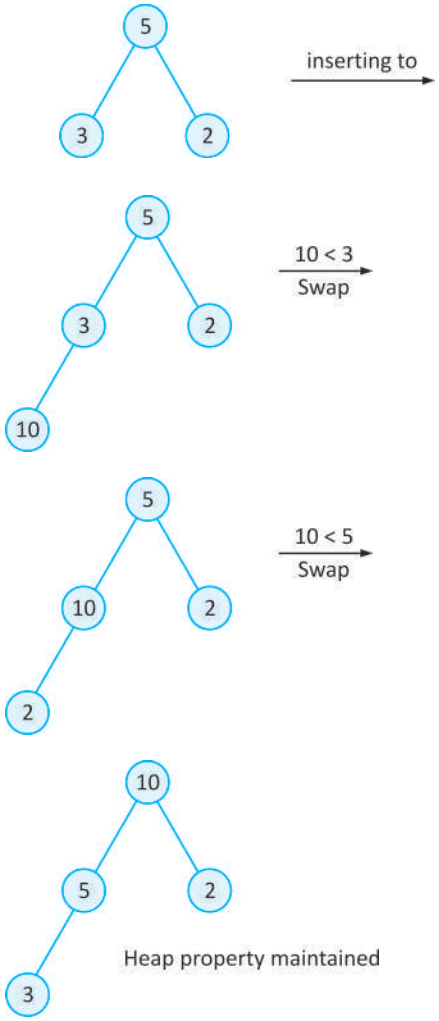
[a]

[b]

Insertion in a Heap :

For this we will talk about max heap only, min heap would be just opposite. So for insertion we create a new node and place it at the end. Now this node may or may not be at its correct position.

To take it to correct position we compare this node/value with parent, if parent is smaller then we swap and we keep repeating it until the parent is greater or it reaches the root.



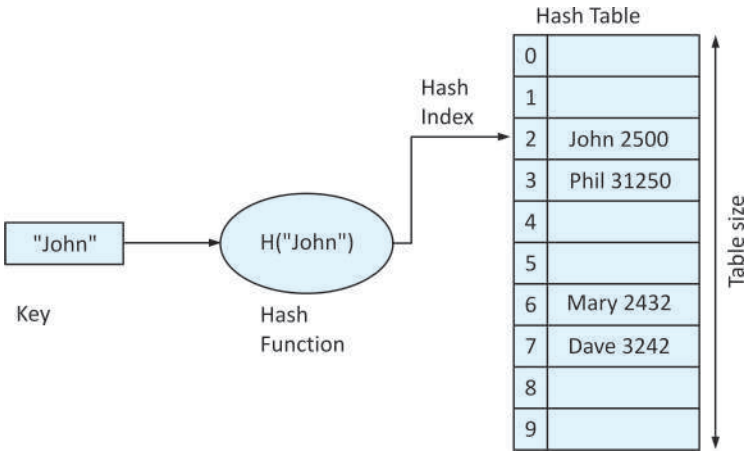
```
//Constructing a maxheap
class Heap {
    vector<int> v;
    int sze;

    int parent(int i) { return i / 2;}
    int left(int i) { return 2 * i; }
    int right(int i) { return 2 * i + 1; }

    void heapify(int i){

        // left child comparison
        int posOfLargest = i;
        if (left(i) <= sze && v[left(i)] > v[i]){
            posOfLargest = left(i);
        }

        if (right(i) <= sze && v[right(i)] > v[posOfLargest]){
            posOfLargest = right(i);
        }

        if (i != posOfLargest){
            // I HAVE to swap
            swap(v[i], v[posOfLargest]);
            heapify(posOfLargest);
        }
    }

public:
    Heap() {
        sze = 0;
        v.push_back(-1);     // empty position
    }

    int top() {
        if (empty()) return -1;
        return v[1];
```

```
    }

    bool empty() {
        return sze == 0;
    }

    void pop() {
        if (empty()) return;

        swap(v[1], v[sze]);
        v.pop_back();
        --sze;

        heapify(1);
    }

    void push(int d) {
        // push and compare with the parent
        v.push_back(d);
        ++sze;

        int i = v.size() - 1;
        while(i > 1 && v[i] > v[parent(i)]){
            swap(v[i], v[parent(i)]);
            i = parent(i);
        }
    }
};
```

16 Hashing

In Hashing the idea is to use hash function that converts a given key to a smaller number and uses the small number as index in a table called hash table.

We generate a hash for the input using the hash function, and then store the element using the generated hash as the key in the hash table.



Hash Table : Hash table is a collection of Key-Value pairs. It Used when the searching, insertion of an element is required to be fast.

Operations in hash function:

- **Insert** – T [h(key)] = value ;

  Calculate the hash, use it as the key and store the value in hash table.

- **Delete** – T [h (key)] = NULL ;

  Calculate the hash, reset the value stored in the hash table for that key.

- **Search** – return T [h (key)] ;

  Calculate the hash, find and return the value stored in the hash table for that key.

Hash Collision: when two or more inputs are mapped to same keys as used in the hash table.

Example **:** h(" john ") == h(" joe ")

Collision can not be completely avoided but can be minimised using "good" hash function and a bigger table size.

The chances of hash collision are less, if the table size if a prime number.

Characteristics of a good Hash Function :

- **Uniform Distribution :** For distribution throughout the constructed table.
- **Fast :** The generation of Hash should be very fast, and should not produce any considerable overhead.

Collision Handling Techniques :

1. **Open Hashing (Separate Chaining) :** In this using Linked List , new nodes are added to the list, the key acts as the head pointer, pointing to a number of nodes having different values.
2. **Closed Hashing (Open Addressing) :** In this we find the "next" vacant bucket in Hash Table and store the value in that bucket.
   (a) Linear Probing : We linearly go to every next bucket and see if it is vacant or not.
   (b) Quadratic Probing : We go to the 1st, 4th , 9th … bucket and check if they are vacant or not.
3. **Double Hashing :** Here we subject the generated key from the hash function to a second hash function.

Load Factor : It is a measure of how full the hash table is allowed to get before its capacity is increased.

Load factor ë of a hash table T is defined as follows:

N = number of elements in T ("current size")

M = size of T ("table size")

ë = N/M (" load factor")

Generally if load factor is greater than 0.5, we increase the size of bucket array and rehash all the key value pairs again.

Rehashing :

Process of increasing the size of the hash table when load factor becomes "too high" (defined by a threshold value) , anticipating that collisions would become higher now.

- Typically expand the table to twice its size (but still prime).
- Need to reinsert all existing elements into the new hash table.

```
// Code to construct a hashmap
struct Node {
     string key; // object
    int val;
    Node* next;
```

```
};
class Hashmap {
    Node * *table;
    int sze;
    int capacity;
    void insertIntoList(int idx, Node* tmp) {
        Node*& head = table[idx];    // reference...
        if (head == NULL) {
            head = tmp;
        }
        else {
            tmp->next = head;
            head = tmp;
        }
    }
    double loadFactor() {
        return (double)sze / capacity;
    }
    int hashFunction(const string& s) {
        int hashCode = 0;
        int mul = 31;        // he told me to take prime...research
        const int MOD = capacity;
        for (int i = 0; i < s.size(); ++i) {
            hashCode = (mul * hashCode) % MOD;
            hashCode = (hashCode + s[i] % MOD) % MOD;
            mul = (mul * 31) % MOD;
        }
        return hashCode;
    }
    void rehash() {
        Node** oldTable = table;
        int oldCapacity = capacity;
        capacity = 2 * oldCapacity;
        table = new Node*[capacity](); // WARNING! MUST be initialised
        sze = 0;     // insert function increases size
        for (int i = 0; i < oldCapacity; ++i) {
            Node* head = oldTable[i];
```

```
            while (head != NULL) {
                // insert into newtable
                Node* nextItem = head->next;
                head->next = NULL;
                insert(head);
                head = nextItem;
            }
        }
        delete [] oldTable;
    }
    void insert(Node* tmp) {
        int hashCode = hashFunction(tmp->key);
        int idx = hashCode % capacity;
        insertIntoList(idx, tmp);
        ++sze;
        if (loadFactor() > 0.7) {
            rehash();
        }
    }
public:
    Hashmap() {
        cout << "Welcome, Creating hashmap...\n";
        capacity = 7;
        sze = 0;
        table = new Node*[capacity]();
        // initialise all ptrs to NULL. WARNING!!!
    }
    void insert(const string& s, const int& val) {
        Node* tmp = new Node();
        tmp->key = s;
        tmp->val = val;
        tmp->next = NULL;
        insert(tmp);
    }
    int at(const string& s) {
        // returns the val
        int idx = hashFunction(s) % capacity;
```

```
        Node* head = table[idx];
        // search s in the list
        while (head) {
            if (head->key == s) {
                return head->val;
            }
            head = head->next;
        }
        return -1;
    }

    bool empty() {
        return sze == 0;
    }
    void remove(const string& s) {
        // delete from list
        int idx = hashFunction(s) % capacity;
        Node* head = table[idx];

        // delete element with s from the list
        Node* cur = head;
        Node* pre = NULL;
        while (cur) {
            if (cur->key == s) {
                if (cur == head) {
                    table[idx] = cur->next;
                    delete cur;
                } else {
                    pre->next = cur->next;
                    delete cur;
                }
                --sze;
            }
            pre = cur;
            cur = cur->next;
        }
    }
```

```
    ~Hashmap() {
        for (int i = 0; i < capacity; ++i) {
            Node* head = table[i];
            while (head) {
                Node* tmp = head->next;
                delete head;
                head = tmp;
            }
        }
        delete [] table;
        cout << "Bye! Dtor called\n";
    }
    void printHashMap(){
        for(int i = 0; i < capacity; ++i){
            Node* head = table[i];
            cout << i << "\t:\t";
            while(head){
                cout << head->key << "->";
                head = head->next;
            }
            cout << endl;
        }
    }
};
```

11 Graphs

Introduction :

Graphs are mathematical structures that represent pairwise relationships between objects. A graph is a flow structure that represents the relationship between various objects. It can be visualized by using the following two basic components :

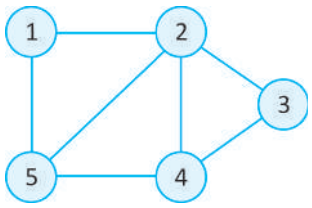
- **Nodes :** These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes A and B and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.
- **Edges :** Edges are the components that are used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

Some Real Life Applications:

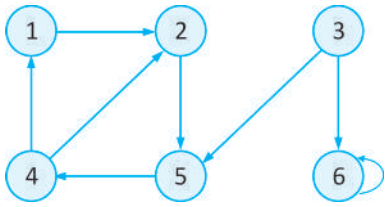
- **Google Maps :** To find a route based on shortest route/Time.
- **Social Networks :** Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
- **Web Search :** Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks, each page is a vertex and the link between two pages is an edge.
- **Recommendation System :** On eCommerce websites relationship graphs are used to show recommendations.

Types of Graphs :

- **Undirected :** An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.



