COM 201Data Structures and Algorithms

Binary Search Trees

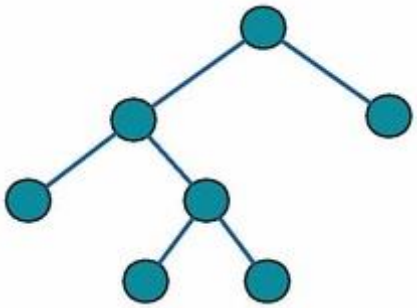
Asst. Prof. Dr. Özge Öztimur Karadağ

Previously

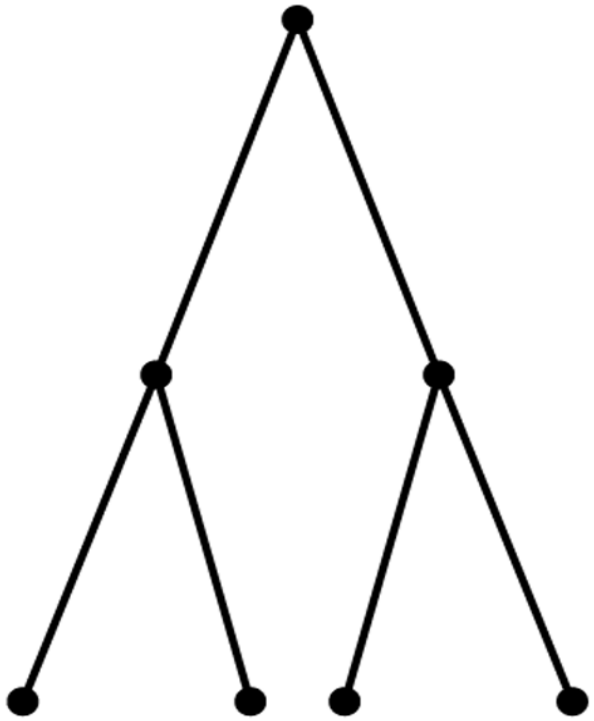
- Nonlinear data structure:
  - Tree
    - Tree Operations
    - Binary Tree

Full Binary Tree

- In a ***full binary tree*** a tree in which every node in the tree has either 0 or 2 children



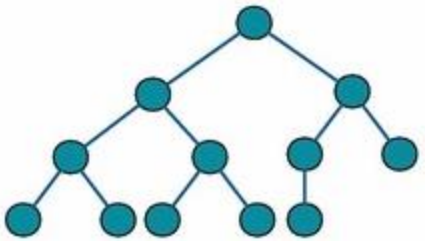
Full Binary Tree – Example



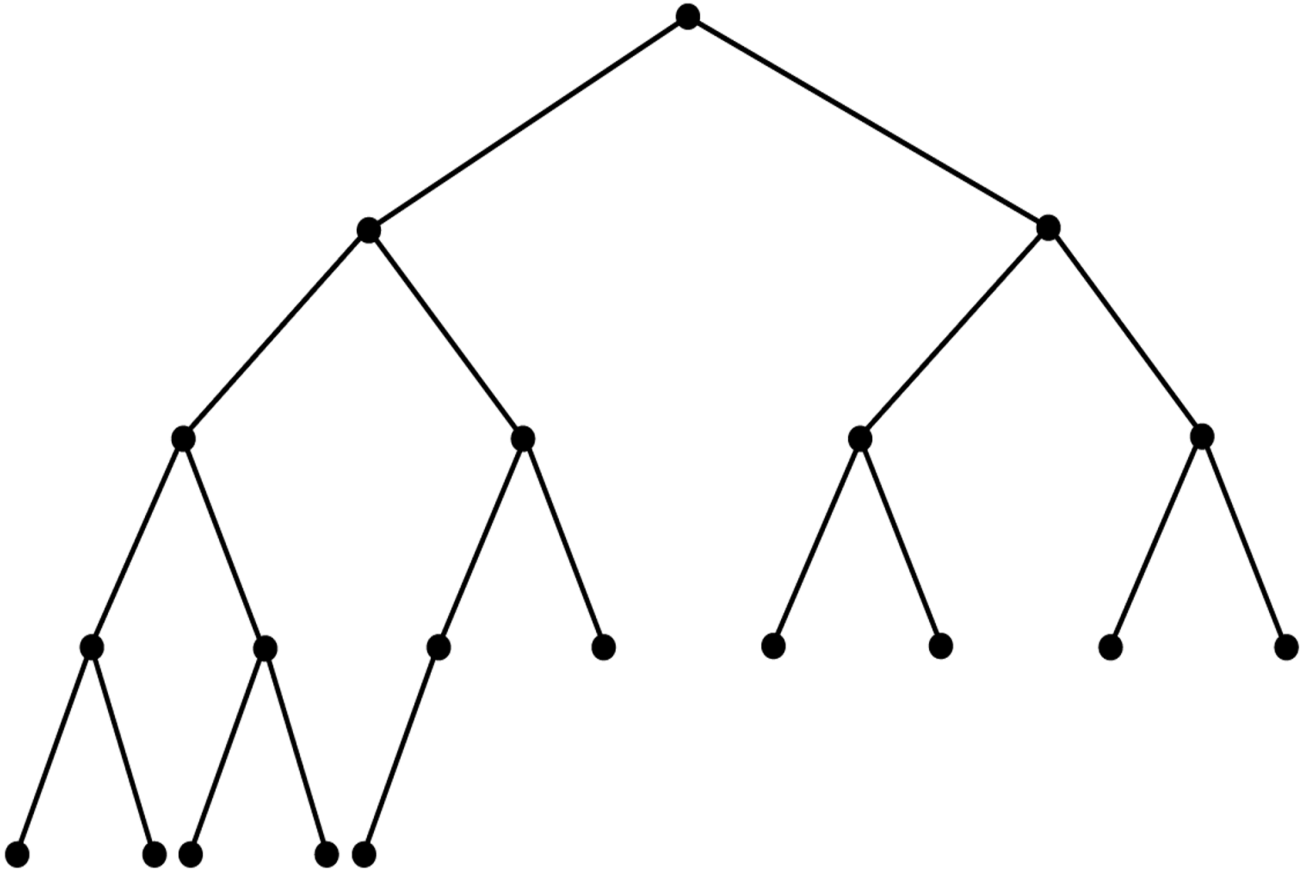
A full binary tree of height 2

Complete Binary Tree

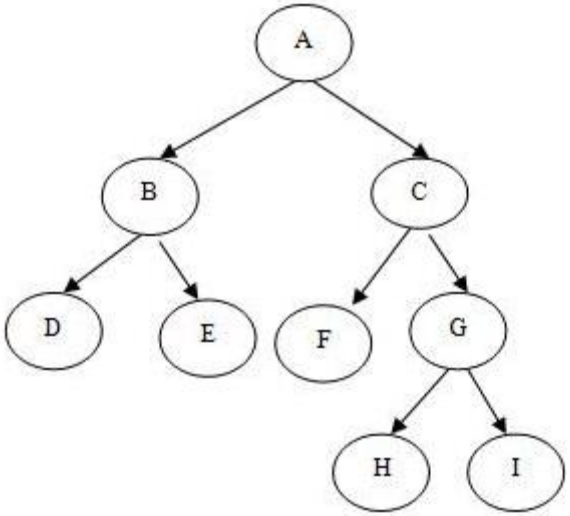
- In a ***complete binary tree*** every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible



Complete Binary Tree – Example

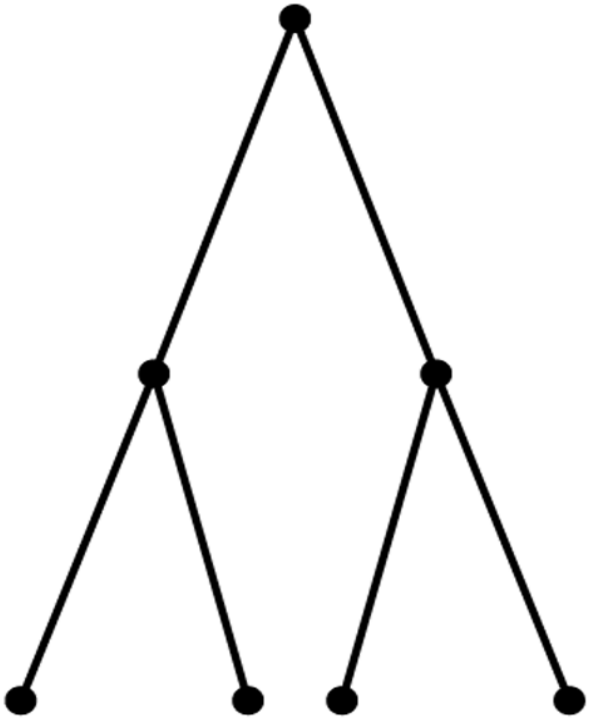


Full but Not Complete Binary Tree – Example



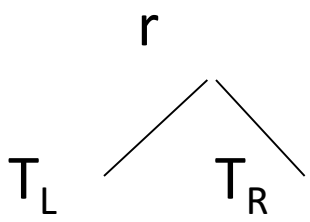
Neither Full Nor Complete Binary Tree – Example

