Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 7: Binary Search Trees (BST)

Reading materials

Goodrich, Tamassia, Goldwasser: Chapter 8

OpenDSA: Chapter 18 and 12

Binary search trees visualizations:

`http://visualgo.net/bst.html`

`http://www.cs.usfca.edu/~galles/visualization/BST.html`

Topics Covered

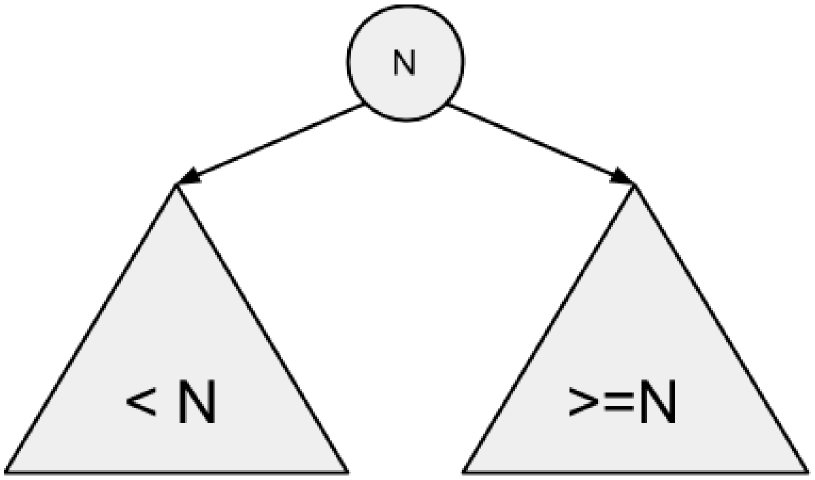
Joanna Klukowska
joannakl@cs.nyu.edu

1 Binary Search Trees (BST)

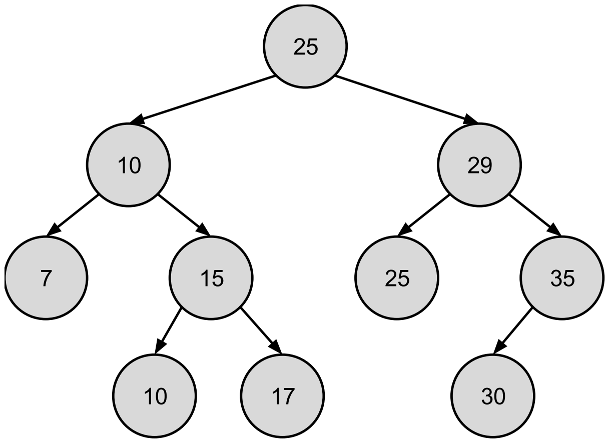
A **binary search tree** is a binary tree with additional properties:

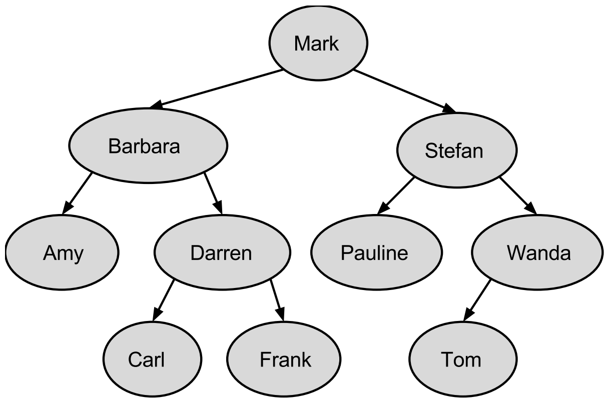
- the value stored in a node is greater than or equal to the value stored in its left child and all its descendants (or left subtree), and
- the value stored in a node is smaller than the value stored in its right child and all its descendants (or its right subtree).

(Well, the right and left really do not matter as long as the implementation is consistent. If we switch the sides, then traversing the tree ”in-order” will yield reverse order of the nodes.)

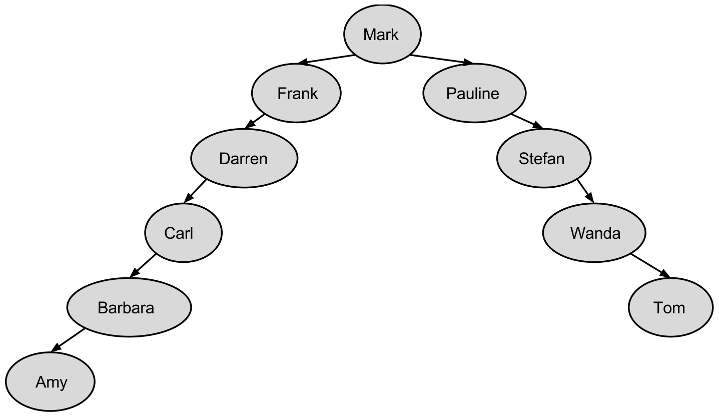


Here are examples of some, pretty well balanced, binary trees. They are also binary search trees.





The binary search trees properties do not prevent trees that are very skinny and have many levels (which will be the source of bad performance for all the algorithms).



In the rest of these notes we will discuss several algorithms related to binary search tree. All of them can be implemented using iterative or recursive approach. For trees, unless the overhead of the method call in the programming language is very large, the time performance of recursive implementations should be similar to the performance of iterative implementations.

