6.2 Search and Rotation on Red-Black Trees

We will now implement the three routines Search, Insert and Delete from the DICTIONARY ADT using the Red-Black Tree data structure.

Since a Red-Black Tree is a BST, we can use the same Search routine as before to search the tree in worst case time $\Theta(\log n)$ (since now the height of the tree is $\Theta(\log n)$ in the worst case). Insert and Delete will also take time $\Theta(\log n)$ but if we use the same routine as before, they will cause violations of one of the three Red-Black properties.

For instance, if we use the regular BST Insert, then we'll add the new node at the bottom of the tree (so both its children are null). Then we have to decide whether to make it red or black. If we make it black, we'll certainly violate property 3 of Red-Black trees. If we make it red, we don’t have to worry about property 3, but we might violate property 2 (if its parent is red).

The following two procedures will be useful in building our Insert and Delete methods:

```plaintext
RotateLeft(Tree T, Node x)
    Node y = rightChild(x);
    rightChild(x) = leftChild(y);
    leftChild(y) = x;

RotateRight(Tree T, Node y)
    Node x = leftChild(y);
    leftChild(y) = rightChild(x);
    rightChild(x) = y;
```

These two methods perform what is referred to as a rotation on the tree $T$. The following is a graphical representation of these two methods. $x$ and $y$ are nodes and $A$, $B$, and $C$ are subtrees.
6.3 Insertion

We’ll use the following procedure to insert a node $x$ into a Red-Black tree:

$$\text{RedBlackInsert}(\text{Tree Root } R, \text{ Node } x)$$

1. Insert($R, x$);
2. $\text{color}(x) = \text{red}$;
3. If property 2 is violated then
   - Fix the tree

Property 2 can be violated only if $\text{parent}(x)$ is red. If $\text{parent}(x) = R$, the root of the tree, then we can just recolor $R$ to be black. This won’t violate property 3 for any node since there is nothing above $R$. If $\text{parent}(x) \neq R$, then we have three cases. We can assume $\text{parent}(\text{parent}(x))$ is colored black since otherwise we would be violating property 2 before we even inserted $x$.

We may need to apply the fixing operations multiple times before the tree is a proper Red-Black tree. Hence, even though $x$ starts off with both children null, we might have moved it upwards in the tree using previous fixing operations, so $x$ might in general have non-null children.

We will now consider the three cases. In each diagram, the objects shown are subtrees of the entire Red-Black tree. There may be nodes above it or nodes below it, except where otherwise specified. $A$ and $B$ are subtrees and $w, x, y$ and $z$ are single nodes. Squares represent black nodes and circles represent red nodes. In every case, we assume that the tree on the left does not violate property 3. Based on this assumption, you should check that the final subtree on the right also does not violate property 3.

Case 1 is the only case which leaves the tree on the right with a violation of Property 3. Cases 2 and 3 produce a proper Red-Black tree on the right without further iterations.

- **Case 1:** $x$’s “uncle” is red. (i.e. node $w$ is red)

![Diagram for Case 1]

The problem here is that $z$’s parent might be red, so we still have a violation of property 2. But notice that we have moved the conflict upwards in the tree. Either we can keep applying case 1 until we reach the root, or we can apply case 2 or case 3 and end the fix up process. If we reach the root by applying case 1 (in other words, $\text{parent}(z)$ is the root and $\text{parent}(z)$ is red, then we just change $\text{parent}(z)$ to black.

- **Case 2:** $x$’s uncle is not red (it’s black or does not exist) and $\text{key}(x) \leq \text{key}(y) \leq \text{key}(z)$ (or $\text{key}(x) \geq \text{key}(y) \geq \text{key}(z)$).
Now there are no violations of either property 2 or property 3, so we are finished.

- **Case 3:** $x$’s uncle is not red and $key(y) \leq key(x) \leq key(z)$ (or $key(y) \geq key(x) \geq key(z)$).

Now we can apply case 2 using $y$ as $x$ and we’re done.

### 6.3.1 Analysis

We know from our previous analysis that **Insert** takes worst case time $\Theta(\log n)$. Now consider the running time of fixing the tree. In the worst case, we might have to apply case 1 until we move the red-red conflict all the way up to the root starting from the bottom of the tree. This takes time $\Theta(\log n)$ since the height of the tree is $\Theta(\log n)$. Combined with the $\Theta(\log n)$ time to do the **Insert**, we find that **RedBlackInsert** takes time $\Theta(\log n)$ in the worst case.

