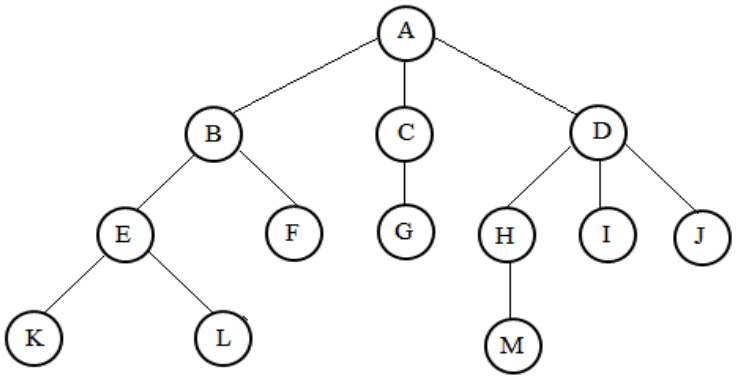
MODULE 4: TREES

DEFINITION

A *tree* is a finite set of one or more nodes such that

- There is a specially designated node called *root*.
- The remaining nodes are partitioned into $n >= 0$ disjoint set $T_1,\ldots,T_n$, where each of these sets is a tree. $T_1,\ldots,T_n$ are called the *subtrees* of the root.

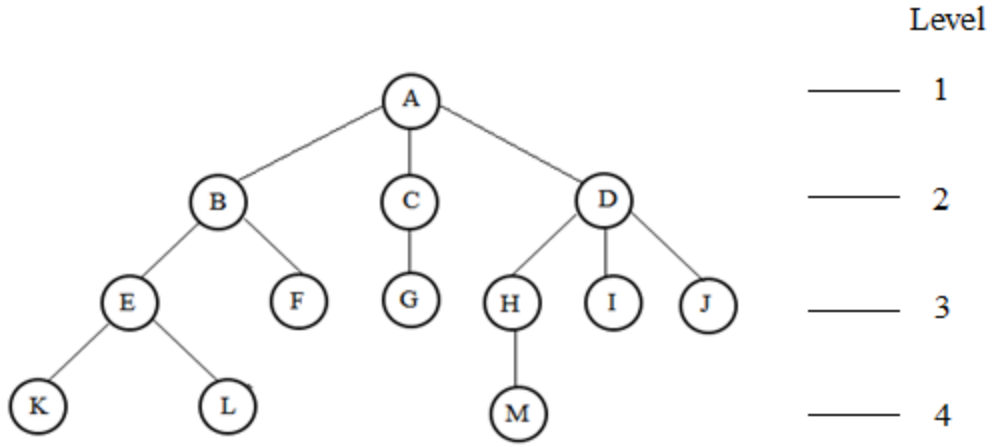


Every node in the tree is the root of some subtree

TERMINOLOGY

- ***Node:*** The item of information plus the branches to other nodes
- ***Degree:*** The number of subtrees of a node
- ***Degree of a tree:*** The maximum of the degree of the nodes in the tree.
- ***Terminal nodes (or leaf):*** nodes that have degree zero or node with no successor
- ***Nonterminal nodes***: nodes that don't belong to terminal nodes.
- ***Parent and Children:*** Suppose N is a node in T with left successor S1 and right successor S2, then N is called the Parent (or father) of S1 and S2. Here, S1 is called left child (or Son) and S2 is called right child (or Son) of N.
- ***Siblings:*** Children of the same parent are said to be siblings.
- ***Edge:*** A line drawn from node N of a T to a successor is called an edge
- ***Path:*** A sequence of consecutive edges from node N to a node M is called a path.
- ***Ancestors of a node:*** All the nodes along the path from the root to that node.
- ***The level of a node:*** defined by letting the root be at level zero. If a node is at level *l*, then it children are at level *l+1*.
- ***Height (or depth):*** The maximum level of any node in the tree

Example



A is the root node
B is the parent of E and F
C and D are the sibling of B
E and F are the children of B
K, L, F, G, M, I, J are external nodes, or leaves
A, B, C, D, E, H are internal nodes
The level of E is 3
The height (depth) of the tree is 4
The degree of node B is 2
The degree of the tree is 3
The ancestors of node M is A, D, H
The descendants of node D is H, I, J, M

Representation of Trees

There are several ways to represent a given tree such as:

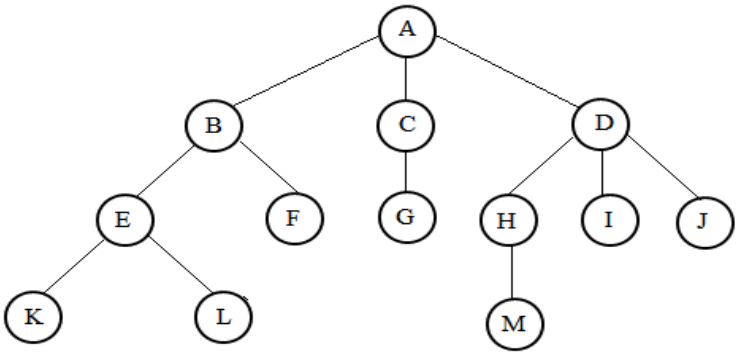


Figure (A)

1. List Representation
2. Left Child- Right Sibling Representation
3. Representation as a Degree-Two tree

List Representation:

The tree can be represented as a List. The tree of **figure (A)** could be written as the list.

(A (B (E (K, L), F), C (G), D (H (M), I, J)))

- The information in the root node comes first.
- The root node is followed by a list of the subtrees of that node.

Tree node is represented by a memory node that has fields for the data and pointers to the tree node's children

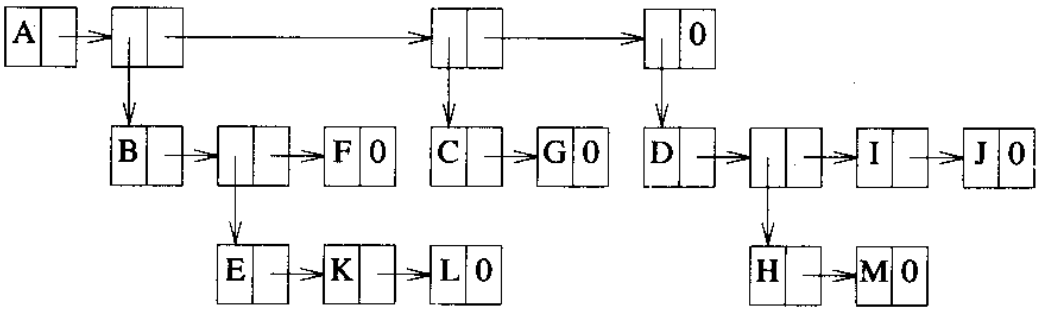


Figure (B): List representation of the tree of figure (A)

Since the degree of each tree node may be different, so memory nodes with a varying number of pointer fields are used.

