**Unit 4: Trees and Graphs (8 Hrs)**

**8.1 Concept and Definitions, Basic Operations in Tree, Tree Height, Level and Depth**

**8.2 Binary Search Tree, Insertion, Deletion, Traversals, Search in BST**

**8.3 AVL tree and Balancing algorithm, Applications of Trees**

**8.4 Definition and Representation of Graphs, Graph Traversal, Minimum Spanning Trees: Kruskal and Prim's Algorithm**

**8.5 Shortest Path Algorithms: Dijksrtra Algorithm**

**Trees**

Linked list usually provide greater flexibility than array, but they are linear structures and it is difficult to use them to organize a hierarchical representation of objects. To overcome these limitations, we create a new data type called a tree that consists of nodes and arcs. A tree can be defined recursively as the following:

1. An empty structure is an empty tree.
2. If t1,……tk are disjointed trees, then the structure whose root has its children the roots of t1,,……… tk is also a tree.
3. Only structures generated by rules 1 and 2 are trees.

**Key Terminologies**

**Root:**  A tree contains a unique first node which is shown at the top of the tree structure. This node is called root of the tree.

**Leaf Nodes**: The nodes which do not have children are called leaf nodes.

**Interior Nodes:** The nodes which have children nodes are called interior nodes.

**Siblings:** If two or more nodes have same parent, then these nodes are called siblings to each other.

**Ancestor:** A node is called ancestor of another node if either it is the parent of that node or it is the parent of some other ancestor of that node.

**Descendents:** A node is called Descendents of another node if it is the child of that node or the child of some other descendents of that node.

**Depth of tree:** The length of the longest path from root to any other node is known as the depth of the tree. Path is the number of edges from root to any node.

**Binary Tree**

A binary tree is type of tree with finite number of elements and is divided into three main parts. The first part is called root of the tree and other two parts are itself binary tree which exists towards left and right of the tree. Each of the elements in the binary tree is considered a node and a node will have three piece of information: data, two references

**Some information of binary tree**

* The binary tree starts at root and grows downward.
* The topmost node is called the root. The root will not have parents. All other nodes can be reached from it by following edges or links.
* The link between two nodes is termed as edge or arc.
* If X is the root of the tree and L and R are its left child/ and right child respectively then X is the father of both L and R. L and R are called siblings or brother
* A node with no left or right child is termed as leaf node. It is also called terminal node.
* An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.
* A path is sequence of edges from some node to another node with more than one edge.
* A path ending in a leaf is known as branch.
* Going from root of the tree to any of the leaf node is termed as descending the tree.
* Going from any leaf of the tree towards root of the tree is termed as climbing the tree.
* The root has level 0 and level of any other node is one more than the level of its parents.
* The height of the binary tree is one more than the number of levels. Another definition of height is the number of nodes in a branch of tree.
* The depth of the binary tree is the maximum level number.
* Number of sons for a node is considered as degree of that node.
* Two binary trees are said to be copies when they have the identical structures and identical nodes at all levels of tree.
* Two binary trees are said to be similar when they have the identical structures only.

**Advantages of Binary Tree**

Searching in Binary Tree becomes faster

Maximum and minimum element can be directly picked up.

It is used for graph traversal

It is used to convert infix expression to postfix and prefix expression

**Types of Binary Tree**

**Strictly Binary Tree**

It is a binary tree with non-empty right and left sub trees. In other words, it is binary tree with every node N has either 0 or 2 tree. The strictly binary tree is also known as extended 2 Tree or simply 2 –tree. Sometimes nodes with 2 children's are known as internal nodes and nodes with 0 children are known as external nodes.

**Complete Binary Tree**

It is a special type of strictly binary tree where all the leaves of the tree reside at the same level. Using the depth we always say a complete binary tree of depth d where all leaves with be at level d

**Properties of Complete Binary Tree**

1. A binary tree of height h with no missing node
2. All leaves are at height h and all other nodes have no children
3. All the nodes that are at a level less than h have two children

**Almost Complete Binary Tree**

A binary tree of depth d is an almost complete binary tree if

* Each leaf in the tree is either at level d or at level d-1
* For any node "nd" in the tree with right child descendant at level d, all the left descendants of nd that are leaves are also at level d. It means nodes should be present in left to right at any level: there should be any missing nodes in the traversal

**Binary Search Tree**

The application of binary tree is searching and sorting. By enforcing certain rules on the values of the elements stored in a binary tree, it could be used to search and sort.

