PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Expression Trees

Dr. Shylaja S S
Ms. Kusuma K V

Expression Tree

An expression tree is built up from the simple operands and operators of an (arithmetic or logical) expression by placing the simple operands as the leaves of a binary tree and the operators as internal nodes.

For each binary operator, the left subtree contains all the simple operands and operators in the left operand of the given operator, and the right subtree contains everything in the right operand.

For a unary operator, one of the two subtrees will be empty. We traditionally write some unary operators to the left of their operands, such as '-'(unary negation) or the standard functions like log() and cos(). Other unary operators are written on the right, such as the factorial function ()! or the function that takes the square of a number, () ^{2}. Sometimes either side is permissible, such as the derivative operator, which can be written as d/dx on the left, or as ()' on the right, or the incrementing operator ++ in the C language (where the actions on the left and right are different). If the operator is written on the left, then in the expression tree we take its left subtree as empty, so that the operand appear on the right side of the operator in the tree, just as they do in the expression. If the operator appears on the right, then its right subtree will be empty, and the operands will be in the left subtree of the operator.

The expression trees of few simple expressions are shown in Figure 1.

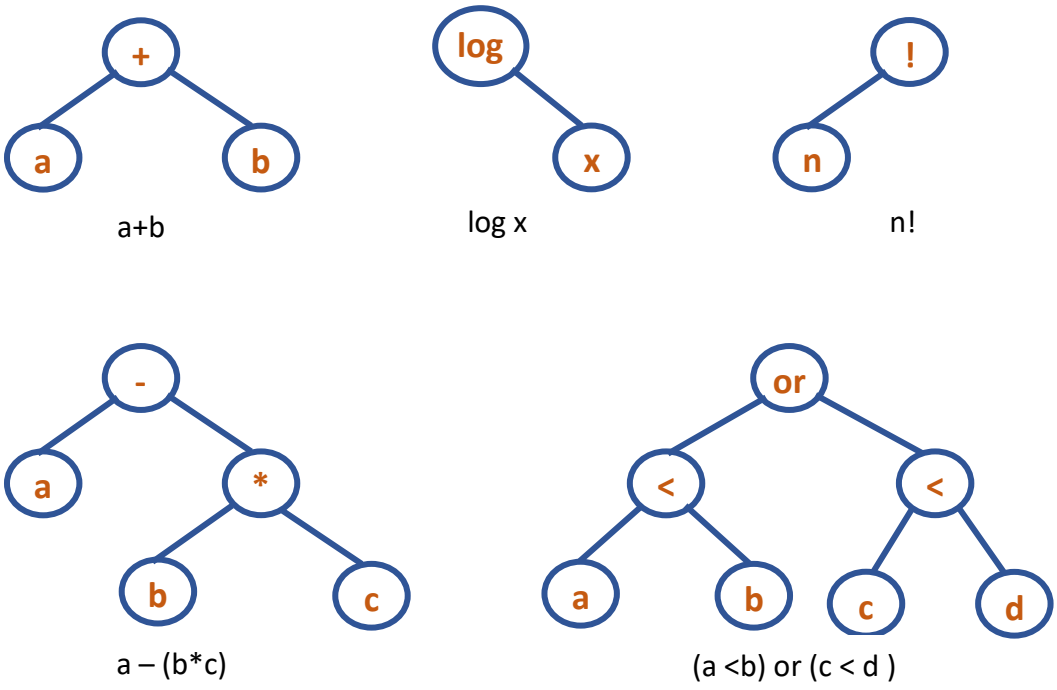


Figure 1: Expression Trees

Construction and Evaluation of an Expression Tree

1) Scan the postfix expression till the end, one symbol at a time
    a) Create a new node, with symbol as info and left and right link as NULL
    b) If symbol is an operand, push address of node to stack
    c) If symbol is an operator
        i) Pop address from stack and make it right child of new node
        ii) Pop address from stack and make it left child of new node
        iii) Now push address of new node to stack
2) Finally, stack has only element which is the address of the root of expression tree

//Binary Expression Tree

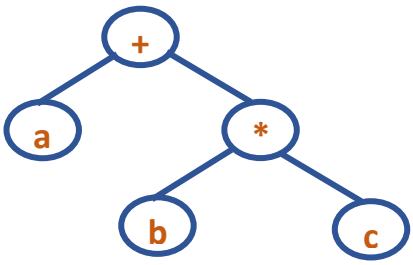
```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#define MAX 50

typedef struct node
{
        char info;
        struct node *left,*right;
}NODE;

typedef struct tree
{
        NODE* root;
}TREE;

typedef struct stack
{
        NODE* s[50];
        int top;
}STACK;

void init_trr(TREE *pt)
{
        pt->root=NULL;
}

void init_stack(STACK *ps)
{
        ps->top=-1;
}
```

```
int push(STACK *ps,NODE* e)
{
	if(ps->top==MAX-1)
		return 0;
	ps->top++;
	ps->s[ps->top]=e;

	return 1;
}

NODE* pop(STACK *ps)
{
	//Assumption: empty condition checked before entering the pop
	NODE *t=ps->s[ps->top];
	ps->top--;

	return t;
}
```

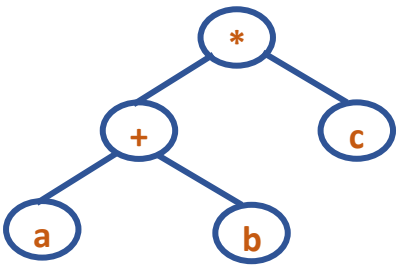
//Expression Tree Evaluation

```
float eval(NODE* r)
{
	float res;
	switch(r->info)
	{
		case '+': return (eval(r->left)+eval(r->right));
				break;
		case '-': return (eval(r->left)-eval(r->right));
				break;
		case '*': return (eval(r->left)*eval(r->right));
				break;
		case '/': return (eval(r->left)/eval(r->right));
				break;
		default: return r->info - '0';
	}
	return res;
}

float eval_tree(TREE *pt)
{
	return eval(pt->root);
}
```

```
int main()
{
        char postfix[MAX];
        STACK sobj;
        TREE tobj;
        NODE *temp;
        init_stack(&sobj);
        printf("Enter a valid postfix expression\n");
        scanf("%s",postfix);

        int i=0;
        while(postfix[i] != '\0')
        {
                temp = (NODE*) malloc(sizeof(NODE));

                temp->info = postfix[i];
                temp->left=NULL;
                temp->right=NULL;

                if(isdigit(postfix[i]))
                        push(&sobj,temp);
                else
                {
                        temp->right=pop(&sobj);
                        temp->left=pop(&sobj);
                        push(&sobj,temp);
                }
                i++;
         }
        tobj.root=pop(&sobj);
        printf("%f",eval_tree (&tobj));
        return 0;
}
```



(i) a + (b*c)



(ii) (a + b)*c

Figure 2: Expressions and their binary tree representations

Traversal of the binary expression trees in Figure 2 is as follows:

Figure 2(i): Preorder: +a*bc	Inorder: a+b*c	Postorder: abc*+
Figure 2(ii): Preorder: *+abc	Inorder: a+b*c	Postorder: ab+c*

It can be observed that traversing the binary expression tree in preorder and postoder yields the corresponding prefix and postfix form of the infix expression. But we can see that the inorder traversal of the binary expression trees doesn't always yield the infix form of the expression. This is because the binary expression tree does not contain parentheses, since the ordering of the operations is implied by the structure of the tree.

General Expression Tree

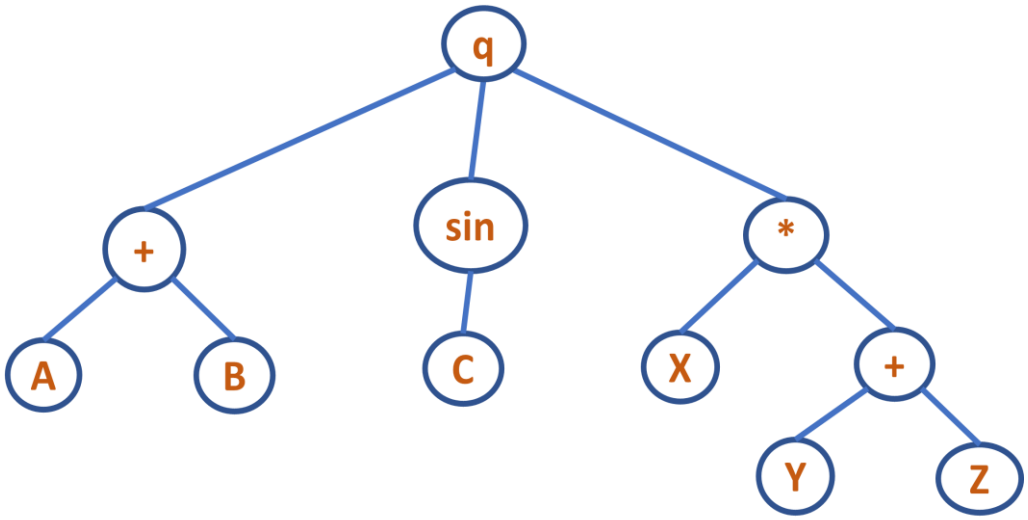


Figure: Tree representation of an arithmetic expression

Here node can be either an operand or an operator

```
struct treenode
{
 short int utype;
  union{
    char operator[MAX];
     float val;
   }info;
  struct treenode *child;
  struct treenode *sibling;
};
typedef struct treenode TREENODE;
```

```
void replace(TREENODE *p)
{
  float val;
  TREENODE *q,*r;
  if(p->utype == operator)
 {
   q = p->child;
   while(q != NULL)
    {
      replace(q);
      q = q->next;
    }
 }
  value = apply(p);
  p->utype = OPERAND;
  p->val = value;
  q = p->child;
  p->child = NULL;
  while(q != NULL)
  {
      r = q;
      q = q->next;
      free(r);
  }
 }
}

float eval(TREENODE *p)
{
 replace(p);
 return(p->val);
 free(p);
 }
```

Constructing a Tree

```
void setchildren(TREENODE *p,TREENODE *list)
{
  if(p == NULL) {
    printf("invalid insertion");
    exit(1);
  }
 if(p->child != NULL) {
   printf("invalid insertion");
   exit(1);
 }
 p->child = list;
}

void addchild(TREENODE *p,int x)
{
 TREENODE *q;
 if(p==NULL)
 {
   printf("void insertion");
   exit(1);
 }
 r = NULL;
 q = p->child;

 while(q != NULL)
 {
   r = q;
   q = q->next;
 }
 q = getnode();
 q->info = x;
 q->next = NULL;

 if(r==NULL)
   p->child=q;
 else
   r->next=q;
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Binary Search Tree (BST) and its Implementation using Dynamic Allocation: Insertion

Dr. Shylaja S S
Ms. Kusuma K V

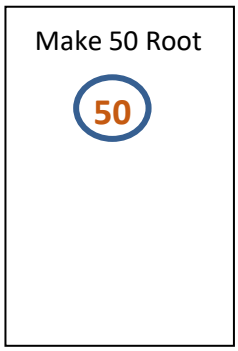
Binary Search Tree: is a Binary tree that is either empty or in which every node contains a key and satisfies the conditions:

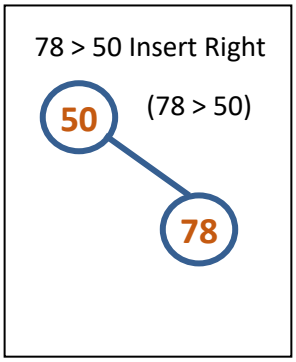
- The key in the left side of a child (if it exists) is less than the key in the parent node
- The key in the right side of a child (if it exists) is greater than the key in the parent node
- The left and right subtrees of the root are again binary search trees

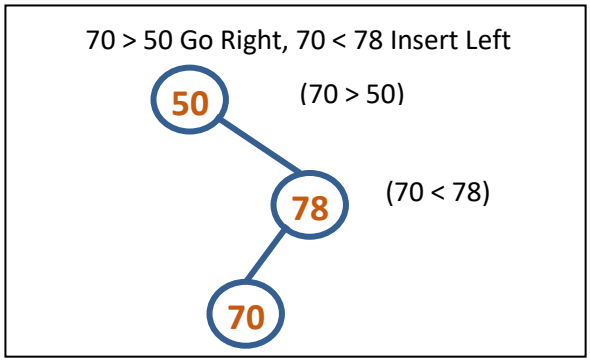
If in case of equal keys we can modify the definition but normally we take distinct keys.

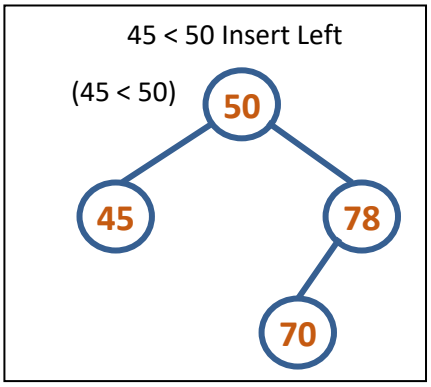
Construct a Binary Search Tree for the elements inserted in the following order:

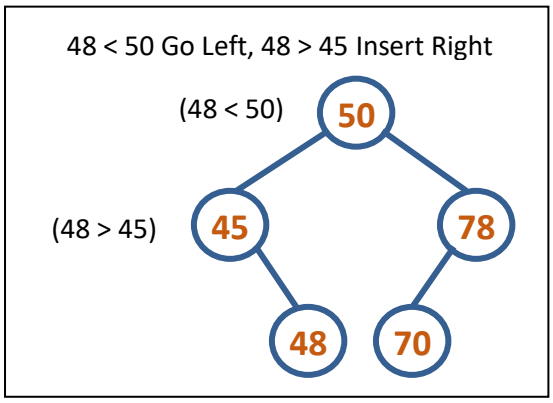
50, 78, 70, 45, 48, 35, 85

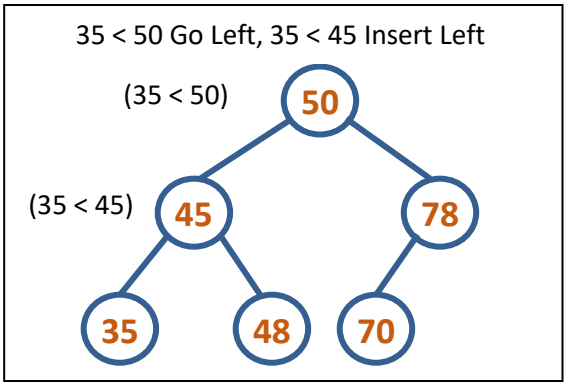


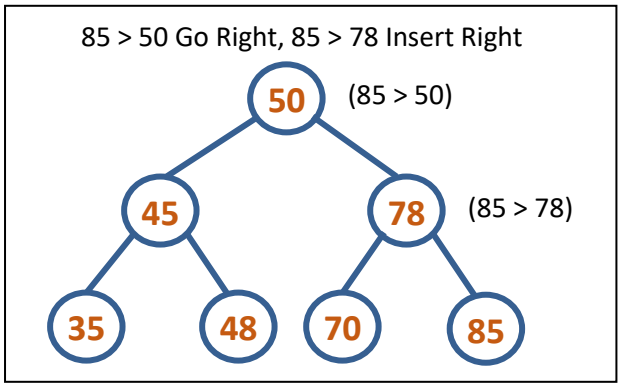












BST: Linked Representation

In linked representation each node in a binary tree has three fields, the left child field, information field and the right child field. Pictorially a node used for linked representation may be as shown in Figure3. If the left subtree is empty then the corresponding node's left child field is set to null. If the right subtree is empty then the corresponding node's right child field is set to null. If the tree itself is empty the root pointer is set to null.

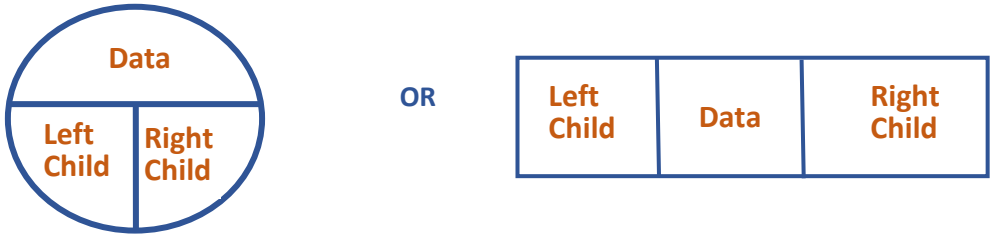


Figure 3: Pictorial representation of a node in Linked representation

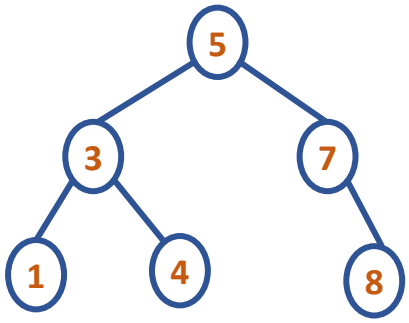


Figure 4: Binary Search Tree

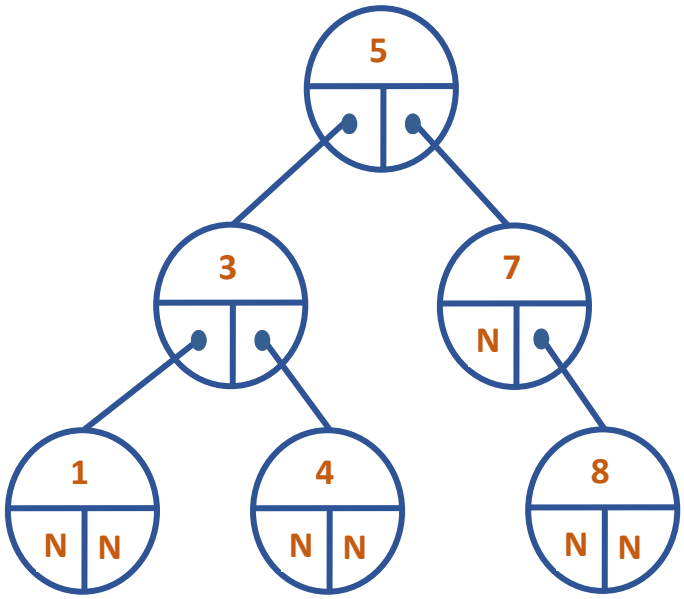


Figure 5: Linked Representation of BST in Figure 4

