Introduction to Data Structures (430.217, 002)
Seoul National University

Project #2 Postcode Management System
Due : 30 Nov 2022, 23:59:59

INSTRUCTIONS

- You are not allowed to copy and paste the code of others (e.g., classmates or website). Please try to work on your own.
- It is desirable to check if your codes run on the environment we have introduced in the Linux tutorial. The version of g++ and c++ should be 7+ and 14+, respectively. If it works only on your PC, there may be deduction of the score.
- All kinds of questions are welcome. If you have problem or find out typos in the given skeleton code, please leave the question on the eTL QnA board. It is recommended to make it open to your classmates. However, we are not going to fix your bugs directly.
- Grading policy will be as follows.
  - 100 points for 100 test cases.
  - Late submission : 10% deduction per 12 hours
  - Memory leak or memory error : 30% deduction
  - Only working on your machine : 20% deduction
  - Any other non-compliance with given rules will be also deducted.

BACKGROUND

Binary Search Tree (BST)

A BST is a tree in which all the nodes follow the bellow properties.

1. Each node has two attributes: Key and Value, and the tree is aligned based on the key.
2. The keys in the left subtree are smaller than its parent node's key.
3. The keys in the right subtree are greater than its parent node's key.
4. Duplicate keys are not allowed. So, if there is already the node with the key in the tree, only the value of the node should be replaced by the new value.
5. Each of the left and right subtrees must be a binary search tree again.

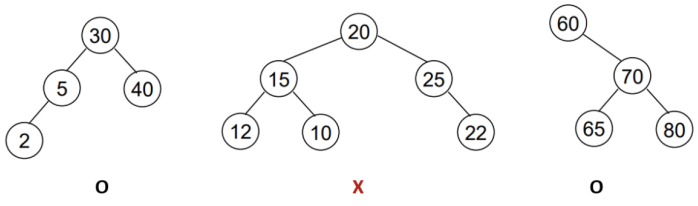

Figure 1: Examples of whether the tree is a binary search tree or not

Figure 1 shows examples of whether each tree is a binary search tree or not. Most of the BST operations (e.g., search, insert, delete, etc.) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree (i.e., worst case). If the tree is perfectly balanced, the height of the tree is always O(log n) where n is the number of nodes in the tree. That means, even in the worst case, the time complexity becomes O(log n). **AVL Tree** and **Red-Black Tree** are the balanced binary search trees.

Introduction to Data Structures (430.217, 002)
Seoul National University

Project #2 Postcode Management System
Due : 30 Nov 2022, 23:59:59

AVL Tree

AVL Tree is height-balanced binary tree. First, we define the balance factor as follows:

balance factor of node n = $h_{left}(n) - h_{right}(n)$

AVL Tree must satisfy that the balance factor of every node n to be 1, 0, or -1. That is, the height of sibling trees differs no more than 1. Figure 2 shows examples of satisfying or not satisfying the condition of the AVL Tree.

