Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 4: Trees, Binary Trees and Binary Search Trees (BST)

Reading materials

Dale, Joyce, Weems: Chapter 8

OpenDSA: Chapter 2

Liang: only in Comprehensive edition,

Binary search trees visualizations:

http://visualgo.net/bst.html

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

Topics Covered

1 Trees 3

2 Binary Trees 5

3 Binary Search Trees (BST) 6

4 Binary Search Tree Node 7

5 Computing the Size of a Tree 7
- 5.1 Recursive approach . . . 8
- 5.2 Iterative approach . . . 8

6 Searching for an Item in a Binary Search Tree 8
- 6.1 Binary Search Algorithm . . . 9
  - 6.1.1 Recursive Approach . . . 9
  - 6.1.2 Iterative Approach . . . 9
- 6.2 `contains()` Method for a Binary Search Tree . . . 10
- 6.3 `get()` Method for a Binary Search Tree . . . 10

7 Traversing a Binary Tree 11
- 7.1 InOrder Traversal . . . 11
  - 7.1.1 Recursive Approach . . . 11
  - 7.1.2 Iterative Approach . . . 12
- 7.2 PreOrder Traversal . . . 13
  - 7.2.1 Recursive Approach . . . 13
  - 7.2.2 Iterative approach . . . 14
- 7.3 PostOrder Traversal . . . 14
  - 7.3.1 Recursive Approach . . . 14

Joanna Klukowska
joannakl@cs.nyu.edu

8 Inserting a Node into a Binary Search Tree **15**

8.1 Order of Insertions 15

9 Removing a Node from a Binary Search Tree **15**

9.1 Removing a Leaf Node 16

9.2 Removing a Node with One Child 16

9.3 Removing a Node with Two Children 17

9.4 Recursive Implementation 18

Joanna Klukowska
joannakl@cs.nyu.edu

1 Trees

A **tree** is a structure in which each node can have multiple successors (unlike the linear structures that we have been studying so far, in which each node always had at most one successor).

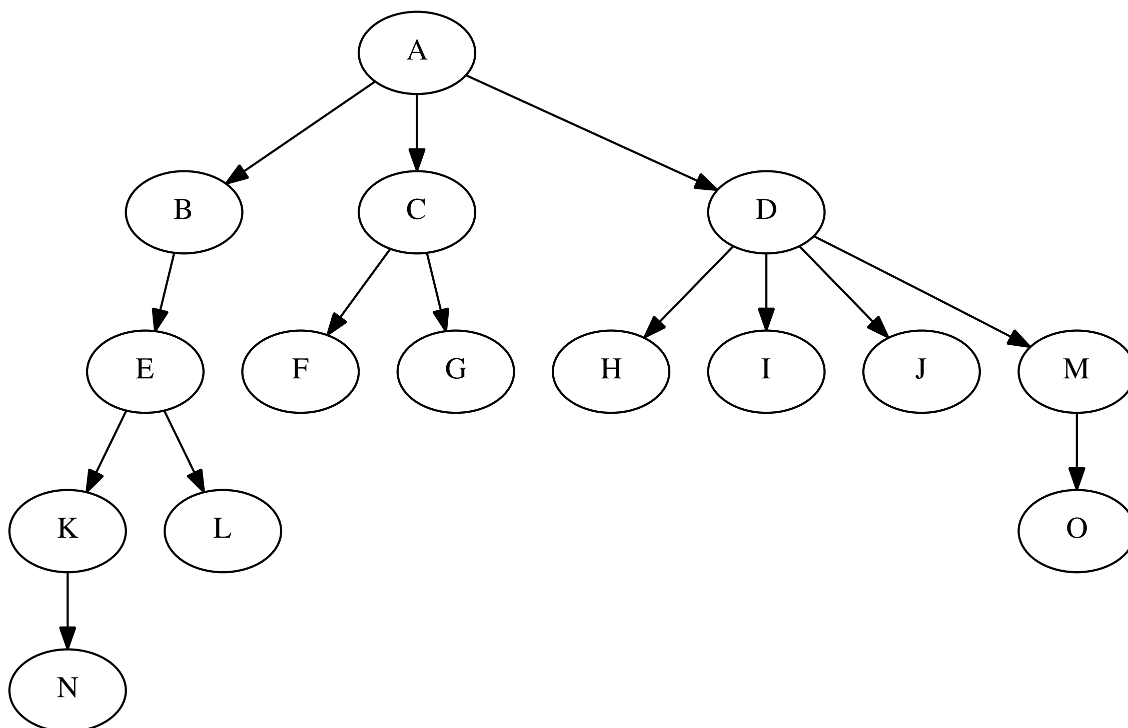
The first node in a tree s called a **root**, it is often called the top level node (YES, in computer science root of a tree is at the top of a tree). In a tree, there is always a unique path from the root of a tree to every other node in a tree - this has an important consequence: there are no cycles in a tree (think of a cycle as a closed path that allows us to go in a cycle infinitely many times).

Any node within a tree can be viewed as a root of its own **subtree** - just take any node, cut of the branch that connects above to the rest of a tree, and it becomes a root with a smaller (possibly empty) tree of its own.