```
//BST Linked Representation
#include<stdio.h>
#include<stdlib.h>

typedef struct node
{
        int info;
        struct node *left,*right;
}NODE;
```

```
typedef struct tree
{
	NODE *root;
}TREE;

void init(TREE *pt)
{
	pt->root=NULL;
}

void creat(TREE *pt)
{
	NODE *temp,*p,*q;
	int wish;

	printf("Enter the root info\n");
	pt->root=(NODE*)malloc(sizeof(NODE));
	scanf("%d",&pt->root->info);

	pt->root->left=NULL;
	pt->root->right=NULL;
	do{
		printf("Enter an element\n");
		temp=(NODE*)malloc(sizeof(NODE));
		scanf("%d",&temp->info);
		temp->left=NULL;
		temp->right=NULL;
		q=NULL;
		p=pt->root;

		while(p!=NULL)
		{
			q=p;
			if(temp->info < p->info)
				p=p->left;
			else
				p=p->right;
		}
		if(temp->info < q->info)
			q->left=temp;
		else
			q->right=temp;
```

```
                printf("Do you wish to add another? 1/0\n");
                scanf("%d",&wish);
        }while(wish);
}

void intr(NODE* p)
{
        if(p!=NULL)
        {
                intr(p->left);
                printf("%d ",p->info);
                intr(p->right);
        }
}

void intrav(TREE *pt)
{
        intr(pt->root);
}

int main()
{
        TREE tobj;

        creat(&tobj);
        intrav(&tobj);

        return 0;
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Binary Search Tree (BST): Deletion

Dr. Shylaja S S
Ms. Kusuma K V

Deletion of a Node in Binary Search Tree

Consider the following 3 cases for deletion of a node in Binary Search Tree so that even after the node is deleted the BST property is preserved.

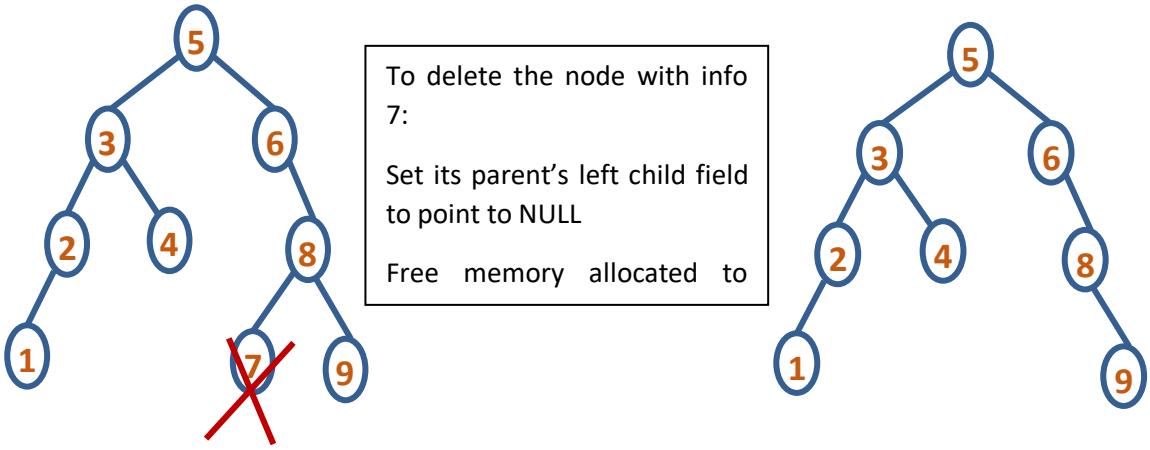
Case 1: Node with no child (leaf node)

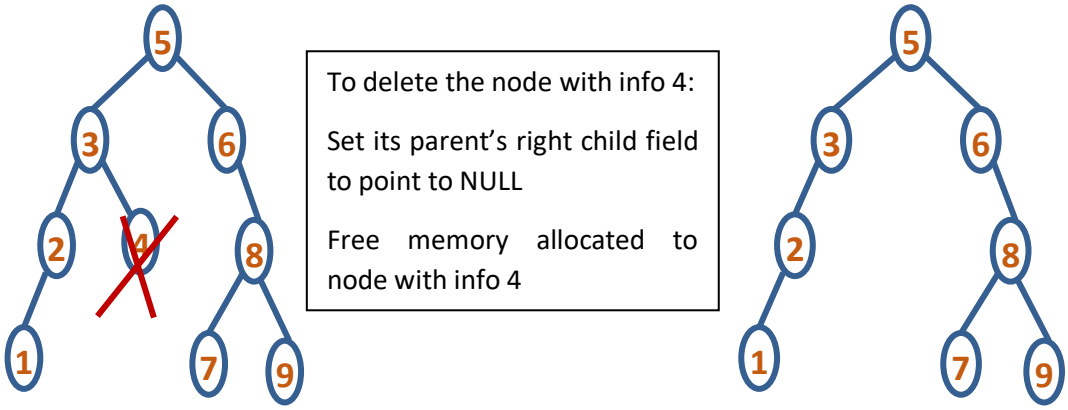
Case 2: Node with 1 child

Case 3: Node with 2 children

Case 1: Deletion of a leaf node

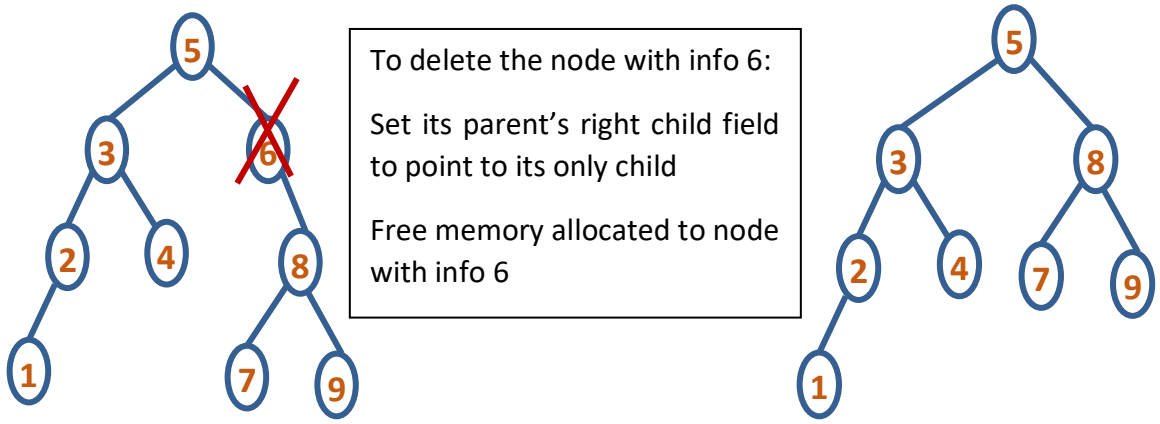
Set its parent's corresponding child field to point to NULL. Free memory allocated to the node.

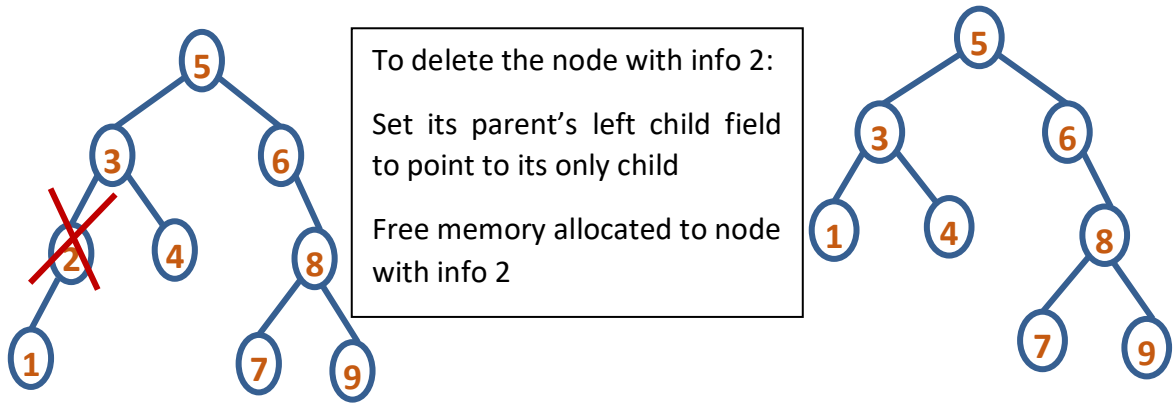




Case 2: Deletion of a node with one child

Connect the node's parent with node's child node





Case 3: Deletion of a node with 2 children

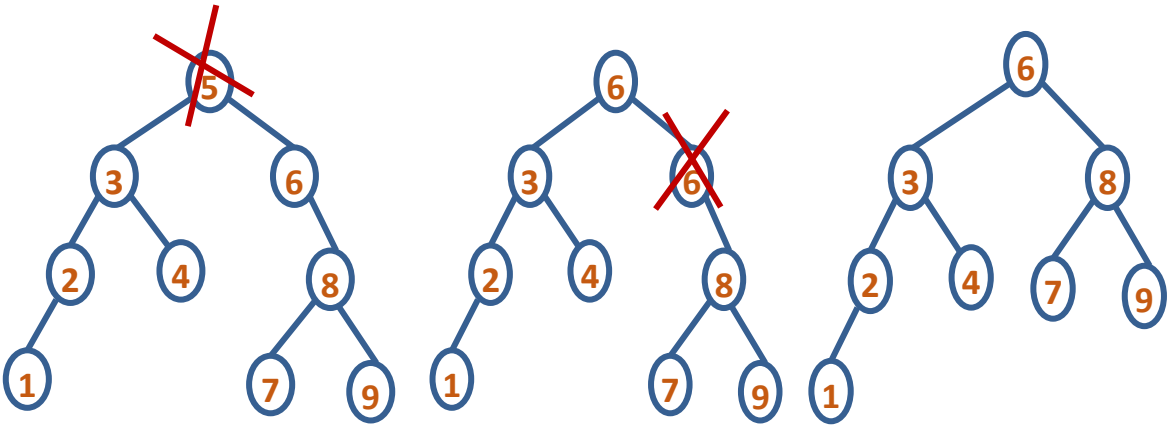
This case can be handled in 2 ways:

Way 1: Replace the node to be deleted with its inorder successor (Smallest in its Right subtree or Leftmost in its Right subtree) and delete that inorder successor.

Way2: Replace the node to be deleted with its inorder predecessor (Largest in its Left subtree) and delete that inorder predecessor.