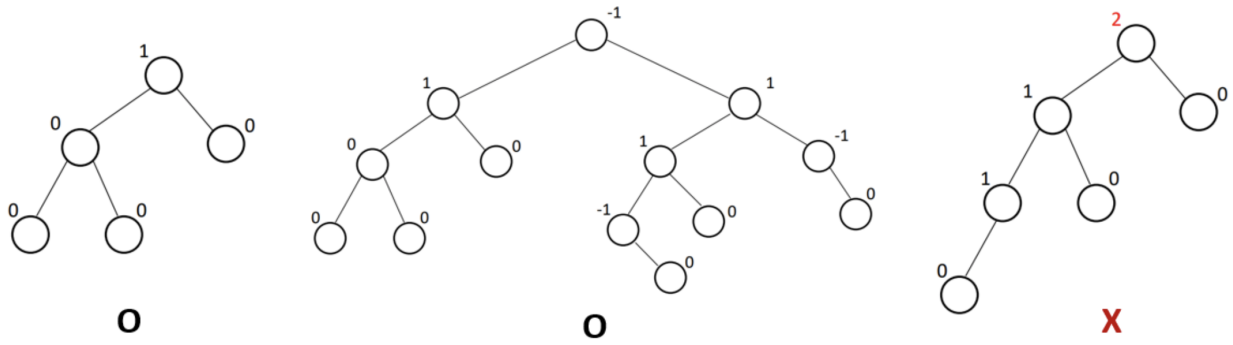

Figure 2: Examples of whether the tree is a AVL tree or not

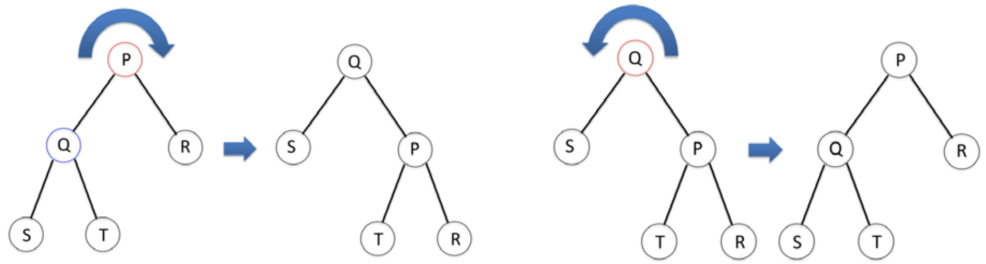

Figure 3: Tree Rotation

Before you start to implement AVL Tree, you should first implement the tree rotation. This is fundamental operation on a balanced binary tree to maintain the balance of the tree for every insertion and deletion. Figure 3-(a) shows right rotation on node P. As shown the figure, the function for right rotation conducts a clockwise rotation. The left child node (i.e., Q) of the target node (i.e., P) moves up and the target node moves down. Likewise, the function for left rotation conducts a counter-clockwise rotation as shown in the Figure 3.

Introduction to Data Structures (430.217, 002)
Seoul National University

Project #2 Postcode Management System
Due : 30 Nov 2022, 23:59:59

Red-Black Tree

Red-Black Tree is null-path-length-balanced binary tree. Red-Black Tree is a tree in which all the nodes follow the bellow properties. Figure 4 shows examples of satisfying or not satisfying the condition of the Red-Black Tree.

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. All leaf(nil) nodes are black nodes.
4. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
    - This property implies that When starting from the root node, the distance to the farthest path is always less than twice the distance to the nearest path.
5. Every path from a node (including root) to any of its descendants NIL nodes has the same number of black nodes(=black height).
    - Black height is greater than the half heights of the tree.

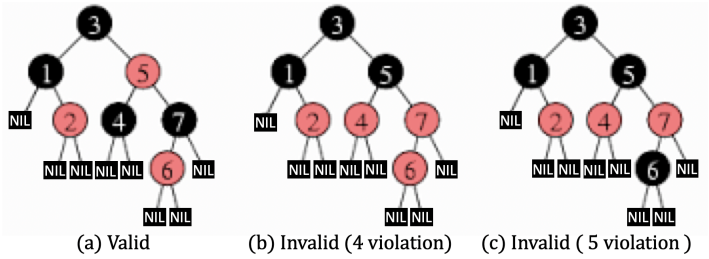

Figure 4: Examples of whether the tree is a Red-Black tree or not

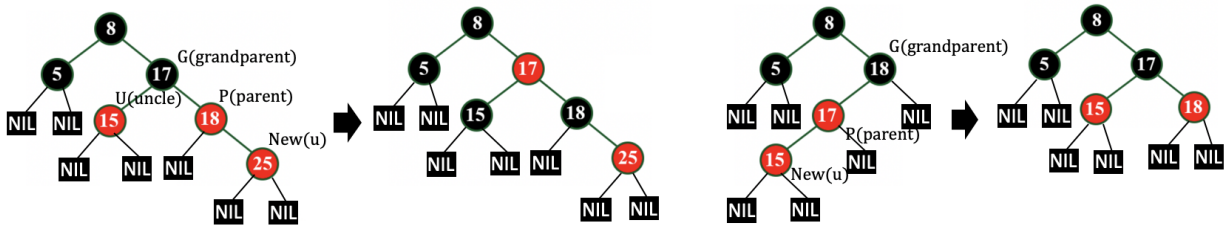

Figure 5: Fundamental operation examples of Red-Black Tree

Before you start to implement Red-Black Tree, you should implement the the tree rotation and node recoloring. These are fundamental operations on a balanced binary tree to maintain the balance of the tree for every insertion and deletion. The examples are shown in the Figure 5. Both example operates appropriately for solving violated property 4. In Figure 5-(a), all property is satisfied by ”node recoloring of U, P and G”. In Figure 5-(b), all property is satisfied by ”tree rotation of G” and ”node recoloring of U and G”.