2 Binary Search Tree Node

The node of a binary search tree needs to store a data item and references to its children. Binary search tree organization requires that the data items stored in the node can be compared to one another. In Java, that means that we want the type that is used as the data type to implement `Comparable` interface. Here is a generic node implementation for BST.

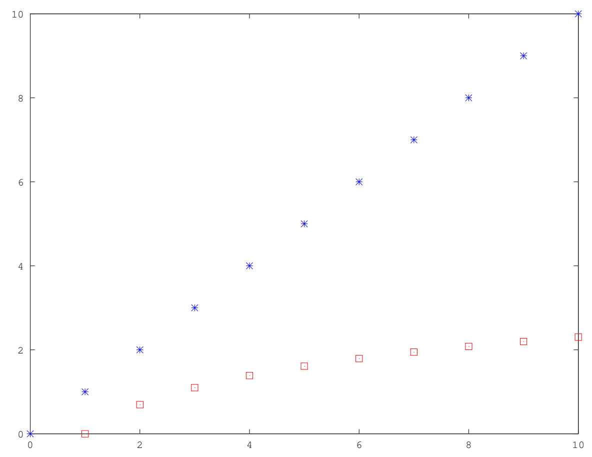
```
class BSTNode <T extends Comparable <T> >
                implements Comparable < BSTNode<T> > {

    private T data;
    private BSTNode <T> left;
    private BSTNode <T> right;

    public BSTNode ( T data ) {
        this.data = data;
    }

    public BSTNode (T data, BSTNode <T> left, BSTNode <T> right ) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
    ...
    //desired setters and getters
    ...
    public int compareTo ( BSTNode <T> other ) {
        return this.data.compareTo ( other.data );
    }

}
```

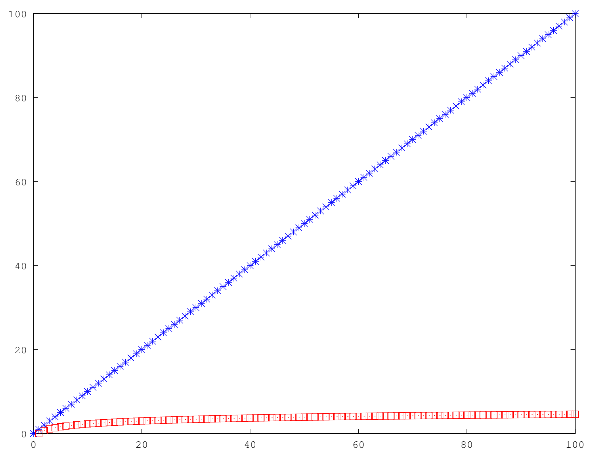
3 Searching for an Item in a Binary Search Tree

The name of the Binary Search Tree suggest that it has something to do with a binary search algorithm. It turns out that the method for searching for an item in a BST is almost the same as searching for an item in a sorted list using binary search method.

3.1 Binary Search Algorithm

If data stored in an array is not sorted, the only way to determine if an item is there or not is by looking at every single array location. That is $O(N)$ operations assuming that we have N array locations. If the array happens to be sorted, the search can done much faster by using binary search algorithm. That reduces the complexity to $O(\log N)$operations assuming that we have N array locations. The difference between N and $\log N$ may not be very significant for small values of N, but becomes very large as N gets larger and larger. See the plots below:





As N gets larger, the plot of N keeps growing while the plot of $\log N$ stays relatively flat.

3.1.1 Recursive Approach

Binary search algorithm can be easily described recursively: We check of the item we are looking for is smaller than the middle element of the array. If it is, then we discard the upper half of the array and repeat the search on the lower half. If it isn't, then we discard the lower half of the array and repeat the search on the upper half. The $O(\log N)$ performance comes from the fact that after each comparison we can discard half of the remaining elements. Remember that in a linear search, after each comparison we are able to discard only one element (the one that we just looked at).

Here is the pseudocode for the recursive binary search algorithm.

```
int binarySearch ( array, key, minIndex, maxIndex )
    if (maxIndex < minIndex )
        key is not in the array (return -1 to indicate key not found)
    else
        midIndex = (minIndex + maxIndex) / 2
        if ( array[midIndex] > key )
            return binarySearch (array, key, minIndex, midIndex-1)
        else if ( array[midIndex] > key )
            return binarySearch (array, key, midIndex+1, maxIndex)
        else

            array[midIndex] is the element we were looking for (return midIndex)
```

3.1.2 Iterative Approach

The iterative approach to binary search is remarkably similar to the recursive method. Here is the pseudocode for the iterative binary search algorithm.

```
int binSearch ( array, key )
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
minIndex = 0
maxIndex = size of array - 1
while (maxIndex >= minIndex )
   midIndex = (minIndex + maxIndex) / 2
   if ( array[midIndex] > key )
      maxIndex =  midIndex - 1
   else if (array[midIndex] < key )
      minIndex = midIndex + 1
   else
      return midIndex;
return -1 to indicate key not found
```

Instead of making recursive calls, we change the values of the minIndex and maxIndex for next iteration (this effectively discards half of the remaining elements on each iteration).