For the deletion of inorder successor/predecessor, case 3 gets reduced to either case 1 or case 2. We know how to solve case1 and case2.

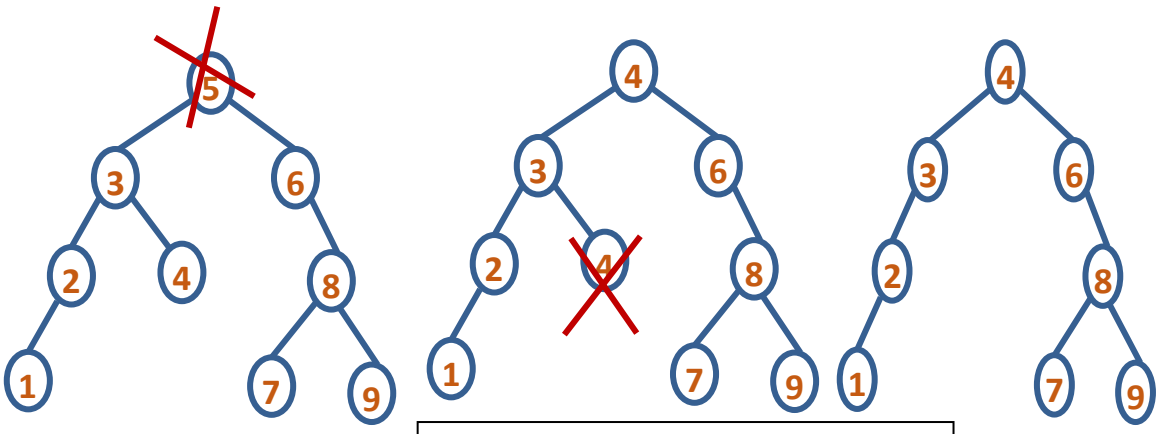
To delete node with info 5 (Way 1: Replace with inorder successor)



Replace 5 with its inorder successor

Now delete that inorder successor
Now case3 has got changed to case2 (In general may change to case2 or case1)

To delete node with info 5 (Way 2: Replace with inorder predecessor)



Replace 5 with its inorder predecessor

Now delete that inorder predecessor
Now case3 has got changed to case1 (In general may change to case2 or case1)

Note: For implementation we shall consider replacing with inorder successor.

```
//BST Deletion
#include<stdio.h>
#include<stdlib.h>
typedef struct node
{
        int info;
        struct node *left,*right;
}NODE;
typedef struct tree
{
        NODE *root;
}TREE;

void init(TREE  *pt)
{
        pt->root=NULL;
}

NODE* createNode(int e)
{
        NODE *temp=malloc(sizeof(NODE));
        temp->left=NULL;
        temp->right=NULL;
        temp->info=e;
        return temp;
}

void create(TREE *pt)
{
        NODE *p,*q;
        int e, wish;

        printf("Enter info\n");
        scanf("%d",&e);

        pt->root=createNode(e);

        do{
                printf("Enter info\n");
                scanf("%d",&e);

                q=NULL;
```

```
                p=pt->root;

                while(p!=NULL)
                {
                        q=p;

                        if(e < p->info)
                                p = p->left;
                        else
                                p = p->right;
                }

                if(e < q->info)
                        q->left = createNode(e);
                else
                        q->right = createNode(e);

                printf("Do you wish to continue\n");
                scanf("%d",&wish);
        }while(wish);
}

void io(NODE* r)
{
        if(r!=NULL)
        {
                io(r->left);
                printf("%d ",r->info);
                io(r->right);
        }
}

void inorder(TREE *pt)
{
        io(pt->root);
}
```

```
NODE* delNode(NODE *r,int ele)
{
        NODE *temp,*p;

        if(r==NULL)
                return r;

        if(ele < r->info)
                r->left = delNode(r->left,ele);
        else if(ele > r->info)
                r->right =  delNode(r->right,ele);
        else
        {
                if(r->left == NULL)                     //1 right child or No children
                {
                        temp=r->right;
                        free(r);
                        return temp;
                }
                else if(r->right == NULL)               //1 left  child or No children
                {
                        temp=r->left;
                        free(r);
                        return temp;
                }
                else
                {       //Node to be deleted has both children
                        //Finding p's leftmost child which is the inorder successor
                        p=r->right;
                        while(p->left != NULL)
                                p=p->left;

                        r->info=p->info;
                        r->right=delNode(r->right, p->info);
                }
        }
        return r;
}
void deleteNode(TREE *pt,int e)
{
        pt->root=delNode(pt->root,e);
}
```

```
int main()
{
        int e;
        TREE t;
        init(&t);
        create(&t);
        inorder(&t);
        printf("Enter the element to be deleted\n");
        scanf("%d",&e);
        deleteNode(&t,e);
        printf("After deletion\n");
        inorder(&t);

        return 0;
}
```

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
BST: Implementation using Arrays

Dr. Shylaja S S
Ms. Kusuma K V

Implicit Array Representation of BST

In the implicit array representation, an array element is allocated whether or not it serves to contain a node of a tree. We must, therefore, flag unused array elements as nonexistent, or null, tree nodes. This may be accomplished by adding a logical flag field, used, to each node. Each node then contains two fields: info and used.

```
typedef struct node
{
        int info;
        int used;
}NODE;
```

NODE[p].used is TRUE if node p is not a null node and FALSE if it is a null node.

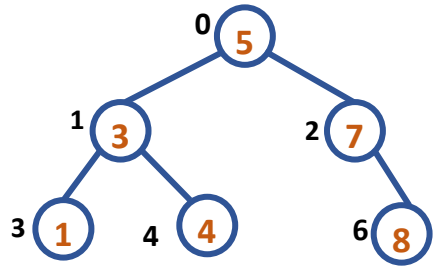


Figure 2: Binary Search Tree

Position: p	0	1	2	3	4	5	6
info	5	3	7	1	4		8
used	1	1	1	1	1	0	1

Array Representation of Binary Search Tree in Figure 2

//BST Array Implementation

```
#include<stdio.h>

typedef struct tree_node
{
        int info;
        int used;
}TREE;

#define MAXNODES 50

void init(TREE t[MAXNODES])
{
        for(int i=0;i<MAXNODES;i++)
                t[i].used=0;
}
```

```
void create(TREE *bst)
{
	int ele, wish;

	printf("Enter the root element\n");
	scanf("%d",&bst[0].info);
	bst[0].used=1;

	do{
		printf("Enter an element\n");
		scanf("%d",&ele);

		int p=0;

		while(p<MAXNODES && bst[p].used)
		{
			if(ele<bst[p].info)
				p=2*p+1;
			else
				p=2*p+2;
		}

		if(p>=MAXNODES)
			printf("Insertion not possible\n");
		else
		{
			bst[p].info=ele;
			bst[p].used=1;
		}
		printf("Do you wish to add another\n");
		scanf("%d",&wish);
	}while(wish);
}
void inorder(TREE* bst, int r)
{
	if(bst[r].used)
	{
		inorder(bst,2*r+1);
		printf("%d ",bst[r].info);
		inorder(bst,2*r+2);
	}
}
```

```
int main()
{
        TREE bst[MAXNODES];

        init(bst);
        create(bst);
        inorder(bst,0);
        return 0;
}
```

Note that under this representation it is always required to check that the range (NUMNODES) has not been exceeded whenever we move down the tree.

Array Implementation is suitable for a complete binary tree. Otherwise there will be many vacant positions in between. In such cases we may look into an alternate representation of tree nodes, i.e., the Linked representation, which makes use of the left and right pointers along with the info field.

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

AVL TREES

Abstract

Balanced tree, Need for Balanced tree, definition, AVL Trees, Rotation.

Dr.Sandesh and Saritha

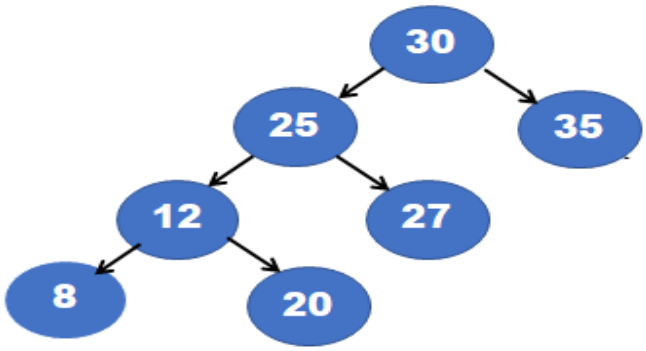
Sandesh_bj@pes.edu

Saritha.k@pes.edu

Balanced tree

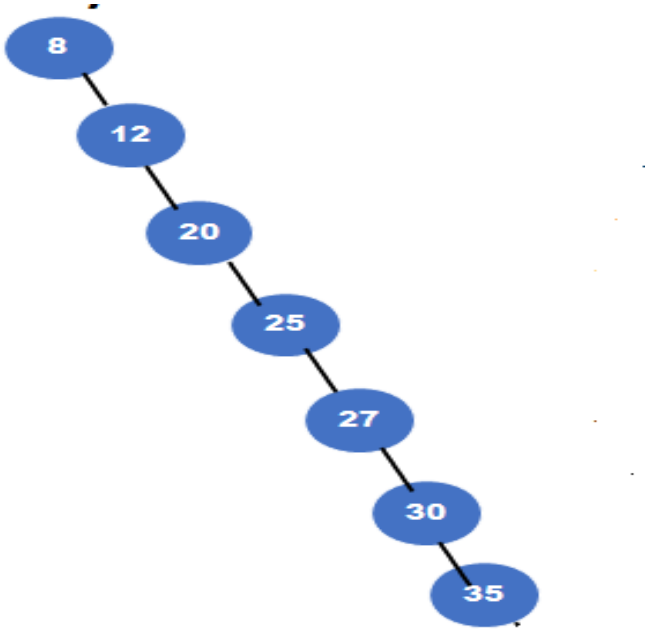
Most operations on a Binary search tree take time directly proportional to the height of the tree, so it is important to keep the height of the tree small.

A balanced binary search tree is a tree that naturally keeps its height small for a sequence of insertion and deletion operation.



Balanced Binary search tree

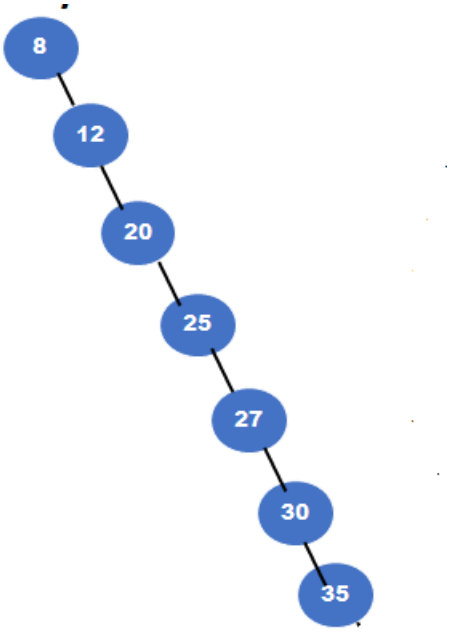
The tree shown above is balanced because the difference between the heights of left subtree and right subtree is not more than one.



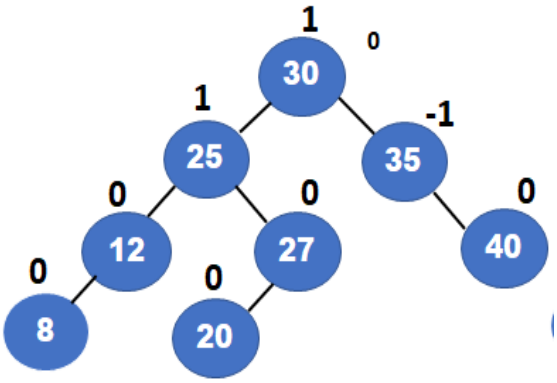
The tree shown above is unbalanced because the tree right side is 6 leaves taller than the left hand side.

Why do Binary search trees have to be balanced?

Balanced search tree decreases the number of comparisons required to find a data element. For example consider a below unbalanced tree, to search for an element 35 in the below tree seven comparisons is required. Whereas in balanced tree it requires only 2 comparisons which means the search performance increased by 50% in a balanced tree.



(i)Unbalanced tree



(ii) Balanced tree

There are different algorithms used to balance the binary search tree such as

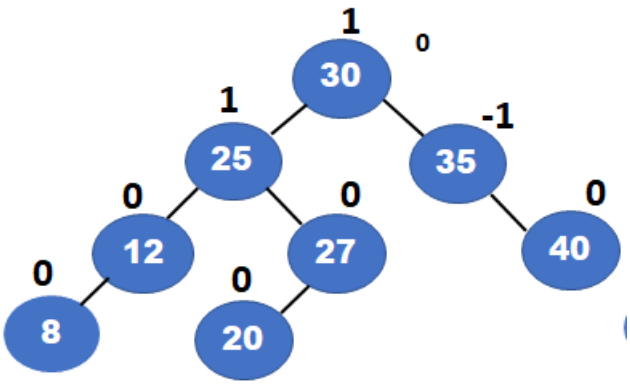
1. AVL trees
2. B-tree
3. Red Black trees

All these data structures force the tree to remain balanced and therefore guarantee performance.

AVL Tree

An AVL tree is height balanced Binary search tree invented by Adelson-Velskii and Landis. In AVL tree the heights of left and right sub-trees of a root differ by at-most one. Every node in the AVL tree is associated with balance factor that is left higher, equal-height or right-higher accordingly ,respectively as the left subtree has height greater than, equal to, or less than that of the right subtree.

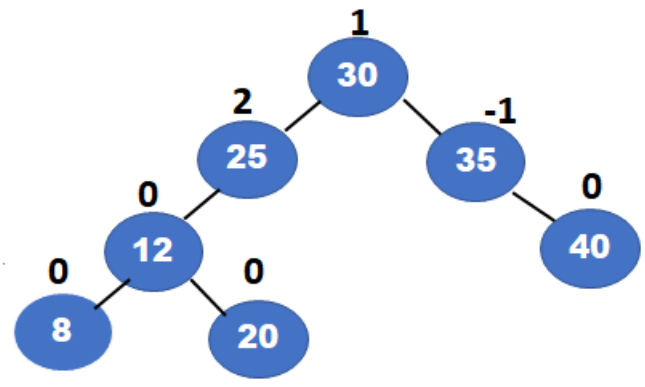
Example of AVL tree:



AVL tree

The tree shown above is an AVL tree as the difference between height of left subtree and right subtree of every node is at most one.

