UNIT IV

NON-LINEAR DATA STRUCTURES

NON LINEAR DATA STRUCTURES

Non Linear Data Structures are those in which data elements are not accessed in sequential fashion.

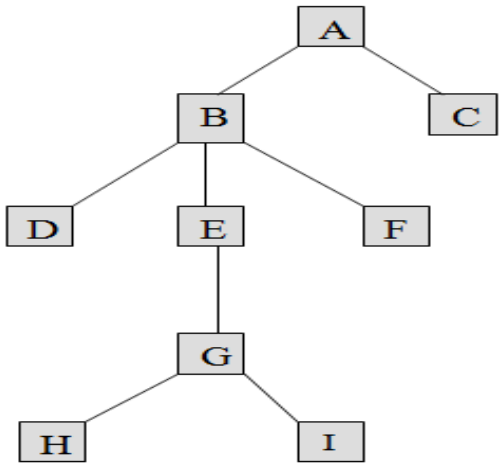
Eg: trees, graphs

TREES :

A tree is a Non-Linear Data Structure which consists of set of nodes called vertices and set of edges which links vertices

Terminology:

- **Root Node:** The starting node of a tree is called Root node of that tree
- **Terminal Nodes:**The node which has no children is said to be terminal node or leaf .
- **Non-Terminal Node:** The nodes which have children is said to be Non-Terminal Nodes
- **Degree:** The degree of a node is number of sub trees of that node
- **Depth:** The length of largest path from root to terminals is said to be depth or height of the tree
- **Siblings:** The children of same parent are said to be siblings
- **Ancestors:** The ancestors of a node are all the nodesalong the path from the root to the node



Property	Value
Number of nodes	:9
Height	:4
Root Node	:A
Leaves	:ED, H, I, F, C
Interior nodes	:D,E,G
Number of levels	:5
Ancestors of H	:I
Descendants of B	:D,E, F
Siblings of E	:D, F

Binary Trees:

Binary trees are special class of trees in which max degree for each node is 2

Recursive definition:

A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.Any tree can be transformed into binary tree.By left child-right sibling representation.

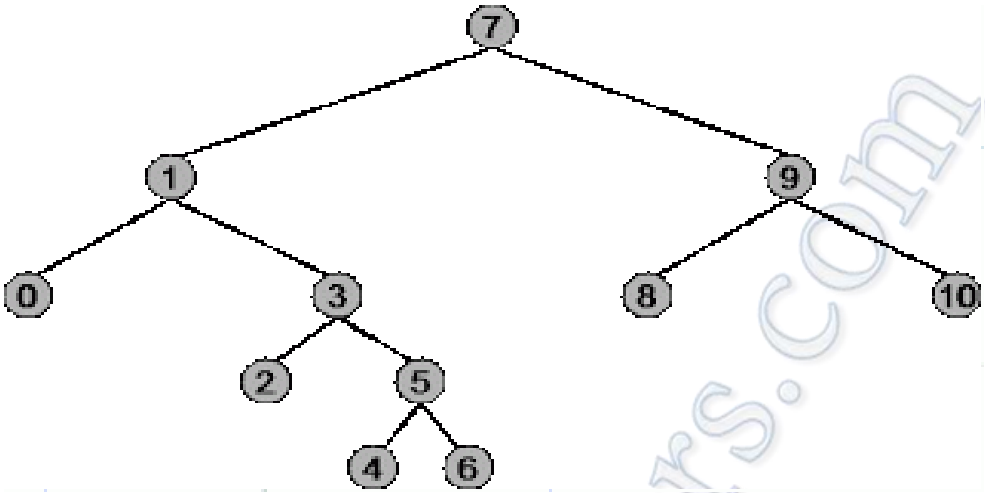
Binary Tree Traversal Techniques:

Preorder traversal: To traverse a binary tree in Preorder, following operations are carried-out

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree

Therefore, the Preorder traversal of the above tree will outputs:

7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10



Inorder traversal: To traverse a binary tree in Inorder, following operations are carried-out

1. Traverse the left most subtree starting at the left external node
2. Visit the root
3. Traverse the right subtree starting at the left external node.