For a tree of degree k, the node structure can be represented as below figure. Each child field is used to point to a subtree.

Left Child-Right Sibling Representation

The below figure show the node structure used in the left child-right sibling representation

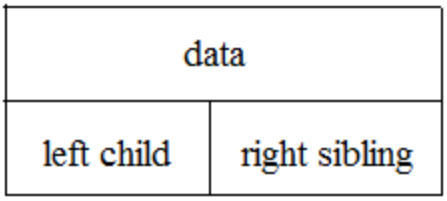


Figure (c): Left child right sibling node structure

To convert the tree of Figure (A) into this representation:

1. First note that every node has at most one leftmost child
2. At most one closest right sibling.

Ex:

- In Figure (A), the leftmost child of A is B, and the leftmost child of D is H.
- The closest right sibling of B is C, and the closest right sibling of H is I.
- Choose the nodes based on how the tree is drawn. The left child field of each node points to its leftmost child (if any), and the right sibling field points to its closest right sibling (if any).

Figure (D) shows the tree of Figure (A) redrawn using the left child-right sibling representation.

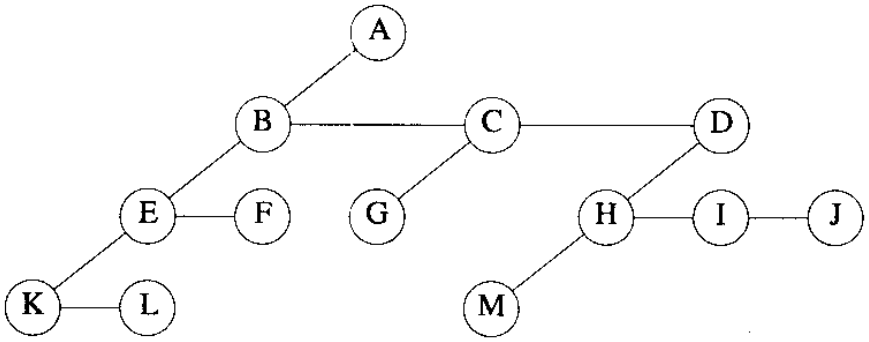


Figure (D): Left child-right sibling representation of tree of figure (A)

Representation as a Degree-Two Tree

To obtain the degree-two tree representation of a tree, simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees. This gives us the degree-two tree displayed in Figure (E).

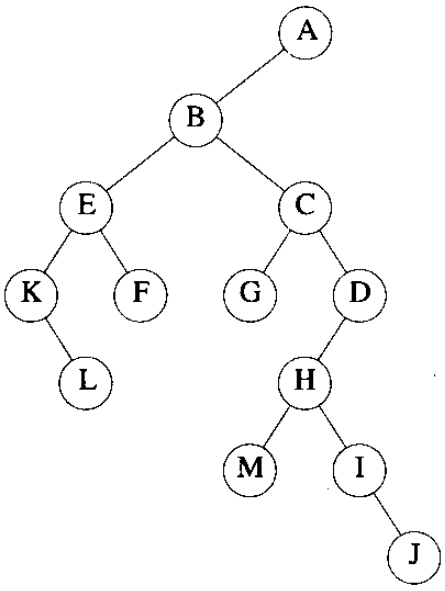


Figure (E): degree-two representation

In the degree-two representation, a node has two children as the left and right children.

BINARY TREES

Definition: A binary tree T is defined as a finite set of nodes such that,

- T is empty or
- T consists of a root and two disjoint binary trees called the left subtree and the right subtree.

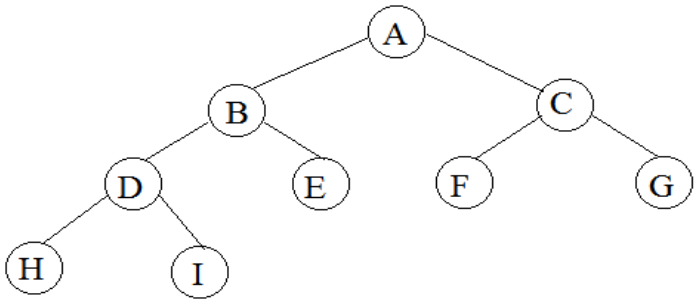


Figure: Binary Tree

Different kinds of Binary Tree

1. Skewed Tree

A skewed tree is a tree, skewed to the left or skews to the right.

or

It is a tree consisting of only left subtree or only right subtree.

- A tree with only left subtrees is called Left Skewed Binary Tree.
- A tree with only right subtrees is called Right Skewed Binary Tree.

2. Complete Binary Tree

A binary tree T is said to complete if all its levels, except possibly the last level, have the maximum number node 2^i, $i \geq 0$ and if all the nodes at the last level appears as far left as possible.

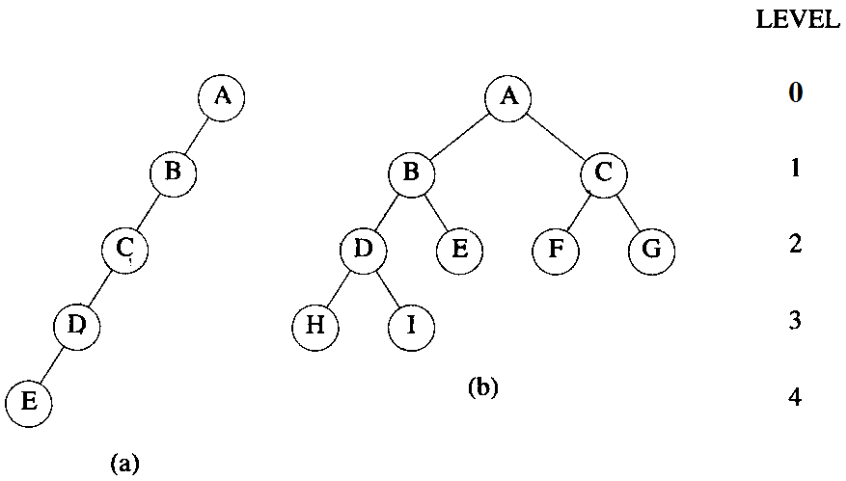


Figure (a): Skewed binary tree Figure (b): Complete binary tree

3. Full Binary Tree

A full binary tree of depth 'k' is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 1$.