IMPLEMENTATION

1 AVL tree (50 points)

The goal is to complete the skeleton code, "avltree.h", by filling in all the TODOs. The functions you need to implement are summarized as follows:

In BSTree class of "bstree.h":

- **AVLNode<T,U>* insert** (AVLNode<T,U>*& node, const T& key, const U& value):
  - Allocate a memory for the new node and insert the node with key and value. Note that the pointer of a particular node should be returned to execute function recursively.
- **AVLNode<T,U>* remove** (AVLNode<T,U>*& node, const T& key):
  - Remove the node with key in the tree. Note that the pointer of a particular node should be returned to execute function recursively.
- **U search** (AVLNode<T,U>*& node, const T& key):
  - Return the value of node with the key. Return "" if there is no such key.
- **void removeall** (AVLNode<T,U>*& node):
  - For destructor, deallocate and remove all nodes in the tree. There must be NO memory leak until the program has shut down. You will get points deducted if you let memory leaked.
- **AVLNode<T,U>* rotate_left** (AVLNode<T,U>*& node):
  - Left rotation on the given node (i.e., counter-clockwise).
- **AVLNode<T,U>* rotate_right** (AVLNode<T,U>*& node):
  - Right rotation on the given node (i.e., clockwise).

* Tree Rotations

There are 4 different cases of tree rotations. You should not only make the tree balanced, but also satisfy the condition of the binary search tree. You can easily understand the situation with following figures.

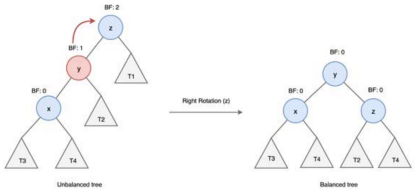

Figure 6: LL (Left Left) case

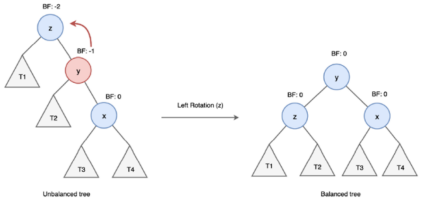

Figure 7: RR (Right Right) case

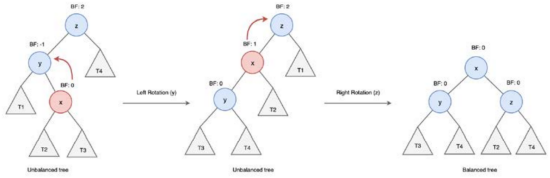

Figure 8: LR (Left Right) case

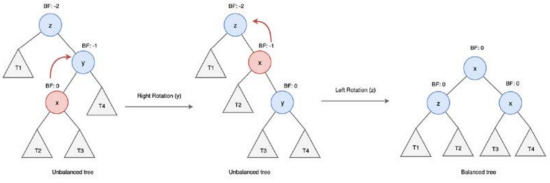

Figure 9: RL (Right Left) case

The balance factor of the node z is 2 or -2 for each case. The “rotate” operation should be performed to make the subtrees balanced. For LL and RR cases, only one rotation is processed. For LR and RL, two different rotations are processed sequentially. Some subtrees (T2 in LL case) are detached from original parent nodes (y) and attached to new parent nodes (z). The “rotate_left” and “rotate_right” function return one node pointer to make the process recursive

1.1 Insert Implementation (20pts)

AVLNode<T,U>* insert (<T,U>*& node, const T& key, const U& value):
If there is no node that has the given key, create a new node, place it in the right position, and store the data. The newly added node must be placed properly as a leaf. If there already is a node that has the given key, then update the value of the node with key, rather than making a new one. You should update the height of the nodes. And then check the balance factor (height of the left subtree – height of the right sub tree) of the node. If the balance factor of every node is either -1,0, or 1, you do nothing. If it is greater than 1 or smaller than -1, you need to do "rotate" to the particular subtree.