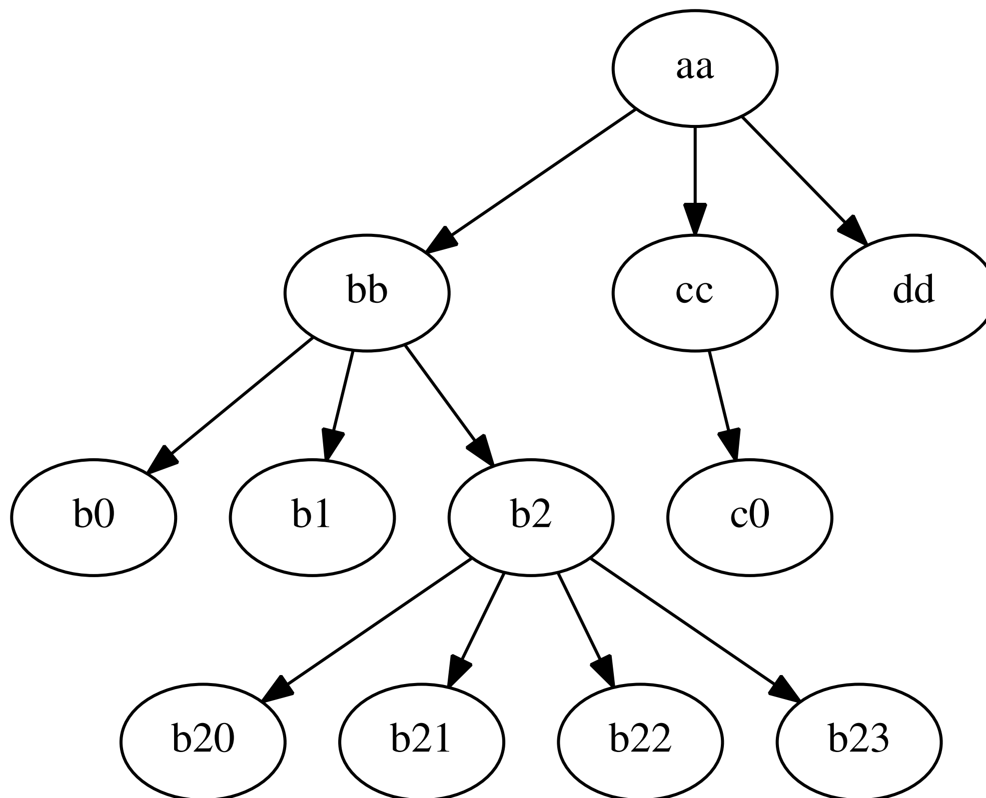
The nodes at the end of each path that leads from root downwards are called **leaves**. The other way to think about it is that leaves are the nodes that do not point to any other nodes (or point only to null). In a linear structure there was only one such node indicating the end of the list. In trees we have many such nodes.

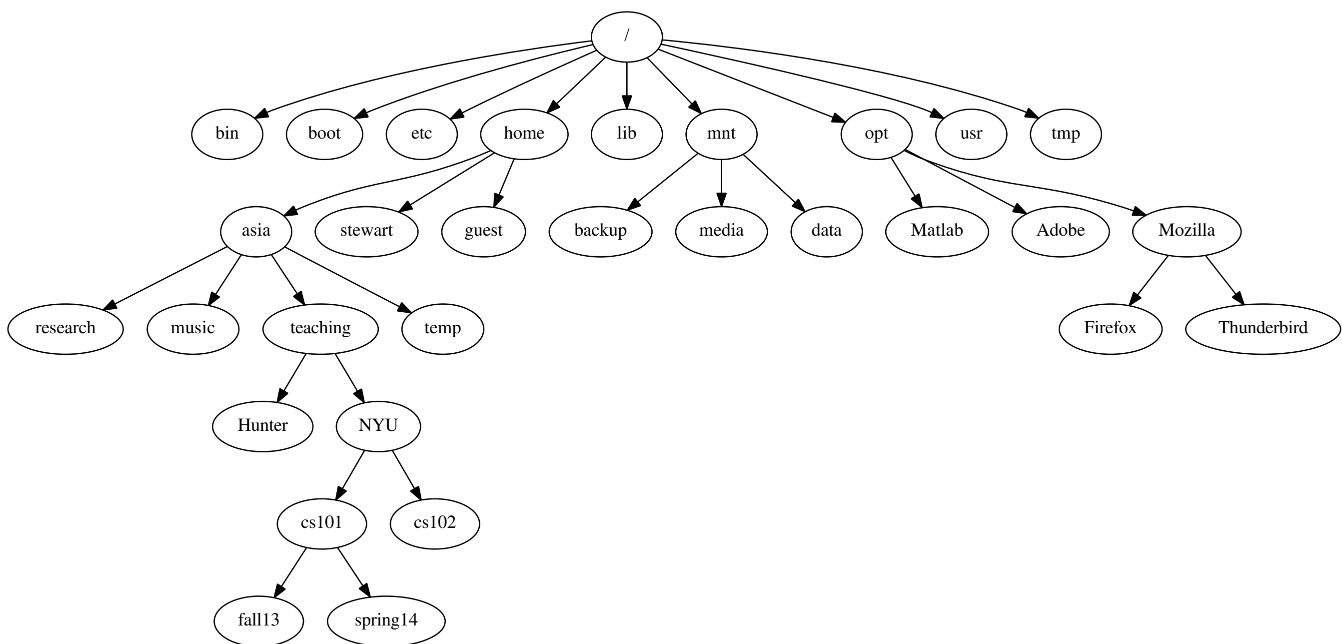
Given a node in a tree, its successors (nodes connected to it in a level below) are called its **children**. **Descendants** of a node are its children, and the children of its children, and the children of the children of its children, and ... - all the nodes below that are connected to that node.

Given a node in a tree, its predecessor (node that connects to it in a level above - there is only one such node) is called its **parent**. **Ascendants** of a node are its parent, and the parent of the parent, and ... - all the nodes along the path from itself to the root.



