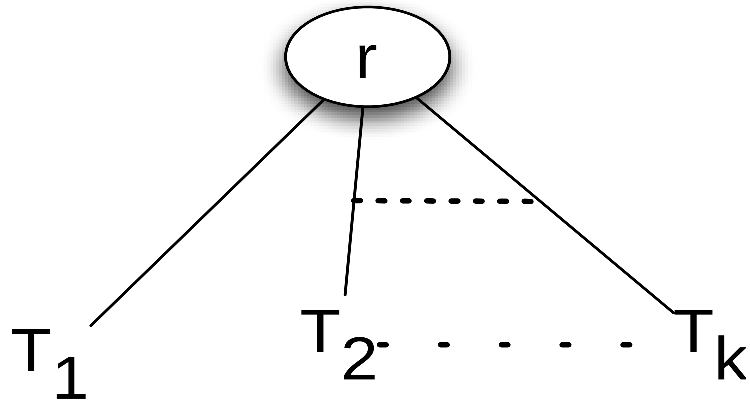
Trees

Initially prepared by Dr. İlyas Çiçekli; improved by various Bilkent CS202 instructors.

What is a Tree?

- T is a **tree** if either
  - T has no nodes, or
  - T is of the form:



where r is a node and T_1, T_2, ..., T_k are trees.

Tree Terminology

Parent – The parent of node n is the node directly above in the tree.

Child – The child of node n is the node directly below in the tree.

- If node m is the parent of node n, node n is the child of node m.

Root – The only node in the tree with no parent.

Leaf – A node with no children.

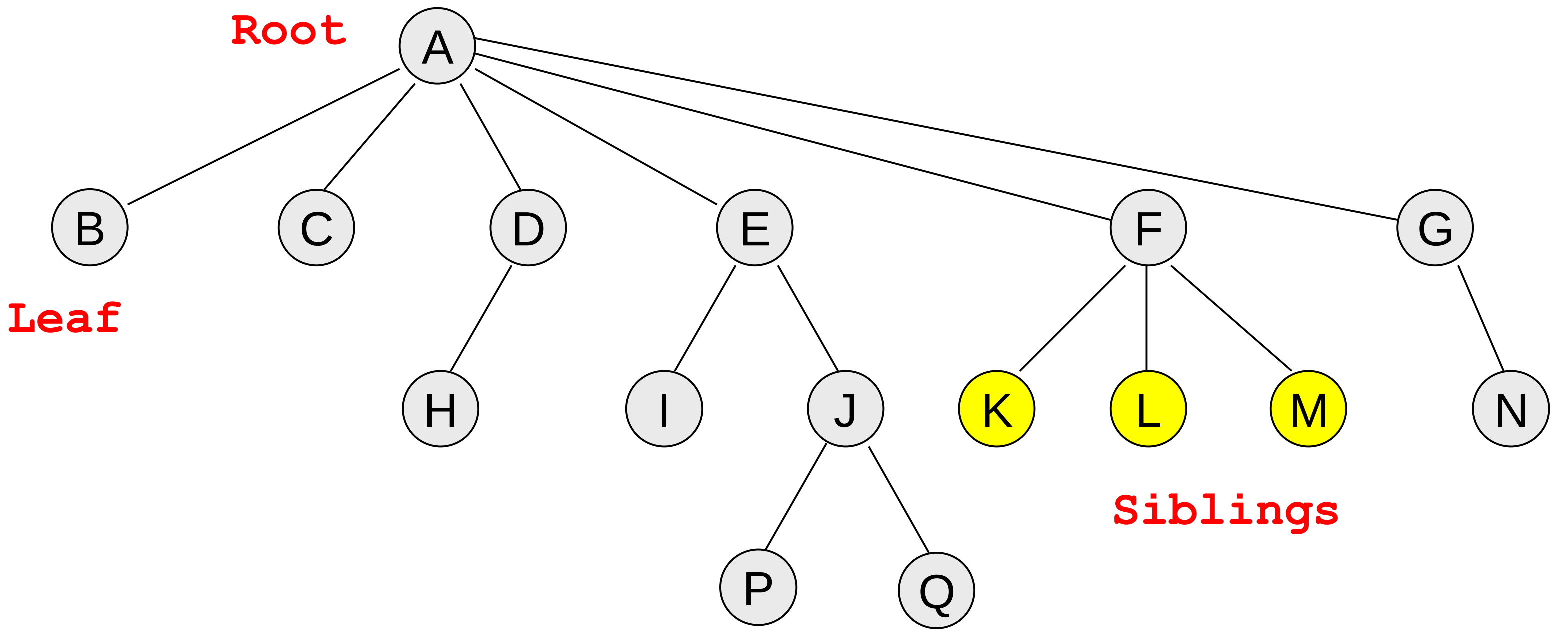
Siblings – Nodes with a common parent.

Ancestor – An ancestor of node n is a node on the path from the root to n.

Descendant – A descendant of node n is a node on the path from n to a leaf.

Subtree – A subtree of node n is a tree that consists of a child (if any) of n and the child's descendants (a tree which is rooted by a child of node n)

A Tree – Example



–Node *A* has 6 ***children***: B, C, D, E, F, G.

–B, C, H, I, P, Q, K, L, M, N are ***leaves*** in the tree above.

–K, L, M are ***siblings*** since F is parent of all of them.

What is a Tree?

- The root of each sub-tree is said to be ***child*** of r, and
  r is the ***parent*** of each sub-tree's root.

- If a tree is a collection of N nodes, then it has N-1 edges. **Why?**

- A ***path*** from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \dots, n_k$ such that n_i is parent of n_{i+1} $(1 \leq i < k)$
  - There is a path from every node to itself.
  - There is exactly one path from the root to each node. **Why?**

non-circular

Level of a node

Level – The level of node n is the number of nodes on the path from root to node n.

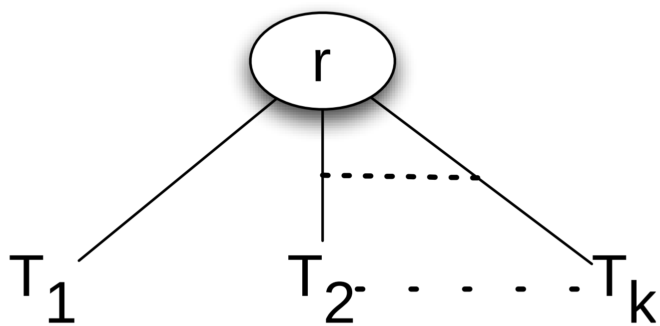
Definition: *The level of node n in a tree T*

- If n is the root of T, the level of n is 1.
- If n is not the root of T, its level is 1 greater than the level of its parent.

Height of A Tree

Height – number of nodes on **longest path** from the root to any leaf.

- The height of a tree T in terms of the levels of its nodes is defined as:
  - If T is empty, its height is 0
  - If T is not empty, its height is equal to the maximum level of its nodes.
- Or, the height of a tree T can be defined as recursively as:
  - If T is empty, its height is 0.
  - If T is non-empty tree, then since T is of the form:

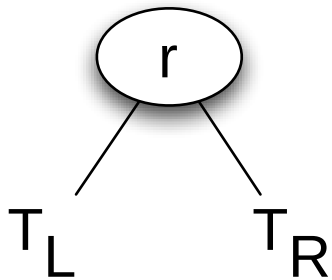


$$\textbf{\textit{height(T)}} = \mathbf{1} + \textbf{\textit{max}}\{height(T_1), height(T_2), \ldots, height(T_k)\}$$

Binary Tree

- *A binary tree T is a set of nodes with the following properties:*
  - *The set can be empty.*
  - *Otherwise, the set is partitioned into three disjoint subsets:*
    - *a tree consists of a distinguished node r, called* ***root****, and*
    - *two possibly empty sets are binary tree, called* ***left*** *and* ***right subtrees*** *of r.*

- T is a **binary tree** if either
  - T has no nodes, or
  - T is of the form:



where r is a node and T_L and T_R are binary trees.

Binary Tree Terminology

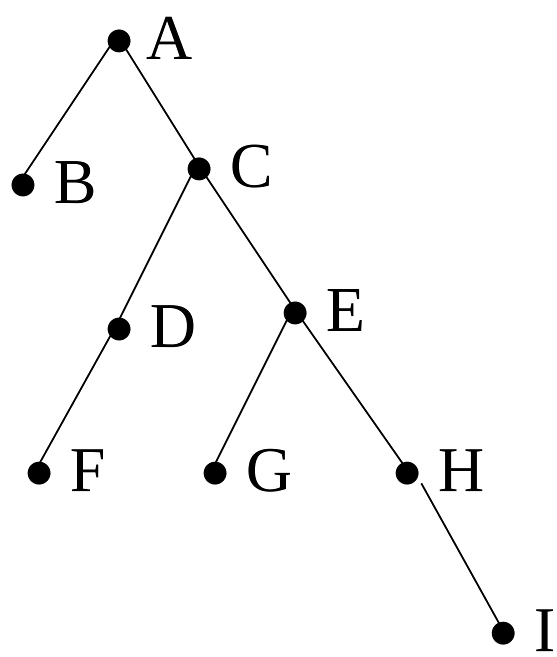
Left Child – The left child of node n is a node directly below and to the left of node n in a binary tree.

Right Child – The right child of node n is a node directly below and to the right of node n in a binary tree.

Left Subtree – In a binary tree, the left subtree of node n is the left child (if any) of node n plus its descendants.

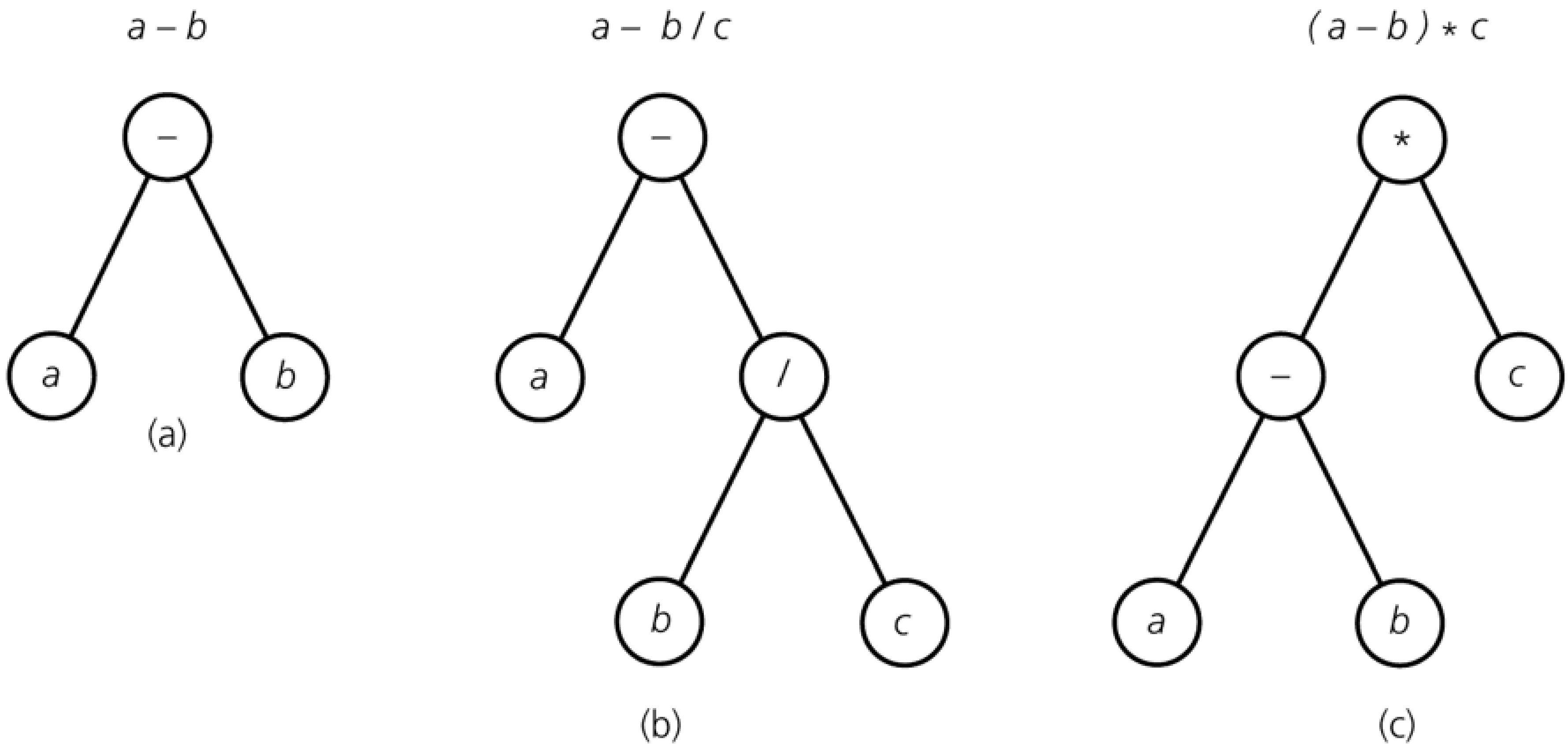
Right Subtree – In a binary tree, the right subtree of node n is the right child (if any) of node n plus its descendants.