Binary search tree is a tree in which value of each node in the tree is greater than the value of node in its left (if exists) and it is less than the value in its right child (if it exists).

As name suggests, binary search tree (BST) is used for searching purpose. For every node N in the BST, the following property will be true.

* The data at left node will be smaller than data at node N
* The data at right node will be larger than data at node N

**Figure: Binary Search Tree**

**Implementing Binary Trees**

Binary trees can be implemented in at least two ways: as arrays and as linked structures. To implement a tree as an array, a node is declared as an object with information filed two “reference” fields.

**Array Implementation of Binary Tree**

The sequential representation of binary tree uses array for storing the data for each node. This is very simple. If any parent node is stored at index I then its left child will be stored at 2\*I+1 and right child will be stored at 2\*I+2. The root of the tree is stored at first index of the array (index 0).

The array representation of above BST is as shown below:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 12 | 9 | 18 | 1 |  | 14 | 21 |  |  |  |

It is clear that the most of the locations in array are empty. This causes wastage of memory space. That is, the array representation of binary tree is quite inefficient. In general for a binary tree with height H, the size of the array will be approximately be 2H.

Besides this, locations of children must be known to insert a new node and these locations may need to be located sequentially. After deleting a node from a tree, a hole in the array would have to be eliminated; this may lead to populating the array with many unused cells.

**Linked List Representation of Binary Tree**

The linked list representation is the most popular, efficient and most frequently used representation of binary tree. In the linked list representation every node is represented by data, a reference to left child and a reference to right child. The node of binary tree can be created as follows:

The structure of node of binary tree can be defined in as follows:

struct BNode

{

int info;

struct BNode \*left;

struct BNode \*right;

}

**Operations on Binary Tree**

There are a lot of operations that can be performed in binary tree. Some main operations are as follows:

* Searching a node
* Inserting a node
* Traversing the tree
* Deleting a node

**Binary Tree Traversal**

Traversing a tree means visiting each and every node in a specified order. This process is not commonly not used as finding, inserting and deleting nodes. One reason for this is that traversal is not particularly fast. But traversing a tree is useful in some circumstances and the algorithm in interesting. The tree traversal is a way in which each node in the tree is visited exactly once in some systematic order. There are three popular methods of three traversal.

* Pre-order Traversal
* In-order Traversal
* Post-order Traversal

**Pre-order Traversal**

In pre-order traversing first root node is visited followed by left sub-tree and right sub-tree.

It is define as follows:

* Visit the root
* Traverse the left sub-tree in pre-order
* Traverse the right sub-tree in pre-order

**The pre-order traversal of above tree is:**

**C function for preorder traversal**

void preorder(BSTNode \*r)

{

if(r!=NULL)

{

printf("%d ",r->data);

preorder(r->left);

preorder(r->right);

}

}

**In-order Traversal**

The in-order traversal of a nonempty binary tree is defined as follows:

* Traverse the left sub-tree in order
* Visit the root
* Traverse the right-sub-tree in in-order

The in-order traversal of the given tree is:

**The C function for in-order traversal is**

void inorder(BSTNode \*r)

{

if(r!=NULL)

{

inorder(r->left);

printf("%d ",r->data);

inorder(r->right);

}

}

**Post-order Traversal**

The post-order traversal of binary tree is defined as:

* Traverse the left-sub-tree in post-order
* Traverse the right sub-tree in post-order
* Visit the root

The post order traversal of above tree is:

**The C function for post order traversal is:**

void postorder(BSTNode \*r)

{

if(r!=NULL)

{

postorder(r->left);

postorder(r->right);

printf("%d ",r->data);

}

}

**Binary Search Tree**

A binary search tree(BST) is a binary tree that is either empty or in which every node contains a key and satisfies following conditions.