Full and Complete Binary Tree – Example



Height of Binary Tree

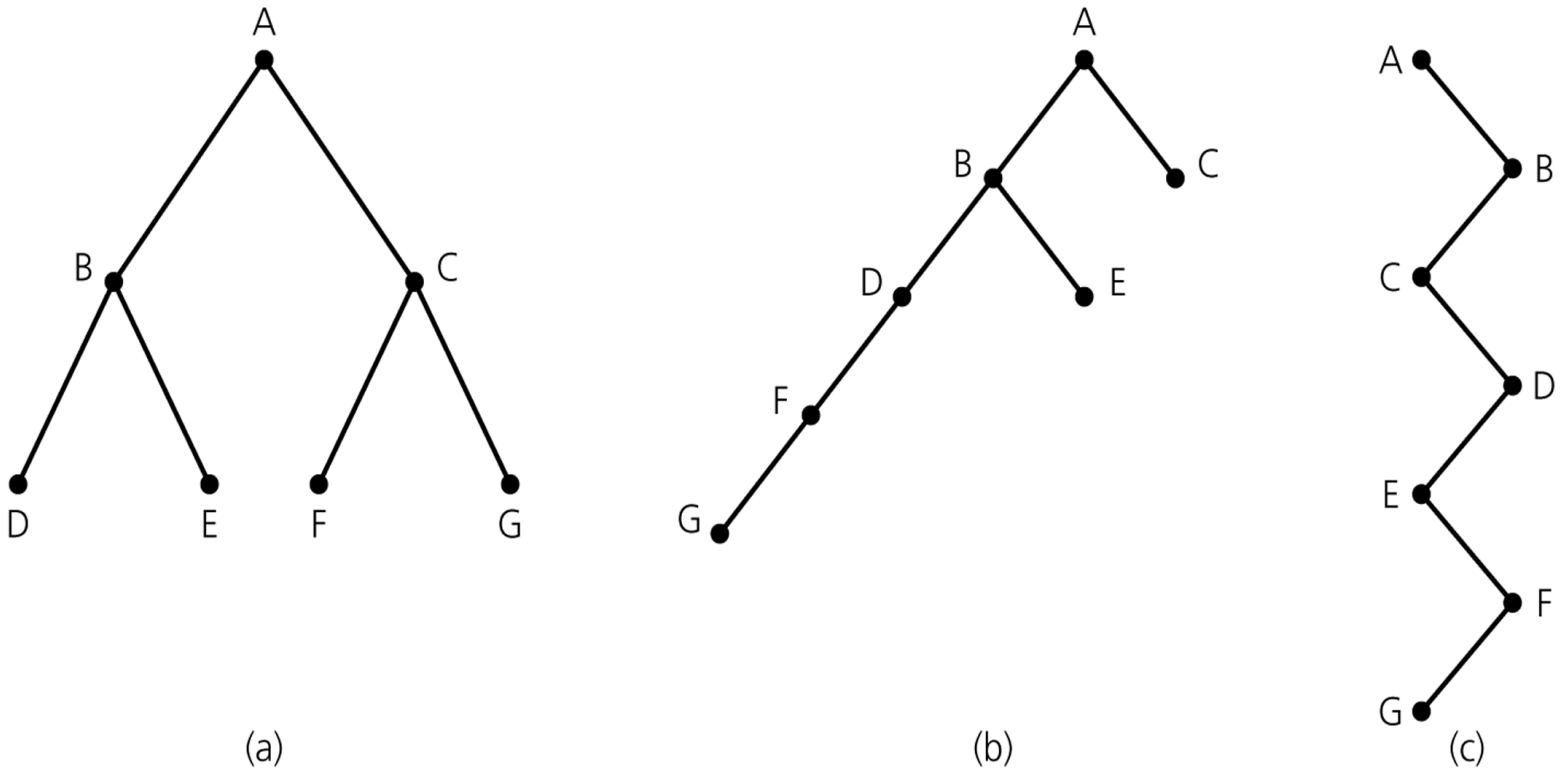
- The height of a binary tree T can be defined *recursively* as:
  - If T is empty, its height is -1.
  - If T is non-empty tree, then since T is of the form



the height of T is 1 greater than the height of its root's taller subtree; i.e.

$$height(T) = 1 + max\{height(T_L), height(T_R)\}$$

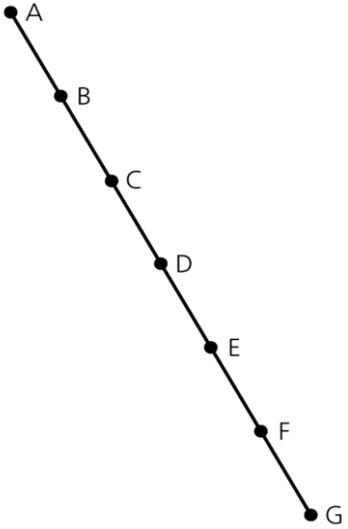
Height of Binary Tree (cont.)



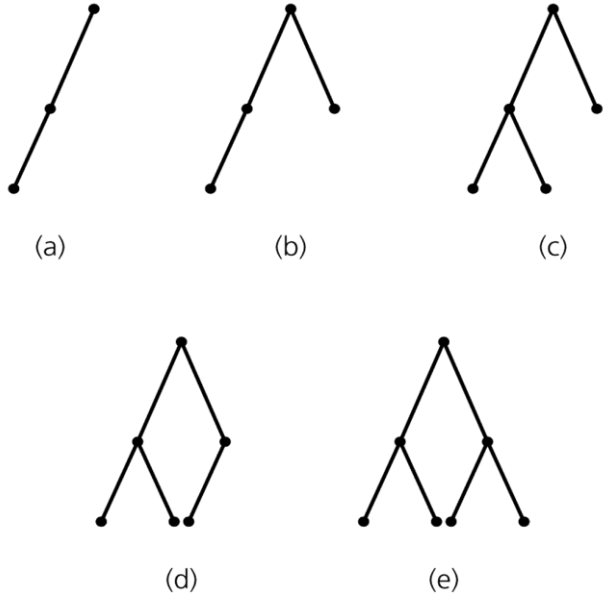
Binary trees with the same nodes but different heights

Maximum and Minimum Heights of a Binary Tree

- The maximum height of a binary tree with N nodes is N-1.
- Each level of a minimum height tree, except the last level, must contain as many nodes as possible



A maximum-height binary tree with seven nodes



Some binary trees of height 2

All but (a) is minimum-height

Some Height Theorems

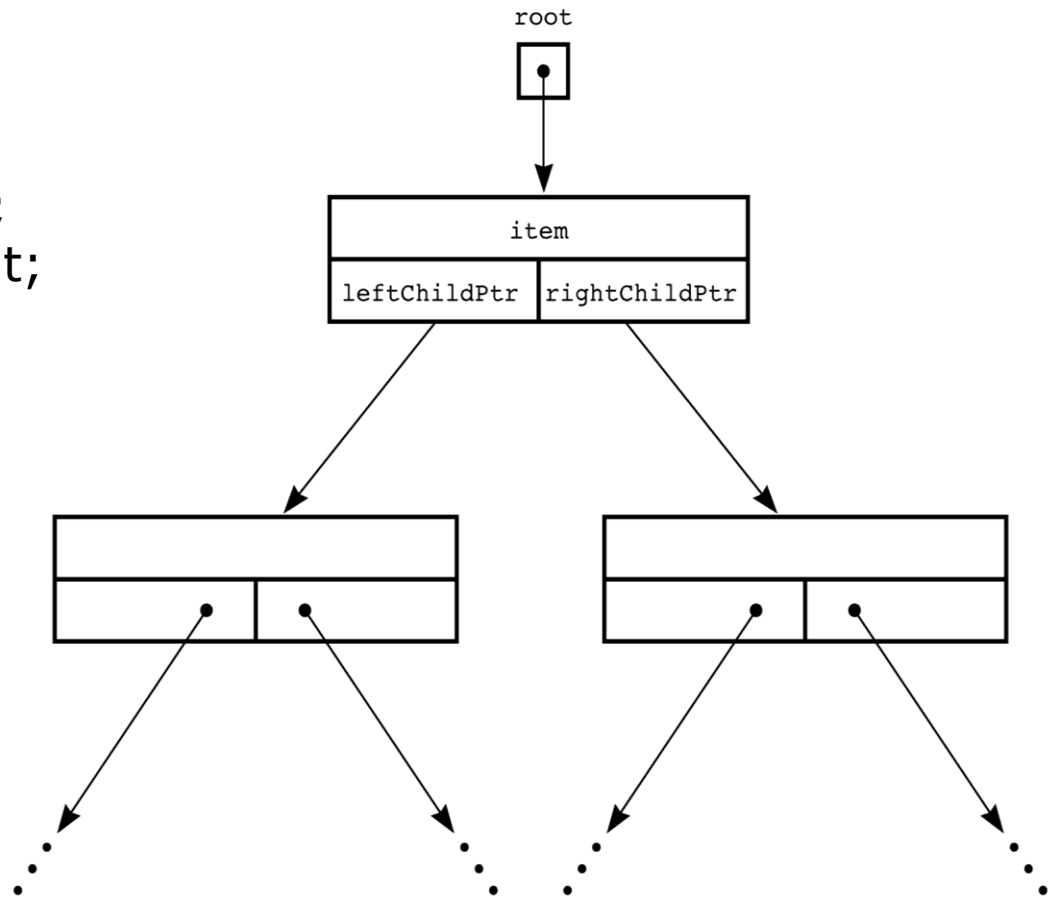
Theorem 1: The maximum number of nodes that a binary tree of height h can have is $2^{h+1}-1$.

Theorem 2: The minimum height of a binary tree with N nodes is $\lceil \log_2(N+1)-1 \rceil$.

Let's prove these two theorems.

