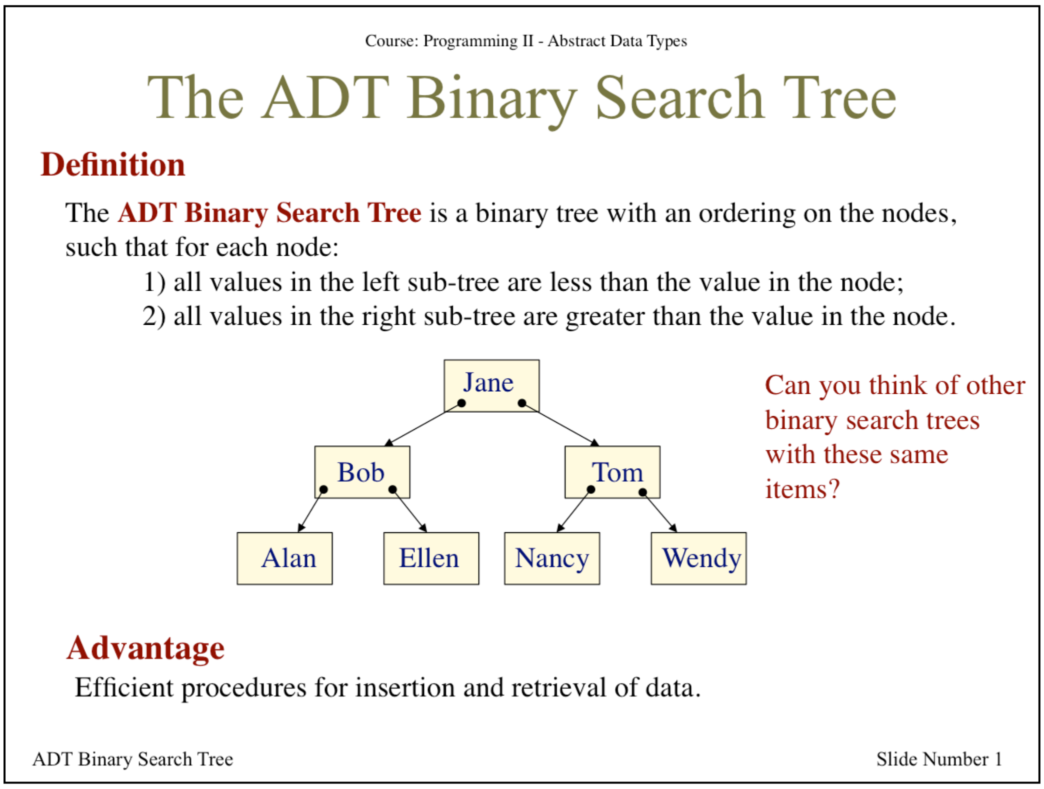
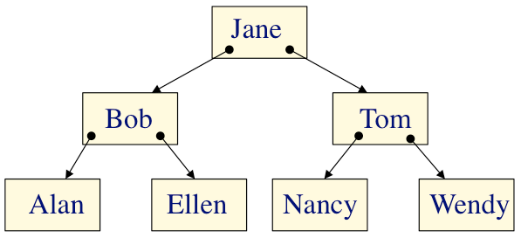

Course: Programming II - Abstract Data Types

The ADT Binary Search Tree

Definition

The **ADT Binary Search Tree** is a binary tree with an ordering on the nodes, such that for each node:

1) all values in the left sub-tree are less than the value in the node;
2) all values in the right sub-tree are greater than the value in the node.



Can you think of other binary search trees with these same items?

Advantage

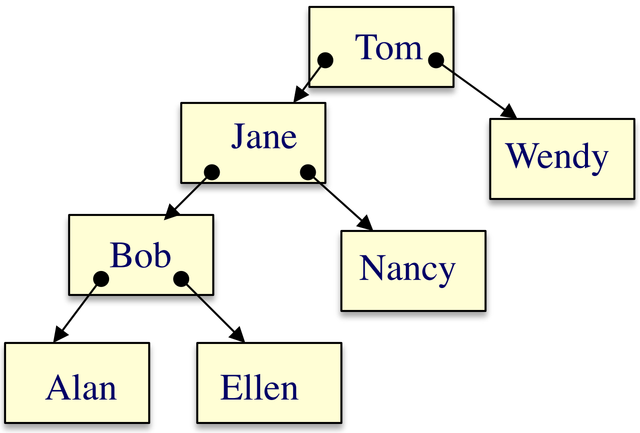
Efficient procedures for insertion and retrieval of data.

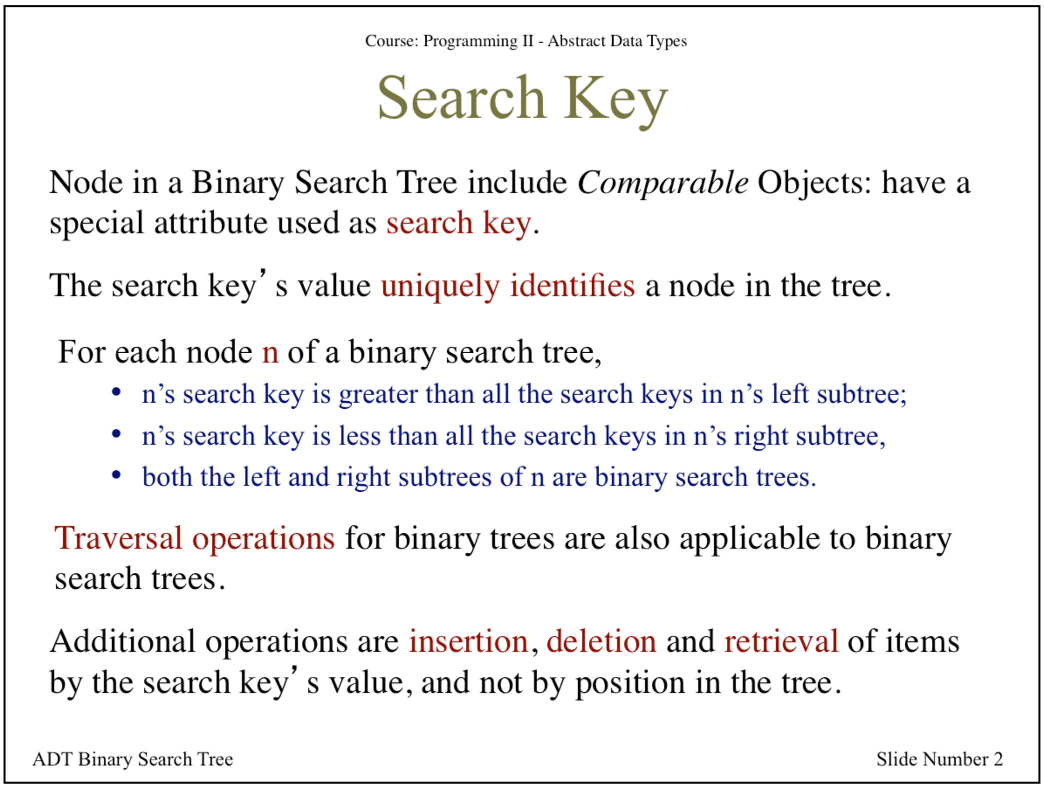
ADT Binary Search Tree Slide Number 1

Searching for a particular item is an operation for which the ADT binary trees are not well suited, as in the worst case scenario we will have to visit all the nodes of the tree. **Searching in a binary tree can therefore be as inefficient as performing a sequential search of an array**.

Binary search trees are particular type of binary trees that corrects this deficiency by organising its own data by value. A binary search tree is a binary tree whose nodes contain Comparable Objects, and are organised as follows. For each node: (a) all the nodes in the left sub-tree are less than the value in the node, and (b) all the nodes in the right sub-tree are greater than the value in the node. This organisation of data enables more efficient search of given data elements, as it allows search to be done by value instead of by position.

The configuration of a binary search tree is not unique. That is we can form several different binary search trees from the same set of data. The tallest binary search tree that can be formed from a given set of data is a ordered linked list! For instance, an alternative search tree to the one given in this slide can be the following. As we will see during this lecture the shape of the tree is a contributing factor to the computational time of search and retrieve operations




Course: Programming II - Abstract Data Types

Search Key

Node in a Binary Search Tree include *Comparable* Objects: have a special attribute used as search key.

The search key' s value uniquely identifies a node in the tree.

For each node n of a binary search tree,

- n's search key is greater than all the search keys in n's left subtree;
- n's search key is less than all the search keys in n's right subtree,
- both the left and right subtrees of n are binary search trees.

Traversal operations for binary trees are also applicable to binary search trees.

Additional operations are insertion, deletion and retrieval of items by the search key' s value, and not by position in the tree.

ADT Binary Search Tree Slide Number 2

A binary search tree is often used in situations where the data stored in the tree contain more than one field (attribute). For example, each item in the tree might contain the name of a person, his/her ID number, his/her address, etc. Often one of these fields is used as the sort key among the items in the tree. For instance, the person' s ID, which is unique for each person, can be used as the information by which the nodes in the tree are sorted. This information is called "search key", as we have already seen in the case of ordered lists.

The search task would be, for instance, the operation of searching for a person whose ID number is equal to a given value. Searching is then the operation of finding the particular item in the tree whose key value is equal to the value given as input.

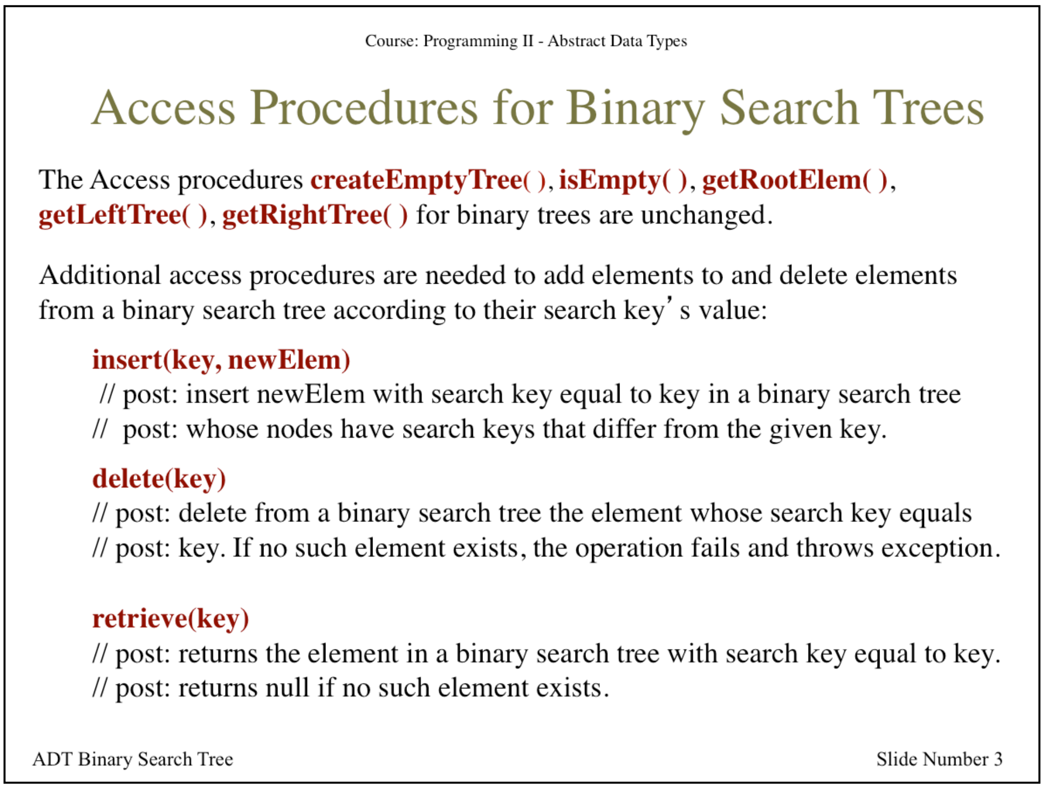
Given this concept of search key, we can say that a binary search tree can be recursively defined as follows. For each node n,

n' s search key is greater than all the search keys in the n' s left subtree;
n' s search key is less than all the search leys in the n' s right subtree,
Both the left and right subtrees of n are binary search trees.

