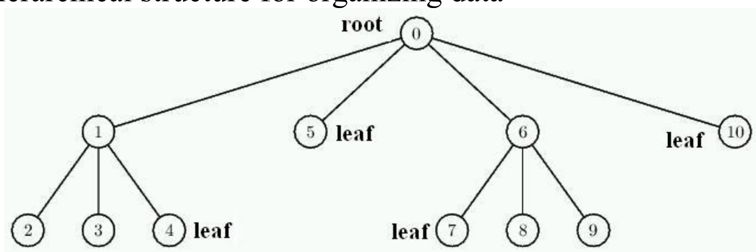
CSCI 3110
Tree, Binary tree, Binary search tree

A tree is an undirected simple graph *G* that is connected and has no simple cycles.

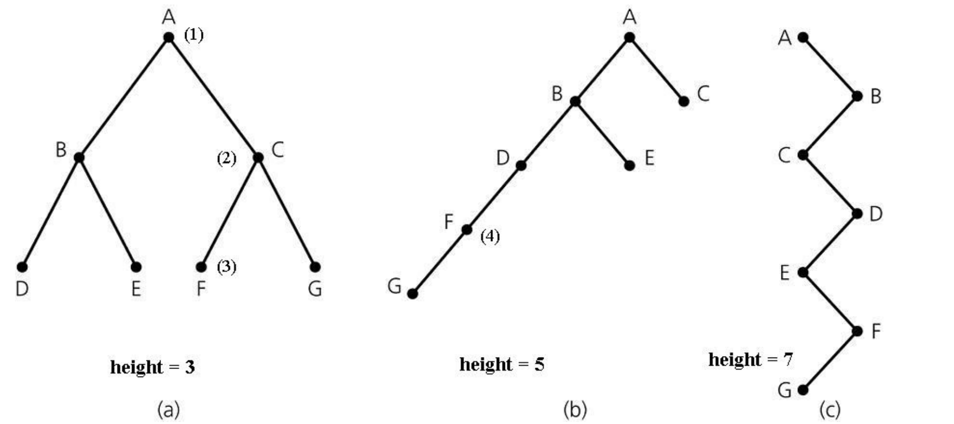
- A graph is connected if there is at least one path between any two vertices
- It is a hierarchical structure for organizing data

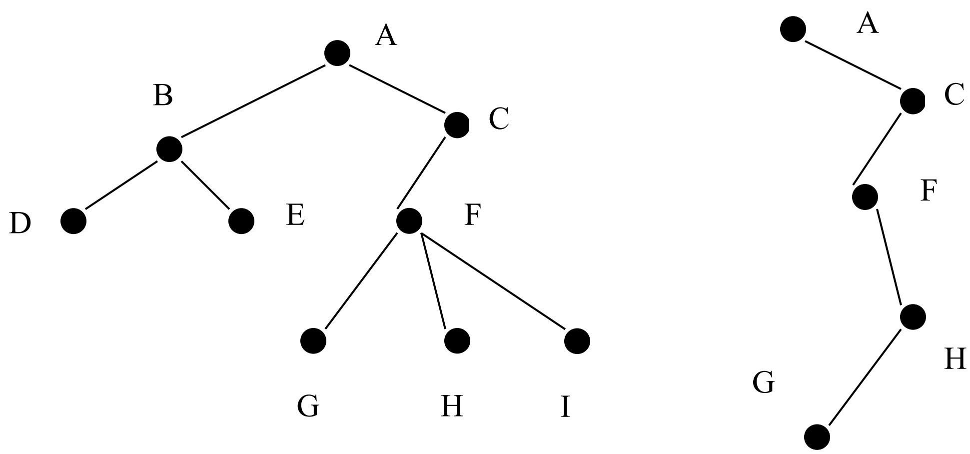


Tree terminologies:

- Parent of node *n*
  - The node directly above node *n* in the tree
- Child of node *n*
  - a node directly below node *n* in the tree
- Root
  - The only node in the tree with no parent
  - When referring to a tree, we need to specify its root. A graph can be viewed as different trees by choosing different vertex as the root
- Leaf
  - A node with no children
- Siblings
  - Nodes with a common parent
- Ancestor of node *n*
  - A node on the path from the root to *n*
- Descendant of node *n*
  - A node on the path from *n* to a leaf
- Degree of a node
  - Number of child nodes for the node
- Subtree : any node in the tree together with all its descendants
- Subtree of a node n : subtree rooted at a child of n
- Height of a tree
  - The number of nodes on the longest path from the root to a leaf
- Level of a node *n*
  - If *n* is the root, it is at level 1
  - Otherwise, its level is 1 greater than the level of its parent.

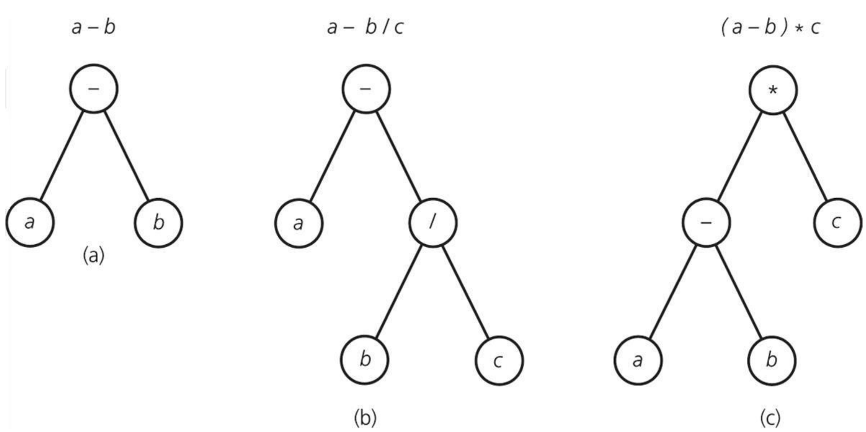
Examples of these concepts in the following tree?



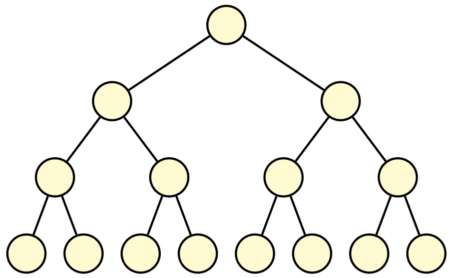


Binary tree – tree where each node has at most 2 child nodes, called left child (if any) and right child (if any).

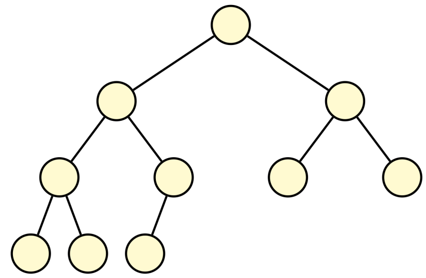
- Equivalently, each node in a binary tree has at most two subtrees, called left subtree T_L (if any) and right subtree T_R (if any).