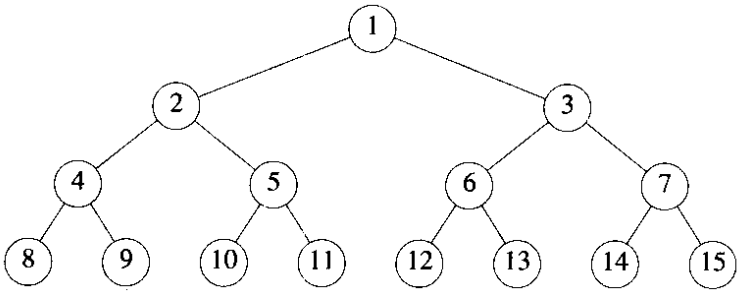
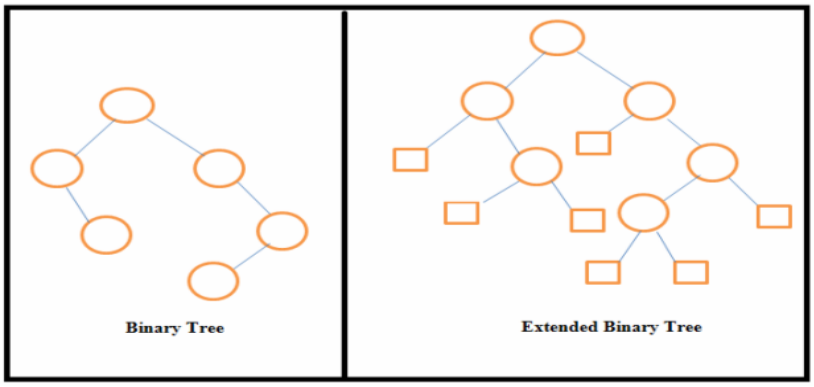


Figure: Full binary tree of level 4 with sequential node number

4. Extended Binary Trees or 2-trees

An *extended binary tree* is a transformation of any binary tree into a complete binary tree. This transformation consists of replacing every null subtree of the original tree with "special nodes." The nodes from the original tree are then *internal nodes*, while the special nodes are *external nodes*.

For instance, consider the following binary tree.



The following tree is its extended binary tree. The circles represent internal nodes, and square represent external nodes.

Every internal node in the extended tree has exactly two children, and every external node is a leaf. The result is a complete binary tree.

PROPERTIES OF BINARY TREES

Lemma 1: [Maximum number of nodes]:
(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Proof:
(1) The proof is by induction on i.
Induction Base: The root is the only node on level i = 1. Hence, the maximum number of nodes on level i =1 is $2^{i-1} = 2^0 = 1$.
Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level i -1is 2^{i-2}
Induction Step: The maximum number of nodes on level i -1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1, or 2^{i-1}

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=0}^{k} (\text{maximum number of nodes on level i}) = \sum_{i=0}^{k} 2^{i-1} = 2^{k-1}$$

Lemma 2: [Relation between number of leaf nodes and degree-2 nodes]:
For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof: Let n_1 be the number of nodes of degree one and n the total number of nodes.
Since all nodes in T are at most of degree two, we have

$$n = n_0 + n_1 + n_2 \qquad (1)$$

Count the number of branches in a binary tree. If B is the number of branches, then

$$n = B + 1.$$

All branches stem from a node of degree one or two. Thus,

$$B = n_1 + 2n_2.$$

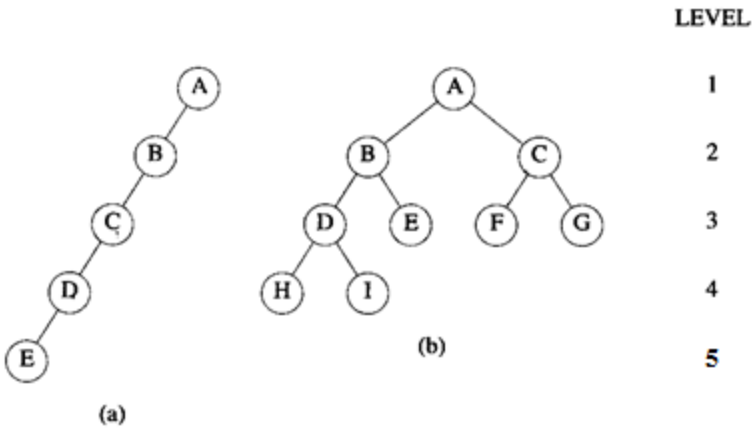
Hence, we obtain

$$n = B + 1 = n_1 + 2n_2 + 1 \qquad (2)$$

Subtracting Eq. (2) from Eq. (1) and rearranging terms, we get

$$\mathbf{n_0 = n_2 + 1}$$

Consider the figure:



Here, For Figure (b) $n_2=4$, $n_0= n_2+1= 4+1=5$
Therefore, the total number of leaf node=5

BINARY TREE REPRESENTATION

The storage representation of binary trees can be classified as

1. Array representation
2. Linked representation.

Array representation:

- A tree can be represented using an array, which is called sequential representation.
- The nodes are numbered from 1 to n, and one dimensional array can be used to store the nodes.
- Position 0 of this array is left empty and the node numbered *i* is mapped to position *i* of the array.

Below figure shows the array representation for both the trees of figure (a).

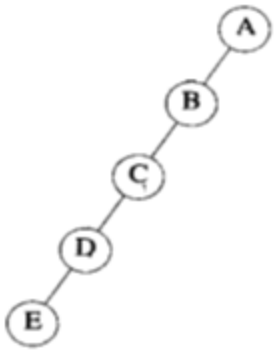


Figure 1(a) Skewed binary tree

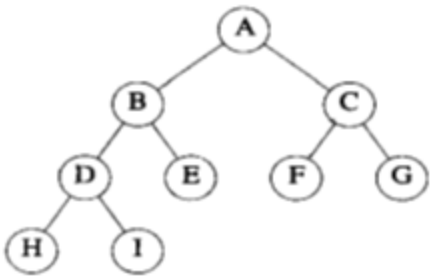
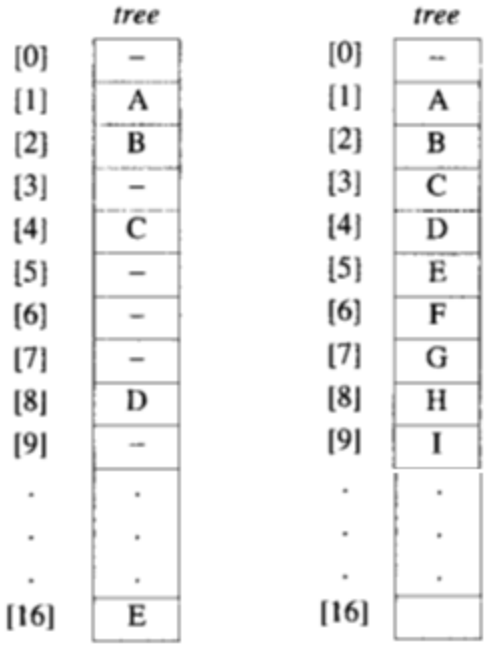


Figure 1(b) Complete binary tree



(a). Tree of figure 1(a) **(b). Tree of figure 1(b)**

- For complete binary tree the array representation is ideal, as no space is wasted.
- For the skewed tree less than half the array is utilized.

Linked representation:

The problems in array representation are:

- It is good for complete binary trees, but more memory is wasted for skewed and many other binary trees.
- The insertion and deletion of nodes from the middle of a tree require the movement of many nodes to reflect the change in level number of these nodes.

