School of Electrical Sciences
IIT Bhubaneswar
Programming and Data Structure

Instructor: P Bera & Goutam Biswas *Spring Semester 2015-2016*

1 Rooted Tree and Binary Tree

1.1 Rooted Tree

Definition 1: A *tree* is a *non-empty* finite set T where

1. there is a designated element r called the *root*.
2. The the set $T \backslash \{r\}$, if non-empty, is partitioned in disjoint subsets $T_1, \cdots, T_k$.
3. Each T_i, $i = 1, \cdots, k$ is a tree, known as a *subtree* of T.

It is clear that the definition is recursive. A tree may be drawn as a connected acyclic graph with the root at the top (convention). There are different terms associated with a rooted-tree T. They are defined as follows:
Definition 2:

1. Elements of a tree are called *nodes*. Information is stored in nodes, when a rooted tree is used as a data structure.
2. Each node has 0 or any finite number of subtrees. The number of subtrees of a node is called its *degree*.
3. A node is called a *leaf node* if its degree is 0. Other nodes are non-leaf or *internal nodes*.
4. Each subtree of the root r of T is also a tree. So each of them has its root. These are the *children* of r, and r is the *parent* of these subtree roots. The root r does not have any parent and a leaf node does not have any children.
5. Each child node is *logically connected* to its parent. This connection is called the *edge* or *link* between the parent and the child. These edges are the edges of the acyclic graph of the tree.
6. Children of the same parent are *siblings*.
7. All the nodes on the path (following the edges) from the root r to any node p in T are the *ancestors* of p.
8. The degree of a tree is the maximum degree of any node in the tree.
9. The *level* or *depth* of the root r is 1 (some people take it as 0). If the level of a node p in a tree is l, then the level of its children (if any) is $l + 1$.
10. The *height* (h) or *depth* of a tree is the value of the maximum level of any node of the tree.

Ex 1.

1. If a tree T has n nodes, then there are $n-1$ edges.

2. If a tree T has n nodes, the minimum number of leaf nodes is 1.

Proof:

1. The proof is by induction on n.
   Basis: It is true for $n = 1$ as there is no child and so there is no *edge*.
   Induction & Hypothesis: Assume that the proposition is true for n. The $(n+1)^{th}$ node is connected to the tree by an edge. So it is true for $n+1$.

2. There are $n-1$ edges, each age is connected to an internal node. So there can at most be $n-1$ internal nodes and only one leaf node.

QED.

1.2 Binary Tree

Binary tree is a rooted tree where the maximum degree of any node is 2. But there is a small difference in the definition of a binary tree.
Definition 3: A *binary tree* is a finite set of nodes. If it is non-empty it has an element called root and two disjoint subtrees known as *left subtree* and *right subtree*.

So a binary tree can be *empty*. It is known as a *empty* or *null tree*. In a binary tree we differentiate between *left* and *right* subtrees. A binary tree may be viewed as a graph or may be written as a 3-tuple,

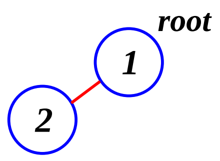
$$(\text{left-sub-tree}, \text{root}, \text{right-sub-tree}).$$

We write a *null tree* as '$\perp$'.

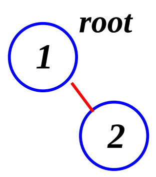
Example 1.

1. $\perp$, $(\perp, 1, \perp)$, $T_1 = ((\perp, 2, \perp), 1, \perp)$, $T_2 = (\perp, 1, (\perp, 2, \perp))$. Note that $T_1 \neq T_2$.

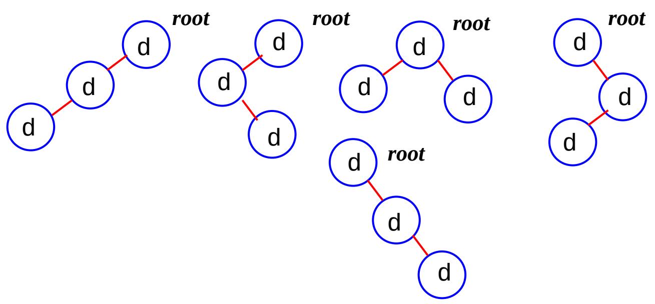
T_1 :



T_2 :



2. If we ignore the data, there is one *null* tree; one tree with a single node: $(\perp, d, \perp)$; two trees with two nodes: $((\perp, d, \perp), d, \perp)$ and $(\perp, d, (\perp, d, \perp))$; five trees with three nodes: $(((\perp, d, \perp), d, \perp), d, \perp)$, $((\perp, d, (\perp, d, \perp)), d, \perp)$, $((\perp, d, \perp)), d, (\perp, d, \perp))$, $(\perp, d, ((\perp, d, \perp), d, \perp))$, $(\perp, d, (\perp, d, (\perp, d, \perp)))$



<u>*Ex 2.*</u>

1. Take the level of the root as 1 and prove that the maximum number of nodes at level l is 2^{l-1}.
2. Maximum number of nodes in a l level binary tree is $2^l - 1$.
3. Give a recursive definition of B_n, the number of binary trees of n nodes.
4. Let a binary tree has n_0 number of leaf nodes and n_2 number of nodes with two children. Prove that $n_0 = n_2 + 1$.

