# Exceptional Code

Coding away life...

Wednesday, August 17, 2011

## A Binary Search Trees Tutorial

This is a short tutorial on Binary Search Trees covering the basic operations that are often done on such trees.

A Binary Search Tree (BST) is a tree data structure that has the following property:

**For any node x in the tree, the left subtree rooted at x only contains nodes with values less than or equal to the value stored at x; and the right subtree rooted at x only contains nodes with values greater than the value at x. Also, the left subtree and the right subtree of x must be binary trees themselves**.



Figure: A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

Image source: Wikipedia

BSTs have sub-linear (logarithmic) average case complexity for element insertion and searching. Once a BST is built, sorting it can be done in linear time. All we need is traverse the tree in inorder.

There are a number of operations that can be done on a BST. We will describe those here in some details with Java code examples.

First let's see how we can represent a tree node in Java:

```
class TreeNode
{
  int data;
  TreeNode parent;
  TreeNode left;
  TreeNode right;

  public TreeNode(int data)
  {
    this.data = data;
    parent = left = right = null;
  }

  public void setData(int data)
  {
    this.data = data;
  }
```

**Popular Posts**


Coding up a Trie (Prefix Tree)


Solving the Boggle Game - Recursion, Prefix Tree, and Dynamic Programming

External Sorting for sorting large files in disk

A Binary Search Trees Tutorial

Generating all permutations, combinations, and power set of a string (or set of numbers)

**Followers**

Join this site
with Google Friend Connect

Members (13)

Already a member? Sign in

**Blog Archive**

► 2013 (2)
► 2012 (4)
▼ 2011 (4)
▼ August (1)
A Binary Search Trees Tutorial
► July (3)

**About Me**

**Golam Kawsar**
NY, NY, United States

I am a software developer living in the most vibrant city in the world - New York. I dream of a day when software will be more intelligent than humans, but humans will still control it :) I am originally from Bangladesh, a small and beautiful country in South Asia. In Bangladesh, we are often known and called by our nick names. My nick name is Bilash.

View my complete profile

```
  public void setLeft(TreeNode node)
  {
    left = node;
  }

  public void setRight(TreeNode node)
  {
    right = node;
  }

  public void setParent(TreeNode node)
  {
    parent = node;
  }

  public int getData()
  {
    return data;
  }

  public TreeNode getLeft()
  {
    return left;
  }

  public TreeNode getRight()
  {
    return right;
  }

  public TreeNode getParent()
  {
    return parent;
  }
}
```

**Inserting a node into a BST**:

Inserting a node into a BST is pretty straightforward. We start checking nodes starting from the root node. At each node if the value is greater than the value in the node to be inserted then we move to the left child, otherwise we move to the right child. We repeat this until there is no more node to be traversed. The last node will be the parent node of the node to be inserted. Now if the node to be inserted has a smaller value than the current node then we set it as the left child of the current node otherwise we set it as the right child.

Java code:

```
public static void insert(TreeNode node)
{
  TreeNode x, y;
      // root is a global variable and is the root of the tree
  x = y = root; // Assume root is initialized to null

  while (x != null)
  {
    if (x.getData() > node.getData())
    {
      y = x;
      x = x.getLeft();
    }
    else
    {
      y = x;
      x = x.getRight();
    }
  }

  // y will be the parent of node
  if (y == null)
  {
    root = node;
    return;
  }
```

```
  if (y.getData() > node.getData())
    y.setLeft(node);
  else
    y.setRight(node);

  node.setParent(y);
}
```

**Deleting a node from a BST**:

When deleting a node from a BST, there can be 3 different scenarios:
i) The node does not have any children: This is kind of a trivial case. When there is no children all we need is removing the node from its parent.
ii) The node has only one child: This is also a relatively simple case. Since the node has only one child, we will simply replace the node with this lone child without violating the BST sorted order property.
iii) The node has two children: Now, this is a bit tricky :) Since there are two children we have to be careful not to break the sorted order of the BST. To keep the sorted order intact, we need to replace the node to be deleted with its successor so that the order is maintained in the absence of the node. Once a successor is found, we need to replace the node to be deleted with it. It turns out the successor can have at most one child. So if the successor has a child we will have to replace the successor with that child after the successor already replaced the node to be deleted.