As with all the other ADTs seen so far, a Binary Search Tree has operations that involve inserting to, deleting and retrieving data from a given binary search tree. Similarly to the case of ordered lists. these operations are done by searching on the key value rather than by its position.

The operations of insertion, deletion and retrieval of elements by their search key values are what extend a basic binary tree into a Binary Search Tree.

The three main traversal algorithms we have seen for binary trees in the previous lecture note are also applicable to the ADT Binary Search Tree.



Course: Programming II - Abstract Data Types

Access Procedures for Binary Search Trees

The Access procedures **createEmptyTree( )**, **isEmpty( )**, **getRootElem( )**, **getLeftTree( )**, **getRightTree( )** for binary trees are unchanged.

Additional access procedures are needed to add elements to and delete elements from a binary search tree according to their search key's value:

insert(key, newElem)
// post: insert newElem with search key equal to key in a binary search tree
// post: whose nodes have search keys that differ from the given key.

delete(key)
// post: delete from a binary search tree the element whose search key equals
// post: key. If no such element exists, the operation fails and throws exception.

retrieve(key)
// post: returns the element in a binary search tree with search key equal to key.
// post: returns null if no such element exists.

ADT Binary Search Tree   Slide Number 3

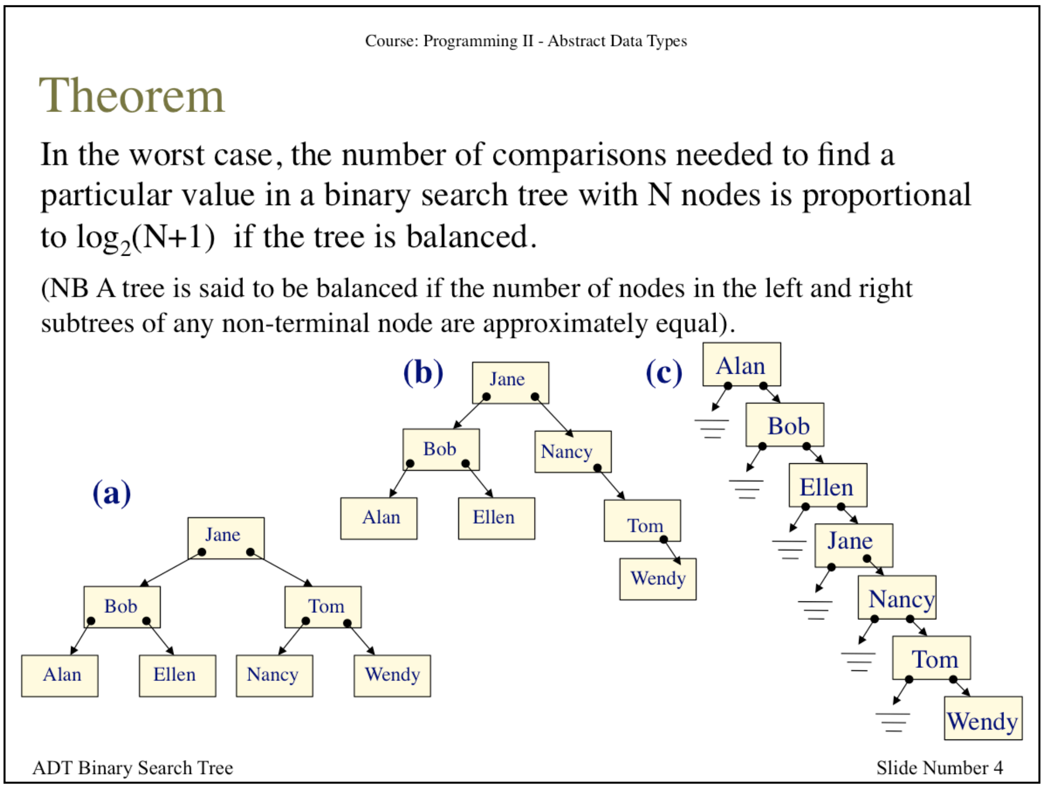
The binary search tree includes some of the basic operations for the ADT's binary tree seen in the previous lecture. In particular, the default constructor, the operations `isEmpty( )`, `getRootElem( )`, `getLeftTree( )`, and `getRightTree( )` that we have defined for a binary tree are also applicable to binary search trees.

Note that the constructor `createTree(root, leftTree, rightTree)` cannot be used in the case of binary search trees. We'll say why later on. Similarly the operation of adding a left sub-tree, or adding a right sub-tree cannot be directly used for binary search trees. This is because arbitrary insertion or deletion of nodes (or sub-trees) would not necessarily guarantee preservation of the ordering. The attachment of two binary search subtrees to a root node would not necessary give a binary search tree. Also the shape of trees so created would not necessary be balanced causing the search in the so generated binary search tree less efficient.

Therefore, the ADT binary search tree includes additional access procedures that allow insertion, deletion, retrieval of elements from/to a binary search tree by their (search key) values and not by their position (as we have seen so far with ADTs like lists, stacks, queues).

In the remainder of this lecture we will see the algorithms for implementing these three new access procedures: insertion, deletion and retrieval. They all make use of a search strategy that searches for an element with key equal to a given search key.

Because a binary search tree is a recursive structure, it is natural to formulate the algorithms for these additional operations in a recursive manner.

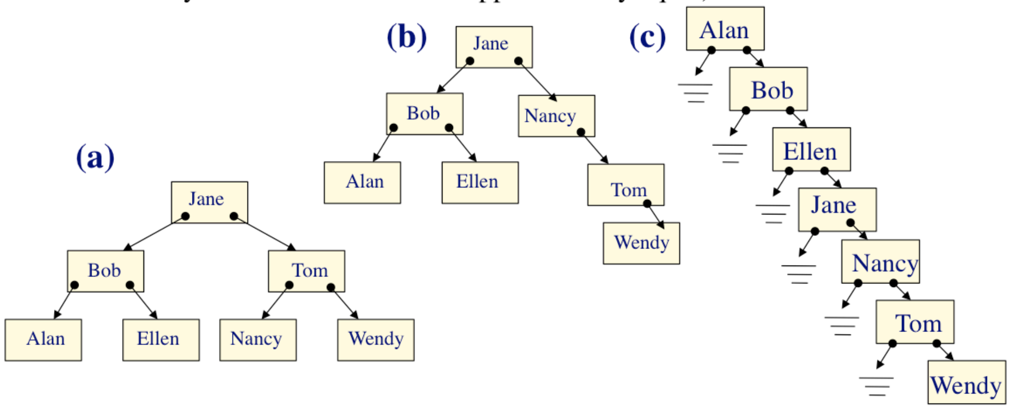


Course: Programming II - Abstract Data Types

Theorem

In the worst case, the number of comparisons needed to find a particular value in a binary search tree with N nodes is proportional to $\log_2(N+1)$ if the tree is balanced.

(NB A tree is said to be balanced if the number of nodes in the left and right subtrees of any non-terminal node are approximately equal).



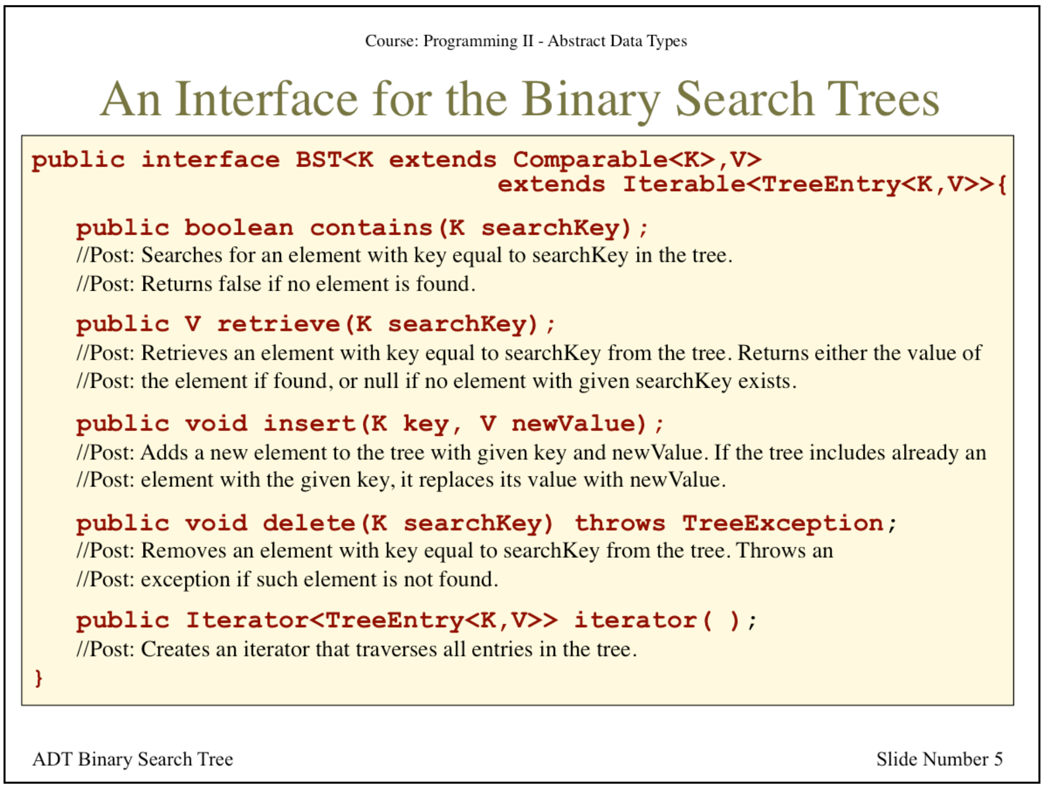
ADT Binary Search Tree

Slide Number 4

As we have already mentioned earlier, many different binary search trees can be created for a given collection of data. In this slide we give examples of three binary search trees, which have the same data but different shapes.

The search strategy can still be applied to each of these three different trees. However, it will work more efficiently on shorter trees and less efficiently on trees with higher number of levels (i.e. taller trees). For example, with the binary search tree (c) the search for Wendy will be of the order O(n) where n is the height of the tree as it will have to inspect every node in the tree before locating Wendy. The search for Wendy in tree (a) would be of the order O(3). So, given a binary search tree of height h, the search algorithm is of the order O(h).

Binary search trees have to be as shorter as possible. The tree (a) is in fact, the shortest binary search tree we can have for the given set of data. Full trees are always such shortest trees. For such trees the search algorithm is of the order $O(\lceil \log_2(n+1\rceil)$ where n is the number of nodes in a complete binary search tree.


