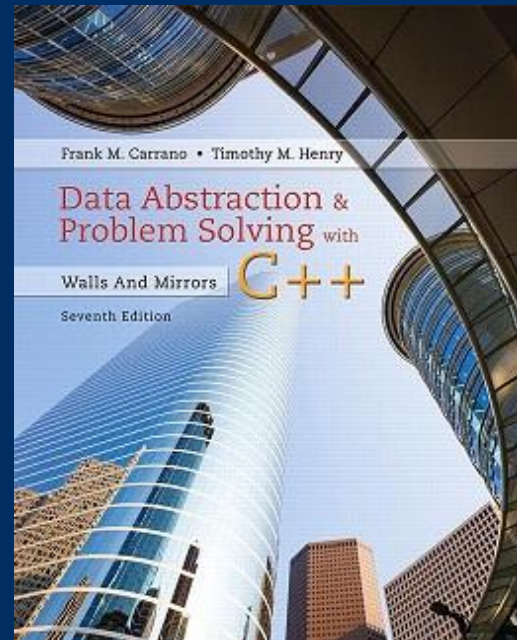


Chapter 15

Trees

CS 302 - Data Structures

M. Abdullah Canbaz



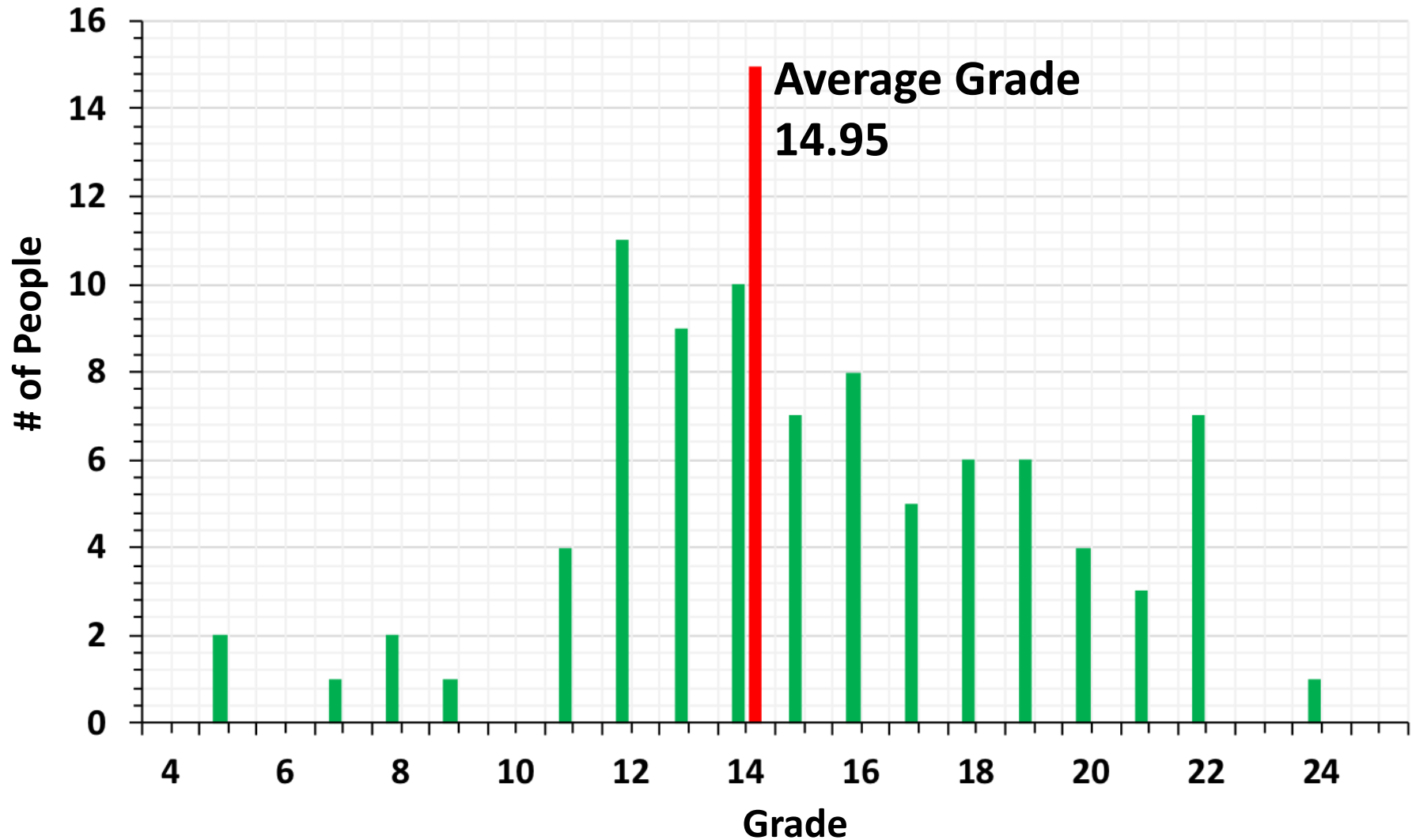


Reminders

- Assignment 4 is available
 - Due April 2nd at 2pm
- TA
 - Shehryar Khattak,
Email: *shehryar [at] nevada {dot} unr {dot} edu*,
Office Hours: Friday, 11:00 am - 1:00 pm at ARF 116