Therefore, the Inorder traversal of the above tree will outputs:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Postorder traversal: To traverse a binary tree in Postorder, following operations are carried-out

1. Traverse all the left external nodes starting with the left most subtree which is then followed by bubble-up all the internal nodes
2. Traverse the right subtree starting at the left external node which is then followed by bubble-up all the internal nodes
3. Visit the root.

Therefore, the Postorder traversal of the above tree will outputs:

0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

Binary Search Tree:

Binary search tree (BST), sometimes also called an ordered or sorted binary tree, is a node-based binary tree with the following properties:

1. Each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree.
2. There must be no duplicate nodes.

3. A unique path exists from the root to every other node.
4. The left sub-tree contains only nodes with keys less than the parent node;
5. The right sub-tree contains only nodes with keys greater than the parent node.
6. The left and right subtree each must also be a binary search tree.

BSTs are also dynamic data structures, and the size of a BST is only limited by the amount of free memory in the operating system. The main advantage of binary search trees is that it remains ordered, which provides quicker search times than many other data structures.

Binary Node class:

```
classBinaryNode
{
int element;
BinaryNode *left;
BinaryNode *right;
BinaryNode(constint&theelement, BnaryNode *lt, BinaryNode *rt)
{
element=theElement;
left=lt;
right=rt;element
}
};
```

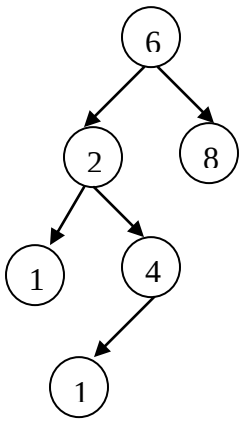
Valid Operations on Binary Search Tree:

- Inserting an element
- Deleting an element
- Searching for an element
- Traversing

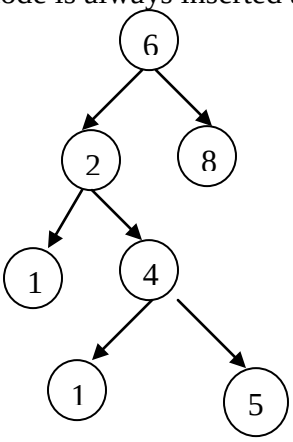
Inserting a node

A naïve algorithm for inserting a node into a BST is that,

1. We start from the root node.
2. If the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root.
3. We continue this process until we find a null pointer (or leaf node) where we cannot go any further.
4. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. Note that a new node is always inserted as a leaf node.



3

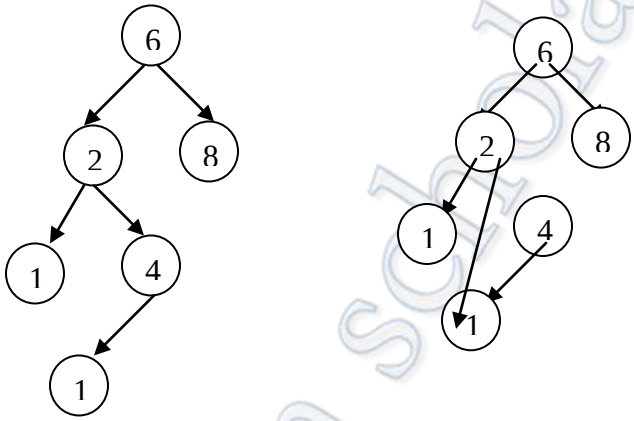


Above diagram shows inserting an element in binary search tree. First one befor inserting 5 second one after inserting 5.

```
intsert(int&x, BinaryNode&t)
{
if(t==NULL)
{
t=new BinaryNode(x,NULL,NULL);
elseif(x<t→element)
insert(x, t→left);
elseif(x>t→element)
insert(x,t→right;
else
{
//do nothing. Duplicate element
}
```

Deletion

Deleting a node with one child:

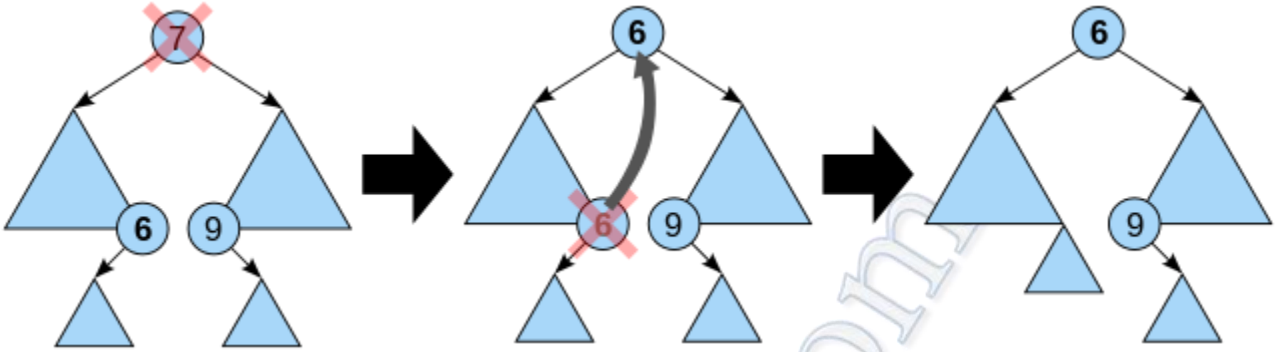


Before delete After deleting 4

There are three possible cases to consider:

- **Deleting a leaf (node with no children):** Deleting a leaf is easy, as we can simply remove it from the tree.
- **Deleting a node with one child:** Remove the node and replace it with its child.
- **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*. Copy the value of *R* to *N*, then recursively call delete on *R* until reaching one of the first two cases.

Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



FindMin()

```
findMin(BinaryNode *t)
{
        if(t==NULL)
        return NULL;
        if(t→left=NULL)
                return t;
        returnfindMin(t→left);
}
```

FindMax()

```
findMax(BinaryNode n*t)
{
        if(t!=NULL)
        while(t→right!=NULL)
                t=t→right;
        return t;
}
```

Deleting a node with two children from a binary search tree

First the rightmost node in the left subtree, the inorder predecessor **6**, is identified. Its value is copied into the node being deleted. The inorder predecessor can then be easily deleted because it has at most one child. The same method works symmetrically using the inorder successor labelled **9**.Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an unbalanced tree, so some implementations select one or the other at different times.

```
remove(constint&x, BinaryNode *&t)
{
```

```
        if(t==NULL)
        return;
        if(x<t→element)
        remove(x, t→left);
        if(x>t→element)
        remove(x, t→right)
        elseif(t→left!= NULL && t→right!=NULL)//two children
        {
        t→elemnt = findMin(t→right) →elemnt;
        remove(x,t→right);
        }
        else
        {
        BinaryNode *oldNode=t;
        t=(t→left!=NULL)?t→left:t→right;
        deleteoldNode;
        }
}
```

Searching

Searching a binary search tree for a specific key can be a recursive or an iterative process.

1. We begin by examining the root node.
2. If the tree is *null*, the key we are searching for does not exist in the tree.
3. Otherwise, if the key equals that of the root, the search is successful and we return the node.
4. If the key is less than that of the root, we search the left subtree.
5. Similarly, if the key is greater than that of the root, we search the right subtree.
6. This process is repeated until the key is found or the remaining subtree is *null*.
7. If the searched key is not found before a *null* subtree is reached, then the item must not be present in the tree.

This is easily expressed as a recursive algorithm:

```
function Find-recursive(key, node):  // call initially with node = root
if node = Null or node.key = key then
return node
else if key <node.key then
return Find-recursive(key, node.left)
else
return Find-recursive(key, node.right)
```

GRAPH THEORY

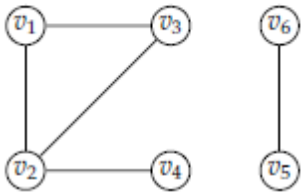
Graph:

Graph is a diagram consisting of points, called vertices, joined together by lines, called edges. Each edge joins exactly two vertices.