Binary Tree -- Example



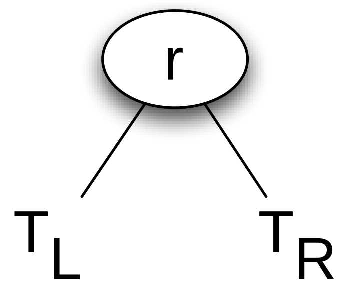
- A is the root.
- B is left child of A,
  C is right child of A.
- D doesn't have a right child.
- H doesn't have a left child.
- B, F, G and I are leaves.

Binary Tree – Representing Algebraic Expressions



Height of Binary Tree

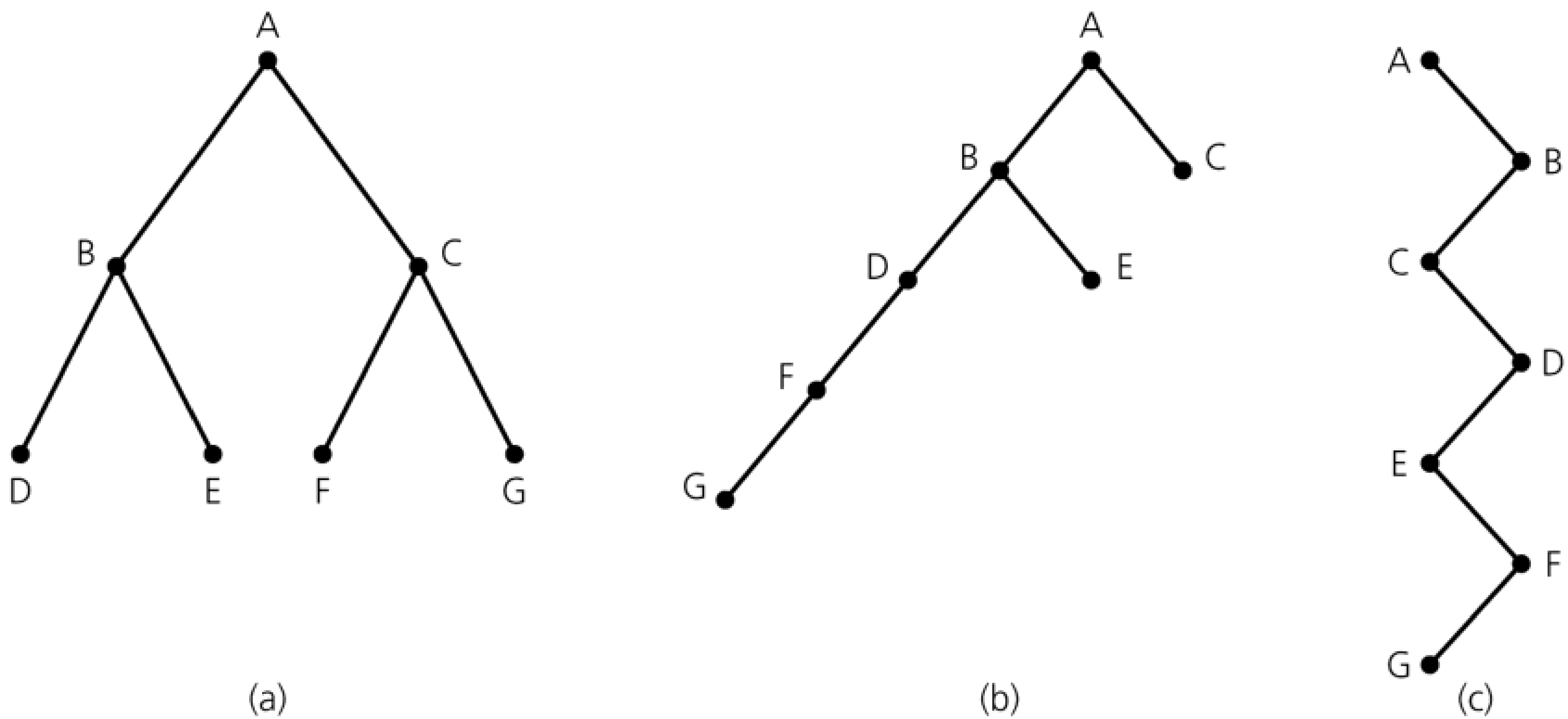
- The height of a binary tree T can be defined as recursively as:
  - If T is empty, its height is 0.
  - If T is non-empty tree, then since T is of the form ...



... height of T is 1 greater than height of its root's taller subtree; ie.

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

Height of Binary Tree (cont.)

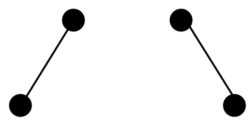


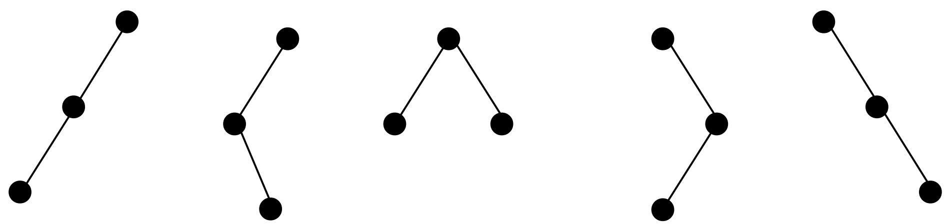
Binary trees with the same nodes but different heights

Number of Binary trees with Same # of Nodes

n=0 ➔ empty tree

n=1 ➔  (1 tree)

n=2 ➔  (2 trees)

n=3 ➔  (5 trees)

n is even ➔ $$NumBT(N) = 2\sum_{i=0}^{(n-1)/2} (NumBT(i)NumBT(n-i-1))$$

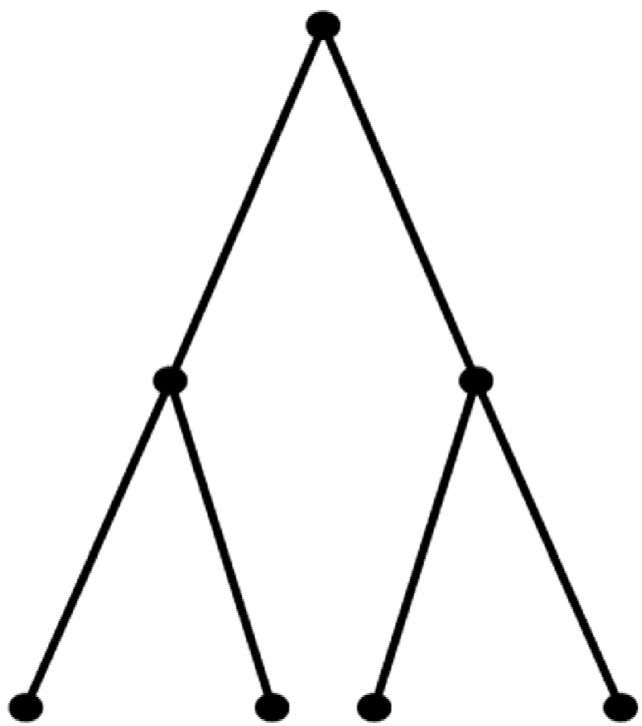
n is odd ➔ $$NumBT(N) = 2\sum_{i=0}^{((n-1)/2)-1} (NumBT(i)NumBT(n-i-1))$$

$$+\ NumBT((n-1)/2)NumBT((n-1)/2)$$

Full Binary Tree

- In a ***full binary tree*** of height h, all nodes that are at a level less than h have two children each.
- Each node in a full binary tree has left and right subtrees of the same height.
- Among binary trees of height h, a full binary tree has as many leaves as possible, and **leaves all are at level h**.
- A full binary tree has **no missing nodes**.
- Recursive definition of full binary tree:
  - If T is empty, T is a full binary tree of height 0.
  - If T is not empty and has height h>0, T is a full binary tree if its root's subtrees are both full binary trees of height h-1.

Full Binary Tree – Example



A full binary tree of height 3

Complete Binary Tree

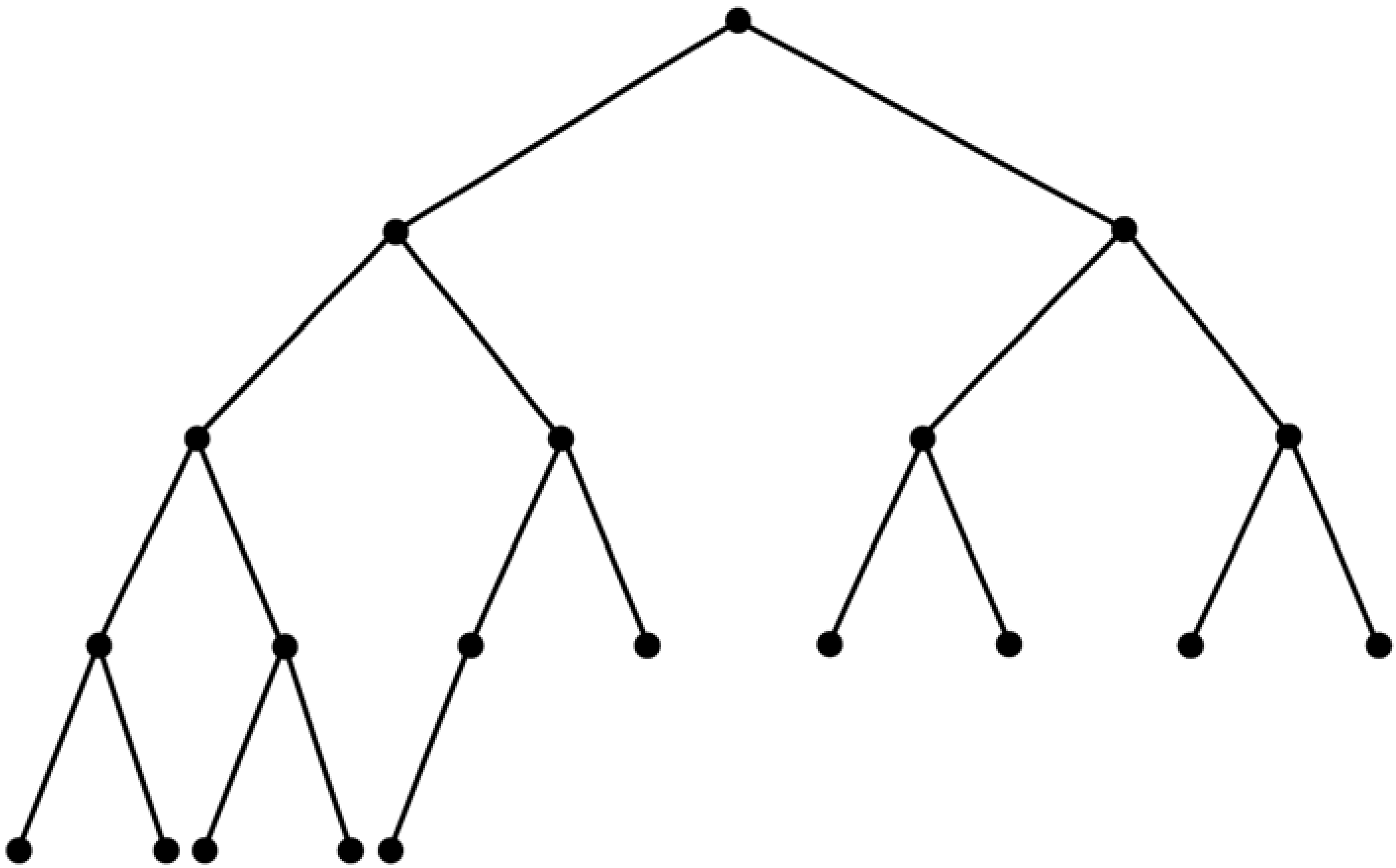
- A ***complete binary tree*** of height h is a binary tree that is **full down to level h-1**, with level h filled in from left to right.

- A binary tree T of height h is complete if
  1. All nodes at level h-2 and above have two children each, and
  2. When a node at level h-1 has children, all nodes to its left at the same level have two children each, and
  3. When a node at level h-1 has one child, it is a left child.

  last level filling the tree left to right

- A full binary tree is a complete binary tree.

Complete Binary Tree – Example



Balanced Binary Tree

- A binary tree is ***balanced*** (or ***height balanced***), if the height of any node's right subtree and left subtree **differ no more than 1**.

- A complete binary tree is a balanced tree. Why?

