CS302 - Data Structures *using C++*

Topic: Trees

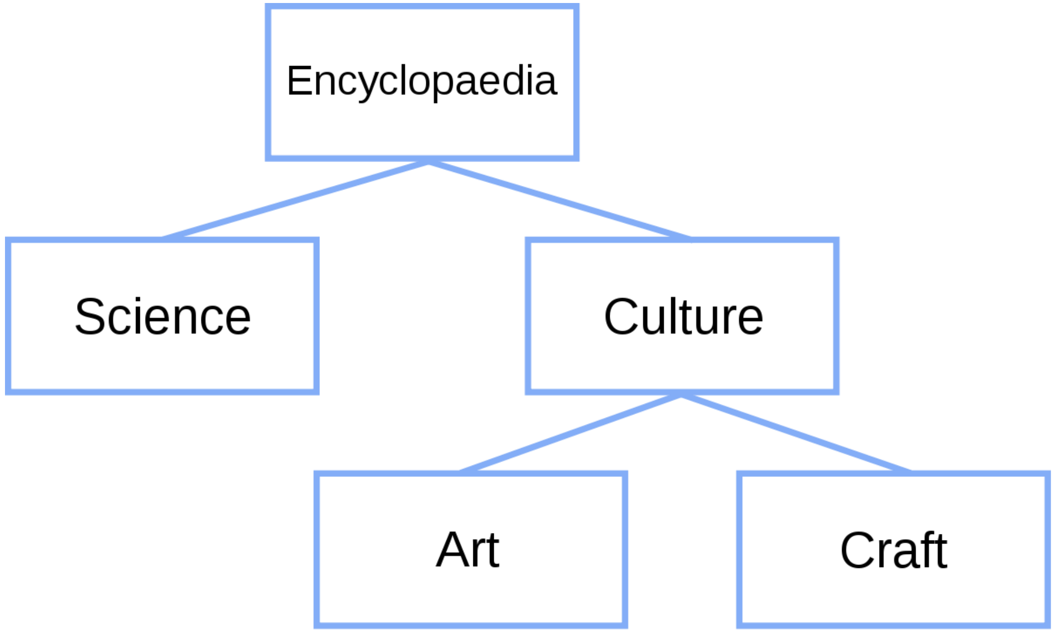
Kostas Alexis

Trees

- List, stacks, and queues are linear in their organization of data.
  - Items are one after another
- In this section, we organize data in a nonlinear hierarchical form.
  - Item can have more than one immediate successor.

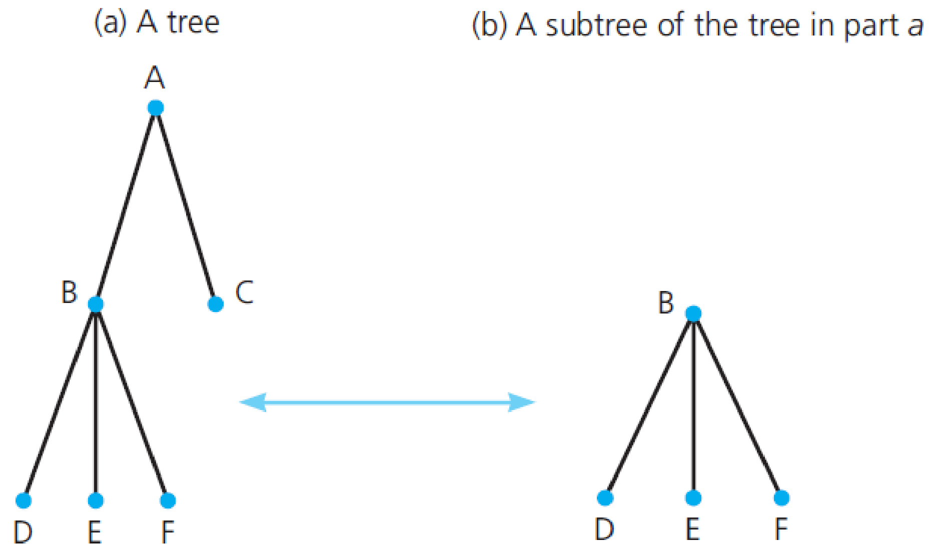
Terminology

- Use trees to represent relationships



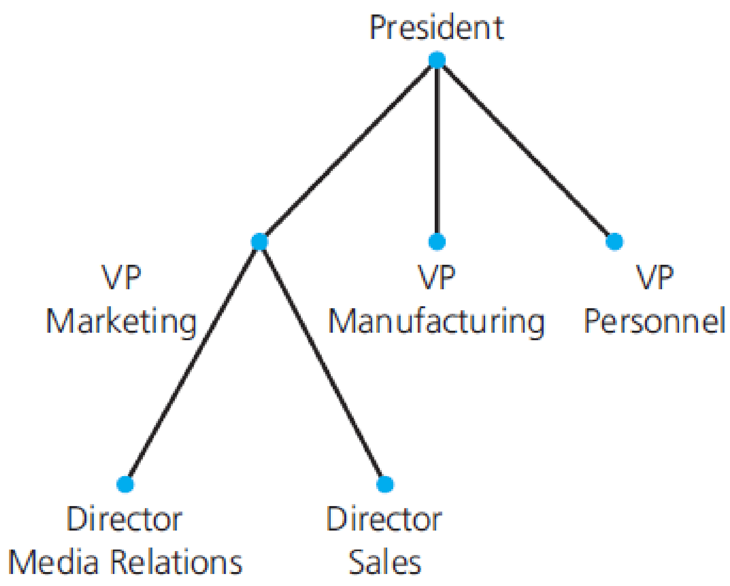
Terminology

- Trees are hierarchical in nature
  - Means a parent-child relationship between nodes
- A tree and one of its subtrees