Definition of Graph

A graph G = (V, E) consists of a (finite) set denoted by V or V(G) and a collection E, or E(G), of unordered pairs {u, v} of distinct elements from V. Each element of V is called a vertex or a point or a node, and each element of E is called an edge or a line or a link.

The figure below is a geometric representation of the graph *G* with *VG* = {*v*1, *v*2, *v*3, *v*4, *v*5, *v*6} and *EG* = {*v*1*v*2, *v*1*v*3, *v*2*v*3, *v*2*v*4, *v*5*v*6}.



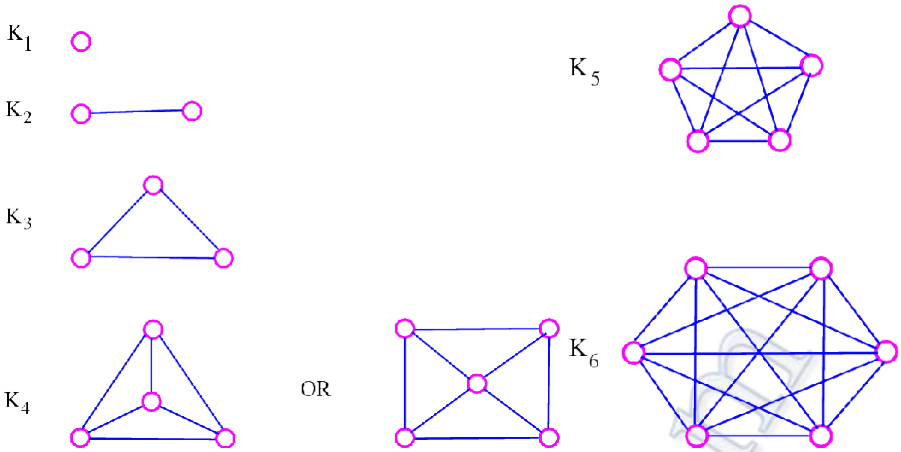
Node: A node, usually drawn as a circle, represents an item that can be related to other items or nodes. Nodes are sometimes referred to as vertices.

Edge: An edge represents a connection between nodes. Edges can be either directed or undirected, depending on the type of graph. Edges can also have weights, which may correspond to strength of relationship or distance between edges.

Directed Graph: A directed graph is one in which edges connect nodes in only one direction.
Undirected Graph: An undirected graph is one in which edges connect nodes bidirectionally (in both directions).
Connected: A graph is connected if from any node you can reach any other node.
Disconnected: A graph is disconnected if certain groups of nodes form an island that has no connection to the rest of the graph.

Complete Graphs

A computer graph is a graph in which every two distinct vertices are joined by exactly one edge. The complete graph with *n* vertices is denoted by K_n.The following are the examples of complete graphs.



The graph $K_n$is regular of degree n-1, and therefore has $n(n-1)/2$ edges, by consequence 3 of the handshaking lemma.

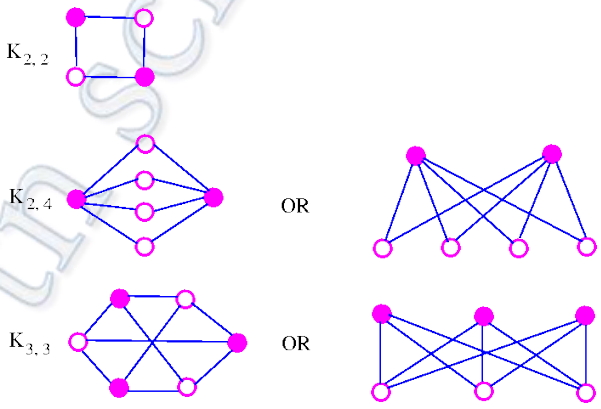
Bipartite Graphs

A bipartite graph is a graph whose vertex-set can be split into two sets in such a way that each edge of the graph joins a vertex in first set to a vertex in second set.

