Lecture 02/24/17

Lecturer: Xiaodi Wu

Reading: Chapter 13.3
Red-Black Trees: Insertion

Binary Tree Operations

- Tree-Search: $O(h) = O(\log(n))$.
- Tree-Insert, Tree-Delete: $O(h) = O(\log(n))$.
- Search: won’t lead to any violation; Insert, Delete: potentially lead to violations of red-black trees.

Tools to Deal with Violations

- Need to somehow change the tree structure.
- Fix violations without having new violations.
- Rotations: the first tool to make the chance, keeping the binary search tree property.
- Insertion: perform normal Tree-Insert (color red) and then RB-Insert-FixUP.
Red-Black Trees: Insertion

Binary Tree Operations

- Tree-Search: $O(h) = O(\log(n))$.
- Tree-Insert, Tree-Delete: $O(h) = O(\log(n))$.
- Search: won’t lead to any violation; Insert, Delete: potentially lead to violations of red-black trees.

Tools to Deal with Violations

- Need to somehow change the tree structure.
- Fix violations without having new violations.
- **Rotations**: the first tool to make the chance, keeping the **binary search tree** property.
- **Insertion**: perform normal Tree-Insert (color red) and then RB-Insert-FlxUP.
Red-Black Trees: Insertion

We can insert a node into an n-node red-black tree in $O(\lg n)$ time. To do so, we use a slightly modified version of the TREE-INSERT procedure (Section 12.3) to insert node $z$ into the tree $T$ as if it were an ordinary binary search tree, and then we color $z$ red. (Exercise 13.3-1 asks you to explain why we choose to make node $z$ red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT.$T; z/$ inserts node $z$, whose key is assumed to have already been filled in, into the red-black tree $T$.

```
RB-INSERT($T, z$)
1    $y = T.nil$
2    $x = T.root$
3    while $x \neq T.nil$
4        $y = x$
5        if $z.key < x.key$
6            $x = x.left$
7        else $x = x.right$
8    $z.p = y$
9    if $y == T.nil$
10        $T.root = z$
11    elseif $z.key < y.key$
12        $y.left = z$
13    else $y.right = z$
14    $z.left = T.nil$
15    $z.right = T.nil$
16    $z.color = RED$
17    RB-INSERT-FIXUP($T, z$)
```
What are possible violations?

Red-Black Tree Properties: NO violation of binary search tree properties

- **Color Property**: every node is either red or black.
- **Root Property**: the root is black.
- **External Property**: every external node (leaf, NIL) is black.
- **Internal Property**: the children of a red node are black.
- **Depth Property**: For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. (Equivalently, each node replaced by the root.)
What are possible violations?

Red-Black Tree Properties: NO violation of binary search tree properties

▶ **Color Property**: every node is either red or black.
▶ **Root Property**: the root is black.
▶ **External Property**: every external node (leaf, NIL) is black.
▶ **Internal Property**: the children of a red node are black.
▶ **Depth Property**: For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. (Equivalently, each node replaced by the root.)

**Violations**: Root Property (**easy fix?**) and Internal Property after inserting a red node.
Internal Property Violation

Internal Property: the children of a red node are black

- Let the current node be $z$ which is red.
- Violation implies the parent of $z$ is also red. Note that is the only violation.
Internal Property Violation

Internal Property: the children of a red node are black

- Let the current node be $z$ which is red.
- Violation implies the parent of $z$ is also red. Note that is the only violation.
- Case 1: $z$’s uncle is also red. $O(\log(n))$ fix.
Internal Property Violation

Internal Property: the children of a red node are black

- Let the current node be $z$ which is red.
- Violation implies the parent of $z$ is also red. Note that is the only violation.
- Case 1: $z$’s uncle is also red. $O(\log(n))$ fix.
- Case 2, 3: $z$’s uncle is black. $O(1)$ fix.
Red-Black Trees: Insertion-FIX-Case-1

Figure 13.5  Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since \( z \) and its parent \( y \) are both red. We take the same action whether \( z \) is a right child or \( y \) is a left child. Each of the subtrees \( \alpha, \beta, \gamma, \delta, \varepsilon \), and \( \text{new } z \) has a black root, and each has the same black-height.