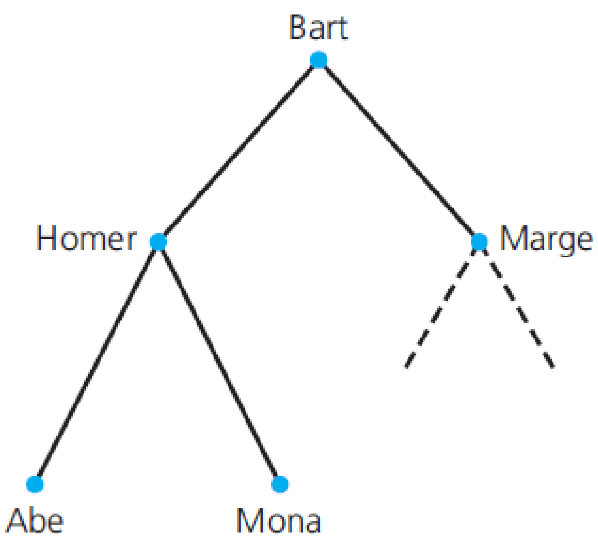


Terminology

- Further visual examples



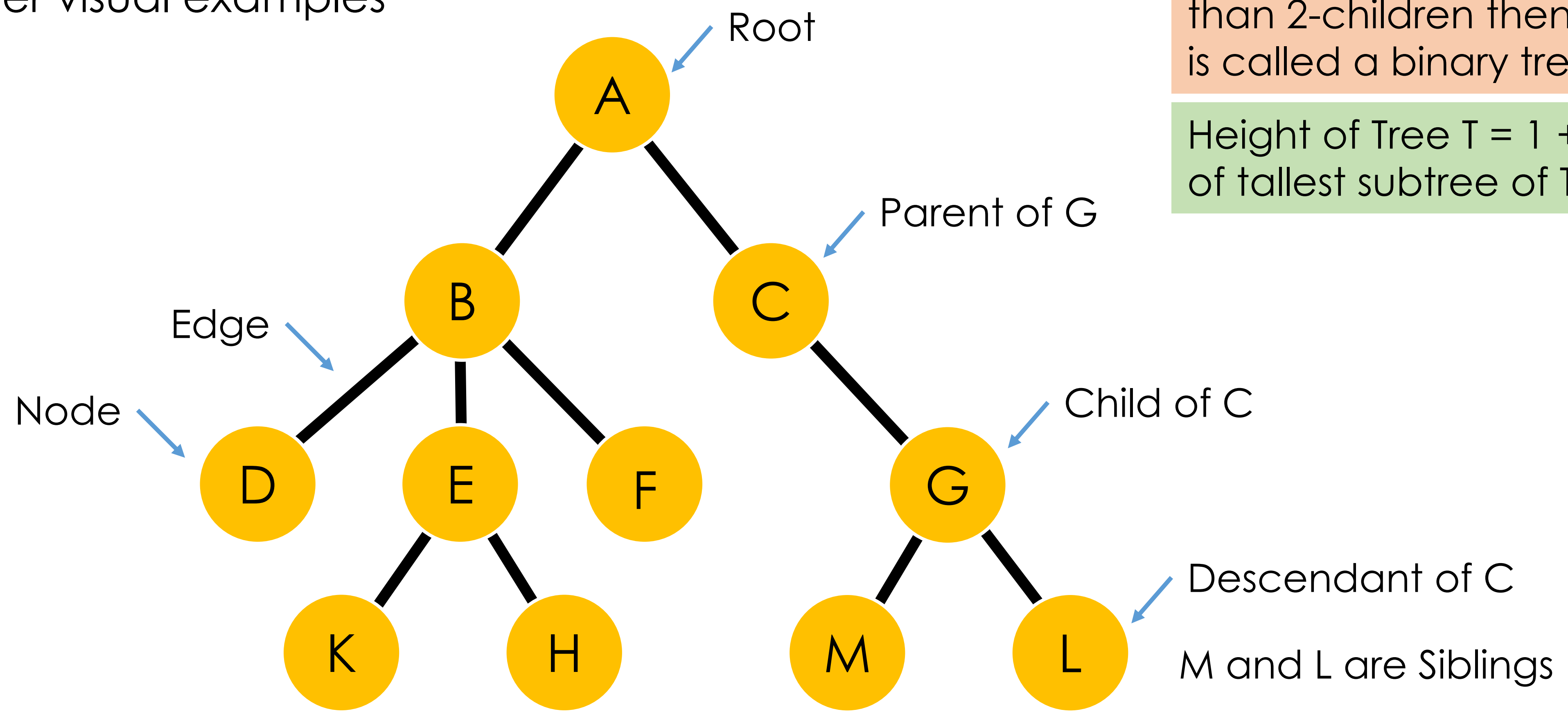
(a) An organization chart



(b) A family tree

Terminology

- Further visual examples



Node with no children: leaf

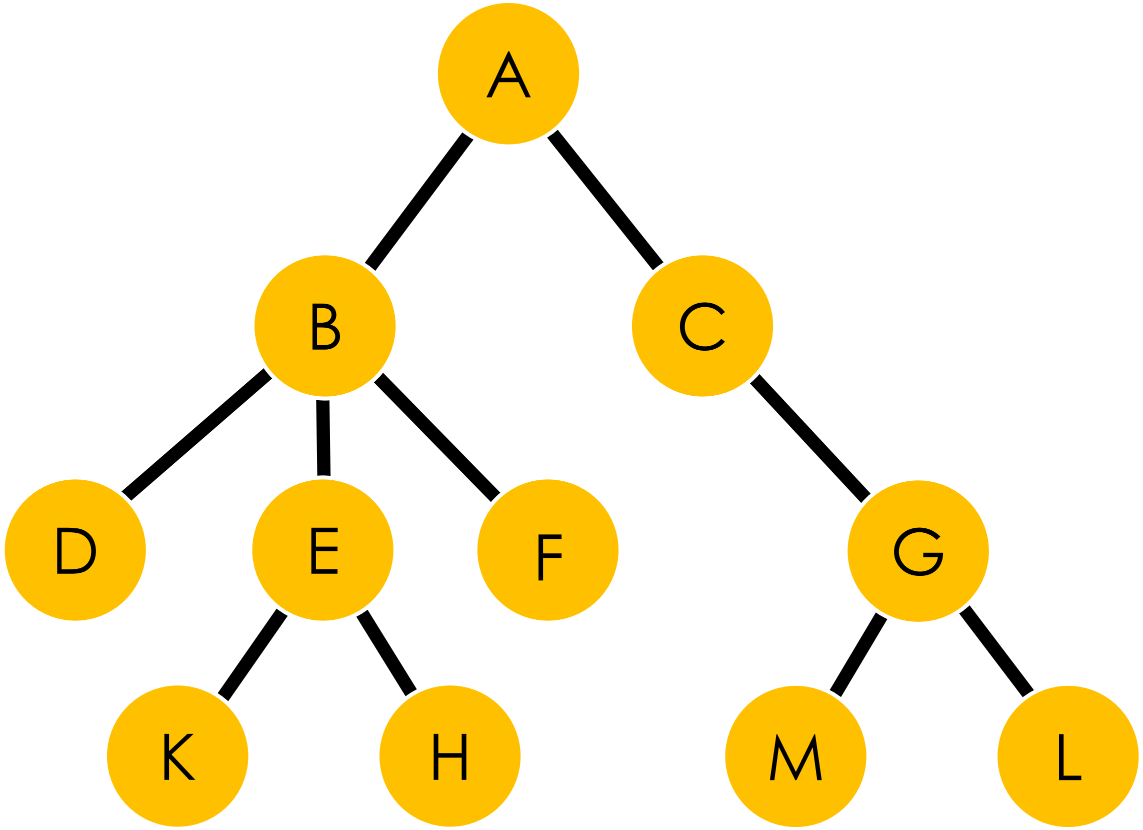
If each node has no more than n-children then this tree is called n-ary tree

If each node has no more than 2-children then this tree is called a binary tree

Height of Tree T = 1 + height of tallest subtree of T

Terminology

- Length of a path = number of edges
- Depth of a node x = length of path from root to x
- Height of node x = length of longest path from x to leaf
- Depth and height of tree = height of root
- The label of a node: A, B, C …

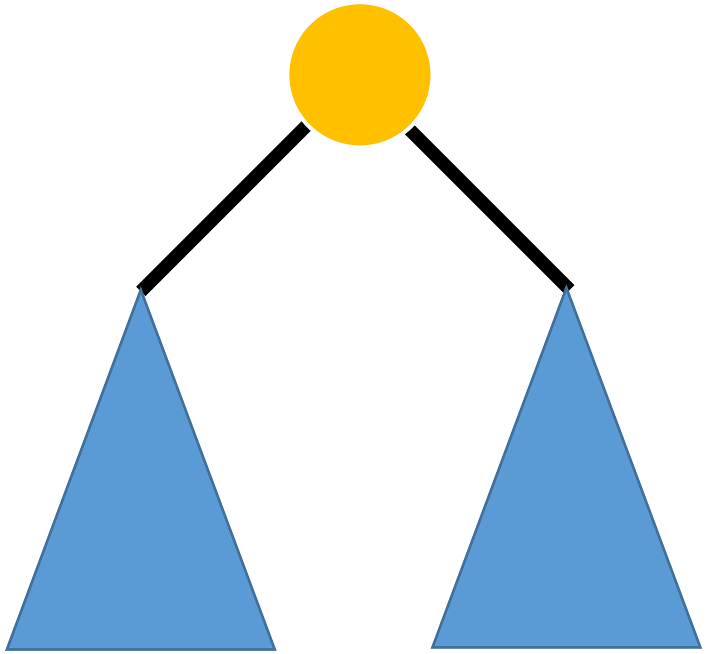


Terminology

- **Graph-theoretic definition of a Tree:** A tree is a graph for which there exists a node, called root such that
  - For any node x, there exists exactly one path from the root to x
- **Recursive Definition of a Tree:** A tree is either
  - Empty or
  - It has a node called the root, followed by zero or more trees called subtrees

Terminology

- Think of a Tree in a recursive manner



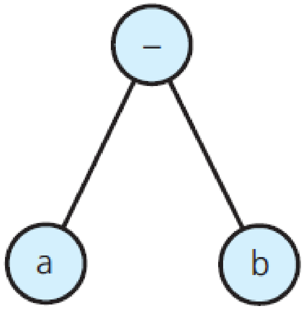
Kinds of Trees (first examples)

- General Tree
  - Set **T** of one or more nodes
  - **T** is partitioned into disjoint subsets
- Binary Tree
  - Set of **T** nodes – either empty or partitioned into disjoint subsets
  - Single node **r**, the root
  - Two (possibly empty) sets – left and right subtrees

Kinds of Trees

- Binary Trees that represent algebraic expressions

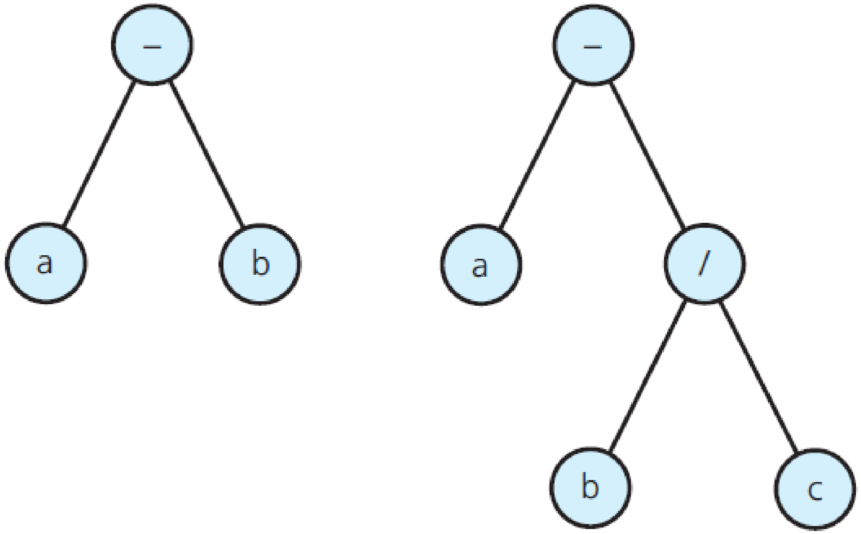
(a) a – b



Kinds of Trees

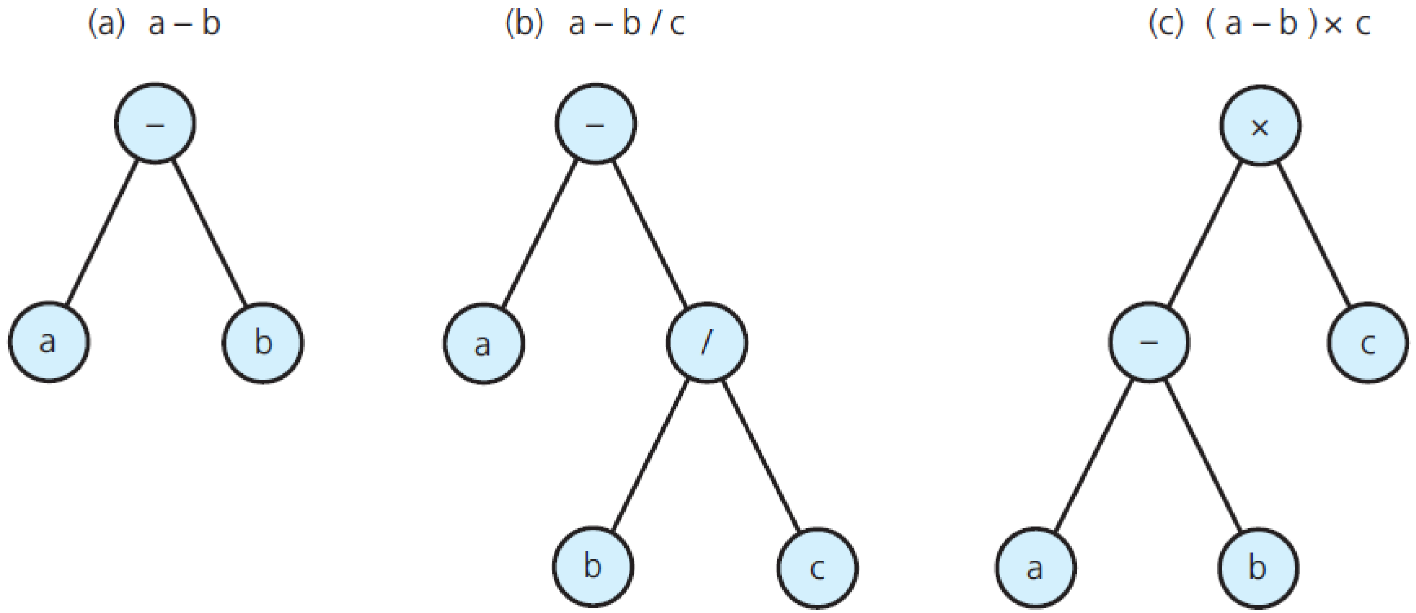
- Binary Trees that represent algebraic expressions

