

EECE 2560: Fundamentals of Engineering Algorithms

Trees



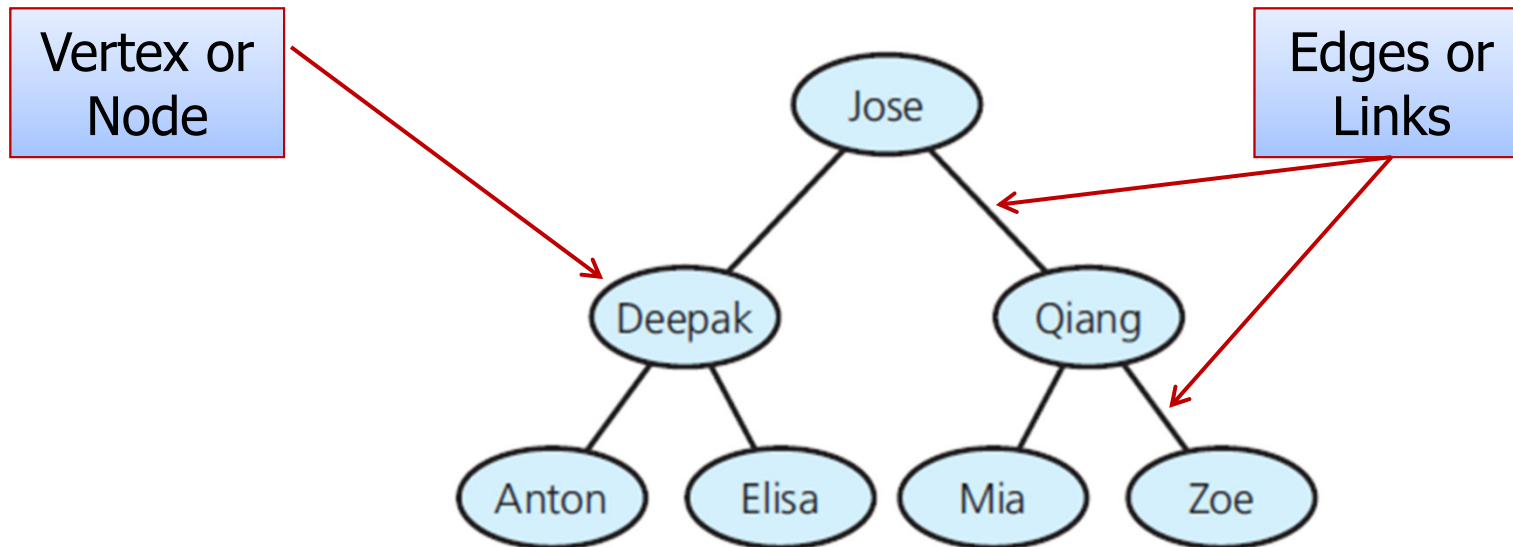
Trees

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



Terminology (1 of 2)

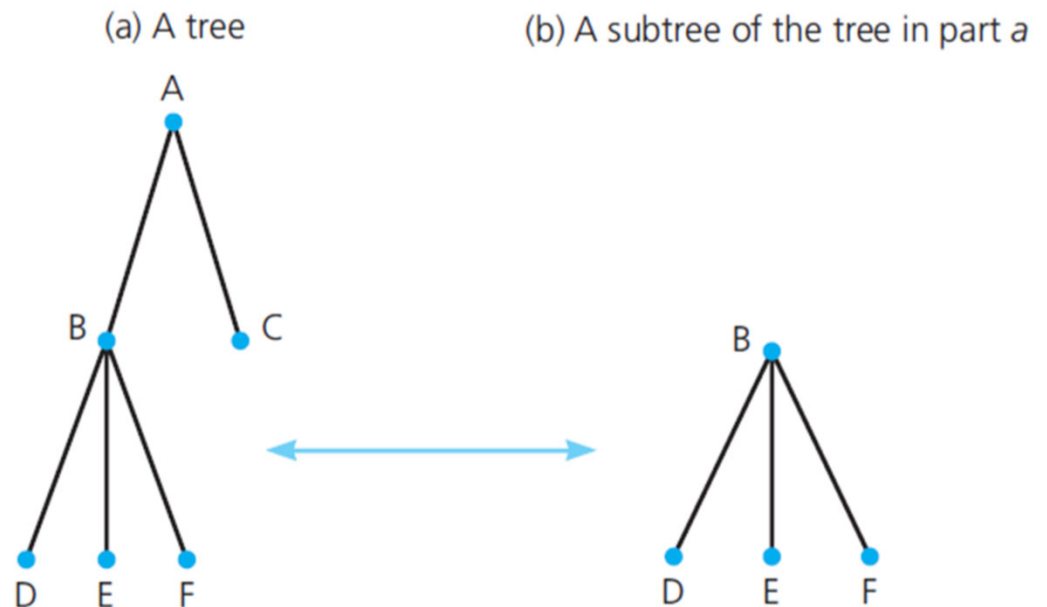
- Use trees to represent relationships





Terminology (2 of 2)

- Trees are hierarchical in nature. Means a parent-child relationship between nodes.
 - Node **A** is the **root** of the tree (it has no parent).
 - Nodes **B** and **C** are **children** of node **A**, where **A** is their **parent**.
 - **B** and **C** are called **siblings**.
 - The leftmost child **D** is called the **oldest child**, or **first child**, of **B**.
 - **C**, **D**, **E**, and **F** are **leaves** of the tree. They have no children.
 - An example of a **subtree** is shown.