These problems can be easily overcome by linked representation

Each node has three fields,

- LeftChild - which contains the address of left subtree
- RightChild - which contains the address of right subtree.
- Data - which contains the actual information

C Code for node:

```
typedef struct node *treepointer;
typedef struct {
                int data;
                treepointer leftChild, rightChild;
        }node;
```

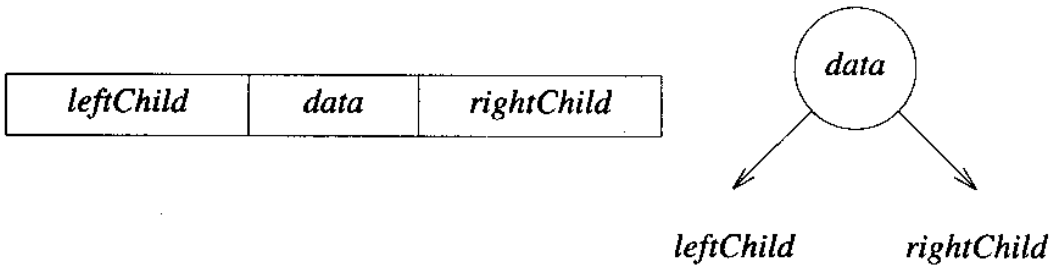


Figure: Node representation

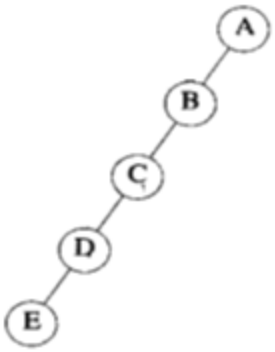


Figure 1(a) Skewed binary tree

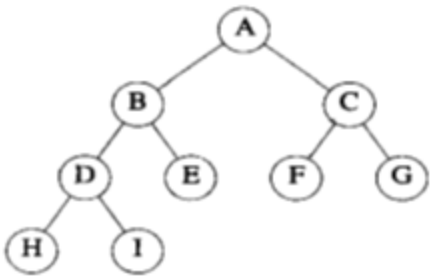
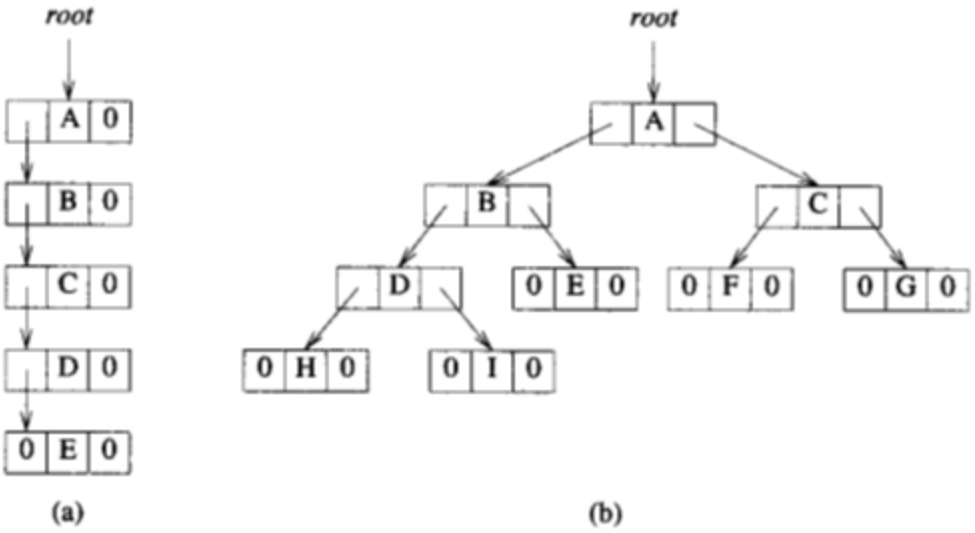


Figure 1(b) Complete binary tree



Linked representation of the binary tree

BINARY TREE TRAVERSALS

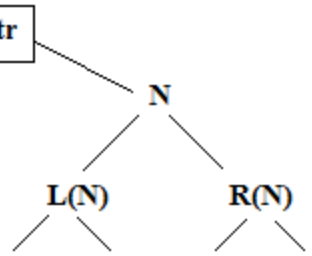
Visiting each node in a tree exactly once is called tree traversal

The different methods of traversing a binary tree are:

1. Preorder
2. Inorder
3. Postorder
4. Iterative inorder Traversal
5. Level-Order traversal

1. Inorder: Inorder traversal calls for moving down the tree toward the left until you cannot go further. Then visit the node, move one node to the right and continue. If no move can be done, then go back one more node.

Let ptr is the pointer which contains the location of the node N currently being scanned.
L(N) denotes the leftchild of node N and R(N) is the right child of node N



Recursion function:

The inorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

```
void inorder(treepointerptr)
{
        if (ptr)
        {
                inorder (ptr→leftchild);
                printf ("%d",ptr→data);
                inorder (ptr→rightchild);
          }
    }
```

2. Preorder: Preorder is the procedure of visiting a node, traverse left and continue. When you cannot continue, move right and begin again or move back until you can move right and resume.

Recursion function:

The Preorder traversal of a binary tree can be recursively defined as

- Visit the root
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder

```
void preorder (treepointerptr)
{
        if (ptr)
        {
                printf ("%d",ptr→data)
                preorder (ptr→leftchild);
                preorder (ptr→rightchild);
          }
 }
```

3. Postorder: Postorder traversal calls for moving down the tree towards the left until you can go no further. Then move to the right node and then visit the node and continue.

Recursion function:

The Postorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root

```
void postorder(treepointerptr)
{
        if (ptr)
        {
                postorder (ptr→leftchild);
                postorder (ptr→rightchild);
                printf ("%d",ptr→data);
         }
  }
```

4. Iterative inorder Traversal:

Iterative inorder traversal explicitly make use of stack function.

The left nodes are pushed into stack until a null node is reached, the node is then removed from the stack and displayed, and the node's right child is stacked until a null node is reached. The traversal then continues with the left child. The traversal is complete when the stack is empty.

```
void iterInorder(treePointer node)
{
   int top = -1; /* initialize stack */
   treePointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node→leftChild)
         push(node); /* add to stack */
      node = pop(); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node→data);
      node = node→rightChild;
   }
}
```

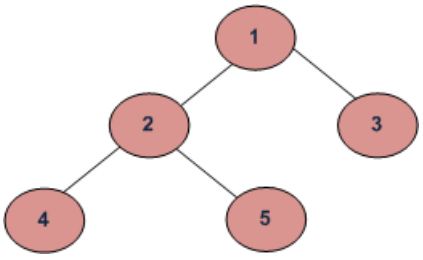
Program : Iterative inorder traversal

5. Level-Order traversal:

Visiting the nodes using the ordering suggested by the node numbering is called level ordering traversing.