Joanna Klukowska
joannakl@cs.nyu.edu

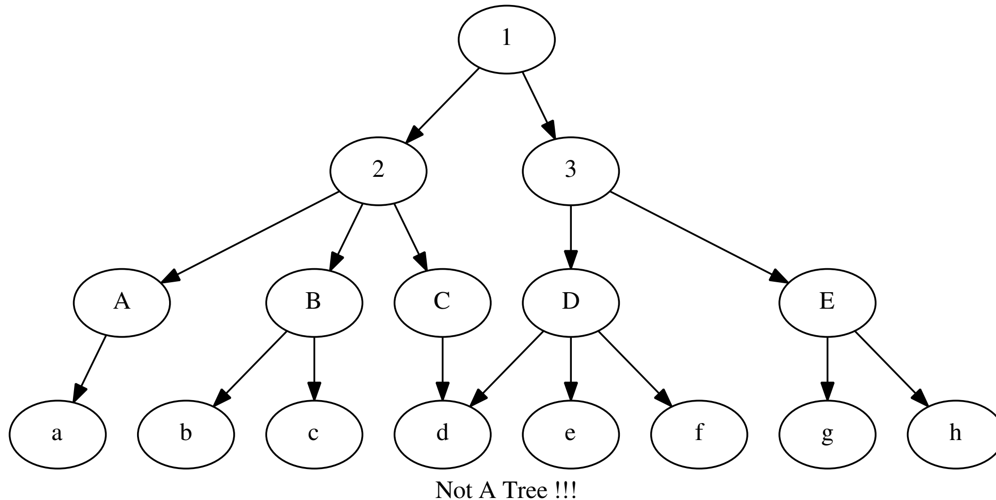




Linux directory structure

Joanna Klukowska
joannakl@cs.nyu.edu

THIS IS NOT A TREE !!! Why?



Not A Tree !!!

2 Binary Trees

A **binary tree** is a special kind of tree in which each node can have at most two children: they are distinguished as a **left child** and a **right child** . The subtree rooted at the left child of a node is called its **left subtre**e and the subtree rooted at the right child of a node is called its **right subtree**.

Level of a node refers to the distance of a node from the root. Root is a distance zero from itself, so the level of the root is 0 (or root is at level zero in the tree). Children of the root are at level 1, "grandchildren" or children of the children of the root are at level 2, etc. The **depth** of a tree is equal to the level of the deepest node.

The **height** of a tree seems to have a slightly different definition depending on the source. I think a most common definition is that the height is one more than the largest level of any node in the tree (i.e., it is equal to the depth + 1). In our textbook, the term depth is not defined, and the height is defined as the largest level of any node (i.e., it is equal to the depth of the tree).You can pick your own - just be consistent. We will be using the height of the tree in discussion about the performance. The **height of any node** is its distance from the furthest leaf (again the difference of one happens when we view the leaves as having the height 0 or the height 1). The height of a tree is simply the height of its root.

In binary trees there are at most 2^L nodes at level L. If a given level L has exactly 2^L nodes, then it is considered **full**. If level L is not full, then none of the levels $> L$ can be full. A **tree is full** if every level in the tree is full. A **tree is complete** if every level, except the last one, is full and the last level is filled from left to right.

Given N nodes, the "shortest" binary tree that we can construct has $\lfloor \log_2 N \rfloor + 1$ levels. We will come back to this idea when we talk about efficiency of trees.

Given N nodes, the "tallest" binary tree that we can construct has $N-1$ levels. Why?