- Later, we look at other height balanced trees.
  – AVL trees
  – Red-Black trees, ...

Maximum and Minimum Heights of a Binary Tree

- **Efficiency** of most binary tree operations **depends on tree height**.

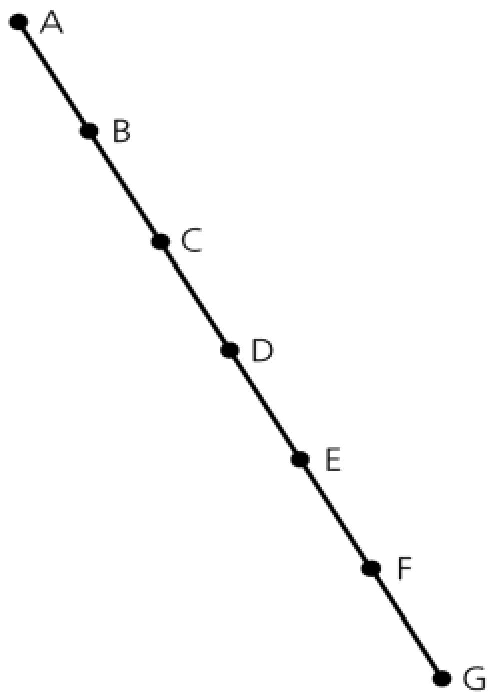
- **E.g. maximum number of key comparisons** for retrieval, deletion, and insertion operations for BSTs is the height of the tree.

- The maximum of height of a binary tree with n nodes is n. How?

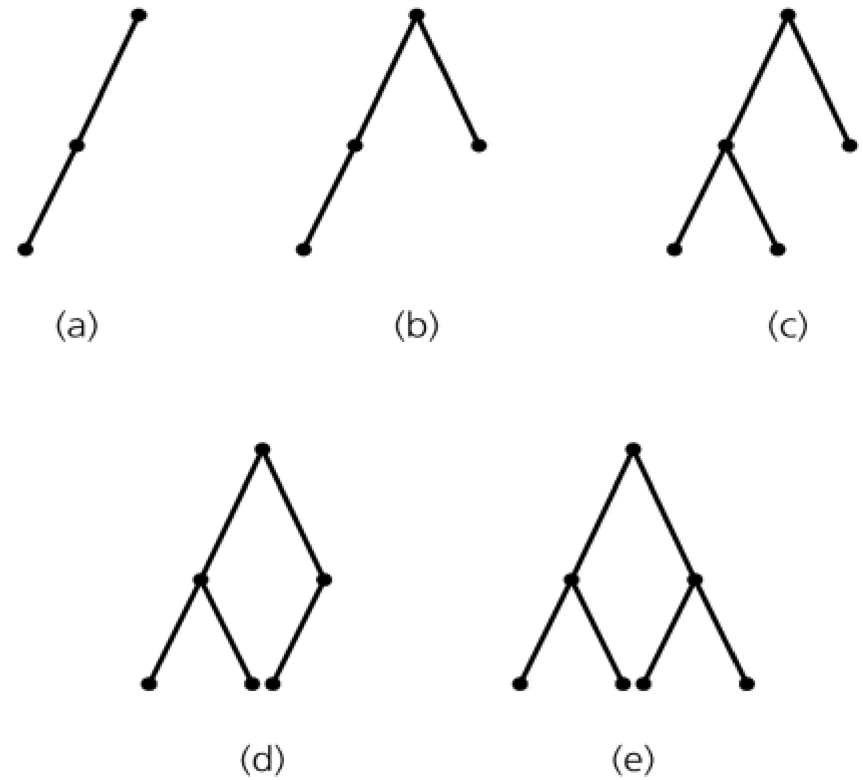
  linear (chain)

- Each level of a minimum height tree, except the last level, must contain as many nodes as possible.
  - Should the tree be a Complete Binary Tree?

Maximum and Minimum Heights of a Binary Tree

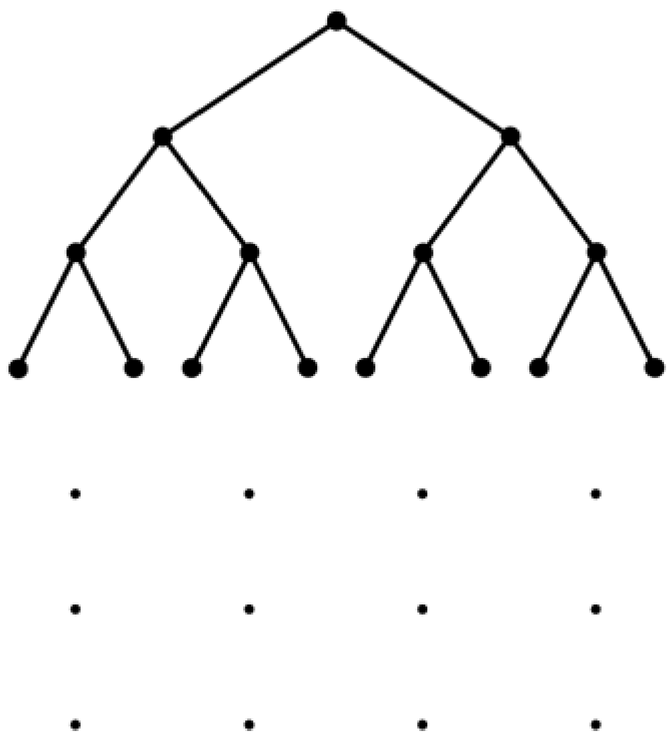


A maximum-height binary tree with seven nodes



Some binary trees of height 3

Counting the nodes in a full binary tree of height *h*



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}	$2^h - 1$

Some Height Theorems

Theorem: A full binary tree of height $h \geq 0$ has 2^h-1 nodes.

- The maximum number of nodes that a binary tree of height h can have is 2^h-1.

- We cannot insert a new node into a full binary tree without increasing its height.

Some Height Theorems

Theorem 10-4: The minimum height of a binary tree with n nodes is $\lceil \log_2(n+1) \rceil$.

Proof: Let h be the smallest integer such that $n \leq 2^h-1$. We can establish following facts:

Fact 1 – A binary tree whose height is $\leq$ h-1 has $<$ n nodes.

- Otherwise h cannot be smallest integer in our assumption.

Fact 2 – There exists a complete binary tree of height h that has exactly n nodes.

- A full binary tree of height h-1 has $2^{h-1}-1$ nodes.
- Since a binary tree of height h cannot have more than 2^h-1 nodes.
- At level h, we will reach n nodes.

Fact 3 – The minimum height of a binary tree with n nodes is the smallest integer h such that $n \leq 2^h-1$.

So, ➔ $2^{h-1}-1 < n \leq 2^h-1$

➔ $2^{h-1} < n+1 \leq 2^h$

➔ $h-1 < \log_2(n+1) \leq h$

Thus, ➔ $h = \lceil \log_2(n+1) \rceil$ is the minimum height of a binary tree with n nodes.

- UML Diagram for **BinaryTree ADT**

- What is an **ADT**?

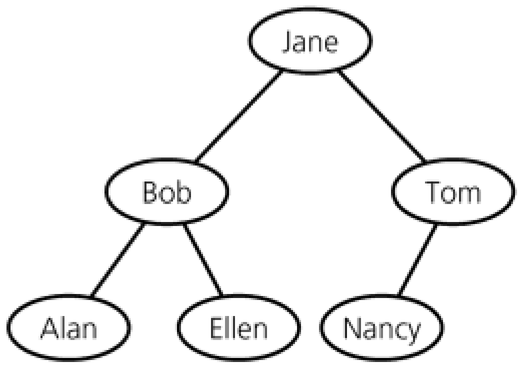
Binary tree
root *left subtree* *right subtree*
createTree() *destroyBinaryTree()* *isEmpty()* *getRootData()* *setRootData()* *attachRight()* *attachLeftSubtree()* *attachRightSubtree()* *detachLeftSubtree()* *detachRightSubtree()* *getLeftSubtree()* *getRightSubtree()* *preorderTraverse()* *inorderTraverse()* *postorderTraverse()*

An Array-Based Implementation of Binary Trees

```
const int MAX_NODES = 100;                // maximum number of nodes
typedef string TreeItemType;

class TreeNode {                          // node in the tree
private:
        TreeNode();
        TreeNode(const TreeItemType& nodeItem, int left, int right);

        TreeItemType item;                // data portion
        int leftChild;                    // index to left child
        int rightChild;                   // index to right child

        // friend class - can access private parts
        friend class BinaryTree;
};

// An array of tree nodes
TreeNode[MAX_NODES] tree;
int  root;
int  free;
```

An Array-Based Implementation (cont.)

(a)



(b) `tree`

	`item`	`leftChild`	`rightChild`
0	Jane	1	2
1	Bob	3	4
2	Tom	5	–1
3	Alan	–1	–1
4	Ellen	–1	–1
5	Nancy	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
⋮	⋮	⋮	⋮

`root`: 0

`free`: 6

Free list

- A ***free list*** keeps track of available nodes.
- To insert a new node into the tree, we first obtain an available node from the free list.
- When we delete a node from the tree, we have to place into the free list so that we can use it later.

An Array-Based Representation of a Complete Binary Tree

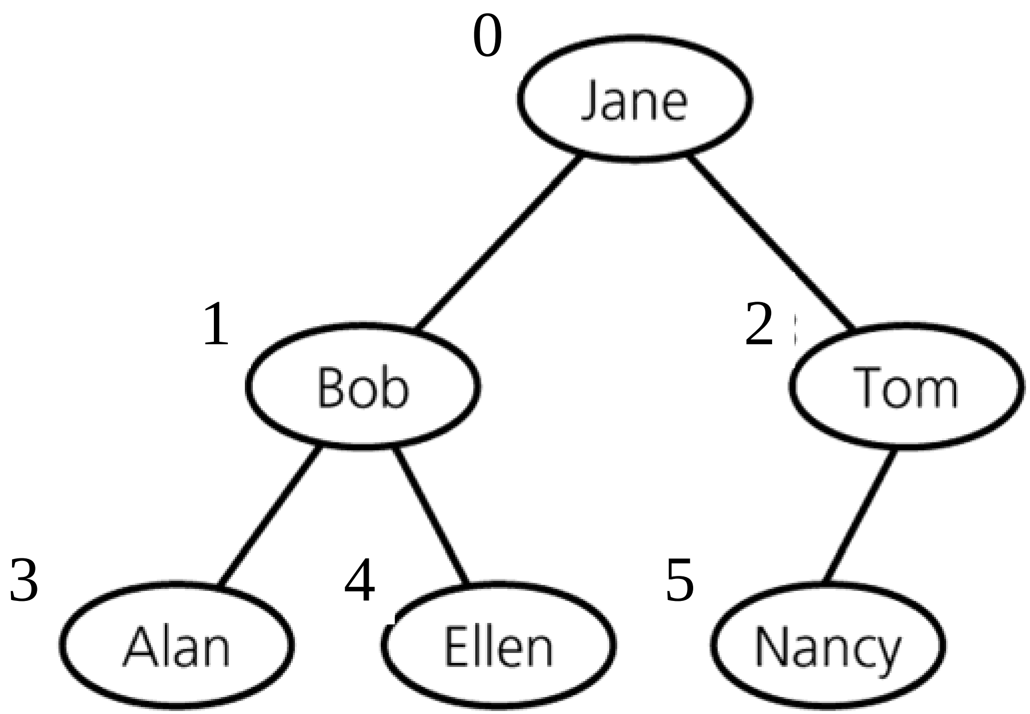
- If we know that our binary tree is a **complete binary tree**, we can use a simpler array-based representation for complete binary trees
  - **without** using **leftChild, rightChild** links

- We can number the nodes level by level, and left to right (starting from 0, the root will be 0). If a node is numbered as i, in the ith location of the array, `tree[i]`, contains this node without links.

- Using these numbers we can find leftChild, rightChild, and parent of a node i.