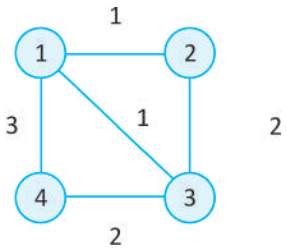
- **Directed :** A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



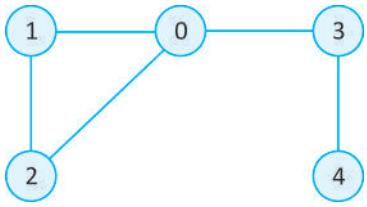
- **Weighted :** In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:

  - 1 -> 2 -> 3
  - 1 -> 3
  - 1 -> 4 -> 3

  Therefore the total cost of each path will be as follows: The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units



- **Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

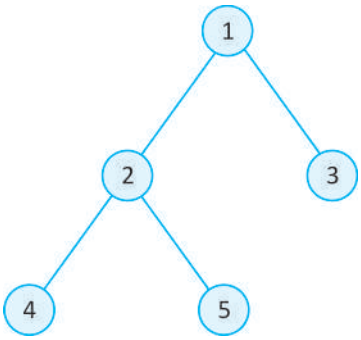


Cycle presents in the above graph (0->1->2)

A tree is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has N - 1 edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

***Note*:** A root node has no parent.

A tree cannot contain any cycles or self loops, however, the same does not apply to graphs.



(**Tree : There is no any cycle or self loop in this graph**)

Graph Representation

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

1. Adjacency Matrix

An adjacency matrix is a $V * V$ binary matrix A. Element $A_{i, j}$ is 1 if there is an edge from vertex i to vertex j else $A_{i, j}$ is 0.

Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in $A_{i, j}$ the weight or cost of the edge will be stored.

In an undirected graph, if $A_{i, j} = 1$, then $A_{j, i} = 1$.

In a directed graph, if $A_{i, j} = 1$, then may or may not be 1.

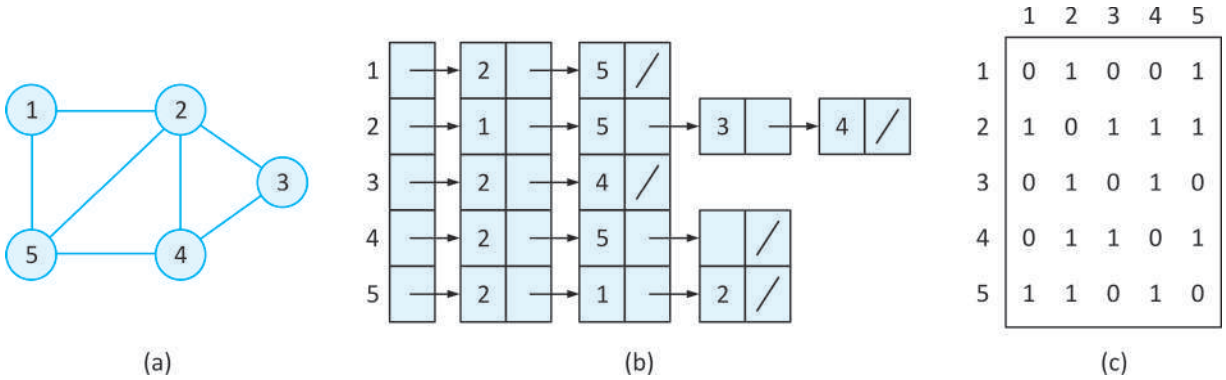
Adjacency matrix provides constant time access (O(1)) to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is $O(V^2)$.

2. Adjacency List

The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array A_i is a list, which contains all the vertices that are adjacent to vertex i.

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list A_i then vertex i will be in list A_j.

The space complexity of adjacency list is O(V + E)because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.

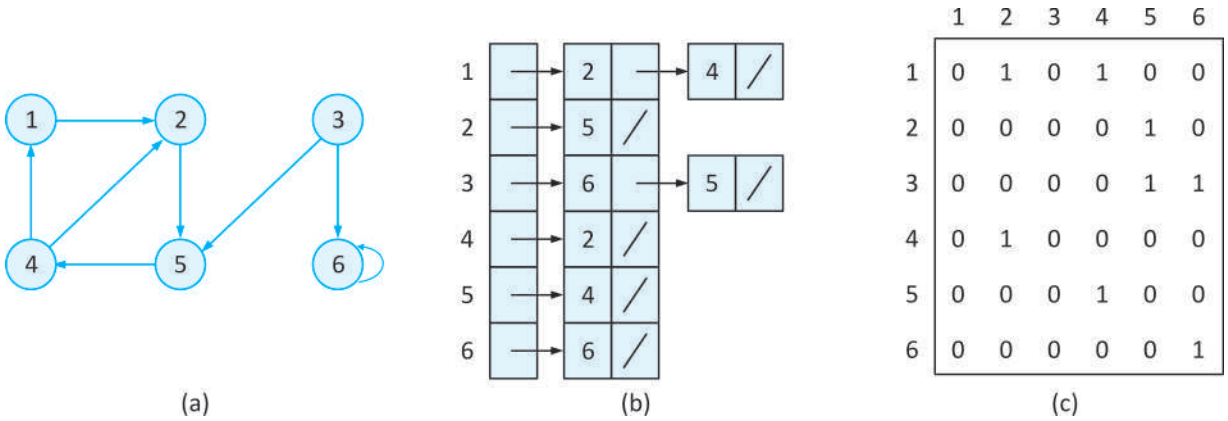


Two representations of an undirected graph.

(a) An undirected graph G having five vertices and seven edges

(b) An adjacency-list representation of G

(c) The adjacency-matrix representation of G



Code for Adjacency list representation of a graph :

```
#include<iostream>
#include<list>
using namespace std;
class Graph{
      int V;
      list<int> *l;
public:
      Graph(int v){
            V = v;
            //Array of Linked Lists
            l = new list<int>[V];
      }
      void addEdge(int u,int v,bool bidir=true){
```

```
l[u].push_back(v);
            if(bidir){
                  l[v].push_back(u);
            }
      }
      void  printAdjList(){
            for(int  i=0;i<V;i++){
                        cout<<i<<“->”;
                        //l[i] is a linked list
                        for(int  vertex:  l[i]){
                              cout<<vertex<<“,”;
                        }
                        cout<<endl;
            }
      }
};
int  main(){
      // Graph has 5 vertices number from 0 to 4
      Graph  g(5);
      g.addEdge(0,1);
      g.addEdge(0,4);
      g.addEdge(4,3);
      g.addEdge(1,4);
      g.addEdge(1,2);
      g.addEdge(2,3);
      g.addEdge(1,3);
      g.printAdjList();
return  0;
}
```

Graph Traversal :

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.

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

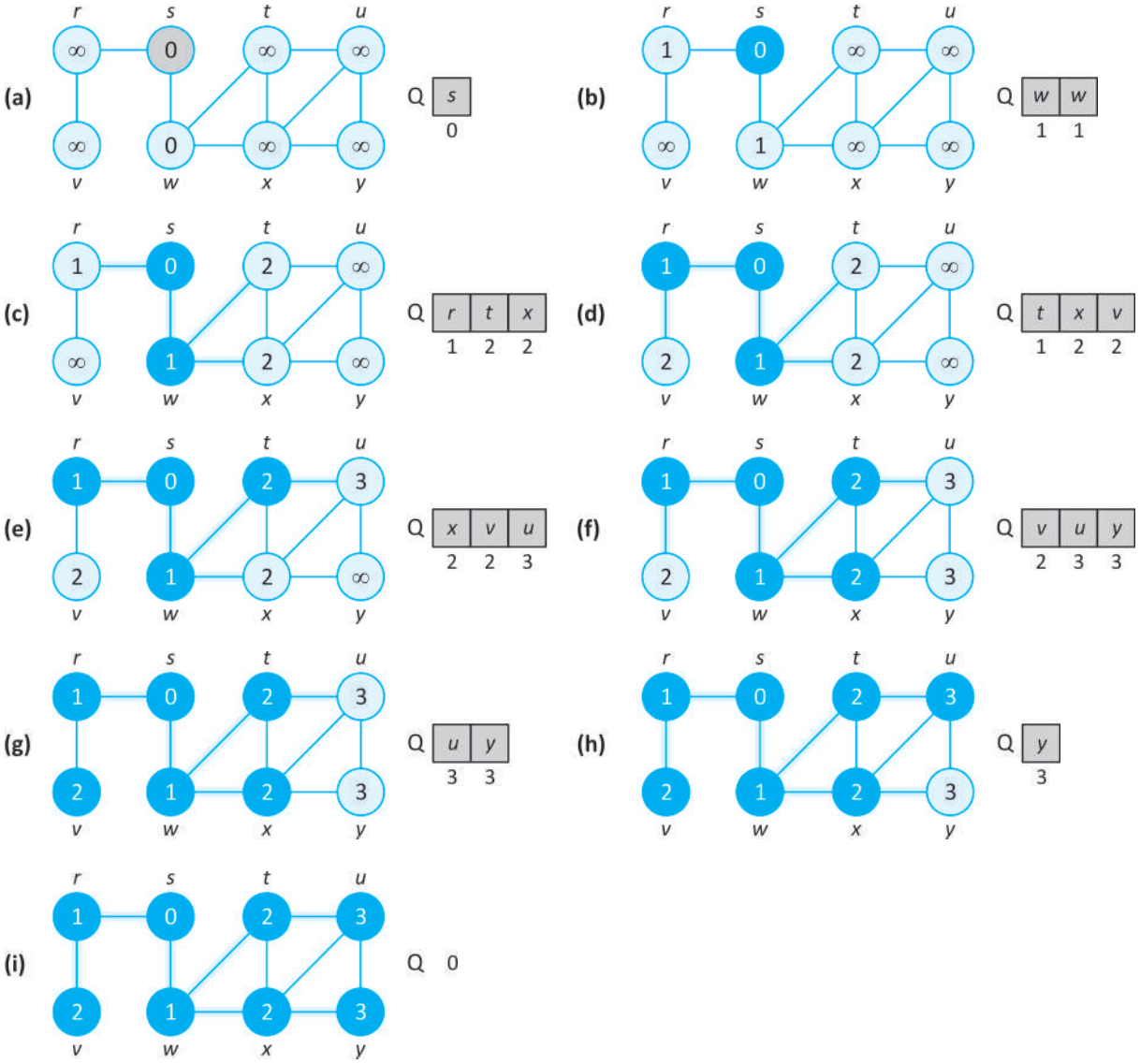
Breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is *discovered* the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If (*u*, v) " *E* and vertex *u* is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex *s*. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex *u*, the vertex v and the edge (*u*, v) are added to the tree. We say that *u* is the *predecessor* or *parent* of v in the breadth-first tree.

