Stephen Chambers (smx227)
CS858

Assignment 4 Writeup

1. **Briefly list any parts of your program which are not fully working. Include transcripts or plots showing the successes or failures. Is there anything else that we should know when evaluating your implementation work?**

   The red black tree works correctly and is faster than BST and a linked list. The plots are attached.

2. **If you didn't do it in class, write down precise psuedo-code for binary tree insertion without looking at any books or notes. Now, prove that your psuedo-code is correct. If you discover a flaw, you are allowed to write new psuedo-code without any penalty, but please turn in both the original and the final versions.**

   Binary Tree Invariant: Assuming that duplicates are not inserted, the invariant for a binary tree is that for any node *n*, all of the nodes in *n*'s left subtree are smaller than *n,* and all of the nodes in *n*'s right subtree are larger than *n*.

   Find-parent is called with *z*, the node we want to insert, *t*, the root of the tree, and *null*. In order to prove correctness, we will go through all of the insertion cases, verifying that the invariant holds.

   a) Tree is empty
      If the tree is empty, *t* is NULL. In find-parent, if cur is NULL, parent is returned, which is passed into the function as NULL. In the insert method, if parent was null, the root is set to z. The invariant holds because z is the only node in the tree.

   b) Tree's root element is larger than z
      When find-parent is called, z.key will be smaller than cur.key, because the root element is larger than z. Therefore, the routine will recurse on cur's left. This same process will repeat, constantly making sure that find-parent will return a node in the correct subtree. The invariant holds because at each node, the algorithm compares z.key to that nodes key and 'places' the node in the correct subtree. Assuming that the algorithm makes sure the invariant holds for the last node, which it does in the 'else' block of insert, it is guaranteed that the invariant described above will hold.

   c) Tree's root element is smaller than z
      When find-parent is called, z.key will be larger than cur.key, because the root element is smaller than z. Therefore, the routine will recurse on cur's right. This same process will repeat, constantly making sure that find-parent will return a node in the correct subtree. The invariant holds because at each node, the algorithm compares z.key to that nodes key and 'places' the node in the correct subtree. Assuming that the algorithm makes sure the invariant holds for the last node, which it does in the 'else' block of insert, it is guaranteed that the invariant described above will hold

3. **Exercise 13.1–7 in CLRS.**
   The largest possible ratio of red internal nodes to black internal nodes is 2:1. This is because of property 4 of a binary tree, stating that "If a node is red, then both its children are black". Assuming that we always have the maximum amount of red nodes in our red black tree:
   1. If the root at depth 0 is black, there is a maximum of 2 red children at depth 1.
   2. If there are 2 red children at depth 1, there *must* be 4 black children at depth 2.
   3. If there are 4 black children at depth 2, there is a maximum of 8 red children at depth 3.
   4. If there are 8 red children at depth 3, there *must* be 16 black children at depth 4, etc.

   The smallest possible ratio is 0, when the tree is perfectly balanced (i.e. one black root node).

4. **Exercise 13.3–1 in CLRS.**
   This would not allow the tree to be balanced. If every node that was inserted were black, inserting a sorted list of numbers would cause the same behavior that an unbalanced BST would. This is because of line 1 in RB-INSERT-FIXUP. Every node we are inserting is black, therefore, z's parent will never be red, and the tree will never rotate itself to preserve balance.

5. **Exercise 13.3–4 in CLRS.**

   The following points are when a node is colored red in RB-INSERT-FIXUP. For each point, I will prove that T.NIL cannot be colored red in that scenario.

   a) In cases 1 and 3, z's grandparent is colored red. If z's grandparent is T.NIL, then z must be the root. However, the algorithm never would have reached case 1 or case 3 if z was the root, because line 1 specified that z's parent has to be red. Since z is the root, z's parent is black, and the algorithm would skip to line 16, color the root black, and return.

   b) In the two symmetric cases to case 1 and case 3, T.NIL cannot be red for the same reasons.

6. **(Those in 858 only) Exercise 13.4–6 in CLRS.**

   1. Case 1 is only executed if *w* is red.
   2. The node *w* is defined to be *x*'s sibling, which means they have the same parent.
   3. If *w* is red, *w*'s parent cannot be red, because that would violate property 4 of the red black tree (if a node is red, both its children are black).
   4. Therefore, w's and x's parent must be black

7. **What suggestions do you have for improving this assignment in the future?**
   Being provided test files containing insertions/deletions to mirror the cases we discussed in class would've been tremendously helpful.