Midterm Review



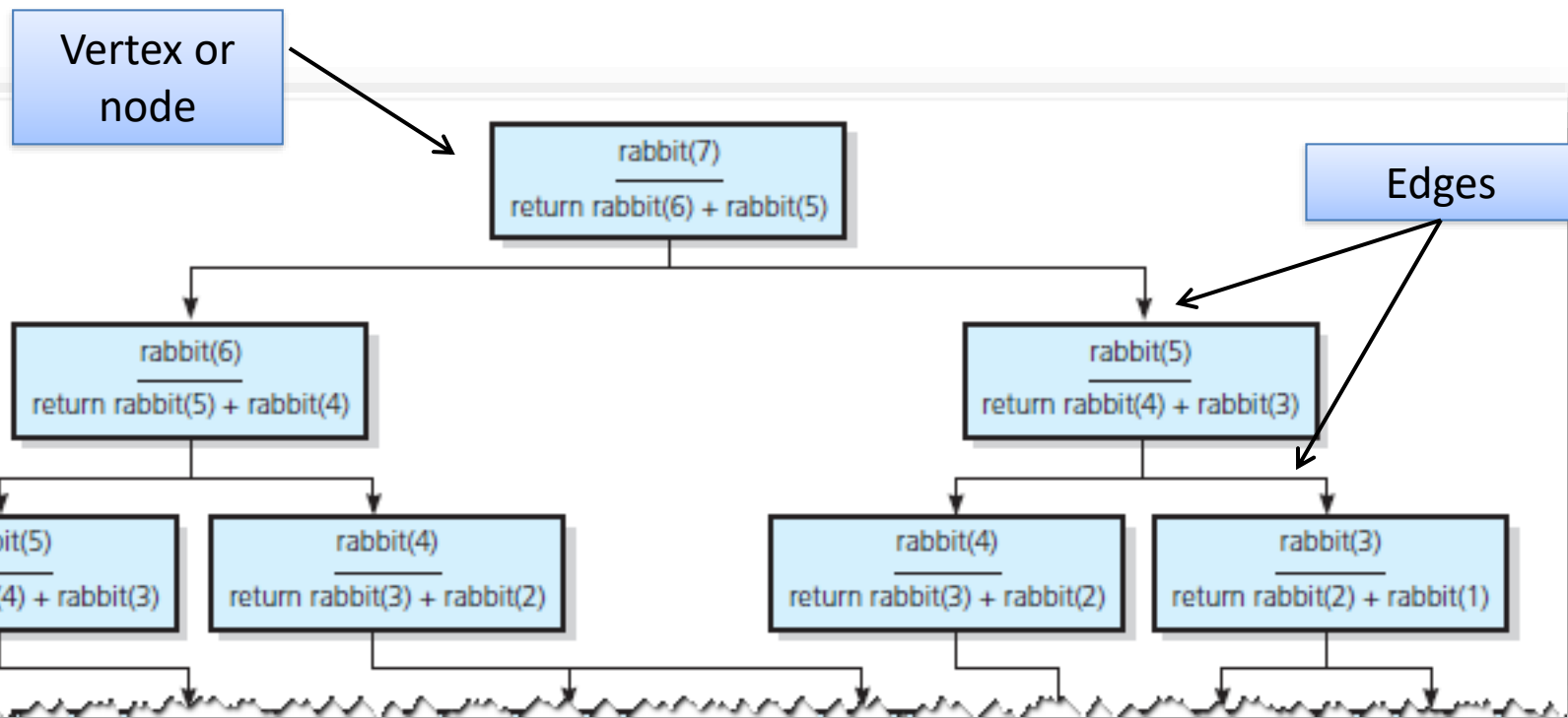




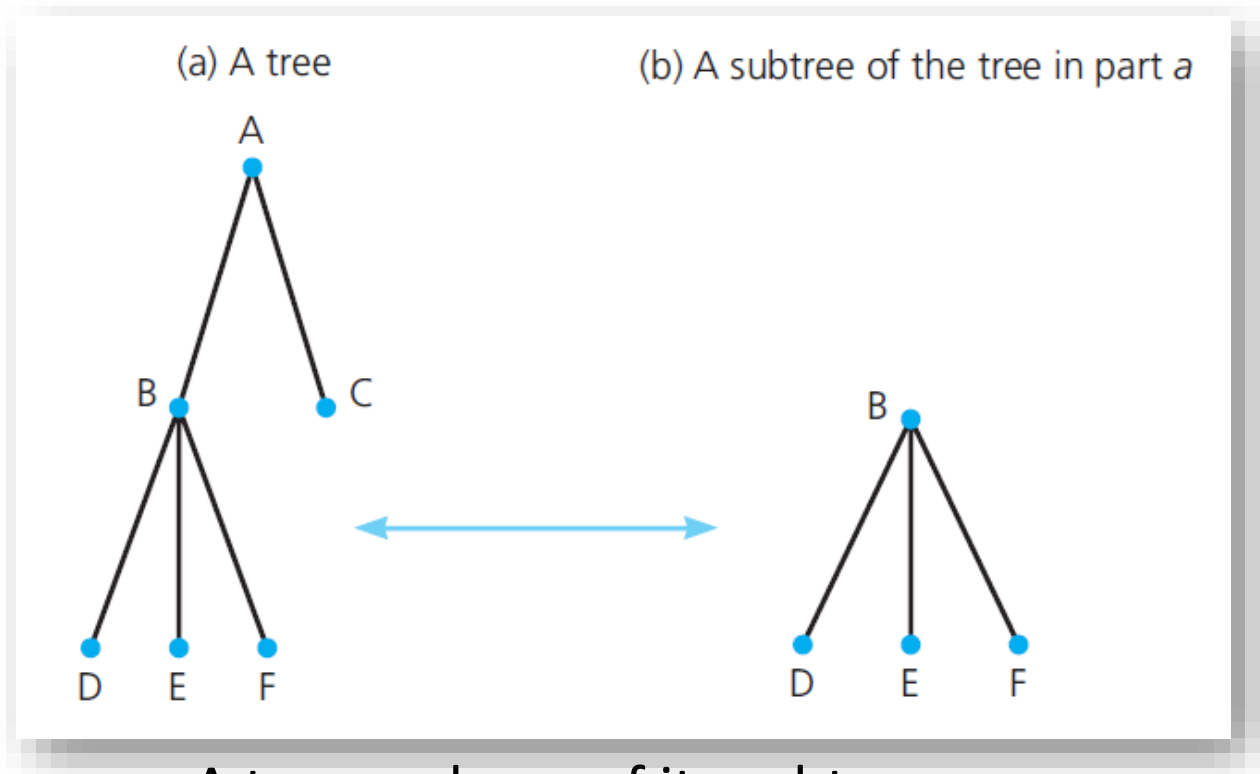
Trees

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

- Use trees to represent relationships

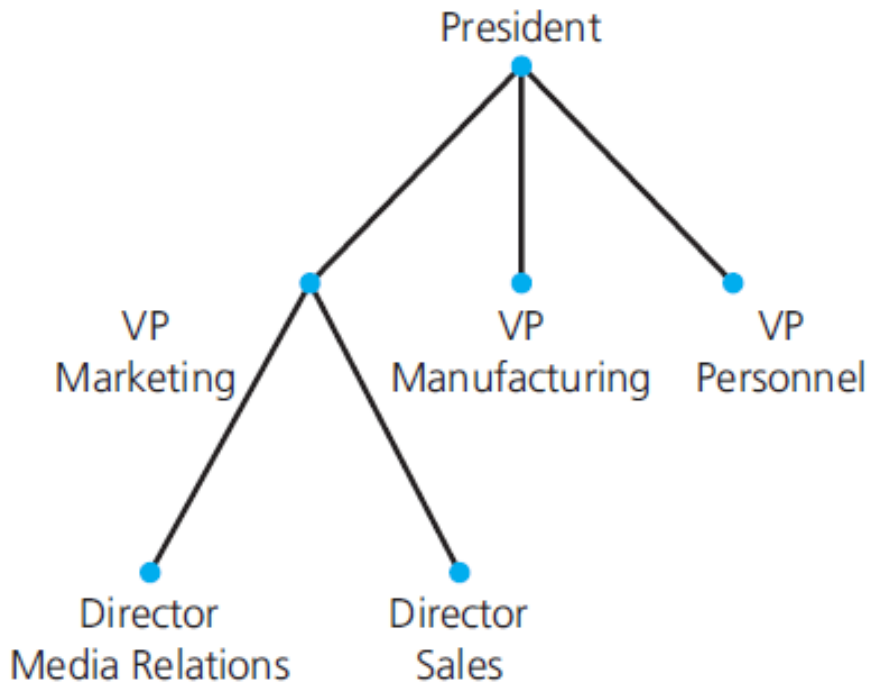


- Trees are hierarchical in nature
 - Means a parent-child relationship between nodes

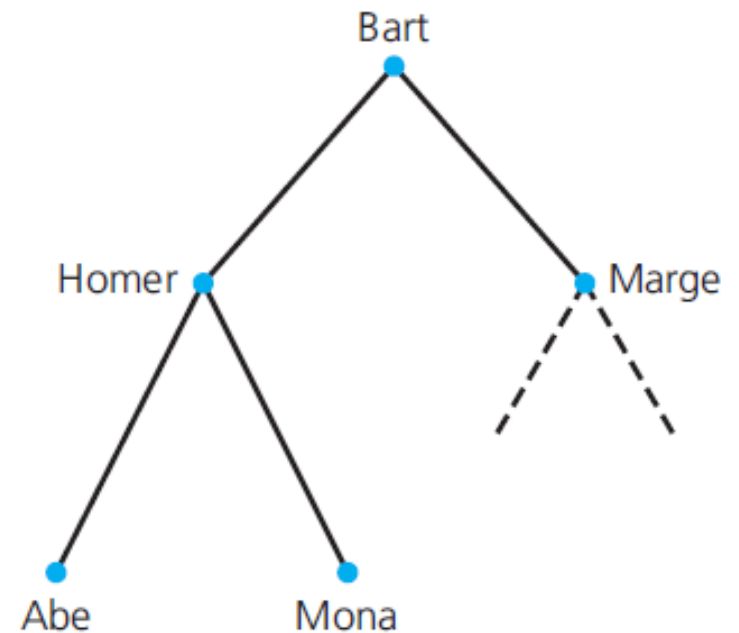


A tree and one of its subtrees