3.2 `contains()` Method for a Binary Search Tree

Having reviewed the binary search algorithm it should be very easy to write an implementation of `contains()` method for a BST. Given an item, we want to determine if there is a node in the tree that stores an item that is equal to the one in a parameter. This algorithm is, once again, easy to state recursively: If the current node is empty, then we failed in finding the item. If the item is smaller than the one at a current node, than we should repeat the search with the node's left child. If the item is larger than the one at a current node, then we should repeat the search with the node's right subtree. Otherwise, we found the item in the current node.

Here is the pseudocode for recursive algorithm.

```
boolean recContains ( item, currentNode )
    if currentNode == null
       return false
    else if (  item < currentNode.item )
       return recContains( item, currentNode.left )
    else if (item > currentNode.item )
       return recContains (item, currentNode.right )
    else
       return true
```

The iterative approach is very similar and you should try to write it as an exercise.

The above method should be used with a wrapper that starts the search at the root of the tree.

4 Inserting a Node into a Binary Search Tree

Adding new nodes to the tree always creates an new leaf. To insert a new node, we need to navigate through the tree to find a right place for it. Starting at the tree, we decide if we need to go left or right depending on the value stored in the root node, then the process is repeated for the subtree, and so on, until we find a node that has a null reference in the correct place to add a node with our new data.

So far, it seems that the recursive algorithm should look something like this

```
recAdd ( BSTNode<T> node, T newData )
    if ( newData < node.data )
       recAdd ( node.left, data )
```

```
        else if
           recAdd (node.right, data )
```

But there is a problem here: we do not have a base case, so when are we actually going to add the node? The solution is to add a base case that creates a new node when the node reference in the parameter is null. To get this new node attached to the tree, we need to return its value and turn the recursive calls into assignment statements.

```
    BSTNode<T> recAdd ( BSTNode<T> node, T newData )
        if ( node == null)
           create new node containing newData
           return reference to that node
        if ( newData < node.data )
           node.left = recAdd ( node.left, newData )
        else
           node.right = recAdd (node.right, newData )
        return node;
```

As usual, the recursive method should be a private method and we should have a public method that calls the recursive one with the root node.

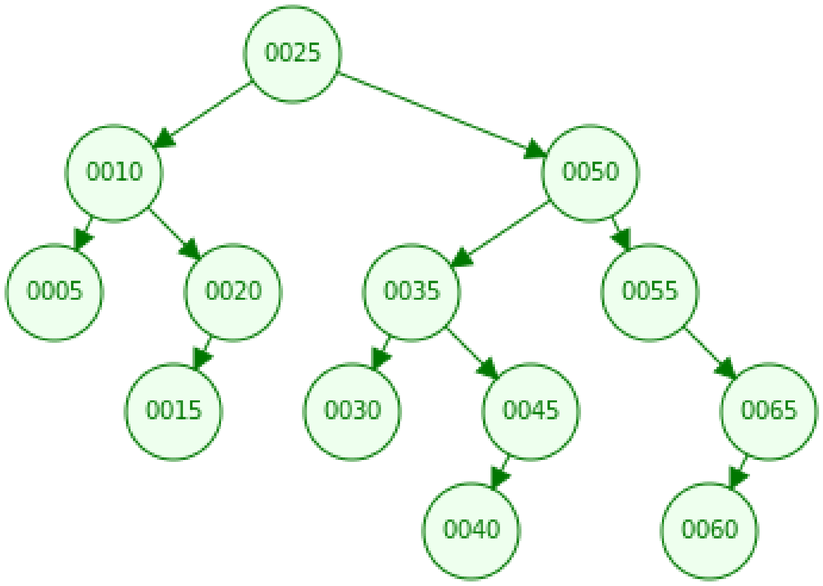
```
    add ( T newData )
        root = recAdd( root, newData );
```

4.1 Order of Insertions

Depending on the order of insertions, the tree may stay nicely balanced (bushy and shallow) or it can become very unbalanced (skinny and deep).

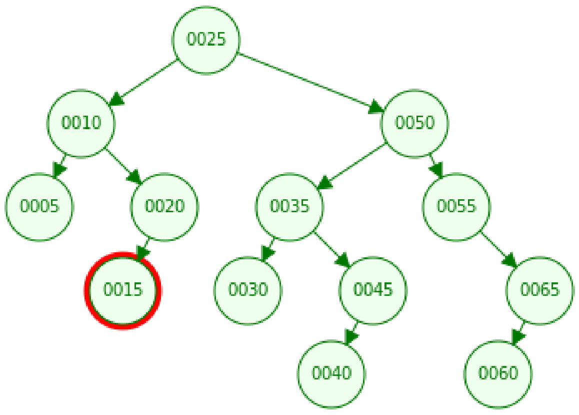
5 Removing a Node from a Binary Search Tree

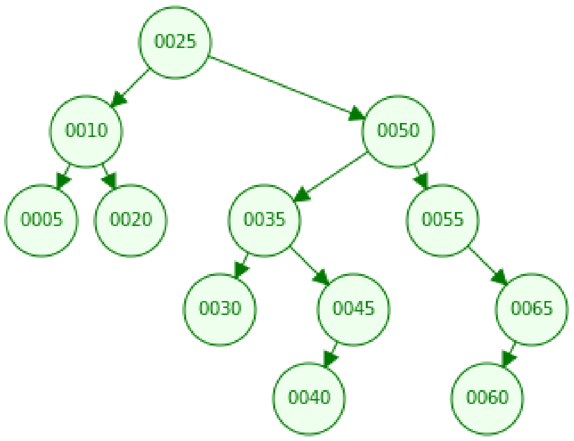
The method for removing a node from a binary search tree is the most involved from all the BST methods that we looked at so far. We will consider it in a case by case basis starting from the simplest one. We will work with the following tree in this section.