Complete Bipartite Graph

A complete bipartite graph is a bipartite graph in which each vertex in the first set is joined to each vertex in the second set by exactly one edge. The complete bipartite graph with r vertices and 3 vertices is denoted by $K_{r,s}$.

The following are some examples.

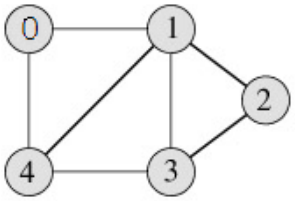


Note that $K_{r,s}$ has r+s vertices (r vertices of degrees, and s vertices of degree r), and r*s edges. Note also that $K_{r,s}= K_{s,r}$.

An Important Note: A complete bipartite graph of the form $K_{r,s}$ is called a star graph.

Graph Representation:

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.
1. Adjacency Matrix
2. Adjacency List
There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

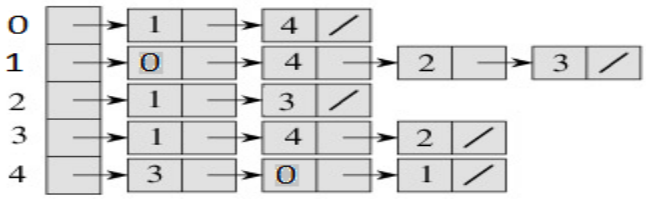
The adjacency matrix for the above example graph is:

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

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List:
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Graph Traversal

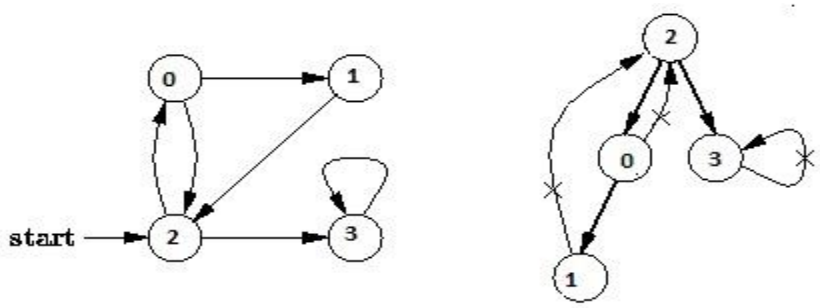
To traverse means to visit the vertices in some systematic order. There are two Graph Traversing techniques: breadth first search (BFS) and depth first search (DFS). Both of these construct spanning trees with certain properties useful in other graph algorithms. We'll start by describing them in undirected graphs, but they are both also very useful for directed graphs.

Depth-first search

A depth-first search (DFS) is an algorithm for traversing a finite graph. DFS visits the child nodes before visiting the sibling nodes; that is, it traverses the depth of any particular path before exploring its breadth. A stack is generally used when implementing the algorithm.

The algorithm begins with a chosen "root" node; it then iteratively transitions from the current node to an adjacent, unvisited node, until it can no longer find an unexplored node to transition to from its current location. The algorithm then backtracks along previously visited nodes, until it finds a node connected to yet more uncharted territory. It will then proceed down the new path as it had before, backtracking as it encounters dead-ends, and ending only when the algorithm has backtracked past the original "root" node from the very first step.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Depth First Traversal of the following graph is 2, 0, 1, 3



Pseudo code

Input: A graph *G* and a vertex *v* of G

Output: A labeling of the edges in the connected component of v as discovery edges and back edges

```
1  procedure DFS(G,v):
2      label v as explored
```

```
3     for all edges e in G.adjacentEdges(v) do
4        if edge e is unexplored then
5           w ← G.adjacentVertex(v,e)
6           if vertex w is unexplored then
7              label e as a discovery edge
8              recursively call DFS(G,w)
9           else
10             label e as a back edge
```

Applications

Following are the problems that use DFS as a bulding block.
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 edges. (See this for details)
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
See this for details.
4) Topological Sorting
See this for details.
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! See this for details.
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. (See this for DFS based algo for finding Strongly Connected Components)

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.)