* All keys in the left sub-tree of the root are smaller than the key in the root node.
* All keys in the right sub-tree of the root are greater than the key in the root node.
* The left and right sub-tree of the root are again binary search trees

**Construct a BST from the following sequence of numbers: 14, 15, 4, 9,7,18,3,5,16,,4,20,17,9**

**Operations on Binary Search Tree (BST)**

Following operations can be performed on BST

* **Search (k,T)** : Search for key k in the tree T. If k is found in some node of the tree then return true otherwise return false.
* **Insert(k,T):** Insert a new node with value k in the info field in the tree T such that the property of tree is maintained.
* **Delete(k,T)** : Delete a node with value k in the info field from the tree T such that the property of BST is maintained
* **findMin(T)** : Finds the minimum element in tree T
* **findMax(T)**: Finds the maximum element in the tree T

**Searching a Binary Search Tree**

An algorithm for locating an element in binary search tree is quite straightforward. For every node, compare the key to be located with the value stored in the node root. If the key is equal to value then stop. If the key is less than the value, go to the left subtree and try again. If key is greater than that value, try the right subtree. If it is same, obviously the search can be discontinued.

The algorithm for searching a node in BST is as follows:

Step 1: start

Step 2: Set curr = root;

Step 3: while curr !=null

Step 3: If curr->data == key

Then return true;

Step 4 otherwise if key> curr->data

Curr = curr->right

Step 5: Else

Curr = curr->left;

Step 6: End While

Step 7: stop

The C function for searching binary search tree is:

int search(int key)

{

struct BNode \*temp = root;

int b=0;

while(temp!=NULL)

{

if(temp->data==key)

{

b = 1;

break;

}

else if(key<temp->data)

temp = temp->left;

else

temp = temp->right;

}

return b;

}

**Tree Traversal**

Tree traversal is the process of visiting each node in the tree exactly once. Traversal may be interpreted as putting all nodes on one line or linearizing a tree.

The definition of traversal specifies only one condition-visiting each node only one time but it does not specify the order in which the nodes are visited. For a tree with n nodes, there are n! different traversals. Most of the traversal has no use, so we restrict our attention two classes only, namely, breadth first search and depth first search traversal.

**Preorder Traversal**

To traverse a nonempty binary tree in **preorder**, we perform the following three operations:

* Visit the root
* Traverse the left tree in preorder
* Traverse the right sub tree in preorder

**The C function for preorder traversal is:**

void preorder(struct BNode \*r)

{

if(r!=NULL)

{

printf("%d\t",r->data);

preorder(r->left);

preorder(r->right);

}

}

**Inorder Traversal**

To traverse a nonempty binary tree in **inorder**, we perform the following three operations:

* Traverse the left sub tree in inorder
* Visit the root
* Traverse the right sub tree inorder

**The C function for inorder traversal is:**

void inorder(struct BNode \*r)

{

if(r!=NULL)

{

inorder(r->left);

printf("%d\t",r->data);

inorder(r->right);

}

}

**Postorder Traversal**

To traverse a non-empty tree in **Postorder**, we perform the following

* Traverse the left sub tree in Postorder
* Traverse the right sub tree in Postorder
* Visit the root

The C function for postorder Traversal is:

void postorder(struct BNode \*r)

{

if(r!=NULL)

{

postorder(r->left);

postorder(r->right);

printf("%d\t",r->data);

}

}

**The preorder traversal will print: 13 8 7 9 25 20 31 29**

**The inorder traversal will print: 7 8 9 13 20 25 29 31**

**The Postorder traversal will print: 7 9 8 20 29 31 25 30**

**Insertion**

Searching a binary tree does not modify the tree. It scans the tree in a predetermined way to access some or all of the keys in the tree but the tree itself remains undisturbed. There are certain operations that always make some change on the tree, such as adding nodes, deleting nodes, and modifying elements, merging trees and balancing trees to reduce their height.

To insert a new node with key el, a tree node with a dead end has to be reached, and new node has to be attached to it. Such a tree node is found using the same technique as used in tree searching.

The key el is compared to the key of the node currently being examined during tree scan. If el is less than that key, the left child (if any) of p is tried; otherwise the right child is tested. If the child of p to be tested is empty, the scanning is discontinued and the new node becomes this child.

