Data Structures and Advanced Programming

Robin Bing-Yu Chen
National Taiwan University

Trees & Implementations

- Specifying the Tree
- Specifying the ADT Tree
- Specifying the ADT Binary Tree
- Specifying the ADT Binary Search Tree
- Implementations of the ADT Tree

Categories of Data-Management Operations (1/2)

- ADT operations fit into at least one of these categories:
  - Insert data into a data collection.
  - Delete data from a data collection.
  - Ask questions about the data in a data collection.
- **Position-oriented ADTs**: Operations that
  - Insert data into the i^{th} position
  - Delete data from the i^{th} position
  - Ask a question about the data in the i^{th} position
  - Examples: list, stack, queue.
- **Value-oriented ADTs**: Operations that
  - Insert data according to its value.
  - Delete data knowing only its value.
  - Ask a question about data knowing only its value.
  - Examples: sorted list, priority queue.

Categories of Data-Management Operations (2/2)

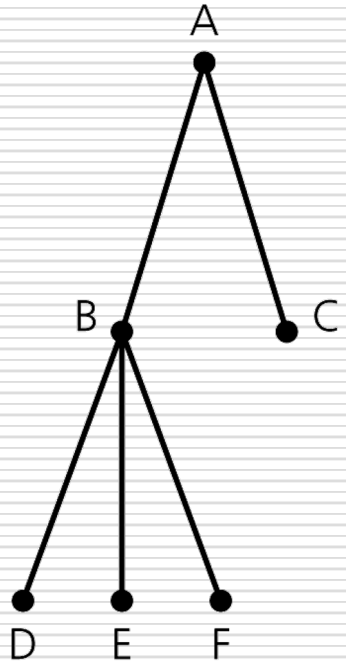
- In this chapter, we talk about "**Tree**", a very useful ADT in many real-world applications.

- Position-oriented ADT:
  - **binary tree**.

- Value-oriented ADT:
  - **binary search tree**.

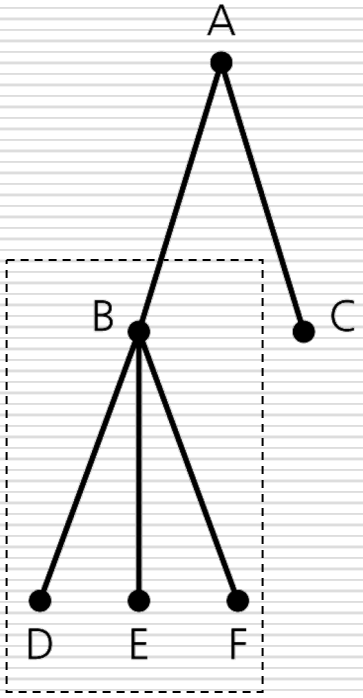
Terminology (1/3)

- Trees are composed of nodes (vertex) and edges.
- Trees are hierarchical.
  - Parent-child relationship between two nodes.
    - E.g., B and C are children of A.
- Each node in a tree has at most one parent.
  - Except root, has no parent.
    - E.g., A.
- A node that has no children is called a leaf of the tree.
  - E.g., C, D, E, F.



Terminology (2/3)

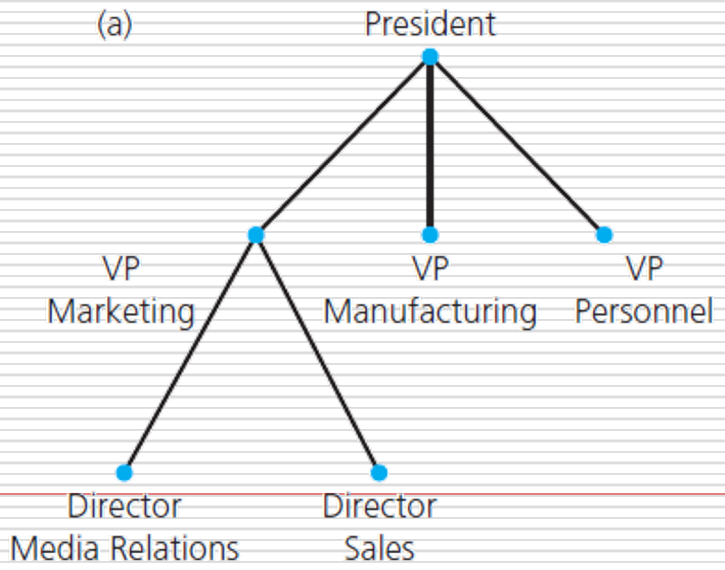
- **Sibling** relationship: children of the same parent.
  - E.g., B and C.
- **Ancestor-descendant** relationships among nodes.
  - A is an ancestor of D.
  - D is a descendant of A.
  - The root of any tree is an ancestor of every node in that tree.
- **Subtree** of a tree: Any node and its descendants.

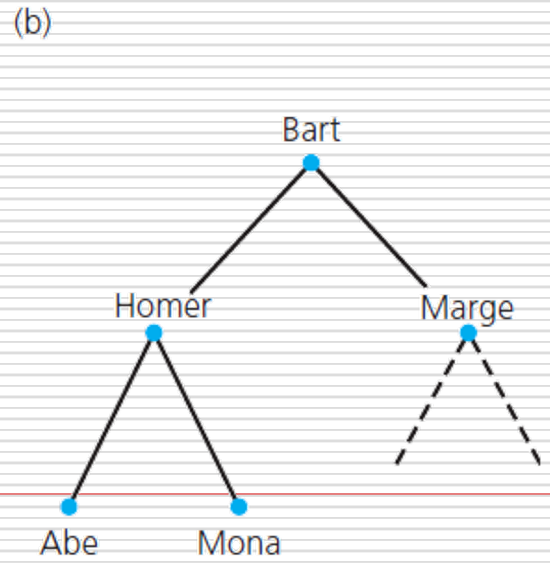


A subtree example.

Terminology (3/3)

- The hierarchical property of tree:
  - Can be used to represent information that itself is hierarchical in nature.
    - E.g., organization chart or family tree

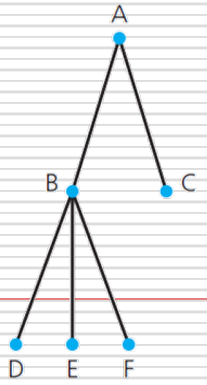




Kinds of Trees

- A **general tree** *T* is a set of one or more nodes such that *T* is partitioned into disjoint subsets:
  - A single node *r*, the root.
  - Sets that are general trees, called subtrees of *r*.

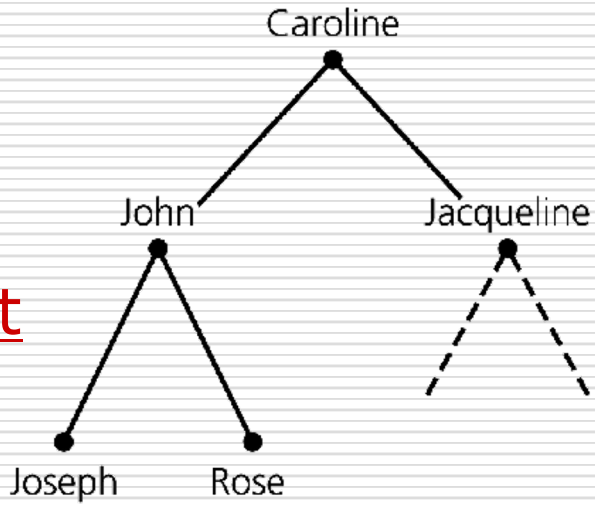
- An ***n*-ary** tree *T* is a set of nodes that is either empty or partitioned into disjoint subsets:
  - A single node *r*, the root.
  - *n* possibly empty sets that are *n*-ary subtrees of *r*.



← An example of 3-ary tree

A Binary Tree (1/2)

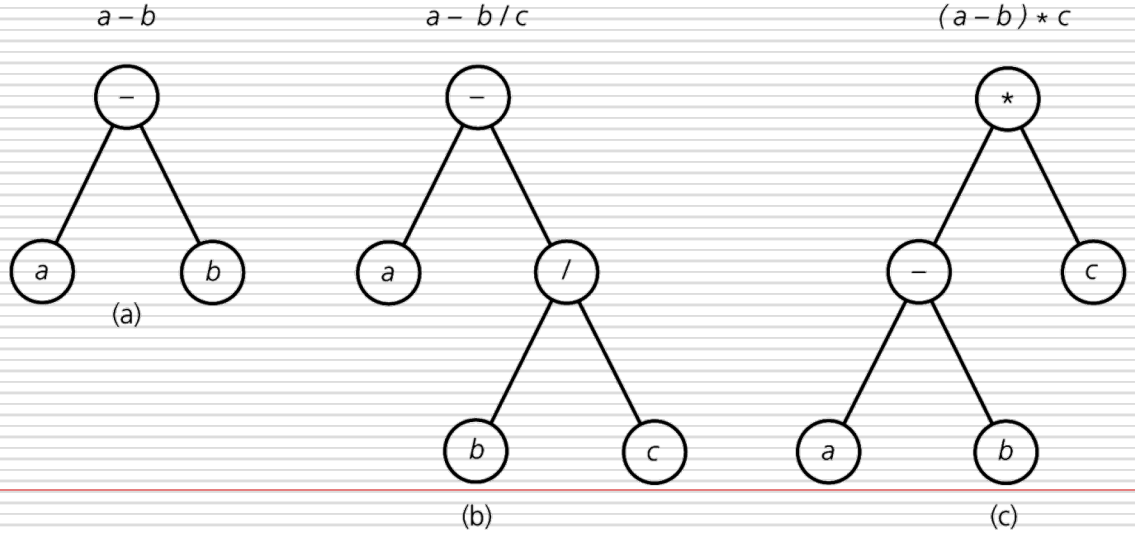
- A binary tree is a set *T* of nodes such that either:
  - *T* is empty, or
  - *T* is partitioned into disjoint subsets:
    - A single node *r*, the root.
    - Two possibly empty sets that are binary trees, called the left subtree of *r* and the right subtree of *r*.



