



## Tutorial #12

(March 12<sup>th</sup>, 2013)

### Objectives:

- Search Operation in the BST and its implementation.
- Deletion Operation in the BST and its implementation.

### Search Operation in the BST and its implementation:

Searching for a value in a BST is very similar to insert operation. Search algorithm traverses the tree "in-depth", choosing appropriate way to go, following binary search tree property and compares value of each visited node with the one we are looking for. Algorithm stops in two cases:

- A node with necessary value is found.
- Algorithm has no way to go.

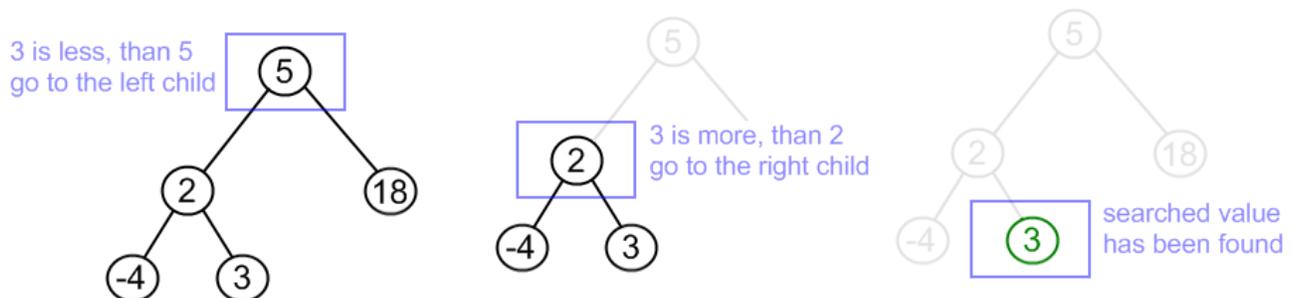
### Search algorithm in detail:

Now, let's see more detailed description of the search algorithm. Like an insert operation, and almost every operation on BST, search algorithm utilizes recursion. Starting from the root,

1. Check, whether value in current node and searched value are equal. If so, **value is found**. Otherwise,
2. If searched value is less than the node's value:
  - If current node has no left child, **searched value doesn't exist in the BST**;
  - Otherwise, handle the left child with the same algorithm..
3. If a new value is greater, than the node's value:
  - If current node has no right child, **searched value doesn't exist in the BST**;
  - Otherwise, handle the right child with the same algorithm.

### Example

Search for 3 in the tree shown below.



## Code snippets:

As in insert operation, check first if root exists. If not, tree is empty, and, therefore, searched value doesn't exist in the tree. This check can be done in the *BinarySearchTree* class. Principal algorithm is implemented in the *BSTNode* class.

Code
------

## Deletion Operation in the BST and its implementation:

Remove operation on binary search tree is more complicated, than insert and search. Basically, in can be divided into two stages:

- Search for a node to remove.
- If the node is found, run remove algorithm.

## Remove algorithm in detail:

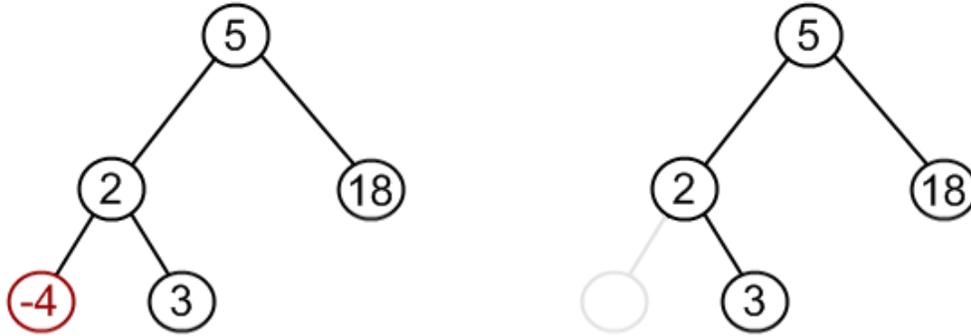
Now, let's see more detailed description of a remove algorithm. First stage is identical to the insert and the search algorithms, except we should track the parent of the current node. Second part is tricky. There are three cases, which are described below.

**1. Node to be removed has no children:**

This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

**Example:**

Remove -4 from a BST.