Perfect Binary Tree



Complete Binary Tree

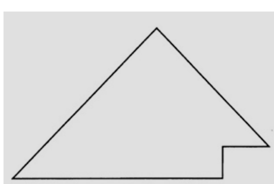


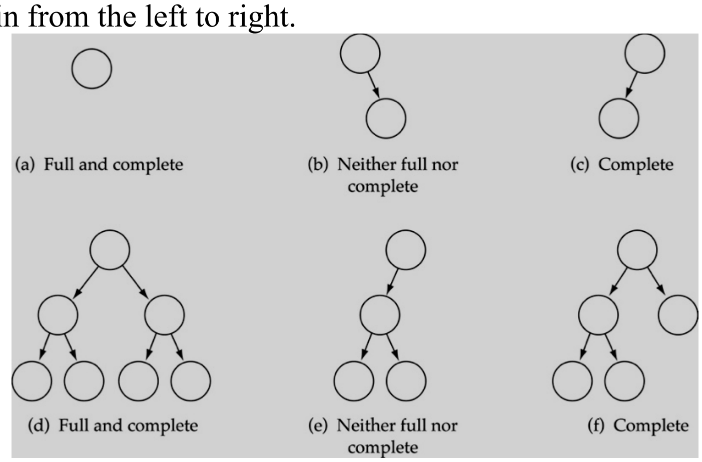
Perfect binary tree - a binary tree of height h, where all nodes at levels less than h have 2 children nodes

- How many nodes are in a perfect binary tree of height h?
  $2^0+2^1+2^2+2^3....2^{(h-1)}=2^h-1$
- Given a perfect binary tree with N nodes, what is the height of the tree?
  $2^h-1 = N \quad ➔ \quad h = \log2(N+1)$

Complete binary tree:

- a binary tree that is either full or full through the next to last level
- The last level is filled in from the left to right.



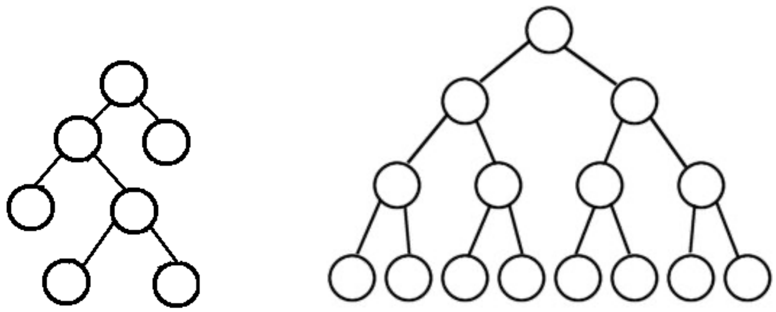


Complete binary tree : binary tree full down the level (h-1), with level h filled in from left to right

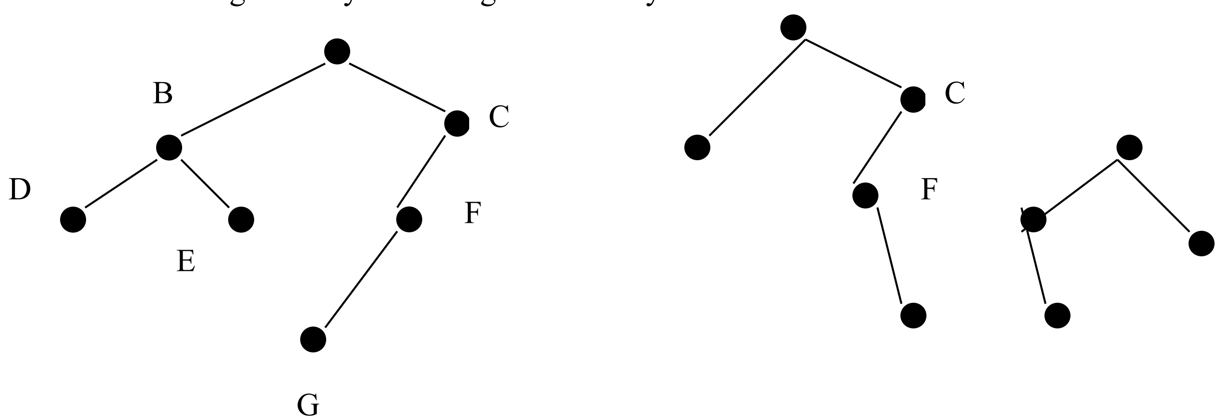
- All nodes at level <= h-2 have 2 children
- When a node at level h-1has children, all nodes to its left at the same level have 2 children each
- When a node at level h-1 has one child, it is a left child

Full binary tree:

- a binary tree in which **all of the nodes have either 0 or 2 offspring**, and
- a binary tree in which all nodes, except the leaf nodes, have two offspring.