The algorithm for inserting an item into binary search tree is as follows:

1. Start
2. Set r = root, save = root
3. Create new node

BNode temp = (struct BNode \*) malloc (sizeof(BNode))

1. Temp->info = data, temp->right = null, temp->right = null
2. If root == null

root = temp;

1. Otherwise
2. While (r!=null)

7.1 Save = r

7.2 if(data<r->data)

r = r->left;

else

r = r->right;

1. If(data<save->data)

Save->left = temp;

1. Else
2. Save->right = temp;
3. stop

**The C function for insertion into BST is:**

void insert(int num)

{

BNode \*r = root;

BNode \*save = root;

struct BNode \*temp = (struct BNode \*)malloc(sizeof(struct BNode));

temp->data = num;

temp->left = NULL;

temp->right = NULL;

if(root==NULL)

{

root = temp;

}

else

{

while(r!=NULL)

{

save = r;

if(num<r->data)

r = r->left;

else

r = r->right;

}

if(num<save->data)

save->left = temp;

else

save->right = temp;

}

}

**Deletion**

Deleting a node is another operation to maintain a binary Search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has. There are three cases of deleting a node from the binary search tree.

1. The node is a leaf; it has no children: This is the easiest case to deal with. In this case simply delete a node and set null pointer to its parents along the side of deleted node. Suppose that we are deleting a node with key 5 in the following BST
2. The node has one child: In this case the child of the node to be deleted is appended to its parent node. Let us consider the following BST where node having key 16 is deleted.
3. The node has two children. In this case, the deleted node is replaced by the smallest node in the right sub-tree or the largest node in the left subtree. Let us consider the following BST. We have to delete node with key 10. After deleting 10, we replace this node by either 9(largest element in the left subtree) or 14(smallest element in the right subtree).

Algorithm for deletion of a node in BST.

1. Start
2. If a node to be deleted is a left leaf node then simply delete this node and set null pointer to its parent’s left pointer.
3. If a node to be deleted is right leaf node then simply delete this node and set null pointer to it’s parent’s right child.
4. If a node to be deleted has one child then connect it’s child pointer with it’s parent pointer and delete it from the tree.
5. If a node to be deleted has two children then replace the node being deleted either by
6. Right most of its left sub sub-tree or
7. Left most node of it’s right sub-tree.
8. End

The C function for deletion of node is as follows:

void deletenode(int data)

{

struct BNode \*move,\*back,\*temp,\*ptr,\*prev;

ptr = root;

if(ptr==NULL)

{

printf("nEmpty tree..............\n");

return;

}

else

{

move = back = ptr;

while(move->data!=data)

{

back=move;

if(data<move->data)

move=move->left;

else

move=move->right;

}

if(move->left!=NULL&&move->right!=NULL)

{

temp=move->right;

back = move;

while(temp->left!=NULL)

{

move=temp;

temp=temp->left;

}

back->data=temp->data;

if(back==move)

move->right = temp->right;

else

move->left = temp->right;

free(temp);

return;

}

if(move->left==NULL&&move->right==NULL)

{

if(back->right==move)

back->right=NULL;

else

back->left=NULL;

free(move);

return;

}

if(move->left==NULL&&move->right!=NULL)

{ if(back->left==move)

back->left=move->right;

else

back->right=move->right;

free(move);

return;

}

if(move->left!=NULL&&move->right==NULL)

{

if(back->left==move)

back->left=move->left;

else

back->right=move->left;

free(move);

return;

}

}

}

The complete C program for implementing BST is as follows:

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct BNode

{

struct BNode \*left;

int data;

struct BNode \*right;

};

struct BNode \*root = NULL;

void insert(int);

int search(int);

void preorder(struct BNode \*);

void inorder(struct BNode \*);

void postorder(struct BNode \*);

void deletenode(int k);

int main()

{

insert(30);

insert(20);

insert(40);

insert(15);

insert(13);

insert(25);

insert(23);

insert(22);

insert(35);

insert(45);

insert(27);

insert(42);

insert(17);

printf("BST in preorder Traversal\n");

inorder(root);

if(search(30))

printf("\nItem found");

else

printf("\nItem not found");

deletenode(30);

printf("\nAfter Deletion\n");

inorder(root);

getch(gdq mgk);

return 0;