Example for not an AVL tree



Not an AVL tree

The above figure is not an AVL tree as the difference between height of left subtree and right subtree of root node is 2

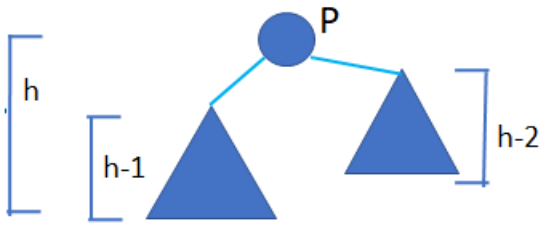
An AVL tree has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.
3. Height of AVL tree is always logarthmic in n.

An AVL tree has balance factor calculated at every node. For every node, height of left and right sub-trees can differ by no more than 1.

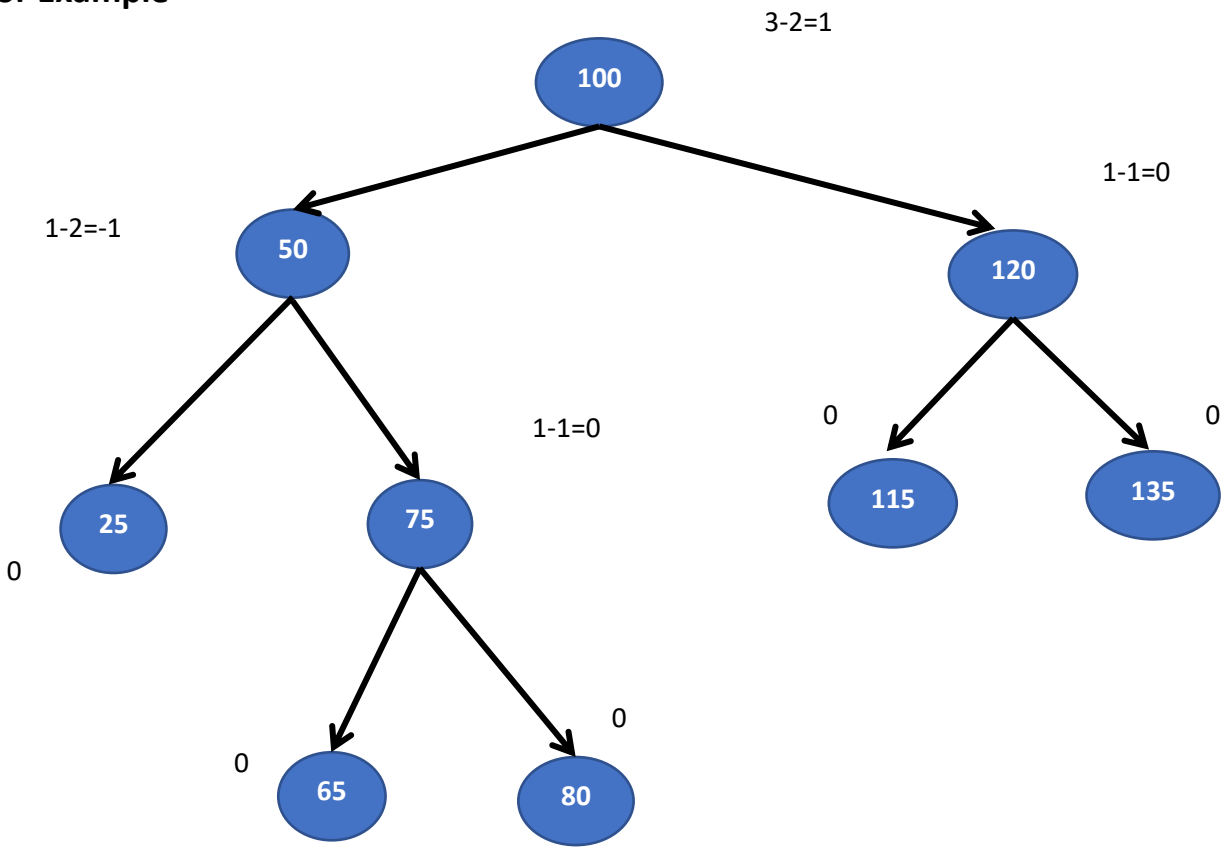
The balance factor of any node in an AVL tree is given by:

Balance factor=Height (left subtree)-Height (right subtree)



Balance factor = (h-1)-(h-2)

For Example



AVL tree rotations:

To balance the AVL tree after the insertions and deletion operations the four kinds of rotations are used. Rotations are an adjustment to the tree, around a node, that maintains the required ordering in the binary search tree.

1. Single Rotation
   a) Left rotation
   b) Right rotation
2. Double Rotation
   a) Left-Right rotation
   b) Right –Left rotation

1. Left Rotation

The tree is rotated left side to rebalance the tree, when a node is inserted into the right subtree of the right subtree. For example if the insertion order is 3, 5 and 7 the tree becomes imbalanced after the insertion of node 7. So we perform Left rotation (rotate in anti- clockwise) to balance the tree.

For example

Right unbalanced tree Left Rotation Balanced tree

2. Right Rotation

The tree is rotated right side to rebalance the tree, when a node is inserted in the left subtree of the left subtree. For Example if the insertion order is 8,6,4 the tree becomes imbalanced after the insertion of node 4 so to balance the tree we perform Right rotation(rotate clockwise).

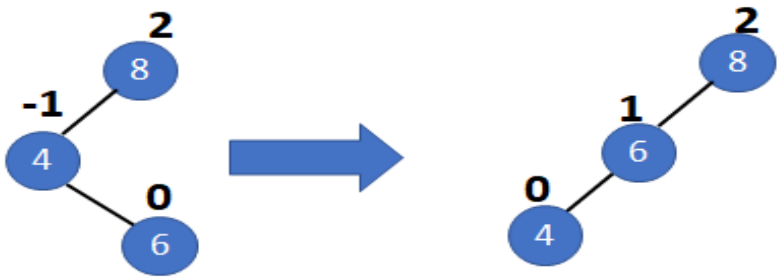
Left unbalanced tree right rotation Balanced tree

3. Left-Right Rotation

In Left-Right Rotation two rotations are performed to balance the tree. The tree is first rotated on left side and then on right side.

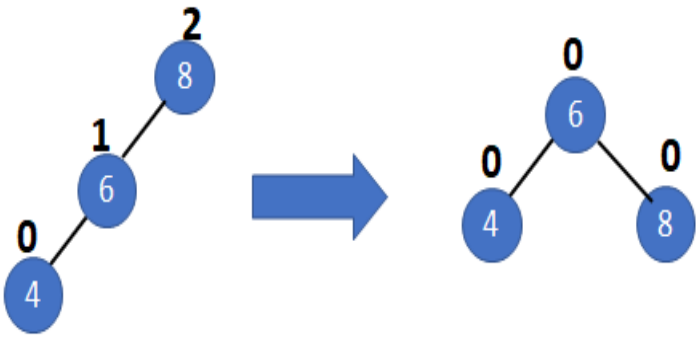
For example in the below unbalanced tree 8 is the root node, 4 is the left child of 8 and 6 is the Right child of 4. After left rotation on 4 and 6 ,the tree

structure looks like as shown below 8 is the root node,6 is the left child of 8 and 4 is the left child of 6



Unbalanced tree Left rotation unbalanced tree

The tree obtained after left rotation is unbalanced so again the tree is rotated right side inorder to balance the AVL tree. So a balanced tree with 6 as the root node,4 will become the left child of 6 and 8 is the right child of 8 is obtained.

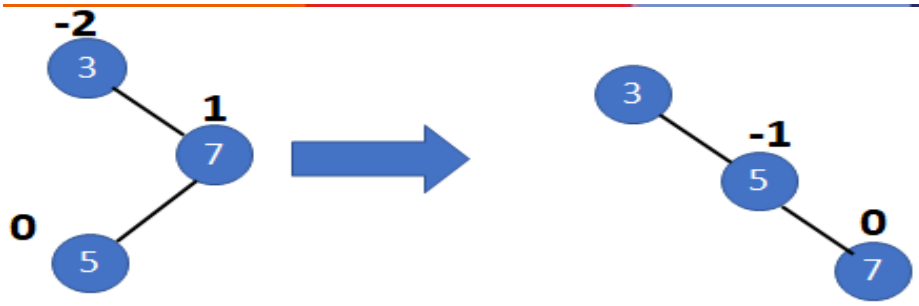


Unbalanced tree Right rotation Balanced Tree

4. Right-Left Rotation

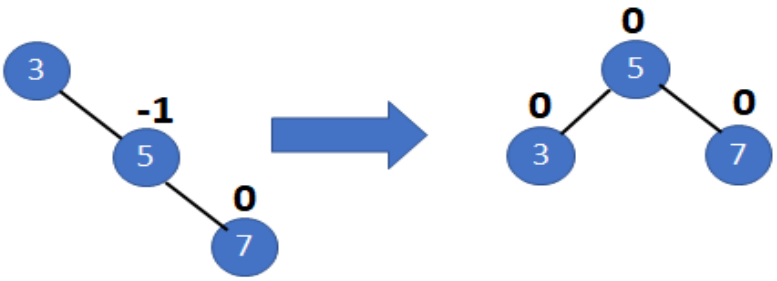
In Right-Left Rotation, first the tree is rotated right side and then to the left side.

For Example in the below unbalanced graph, 3 is the root node 7 is the right child of 3 and 5 is the left child of 7.After applying first right rotation on 7 and 5, then we get a unbalanced tree with 3 as the root node, 5 is the right child of 3 and 7 is the right child of 5.



Unbalanced tree Right rotation unbalanced tree

Then the tree is rotated left side inorder to obtain the balanced tree where 5 is the root node, 3 will be left child of 5 and 7 will be right child of 5.



Unbalanced tree Left rotation Balanced tree

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
N-ary Trees and Forest

Dr. Shylaja S S
Ms. Kusuma K V

Tree: is a Non Linear Data Structure

Definition: Finite nonempty set of elements

- One element is the root
- Remaining elements are partitioned into m≥0 disjoint subsets each of which is itself a tree

Ordered Tree: a tree in which subtrees of each node form an ordered set

- In such a tree we define first, second, ..., last child of a particular node
- First child is called the oldest child and last child the youngest child

Figure 1 shows an ordered tree with A as its root, B is the oldest child and D is the youngest child of A. E is the oldest and F is the youngest child of B.

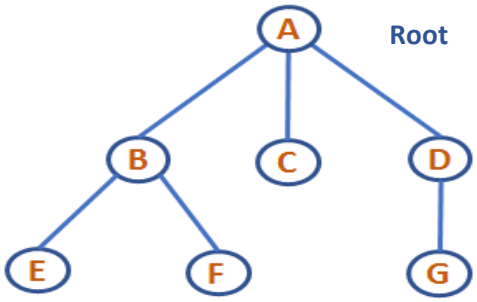


Figure 1: Ordered Tree

Note: Here on we will refer an ordered tree as just a tree

n-ary tree: A rooted tree in which each node has no more than n children.

A binary tree is an n-ary tree with n=2. Figure 2 shows an n-ary tree with n=5.

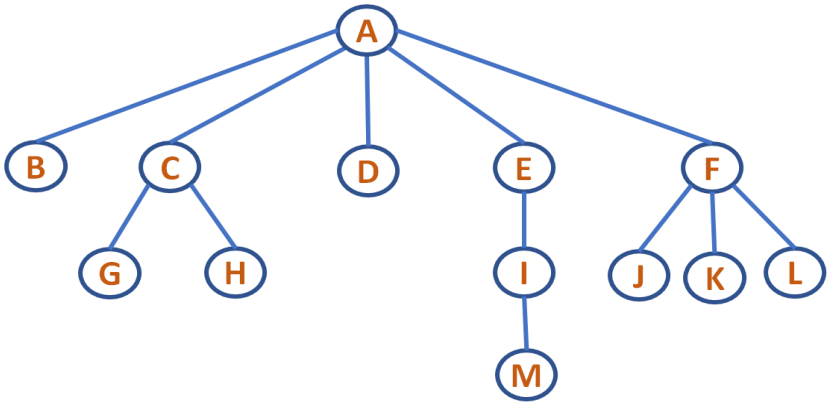


Figure 2: n-ary tree, n=5

Forest: is an ordered set of ordered trees

In the representation of a binary tree, each node contains an information field and two pointers to its two children. Trees may be represented as an array of tree nodes or a dynamic variable may be allocated for each node created. But how many pointers should a tree node contain? The number of children a node can have varies and may be as large or as small as desired.

```
#define MAXCHILD 20
struct treenode{
        int info;
        struct treenode *child[MAX];
};
```

where MAX is a constant

Restriction with the above implementation is that a node cannot have more than MAX children. Therefore the tree cannot be expanded.

Consider an alternative implementation as follows: All the children of a given node are linked and only the oldest child is linked to the parent

A node has link to first child and a link to immediate sibling

```
struct treenode{
        int info;
        struct treenode *child;
        struct treenode *sibling;
};
```

Conversion of an n-ary tree to a Binary Tree

Using Left Child – Right Sibling Representation an n-ary tree can be converted to a binary tree as follows:

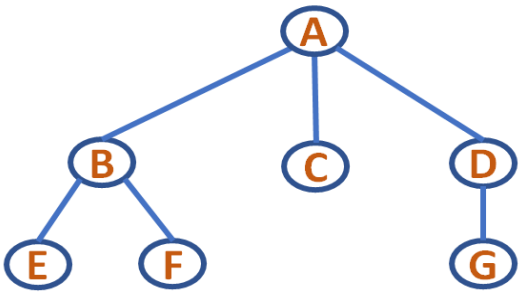
1) Link all the siblings of a node

2) Delete all links from a node to its children except for the link to its leftmost child

The **left child in binary tree** is the node which is the oldest child of the given node in an n-ary tree, and the **right child is the node to the immediate right of the given node** on the same horizontal line. Such a binary tree will not have a right sub tree.