(a) a – b (b) a – b / c



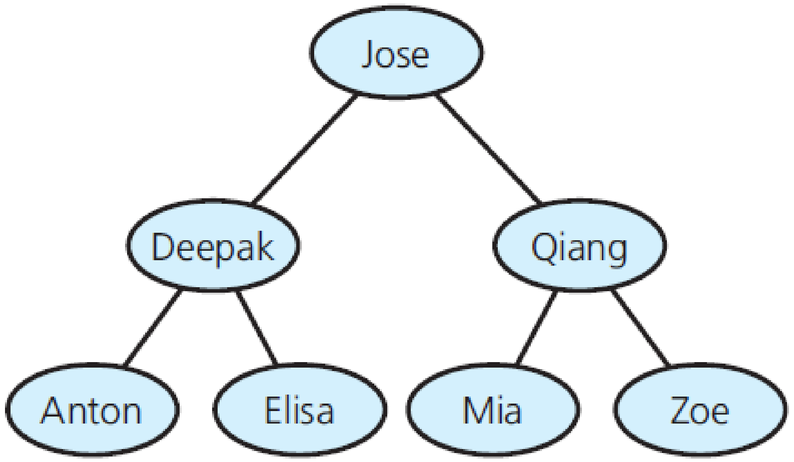
Kinds of Trees

- Binary Trees that represent algebraic expressions



Kinds of Trees

- Binary Search Tree of names

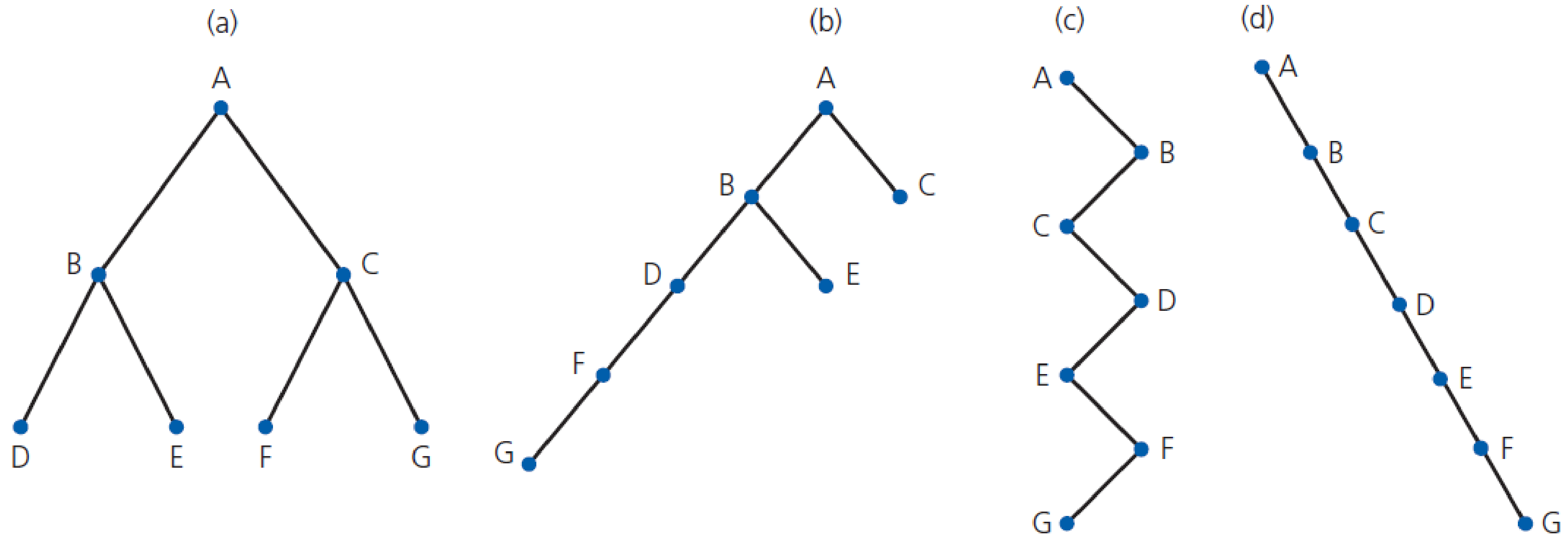


The Height of Trees

- Level of a node, **n**
  - If **n** is root, level 1
  - If **n** not the root, level is 1 greater than level of its parent
- Height of a tree
  - Number of nodes on longest path from root to a leaf
  - **T** empty, height 0
  - **T** not empty, height equal to max level of nodes

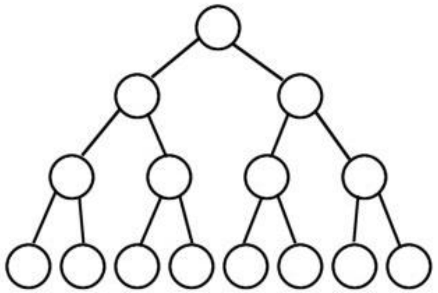
The Height of Trees

- Binary Trees with the same nodes but different heights

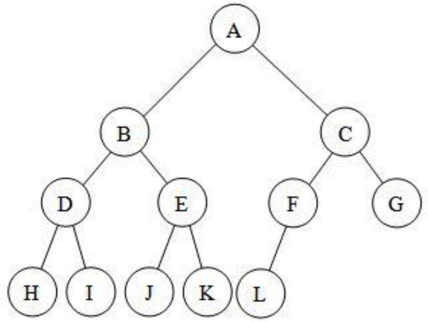


Full, Complete, and Balanced Binary Trees

- A **full binary tree** (sometimes also called a proper tree or a 2-tree) is a tree in which every node other than the leaves has two children.



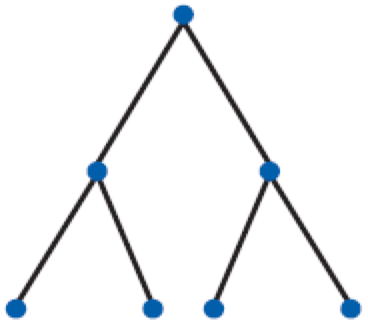
- A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled and all nodes are as far left as possible.



AUTONOMOUS ROBOTS LAB

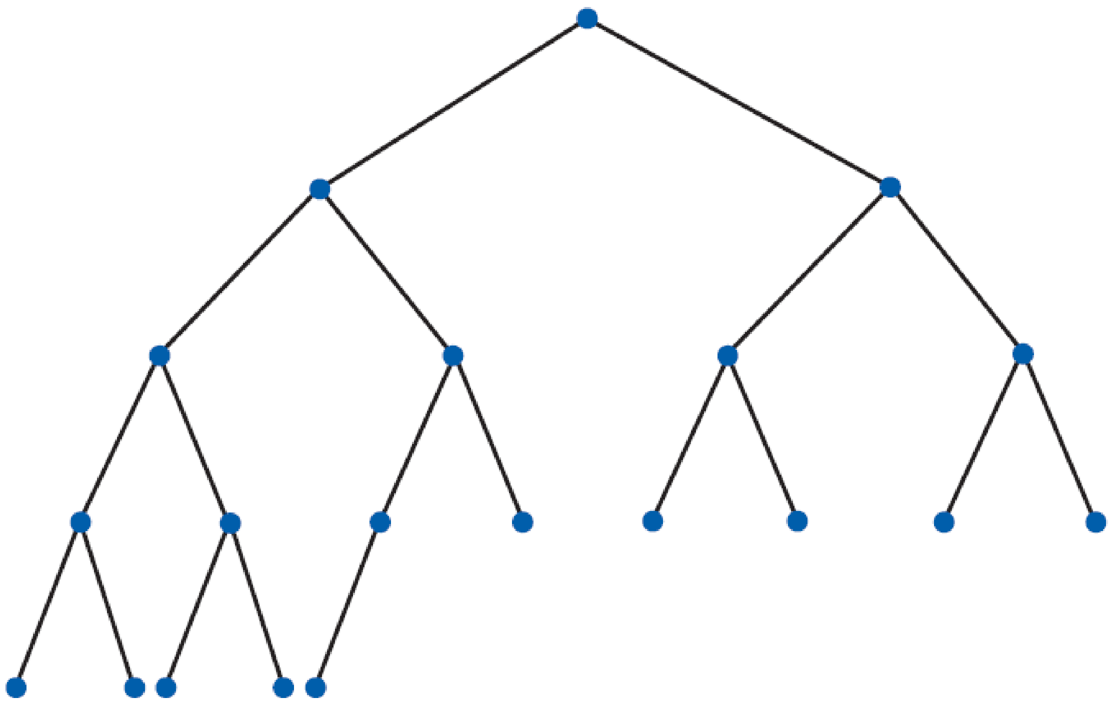
Full, Complete, and Balanced Binary Trees

- A **full binary tree** of height 3



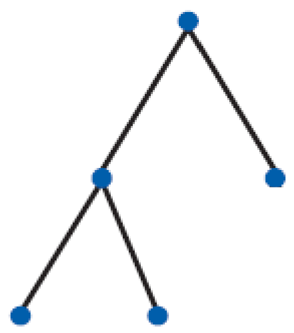
Full, Complete, and Balanced Binary Trees

- A **complete binary tree**



Full, Complete, and Balanced Binary Trees

- A **balanced binary tree** is a binary tree in which the left and right subtrees of every node differ in height by no more than 1.

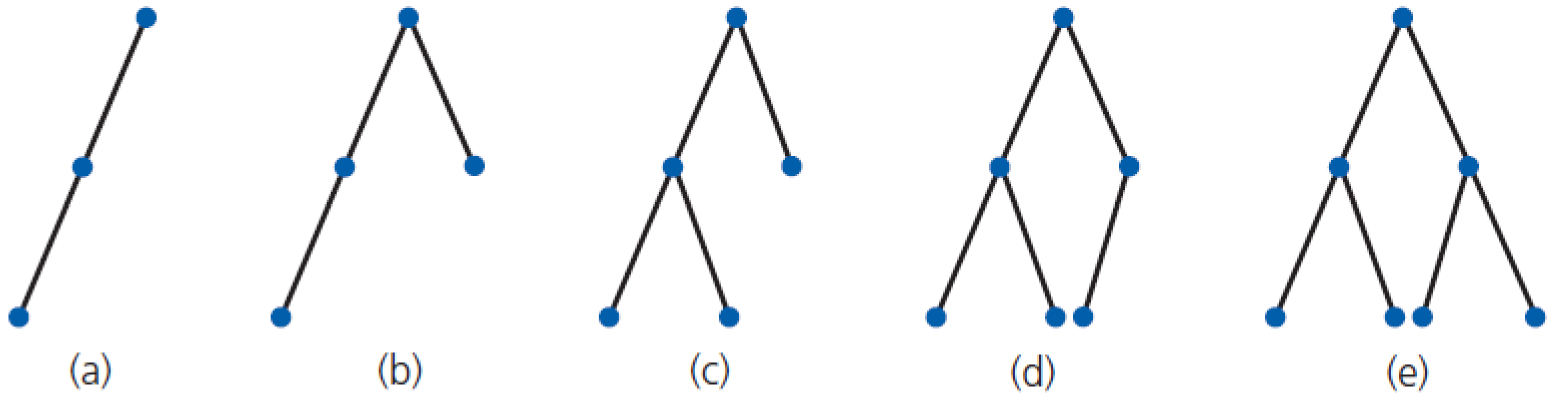


Max and Min Heights of a Binary Tree

- Binary Tree with **n** nodes
  - Max height is **n**
- To minimize the height of a binary tree of n nodes
  - Fill each level of a tree as completely as possible
  - A complete tree meets this requirement