A Pointer-Based Implementation of Binary Trees

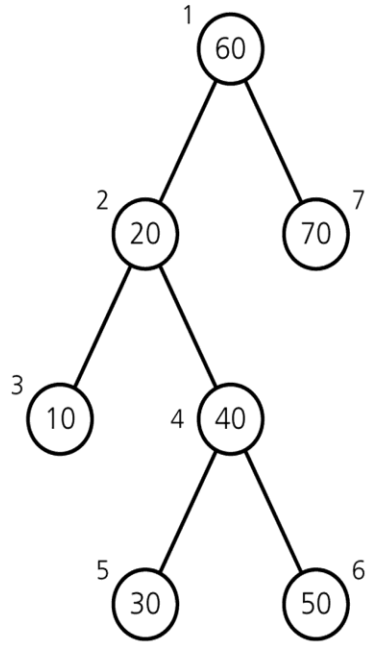
```
struct BinaryNode {
   Object          element;
   struct BinaryNode *left;
   struct BinaryNode *right;
};
```



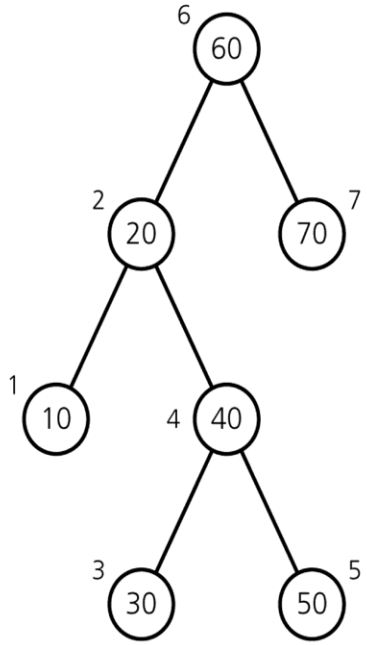
Binary Tree Traversals

- **Preorder Traversal**
  - the node is visited before its left and right subtrees,
- **Postorder Traversal**
  - the node is visited after both subtrees.
- **Inorder Traversal**
  - the node is visited between the subtrees,
  - Visit left subtree, visit the node, and visit the right subtree.

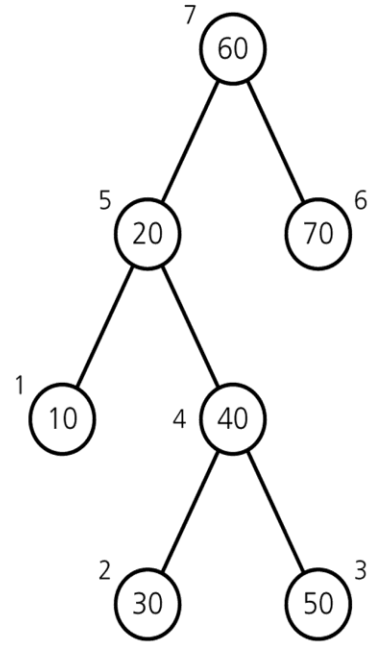
Binary Tree Traversals



(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.)

Preorder

```
void preorder(struct tree_node * p)
{   if (p !=NULL) {
    cout << p->data << endl;
    preorder(p->left_child);
    preorder(p->right_child);
}
}
```

Inorder

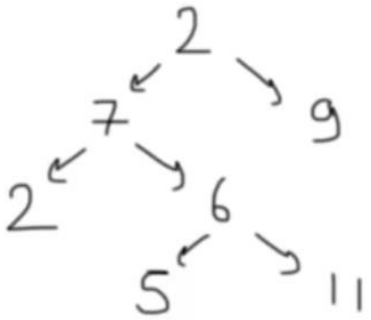
```
void inorder(struct tree_node *p)
{   if (p !=NULL) {
    inorder(p->left_child);
    cout << p->data << endl;
    inorder(p->right_child);
}
}
```

Postorder

```
void postorder(struct tree_node *p)
{   if (p !=NULL) {
    postorder(p->left_child);
    postorder(p->right_child);
    cout << p->data << endl;
  }
}
```

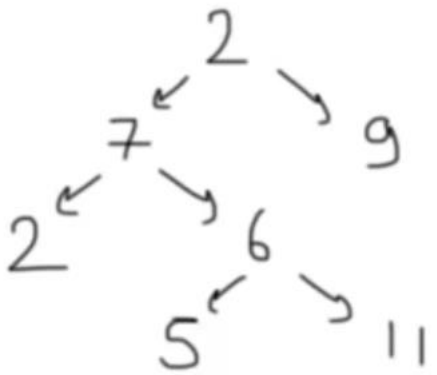
Finding the maximum value in a binary tree

```
int FindMax(struct tree_node *p)
{
}
```



Adding up all values in a Binary Tree

```
int add(struct tree_node *p)
{
}
```



Exercises

- Write a function that will count the leaves of a binary tree.

```
unsigned int getLeafCount(struct node* node)
{
  if(node == NULL)
    return 0;
  if(node->left == NULL && node->right==NULL)
    return 1;
  else
    return getLeafCount(node->left)+
           getLeafCount(node->right);
}
```

- Write a function that'll find the height of a BT.

```
int maxDepth(struct node* node)
{
   if (node==NULL)
       return 0;
   else
   {
       /* compute the depth of each subtree */
       int lDepth = maxDepth(node->left);
       int rDepth = maxDepth(node->right);

       /* use the larger one */
       if (lDepth > rDepth)
           return(lDepth+1);
       else return(rDepth+1);
   }
}
```

Binary Search Trees

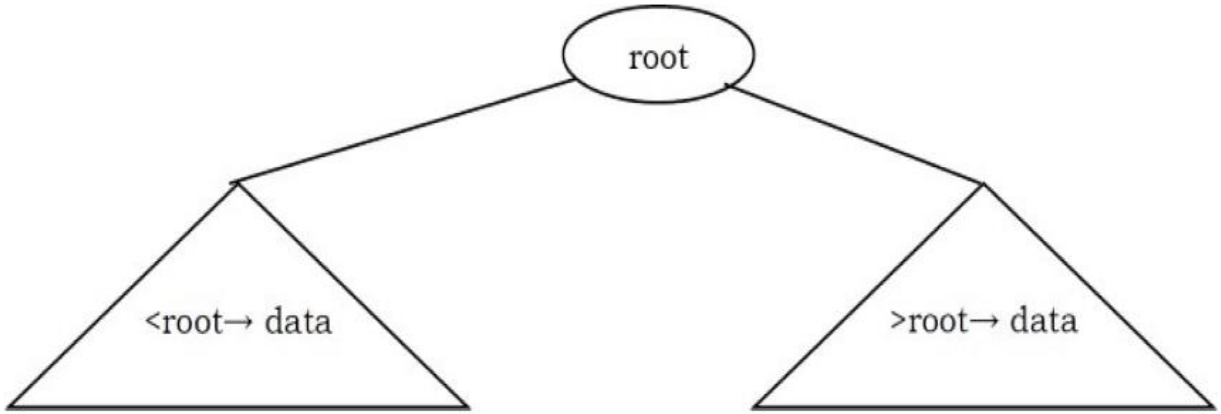
- An important application of binary trees is their use in searching.
- ***Binary search tree*** is a binary tree in which every node X contains a data value that satisfies the following:
  a) all data values in its left subtree are smaller than the data value in X
  b) the data value in X is smaller than all the values in its right subtree.
  c) the left and right subtrees are also binary search tees.

Binary Search Tree Property

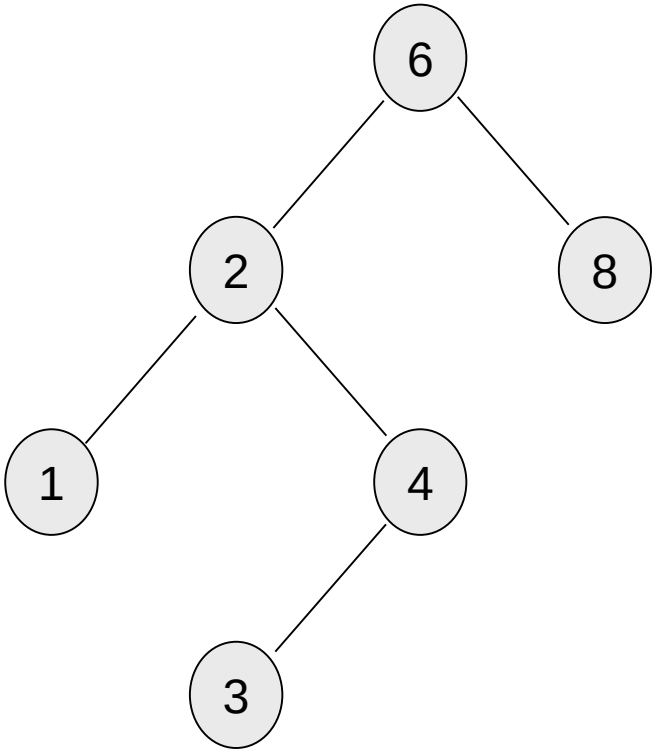
In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called binary search tree property. Note that, this property should be satisfied at every node in the tree.

- The left subtree of a node contains only nodes with keys less than the nodes key.
- The right subtree of a node contains only nodes with keys greater than the nodes key.
- Both the left and right subtrees must also be binary search trees.