5.1 Removing a Leaf Node

The leaf nodes are the ones that have no children (both left and right references are null). If we need to remove a leaf node, all we need to do is to disconnect it from its parent. If we remove node labeled 15 from our tree, we first find it (figure on the left), and then disconnect it from the node with label 20 (figure on the right).

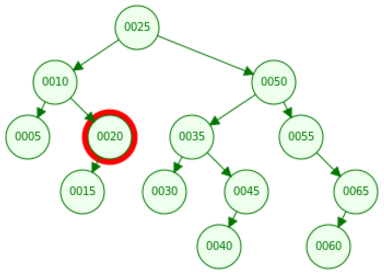


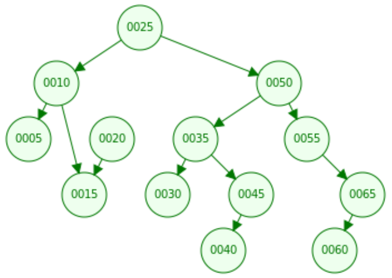


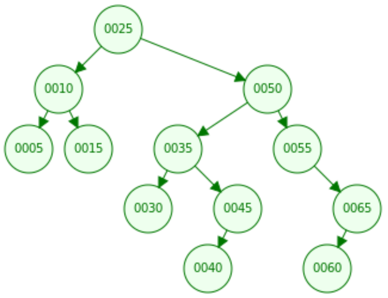
The tricky part is to know who the parent of node labeled 15 is once we find it. In this, and all the other cases, as we are searching for the node, we will keep a reference to its parent.

5.2 Removing a Node with One Child

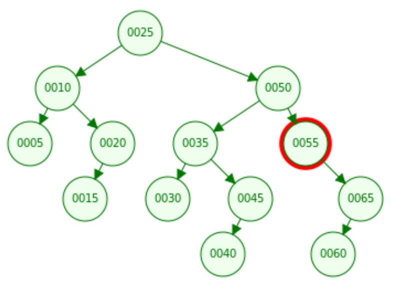
The nodes with one child have either left or right reference pointing to a subtree, but the other one is pointing to null. When we delete a node like this, its parent adopts its child subtree (the entire branch is just pulled up). If we remove node labeled 20 from out tree, we first find it (figure on the left), and then connect that node's parent's right reference to that node's left reference (figure in the middle). By removing 20, its parent has a one free reference so it can take on its single "grandchild" as its own child. After that 20 is no longer referenced (or pointed to) by anything, so it is effectively removed from the tree (figure on the right).

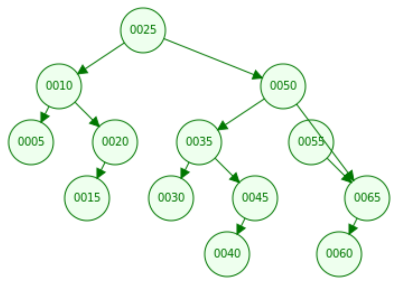


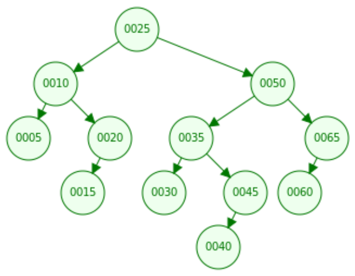




It does not matter if there is an entire (possibly large) subtree rooted at the node that we need to remove, as long as it is only one of the two possible subtrees. It also does not matter if it is a left or a right subtree that points to null. The procedure is still the same. Removing 55 from our tree is done in the same fashion as removing 20.







5.3 Removing a Node with Two Children

When the node to be removed has two children they cannot be simply adopted by its parent - the parent has a limit of having two children so, in general, we cannot assign two extra children to it. Even if its other reference was null, the two "orphaned" subtrees both contains nodes that are smaller/larger (but not both) that the parent of the node to be removed, so they cannot simply become its left and right child.

So what can we do?

If we remove node labeled 50 from our tree, we need to rearrange the nodes in the entire right subtree of the tree rooted at 25, so that 1) 50 is no longer on the tree, 2) the remaining tree maintains the BST properties. Depending on the shape of the two subtrees rooted at the removed node, there might be different solutions that appear to be the "simplest" ones. But, when the remove algorithm has to decide which options to use, we do not want it to have to analyze the entire subtrees in order to chose which approach to take. Fortunately, there is one general approach that works independent of the shape of the two subtrees. We will look for a suitable replacement for 50 so that no other nodes (well, almost no other nodes) in the two subtrees need to be changed. Such suitable replacements are either the largest node in the left subtree, or the smallest node in the right subtree. Looking at the shapes of the subtrees and the values stored in them, we realize that the largest node in the left subtree is the "rightmost" node in the left subtree. Similarly, the smallest node in the right subtree is the "leftmost" node in that subtree.