}

void insert(int num)

{

BNode \*r = root;

BNode \*save = root;

struct BNode \*temp = (struct BNode \*)malloc(sizeof(struct BNode));

temp->data = num;

temp->left = NULL;

temp->right = NULL;

if(root==NULL)

{

root = temp;

}

else

{

while(r!=NULL)

{

save = r;

if(num<r->data)

r = r->left;

else

r = r->right;

}

if(num<save->data)

save->left = temp;

else

save->right = temp;

}

}

void preorder(struct BNode \*r)

{

if(r!=NULL)

{

printf("%d\t",r->data);

preorder(r->left);

preorder(r->right);

}

}

void inorder(struct BNode \*r)

{

if(r!=NULL)

{

inorder(r->left);

printf("%d\t",r->data);

inorder(r->right);

}

}

void postorder(struct BNode \*r)

{

if(r!=NULL)

{

postorder(r->left);

postorder(r->right);

printf("%d\t",r->data);

}

}

int search(int key)

{

struct BNode \*temp = root;

int b=0;

while(temp!=NULL)

{

if(temp->data==key)

{

b = 1;

break;

}

else if(key<temp->data)

temp = temp->left;

else

temp = temp->right;

}

return b;

}

void deletenode(int data)

{

struct BNode \*move,\*back,\*temp,\*ptr,\*prev;

ptr = root;

if(ptr==NULL)

{

printf("nEmpty tree..............\n");

return;

}

else

{

move = back = ptr;

while(move->data!=data)

{

back=move;

if(data<move->data)

move=move->left;

else

move=move->right;

}

if(move->left!=NULL&&move->right!=NULL)

{

temp=move->right;

back = move;

while(temp->left!=NULL)

{

move=temp;

temp=temp->left;

}

back->data=temp->data;

if(back==move)

move->right = temp->right;

else

move->left = temp->right;

free(temp);

return;

}

if(move->left==NULL&&move->right==NULL)

{

if(back->right==move)

back->right=NULL;

else

back->left=NULL;

free(move);

return;

}

if(move->left==NULL&&move->right!=NULL)

{ if(back->left==move)

back->left=move->right;

else

back->right=move->right;

free (move);

return;

}

If (move->left!=NULL&&move->right==NULL)

{

If (back->left==move)

back->left=move->left;

else

back->right=move->left;

free (move);

return;

}

}

}

**AVL Tree**

The first balanced binary tree is the AVL tree, named after their inventor Adelson, Velski and Landis. AVL trees are height balanced binary search tree. AVL tree checks the height of the left and right sub-trees and assures that the difference is not more than 1. This difference is called the **Balancing Factor**. AVL tree is a binary search tree where the balance factor at each node is -1,0, or 1.

The balance factor for AVL tree is defined as:

**Balance Factor = height(left subtree)-height(right subtree)**

If the difference in the height is greater than 1 then the tree is not balanced and we need to make height balance by using some rotation techniques. To balance AVL tree we perform following kinds of rotations.

* Left Rotation
* Right Rotation
* Left Right Rotation
* Right Left Rotation

The first two rotations are single rotations and last two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2.

**Left Rotation:** If a tree becomes unbalanced when a node is inserted into the right of the right subtree then we perform a single left.

**Right Rotation**

AVL tree may become unbalanced if a node is inserted in the left of left subtree. The tree then needs a right rotation

**Left Right Rotation :** Double rotation are slightly complex. A left right rotation is a combination of left rotation followed by right rotation.

A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

**Right Left Rotation :** The second type of double rotation is Right -Left Rotation. It is a combination of right rotation followed by left rotation.

Procedure to construct an AVL Tree: To construct an AVL tree from given set of data we must consider following steps.

1. Take a first element of given array of elements and make it as first node of AVL tree
2. Set next (second) node either left or right side of given AVL tree
3. If node(1)>node(2) then set node(2) in left of node(1)
4. Otherwise set node(2) in right side of node(1)
5. Continue this process until all the elements are not included in resulting AVL tree and maintain balance factor for each node either -1 or 0 or 1
6. If balance factor of a particular node is not given range(-1 to 1) then rotate either single or double.
7. If straight path takes place then perform single rotation
8. Otherwise perform double rotation.