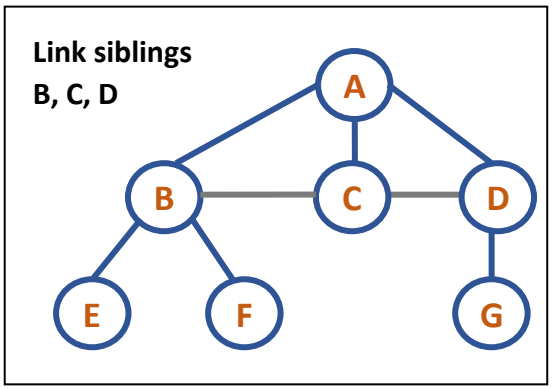
The node structure looks as shown below:

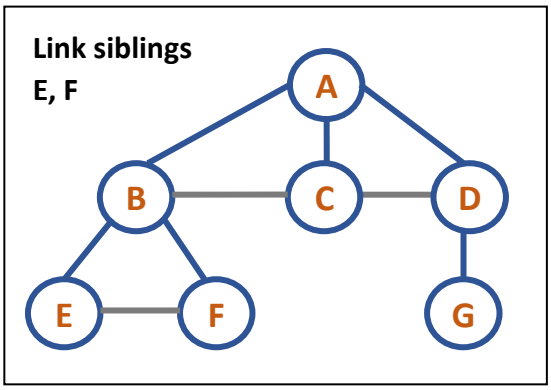
Consider the 3-ary tree shown below:



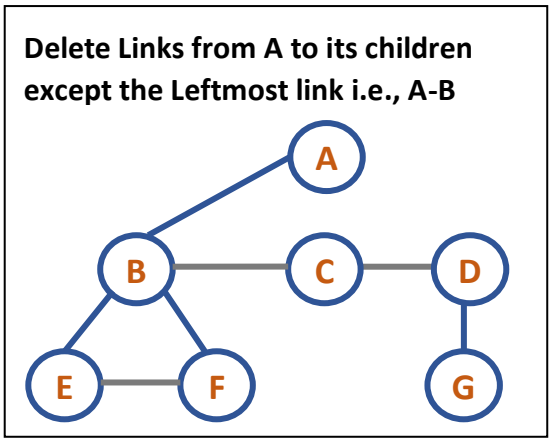
3 - ary tree

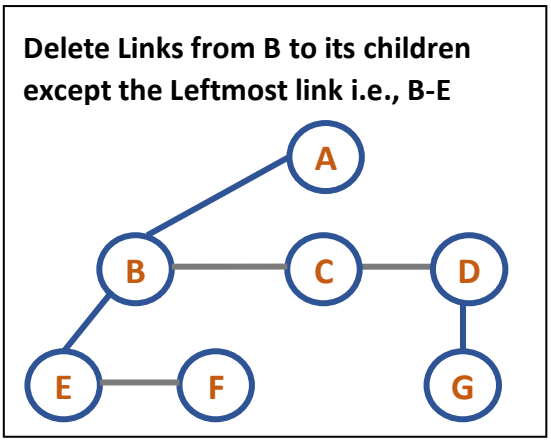
First step in the conversion is to Link all the siblings of a node.





Second step is to delete all the links from a node to its children except for the link to its leftmost child.

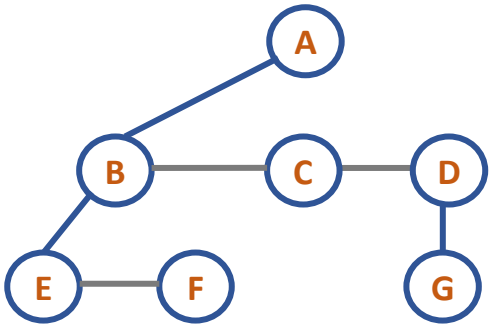




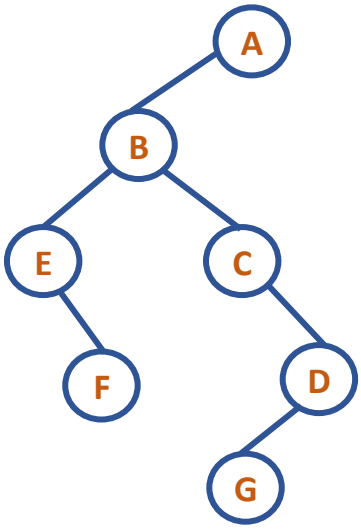
There are no more multiple links from any parent node to its children. The tree so obtained is the binary tree. But it doesn't look like one. Use the left child-right sibling relationship and make the tree look like a binary tree i.e., for any given node, its leftmost child will become the left child and immediate sibling becomes the right child. The binary tree so obtained will always have an empty right subtree for the root node. This is because the root of the tree we are transforming has no siblings.

For node A: Left child is B and has no Right child
For Node B: Left child is E and Right child is C
For Node C: No Left child and Right child is D
For node D: Left child is G and has no Right child
For Node E: No Left child and Right child is F
For Node F and Node G: No children (No Left child: because they do not have a child and no Right child: because they do not have a sibling towards right)

Left Child – Right Sibling representation of the given 3-ary tree



Corresponding Binary Tree representation of the given 3-ary tree



Conversion of a Forest to a Binary Tree

In the n-ary tree to binary tree conversion as stated above we saw that the right subtree for the root node of the binary tree is always empty. This is because the root of the tree we are transforming has no siblings.

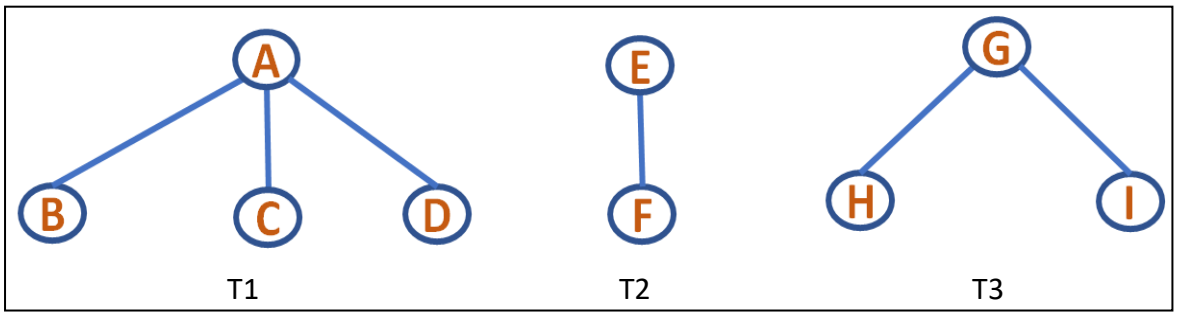
On the other hand, if we have a forest then these can all be transformed into a single binary tree as follows:

1) First obtain the binary tree representation of each of the trees in the forest
2) Link all the binary trees together through the right sibling field of the root nodes

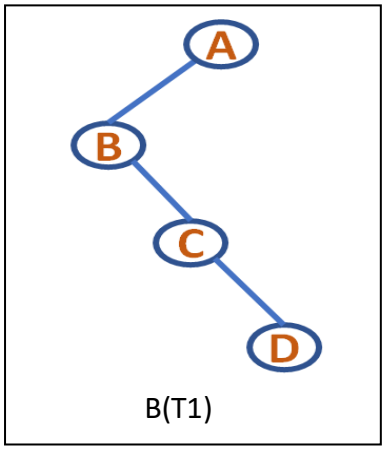
Conversion of a Forest to a Binary Tree can be formally defined as follows:

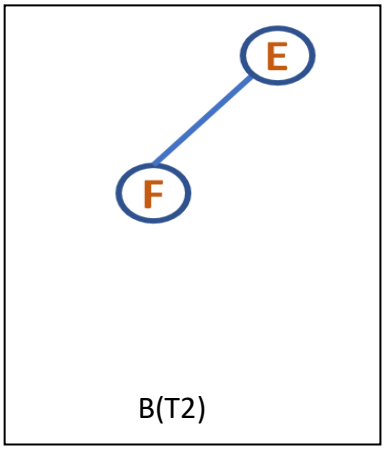
- If $T_1,...,T_n$ is a forest of n trees, then the binary tree corresponding to this forest, denoted by $B(T_1, ..., T_n)$:
  - is empty if $n = 0$
  - has root equal to root (T_1)
  - has left subtree equal to $B(T_{11}, T_{12}, ...,T_{1m})$ where T_{11} ,..., T_{1m} are the subtrees of root(T_1)
  - has right subtree $B(T_2, ..., T_n)$

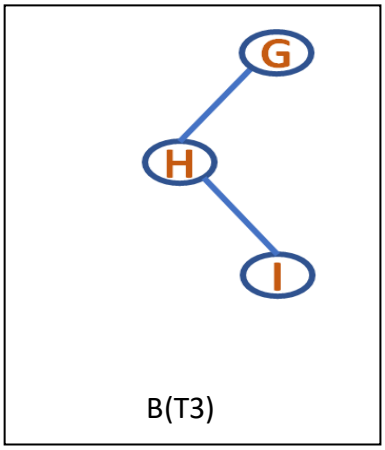
Consider the following Forest with three Trees:

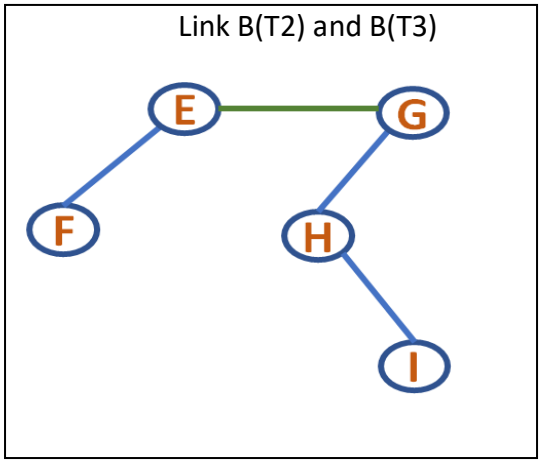


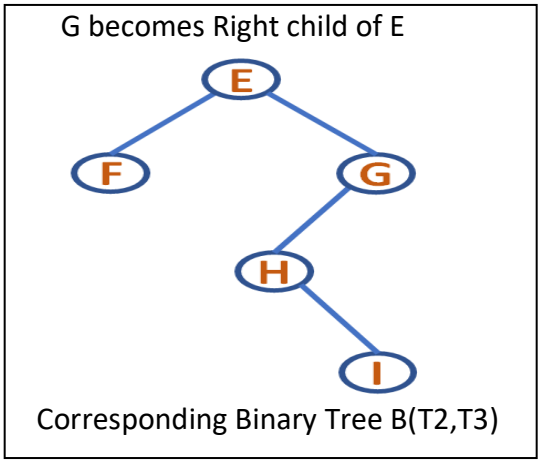
Corresponding Binary Trees:

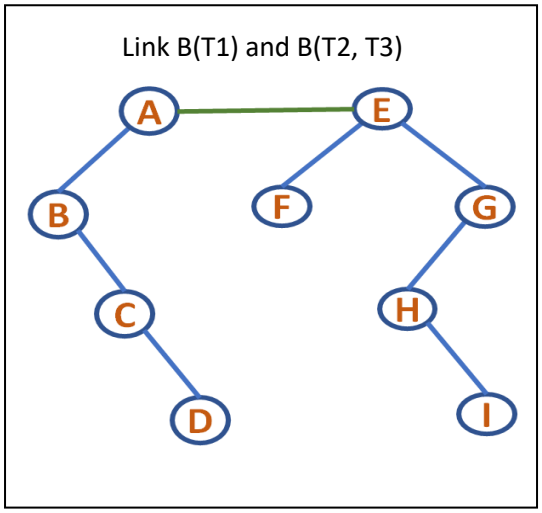


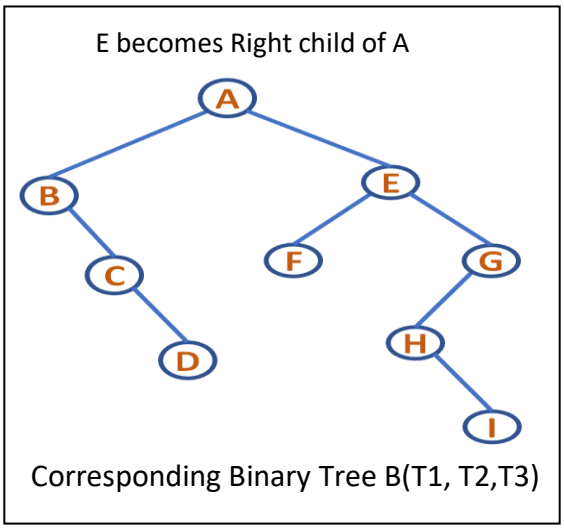












Tree Traversal

The traversal methods for binary trees induce traversal methods for forests. The preorder, inorder, or postorder traversals of a forest may be defined as the preorder, inorder, or postorder traversals of its corresponding binary tree.

```
struct treenode{
        int info;
        struct treenode *child;
        struct treenode *sibling;
};
```

With the treenode implemented as having pointers to first child and immediate sibling, the traversal preorder, inorder and postorder for a tree are defined as below:

Preorder:

1. Visit the root of the first tree in the forest
2. Traverse in preorder the forest formed by the subtrees of the first tree, if any
3. Traverse in preorder the forest formed by the remaining trees in the forest, if any

```
void preorder(TREE *root)
{
   if(root!=NULL)
   {
     printf(" %d ",root->info);
     preorder(root->child);
     preorder(root->sibling);
  }
}
```

Inorder:

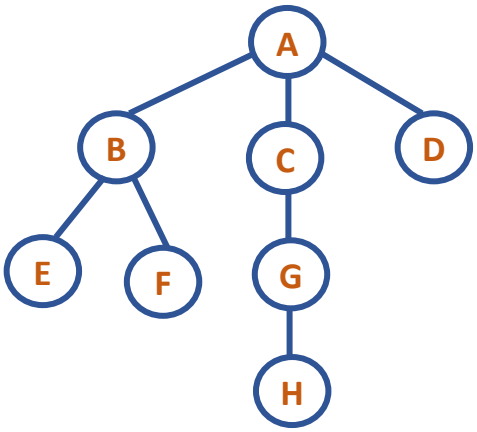
1. Traverse in inorder the forest formed by the subtrees of the first tree, if any
2. Visit the root of the first tree in the forest
3. Traverse in inorder the forest formed by the remaining trees in the forest, if any

```
void inorder(TREE *root)
{
  if(root!=NULL)
    {
      inorder(root->child);
      printf(" %d ",root->info);
      inorder(root->sibling);
    }
}
```

Postorder:

1. Traverse in postorder the forest formed by the subtrees of the first tree, if any
2. Traverse in postorder the forest formed by the remaining trees in the forest, if any
3. Visit the root of the first tree in the forest

```
void postorder(TREE *root)
{
  if(root!=NULL)
    {
      postorder(root->child);
      postorder(root->sibling);
      printf(" %d ", root->info);
    }
}
```



Traversal of the above n-ary Tree:

Preorder: ABEFCGHD

Inorder: EFBHGCDA

Postorder: FEHGDCBA

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPHS

Abstract

Introduction, Properties, Representation of graphs: Adjacency Matrix, Adjacency List

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Overview:

Graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of edges. We will often denote n = |V|, e = |E|. It consists of set of nodes and arcs. Each arch in a graph is specified by a pair of nodes.