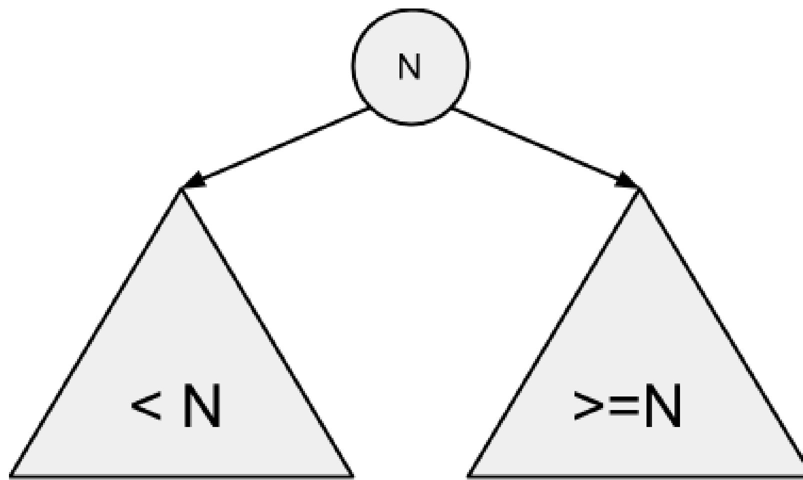
Joanna Klukowska
joannakl@cs.nyu.edu

3 Binary Search Trees (BST)

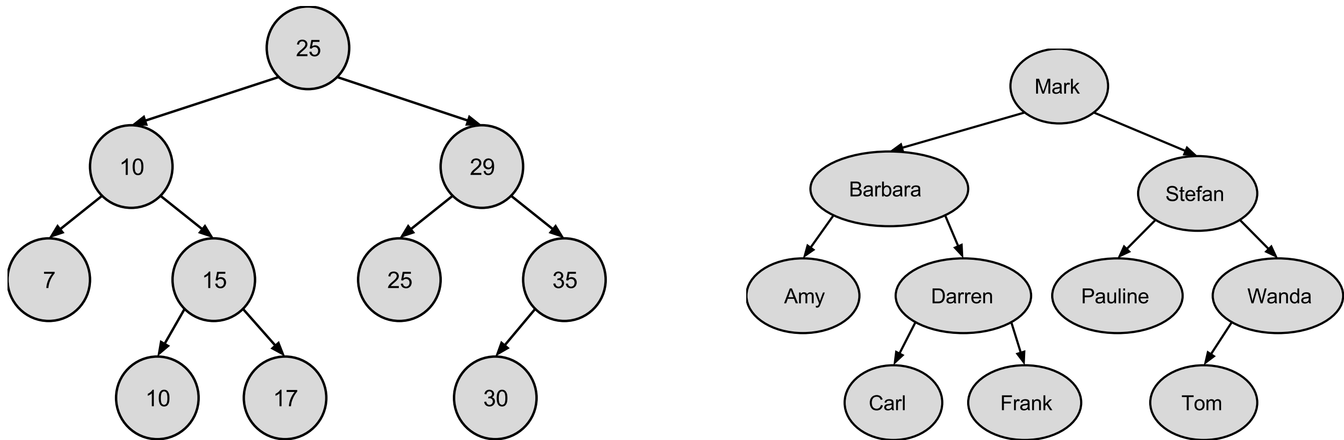
A **binary search tree** is a binary tree with additional properties:

- 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.)

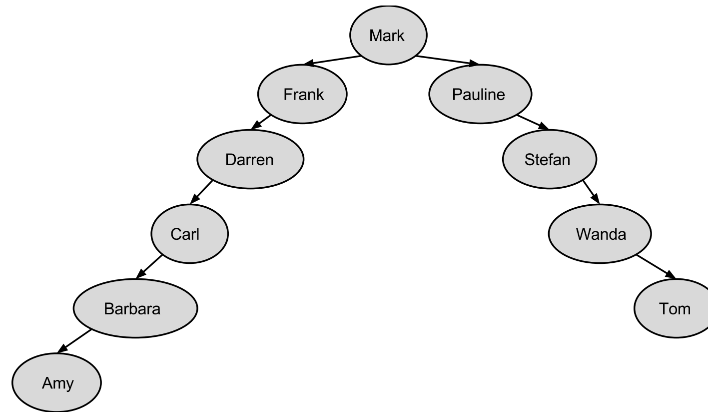


Here are examples of some, pretty well balanced, binary trees. They are also binary search trees.



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).

Joanna Klukowska
joannakl@cs.nyu.edu



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.

4 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 );
    }
}
```

5 Computing the Size of a Tree

Trees are inherently recursive structures: if we pick any node in a tree and disconnect it from its parent, we are still left with a tree (a smaller one, but a tree). This makes recursive algorithms very intuitive.

Joanna Klukowska
joannakl@cs.nyu.edu

5.1 Recursive approach

How are we going to compute the number of nodes in the tree? Simple! If the tree has no nodes, then its size is zero. If the tree has at least one node it is the root node. We can "ask" its children for the sizes of trees rooted at them, and then add the two numbers together and add one for the root. This gives us total number of nodes in the tree. Here is the pseudocode for recursive algorithm for the size method of a BST class.

```
int recSize ( BSTNode<T> root )
    if root == null
        return 0
    else
        return recSize(root.left) + recSize(root.right) + 1;
```

Such method would be a private method in the class and would be called from a public wrapper method. Why? Because we should not allow the client of the class to have access to the root of the tree (what if they set it to null, for example?), so we need to have a way of calling the recursive method ourselves.

```
int size ()
    return recSize( rootOfTheTree)
```

5.2 Iterative approach

The same algorithm implemented recursively is much more complicated. When we looked at iterative and recursive implementations of the size method for the linked list they were not significantly different in complexity and, at least for some of us, the iterative method was a bit more intuitive. This is not true when implementing the size method for trees, because at each node we have multiple branches, so we need to keep track of unexplored branches as we explore the others.

Here is the pseudocode for the iterative implementation of a size method.

```
int size ( )
    set counter to 0
    if tree is not empty (root is not null)
        create an empty stack
        push the root of this tree onto the stack
        while stack is not empty
            set current reference to top of the stack
              remove the top from the stack
            increment counter
            if current has a left child
              push left child onto the stack
            if current has a right child
              push right child onto the stack
    return counter
```

The stack allows us to put the unexplored nodes on hold while we explore other nodes. An empty stack indicates that all the nodes in the tree have been counted (or that we have a bug in our code, but lets be optimistic).

Can you think of a way of writing the above algorithm using a queue?

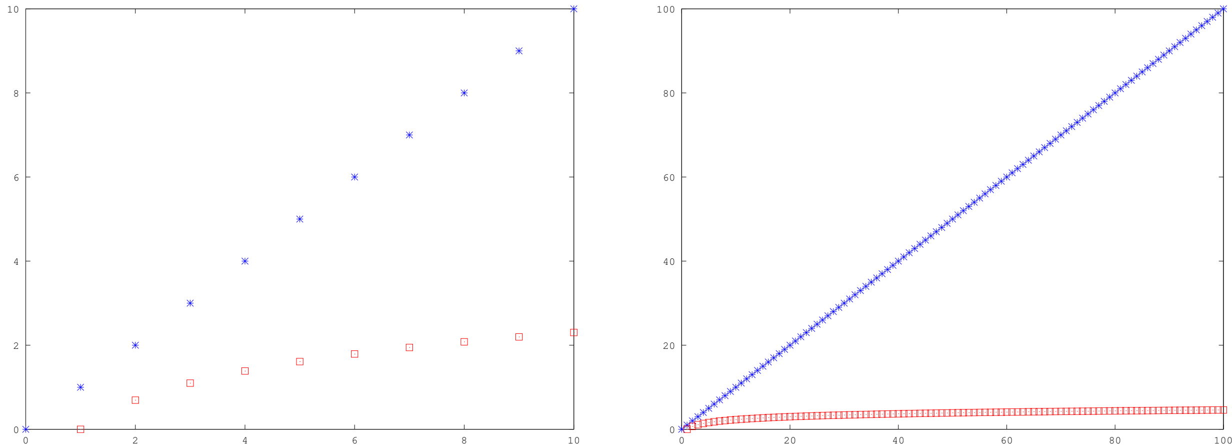
6 Searching for an Item in a Binary Search Tree

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.

Joanna Klukowska
joannakl@cs.nyu.edu

6.1 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.

6.1.1 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)
```