Balanced binary tree: binary tree is balanced, if the height of any node's left subtree differs from the height of any node's right subtree by no more than 1



If a binary tree is perfect, is it always balanced?
If a binary tree is complete, is it always balanced?
If a binary tree is full, is it always balanced?
If a binary tree is balanced, is it always full?
If a binary tree is balanced, is it always complete?

Operations of a binary tree

- Create an empty binary tree BinaryTree(); // default constructor
  Create a one-node(the root) binary tree BinaryTree(const TreeItemType root);

- Create a binary tree given the root, and its subtrees
  BinaryTree(const TreeItemType root, BinaryTree left, BinaryTree right);

- Destroy a binary tree
  ~BinaryTree();

- Determine whether a binary tree is empty
  isEmpty();

- Determine or change the data in the binary tree's root
  TreeItemType getRootData();
  void setRootData(TreeItermType data);

- Attach a left or right node to the binary tree's root
  void attachLeft(TreeItemType item) ;
  void attachRight(TreeItermType item);

- Detach a left or right subtree of the binary tree's root
  void detachLeftSubtree(BinaryTree& leftTree);
  void detachRightSubTree(BinaryTree& rightTree);

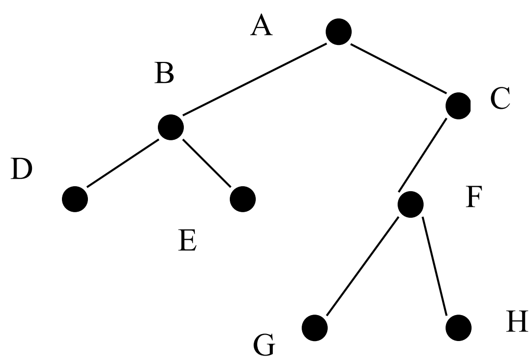
- Return a copy of the left or right subtree of the binary tree's root
  BinaryTree leftSubtree(void);
  BinaryTree rightSubTree(void);

- Traverse the nodes in a binary tree
  preorder: void preorderTraverse(FunctionType visit);
  inorder: void inorderTraverse(FunctionType visit);
  postorder: void postorderTraverse(FunctionType visit);

Traversal of a binary tree

A traversal algorithm for a binary tree visits each node in the tree

- Preorder traversal : visit root first, then left child, than right child (ABDECFGHI)
- Inorder traversal : visit left child first, then visit root, then visit right child
- Postorder traversal : visit left node, then visit right child node, then visit root



Preorder traversal (used to copy a tree)

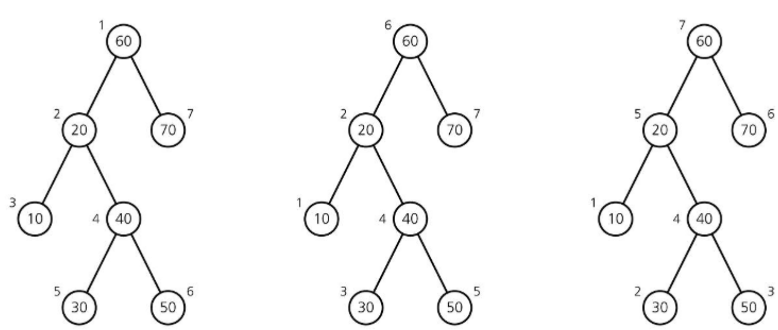
```
preorder ( BinaryTree tree, FunctionType visit ) {
   if ( tree is not empty ) {
      visit(the root of tree);
      preorder(Left subtree of tree's root, visit);
      preorder(Right subtree of tree's root, visit);
   }
}
```

Inorder traversal (used to visit nodes in ascending order of keys, to extract inorder expression)

```
inorder ( BinaryTree tree, FunctionType visit ) {
   if ( tree is not empty ) {
      inorder(Left subtree of tree's root, visit);
      visit(the root of tree);
      inorder(Right subtree of tree's root, visit);
   }
}
```