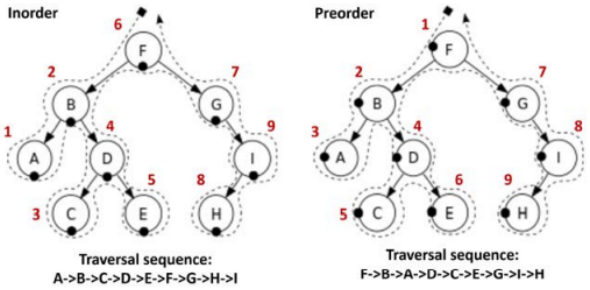

Figure 10: Tree traversal examples of inorder and preorder traversal

You can check your tree structure by using the functions of inorder() and preorder() which are provided to help you debug your code. In case of BST, inorder traversal gives nodes in increasing order and preorder traversal is a topologically sorted as shown in the Figure 10. To get more detail, refer to the link below.
https://en.wikipedia.org/wiki/Tree_traversal

1.2 Remove Implementation (30pts)

AVLNode<T,U>* remove (AVLNode<T,U>*& node, const T& key):
Find the node that has the given key and remove that node. Note that you should deallocate the node using "delete" operation after removing the node. Even after the remove operation, the tree should preserve inorder traversal sequence (i.e., binary search tree property).
For remove operation, first, you should consider 3 cases in insertion of Binary Search Tree as follows:

1. The node with the given key is not in a tree.
   - Do nothing.
2. The node is a leaf.
   - Just delete that node.
3. The node has one or more child nodes.
   - Find the max key from its left subtree or the min key from its right subtree. Then, the node to be deleted can be replaced with the max/min node. **(We unify min node from its right subtree if the node has two child nodes.)** If the node to be replaced has also one or more child, then you just repeat this process recursively. The figure 5 shows an example of the case 3.

After that, update the height of the nodes and check the balance factor. Do "rotate" if the tree is unbalanced. Same tree rotations are applied as the "insert".

2 Red-Black tree (50 points)

The goal is to complete the skeleton code, "RBtree.h", by filling in all the TODOs. The functions you need to implement are summarized as follows:

In RBTree class of RBtree.h":

- **RBNode<T,U>* insert** (RBNode<T,U>*& node, const T& key, const U& value):
  - Allocate a memory for the new node and insert the node with key and value. Note that the pointer of a particular node should be returned to execute function recursively.
- **RBNode<T,U>* remove** (RBNode<T,U>*& node, const T& key):
  - Remove the node with key in the tree. Note that the pointer of a particular node should be returned to execute function recursively.
- **U search** (RBNode<T,U>*& node, const T& key):
  - Return the value of node with the key. Return "" if there is no such key.
- **void removeall** (RBNode<T,U>*& node):
  - For destructor, deallocate and remove all nodes in the tree. There must be NO memory leak until the program has shut down. You will get points deducted if you let memory leaked.
- **RBNode<T,U>* rotate_left** (RBNode<T,U>*& node):
  - Left rotation on the given node (i.e., counter-clockwise).
- **RBNode<T,U>* rotate_right** (RBNode<T,U>*& node):
  - Right rotation on the given node (i.e., clockwise).

2.1 Insert Implementation (20pts)

RBNode<T,U>* insert (<T,U>*& node, const T& key, const U& value):
If there is no node that has the given key, create a new node N with red color, place it in the right position, and store the data. The newly added node must be placed properly as a leaf. If there already is a node that has the given key, then update the value of the node with key, rather than making a new one. After that, you should check it the insertion violated the red-black tree properties mentioned in background. If it did, you should fix it using MECE approach.

There are several cases you need to consider. Let us suppose P is a parent node, U is an uncle node, S is a sibling node and G is a grandparent node of N.

- **Case 1: Red-Black is empty.**
  - Make N the root of the tree and color it black.
- **Case 2: P is black.**
  - Do nothing because it cannot violate any of properties.
- **Case 3-1: P is red and U is red.**
  - Recolor node of P, U, and G. That means, P becomes black, U becomes black, and G becomes red. If G a root of tree, don't recolor G due to property 2.

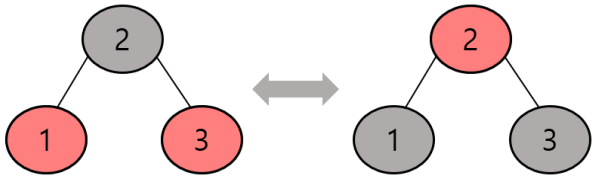

Figure 11: Recoloring without violation#5

- **Case 3-2: P is red and U is black (or NIL).**
  - **Case 3-2-1: P is right(left) child of G and N is right(left) child of P.**
    - Recolor G node and P node. Perform left(right) rotation on G.
  - **Case 3-2-2: P is right(left) child of G and N is left(right) child of P.**
    - First do the right(left) rotation on P. Next process is same with 3.2.1.

2.2 Remove Implementation (30pts)

RBNode<T,U>* remove (RBNode<T,U>*& node, const T& key):
Find the node that has the given key and remove that node. Note that you should deallocate the node using "delete" operation after removing the node. Even after the remove operation, the tree should preserve inorder traversal sequence (i.e., binary search tree property).

