The ADT Binary Search Tree

The Binary Search Tree is a particular type of binary tree that enables easy searching for specific items.

**Definition**

The **ADT Binary Search Tree** is a binary tree which has an ordering imposed 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.

**Applications**

- efficient procedures for inserting and retrieving data.

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

Searching for a particular item is an operation for which the ADT binary trees are not well suited. Binary search trees are particular type of binary trees that correct this deficiency by organising its data by value. Each node in a binary search tree satisfies the two properties that (a) all the nodes in its 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 to search a binary tree for a particular data item, given its value instead of its position, and in certain conditions this search can be quite efficient, as we will see later in this lecture.
Items in a Binary Search Tree often 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.

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 often called the “search key”.

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 keys 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, deleting, and retrieving data from a given binary search tree. The main difference compared with what we have seen so far is that in the case of a binary search tree, these operations are done by searching on the key value rather than by its position in the data structure, as we have seen for instance in the case of linked lists.

The operations of insertion, deletion and retrieval of items 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 notes are also applicable to the ADT Binary Search Tree.
Access Procedures for Binary Search Trees

- The Access procedures `createEmptyTree()`, `isEmpty()`, `getRootItem()`, `getLeftTree()`, `getRightTree()` for binary trees are unchanged.

- Additional access procedures are needed to add items to and delete items from a binary search tree according to their search key’s value:
  1. **insert(newItem)**
     // post: insert newItem in a binary search tree, whose nodes have search keys
     // that differ from the newItem’s search key.
  2. **delete(searchKey)**
     // post: delete from a binary search tree, the item whose search key equals
     // searchKey. If no such item exists, the operation fails and throws exception.
  3. **retrieve(searchKey)**
     // post: returns the item in a binary search tree whose search key equal
     // searchKey. Returns null if no such item exists.

As it is a binary tree, the binary search tree includes the basic operations for the ADT’s binary tree seen in the previous lecture. In particular, the default constructor `createEmptyTree()`, the operations `isEmpty()`, `getRootItem()`, `getLeftTree()`, and `getRightTree()` 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.

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

In the remainder of this lecture we will see the implementation of these three new access procedures: insertion, deletion and retrieval. They all make use of an auxiliary method which is `search(searchKey)` that searches for an item whose search key is equal to `searchKey`.

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

Let’s start with a recursive definition of the algorithm for searching for a given item in a binary search tree. This is given in the next slide.
The Search Strategy in Binary Search Trees

Pseudocode

```
search(TreeNode Node, newItem)
// search the binary search tree with root Node for newItem
if (Node is empty)
    { the desired item is not found }
else if (newItem == Node’s item)
    { the desired item is found }
else if (newItem < Node’s item)
    { search(Node.getLeftTree( ), newItem) }
else
    { search(Node.getRightTree( ), newItem) }
```

All the 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 item has to be inserted so as to preserve the ordering property, and in the cases of deletion and retrieval, it locates the item that needs to be either deleted or returned as output.

Let’s therefore start with understanding how the searching strategy can be defined recursively for a binary search tree. Let’s assume for simplicity that the item in the tree contains only one value and that this value is unique in the tree (i.e. the tree does not include more than one instance of any item).

Assume that we pass to the search strategy the root of a binary search tree and the Item. Then if the root node is null, the item does not exist in the tree. If it is not null, than we need to compare the item in the root with the newItem given in input. If they are equal than we have found the node. Otherwise we need to check whether newItem is bigger or smaller than the value in the root. In the first case, because we are dealing with binary search trees, we are sure that the item, if it exists, should be in the right sub-tree. If it is smaller than the item in the root, then it should be in the left sub-tree. So we get the appropriate new parameters and call recursively the same procedure on the relevant sub-tree.