We need to pick one of these suitable candidates. It does not matter which one, so we will go with the smallest node in the left subtree. It is easy to find it: we take the left from the node to be removed (i.e., 50) and then keep on going right until we cannot go right anymore.

The algorithm for finding the rightmost node in a left subtree of a given node n is as follows.

```
T getPredecessor ( BSTNode<T> n )
    if (n.left == null)
        that should not happen
        return error condition
    else
        BSTNode <T> current = n.left
        while (current.right != null )
            current = current.right
        return current.data
```

In our tree, this will yield 45, so we can use the data from node labeled 45 to replace the data from the node labeled 50. This effectively removes 50 from the tree, but we now have two nodes with label 45 containing the same data. We need to remove the second 45!!!

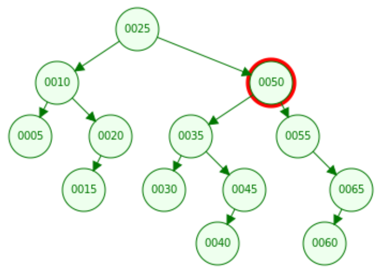
Removing the rightmost node of the left subtree is guaranteed to fall under one of the simpler cases: either it is a leaf node, or it has only a left child. It is guaranteed not to have a right child, otherwise it would not have been a rightmost child of the subtree.

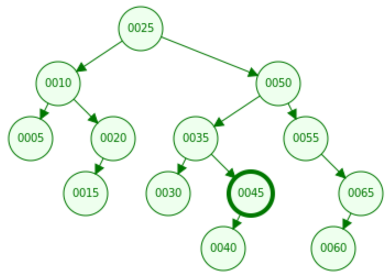
The following figures show the steps of the removal of 50:

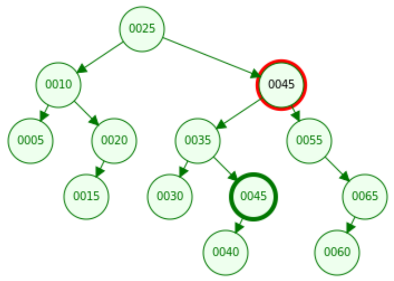
1. find the node with label 50
2. find its predecessor in the tree (the rightmost node it its left subtree), that's 45

Joanna Klukowska
joannakl@cs.nyu.edu

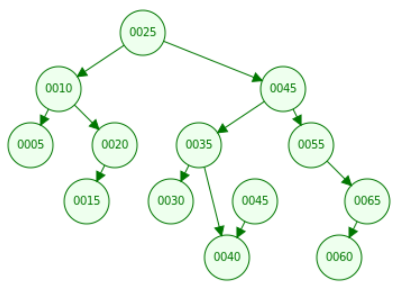
3. use 45 to replace 50

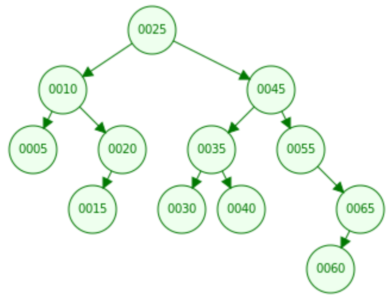






and then remove 45 using the leaf or one child approach





5.4 Recursive Implementation

The implementation of the remove operation involves several smaller methods. The only recursion in it is involved in finding the node that we want to remove. We will use a top-down approach to discuss itand leave details for later implementation.

The client of our BST class needs to be able to call a remove method on a tree passing to it a parameter that indicates the data item that should be removed from the tree (data, not the node, because technically the client does not know anything about the node structure, they just add and remove the data from a tree). Our client side method remove the data item from a tree if it is there, an leaves the tree unchanged otherwise.

```
remove ( T item )
    root = recRemove( root, item )
```

The client side remove method does not do much: it calls the recursive remove that does the actual work (well, not all of it, recursive remove needs some help from other methods too). Our recursive remove returns the reference to the tree that it just modified. The reasons for it are similar to the reasons why our recursive add method returned the reference to the tree that it modified. It also allows us to not have to worry about the special case of removing the root node. The recursive remove first locates the node and then removes it and returns the reference to the new, possibly modified, tree.

```
BSTNode <T> recRemove (BSTNode <T> node, T item )
    if (node == null)
       do nothing, the item is not in the tree
    else if ( item < node.data )
       node.left = recRemove ( node.left, item)  //search in the left subtree
    else if ( item > node.data )

       node.right = recRemove ( node.right, item ) //search in the right subtree
    else //found it!
```

```
            //remove the data stored in the node
            node = remove( node)
        return node
```

The actual removal depends on which of the previously discussed cases we are in.

```
    BSTNode<T> remove ( BSTNode<T> node )

        if (node.left == null )
            return node.right
        if (node.right == null )
            return node.left
        //otherwise we have two children
        T data = getPredecessor ( node )
        node.data = data
        node.left = recRemove ( node.left, data )
        return node
```

6 Balanced Binary Search Trees

We often discussed the fact that the performance of insertion and removal operations in a binary search tree can become very poor (comparable to linked list) if the binary search tree is not nice and bushy. We will look into ways of avoiding such degenerative situation.