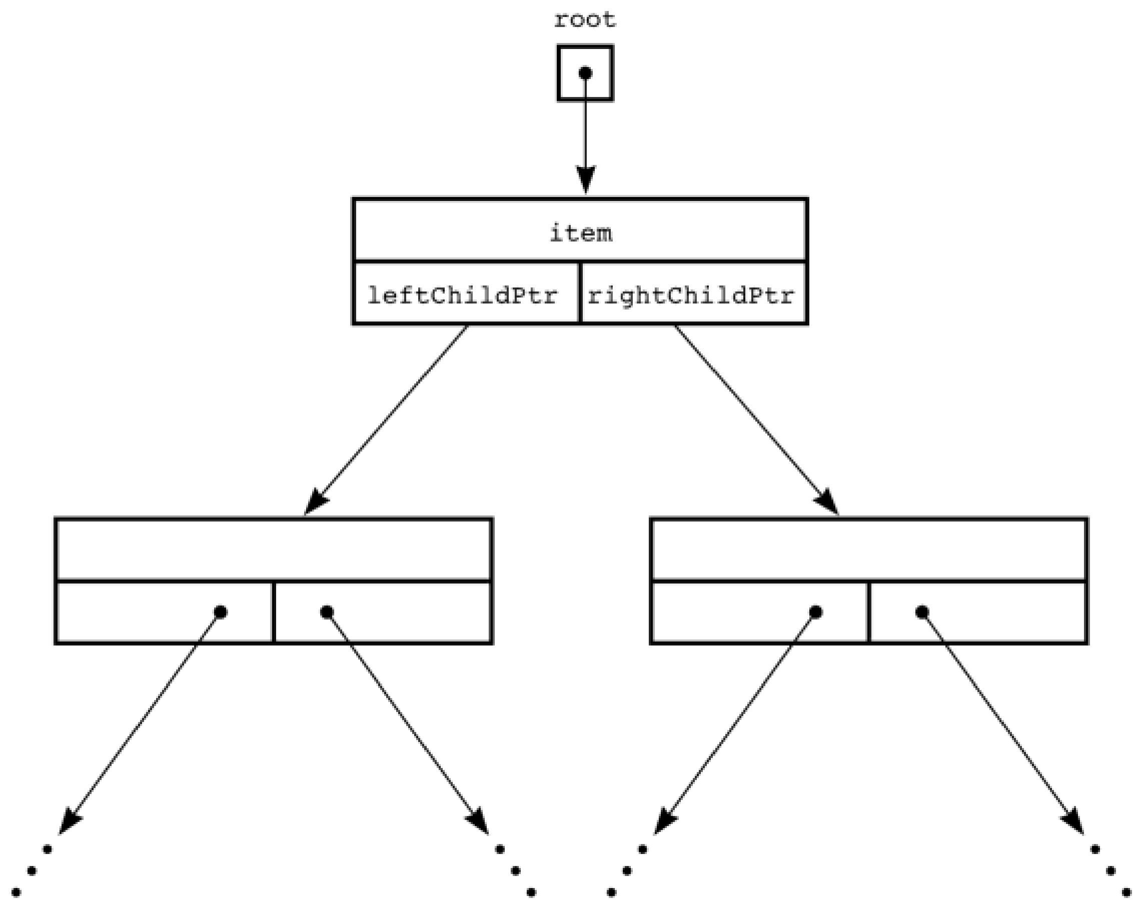
The left child (if it exists) of node i is	`tree[2*i+1]`
The right child (if it exists) of node i is	`tree[2*i+2]`
The parent (if it exists) of node i is	`tree[(i-1)/2]` integer division

An Array-Based Representation of a Complete Binary Tree (cont.)



0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	
7	

Pointer-Based Implementation of Binary Trees



A Pointer-Based Implementation of a Binary Tree Node

```
typedef string TreeItemType;

class TreeNode {          // node in the tree
private:
   TreeNode() {}
   TreeNode(const TreeItemType& nodeItem,
      TreeNode *left = NULL,
      TreeNode *right = NULL)
      :item(nodeItem),leftChildPtr(left),rightChildPtr(right) {}

   TreeItemType item;       // data portion
   TreeNode *leftChildPtr;  // pointer to left child
   TreeNode *rightChildPtr; // pointer to right child

   friend class BinaryTree;
};
```

Binary Tree – TreeException.h

```
class TreeException : public exception{

private:
   string msg;

public:
       virtual const char* what() const throw()
       {
            return msg.c_str();
       }
  TreeException(const string & message =""):
       exception(), msg(message) {};
       ~TreeException() throw() {};

}; // end TreeException
```

Binary tree
root *left subtree* *right subtree*
createTree() *destroyBinaryTree()* *isEmpty()* *getRootData()* *setRootData()* *attachRight()* *attachLeftSubtree()* *attachRightSubtree()* *detachLeftSubtree()* *detachRightSubtree()* *getLeftSubtree()* *getRightSubtree()* *preorderTraverse()* *inorderTraverse()* *postorderTraverse()*

The BinaryTree Class

- **Properties**
  - TreeNode * root

- **Constructors**
  - BinaryTree();
  - BinaryTree(const TreeItemType& rootItem);
  - BinaryTree(const TreeItemType& rootItem,
    BinaryTree& leftTree, BinaryTree& rightTree);
  - BinaryTree(const BinaryTree& tree);

    void copyTree(TreeNode *treePtr, TreeNode* & newTreePtr) const;

- **Destructor**
  - ~BinaryTree();

    void destroyTree(TreeNode * &treePtr);

BinaryTree: Public Methods

- bool isEmpty()
- TreeItemType rootData() const throw(TreeException)
- void setRootData(const TreeItemType& newItem)
- void attachLeft(const TreeItemType& newItem)
- void attachRight(const TreeItemType& newItem)
- void attachLeftSubtree(BinaryTree& leftTree)
- void attachRightSubtree(BinaryTree& rightTree)
- void detachLeftSubtree(BinaryTree& leftTree)
- void detachRightSubtree(BinaryTree& rightTree)
- BinaryTree leftSubtree()
- BinaryTree rightSubtree()
- void preorderTraverse(FunctionType visit_fn)
- void inorderTraverse(FunctionType visit_fn)
- void postorderTraverse(FunctionType visit_fn)
  - FunctionType is a pointer to a function:
    - typedef void (*FunctionType)(TreeItemType& anItem);

BinaryTree: Implementation

- The complete implementation is in your text book
- In class, we will go through only some methods
  - Skipping straightforward methods
    - Such as `isEmpty`, `rootData`, and `setRootData` functions
  - Skipping some details
    - Such as throwing exceptions

```cpp
// Default constructor
BinaryTree::BinaryTree() : root(NULL) {

}


// Protected constructor
BinaryTree::BinaryTree(TreeNode *nodePtr) : root(nodePtr) {

}


// Constructor
BinaryTree::BinaryTree(const TreeItemType& rootItem) {
        root = new TreeNode(rootItem, NULL, NULL);
}
```

```
// Constructor
BinaryTree::BinaryTree(const TreeItemType& rootItem,
                                   BinaryTree& leftTree, BinaryTree& rightTree) {
        root = new TreeNode(rootItem, NULL, NULL);
        attachLeftSubtree(leftTree);
        attachRightSubtree(rightTree);
}

void BinaryTree::attachLeftSubtree(BinaryTree& leftTree) {
        // Assertion: nonempty tree; no left child
        if (!isEmpty() && (root->leftChildPtr == NULL)) {
                root->leftChildPtr = leftTree.root;
                leftTree.root = NULL    //done so can't reach the subtree with using this pointer again, only reach it
                                        with using the whole trees root pointer
        }
}

void BinaryTree::attachRightSubtree(BinaryTree& rightTree) {
        // Left as an exercise
}
```

```
// Copy constructor
BinaryTree::BinaryTree(const BinaryTree& tree) {
        copyTree(tree.root, root);
}


// Uses preorder traversal for the copy operation
// (Visits first the node and then the left and right children)
void BinaryTree::copyTree(TreeNode *treePtr, TreeNode *& newTreePtr) const {

        if (treePtr != NULL) {          // copy node
                newTreePtr = new TreeNode(treePtr->item, NULL, NULL);
                copyTree(treePtr->leftChildPtr, newTreePtr->leftChildPtr);
                copyTree(treePtr->rightChildPtr, newTreePtr->rightChildPtr);
        }
        else
                newTreePtr = NULL;  // copy empty tree
}
```

```
// Destructor
BinaryTree::~BinaryTree() {
	destroyTree(root);
}


// Uses postorder traversal for the destroy operation
// (Visits first the left and right children and then the node)
void BinaryTree::destroyTree(TreeNode *& treePtr) {

	if (treePtr != NULL){
		destroyTree(treePtr->leftChildPtr);
		destroyTree(treePtr->rightChildPtr);
		delete treePtr;
		treePtr = NULL;
	}
}
```

Binary Tree Traversals

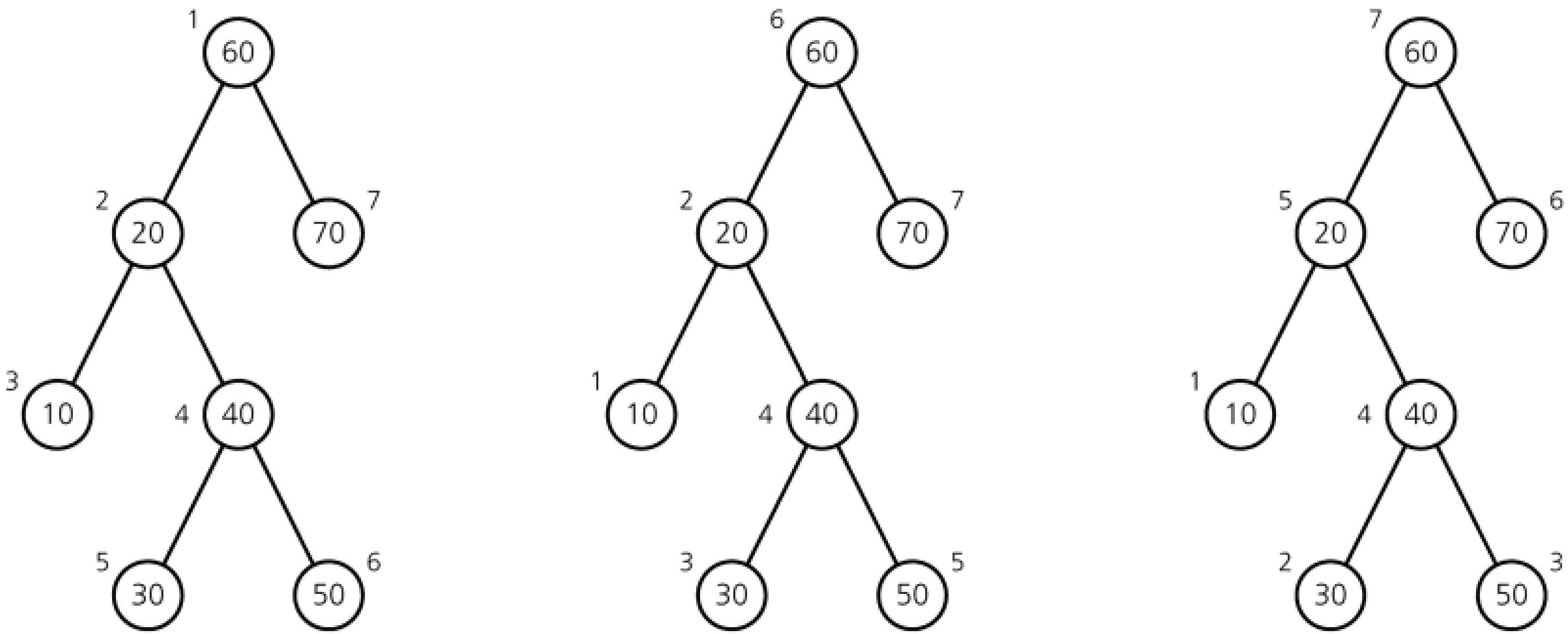
6 different traversals

- **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.)

```
void BinaryTree::preorderTraverse(FunctionType visit) {
        preorder(root, visit);
}
void BinaryTree::inorderTraverse(FunctionType visit) {
        inorder(root, visit);
}
void BinaryTree::postorderTraverse(FunctionType visit) {
        postorder(root, visit);
}
```

Remember that:

FunctionType is a pointer to a function

- Variables that point to the address of a function
- `typedef void (*FunctionType)(TreeItemType& anItem);`

Example of using inorderTraverse function:

- `void display(TreeItemType& anItem) { cout << anItem << endl; }`
- `BinaryTree T1;`
  `T1.inorderTraverse(display);`

```
void BinaryTree::preorder(TreeNode *treePtr, FunctionType visit) {
        if (treePtr != NULL) {
                visit(treePtr->item);
                preorder(treePtr->leftChildPtr, visit);
                preorder(treePtr->rightChildPtr, visit);
        }
}

void BinaryTree::inorder(TreeNode *treePtr, FunctionType visit) {
        if (treePtr != NULL) {
                inorder(treePtr->leftChildPtr, visit);
                visit(treePtr->item);
                inorder(treePtr->rightChildPtr, visit);
        }
}

void BinaryTree::postorder(TreeNode *treePtr, FunctionType visit) {
        if (treePtr != NULL) {
                postorder(treePtr->leftChildPtr, visit);
                postorder(treePtr->rightChildPtr, visit);
                visit(treePtr->item);
        }
}
```

Complexity of Traversals

What is the complexity of each traversal type?