Algorithm :

```
BFS(G, s)
    for each vertex u ∈ V [G] – {s}
        do color[u] ← WHITE
           d[u] ← ∞
           π[u] ← NIL
    color[s] ← GRAY
    d[s] ← 0
    π[s] ← NIL
    Q ← 0
    ENQUEUE(Q, s)
    while Q ≠ 0
       do u ← DEQUEUE(Q)
          for each ∈ Adj[u]
              do if color[v] = WHITE
                 then color[v] ← GRAY
```

```
    d[v] ← d[u] + 1
                    π[v] ← u
                    ENQUEUE(Q, v)

    color[u] ← BLACK
```



The operation of BFS on an undirected graph. Tree edges are shown shaded as they are prduced by BFS. Within each vertex vertex u is shown d[u]. The queue Q is shown at the beginning of each iteration of the while loop of lines 10-18. Vertex distances are shown next to vertices in the queue.

Code :

```
#include<iostream>
#include<map>
#include<list>
#include<queue>
using namespace std;

template<typename T>
class Graph{

    map<T,list<T> > adjList;

public:
    Graph(){

    }
    void addEdge(T u, T v,bool bidir=true){

        adjList[u].push_back(v);
        if(bidir){
            adjList[v].push_back(u);
        }
    }

    void print(){
```

Time Complexity :

Time complexity and space complexity of above algorithm is O(V + E) and O(V).

Application of BFS

1. **Shortest Path and Minimum Spanning Tree for unweighted graph** In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

   Code for finding shortest path using BFS :

```
#include<iostream>
#include<map>
#include<list>
```

```
#include<queue>
using namespace std;
template<typename T>
class Graph{
            map<T,list<T> > adjList;
public:
            Graph(){
 }
            void addEdge(T u, T v,bool bidir=true){
                  adjList[u].push_back(v);
            if(bidir){
                  adjList[v].push_back(u);
            }
      }

      void print(){
            //Iterate over the map
            for(auto i:adjList){
                  cout<<i.first<<"->";
            //i.second is LL
            for(T entry:i.second){
                  cout<<entry<<",";
            }
            cout<<endl;
      }
}
voidbfs(T src){
      queue<T> q;
      map<T,int>dist;
      map<T,T> parent;
      for(auto i:adjList){
            dist[i.first] = INT_MAX;
      }
      q.push(src);
```

```
                dist[src] = 0;
                parent[src] = src;
                while(!q.empty()){
                        T node = q.front();
                        cout<<node<<" ";
                        q.pop();
                        // For the neigbours of the current node, find out the nodes which
                are not visited
                        for(intneigbour :adjList[node]){
                                if(dist[neigbour]==INT_MAX){
                                        q.push(neigbour);
                                        dist[neigbour] = dist[node]  + 1;
                                        parent[neigbour] = node;
                                }
                        }
                }
                //Print the distance to all the nodes
                for(auto i:adjList){
                        T node = i.first;
                        cout<<"Dist of "<<node<<" from "<<src<<" is "<<dist[node]<<endl;
                }
        }
};
int main(){
        Graph<int> g;
        g.addEdge(0,1);
        g.addEdge(1,2);
        g.addEdge(0,4);
        g.addEdge(2,4);
        g.addEdge(2,3);
        g.addEdge(3,5);
        g.addEdge(3,4);
        g.bfs(0);
}
```

2. **Peer to Peer Networks :** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3. **Crawlers in Search Engines :** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
4. **Social Networking Websites :** In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5. **GPS Navigation systems :** Breadth First Search is used to find all neighboring locations.
6. **Broadcasting in Network :** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7. **In Garbage Collection :** Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
8. **Cycle detection in undirected graph :** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.
9. **Ford–Fulkerson algorithm :** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).
10. **To test if a graph is Bipartite :** We can either use Breadth First or Depth First Traversal.
11. **Path Finding :** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
12. **Finding all nodes within one connected component :** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

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 be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

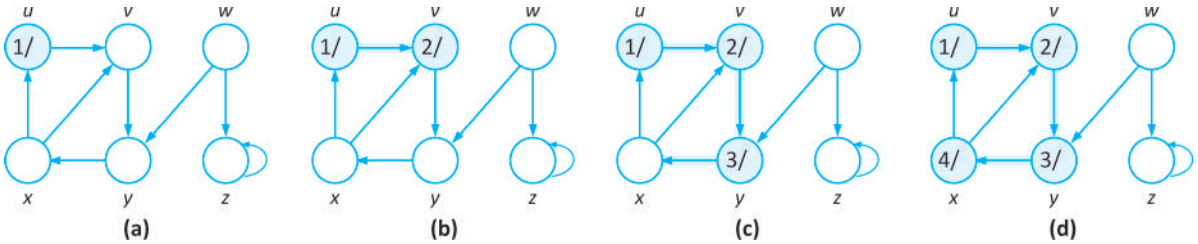
- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

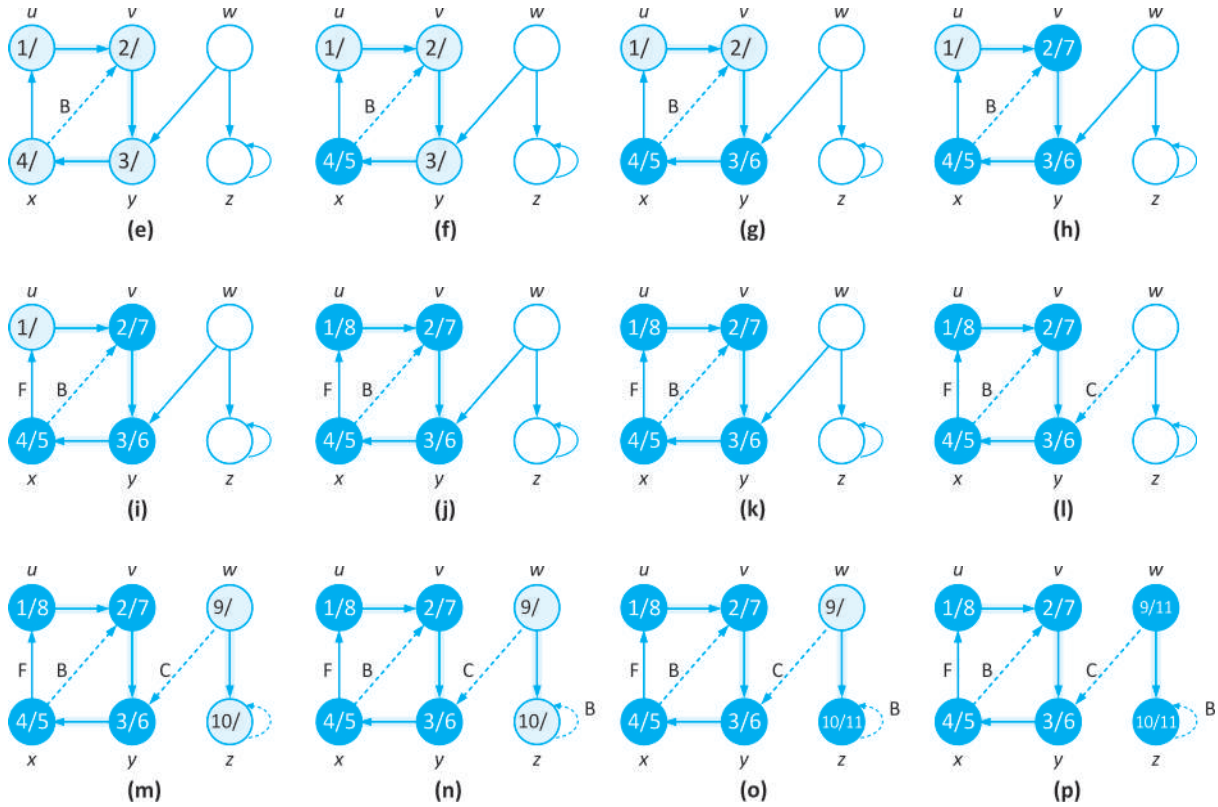
- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

  As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Algorithm :

```
DFS(G)
for each vertex u ∈ V[G]
     do color [u] ← WHITE
         p[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
     do if color[u] = WHITE
           then DFS-VISIT(u)
DFS-VISIT(u)
color[u] ← GRAY        White vertex u has just been discovered
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]          Explore edge (u, v)
     do if color[v] = WHITE
           then π[v] ← u
                DFS-VISIT(v)
color[u] BLACK          Blacken u; it is finished
f[u] ← time ← time + 1
```





The progress of the depth-first-search algorithm DFS on a directed path. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

Code :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;

template<typename T>
class Graph{

    map<T,list<T> > adjList;

public:
    Graph(){

    }
    void addEdge(T u, T v,bool bidir=true){
        adjList[u].push_back(v);
```

```
            if(bidir){
                  adjList[v].push_back(u);
            }
      }
      void  print(){
            //Iterate over the map
            for(auto  i:adjList){
                  cout<<i.first<<"->";

                  //i.second is LL
                  for(T  entry:i.second){
                        cout<<entry<<",";
                  }
                  cout<<endl;
            }
      }
      void  dfsHelper(T  node,map<T,bool>  &visited){
            //Whenever to come to a node, mark it visited
            visited[node]  =  true;
            cout<<node<<"  ";
            //Try to find out a node which is neigbour of current node and not
yet  visited
            for(T  neighbour:  adjList[node]){
                  if(!visited[neighbour]){
                        dfsHelper(neighbour,visited);
                  }
            }
      }
      void  dfs(T  src){
            map<T,bool>  visited;
            dfsHelper(src,visited);
      }
};
int  main(){

      Graph<int>  g;
      g.addEdge(0,1);
      g.addEdge(1,2);
      g.addEdge(0,4);
```

```
g.addEdge(2,4);
     g.addEdge(2,3);
     g.addEdge(3,4);
     g.addEdge(3,5);
     g.dfs(0);


return 0;
}
```

Time Complexity :

Time complexity of above algorithm is O(V + E).

Application of DFS :

1. For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.
2. **Detecting cycle in a graph :** A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edge.

Code for detect a cycle in a graph :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;
template<typename T>
class Graph{
     map<T,list<T> > adjList;
public:
     Graph(){
     }
     void addEdge(T u, T v,bool bidir=true){
           adjList[u].push_back(v);
           if(bidir){
                 adjList[v].push_back(u);
           }
     }
     bool isCyclicHelper(T node,map<T,bool> &visited,map<T,bool> &inStack){
```

```
                //Processing the current node - Visited, InStack
                visited[node] = true;
                inStack[node] = true;
```

3. **Path Finding :** We can specialize the DFS algorithm to find a path between two given vertices u and z.

   (i) Call DFS(G, u) with u as the start vertex.

   (ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

   (iii) As soon as destination vertex z is encountered, return the path as the contents of the stack.

4. **Topological Sorting** : In the field of computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering.

