CSE221 Data Structures (Fall 2021)
Instructors: Prof. Young-ri Choi
TAs: Gyeongchan Yun, Jin Yang and Ju Young Bang
Due date: Dec 19, 2021, 11:59 pm.

Notes:

- Be aware of plagiarism. You are allowed to use Piazza for QnA, but do **NOT** discuss with your classmates **directly**.
- Check correctness of your code on **uni server** before final submission.
- Please carefully **follow the grading strategy** on the last part. We will not tolerate any issues caused by not following the strategy.
- Do **NOT** change the declaration of **given functions in the skeleton code**
- You can add your own global/static variable, function or classes, but do **NOT** use STL or other libraries except provided.
- Grading: 20 test cases (5pt for each test case).
- For grading, we only consider 3 header files in **assignment4/** directory: '**`Tree.h`**', '**`AVLTree.h`**', '**`RBTree.h`**'
- **IMPORTANT:** For this assignment, we will **NOT accept late submissions**.

Preparation Method

To start your assignment, you have to update your local git repository. Enter the below command.

```
$ cd uni+student_id
$ git pull origin master
```

If successfully updated, you will be able to see the assignment4/ directory.

Assignment 4: Balanced Binary Search Trees

In this assignment, you need to implement Balanced Binary Search Trees. We provide you `Tree.h` that contains a complete implementation of binary search tree and several skeleton files including `AVLTree.h` and `RBTree.h`. You should notice that class `AVLTree_t` in `AVLTree.h` and class `RBTree_t` in `RBTree.h` inherit class `Tree_t` in `Tree.h`. Each Balanced BST will need to handle all operators that we learnt from lecture 12 to 14. `AVLTree.h` and `RBTree.h` contain a set of functions you should implement for your Balanced BST. Please read comments in the files we provide for more details on how each function should work.

1. Rotation Implementation (total 20 pts)

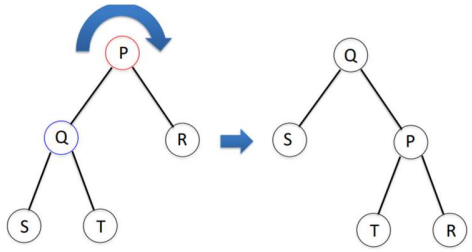

Figure 1

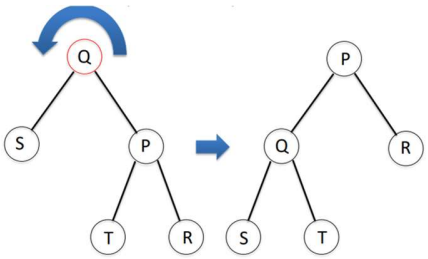

Figure 2

You need to implement one function in **`Tree.h`**:

- `void rotate(Node_t<keyT, valT>* n, bool direction)`

It receives one node `Node_t<keyT, valT>* n` and a direction of the rotation `bool direction` as arguments. Note that `Node_t* n` could be any node in the tree including the root and you could handle the parent node's pointers.

If the `direction` is **`true`**, then the function conducts a **clockwise** rotation and vice versa. For example, in Figure 1, if the function received node P and `true` direction as its argument, `rotate()` should do a clockwise rotation on P. Likewise, in Figure 2, if the function received node Q and **`false`** direction, then it should do a **counter-clockwise** rotation on Q. You can get more detail in lecture 12 for how rotation properly works.

2. AVL Tree Implementation (total 40 pts)

In this part, you need to implement two functions in **`AVLTree.h`**:

- `void insert(keyT key, valT value)`
  If there is no node that has the given key, `insert()` creates a new one and places it in the correct place, and it finally stores the data. Note that if there is already a node that has the given key, **update the data**, rather than making a new one. Also, you need to save the balance factor of each node .

- `bool remove(keyT key)`
  Find the node that has the given key and remove that node. Also, you need to save the balance factor of each node .
  After the removal process, the tree must keep all AVL tree conditions. For consistency, the function must traverse and **bring the node from the**

right subtree when the tree has both left and right subtrees. Also, note that the function **prefers single rotation over double rotation when both are possible** to re-balance the tree, i.e., LL over LR and RR over RL (The definition of notations are on pages 25-31 in Lecture 12).
If it tries to remove a non-existing node or the tree doesn't have the root node, `remove()` returns `false`.

Notes:

- Use `meta` variable in `Node_t` to store the balance factor.
- `check_bf()` function is provided to help you debug your code.
- We have included a few functions that you may find helpful, though it is not mandatory to implement them. Please check them in the skeleton code.