The nodes in a tree are numbered starting with the root on level 1 and so on.

Firstly visit the root, then the root's left child, followed by the root's right child. Thus continuing in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.



Level order traversal: 1 2 3 4 5

Initially in the code for level order add the root to the queue. The function operates by deleting the node at the front of the queue, printing the nodes data field and adding the nodes left and right children to the queue.

Function for level order traversal of a binary tree:

```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
   int front = rear = 0;
   treePointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty tree */
   addq(ptr);
   for (;;) {
      ptr = deleteq();
      if (ptr) {
         printf("%d",ptr→data);
         if(ptr→leftChild)
            addq(ptr→leftChild);
         if (ptr→rightChild)
            addq(ptr→rightChild);
      }
      else break;
   }
}
```

Program : Level-order traversal of a binary tree

ADDITIONAL BINARY TREE OPERATIONS

1. Copying a Binary tree

This operations will perform a copying of one binary tree to another.

C function to copy a binary tree:

```
treepointer copy(treepointer original)
{  if(original)
    {  MALLOC(temp,sizeof(*temp));
       temp→leftchild=copy(original→leftchild);
       temp→rightchild=copy(original→rightchild);
       temp→data=original→data;
       return temp;
    }
   return NULL;
}
```

2. Testing Equality

This operation will determin the equivalance of two binary tree. Equivalance binary tree have the same strucutre and the same information in the corresponding nodes.

C function for testing equality of a binary tree:

```
int equal(treepointer first,treepointer second)
{
        return((!first && !second) || (first && second && (first→data==second→data)
        && equal(first→leftchild,second→leftchild) && equal(first→rightchild,
         second→rightchild))
}
```

This function will return TRUE if two trees are equivalent and FALSE if they are not.

3. The Satisfiability problem

- Consider the formula that is constructed by set of variables: $x_1, x_2, \ldots, x_n$ and operators $\wedge(and), \vee(or), \neg(not)$.
- The variables can hold only of two possible values, *true* or *false*.
- The expression can form using these variables and operators is defined by the following rules.
    - A variable is an expression
    - If x and y are expressions, then $\neg x, x \wedge y, x \vee y$ are expressions
    - Parentheses can be used to alter the normal order of evaluation $(\neg > \wedge > \vee)$

Example: $x_1 \vee (x_2 \wedge \neg x_3)$ If x_1 and x_3 are *false* and x_2 is *true*

$= false \vee (true \wedge \neg false)$

$= false \vee true$

$= true$

The satisfiablity problem for formulas of the propositional calculus asks if there is an assignment of values to the variable that causes the value of the expression to be *true*.

Let's assume the formula in a binary tree

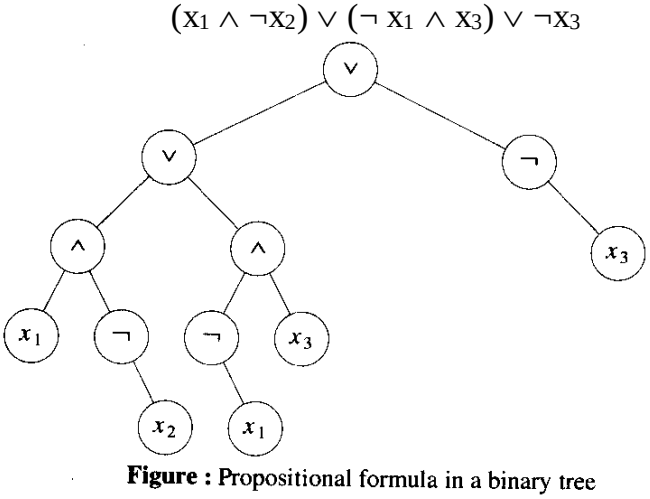


Figure : Propositional formula in a binary tree

The inorder traversal of this tree is

$$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_3 \vee \neg x_3$$

The algorithm to determine satisfiablity is to let (x_1, x_2, x_3) takes on all the possible combination of true and false values to check the formula for each combination.

For *n* value of an expression, there are 2^n possible combinations of *true* and *false*
For example n=3, the eight combinations are (t,t,t), (t,t,f), (t,f,t), (t,f,f), (f,t,t), (f,t,f), (f,f,t), (f,f,f).
The algorithm will take $O(g\ 2^n)$, where g is the time to substitute values for x_1, x_2,… x_n and evaluate the expression.

Node structure:
For the purpose of evaluation algorithm, assume each node has four fields:

left_child	*data*	*value*	*right_child*

Figure : Node structure for the satisfiability problem

Define this node structure in C as:

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *tree_pointer;
typedef struct node {
        tree_pointer left_child;
        logical      data;
        short int    value;
        tree_pointer right_child;
        } ;
```

Satisfiability function: The first version of Satisfiability algorithm

```
for (all 2^n possible combinations) {
   generate the next combination;
   replace the variables by their values;
   evaluate root by traversing it in postorder;
   if (root→value) {
      printf(<combination>);
      return;
   }
}
printf("No satisfiable combination\n");
```

THREADED BINARY TREE

The limitations of binary tree are:

- In binary tree, there are n+1 null links out of 2n total links.
- Traversing a tree with binary tree is time consuming.

These limitations can be overcome by threaded binary tree.

In the linked representation of any binary tree, there are more null links than actual pointers. These null links are replaced by the pointers, called **threads**, which points to other nodes in the tree.

To construct the threads use the following rules:

1. Assume that **ptr** represents a node. If ptr→leftChild is null, then replace the null link with a pointer to the inorder predecessor of ptr.
2. If ptr →rightChild is null, replace the null link with a pointer to the inorder successor of ptr.

Ex: Consider the binary tree as shown in below figure:

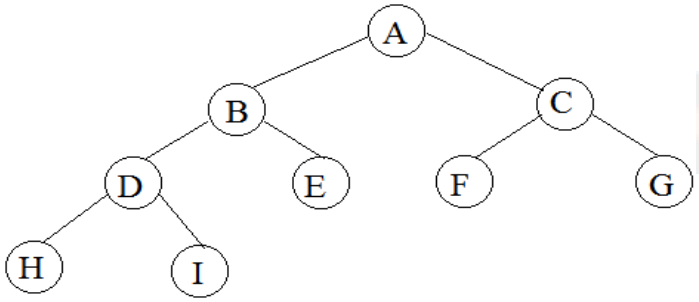


Figure A: Binary Tree

There should be no loose threads in threaded binary tree. But in **Figure B** two threads have been left dangling: one in the left child of *H*, the other in the right child of G.

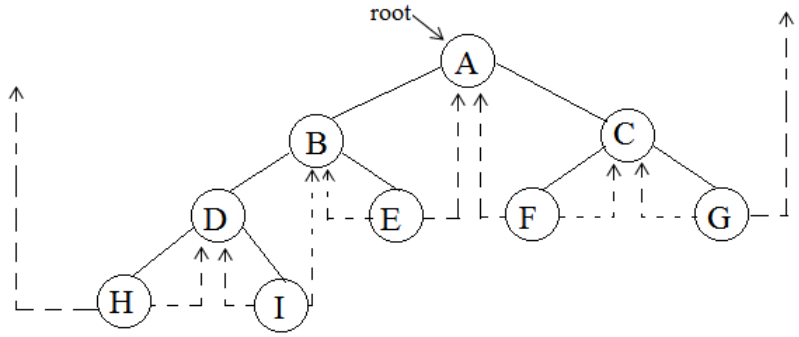


Figure B: Threaded tree corresponding to Figure A

In above figure the new threads are drawn in broken lines. This tree has 9 node and 10 0-links which has been replaced by threads.

When trees are represented in memory, it should be able to distinguish between threads and pointers. This can be done by adding two additional fields to node structure, ie., *leftThread* and *rightThread*

- If `ptr→leftThread = TRUE`, then `ptr→leftChild` contains a thread, otherwise it contains a pointer to the left child.
- If `ptr→rightThread = TRUE`, then `ptr→rightChild` contains a thread, otherwise it contains a pointer to the right child.

Node Structure:

The node structure is given in C declaration