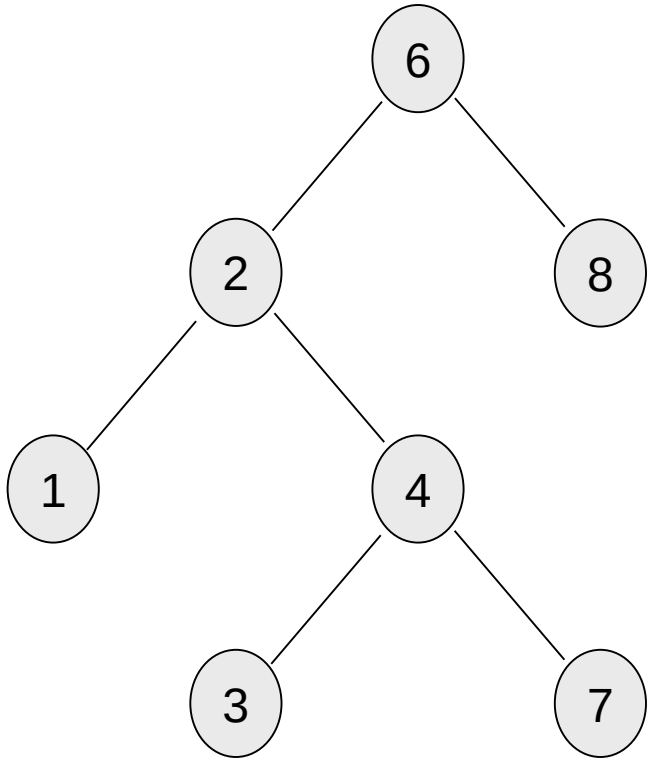
Binary Search Tree Property



Example

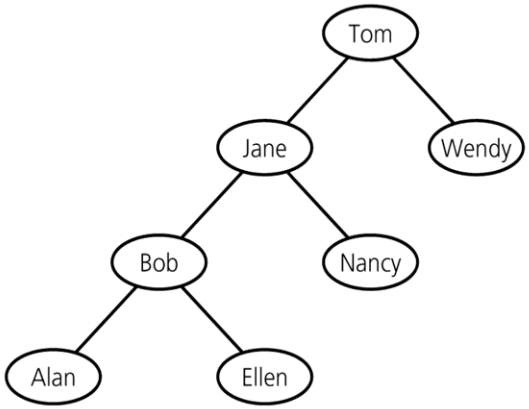


A *binary search tree*

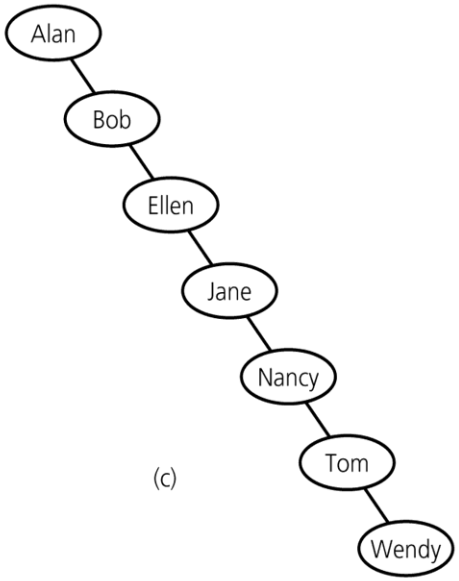


Not a *binary search tree,* but a *binary tree*

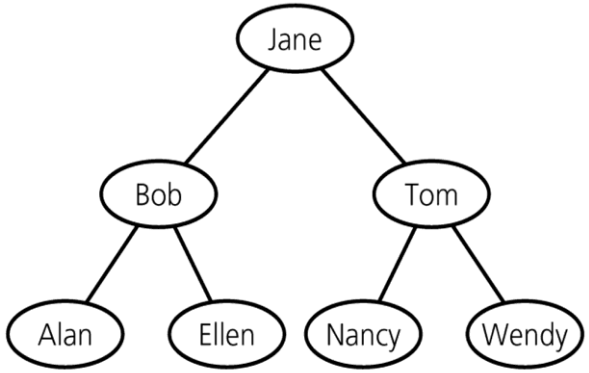
Binary Search Trees – containing same data

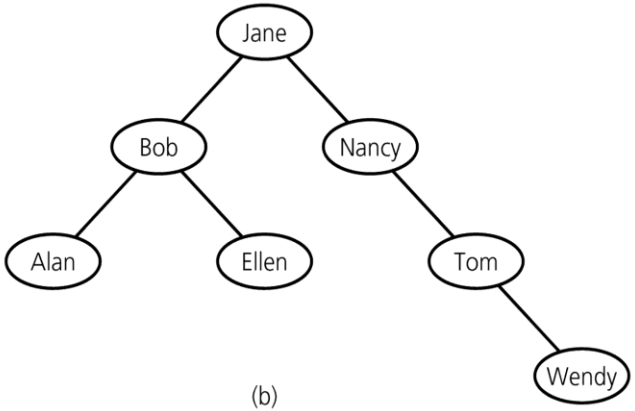


(a)



(c)





(b)

Binary Search Tree Declaration

There is no difference between regular binary tree declaration and binary search tree declaration. The difference is only in data but not in structure. But for our convenience we change the structure name as:

```
struct BinarySearchTreeNode{
    int data;
    struct BinarySearchTreeNode *left;
    struct BinarySearchTreeNode *right;
};
```

Operations on Binary Search Trees

Main operations: Following are the main operations that are supported by binary search trees:

- Find Minimum / Maximum element in binary search trees
- Inserting an element in binary search trees
- Deleting an element from binary search trees

Auxiliary operations: Checking whether the given tree is a binary search tree or not

- Finding *kth*-smallest element in tree
- Sorting the elements of binary search tree and many more

Important Notes on Binary Search Trees

- Given a BST, how to get a sorted list?
  - Since root data is always between left subtree data and right subtree data, performing inorder traversal on binary search tree produces a sorted list.
- Take advantage or recursion:
  - While solving problems on binary search trees, first we process left subtree, then root data, and finally we process right subtree. This means, depending on the problem, only the intermediate step (processing root data) changes and we do not touch the first and third steps.

Important Notes on Binary Search Trees (Continued)

- How to search for an item?
  - If we are searching for an element and if the left subtree root data is less than the element we want to search, then skip it. The same is the case with the right subtree. Because of this, binary search trees take less time for searching an element than regular binary trees. In other words, the binary search trees consider either left or right subtrees for searching an element but not both.
- What is the time complexity of basic operations (insertion, deletion, search)
  - The basic operations that can be performed on binary search tree (BST) are insertion of element, deletion of element, and searching for an element. While performing these operations on BST the height of the tree gets changed each time. Hence there exists variations in time complexities of best case, average case, and worst case.

Operations on BSTs

- Most of the operations on binary trees are O(log*N*).
  - This is the main motivation for using binary trees rather than using ordinary lists to store items.
  - The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with node n, such operations runs in O(logn) worst-case time. If the tree is a linear chain of n nodes (skew-tree), however, the same operations takes O(n) worst-case time.

- Most of the operations can be implemented using recursion.
  - we generally do not need to worry about running out of stack space, since the average depth of binary search trees is O(log*N*).

The BinaryNode class

```
template <class T>
class BinaryNode
{
   T element;   // this is the item stored in the node
   BinaryNode *left;
   BinaryNode *right;

   BinaryNode( const T & theElement, BinaryNode *lt,
     BinaryNode *rt ) : element( theElement ), left( lt ),
     right( rt ) { }
};
```

find

```
/**
 * Method to find an item in a subtree.
 * x is item to search for.
 * t is the node that roots the tree.
 * Return node containing the matched item.
 */
template <class T>
BinaryNode<T> *
find( const T & x, BinaryNode<T> *t ) const
{
  if( t == NULL )
      return NULL;
  else if( x < t->element )
      return find( x, t->left );
  else if( t->element < x )
      return find( x, t->right );
  else
      return t;    // Match
}
```

Finding an Element in Binary Search Trees (Continued)

```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL )
        return NULL;
    if( data < root→data )
        return Find(root→left, data);
    else if( data > root→data )
        return( Find( root→right, data );
    return root;
}
```

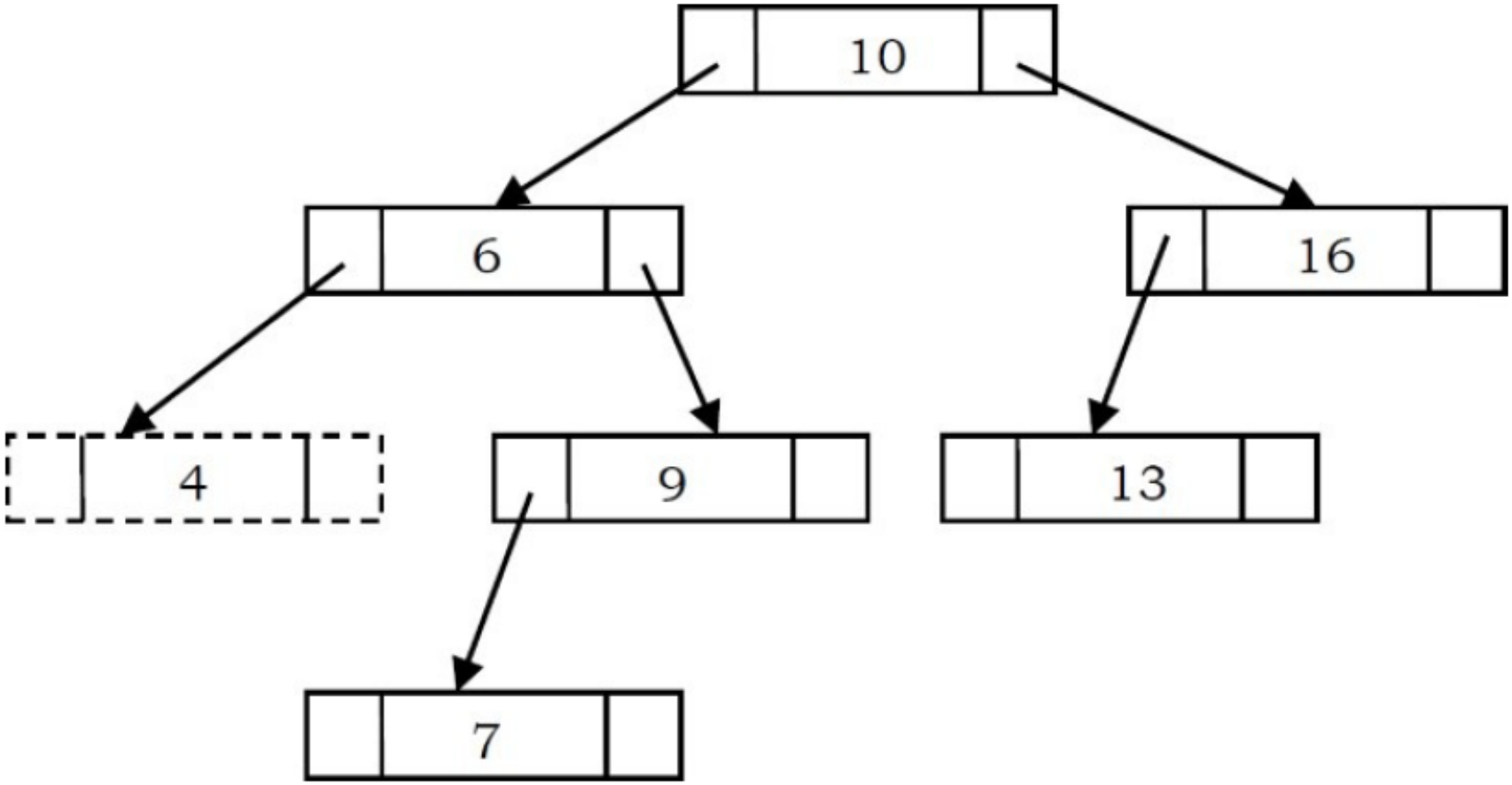
Time Complexity: O(n) in worst case (when BST is a skew tree).
Space Complexity: O(n) for recursive stack.