There are three types of graph

1. Directed graph

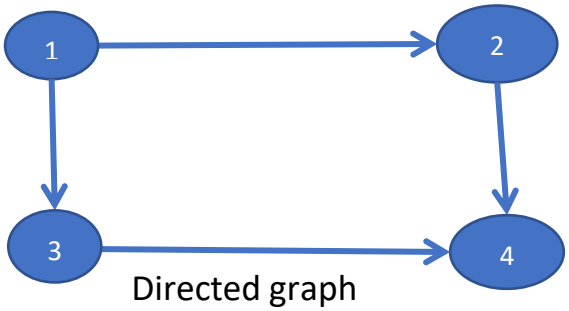
2. Undirected graph

3. Weighted graph

1. Directed graph: A graph G= (V, E) in which every edge is directed is called directed graph. In directed graph pair of vertices representing any edge is ordered. For example in the below fig < v_1, v_2 > and < v_2, v_1> represents the different edge. Directed graph is also known as digraph.

Examples: 1

2.



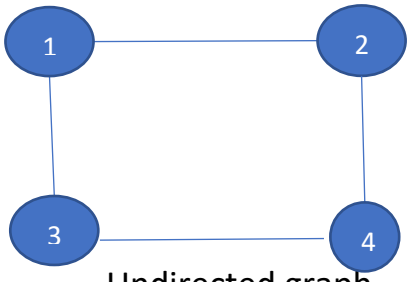
Directed graph

In the above Example.2 V={1,2,3,4} is set of vertices and E={<1,2>,<2,4>,<3,4>,<1,3>} is set of edges. Since all the edges are directed it is a directed graph

2. Undirected graph: The graph G= (V, E) in which every edge is undirected .In undirected graph the pair of vertices representing any edge is unordered.
Example1: The < v_1, v_2 > and < v_2, v_1> represents the same edge.

Example.1

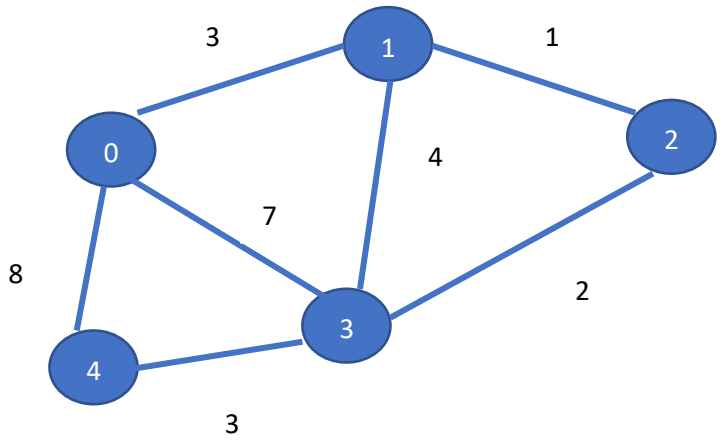
Example.2



Undirected graph

In the above example.2 V={1,2,3,4} is the set of vertices and E={(1,2),(1,3),(2,1),(2,4),(4,2),(3,4),(4,3),(3,1)} is the set of edges. Since all the edges are undirected the above graph is undirected graph

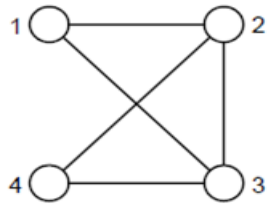
3. Weighted graph or Network: A weighted graph is a graph where each edge has a numerical value called weight. Given below is an example of a weighted graph. Weighted graph can be directed and undirected.



Weighted graph

Graph terminologies:

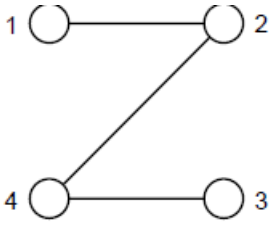
Adjacent Nodes: When there is an edge from one node to another then these nodes are called adjacent nodes.



Adjacent Node

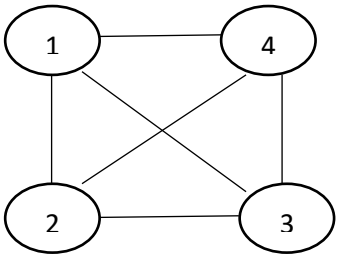
In the above graph 2 is adjacent to 3 and 4 is adjacent to 3 and so on

Path: A path from edges u_0 to a node u_n is a sequence of nodes u_0, u_1, u_2.....u_{n-1},u_n. Here u_0 is adjacent to u_1, u1 is adjacent to u_2 and u_{n-1} is adjacent to u_{n-1}. In the below graph, the path from vertex 1 to 3 is denoted by (1,2,4,3) which can also be written as (1, 2), (2, 4), (4, 3).



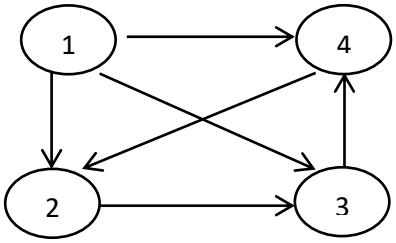
Path

Length of the path: Length of the path is the total number of edges included in the path from the source node to destination node.



Undirected graph

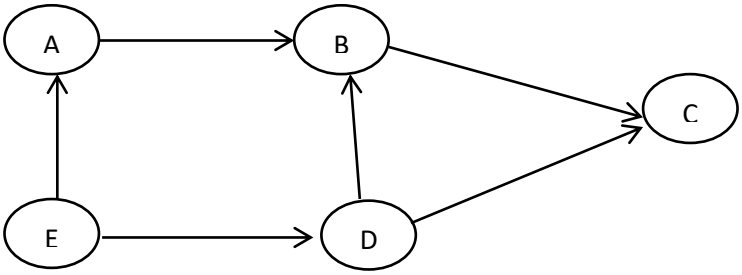
For example in the above undirected graph ,the path(1,2,3,4) has length 3 since there are three edges(1,2),(2,3),(3,4). The path 1, 2, 3 has length 2 since there are 2 edges (1, 2), (2, 3).



Directed graph

Similarly in the directed graph, the path <1,2,3,4> has length 3 since there are 3 edges <1,2>,<2,3>,<3,4> and the path <1,2,3> has length 2 since there are 2 edges <1,2>,<2,3>.

Degree: In an undirected graph, the total number of edges linked to a node is called degree of the node. In digraph there are 2 degrees for every node called indegree and outdegree.

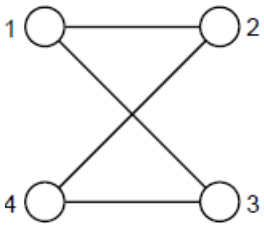


Degree

In-degree: The in-degree of a node n is the total number of arcs that have n as the head. For example C is receiving 2 edges so indegree of C is 2.

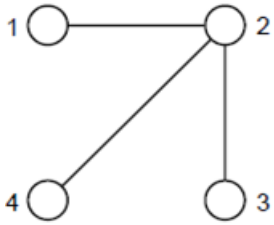
Outdegree: The outdegree of a node n is the total number of arcs that have n as the tail. For example outdegree of D is 1.

Cycle graph: A path from node to itself be called a cycle or cycle is path in which first and last vertices are same. A graph with at-least one cycle is called cyclic graph. A directed acyclic graph is called dag. For example the path<1, 2, 4, 3, and 1> is cycle since the first node and the last node are same. It can be represented as <1,2>,<2,4>,<4,3>,<3,1>.



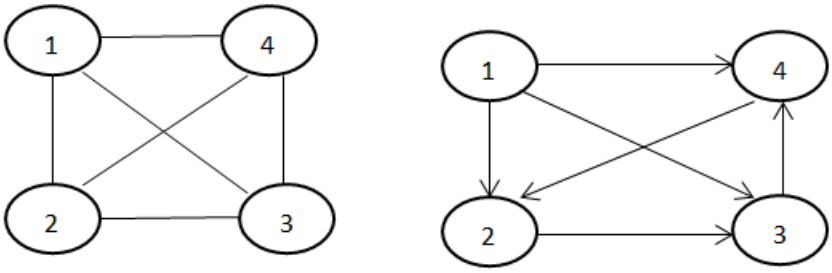
Cycle

Acyclic graph: A graph with no cycle is called acyclic graph. Tree is an acyclic graph.



Acyclic graph

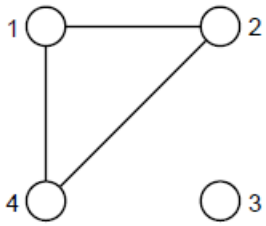
Connected: A graph is called connected if there is a path from any vertex to any other vertex. A tree is defined as connected undirected graph with no cycles. Example



Connected graph

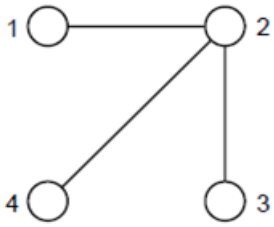
Disconnected graph: if there exist atleast one vertex in a graph that cannot be reached from the other vertices in the graph,then such graph is aclled

disconnected graph.In the below example vertex 3 cannot be reached by any other vertices in the graph.



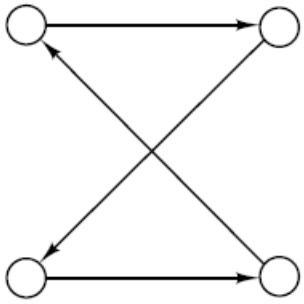
Disconnected graph

Tree: A tree is defined as connected undirected graph with no cycles.



Tree

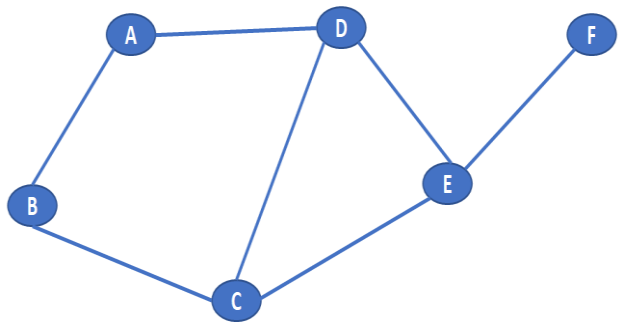
Directed cycle: In directed graph all the edges in a path or cycle have same direction



Directed cycle

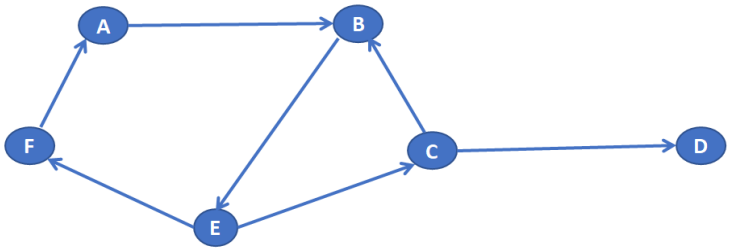
Properties of graph (referred from geeks for geeks):

1. **Undirected graph**
   a. The number of possible pairs in an m vertex graph is m*(m-1).
   b. The number of edges in an undirected graph is m*(m-1)/2 since the edge (u, v) is same as the edge(v, u).



Undirected graph

2. **Directed graph**
   a. The number of possible pairs in an m vertex graph is m*(m-1)
   b. The number of edges in an directed graph is m*(m-1) since the edge(u, v) is not the same as the edge(v, u)
   c. The number of edges in an directed graph is<=m*(m-1)



Directed graph

Representation of graph

Graph representation is the method to store the graphs in computer memory. Graph is represented using a set of vertices, and for each vertex, the vertices are

directly connected to it by an edge. For a weighted graph weight will be associated with each edge. The choice of graph representation depends on the application and the types of operation performed and ease of use.

There are 2 ways of representing the graph

1. Adjacency matrix.
2. Adjacency List.

1. Adjacency Matrix:

Let G= (V, E) be a graph where V is the set of vertices and E is the set of edges. Let N be the number of vertices in the graph G. The adjacency matrix A of a graph G is defines as

A[i][j] = 1 if there is an edge from vertex i to j

0 if there is no edge from vertex i to j

The adjacency matrix of a graph is a Boolean square matrix or bit matrix with n rows and n columns with entries zeros and ones.

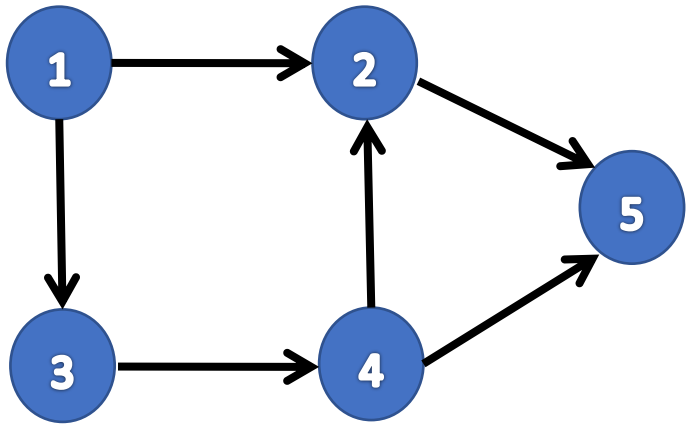
In an undirected graph, if there exist an edge (i, j),then A[i][j] and A[j][i] is made one since (i, j) is similar to (j, i).