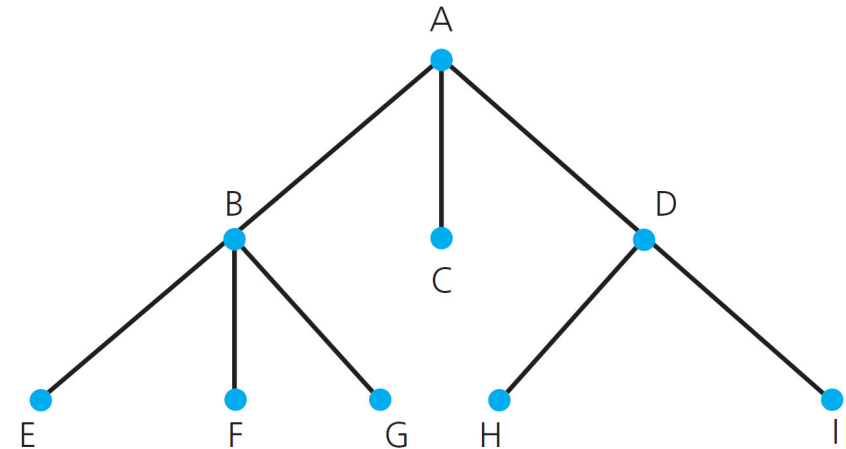




Kinds of Trees

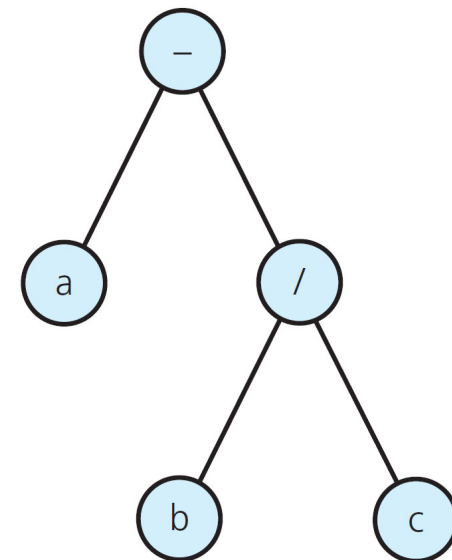
■ General tree

- Each node can have any number of children nodes.



■ Binary tree

- Each node has up to two pointers: one to the left subtree and the other to the right subtree.
- One or both pointers could be null (i.e., empty subtree)





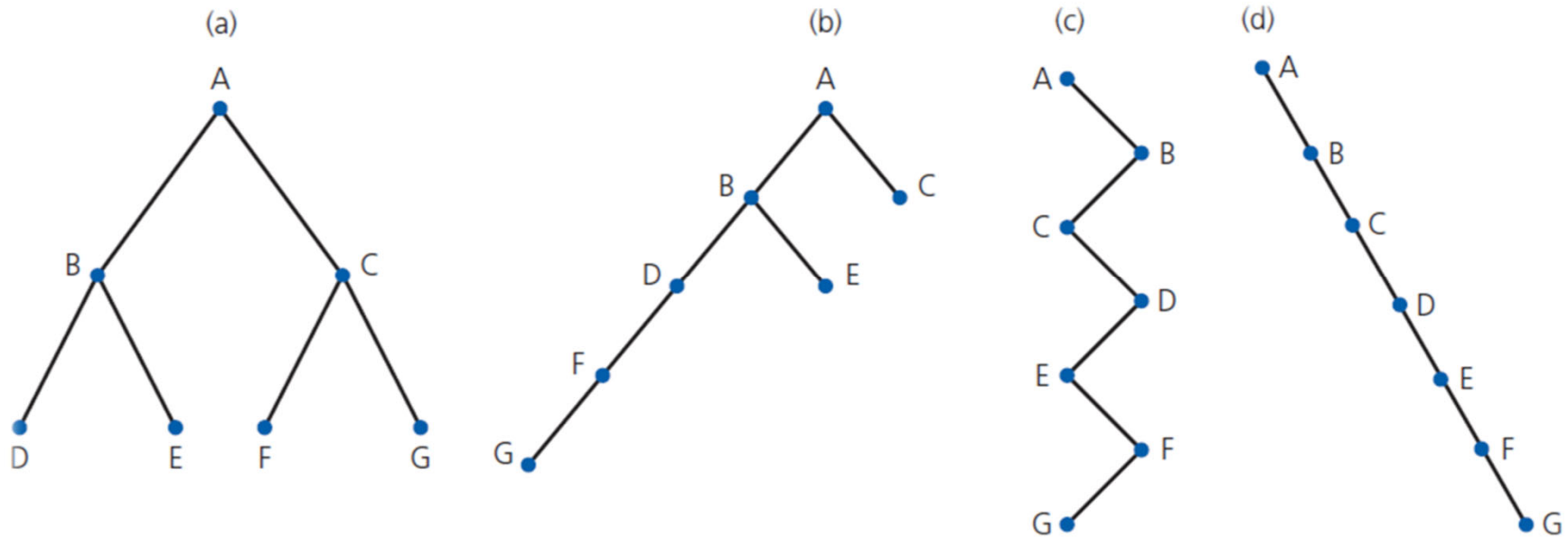
The Height of Trees

- Level of node n
 - If n is root, its level is 1
 - If n is not the root, its level is 1 greater than the 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

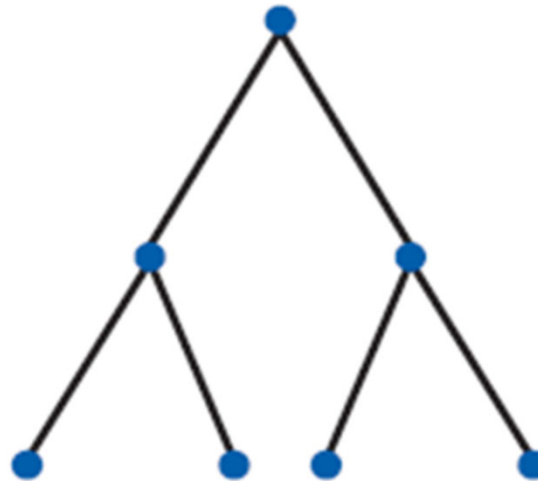


Binary trees with the same nodes but with heights, respectively, 3, 5, 7, and 7.