(a) An organization chart



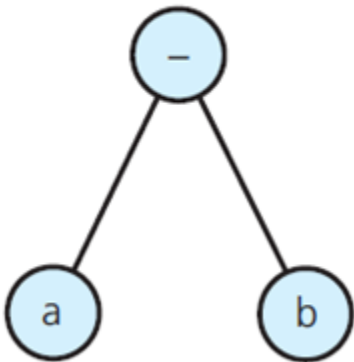
(b) A family tree



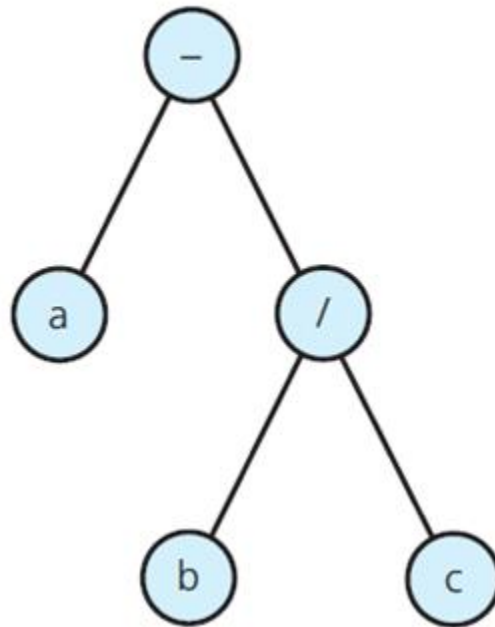
- General tree
 - Set T of one or more nodes
 - T is partitioned into disjoint subsets
- Binary tree
 - Set of T nodes – either empty or partitioned into disjoint subsets
 - Single node r , the root
 - Two (possibly empty) sets – left and right subtrees