Code for printing Topological sorting of DAG :

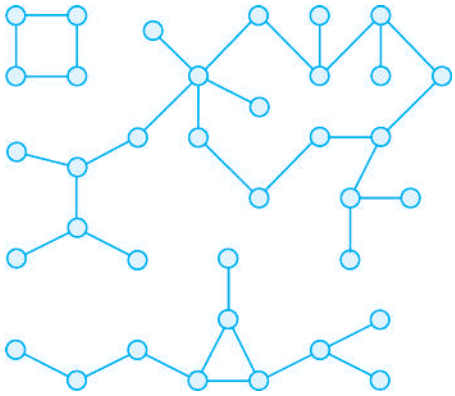
```
#include<iostream>
#include<map>
#include<list>
#include<queue>
using namespace std;
template<typename T>
class Graph{
     map<T,list<T> > adjList;
 public:
     Graph(){
     }
     void addEdge(T u, T v,bool bidir=true){
           adjList[u].push_back(v);
           if(bidir){
                 adjList[v].push_back(u);
           }
     }
     void topologicalSort(){
```

```
queue<T> q;
            map<T,bool> visited;
            map<T,int> indegree;
            for(auto i:adjList){
                        //i is pair of node and its list
                        T node = i.first;
                        visited[node] = false;
                        indegree[node] = 0;
            }
            //Init the indegrees of all nodes
            for(auto i:adjList){
                  T u = i.first;
                  for(T v: adjList[u]){
                  indegree[v]++;
                  }
            }
            //Find out all the nodes with 0 indegree
            for(auto i:adjList){
                  T node = i.first;
                  if(indegree[node]==0){
                        q.push(node);
                  }
            }
            //Start with algorithm
            while(!q.empty()){
                  T node = q.front();
                  q.pop();
                  cout<<node<<"->";
                  for(T neighbour:adjList[node]){
                        indegree[neighbour]--;
                  if(indegree[neighbour]==0){
                        q.push(neighbour);
                  }
            }
```

```
            }
        }
};

int main(){
    Graph<string> g;
    g.addEdge("English","ProgrammingLogic",false);
    g.addEdge("Maths","Programming Logic",false);
    g.addEdge("Programming Logic","HTML",false);
    g.addEdge("Programming Logic","Python",false);
    g.addEdge("Programming Logic","Java",false);
    g.addEdge("Programming Logic","JS",false);
    g.addEdge("Python","WebDev",false);
    g.addEdge("HTML","CSS",false);
    g.addEdge("CSS","JS",false);
    g.addEdge("JS","WebDev",false);
    g.addEdge("Java","WebDev",false);
    g.addEdge("Python","WebDev",false);
    g.topologicalSort();

return0;
}
```

5. **To test if a graph is bipartite :** We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black.
6. **Finding Strongly Connected Components of a graph :** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
7. **Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
8. **For finding Connected Components :** In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Example **:**



In the above picture, there are three connected components.

Code for finding connected components :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;

template<typename T>
class Graph{
      map<T,list<T> > adjList;
public:
      Graph(){
      }
      void addEdge(T u, T v,bool bidir=true){
              adjList[u].push_back(v);
              if(bidir){
                      adjList[v].push_back(u);
              }
       }
       void print(){
            //Iterate over the map
            for(auto i:adjList){
                  cout<<i.first<<"->";
                  //i.second is LL
```

```
            for(T  entry:i.second){
                  cout<<entry<<“,”;
            }
            cout<<endl;
      }
}
void  dfsHelper(T  node,map<T,bool>  &visited){
      //Whenever  to  come  to  a  node,  mark  it  visited
      visited[node]  =  true;
      cout<<node<<“  “;
      //Try to find out a node which is neigbour of current node and not yet visited
      for(T  neighbour:  adjList[node]){
            if(!visited[neighbour]){
                  dfsHelper(neighbour,visited);
            }
      }
}
void  dfs(T  src){
      map<T,bool>  visited;
      int  component  =  1;
      dfsHelper(src,visited);
      cout<<endl;
      for(auto  i:adjList){
            T  city  =  i.first;
            if(!visited[city]){
                  dfsHelper(city,visited);
                  component++;
            }
      }
cout<<endl;
            cout<<“The  current  graph  had  “<<component<<“  components”;
      }
};
int  main(){
      Graph<string>  g;
```

```
g.addEdge("Amritsar","Jaipur");
      g.addEdge("Amritsar","Delhi");
      g.addEdge("Delhi","Jaipur");
      g.addEdge("Mumbai","Jaipur");
      g.addEdge("Mumbai","Bhopal");
      g.addEdge("Delhi","Bhopal");
      g.addEdge("Mumbai","Bangalore");
      g.addEdge("Agra","Delhi");
      g.addEdge("Andaman","Nicobar");
      g.dfs("Amritsar");
return 0;
}
```

Minimum Spanning Tree

What is a Spanning Tree?

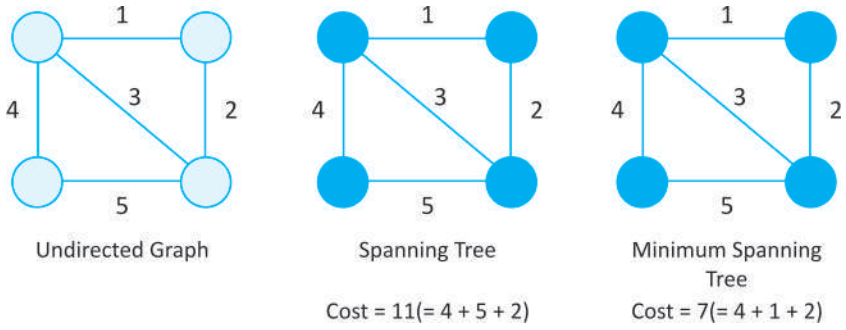
Given an undirected and connected graph G=(V,E) , a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is **"Disjoint Sets"**:

Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

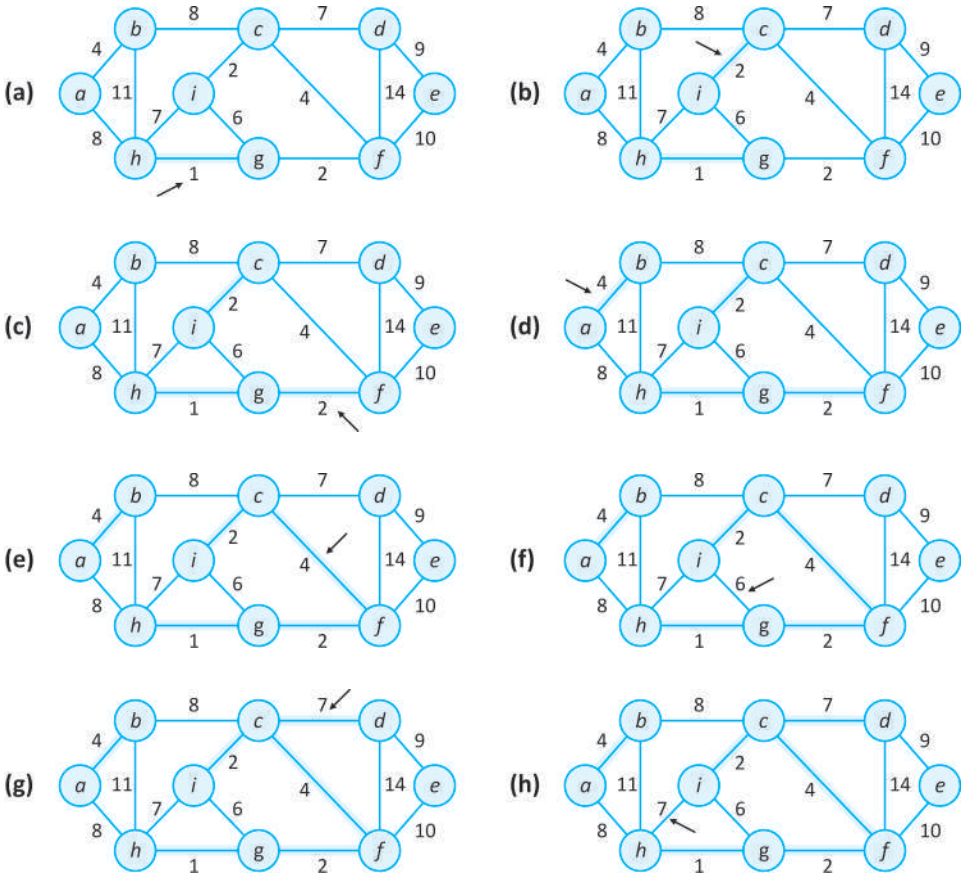
In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first.

Note **:** Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Algorithm :

```
MST-KRUSKAL(G, w)
A ← 0
For each vertex v ∈ V[G]
    Do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) ∈ E, taken in nondecreasing order by weight
    do if FIND-SET(u)≠ FIND-SET(v)
          then A ← A ∪ {(u, v)}
               UNION(u, v)
return A
```

Here **A** is the set which contains all the edges of minimum spanning tree.



The execution of Kruskal's algorithm on the graph from figure. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Code :

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void initialize()
{
    for(int i = 0;i < MAX;++i)
```

```
        id[i] = i;
}

int root(int x)
{
    while(id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
    }
    return x;
}

void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}

long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
        // Selecting edges one by one in increasing order from the beginning
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check if the selected edge is creating a cycle or not
        if(root(x) != root(y))
        {
            minimumCost += cost;
            union1(x, y);
        }
```

```
  }
    return minimumCost;
}

int main()
{
   int x, y;
    long long weight, cost, minimumCost;
    initialize();
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
         p[i] = make_pair(weight, make_pair(x, y));
    }
    // Sort the edges in the ascending order
    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity :

The total running time complexity of kruskal algorithm is O(V log E).

Prim's Algorithm :

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps :

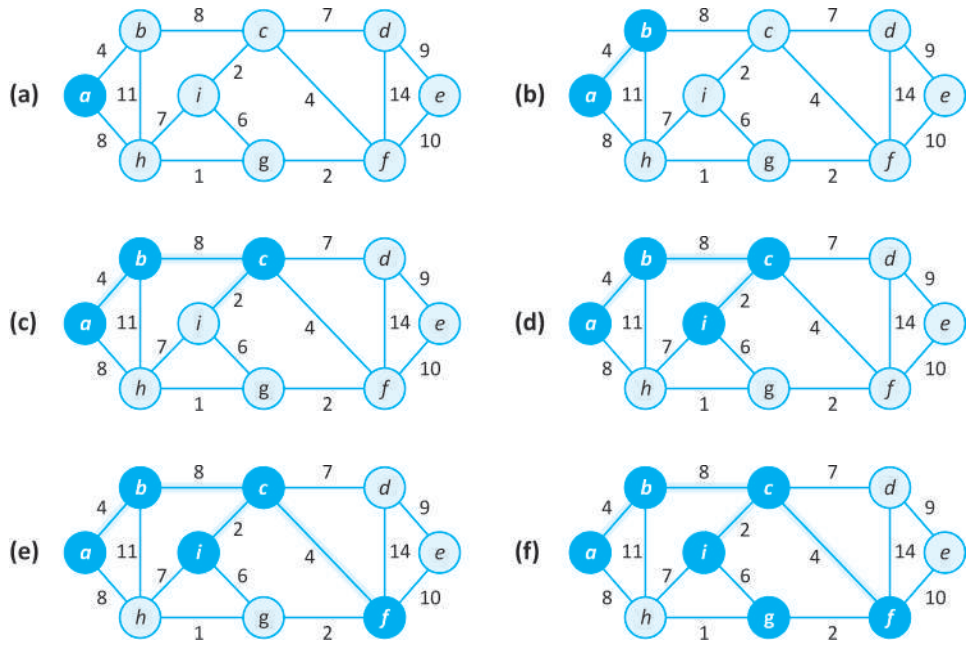
- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

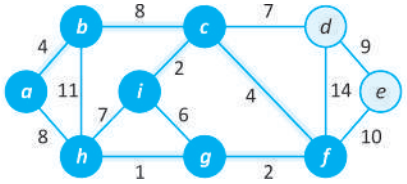
  In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex.