```
typedef struct threadTree *threadPointer
typedef struct{
              short int leftThread;
              threadPointer leftChild;
              char data;
              threadPointer rightChild;
              short int rightThread;
         }threadTree;
```

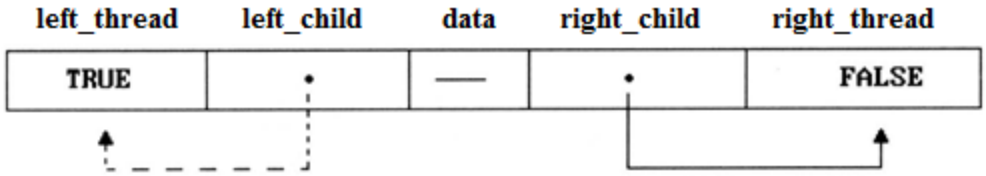
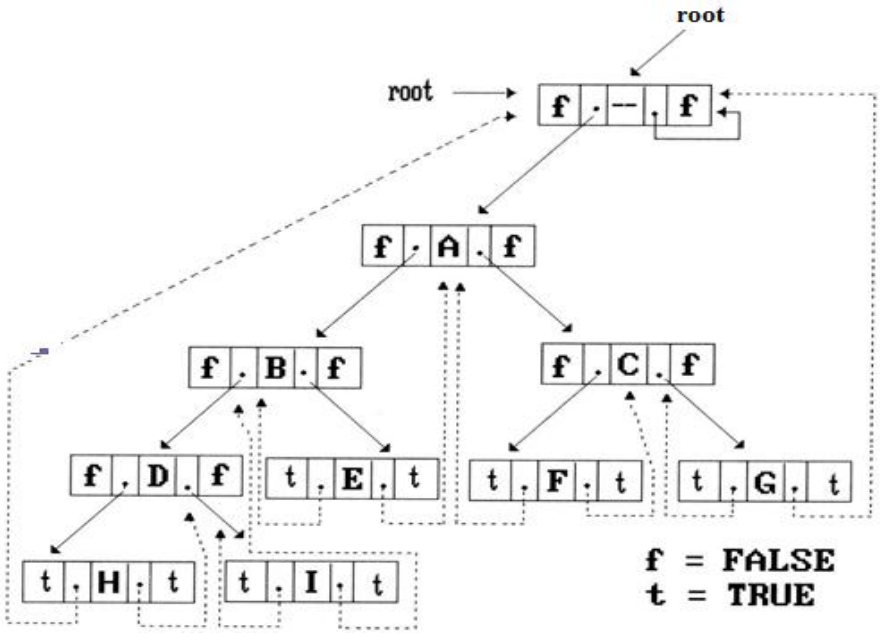


Figure An empty threaded tree

The complete memory representation for the tree of figure is shown in Figure C



The variable ***root*** points to the header node of the tree, while **root →leftChild** points to the start of the first node of the actual tree. This is true for all threaded trees. Here the problem of the loose threads is handled by pointing to the head node called ***root***.

Inorder Traversal of a Threaded Binary Tree

- By using the threads, an inorder traversal can be performed without making use of a stack.
- For any node, **ptr**, in a threaded binary tree, if **ptr→rightThread =TRUE,** the inorder successor of **ptr** is **ptr →rightChild** by definition of the threads. Otherwise we obtain the inorder successor of **ptr** by following a path of **left-child links** from the **right-child** of **ptr** until we reach a node with **leftThread = TRUE.**
- The function insucc **( )** finds the inorder successor of any node in a threaded tree without using a stack.

```
threadedpointer insucc(threadedPointer tree)
{       /* find the inorder successor of tree in a threaded binary tree */
        threadedpointer temp;
        temp = tree→rightChild;
        if (!tree→rightThread)
           while (!temp→leftThread)
                   temp = temp→leftChild;
        return temp;
```

Program: Finding inorder successor of a node

To perform inorder traversal make repeated calls to insucc ( ) function

```
void tinorder (threadedpointer tree)
{
Threadedpointer  temp = tree;
for(; ;){
                temp = insucc(temp);
                if (temp == tree)     break;
                printf("%3c", temp→data
     }
}
```

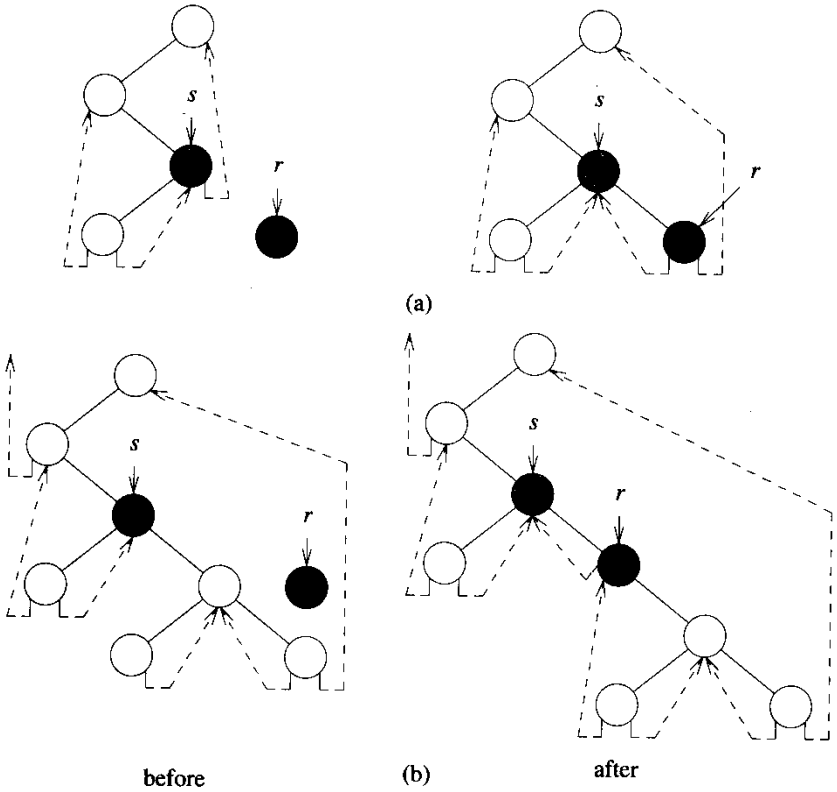
Program: Inorder traversal of a threaded binary tree

Inserting a Node into a Threaded Binary Tree

In this case, the insertion of **r** as the right child of a node **s** is studied.

The cases for insertion are:

- If ***s*** has an **empty** right subtree, then the insertion is simple and diagrammed in Figure
- If the right subtree of ***s*** is **not empty**, then this right subtree is made the right subtree of ***r*** after insertion. When this is done, r becomes the inorder predecessor of a node that has a **leftThread == true** field, and consequently there is a thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s.




