Red-Black Trees

By Roy Dinh

A **red-black tree** is a **binary search tree (BST)** with the following facts:

- Every node is **red** or **black**
- The root is **black**
- Every leaf is **NIL** and **black**
- If a node is **red**, then both children are **black**
- All paths from the root to any leaf has the same number of black nodes

Performance Analysis

We will be measuring the time complexity of:

- Insertion
- Deletion
- Searching

(all functions can be found in the zip including helper functions)

In addition, we will compare red-black trees with other data structures like:

- AVL trees
- Normal binary search trees

Theorem

A red-black tree with n nodes has a height:

$$\mathbf{h \leq 2\log(n + 1)}$$

(Proof in the slides)

To simplify, **h ≤ logn**

This fact will be used to find the runtimes of the operations.

Insertion

Suppose we wanted to insert node z into a red-black tree of size n.

The processes RBLocateParent, takes **O(logn)** time because it traverses through the tree with height being logn.

Similarly, the functions RBInsertFixupB and RBInsertFixupA take worst case **O(logn)** time as well because they depend on the height of the tree.

So insertion in a red-black tree is **Θ(logn)**.
C + logn + logn + logn = 3logn ∈ Θ(logn)

```
void Tree::RBTreeInsert(Node* z) {
    z->left = NIL;
    z->right = NIL;

    Node* y = RBLocateParent(z);
    z->parent = y;
    if (y == NIL) {
        root = z;
    } else if (z->key < y->key) {
        y->left = z;
    } else {
        y->right = z;
    }
    z->color = RED;

    if (z->parent->color == RED
        && Sibling(z)->color == BLACK
        && z == z->parent->right) {
            RBInsertFixupB(z);
    } else {
        RBInsertFixupA(z);
    }
    root->color = BLACK;
}
```

Deletion

Suppose we want to delete a node z from a tree of size n.

The function call, Minimum takes **O(logn)** time because it goes down the tree to find the minimum key value of a node.

The function, DeleteFixup, takes **O(logn)** time as well because it traverses in worst case scenario through the height the tree to maintain properties of a red-black tree.

So deletion in a red-black tree takes **Θ(logn)**.
C + logn + logn = 2logn ∈ Θ(logn)

```
void Tree::Delete(Node* z) {
    Node* v = z;
    Color color = v->color;
    Node* x;
    if (z->left == NIL) {
        x = z->right;
        Shift(z, x);
    }else if (z->right == NIL) {
        x = z->left;
        Shift(z, x);
    }else {
        v = Minimum(z->right);
        color = v->color;
        x = v->right;
        if (v->parent == z) {
            x->parent = v;
        }else {
            Shift(v, v->right);
            v->right = z->right;
            v->right->parent = v;
        }
        Shift(z, v);
        v->left = z->left;
        v->left->parent = v;
        v->color = z->color;
    }
    if (color == BLACK) {
        DeleteFixup(x);
    }
}
```

Searching

Suppose we want to search for a node x with key ‘key’ in a tree with size n.

The while loops shows that the search will continue until a NIL leaf node is reached or until the node is found. Two cases can occur:

- Loop runs until leaf node NIL is reached, therefore traversing the height of the tree with **O(logn)** time.
- The node we want is found in best case **O(1)** or in worst case we have to go down the height of the tree in **O(logn)** time.

Therefore, searching in a red-black tree takes **Θ(logn)**.
C + logn = logn ∈ Θ(logn)

```
bool Tree::Search(Node* x, int key) {
    bool found = false;
    while (x != NIL && key != x->key) {
        if ( key < x->key) {
            x = x->left;
        }
        else {
            x = x->right;
        }
    }
    if (x != NIL) {
        found = true;
    }
    return found;
}
```

Compare with AVL Trees

AVL trees are self balancing BSTs with a similar runtime for search, delete and insert.

Insertion: The only difference is that when inserting, the balance factor is checked and rotations occur to keep the tree balanced. Each rotation takes O(1) time and rotations in worst case scenario may happen at each level in the tree, hence why it takes **Θ(logn)** to insert.

Deletion: After removing a node, the tree may become imbalanced which may take rotations to fix. Similar to insertion, in worst case scenario a rotation might occur in each level making deletion **Θ(logn)**.

Searching: Since AVL trees are balanced, in worst case searching for an element will take the height of the tree, hence it takes **Θ(logn)**.

Compare with Regular Binary Search Tree

A regular BST is not always balanced like AVL and red-black trees. So the height will differ. Height: **h ≤ n** because in worst case, all nodes are in a linked list.