Proof:

1. The proof is by induction on l.
   Basis: There is one node (root) at level 1.
   Hypothesis & Induction: There are 2^{l-1} nodes at level l. Two nodes can be attached to each of them, so there are 2^l nodes at level $l+1$.
2. This is $1 + 2 + \cdots + 2^{l-1} = 2^l - 1$.
3. $B_0 = 1$, $B_n = \sum_{i=0}^{n-1} B_i B_{n-i-1}$, where $n \geq 1$. There are several formula for $B_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{2(2n-1)}{n+1} B_{n-1}$.
4. Let the tree has n_1 number of nodes with one children. So the total number of edges are $n_0 + n_1 + n_2 - 1$. Counting in other way, the number of edges are $2n_2 + n_1$. Equating them we get $n_0 + n_1 + n_2 = 2n_2 + n_1$, implies the result.
   Another way is to prove it by induction on n.
   Basis $(n = 1)$*:* One leaf node and no node of two children.
   Hypothesis & Induction: Assume that the proposition be true for some k. When we attach the next node there are two possibilities, (i) it is attached to a leaf node - the number of leaf nodes and the number of nodes with two children remain unchanged; (ii) it is attached to a node having one child - this increases both the number of leaf nodes and the number of nodes with two children by one.

QED.

<u>*Definition 4:*</u> A binary tree is said to be *full* if each node has either no children or two children. Some authors use different definition for *full binary tree*.
An *almost complete binary tree* of level l is such that

1. If $l = 1$, then there is one node, the root.

2. If $l > 1$, then there are $2^{l-1} - 1$ nodes upto level $l - 1$ (saturated).

3. Nodes at level l are filled from left to right.

Ex 3. Prove that in a *full binary tree* the number of nodes is odd. And if there are n nodes, then $\frac{n-1}{2}$ nodes are internal and $\frac{n+1}{2}$ are leaf nodes.
Proof: Number of nodes start from 1 and increases by 2.
Proof by induction on n. If $n = 1$, there is no internal node and 1 leaf node. If it is true for n nodes, then adding two nodes reduces one leaf node to an internal node and adds two new leaf nodes. QED.

1.3 Representation of Binary Tree

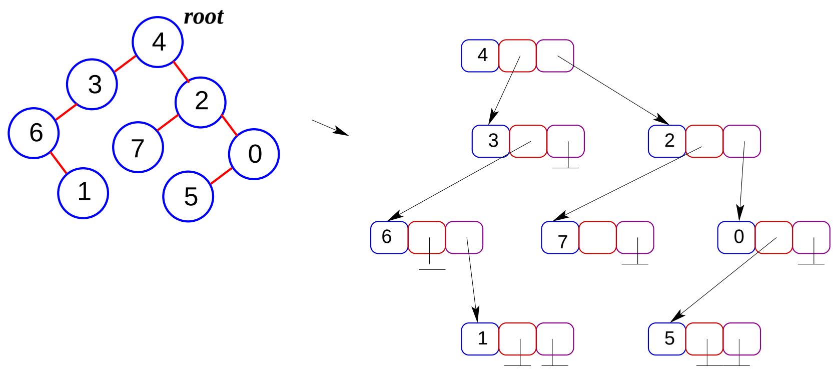
An *almost complete* binary tree with n nodes can be stored in a 1-dimensional array (index: $\{0, 1, \cdots, n-1\}$). The root is stored at index 0. If a node is stored at index i, then

- if $i = 0$, it has no parent. Otherwise the index of the parent is $\lfloor \frac{i-1}{2} \rfloor$ i.e. parent of node 5 and 6 is 2.

- If $2i + 1 < n$, then the index of the *left-child* of i is $2i + 1$. Similarly if $2i + 2 < n$, then the index of the *right-child* of i is $2i + 2$.

But the scheme is space inefficient if the tree is skewed. A skewed tree of n nodes may have height n (root at height 1). But the previous scheme will use 2^{n-1} to $2^n - 1$ array locations. So a large number of locations will remain unused.

An alternate to this is a *linked representation*, which uses three essential fields - *data*, a link to the root of the *left subtree* and a link to the root of the *right subtree*. There may also be a link to the *parent node*.



This can be easily implemented using self-referencing structure of C language.

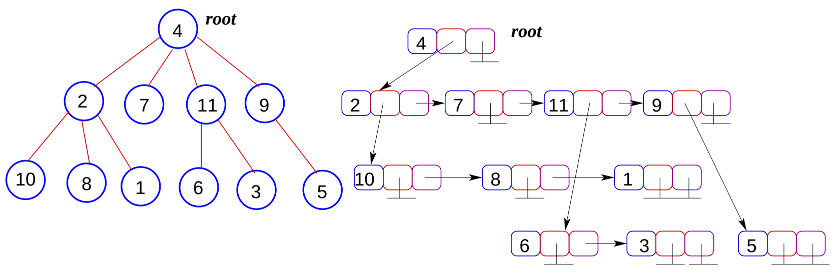
```
struct binaryTreeNode {
        int data ;
        struct binaryTreeNode *leftChild, *rightChild /*, *parent*/ ;
} ;
typedef struct binaryTreeNode *binaryTree ;
```

1.4 Representation of Tree with degree k

We know that the degree of a rooted-tree is the maximum number of children of any internal node. The structure of a node of a degree-k tree may have a data field and k number of links to k child subtrees. If for a particular node the number of subtrees are less than k, then some of the links may be *null*. But this may be space inefficient if most of the nodes have smaller number of children compared to the degree of the tree.

```
#define DEG 20
struct rootedTreeNode {
       int data;
       struct rootedTreeNode *children[DEG];
};
typedef struct rootedTreeNode *rootedTree;
```

An alternate representation of a degree-k tree uses a binary tree like representation. The fields are - *data*, link to *left-child* and link to *right-sibling*.



```
struct rootedTree {
       int data;
       struct rootedTree *leftChild, *rightSibling;
};
typedef struct rootedTreeNode *rootedTree;
```

1.5 Axiomatic Definition of a Binary Tree (ADT)

T	:	The set of binary trees of element of type *Item*,
I	:	The set of values of type *Item*,
B	:	The set of *Boolean* values,
M	:	The set of *Methods*.