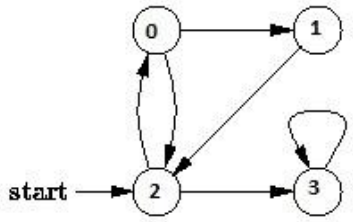
Breadth-first search

BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic algorithm.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not

yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Breadth First Traversal of the following graph is 2, 0, 3, 1.



Algorithm

1. Enqueue the root node.
2. Dequeue a node and examine it.
   - If the element sought is found in this node, quit the search and return a result.
   - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. Repeat from Step 2.

Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.

Pseudocode

Input: A graph *G* and a root *v* of G
Output: The node closest to v in G satisfying some conditions or null if no such a node exists in G

```
1  procedure BFS(G,v):
2      create a queue Q
3      enqueuev onto Q
4      mark v
5      whileQ is not empty:
6          t ← Q.dequeue()
7          ift is what we are looking for:
8              return t
9          for all edges e in G.adjacentEdges(t) do
12             o ← G.adjacentVertex(t,e)
13             if o is not marked:
14                 mark o
15                 enqueueo onto Q
16     return null
```

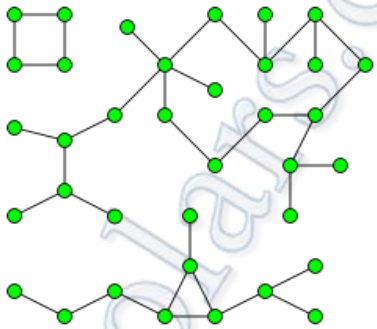
Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

- Finding all nodes within one connected component
- Finding the shortest path between two nodes *u* and *v* (with path length measured by number of edges)
- Testing a graph for bipartitions
- Ford–Fulkerson method for computing the maximum flow in a flow network
- The flood fill algorithm for marking contiguous regions of a two dimensional image or n-dimensional array
- The analysis of networks and relationships

CONNECTED COMPONENT

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. For example, the graph shown below has three connected components. A graph that is itself connected has exactly one connected component, consisting of the whole graph.



An equivalence relation

An alternative way to define connected components involves the equivalence classes of an equivalence relation that is defined on the vertices of the graph. In an undirected graph, a vertex v is reachable from a vertex u if there is a path from u to v. In this definition, a single vertex is counted as a path of length zero, and the same vertex may occur more than once within a path. Reachability is an equivalence relation, since:

1. It is ***reflexive***: There is a trivial path of length zero from any vertex to itself.
2. It is ***symmetric***: If there is a path from u to v, the same edges form a path from v to u.
3. It is ***transitive***: If there is a path from u to v and a path from v to w, the two paths may be concatenated together to form a path from u to w.

The number of connected components

The number of connected components is an important topological invariant of a graph. In topological graph theory it can be interpreted as the zerothBetti number of the graph. In algebraic graph theory it equals the multiplicity of 0 as an eigenvalue of the Laplacian matrix of the graph. It is also the index of the first nonzero coefficient of the chromatic polynomial of a graph.

Numbers of connected components play a key role in the Tutte theorem characterizing graphs that have perfect matchings, and in the definition of graph toughness.

Algorithms

It is straightforward to compute the connected components of a graph in linear time (in terms of the numbers of the vertices and edges of the graph) using either breadth-first search or depth-first search. In either case, a search that begins at some particular vertex v will find the entire connected component containing v (and no more) before returning. To find all the connected components of a graph, loop through its vertices, starting a new breadth first or depth first search whenever the loop reaches a vertex that has not already been included in a previously found connected component. Hopcroft and Tarjan describe essentially this algorithm, and state that at that point it was "well known".

SETS

In computer science, a **set** is an abstract data structure that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set.

Some set data structures are designed for **static sets** that do not change with time, and allow only query operations such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and/or deletion of elements from the set.

Operations

Typical operations that may be provided by a static set structure *S* are

1. element_of(*x*,*S*): checks whether the value *x* is in the set *S*.
2. empty(*S*): checks whether the set *S* is empty.
3. size(*S*): returns the number of elements in *S*.
4. enumerate(*S*): yields the elements of *S* in some arbitrary order.
5. pick(*S*): returns an arbitrary element of *S*.
6. build(*x*1,*x*2,…,*xn*,): creates a set structure with values *x*1,*x*2,…,*xn*.