Course: Programming II - Abstract Data Types

An Interface for the Binary Search Trees

```
public interface BST<K extends Comparable<K>,V>
                                    extends Iterable<TreeEntry<K,V>>{

   public boolean contains(K searchKey);
   //Post: Searches for an element with key equal to searchKey in the tree.
   //Post: Returns false if no element is found.

   public V retrieve(K searchKey);
   //Post: Retrieves an element with key equal to searchKey from the tree. Returns either the value of
   //Post: the element if found, or null if no element with given searchKey exists.

   public void insert(K key, V newValue);
   //Post: Adds a new element to the tree with given key and newValue. If the tree includes already an
   //Post: element with the given key, it replaces its value with newValue.

   public void delete(K searchKey) throws TreeException;
   //Post: Removes an element with key equal to searchKey from the tree. Throws an
   //Post: exception if such element is not found.

   public Iterator<TreeEntry<K,V>> iterator( );
   //Post: Creates an iterator that traverses all entries in the tree.
}
```

ADT Binary Search Tree   Slide Number 5

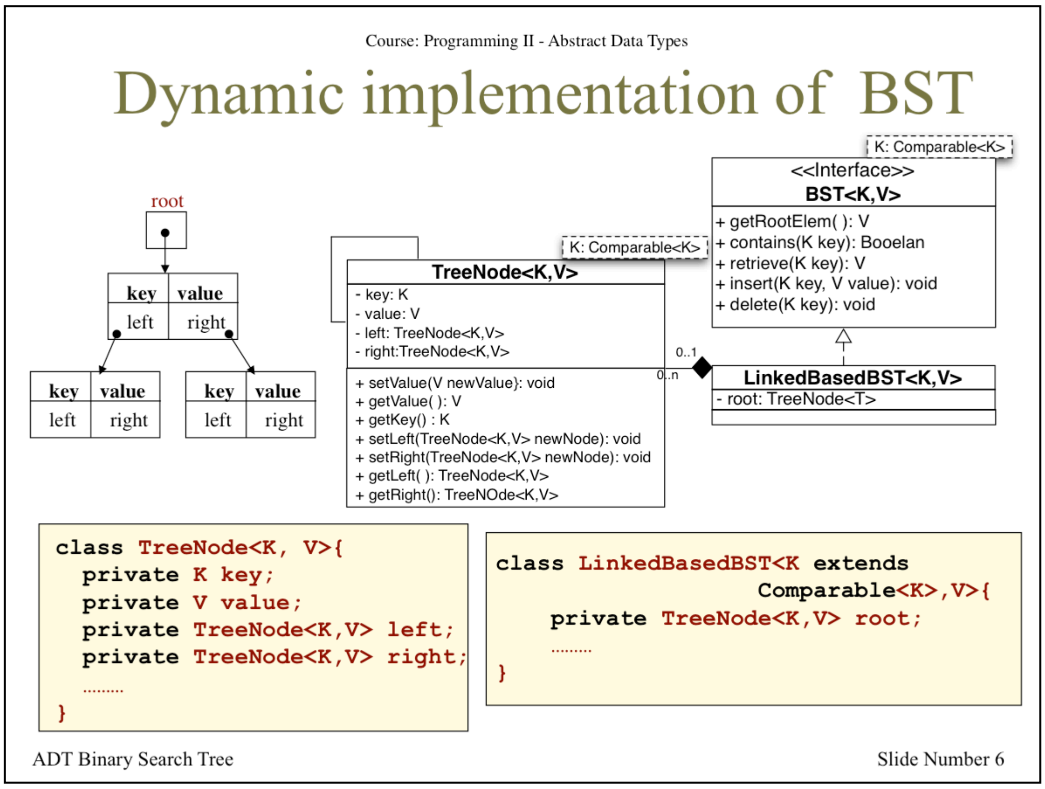
This is an example interface for a binary search tree with the full specification for each of the main access procedures used for binary search trees. Note that the interface extends in this case the Java interface Iterable<T>.

To include also the basic access procedures for a binary tree, which are still applicable to a binary search tree, we could have an interface called `TreeInterface<T>`, that has the basic operations: An example is given below:

```
public interface TreeInterface<T>{
   public T getRootElem();
   public boolean isEmpty();
   public void clear();
 }
```

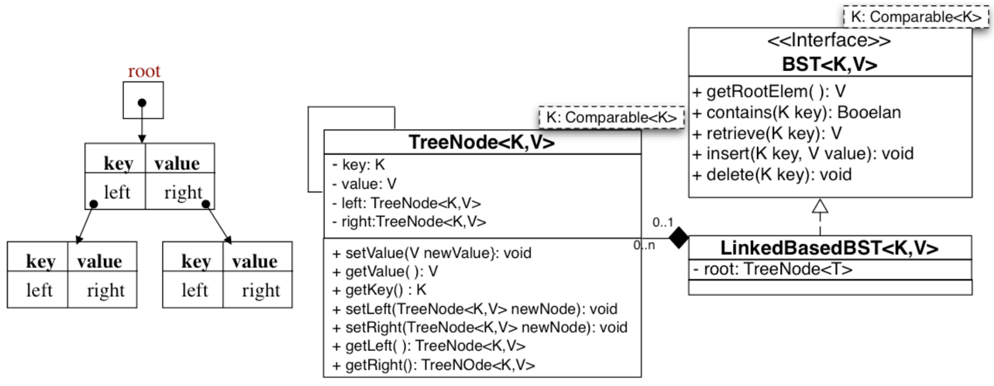
and declare that the interface `BST<K,V>` extends `TreeInterface<V>`.

The iterator method can create an Iterator object that traverses the binary search tree using one of the tree traversal policies that we have seen in the previous lecture. In implementing such a policy you will need to use an iterative approach.



Course: Programming II - Abstract Data Types

Dynamic implementation of BST



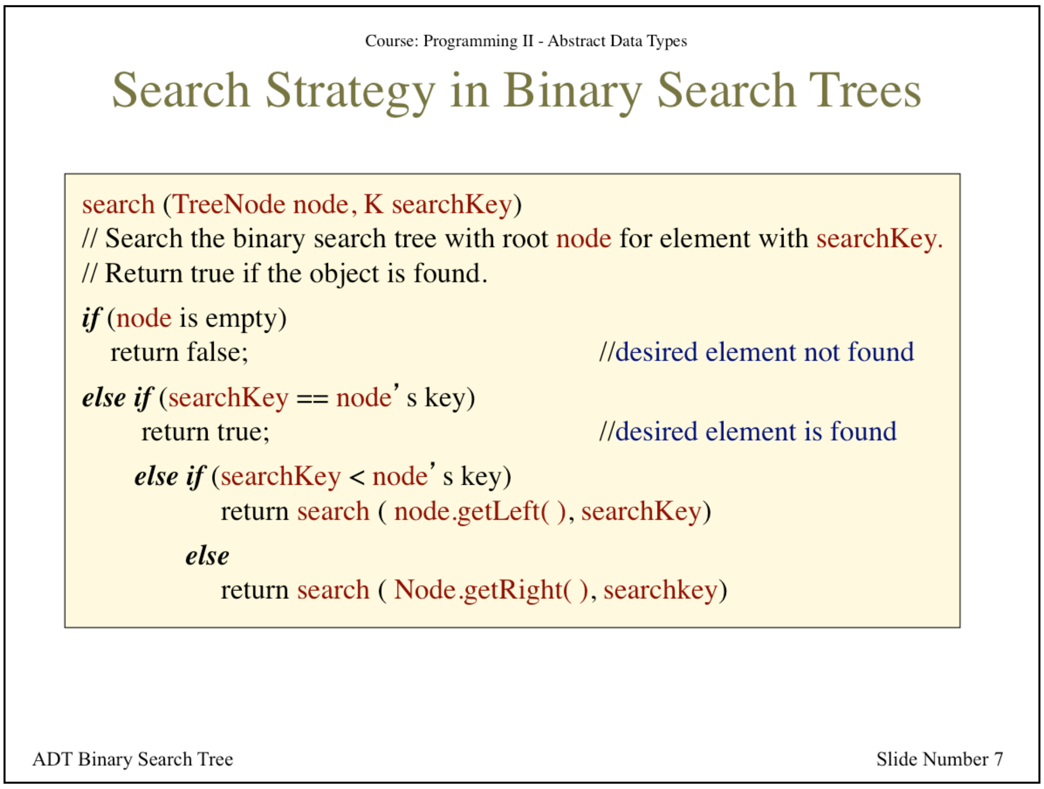
```
class TreeNode<K, V>{
  private K key;
  private V value;
  private TreeNode<K,V> left;
  private TreeNode<K,V> right;
  .........
}
```

```
class LinkedBasedBST<K extends
                   Comparable<K>,V>{
   private TreeNode<K,V> root;
   .........
}
```

ADT Binary Search Tree   Slide Number 6

The entry of a binary search tree can be implemented using a generic class with two parameters, one for the searchkey value and the other for the actual value to be stored in the tree. As we have already seen for ordered linked lists, the type parameter for the search key value has to have the bound expression `K extends Comparable<K>`, to constraint the type of `searchkey` to be of any class that implements the interface `Comparable<K>`.

Note that you could have also defined the TreeNode class using an interface that makes public only the operations of getting the key and getting the value and leaves the other operations (setting the key, setting the value, setting the left, setting right, getting left and getting right protected instead of public). You could also think of using an inner class TreeNode as part of your `LinkedBasedBST` class.


Course: Programming II - Abstract Data Types

Search Strategy in Binary Search Trees

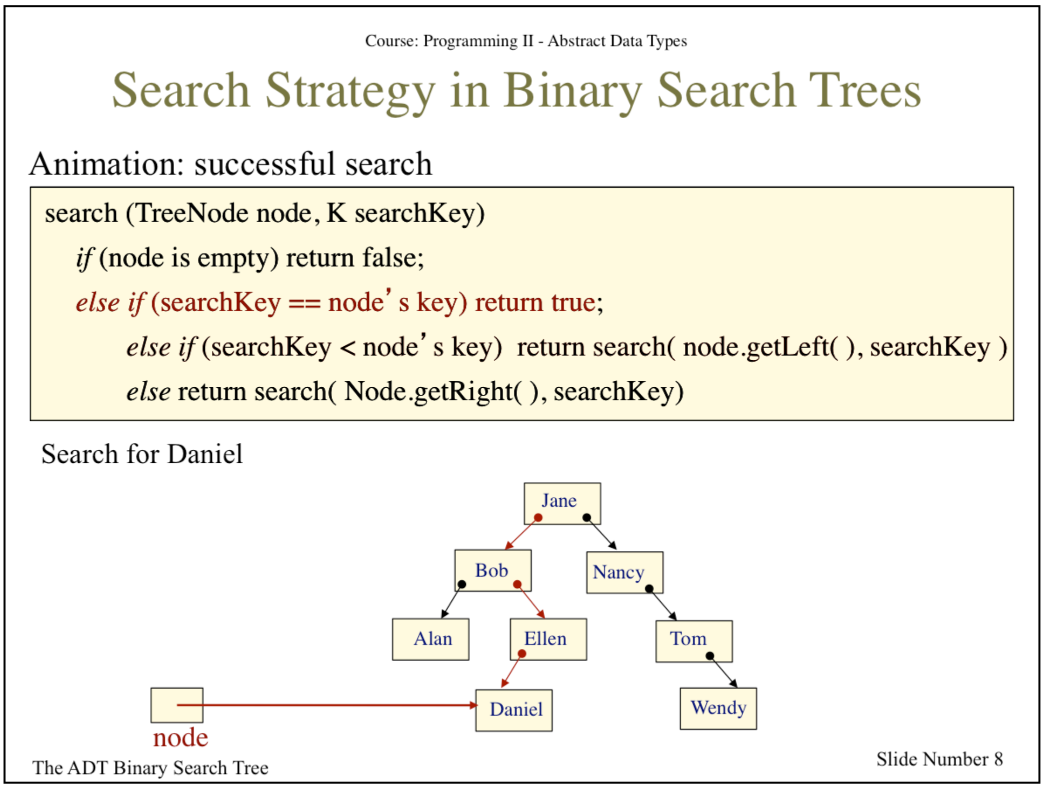
```
search (TreeNode node, K searchKey)
// Search the binary search tree with root node for element with searchKey.
// Return true if the object is found.
if (node is empty)
   return false;                          //desired element not found
else if (searchKey == node’s key)
       return true;                       //desired element is found
     else if (searchKey < node’s key)
            return search ( node.getLeft( ), searchKey)
          else
            return search ( Node.getRight( ), searchkey)
```

ADT Binary Search Tree   Slide Number 7

All three operations of *insertion*, *deletion* and *retrieval* make use of a search strategy. This search strategy allows us, in the case of insertion, to locate the place in the tree where the new element has to be inserted so as to preserve the ordering of the binary search tree, and in the cases of deletion and retrieval, it locates the element (if any) in the tree that needs to be either deleted or returned as output.