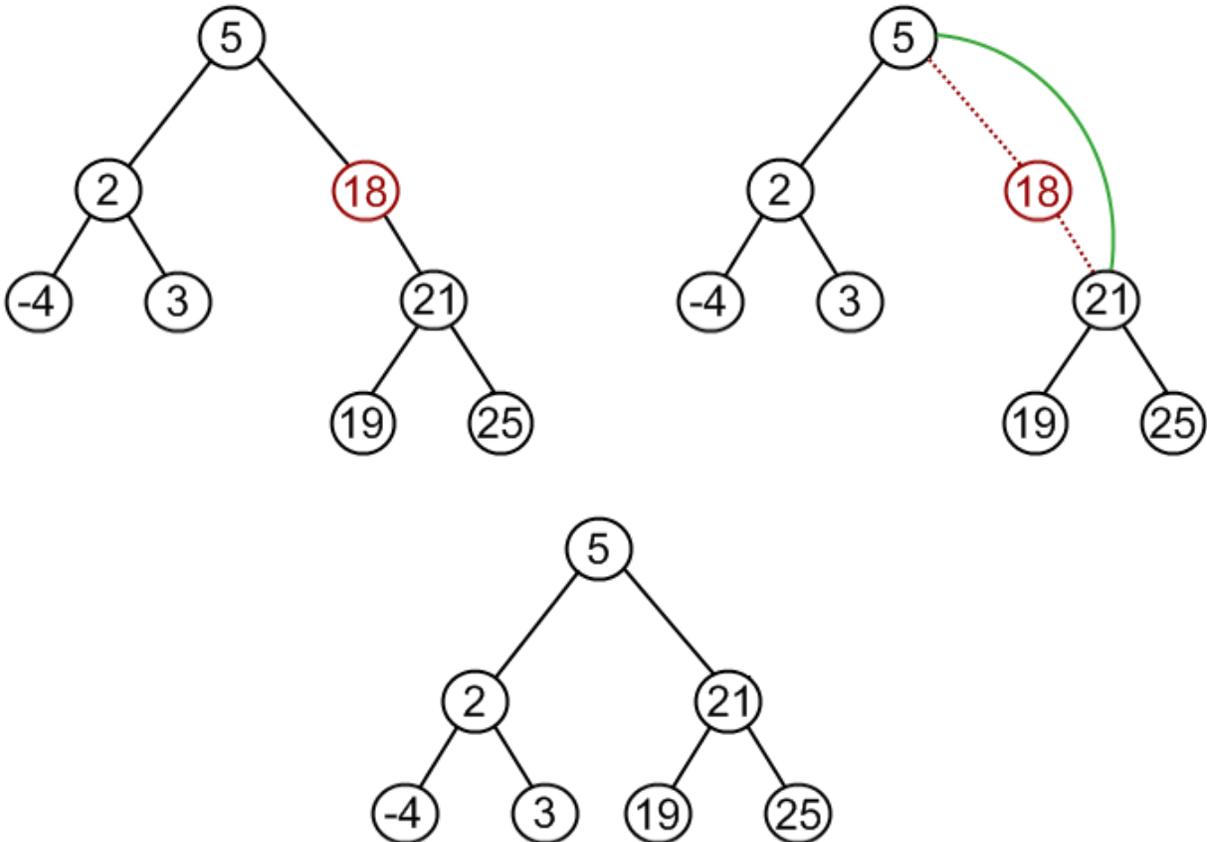


**2. Node to be removed has one child:**

In this case, node is cut from the tree and algorithm links single child (with its sub-tree) directly to the parent of the removed node.

**Example:**

Remove 18 from a BST.



### 3. Node to be removed has two children:

This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example BSTs shown in Figure (1) contain the same values {5, 19, 21, 25}. To transform first tree into second one, we can do following:

- Choose minimum element from the right sub-tree (19 in the example);
- Replace 5 by 19;
- Hang 5 as a left child.

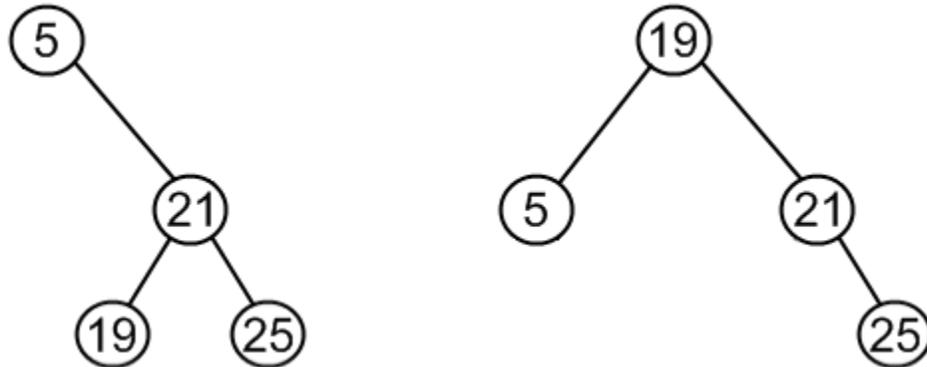


Figure (1)

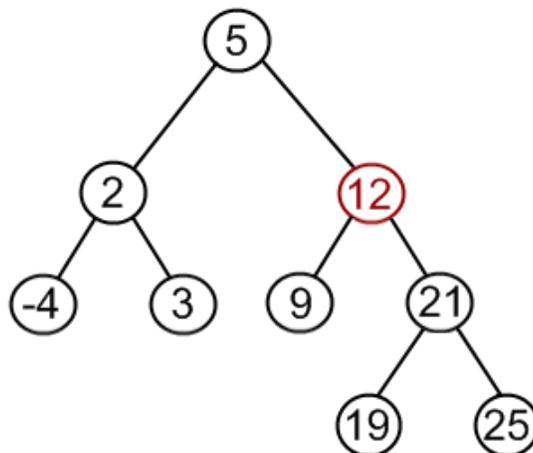
The same approach can be utilized to remove a node, which has two children:

- Find a minimum value in the right sub-tree;
- Replace value of the node to be removed with found minimum. Now, right sub-tree contains a duplicate!
- Apply remove to the right sub-tree to remove a duplicate.