6.1.2 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).

6.2 `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, BSTNode 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.

6.3 `get()` Method for a Binary Search Tree

The only difference between `contains()` and `get()` methods is that the `get()` method actually returnes a reference. Having such a reference the calling program has direct access to the data stored in a tree, without having access to the nodes and tree structure itself.

Here is the pseudocode for recursive algorithm (it is almost identical to the one in the previous section).

```
typeOfItem get ( item, BSTNode currentNode )
    if currentNode == null
       return null
    else if (  item < currentNode.item )
       return get( item, currentNode.left )
    else if (item > currentNode.item )
       return get (item, currentNode.right )
    else
       return current.item
```

Again, the iterative approach is very similar and you should make sure you can write both pseudocode and actual Java code for it (the Java code should be done using generics for best portability).

Joanna Klukowska
joannakl@cs.nyu.edu

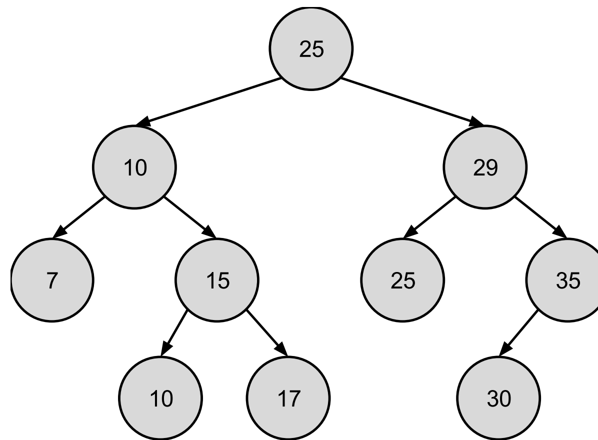
7 Traversing a Binary Tree

We often need to perform an operation on every node in a tree and sometimes the order matters. There are multiple approaches to tree traversals and we will discuss three of them: inorder, preorder and postorder.

7.1 InOrder Traversal

The first one is **inorder traversal**. As the name suggests it visits the nodes according to their ordering (the same ordering that determines their placement into a BST).

Example: Given the following tree, the inorder traversal should visit (and process) the nodes in the increasing order of integers.



Inorder traveral: 7, 10, 10, 15, 17, 25, 25, 29, 30, 35.

7.1.1 Recursive Approach

In a perfectly balanced binary search tree one can think of the root as (approximately) middle element: the nodes with values smaller than the root are in its left subtree - so we need to process them before the root, and the nodes with values larger than the root are in its right subtree - so we need to process them after the root. The same is true about every other node in the tree, not only the root. And once again, this yields a very intuitive recursive algorithm for inorder tree traversal:

- process the nodes in the left subtree of a node
- process the node itself
- process the nodes in the right subtree of a node

In the above, the word "process" is intentially vague. The actual action depends on what exactly we need to do. If we want to print the content of the BST in alphabetical order, then "process" means "print". If we need to perform some other computation based on the data stored in the node, then "process" means "perform the computation".

Here is the pseudocode for a recursive inorder traversal algorithm.

```
recInorderTraversal ( BSTNode node )

    if node is not null
       recInorderTraversal( node.left )
       process the node
       recInorderTraversal( node.right )
```

Joanna Klukowska
joannakl@cs.nyu.edu

7.1.2 Iterative Approach

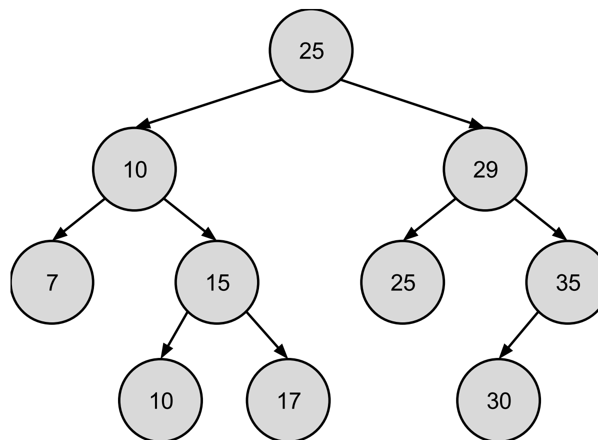
The iterative approach is significantly different and requires an extra data structures to keep track of all the nodes ”on hold”: as we descend down the tree to find the smallest node to be printed first, we need to keep track of all the nodes on the way so we know where to return. As with the iterative `size()` method implementation we going to make use of a stack.