In the directed graph if there exist an edge <i,j>, then A[i][j]=1

If there exist no edge between vertex i and j then A[i][j]=0

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

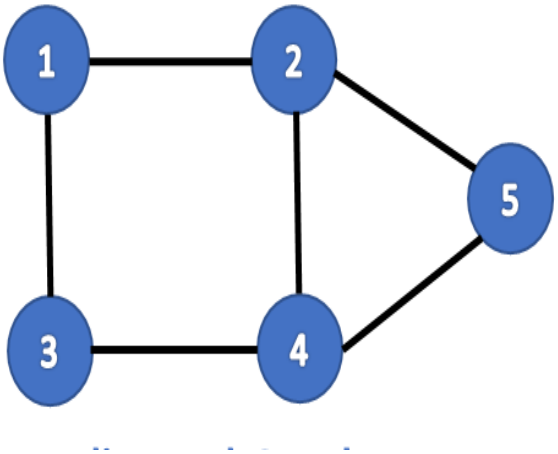
1. Directed graph



Directed graph

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	0	1	0	0	1
5	0	0	0	0	0

2. Undirected graph

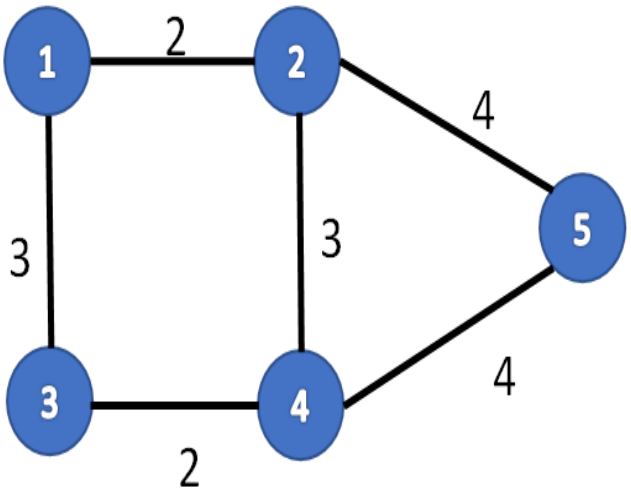


Undirected graph

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	1
3	1	0	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

3. Weighted graph:

In the weighted graph, distance or cost between the nodes are represented on the edge cost/distance value specified on the edge between adjacent nodes are used for representation in the adjacency matrix.



Weighted graph

	1	**2**	**3**	**4**	**5**
1	0	2	3	0	0
2	2	0	0	3	4
3	3	0	0	2	0
4	0	3	2	0	4
5	0	4	0	4	0

C representation of graph:

The number of nodes in the graph is constant, that is arcs may be added or deleted but the nodes may not. A graph with 25 nodes could be declared as

A graph with 25 nodes can be declared as

```
#define MAX 25

struct node

{

                    //information associated with each node

};

struct arc

{

                    int adj;//information associated with each arc

};

struct graph

{

                    struct node nodes[MAX];

                    struct arc arcs[MAX][MAX];

};

struct graph g;
```

Each node in a graph is represented by an integer number from zero to MAX-1, and the array field nodes represent the appropriate information assigned to each node. The array field an arcs is a two dimensional array representing every possible ordered pair of nodes. The value of g.arcs[i][j].adj is either one or zero depending on whether a node i is adjacent to j.

A weighted graph with fixed number of nodes can be declared as

```
struct arc
{
                    int adj;
                    int weight;
};
struct arc g[MAX][MAX];
```

Drawback of adjacency matrix representation of graph is it requires advance knowledge of the number of nodes. A new matrix must be created for each addition and deletion of a node. Adjacency matrix representation is inefficient in real world situation where a graph may have hundreds of nodes. An alternate solution to this is to use adjacency list.

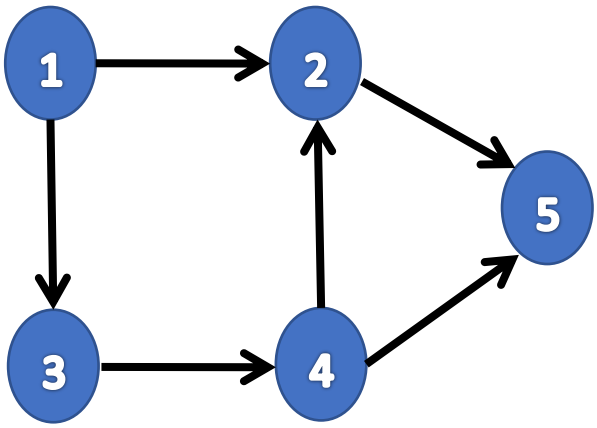
Function to read adjacency matrix

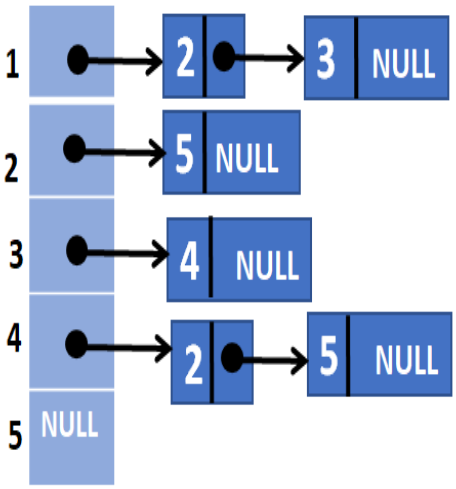
```
void read_admat(int A[10][10], int n)
{
    int I,j;
     for(i=0; i<n; i++)
    {
          for(j=0; j<n; j++)
           {
              scanf("%d", &A[i][j]);
            }
     }
}
```

2. Adjacency Linked List:

Adjacency linked list is an array of n linked list in which every vertex of a graph contains the list of adjacent vertices

1. Directed graph





Nodes adjacent to 1 are 2 and 3

Nodes adjacent to 2 is 5

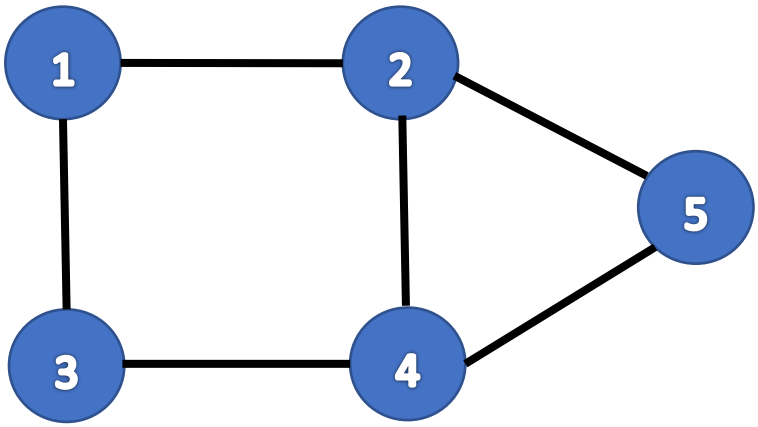
Nodes adjacent to 3 is 4

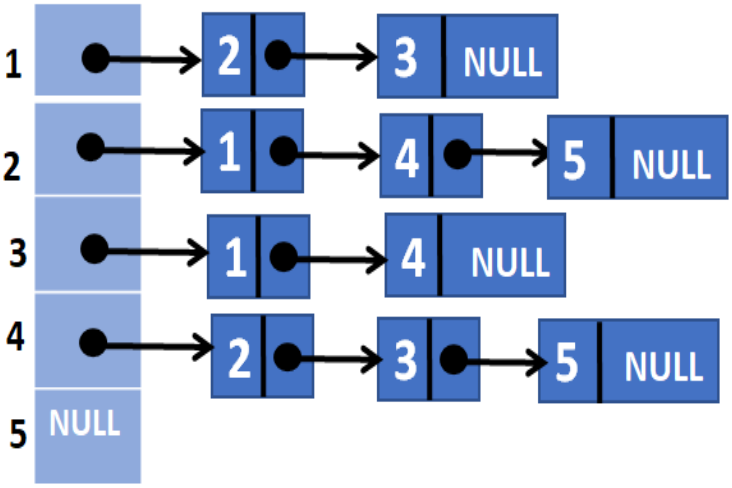
Nodes adjacent to 4 is 2 and 5

No Nodes adjacent to 5

Directed graph

2. Undirected graph





Nodes adjacent to 1 are 2 and 3

Nodes adjacent to 2 are 1, 4 and 5

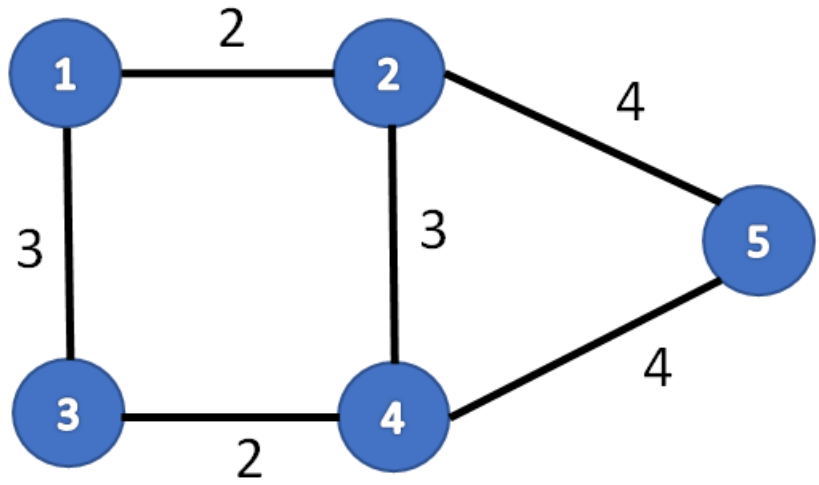
Nodes adjacent to 3 are 1 and 4

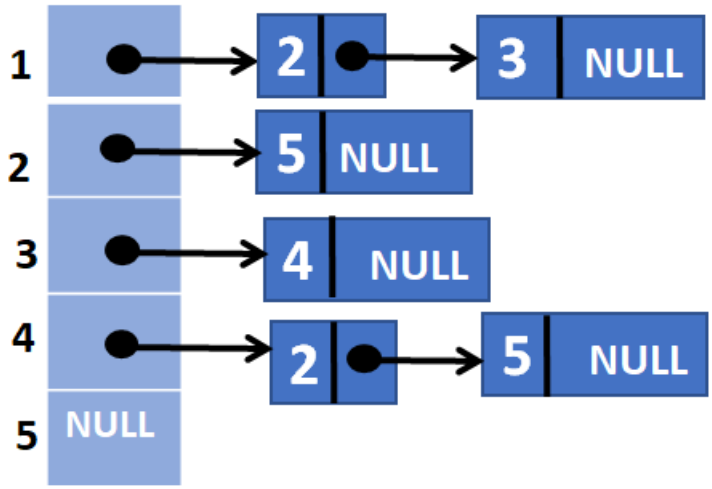
Nodes adjacent to 4 are 2, 3 and 5

No Nodes adjacent 5

Undirected graph

3. Weighted graph





Weighted graph

C representation of adjacency list

Two types of allocated nodes are needed since each graph carries some information.(arcs does not carry any info).

1. header nodes

arcptr	info	nextnode

A Sample header node

Each header node contains an info field and two pointers. The first of these to is to adjacency list of arcs emanating from the graph node, and the second is to next header node in the graph.

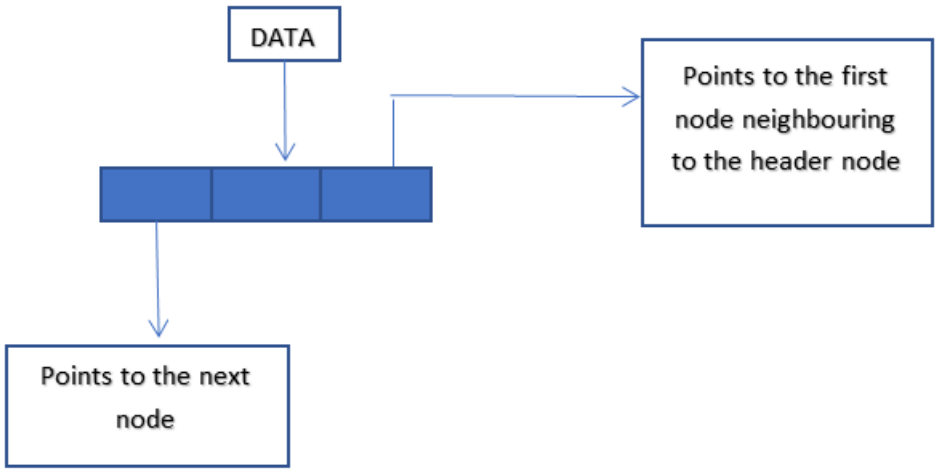
2. Adjacency list nodes

Ndptr	nextarc

A sample list node representing an arc

Each arc node contains two pointers, one for nextarc node in adjacency list and the other to the header node representing the graph that terminates the arc. Refer to the diagram from textbook T1 8.3.1 page no 543.

Header nodes and list nodes have different formats and must be represented by different structures. However for simplicity we make assumption that both header and the list nodes have same format and contain two pointers and a single integer information field.



1. Using array implementation the nodes are declared as

```
#define MAX 50

struct node

{

    int info;

     int point;

      int next;

};

struct node NODE[MAX];
```

In the case of header node, node[p] represents a graph node A, node[p].info represents the information associated with the graph node, node[p].next points to the next graph node and node[p].point points to the first list node representing an arc emanating from A. In the case of a list node, node[p] represents an arc <A, B>, node[p].info represents the weight of arc, node[p].point points to the header node representing the graph node B.

2. Using dynamic implementation

```
struct node

{

    int info;

    strut node *pointer;

    struct node *next;

};

struct node *nodeptr;
```

Function to read the adjacency list

```
void read_adlist(nodeptr a[],int n)
{
    int I,j,m ele;
    for(i=0;i<n;i++)
     {
        printf("enter the number of nodes adjacent to %d:",i);
         scanf("%d",&m);
         if(m==0) continue;
         printf("Enter the nodes adjacent to %d",i);
        for(j=0;j<m;j++)
        {
         scanf("%d",&ele);
        a[i]=insert_rear(ele,a[i]); //Function to insert an element at the rear of the list
         }
    }
}
```

Advantages of using Adjacency list:

1. Adding a new vertex is easy

2. Graphs can be stored in more compact form

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPHS

Abstract

Implementation of Graph using Adjacency Matrix

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Implementation of graph using adjacency matrix

C representation of graph:

A graph with 25 nodes can be declared as

```
#define MAX 25
struct node
{
                    //information associated with each node
};
struct arc
{
                    int adj;//information associated with each arc
};
struct graph
{
    struct node nodes[MAX];
    struct arc arcs[MAX][MAX];
};
struct graph g;
```

Each node in a graph is represented by an integer number from zero to MAX-1, and the array field nodes represent the appropriate information assigned to each node. The array field an arc is a two dimensional array representing every possible ordered pair of nodes.