The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The \( while \) loop continues with node \( y \)'s grandparent \( p \) as the new root. Any violation of property 4 can only occur between the new root, which is red, and its parent, if it is red as well.

If node \( y \) is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since \( y \) is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to \( y \).

If node \( y \) is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made \( y \) red and left \( p \) alone. If \( p \) was black, there is no violation of property 4. If \( p \) was red, coloring \( y \) red created one violation of property 4 between \( y \) and \( y \).

Case 2: \( y \)'s uncle \( z \) is black and \( z \) is a right child.

Case 3: \( y \)'s uncle \( z \) is black and \( z \) is a left child.

In cases 2 and 3, the color of \( z \)'s uncle \( z \) is black. We distinguish the two cases according to whether \( z \) is a right or left child of \( p \). Lines 10–11 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node \( y \) is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 12–14), in which node \( y \) is a left child. Because \(... use a left rotation to transform the situation into case 3 (lines 12–14), in which node \( y \) is a left child. Because
Red-Black Trees: Insertion-FIX-Case-2-3

Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP.

Since a case 1, property 4 is violated in either case 2 or case 3 because $\alpha$ and its parent $\beta$ are both red. Each of the subtrees $\gamma$, $\delta$, and $\epsilon$ has a black root ($\gamma$, $\delta$, and $\epsilon$ from property 4, and $\epsilon$ because otherwise we would be in case 1), and each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The while loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

Both $\alpha$ and $\beta$ are red, the rotation affects neither the black-height of nodes $\gamma$ nor property 5. Whether we enter case 3 directly or through case 2, $\gamma$'s uncle $\delta$ is black, since otherwise we would have executed case 1. Additionally, the node $\alpha$ exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving $\alpha$ up one level in line 10 and then down one level in line 11, the identity of $\alpha$ remains unchanged. In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done. The while loop does not iterate another time, since $\alpha$ is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $\alpha$ will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes $\alpha$ point to $\delta$, which is red. No change to $\alpha$ or its color occurs in cases 2 and 3.
b. Case 3 makes $\alpha$ black, so that if $\alpha$ is the root at the start of the next iteration, it is black.
c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3. Since node $\alpha$ is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.
To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps. First, we shall determine what violations of the red-black properties are introduced in RB-INSERT-FIXUP when node \( z \) is inserted and colored red. Second, we shall examine the overall goal of the while loop in lines 1–15. Finally, we shall explore each of the three cases within the while loop's body and see how they accomplish the goal. Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree.

Which of the red-black properties might be violated upon the call to RB-INSERT-FIXUP?

Property 1 certainly continues to hold, as does property 3, since both children of the newly inserted red node are the sentinel \( T : nil \).

Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node \( z \) replaces the (black) sentinel, and node \( z \) is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations are due to \( z \) being colored red. Property 2 is violated if \( z \) is the root, and property 4 is violated if \( z \)'s parent is red. Figure 13.4(a) shows a violation of property 4 after the node \( z \) has been inserted.

Case 2 falls through into case 3, and so these two cases are not mutually exclusive.
Red-Black Trees: Insertion-FIX-Example

Figure 13.4: The operation of RB-INSERT-FIXUP.

(a) An old tree after insertion. Because both $z$ and its parent $p$ are red, a violation of property 4 occurs. Since $z$'s uncle $y$ is red, case 1 in the code applies. We recolor nodes and move the pointer $z$ up the tree, resulting in the tree shown in (b).

Once again, $z$ and its parent are both red, but $z$'s uncle $y$ is black. Since $z$ is the right child of $z$: $p$, case 2 applies. We perform a left rotation, and the tree that results is shown in (c).

Now, $z$ is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in (d), which is a legal red-black tree.
13.3 Insertion

Figure 13.4 The operation of RB-INSERT-FIXUP.

(a) An old tree after insertion. Because both $z$ and its parent $y$ are red, a violation of property 4 occurs. Since $z$'s uncle $y$ is red, case 1 in the code applies. We recolor nodes and move the pointer $z$ up the tree, resulting in the tree shown in (b).