Non-Recursive Version of Finding an Element in Binary Search Trees

```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL )
        return NULL;
    while (root) {
        if(data == root→data)
            return root;
        else if(data > root→data)
            root = root→right;
        else root = root→left;
    }
    return NULL;
}
```

Time Complexity: O(n), Space Complexity: O(1)

Finding An Element in Binary Search Trees - Example



findMin (recursive implementation)

```
/**
 * method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
template <class T>
BinaryNode<T> *
findMin( BinaryNode<T> *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

findMax (nonrecursive implementation)

```
/**
 *method to find the largest item in a subtree t.
 *Return node containing the largest item.
 */
template <class T>
BinaryNode<T> *
findMax( BinaryNode<T> *t ) const
{
  if( t != NULL )
    while( t->right != NULL )
       t = t->right;
  return t;
}
```

Insert operation

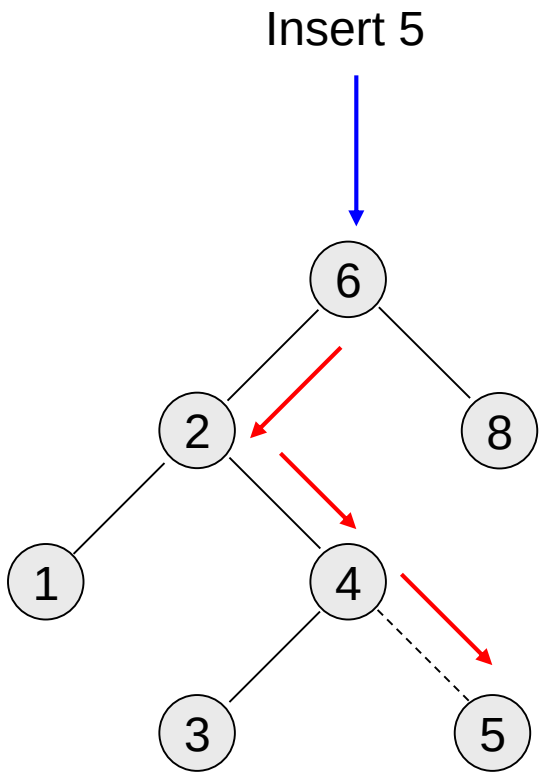
Algorithm for inserting X into tree T:

- Proceed down the tree as you would with a find operation.
- if X is found
  do nothing, (or “update” something)
  else
  insert X at the last spot on the path traversed.

Example

- As an example let us consider the following tree. The dotted node indicates the element (5) to be inserted. To insert 5, traverse the tree using *find* function. At node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct location for insertion.

- What about duplicates?



Inserting an Element from Binary Search Tree (Continued)

```
struct BinarySearchTreeNode *Insert(struct BinarySearchTreeNode *root, int data) {
    if( root == NULL ) {
        root = (struct BinarySearchTreeNode *) malloc(sizeof(struct  BinarySearchTreeNode));
        if( root == NULL ) {
            printf("Memory Error");
            return;
        }
        else {
            root→data = data;
            root→left = root→right = NULL;
        }
    }
    else {
        if( data < root→data )
            root→left = Insert(root→left,  data);
        else if( data > root→data )
            root→right = Insert(root→right,  data);
    }