```
MST-PRISM(G, w, r)
   for each u ∈ V[G]
        do key[u] ← ∞
           p[u] ← NIL
   key[r] ← 0
   Q ← V[G]
   while Q ≠ 0
       do u ← EXTRACT-MIN(Q)
          for each v ∈ Adj[u]
              do if v ∈ Q and w(u, v) < key[v]
                    then π[v] ← u
                         key[v] ← w(u, v)
```

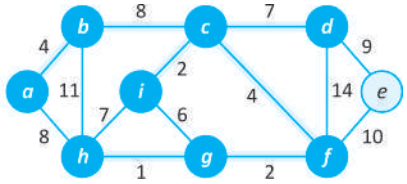
The prims algorithm works as shown in below graph.



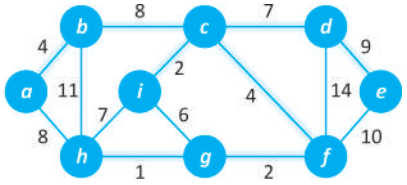
(g)



(h)



(i)



Code :

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>
using namespace std;
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
   priority_queue<PII, vector<PII>, greater<PII> > Q;
  int y;
   long long minimumCost = 0;
  PII p;
   Q.push(make_pair(0, x));
   while(!Q.empty())
  {
       // Select the edge with minimum weight
       p = Q.top();
       Q.pop();
       x = p.second;
       // Checking for cycle
```

```
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
       {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
       }
   }
   return minimumCost;
}
int main()
{
   int nodes, edges, x, y;
   long long weight, minimumCost;
   cin >> nodes >> edges;
   for(int i = 0;i < edges;++i)
   {
        cin >> x >> y >> weight;
         adj[x].push_back(make_pair(weight, y));
         adj[y].push_back(make_pair(weight, x));
   }
   // Selecting 1 as the starting node
   minimumCost = prim(1);
   cout << minimumCost << endl;
   return 0;
}
```

Shortest Path Algorithms

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

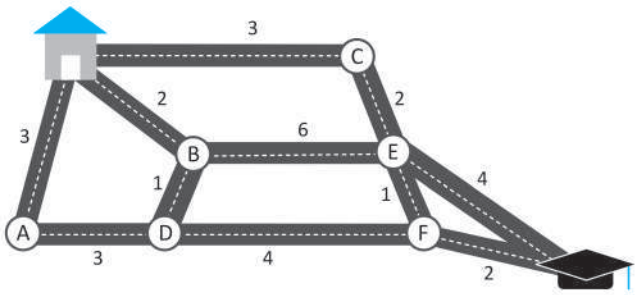
This problem could be solved easily using (BFS) if all edge weights were (1), but here weights can take any value. There are some algorithm discussed below which work to find Shortest path between two vertices.

1. Single Source Shortest Path Algorithm :

In this kind of problem, we need to find the shortest path of single vertices to all other vertices.

Example **:**

Find the shortest path from home to school in the following graph:



A weighted graph representing roads from home to school

The shortest path, which could be fund using Dijkstra's algorithm, is

Home → B → D → F → School

There are two algorithm works to find Single source shortest path from a given graph.

A. Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Pseudo Code :

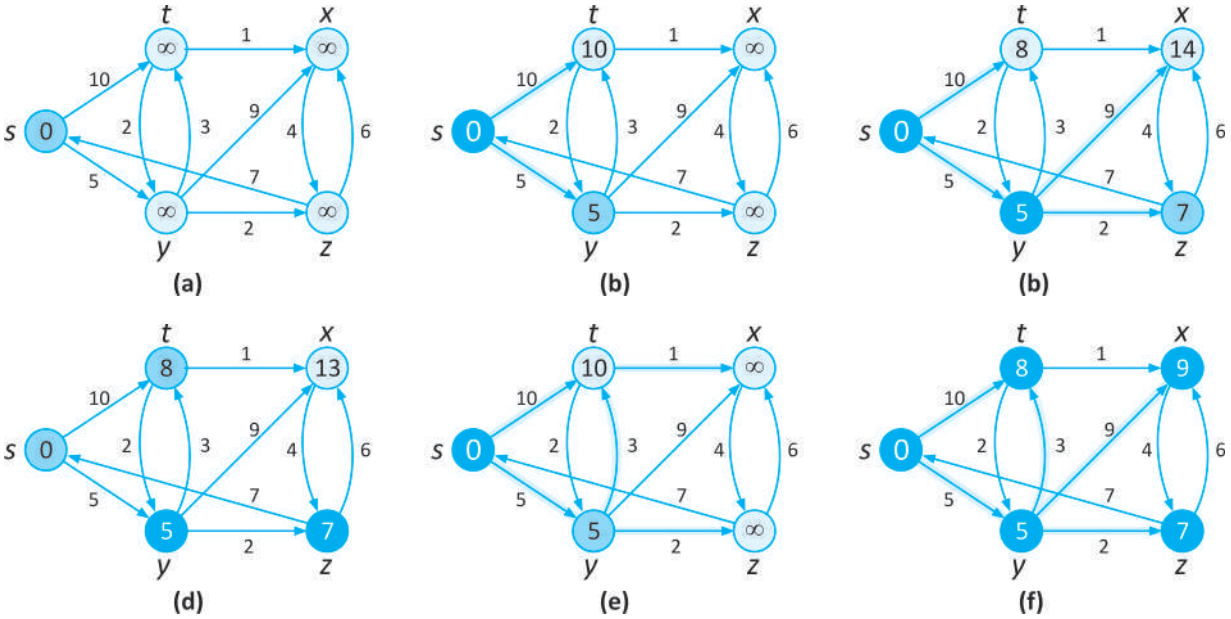
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← 0
Q ← V[G]
while Q ≠ 0
```

```
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

For relaxing an edge of given graph, Algorithm works likes this :

```
RELAX(u, v, w)
   if d[v] > d[u] + w(u, v)
      then d[v] ← d[u] + w(u, v)
           π[v] ← u
```

Example :



The execution of Dijkstra's algorithm. The source *s* is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set *S*, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop. The shaded vertex has the minimum *d* value and is chosen as vertex *u* . **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex *u* in line 5 of the next iteration. The *d* and À values shown in part (f) are the final values.

Code :

```
#include<bits/stdc++.h>
using namespace std;
template<typename T>
class Graph{
    unordered_map<T, list<pair<T,int> > > m;
public:
```

```
void addEdge(T u,T v,int dist,bool bidir=true){
     m[u].push_back(make_pair(v,dist));
    if(bidir){
         m[v].push_back(make_pair(u,dist));
    }
}
void printAdj(){
    //Let try to print the adj list
    //Iterate over all the key value pairs in the map
    for(auto j:m){
         cout<<j.first<<"->";
         //Iterater over the list of cities
         for(auto l: j.second){
                  cout<<"("<<l.first<<","<<l.second<<")";
        }
         cout<<endl;
    }
}
void dijsktraSSSP(T src){
     unordered_map<T,int> dist;
     //Set all distance to infinity
    for(auto j:m){
         dist[j.first] = INT_MAX;
    }
   //Make a set to find a out node with the minimum distance
    set<pair<int, T> > s;
   dist[src] = 0;
    s.insert(make_pair(0,src));
   while(!s.empty()){
        //Find the pair at the front.
        auto p =   *(s.begin());
        T node = p.second;
        int nodeDist = p.first;
        s.erase(s.begin());
       //Iterate over neighbours/children of the current node
        for(auto childPair: m[node]){
```

```
                if(nodeDist + childPair.second < dist[childPair.first]){
                    //In the set updation of a particular is not possible
                     // we have to remove the old pair, and insert the new pair
to   simulation updation
                    T dest = childPair.first;
                     auto f = s.find( make_pair(dist[dest],dest));
                    if(f!=s.end()){
                        s.erase(f);
                    }
                    //Insert the new pair
                     dist[dest] = nodeDist + childPair.second;
                     s.insert(make_pair(dist[dest],dest));
                }
            }
        }
        //Lets print distance to all other node from src
        for(auto d:dist){
             cout<<d.first<<" is located at distance of  "<<d.second<<endl;
        }
    }
};
int main(){
    Graph<int> g;
    g.addEdge(1,2,1);
    g.addEdge(1,3,4);
    g.addEdge(2,3,1);
    g.addEdge(3,4,2);
    g.addEdge(1,4,7);
    //g.printAdj();
   // g.dijsktraSSSP(1);
    Graph<string> india;
     india.addEdge("Amritsar","Delhi",1);
     india.addEdge("Amritsar","Jaipur",4);
     india.addEdge("Jaipur","Delhi",2);
 india.addEdge("Jaipur","Mumbai",8);
     india.addEdge("Bhopal","Agra",2);
```

```
    india.addEdge("Mumbai","Bhopal",3);
    india.addEdge("Agra","Delhi",1);
   //india.printAdj();
    india.dijsktraSSSP("Amritsar");
return 0;
}
```

Time Complexity : Time Complexity of Dijkstra's Algorithm is $O(V^2)$ but with min-priority queue it drops down to $O(V + E \log V)$.

B. Bellman Ford's Algorithm

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most n – 1 edges, because the shortest path couldn't have a cycle.

So why shortest path shouldn't have a cycle ?

