Quoc Ho 001200151

CPSC 3620 Project
Red Black Tree

i) Red Black Tree:

Red Black Tree (RBT) is a self-balancing Binary Search Tree.

Properties:

1. Every node is either RED or BLACK
2. Root node is BLACK.
3. Every leaf (NULL node) is BLACK
4. If a node is RED, then both its children are BLACK.
5. Every path from a given node to any of its descendant leaf (NULL nodes) contains the same number of BLACK nodes.

Operations:

1. Search (Find): O(log n)z

Perform the same as AVL tree. Start from the root, if less than the current node then travel left, if greater then travel right.

2. Insert: O(log n)

In RBT, we use recoloring and rotation to do balancing. We try recoloring first, if it does not work then we try for rotation. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Depend on the color of the nearby nodes, there are several cases of insertions in RBT to insert a new node x:

a) Perform BST insertion and make the color of the newly inserted nodes as RED.
b) If x is root, change color of x as BLACK.
c) Do the following if the color of x's parent is not BLACK or x is not root:

1. If x's uncle is RED
    i) Change color of parent and uncle to BLACK
    ii) Color of grandparent as RED
    iii) Change x = x's grandparent, repeat i and ii for new x.
2. If x's uncle is BLACK, then there can be four configurations for x, x's parent (p) and x's grandparent (g):
    i) Left Left Case: p is left child of g and x is left child of p
    ii) Left Right Case: p is left child of g and x is right child of p

iii) Right Right Case: mirror of case i
iv) Right Left Case: mirror of case ii

3. Delete: O(log n)

We check the color of siblings to decide the appropriate case. The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to convert this double black to single black.

__**Deletion steps:**__ Let v be the node to be deleted and u be the child that replaces v.

1. Perform BST deletion.

2. If either u or v is RED: we mark the replaced child as BLACK. Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.

3. If both u and v are BLACK:

3.1. Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.

3.2. Do following while the current node u is double black and it is not root. Let sibling of node be s.

a) If sibling s if BLACK and at least one of sibling's children is RED: perform rotation(s). Let the child of s be r. This case can be divided into 4 subcases depending on the positions of s and r:

i) Left Left Case: s is left child of its parent and r is left child of s or both children of s are RED.

ii) Left Right Case: s is left child of its parent and r is right child.

iii) Right Right Case: s is right child of its parent and r is right child of s or both children of s are RED.

iv) Right Left Case: s if right child of its parent and r is left child of s.

b) If sibling is BLACK and its both children are BLACK: perform recoloring and recur for the parent if parent is BLACK.

c) If sibling is RED: perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black. This mainly converts the tree to black sibling case (by rotation) and leads to case **(a)** or **(b)**. This case can be divided in two subcases.

i) Left Case: s is left child of its parent. We right rotate the parent p.

ii) Right Case: s is right child of its parent. We left rotate the parent p.

3.3. If u is root, make it single black and return.

ii) Operations worst-case running time:

Find(): O(log n)
Insert(): O(log n)
Delete(): O(log n)
Print(): O(n)
Worst case running time: O(n log n)
Space requirement: O(n)

- The program has a problem with balancing height property after insertion some values. The height different can be > 1.
- The program can not display a CBT with high input elements properly.
- The program has a problem with coloring after insertion.
- Rebalancing after deletion operations works fine.
- Error when try to delete the whole RBT because of property 2 where the root is BLACK. Using exception should fix the issue.

iii) Experiment: Testing with CodeBlock on Windows 8.1 for run time. Used ressort monitor for RAM usage.

1) Insert: 1 → 10
No Delete operation:
Time 1: 0.037s
Time 2: 0.023s
Time 3: 0.022s
Memory Usage: 72 KB

2) Insert: 1 → 100
Time 1: 0.152s

Time 2: 0.095s
Time 3: 0.104s
Memory Usage: 72 KB

3) Insert: 1 → 1000
Time 1: 1.104s
Time 2: 1.154s
Time 3: 1.161s
Memory Usage: 72 KB

4) Insert: 1→ 10,000
Time 1: 16.326s
Time 2: 16.282s
Time 3: 16.514s
Memory Usage: 72 KB

5) Insert: 1 → 100
Delete: 25→ 75
Time 1: 0.081s
Time 2: 0.086s
Time 3: 0.093s
Memory Usage: 72 KB

6) Insert: 1 → 500
Deletion: 2 → 500
Time 1: 0.029s
Time 2: 0.032s
Time 3: 0.030s
Memory Usage: 72 KB

7) Insert: 1 → 5000
Deletion: 2 → 5000
Time 1: 1.055s
Time 2: 1.077s
Time 3: 1.055s
Memory Usage: 72 KB

iv) The running times of the same test sample had a very small margin with CodeBlocks. So the running time should be accurate. Memory usage measurement was only visible when comparing between extremely huge input size.

The running time is mainly consisted of print(). When inserted 1→ 10000, the running time without print() is 2.064s. However, when inserted 1→ 5000, and delete 2→ 5000, the running time without print() is 1.044s. That was within the average of the test running time with print(). It was expected because print() was the last function to run, which only had 1 variable to be printed, after insert() then delete().

v) Theoretically, the running time without print() should be smaller than with print() running. As we could see the huge improvement with test 4, without print() the running time was 2.064s compare to 16.326s with print(). This was because there was no delete() so the whole tree was printed.

The print() paid little role when the whole tree was inserted then deleted. This leave only the root or no tree to be run, so the running time for print() is O(1). Only running time of delete() and insert() are counted.

Conclusion: the actual running time reflects the theoretically running time.