Syntax:

$$\begin{aligned}
\mathit{Init}() &\longrightarrow t : T \\
\mathit{IsEmpty}(t) &\longrightarrow b : B \\
\mathit{MakeTree}(t_1, v, t_2) &\longrightarrow t : T \\
\mathit{LeftSubtree}(t) &\longrightarrow \begin{cases} \mathit{error} & \text{if } \mathit{IsEmpty}(t) = \mathit{true} \\ t' : T & \text{otherwise} \end{cases} \\
\mathit{RightSubtree}(t) &\longrightarrow \begin{cases} \mathit{error} & \text{if } \mathit{IsEmpty}(t) = \mathit{true} \\ t' : T & \text{otherwise} \end{cases} \\
\mathit{Root}(t) &\longrightarrow \begin{cases} \mathit{error} & \text{if } \mathit{IsEmpty}(t) = \mathit{true} \\ v : I & \text{otherwise} \end{cases}
\end{aligned}$$

Axioms:

$$\begin{aligned}
\mathit{IsEmpty}(\mathit{Init}()) &= \mathit{true} \\
\mathit{IsEmpty}(\mathit{MakeTree}(t_1, v, t_2)) &= \mathit{false} \\
\mathit{LeftSubtree}(\mathit{Init}()) &= \mathit{error} \\
\mathit{RightSubtree}(\mathit{Init}()) &= \mathit{error} \\
\mathit{Root}(\mathit{Init}()) &= \mathit{error} \\
\mathit{LeftSubtree}(\mathit{MakeTree}(t_1, v, t_2)) &= t_1 \\
\mathit{RightSubtree}(\mathit{MakeTree}(t_1, v, t_2)) &= t_2 \\
\mathit{Root}(\mathit{MakeTree}(t_1, v, t_2)) &= v
\end{aligned}$$

1.6 Traversal in Binary Tree

Visiting the nodes (data) of a binary tree and doing computation is an important operation. An operation may be simply printing the data. If we decompose the tree as l (left sub-tree), R (root), and r (right sub-tree), there are $3! = 6$ possible ways of visiting the nodes (traversals) - lRr, lrR, Rlr, Rrl, rlR, and rRl. But if we decide that the left sub-tree will be visited before the right sub-tree, then the first three are the only possibilities - lRr is called the *inorder traversal*, lrR is the *postorder traversal* and Rlr is the *preorder traversal*.

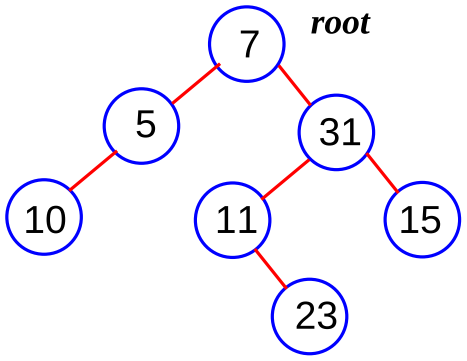
1. lRr - Inorder traversal - perform inorder traversal on the left subtree, compute on the root node (may be just print the data), perform inorder traversal on the right subtree.

2. lrR - Postorder traversal - perform postorder traversal on the left subtree, perform inorder traversal on the right subtree, then compute on the root node.

3. Rlr - Preorder traversal - first compute on the root node, then perform preorder traversal on the left subtree and perform inorder traversal on the right subtree.

It is clear from the description that the definition of these three traversals are inductive (recursive) in nature. So they directly or indirectly use a *stack*.

A binary tree can also be traversed *level-by-level* from left to right. This is known as *level-order traversal*.



Following are outcome of different traversals on the above tree.

- *Pre-order traversal:* `7 5 10 31 11 23 15`
- *In-order traversal:* `10 5 7 11 23 31 15`
- *Post-order traversal:* `10 5 23 11 15 31 7`
- *Level-order traversal:* `7 5 31 10 11 15 23`