There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

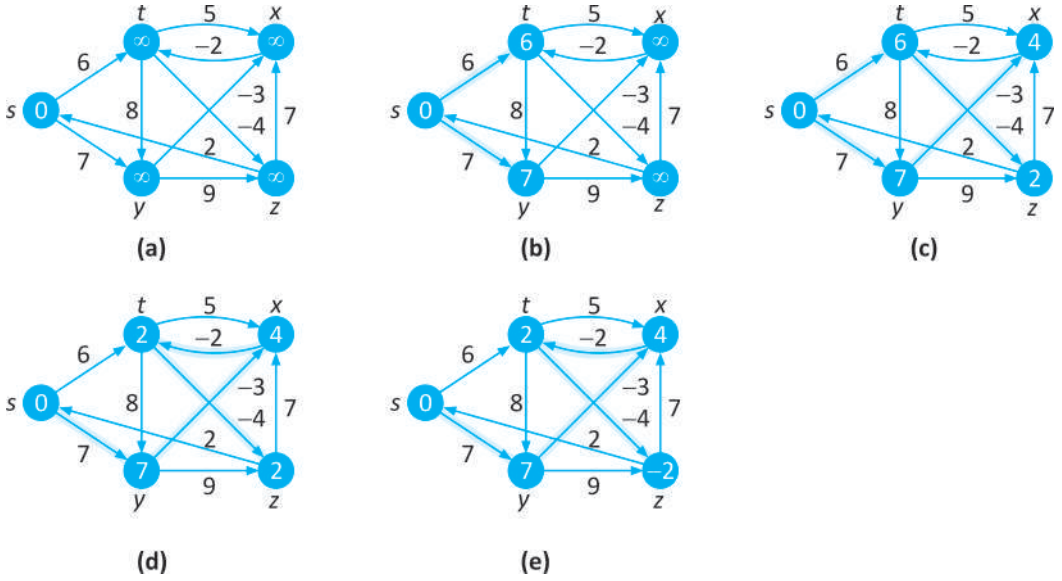
- The outer loop traverses from 0 to n – 1.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Pseudo Code :

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| - 1
     do for each edge (u, v) ∈ E[G]
            do RELAX(u, v, w)
for each edge (u, v) ∈ E[G]
     do if d[v] > d[u] + w(u, v)
           then return FALSE
return TRUE
```

Example :



All-Pairs Shortest Paths :

The all-pairs shortest path problem is the determination of the shortest graph distances between every pair of vertices in a given graph. The problem can be solved using n applications of Dijkstra's algorithm at each vertex if graph doesn't contains negative weight. We use two algorithms for finding All-pair shortest path of a graph.

1. Floyd-Warshall's Algorithm
2. Johnson's Algorithm

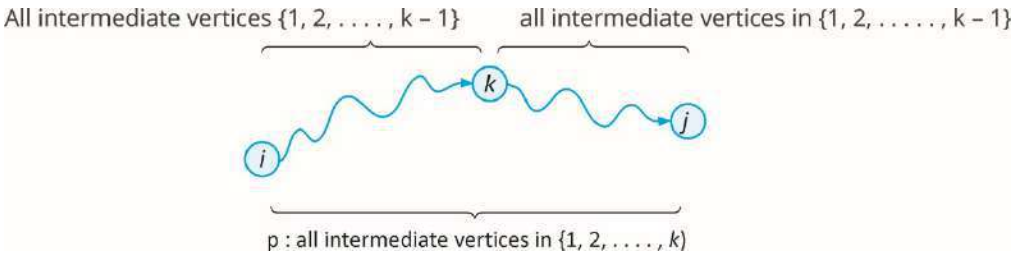
Floyd-Warshall's Algorithm :

Here we use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. The resulting algorithm, known as the ***Floyd-Warshall algorithm***. Floyd–Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in $O(V^3)$, where V is the number of vertices in a graph.

The Floyd-Warshall algorithm is based on the following observation. Under our assumption that the vertices of G are $V = \{1, 2, \ldots, n\}$, let us consider a subset $\{1,2,\ldots,k\}$ of vertices for some k. For any pair of vertices i, $j – V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. The Floyd- Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k – 1\}$. The relationship depends on whether or not k is an intermediate vertex of path p. There would two possibilities :

1. If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1,2,\ldots,k – 1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1,2,\ldots,k –1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

2. If *k* is an intermediate vertex of path *p*, then we break *p* down into *i* -> *k* -> *j* as *p*1 is a shortest path from *i* to *k* *and p*2 is a shortest path from *k* to *j. Since* *p*1 is a shortest path from *i* to *k* with all intermediate vertices in the set {1, 2, . . . , *k*}. Because vertex *k* is not an intermediate vertex of path *p*1, we see that *p*1 is a shortest path from *i* to *k* with all intermediate vertices in the set {1, 2, . . . , *k* – 1}. Similarly, *p*2 is a shortest path from vertex *k* to vertex *j* with all intermediate vertices in the set {1,2,...,*k* – 1}.



Path P is a shortest path from vertices *i* to vertex j, and k is the highest=numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set {1, 2, ... , k – 1}. The same holds for path p_2 from vertex k to vertex j.

Let $d_{ij}^{(k)}$ be the weight of shortest path from vertex *i* to vertex *j* for which all intermediate vertices are in the set {1, 2, . . . , *k*}. A recursive definition following the above discussion is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if} \quad k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if} \quad k \geq 1 \end{cases}$$

The Algorithm Steps:

For a graph with V vertices:

- Initialize the shortest paths between any 2 vertices with Infinity.
- Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all V vertices as intermediate nodes.
- Minimize the shortest paths between any 2 pairs in the previous operation.
- For any 2 vertices (i , j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be:

  Min (dist[i][k] + dist[k][j] , dist[i][j])

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k , j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

Constructing a shortest Path

For construction of the shortest path, we can use the concept of predecessor matrix π to construct the path. We can compute the predecessor matrix π "on-line" just as the Floyd-Warshall algorithm computes the matrices $D(k)$. Specifically, we compute a sequence of matrices $\pi^{(k)}$ where $\pi^{(k)}$ is defined to be the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set {1,2,...,k}. We can give a recursive formulation of $\pi^{(k)}$ as

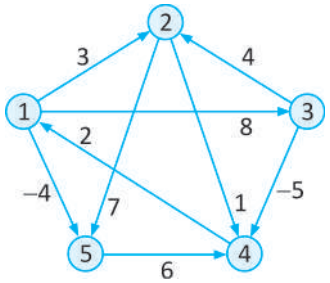
For k = 0 :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NILL} & \text{if} \quad i = j \text{ or } w_{ij} = \infty \\ i & \text{if} \quad i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

And For 1<=k<=V

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if} \quad d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ i & \text{if} \quad d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Consider the below graph and find distance (D) and parents (π) matrix for this graph



Computed distance (D) and parents (π) matrix for the above graph :

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Implemented Code :

```
#include<bits/stdc++.h>
#include<iostream>
#include<unordered_map>
#define INF 99999
#define V 5
using namespace std;
template<typename T>
class Graph
{
   unordered_map<T, list<pair<T, int> > > m;
   int graph[V][V];
   int parent[V][V];

public:
 //Adjacency list representation of the graph
 void addEdge(T u, T v, int dist,bool bidir = false)
 {
      m[u].push_back(make_pair(v,dist));
     /*if(bidir){
```

```
            m[v].push_back(make_pair(u,dist));
        }*/

    }
    //Print Adjacency list
    void printAdj()
    {
        for(auto j:m)
        {
            cout<<j.first<<"->";
            for(auto l: j.second)
            {
                cout<<"("<<l.first<<","<<l.second<<")";
            }
            cout<<endl;
        }
    }
    void matrix_form(int u, int v , int w)
    {
      graph[u-1][v-1] = w;
      parent[u-1][v-1] = u;
      return;
    }

//Adjacency matrix representation of the graph
    void matrix_form2()
    {
        for(int i=0;i<V;i++)
        {
            for(int j=0;j<V;j++)
            {
                if(i==j)
                {
                    graph[i][j]=0;
                    parent[i][j]=0;
                }
```

```
else
            {
                graph[i][j]=INF;
                parent[i][j]=0;
            }
        }
    }
    return;
}
//Print Adjacency matrix
 void print_matrix()
{
     for(int i=0;i<V;i++)
    {
         for(int j=0;j<V;j++)
        {
             if(graph[i][j]==INF)
             cout<<"INF"<<"   ";
            else
             cout<<graph[i][j]<<"    ";
        }
         cout<<endl;
    }
}
//Print predecessor matrix
 void printParents(int p[][V])
{

     cout<<"Parents Matrix"<<"\n";
  for (int i = 0; i < V; i++)
  {
      for (int j = 0; j < V; j++)
      {
          if(p[i][j]!=0)
         cout<<p[i][j]<<"   ";
          else
```

```
          {
           cout<<"NIL"<<" ";
           //cout<<p[i][j]<<"   ";
          }
       }
         cout<<endl;
   }
 }
 //All pair shortest path matrix i.e D
 void printSolution(int dist[][V])
{

            cout<<"Following matrix shows the shortest distances"
           " between every pair of vertices"<<"\n";
   for (int i = 0; i < V; i++)
   {
        for (int j = 0; j < V; j++)
       {
            if (dist[i][j] == INF)
                cout<<"INF   ";
           else
                cout<<dist[i][j]<<"   ";
       }
         cout<<endl;
   }
}
 //Print the shortest path , distance and all intermediate vertex
 void print_path(int p[][V], int d[][V])
{
  // cout<<"Hello"<<"\n";
   for(int i=0;i<V;i++)
   {
        for(int j=0;j<V;j++)
       {
  // cout<<"Hello1"<<"\n";
           if(i!=j)
```

```
            {
             cout<<“Shortest path from “<<i+1<<“ to “<< j+1<<“ => “;
             cout<<“[Total Distance : “<<d[i][j]<<“ ( Path : “;
             //cout<<“Hello1”<<“\n”;
            int k=j;
            int l=0;
            int a[V];
            a[l++] = j+1;
             while(p[i][k]!=i+1)
            {
                 //cout<<“Hello1”<<“\n”;
                 a[l++]=p[i][k];
                k=p[i][k]-1;
            }
            a[l]=i+1;
             //cout<<“Hello1”<<“\n”;
             for(int r =l;r>0;r–)
            {
                 //cout<<“Hello1”<<“\n”;
              cout<<a[r]<<“ ––> “;
            }
             cout<<a[0]<<“ )”;
            cout<<endl;

            }
       }
   }
}
 //Floyd Warshall Algorithm
void floydWarshall ()
{
   int dist[V][V], i, j, k;
   int parent2[V][V];
   for (i = 0; i < V; i++)
       for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
```

```
   for(int  i=0;i<V;i++)
   {
        for(int j=0;j<V;j++)
       {
            parent2[i][j] = parent[i][j];
       }
   }

   for (k = 0; k < V; k++)
   {
       for (i = 0; i < V; i++)
       {
           for (j = 0; j < V; j++)
           {
                if (dist[i][k] + dist[k][j] < dist[i][j])
               {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    parent2[i][j] = parent2[k][j];
               }
           }
       }
   }

   printSolution(dist);
   cout<<"\n\n";
   printParents(parent2);
   cout<<"\n\n";
   cout<<"All pair shortest path is given below :"<<"\n";
   print_path(parent2, dist);
}
};
//Main  Function
int  main()
{
   Graph<int> g;
    g.matrix_form2();
```

```
  // g.make_parent();
 //Intializing the graph given in above picture
    g.addEdge(1,2,3);
    g.matrix_form(1,2,3);
    g.addEdge(1,3,8);
    g.matrix_form(1,3,8);
    g.addEdge(1,5,-4);
    g.matrix_form(1,5,-4);
    g.addEdge(2,4,1);
    g.matrix_form(2,4,1);
    g.addEdge(2,5,7);
    g.matrix_form(2,5,7);
    g.addEdge(3,2,4);
    g.matrix_form(3,2,4);
    g.addEdge(4,3,-5);
    g.matrix_form(4,3,-5);
    g.addEdge(4,1,2);
    g.matrix_form(4,1,2);
    g.addEdge(5,4,6);
    g.matrix_form(5,4,6);
    cout<<"Graph in the form of adjecency list representation : "<<"\n";
    g.printAdj();
    cout<<"Graph in the form of matrix representation : "<<"\n";
    g.print_matrix();
    cout<<"\n\n";
    g.floydWarshall();