- Binary trees that represent algebraic expressions

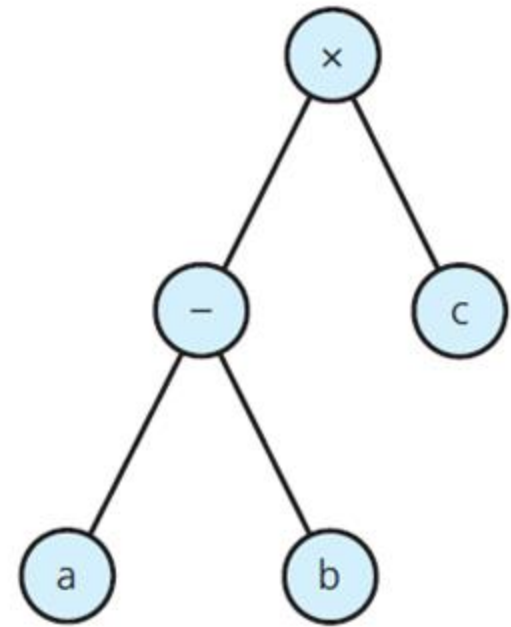
(a) $a - b$



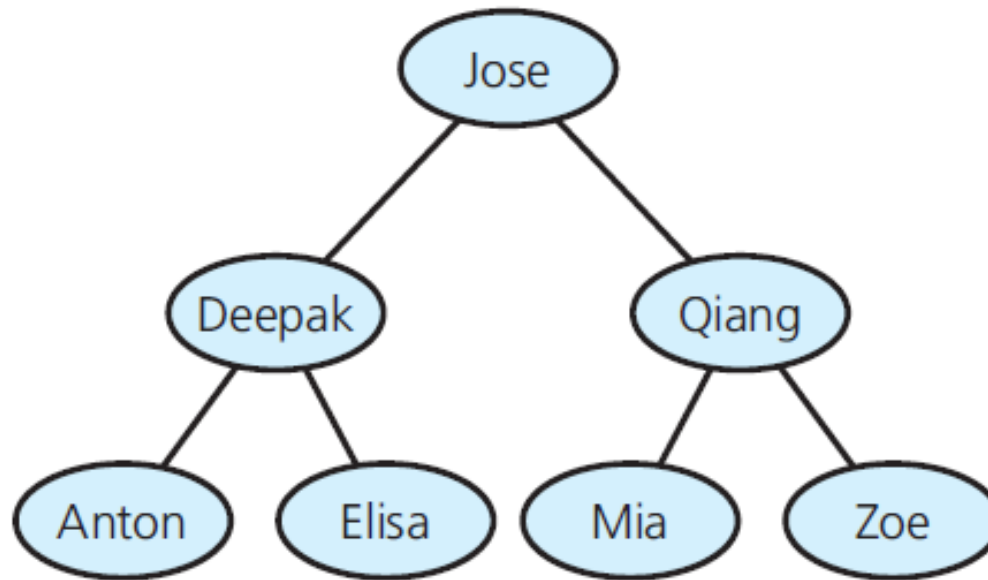
(b) $a - b / c$



(c) $(a - b) \times c$



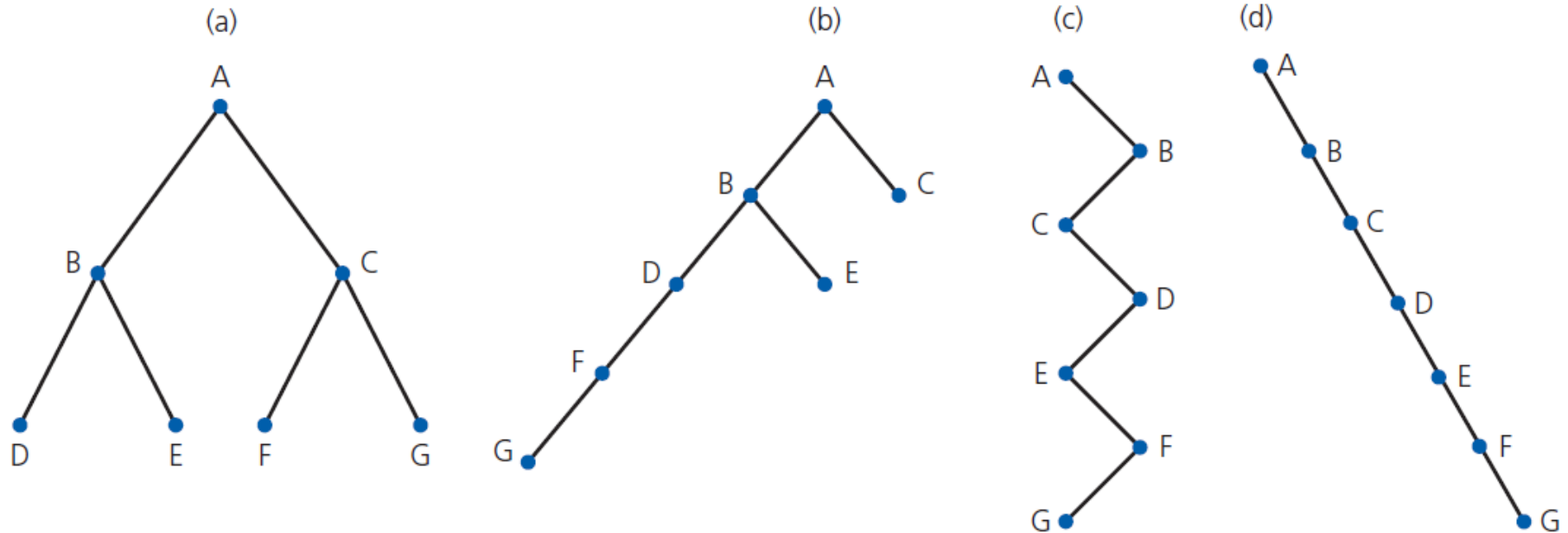
- A binary search tree of names





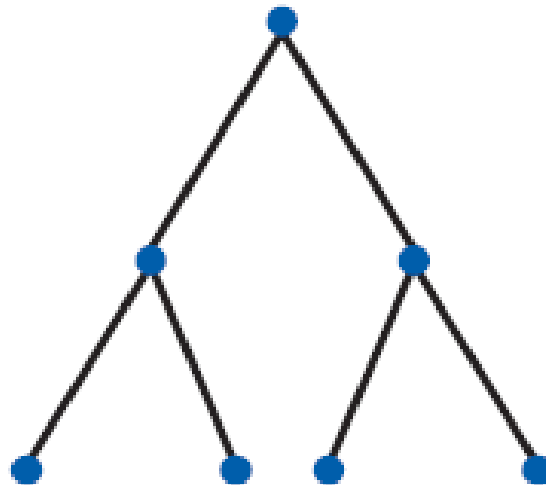
The Height of Trees