**Deleting data element from AVL Tree**

Deleting element from AVL tree is same as that of deleting element from BST. Slightly different is that here we need to check the balance factor of each node after deleting the node from the tree. Initially we need to search the node to be deleted. The node to be deleted could be a leaf node, a node with one child node or node of two child etc.

**Expression Tree**

A binary tree with each leaf node contains the operands and internal node contains the operators of given expression is called expression tree. During constructing expression tree from given expression we take an operator with lowest precedence as root node and set subtree and right subtree around the root node. Again left subtree and right subtree acts an expression tree recursively.

**Example:** Construct an expression tree of given infix expression: A\*B-(C+D)\*(P/Q)

A\*B

(C\*D)\*(P/Q)

Step 1

C+D

P/Q

Step 2

The expression tree for a given expression can be built recursively from the following rules

* The expression tree for a single operand is single root node that contains it.
* If **E1** and **E2** are expressions represented by expression trees **T1** and **T2** and if **op** is an operator, then the expression tree for the expression **E1** **op** **E2** is the tree with root node containing **op** and sub-trees **T1** and **T2**.

An expression has three representations, depending upon which traversal algorithm is used to traverse its tree. The preorder traversal produces the prefix representation, the inorder traversal produces the infix representation and the post order traversal produces the postfix representation of the expression. The postfix expression is also called reverse police notation or RPN.

**Prefix Representation of given Expression**

An expression is called prefix expression if operands are followed by operands. This means towards left side operators are present and right side of expression operands are present.

Example: A prefix expression is:

+\*-A B C/6 B D

**Infix Representation of given Expression**

An expression is called infix expression if every operand surrounds by any two operands.

Example: An infix expression:

A-B\*C+6/B+D

**Postfix Representation of given Expression**

An expression is called postfix expression if operators are followed by operands. This means towards left side operands are present and right side of the expression operators are present.

Example: A postfix expression is:

A B -C \*6 A D +/ +

**Evaluating an Expression from its Postfix Representation**

To evaluate an expression represented in postfix, scan the representation from left to right and perform the following operations:

1. Create a stack for operands
2. Repeat steps 3-9 until the end of representation is reached.
3. Read the next token t from the representation
4. If it is an operand the push its value onto the stack
5. Otherwise, do steps 6-9.
6. Pop top operands 'a' from the stack
7. Pop second top operand 'b' from the stack
8. Evaluate given operation of operands 'a' and 'b' as c=a t b
9. Push result 'c' onto the stack
10. Return the top element on the stack.

**Example: Evaluate the expression [A B -C\*B D+\*] where A = 5, B=2 and C = 3 and D=1**

Input Tape

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | - | C | \* | A | D | + | \* |  |

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Stack

Constructing an expression from postfix expression

1. Scan given postfix expression from left to right
2. If scanned symbol is operand then

Make a tree with only one node and push its pointer to the top of the stack

1. If scanned symbol is operator then
2. Pop top element and place it to T2
3. Again pop top of the element and place it to T1
4. Make a new tree with root node as given operator and its left and right subtree as T1 and T2 respectively
5. Continue this process until scanning pointer do not reach at last location of given postfix expression

**Example : Construct an expression tree of the postfix expression A B + C D E + \* $**

Constructing an Expression tree from prefix expression

1. Start
2. Scan given postfix expression from right to left
3. If scanned symbol is operand then
4. Make a tree with only one node and push its pointer to the top of the stack
5. If scanned symbol is operator then
6. Pop top element and place it in T1
7. Again pop top element and place it to T2
8. Make a new tree with root node as given operator and its left and right sub-tree as T1 and T2 respectively
9. Continue this process until scanning pointer do not reach at last location of given postfix expression
10. Stop

**Example : Construct an expression tree of the prefix expression $+/ZAB\*CD**

**Graphs**

A simple graph G is composed of a finite set of vertices and a finite set of edges. A graph is denoted as G = (V,E). The elements of V are the vertices and those of E are the edges. The vertex set of V are denoted by VG and edge set is denoted by EG. Thus G = (VG, EG). The number of vertices is denoted by |V| and number of edges is denoted by |E|.