Postorder traversal (used to extract postorder expression from a BST)

```
postorder ( BinaryTree tree, FunctionType visit ) {
   if ( tree is not empty ) {
      postorder(Left subtree of tree's root, visit);
      postorder(Right subtree of tree's root, visit);
      visit(the root of tree);
   }
}
```



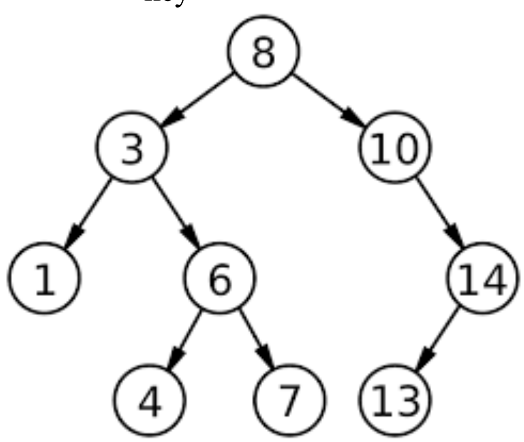
(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

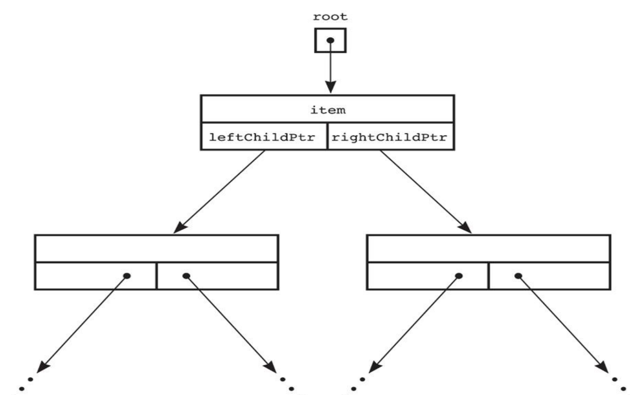
(Numbers beside nodes indicate traversal order.)

The function visit is provided and passed by clients as the computation performed for each node.

A Binary Search Tree (BST)

- A binary tree that has the following properties for each node *n*
  - *n*'s value is greater than all values in its left subtree T_L
  - *n*'s value is less than all values in its right subtree T_R
  - Both T_L and T_R are binary search trees
- A deficiency of the ADT binary tree which is corrected by the ADT binary search tree
  - Searching for a particular item
- A data item in a binary search tree has a specially designated search key
  - A search key is the part of a record that identifies it within a collection of records
- *KeyedItem* class
  - Contains the search key as a data field and a function for accessing the search key





Defining a Binary Search Tree :

```
struct treeNode;
typedef treeNode * ptrType;

struct treeNode
{  treeItemType Item;
   ptrType      LChildPtr, RChildPtr;
   treeNode(const treeItemType& NodeItem, ptrType L, ptrType R);
}; // end struct

treeNode::treeNode(const treeItemType& NodeItem, ptrType L, ptrType R):
Item(NodeItem), LChildPtr(L), RChildPtr(R)
{
}
```

Binary search tree traversal examples:

1. Binary Search Tree Operation: **Search**
   - A recursive search function
     - Searches the binary search tree *bst* for the item whose search key is *key*.
     - Searches recursively in the right or left subtree (depending on the value of the item) until *key* matches the search key of the node's item

```
Search ( BinarySearchTree bst, KeyType key)
{
        if ( bst is empty )
                The desired record is not found, throw an exception
        else if (key == search key of root's item)
                The desired record is found
        else if ( key < search key of root's item)
                Search(left subtree of bst, key)
        else // key > search key of root's item
                Search(left subtree of bst, key)
}
```