Consider the binary search tree given in the next slide, and suppose that we want to search for Ellen. Jane is the root node of the tree, so if Ellen is present in the tree it must be in Jane’s left sub-tree, because Ellen comes before Jane (we use here standard lexicographic ordering of strings which we use also when we search for a word in a dictionary). Now Bob is in the root of Jane’s left sub-tree and this left sub-tree is also a binary search tree. So we can use exactly the same strategy for searching for Ellen in Jane’s left sub-tree. Specifically, since Ellen is greater than Bob, Ellen must be in Bob’s right sub-tree. Bob’s right sub-tree is also a binary search tree. Now it happens that Ellen is in the root of Bob’s right sub-tree. Thus the search operation has located Ellen. If this wasn’t the case both right and left sub-trees of Bob’s right child would have been equal to null. This would have allowed the procedure to conclude that Ellen
Theorem

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

Many different binary search tree can be created for a given collection of items. In this slide we give examples of three binary search trees, which have the same items but different forms.

The search strategy can still be applied to each of these three different trees. However, it will work more efficiently on some binary search trees than others. For example, with the binary search tree (b) given in this slide, the search tree will have to inspect every node in the tree before locating Wendy. This same tree has in fact the same structure as a sorted linear linked list and offers no advantage over the list in efficiency.

In contrast, with the full tree (a) given in this slide, the search algorithm will inspect only the nodes that contain the values Jane, Tom and Wendy. This full tree is indeed an example of a balanced binary search tree and the height, 3, is equal to the value of $\log_2(7+1)$. If the number of items in the BST, $n$, is not equal to $2^i-1$ for some $i$, the height of the tree is $\lceil \log_2(n+1) \rceil$, the next integer above $\log_2(n+1)$. 
In this slide we give the implementation of the three additional operations for binary search trees. Each of these operations takes in input a new item or the search key’s value of an item and either adds the new item to the tree or deletes/retrieves the given item from the tree.

Note that none of these three access procedures takes as parameter a tree node, but just the item value or the value of its search key. The search algorithm, given before, takes instead as parameter the root of a given binary search tree and passes in each recursive call the roots of respective sub-trees. This searching process is performed in each of these three access procedures by auxiliary methods: the method `insertItem` for the access procedure insert, the method `deleteItem` for the access procedure delete, and the method `retrieveItem` for the access procedure retrieve.

Each of these three auxiliary methods is a recursive algorithm that goes through the tree structure. In fact each of these three methods takes as parameter a reference variable to a root Node. This parameter refers either to the root node of a given tree (when the method is first called), or to the root node of the sub-trees considered during the recursion.

Since these recursive methods are only auxiliary, they should be declared to be private to 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 the actual Java code implementation.
NOTE

With Binary Search trees, we need to restrict the user’s access to constructor operations to only “createEmptyTree” and “insert”.

The constructor “createBTree(rootItem, leftTree, rightTree )” must not be used by a client class. This will ensure that the Binary Search Tree retains its defined ordered property, as items may only be added to the tree by using the “insert” method.

The access procedure “insert’ may use createBTree and the other binary tree constructor methods as private auxiliary methods.
The **insert** Access Procedure

The main problem is to identify the correct place in a binary search tree to insert the new item.

We can use the search algorithm:

1. Imagine the new item already exists in the tree
2. Use the search algorithm to search for it.
3. The search algorithm thus stops always at an empty sub-tree.
4. This is the correct position to add the new item.
5. The new item is added as a new leaf:
   - i.e. by changing null reference of existing leaf to refer to the new item node.

**Example: insert Frank**

Suppose that we want to insert a new item 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 value “Frank”. Again, we are assuming that the node includes only one attribute, which works also as a search key. As a first step, imagine that we want instead to search for this item assuming that it is already present in the tree. We can do this using the search algorithm given in the previous slides. The search algorithm will first search the tree rooted at Jane, then the tree rooted at Bob, and then the tree rooted at Ellen. Because this last tree is empty, the search algorithm has reached its base case and will terminate with the report that Frank is not present.