- 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

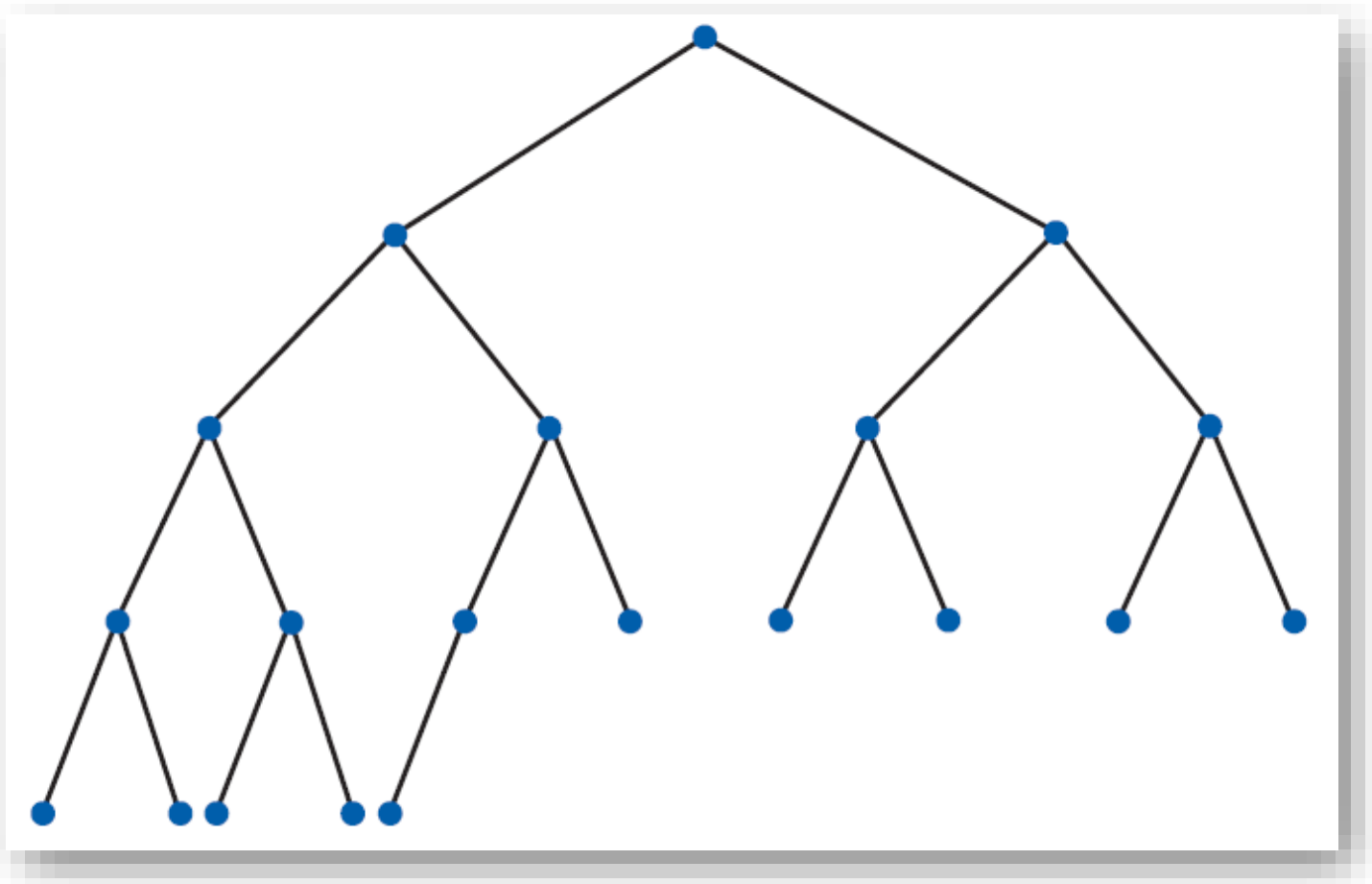


- Binary trees with the same nodes but different heights

- A full binary tree of height 3



- A complete binary tree

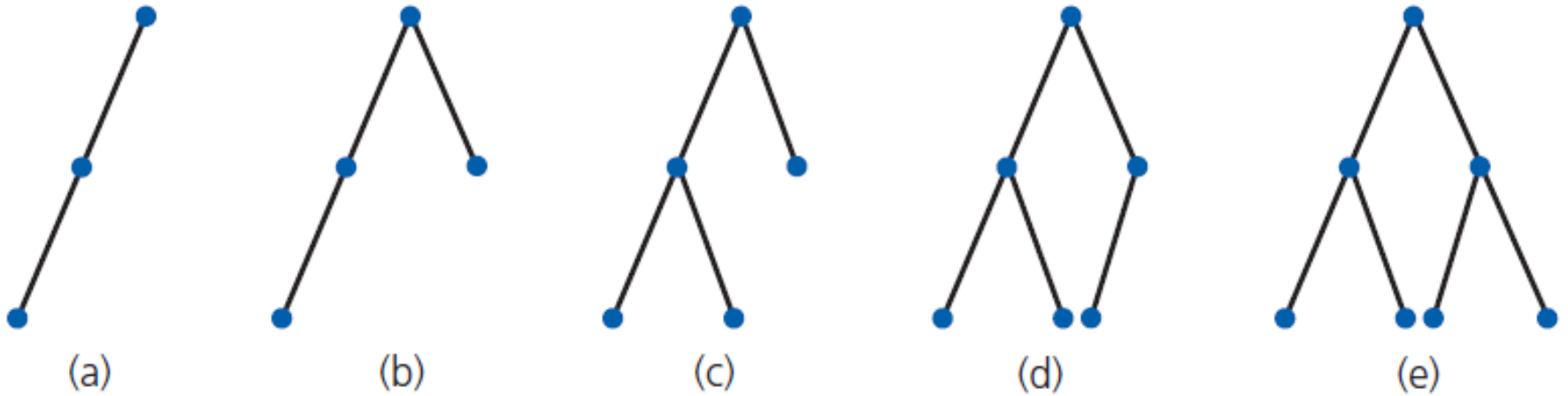




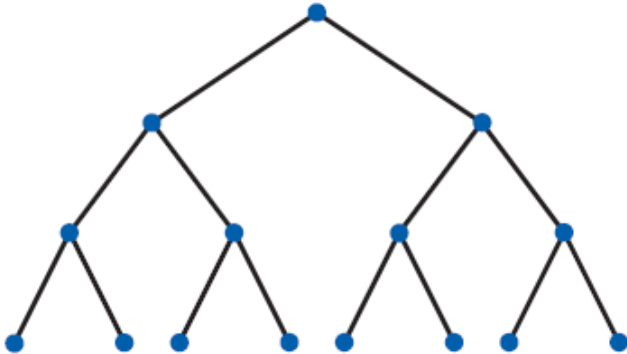
The Maximum and Minimum Heights of a Binary Tree