The different operations performed on adjacency matrix are

1. To add an arc from node1 to node2

```
void join(int adj[]MAX],int node1,int node2)
 {
    adj[node1][node2]=TRUE;
}
```

2. To delete an arc from node1 to node2 if it exists

```
void remv(int adj[][MAX],int node1,int node2)
{
  adj[node1][node2]=FALSE;
}
```

3. To check whether arc exists between node1 and node2

```
int adjacent(int adj[][MAX],int node1,int node2)
{
   return((adj[node1][node2]==TRUE)?TRUE:FALSE);
}
```

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPHS

Abstract

Implementation of Graph using Adjacency List.

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Implementation of adjacency List:

C representation of adjacency list

1.using array implementation

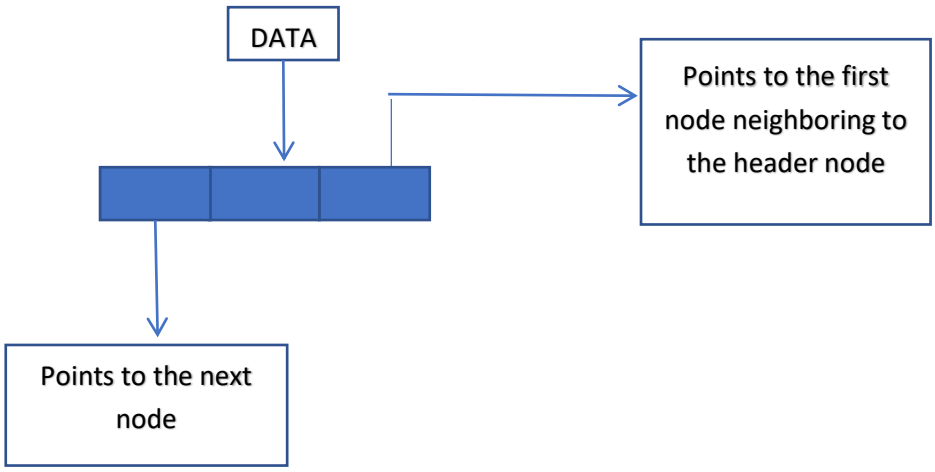
```
#define MAX 50
struct node
{
  int info;
  int point;
  int next;
};
struct node NODE[MAX];
```

2.Using dynamic implementation

```
struct node
{
   int info;
  strut node *pointer;
   struct node *next;
};
struct node *nodeptr;
```

Two lists are maintained for adjacency of list. The first list is used to store information of all the nodes. The second list is used to store the information of adjacent nodes for each node of a graph.

Header node is used to represent each list, which is assumed to be the first node of a list as shown below.



The different operations performed on adjacency list are

1. To create an arc between two nodes

```
void jointwt(int p,int q,int wt)
    {
        int r,r2;
        r2=-1;
       r=node[p].point;
       While(r>=0 && node[r].point!=q)
        {
            r2=r;
            r=node[r].next;
         }
       If(r>=0)
        {
            node[r].info=wt;
```

```
                    return;
                }
            R=getnode();
            node[r].point=q;
            node[r].next=-1;
             node[r].info=wt;
             (R2<0)?(node[p].point=r):(node[r2].next=r);
}
```

2. To remove an arc between two nodes if it exists:

```
Void remv(int p,int q)
{
    int r,r2;
   r2=-1;
   r=node[p].point;
  while(r>=0 ** node[r].point !=q)
 {
     r2=r;
    r=node[r].next;
}
If(r>=0)
{
   (r2<0)?(node[p].point=node[r].next):(node[r2].next=node[r].next);
   freenode(r);
   return;
}
```

```
}
```

3. Checks whether a node is adjacent to another node

```
int adjacent(int p,int q)
{
   int r;
    r=node[p].point;
   while(r>=0)
        If(node[r].point==q)
            return(TRUE)
         else
            r=node[r].next;
              return(FALSE);
}
```

4. Find a node with a specific information in a graph:

```
int findnode(int graph,int x)
{
        int p;
        p=graph;
        while(p>=0)
                if (node[p].info==x)
                        return(p);
                else
                        p=node[p].next;
```

```
        return(-1);
}
```

5. Add a node with specific information to a graph.

```
int addnode(int *graph,int x)
{
    int p;
    p=getnode();
    node[p].info=x;
    node[p].point=-1;
    node[p].next=*graph;
    *graph=p;
    return(p);
}
```

PES UNIVERSITY

Department of Computer science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

GRAPH TRAVERSAL

Abstract

Traversals, Breadth first search, Depth first search, Implementation, Advantages, Disadvantages and Applications

Dr.Sandesh and Saritha

Sandesh_bj@pes.edu

Saritha.k@pes.edu

Graph traversal:

Many graph algorithms requires a systematic way to examine all the nodes and edges of a graph. Traversing is a process of visiting each and every node of a graph.

Defining a traversal that relates to the structure of a graph is more complex than for a list or tree due to following reasons

1. A graph has no first node or root node. Therefore the graph can be traversed from any node as starting node and once the starting node is selected there may be few nodes which are not reachable from the starting node. Traversal algorithm faces the problem of selecting another starting point.

2. In a graph to reach a particular node multiple paths may be available.

3. There is no natural order among the successors of particular node. Thus there is no prior order in which the successors of a particular node should be visited.

4. A graph may have more than one predecessor and therefore there is a possibility for a node to be visited before one of its predecessors. For example if node A is a successor of nodes B and C, A may be visited after B but before C.

There are two standard algorithms for traversal of graphs. They are:

• Breadth first search (BFS): The BFS will use a queue as an auxiliary structure to hold nodes for future processing

• Depth first Search (DFS): DFS algorithm traverses a graph in a depth-ward motion and uses a stack to remember to get the next vertex to start a search, when dead end occurs in any iteration.

Depth First Search:

In Depth first search method, an arbitrary node of a graph is taken as starting node and is marked as visited. On each iteration, the algorithm proceeds to an unvisited vertex which is adjacent to the one which is currently in. Process continues until a vertex with no adjacent unvisted vertex is found. When the dead end is reached the control returns to the previous node and continues to visit all the unvisited vertices. The algorithm ends after backing up to the starting vertex, when the latter being a dead end. The algorithm has to restart at an arbitrary vertex, if there still remain unvisited vertices.

This process is implemented using stack. A vertex is inserted into the stack when the vertex is visited for the first time and deletes a vertex when the visit of the vertex ends.

Iterative procedure to traverse a graph is shown below:

1. Select a node u as a start vertex, push u onto the stack and mark it as visited.

2. While stack is not empty

    For vertex u on top of the stack, find the next immediate adjacent vertex

    If v is adjacent

        If a vertex v is not visited then

            Push it on to stack and mark it as visited

        else

            Ignore the vertex

        End if

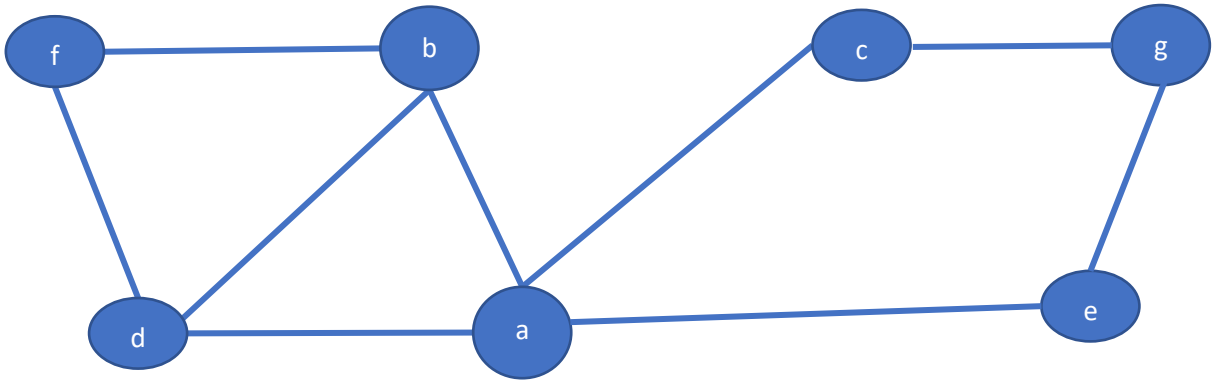
    Else

        Remove the vertex from the stack

    End if

   End while

3. Repeat step1 and step2 until all the vertices in the graph are visited.



	Stack	**V(adjacent vertex)**	**Nodes visited(S)**	**Pop(stack)**
Initial step	a	-	a	
	a	b	a,b	-
	a,b	d	a,b,d	-
	a,b,d	f	a,b,d,f	-
	a,b,d,f	-	a,b,d,f	f
	a,b,d	-	a,b,d,f	d
	a,b	-	a,b,d,f	b
	a	c	a,b,d,f,c	-
	a,c	g	a,b,d,f,c,g	-
	a,c,g	e	a,b,d,f,c,g,e	-
	a,c,g,e	-	a,b,d,f,c,g,e	e
	a,c,g	-	a,b,d,f,c,g,e	g
	a,c	-	a,b,d,f,c,g,e	c
	a	-	a,b,d,f,c,g,e	a

Step 1: Insert a source vertex *a* onto the stack and add *a* to node visited(S)
Step 2: Push the node adjacent to *a* that is *b* onto the stack (alphabetical order) and add to S if not present in S (Also note that in DFS only one adjacent vertex which is not visited is considered).
Step 3: push the node adjacent to *b*, that is *d* onto stack and add to S if not present.
Step 4: push the node adjacent to *d* that is *f* onto the stack and add to S if not visited.
Step 5: Since the node adjacent to *f* is already visited. Pop *f* from the stack.
Step 6: Since the node adjacent to *d* is already visited, pop *d* from the stack.
Step 7: Pop *b* from the stack.
Step 8: Since there is adjacent node from *a* to *c* we add c to stack and S
Step 9: Push the node adjacent to *c* that is *g* to stack and add to S.
Step 10: push the node adjacent to *g* that is e to stack and add to S.
Step 11: Since the node adjacent to *e* is visited, pop *e* from the stack.
Step 12: Since the node adjacent to *g* is visited, pop *g* from stack
Step 13: Since the node adjacent to *c* is visited, pop *c* from stack.
Step 14. Pop *a* from stack
Thus nodes reachable from *a* are *a, b, c, d, e, f, g.*

Advantages of Depth First Search:

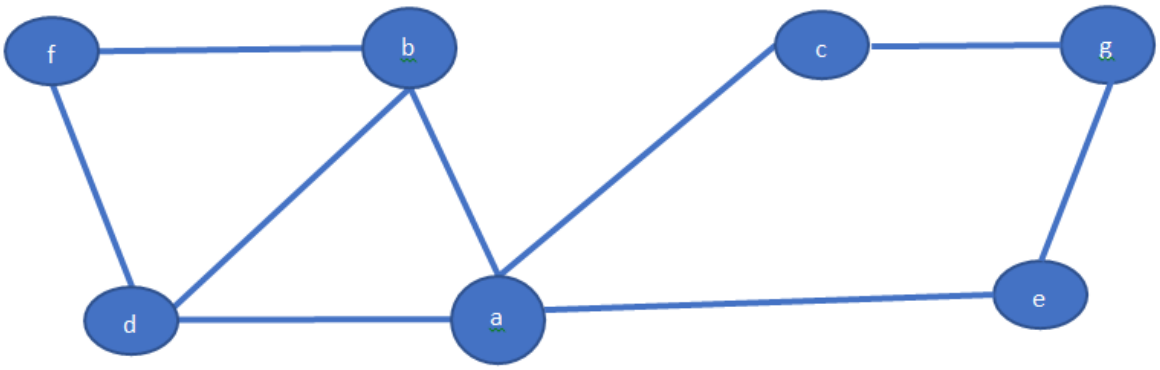
1. Consumes less memory
2. Finds the larger distant element from source vertex in less time.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever. To eliminate this we keep a list of states previously visited, and never permit search to return to any of them.
2. We cannot come up with shortest solution to the problem.

Breadth First Search: Breadth first search uses the queue data structure for traversing vertices of the graph. Any node of the graph can act as the starting node. Using the starting node, all other nodes of the graph are traversed. To prevent repeated visit to the same node, an array is maintained which keeps track of visited node.

In breadth first search algorithm the queue is initialized with the starting vertex and is marked as visited. On each iteration, the algorithm identifies all the unvisited vertices adjacent to the front vertex, marks them as visited and adds them to the queue; after that ,the front vertex is removed from the queue.



	U(deleted from queue)	V(adjacent vertex)	Nodes visited(s)	Queue
Initial node			a	a
	a	b,c,d,e	a,b,c,d,e	b,c,d,e
	b	a,d,f	a,b,c,d,e,f	c,d,e,f
	c	a,g	a,b,c,d,e,f,g	d,e,f,g
	d	a,b,f	a,b,c,d,e,f,g	e,f,,g,
	e	a,g	a,b,c,d,e,f,g	f,g
	f	b,d	a,b,c,d,e,f,g	g
	g	c,e	a,b,c,d,e,f,g	empty

Step 1: Insert a source vertex *a* into the queue and add *a* to node visited(S).

Step 2: Delete an element a from queue and find all the adjacent nodes of .The nodes adjacent to *a* are *b, c, d* and *e*. Add these nodes to S only if it is not present in S. So we add *b, c, d* and *e* to S

Step 3: Delete b from queue and find all the adjacent nodes to *b* i.e. *a, d, f* and add to S only if it not present in S, so only *f* is added to S.

Step 4: Delete *c* from queue, Find all the adjacent nodes to *c*, and add those nodes to S if it is not present in S. So add only *g* to S.

Step 5: Delete *d* from queue, Find all the adjacent nodes of *d* and add those to S if not present. All the adjacent nodes of *d* is already in S.

Step 6: Delete *e* from queue, find all the adjacent nodes of *e*. i.e *a* and *g* .Since *g* is not in S we add *g* to S.

Step7: Delete *f* from queue; find all the nodes adjacent to *f.* add to S if not present in S

Step 8: Delete *g* from queue, find all the adjacent nodes to *g*. Since all the adjacent nodes of *g* are already present in S we don't add to S.

And the queue is empty so the nodes reachable from Source *a*: *a, b, c, d, e, f, g*

Pseudo code:

```
Insert source u to queue
Mark u as visited
While queue is not empty
        Delete a vertex u from queue
        Find the vertices v adjacent to u
        If v is not visited
            Print v
            Mark v as visited
            Insert v to queue
        End if
```

End while

Advantages of Breadth First Search:

1. BFS algorithm is used find the shortest path between vertices
2. Always finds optimal solutions.

Disadvantages of BFS:

All of the connected vertices must be stored in memory. So consumes more memory