Red Black Trees

- Each node has a "color" field in addition to a key, left, right, and parent pointer
- If the child or parent of a node does not exist, the corresponding pointer field will contain the value NIL
- We will say that these NIL’s are pointers to external nodes (leaves) of the tree, and say that all key-bearing nodes are internal nodes of the tree

Outline

- What is a RB-Tree

Red-Black Properties

A BST is a red-black tree if it satisfies the RB-Properties

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Example RB-Tree

- A RB-Tree is a balanced binary search tree
- The height of the tree is always $O(\log n)$ where $n$ is the number of nodes in the tree
**Black Height**

- **Black-height** of a node $x$, $bh(x)$ is the number of nodes on any path from, but not including $x$ down to a leaf node.
- Note that the black-height of a node is well-defined since all paths have the same number of black nodes.
- The black-height of an RB-Tree is just the black-height of the root.

**Key Lemma**

- **Lemma**: A RB-Tree with $n$ internal nodes has height at most $2 \log(n+1)$.
- **Proof Sketch**: 1) The subtree rooted at the node $x$ contains at least $2^{bh(x)} - 1$ internal nodes.
- 2) For the root $r$, $bh(r) \geq h/2$, thus $n \geq 2^{h/2} - 1$. Taking logs of both sides, we get that $h \leq 2 \log(n+1)$.

**Proof**

1) The subtree rooted at the node $x$ contains at least $2^{bh(x)} - 1$ internal nodes.

- Show by induction on the height of $x$.
- BC: If the height of $x$ is 0, then $x$ is a leaf, and subtree rooted at $x$ does indeed contain $2^0 - 1 = 0$ internal nodes.
- IS: Consider a node $x$ which is an internal node with two children (all internal nodes have two children). Each child has black-height of either $bh(x)$ or $bh(x) - 1$ (the former if it is red, the latter if it is black). Since the height of these children is less than $x$, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. This implies that the subtree rooted at $x$ has at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes. This proves the claim.

**Left-Rotate**

- Left-Rotate($x$) takes a node $x$ and "rotates" $x$ with its right child.
- Right-Rotate is the symmetric operation.
- Both Left-Rotate and Right-Rotate preserve the BST Property.
- We'll use Left-Rotate and Right-Rotate in the RB-Insert procedure.

**Picture**

- Diagram showing the Left-Rotate($x$) and Right-Rotate($y$) operations.

**Maintenance?**

- How do we ensure that the Red-Black Properties are maintained?
- I.e. when we insert a new node, what do we color it? How do we re-arrange the new tree so that the Red-Black Property holds?
- How about for deletions?
Example

Binary Search Tree Property

• Let $x$ be a node in a binary search tree. If $y$ is a node in the left subtree of $x$, then $\text{key}(y) \leq \text{key}(x)$. If $y$ is a node in the right subtree of $x$ then $\text{key}(y) \geq \text{key}(x)$

In-Class Exercise

Show that Left-Rotate($x$) maintains the BST Property. In other words, show that if the BST Property was true for the tree before the Left-Rotate($x$) operation, then it’s true for the tree after the operation.

• Show that after rotation, the BST property holds for the entire subtree rooted at $x$
• Show that after rotation, the BST property holds for the subtree rooted at $y$
• Now argue that after rotation, the BST property holds for the entire tree

RB-Insert($T, z$)

1. Set left($z$) and right($z$) to be NIL
2. Let $y$ be the last node processed during a search for $z$ in $T$
3. Insert $z$ as the appropriate child of $y$ (left child if $\text{key}(z) \leq \text{key}(y)$, right child otherwise)
4. Color $z$ red
5. Call the procedure RB-Insert-Fixup

RB-Insert-Fixup($T, z$)

```
RB-Insert-Fixup(T,z){
  while (color(p(z)) is red){
    case 1: z's uncle, y, is red{
      do case 1
    }
    case 2: z's uncle, y, is black and z is a right child{
      do case 2
    }
    case 3: z's uncle, y, is black and z is a left child{
      do case 3
    }
  }
  color(root(T)) = black;
}
```
Case 2 and 3

Loop Invariant

At the start of each iteration of the loop:

- Node \( z \) is red
- If parent(\( z \)) is the root, then parent(\( z \)) is black
- If there is a violation of the red-black properties, there is at most one violation, and it is either property 2 or 4. If there is a violation of property 2, it occurs because \( z \) is the root and is red. If there is a violation of property 4, it occurs because both \( z \) and parent(\( z \)) are red.

Pseudocode

- Detailed Pseudocode for RB-Insert and RB-Insert-Fixup is in the book, Chapter 13.3
- There's also a detailed proof of correctness for RB-Insert-Fixup in the same Chapter