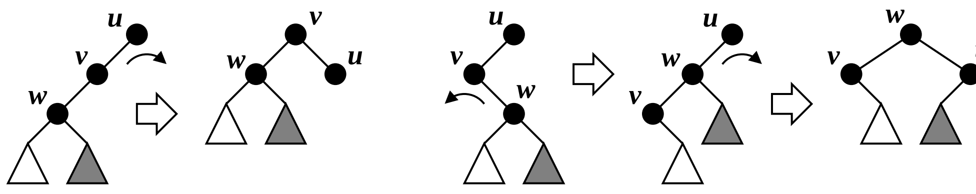
CS69001 Computing Laboratory – I

Assignment No: B3

Date: 13–August–2018

Let T be a binary search tree (BST) with n nodes. Let $h = h(T)$ denote the height of T (length of a longest path from the root to a leaf node). Smaller values of h are desirable for enhancing search, insertion and deletion efficiency. In the best case, $h = \lceil \log_2 n \rceil$, whereas in the worst case, $h = n - 1$. We call T height-balanced if $h = \Theta(\log n)$. There are well-known algorithms for balancing or rebalancing the heights of BSTs. In this assignment, you deal with local adjustments in T so as to decrease its height at a reasonable cost of $O(n)$. This technique is not guaranteed to lead to a height-balanced tree, but can practically improve the height of T considerably.

To this end, we define a 2-chain in T as a sequence u, v, w of three nodes satisfying the conditions: (i) each of u and v has only one child, (ii) v is the only child of u, and (iii) w is the only child of v. If v is the left child of u, and w is the left child of v, we call u, v, w an LL chain. We can analogously define LR, RL, and RR chains. The following figure illustrates an LL chain and an LR chain. This assignment deals with removal of all 2-chains from T. The final tree must be a BST storing the same keys as the input tree T.



Removing LL chain Removing LR chain

As the above figure indicates, 2-chains are eliminated by using rotations. The case of LL and RR chains is simpler. The left side of the figure demonstrates that a right rotation at u removes the LL chain starting at u. Handle the case of RR chains symmetrically.

The right side of the above figure illustrates the case of an LR chain. A right rotation at u converts the chain u, v, w to an RL chain v, u, w. A double rotation helps us achieve the desired objective. We first make a left rotation at v. This makes u, w, v line up along left-child links. If the right subtree of w (the shaded tree) is non-empty, the earlier 2-chain is already removed, but the eventual goal of reducing the height at u is not yet fully addressed (the unshaded subtree is still at a distance of two from u). A second rotation (right) at u now takes care of both the objectives: chain removal and height shrinking. The symmetric case of RL chains can be handled analogously.

Part 1: Define a data type for storing a node of a BST T. Each node is supposed to store only an integer-valued key and two child pointers. Do not include any other field (like parent pointers) in a node. Implement the standard BST insertion procedure in a function **`insert`**. Notice that you do not store duplicate keys in a BST. If a key already present in the BST is attempted to be inserted, the result is no change in the tree.

Part 2: Write functions for the **`preorder`** and **`inorder`** printing of the keys of a BST T. Write another function **`height`** to compute and return the height of T. Write yet another function **`nodecount`** to compute and return the number of nodes in T.

The *level* of a node u in a BST (or binary tree) T with root r is the length of the unique r,u-path in T. Thus, the root itself is at level 0, its children are at level 1, the grandchildren of the root are at level 2, and so on. Write a function **`avglevel`** to compute the average of the levels of all the nodes in the tree T. You may instead write a function **`sumlevel`** to return the sum of the levels of the nodes in T. This sum, divided by the number n of nodes in T (as returned by **`nodecount`**), gives the average level.

Part 3: Write a function **`count2chains`** to get a statistics of the four types of chains (LL, LR, RL, and RR) in a BST T. The function should print the counts of chains of individual types, and also the total count of 2-chains in T.

Part 4: Write the functions **`lrotate`** and **`rrotate`** to perform left and right rotations at a specified node in the BST T. These functions perform *single* rotations. If you consider it handy, you may also right a couple of functions performing double rotations.

Part 5: Write a function **`rm2chains`** to remove all the 2-chains in a BST T. Your function should make a preorder traversal though the tree, and make single or double rotations whenever a 2-chain is located. In order to simplify the process, you may create a dummy node D, let it store the key $+\infty$, and set its left- and right-child pointers point to the root of the original tree and an empty tree, respectively. After **`rm2chains`** (the outermost call) returns, the root of the BST should be changed to the left child of D—this is needed, because a rotation, if made at the initial root of T, introduces a different node at the root position.

The *main*() function

- The user first enters the number **`nins`** of insertions that (s)he wishes to make in an initially empty BST T. (S)he then supplies **`nins`** integer-valued keys. Call **`insert`** to insert the supplied keys to T. Let n be the number of nodes in T after **`nins`** insertions. The user is allowed to supply duplicate keys, so n may be smaller than **`nins`**.
- Print the **`preorder`** and **`inorder`** listings of the keys in T. Also, print the height and the average level of T (see Part 2). You may also print the output of **`nodecount`**.
- Call the function **`count2chains`** of Part 3 to print the 2-chain statistics of the current tree T.
- Call **`rm2chains`** to remove all the 2-chains of all types from T.
- Again print the information about the tree: the preorder and inorder listing of the keys, the height and the average level, and the new 2-chain statistics (should be all zeros).

Sample output

```
Enter nins: 34
Enter keys: 325 318 359 166 356 175 181 349 328 341 329 316 335 183 206 208 190
            213 197 306 299 290 335 286 258 243 255 234 277 306 253 222 228 267

+++ Initial tree
    Preorder :  325 318 166 175 181 316 183 206 190 197 208 213 306 299 290 286
                258 243 234 222 228 255 253 277 267 359 356 349 328 341 329 335
    Inorder  :  166 175 181 183 190 197 206 208 213 222 228 234 243 253 255 258
                267 277 286 290 299 306 316 318 325 328 329 335 341 349 356 359
    Height of tree  = 18
    Average level   = 8.750000
    Number of nodes = 32

+++ Counts of 2-chains
    LL: 5, LR: 5, RL: 3, RR: 3, Total: 16

+++ Removing 2-chains

+++ New tree
    Preorder :  325 175 166 316 183 181 206 190 197 213 208 299 286 258 243 228
                222 234 255 253 277 267 290 306 318 356 341 329 328 335 349 359
    Inorder  :  166 175 181 183 190 197 206 208 213 222 228 234 243 253 255 258
                267 277 286 290 299 306 316 318 325 328 329 335 341 349 356 359
    Height of tree  = 11
    Average level   = 5.531250

+++ Counts of 2-chains
    LL: 0, LR: 0, RL: 0, RR: 0, Total: 0
```

Submit a single C/C++ source file. Do not use global/static variables. Do not invoke any STL features.