Following is a C language code for binary tree and traversals.

```
// binTree.h: Binary tree header file.

#ifndef _BINTREE_H
#define _BINTREE_H

#define ERROR 1
#define OK 0
#define TRUE 1

struct binaryTreeNode {
       int data;
       struct binaryTreeNode *leftChild, *rightChild, *parent;
};
typedef struct binaryTreeNode *binaryTree;

binaryTree createNullTree();
int isEmptyBinTree(binaryTree);
int isRoot(binaryTree) ;
binaryTree makeBinTree(
           binaryTree ,
           int,
           binaryTree
           );
```

```
binaryTree leftSubTree(binaryTree, int *);
binaryTree rightSubTree(binaryTree, int *);
binaryTree parent(binaryTree, int *);

void inorderTravel(binaryTree);
void preorderTravel(binaryTree);
void postorderTravel(binaryTree);
void levelOrderTravel(binaryTree);

int leafCount(binaryTree) ;
binaryTree copyTree(binaryTree);
#endif

// binTree.c: binary tree implementation file

#include "binTree.h"
#include "queue.h"
#include <stdio.h>
#include <stdlib.h>

binaryTree createNullTree() {
           return NULL;
}

int isEmptyBinTree(binaryTree bp) {
    return bp == NULL;
}

int isRoot(binaryTree bp) {
    if(isEmptyBinTree(bp)) return TRUE ;
    return bp == bp -> parent ;
}

binaryTree makeBinTree( binaryTree lp,
                        int n,
                        binaryTree rp
                      ) {
           binaryTree temp ;
           temp = (binaryTree)malloc(sizeof(*temp)) ;
           temp -> data = n ; temp -> parent = temp ;
           temp -> leftChild = lp ; temp -> rightChild = rp ;
           if(!isEmptyBinTree(lp)) lp -> parent = temp ;
           if(!isEmptyBinTree(rp)) rp -> parent = temp ;
           return temp ;
}

binaryTree leftSubTree(binaryTree bp, int *err) {
           if(!isEmptyBinTree(bp)) {
              *err = OK ;
              return bp -> leftChild ;
           }
           *err = ERROR ;
           return NULL ;
}

binaryTree rightSubTree(binaryTree bp, int *err) {
           if(!isEmptyBinTree(bp)) {
              *err = OK ;
              return bp -> rightChild ;
           }
           *err = ERROR ;
           return NULL ;
}

binaryTree parent(binaryTree bp, int *err) {
           if(!isEmptyBinTree(bp)) {
              *err = OK ;
              return bp -> parent ;
           }
           *err = ERROR;
           return NULL ;
}

void inorderTravel(binaryTree bp) {
     if(!isEmptyBinTree(bp)) {
        inorderTravel(bp -> leftChild) ;
        printf("*%d*", bp -> data) ;
        inorderTravel(bp -> rightChild) ;
     }
}

void preorderTravel(binaryTree bp) {
     if(!isEmptyBinTree(bp)) {
        printf("*%d*", bp -> data) ;
        preorderTravel(bp -> leftChild) ;
        preorderTravel(bp -> rightChild) ;
     }
}

void postorderTravel(binaryTree bp) {
     if(!isEmptyBinTree(bp)) {
        postorderTravel(bp -> leftChild) ;
        postorderTravel(bp -> rightChild) ;
        printf("*%d*", bp -> data) ;
     }
}

int leafCount(binaryTree bp) {
int  err ;
```

```
if(isEmptyBinTree(bp)) return 0 ;
if(isEmptyBinTree(rightSubTree(bp, &err)) &&
   isEmptyBinTree(leftSubTree(bp, &err))) return 1 ;
return leafCount(rightSubTree(bp, &err)) +
       leafCount(leftSubTree(bp, &err)) ;
}

binaryTree copyTree(binaryTree bp){
           if(isEmptyBinTree(bp)) return createNullTree();
           return makeBinTree(copyTree(bp->leftChild),
                              bp->data,
                              copyTree(bp->rightChild));
}

void levelOrderTravel(binaryTree bp){
     queue q;
     binaryTree bt;

     if(isEmptyBinTree(bp)) return ;
     init(&q);
     add(&q, bp);
     while(!isEmptyQ(q)){
           front(q, &bt);
           if(!isEmptyBinTree(bt -> leftChild))
              add(&q, bt->leftChild);
           if(!isEmptyBinTree(bt -> rightChild))
              add(&q, bt->rightChild);
           printf("*%d*", bt->data);
           delete(&q);
     }
}

// queue.h : queue on self-referencing structure with a
//           dummy node

#ifndef _QUEUE_H
#define _QUEUE_H

#include <stdio.h>
#include <stdlib.h>
#include "binTree.h"
#define ERROR 1
#define OK 0

struct queue {
       binaryTree data ;
       struct queue *next ;
};
typedef struct queue node;
typedef struct {
        struct queue *front, *rear ;
} queue ;

#define isEmptyQ(q) ((q).front == (q).rear)
void init(queue *) ;
int add(queue *, binaryTree) ;
int delete(queue *);
int front(queue, binaryTree *) ;
void printQ(queue);
#endif

// queue.c : queue on self referencing structure with a dummy node

#include "queue.h"
void init(queue *qP) {
     node *temp ;
     temp = (node *)malloc(sizeof(node)) ;
     qP->front=qP->rear=temp ;
}

int add(queue *qP, binaryTree n) {
    node *temp ;

    temp=(node *)malloc(sizeof(node)) ;
    temp->data=n; qP->rear->next=temp ;
    qP->rear = temp ;
    return OK ;
}

int front(queue q, binaryTree *v) {
    if(isEmptyQ(q)) return ERROR ;
    *v= q.front->next->data ;
    return OK ;
}

int delete(queue *qP) {
    node *temp ;

    if(isEmptyQ(*qP)) return ERROR ;
    temp = qP->front ;
    qP->front=qP->front->next ;
    free(temp) ;
    return OK ;
}

// testBinTree.c : testing binary tree functions

#include <stdio.h>
#include "binTree.h"
```

```
int main()
{
 binaryTree t, t1 ;

 t =  makeBinTree(
          makeBinTree(
              makeBinTree(
                  createNullTree(),
                  10,
                  createNullTree()
              ),
              5,
              createNullTree()
          ),
          7,
          makeBinTree(
              makeBinTree(
                  createNullTree(),
                  11,
                  makeBinTree(
                      createNullTree(),
                      23,
                      createNullTree()
                  )
              ),
              31,
              makeBinTree(
                  createNullTree(),
                  15,
                  createNullTree()
              )
          )
      );

 printf("preOrder traversal: ");
 preorderTravel(t) ;
 printf("\n") ;

 printf("inOrder traversal: ");
 inorderTravel(t) ;
 printf("\n") ;

 printf("postOrder traversal: ");
 postorderTravel(t) ;
 printf("\n") ;

 printf("levelOrder traversal: ");
 levelOrderTravel(t) ;
 printf("\n") ;
 printf("Leaf Nodes count: %d\n", leafCount(t));

 printf("tree copied: inOrder: ");
 t1 = copyTree(t);
 inorderTravel(t1) ;
 printf("\n") ;

 return 0 ;
}
```

A *make file* -

```
a.out: testBinTree.o binTree.o queue.o
cc testBinTree.o binTree.o queue.o

testBinTree.o: testBinTree.c binTree.h
cc -Wall -c testBinTree.c

binTree.o: binTree.c binTree.h queue.h
cc -Wall -c binTree.c

queue.o: queue.c binTree.h
cc -Wall -c queue.c

clean:
rm a.out testBinTree.o binTree.o queue.o
```

2 Binary Search Tree

A *dynamic set* is a collection of data where new data can be inserted and existing data can be deleted. Each data has a *key* and a data is uniquely identified by the *key*. It may be necessary to search data corresponding to a *key*. If the set of keys are totally ordered, it may be necessary to find data corresponding to the smallest and the largest key values. It may also necessary to find the successor or the predecessor of a *key*. Printing all elements of the set following the sequence of keys may also be required.