Let’s therefore start with understanding how the search strategy can be defined recursively for a binary search tree.

Assume that we pass to the search strategy the root of a binary search tree and the searchKey value of the element that we want to find. Then if the root node is null, the element does not exist in the tree. This condition is reached either at the very beginning if the tree is empty or as base case of the recursion when a (left or right) subtree of a leaf node is reached (these are clearly empty trees). If the root of the current tree considered in a recursive call is not null, than we can compare the element’s key in the root with the given searchKey. If they are equal than we have found the element. Otherwise we need to check whether the given searchKey is smaller or bigger than the key of the root node of the tree under consideration. If it is smaller, the node we are looking for will be in the left subtree. If it is bigger the node that are looking for will be in the right sub-tree. This is clearly because we are dealing with binary search trees and we are sure that the elements with smaller (resp. bigger) key than a root node are all located in the left (resp. right) subtree of the root node. This allows us to decide which next sub-tree to search. In the recursive call we pass the root of the chosen subtree.



Course: Programming II - Abstract Data Types

Search Strategy in Binary Search Trees

Animation: successful search

```
search (TreeNode node, K searchKey)
    if (node is empty) return false;
    else if (searchKey == node’s key) return true;
        else if (searchKey < node’s key)  return search( node.getLeft( ), searchKey )
        else return search( Node.getRight( ), searchKey)
```

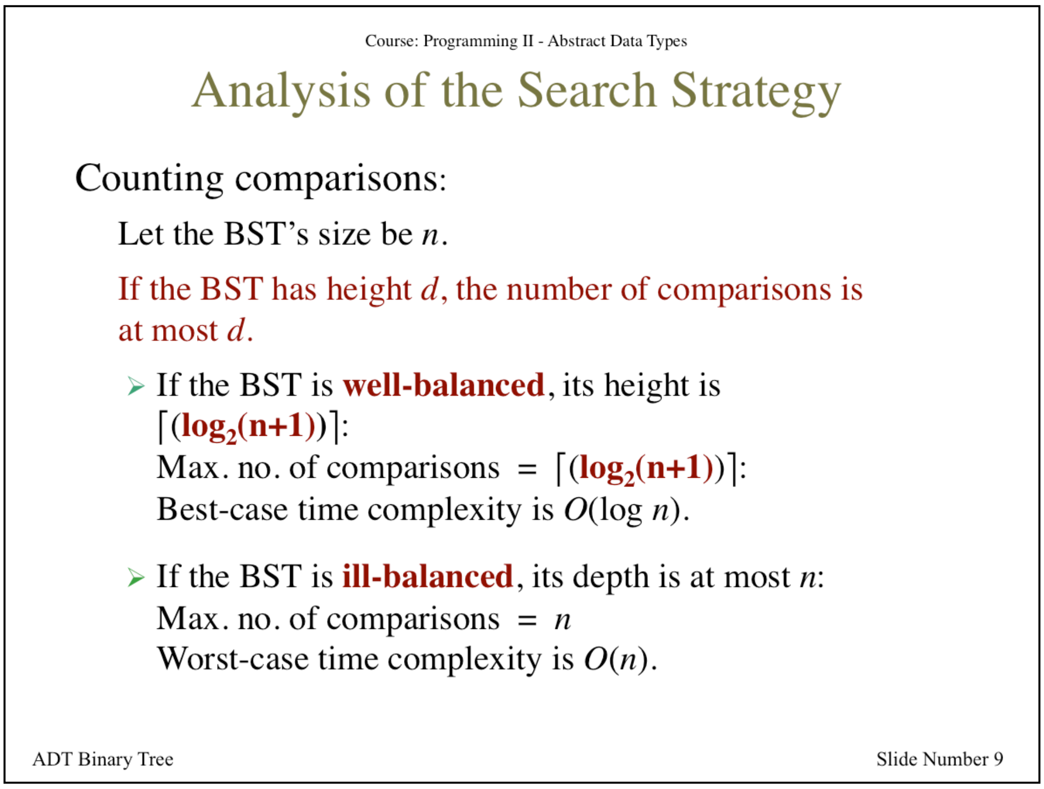
Search for Daniel

Jane
Bob
Nancy
Alan
Ellen
Tom
Daniel
Wendy
node

The ADT Binary Search Tree

Slide Number 8

Consider the binary search tree given in this slide and the task of searching an element with key “Daniel”. This slide shows an animation of the search algorithm. At the beginning, the algorithm starts with the parameter Node given by the root of the tree. Since “Daniel < Jane”, the search method is called again by with the node parameter given by Jane.getLeft(), which is Bob. Now “Daniel > Bob” so the search is called recursively again but with parameter Bob.getRight, which is Ellen. Now “Daniel < Ellen”, so the search is called recursively again but with parameter Ellen.getLeft(), which is Daniel. In this call the current Node’s key (i.e. Daniel) is equal to the givens searchKey so the procedure stops and returns true.



Analysis of the Search Strategy

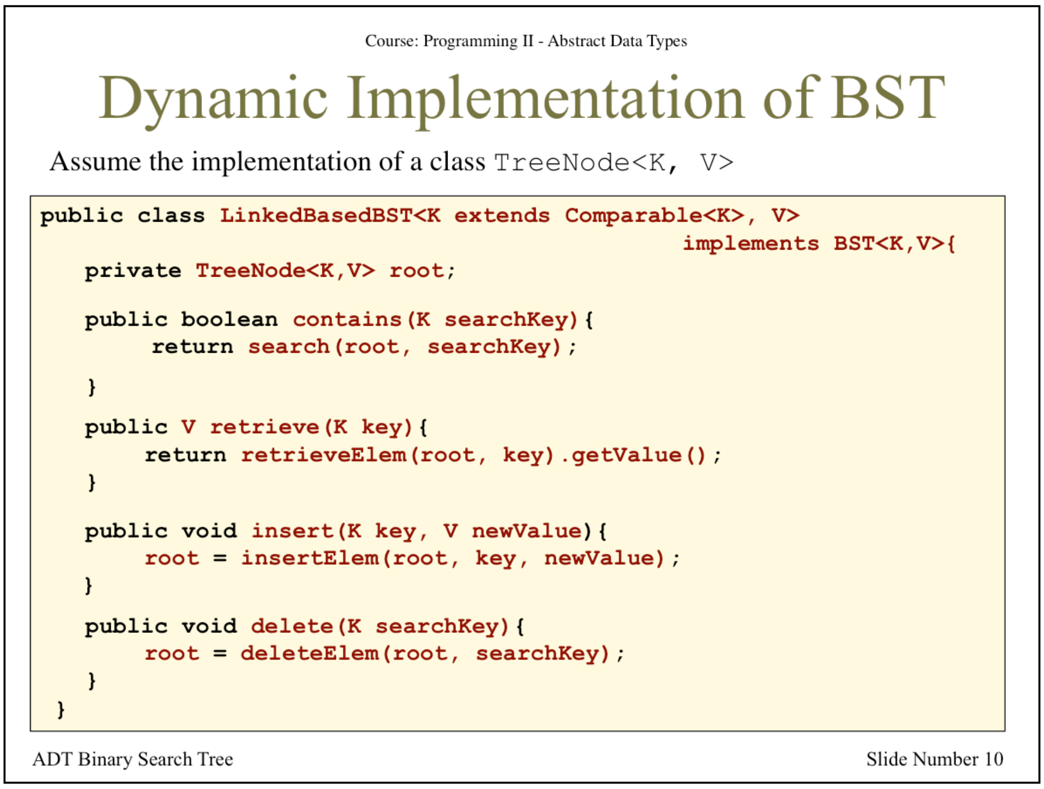
Counting comparisons:

Let the BST's size be n.

If the BST has height d, the number of comparisons is at most d.

- If the BST is **well-balanced**, its height is $\lceil(\mathbf{\log_2(n+1)})\rceil$:
  Max. no. of comparisons = $\lceil(\mathbf{\log_2(n+1)})\rceil$:
  Best-case time complexity is $O(\log n)$.

- If the BST is **ill-balanced**, its depth is at most n:
  Max. no. of comparisons = n
  Worst-case time complexity is $O(n)$.



Dynamic Implementation of BST

Assume the implementation of a class `TreeNode<K, V>`

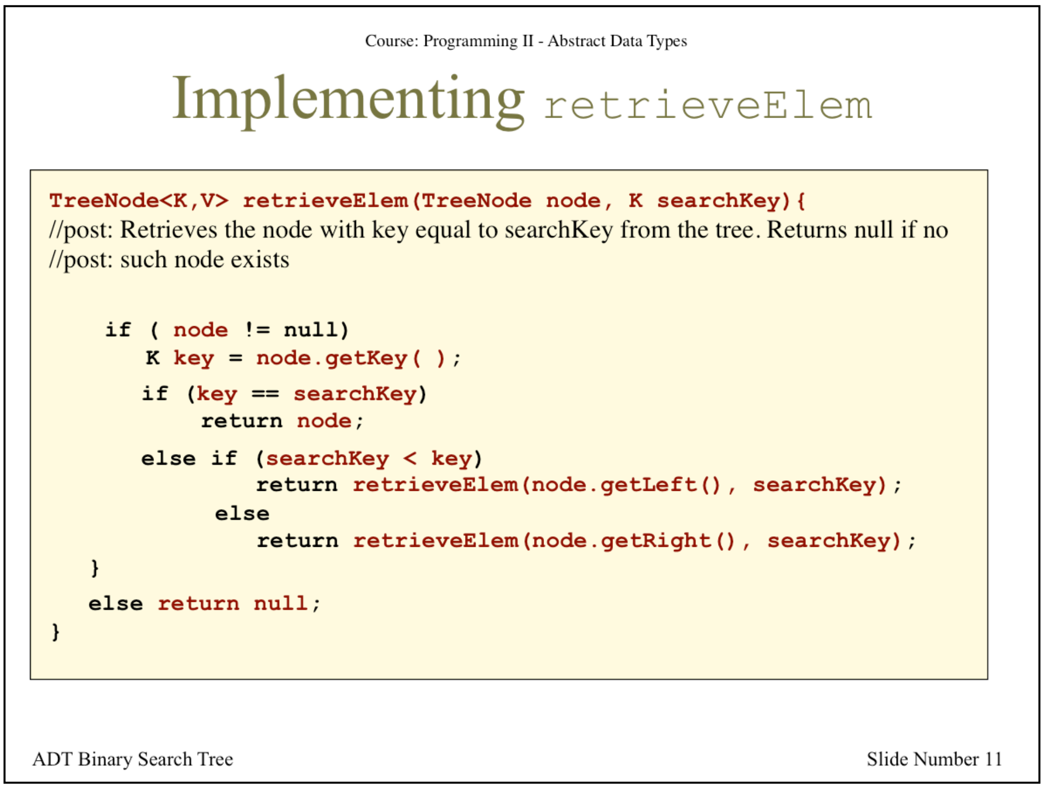
```
public class LinkedBasedBST<K extends Comparable<K>, V>
                                              implements BST<K,V>{
   private TreeNode<K,V> root;

   public boolean contains(K searchKey){
        return search(root, searchKey);
   }
   public V retrieve(K key){
        return retrieveElem(root, key).getValue();
   }
   public void insert(K key, V newValue){
        root = insertElem(root, key, newValue);
   }
   public void delete(K searchKey){
        root = deleteElem(root, searchKey);
   }
 }
```

In this slide we give the implementation of the four main access procedures for binary search trees. Note that none of these access procedures takes as parameter a tree node, but just the value or the key of the element in question (to search, delete or add) search key or both (as in the case of insert access procedure). The search algorithm given before takes as parameter also the root of a given binary search tree and and passes in each recursive call the root of the sub-trees recursively visited. This search strategy is directly used by the `contains` access procedure. The other three access procedures follow a similar search strategy, i.e. the auxiliary methods `insertElem` for the access procedure `insert`, `deleteElem` for the access procedure `delete`, and `retrieveElem` for the access procedure `retrieve`.