Insertion: Because height in worst case is n, inserting worst case scenario is **O(n)** while if the tree was balanced, it would be O(logn).

Deletion: Similar to insertion, since in worst case the node we want to delete is at the end of the linked list of size n, worst case run time is **O(n)**, but if it was balanced, it would be O(logn).

Searching: The same reasoning for insertion and deletion applies, so in the worst case scenario, it would take **O(n)** time to find an element.

So in comparison, doing operations on a BST is slower than a red-black and AVL tree.

Specifics of Inserting, Deleting, and Searching in a Red-Black Tree

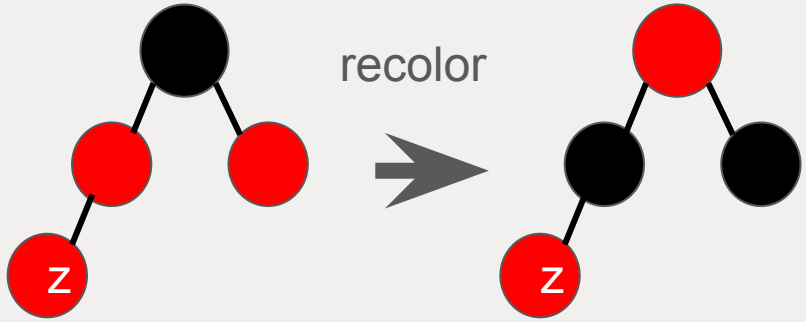
How do these algorithms work?

Inserting

When we insert a node into the tree, the new node should be **red**. Once we insert this new node, there are some cases we need to consider in order to “fix” the red-black tree after inserting. Let ‘z’ be the node we insert.

Case 1: Node z has a **red** parent and **red** uncle (parent’s sibling).

In this case, since z is red and has a red parent, this violates one of the properties. So we make both z.parent and uncle **black** and make the grandparent (the parent of z.parent and uncle) **red**.

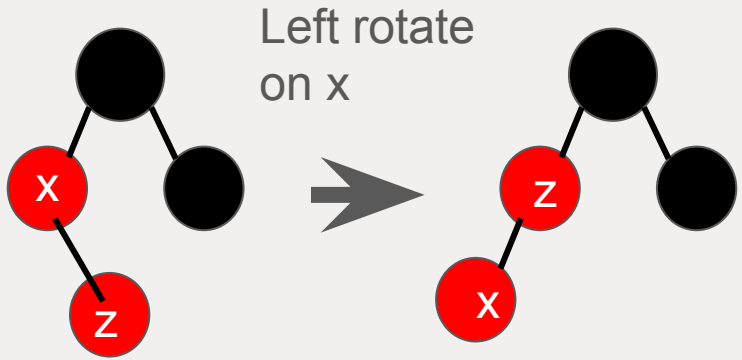


Inserting cont.

Case 2: The parent of z, x, is **red** and z's uncle is **black**. Also z is a right child and its parent is a left child.

