

*I used 1 late day token in this assignment.

1. Algorithm

Search function

This function is added for convenience. Search function starts from the root and goes down the tree by comparing the current Entry's keys and the target key. If it finds the entry with the same key, it returns the pointer. If not, it goes down until it reaches null, and return the pointer to the IntEntry right above it.

Insert

first, it checks whether the tree is empty or not. If it is empty it inserts it to the root. After the first insert, it uses search function and find the IntEntry to be inserted. Then, insert the new entry it to the correct place by comparing the key. If the Entry is already there then we increment the value. So the value is equal to the number we inserted key.

find

It uses the search function to find the Entry with same key. if the key is found, we return the value. If not found, return 0.

inordertravel

This function is added for convenience. It starts from the root and uses recursion of left child, and then right child. inOrderTravel function compares the current node's value with the left and right integer argument so that it only goes into the recursion when it is inside the range.

countRange

countRange function uses inordertravel function to calculate the sum of values in the range,

Remove

If the tree is empty, it returns false. If it is not empty, it uses the search function to find the Entry with the target key. If it is not found, we return false.

When the Entry is found and the value is more than 1, we just decrement the value. If the Entry value is 1, then we must remove the Entry from the tree. From here we divide into 4 cases. The first 3 cases is where the target IntEntry has NULL as child. It just simply connect the target's parent with target's child and

delete the target. The last case is where the target IntEntry is an Internal node. We start by finding the LMI Node(the next node when traversed inorder way). This process is done by findLMINode function which is made for convenience. We move the key and value of the LMI Node to the target node, Then, we erase the LMI Node.

RB Insert

If the RB tree is empty. We just add the new Black entry to the root. After the first insert. We insert the new node which is Red, by the same algorithm of insert above. If the new inserted node and the parent node is red, double red problem is occurred. If so we go to doubleRed function which takes the pointer to newly inserted node and solves the problem.

doubleRed function checks whether it needs to be restructured or recolored by looking at the color of parent's sibling. If restructure is needed, it goes to restructure function. the restructure function classifies into 4 cases and solves the problem. The important part of restructure function is that it doesn't change the position of the grandparent Entry and only change the position and values of parent and its sibling. This allows us to remain the relationship with parent of grandparent Entry, and keep the color of the grandparent Node.

If recoloring is needed we just change the color of parent, and sibling as black. Then, if grandparent is not the root we the color to red. Since another double red problem can appear, we check for it and solve it by recursion.

RB remove

RBT remove first find the target node do delete, Similar to BST remove, it checks the value of the Entry and decrement the value if it is more than 1. If the value if 1, than we delete it in the same way as remove above. But one thing different is that we check for the double black while deleting process. If the double black node is detected, we set the flag to 1, and save the pointer to the double black node. Then we put that pointer and the pointer to the parent as an input to doubleBlack function. This function will receive these two argument and fix the problem.

doubleBlack function divides doubleBlack problem into 3 cases, restructure, recoloring, and adjustment.

restructure process uses the restructure function described above. recoloring is very simple color change process but it can lead to another doubleBlack problem so we solved it using recursion. adjustment process require restructure process too, after that the doubleBlack Node can be solved by recursion of doubleBlack function.

2. How I tested the correctness of operations.

Since BST insert and BST remove is in the part of RBT insert and RBT remove, I was confident that if I correctly implemented RBT insert and RBT remove, BST insert & remove will also be correct.

To check if I correctly implemented RBT insert & remove, I created 3 public bool functions

- isRBTreeDepthPropertyValid()
- isRBTreeInternalPropertyValid()
- isRBTreeRootPropertyValid()

The function that checks depth property and internal property uses recursion to check these properties. And the function that checks root property simply compare the root's color.