Since these recursive methods are only auxiliary, they should be declared private within the class `BinarySearchTree`.

Let's see how to implement each of these auxiliary access procedures. We'll first see the basic algorithmic idea, and then we'll give a pseudocode for each of them.

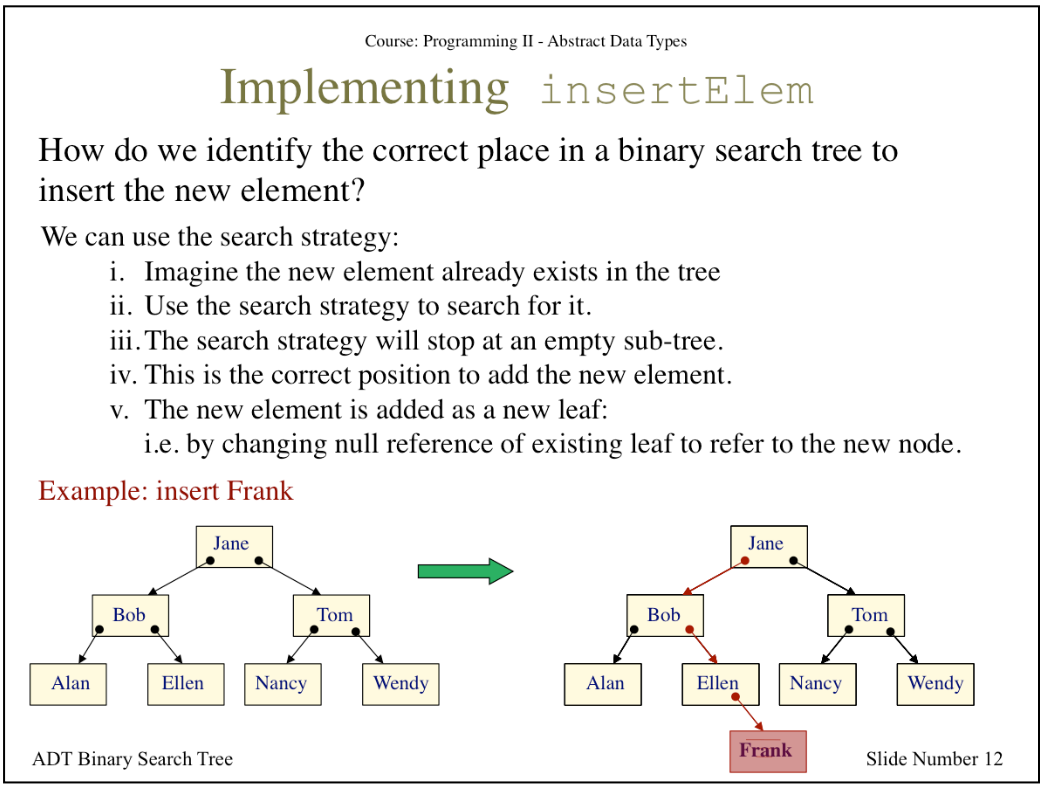


Implementing `retrieveElem`

```
TreeNode<K,V> retrieveElem(TreeNode node, K searchKey){
//post: Retrieves the node with key equal to searchKey from the tree. Returns null if no
//post: such node exists

    if ( node != null)
       K key = node.getKey( );
       if (key == searchKey)
           return node;
       else if (searchKey < key)
               return retrieveElem(node.getLeft(), searchKey);
            else
               return retrieveElem(node.getRight(), searchKey);
   }
   else return null;
}
```

This is an auxiliary method used by the access procedure retrieve(K key) defined in the previous slide. The algorithm is similar to the search algorithm described previously but with the difference that in this case we are not returning a Boolean value but the actual node in the tree that has a key value equal to the given searchkey.



Course: Programming II - Abstract Data Types

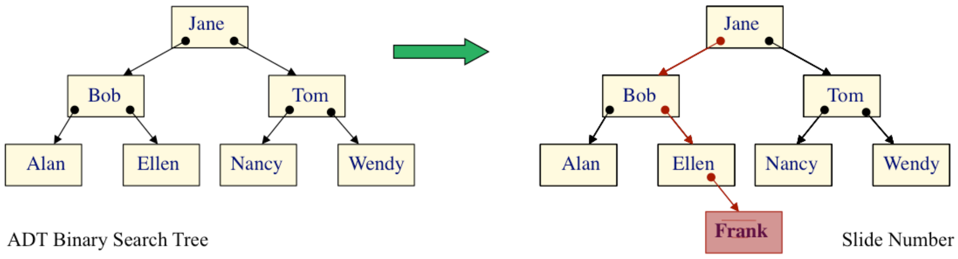
Implementing `insertElem`

How do we identify the correct place in a binary search tree to insert the new element?

We can use the search strategy:

i. Imagine the new element already exists in the tree
ii. Use the search strategy to search for it.
iii. The search strategy will stop at an empty sub-tree.
iv. This is the correct position to add the new element.
v. The new element is added as a new leaf:
   i.e. by changing null reference of existing leaf to refer to the new node.

Example: insert Frank



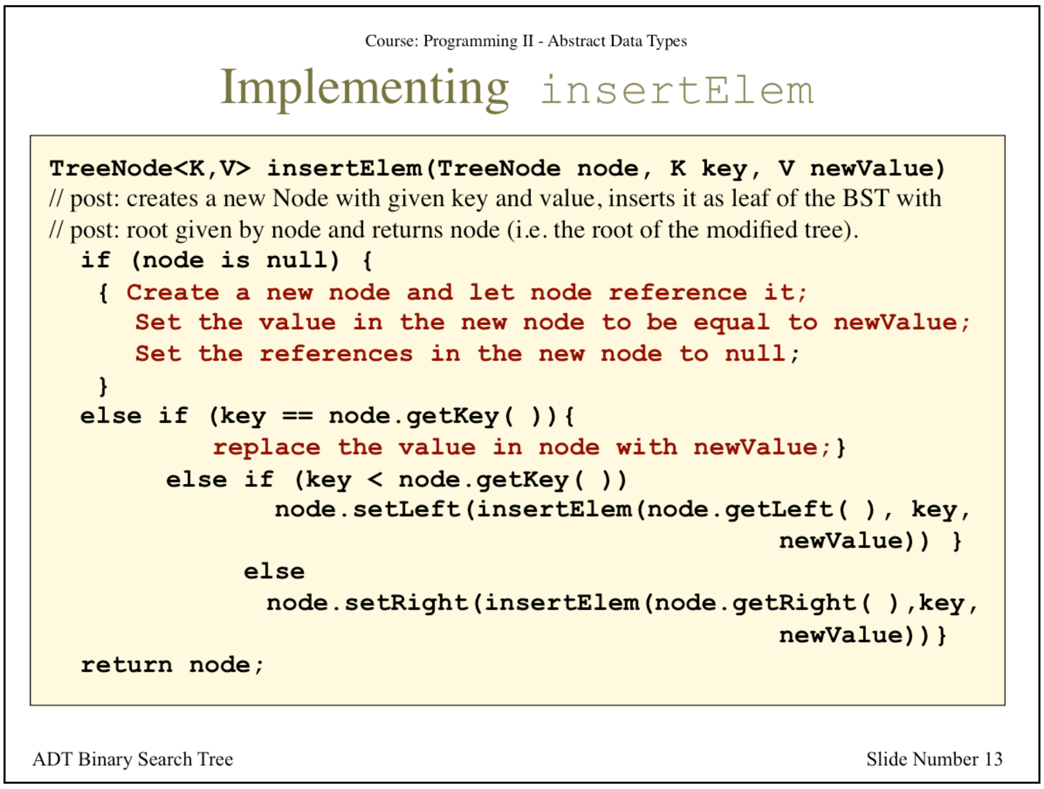
ADT Binary Search Tree Slide Number 12

Suppose that we want to insert a new element in the tree. Consider the binary search tree given in the left hand side of this slide, and suppose that we want to insert in this tree a new node with key “Frank”. Let’ s assume that the tree does not include already the entry we want to add.

As a first step, imagine that we want to search for this element assuming that it is already present in the tree. We can do this using the search strategy given in the previous slides. We will first search the tree rooted at Jane, then the tree rooted at Bob, then the tree rooted at Ellen, and finally the right subtree of Ellen. Because this last tree is empty, the search strategy has reached its base case of recursion. If we were only searching a node we would return false. But since we want to add a new node, the base case of the recursive step is where the new node is created and added to the tree.

If Frank were present it would have been the right child of Ellen, which has been found to be empty. This indicates that a good place to insert Frank is as the right child of the node Ellen. Because we have used the search strategy, we can also be sure that inserting the new node at this identified position would still make the new tree satisfy the properties of a binary search tree.

Note that using this search strategy to determine where in the tree to insert a new node always leads to an easy insertion. No matter what the new element is, the search will always terminate at an empty sub-tree if the entry is not already in the tree. Thus, the search always tells us to insert the new element as a new leaf. Adding a new leaf requires changing appropriate reference variables in the parent node.


Course: Programming II - Abstract Data Types

Implementing `insertElem`

```
TreeNode<K,V> insertElem(TreeNode node, K key, V newValue)
// post: creates a new Node with given key and value, inserts it as leaf of the BST with
// post: root given by node and returns node (i.e. the root of the modified tree).
  if (node is null) {
   { Create a new node and let node reference it;
     Set the value in the new node to be equal to newValue;
     Set the references in the new node to null;
   }
  else if (key == node.getKey( )){
          replace the value in node with newValue;}
        else if (key < node.getKey( ))
               node.setLeft(insertElem(node.getLeft( ), key,
                                               newValue)) }
             else
              node.setRight(insertElem(node.getRight( ),key,
                                               newValue))}
  return node;
```