Here is the pseudocode for an iterative inorder traversal algorithm.

```
inorderTraversal ( )

    if tree is not empty (root is not null)
        create an empty stack
        set current to root of this tree
        set done to false
        while not done
            if current is not null
              push current onto the stack
              current = current.left
            else if stack is not empty
              current = top of the stack
              remove the item from top of the stack
              process current
              current = current.right
            else
              set done to true
```

This is not as inuitive as the recursive version, but it might be easier to follow with an example. We are going to apply this algorithm to the following tree:



State of the inOrder traversal program after each change to current when run with the above tree.

`iter` indicates iteratiion number of the while loop

`current` indicates the value of the node pointed to by the current pointer

`stack` indicates the content of the stack used for keeping track of nodes that we need to return to

`processed` indicates which nodes have been already processed

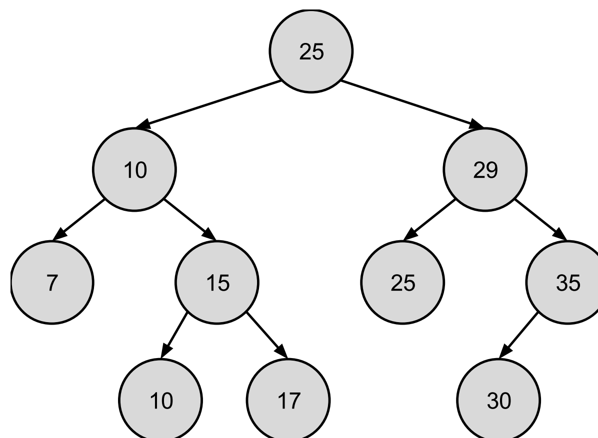
Joanna Klukowska
joannakl@cs.nyu.edu

```
iter:  0 current:   25 stack: []           processed: []
iter:  1 current:   10 stack: [25]         processed: []
iter:  2 current:    7 stack: [25, 10]     processed: []
iter:  3 current: null stack: [25, 10, 7]  processed: []
iter:  4 current:    7 stack: [25, 10]     processed: []
iter:  4 current: null stack: [25, 10]     processed: [7]
iter:  5 current:   10 stack: [25]         processed: [7]
iter:  5 current:   15 stack: [25]         processed: [7, 10]
iter:  6 current:   10 stack: [25, 15]     processed: [7, 10]
iter:  7 current: null stack: [25, 15, 10] processed: [7, 10]
iter:  8 current:   10 stack: [25, 15]     processed: [7, 10]
iter:  8 current: null stack: [25, 15]     processed: [7, 10, 10]
iter:  9 current:   15 stack: [25]         processed: [7, 10, 10]
iter:  9 current:   17 stack: [25]         processed: [7, 10, 10, 15]
iter: 10 current: null stack: [25, 17]     processed: [7, 10, 10, 15]
iter: 11 current:   17 stack: [25]         processed: [7, 10, 10, 15]
iter: 11 current: null stack: [25]         processed: [7, 10, 10, 15, 17]
iter: 12 current:   25 stack: []           processed: [7, 10, 10, 15, 17]
iter: 12 current:   29 stack: []           processed: [7, 10, 10, 15, 17, 25]
iter: 13 current:   25 stack: [29]         processed: [7, 10, 10, 15, 17, 25]
iter: 14 current: null stack: [29, 25]     processed: [7, 10, 10, 15, 17, 25]
iter: 15 current:   25 stack: [29]         processed: [7, 10, 10, 15, 17, 25]
iter: 15 current: null stack: [29]         processed: [7, 10, 10, 15, 17, 25, 25]
iter: 16 current:   29 stack: []           processed: [7, 10, 10, 15, 17, 25, 25]
iter: 16 current:   35 stack: []           processed: [7, 10, 10, 15, 17, 25, 25, 29]
iter: 17 current:   30 stack: [35]         processed: [7, 10, 10, 15, 17, 25, 25, 29]
iter: 18 current: null stack: [35, 30]     processed: [7, 10, 10, 15, 17, 25, 25, 29]
iter: 19 current:   30 stack: [35]         processed: [7, 10, 10, 15, 17, 25, 25, 29]
iter: 19 current: null stack: [35]         processed: [7, 10, 10, 15, 17, 25, 25, 29, 30]
iter: 20 current:   35 stack: []           processed: [7, 10, 10, 15, 17, 25, 25, 29, 30]
iter: 20 current: null stack: []           processed: [7, 10, 10, 15, 17, 25, 25, 29, 30, 35]
```

7.2 PreOrder Traversal

In the **preorder traversal** of the tree, we visit the node before exploring its children.

Example: Given the following tree, the preorder traversal visits the root, then visits its left subtree and then visits its right subtree.



Preorder traveral: 25, 10, 7, 15, 10, 17, 29, 25, 35, 30.

7.2.1 Recursive Approach

The preorder traversal algorithm is intuitively described by the sequence of three steps mentioned already above:

Joanna Klukowska
joannakl@cs.nyu.edu

- process the node itself
- process the nodes in the left subtree of the node
- process the nodes in the right subtree of the node

The pseudocode for a recursive preorder traversal algorithm does not differ much:

```
recInorderTraversal ( BSTNode node )
    if node is not null
       process the node
       recInorderTraversal( node.left )
       recInorderTraversal( node.right )
```

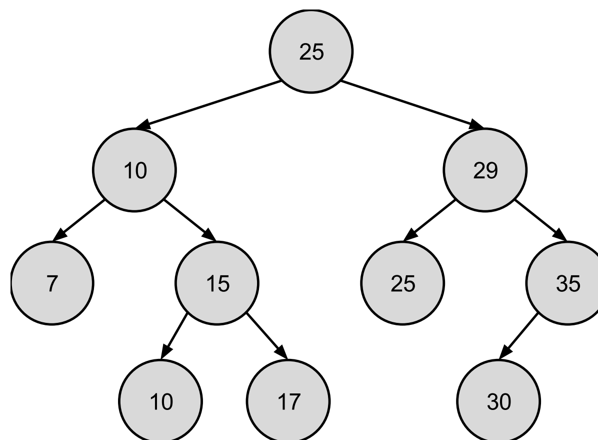
7.2.2 Iterative approach

Exercise: try to derive the pseudocode for the preorder traversal on your own to see if you understand the concepts.