Java code:

```
public static void delete(TreeNode node)
{
  // Case 1: node does not have a child, just delete it
  if (node.getLeft() == null && node.getRight() == null)
  {
    if (node.getParent() != null && node.getParent().getLeft() == node)
      node.setLeft(null);
    else if (node.getParent() != null && node.getParent().getRight() == node)
      node.setRight(null);
  }
  // Case 2: node has only one child, splice the child with its parent
  else if (node.getLeft() == null || node.getRight() == null)
  {
    if (node.getParent() != null)
    {
      TreeNode x = node.getLeft() == null ? node.getRight() : node.getLeft();
      if (node.getParent().getLeft() == node)
        node.getParent().setLeft(x);
      else
        node.getParent().setRight(x);
    }
  }
  // Case 3: node has both children, set the successor of the node to its parent
  else
  {
    TreeNode x = findSuccessor(node); // x will have at most one child
    // Instead of deleting we can just copy the successor's data over to the node to be
deleted
    node.setData(x.getData());
    // Now delete the successor and set its child (if any) to its parent
    TreeNode nodeChild = x.getLeft() == null ? x.getRight() : x.getLeft();
    if (x.getLeft() != null)
    {
      if (x.getParent().getLeft() == x)
        x.getParent().setLeft(nodeChild);
      else
        x.getParent().setRight(nodeChild);
    }
    else
    {
      if (x.getParent().getLeft() == x)
        x.getParent().setLeft(nodeChild);
      else
        x.getParent().setRight(nodeChild);
    }
  }
}
```

**Finding the minimum value in the tree**:

In a BST, all nodes to the left of a node contains smaller (or equal) values than the value stored in the node itself. This gives us a clue about how to tackle the problem of finding the minimum value stored in the BST. If you think about it, the node with the minimum value in a BST is actually the leftmost node in the tree. If it wasn't then there would be node(s) in the tree that are on the right of some some node but contain smaller values than the node itself, thus violating the BST contract.

Here is the Java code for finding the minimum value node in a BST:

```
public TreeNode findMinimum(TreeNode root)
{
  if (root == null)
    return null;

  if (root.getLeft() != null)
    return findMinimum(root.getLeft());

  return root;
}
```

**Finding the maximum value in the tree**:

Similar to the logic as in finding the minimum value, the maximum value node will be the right most node in the tree. The Java code for finding the maximum value node in a BST:

```
public static TreeNode findMaximum(TreeNode root)
{
  if (root == null)
    return null;

  if (root.getRight() != null)
    return findMaximum(root.getRight());

  return root;
}
```

One useful property of the BST is that if it is traversed in inorder we get a sorted list of values stored in the tree nodes. There are two operations that are often done in relation to maintaining this sorted order of a BST: finding the successor node and predecessor node of a given node.

**Finding the successor node of a given node**:

The successor node is the node with the next bigger number in the tree. Since all numbers to the left are smaller than the current node the successor node has to be located in the right subtree. Now, since it is the next bigger number of the current node, it has to be the smallest of all numbers in the right subtree. So, essentially we are looking for the minimum number in the right subtree of the current node. In the case that there is no right child of the node, we need to look up the tree and figure out the successor. The successor is the first ancestor whose left subtree has this node as the largest number. In other words: the first ancestor of this node whose left child is also an ancestor of this node. The intuition is: as we traverse left up the tree we traverse smaller values, the first node on the right is the next larger number.

Here is the Java code:

```
public static TreeNode findSuccessor(TreeNode node)
{
  if (node == null)
    return null;

  if (node.getRight() != null)
    return findMinimum(node.getRight());

  TreeNode y = node.getParent();
  TreeNode x = node;
  while (y != null && x == y.getRight())
  {
    x = y;
    y = y.getParent();
  }
```

```
  return y;
}
```

**Finding the predecessor node of a given node**:

The predecessor of a given node is the node containing the next smaller value. Finding the predecessor follows symmetric rules that we used for finding the successor.

Java code for finding the predecessor:

```
public static TreeNode findPredecessor(TreeNode node)
{
  if (node == null)
    return null;

  if (node.getLeft() != null)
    return findMaximum(node.getLeft());

  TreeNode y = node.getParent();
  TreeNode x = node;
  while (y != null && x == y.getLeft())
  {
    x = y;
    y = y.getParent();
  }

  return y;
}
```

**Determining whether a tree is a BST or not**:

Sometimes we already have a binary tree that we need to determine whether it is a BST or not. This is an interesting problem and can be really solved with a simple recursive solution.