O(n) --> visits every node

- Preorder traversal

- Postorder traversal

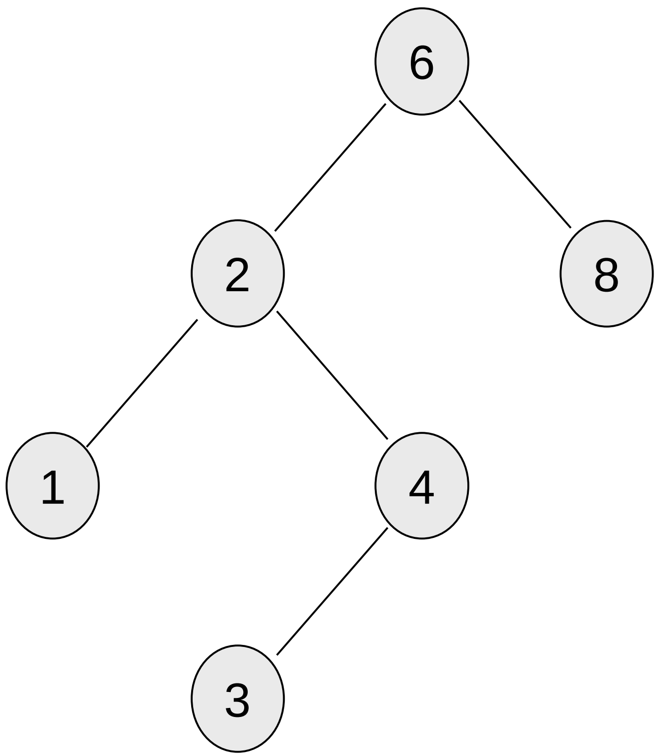
- Inorder traversal

Binary Search Tree

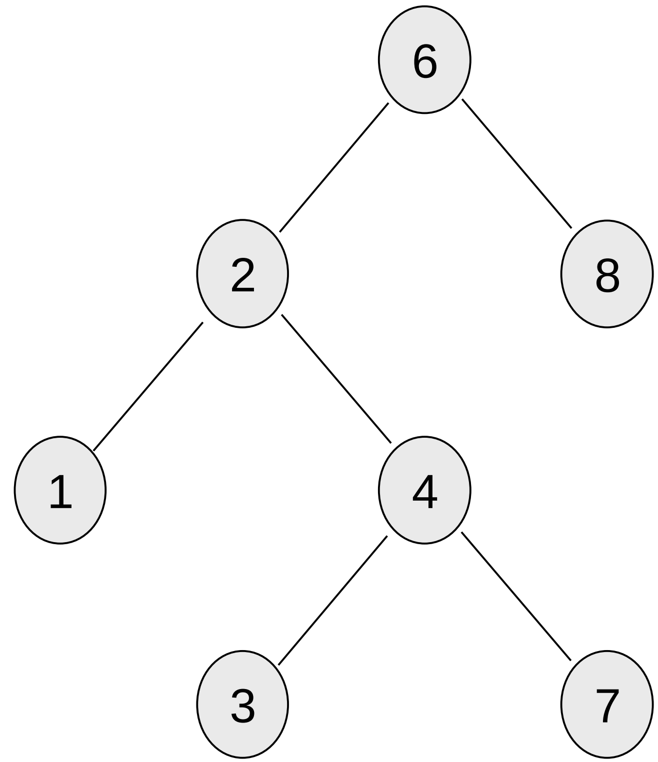
- An important application of binary trees is their use in searching.

- ***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 data value in X
  b) all data values in its **right subtree are larger** than data value in X
  c) the left and right subtrees are also binary search trees

Binary Search Tree

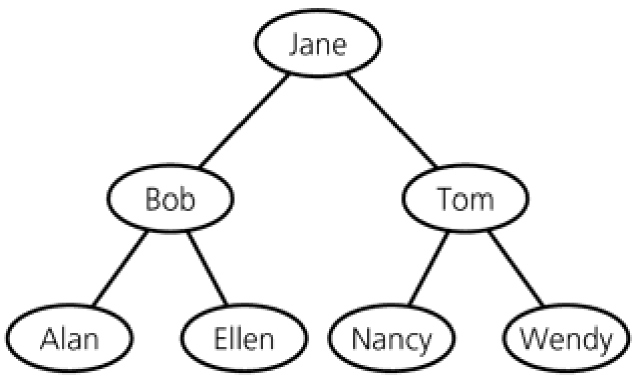


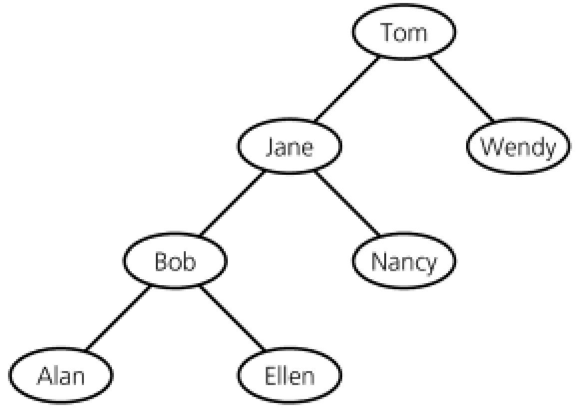
A *binary search tree*



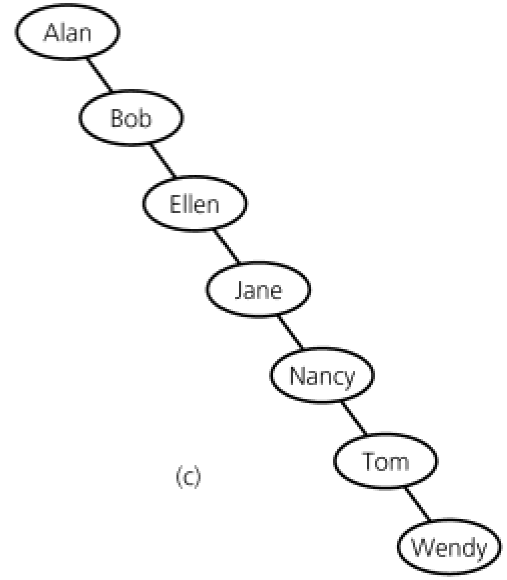
Not a *binary search tree*,
but a *binary tree* **Why?**

Binary Search Trees – containing same data

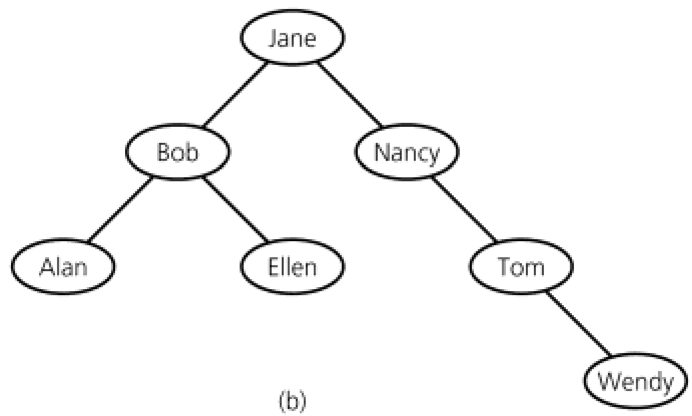




(a)



(c)



(b)

BinarySearchTree Class – UML Diagram

BinarySearchTree
root *left subtree* *right subtree*
createBinarySearchTree() *destroyBinarySearchTree()* *isEmpty()* *searchTreeInsert()* *searchTreeDelete()* *searchTreeRetrieve()* *preorderTraverse()* *inorderTraverse()* *postorderTraverse()*

The KeyedItem Class

```
typedef desired-type-of-search-key KeyType;

class KeyedItem {
public:
        KeyedItem() { }
        KeyedItem(const KeyType& keyValue) : searchKey(keyValue) { }

        KeyType getKey() const {
                return searchKey;
        }

private:
        KeyType searchKey;
        // ... and other data items
};
```

The TreeNode Class

```
typedef KeyedItem TreeItemType;

class TreeNode {     // a node in the tree
private:
        TreeNode() { }
        TreeNode(const TreeItemType& nodeItem,TreeNode *left = NULL,
                                                TreeNode *right = NULL)
        : item(nodeItem), leftChildPtr(left), rightChildPtr(right){ }


        TreeItemType item;                  // a data item in the tree
        TreeNode *leftChildPtr;        // pointers to children
        TreeNode *rightChildPtr;


    // friend class - can access private parts
    friend class BinarySearchTree;
};
```

The BinarySearchTree Class

- **Properties**
  - `TreeNode * root`
- **Constructors**
  - `BinarySearchTree();`
  - `BinarySearchTree(const BinarySearchTree& tree);`
- **Destructor**
  - `~BinarySearchTree();`

The BinarySearchTree Class

- **Public methods**

  - bool isEmpty() const;

  - void searchTreeRetrieve(KeyType searchKey, TreeItemType& item);
  - void searchTreeInsert(const TreeItemType& newItem);
  - void searchTreeDelete(KeyType searchKey);

  - void preorderTraverse(FunctionType visit);
  - void inorderTraverse(FunctionType visit);
  - void postorderTraverse(FunctionType visit);

  - BinarySearchTree& operator=(const BinarySearchTree& rhs);

The BinarySearchTree Class

- **Protected methods**

  - void retrieveItem(TreeNode *treePtr, KeyType searchKey,
    TreeItemType& item);

  - void insertItem(TreeNode * &treePtr,const TreeItemType& item);

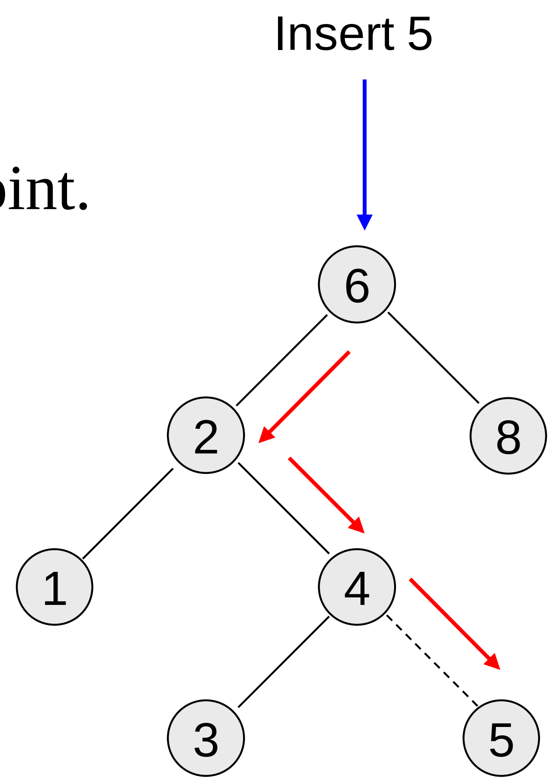
  - void deleteItem(TreeNode * &treePtr, KeyType searchKey);
  - void deleteNodeItem(TreeNode * &nodePtr);
  - void processLeftmost(TreeNode * &nodePtr, TreeItemType& item);

Searching (Retrieving) an Item in a BST

```
void BinarySearchTree::searchTreeRetrieve(KeyType searchKey,
                                          TreeItemType& treeItem) const throw(TreeException) {
        retrieveItem(root, searchKey, treeItem);
}


void BinarySearchTree::retrieveItem(TreeNode *treePtr, KeyType searchKey,
                                    TreeItemType& treeItem) const throw(TreeException) {

        if (treePtr == NULL)
                throw TreeException("TreeException: searchKey not found");
        else if (searchKey == treePtr->item.getKey())
                treeItem = treePtr->item;
        else if (searchKey < treePtr->item.getKey())
                retrieveItem(treePtr->leftChildPtr, searchKey, treeItem);
        else
                retrieveItem(treePtr->rightChildPtr, searchKey, treeItem);
}
```