Each node in a binary tree has no more than two children

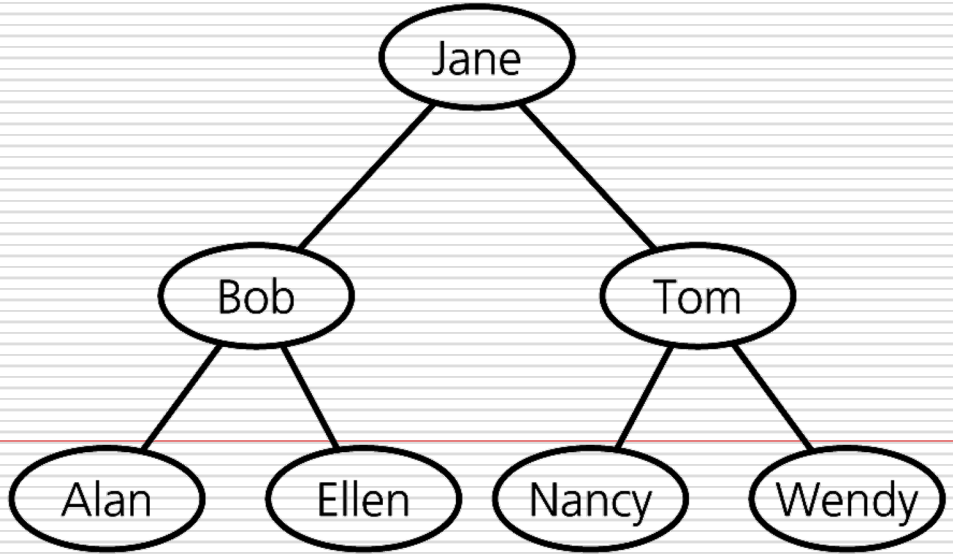
A Binary Tree (2/2)

- You can use binary trees to represent algebraic expressions.
  - Leafs are operands.
  - Evaluation manner: bottom-up.
    - The hierarchy specifies an unambiguous order for evaluating an expression.
    - Parentheses do not appear in these tree.



A Binary **Search** Tree

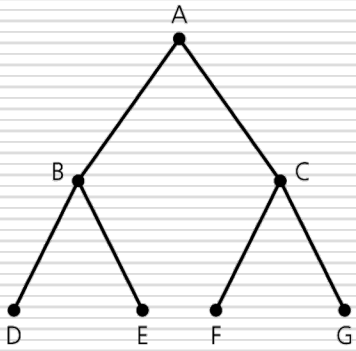
- A binary search tree is a binary tree.
- But has the following properties for each node *n*:
  - *n*'s value is > all values in *n*'s left subtree T_L.
  - *n*'s value is < all values in *n*'s right subtree T_R.
  - Both T_L and T_R are binary search trees.



The Height of Trees (1/3)

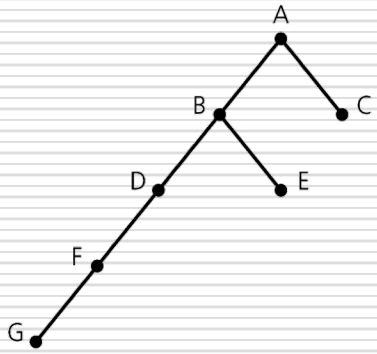
- Height of a tree:
  - Number of nodes along the longest path from the root to a leaf.

Height 3



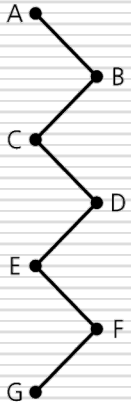
(a)

Height 5



(b)

Height 7

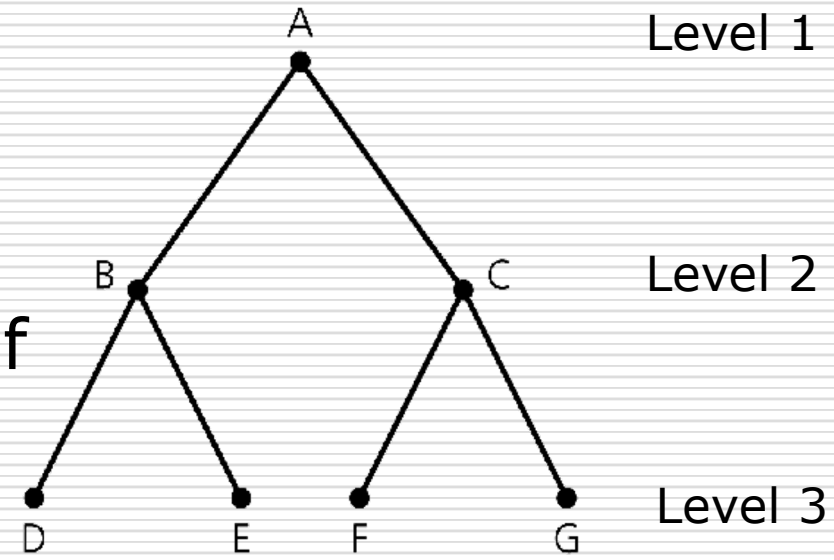


(c)

Trees with the same nodes but different heights!!

The Height of Trees (2/3)

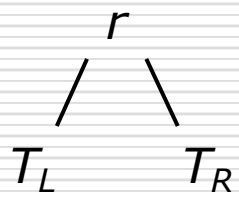
- Level of a node *n* in a tree *T*:
  - If *n* is the root of *T*, it is at level 1.
  - If *n* is not the root of *T*, its level is 1 greater than the level of its parent.



The Height of Trees (3/3)

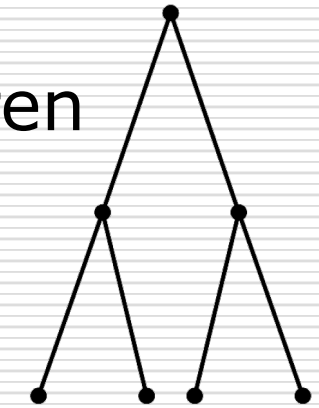
- Height of a tree *T* defined in terms of the levels of its nodes.
  - If *T* is empty, its height is 0.
  - If *T* is not empty, its height is equal to the maximum level of its nodes.

- A **recursive** definition of height:
  - If *T* is empty, its height is 0
  - If *T* is not empty,
    $height(T) = 1 + \max\{height(T_L), height(T_R)\}$



Full Binary Trees

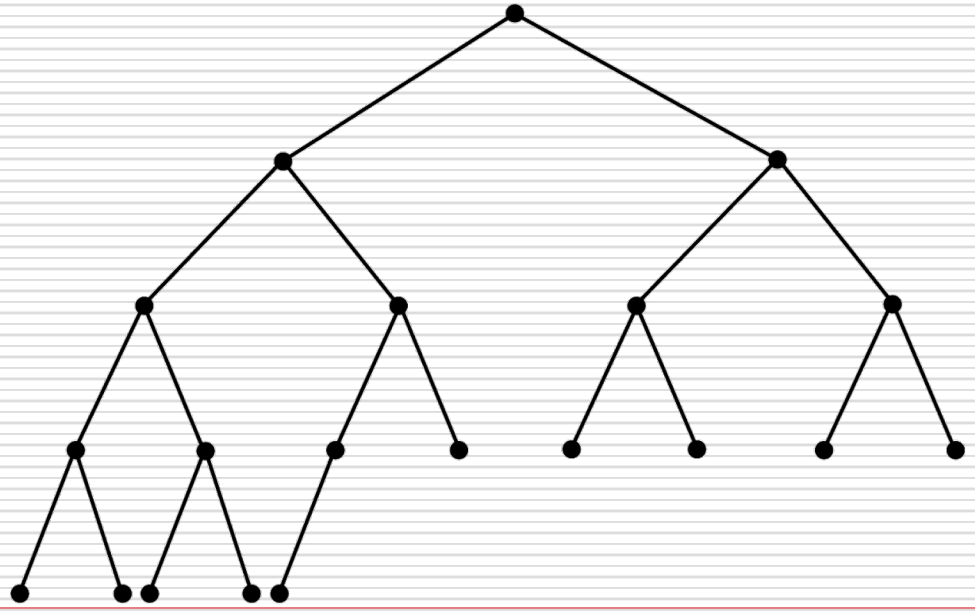
- A binary tree of height *h* is *full* if
  - Nodes at levels < *h* have two children each.



- Recursive definition:
  - 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 root's subtrees **are both** full binary trees of height *h* – 1.

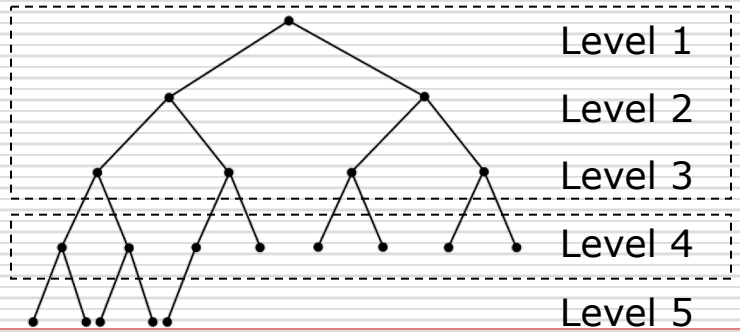
Complete Binary Trees (1/2)

- A binary tree of height *h* is *complete* if
  - it is full to level *h* – 1.
  - level *h* is filled from left to right.



Complete Binary Trees (2/2)

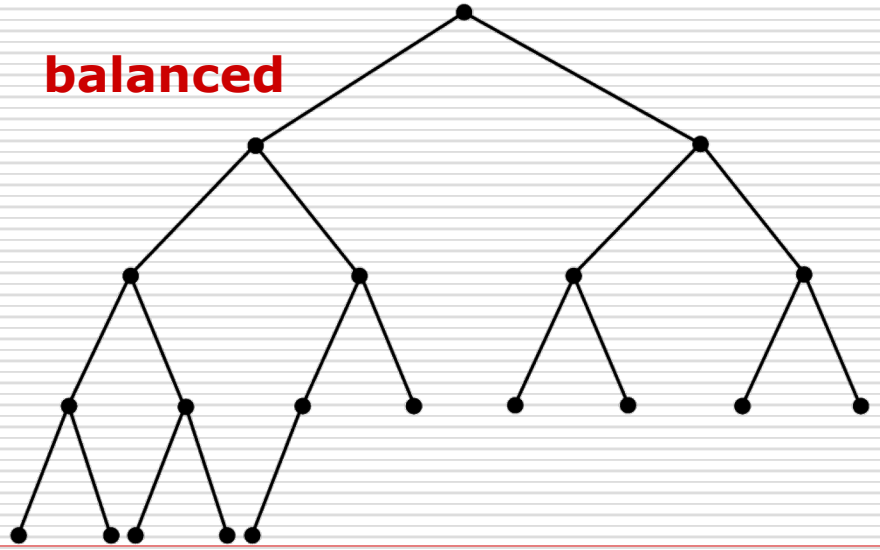
- Another definition:
  - A binary tree of height *h* is complete if
    - all nodes at levels <= *h* – 2 have two children each.
    - when a node at level *h* – 1 has children, all nodes to its left at the same level have two children each.
    - when a node at level *h* – 1 has one child, it is the left child.