A directed graph or digraph G = (V, E) consists of a nonempty set V of vertices and a set of edges where each edge is a pair of vertices from V. The difference is that one edge of a simple graph is of the form {vi,vj} and for such an edge {vi,vj} = {vj,vi}. In digraph, the edge is of the form (vi,vj) and in this case, (vi,vj)≠(vj,vi). Unless necessary, this distinction in notation will be disregarded and edge between vertices vi and vj is will be referred to as edge (vi,vj).

A multigraph is a graph in which two vertices can be joined by multiple edges.

A path from vi to vn is a sequence of edges edge(v1,v2), edge(v2,v3),….(vn-1,vn) and is denoted as path v1,v2,v3,……………..vn-1,vn. If v1 = vn and no edge is repeated, then the path is called a circuit. If all vertices in a circuit are different, then it is called a cycle.

The graph is called a weighted graph if each edge has an assigned number. Depending on the context in which such graphs are used, the number assigned to an edge is called its weight, cost, distance, length or some other name.

A graph with n vertices is called complete and is denoted by Kn if for some each pair of distinct vertices there is exactly one edge connecting them: that is, each vertex can be connected to any other vertex.

**Graph Representation**

There are two standard ways to represent a graph G = (V,E) namely adjacency lists (linked list representation) and adjacency matrix. Either way is applicable to both directed and undirected graph.

Adjacency List: The adjacency list representation of a graph G = (G,E) consists of an array Adj of |V| lists, one for each vertex in V. For each u ɛ V, the adjacency list Adj[u]

Contains all the vertices v such that there is an edge (u,v) ɛ E. That is, Adj[u] consists of all vertices adjacent to u in G.

a

b

c

d

e

b

c

a

c

d

a

b

e

b

d

c

e

Figure: Adjacency list representation of undirected graph

a

b

c

d

e

b

d

e

e

f

b

e

f

**Figure: Adjacency list representation of directed graph**

The adjacency list representation is usually preferred to represent sparse graphs those for which |E| is much less than |V|2

A potential disadvantage is of the adjacency list representation is that there is no quicker way to determine if a given edge (u,v) is present in the graph than to search for v in the adjacency list Adj[u].

Adjacency Matrix: Given a simple graph G = (V,E) with |V| = n, we assume that the vertices are numbered 1,2, 3…. |V| in some arbitrary manner. Then the adjacency matric of graph G is n\*n matric A = (aij) such that

aij = 1 if there is an edge from vertex i to vertex j

= 0 otherwise

The adjacency matrices for above two graphs are as shown below:

a b c d e f

a 0 1 0 1 0 0

b 0 0 0 0 1 0

c 0 0 0 0 1 0

d 0 1 0 0 0 0

e 0 0 0 1 0 0

f 0 0 0 0 1 0

a b c d e

a 0 1 1 0 1

b 1 0 1 1 0

c 1 1 0 0 1

d 0 1 0 0 1

e 0 0 1 1 0

**Figure:** Adjacency matric for undirected Graph(left) and for directed graph (right)

An adjacency matrix representation may be preferred when graph is dense that is when |E| is close to |V|2.

Since there are n vertices and we may order vertices in n! ways. Hence there are n! possible adjacency matrices for a graph with n vertices

When number of edges is far less than |V|2 then we use adjacency list otherwise sparse matrix is generated.

**Graph Traversal:-** There are number of approaches used for solving problems on graphs. One of the most important approaches is based on the notion of systematically visiting all vertices and edges of a graph. The reason for this is that these traversals impose a type of three on the graph, and trees are usually much easier to reason about than general graphs.

**Breadth First Search:** This is one of the simplest methods of graph searching. We choose some vertex arbitrarily as a root. We add all the vertices and edges that are incident in the root. The newly added vertices will become the vertices at the level 1 of the BFS tree. From the vertices of level 1, we find other vertices that are reachable from level vertices. We continue until all the vertices are visited.

The algorithm is as follows:

BFS(G,S)

{

T = {s}

L = φ // Empty Queue

EnQueue(L,S);

While(L!= φ)

{

V = dequeue(L);

For each neighbor w to v

If(w≠L and w !ɛ T)

{

EnQueue(L,w)

T = T U {w}

}

}

}

**Analysis:** Let number of vertices be n. All vertices are put once in the queue. This can be done in O(n) time. For each vertex in queue their adjacent vertices are taken for and this again takes O(n) time (when graph is complete). Thus, total time required is O(n2).