Inserting an Item into a BST

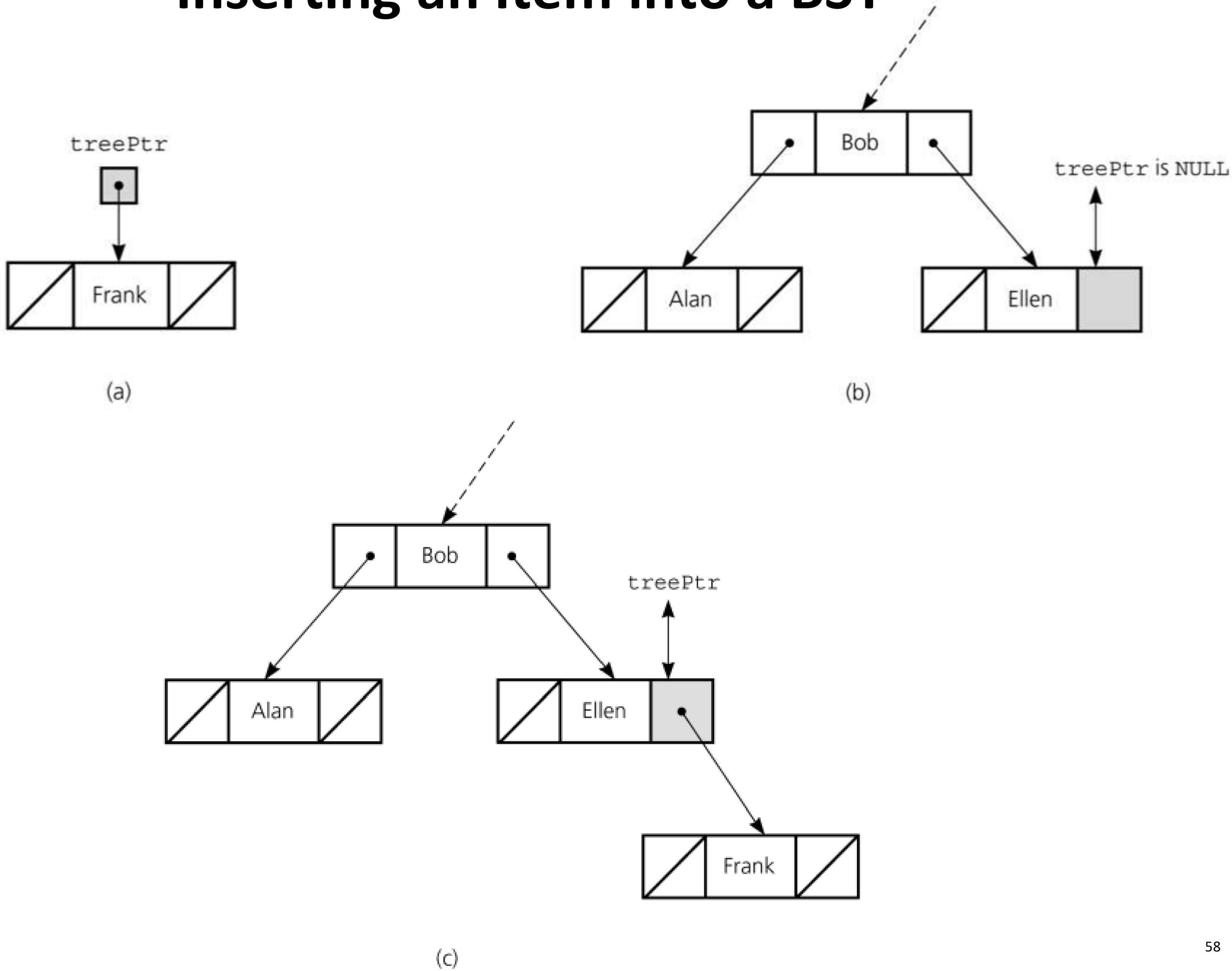
Search determines the insertion point.



Inserting an Item into a BST

```
void BinarySearchTree::searchTreeInsert(const TreeItemType& newItem) {
        insertItem(root, newItem);
}


void BinarySearchTree::insertItem(TreeNode *& treePtr,
                        const TreeItemType& newItem) throw(TreeException) {

        // Position of insertion found; insert after leaf
        if (treePtr == NULL) {
                treePtr = new TreeNode(newItem, NULL, NULL);
                if (treePtr == NULL)
                        throw TreeException("TreeException: insert failed");
        }
        // Else search for the insertion position
        else if (newItem.getKey() < treePtr->item.getKey())
                insertItem(treePtr->leftChildPtr, newItem);
        else
                insertItem(treePtr->rightChildPtr, newItem);
}
```

Inserting an Item into a BST



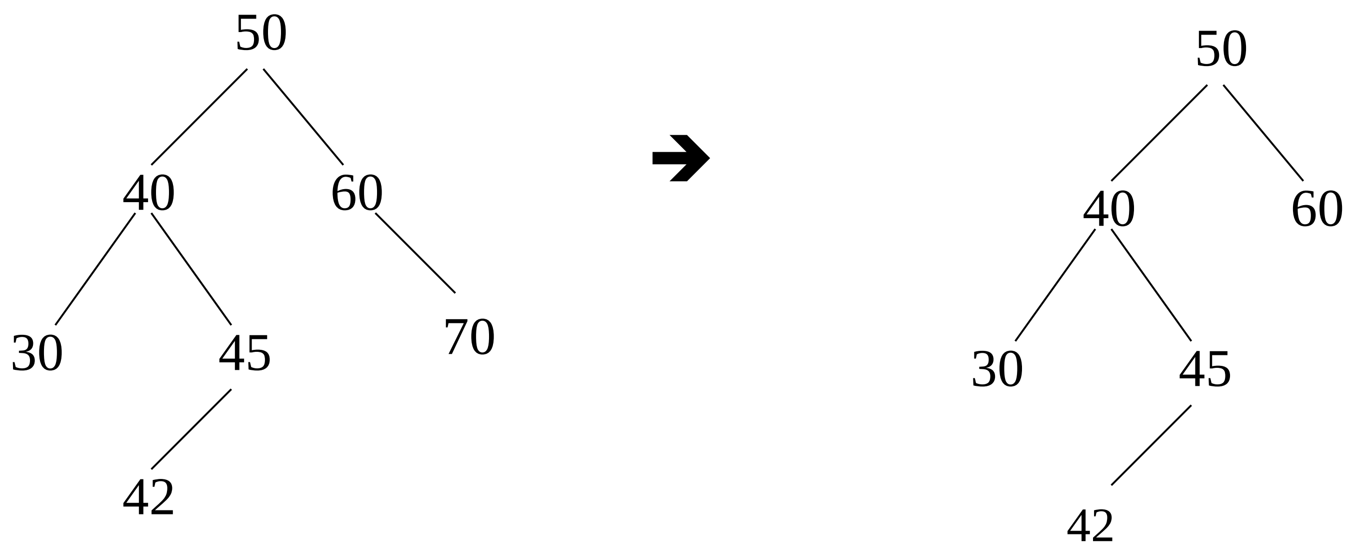
Deleting An Item from a BST

- To delete an item from a BST, we have to locate that item in that BST.

- The deleted node can be:
  - *Case 1* – A leaf node.
  - *Case 2* – A node with only with child
    (with left child or with right child).
  - *Case 3* – A node with two children.

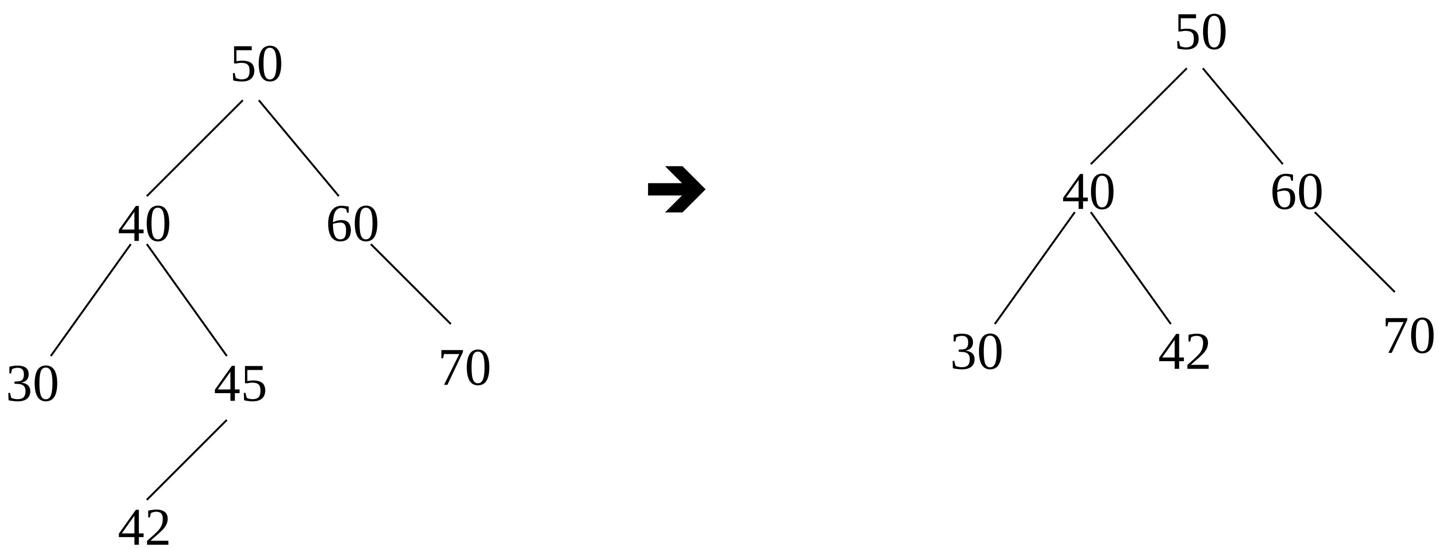
Deletion – Case 1: A Leaf Node

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



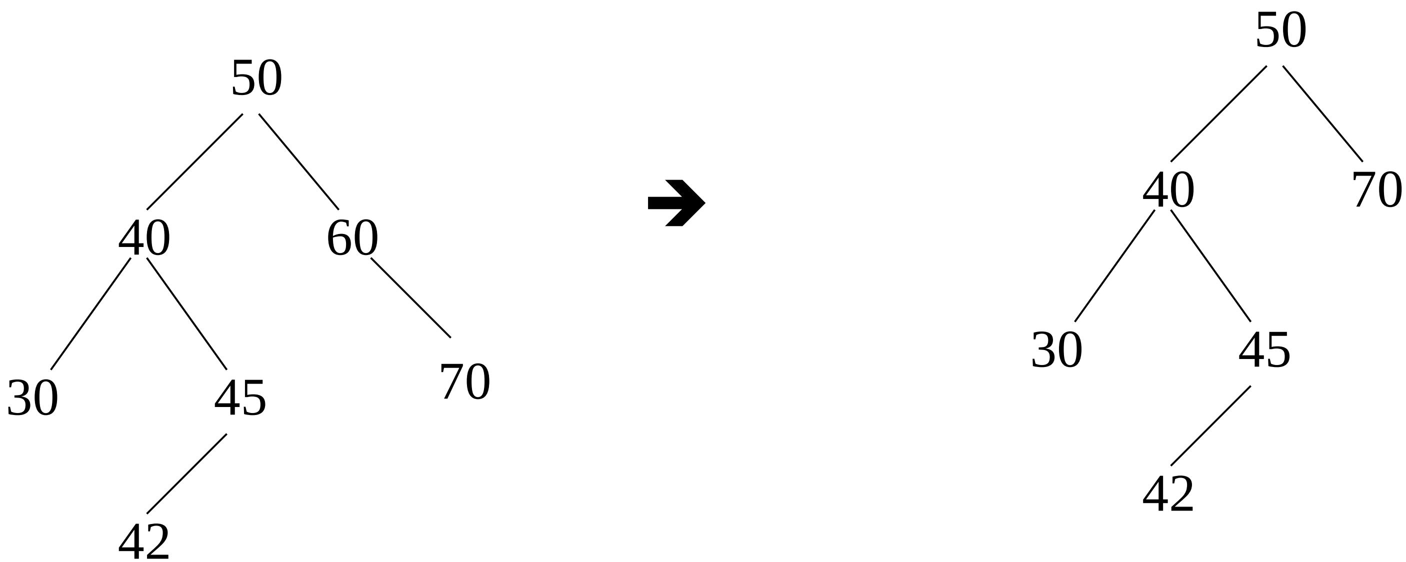
Delete 70 (A leaf node)

Deletion – Case 2: A Node with only a left child



Delete 45 (A node with only a left child)