7.3 PostOrder Traversal

Finally, in the **postorder traversal** of the tree, we visit the node after exploring its children.

Example: Given the following tree, the postorder traversal visits its left subtree, then visits its right subtree and then visits the root.



Postorder traveral: 7, 10, 17, 15, 10, 25, 30, 35, 29, 25.

7.3.1 Recursive Approach

The postorder traversal algorithm is intuitively described by the sequence of three steps mentioned already above:

- process the nodes in the left subtree of a node
- process the nodes in the right subtree of a node
- process the node itself

The pseudocode for a recursive preorder traversal algorithm does not differ much:

```
recInorderTraversal ( BSTNode node )
    if node is not null
       recInorderTraversal( node.left )
       recInorderTraversal( node.right )
       process the node
```

Joanna Klukowska
joannakl@cs.nyu.edu

8 Inserting a Node into a Binary Search Tree

Adding new nodes to the tree always happens at the leaves. To insert a new node, we need to navigate through the tree to finds 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 right 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
        recAdd (node.right, data )
```

But there is a problem here: we do not have a basecase, 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 int 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 );
```

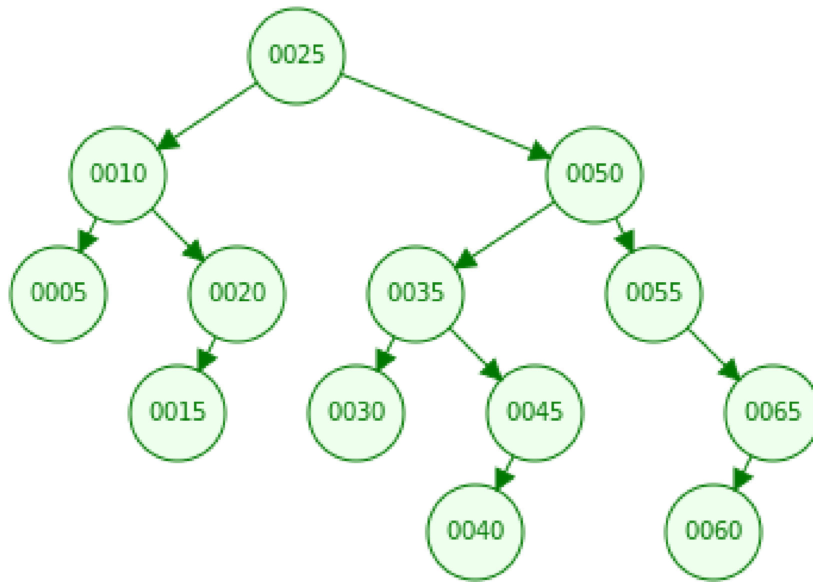
8.1 Order of Insertions

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

9 Removing a Node from a Binary Search Tree

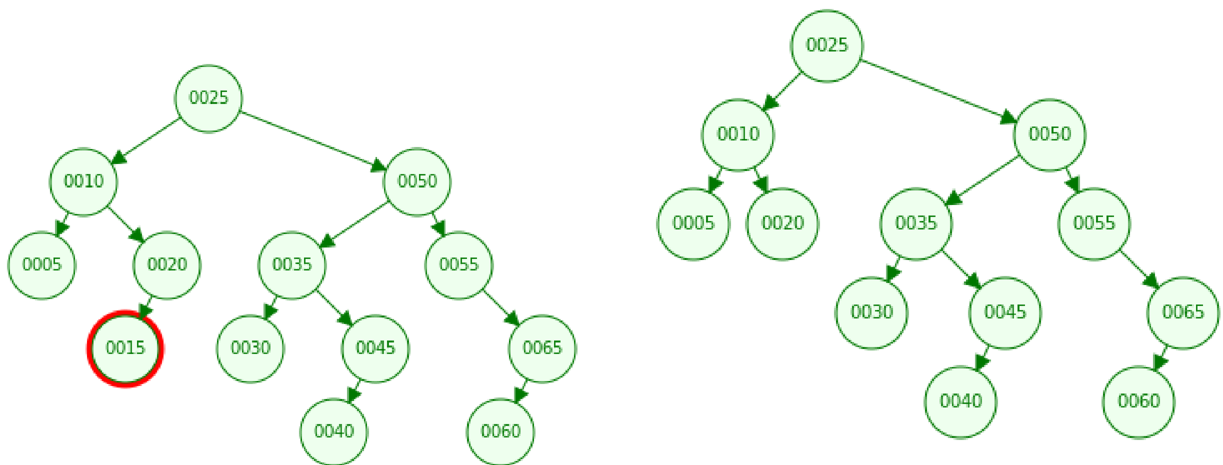
The method for removing a node from a binary search tree is the most involved from all the BST methods that we looked at so far. We will consider it in a case by case basis starting from the simplest one. We will work with the following tree in this section.

Joanna Klukowska
joannakl@cs.nyu.edu



9.1 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).