    return root;
}
```

Insertion into a BST

```
/* method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 * Set the new root.
 */
template <class T>
void insert( const T & x,
             BinaryNode<T> * & t ) const
{
   if( t == NULL )
      t = new BinaryNode<T>( x, NULL, NULL );
   else if( x < t->element )
      insert( x, t->left );
   else if( t->element < x )
      insert( x, t->right );
   else
      ;  // Duplicate; do nothing
}
```

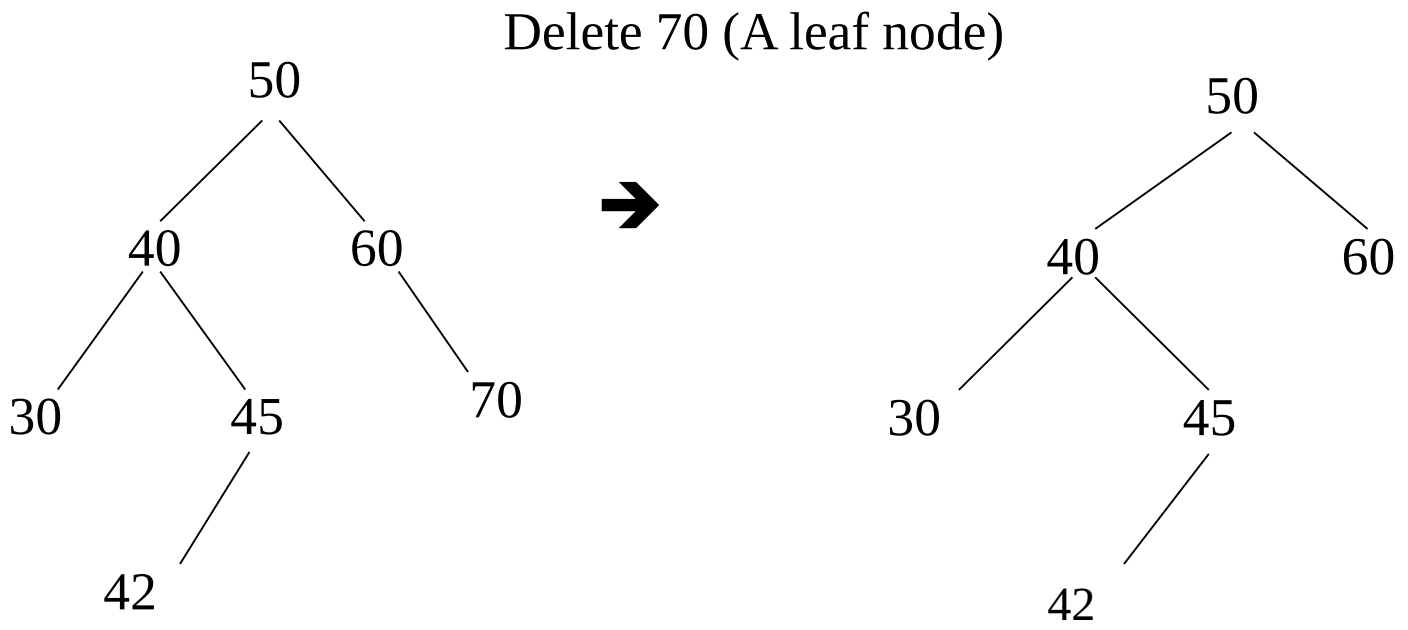
Deletion operation

There are three cases to consider:

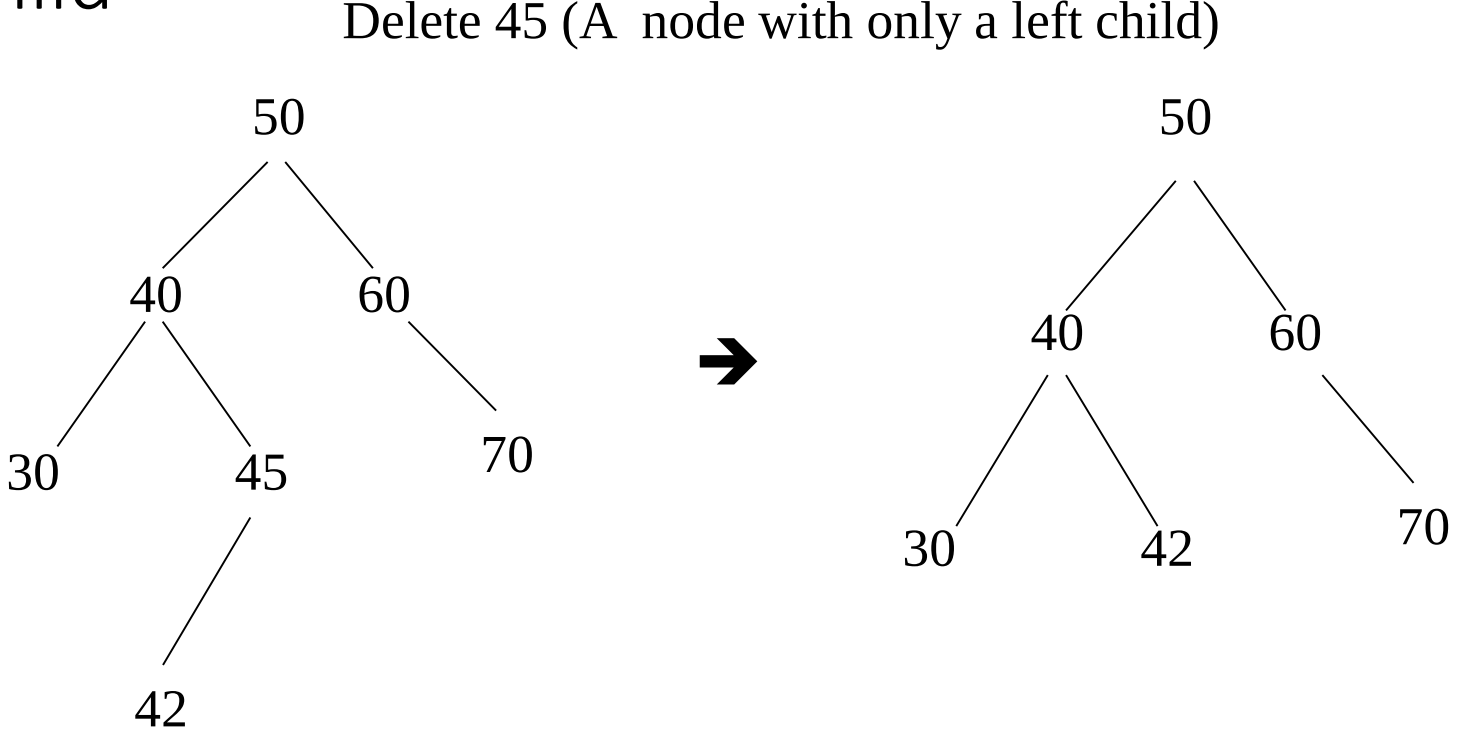
1. Deleting a leaf node
   - Replace the link to the deleted node by NULL.
2. Deleting a node with one child:
   - The node can be deleted after its parent adjusts a link to bypass the node.
3. Deleting a node with two children:
   - The deleted value must be replaced by an existing value that is either one of the following:
     - The largest value in the deleted node's left subtree
     - The smallest value in the deleted node's right subtree.

Deletion – Case1: A Leaf Node

To remove the leaf containing the item, we have to set the pointer in its parent to NULL.

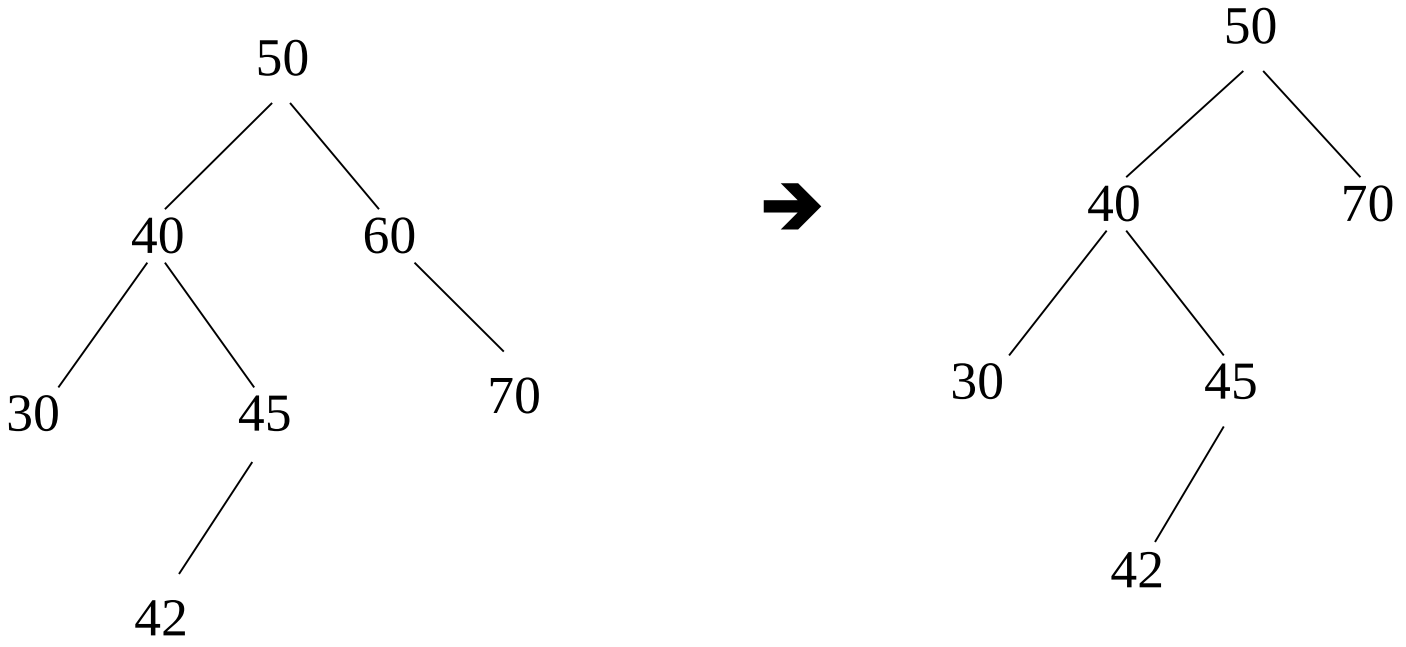


Deletion – Case2: A Node with only a left child



Deletion – Case2: A Node with only a right child

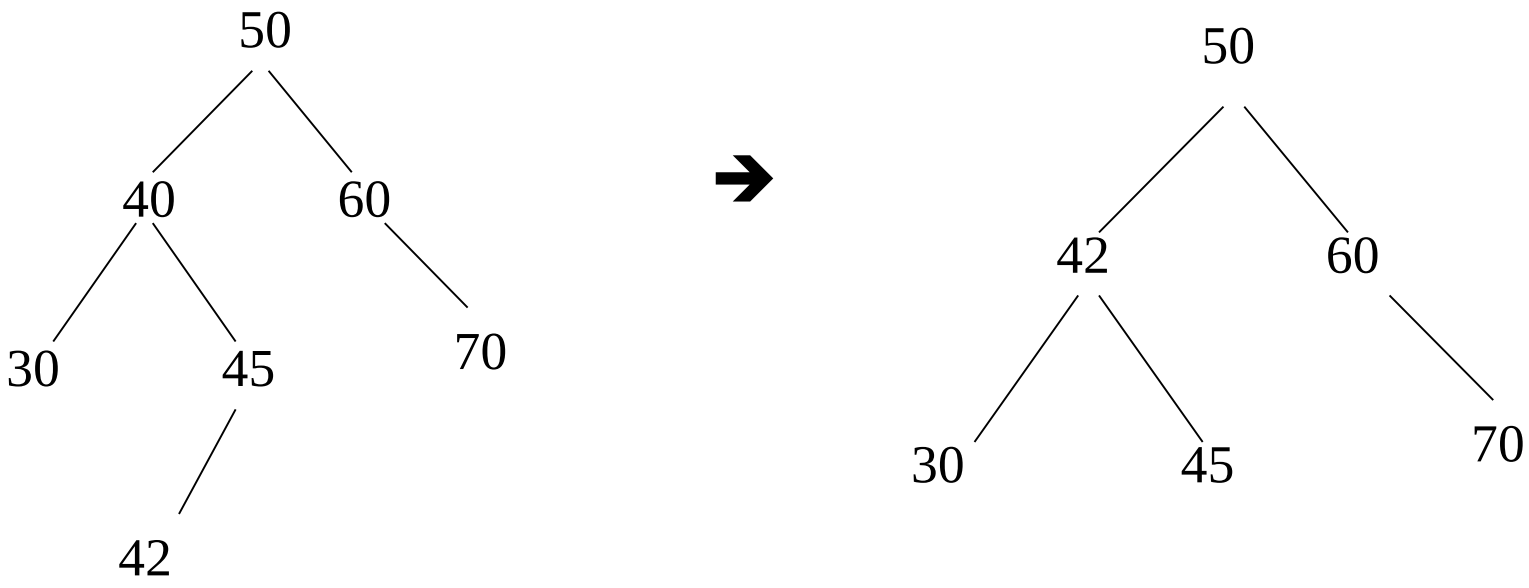
Delete 60 (A node with only a right child)



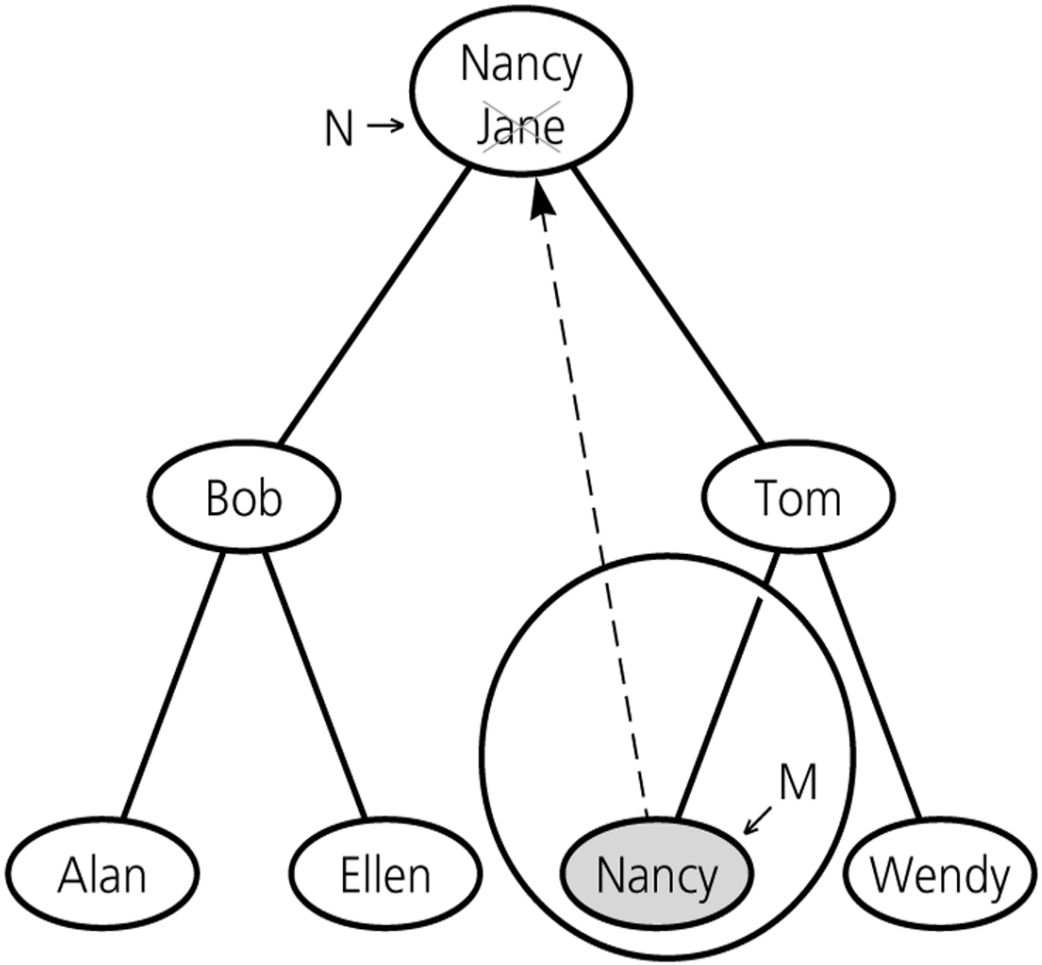
Deletion – Case3: A Node with two children

- Locate the inorder successor of the node.
- Copy the item in this node into the node which contains the item which will be deleted.
- Delete the node of the inorder successor.

Delete 40 (A node with two children)



Deletion – Case3: A Node with two children

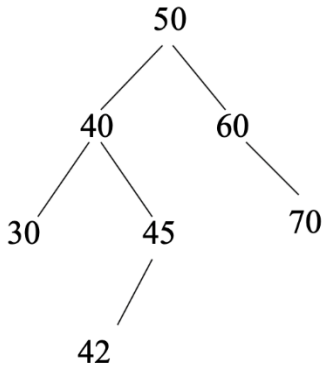


Deleting an Element from Binary Search Tree (Continued)