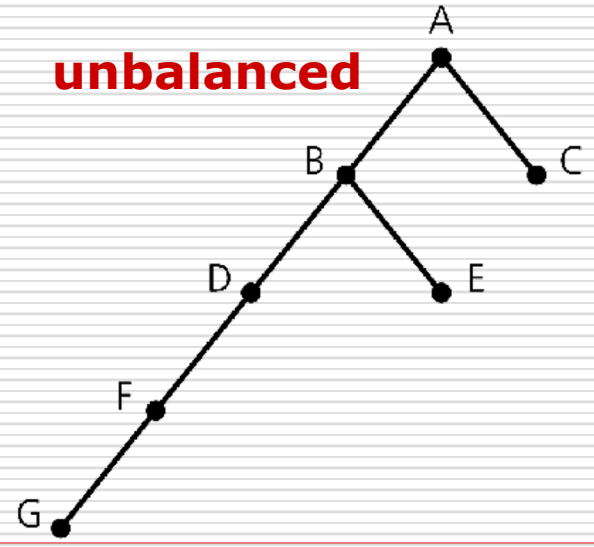
Balanced Binary Trees

- A binary tree is *balanced* if the heights of any node's two subtrees differ by no more than 1.
  - Complete binary trees are balanced.
  - Full binary trees are complete and balanced.

balanced

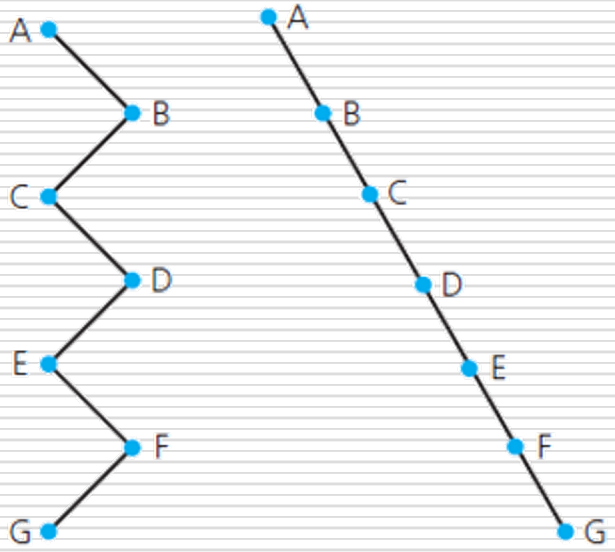


unbalanced



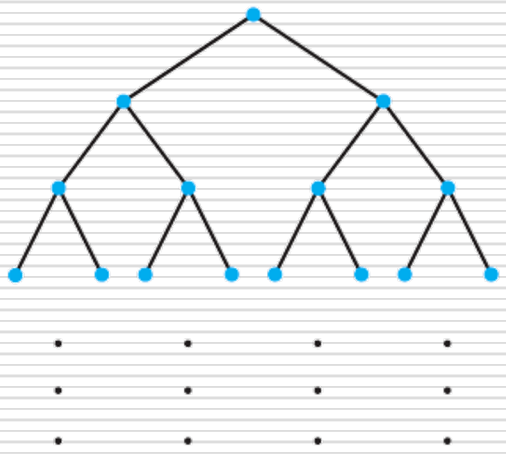
The Maximum Height of a Binary Tree

- The maximum height of an *n*-node binary tree is *n*.
  - By giving each internal node exactly one child.



The Minimum Height of a Binary Tree (1/2)

- To minimize the height of a binary tree given *n* nodes, you must fill each level of the tree as completely as possible.



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 Minimum Height of a Binary Tree (2/2)

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

- Complete trees and full trees have minimum height.

The ADT Binary Tree

- As an abstract data type, the binary tree has operations that:
  - **Add** and **remove** nodes
  - **Set** or **retrieve** the data in the root of the tree
  - **Test** whether the tree is empty
  - **Traversal** operations that visit every node in a binary tree
    - Visit → doing **something** with or to the node.

Traversals of a Binary Tree (1/6)

- A traversal visits **each node** in a tree.
  - You do something with or to the node during a visit.
  - For example, **display** or **modify** the data in the node.
    - Assume that visiting means displaying data.

- With the recursive definition of a binary tree, you can construct a recursive traversal algorithm.
  - If the tree is not empty, the traversal algorithm must perform three tasks:
    - Display the data in the root.
    - Traverse the two subtrees (recursive … smaller problems).
  - The base case: If the tree is empty, the algorithm takes no action.

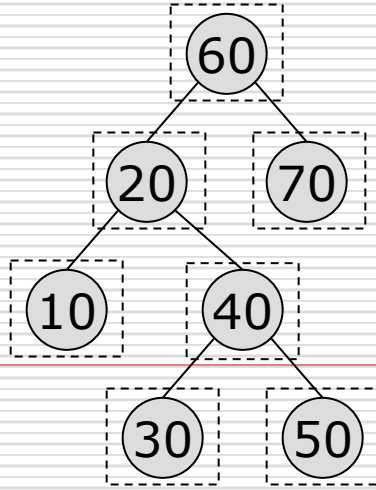
Traversals of a Binary Tree (2/6)

- The algorithm has ***three choice of when to visit the root***.
  - Preorder traversal
    - Visit root before visiting its subtrees.
    - i.e., Before the recursive calls.
  - Inorder traversal
    - Visit root between visiting its subtrees.
    - i.e., Between the recursive calls.
  - Postorder traversal
    - Visit root after visiting its subtrees.
    - i.e., After the recursive calls.

Traversals of a Binary Tree (3/6)

- The pseudo code of the preorder traversal algorithm:

```
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)
   }
```

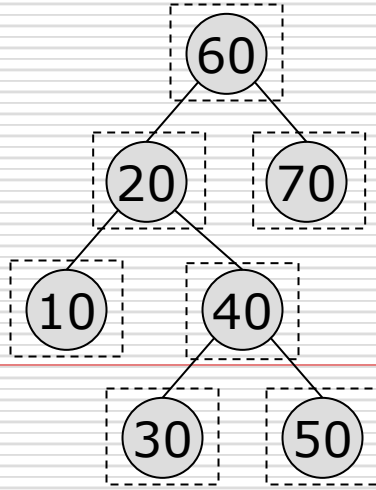


Preorder: 60 20 10 40 30 50 70

Traversals of a Binary Tree (4/6)

- The pseudo code of the postorder traversal algorithm:

```
postorder(in binTree:Binarytree): void
   if(binTree is not empty)
   {  postorder(Left subtree of binTree's root)
      postorder(Right subtree of binTree's root)
      Display the data in the root of binTree
   }
```



Postorder: 10 30 50 40 20 70 60

Traversals of a Binary Tree (5/6)

- Each of these traversals visits every node in a binary tree exactly once.
  - *n* visits occur for a tree of *n* nodes.
- Each visit performs the same operations on each node, independently of *n*.
  - It must be O(1).
- Thus, each traversal is O(*n*).

Traversals of a Binary Tree (6/6)

- The task of tree traversal can be more than to display each item when you visit it.
  - Copy the item or even alter it.
  - Thus, you might devise many different traversal operations:
    - `preorderTraverseAndDisplay`, `preorderTraverseAndCopy`, …

- A better solution: devise a traversal operation can call a function to perform a task on each item in the tree.
  - Function of displaying, copying, or altering items.
  - The client defines and passes this function as an argument to the traversal operation.
  - For instance: `bintree.preorderTraverse(display);`
  - Next chapter will discuss the implementation details.

Binary Tree Operations (1/4)

- Data:
  - A finite number of objects in hierarchical order.
- Operations:

`isEmpty()`	**Task:** Tests whether this binary tree is empty. **Input:** None. **Output:** True if the binary tree is empty; otherwise false.
`getHeight()`	**Task:** Gets the height of this binary tree. **Input:** None. **Output:** The height of the binary tree.
`getNumberOfNodes()`	**Task:** Gets the number of nodes in this binary tree. **Input:** None. **Output:** The number of nodes in the binary tree.

Binary Tree Operations (2/4)

`getRootData()`	**Task:** Gets the data that is in the root of this binary tree. **Input:** none. **Output:** The root's data
`setRootData(newData)`	**Task:** Replaces the data item in the root of this binary tree with `newData`, if the tree is note empty. However, if the tree is empty, **inserts a new root node** whose data item is `newData` into the tree **Input:** `newData` is the data item. **Output:** None.
`add(newData)`	**Task:** Adds a new node containing a given data item into this binary tree. **Input:** `newData` is the data item. **Output:** True if the addition is successful, or false if not.

Binary Tree Operations (3/4)

`remove(data)`	**Task:** Removes the node containing the given data item from this binary tree. **Input:** `data` is the data item. **Output:** True if the removal is successful, or false if not.
`clear()`	**Task:** Removes all nodes from this binary tree. **Input:** None. **Output:** None. (The tree is empty.)
`getEntry(anEntry)`	**Task:** Gets a specific entry in this binary tree. **Input:** `anEntry` is the desired data item. **Output:** The entry in the binary tree that matches `anEntry`. Throws an exception if the entry is not found.

Binary Tree Operations (4/4)

`contains(data)`	**Task:** Tests whether the given data item occurs in this binary tree. **Input:** `data` is the data item. **Output:** True if the binary tree contains the given data item, or false if not.
`preorderTraverse(visit)` **`inorderTraverse(visit)`** **`postorderTraverse(visit)`**	**Task:** Traverses this binary tree in preorder/inorder/postorder and calls the function `visit` once for each node. **Input:** `visit` is a client-defined function that performs an operation on or with the data in each visited node. **Output:** None.

The ADT Binary **Search** Tree

- The ADT binary **search** tree is suitable for searching a tree for a particular data item.

- For each node *n* in 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.

- This organization of data enables you to search a binary search tree for a particular data item efficiently.

Binary Search Tree Operations (1/2)

- As an ADT, the binary search tree has operations of **inserting**, **removing**, and **retrieving** data.
  - Unlike the position-oriented ADTs stack, list, and queue, but like the ADT sorted list, the insertion, removal, and retrieval operations are by value, not by position.
  - The traversal operations that you just saw for a binary tree apply to a binary search tree without change.
    - Because a binary search tree is a binary tree!!

Binary Search Tree Operations (2/2)

- Data:
  - A finite number of objects in hierarchical order.
- Operations:

`add(newEntry)`	**Task:** Inserts `newEntry` into this binary search tree such that the properties of a binary search tree are maintained. **Input:** `newEntry` is the data item to be inserted. Assumes the entries in the tree are distinct and differ from `newEntry` **Output:** True if the insertion is successful, or false if not.
`remove(anEntry)`	**Task:** Removes the given entry from this binary search tree such that the properties of a binary search tree are maintained. **Input:** `anEntry` is the entry to remove. **Output:** True if the removal is successful, or false if not.
The methods **`isEmpty`**, **`getHeight`**, **`getNumberOfNodes`**, **`getRootData`**, **`clear`**, **`getEntry`**, **`contains`**, **`preorderTraverse`**, **`inorderTraverse`**, and **`postorderTraverse`** have the same specifications as for a binary tree.	