The strategy is to replace x with z by calling a left rotate on x

Then to fix this state, we apply case 3…

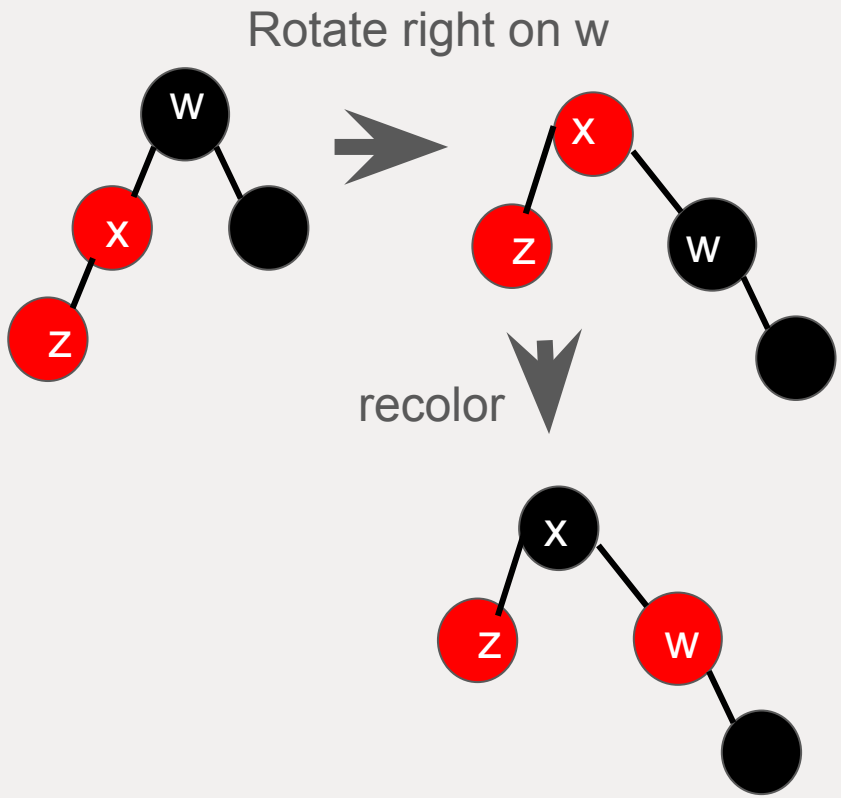


Inserting cont.

Case 3: The parent of z, x, is **red** an z's uncle is **black**. And if z is a left child and its parent is a left child.

We want to do a rotate right on the grandparent of z, w, then:

- Color the parent of z **black**
- Color the sibling of z **red**



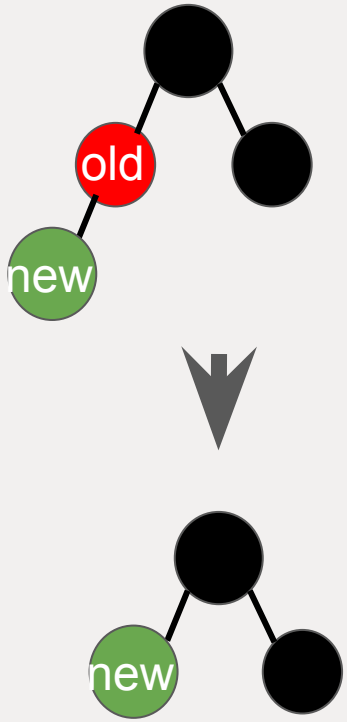
All of these cases are implemented by functions: RBInsertFixupA, RBInsertFixupB, and RBInsertFixupC.

Deleting

The delete function uses a helper function, shift/transplant, to delete nodes.

For example, we want to get rid of the node “old” so we replace “old” with “new”.

Let the node to be deleted be ‘z’.



Deleting cont.

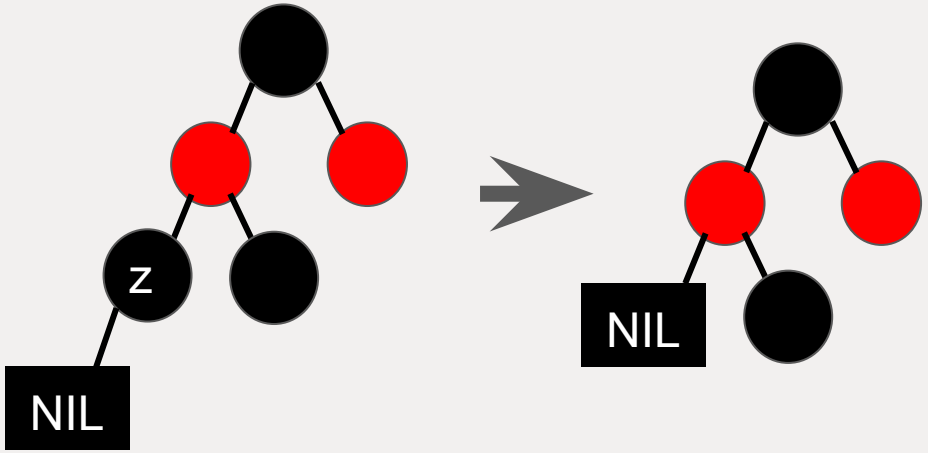
Case 1: If z's left child is NIL

We want to replace z with it's left child using shift/transplant

Case 2: If z's right child is NIL

We want to replace z with its right child using shift/transplant

We also want to keep track of the original color of the node to be deleted in these cases. Let v be this node that keeps track of the color.