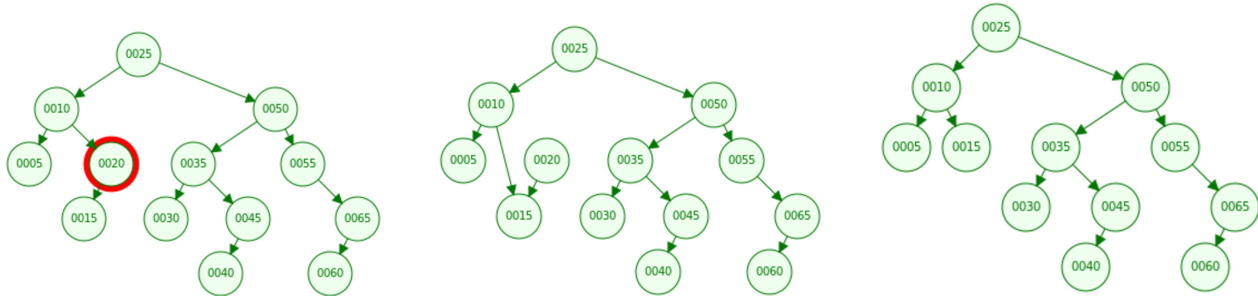
The tricky part is to know who the parent of node lebeled 15 is once we find it. In this, and all the other cases, as we are searching for the node, we will keep a reference to its parent.

9.2 Removing a Node with One Child

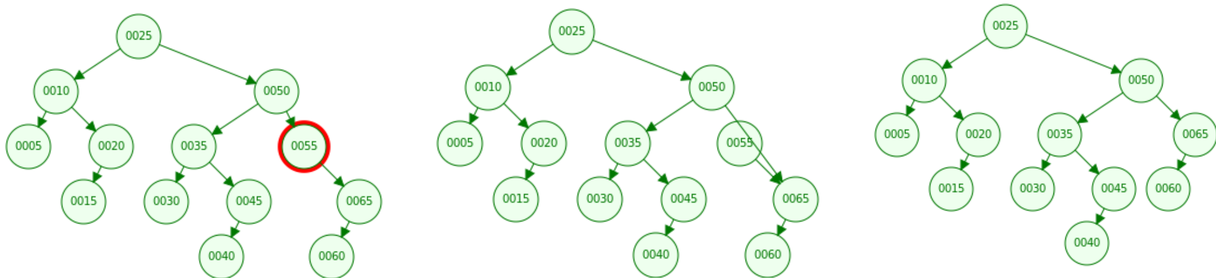
The nodes with one child have either left or right reference pointing to a subtree, but the other one is pointing to null. When we delete a node like this, its parent adopts its child subtree (the entire branch is just pulled up). If we remove nodel labeled 20 from out tree, we first find it (figure 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

Joanna Klukowska
joannakl@cs.nyu.edu

its onw 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.



9.3 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 mantains 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. Fortunatelly, 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
```

Joanna Klukowska
joannakl@cs.nyu.edu

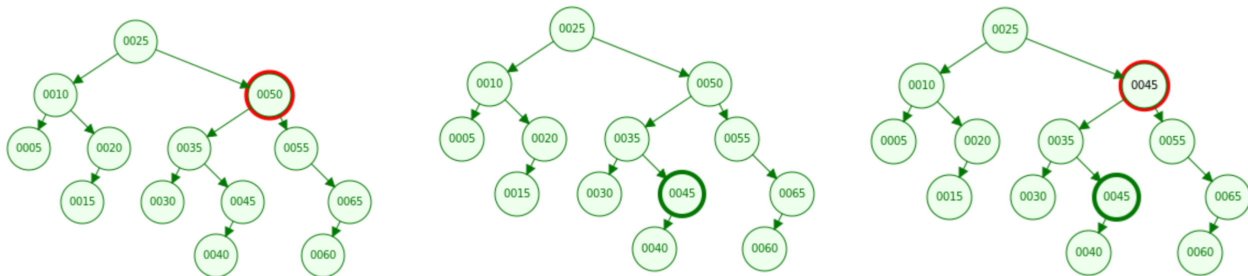
```
BSTNode <T> current = n.left
while (current.right != null )
   current = current.right
return current.data
```

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!!!

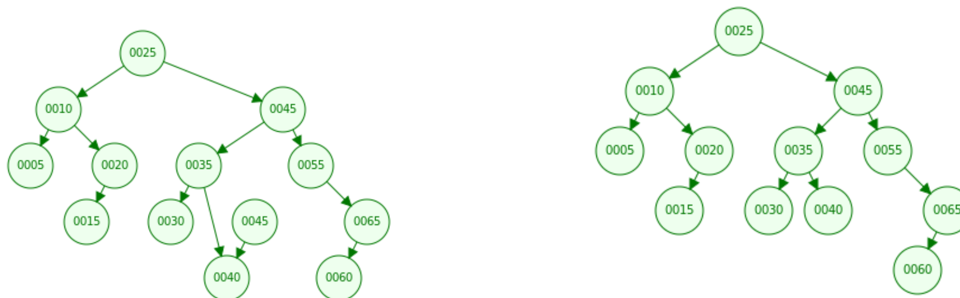
Removing the rightmost node of the left subtree is guaranteed to fall under one of the simpler cases: either it is a leaf node, or it has only a left child. It is guaranteed not to have a right child, otherwise it would not have been a rightmost child of the subtree.

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

1. find the node with label 50
2. find its predocessor in the tree (the rightmost node it its left subtree), that's 45
3. use 45 to replace 50



and then remove 45 using the leaf or one child approach



9.4 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 )
```

Joanna Klukowska
joannakl@cs.nyu.edu

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 subtre
    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
```