For the discussion of balanced trees we will use the following two definitions:

Full binary tree A binary tree in which all of the leaves are on the same level and every non-leaf node has two children. If we count levels starting at zero at the root, then the full binary tree has exactly 2^L nodes at level L.

Complete binary tree A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible.

6.1 Adding a `balance()` Method

The textbook describes an approach of balancing a tree. The burden and responsibility is put on the user of the data structure to periodically call a balance() method that takes the current tree and turns it into a nice and bushy structure with the height of approximately $\log_2 N$ where N is the number of nodes in the tree.

There are good and bad approaches to implementing such a method. See the section 8.8 in the textbook for details of several different approaches. The best one uses our favorite tool of recursion and inserts existing nodes into a new tree in an order that guarantees a nice balanced tree. **The computational complexity of this balancing of the tree is O($N \log_2 N$).**

The following pseudocode describes the steps needed to implement a balance method for the tree.

```
    balance()

        perform inorder traversal of the current tree and save the nodes/elements
          nodeList = currentTree.inorderTraversal()
        create a new empty binary search tree called newBST
          newBST = new BST();
        call method insertNodes on the newBST as follows:
          newBST.insertNodes( nodeList, 0, nodeList.size()-1 )


    isertNodes( nodeList, first, last )

        if (first == last)
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
            this.insert( nodeList[first] )
        else if (first+1 == last)
            this.insert( nodeList[first] )
            this.insert( nodeList[last] )
        else
            mid = (first + last) / 2
            this.insert ( nodeList[mid] )
            this.insertNodes ( first, mid-1 )
            this.insertNodes ( mid+1, last )
```

The `balance()` method performs an inorder traversal saving all the nodes in a list, creates a new empty tree and then adds all the nodes into the tree one by one using the recursive method `insertNodes`. The recursive `insertNodes` method continues by finding the best candidate for the root of the tree (that is the middle element) and then finds the best candidates for roots of left and right subtrees of the root (the middle of the first half and the middle of the second half), etc.

The big **advantage** of this approach is that the user of our binary tree can decide when it is worth to spend the extra computation time to balance the tree. Unfortunately, this is also the **drawback** - the user has to anticipate when it makes sense to balance the tree. If balancing is done too frequently, then that process adds a lot of computation to the tree processing. If it is not done frequently enough, then the tree may become a linked list.

Additionally, the cost of running the balance() method is exactly the same for a tree that is almost balanced and for a tree that looks like a linked list.

7 Self-Balancing Binary Search Trees

Another solution (not discussed in the book) is implementation of the **self-balancing binary search trees**. The add and remove operations of the tree are "willing to tolerate" some level of imbalance (i.e. the tree does not have to be complete), but as soon as this is violated, these methods readjust the tree to keep it balanced. Such readjustments do not require the same type of procedures as re-balancing the entire tree. They can be done **locally**, because the imbalance was caused by most recent addition and removal.

Keeping the binary search tree balanced becomes up to the tree itself and not its user.

There are several different approaches to implementing self-balancing binary search trees.

7.1 AVL Tree

The AVL tree (named after its inventors Adelson-Velskii and Landis) is the original self-balancing binary search tree. The computational time of insertions and removals (even with re-balancing) remains $O(\log_2 N)$. **The balancing is done based on height: the heights of the two subtrees of any given node cannot differ by more than 1**. This does not mean that the tree is complete, but it is close enough to complete that the performance of insertions and removals stays even in the worst case at $O(\log_2 N)$, not $O(N)$ as was the case with the ordinary BST. This is achievable by applying one or two (relatively) simple rotations at the nodes that become unbalanced.

Wikipedia has an interesting article about AVL trees with good explanation of rotations and some pseudocode: http://en.wikipedia.org/wiki/AVL_tree

And you can animate creation of the tree and re-balancing at http://www.cs.usfca.edu/~galles/visualization/AVLtree.html.

Reminder: The **height** of any node is calculated from the bottom of the tree up. The leaves are all at height 0 (or 1 according to some definitions). The height of any internal node is calculated recursively as the maximum of the heights of its children + 1.

After an insert or remove operations we need to check all the nodes in the tree from the one added/removed up to the root to determine if any imbalance was created. Because the re-balancing is performed on each operation, the changes are always small and they are only needed along the path from the root to the newly created or removed node. In fact, there are only 4 different ways in which the node can be out of balance (according to the rule that requires that the heights of its two subtrees differ by no more than 1).

Joanna Klukowska
joannakl@cs.nyu.edu

7.1.1 When balancing is necessary

In order to keep the tree balanced, we need to make adjustments when they become necessary. This is decided based on the height difference of the subtrees of a given node. The balance factor is the difference between the heights of the left and right subtrees.

```
int balanceFactor ( Node n )
    if ( n.right == null )
       return -node.height
    if ( n.left == null )
       return node.height
    return node.right.height - node.left.height;
```

In order to compute the balance factor and determine which rotation (if any) is needed at that node, we need to know the heights associated with every node. When we update the height of any given node, we make an assumption (easily satisfiable) that the heights stored in the children of the node are correct.