A *binary search tree (bst)* is a binary tree where data (*key*) is present in every node. The set of keys is *totally ordered*. For every node x in a binary search tree, the key values present in the *left subtree* are less than ($<$) the key present at x. Similarly, the key values present in the *right subtree* are greater than ($>$) the key present in x.

A binary search tree is called *height-balanced* if for each node x, the difference of heights between the *left subtree* and the *right subtree* is at most *one (1)*. But in general a binary search tree may not be height-balanced. It may even degenerate to a *linear chain* depending on data (the arrival order). The best and average heights h of an n node binary search tree is $\mathbf{O}(\log \mathbf{n})$. The worst case height of a binary search tree is $\mathbf{O}(\mathbf{n})$. A dynamic set can be stored in a *binary search tree*.

Type deceleration of a binary search tree node in C language is

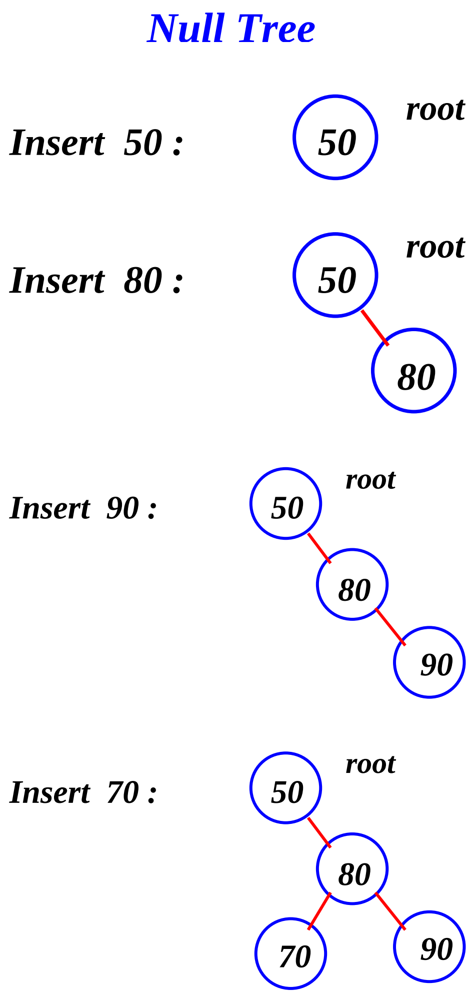
```
typedef struct binaryTreeNode {
       int data ;
       struct binaryTreeNode
              *leftChild, *rightChild, *parent ;
} *binStree;
```

1. Search in a binary search tree.

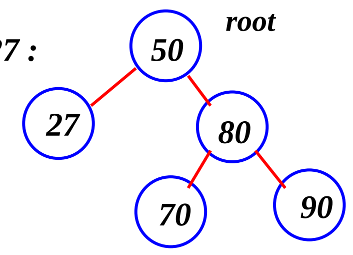
   ```
   binStree searchbSt(binStree t, int key) {
           if(isNullbSt(t)) return t ;
           if(t->data == key) return t ;
           if(key < t->data) return searchbSt(t->leftChild, key) ;
           return searchbSt(t->rightChild, key) ;
   }
   ```

   The time complexity for search is $O(h)$, where h is the height of the tree. As the worst case height is of $O(n)$, the search may take $O(n)$ steps in a skewed bst.

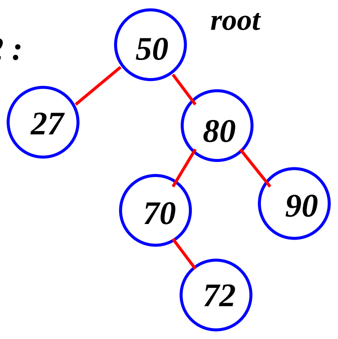
2. Insertion in a binary search tree.



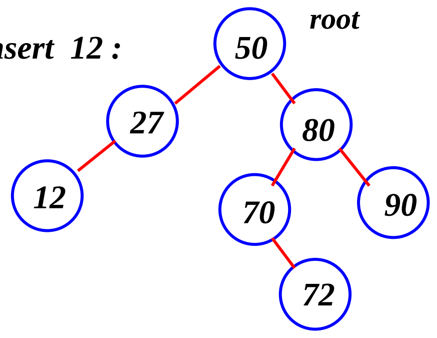
Insert 27 :



Insert 72 :

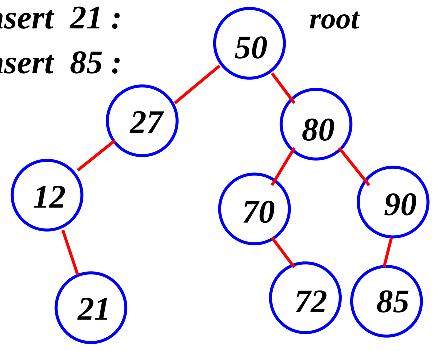


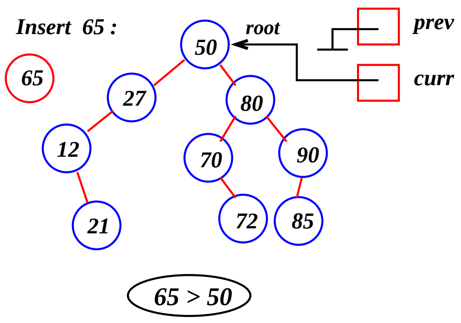
Insert 12 :