Full Binary Tree

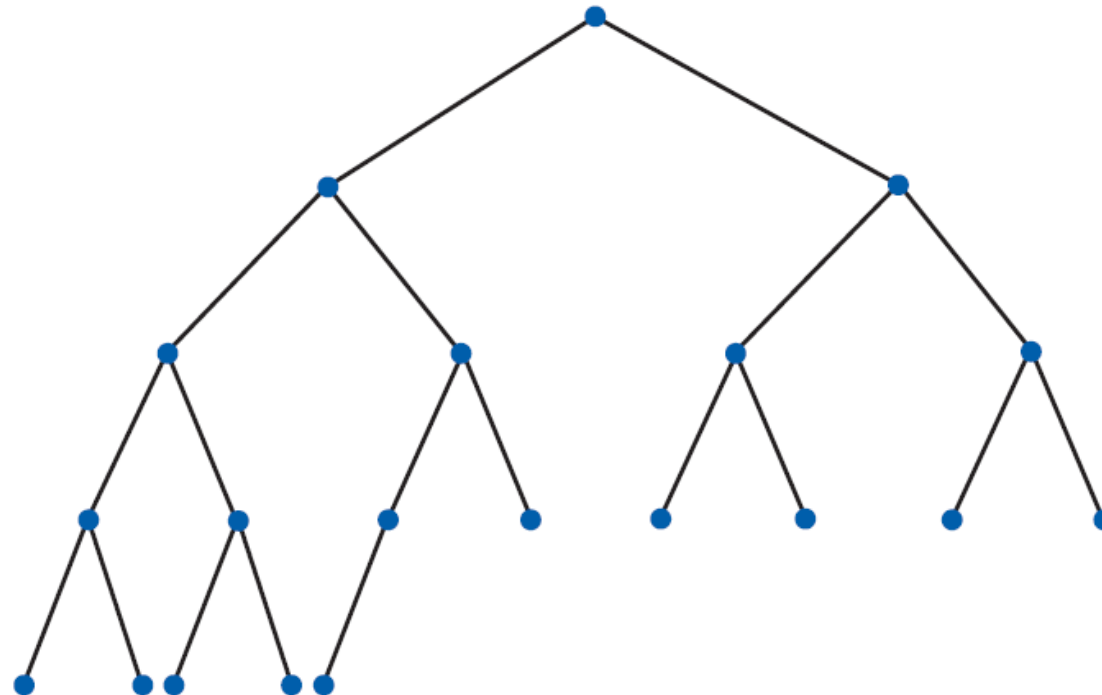
In a **full binary tree** of height h , all nodes that are at a level less than h have two children each.





Complete Binary Tree

In 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.

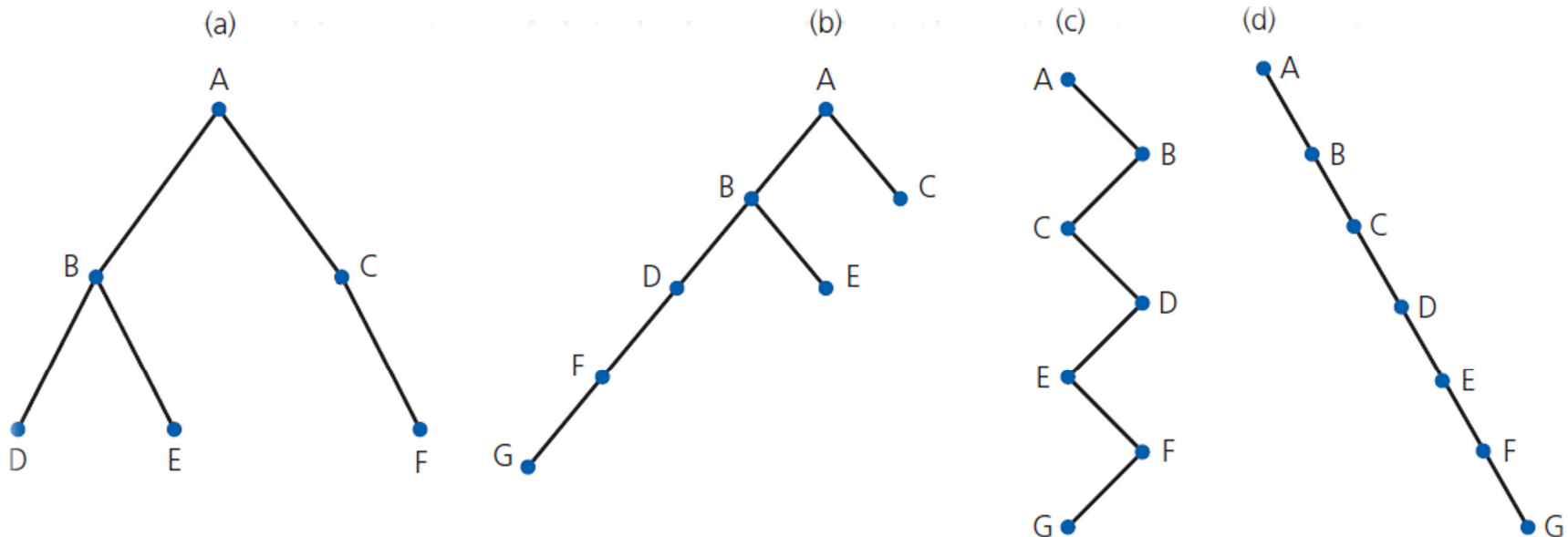




Balanced Binary Tree

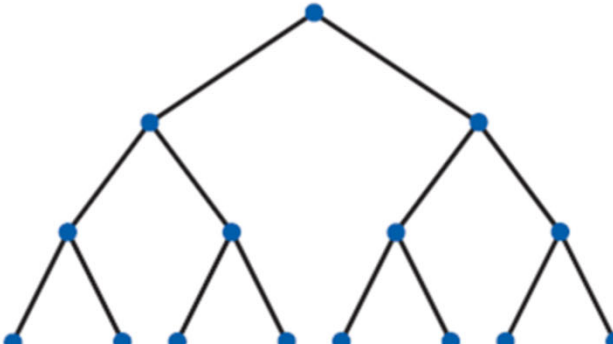
In a **balanced binary tree** the height of any node's right subtree differs from the height of the node's left subtree by no more than 1.

- The binary tree (a) is balanced, but the other trees are not balanced.