Also from aggregate analysis, we can write the algorithm as O(E+V) as inner loop executes E times in total.

**Example: Use breadth first search to find a BFS tree of the following graph:**

**aaa**

Let a be the start vertex:

**Depth First Search:-** This another technique that can be used to search the graph. In this algorithm, each vertex v is visited and then unvisited vertex adjacent to v is visited. If a vertex v has no adjacent vertices or all of its adjacent vertices have been visited, we backtrack to the predecessor of v. The traversal is finished if this visiting and backtracking process leads to a first vertex where the traversal started. If there are still some unvisited vertices in the graph, the traversal continues restarting for one of the unvisited vertices.

**Depth First Search can be written as:**

**DFS (G,s)**

{

T = {s}

Traverse(s);

}

Traverse (v)

{

For each vertex w adjacent to v and not yet in T

{

T = T U {w}

Traverse (w)

}

}

**Analysis:** The complexity of DFS algorithm can be written as T(n) = T(n-1) +n. Solving we get T(n) = O(n2)

Also from aggregate analysis, we write the complexity as O(V+E) because traverse function is invoked V times at maximum and for loop executes O€ times in total.

**Example: Use depth first search to find a spanning tree of the following f=graph:**

Choose a as initial vertex then we have:

**Spanning Tree:** A spanning tree of a graph consists of all vertices and some of the edges so that graph does not contain a cycle. For example, for the following graph.

The spanning tree:

**minimum spanning Tree:** A minimum spanning tree (MST) is defined for a weighted graph. It is the spanning tree for a weighted graph. It is the spanning graph whose sum of weight of edges in the spanning tree is minimum. For example for the following graph.

2

5

3

4

6

2

5

3

4

6

1

2

The minimum spanning tree

For finding a MST for a graph, there are two most popular algorithm given by Kruskal and Prim

**Kruskal’s Algorithm**

The problem of finding MST can be solved by using Kruskal’s algorithm. Here we put the set of edges in non-decreasing order of their weights. The selection of each edge in sequence then guarantees that the total cost that would form will be minimum. Here G is a graph, V is as set of n vertices and E as set of edges.

**Algorithm:**

KruskalMST(G)

{

T = {v}

S = set of edges sorted in non-decreasing order of weight

While (|T|<n-1 and E!=φ)

{

Select (u,v) from S in order

Remove (u,v) from E

If (u,v) does not create a cycle in T

T = T U {(u,v)}

}

}

**Analysis:**Creating n tree forest at beginning takes **O(V)** time. Creating of the set S takes **O(ElogE)** time. The while loop executes O(V) time and steps inside the loop take almost constant time. So, total time taken is **O(ElogE)** which is asymptotically equivalent to **O(E logV)**.

**Example: Find the MST and its weight of the given graph.**

2

5

3

4

6

1

8

3

2

1

2

1

1

3

2

1

4

**Prim's Algorithm**

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of edges of that graph which

* Form a tree that includes every vertex
* Has the minimum sum of weights among all the trees that can be formed from the graph.

The idea behind this algorithm is just take any arbitrary vertex and choose the edge with minimum weight incident on the chosen vertex. Add the vertex and continue the above process until the vertices are not added to the list. Remember the cycle must be avoided

Algorithm

1. Start
2. Initialize the minimum spanning tree with a vertex chosen at random
3. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
4. Keep repeating step 3 until we get a minimum spanning tree
5. Stop

Pseudocode

PrimMST(G)

{

T = ø

S = {s}

while(S!=T)

{

e = (u,v) and edge of minimum weight incident to vertices in T and not forming simple circuit in T if added to T.

T = T U {(u,v)}

S = S U {v};

}

}

Analysis:

In the above algorithm while loop executes O(V). The edge of minimum weight incident on a vertex can be found in O(E), so the total time is O(EV). We can improve the performance of the above algorithm by choosing better data structure as priority queue and normally it will be seen that the running time of prim's algorithm is O(ELogV).