ADT Binary Search Tree   Slide Number 13

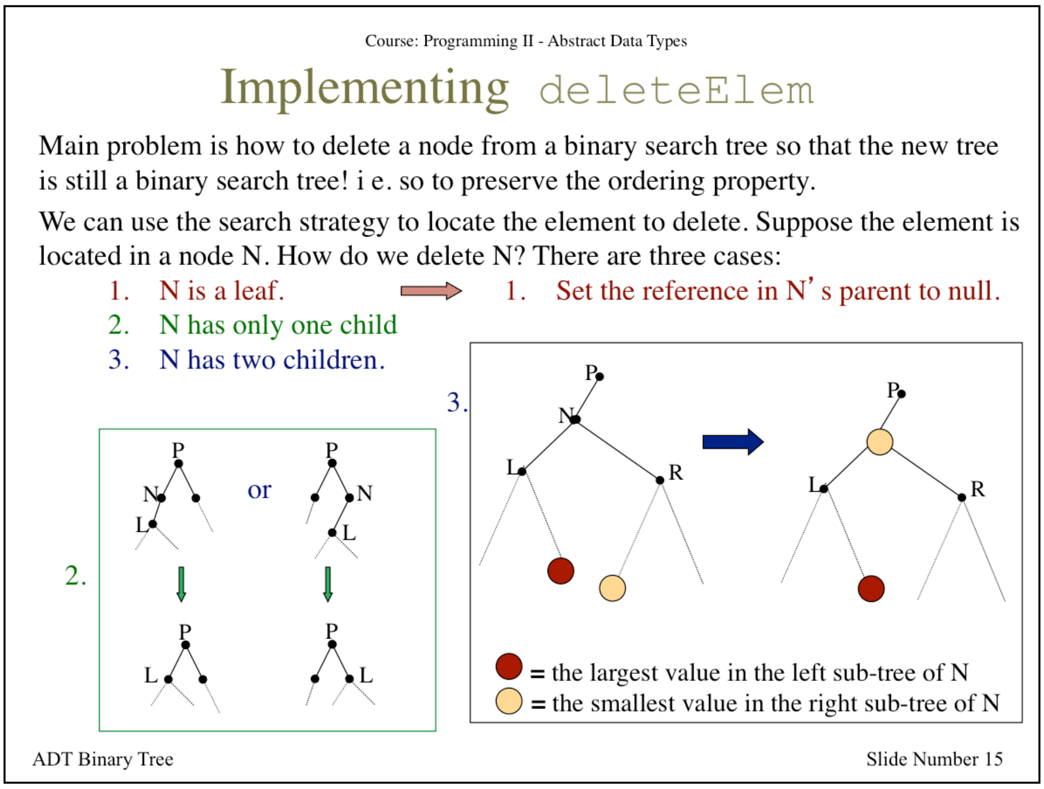
Note that we have made use here of a recursive call of the method `insertElem`. The main difference between this pseudo code and the pseudo code of the search strategy is that in this case when we find the null node while we traverse the tree we don't return false, but we actually insert the new node in the tree. The important thing to understand is how this recursive algorithm sets the "leftChild" or the "rightChild" of an identified parent node to reference the new node.
If the tree is empty before the insertion, then the external reference to the root of the tree would be null, and so the method would not make a recursive call. This is indeed one of the base cases of this recursive method. When this situation occurs, a new tree node must be created and initialised. The method then returns the reference to this node. The calling environment will then set the left or right child its current node to be equal to the returned (newly created) node. This is done by the method call `treeNode.setLeft`, to set the returned node to be the new left child of the parent, or by the method call `treeNode.setRight` to set the returned node to be the new right child of the parent, within the context of the calling recursion.

Analysis of insertion in BST

Counting comparisons:

No. of comparisons is the same as for BST search

- Whether a BST is **well-balanced** or **ill-balance**, depends on the order of insertion.
  - If the inserted elements are randomly ordered, the BST will probably be reasonably well-balanced.
  - If the inserted elements happen to be in ascending (or descending) order, the BST will be extremely ill-balanced.



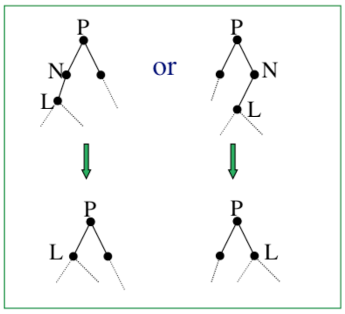
Implementing `deleteElem`

Main problem is how to delete a node from a binary search tree so that the new tree is still a binary search tree! i e. so to preserve the ordering property.

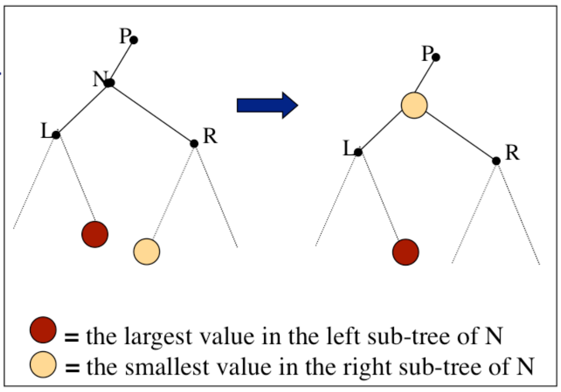
We can use the search strategy to locate the element to delete. Suppose the element is located in a node N. How do we delete N? There are three cases:

1. N is a leaf. → 1. Set the reference in N's parent to null.
2. N has only one child
3. N has two children.

2.



3.



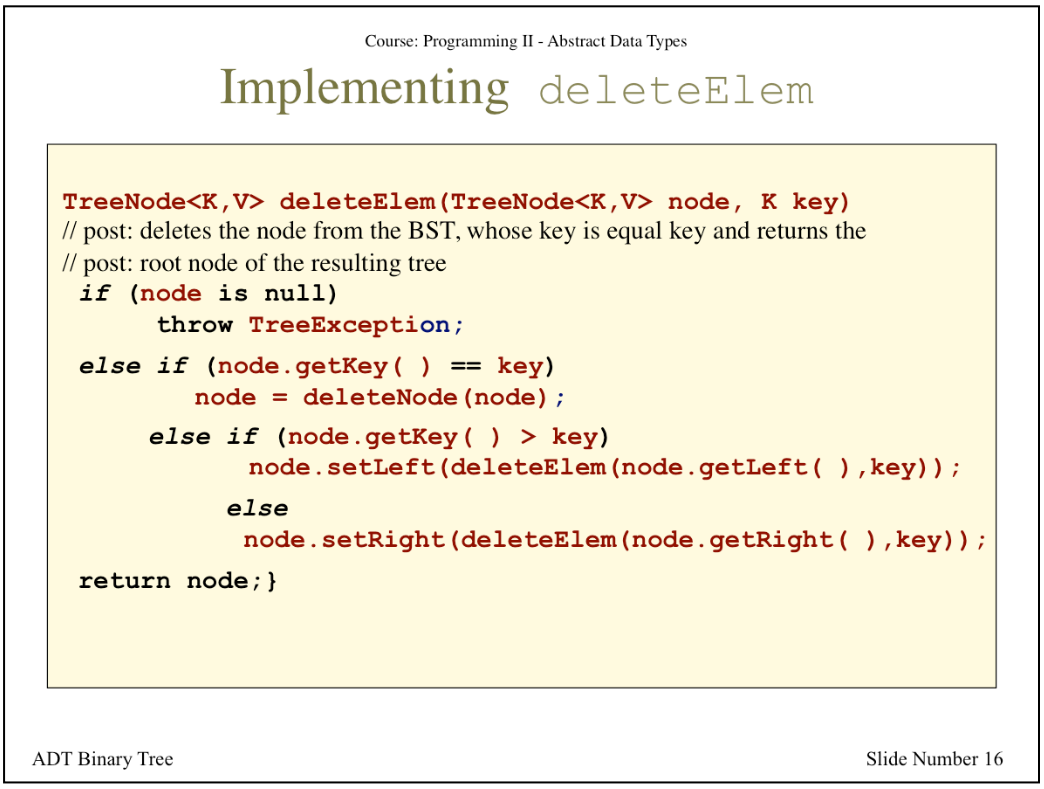
● = the largest value in the left sub-tree of N
○ = the smallest value in the right sub-tree of N

The deletion operation is more complex than insertion. We present here the algorithmic idea for this auxiliary method, and its pseudocode.

We can still use a search strategy for locating the node that includes the value to be deleted. But then the question is how do we delete the node and guarantee that the resulting tree is still a binary search tree containing all the remaining items?

When we locate the particular node, say N, the node can be one of the following three possible cases: (1) the node N is a leaf node, in which case removing it means simply set the value of its parent's reference variable to null. (2) The node has only one child, a left child or a right child. These two cases are symmetrical, so we just consider the case when N has only the left child. We could think of simply removing N by making its left child (say L) replace the node N. It is not difficult to see that this operation still preserves the ordering of a binary search tree whether N is in the left or right subtree.

The more difficult case is the third case when the node N has two children. We can't just delete N by replacing it with the two children, so we need to think of alternative solutions. The simplest way is not to delete N but to find a node in the tree that is easy to delete (i.e. a node that is a leaf or that has only one child), replace N with this node and delete this node from the tree. But which node shall we pick? We can't take any arbitrary leaf node, because otherwise the ordering will not be respected any longer. On the other hand, if we were going to pick the right most node in the left sub-tree of N, this would have a value that is still smaller than N and bigger then any other node in the left sub-tree of N, and also smaller than any node in the right subtree of N. In the same way, if we pick the leftmost node in the right sub-tree of N, this is a node that is the smallest value greater then N and therefore it would still be less than any other value in the right sub-tree of N; it is also greater than any value of the left sub-tree of N. Either of these two leaf nodes would be a good replacement for node N, as it will preserve the ordering and permit N to be deleted. In this slide we have chosen the second case, and replaced N with the leftmost node in the right sub-tree of N.


Course: Programming II - Abstract Data Types

Implementing `deleteElem`

```
TreeNode<K,V> deleteElem(TreeNode<K,V> node, K key)
// post: deletes the node from the BST, whose key is equal key and returns the
// post: root node of the resulting tree
 if (node is null)
      throw TreeException;
 else if (node.getKey( ) == key)
           node = deleteNode(node);
      else if (node.getKey( ) > key)
              node.setLeft(deleteElem(node.getLeft( ),key));
           else
              node.setRight(deleteElem(node.getRight( ),key));
 return node;}
```

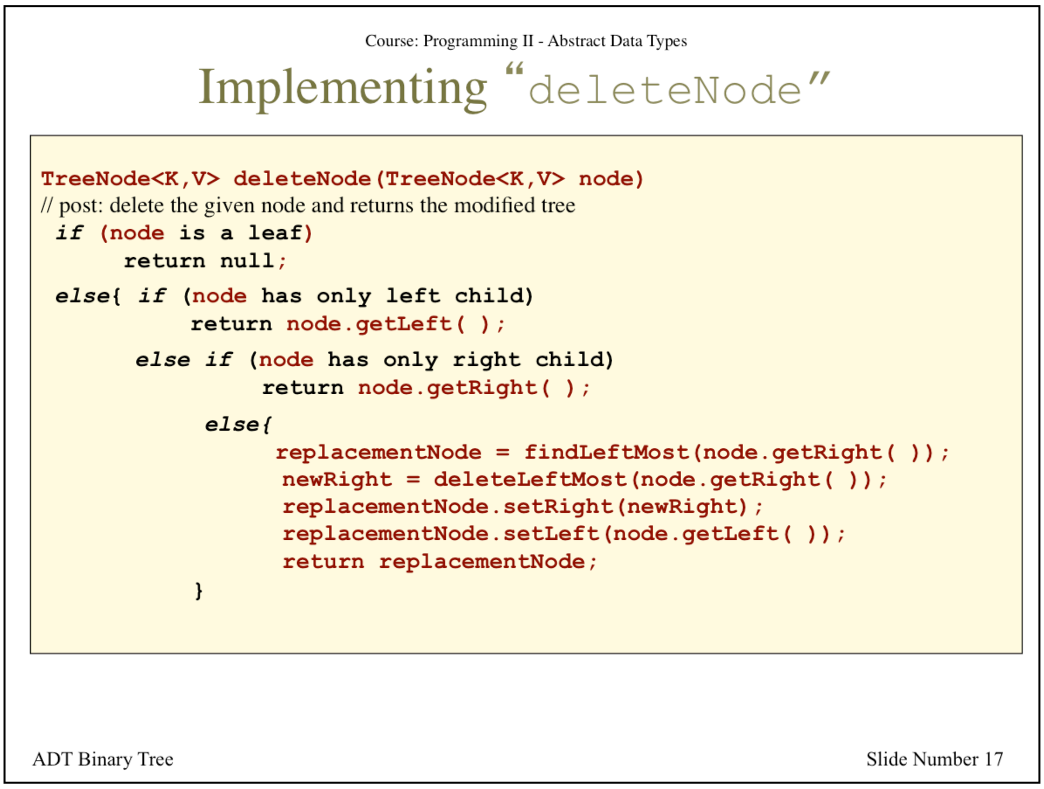
ADT Binary Tree

Slide Number 16

This slide gives the algorithm for the recursive pseudocode of the auxiliary method “deleteElem”, which deletes a node from a Binary Search Tree and returns the new Binary Search Tree. Of course if the node is not in the tree, an exception is thrown.