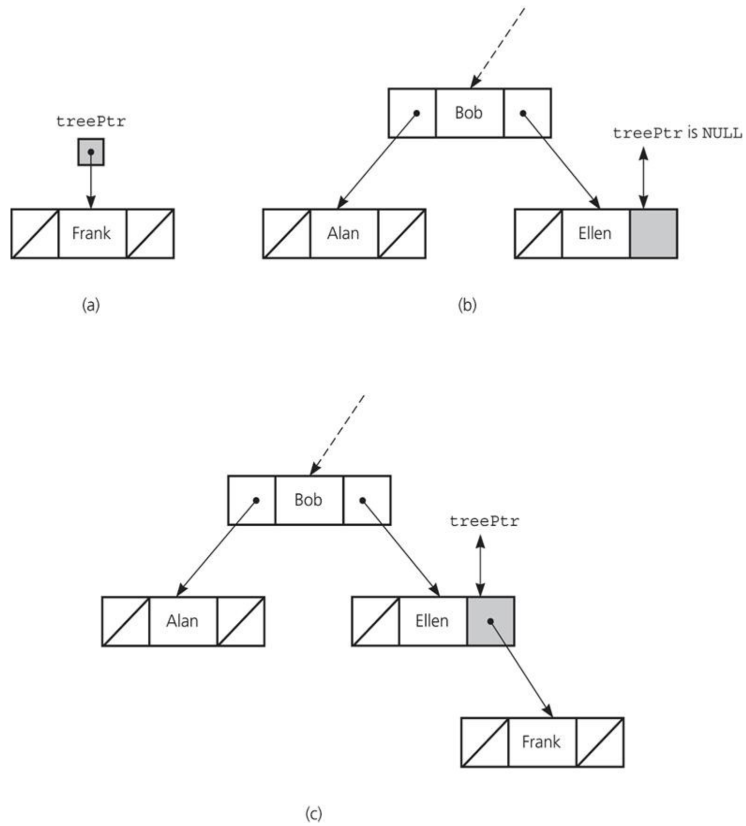
2. Binary Search Tree Operation: **Insert**

   Inserts *newItem* into the binary search tree to which *treePtr* points

```
InsertItem( TreeNodePtr treePtr, TreeItemType newItem )
// Question: Should treePtr be passed by value or by reference????
{
    if ( treePtr == nullptr )
            Create a new Node, Initialize it appropriately, and let treePtr point
            to it.
    else if ( newItem.getKey() < treePtr→item.getKey() )
            InsertItem( treePtr→leftChildPtr, newItem );
    else
            InsertItem( treePtr→rightChildPtr, newItem );
}
```

How are these functions called?
If these functions are member functions of a class, how should them be called?
Why is it necessary to have a <u>wrapper function</u> in these cases?

?? How to write this function as a member function of BST class ??
Ans: a wrapper function is needed to deal with recursive member function that require the access of the private data, the root, of the class



(a) Insertion into an empty tree; (b) search terminates at a leaf; (c) insertion at a leaf

<u>Practice Question:</u>
How to build a binary search tree by inserting the following key values one by one into an empty tree ? 45, 23, 100, 80, 56, 87, 5, 25

3. Binary Search Tree Operation: **Delete**

- Three possible cases for deleting a node N
    1. N is a leaf
        - Set the pointer in N's parent to nullptr (or NULL)
    2. Delete a node with only left child
    3. Delete a node with only right child
    4. N has two children
        - Locate another node M that is the leftmost node in N's right subtree
          M's search key is called the in-order successor of N's search key
        - Copy the item that is in node M to node N
        - Update the pointer to M to point to M's right subtree.
        - Remove the node M from the tree

4. How to determine the level of each node in a binary (search) tree? (use pre-order traversal)

function call: AssignLevel(**root**, 1); // assuming root at level 1.
function definition:

```
void  AssignLevel(ptrType   nodePtr, int level)
{     if (nodePtr != nullptr)
      {   cout << "Node with key" << nodePtr→Item.Key()
               << "is at level " << level << endl;
      nodePtr→PutLevel(level);
          AssignLevel(nodePtr→LChildPtr, level+1);
          AssignLevel(nodePtr→RChildPtr, level+1);
      }
}
```