Counting the Nodes

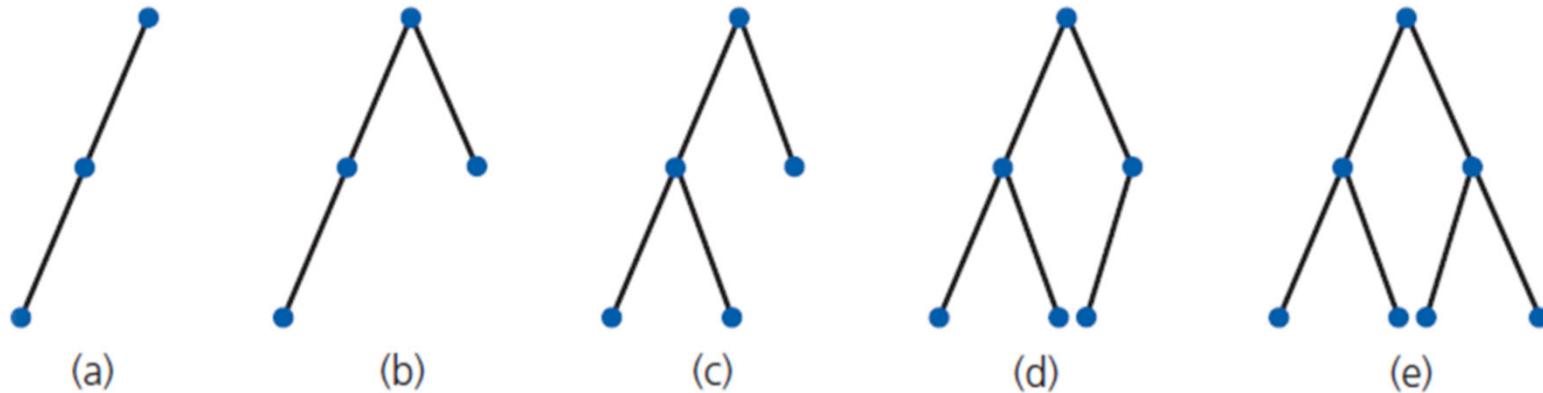
	Level	Number of nodes at this level	Total number of nodes at this level and all previous levels
	1	$1 = 2^0$	$1 = 2^1 - 1$
	2	$2 = 2^1$	$3 = 2^2 - 1$
	3	$4 = 2^2$	$7 = 2^3 - 1$
	4	$8 = 2^3$	$15 = 2^4 - 1$
•	•	•	•
•	•	•	•
•	•	•	•
	h	2^{h-1}	$2^h - 1$

Counting the nodes in a full binary tree of height h



The Maximum and Minimum Heights

- Binary tree with n nodes
 - Maximum 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
 - The minimum height of a binary tree with n nodes is $\lceil \log_2 (n + 1) \rceil$.
 - *Note:* if it is full, then $n = 2^0 + 2^1 + \dots + 2^{h-1} = 2^h - 1$





The ADT Binary Tree

- Operations of ADT binary tree
 - Add, remove
 - Set, retrieve data
 - Test for empty
 - Traversal operations that visit every node



Traversals of a Binary Tree

- Traversal means to visit each item in the ADT.
- Visiting a binary tree node can mean displaying the node's content, copying the content into another data structure, or altering that content.
- Pseudocode for general form of a recursive traversal algorithm:

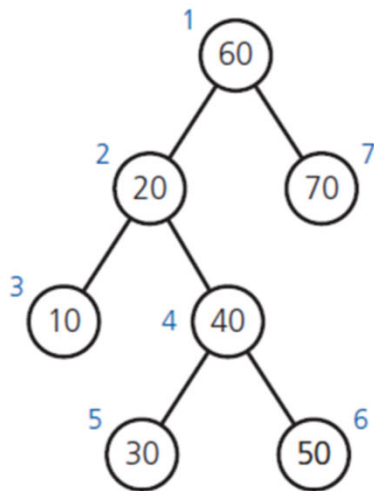
```
if (T is not empty)  
{  
    Display the data in T's root  
    Traverse T's left subtree  
    Traverse T's right subtree  
}
```

- Traversal can visit nodes in several different orders.

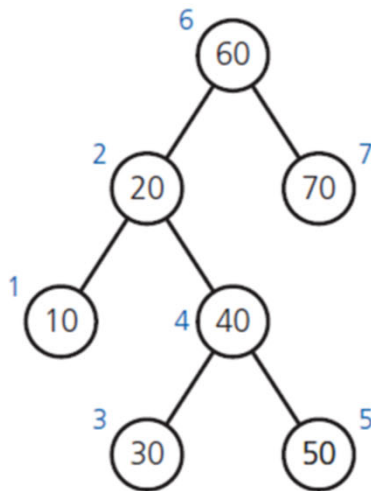


Traversals Options

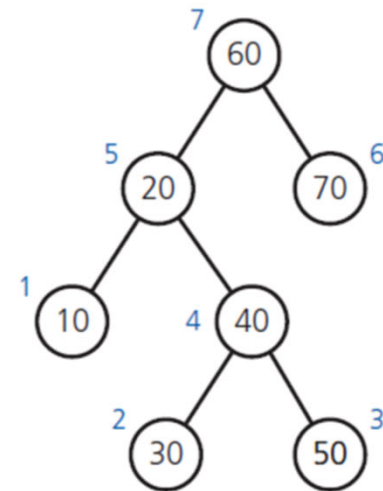
- Options for when to visit the **root**
 - **Preorder**: it traverses *root* → *left subtree* → *right subtree*
 - **Inorder**: it traverses *left subtree* → *root* → *right subtree*
 - **Postorder**: it traverses *left subtree* → *right subtree* → *root*
- Note traversal is $O(n)$



(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  
    }  
}
```



Binary Tree Operations

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
```



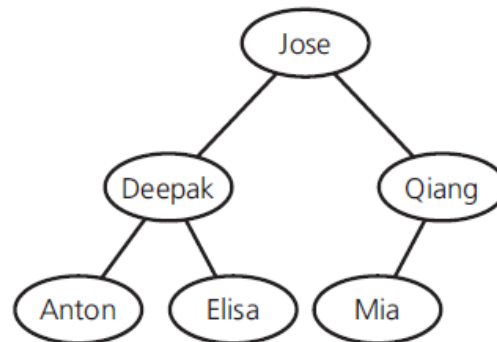
Nodes in a Binary Tree

- Representing tree nodes
 - Must contain both data and the “locations” of the node’s children
 - Each node will be an object
- Array-based
 - Children locations are array indices
- Link-based
 - C++ pointers are used to store the children locations.



Array-Based Representation

(a)