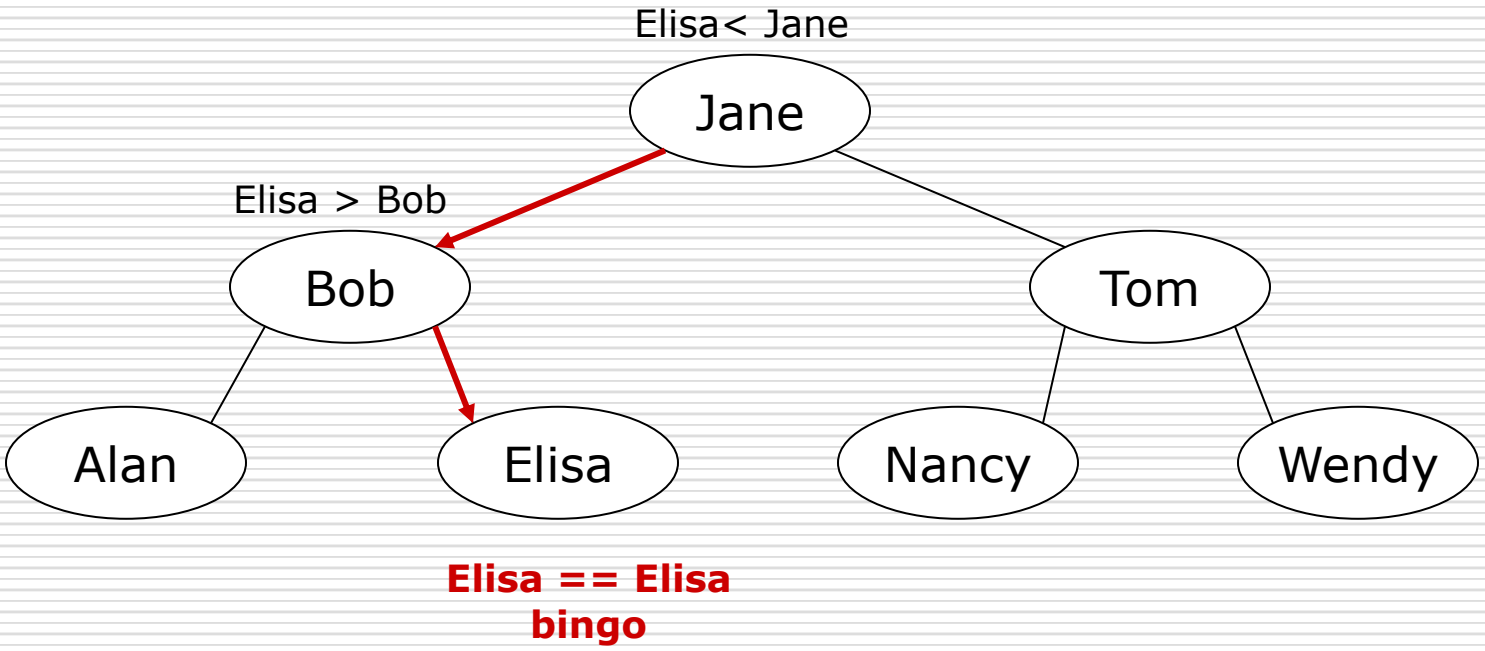
ADT Binary Search Tree: **Search** Algorithm (1/4)

- Because a binary search tree is recursive by nature, it is natural to formulate recursive algorithms for operations on the tree.

```
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 // searchKey > root item
            search(Right subtree of bstTree, target)
```

ADT Binary Search Tree: Search Algorithm (2/4)

- Suppose you want to locate Elisa's record in the binary tree.



ADT Binary Search Tree: Search Algorithm (3/4)

- As you will see, this search algorithm is the basis of the other operations on a binary search tree!!
- Note that the shape of the tree in no way affects the validity of the search algorithm.
- However, the search algorithm works more efficiently on some trees than on others!!

ADT Binary Search Tree: Search Algorithm (4/4)

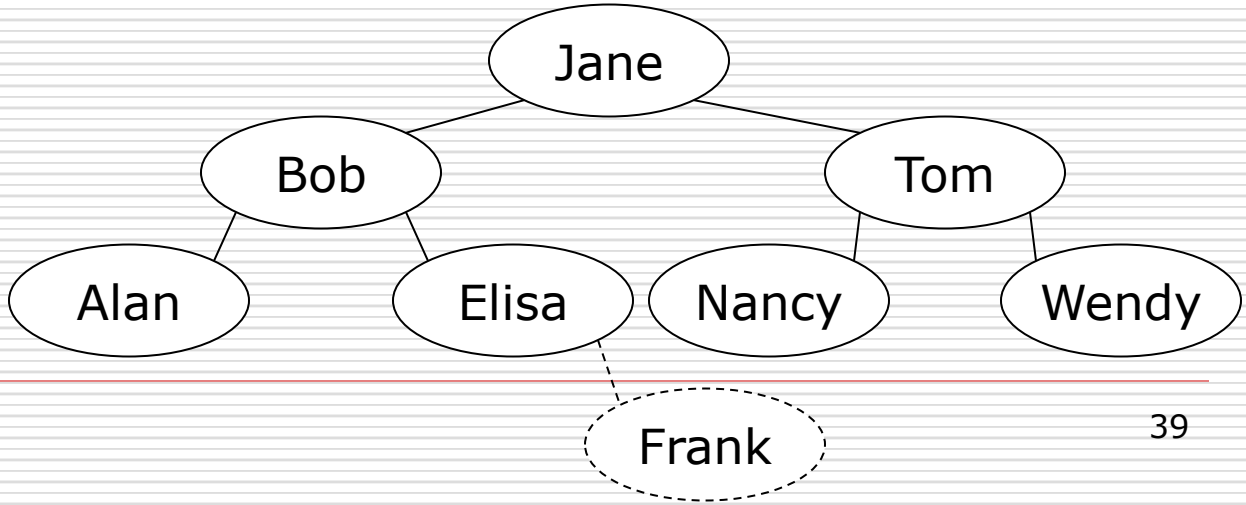
- The shape of a binary search tree **affects the efficiency of its operations**.

- The more balanced a binary search tree is, the farther it is from a linear structure.
  - The behavior of the search algorithm will be to a binary search of an array!!

ADT Binary Search Tree: **Insertion** (1/2)

- Suppose that you want to insert a record for Frank into the binary search tree.

- Imagine that you instead want to search for the item with Frank.
  - Frank must be the right child of Elisa!!



ADT Binary Search Tree: Insertion (2/2)

- Search always tells you the insertion point.
  - It will always terminate at an empty subtree.
  - Or … search always tells you to insert the item **as a new leaf**!!

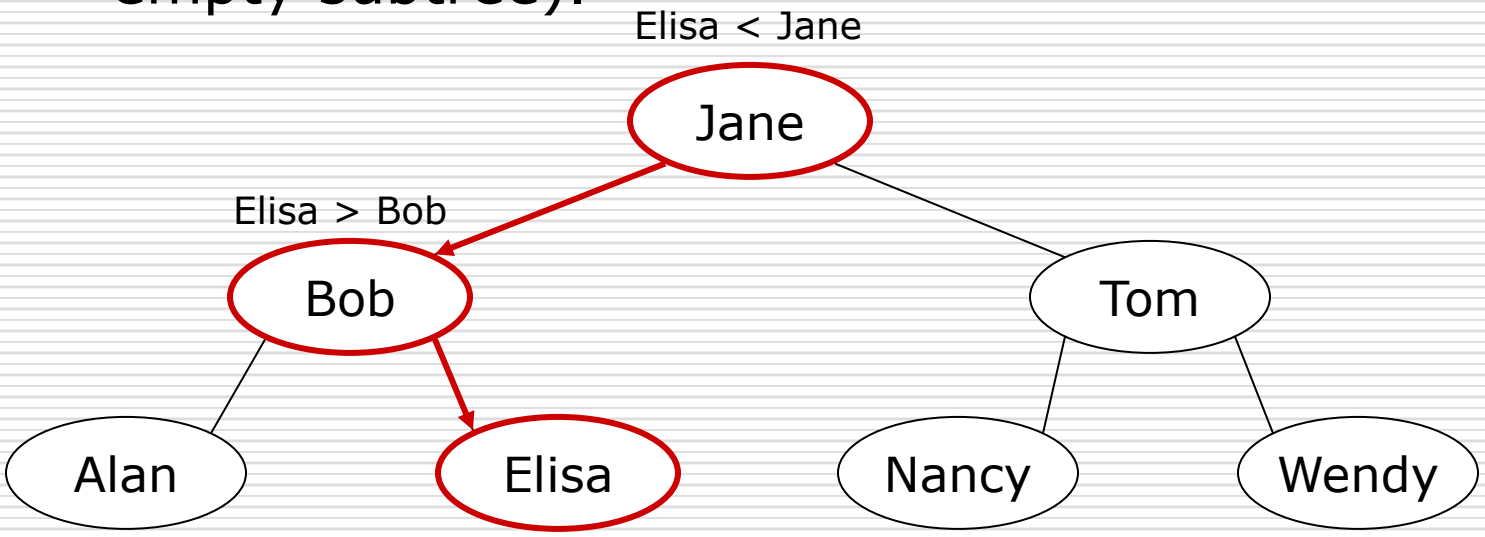
ADT Binary Search Tree: **Traversal**

- Traversals for a binary search tree are the same as the traversals for a binary tree.

- The *inorder* traversal of a binary search tree visits the tree's nodes ***in sorted search-key order***.

The Efficiency of Binary Search Tree Operations (1/7)

- Each operation compares the a specified value *v* to the entries in the nodes along a path through the tree.
  - Start at the root.
  - Terminate at the node that contains *v* (or an empty subtree).



Elisa == Elisa bingo

3 comparisons

The Efficiency of Binary Search Tree Operations (2/7)

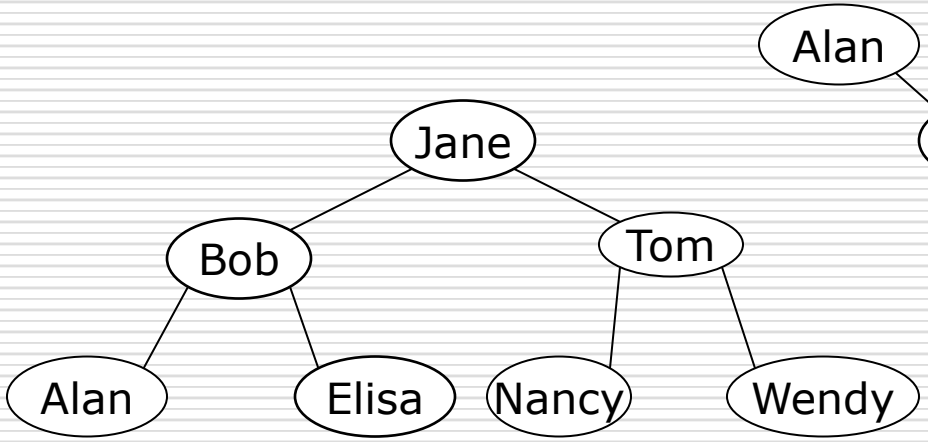
- Thus, the efficiency of each operation is related (proportional) to the number of nodes along the comparison path.