```
struct BinarySearchTreeNode *Delete(struct BinarySearchTreeNode *root, int data) {
    struct BinarySearchTreeNode *temp;
    if( root == NULL )
        printf("Element not there in tree");
    else if(data < root→data)
            root→left = Delete(root→left, data);
    else if(data > root→data )
        root→right = Delete(root→right, data);
    else {
        //Found element
        if( root→left && root→right ) {
            /* Replace with largest in left subtree */
            temp = FindMax( root→left );
            root→data = temp→data;
            root→left = Delete(root→left, root→data);
        }
        else {
                /* One child */
                temp = root;
                if( root→left == NULL )
                    root = root→right;
                if( root→right == NULL )
                    root = root→left;
                free( temp );
        }
    }
    return root;
}
```

Deletion routine for BST

```
template <class T>
void remove( const T & x,
             BinaryNode<T> * & t ) const
{
   if( t == NULL )
      return;   // Item not found; do nothing
   if( x < t->element )
      remove( x, t->left );
   else if( t->element < x )
      remove( x, t->right );
   else if( t->left != NULL && t->right != NULL {
       t->element = findMin( t->right )->element;
       remove( t->element, t->right );
   }
   else   {
      BinaryNode<T> *oldNode = t;
      t = ( t->left != NULL ) ? t->left : t->right;
      delete oldNode;
    }
}
```



Analysis of BST Operations

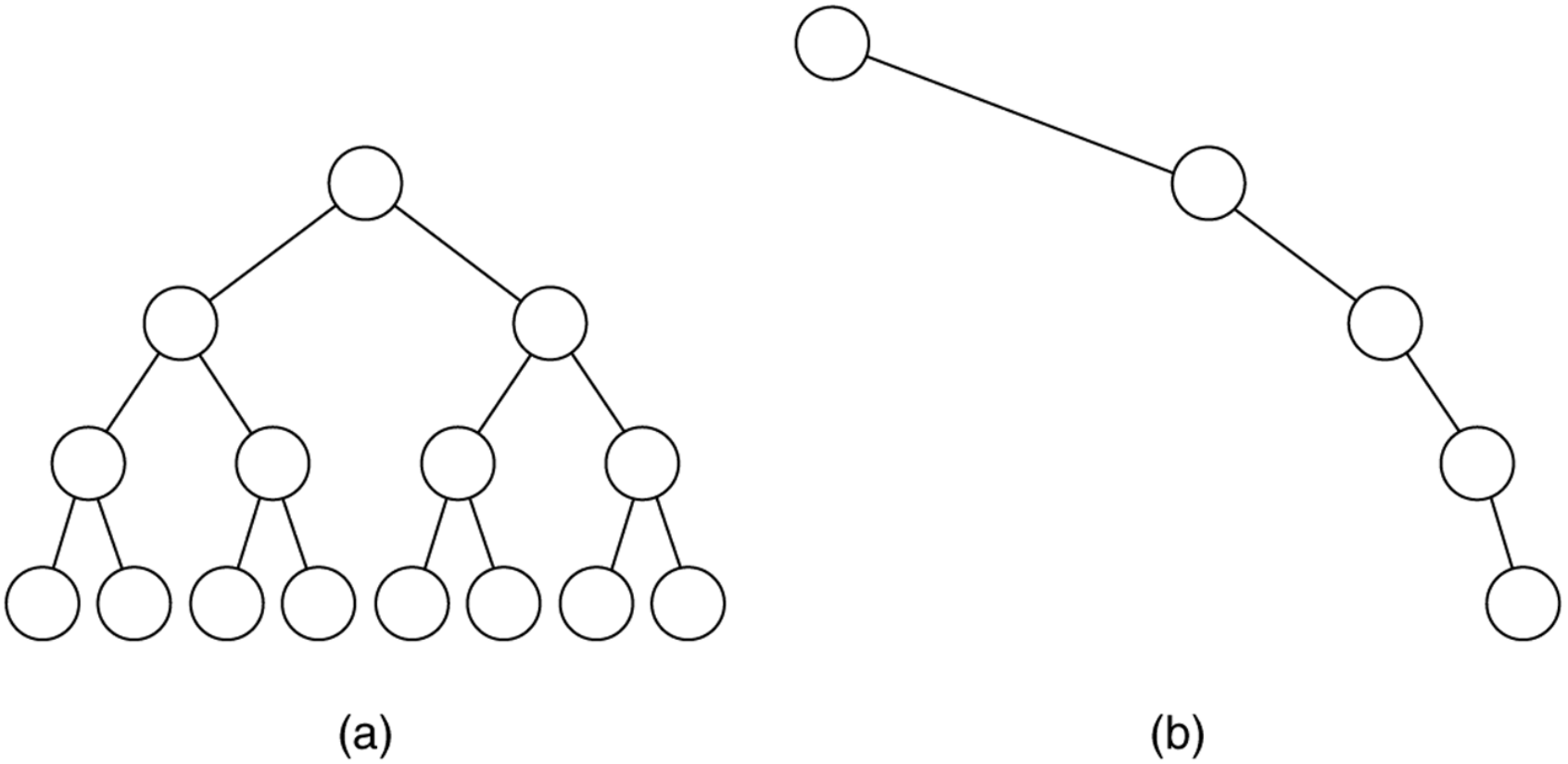
- The cost of an operation is proportional to the depth of the last accessed node.
- The cost is logarithmic for a well-balanced tree, but it could be as bad as linear for a degenerate tree.
- In the best case we have logarithmic access cost, and in the worst case we have linear access cost.

Minimum Height

- Insertion in search-key order produces a maximum-height binary *search* tree.
- Insertion in random order produces a near-minimum-height binary *search* tree.
- That is, the height of an n-node binary search tree is:
  - *Best Case* – $\lceil \log_2(n+1)-1 \rceil$ (see slide 25) ➔ $O(\log_2 n)$
  - *Worst Case* – n-1 ➔ O(n)
  - *Average Case* – close to $\lceil \log_2(n+1)-1 \rceil$ ➔ $O(\log_2 n)$
    - In fact, $1.39\log_2 n$

Figure 19.19

(a) The balanced tree has a height of log *N*; (b) the unbalanced tree has a height of *N* – 1.



Number of BSTs

Suppose we're inserting *n* items into an empty binary search tree to create a binary search tree with *n* nodes,

➔ How many different binary search trees with n nodes, and

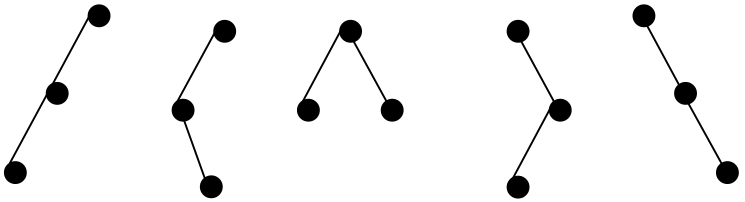
➔ What are their probabilities,

There are n! different orderings of n keys.
But how many different binary search trees with n nodes?

n=0 ➔ 1 BST (empty tree)
n=1 ➔ 1 BST (a binary tree with a single node)
n=2 ➔ 2 BSTs
n=3 ➔ 5 BSTs

Number of BSTs

n=3 ➔



Probabilities:	1/6	1/6	2/6	1/6	1/6
Insertion Order:	3,2,1	3,1,2	2,1,3 2,3,1	1,3,2	1,2,3

In general, given n elements, the number of binary search trees that can be made from those elements is given by the n^{th} Catalan Number (denoted C_n).

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

Order of Operations on BSTs

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

Treesort

- We can use a binary search tree to sort an array.