Deletion – Case 2: A Node with only a right child



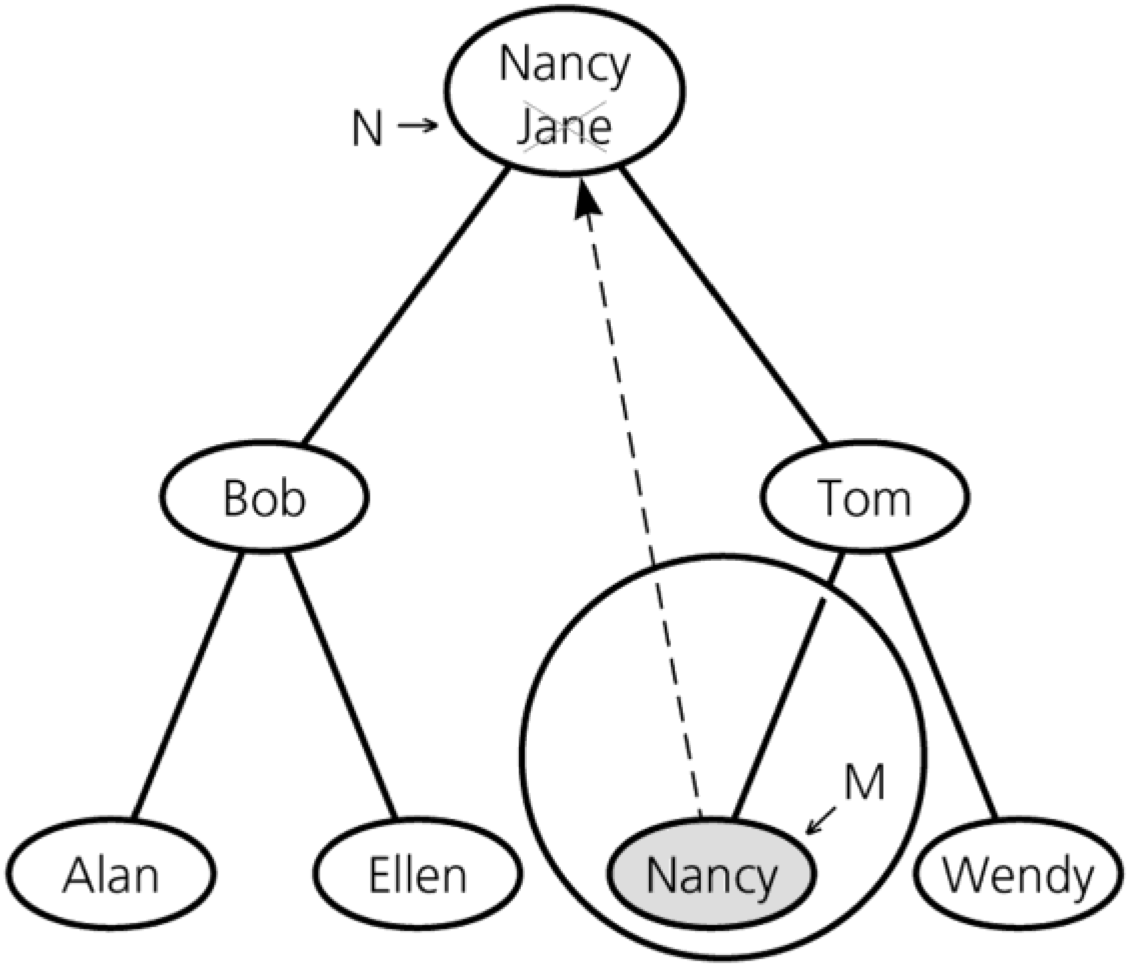
Delete 60 (A node with only a right child)

Deletion – Case 3: A Node with two children

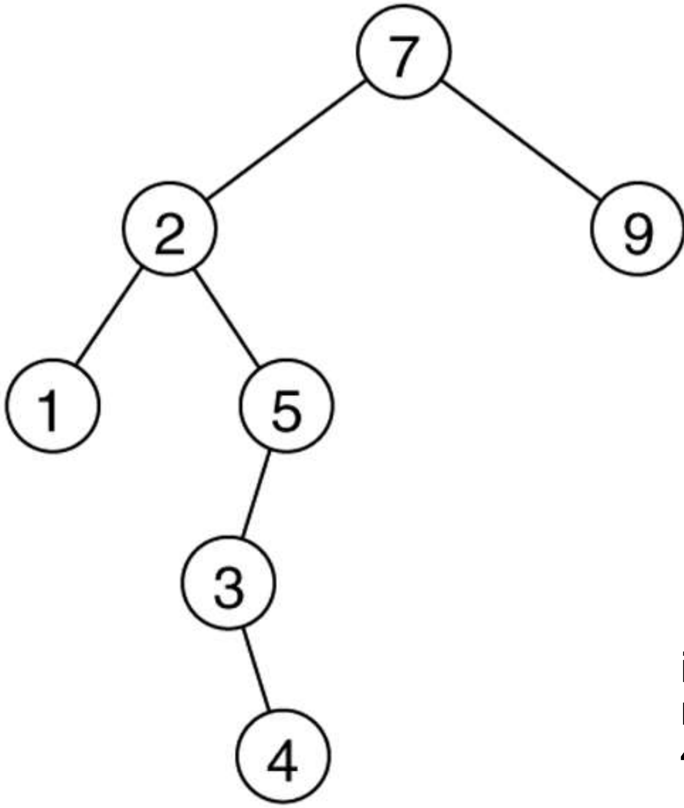
- **Locate** the inorder successor of the node.
- **Copy** the item in this node into the node which contains the item which will be deleted.
- **Delete** the node of the inorder successor.

Delete 40 (A node with two children)

Deletion – Case 3: A Node with two children



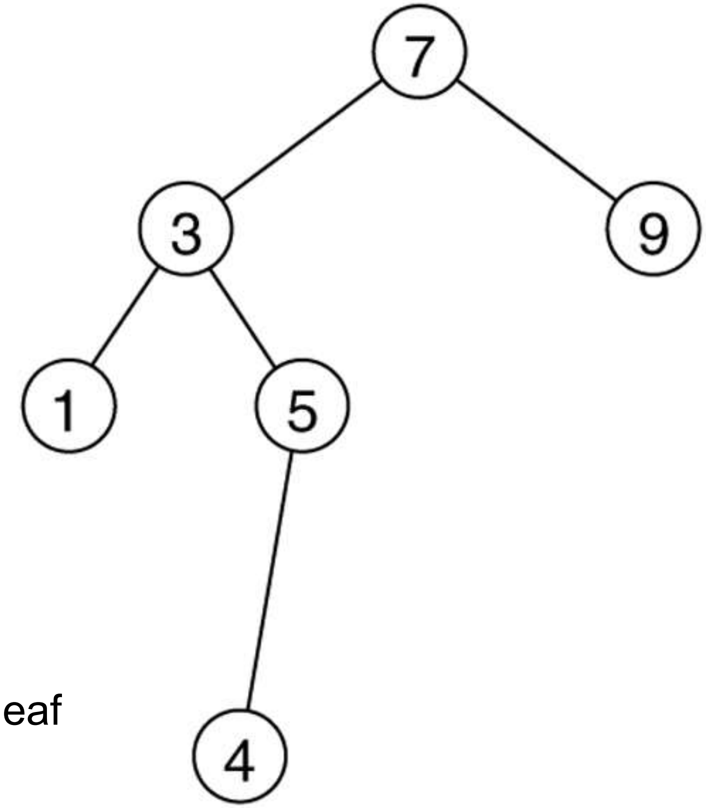
Deletion – Case 3: A Node with two children



(a)

➔

inorder successor = 3
right subtree's leftmost leaf
4 is right leaf here



(b)

Delete 2

Deletion from a BST

```
void BinarySearchTree::searchTreeDelete(KeyType searchKey)
                                                        throw(TreeException) {
        deleteItem(root, searchKey);
}

void BinarySearchTree::deleteItem(TreeNode * &treePtr, KeyType searchKey)
                                                        throw(TreeException) {
        if (treePtr == NULL) // Empty tree
                throw TreeException("Delete failed");

        // Position of deletion found
        else if (searchKey == treePtr->item.getKey())
                deleteNodeItem(treePtr);

        // Else search for the deletion position
        else if (searchKey < treePtr->item.getKey())
                deleteItem(treePtr->leftChildPtr, searchKey);
        else
                deleteItem(treePtr->rightChildPtr, searchKey);
}
```

Deletion from a BST

```
void BinarySearchTree::deleteNodeItem(TreeNode * &nodePtr) {
        TreeNode *delPtr;
        TreeItemType replacementItem;


        // (1)  Test for a leaf
        if ( (nodePtr->leftChildPtr == NULL) &&
           (nodePtr->rightChildPtr == NULL) ) {
                delete nodePtr;
                nodePtr = NULL;
        }


        // (2)  Test for no left child
        else if (nodePtr->leftChildPtr == NULL){
                delPtr = nodePtr;
                nodePtr = nodePtr->rightChildPtr;
                delPtr->rightChildPtr = NULL;
                delete delPtr;
        }
```

right child will replace deleted node

swap

Deletion from a BST

```
// (3)  Test for no right child
else if (nodePtr->rightChildPtr == NULL) {
            // ...                              left child will replace deleted node
            // Left as an exercise
}


// (4)  There are two children:
//      Retrieve and delete the inorder successor          right subtree's leftmost leaf
else {
            processLeftmost(nodePtr->rightChildPtr, replacementItem);
            nodePtr->item = replacementItem;
}

}
```

Deletion from a BST

```
void BinarySearchTree::processLeftmost(TreeNode *&nodePtr,
                                       TreeItemType& treeItem){

    if (nodePtr->leftChildPtr == NULL) {
        treeItem = nodePtr->item;
        TreeNode *delPtr = nodePtr;
        nodePtr = nodePtr->rightChildPtr;
        delPtr->rightChildPtr = NULL; // defense
        delete delPtr;
    }
    else
        processLeftmost(nodePtr->leftChildPtr, treeItem);
}
```

copy inorder successor, change it to nodeToDelete and delete the node

inorder successor might have a right child

Traversals

- The traversals for binary search trees are same as the traversals for the binary trees.

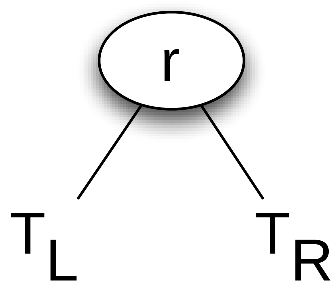
Theorem: Inorder traversal of a binary search tree will visit its nodes in sorted search-key order.

Proof: Proof by induction on the height of the binary search tree T.

Basis: h=0 ➔ no nodes are visited, empty list is in sorted order.

Inductive Hypothesis: Assume that the theorem is true for all k, $0 \leq k < h$

Inductive Conclusion: We have to show that the theorem is true for $k=h>0$. T should be:



Since the lengths of T_L and T_R are less than h, the theorem holds for them. All the keys in T_L are less than r, and all the keys in T_R are greater than r. In inorder traversal, we visit T_L first, then r, and then T_R. Thus, the theorem holds for T with height k=h.

Minimum Height

- **Complete trees and full trees have minimum height.**
- The height of an n-node binary search tree ranges from $\lceil \log_2(n+1) \rceil$ to n.

- **Insertion in search-key order** produces a maximum-height BST.
- **Insertion in random order** produces a near-minimum-height BST.

- That is, the height of an n-node binary search tree
  - *Best Case –* $\lceil \log_2(n+1) \rceil$ → $O(\log_2 n)$
  - *Worst Case –* n → $O(n)$
  - *Average Case –* close to $\lceil \log_2(n+1) \rceil$ → $O(\log_2 n)$

Average Height

- If we insert n items into an empty BST to create a BST with n nodes,
  - How many different binary search trees with n nodes?
  - What are their probabilities?

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

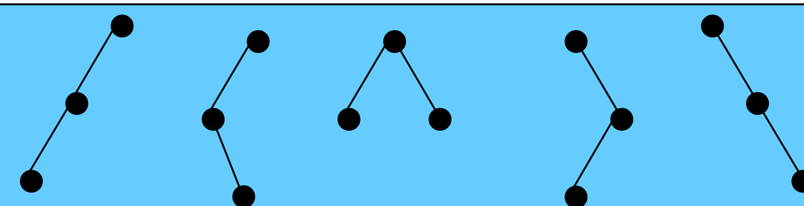
    n=0 → 1 BST (empty tree)

    n=1 → 1 BST (a binary tree with a single node)