- The maximum number of comparisons is the number of nodes along the longest path from root to a leaf-that is, the tree's height.

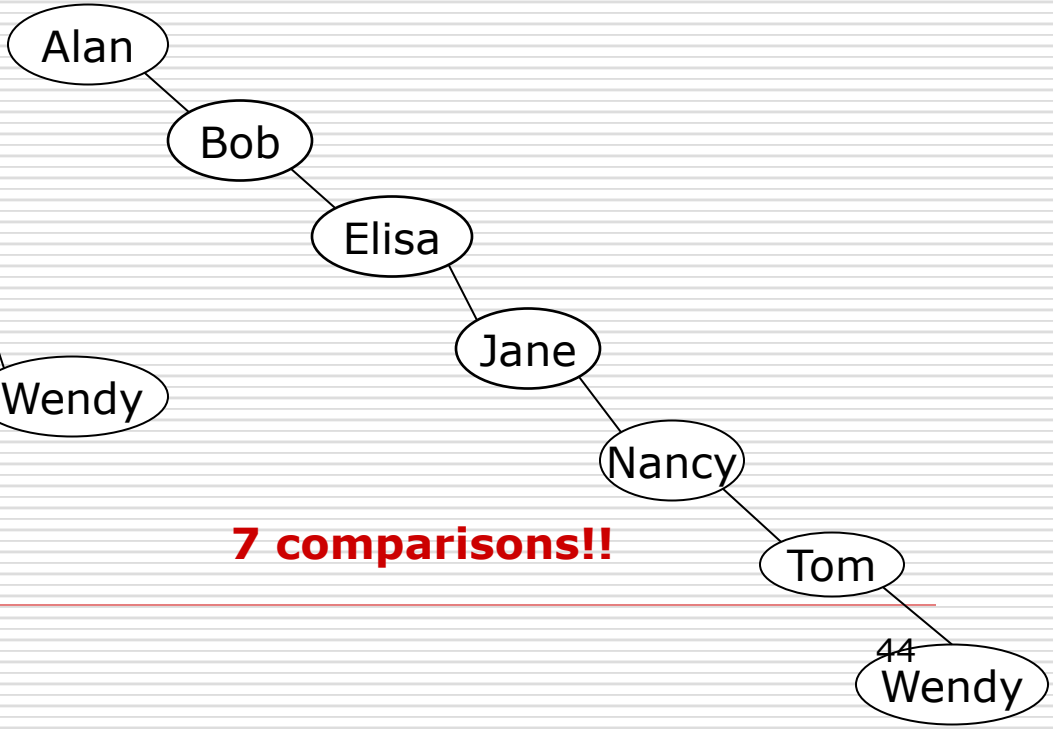
The Efficiency of Binary Search Tree Operations (3/7)

- However, several different binary search trees are possible for the same data!!

- To locate "Wendy":



3 comparisons



7 comparisons!!

The Efficiency of Binary Search Tree Operations (4/7)

- The order in which insertion and deletion operations are performed on a binary search tree affects its height.
- Now, we try to analyze the maximum and minimum heights of a binary search tree with *n* nodes.
- **The maximum height of a binary tree with *n* nodes is *n*.**
  - By giving each internal node exactly one child.

The Efficiency of Binary Search Tree Operations (5/7)

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

The Efficiency of Binary Search Tree Operations (6/7)

- It can be proven mathematically that if the insertion and deletion operations **occur in a random order**, the height of the binary search tree will be quite close to $\log_2 n$.
  - In many applications, it is realistic to expect the operations to occur in random order.

The Efficiency of Binary Search Tree Operations (7/7)

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*)

The Nodes in a Binary Tree

- The first step is to choose a data structure to represent nodes.
- Since each node must contain both data and "pointers" to the node's children, it is natural to make each node an object!!
  - Thus, we will use a C++ class to define the nodes in the tree.
- If we place these nodes in an **array**, the "pointers" in the nodes are array indices.
- If the nodes are a part of a **linked chain**, we use C++ **pointers** to link them together.

An **Array**-Based Representation (1/3)

- An array-based implementation of a tree uses an array of nodes, so a class of such trees could have the following data members:
  - `TreeNode<ItemType> tree[MAX_NODES]; // array of tree nodes`
  - `int root; // index of root`
  - `int free; // index of free list`

- root is an index to the tree's root node within the array tree.
  - If the tree is empty, `root` is -1.

- As the tree changes due to insertions and removals, its nodes may not be in contiguous elements of the array!!
  - You have to establish a collection of available nodes, which is called a free list.
  - `free` is the index to the first node in the free list.

An Array-Based Representation (2/3)

- Let's name our class of nodes **`TreeNode`**.
  - Here, we focus on binary tree!!

```
template<class ItemType>
class TreeNode
{
private:
  ItemType item;       // data portion
  int leftChild;       // index to left child
  int rightChild;      // index to right child
public:
  TreeNode();
  TreeNode(const ItemType &nodeItem, int left, int right);
  ...
};
```

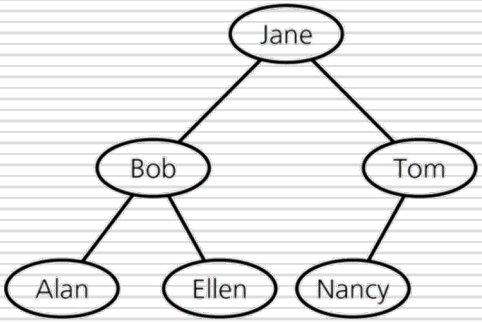
-1 if a node has no left/right child

The class **`TreeNode`** for an array-based implement of the ADT binary tree.

An Array-Based Representation (3/3)

- **`free`** is the index of the first node in the free list, but the next available node is not necessarily at index **`free + 1`**!!
- We link the available nodes by making the **`rightChild`** member of each node be the index of the next node in the free list.

(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 **Link**-Based Representation (1/3)

- The most common way of implementing a tree is to use C++ pointers to link the nodes in the tree.

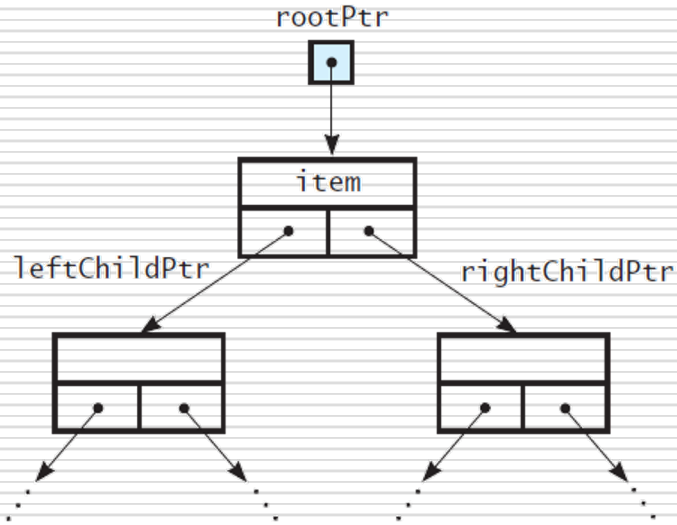
```
...
template<class ItemType>
class BinaryNode
{
private:
   ItemType item;                          // data portion
   BinaryNode<ItemType>* leftChildPtr; // pointer to left child
   BinaryNode<ItemType>* rightChildPtr;// pointer to right child

public:
  ...
```

A Link-Based Representation (2/3)

```
  void setItem(const ItemType& anItem);
  ItemType getItem() const;

  bool isLeaf() const;

  BinaryNode<ItemType>* getLeftChildPtr() const;
  BinaryNode<ItemType>* getRightChildPtr() const;

  void setLeftChildPtr(BinaryNode<ItemType>* leftPtr);
  void setRightChildPtr(BinaryNode<ItemType>* rightPtr);
}; // end BinaryNode
...
```

A Link-Based Representation (3/3)

- A class of link-based binary trees will declare one data member:
  - A pointer **`rootPtr`** points to the tree's root.
  - If the tree is empty, **`rootPtr`** contains **`nullptr`**.
  - **`rootPtr->getLeftChildPtr()`** points to the root of the left subtree.
  - **`rootPtr->getRightChildPtr()`** points to the root of the right subtree.
  - If either of these subtrees is empty, the pointer to it would be **`nullptr`**.



An Interface Template for the ADT Binary Tree

```
...
template<class ItemType>
class BinaryTreeInterface
{
public:
   virtual bool isEmpty() const = 0;
   virtual int getHeight() const = 0;
   virtual int getNumberOfNodes() const = 0;
   virtual ItemType getRootData() const = 0;
   virtual void setRootData(const ItemType& newData) = 0;
   virtual bool add(const ItemType& newData) = 0;
   virtual bool remove(const ItemType& data) = 0;
   virtual void clear() = 0;
   virtual ItemType getEntry(const ItemType& anEntry) const
                    throw(NotFoundException) = 0;
   virtual bool contains(const ItemType& anEntry) const = 0;
   virtual void preorderTraverse(void visit(ItemType&)) const = 0;
   virtual void inorderTraverse(void visit(ItemType&)) const = 0;
   virtual void postorderTraverse(void visit(ItemType&)) const = 0;
}; // end BinaryTreeInterface
...
```

A **Link**-Based Implementation of the ADT Binary Tree (1/13)

- The header file of the class **`BinaryNodeTree`**:
  - The protected methods – called by the public methods to perform their operations recursively.
  - These methods require pointers (implementation details) as arguments. As such, they should not be public and available to clients of the class.

```
...
template<class ItemType>
class BinaryNodeTree : public BinaryTreeInterface<ItemType>
{
private:
   BinaryNode<ItemType>* rootPtr;

protected:
   int getHeightHelper(BinaryNode<ItemType>* subTreePtr) const;
   int getNumberOfNodesHelper(BinaryNode<ItemType>* subTreePtr) const;
...
```

A Link-Based Implementation of the ADT Binary Tree (2/13)

- The public section declares constructors, allowing a client to define binary trees in a variety of circumstances:
  - That is empty
  - From data for its root, which is its only node
  - From data for its root and form its two subtrees

```
public:
   //-----------------------------------------------------------
   // Constructor and Destructor Section.
   //-----------------------------------------------------------
   BinaryNodeTree();
   BinaryNodeTree(const ItemType& rootItem);
   BinaryNodeTree(const ItemType& rootItem,
                  const BinaryNodeTree<ItemType>* leftTreePtr,
                  const BinaryNodeTree<ItemType>* rightTreePtr);
   BinaryNodeTree(const BinaryNodeTree<ItemType>& tree);
   virtual ~BinaryNodeTree();
```

copy constructor

A Link-Based Implementation of the ADT Binary Tree (3/13)

- For example, the following statements invoke these three constructors:
  - `BinaryNodeTree<string> tree1;`
  - `BinaryNodeTree<string>* tree2Ptr = new BinaryNodeTree<string>("A");`
  - `BinaryNodeTree<string>* tree3Ptr = new BinaryNodeTree<string>("B");`
  - `BinaryNodeTree<string>* tree4Ptr = new BinaryNodeTree<string>("C", tree2Ptr, tree3Ptr);`

- **`tree1`** is an empty binary tree;
- **`tree2Ptr`** and **`tree3Ptr`** each point to binary trees that have only a root node.
- **`tree4Ptr`** points to a binary tree whose root contains "C" and has subtrees pointed to by **`tree2Ptr`** and **`tree3Ptr`**.

