Lab 10, Binary Search Tree

Contents

Binary Search Tree

On the algs4 website, we are given a recursive implementation of binary search tree.

https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/BST.java.html

In this lab we will be writing a non-recursive one.

Binary Search Tree

At the very begining, we should define the basic structure. This is important since we want to "maintain" a structure instead of just using the algorithm. This is a basic tree node definition.

```
public class TreeNode {

  public Key key;
  public Value value;

  public TreeNode left = null;
  public TreeNode right = null;
  public TreeNode parent = null;


  public TreeNode( Key key, Value value ) {
      this.key = key;
      this.value = value;
  }
}
```

Binary Search Tree

Note that we used generic here in order to store various types in the node.

```
public class TreeNode <Key extends Comparable<Key>, Value> {

  public Key key;
  public Value value;

  public TreeNode left = null;
  public TreeNode right = null;
  public TreeNode parent = null;


  public TreeNode( Key key, Value value ) {
    this.key = key;
    this.value = value;
  }
}
```

Binary Search Tree

With our tree node defined, we can easily define the binary tree data structure.

```
public class BSTree<Key extends Comparable<Key>, Value> {

  private TreeNode root = null;
  private int size = 0;

}
```

In a basic bst implementation, we use a root node to manage the whole tree. We also use a size variable to record the number of nodes in the tree.

Binary Search Tree

First we should implement the get and put methods to provide basic functionality.

```
public Value get( Key key ) {
  if( key == null )
    return null;
  TreeNode node = root;
  while( node != null ) {
    int compare = node.key.compareTo(key);
    if( compare < 0 )
      node = node.right;
    else if( compare > 0 )
      node = node.left;
    else
      return node.value;
  }
  return null;
}
```

Binary Search Tree

Always be aware of the "null". Any parameter of your methods could be null (except those primitive types). The "root" variable could also be null.

Some implementations allows you to store a null value in the tree. You can make your own decision when implementing your own version.

```
public Value put( Key key, Value value ) {
  if( key == null || value == null )
    return null;
  if( root == null ) {
    root = new TreeNode<Key,Value>(key, value);
    return null;
  }
```

Binary Search Tree

Keep working on the "put" method. We go down in the tree until we find a null value and we insert the node here.

```
TreeNode<Key,Value> node = root;
while( node != null ) {
  int compare = node.key.compareTo(key);
  if( compare < 0 ) {
    if( node.right == null ) {
      node.right = new TreeNode<Key,Value>(key, value);
      node.right.parent = node;
      size++;
      return null;
    } else
      node = node.right;
  }
```

Binary Search Tree

If we actually find the node in the tree, we could replace the value in the tree node with the new value.

In reality the implementation should rely on your requirement.

```
else {
  Value ret = node.value;
  node.value = value;
  return ret;
}
```

Binary Search Tree

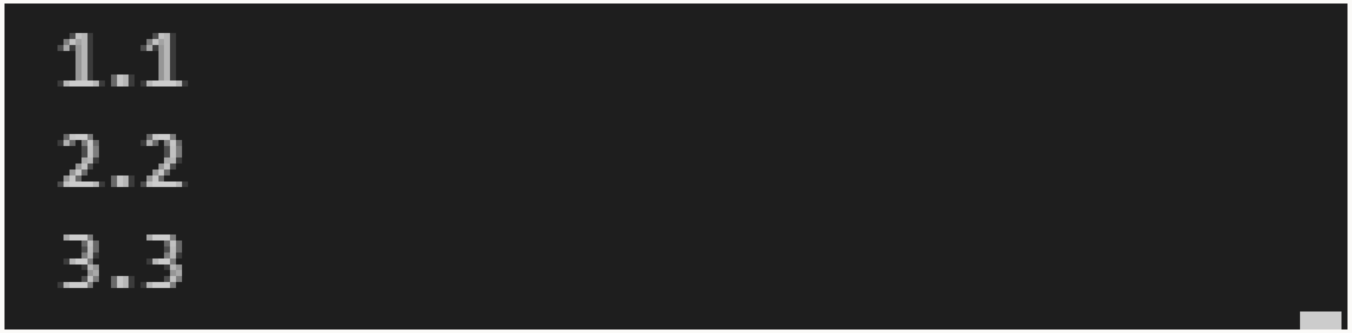
We have finished "get" and "put" method. Let's write a simple test.

```
public static void main( String[] args ) {
  BSTree<Integer, Double> tree = new BSTree<>();
  tree.put(1, 1.1);
  tree.put(2, 2.2);
  tree.put(3, 3.3);

  System.out.println(tree.get(1));
  System.out.println(tree.get(2));
  System.out.println(tree.get(3));
}
```

Binary Search Tree

If the above test shows right result, we can move on the the next step.



```
1.1
2.2
3.3
```

Binary Search Tree

In the next step, we should write the "delete" method. But before that, let's implement the "delMin" method to delete the minimal node in the subtree.

First we should also check the whether the node is null.

```
private TreeNode delMin( TreeNode node ) {
  if( node == null )
      return null;
```

Binary Search Tree

Then we iteratively go to the left child node to get the minimal node here. Then we just remove the node.

```
while( node.left != null )
  node = node.left;
if( node == root )
  root = node.right;
else if( node == node.parent.left )
  node.parent.left = node.right;
else
  node.parent.right = node.right;
if( node.right != null )
  node.right.parent = node.parent;
size --;
return node;
```

Binary Search Tree

With delMin implemented, we can delete a node with a given key.

First we should also check null value.

```
public Value delete( Key key ) {
  if( root == null || key == null )
    return null;
```

Binary Search Tree

If the given key is smaller than the node key, node go to left or right.

```
TreeNode node = root;
while( node != null ) {
  int compare = node.key.compareTo(key);
  if( compare < 0 )
    node = node.right;
  else if( compare > 0 )
    node = node.left;
```

Binary Search Tree

When we find the node, we should remove it. If the node has zero or one child, it is simple.

```
if( node.left == null ) {
  if(node.right != null)
    node.right.parent = node.parent;
  if( node == root )
    root = node.right;
  else if( node == node.parent.left )
    node.parent.left = node.right;
  else
    node.parent.right = node.right;
  size --;
  return node.value;
}
```

Binary Search Tree

If the node contains two child, we should replace it with the minimal node in the right subtree.

```
TreeNode minNode = delMin(node.right);
Value ret = node.value;
node.key = minNode.key;
node.value = minNode.value;
size--;
return ret;
```

In class exercise: allow duplication

Modify the BST implementation such that duplicate keys in trees are allowed:

- The left child is smaller than its parent.
- The right child is larger than **or equal to** its parent.
- The get(Key) method returns a list of all values whose key is equal to the given key.
- The delete(Key) method deletes all nodes whose key equals to the given key.