The algorithm is again similar to the search strategy given before, with the main difference that when the node is found, it needs to be deleted The main issue is then how do we delete this node. The actual operation of deleting a node once it has been found is performed by an other auxiliary method “deleteNode”, which is defined in the next slide.

Note that the first *if* statement checks whether the root of the given tree already includes the key of the element we want to delete. If not, at each recursive call the value of the parameter node will be the root of the particular sub-tree considered in the recursion.



Course: Programming II - Abstract Data Types

Implementing “deleteNode”

```
TreeNode<K,V> deleteNode(TreeNode<K,V> node)
// post: delete the given node and returns the modified tree
 if (node is a leaf)
      return null;
 else{ if (node has only left child)
           return node.getLeft( );
      else if (node has only right child)
              return node.getRight( );
          else{
               replacementNode = findLeftMost(node.getRight( ));
               newRight = deleteLeftMost(node.getRight( ));
               replacementNode.setRight(newRight);
               replacementNode.setLeft(node.getLeft( ));
               return replacementNode;
          }
```

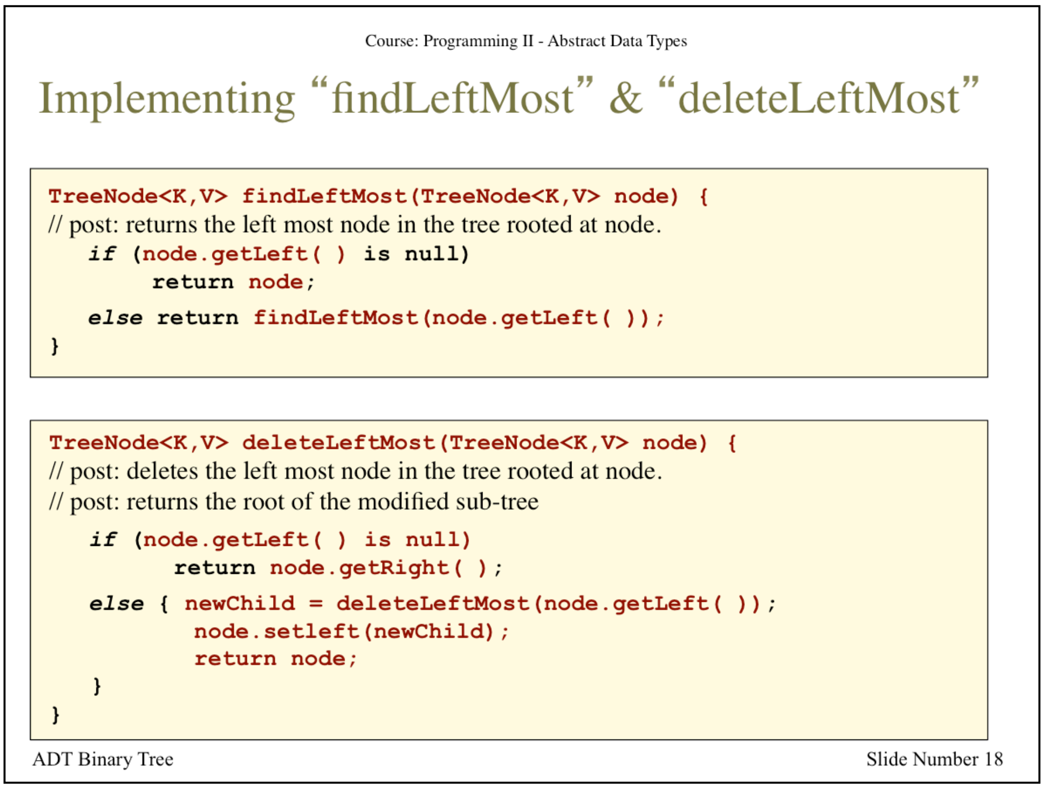
ADT Binary Tree Slide Number 17

This is also an auxiliary method used by the deleteElem private procedure of a binary search tree. Once a given node to delete has been found in the tree, this method is used on the node with the found key to handle the case of deletion. We check first whether this node is a leaf. If so, then the method has just to return null, since this will be the value of the new leaf once the node has been deleted.

If the current node is not a leaf, we check the other possible cases discussed in the previous slide. The node can have just the left child, or just the right child or both children. In the first case, the left child is returns as it will be attached to the node in the calling procedure. Similarly for the right child.

More complex is the case when the node has two children. In this case, as we have already diagrammatically described in slide 15, we need to construct a new subtree but with its root’ value given by the value of the leftmost node of the right subtree of the node that need to be deleted. We have then to find the left most node of the right subtree of the current node, delete this left most node from this right subtree and set the right link of this new node to be the new right subtree. The left link of this new node should then be set to be the left subtree of the originally identified treeNode. The so created new node will then be returned as new root of the subtree currently examined by the deleteElem calling procedure.

What is left to see are the algorithms for finding the left most node in a given binary search tree, and for deleting the left most node of a given binary search tree.



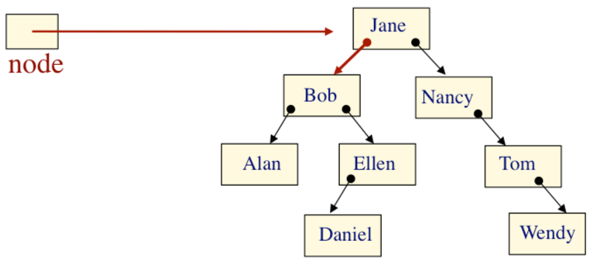
Implementing “findLeftMost” & “deleteLeftMost”

```
TreeNode<K,V> findLeftMost(TreeNode<K,V> node) {
// post: returns the left most node in the tree rooted at node.
   if (node.getLeft( ) is null)
        return node;
   else return findLeftMost(node.getLeft( ));
}
```

```
TreeNode<K,V> deleteLeftMost(TreeNode<K,V> node) {
// post: deletes the left most node in the tree rooted at node.
// post: returns the root of the modified sub-tree
   if (node.getLeft( ) is null)
         return node.getRight( );
   else { newChild = deleteLeftMost(node.getLeft( ));
          node.setleft(newChild);
          return node;
   }
}
```

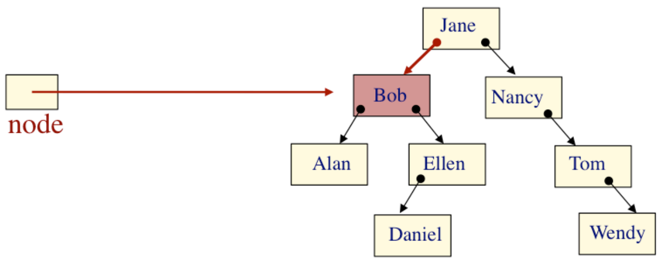
These are the other two auxiliary methods for finding the leftmost node of a given subtree, and deleting the leftmost node of a given subtree and returning the new tree.

delete Bob



```
deleteElem(TreeNode<K,V> node, K key)
 if (node is null)
      throw TreeException;
 else if (node.getKey( ) == key)
         node = deleteNode(node);
     else if (node.getKey( ) > key)
           node.setLeft(deleteElem(node.getLeft( ),key));
          else
           node.setRight(deleteElem(node.getRight( ),key));
 return node;
```

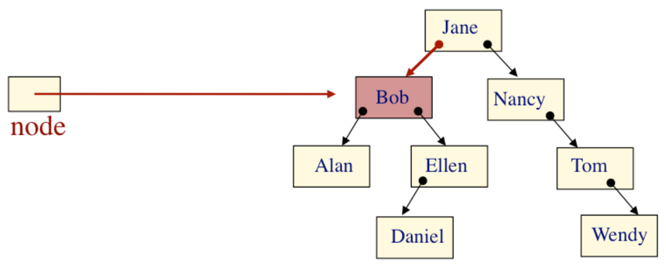
delete Bob



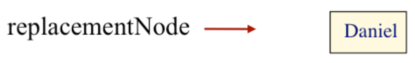
node =

```
deleteElem(TreeNode<K,V> node, K key)
 if (node is null)
      throw TreeException;
 else if (node.getKey( ) == key)
         node = deleteNode(node);
      else if (node.getKey( ) > key)
            node.setLeft(deleteElem(node.getLeft( ),key));
           else
            node.setRight(deleteElem(node.getRight( ),key));
 return node;
```

delete Bob

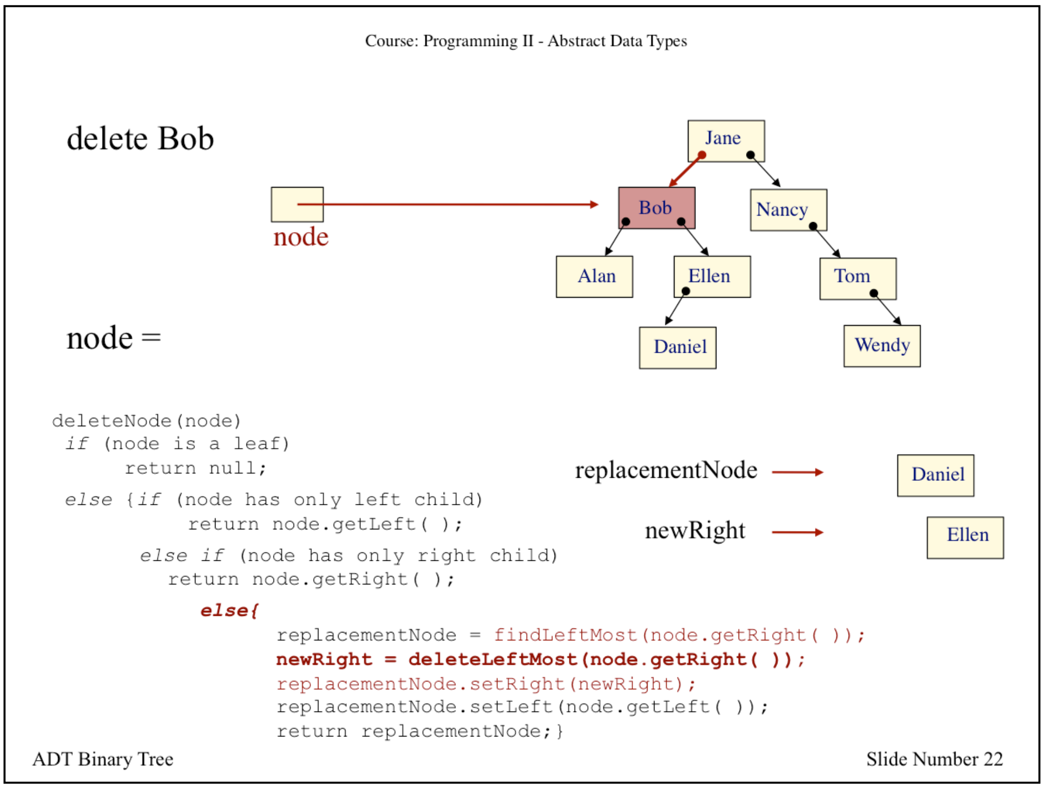


node =

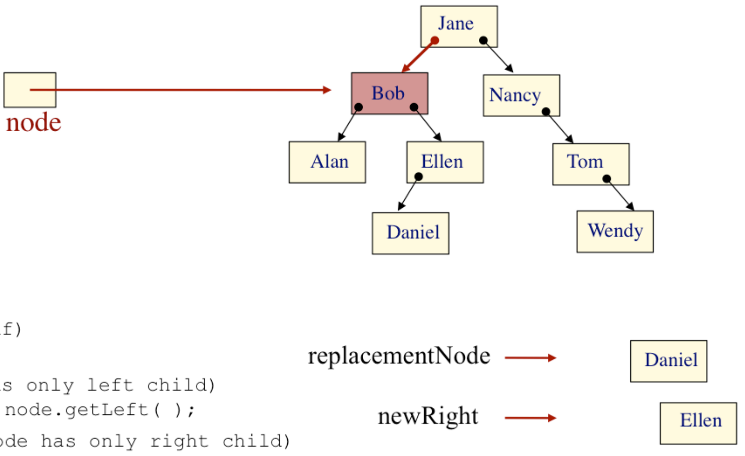


```
deleteNode(node)
 if (node is a leaf)
      return null;
 else {if (node has only left child)
           return node.getLeft( );

       else if (node has only right child)
         return node.getRight( );
            else{
                  replacementNode = findLeftMost(node.getRight( ));
                  newRight = deleteLeftMost(node.getRight( ));
                  replacementNode.setRight(newRight);
                  replacementNode.setLeft(node.getLeft( ));
                  return replacementNode; }
```