}
```

18 Dynamic Programming

Dynamic Programming :

It is a ***technique*** of solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those sub-problems just once, and storing their solutions.

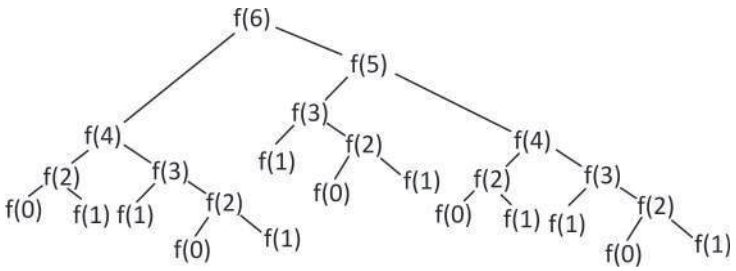
Dynamic Programming avoids repeated computations and thus is an optimization over standard solutions.

Dynamic Programming can be applied to any problem if that satisfies these two criteria.

1. Overlapping Subproblem :

One or more task is computed multiple times

Consider recursion tree of calls for fibonnaci(6)



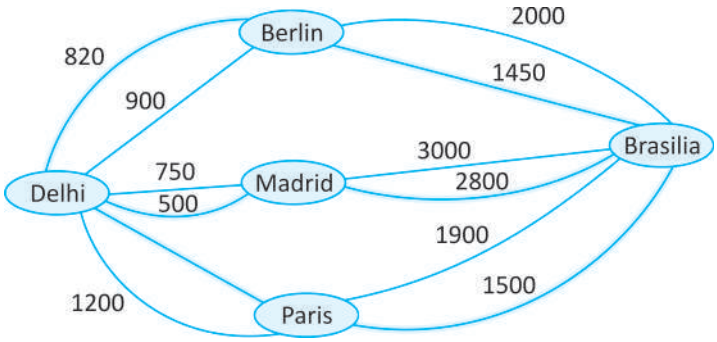
f(4) $\rightarrow$ 2 times

f(3) $\rightarrow$ 1 time independently, and 2times inside f(4)

2. Optimal Substructure :

The optimal solution of the smaller problem helps building the optimal solution to the larger problem.

That means, when building your solution for a problem of size n, you split the problem to smaller problems, one of them of size n'. Now, you need only to consider the optimal solution to n', and not all possible solutions to it, based on the optimal substructure property.



There exists multiple paths from one city to another. But to choose the optimal (smallest) route from Delhi to Brasilia, you consider only the optimal (smallest) paths from Delhi to intermediate cities and from intermediate cities to Brasilia.

Presence of alternate routes is not taken into consideration to find the optimal path.

To avoid recomputation of overlapping subproblems, either of two approachs can be used.

1. **Memoization (Top-Down) :** Calculate the solution for a problem and all its subproblem as and when it is required. The next time they are required just use the value. Here all the subproblems are not computed, only those that are required are computed.

   This is the top-down approach which is used in recursive problems.

2. **Tabulation (Bottom-up):** The solution of all smaller sub-problems is computed before hand and then the solution of actual problem is solved.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of **Fibonacci numbers.**

Fibonacci (n) = 1; if n = 0

Fibonacci (n) = 1; if n = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21, 35...

A code for it using pure recursion:

```
int fib (int n) {
if (n < 2)
return 1;
return fib(n-1) + fib(n-2);
}
```

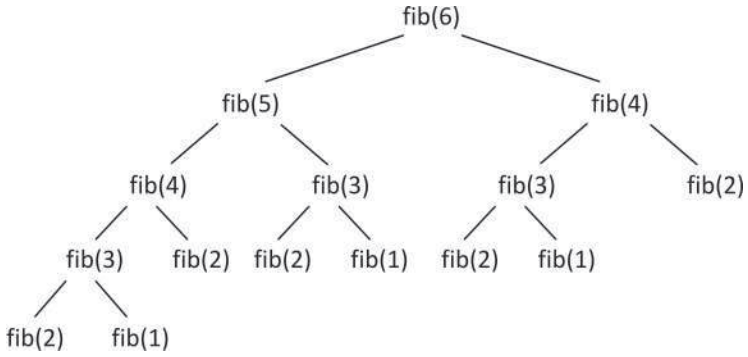
Using Dynamic Programming approach with memoization:

```
void fib () {
fib[0] = 1;
fib[1] = 1;
for (int i = 2; i<n; i++)
fib[i] = fib[i-1] + fib[i-2];
}
```

Q. Are we using a different recurrence relation in the two codes? **No**

Q. Are we doing anything different in the two codes? **Yes**

Let's Visualize :



Here we are running fibonacci function multiple times for the same value of n, which can be prevented using memoization.

Optimization Problems :

Dynamic Programming is typically applied to optimization problems. In such problems there can be any possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem.

❑ MINIMUM COIN CHANGE

Given a value N, if we want to make change for N cents, and we have infinite supply of each of C = { C1, C2, ... , CM} valued coins, what is the minimum number of coins to make the change?

Example **:**

Input: coins[] = {25, 10, 5}, N = 30

Output: Minimum 2 coins required

We can use one coin of 25 cents and one of 5 cents

Input: coins[] = {9, 6, 5, 1}, N = 13

Output: Minimum 3 coins required

We can use one coin of 6 + 6 + 1 cents coins.

Recursive Solution :

Start the solution with initially sum = N cents and at each iteration find the minimum coins required by dividing the problem in subproblems where we take {C1, C2, ..., CM} coin and decrease the sum N by C[i] (depending on the coin we took). Whenever N becomes 0, this means we have a possible solution. To find the optimal answer, we return the minimum of all answer for which N became 0.

If N == 0, then 0 coins required.

If N > 0

minCoins(N , coins[0..m-1]) = min {1 + minCoins(N-coin[i] ,coins[0....m-1])}

where i varies from 0 to m-1 and coin[i] <= N

Code :

```
int minCoins(int coins[], int m, int N)
{
    // base case
     if (N == 0)
       return 0;
    // Initialize result
    int res = INT_MAX;
    // Try every coin that has smaller value than V
    for (int i=0; i<m; i++)
     {
      if (coins[i] <= N)
        {
           int sub_res = 1 + minCoins(coins, m, N-coins[i]);
            // see if result can minimized
            if (sub_res < res)
                res = sub_res;
        }
     }
   return res;
}
```

Dynamic Programming Solution :

Since same suproblems are called again and again, this problem has Overlapping Subprolems property. Like other typical Dynamic Programming(DP) problems, re-computations of same subproblems can be avoided by constructing a temporary array dp[] and memoizing the computed values in this array.

1. Top Down DP

Code :

```
int minCoins(int N, int M)
{
   // if we have already solved this subproblem
   // return memoized result
   if(dp[N] != -1)
    return dp[N];
   // base case
   if (N == 0)
     return 0;
   // Initialize result
   int res = INF;
   // Try every coin that has smaller value than N
    for (int i=0; i<M; i++)
     {
       if (coins[i] <= N)
         {
            int sub_res = 1 + minCoins(N-coins[i], M);
            // see if result can minimized
            if (sub_res < res)
               res = sub_res;
         }
     }
   return dp[N] = res;
}
```

2. **Bottom Up DP :** ith state of dp : dp[i] : Minimum number of coins required to sum to i cents.

Code :

```
int minCoins(int N, int M)
{
  //Initializing all values to infinity i.e. minimum coins to make any
  //amount of sum is infinite
     for(int i = 0;i<=N;i++)
        dp[i] = INF;
 //Base case i.e. minimum coins to make sum = 0 cents is 0
     dp[0] = 0;
  //Iterating in the outer loop for possible values of sum between 1 to N
   //Since our final solution for sum = N might depend upon any of these
values
    for(int i = 1;i<=N;i++)
    {
        //Inner loop denotes the index of coin array.
        //For each value of sum, to obtain the optimal solution.
      for(int j = 0;j<M;j++)
        {
         //i -> sum
         //j -> next coin index
          //If we can include this coin in our solution
           if(coins[j] <= i)
             {
               //Solution might include the newly included coin.
               dp[i] = min(dp[i], 1 + dp[i - coins[j]]);
             }
        }
   }
 //for(int i = 1;i<=N;i++)cout<<i<<" "<<dp[i]<<endl;
 return dp[N];
}
T = O(N*M)
```

❑ LONGEST INCREASING SUBSEQUENCE

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 9, 3, 5, 4, 11, 7, 8} is 4 and LIS is {3, 4, 7, 8}.

Recurrence relation :

```
L(i) = 1 + max{ L(j) } where 0 < j < i and arr[j] < arr[i];
or
L(i) = 1, if no such j exists.
return max(L(i)) where 0 < i < n
```

Code :

```
int LIS(int n)
{
  int i,j,res = 0;
  /* Initialize LIS values for all indexes */
  for (i = 0; i < n; i++ )
   lis[i] = 1;
  /* Compute optimized LIS values in bottom up manner */
  for (i = 1; i < n; i++ )
   for (j = 0; j < i; j++ )
     if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
       lis[i] = lis[j] + 1;
  /* Pick maximum of all LIS values */
  for (i = 0; i < n; i++ )
   if (res < lis[i])
    res = lis[i];
 return res;
}
```

T = O(n^2)

We can also print the Longest Increasing Subsequence as:

```
void print_lis(int n)
{
//denotes the current LIS
//initially it is equal to the LIS of the whole sequence
 int cur_lis = LIS(n);
//denotes the previous element printed
```

```
//to print the LIS, previous element printed must always be larger than current
//element (if we are printing the LIS backwards)
//Initially set it to infinity
 int prev = INF;
 for(int i = n-1;i>=0;i–)
 {
  //find the element upto which the LIS equal to the cur_LIS
  //and that element is less than the previous one printed
  if(lis[i] == cur_lis && arr[i] <= prev)
  {
   cout<<arr[i]<<“ “;
   cur_lis–;
   prev = arr[i];
  }
  if(!cur_lis)
   break;
 }
return;
}
```

❑ LONGEST COMMON SUBSEQUENCE

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘“acefg”, ... etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

Example **:**

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4

Recurrence Relation :

LCS(str1, str2, m, n) = 0, if m = 0 or n = 0 //Base Case

LCS(str1, str2, m, n) = 1 + LCS(str1, str2, m-1, n-1), if str1[m] = str2[n]

LCS(str1, str2, m, n) = max{LCS(str1, str2, m-1, n), LCS(str1, str2, m, n-1)}, otherwise

LCS can take value between 0 and min(m,n).

Code :

```
int lcs( char *str1, char *str2, int m, int n )
{
//Following steps build L[m+1][n+1] in bottom up fashion. Note
//that L[i][j] contains length of LCS of str1[0..i-1] and str2[0..j-1]
for (int i=0; i<=m; i++)
 {
  for (int j=0; j<=n; j++)
   {
    if (i == 0 || j == 0)
     LCS[i][j] = 0;
    else if (str1[i-1] == str2[j-1])
      LCS[i][j] = LCS[i-1][j-1] + 1;
    else
       LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1]);
   }
 }
//Print LCS
 int i = m, j = n, index = LCS[m][n];
//array containing the final LCS
 char *lcs_arr = new char[index];
 while (i > 0 && j > 0)
  {
  // If current character in str1[] and str2[] are same, then
  // current character is part of LCS
   if (str1[i-1] == str2[j-1])
    {
    // Put current character in result
     lcs_arr[index-1] = str1[i-1];
    // reduce values of i, j and index
      i–;
      j–;
```