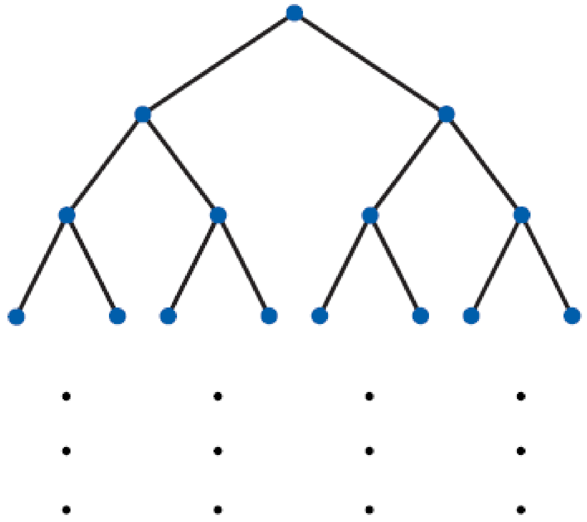
Max and Min Heights of a Binary Tree

- Binary Trees of height 3



Max and Min Heights of a Binary Tree

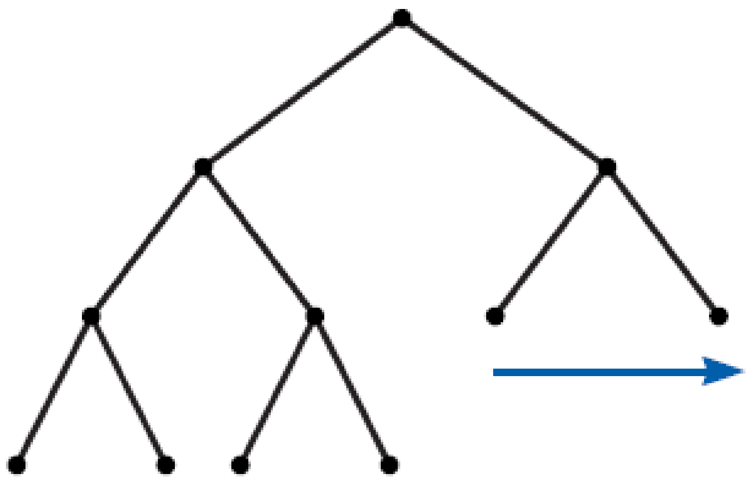
- Counting the nodes in a full binary tree of height h



Level	Number of nodes at this level	Total number of nodes at this level and all 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

Max and Min Heights of a Binary Tree

- Filling in the last level of a tree



The ADT Binary Tree

- Operations of ADT binary tree
  - **Add, remove**
  - **Set, retrieve data**
  - **Test for empty**
  - **Traversal operations that visit every node**
- Traversal can visit nodes in several different orders

Traversals of a Binary Tree

- Pseudocode for general form of a recursive traversal algoritm

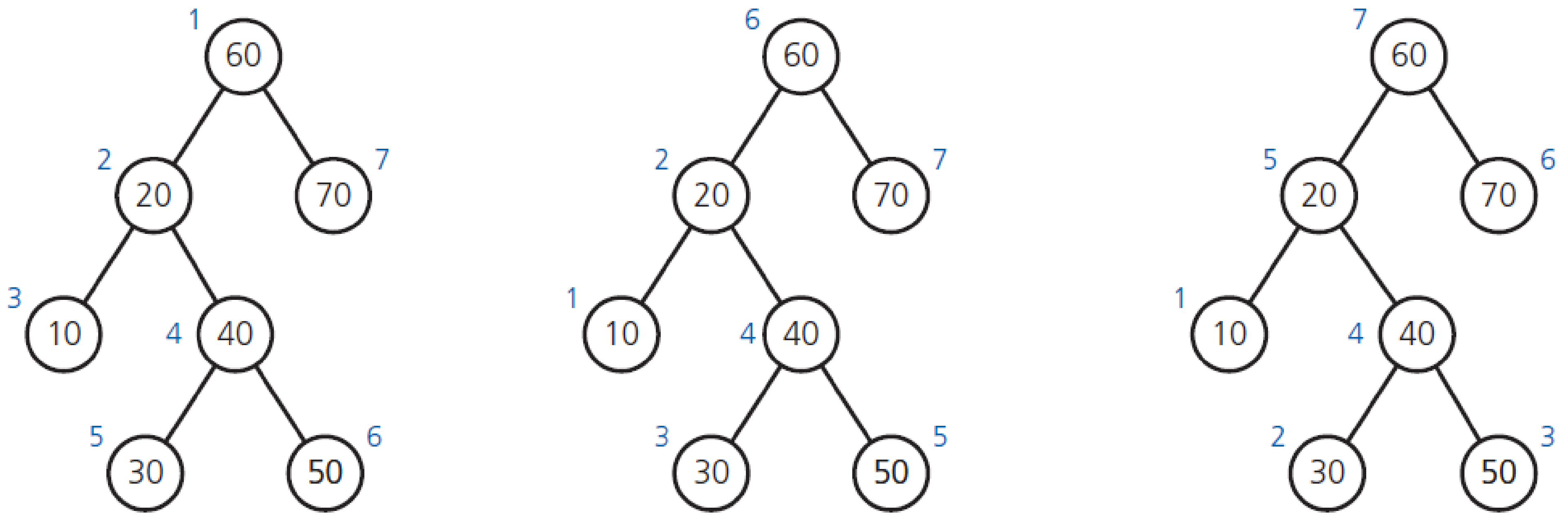
```
if (T is not empty)
{
    Display the data in T’s root
    Traverse T’s left subtree
    Traverse T’s right subtree
}
```

Traversals of a Binary Tree

- Options for when to visit the root (defines traversal type)
  - **Preorder:** before it traverses both subtrees
  - **Inorder:** after it traverses left subtree, before it traverses right subtree
  - **Postorder:** after it traverses both subtrees
- Node traversal is O(n)

Traversals of a Binary Tree

- Three traversals of a binary 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.)

Traversals of a Binary Tree

- Preorder traversal algorithm
  - Process (visit) root
  - Traverse(Left subtree of Tree's root)
  - Traverse(Right subtree of Tree's root)
- Exploits recursive character of how a tree is built
- A type of depth-first traversal

Traversals of a Binary Tree

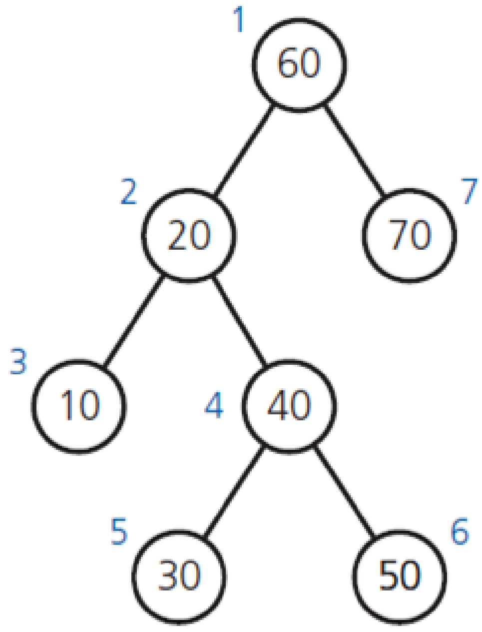
- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```