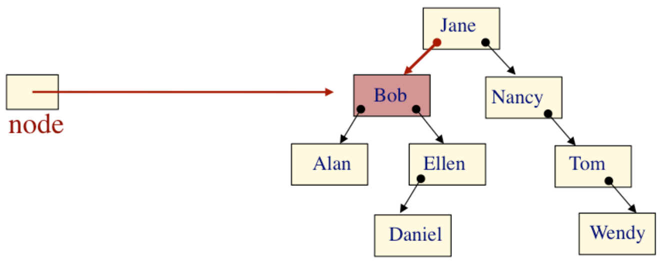
delete Bob



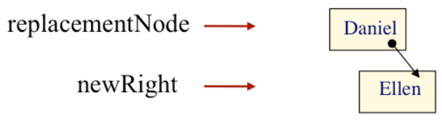
node =

```
deleteNode(node)
 if (node is a leaf)
      return null;
 else {if (node has only left child)
           return node.getLeft( );
       else if (node has only right child)
          return node.getRight( );
            else{
                  replacementNode = findLeftMost(node.getRight( ));
                  newRight = deleteLeftMost(node.getRight( ));
                  replacementNode.setRight(newRight);
                  replacementNode.setLeft(node.getLeft( ));
                  return replacementNode;}
```

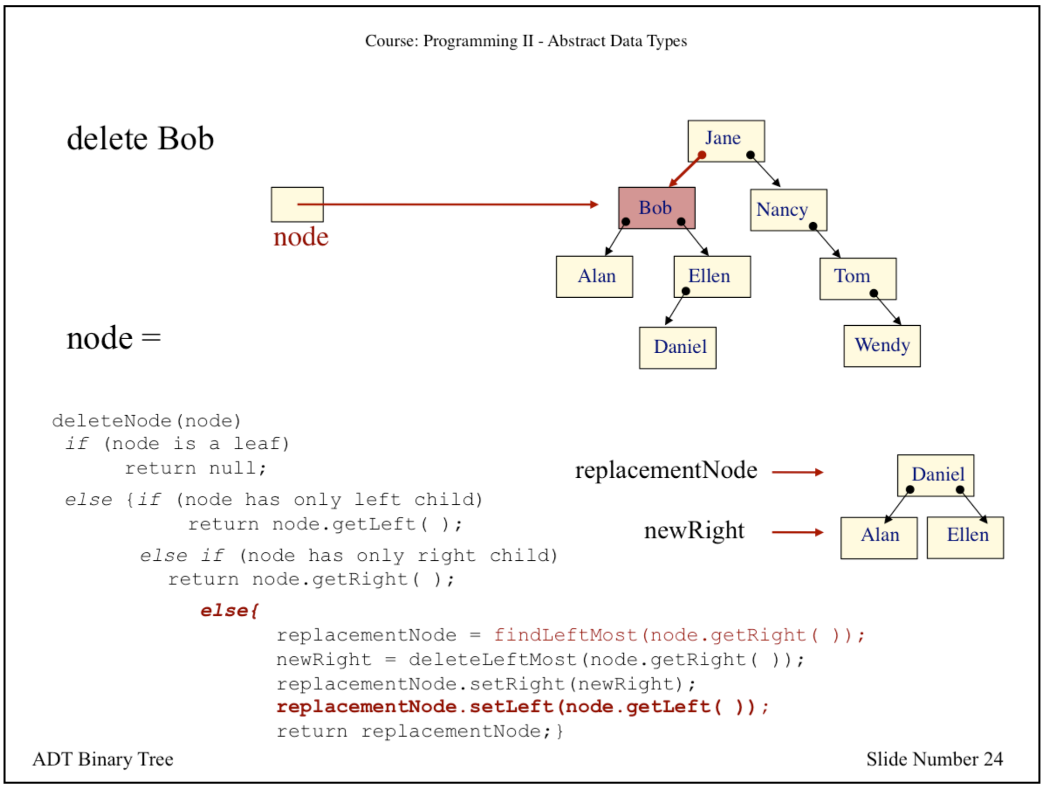
delete Bob



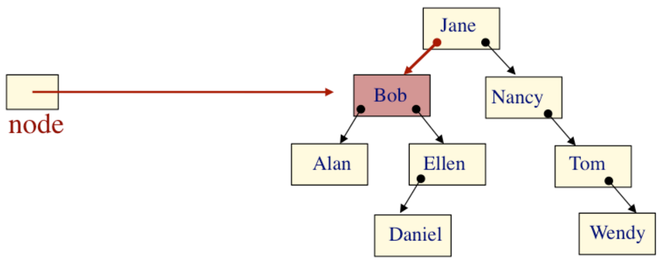
node =



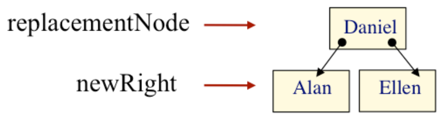
```
deleteNode(node)
 if (node is a leaf)
      return null;
 else {if (node has only left child)
           return node.getLeft( );
       else if (node has only right child)
         return node.getRight( );
           else{
                  replacementNode = findLeftMost(node.getRight( ));
                  newRight = deleteLeftMost(node.getRight( ));
                  replacementNode.setRight(newRight);
                  replacementNode.setLeft(node.getLeft( ));
                  return replacementNode; }
```



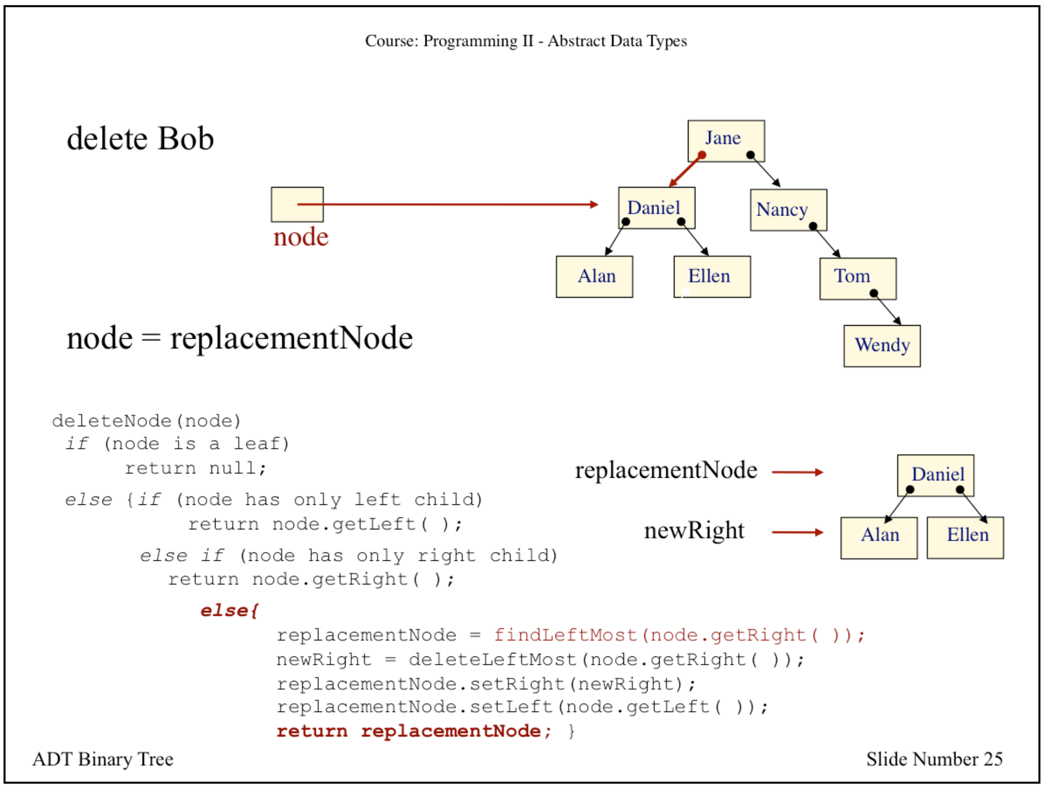
delete Bob



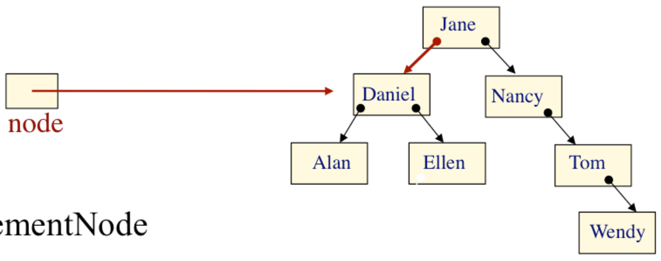
node =



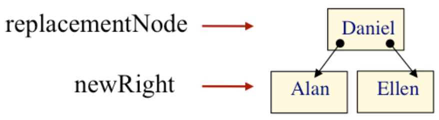
```
deleteNode(node)
 if (node is a leaf)
      return null;
 else {if (node has only left child)
           return node.getLeft( );
       else if (node has only right child)
         return node.getRight( );
            else{
                  replacementNode = findLeftMost(node.getRight( ));
                  newRight = deleteLeftMost(node.getRight( ));
                  replacementNode.setRight(newRight);
                  replacementNode.setLeft(node.getLeft( ));
                  return replacementNode;}
```



delete Bob



node = replacementNode



```
deleteNode(node)
 if (node is a leaf)
      return null;
 else {if (node has only left child)
           return node.getLeft( );
       else if (node has only right child)
         return node.getRight( );
            else{
                  replacementNode = findLeftMost(node.getRight( ));
                  newRight = deleteLeftMost(node.getRight( ));
                  replacementNode.setRight(newRight);
                  replacementNode.setLeft(node.getLeft( ));
                  return replacementNode; }
```

Summary

- The binary search tree allows the use of a binary search-like strategy to search for an element with a specified key.
- An in-order traversal of a binary search tree visits the tree's nodes in sorted order. Binary search trees come in many shapes.
- The height of a binary search tree with n nodes can range from a minimum of $\log_2(n+1)$, to a maximum of n.
- The shape of a binary search tree determines the efficiency of its operations. The closer a binary search tree is to a balanced tree, the closer to $\log_2(n)$ would be the maximum number of comparisons needed in order to find a particular element.