The BST property - that *every* node on the right subtree has to be larger than the current node and *every* node on the left subtree has to be smaller (or equal) than the current node - is the key to figuring out whether a tree is a BST or not. On a first thought it might look like we can simply traverse the tree and at every node check whether the node contains a value larger than the value at the left child and smaller than the value on the right child, and if this condition holds for all the nodes in the tree then we have a BST. This is the so called Greedy approach, making a decision based on local properties. But this approach clearly won't work for the following tree:

```
    20
   /  \
 10    30
      /  \
     5    40
```

In the tree above, at every node the condition that the node contains a value larger than its left child and smaller than its right child hold, still its not a BST: the value 5 is on the right subtree of the node containing 20, a violation of the BST property!

So how do we solve this? It turns out that instead of making a decision based solely on a node and its children's values, we also need information flowing down from the parent as well. In the case of the tree above, if we could remember about the node containing the value 20 we could see that the node with value 5 is violating the BST property contract.

So the condition we need to check at each node is that: a) if the node is the left child of its parent, then it must be smaller (or equal) than the parent and it must pass down the value from its parent to its right subtree to make sure none of the nodes in that subtree is greater the parent, and similarly b) if the node is the right child of its parent, then it must be larger than the parent and it must pass down the value from its parent to its left subtree to make sure none of the nodes in that subtree is greater the parent.

A simple but elegant recursive solution in Java can explain this further:

```
public static boolean isBST(TreeNode node, int leftData, int rightData)
{
  if (node == null)
    return true;

  if (node.getData() > leftData || node.getData() <= rightData)
    return false;
```

```
  return (isBST(node.left, node.getData(), rightData) && isBST(node.right, leftData,
node.getData()));
}
```

The initial call to this function can be something like this:

```
if (isBST(root, Integer.MAX_VALUE, Integer.MIN_VALUE))
  System.out.println("This is a BST.");
else
  System.out.println("This is NOT a BST!");
```

Essentially we keep creating a valid range (starting from [ MIN_VALUE, MAX_VALUE]) and keep shrinking it down foe each node as we go down recursively.

So, here is my short and sweet primer on Binary Search Trees. Hope you found it useful. Let me know if there is a bug or a possible improvement in runtime or space.

Posted by Golam Kawsar at 8:15 PM

+3 Recommend this on Google

Labels: Binary Search Trees, BST, Tree manipulations, Trees

## 3 comments:

**Peky_69** August 18, 2011 at 12:50 AM

Reply

**Unknown** August 18, 2011 at 1:10 AM

Great article, I'd add extension to it, first, about creating BST out of any binary tree and about balancing BSTs, than it'll be the extra exceptional code (Unhandled exception: Info about this comment could not be read :))

Reply

**Golam Kawsar** August 18, 2011 at 5:25 PM

OK, will try to add those two (creating BST out of any binary tree and about balancing BSTs) sometime in the future!

Thanks for taking time to read it:)

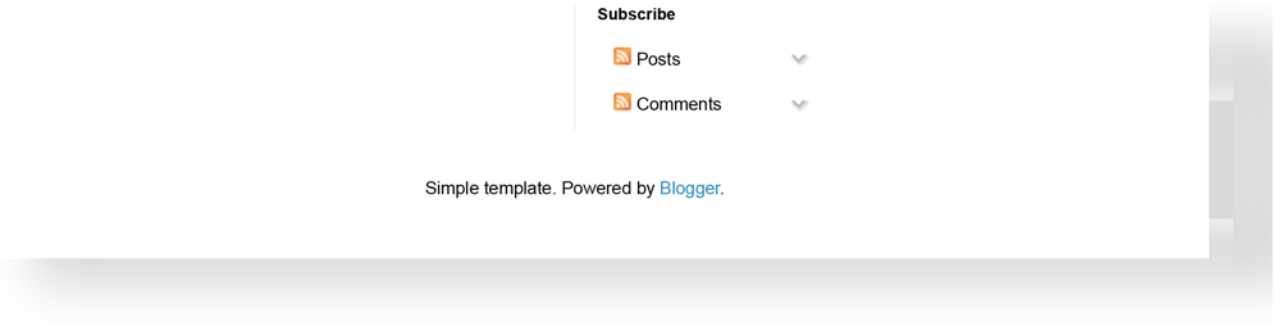
Reply

**Comment as:** Select profile...

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)


Subscribe
Posts
Comments
Simple template. Powered by Blogger.