Traversals of a Binary Tree

- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```

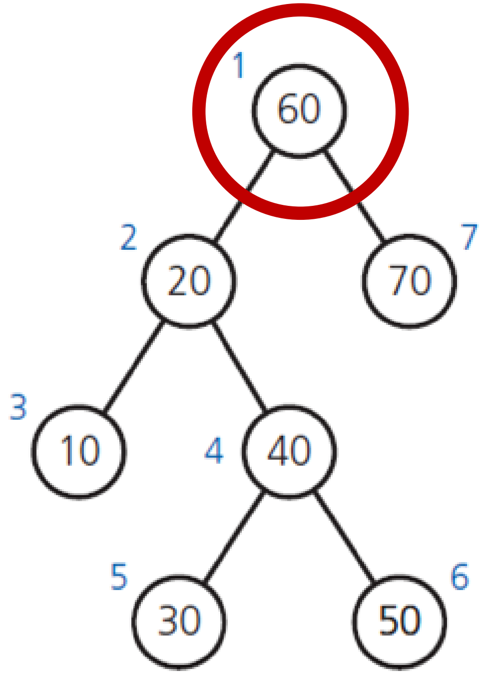


(a) Preorder: 60, 20, 10, 40, 30, 50, 70

Traversals of a Binary Tree

- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```

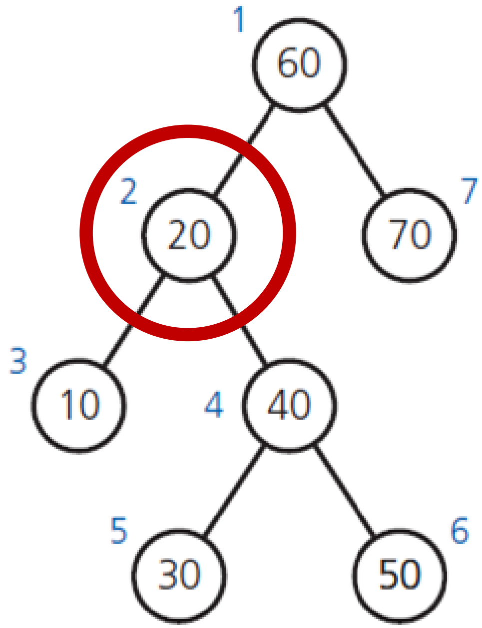


(a) Preorder: 60, 20, 10, 40, 30, 50, 70

Traversals of a Binary Tree

- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```

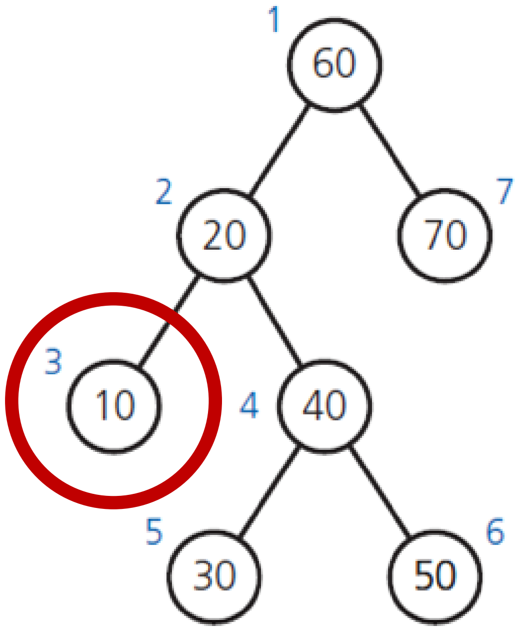


(a) Preorder: 60, 20, 10, 40, 30, 50, 70

Traversals of a Binary Tree

- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```

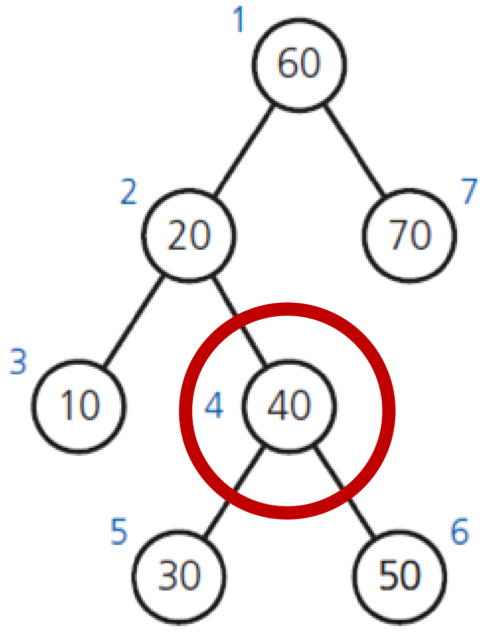


(a) Preorder: 60, 20, 10, 40, 30, 50, 70

Traversals of a Binary Tree

- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```

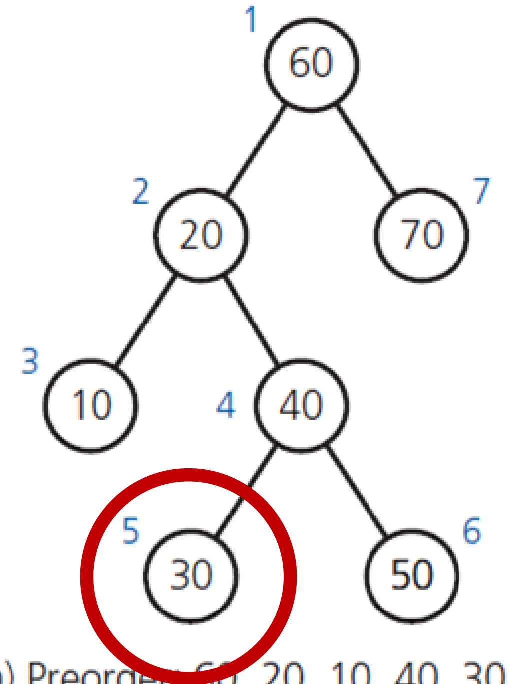


(a) Preorder: 60, 20, 10, 40, 30, 50, 70

Traversals of a Binary Tree

- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```

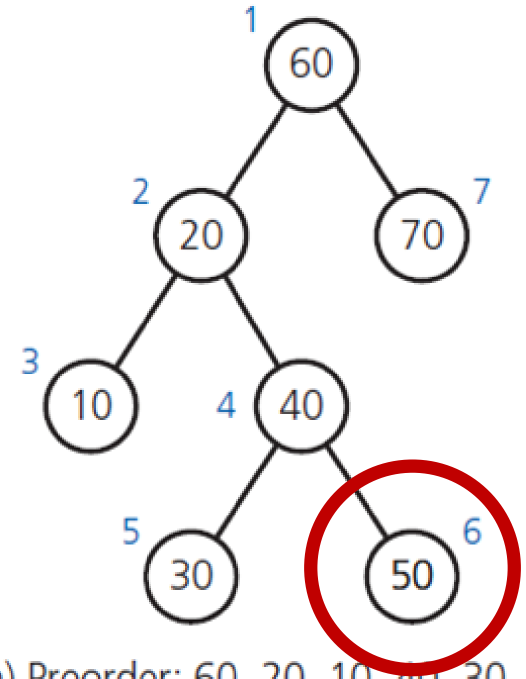


(a) Preorder: 60, 20, 10, 40, 30, 50, 70

Traversals of a Binary Tree

- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```

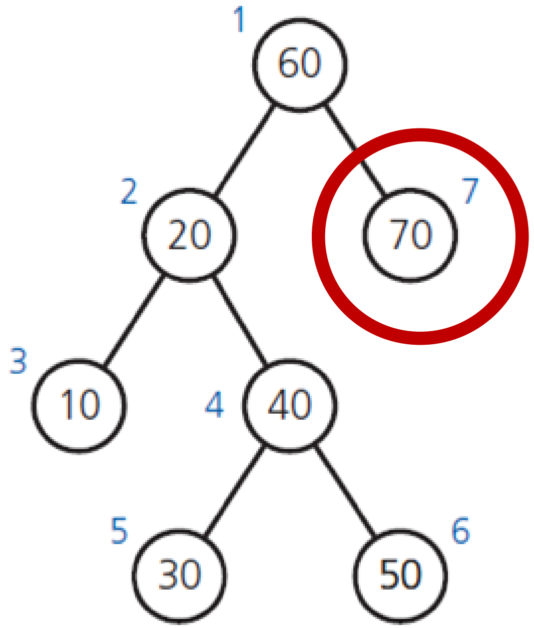


(a) Preorder: 60, 20, 10, 40, 30, 50, 70

Traversals of a Binary Tree

- Preorder traversal algorithm

```
// Traverses the given binary tree in preorder
// Assumes that "visit a node" means to process the node's data item
preorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        Visit the root of binTree
        preorder(Left subtree of binTree's root)
        preorder(Right subtree of binTree's root)
    }
}
```



(a) Preorder: 60, 20, 10, 40, 30, 50, 70

Traversals of a Binary Tree

- Inorder traversal algorithm
  - Traverse(Left subtree of Tree's root)
  - Process (visit) root
  - Traverse(Right subtree of Tree's root)

Traversals of a Binary Tree

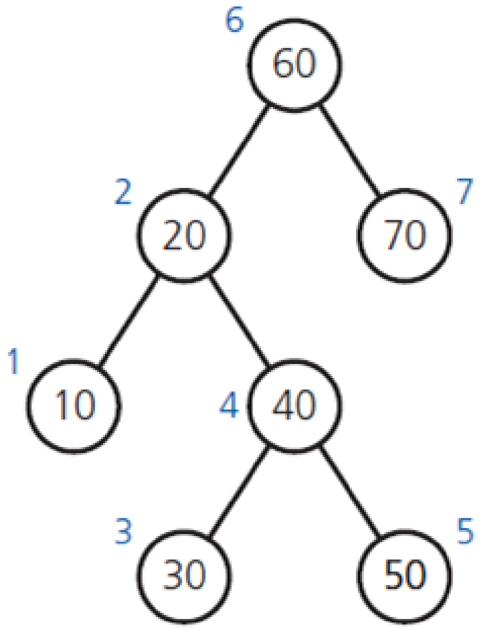
- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

Traversals of a Binary Tree

- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

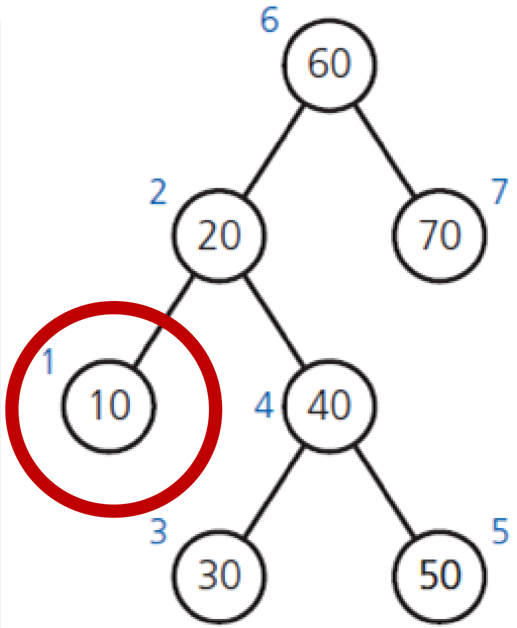


(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Traversals of a Binary Tree

- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

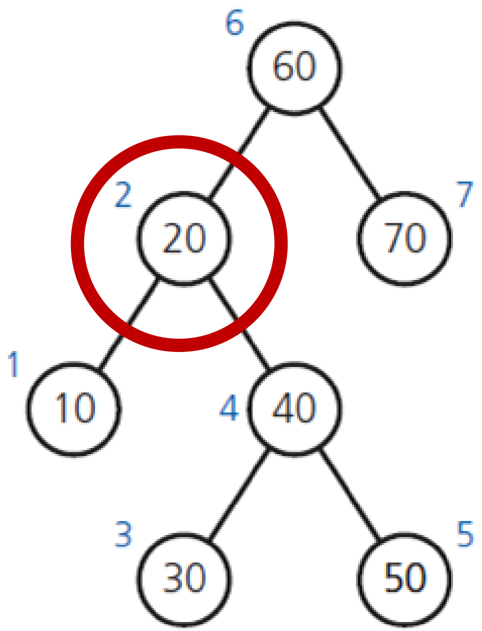


(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Traversals of a Binary Tree

- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

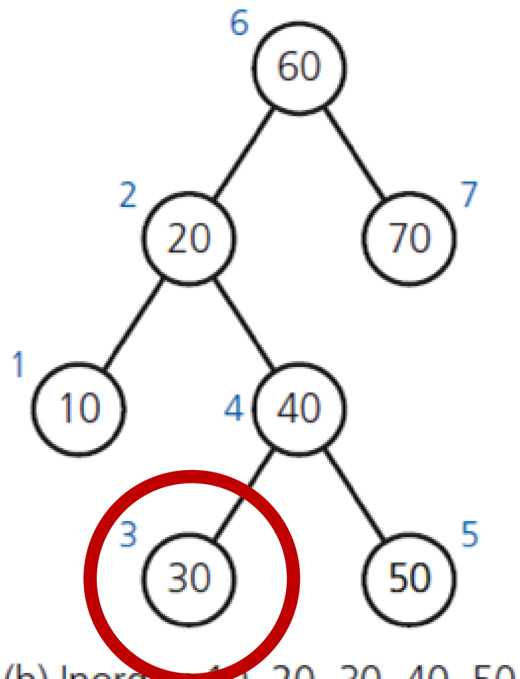


(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Traversals of a Binary Tree

- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

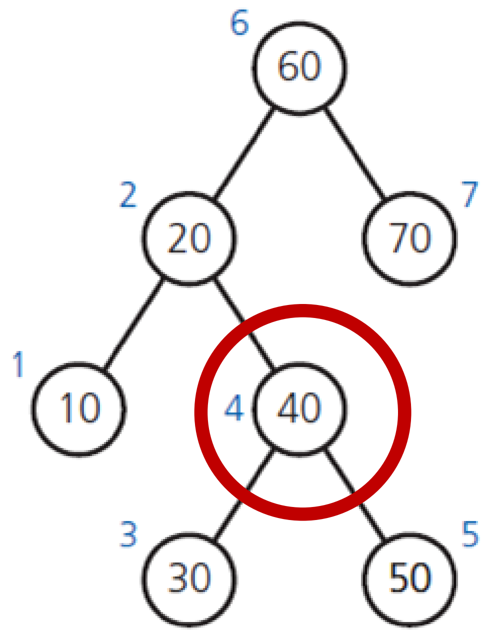


(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Traversals of a Binary Tree

- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

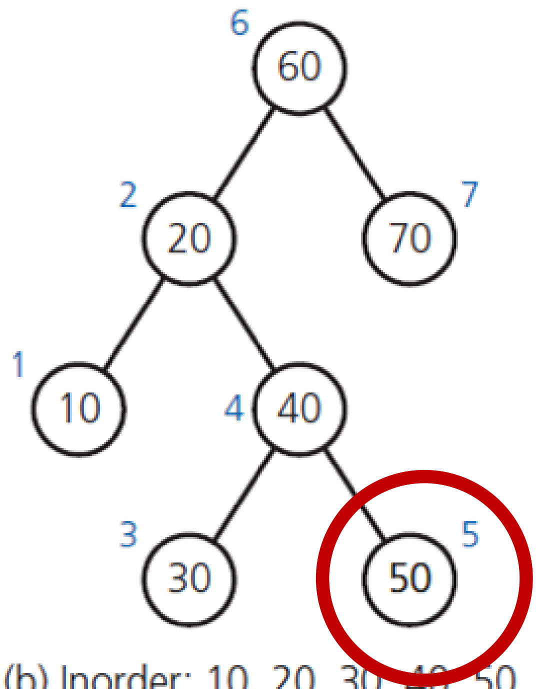


(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Traversals of a Binary Tree

- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

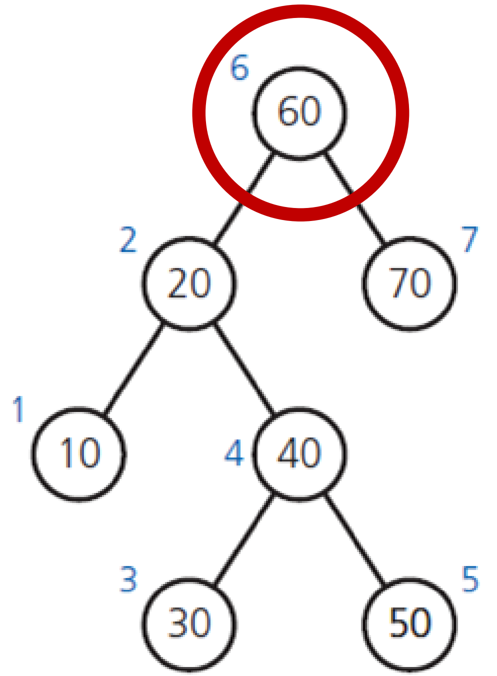


(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Traversals of a Binary Tree

- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

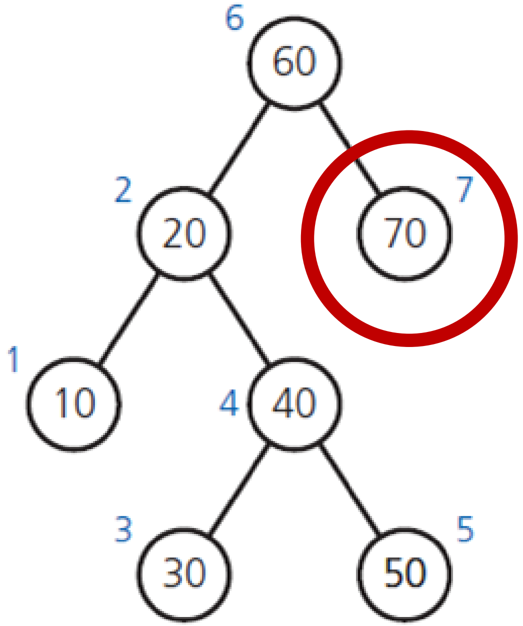


(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Traversals of a Binary Tree

- Inorder traversal algorithm

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```



(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Traversals of a Binary Tree

- Postorder traversal algorithm
  - Traverse(Left subtree of Tree's root)
  - Traverse(Right subtree of Tree's root)
  - Process (visit) root

Traversals of a Binary Tree

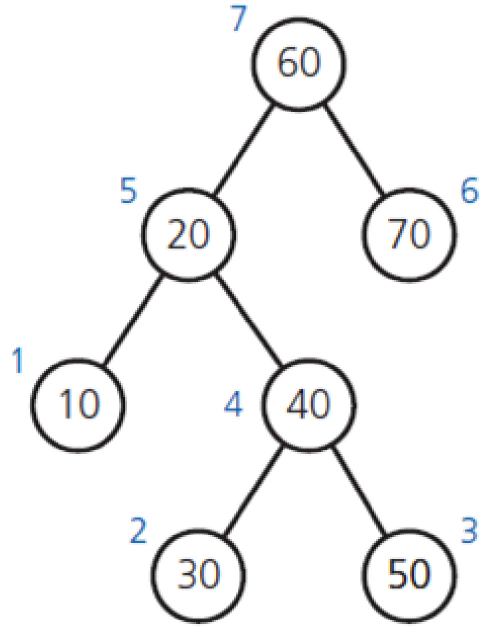
- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

Traversals of a Binary Tree

- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

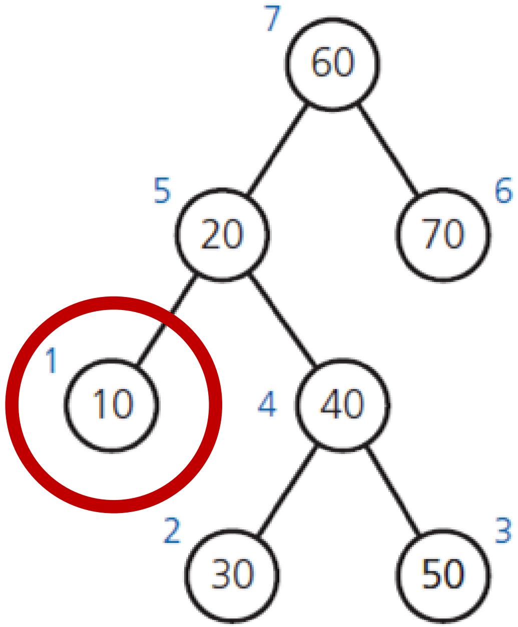


(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Traversals of a Binary Tree

- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

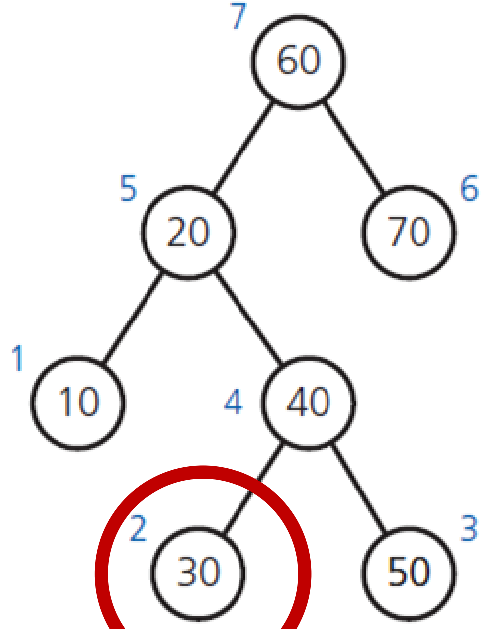


(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Traversals of a Binary Tree

- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

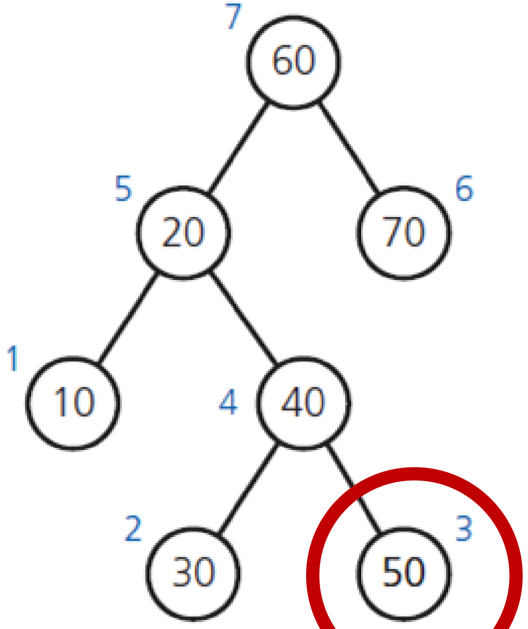


(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Traversals of a Binary Tree

- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

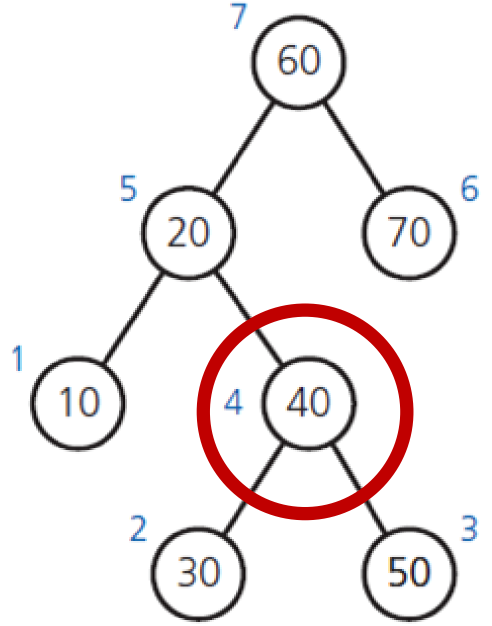


(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Traversals of a Binary Tree

- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

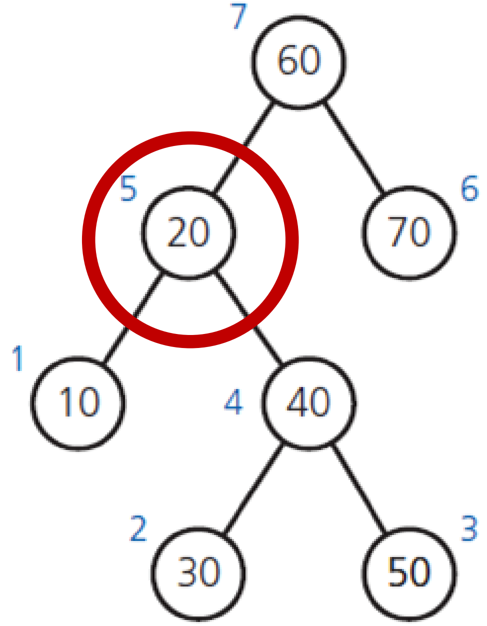


(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Traversals of a Binary Tree

- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

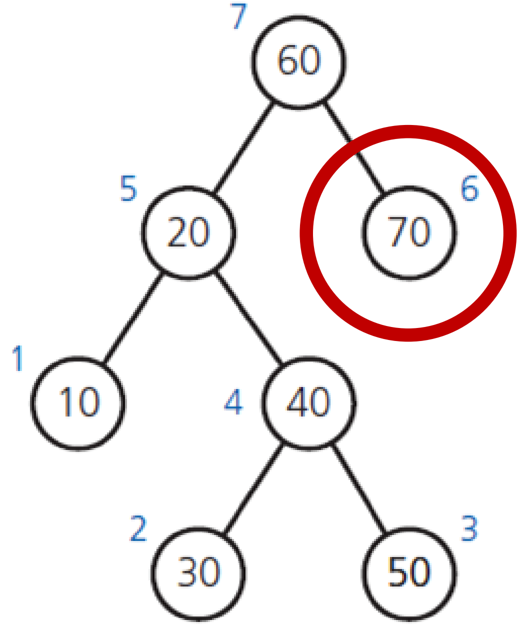


(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Traversals of a Binary Tree

- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```

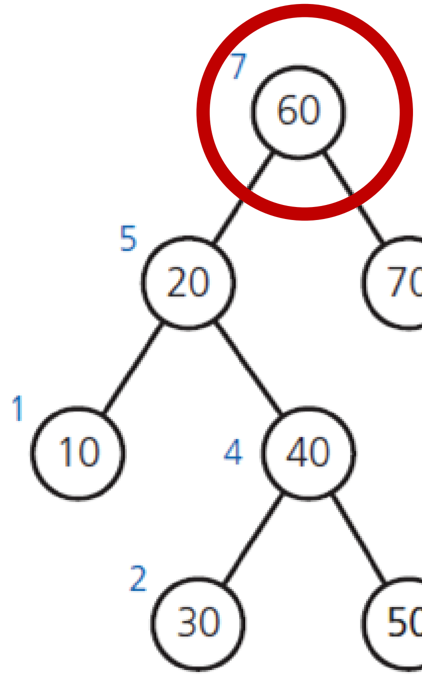


(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Traversals of a Binary Tree

- Postorder traversal algorithm

```
// Traverses the given binary tree in postorder
// Assumes that "visit a node" means to process the node's data item
postorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        postorder(Left subtree of binTree's root)
        postorder(Right subtree of binTree's root)
        Visit the root of binTree
    }
}
```



(c) Postorder: 10, 30, 50, 40, 20, 70, 60

Binary Tree Operations

- UML diagram for the class BinaryTree

UML Notation
BinaryTree

`+isEmpty(): boolean`
`+getHeight(): integer`
`+getNumberOfNodes(): integer`
`+getRootData(): ItemType`
`+setRootData(newData: ItemType): void`
`+add(newData: ItemType): boolean`
`+remove(target: ItemType): boolean`
`+clear(): void`
`+getEntry(target: ItemType): ItemType`
`+contains(target: ItemType): boolean`
`+preorderTraverse(visit(item: ItemType): void): void`
`+inorderTraverse(visit(item: ItemType): void): void`
`+postorderTraverse(visit(item: ItemType): void): void`

Interface for the ADT Binary Tree

- An interface template for the ADT Binary Tree

```
#ifndef BINARY_TREE_INTERFACE_
#define BINARY_TREE_INTERFACE_
#include "NotFoundException.h"

template<class ItemType>
class BinaryTreeInterface
{
public:
     // Tests whether this binary tree is empty
     // @return True if the binary tree is empty, or false if not
     virtual bool isEmpty() const = 0;

     // Gets the height of this binary tree
     // @return The height of the binary tree
     virtual int getHeight() const = 0;

     // Gets the number of nodes in this binary tree
     // @return The number of nodes in the binary tree
     virtual int getNumberOfNodes() const = 0;

     // Gets the data that is in the root of the binary tree
     // @pre: The binary tree is not empty
     // @post: The root's data has been returned, binary tree is unchanged
     // @return: The data n the root of the binary tree
     virtual ItemType getRootData() const = 0
```

Interface for the ADT Binary Tree

- An interface template for the ADT Binary Tree

```
// Replaces the data in the root of this binary tree with the given data,
// if the tree is not empty. However, if the tree is empty, inserts a new
// root node containing the given data into the tree
// @pre None
// @post The data in the root of the binary tree is as given
// @param newData The data for the root
virtual bool setRootData(const ItemType& newData) = 0;

// Adds the given data to this binary tree
// @param newData The data to add to the binary tree
// @post The binary tree contains the new data
// @return True if the addition is successful, or false if not
virtual bool add(const ItemType& newData) = 0;

// Removes the specified data from this binary tree
// @param target The data to remove from the binary tree
// @return True if the remove is successful, or false if not
virtual bool remove(const ItemType& target) = 0;

// Removes all data from this binary tree
virtual void clear() = 0;

// Retrieves the specified data from this binary tree
// @post The desired data has been returned, and the binary tree is unchanged.
//       If no such data was found, exception is thrown
// @param target The data to locate
// @return The data in the binary tree that matches the given data
virtual ItemType getEntry(const ItemType& target) const = 0;
```

Interface for the ADT Binary Tree

- An interface template for the ADT Binary Tree

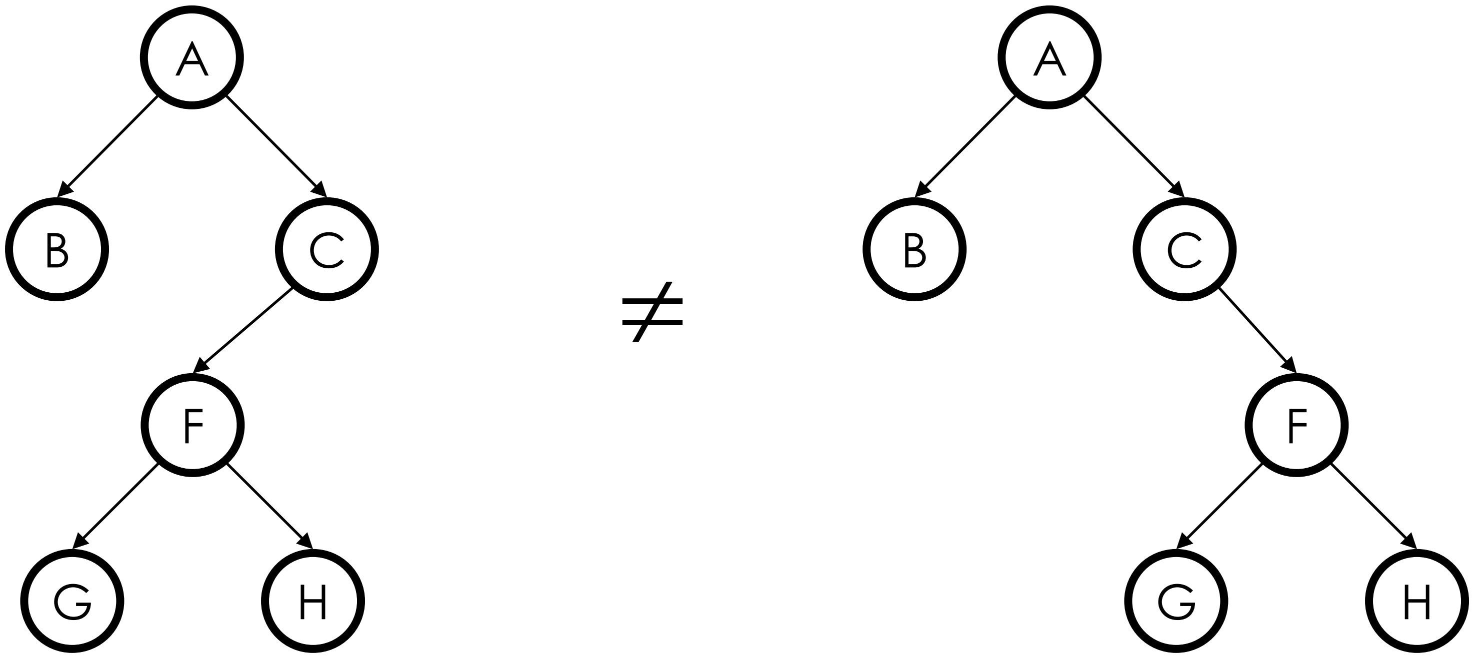
```
      // Tests whether the specified data occurs in this binary tree
      // @post The binary tree is unchanged
      // @param target The data to find
      // @return True if data matching the target occurs in the tree, or false otherwise
      virtual bool contains(const ItemType& target) const = 0;

      // Traverses this binary tree in preorder (inorder, postorder) and
      // calls the function visit once for each node
      // @param visit A client-defined function that performs an operation on either each visited node or its data
      virtual void preorderTraverse(void visit(ItemType&)) const = 0;
      virtual void inorderTraverse(void visit(ItemType&)) const = 0;
      virtual void postorderTraverse(void visit(ItemType&)) const = 0;

      // Destroys this tree and frees its assigned memory
      virtual ~BinaryTreeInterface() { }
}; // end BinaryTreeInterface
#endif;
```

The ADT Binary Search Tree

- Recursive definition of a binary search tree
  - The value of **n** is greater than all values in its left subtree $\mathbf{T_L}$
  - The value of **n** is less than all values in its right subtree $\mathbf{T_R}$
  - Both $\mathbf{T_L}$ and $\mathbf{T_R}$ are binary search trees