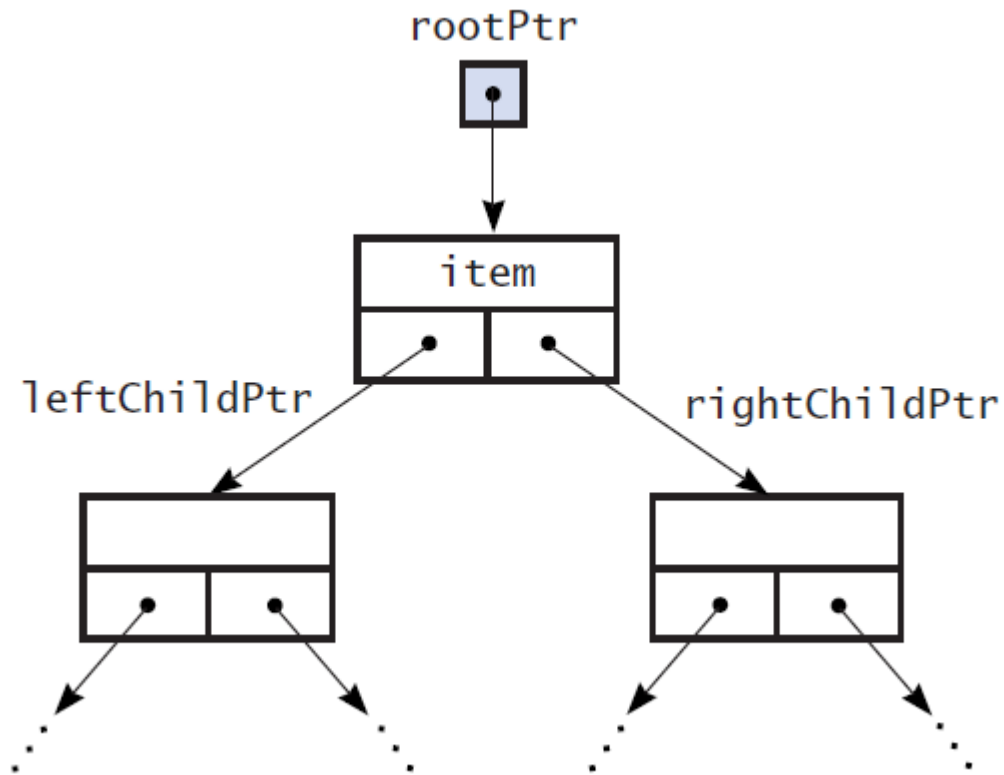
(b)

The array tree				root
	item	leftChild	rightChild	
0	Jose	1	2	0
1	Deepak	3	4	free
2	Qiang	5	-1	6
3	Anton	-1	-1	
4	Elisa	-1	-1	
5	Mia	-1	-1	
6	?	-1	7	Free list
7	?	-1	8	
8	?	-1	9	
•	•	•	•	
•	•	•	•	
•	•	•	•	

- a) A binary tree of names.
b) Its implementation using the array **tree**.



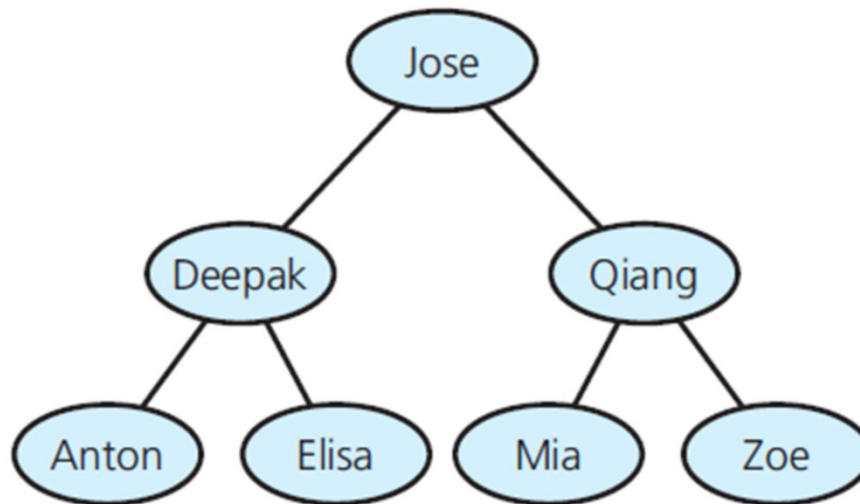
Link-Based Representation





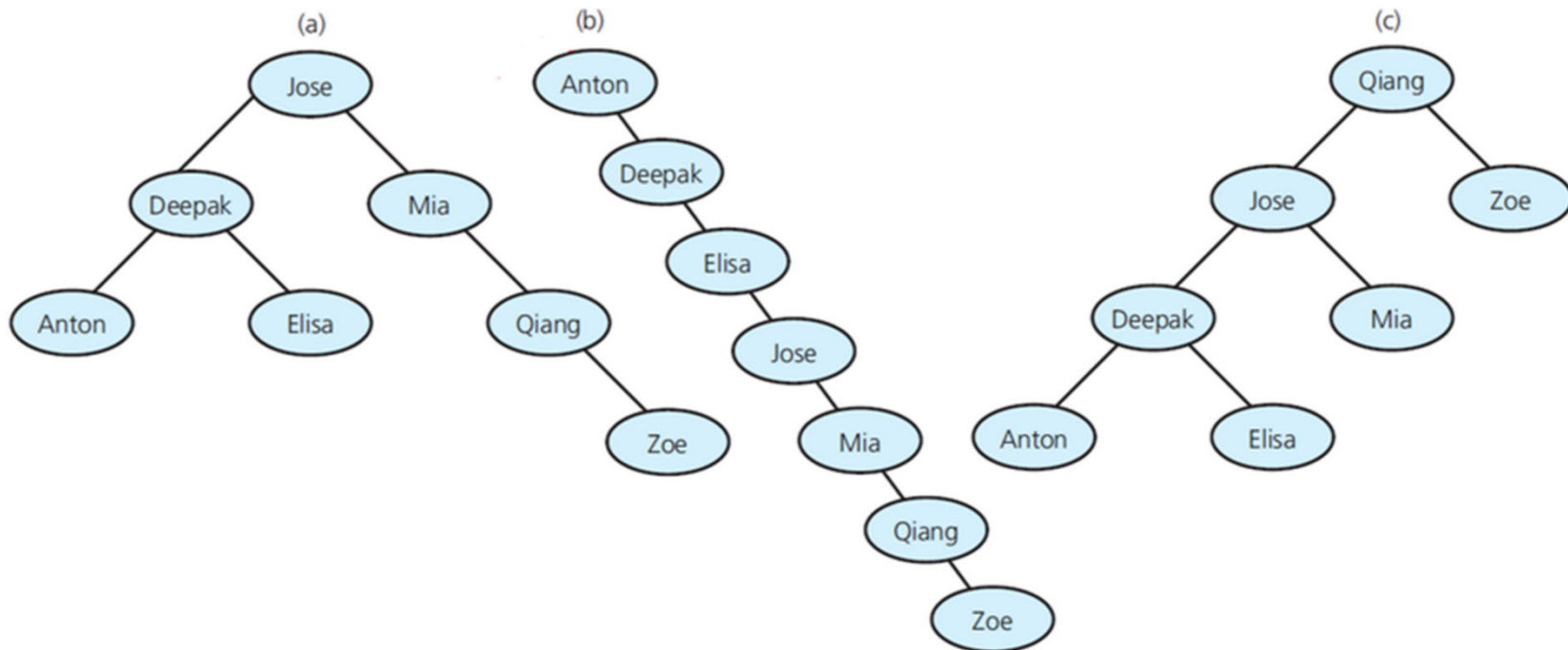
The ADT Binary Search Tree

- Recursive definition of a binary search tree (BST)
 - The value of the BST root is:
 - greater than all values in its left subtree T_L and
 - less than all values in its right subtree T_R .
 - Both T_L and T_R are binary search trees.