```
            index–;
        }
         // If not same, then find the larger of two and
        // go in the direction of larger value
        else if (LCS[i-1][j] > LCS[i][j-1])
          i–;
        else
          j–;
    }
    // Print the lcs
     cout << "LCS of " << str1 << " and " << str2 << " is "<< lcs_arr << endl;
    //L[m][n] contains length of LCS for str1[0..n-1] and
     str2[0..m-1]
     return LCS[m][n];
    }
```

T = O(mn)

- **EDIT DISTANCE**

Problem Statement: Given two strings str1 and str2 and below operations can be performed on str1. Find min number of edits(operations) required to convert str1 to str2.

- Insert
- Remove
- Replace

All the above operations are of equal cost.

http://www.spoj.com/problems/EDIST/

Example :

str1 = "cat"

str2 = "cut"

Replace 'a' with 'u', min number of edits = 1

str1 = "sunday"

str2 = "saturday"

Last 3 characters are same, we only need to replace "un" with "atur".

Replace n—>r and insert 'a' and 't' before 'u', min number of edits = 3

Recurrence Relation :

if str1[m] = str2[n]

editDist(str1, str2, m, n) = editDist(str1, str2, m-1, n-1)

else

editDist(str1, str2, m, n) = 1 + min{editDist(str1, str2, m-1, n) //Remove

editDist(str1, str2, m, n-1) //Insert

editDist(str1, str2, m-1, n-1) //Replace

}

Transform "Sunday" to "Saturday" :

Last 3 are same, so ignore. We will transform Sun —> Satur

(Sun, Satur) —> (Su, Satu) //Replace 'n' with 'r', cost= 1

(Su, Satu) —> (S, Sat) //Ignore 'u', cost = 0

(S, Sat) —> (S,Sa) //Insert 't', cost = 1

(S,Sa) —> (S,S) //Insert 'a', cost = 1

(S,S) —> ('','') //Ignore 'S', cost = 0

('', '') —> return 0

Dynamic Programming :

```
int editDist(string str1, string str2){
int m = str1.length();
int n = str2.length();
// dp[i][j] -> the minimum number of edits to transform str1[0...i-1] to
str2[0...j-1]
//Fill up the dp table in bottom up fashion
for(int i = 0;i<=m;i++)
{
for(int j = 0;j<=n;j++)
{
//If both strings are empty
 if(i == 0 && j == 0)
   dp[i][j] = 0;
//If first string is empty, only option is to
//insert all characters of second string
```

```
//So number of edits is the length of secondstring
else if(i == 0)
  dp[i][j] = j;
//If second string is empty, only option is to
//remove all characters of first string
//So number of edits is the length of first string
else if(j == 0)
  dp[i][j] = i;
//If the last character of the two strings are
//same, ignore this character and recur for the
//remaining string
else if(str1[i-1] == str2[j-1])
  dp[i][j] = dp[i-1][j-1];
//If last character is different, we need at least one
//edit to make them same. Consider all the possibilities
//and find minimum
else
  dp[i][j] = 1 + min(min(
             dp[i-1][j], //Remove
             dp[i][j-1]), //Insert
             dp[i-1][j-1] //Replace
              );
  }
}
//Return the most optimal solution
return dp[m][n];
}
```

- **0-1 KNAPSACK PROBLEM**

For each item you are given its weight and its value. You want to maximize the total value of all the items you are going to put in the knapsack such that the total weight of items is less than knapsack's capacity. What is this maximum total value?

http://www.spoj.com/problems/KNAPSACK/

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

1. Maximum value obtained by n-1 items and W weight (excludingnth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

Recurrence Relation :

```
//Base case
//If we have explored all the items all we have reached the maximum capacity
of Knapsack
if (n=0 or W=0)
 return 0
//If the weight of nth item is greater than the capacity of knapsack, we cannot
include //this item
if (wieght[n] > W)
 return solve(n-1, W)
otherwise
 return max{ solve(n-1, W), //We have not included the item
solve(n-1, W-weight[n]) //We have included the item in the knapsack
}
```

If we build the recursion tree for the above relation, we can clearly see that the **property of overlapping sub-problems** is satisfied. So, we will try to solve it using dynamic programming.

Let us define the dp solution with states i and j as
dp[i,j] —> max value that can be obtained with objects u
pto index i and knapsack capacity of j.

The most optimal solution to the problem will be **dp[N][W]** i.e. max value that can be obtained upto index N with max capacity of W.

Code :

```
int knapsack(int N, int W)
{
 for(int i = 0;i<=N;i++)
  {
   for(int j = 0;j<=W;j++)
    {
     //Base case
```

```
      //When no object is to be explored or our knapsack's capacity is 0
     if(i == 0 || j == 0)
       dp[i][j] = 0;
     //When the wieght of the item to be considered is more than the
     //knapsack's capacity, we will not include this item
      if(wt[i-1] > j)
        dp[i][j] = dp[i-1][j];
      else
        dp[1][j] = max(
                  //If we include this item, we get a value of val[i-1] but the
                  //capacity of the knapsack gets reduced by the weight of that
                   //item.
                    val[i-1] + dp[i-1][j - wt[i-1]],
                     //If we do not include this item, max value will be the
                 //solution obtained by taking objects upto index i-1, capacity
                     //of knapsack will remain unchanged.
                    dp[i-1][j]);
      }
   }
 return dp[N][W];
}
```

Time Complexity = O(NW)

Space Complexity = O(NW)

Can we do better?

If we observe carefully, we can see that the dp solution with states (i,j) will depend on state (i-1, j) or (i-1, j-wt[i-1]). In either case the solution for state (i,j) will lie in the **i-1**th row of the memoization table. So at every iteration of the index, we can copy the values of current row and use only this row for building the solution in next iteration and no other row will be used. Hence, at any iteration we will be using only a **single** row to build the solution for current row. Hence, we can reduce the space complexity to just **O(W).**

Space-Optimized DP Code :

```
int knapsack(int N, int W)
{
 for(int j = 0;j<=W;j++)
       dp[0][j] = 0;
```

```
for(int i = 0;i<=N;i++)
{
   for(int j = 0;j<=W;j++)
   {
   //Base case
   //When no object is to be explored or our knapsack's capacity is 0
    if(i == 0 || j == 0)
       dp[1][j] = 0;
   //When the wieght of the item to be considered is more than the
   //knapsack's capacity, we will not include this item
    if(wt[i-1] > j)
       dp[1][j] = dp[0][j];
    else
       dp[1][j] = max(
                //If we include this item, we get a value of val[i-1] but the
                //capacity of the knapsack gets reduced by the weight of that
               //item.
                  val[i-1] + dp[0][j - wt[i-1]],
                //If we do not include this item, max value will be the
                //solution obtained by taking objects upto index i-1, capacity
                //of knapsack will remain unchanged.
                dp[0][j]);
    }
        //Here we are copying value of current row into the previous row,
         //which will be used in building the solution for next iteration ofrow.
     for(int j = 0;j<=W;j++)
         dp[0][j] = dp[1][j];
   }
return  dp[1][W];
}
```

Time Complexity: O(N*W)

Space Complexity: O(W)

19 Tries

Tries :

A Trie, (also known as a prefix tree) is a special type of tree used to store associative data structures. A trie (pronounced try) gets its name from re***trie***val - its structure makes it a stellar matching algorithm.

If we need to search a *word of length m*, in a paragraph of *n lines*, if normal string searching is used then it would take *O*(*m*n*) time, but as the number of lines increases, this complexity becomes unbearably slow and gives a bad user experience.

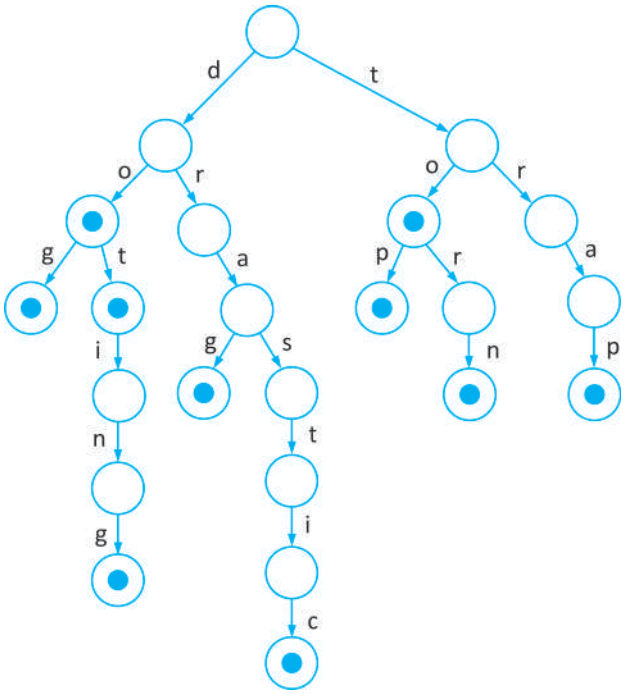
Think of using Microsoft Word, and searching a word in it. Would you like the software if it takes minutes to search?

This is where Trie comes in the picture, and is suitable data structure for applications like dictionary, word processing, etc.

General Implementation :

We need to make a class Node to represent each node. The class Node contains a Character data, a boolean is Terminal which determines if a word ends at this node or not and a HashMap of Character and Node, children.

Initially we add '\n' at the root node and all the words are its children. The node marked with red circle shows that the boolean isTerminal for them is true. Some of the words in the trie are dog, trap, drag, tap, etc.



Time Complexity for a Trie :

The formation of trie has a time complexity of O(n) where n is length of the paragraph but once the trie is created, the time for searching a word is only **O(m)** where m is the length of the word.

So searching a word has been reduced from O(n*m) to O(m). This makes finding a word very fast because generally the size of word is small and we get nearly **constant searching time**.

Following is the code in C++ for creating a class Trie

```
class Node {
public:
      bool isterminal;
      Node* characters[26];
      Node(){
            isterminal = false;
            fill(characters, characters + 26, (Node*)NULL);
      }
};
class Trie{
      Node * root;
public:
      Trie(){
            root = new Node();
      }
      int charToInt(char c){
            return c - 'a';
      }
      void addWord(char word[]){
            Node * cur = root;
            for(int i = 0; word[i] != '\0'; ++i){
                  int idx = charToInt(word[i]);
                  if (cur->characters[idx] == NULL){
                        //furthur levels do NOT exist
                        cur->characters[idx] = new Node();
                  }
                  cur = cur->characters[idx];
            }
            cur->isterminal = true;
      }
      bool searchWord(char word[]){
            Node * cur = root;
            for(int i = 0; word[i] != '\0'; ++i){
                  int idx = charToInt(word[i]);
                  if (!cur->characters[idx]) return false;
                  cur = cur->characters[idx];
            }
            return cur->isterminal;
      }
};
```