Once again, $z$ and its parent are both red, but $z$'s uncle $y$ is black. Since $z$ is the right child of $y$, case 2 applies. We perform a left rotation, and the tree that results is shown in (c). Now, $z$ is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in (d), which is a legal red-black tree.
Red-Black Trees: Insertion-FIX-Example

13.3 Insertion

Figure 13.4
The operation of RB-INSERTFIXUP.

(a) An node after insertion. Because both and its parent are red, a violation of property 4 occurs. Since 's uncle is red, case 1 in the code applies. We recolor nodes and move the pointer up the tree, resulting in the tree shown in (b).

Once again, and its parent are both red, but 's uncle is black. Since is the right child of , case 2 applies. We perform a left rotation, and the tree that results is shown in (c). Now, is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in (d), which is a legal red-black tree.
Red-Black Trees: Insertion-FIX-Example

Figure 13.4 The operation of RB-INSERT-FIX-UP.

(a) An old tree after insertion. Because both $z$ and its parent $p$ are red, a violation of property 4 occurs. Since $z$'s uncle $y$ is red, case 1 in the code applies. We recolor nodes and move the pointer $z$ up the tree, resulting in the tree shown in (b).

Once again, $z$ and its parent are both red, but $z$'s uncle $y$ is black. Since $z$ is the right child of $p$, case 2 applies. We perform a left rotation, and the tree that results is shown in (c). Now, $z$ is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in (d), which is a legal red-black tree.
The while loop in lines 1–15 maintains the following three-part invariant at the start of each iteration of the loop:

a. Node $z$ is red.

b. If $z.p$ is the root, then $z.p$ is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because $z$ is the root and is red. If the tree violates property 4, it is because both $z$ and $z.p$ are red.
Red-Black Trees: Insertion-FIX-Case-1

Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP.

Property 4 is violated, since and its parent are both red. We take the same action whether (a) is a right child or (b) is a left child. Each of the subtrees , , , and has a black root, and each has the same black-height.

The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The loop continues with node 's grandparent as the new . Any violation of property 4 can only occur between the new , which is red, and its parent, if it is red as well.

If node is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to . If node is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made red and left alone. If was black, there is no violation of property 4. If was red, coloring red created one violation of property 4 between and .

Case 2: 's uncle is black and is a right child.

Case 3: 's uncle is black and is a left child.

In cases 2 and 3, the color of 's uncle is black. We distinguish the two cases according to whether is a right or left child of . Lines 10–11 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 12–14), in which node is a left child. Because...
13.3 Insertion

Figure 13.6 Cases 2 and 3 of the procedure RB-INSERTFIXUP.

A since a case1, property4 is violated in either case 2 or case 3 because

and its parent

are both red. Each of the subtrees

\( \alpha \), \( \beta \), \( \gamma \), and

has a black root (\( \alpha \), \( \beta \), \( \gamma \), \( \delta \)), and because otherwise we would be in case 1), and
each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves

while loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

both

and

are red, the rotation affects neither the black-height of nodes

nor property 5. Whether we enter case 3 directly or through case 2,

’s uncle

is black, since otherwise we would have executed case 1. Additionally, the

node

exists, since we have argued that this node existed at the time that

lines 2 and 3 were executed, and after moving

up one level in line 10 and then
down one level in line 11, the identity of

remains unchanged. In case 3,
we execute some color changes and a right rotation, which preserve property 5,
and then, since we no longer have two red nodes in a row, we are done. The

while loop does not iterate another time, since

is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just
argued,
will be black upon the next test in line 1, and the loop body will not
execute again.)

a. Case 2 makes
point to
, which is red. No change to

occurs in cases 2 and 3.

b. Case 3 makes
black, so that if
is the root at the start of the next
iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

Since node
is not the root in cases 2 and 3, we know that there is no viola-
tion of property 2. Cases 2 and 3 do not introduce a violation of property 2,
since the only node that is made red becomes a child of a black node by the
rotation in case 3.

Cases 2 and 3 correct the lone violation of property 4, and they do not intro-
duce another violation.