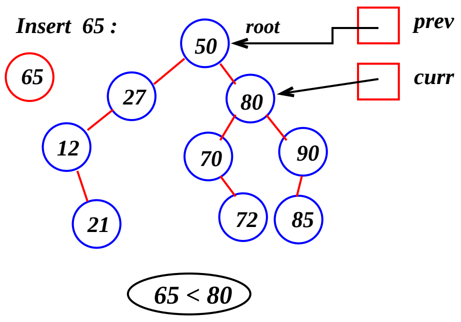


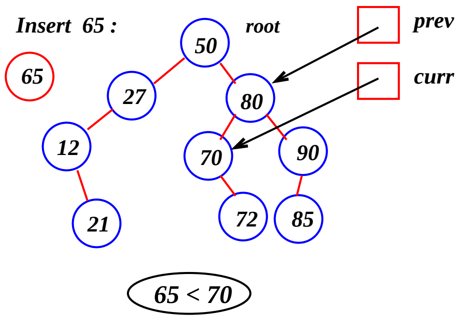
Insert 21 :

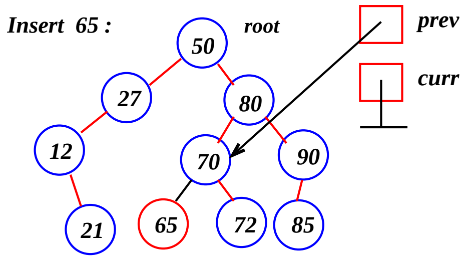
Insert 85 :







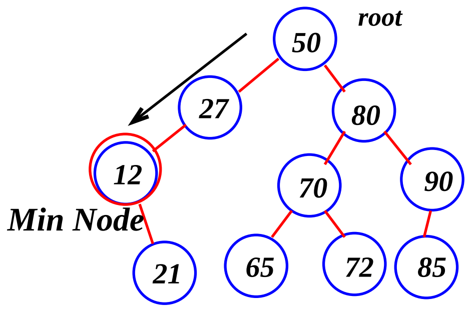




Left Child is Null

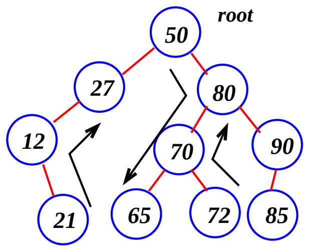
The time complexity of insert is $O(h)$, where h is the height of the tree. In worst case this can be $O(n)$, where n is the number of nodes in the tree.

3. The minimum key in the binary search tree

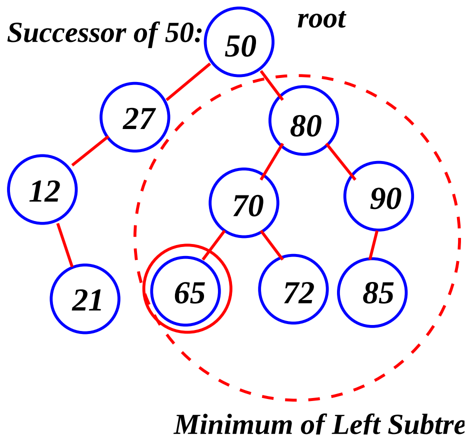


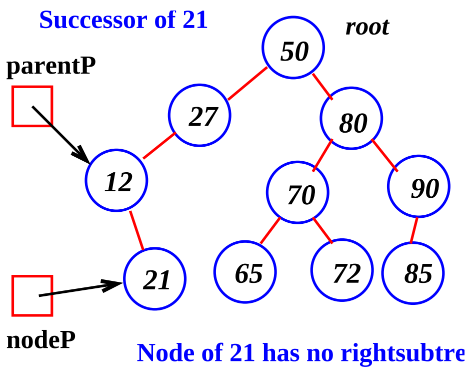
The time complexity is $O(h)$, where h is the height of the tree.

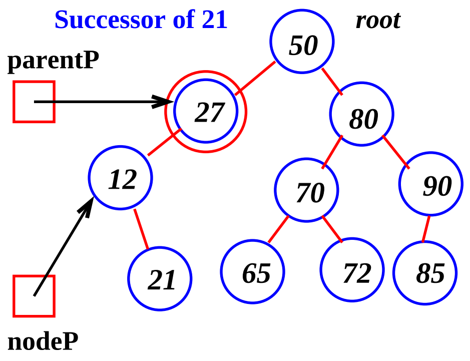
4. The *successor* of a node 'x' is the node 'y', if the key of the node 'y' follows immediately the key of the node 'x' in order. The successor is not present in the tree for the *largest* key. The *predecessor* of a node is defined similarly.



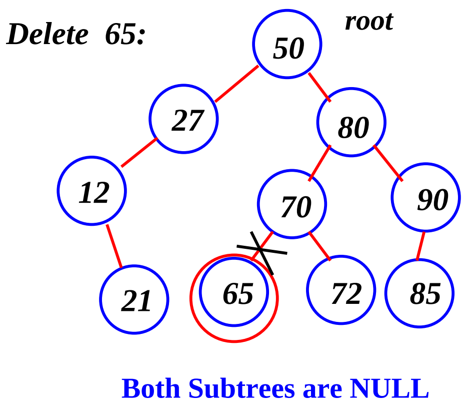
Successor(21) => 27,Successor(50) => 65,
Successor(72) => 80,No successor of 90