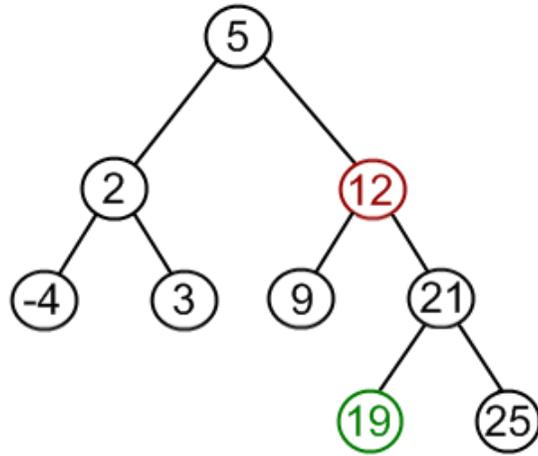
Notice that, the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

#### Example:

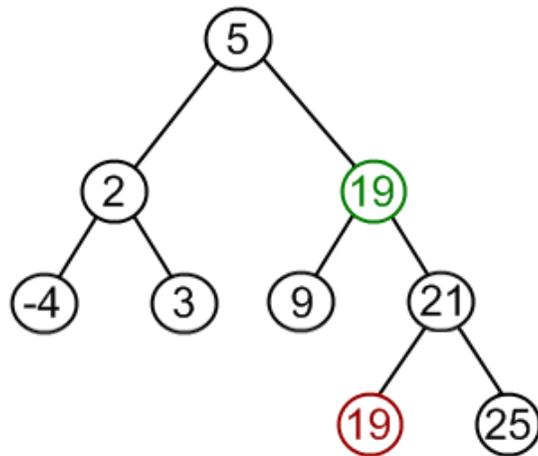
Remove 12 from a BST.



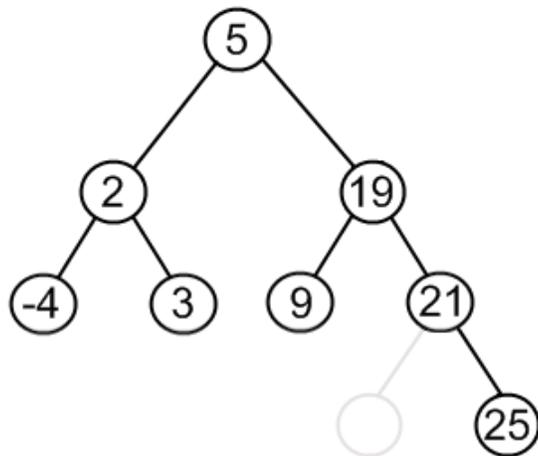
Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.



Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.



Remove 19 from the left sub-tree.



## Code snippets:

First, check first if root exists. If not, tree is empty, and, therefore, value, that should be removed, doesn't exist in the tree. Then, check if root value is the one to be removed. It's a special case and there are several approaches to solve it. We propose the *dummy root* method, when dummy root node is created and real root hanged to it as a left child. When remove is done, set root link to the link to the left child of the dummy root.

Code
<pre>public class BinarySearchTree {     ...     public boolean remove(int value) {         if (root == null)             return false;         else {             if (root.getValue() == value) {                 BSTNode auxRoot = new BSTNode(0);                 auxRoot.setLeftChild(root);                 boolean result = root.remove(value, auxRoot);                 root = auxRoot.getLeft();                 return result;             } else {                 return root.remove(value, null);             }         }     } }  public class BSTNode {     ...     public boolean remove(int value, BSTNode parent) {         if (value &lt; this.value) {             if (left != null)                 return left.remove(value, this);             else                 return false;         } else if (value &gt; this.value) {             if (right != null)                 return right.remove(value, this);             else                 return false;         } else {             if (left != null &amp;&amp; right != null) {                 this.value = right.minValue();                 right.remove(this.value, this);             } else if (parent.left == this) {                 parent.left = (left != null) ? left : right;             } else if (parent.right == this) {                 parent.right = (left != null) ? left : right;             }             return true;         }     } }</pre>

```
public int minValue() {  
    if (left == null)  
        return value;  
    else  
        return left.minValue();  
    }  
}
```