A Link-Based Implementation of the ADT Binary Tree (4/13)

- The public methods inherited from **`BinaryTreeInterface`.**

```
//------------------------------------------------------------
// Public BinaryTreeInterface Methods Section.
//------------------------------------------------------------
bool isEmpty() const;
int getHeight() const;
int getNumberOfNodes() const;
ItemType getRootData() const throw(PrecondViolatedExcep);
void setRootData(const ItemType& newData);
bool add(const ItemType& newData); // Adds a node
bool remove(const ItemType& data); // Removes a node
void clear();
ItemType getEntry(const ItemType& anEntry) const throw(NotFoundException);
bool contains(const ItemType& anEntry) const;
```

A Link-Based Implementation of the ADT Binary Tree (5/13)

- The traversal methods inherited from **`BinaryTreeInterface`**.

```
   //------------------------------------------------------------
   // Public Traversals Section.
   //------------------------------------------------------------
   void preorderTraverse(void visit(ItemType&)) const;
   void inorderTraverse(void visit(ItemType&)) const;
   void postorderTraverse(void visit(ItemType&)) const;

   //------------------------------------------------------------
   // Overloaded Operator Section.
   //------------------------------------------------------------
   BinaryNodeTree& operator=(const BinaryNodeTree& rightHandSide);
}; // end BinaryNodeTree
...
```

A Link-Based Implementation of the ADT Binary Tree (6/13)

- Here, we examine the most significant methods:
- The constructors:

```
template<class ItemType>
BinaryNodeTree<ItemType>::BinaryNodeTree() : rootPtr(nullptr)
{
}  // end default constructor

template<class ItemType>
BinaryNodeTree<ItemType>::BinaryNodeTree(const ItemType& rootItem)
{
   rootPtr = new BinaryNode<ItemType>(rootItem, nullptr, nullptr);
}  // end constructor

template<class ItemType>
BinaryNodeTree<ItemType>::BinaryNodeTree(const ItemType& rootItem,
                                         const BinaryNodeTree<ItemType>* leftTreePtr,
                                         const BinaryNodeTree<ItemType>* rightTreePtr)
{
    rootPtr = new BinaryNode<ItemType>(rootItem, copyTree(leftTreePtr->rootPtr),
                                                 copyTree(rightTreePtr->rootPtr));
}  // end constructor
```

A Link-Based Implementation of the ADT Binary Tree (7/13)

- The protected method **copyTree** uses a recursive preorder traversal to copy each node in the tree.

```
template<class ItemType>
BinaryNode<ItemType>* BinaryNodeTree<ItemType>::
          copyTree(const BinaryNode<ItemType>* treePtr) const
{
   BinaryNode<ItemType>* newTreePtr = nullptr;

   // Copy tree nodes during a preorder traversal
   if (treePtr != nullptr)
   {  // Copy node
      newTreePtr = new BinaryNode<ItemType>(treePtr->getItem(), nullptr, nullptr);
      newTreePtr->setLeftChildPtr(copyTree(treePtr->getLeftChildPtr()));
      newTreePtr->setRightChildPtr(copyTree(treePtr->getRightChildPtr()));
   }  // end if

   return newTreePtr;
}  // end copyTree
```

A Link-Based Implementation of the ADT Binary Tree (8/13)

- The copy constructor then looks like this:

```
template<class ItemType>
BinaryNodeTree<ItemType>::BinaryNodeTree(const BinaryNodeTree<ItemType>& tree)
{
   rootPtr = copyTree(tree.rootPtr);
}  // end copy constructor
```

A Link-Based Implementation of the ADT Binary Tree (9/13)

- The destructor calls **`destroyTree(rootPtr)`** to delete each node in the tree in a recursive postorder manner.

```
template<class ItemType>
void BinaryNodeTree<ItemType>::destroyTree(BinaryNode<ItemType>* subTreePtr)
{
   if (subTreePtr != nullptr)
   {
      destroyTree (subTreePtr->getLeftChildPtr());
      destroyTree (subTreePtr->getRightChildPtr());
      delete subTreePtr;
   }
}  // end destroyTree
```

- The destructor then only needs to make the call **`destroyTree(rootPtr)`**.

```
template<class ItemType>
BinaryNodeTree<ItemType>::~BinaryNodeTree()
{
   destroyTree(rootPtr);
}  // end destructor
```

A Link-Based Implementation of the ADT Binary Tree (10/13)

- Another recursive example:

```
template<class ItemType>
int BinaryNodeTree<ItemType>::getHeightHelper(BinaryNode<ItemType>* subTreePtr) const
{
   if (subTreePtr == nullptr)
      return 0;
   else
      return 1 + max(getHeightHelper(subTreePtr->getLeftChildPtr()),
                     getHeightHelper(subTreePtr->getRightChildPtr()));
}  // end getHeightHelper

template<class ItemType>
int BinaryNodeTree<ItemType>::getHeight() const
{
   return getHeightHelper(rootPtr);
}  // end getHeight
```

A Link-Based Implementation of the ADT Binary Tree (11/13)

- The method **Add**:
  - The specification of the public method add does not indicate where the new node should be in the tree.
    - We have flexibility in how we define the method.

- Let's add the new node so that the resulting tree is balanced.

```
template<class ItemType>
bool BinaryNodeTree<ItemType>::add(const ItemType& newData)
{
   BinaryNode<ItemType>* newNodePtr = new BinaryNode<ItemType>(newData);
   rootPtr = balancedAdd(rootPtr, newNodePtr);

   return true;
}  // end add
```

A Link-Based Implementation of the ADT Binary Tree (12/13)

```
template<class ItemType>
BinaryNode<ItemType>* BinaryNodeTree<ItemType>::balancedAdd(
                                BinaryNode<ItemType>* subTreePtr,
                                BinaryNode<ItemType>* newNodePtr)
{
   if (subTreePtr == nullptr)
      return newNodePtr;

   else
   {
      BinaryNode<ItemType>* leftPtr = subTreePtr->getLeftChildPtr();
      BinaryNode<ItemType>* rightPtr = subTreePtr->getRightChildPtr();

      if (getHeightHelper(leftPtr) > getHeightHelper(rightPtr))
      {
         rightPtr = balancedAdd(rightPtr, newNodePtr);
         subTreePtr->setRightChildPtr(rightPtr);
      } else {
         leftPtr = balancedAdd(leftPtr, newNodePtr);
         subTreePtr->setLeftChildPtr(leftPtr);
      }

      return subTreePtr;
   }
}
```

A Link-Based Implementation of the ADT Binary Tree (13/13)

- The traversals:
  - Since the traversal are recursive (using pointers – implementation details), the public traversal methods each calls a protected method that performs the actual recursion.

function as a parameter

```
template<class ItemType>
void BinaryNodeTree<ItemType>::inorder(
                               void visit(ItemType&),
                               BinaryNode<ItemType>* treePtr) const
{
   if (treePtr != nullptr)
   {
      inorder(visit, treePtr->getLeftChildPtr());
      ItemType theItem = treePtr->getItem();
      visit(theItem);
      // treePtr->setItem(theItem);
      inorder(visit, treePtr->getRightChildPtr());
    }
}

template<class ItemType>
void BinaryNodeTree<ItemType>::inorderTraverse(void visit(ItemType&)) const
{
   inorder(visit,rootPtr);
}
```

A **Link**-Based Implementation of the ADT Binary **Search** Tree (1/20)

- Since a binary search tree is a binary tree, its implementation can use the same node objects as for a binary-tree implementation.
  - We will use the class **`BinaryNode`**.
  - We assume that the data items in the binary search tree are unique!!

- The recursive search algorithm is the basis of the *insertion*, *removal*, and *retrieval* operations on a binary search tree.

A Link-Based Implementation of the ADT Binary Search Tree (2/20)

- Adding a new entry:
  - You insert a new entry into a binary search tree in the same place that the search algorithm would look for it!!
  - Because searching for an entry that is not in the binary search tree always ends at an empty subtree … YOU ALWAYS INSERT A NEW ITEM AS A NEW LEAF!!
  - Adding a leaf requires only a change of the appropriate pointer in the parent.

A Link-Based Implementation of the ADT Binary Search Tree (3/20)

```
template<class ItemType>
int BinarySearchTree<ItemType>::add(const ItemType& newData)
{
   BinaryNode<ItemType>* newNodePtr = new BinaryNode<ItemType>(newData);
   rootPtr = insertInorder(rootPtr, newNodePtr);

   return true;
}
```

```
// pseudocode of insertInorder
insertInorder(subTreePtr: BinaryNodePointer,
                    newNodePtr: BinaryNodePointer): BinaryNodePointer

   if (subTreePtr is nulptr)
      return newNodePtr
   else if (subTreePtr->getItem() > newNodePtr->getItem()) {
      tempPtr = insertInorder(subTreePtr->getLeftChildPtr(), newNodePtr)
      subTreePtr->setLeftChildPtr(tempPtr)
    } else {
      tempPtr = insertInorder(subTreePtr->getRightChildPtr(), newNodePtr)
      subTreePtr->setRightChildPtr(tempPtr)
    }

    return subTreePtr
```

A Link-Based Implementation of the ADT Binary Search Tree (4/20)

- Removing an entry:
  - Is more complicated than adding.
  - First, you use the *search algorithm* to locate the specified item.
    - If it is found, you must remove it from the tree.
  - Assuming that **`removeValue`** locates the target in a particular node *N*.