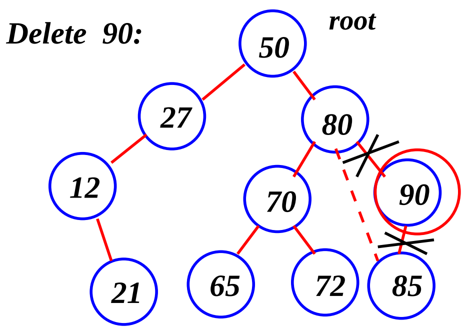




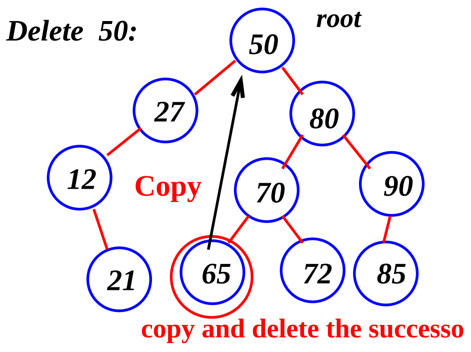


5. Delete a node

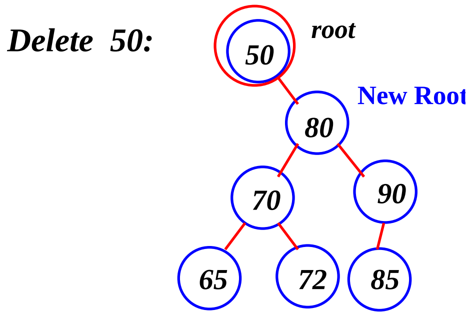




One Subtree is NULL



Both subtrees are present



The worst case time complexity of delete is $O(h)$, where h is the height of the tree, as it may be necessary to find the *successor* of the node to be deleted.

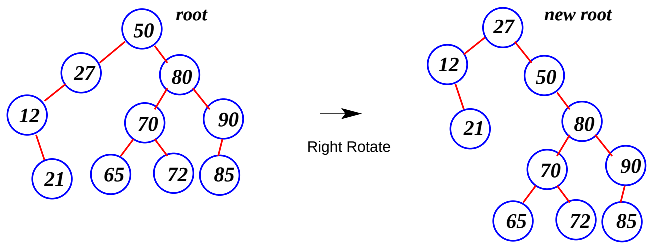
6. A *binary search tree* can be *rotated*. If the left subtree is non-null, the tree can be *rotated right (counter clockwise)* i.e.

   (a) The root of the left subtree will be the new root of the tree.

   (b) The right subtree of the left subtree will be the left subtree of the old root.

   (c) The old root will be the root of the right subtree of the new root.

   An example of a *binary search tree* after rotation.



7. Construct binary search tree of height 3, 4, 5, 6, and 7 with the following data: $\{1, 2, 3, 4, 5, 6, 7\}$.

8. A sorted sequence of the key values can be obtained from a binary-search tree by

   (a) inorder traversal, (b) preorder traversal, (c) postorder traversal, (d) label order traversal, (e) none

   Ans. (a)

9. The worst case time complexity to find the largest element from a binary search tree of n nodes is

   (a) $\Theta(n \log n)$, (b) $\Theta(n)$, (c) $\Theta(\log n)$, (d) $\Theta(1)$, (e) none

   Ans. (b)

10. The worst case time complexity to find the smallest element from a binary search tree of n nodes is

    (a) $\Theta(n \log n)$, (b) $\Theta(n)$, (c) $\Theta(\log n)$, (d) $\Theta(1)$, (e) none

    Ans. (b)

11. Given a set of 4 distinct integers, the number of possible binary search trees are

    (a) one (1), (b) four (4), (c) twelve (12), (d) fourteen (14), (e) none

    Ans. (d)