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

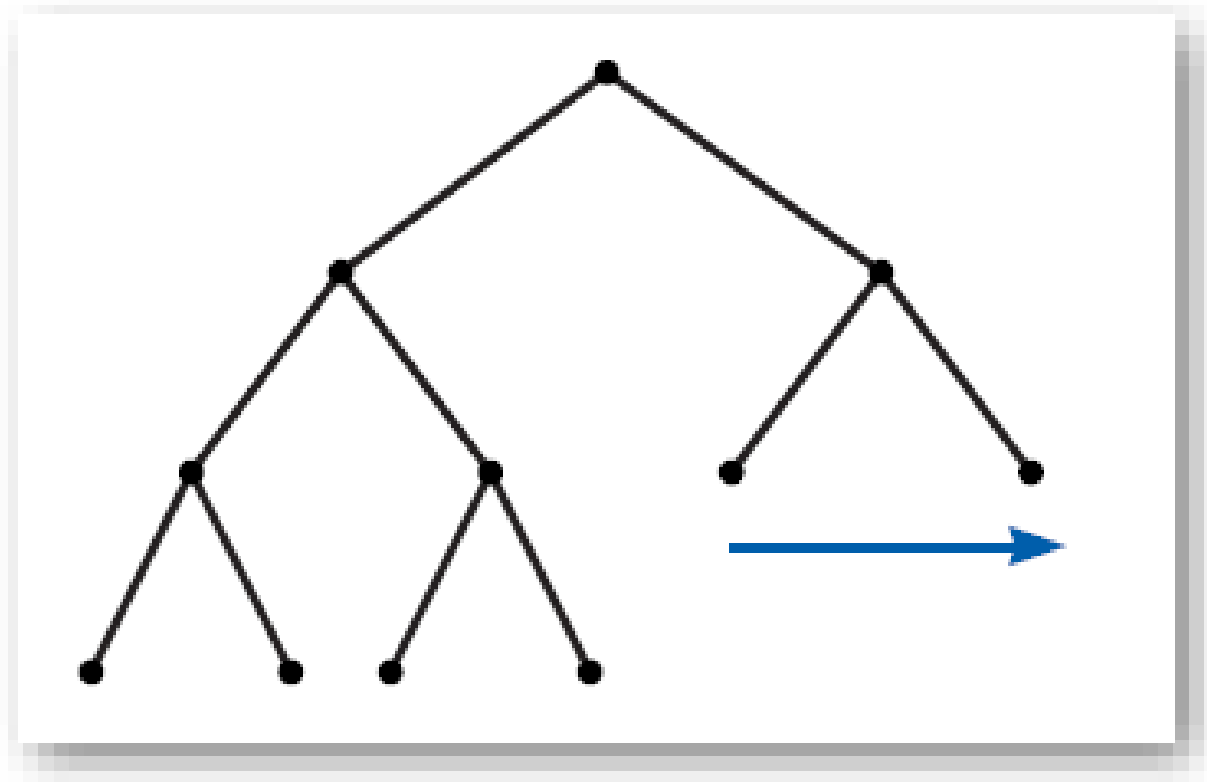
- Binary trees of height 3



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

height n	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}	$2^h - 1$

- 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

- Pseudocode for general form of a recursive traversal algorithm

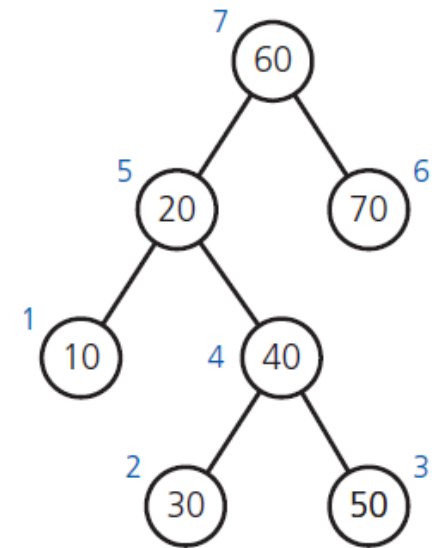
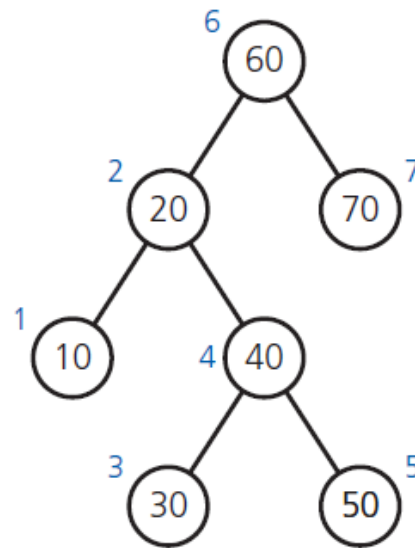
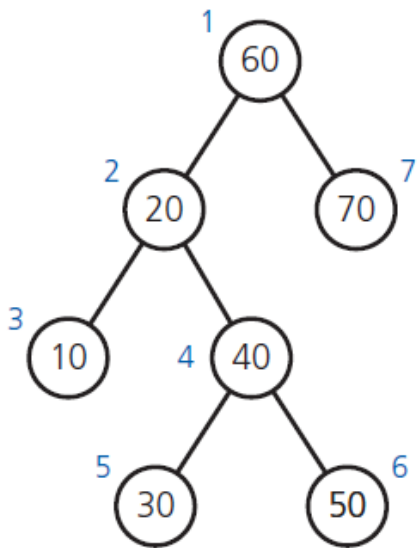
```
if (T is not empty)
{
    Display the data in T's root
    Traverse T's left subtree
    Traverse T's right subtree
}
```



Traversals of a Binary Tree

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

- 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.)