If Frank were present it would have been in 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 algorithm, 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 to determine where in the tree to insert a new node always leads to an easy insertion. No matter what the new item is, the search will always terminate at an empty sub-tree. Thus, the search always tells us to insert the new item as a new leaf. Adding a new leaf requires changing appropriate reference variables in the parent node. The complexity of the “insert” algorithm is therefore virtually the same as that of the corresponding search algorithm.
Implementing “InsertItem” (1)

**pseudocode**

```java
TreeNode insertItem(TreeNode, newItem) // post: inserts newItem into the BST of which treeNode is the root // and returns treeNode.
{
    if (treeNode is null) {
        Create a new node and let treeNode reference it;
        Set the value in the new node to be equal to newItem;
        Set the references in the new node to null;
    }
    else if (newItem < treeNode.getItem( ))
        { treeNode.setLeft (insertItem(treeNode.getLeft( ), newItem)) }
    else
        { treeNode.setRight(insertItem(treeNode.getRight( ), newItem))}
    return treeNode; }
```

Note that we have made use here of a recursive call of the method insertItem. The main difference between this pseudo code and the pseudo code of the search algorithm is that in this case when we find the null node while we traverse the tree we don’t give the answer “item not found”, 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 the base case 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, which should be made the new root of the tree.

This case is also the general base case of a recursive call to the method. When the formal parameter treeNode becomes null, the corresponding actual argument is the “leftChild” or “rightChild” reference to the parent of the empty sub-tree; this reference has therefore value null. As in the case of an empty tree, the method will then return the reference to a new node, which must be set as either the parent’s left child or the parent’s right child. This is done by the method treeNode.setLeft to set the returned node to be the new left child of the parent, or by the method treeNode.setRight to set the returned node to be the new right child of the parent.
The deleteItem Access Procedure

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 as to preserve the ordering property.

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

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

1. Set the reference in N’s parent to null.
2. P
   N
   L
3. P
   N
   L

or

P
L

= the largest value in the left sub-tree of N

P
L

= 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 algorithm 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, there are three main cases to consider: 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. The second case is when 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.

The more difficult case is the last one, 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 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), exchange 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 the node N, as exchanging it with N will preserve the ordering and permit N to be deleted as either now a leaf node or a node with only one child, as above, preserving the binary search tree. In this slide we have chosen the second case, and replaced N with the smallest node in the right sub-tree of N.
Implementing “deleteItem”

```java
TreeNode deleteItem(TreeNode rootNode, Item) {
    // post: deletes Item from the BST with root rootNode and returns the
    // root node of the resulting tree
    if (rootNode is null)
        { throw TreeException; }
    else
    {
        if (rootNode.getItem( ) == Item)
            { rootNode = deleteNode(rootNode); }
        else if (rootNode.getItem( ) > Item)
        { newLeft = deleteItem(rootNode.getLeft( ), Item);
            rootNode.setLeft(newLeft); }
        else
        { newRight = deleteItem(rootNode.getRight( ), Item);
            rootNode.setRight(newRight); }
    }
    return rootNode; }
```

Pseudocode:

This slide gives the algorithm for the recursive method “deleteItem”, 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 algorithm given before, with the main difference that when the item is found, it needs to be deleted. The main issue is then how do we delete this item. The actual operation of deleting a node once it has been found is performed by the 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 item we want to delete. If not, this if case will be used at each recursive call, where the value of rootNode will be the root of the particular sub-tree considered in the recursion.
This method is part of the delete operation in a binary search tree. Once a given node has been found in the tree, this method is then applied on the node in the following way.

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 item has been deleted.

If the current node is not a leaf, then we check the other possible cases discussed in slide 11. The node can have just the left child, or just the right child or both children. In the first case, we would like that the left child becomes the new item in the tree. This is why we just return the left child. 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 11, we have to find the left most node of the right subtree of the current node, get its value, assign this (found) value to the current node (so to delete its current value), and delete the leftmost node in the right sub-tree. This is done by the four instructions given in the slide. At the end the new tree is returned.

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”

**pseudocode**

```java
Item findLeftMost(treeNode) {
    // post: returns the value of the left most node in the tree rooted at treeNode.
    if (treeNode.getLeft() is null) {
        return treeNode.getItem();
    } else {
        return findLeftMost(treeNode.getLeft());
    }
}

treeNode deleteLeftMost(treeNode) {
    // post: deletes the left most node in the tree rooted at treeNode.
    // returns the sub-tree of the deleted node
    if (treeNode.getLeft() is null) {
        return treeNode.getRight();
    } else {
        newChild = deleteLeftMost(treeNode.getLeft());
        treeNode.setLeft(newChild);
        return newNode;
    }
}
```

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.
Summary

- The binary search tree allows the use of a binary search-like algorithm to search for an item with a specified value.

- 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 item.