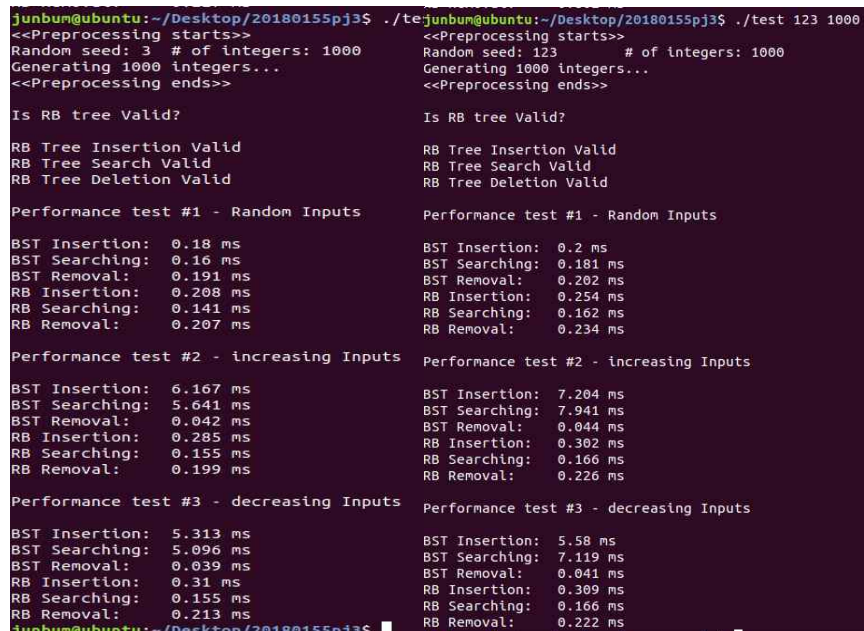
These 3 functions are always asserted at inserting and removing entries inside the tester_private.cpp. Since all of those properties are asserted at all insertion and deletion process, the RBT insert & RBT remove seems correct. (External property is trivial because the bottom of every node is NULL). And because RBT insert & remove is correct, so does BST insert & BST remove.

For, find and countRange function, I tested it with some arbitrary inputs and range several times and they all worked fine. so I assumed it is correct.

2. How I evaluated the performance over a series of operations

I tested the performance of BST & RBT using 3 series of input key. First series is just random **num** input in the range [0 , **maxNum** - 1]. Second series is a input that is in increasing order and third series is a input that is in decreasing order. maxNum is fixed to 1000 in the tester_private.cpp and only num variable can be changed using the terminal input.

3. Timing measurement Result



```
junbum@ubuntu:~/Desktop/20180155pj3$ ./test 123 1000
<<Preprocessing starts>>
Random seed: 3 # of integers: 1000
Generating 1000 integers...
<<Preprocessing ends>>

Is RB tree Valid?
RB Tree Insertion Valid
RB Tree Search Valid
RB Tree Deletion Valid

Performance test #1 - Random Inputs
BST Insertion: 0.18 ms
BST Searching: 0.16 ms
BST Removal: 0.191 ms
RB Insertion: 0.208 ms
RB Searching: 0.141 ms
RB Removal: 0.207 ms

Performance test #2 - increasing Inputs
BST Insertion: 6.167 ms
BST Searching: 5.641 ms
BST Removal: 0.042 ms
RB Insertion: 0.285 ms
RB Searching: 0.155 ms
RB Removal: 0.199 ms

Performance test #3 - decreasing Inputs
BST Insertion: 5.313 ms
BST Searching: 5.096 ms
BST Removal: 0.039 ms
RB Insertion: 0.31 ms
RB Searching: 0.155 ms
RB Removal: 0.213 ms

junbum@ubuntu:~/Desktop/20180155pj3$
```

```
tejunbum@ubuntu:~/Desktop/20180155pj3$ ./test 123 1000
<<Preprocessing starts>>
Random seed: 123 # of integers: 1000
Generating 1000 integers...
<<Preprocessing ends>>

Is RB tree Valid?
RB Tree Insertion Valid
RB Tree Search Valid
RB Tree Deletion Valid

Performance test #1 - Random Inputs
BST Insertion: 0.2 ms
BST Searching: 0.181 ms
BST Removal: 0.202 ms
RB Insertion: 0.254 ms
RB Searching: 0.162 ms
RB Removal: 0.234 ms

Performance test #2 - increasing Inputs
BST Insertion: 7.204 ms
BST Searching: 7.941 ms
BST Removal: 0.044 ms
RB Insertion: 0.302 ms
RB Searching: 0.166 ms
RB Removal: 0.226 ms

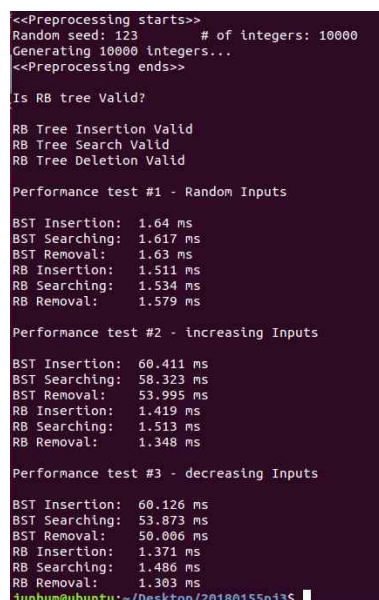
Performance test #3 - decreasing Inputs
BST Insertion: 5.58 ms
BST Searching: 7.119 ms
BST Removal: 0.041 ms
RB Insertion: 0.309 ms
RB Searching: 0.166 ms
RB Removal: 0.222 ms

junbum@ubuntu:~/Desktop/20180155pj3$
```

