Lamess Kharfan

Student ID:10150607

CPSC 331 - Assignment #3

- Let T be a binary search tree storing values of type V with keys of type E and let *key* be any value of type E. If the search algorithm is executed with inputs *key* and T, the algorithm eventually terminates and following are satisfied upon termination:
    - If *key* is a key stored in T then v, the value of the key we are searching for in T is returned.
    - If *key* is not stored in T then a KeyNotFoundException is thrown.
    - Neither T or *key* are changed

Thus, the given algorithm is correct for the search algorithm.

Loop Invariant:

- key is a value of type E
- T is a binary search tree with keys of type E and associated values of type V
- curr != null and v = null
- key and T are unchanged
- The Binary Search Tree property is satisfied

Proof:

Suppose h, the height of a given tree T is -1. Then T is an empty tree, so *key* cannot be stored within T. It is sufficient to show that the program halts with a KeyNotFoundException being thrown since the while loop is never entered and V is never re-assigned any value. T and *key* are never changed.

Since T is an empty tree, curr, which is initially set to the root of the tree T is set to null, since an empty tree has no root, and v is also set to null. So the loop test while(curr != null and v = null), fails, so the loop body is skipped over, and we go to the if statement, if(v = null), is true, so a KeyNotFoundException is thrown.

Now, suppose the height, h, is an integer greater than or equal to -1, and suppose the search algorithm works correctly whenever it's inputs consist of a key *key* of type E, and a tree T whose height is less than or equal to h.
Let T be a binary search tree of height h + 1 greater than or equal to zero. We know T is not empty, so it contains some element with key or type E stored at the root of T (for which the key is accessible as curr.key, since curr begins at the root of T and is not null since T is not empty.

Now, the key we are searching for is either less than curr.key, equal to curr.key, or greater than curr.key.

We consider all of these cases below:

Since the tree is not empty, curr = T.root, which is not null, and V is null, so the loop test passes.

- If *key* is equal to curr.key, then the following tests fails, so we assign the value of curr.key to V, and the loop guard fails at the next iteration, so we skip over the loop body and go to the if statement, if(v = null), which fails, since v is now equal to curr.value, so we execute the else, and return the value of v with *key* or T being changed.

- If *key* is less than curr.key, then we know that the first test failed, so curr.key is not equal to *key.* So we assign the curr.left, the left child of curr, to curr. Now we must loop again, v is no longer null and:

    - Now *key* is either the key of a value stored in T or it is not.
    - If *key* is a value stored in T then it follows the binary search tree property and the key is stored somewhere in the left subtree of T, which is accessed by continually accessing curr.left.
      Since T has height h+1, the height of the left subtree is an integer between -1 and h, it now follows by the inductive hypothesis that the execution of the loop on inputs curr = curr.left as the left subtree, T.left and *key* terminates and v = value of curr.key as the output without changing terminates and v = value of curr.key as the output without changing *key* or the left subtree, which has been accessed by accessing curr.left.

      The output produced, V, is immediately returned as the output of the algorithm, so it is clear that the algorithm terminates and returns an expected output without changing T or *key.*

    - On the other hand, if *key* is not stored in T then it won't be stored in the left subtree, and we won't be able to access it by accessing curr.left. Since the height of the left subtree is an integer between -1 and h, it follows by the inductive hypothesis that the execution of the while on the left subtree by accessing curr.left and v = null terminates without assigning a value to V, and a KeyNotFoundException being thrown without changing the value of T or *key.*

- Finally, if *key* is greater than or equal to curr.key, then both of the previous tests "curr.key = key" and "curr.key < key" so we execute the else of the algorithm, so we assign the curr.right, the right child of curr, to curr. Now we must loop again, v is no longer null and either *key* is stored in T or it is not. Each subcase is considered:
    - If *key* in T then it follows by the Binary Search Tree Property that *key* is stored in the right subtree of T, which we may access by continually accessing curr.right.
      Since T has height h + 1, the height of the right subtree is an integer between -1 and h. It now follows by the inductive hypothesis that the execution of while loop on inputs curr=curr.right and v = null terminates and v is returned with the value of curr.key as output withoutout changing *key* or T.

      The produced output is returned as the output of the while loop and it is clear than the algorithm terminates and eturned an expected output.

    - On the other hand, if *key* is not stored in T then it won't be stored in the right subtree, and we won't be able to access it by accessing curr.right. Since the height of the right subtree is an integer between -1 and h, it follows by the inductive hypothesis that the execution of the while on the right subtree by accessing curr.right and v = null terminates without assigning a value to V, and a KeyNotFoundException being thrown without changing the value of T or *key.*

Thus the algorithm terminates and works correctly with inputs T and *key* in al cases, as is required.

Loop Variant: f(h + 1, h) = h + 1 – h.

- This function us integer valued
- The function decreases by at least one after evert iteration, h increases so

(h+1) – h decreases.

- If f(h+1, h) is less than or equal to zero, which is when h = -1, and the tree is empty, then the function terminates.

Thus, f(h+1, h) terminates and its existence implies that the loop must terminate.

2.

```
insert(T, K key, V value)

curr = root;
parent = null;

if root = null
        root = new bstNode(key, value)

end if

else

        while(curr != null)

                parent = curr;

          if key < curr.key
                curr = curr.left;
          else if key > curr.key
               curr = curr.right
          else
                throw KeyFoundException
        end while

end else

if key < parent.key
    parent.left = bstNode(key, value);
else
   parent.right = bstNode(key, value);
```

3. inOrderTraversal(T)

```
if root = null
        end inorder traversal
Stack bstStack = new Stack
curr = T.root;

while stack.Empty = false or curr != null
        if curr != null
                bstStack.push(curr)
```

```
			curr = curr.left
		else
			node = bstStack.pop;
			curr = node.right;
```

7. The data obtained from running A3Q6.java is what I would expect to obtain.
   - We know that a binary search tree's height is smallest when the tree is fullest, we can obtain this height using the equation: $\log_2(n+1)-1$.
   - We also a binary search tree's height is the biggest when it is a continuous chain of length n, so the maximal height can be obtained from the equation $n-1.$
   - If we use these equations to obtain the maximum and minimum height of 100 binary search trees of each n we tested we get:

	n=100	n=1000	n=10000	N=10000
Minimum	5.56	8.96	12.28	12.60
Maximum	99	999	9999	99999

Now, these values do differ quite a lot from the obtained values, however, one can expect that when n is sufficiently large and a binary search tree T is constructed with size n, it is very likely that the values of the maximum and the minimum will be much closer to the average height of m trees than the calculated maximum and minimum, since when constructing a tree randomly, it is extremely unlikely that we will get a completely full tree, or a continuous chain.

The upper-bound on the average is what I would expect, since it is calculated using the formula $3\log_2(n)$, and the calculated values are greater than that of the actual averages, so this value is as expected.

The worst-case bound on the maximum height of a random red-black tree of size n is also as expected, since the properties of a red-black tree to differ from those of a binary search tree, so the height will in fact differ, and this value was calculated using the formula $\log_2(n+1)$ $2¿$