3. Red Black Tree Implementation (total 40 pts)

In this part, you need to implement two functions in **`RBTree.h`**.
Note that you must construct the RB tree strictly based on lecture 14 (RB tree part 2) for consistency.

- `void insert(keyT key, valT value)`
  If there is no node that has the given key, `insert()` creates a new one and places it in the correct place, and it finally stores the data. Note that if there is already a node that has the given key, **update the data**, rather than making a new one. Also, you need to save the color of each node.

- `bool remove(keyT key)`
  Find the node that has the given key and remove that node. Also, you need to save the color of each node. The color maps (`RED` and `BLACK`) were defined as macro on top of the skeleton code.
  After the removal process, the tree must keep all RB tree conditions. For consistency, the function must traverse and **bring the node from the right subtree** when the tree has both left and right subtrees.
  If it tries to remove a non-existing node or the tree doesn't have the root node, `remove()` returns `false`.

Notes:

- Use `meta` variable in `Node_t` to store node color (red, black).
- `check_rb()` function is provided to help you debug your code.
- We have included a few functions that you may find helpful, though it is not mandatory to implement them. Please check them in the skeleton code.

Grading strategy

(1) We will **compare only the printed outputs**. For example, you can get credit for a certain test case only when your program gives a perfect output for that case. So please **take care of your printing snippets, especially any trace of debugging stages.**

(2) In evaluation time, you will receive a **ZERO score** for the specific test case (among lots of test cases) where your program shows any kind of **GitLab conflict error, compile error, runtime error, or infinite loop** over timeout (10 secs). For your information, any warning messages during compilation don't affect grading. Again, please do a final check on **uni-server**, not on your arbitrary environment because we will conduct the evaluation on uni-server.

Testing Your Code

We have provided three cpp files to test your code. You can use **`Tree_rotate_test.cpp`** for testing rotation, **`AVLtest.cpp`** for AVL Tree implementation, and **`RBtest.cpp`** for Red Black Tree implementation. These files will test your tree implementation by printing the formation of tree via pre-order and level-order traversal after each set of instructions.
You can find the answer output files for each test cpp files: Tree_rotate_test.txt, AVLtest.txt and RBtest.txt.
We emphasize that the given test cases are examples of grading methods, not the exact same test cases that we will use for grading.

Compile/Execution Method

We will compile your code by **ONLY** the Makefile we provided, which explicitly allows C++11 grammar. We emphasize that **you are not allowed to modify the Makefile**. The grading will only be done with the original Makefile. Any compile error occurred due to using different compilation will not be tolerated.

1. **Rotation implementation**
   Compile command:
   ```
   $ make rotate
   ```
   If compilation is done, enter the execution command:
   ```
   $ ./rotate
   ```
2. **AVL Tree Implementation**
   Compile command:
   ```
   $ make avl
   ```
   If compilation is done, enter the execution command:

```
$ ./avl
```

3. **Red Black Tree implementation**
   Compile command:
   ```
   $ make rb
   ```
   If compilation is done, enter the execution command:
   ```
   $ ./rb
   ```

To compile all three test cases, enter the following command.
```
$ make
```
If compilation is done, three executable files: rotate, avl, and rb are generated.

Check Correctness Method

We provide bash script `check_testcase.sh` to check the correctness of original given test cases. This is an **optional** method because **it works only with original given test cases.** In other words, we cannot guarantee that the script works properly after modifying the test cases by yourself. For your information, the grading will be conducted in a similar manner.

1. **Rotation implementation**
   ```
   $ bash check_testcase.sh rotate
   ```
2. **AVL Tree Implementation**
   ```
   $ bash check_testcase.sh avl
   ```
3. **Red Black Tree implementation**
   ```
   $ bash check_testcase.sh rb
   ```

To check all three test cases, enter the following command.
```
$ bash check_testcase.sh all
```
If output of your code is the same as the answer file, for example of rotation implementation, the below message is shown:

```
[INFO] Correct rotate!
```

Submission Method

We only consider 3 header files in **assignment4/** directory for grading:
'`Tree.h`', **'`AVLTree.h`'**, **'`RBTree.h`'**

This assignment must be submitted through GitLab. You will be able to push your source code through the following commands.

```
$ cd uni+student_id/assignment4
$ git add Tree.h  AVLTree.h  RBTree.h
```

```
$ git commit -m “Final commit” (commit message is not
important for grading)
$ git push origin master
```

To double check whether your submitted code has any problem (e.g, git conflict, compile error, etc) or not, please follow the below commands.

```
$ git pull origin master
$ make
```