When defined in the BST class as a method:

define the method:

```
void  BST::AssignLevel(ptrType   nodePtr, int level)
{     if (nodePtr != nullptr)
      {   cout << "Node with key" << nodePtr→Item.Key()
               << "is at level " << level << endl;
          nodePtr→Level = level;
          AssignLevel(nodePtr→LChildPtr, level+1);
          AssignLevel(nodePtr→RChildPtr, level+1);
      }
}
void BST::DetermineLevel()
{
      AssignLevel(root, 1);
}
```

call the method: OneBSTree.DetermineLevel();

5. How to determine the height of a binary (search) tree? (use post-order traversal)
function call: height = FindHeight(root);
function definition:

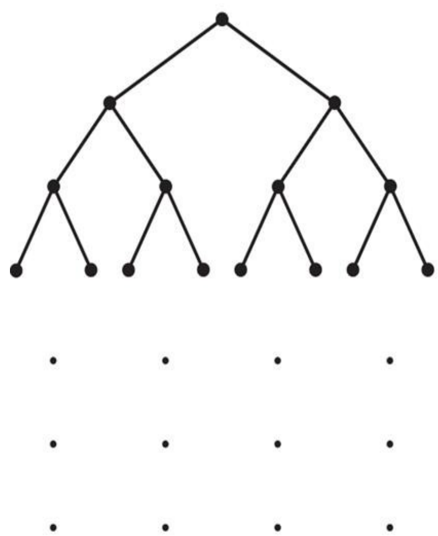
```
int FindHeight(ptrType nodePtr)
{       int leftHeight, rightHeight;
        if (nodePtr != NULL)
        {       leftHeight = FindHeight(nodePtr→LChildPtr)+1;
                rightHeight = FindHeight(nodePtr→RChildPtr)+1;
                if (leftHeight < rightHeight)
                        return rightHeight;
                else
                        return leftHeight;
        }
        else
                return 0;
}
```

6. **Saving and restoring binary search tree**
   - **Restoring the binary search tree to its <u>original shape</u>**
     (use pre-order traversal to save the tree to a file, then rebuild the tree by inserting the items one by one into an empty tree)

   - **Saving the binary search tree in order and <u>Restoring the binary search tree to the minimum height</u>**
     - Uses inorder traversal to save the tree to a file
     - Can be used if the data is sorted and the number of nodes in the tree is known

```
// save the records in a binary search tree in order
void   ReadSave(ofstream& outFile, ptrType nodePtr)
{
   if (nodePtr != nullptr)
   {
            ReadSave(outFile, nodePtr→LChildPtr);

            // Write information in nodePtr to outFile (overloaded << operator)
            outFile << nodePtr→Item << endl;

            ReadSave(outFile, nodePtr→RChildPtr);
   }
}
// restore the binary search tree to minimum height
void        Restore(ifstream & infile, ptrType & nodePtr, int numOfRecords)
{
   if (numOfRecords > 0)
   {
            nodePtr = new  treeNode;

            Restore (infile, nodePtr→LChildPtr, numOfRecords/2);

            infile >> newItem;   // overloaded >> operator
            nodePtr→Item = newItem // overloaded = operator

            Restore(infile, nodePtr→RChildPtr, (numOfRecords-1)/2);
   }
   else
            nodePtr = nullptr;
}
```

The efficiency of Binary Search Tree operations



Level	Number of nodes at this level	Number of nodes at this and previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
•	•	•
•	•	•
•	•	•
h	2^{h-1}	$2^h - 1$

Uses the ADT binary search tree to sort an array of records into search-key order

- Average case: $O(n * logn)$
- Worst case: $O(n^2)$