    n=2 → 2 BSTs

    n=3 → 5 BSTs

<u>When n=3</u>



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

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.**

```
// Sorts n integers in an array anArray into
// ascending order
treesort(inout anArray:ArrayType, in n:integer)

        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: $O(n)$ height = n, for example insert 10 11 12 13 14 15 sorted order
  - 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 same key comparisons of keys as does quicksort when the pivot for each sublist is chosen to be the first key

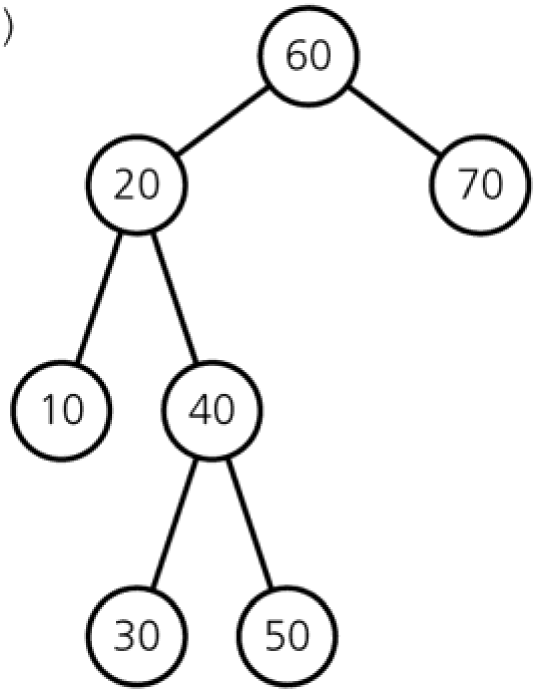
Saving a BST into a file and restoring it to its original shape

- **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

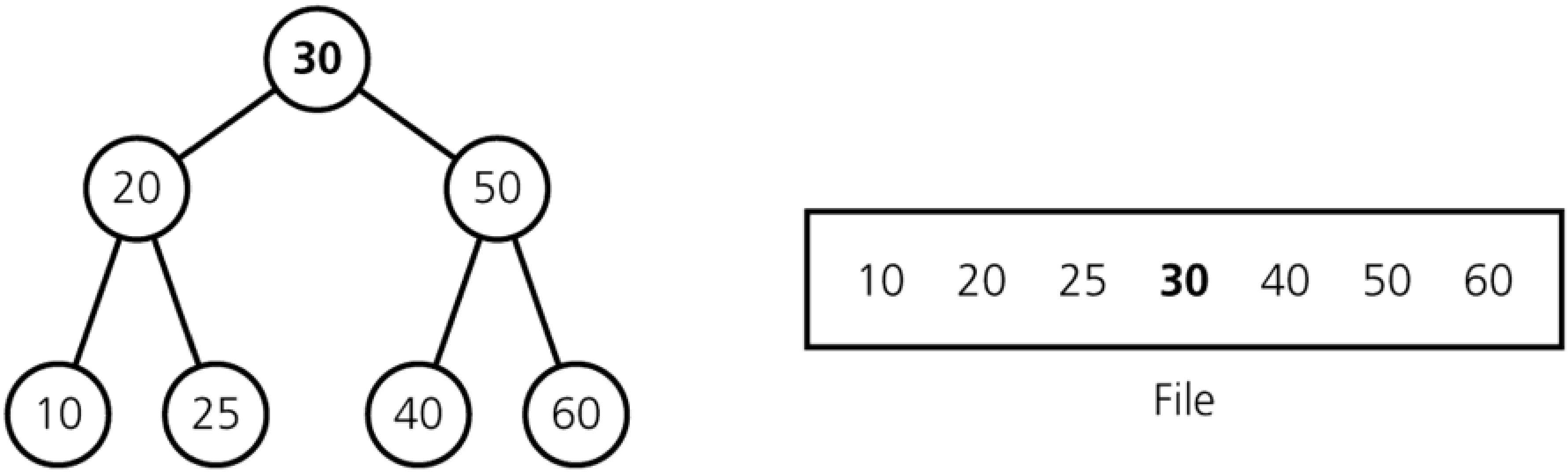
- **Restore:**
  - Read the number of nodes (n)
  - Start with an empty BST
  - Read the nodes from the file one by one to create a minimum-height binary search tree

Building a minimum-height BST

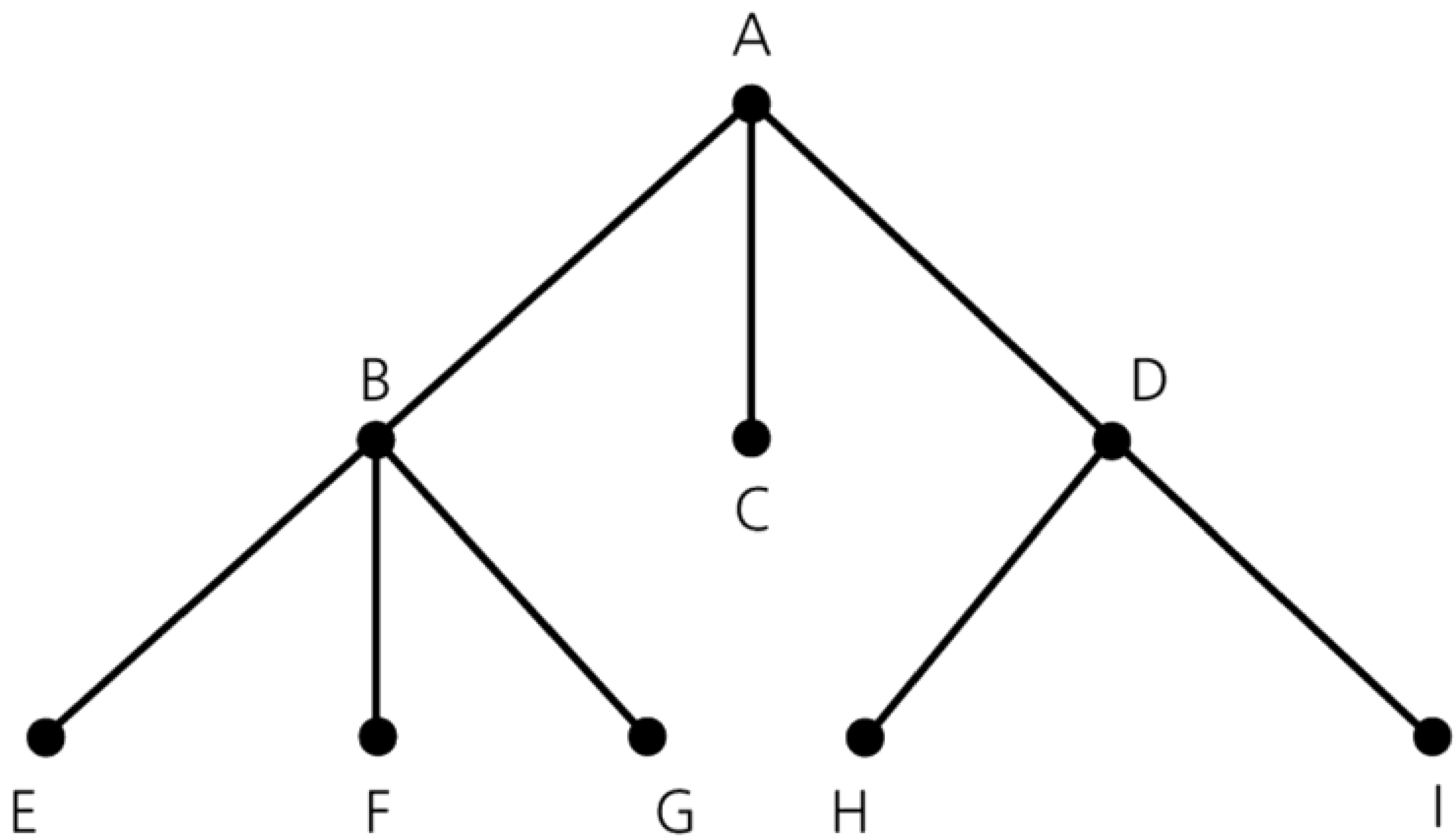
```
// Builds a minimum-height binary search tree from n sorted
// values in a file. treePtr will point to the tree's root.
readTree(out treePtr:TreeNodePtr, in n:integer)

        if (n>0) {
                treePtr = pointer to new node with NULL child pointers

                // construct the left subtree
                readTree(treePtr->leftChildPtr, n/2)

                // get the root
                Read item from file into treePtr->item

                // construct the right subtree
                readTree(treePtr->rightChildPtr, (n-1)/2)
        }
```

integer division

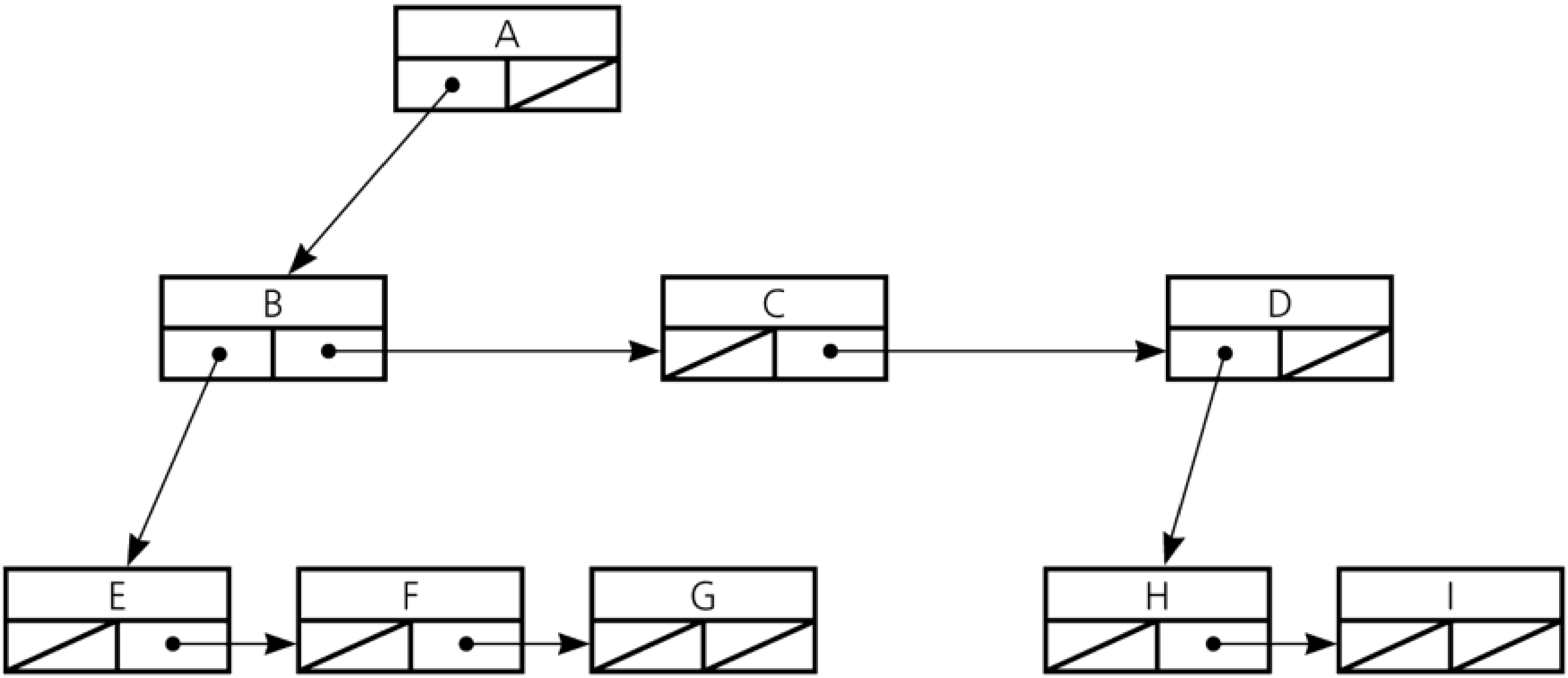
A full tree saved in a file by using inorder traversal



A General Tree

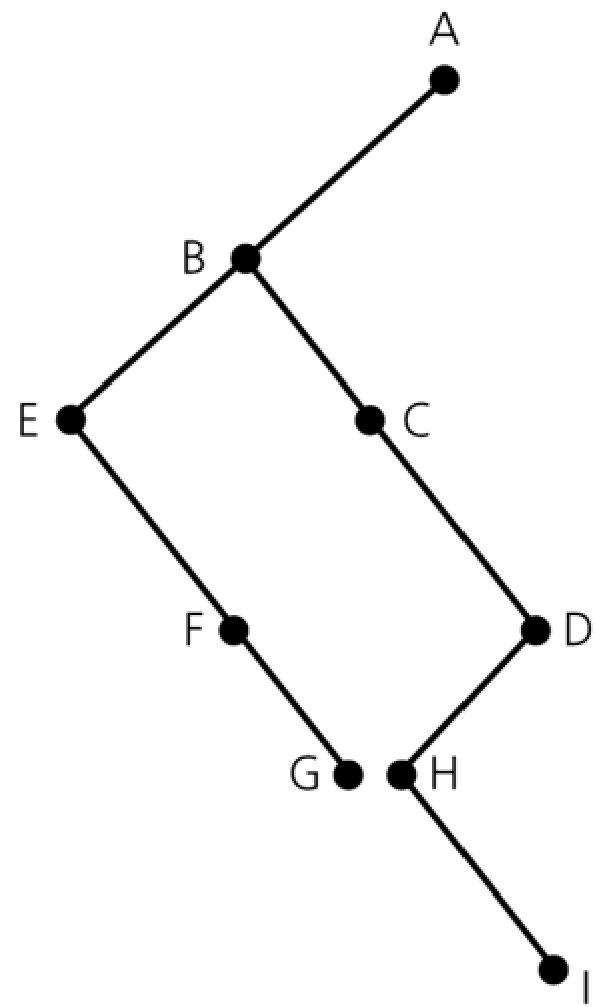


A Pointer-Based Implementation of General Trees



A Pointer-Based Implementation of General Trees

A pointer-based implementation of
a general tree can also represent
a binary tree.



N-ary Tree

An **n-ary tree** is a generalization of a binary whose nodes each can have no more than n children.