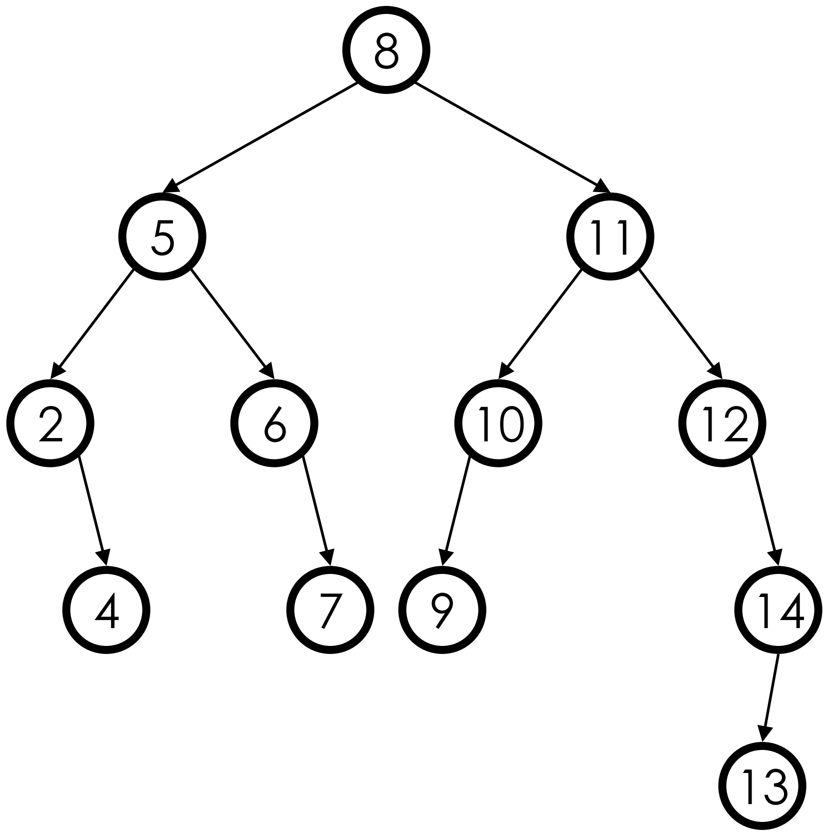
The ADT Binary Search Tree

- Notice: we distinguish between left and right child



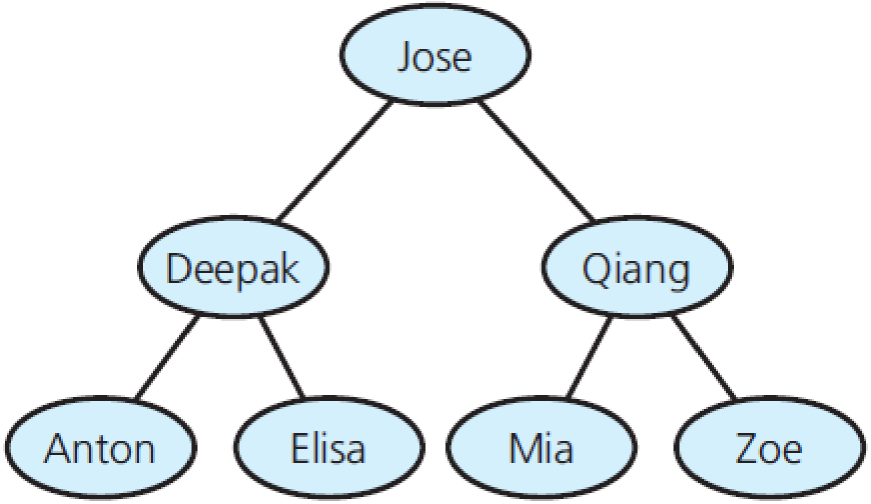
The ADT Binary Search Tree

- Search tree property
  - All keys in left subtree smaller than root’s key
  - All keys in right subtree larger than root’s key
  - Result:
    - Easy to find any given key
    - Inserts/deletes by changing links



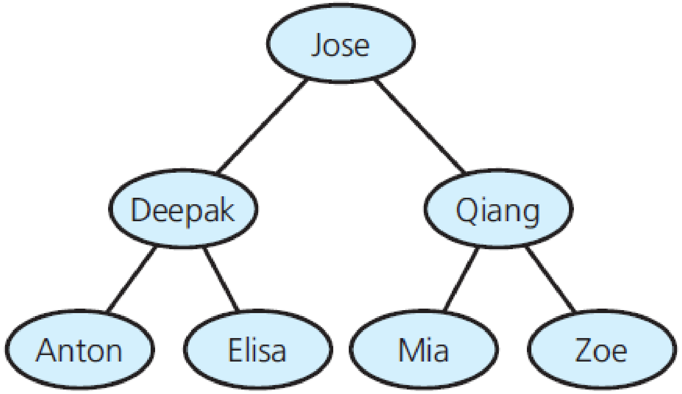
The ADT Binary Search Tree

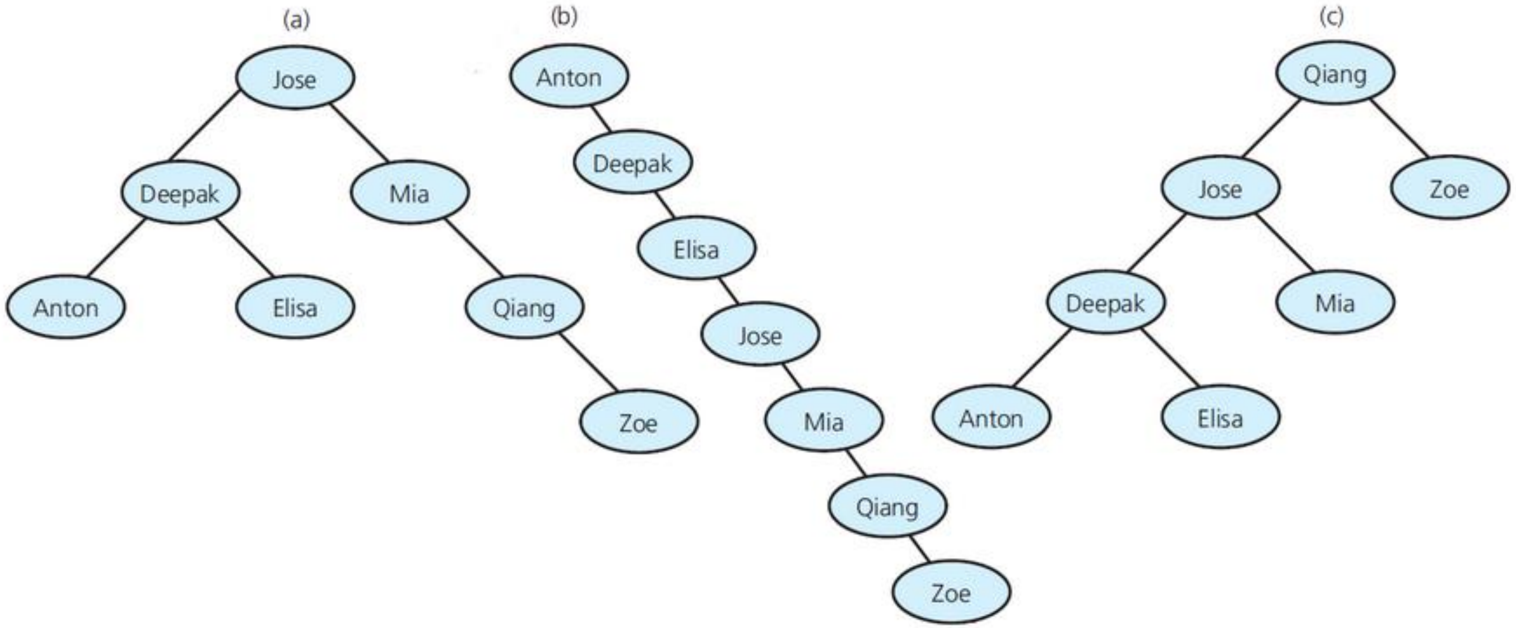
- A binary search tree of names



Binary Search Tree Operations

- Binary search trees with the same data





Binary Search Tree Operations

- Test whether a binary search tree is empty
- Get the height of a binary search tree
- Get the number of nodes in a binary search tree
- Get the data in a binary search tree's root
- Add the given data item to a binary search tree
- Remove the specified data item from a binary search tree
- Remove all data items from a binary search tree
- Retrieve the specified data item in a binary search tree
- Test whether a binary search tree contains specific data
- Traverse the nodes in a binary search tree in preorder, inorder, or postorder sense

These operations define the ADT binary search tree

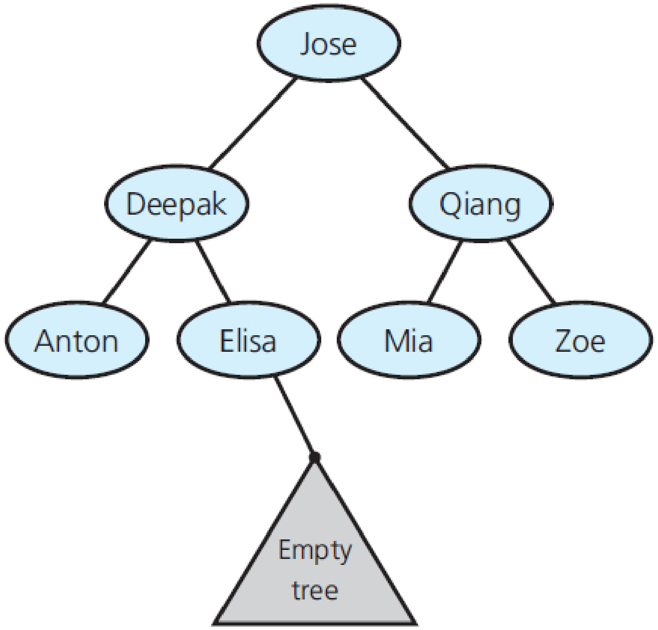
Searching a Binary Search Tree

- Search algorithm for a binary search tree

```
// Searches the binary search tree for a given target value
search(bstTree: BinarySearchTree, target: ItemType)
{
    if (bstTree is empty)
        The desired item is not found
    else if (target == data item in the root of bstTree)
        The desired item is found
    else if (target < data item in the root of bstTree)
        search(Left subtree of bstTree, target)
    else
        search(Right subtree of bstTree, target)
}
```

Creating a Binary Search Tree

- Empty subtree where the search algorithm terminates when looking for Finn



Traversals of a Binary Search Tree

- Inorder traversal of a binary search tree visits tree's nodes in sorted search-key order

```
// Traverses the given binary tree in inorder
// Assumes that "visit a node" means to process the node's data item
inorder(binTree: BinaryTree): void
{
    if (binTree is not empty)
    {
        inorder(Left subtree of binTree's root)
        Visit the root of binTree
        inorder(Right subtree of binTree's root)
    }
}
```

Efficiency of Binary Search Tree Operations

- Max number of comparisons for retrieval, addition, or removal
  - The height of the tree
- Adding entries in sorted order
  - Produces maximum-height binary search tree
- Adding entries in random order
  - Produces near-minimum-height binary search tree

Efficiency of Binary Search Tree Operations

- The Big O for the retrieval, addition, removal, and traversal operations of the ADT binary search tree (BST):

Operation	Average Case	Worst Case
Retrieval	O(logn)	O(n)
Addition	O(logn)	O(n)
Removal	O(logn)	O(n)
Traversal	O(n)	O(n)

Thank you