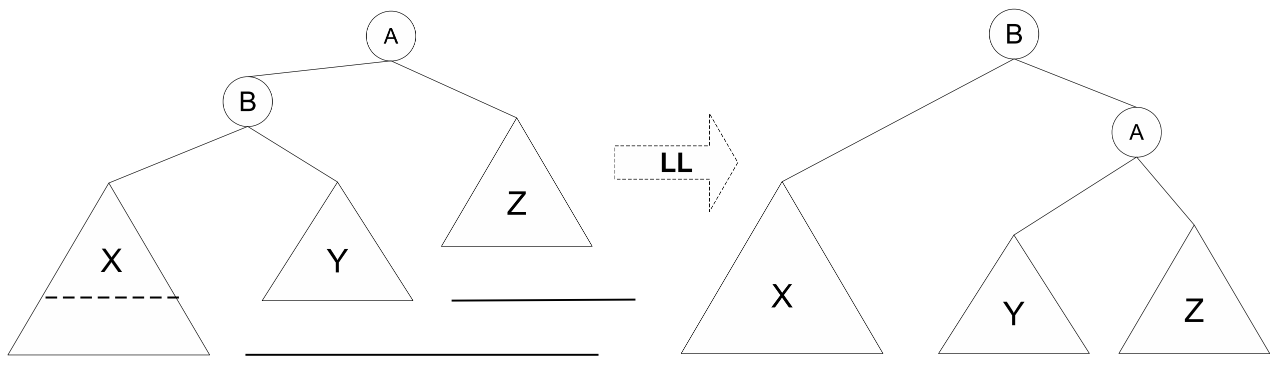
```
void updateHeight ( Node n )
    if node is a leaf
       node.height = 0 //this is sometime set to 1
    else if node.left == null
       node.height = node.right.height + 1
    else if node.right == null
       node.height = node.left.height + 1
    else
       node.height = max( node.right.height, node.left.height) + 1
```

When we add or remove a node in an AVL tree, we need to go back up to the root (along the path from the modified node to the root) and check one by one if any of the nodes requires re-balancing. If the call to `balanceFactor` returns -2 or 2, then we need to perform one of the four possible rotations described below. Notice that the call to balanceFactor should never return a value smaller than -2 or larger than 2 if the tree is maintained properly.

It is important to note that in all of the rotations below we change the structure of the tree (relink the nodes) and the data is never copied from one node to another node.

LL imbalance / LL rotation The imbalance occurs at a node A (note that node A is not the root of the whole tree, it is just a node at which imbalance occurs, there might be a huge tree above it): its left subtree has two more levels than its right tree. In the left subtree of A (i.e. subtree rooted at B) either both subtrees of B have the same height, or the left subtree of B has height one larger than the right subtree.

Fix: rotate the subtree to the right. The right subtree of B becomes the left subtree of A.



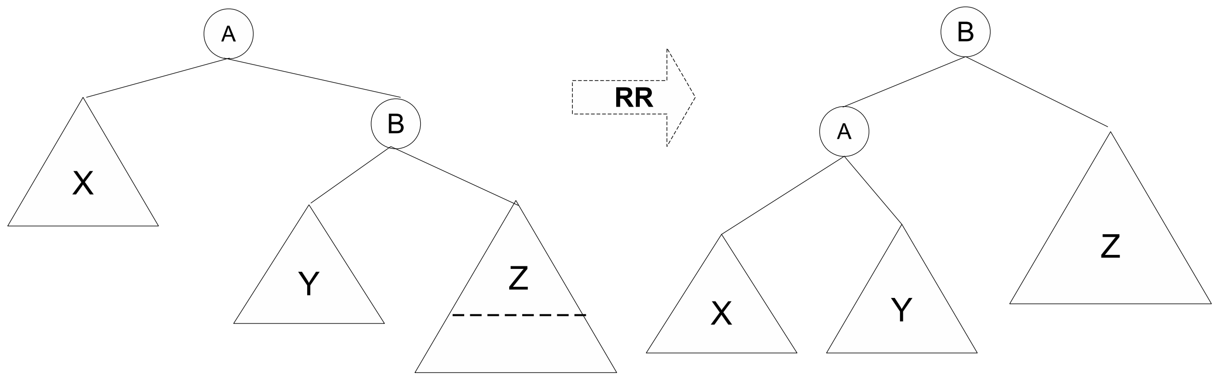
Joanna Klukowska
joannakl@cs.nyu.edu

The following pseudocode describes steps required to perform an LL rotation

```
//returns a reference to the new root of the subtree after
//the LL rotation has been performed

Node  balanceLL ( Node A )

    Node B = A.left

    A.left  = B.right
    B.right  = A

    updateHeight ( A )
    updateHeight ( B )

    return B
```

RR imbalance / RR rotation The imbalance occurs at a node A: its right subtree has two more levels than its right subtree. In the right subtree of A (i.e. subtree rooted at B) either both subtrees of B have the same height, or the right subtree of B has height one larger than the left subtree.

Fix: rotate the subtree to the left. The left subtree of B becomes the right subtree of A.