12. The data present in a binary search tree can be printed in the sorted order in time

    (a) $\Theta(n)$, (b) $O(n)$, (c) $\Theta(n \log n)$, (d) $O(\log n)$, (e) none

    Ans. (a) and (b)

```
/*
 * binStree.h is a header file of an implementation of
 * binary search tree of int data. The implementation file
 * is binStree.c.
 */

#ifndef _BINSTREE_H
#define _BINSTREE_H

#include <stdio.h>
#define ERROR 1
#define OK 0

typedef struct binaryTreeNode {
       int data ;
       struct binaryTreeNode
              *leftChild, *rightChild, *parent ;
} *binStree;

static inline void initbSt(binStree *tP) { *tP = NULL ;}
static inline int isNullbSt(binStree t) { return t == NULL ;}
static inline void printNode(binStree t) {
       if(isNullbSt(t)) return ;
       printf("%d ", t->data) ;
}
int insertbSt(binStree *, int) ;
int insertbSt1(binStree *, int) ;
int inorderbStTravel(binStree t) ;
binStree searchbSt(binStree, int) ;
binStree minimumbSt(binStree) ;
binStree successorbSt(binStree, int) ;
int deletebSt(binStree *, int) ;
binStree rotateRight(binStree bp);

#endif
```

```
/*
 * binStree.c is the implementation file for the
 * binary search tree. The header file is
 * binStree.h
 * Compile as $ cc -Wall -O1 -c binStree.c
 * The optimization is to be set on for inline functions
 * to expand.
 */

#include "binStree.h"
#include <stdlib.h>



int inorderbStTravel(binStree t) {
        int lcount, rcount ;

        if(isNullbSt(t)) return 0 ;
        lcount = inorderbStTravel(t->leftChild) ;
        printNode(t) ;
        rcount = inorderbStTravel(t->rightChild) ;
        return lcount + rcount + 1 ;
}


static int insertbStRec(binStree *tP, struct binaryTreeNode *nP) {
        binStree t ;

        if(isNullbSt(*tP)) { // Null tree
                *tP = nP ;
                nP->parent = NULL ;
                return OK ;
        }
        t = *tP ;
        if(t->data > nP->data){ // Insert in left subtree
                if(t->leftChild == NULL) { // Left subtree null
                        t->leftChild = nP ;
                        nP->parent = t;
                        return OK ;
                }
                else  return insertbStRec(&t->leftChild, nP) ;
        }
        else { // Insert in rightsubtree
                if(t->rightChild == NULL) { // Right subtree null
                        t->rightChild = nP ;
                        nP->parent = t;
                        return OK ;
                }
                else return insertbStRec(&t->rightChild, nP) ;
        }
}


int insertbSt(binStree *tP, int key) {
        struct binaryTreeNode *nP ;

        nP = (struct binaryTreeNode *)malloc(sizeof(struct binaryTreeNode));
        if(nP == NULL) {
                perror("malloc error") ;
                exit(0) ;
        }
        nP->leftChild = nP->rightChild = NULL ;
        nP->data = key ;
        return insertbStRec(tP, nP) ;
}

int insertbSt1(binStree *tP, int key) {
        struct binaryTreeNode *nP, *prev, *curr ;

        nP = (struct binaryTreeNode *)malloc(sizeof(struct binaryTreeNode));
        if(nP == NULL) {
                perror("malloc error") ;
                exit(0) ;
        }
        nP->leftChild = nP->rightChild = NULL ;
        nP->data = key ;

        prev = NULL ;
        curr = *tP ;
        while(curr) {
                prev = curr ;
                if(key <= curr->data) curr = curr->leftChild ;
                else curr = curr -> rightChild ;
        }

        nP->parent = prev ;
        if(prev == NULL) *tP = nP ;
        else
                if(key <= prev->data) prev->leftChild = nP ;
                else prev->rightChild = nP ;

        return OK ;
}

binStree searchbSt(binStree t, int key) {
        if(isNullbSt(t)) return t ;
        if(t->data == key) return t ;
        if(key < t->data) return searchbSt(t->leftChild, key) ;
        return searchbSt(t->rightChild, key) ;
}
```

```
binStree minimumbSt(binStree t) {
        if(isNullbSt(t)) return t ;
        while(t->leftChild) t = t->leftChild ;
        return t ;
}

binStree successorbSt(binStree t, int key) {
        binStree parentP, nodeP = searchbSt(t, key) ;

        if(nodeP == NULL) return NULL ;

        if(nodeP->rightChild != NULL)
                return minimumbSt(nodeP->rightChild) ;

        parentP = nodeP->parent ;
        while(parentP != NULL && nodeP == parentP->rightChild) {
                nodeP = parentP ;
                parentP = nodeP -> parent ;
        }
        return parentP ;
}


int deletebSt(binStree *tP, int key) {
        binStree t = *tP, delP, nodeP, lrP ;

        delP = searchbSt(t, key) ;  // Find the node

        nodeP = delP ;
        if(nodeP == NULL) return ERROR ;

/* If both left and right children are present, the successor of the
 * node is physically deleted. The key of the successor node is
 * copied to the node pointed by ‘delP’.
 */
        if(nodeP->leftChild != NULL && nodeP->rightChild != NULL)
                nodeP = successorbSt(t, key) ;

/* ‘nodeP’ points to a node whose at least one children is NULL.
 * The other child (it may also be NULL) is pointed by ‘lrP’.
 */
        if(nodeP->leftChild != NULL) lrP = nodeP->leftChild ;
        else lrP = nodeP -> rightChild ;

        if(lrP != NULL)  lrP->parent = nodeP->parent ;

        if(nodeP->parent == NULL) *tP = lrP ; // Root is deleted
        else
                if(nodeP == nodeP->parent->leftChild) // Left child deleted
```

```
                        nodeP->parent->leftChild = lrP ;
                else nodeP->parent->rightChild = lrP ; // Right child deleted

        delP->data = nodeP->data ; // Data copy from deleted node
        free(nodeP) ;

        return OK ;
}

binStree rotateRight(binStree bp){
     binStree tp;

     if(isNullbSt(bp) ||
        isNullbSt(bp->leftChild)) return bp;
     tp = bp->leftChild;
     bp->leftChild = tp->rightChild;
     tp->rightChild = bp;
     return tp;
}
```

```
/*
 * testBinStree.c is the main program to test the
 * binary search tree. The header file is binStree.h
 * The object module is binStree.o
 * Compile the executable module as
 * $ cc -Wall -O1 testBinStree.c binStree.o
 */

#include "binStree.h"

int main() {
        binStree t ;

        initbSt(&t) ;
        inorderbStTravel(t) ;
        printf("\n") ;

        insertbSt1(&t, 50) ;
        insertbSt1(&t, 15) ;
        insertbSt1(&t, 60) ;
        insertbSt1(&t, 90) ;
        insertbSt1(&t, 25) ;
        insertbSt1(&t, 35) ;
        insertbSt1(&t, 45) ;
        insertbSt1(&t, 75) ;
        insertbSt1(&t, 10) ;
        insertbSt1(&t, 80) ;
        insertbSt1(&t, 100) ;
        insertbSt1(&t, 5) ;
        inorderbStTravel(t) ;
        printf("\n") ;

//        t = rotateRight(t);
//        printf("After right rotation:\n");
//        inorderbStTravel(t) ;
//        printf("\n") ;

        printf("Successor of %d is ", 60) ;
        printNode(successorbSt(t,60)) ;
        printf("\n") ;

        deletebSt(&t, 50) ;
        inorderbStTravel(t) ;
        printf("\n") ;

        return 0;
}
```

References

[EHSSDM] *Fundamentals of Data Structures in C++*, by Ellis Horowitz, Sartaj Sahni and Dinesh Mehta, Galgotia Pub. Pvt. Ltd., 2009, ISBN 81-751-278-8.