```
void insertRight(threadedPointer Sf threadedPointer r)
{    /* insert r as the right child of s */
       threadedpointer temp;
       r→rightChild = parent→rightChild;
       r→rightThread = parent→rightThread;
       r→leftChild = parent;
       r→leftThread = TRUE;
       s→rightChild = child;
       s→rightThread = FALSE;
        if (!r→rightThread) {
              temp = insucc(r);
              temp→leftChild = r;
        }
}
```

BINARY SEARCH TREE

A dictionary is a collection of pairs (key, item), each pair has a key and an associated item. For example:

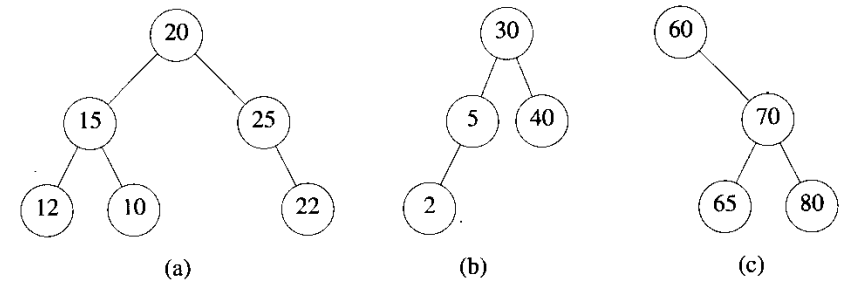
USN	Name
123456	ABCDEF

In the above dictionary USN is a key element and Name is item associated with the key.

Definition: Binary search tree

Binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties.

1. Every node has exactly one key and keys are distinct.
2. The keys (if any) in the left subtree are smaller than the key in the root
3. The keys (if any) in the right subtree are larger than the key in the root
4. The left and right subtrees are also binary search trees



1. Searching in a binary search tree

- Searching a node in binary search tree whose key is k, then searching begins at the root. If the root is NULL, the search tree contains no nodes and the search is unsuccessful.
- If the root is not NULL, compare k with the key in root. If k equals the root's key, then the search terminates successfully.
- If k is less than root's key, then no element in the right subtree can have equal to k. Therefore, search the left subtree of the root.
- If k is larger than root's key value, then search the right subtree of the root

```
element* search(treePointer root, int key)
{/* return a pointer to the element whose key is k, if
    there is no such element, return NULL. */
  if (!root) return NULL;
  if (k == root→data.key) return &(root→data);
  if (k < root→data.key)
    return search(root→leftChild, k);
  return search(root→rightChild, k);
}
```

Program: Recursive search of a binary search tree

```
element* iterSearch(treePointer tree, int k)
{/* return a pointer to the element whose key is k,  if
    there is no such element, return NULL. */
   while (tree) {
      if (k == tree→data.key) return &(tree→data);
      if (k < tree→data.key)
         tree = tree→leftChild;
      else
         tree = tree→rightChild;
      }
   return NULL;
}
```

Program : Iterative search of a binary search tree

Analysis of search and itersearch:

If h is the height of the binary search tree, the time complexity for search or itersearch is O(h)

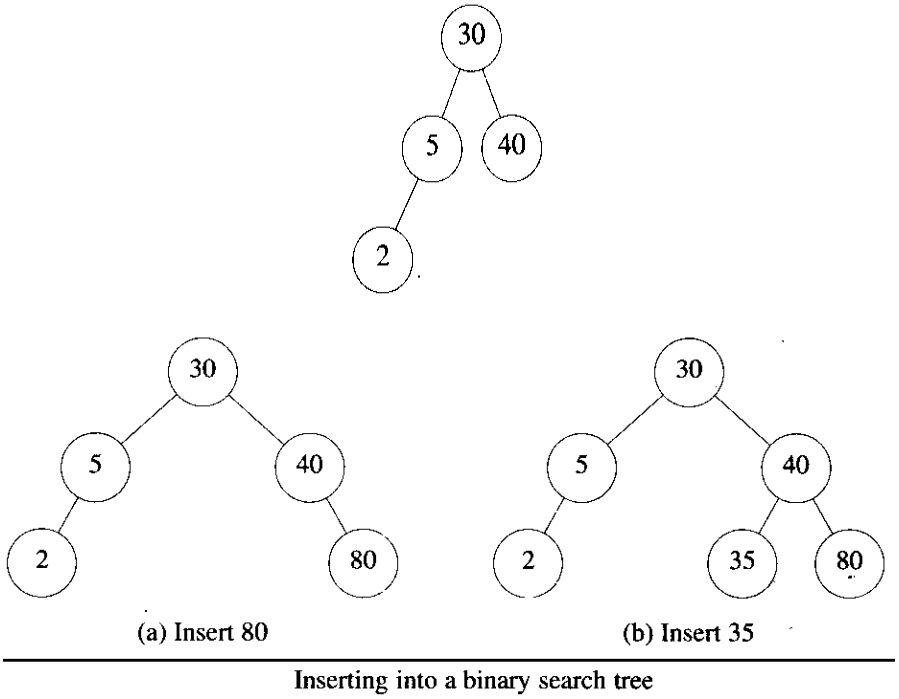
2. Inserting into a binary search tree:

To insert a dictionary pair into a binary search tree whose key is k, verify that the key is different from those of existing pairs. To do this search the tree, if the search is unsuccessful, then insert the pair at the point the search terminated.