Different BSTs with the Same Data



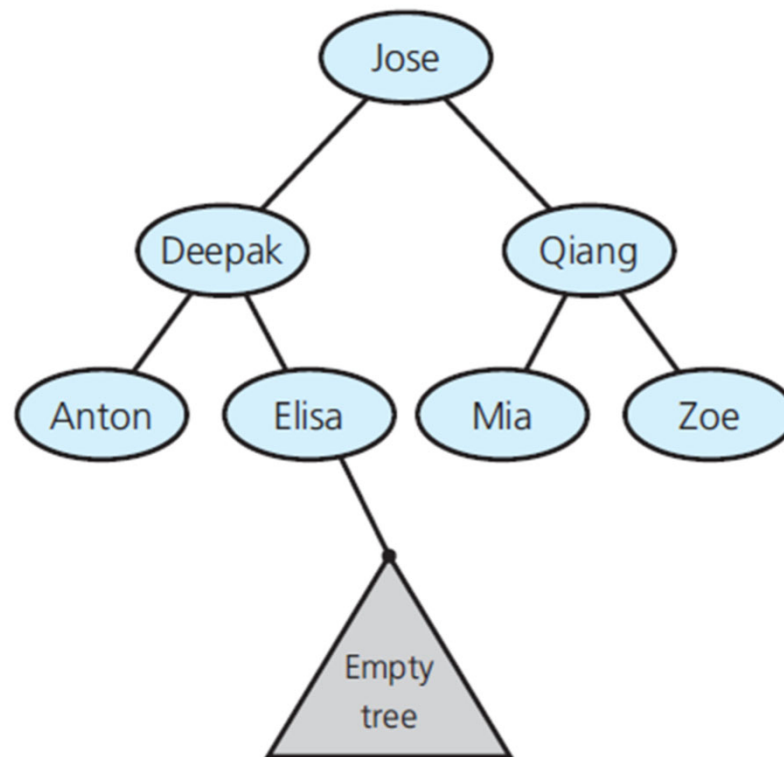


Searching a BST

```
// 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)
}
```



Searching BST Example



When looking for **Finn**, the search algorithm terminates with "item not found"