For remove operation, first, you should consider 3 cases in remove of Binary Search Tree mentioned in Remove Implementation of AVL tree.

When you remove a node, you should check if it violates the attribute, and consider the color to be removed. The color to be removed is determined by the color of the node to be removed when the child of the node is 0 or 1, and the color of the node's successor when the child is 2. Example is shown in the Figure 12. If the color removed is red, do nothing because no violation occurs. If the color removed is black, you should do extra operation to solve violation(#2, #4, #5).

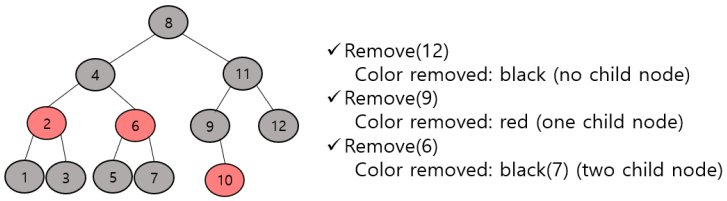

Figure 12: Removed color example

If the removed color is black, to satisfy the properties, an extra black is given to the node that replaces the location of the removed color, and the extra black is counted as one black node.

- If the **root of tree** is removed, the root's successor colors black to solve property #2.
- If a node which has **no child** is removed, property #5(,4) violated. Give nil node(black) extra black.
- If a node which has **one child** is removed, property #5(,4) violated. Give child of the node extra black.
- If a node which has **two children** is removed, property #5(,4) violated. Give the node that replaces the location of the removed color extra black.

If extra black is given to the red node, it becomes 'red and black'. If extra black is given to black node, it becomes 'double black'. 'Red and black' can be changed to black node. But 'doubly black' requires extra operations.

- **Case 1: Right(Left) sibling color is black, the sibling's right(left) child color is red.**
  - Color right(left) sibling to color of parents. Color parent and right(left) child of right(left) sibling to black. Perform left(right) rotation on parent.
- **Case 2: Right(Left) sibling color is black, the sibling's right(left) child color is black, the sibling's left(right) child color of the sibling is red.**
  - Color right(left) sibling to red. Color left(right) child of right(left) sibling to black. Perform left(right) rotation on right(left) sibling. After that, process with case 1.
- **Case 3: Right(Left) sibling color is black, right(left) sibling's two children colors are black.**
  - Color right(left) sibling to red. Pass the extra black from right(left) sibling to parent. If the node becomes 'red and black', it becomes black. If the node becomes 'doubly black', there are two cases. If the node is root, color root to black. If not, delegate to parent to handle case 1,2,3,4.
- **Case 4: Right(Left) sibling color is red.**
  - Color right(left) sibling to black. Color parent to red. Perform left(right) rotation on parent. After that, handle with cases 1,2,3 at the location of a node with extra black.

** Comparison

AVL tree is balanced compared to Red Black tree, but more rotations can occur during insert and remove. Therefore, if **inserts and removes occur frequently** in an application, it is recommended to use the **Red Black tree**. **AVL tree** is more efficient than Red Black tree if there are **fewer inserts and removes and more frequent searches**.

3 Test case

```
tree.insert(16545, "yeonnamdong");
tree.insert(54312, "sinsadong");

tree.insert(11243, "sillimdong");
tree.insert(66443, "bongcheondong");
tree.insert(55443, "jungangdong");
tree.insert(32340, "banpodong");
tree.insert(33450, "dobongdong");
tree.insert(25234, "samseongdong");
tree.remove(54312);
tree.insert(54155, "gurodong");
tree.insert(51211, "banghwadong");

tree.remove(11243);
```