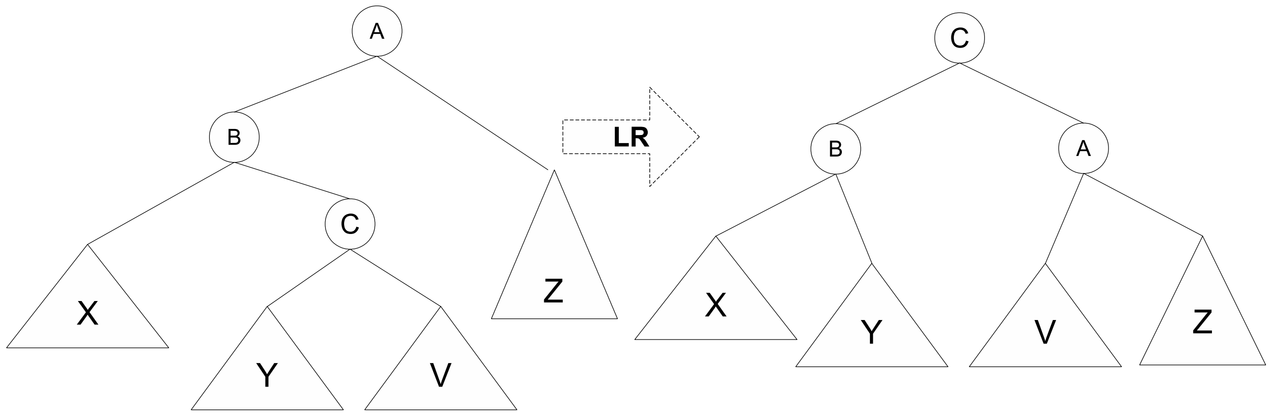


The code that performs this rotation is very similar to the one for the LL rotation.

LR imbalance / LR rotation The imbalance occurs at a node A: its left subtree has two more levels than its right subtree. In the left subtree of A (i.e. subtree rooted at B) the right subtrees of B (whose root is C) has height one larger than the left subtree of B.

Fix: rotate to the left at node B and then rotate to the right at node A.

Joanna Klukowska
joannakl@cs.nyu.edu

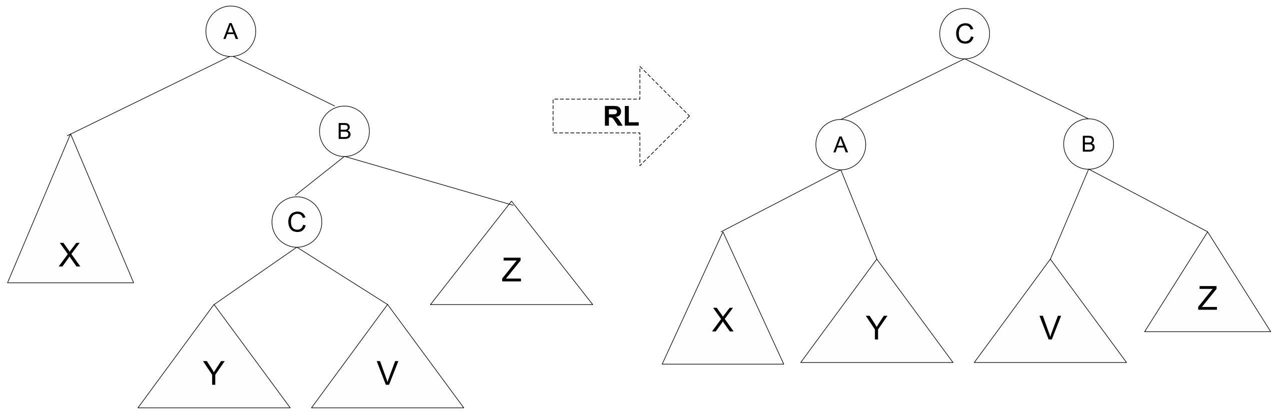


The following pseudocode describes steps required to perform an LR rotation

```
//returns a reference to the new root of the subtree after
//the LR rotation has been performed

Node  balanceLR ( Node A )

    Node B = A.left
    Node C = B.right;

    A.left  = C.right
    B.right = C.left
    C.left  = B
    C.right = A

    updateHeight ( A )
    updateHeight ( B )
    updateHeight ( C )

    return C
```

RL imbalance / RL rotation The imbalance occurs at a node A: its right subtree has two more levels than its left subtree. In the right subtree of A (i.e. subtree rooted at B) the left subtrees of B (whose root is C) has height one larger than the left subtree of B.

Fix: rotate to the right at node B and then rotate to the left at node A.



The code that performs this rotation is very similar to the one for the LR rotation.

Joanna Klukowska
joannakl@cs.nyu.edu

7.2 Red-Black Trees

The red-black tree (invented by R Bayer) is another self-balancing search tree. The computation time of insertions and removals is as well $O(\log_2 N)$, but the balancing is done differently than with the AVL trees. All nodes are assigned one of the two colors (red or black, hence the name) and the coloring has to follow certain properties. If these properties are violated the insert or remove method has to fix the tree.

For the list of all the properties and their consequences see the Wikipedia article at `http://en.wikipedia.org/wiki/Red-black_tree`.

And the animation of creation and modification of a tree is at `http://www.cs.usfca.edu/~galles/visualization/RedBlack.html`.

Java's library `TreeSet<E>` and `TreeMap<K,V>` classes are implemented using the Red-Black trees:

- TreeSet (`http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html`),
- TreeMap (`http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html`).