Traversals of a BST

```
// 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 BST Operations

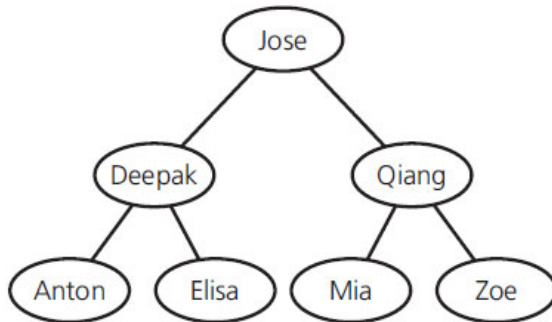
- Max number of comparisons for retrieval, addition, or removal is the *height* of the tree.
- Adding entries in sorted order produces *maximum-height* binary search tree.
- Adding entries in random order might produce near-minimum-height binary search tree.
 - How to produce a minimum-height binary search tree?

<u>Operation</u>	<u>Average case</u>	<u>Worst case</u>
Retrieval	$O(\log n)$	$O(n)$
Addition	$O(\log n)$	$O(n)$
Removal	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$

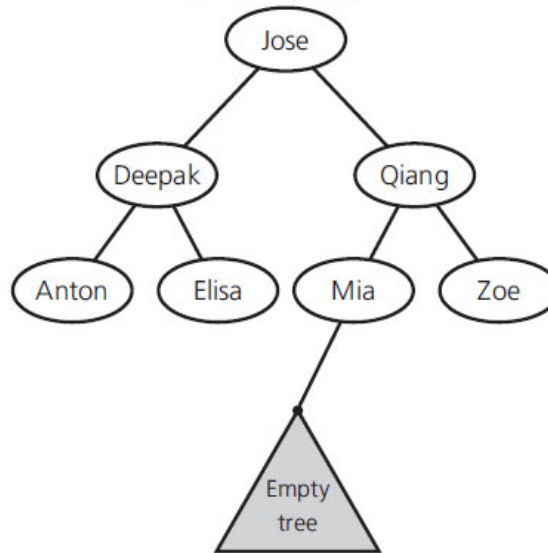


Adding a Node to a BST

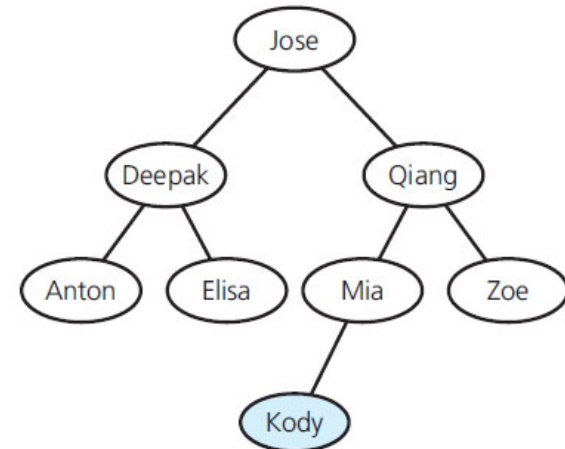
(a) A binary search tree



(b) A search for Kody terminates at an empty subtree



(c) Kody is **added** as a new leaf



Example: Adding **Kody** to a binary search tree



Removing a Node from a BST (1 of 4)

- There are three cases of removing a node from a BST depending on the location of the node to be removed.
- Case 1: The node to be removed is a leaf
 - Remove the node
 - Set pointer in its parent to `nullptr`



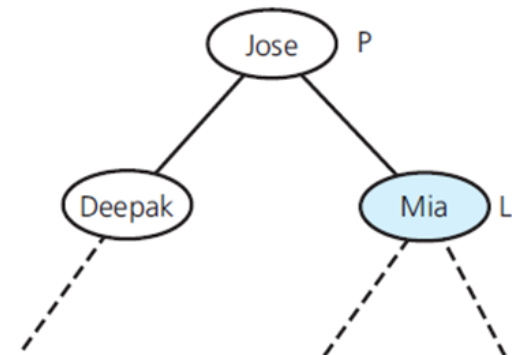
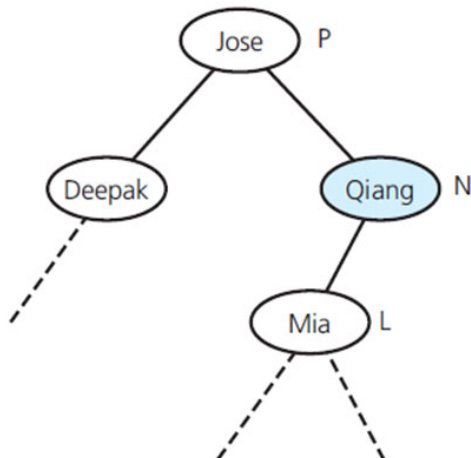
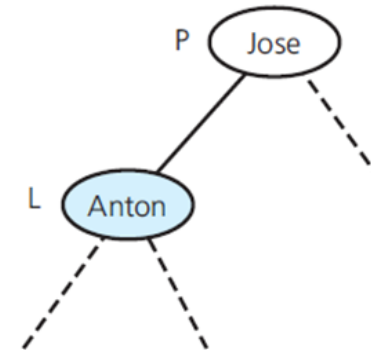
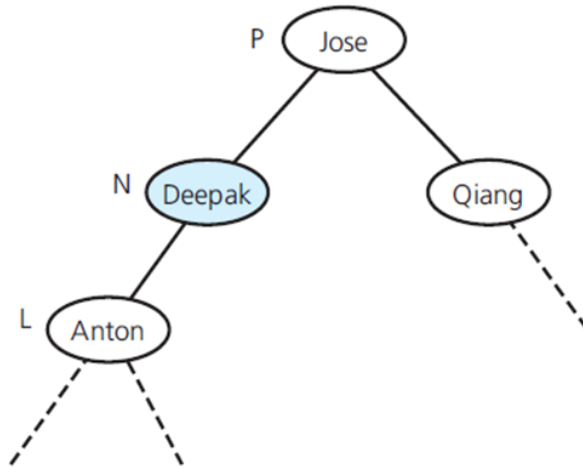
Removing a Node from a BST (2 of 4)

- Case 2: The node, N , to be removed has only one child (left or right)
 - After N is removed, all data items rooted at N 's child, are adopted by N 's parent.
 - All items adopted are in correct order, binary search tree property preserved



Removing a Node from a BST (3 of 4)

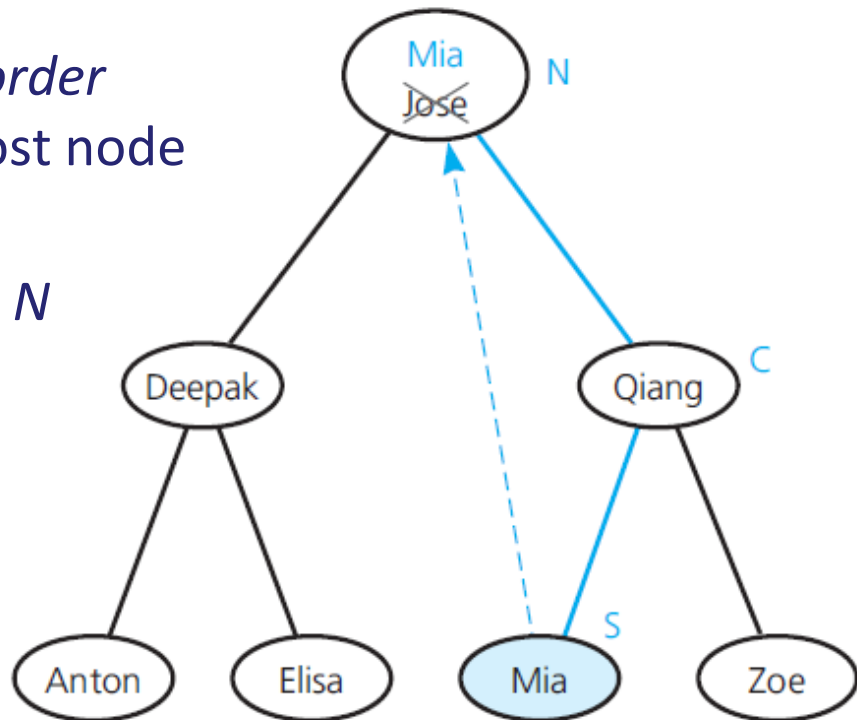
- Case 2 Examples:





Removing a Node from a BST (4 of 4)

- Case 3: The node, N , to be removed has two children
 - Locate node M that is N 's *inorder successor* (the leftmost node in N 's right subtree)
or
Locate node M as N 's *inorder predecessor* (the rightmost node in N 's left subtree)
 - Copy item that is in M to N
 - Remove M from tree





Saving a BST in a File

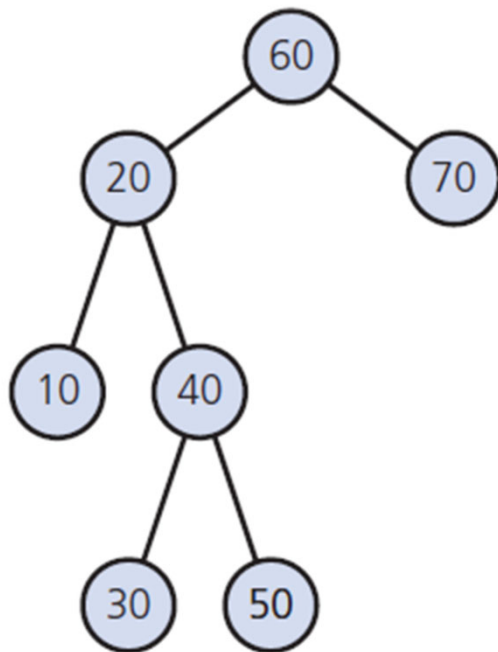
- Assume a program uses a BST to represent a database. As the program terminates, it must save the tree's data in a file so that it can later restore the tree.
- Two approaches can be used:
 1. Saving the BST's contents and then restoring it to its original shape.
 2. Saving the BST's contents and then restoring it to a balanced shape. As balanced BST increases efficiency of ADT operations.



Restoring a BST to its Original Shape

Consider the shown. If you save the tree in **preorder**, you get the sequence: 60, 20, 10, 40, 30, 50, 70.

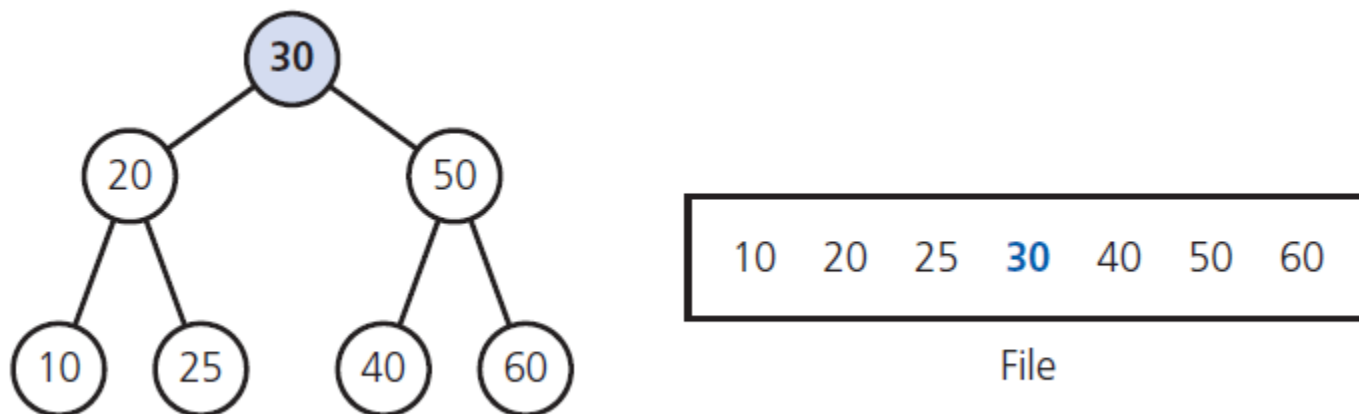
If you then use the add method to insert these values into a BST that is initially empty, you will get the original tree.