The enumerate operation may return a list of all the elements, or an iterator, a procedure object that returns one more value of *S* at each call.

Dynamic set structures typically add:

1. create(*n*): creates a new set structure, initially empty but capable of holding up to *n* elements.
2. add(*S*,*x*): adds the element *x* to *S*, if it is not there already.
3. delete(*S*,*x*): removes the element *x* from *S*, if it is there.
4. capacity(*S*): returns the maximum number of values that *S* can hold.

Some set structures may allow only some of these operations. The cost of each operation will depend on the implementation, and possibly also on the particular values stored in the set, and the order in which they are inserted.

There are many other operations that can (in principle) be defined in terms of the above, such as:

• pop(*S*): returns an arbitrary element of *S*, deleting it from *S*.
• find(*S*,*P*): returns an element of *S* that satisfies a given predicate *P*.

• clear(*S*): delete all elements of *S*.
• union(*S*,*T*): returns the union of sets *S* and *T*.
• intersection(*S*,*T*): returns the intersection of sets *S* and *T*.
• difference(*S*,*T*): returns the difference of sets *S* and *T*.
• subset(*S*,*T*): a predicate that tests whether the set *S* is a subset of set *T*.

Equivalence RFelation:

A relation R is defined on a set if for every pair of elements (a,b). a,b € S, aRb is either true or false. If a RB is true, then we say that a is related to b. An equivalence relation is a relation that satisfies the following three properties:

1. Reflexive: aRa, for all a € S.
2. Symmetric aRb if and only if bRa.
3. Transitive aRb and bRc implies that aRc.

Example:

≥, ≤ is not a equivalence relation because it is not symmetric (a≤bdosen't implies b≤a).

The equivalence class of an element a € S is the subset of S that contains all the elements related to a.

Deciding Equivalence relation on a set (union/find algorithm):

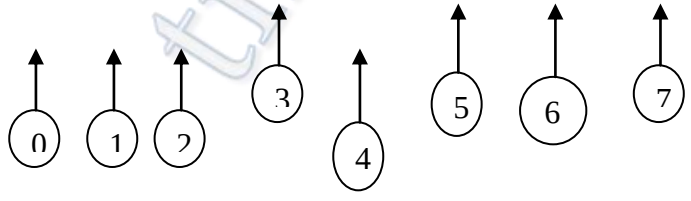
Input: Collection of N sets, each with one element. The initial representation is that all relations except reflexive is false. Each set has different elements, so that $S_i \cap S_j = \emptyset$. This makes the sets disjoint. There are two operations:

1. Find() : Returns the name of the set (i.e. the equivalence class)
2. union() : Adds relations

If we want to add the relation a~b, then we first see if a and b already related. This is done by performing finds on both a and b and then checking whether they are in the same equivalence class. If they are not then we apply union operation which merges two equivalences classes containing a and b into new equivalence class. The algorithm to do this is called as union/find algorithm for this reason.

Find()

Find() returns the name of the set (i.e. the equivalence class), that finds on two elements return the same answer if and only if they are in the same set. We will represent each set by a tree. The name of the tree is given at the root node. Since we only need the root of the parent, we can assume that the tree is stored implicitly in an array; each entry s[i] in the array represents the parent of element i. If i is a root then s[i]=-1. I n the following figure, s[i]—1 for 0≤i<8.

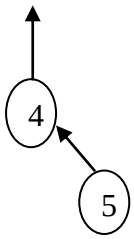


The implicit representation of above tree is :

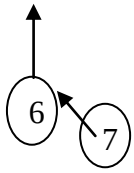
-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7

Union() Two perform union of two sets, we merge the two trees by making the parent link of one tree root link to the root node of other tree. The following figures represent union operation.