Figure 13: Sample test case

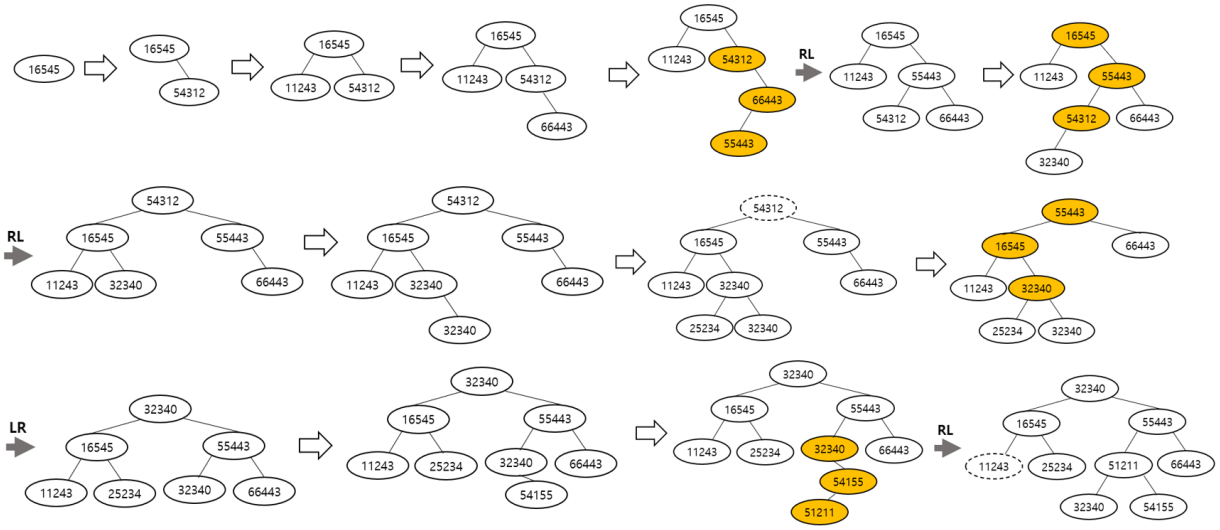

Figure 14: Progress of AVL tree in sample test case

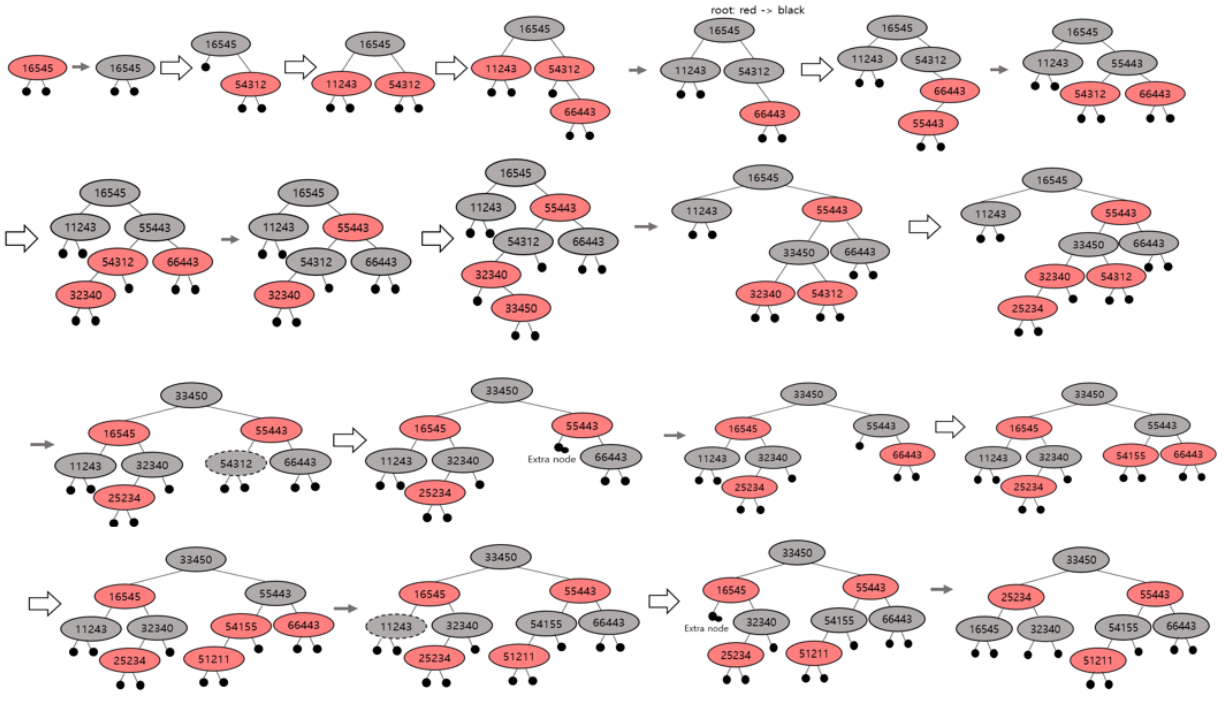

Figure 15: Progress of Red-Black tree in sample test case