Restoring a BST to a Balanced Shape

- If you save a full tree in a file by using an **inorder** traversal, the file will be in sorted order.
- A full tree with exactly $n = 2^h - 1$ nodes for some height h has the exact middle of the data items (the median) in its root (as shown below).
- The left and right subtrees of the root are full trees of $2^{h-1} - 1$ nodes each (that is, half of $n - 1$).
- Thus, you can use a recursive algorithm to create a full binary search tree with n nodes (knowing n is a simple matter of counting nodes as you traverse the tree and then saving this count in the file).





Algorithm to Restore a Full BST

*// Builds a full binary search tree from n sorted values in a file.
// Returns a pointer to the tree's root.*

readFullTree(n: integer): BinaryNodePointer

if (n > 0) {

nodePtr = pointer to new node with children pointers are nullptr

// Construct the left subtree

leftPtr = readFullTree(n/2)

nodePtr->setLeftChildPtr(leftPtr)

// Get the root

rootItem = next item from file

nodePtr->setItem(rootItem)

// Construct the right subtree

rightPtr = readFullTree(n/2)

nodePtr->setRightChildPtr(rightPtr)

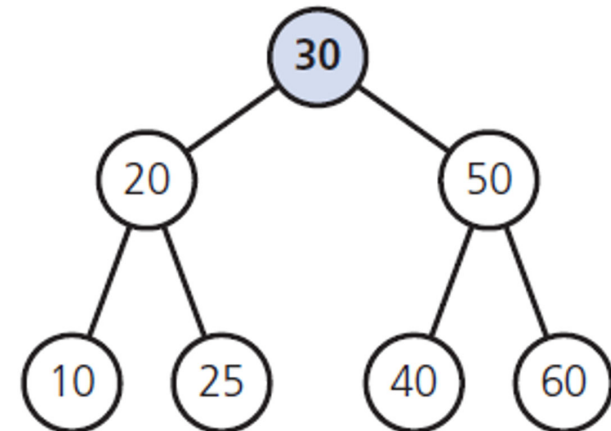
return nodePtr

}

else

return nullptr

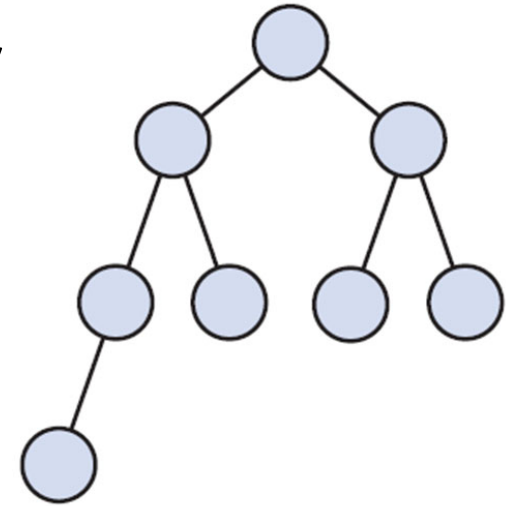
10 20 25 30 40 50 60





Balancing a not Full BST

- The readFullTree algorithm is essentially correct even if the tree is not full as shown.
- If n is odd, both subtrees are of size $n/2$, as before, and the root is automatically accounted for.



- If n is even, however, you have to deal with the root and the fact that one of the root's subtrees will have one more node than the other. In this case, you can arbitrarily choose to put the extra node in the left subtree by making the following changes:

```
leftPtr = readFullTree(n/2)
rightPtr = readFullTree((n-1)/2)
```



Tree Sort

- You can use the ADT BST to sort an array efficiently. The basic idea of the algorithm is simple:

// Sorts the integers in an array into ascending order.

```
treeSort(anArray: array, n: integer)
```

Insert anArray's entries into a binary search tree BST
Traverse BST in inorder.

As you visit the BST's nodes, copy their data items
into successive locations of anArray



Tree Sort Analysis

- Each insertion into a BST requires $O(\log n)$ operations in the average case and $O(n)$ operations in the worst case.
- Thus, n insertions require $O(n \times \log n)$ operations in the average case and $O(n^2)$ operations in the worst case.
- The traversal of the tree involves one copy operation for each of the n entries and so is $O(n)$.
- Because $O(n)$ is less than $O(n \times \log n)$ and $O(n^2)$, tree sort in the average case is $O(n \times \log n)$ and $O(n^2)$ in the worst case.