The captured pictures above are the results of performance tests where the number of interger is 1000, maxNum is 1000.

We can see that for random inputs, BST insert & remove is faster than RBT insert & remove. This seems fine because during the insertion and deletion process, RBT require more job(solving double red, double black problem). But because we make the tree more organized in RBT, the searching process should be faster then BST. And the result seems to match the theory.

For increasing and decreasing inputs, the BST insert takes longer than RBT. This is because BST's height is very long(same as input number of integer). So inserting and searching takes a longer time. But deleting will just simply take out the root element in both cases so it takes very small time.



```
<<Preprocessing starts>>
Random seed: 123 # of integers: 10000
Generating 10000 integers...
<<Preprocessing ends>>

Is RB tree Valid?
RB Tree Insertion Valid
RB Tree Search Valid
RB Tree Deletion Valid

Performance test #1 - Random Inputs
BST Insertion: 1.64 ms
BST Searching: 1.617 ms
BST Removal: 1.63 ms
RB Insertion: 1.511 ms
RB Searching: 1.534 ms
RB Removal: 1.579 ms

Performance test #2 - increasing Inputs
BST Insertion: 60.411 ms
BST Searching: 58.323 ms
BST Removal: 53.995 ms
RB Insertion: 1.419 ms
RB Searching: 1.513 ms
RB Removal: 1.348 ms

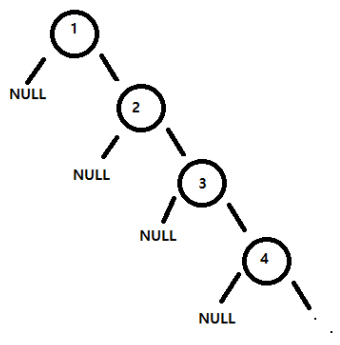
Performance test #3 - decreasing Inputs
BST Insertion: 60.126 ms
BST Searching: 53.873 ms
BST Removal: 50.006 ms
RB Insertion: 1.371 ms
RB Searching: 1.486 ms
RB Removal: 1.303 ms

junbum@ubuntu:~/Desktop/20180155pj3$
```

When the number of interger is more than numMax, the BST performance of test 2 and 3 is very worse even for the deletion, too. It is because the height of the tree is very long (same as numMax). And each time of insert, delete, search, we have to go through it.

4. Explain a scenario (a set of operations) where `IntRBTree` outperforms `IntBST`.

As I explained about the time result in 3. The scenario where RBT outperforms BST is when BST's height is very long. To be such case, the set of operation should like `insert(1)`, `insert(2)`, `insert(3)` ... `insert(999)`, `insert(1000)`, `delete(1000)`, `delete(999)`, ... `delete(2)`, `delete(1)`. If such case, the BST would look like below - the height of BST is equal to number of inserted Integer. But RBT would look very organized and the height of RBT will be $O(\log n)$.



The searching process of RBT must be faster than BST in this case.

If the number of integer is small, the insert and delete of BST could be faster than RBT but as the number grows, RBT insert & delete will become faster (since RBT insert and deletion require searching process too)

5. List all collaborators who discussed with you.

I didn't collaborate with anyone and only used the book and classum.