- 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)  
    }  
}
```


- 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)  
    }  
}
```

- 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  
    }  
}
```

- UML diagram for the class **BinaryTree**

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
```

```
1  /** Interface for the ADT binary tree.
2   @file BinaryTreeInterface.h */
3
4  #ifndef BINARY_TREE_INTERFACE_
5  #define BINARY_TREE_INTERFACE_
6  #include "NotFoundException.h"
7
8  template<class ItemType>
9  class BinaryTreeInterface
10 {
11 public:
12     /** Tests whether this binary tree is empty.
13     @return True if the binary tree is empty, or false if not. */
14     virtual bool isEmpty() const = 0;
15
16     /** Gets the height of this binary tree.
17     @return The height of the binary tree. */
18     virtual int getHeight() const = 0;
19
20     /** Gets the number of nodes in this binary tree.
21     @return The number of nodes in the binary tree. */
22     virtual int getNumberOfNodes() const = 0;
```

- An interface template for the ADT binary tree

```
23
24  /** Gets the data that is in the root of this binary tree.
25     @pre  The binary tree is not empty.
26     @post The root's data has been returned, and the binary tree is unchanged.
27     @return The data in the root of the binary tree. */
28  virtual ItemType getRootData() const = 0;
29
30  /** Replaces the data in the root of this binary tree with the given data,
31     if the tree is not empty. However, if the tree is empty, inserts a new
32     root node containing the given data into the tree.
33     @pre  None.
34     @post The data in the root of the binary tree is as given.
35     @param newData  The data for the root. */
36  virtual void setRootData(const ItemType& newData) = 0;
37
38  /** Adds the given data to this binary tree.
39     @param newData  The data to add to the binary tree.
40     @post  The binary tree contains the new data.
41     @return True if the addition is successful, or false if not. */
42  virtual bool add(const ItemType& newData) = 0;
```

- An interface template for the ADT binary tree

```
41
42  /** Removes the specified data from this binary tree.
43   @param target  The data to remove from the binary tree.
44   @return  True if the removal is successful, or false if not. */
45  virtual bool remove(const ItemType& target) = 0;
46
47  /** Removes all data from this binary tree. */
48  virtual void clear() = 0;
49
50  /** Retrieves the specified data from this binary tree.
51   @post  The desired data has been returned, and the binary tree
52   is unchanged. If no such data was found, an exception is thrown.
53   @param target  The data to locate.
54   @return  The data in the binary tree that matches the given data.*/
55  virtual ItemType getEntry(const ItemType& target) const = 0;
```

- An interface template for the ADT binary tree

- An interface template for the ADT binary tree

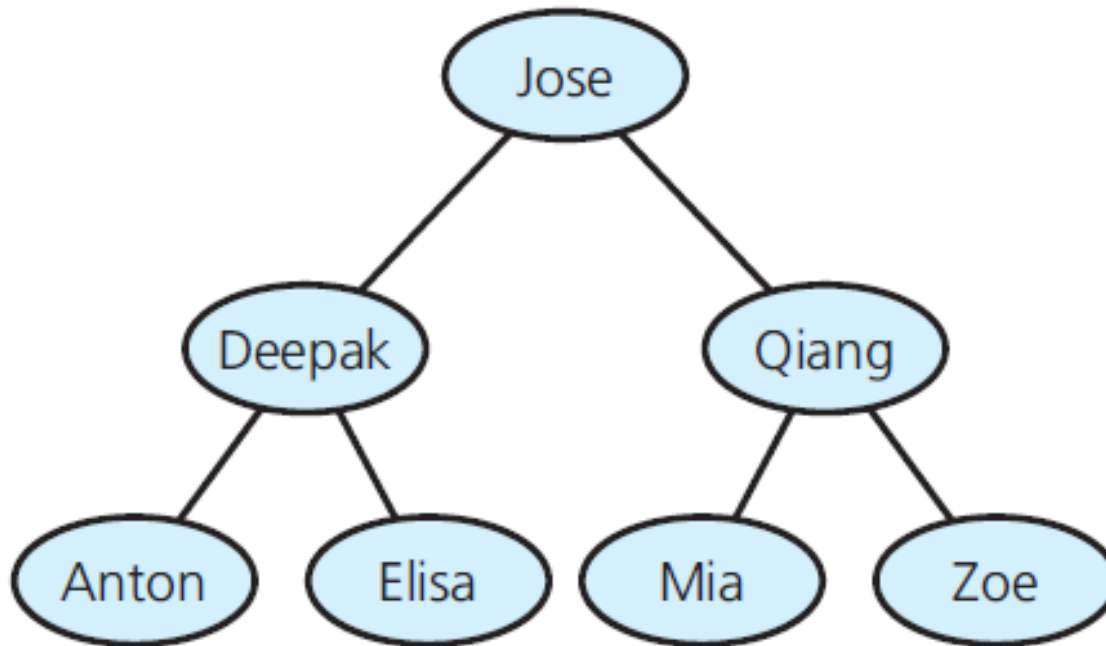
```
57     /** Tests whether the specified data occurs in this binary tree.
58         @post  The binary tree is unchanged.
59         @param target  The data to find.
60         @return  True if data matching the target occurs in the tree, or false if not. */
61     virtual bool contains(const ItemType& target) const = 0;
62
63     /** Traverses this binary tree in preorder (inorder, postorder) and
64         calls the function visit once for each node.
65         @param visit  A client-defined function that performs an operation on
66             either each visited node or its data. */
67     virtual void preorderTraverse(void visit(ItemType&)) const = 0;
68     virtual void inorderTraverse(void visit(ItemType&)) const = 0;
69     virtual void postorderTraverse(void visit(ItemType&)) const = 0;
70
71     /** Destroys this tree and frees its assigned memory. */
72     virtual ~BinaryTreeInterface() { }
73 }; // end BinaryTreeInterface
74 #endif
```