A Link-Based Implementation of the ADT Binary Search Tree (5/20)

```
removeValue(subTreePtr: BinaryNodePointer, target: ItemType,
                                                success: Boolean&): BinaryNodePointer
if(subTreePtr == nullptr)
{
   success = false
   return nullptr

} else if (subTreePtr->getItem() == target) {
   subTreePtr = removeNode(subTreePtr)
   success = true
   return subTreePtr

} else if (subTreePtr->getItem() > target) {
   tempPtr = removeValue(subTreePtr->getLeftChildPtr(), target, success)
   subTreePtr->setLeftChildPtr(tempPtr)
   return subTreePtr

} else {
   tempPtr = removeValue(subTreePtr->getRightChildPtr(), target, success)
   subTreePtr->setRightChildPtr(tempPtr)
   return subTreePtr
}
```

A Link-Based Implementation of the ADT Binary Search Tree (6/20)

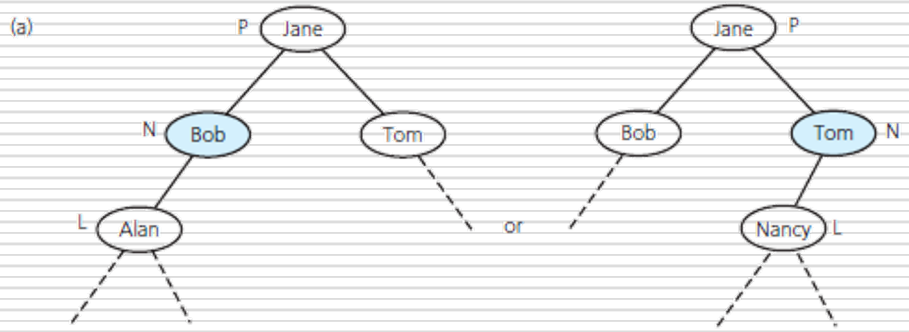
- The essential task is to remove the target *N* from the tree (**`removeNode`**).
  - There are three cases to consider:
    - *N* is a leaf
    - *N* has only one child
    - *N* has two children

- Case 1 is the easiest. You need only set **the pointer in its parent to `nullptr`.**

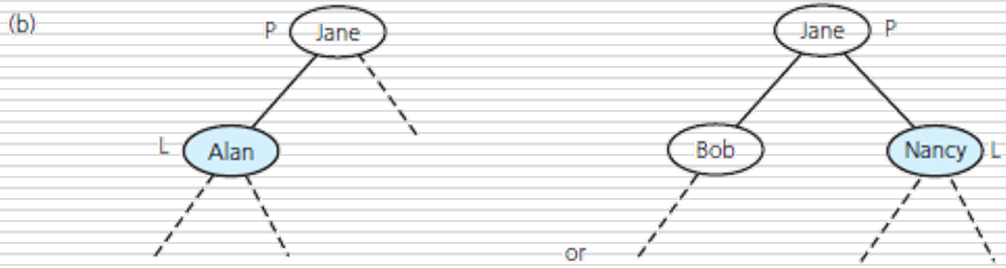
A Link-Based Implementation of the ADT Binary Search Tree (7/20)

- Case 2 is a bit more involved.
  - If *N* has only one child:
    - *N* has only a left child
    - *N* has only a right child
- The two possibilities are symmetrical, so we illustrate the solution for a left child.

A Link-Based Implementation of the ADT Binary Search Tree (8/20)



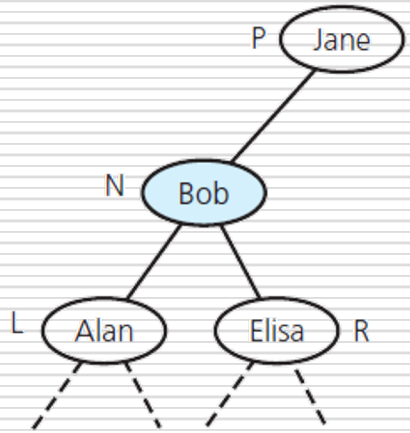
- Let *L* take the place of *N* as one of *P*'s children…



- Does this adoption preserve the binary search tree property?
  - Yes, the binary search tree property is preserved!!

A Link-Based Implementation of the ADT Binary Search Tree (9/20)

- Case 3 is the most difficult case.



- *N*'s parent has room for only one *N*'s children as a replacement for *N*!!

A Link-Based Implementation of the ADT Binary Search Tree (10/20)

- In fact, you can find another node that is easier to delete and delete it instead of *N*.
  - This strategy may sound like cheating…
  - But remember that the client expects only a certain entry to be removed from the ADT.
  - It has no right, because of the WALL between the program and the ADT implementation, to expect a particular node in the tree to be deleted.

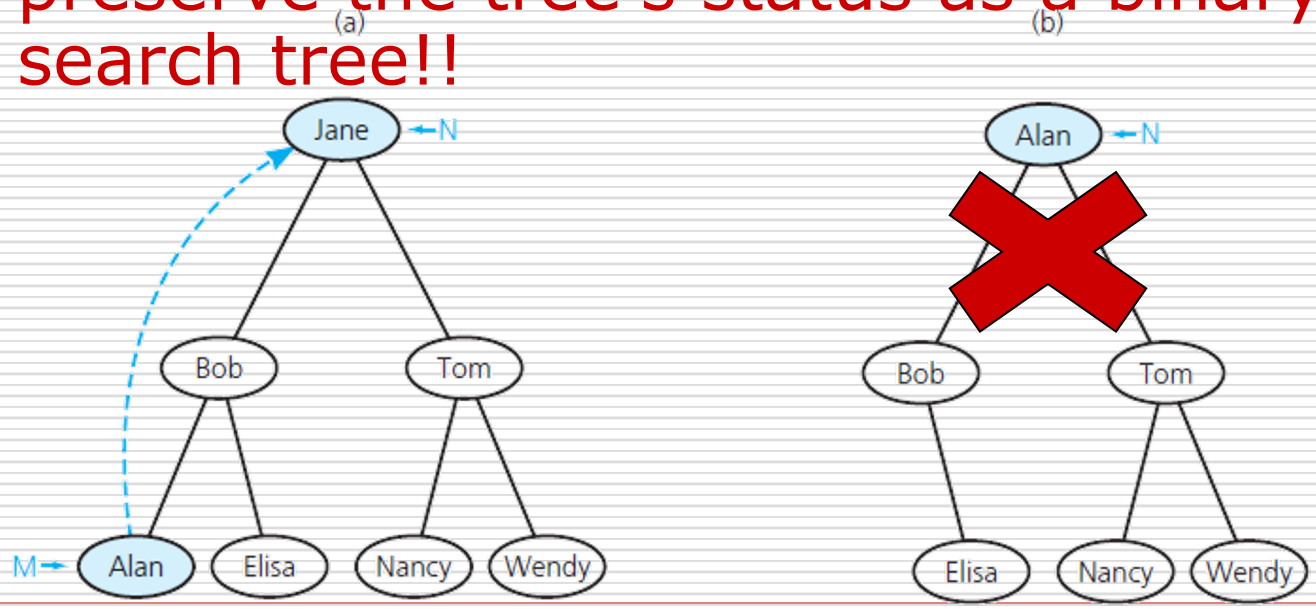
A Link-Based Implementation of the ADT Binary Search Tree (11/20)

- Removing strategy:
  - Locate another node *M* that is easier to remove from the tree than the node *N*.
  - Copy the item that is in *M* to *N*, thus effectively removing from the tree the item originally in *N*.
  - Remove the node *M* from the tree.

- But … what kind of node *M* is easier to remove??

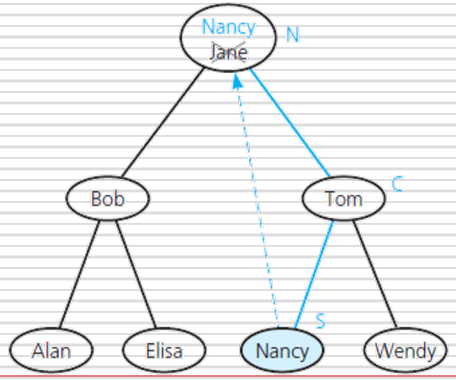
A Link-Based Implementation of the ADT Binary Search Tree (12/20)

- *M* should have a **single child** or **no children**.
  - After replacing *N* by *M*, you must also preserve the tree's status as a binary search tree!!



A Link-Based Implementation of the ADT Binary Search Tree (13/20)

- There are two suitable possibilities for the replacement.
  - You can copy into *N* the item that is immediately before/after *N* in the sorted order.
  - Suppose that we decide to use the node *y* whose entry comes immediately after *N*'s entry *x*.
    - This entry is called *x*'s inorder successor.



A Link-Based Implementation of the ADT Binary Search Tree (14/20)

- How can you locate this node?
- Because *N* has two children, the inorder successor is in the leftmost node of *N*'s right subtree.
  - You follow *N*'s right-child pointer to its right child *C* (must be present).
  - Then, you descend the tree rooted at *C* by taking left branches at each node … until you encounter a node *S* with no left child.
  - Copy the item in *S* into *N*.
  - REMOVE *S*!!
    - Because *S* has no left child you can remove *S* from the tree as one of the two easy cases.

A Link-Based Implementation of the ADT Binary Search Tree (15/20)

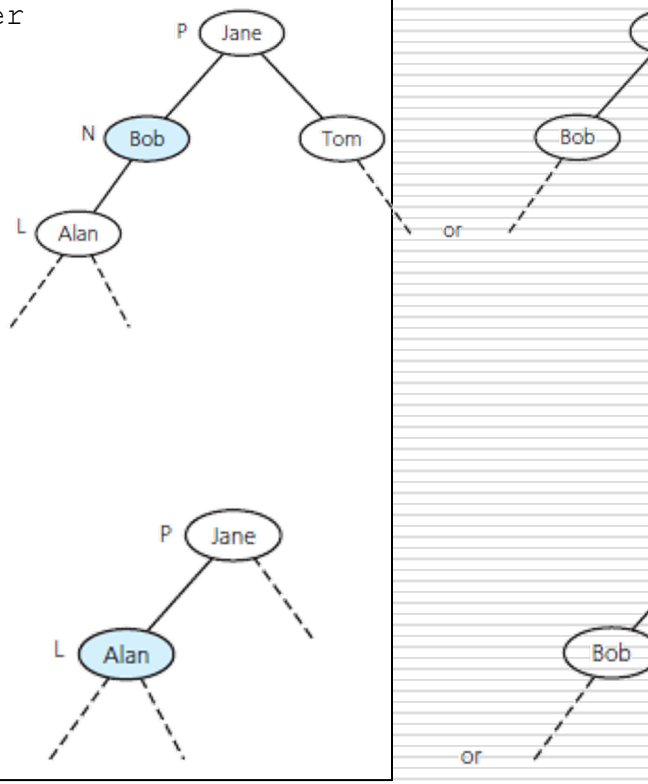
- Pseudocode of **removeNode**

```
removeNode(N:: BinaryNode)

   if (N is a leaf)
      Remove N from the tree
   else if (N has only one child C)
   {
      if (N was a left child of its parent P)
          Make C the left child of P
      else
          Make C the right child of P
   } else {
      Find S, the node that contains N's inorder successor
      Copy the item from node S into node N
      Remove S from the tree by using the previous technique for a leaf or
             a node with one child
   }
```

A Link-Based Implementation of the ADT Binary Search Tree (16/20)

- Pseudo code:

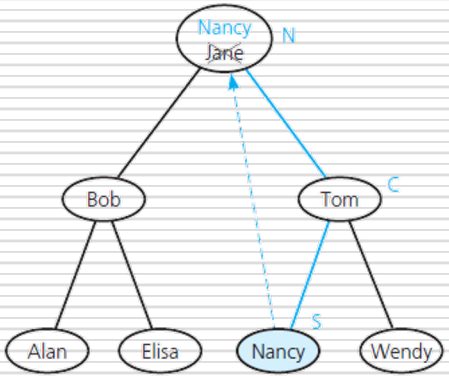
```
removeNode(nodePtr: BinaryNodePointer): BinaryNodePointer

   if (N is a leaf)
   {
      delete nodePtr
      nodePtr = nullptr
      return nodePtr
   }
   else if (N has only one child C)
   {
      if (C is the left child)
          nodeToConnectPtr = nodePtr->getLeftChildPtr()
      else
          nodeToConnectPtr = nodePtr->getRightChildPtr()

      delete nodePtr
      nodePtr = nullptr
      return nodeToConnectPtr

   }
```



A Link-Based Implementation of the ADT Binary Search Tree (17/20)

- Pseudo code:

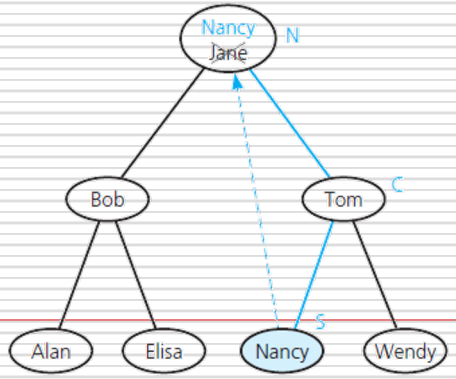
```
else
{
   tempPtr = removeLeftmostNode(nodePtr->getRightChildPtr(), newNodeValue)
   nodePtr->setRightChildPtr(tempPtr)
   nodePtr->setItem(newNodeValue)
   return nodePtr
}
```



newNodeValue : Nancy

A Link-Based Implementation of the ADT Binary Search Tree (18/20)

```
removeLeftmostNode(nodePtr:: BinaryNodePointer,
                   inorderSuccessor: ItemType&): BinaryNodePointer

   if (nodePtr->getLeftChildPtr() == nullptr)
   {
      inorderSuccessor = nodePtr->getItem()
      return removeNode(nodePtr)
   }
   else
   {
      tempPtr = removeLeftmostNode(nodePtr->getLeftChildPtr(),inorderSuccesssor)
      nodePtr->setLeftChildPtr(tempPtr)
      return nodePtr
   }
```



A Link-Based Implementation of the ADT Binary Search Tree (19/20)

- The public method remove:

```
remove (target: ItemType): boolean

   success = false
   rootPtr = removeValue(rootPtr, target, success)
   return success
```

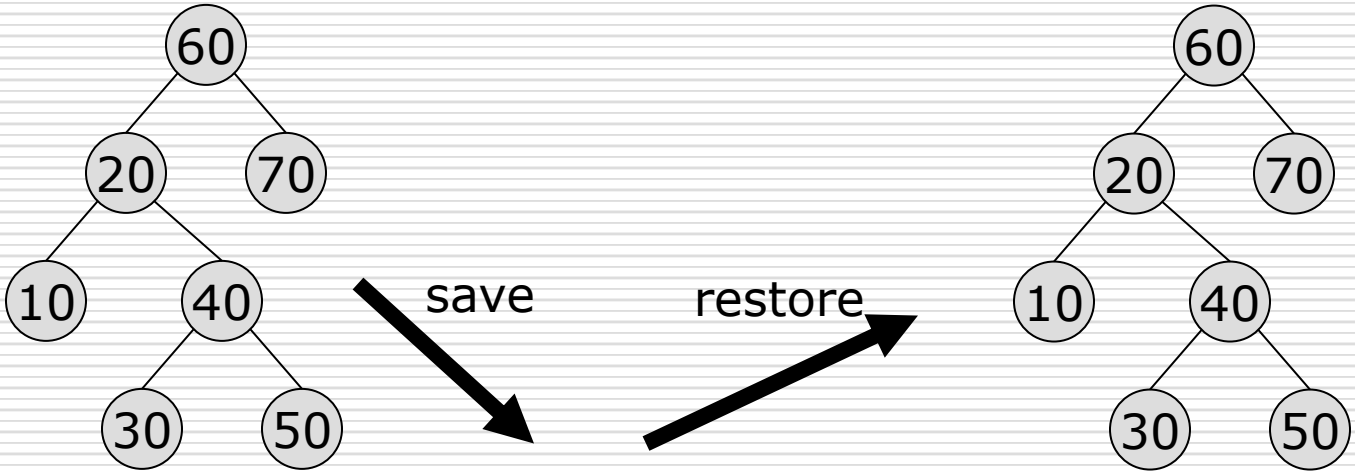
A Link-Based Implementation of the ADT Binary Search Tree (20/20)

- Retrieving an entry – **`getEntry`**:
  - Call **`findNode`** to recursively checks whether the desired target is in a binary search tree.
  - It checks the return value, and returns the desired target or throws an exception

```
findNode (subTreePtr: BinaryNodePointer, target: itemType): BinaryNodePointer

   if(subTreePtr == nullPtr)
      return nullptr
   else if (subTreePtr->getItem()== target)
      return subTreePtr;
   else if (subTreePtr->getItem() >target)
      return findNode(subTreePtr->getLeftChildPtr(), target)
   else
      return findNode(subTreePtr->getRightChildPtr(), target)
```

Saving a Binary Search Tree in a File (1/6)

- Saving a binary search tree and then restoring it to its original shape:
  - Uses **preorder traversal** to save the tree to a file.



Preorder: 60 20 10 40 30 50 70

Saving a Binary Search Tree in a File (2/6)

- Saving a binary search tree and then restoring it to a **balanced shape**:
  - Uses inorder traversal to save the tree to a file.
    - To make the data sorted.
  - To restore, need the number of nodes in the tree.
    - Can determine the middle item and, in turn, the number of nodes in the left and right subtrees of the tree's root.

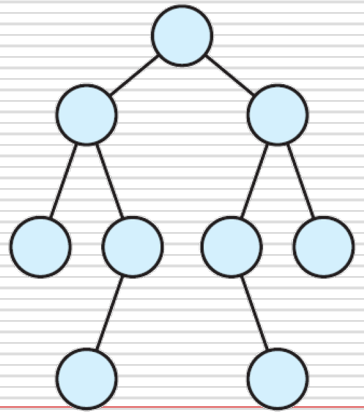
Saving a Binary Search Tree in a File (3/6)

- Restoring a full binary search tree:
  - You can use the following recursive algorithm to create a full binary search tree with *n* nodes (provided you either know or can determine *n* beforehand).

```
readFullTree (treePtr: BinaryNodePointer, n: integer): BinaryNodePointer

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

      // Construct the left subtree
      leftPtr = readFullTree(treePtr->getLeftChildPtr(), n / 2)
      treePtr->setLeftChildPtr(leftPtr)

      // Get the root
      rootItem = next item from file
      treePtr->setItem(rootItem)

      // Construct the right subtree
      rightPtr = readFullTree(treePtr->getRightChildPtr(), n / 2)
      treePtr->setRightChildPtr(rightPtr)

      return treePtr
   }
   else
      return nullptr
```

Saving a Binary Search Tree in a File (4/6)

- If the tree to be restored is not full:
  - The first thing that comes to mind is that the resorted tree should be complete.
  - But … you care only about minimizing the height of the restored tree. It does not matter where the nodes on the last level go.



Saving a Binary Search Tree in a File (5/6)

- The method **`readFullTree`** is correct even if the tree is not full!!

- However, you have to be a bit careful when computing the sizes of the left and right subtrees of the tree's root.
  - If *n* is odd, both subtrees are of size *n*/2.
  - If *n* is even, you have to deal with the fact that one subtree will have one more node than the other.

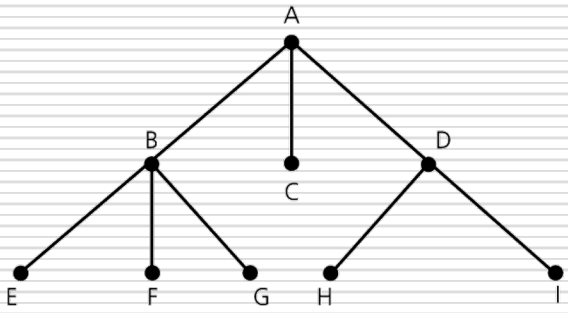
  - In this case, we put the extra node in the left subtree.

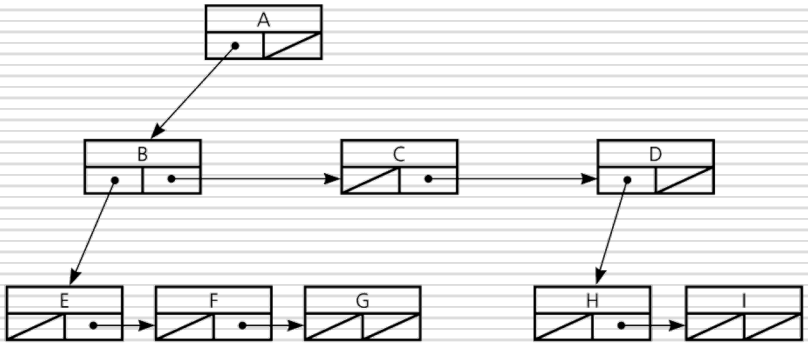
Saving a Binary Search Tree in a File (6/6)

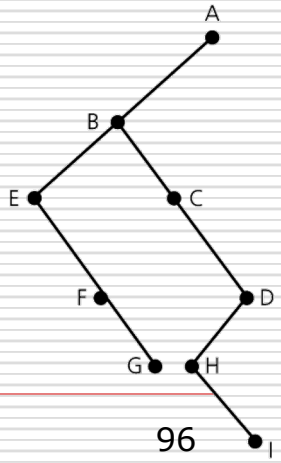
```
readTree(treePtr: BinaryNodePointer, n: integer): BinaryNodePointer

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

      // Construct the left subtree
      leftPtr = readTree(treePtr->getLeftChildPtr(), n / 2)
      treePtr->setLeftChildPtr(leftPtr)

      // Get the root
      rootItem = next item from file
      treePtr->setItem(rootItem)

      // Construct the right subtree
      rightPtr = readTree(treePtr->getRightChildPtr(), (n - 1) / 2)
      treePtr->setRightChildPtr(rightPtr)
      return treePtr
   }
   else
      return nullptr
```

General Trees (1/2)

- Each node can have an arbitrary number of children.
- A binary tree can represent a general tree.
  - Each node has two pointers:
    - The left pointer points to the node's oldest (first) child.
    - The right pointer points to the node's next sibling.







General Trees (2/2)

- An *n*-ary tree is a general tree whose nodes can have no more than *n* children each.
- In implementation, each node points directly to its children.