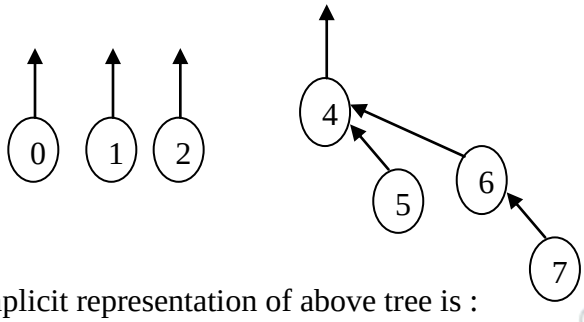
union(4,5):



Union(6,7)



Union(4,6)



The implicit representation of above tree is :

-1	-1	-1	-1	-1	4	4	6
0	1	2	3	4	5	6	7

The following code represents an implementation of the basic algorithm.

```
classDisjSets{
public:
        explicitDijSets(intnumElements);
        int find(int x) const;
        int find(int x);
private:
        vector<int> s;
};
```

Disjoint set initialization routine:

```
//here numElements is the number of initial disjoint sets.
DishSets: :DisjSets(intnemElemets) : s(numElements)
{
for(inti=0; i<s.size;i++)
s[i]—1;
}
```

Union :

```
DisjointSets::unionSets(int root1, int root2)
{
s[root2]=root1;
}
```

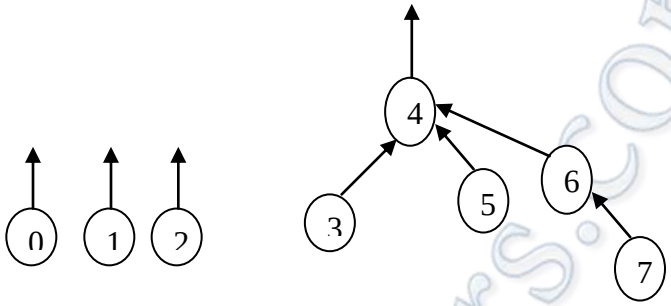
Find:

```
DisjSets::Find(int x)const
```

```
{
if(s[x]<0)
return x;
else
return find(s[x]);
}
```

Smart union algorithm:

The union above were performed arbitrarily, by making the second tree a sub tree of smaller tree. A simple improvement is always to make the smaller tree a sub tree of larger, breaking tie by any method; this approach is called as **union-by-size.** The three unions below are all ties, so we can consider that they were performed by size. If the next operation were union(3,4), then the below figure would form.



The depth of any node is never more than log N. For this a node is initially at depth 0. When its depth is increases as a result of union , it is placed in a tree that at least twice as large as before. Thus the depth can be increased at most log N times.

An alternative implementation, which guarantees that all these trees will have depth at most log N, is **union-by-height**. We keep track of height instead of size, of each tree and perform union by making shallow tree a sub tree of the deerper tree. This is an easy algorithm, since the height of a tree increases only when two equally deeper trees are joined (and then the height goes up by one).

Implicit representation of union-by-size:

-1	-1	-1	4	-5	4	4	6
0	1	2	3	4	5	6	7

Here –5 represents size of the 4th sub tree.

Implicit representation of union-by-height:

-1	-1	-1	4	-3	4	4	6
0	1	2	3	4	5	6	7

Here –3 represents height of the 4th sub tree.

TEXT BOOKS:

1. Deitel and Deitel, “C++, How To Program”, Fifth Edition, Pearson Education, 2005.
2. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C++”, Third Edition, Addison-Wesley, 2007.

REFERENCES:

1. Bhushan Trivedi, “Programming with ANSI C++, A Step-By-Step approach”, Oxford University Press, 2010.

2. Goodrich, Michael T., Roberto Tamassia, David Mount, “Data Structures and Algorithms in C++”, 7th Edition, Wiley. 2004.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", Second Edition, Mc Graw Hill, 2002.
4. Bjarne Stroustrup, “The C++ Programming Language”, 3rd Edition, Pearson Education, 2007.
5. Ellis Horowitz, Sartaj Sahni and Dinesh Mehta, “Fundamentals of Data Structures in C++”, Galgotia Publications, 2007.