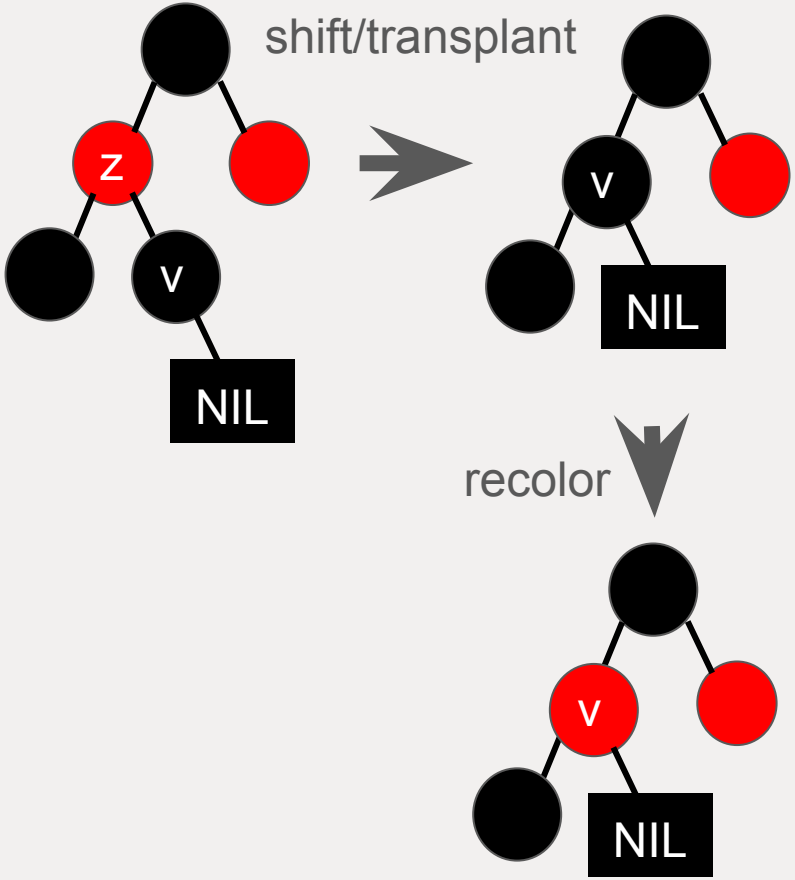
Similar for case 2 except on right side.

Deleting cont.

Case 3: z has 2 children

Let v be the minimum in the in the right subtree of z. We want v to be the successor of z. So use shift/transplant.

After these 3 cases, we must now fix any violations if any nodes, v, were **black**.



Fixing after Delete

Case 1: x’s sibling, w, is red

- Rotate left on parent of x and w, y.
- Make y red and w black
- Update w to be x’s sibling

Case 2: w (x’s sibling) is black and w’s children are black

- Make w red
- X becomes the parent

Fixing after Delete cont.

Case 3: w is black with **red left child** and **black right child**

- Rotate right on w
- Make w red
- Make w.parent black
- Update w

Case 4: w is black and w’s right child is red

- Rotate left on w
- Make w.right black
- Make w the color of w.left
- Make w.left black

Fixing after Delete cont.

Finally, update x to be the root.

The DeleteFixup function loops through and fixes any issues by going through the cases until they are resolved.

It can be concluded that DeleteFixup will take O(logn) because in worst case, it fixes the whole height of the tree. As concluded from before, since DeleteFixup is called in Delete on slide 6, it can be observed that Delete takes O(logn).

Small Note

The algorithms for inserting and deleting are also **symmetric** as well. They will work the opposite way. For example, this can be seen in the slides (first link in references) with RBInsertFixupB and RBInsertFixupC.

Searching

Searching is a lot simpler than inserting and deleting.

Generally, it loops until it reaches a leaf or it finds the specified node. It compares the key or value of the node we are looking for with the nodes around it. For example, if the key is less than a node being compared, say x for example, the algorithm will look in the left subtree of x and continue comparing with each node. It will work similarly if the key was greater than x.

Rationale and Design Choices

With my experience in C and C++ this semester, I decided to do this project with what I had learned in my classes this semester. I went with C++ on a Linux environment to create this project because C++ allows pointers, the use of classes and structs in a header file which allow me to easily define a red-black tree. I also decided to separate all of my functions into separate files to make the overall project cleaner and less messy in one file.

I also decided to have my main.cpp to hold the testing code for my implementation, which allowed me to test my code while implementing the functions. In addition, I made use of a Makefile in order to compile all of my files.

I did not have time to make a visualization for red-black tree so I went with prints in the terminal so I could ensure my implementation worked. After a while, reading it became much easier, but for more complex binary trees, the print statements would not be ideal.

References

https://osu.instructure.com/courses/182295/assignments/4652561

https://www.youtube.com/watch?v=t-oiZnplv7g

https://www.youtube.com/watch?v=lU99loSvD8s

https://www.youtube.com/watch?v=iw8N1_keEWA