```
treesort(inout anArray:ArrayType, in n:integer)
// Sorts n integers in an array anArray
// into ascending order
   Insert anArray's elements into a
     binary search tree bTree

   Traverse bTree in inorder.
     As you visit bTree's nodes, copy their data items
     into successive locations of anArray
```

Treesort Analysis

- Inserting an item into a binary search tree:
  - Worst Case: O(n)
  - Average Case: $O(\log_2 n)$
- Inserting *n* items into a binary search tree:
  - Worst Case: $O(n^2)$ ➔ $(1+2+...+n) = O(n^2)$
  - Average Case: $O(n*\log_2 n)$
- Inorder traversal and copy items back into array ➔ O(n)
- Thus, treesort is

  ➔ **$O(n^2)$ in worst case**, and

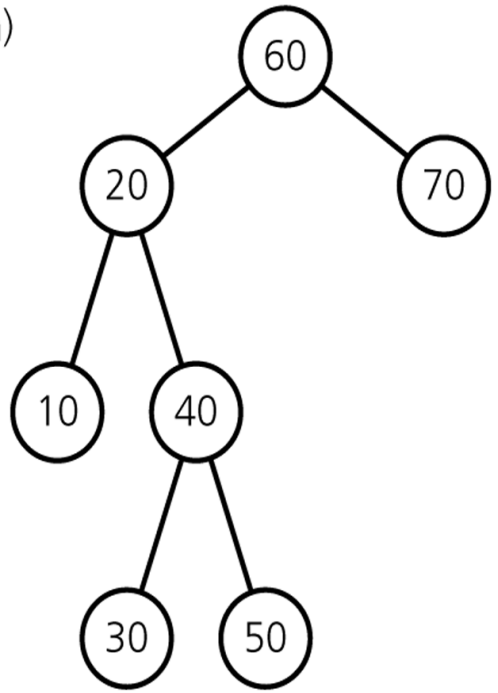
  ➔ **$O(n*\log_2 n)$ in average case**.
- Treesort makes exactly the same comparisons of keys as quicksort when the pivot for each sublist is chosen to be the first key.

Saving a BST into a file, and restoring it to its original shape

- Save:
  - Use a preorder traversal to save the nodes of the BST into a file.
- Restore:
  - Start with an empty BST.
  - Read the nodes from the file one by one, and insert them into the BST.

Saving a BST into a file, and restoring it to its original shape

(a)



(b)

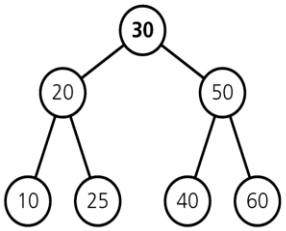
```
bst.searchTreeInsert(60);
bst.searchTreeInsert(20);
bst.searchTreeInsert(10);
bst.searchTreeInsert(40);
bst.searchTreeInsert(30);
bst.searchTreeInsert(50);
bst.searchTreeInsert(70);
```

Preorder: 60 20 10 40 30 50 70

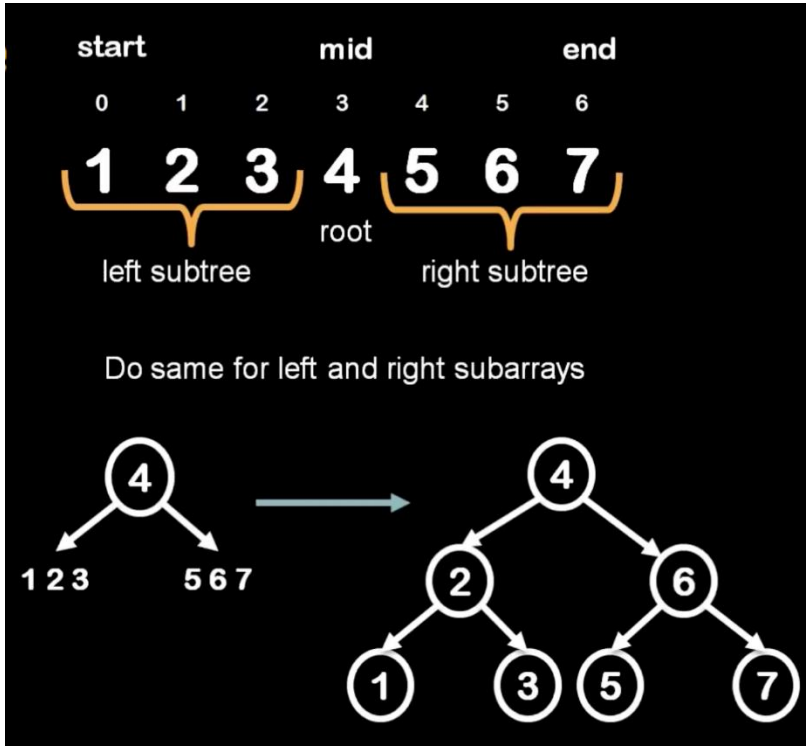
Saving a BST into a file, and restoring it to a minimum-height BST

- Save:
  - Use an inorder traversal to save the nodes of the BST into a file. The saved nodes will be in ascending order.
  - Save the number of nodes (n) in somewhere.

- Restore:
  - Read the number of nodes (n).
  - Start with an empty BST.
  - Put middle element to root, set its left and right BSTs recursively.



Building a minimum-height BST



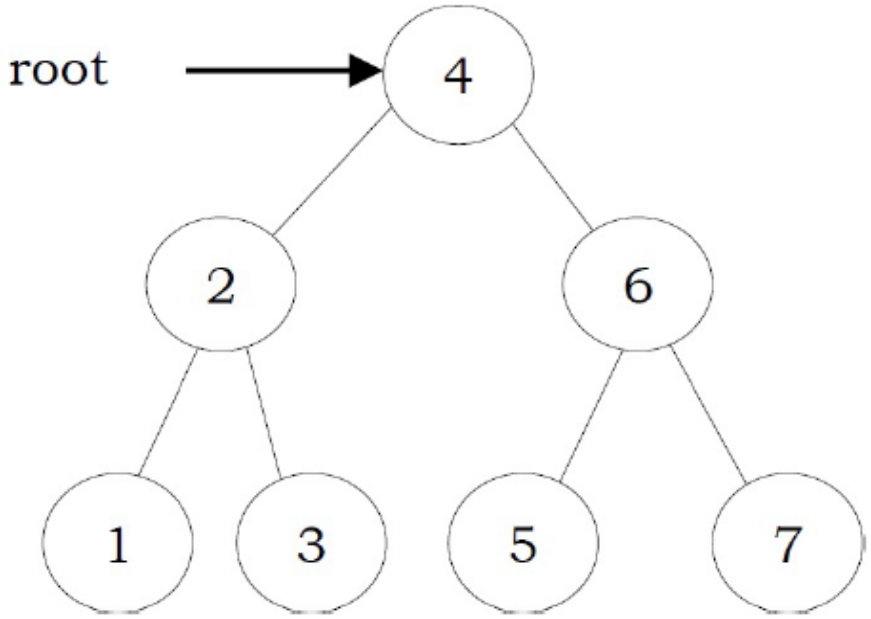

```
private static TreeNode createBST(int[] array,
                                  int start, int end) {

    if(start>end) return null;
    int mid = (start + end)/2;
    TreeNode root = new TreeNode(array[mid]);

    root.setLeft(createBST(array, start, mid-1));
    root.setRight(createBST(array, mid+1, end));

    return root;

}
```

Balanced Binary Search Trees

- We have seen different trees whose worst case complexity is O(*n*), where n is the number of nodes in the tree. This happens when the trees are skew trees. We will try to reduce this worst case complexity to O(*logn*) by imposing restrictions on the heights.
- In general, the height balanced trees are represented with *HB*(*k*), where *k* is the difference between left subtree height and right subtree height. Sometimes *k* is called balance factor.
- e.g. AVL Trees, Red-Black Trees, B-Trees

Full Balanced Binary Search Trees

- In *HB*(*k*), if *k* = 0 (if balance factor is zero), then we call such binary search trees as *full* balanced binary search trees. That means, in *HB*(0) binary search tree, the difference between left subtree height and right subtree height should be at most zero. This ensures that the tree is a full binary tree. For example



References

- Assist. Prof. Yusuf Sahillioğlu Leccture Notes, Metu.
- Assist. Prof. Fatih Soygazi Lecture Notes, ADU.
- Narasimha Karumanchi; “Data Structures and Algorithms Made Easy: Data Structure and Algorithmic Puzzles”, 5th Ed., CareerMonk Publications, 2016.