### 6.4 Deletion

It remains to be seen how Red-Black trees can support deletion in time $\Theta(\log n)$, where $n$ is the number of nodes in the tree. Recall that **RedBlackInsert** was essentially the same as the **Insert** operation for binary search trees except that it was followed by a “fix-up” process whenever the new node (which we colored red) had a red parent.

Recall that $\text{Delete}(R, x)$ for BSTs deletes either node $x$ if $x$ has 0 or 1 children and $\text{succ}(x)$ if $x$ has 2 children. **RedBlackDelete** will be as follows:

**RedBlackDelete**($\text{Tree Root } R, \text{ Node } x$)

$\text{Delete}(R, x)$ but don’t actually delete $x$, instead let $y$ be the
If we perform `Delete(R,x)` on a Red-Black tree $R$, then we remove a node $y$ (which is either $x$ or $\text{succ}(x)$). If $y$ is red then we could not have possibly introduced any violations.

**Exercise.** Why could we have not introduced any violations when deleting a red node?

If $y$ happens to be colored black, the black-height balance of the tree will almost certainly be upset and property 3 of Red-Black trees will be violated. So again, we will need a “fix-up” process.

Recall also that `Delete` always removes a node that has at most one child (if $x$ has two children, then we remove $\text{succ}(x)$, which never has two children). Therefore, we have to worry about only those cases where $y$ is black and $y$ has at most one child.

- **Case A:** $y$ has one child: $y$’s child, call it $w$, must be red since otherwise property 3 would be violated in the subtree rooted at $y$. So we can just remove $y$ and make $w$ black to preserve the black-height balance for any node above $y$.

- **Case B:** $y$ has no children: We can’t apply the above trick if $y$ has no children. Recall that the `null` values at the bottom of the tree are considered black leaves. To preserve the black-height for $y$’s ancestors, we’ll remove $y$ and replace it with a `null` node that is not just black, but “double-black” (denoted by a double circle). But while this upholds property 3 of Red-Black trees, it violates property 1 (that every node must be either red or black).

Now, we consider the problem of removing a double-black node from an arbitrary position in the tree. There are five cases for this; in all, $r$ (which might be `null`) will be the double-black node that we want to remove. You should check that the transformations preserve property 3 and do not introduce any property 2 violations.
• **Case 1: r’s sibling is red:** In this case, we modify the tree so that r’s neighbor is black and then apply one of the other cases. This is so that in general we can rely on the fact that r’s neighbor will be black:

![Diagram of Case 1](image1.png)

Notice that this transformation has moved the double-black node downwards in the tree.

• **Case 2: r’s parent, sibling and nephews are all black:**

![Diagram of Case 2](image2.png)

Notice that this transformation has moved the double-black node upwards in the tree. If this keeps happening, then eventually the root will become the double-black node and we can just change it to black without violating any properties. Otherwise we will be able to apply one of the other cases.

Exercise. Is it possible that Case 1 and Case 2 can conflict with each other by moving the double-black node downwards and then upwards in an infinite loop?

• **Case 3: r’s sibling and nephews are black, r’s parent is red:**

![Diagram of Case 3](image3.png)

We can stop here because we have eliminated the double-black.
- **Case 4: r’s far nephew is red**: r’s parent s can start off as either color here (we’ll denote this by a rectangular node). After the transformation, t takes whatever color s had before.

![Diagram of Case 4](image)

Again, since there are no double-black nodes on the right, we can stop.

- **Case 5: r’s far nephew is black, r’s near nephew is red**: Here we’re going to perform a transformation so that r’s far nephew becomes red. Then we can apply Case 4.

![Diagram of Case 5](image)

### 6.4.1 Analysis

Again, we know that Delete takes \(\Theta(\log n)\) time in the worst case. Now, consider the “fix-up” process.

Case A can be performed in constant time. Case B further breaks down into five cases. Case 1 moves the double black node down one position in the tree. Case 2 moves the double black node up one position in the tree. The other three positions eliminate the double black node in constant time.

If Case 1 is required, then the parent of the double black square is red. Therefore, one of Cases 3-5 are applied and the entire operation is performed in constant time.

Finally, Case 2 moves the double black node up one position in the tree and this will need to be performed \(\Theta(\log n)\) times in the worst case. Therefore, the worst case running time of the “fix-up” operation is \(\Theta(\log n)\).

Therefore, RedBlackDelete has a worst case running time of \(\Theta(\log n)\).