```
void insert(treePointer *node, int k, iType theItem)
{/* if k is in the tree pointed at by node do nothing;
    otherwise add a new node with data = (k, theItem) */
  treePointer ptr, temp = modifiedSearch(*node, k);
  if (temp || !(*node)) {
     /* k is not in the tree */
     MALLOC(ptr, sizeof(*ptr));
     ptr→data.key = k;
     ptr→data.item = theItem;
     ptr→leftChild = ptr→rightChild = NULL;
     if (*node) /* insert as child of temp */
        if (k < temp→data.key) temp→leftChild = ptr;
        else temp→rightChild = ptr;
     else *node = ptr;
  }
}
```

Program : Inserting a dictionary pair into a binary search tree

For example: Insert a pair with key 80 and 35 to the following tree:



Inserting into a binary search tree

- The **insert** function uses the function *modifiedSearch* which searches the binary search tree **node* for the key k.
- If the tree is empty or if k (key) is present, it returns NULL.
- Otherwise, it returns a pointer to the last node of the tree that was encountered during search. The new pair is to be inserted as a child of this node.

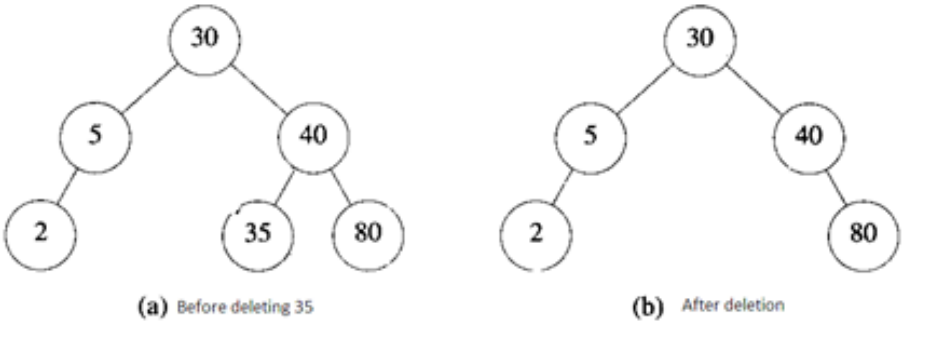
Analysis of insert

The time required to search the tree for k is O(h), where h is its height. The remainder of algorithm takes Θ(1) time. So overall time needed by insert function is O(h).

3. Deletion from a Binary search tree:

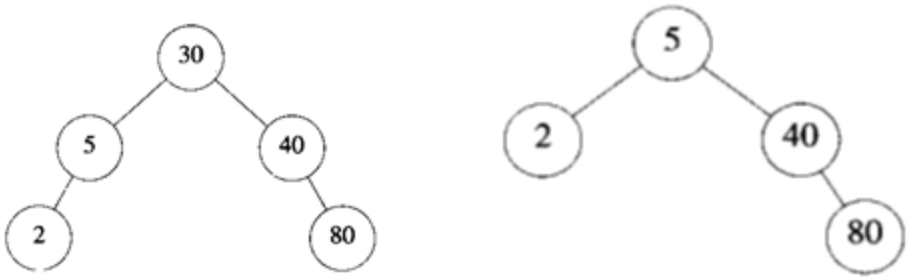
Deletion of a leaf node is easy.

For ex: To delete 35 from the tree, the left child field of its parent is set to 0 and the node is freed. This gives us the tree of Figure (b)



The deletion of a non-leaf that has only child is also easy.

When the pair to be deleted is in a non-leaf node that has two children, the pair to be deleted is replaced by either the largest pair in its left subtree or the smallest one in its right subtree.

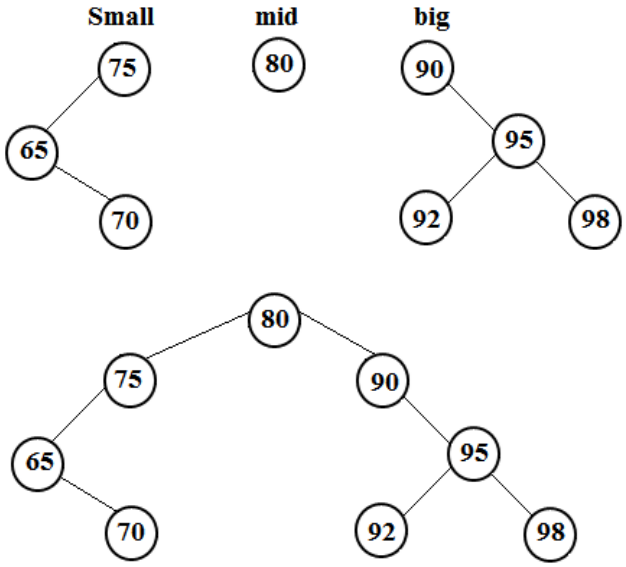


For ex: Delete the pair with the key 30 from the above tree. Then replace it with the largest pair of left subtree or smallest pair of right subtree that is 5 or 40.
Suppose the largest pair in left subtree is opted, then pair with the key 5 is moved into the root.

4. Joining and Splitting Binary Trees

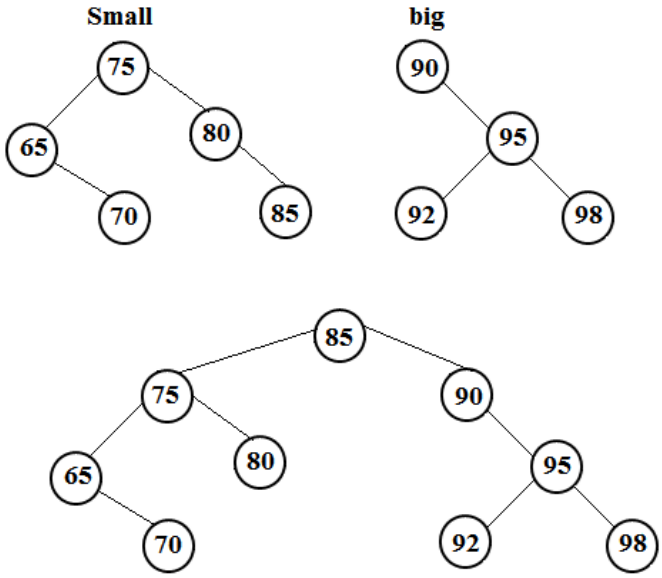
(a) **threeWayJoin (small, mid, big):** This creates a binary search tree consisting of the pairs initially in the binary search trees small and big, as well as the pair mid. It is assumed that each key in small is smaller than mid. key and that each key in big is greater than mid. key. Following the Join, both small and big are empty.

Ex:



(b) **twoWayJoin (small, big):** This joins the two binary search trees small and big to obtain a single binary search tree that contains all the pairs originally in small and big. It is assumed that all keys of small are smaller than all keys of big and that following the join both small and big are empty.

Ex:



(c) **split (theTree, k, small, mid, big):** The binary search tree ***theTree*** is split into three parts: ***small*** is a binary search tree that contains all pairs of ***theTree*** that have key less than **k**, ***mid*** is the pair (if any) in ***theTree*** whose key is **k** and ***big*** is a binary search tree that contains all pairs of ***theTree*** that have key larger than k. Following the split operation ***theTree*** is empty. When ***theTree*** has no pair whose key is k, mid.key