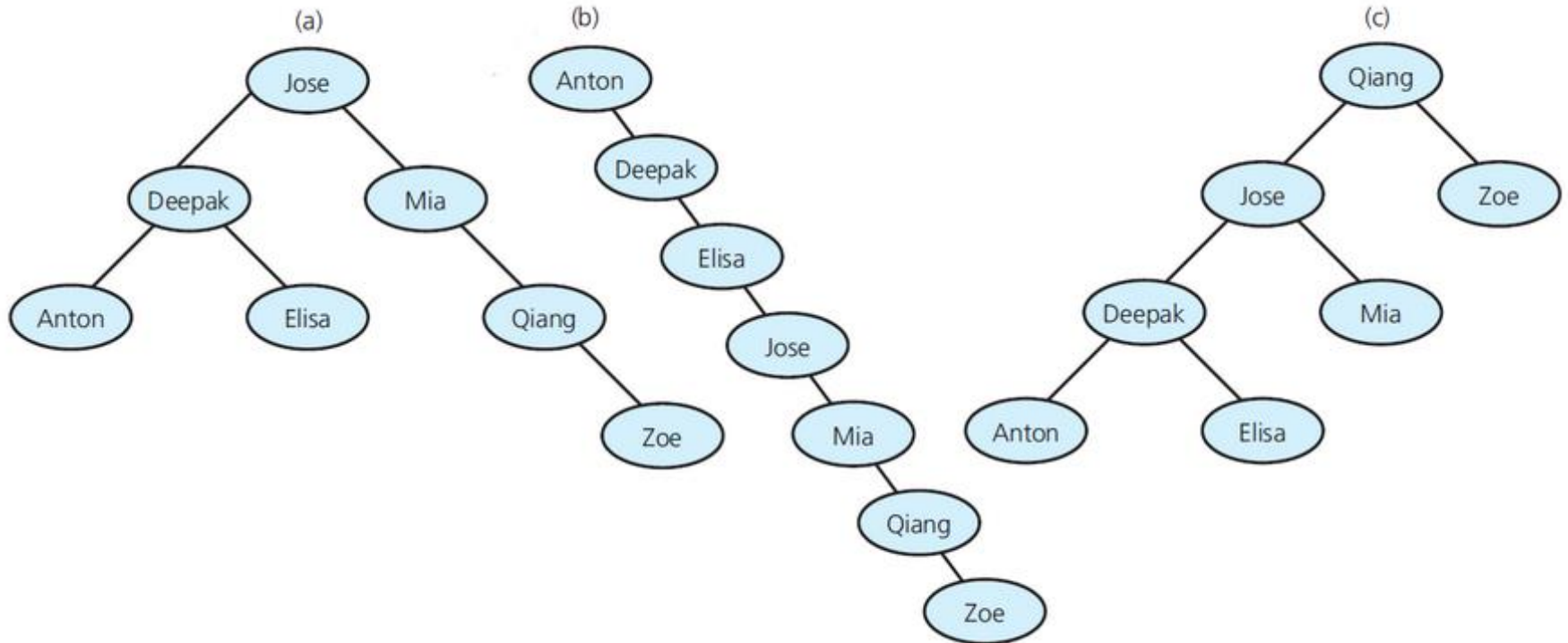


The ADT Binary Search Tree

- Recursive definition of a binary search tree
 - n 's value is greater than all values in its left subtree T_L
 - n 's value is less than all values in its right subtree T_R
 - Both T_L and T_R are binary search trees

- A binary search tree of names





- Binary search trees with the same data

- 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.

- Operations that define the ADT 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)  
}
```

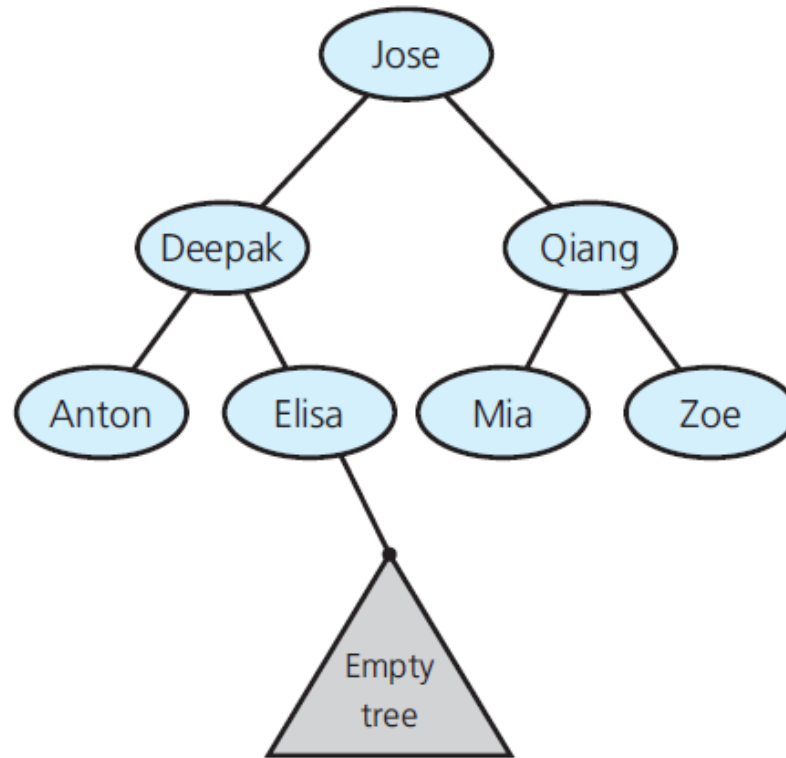
N

Searching a Binary Search Tree

- An array of names in sorted order

Anton	Deepak	Elisa	Jose	Mia	Qiang	Zoe
0	1	2	3	4	5	6

Creating a Binary Search Tree



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

```
// 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)  
    }  
}
```

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



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

Operation	Best case	Average case	Worst Case
Retrieval	$O(1)$	$O(\log n)$	$O(n)$
Addition	$O(1)$	$O(\log n)$	$O(n)$
Removal	$O(1)$	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$	$O(n)$

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

Cracking the Coding Interview Data Structures: Trees

<https://www.youtube.com/watch?v=oSWTXtMgIKE>

