COMP251: Red-black trees

Jérôme Waldispühl & Roman Sarrazin-Gendron
School of Computer Science
McGill University

Based on (Cormen et al., 2002)

Based on slides from D. Plaisted (UNC)
Admin

• Roman’s Office Hours will resume when he will teach again
• My Office Hours will appear in the calendar
• Reduction of TA’s Office Hours
• Release of practice problems with solutions
• Check the schedule (it will be updated this week)
Recap: Balanced Binary Search Trees

- $T$ is a rooted binary tree
- Key of a node $x \geq$ keys in its left subtree.
- Key of a node $x \leq$ keys in its right subtree.
- Use to store keys
- The running time of search/Insert/Delete operations depends on the height of the subtrees
- $\Rightarrow$ Keep the height of subtrees as minimal as possible
Recap: Rotations

Rotations change the tree structure & **preserve the BST property**.

**Proof:** elements in B are ≥ x and ≤ y...

In both cases, everything in A < x < everything in B < y < everything in C
Recap: AVL trees

**Definition:** BST such that the heights of the two child subtrees of any node differ by at most one.

\[ |h_{\text{left}} - h_{\text{right}}| \leq 1 \]

- AVL trees are self-balanced binary search trees.
- Insert, Delete & Search take $O(\log n)$ in average and worst cases.
- To satisfy the definition, the height of an empty subtree is -1
Red-black trees: Overview

• Red-black trees are a variation of binary search trees to ensure that the tree is **balanced**.
  – Height is $O(lg \ n)$, where $n$ is the number of nodes.
• Operations take $O(lg \ n)$ time in the worst case.
• Invented by R. Bayer (1972).
Red-black Tree

• Binary search tree + 1 bit per node: the attribute color, which is either red or black.

• All other attributes of BSTs are inherited:
  – key, left, right, and parent.

• All empty trees (leaves) are colored black.
  – Note: We can use a single sentinel, nil, for all the leaves of red-black tree $T$, with $\text{color}[\text{nil}] = \text{black}$. The root’s parent is also $\text{nil}[T]$. 
Red-black (RB) Properties

1. Every node is either red or black.
2. The root is black.
3. All leaves (nil) are black.
4. If a node is red, then its children are black (i.e. no 2 consecutive red nodes).
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes (i.e. same black height).
Red-black Tree – Example

Note: every internal node has two children, even though \textit{nil leaves are not usually shown}. 
Height of a Red-black Tree

• Height of a node:
  – \( h(x) = \) number of edges in the longest path to a leaf.

• Black-height of a node \( x \), \( bh(x) \):
  – \( bh(x) = \) number of black nodes (including \( nil[T] \)) on the path from \( x \) to leaf, not counting \( x \).

• Black-height of a red-black tree is the black-height of its root.
  – By RB Property 5, black height is well defined.
• Height \( h(x) \):
  #edges in a longest path to a leaf.

• Black-height \( bh(x) \):
  # black nodes on path from \( x \) to leaf, not counting \( x \).

• Property: \( bh(x) \leq h(x) \leq 2 \cdot bh(x) \)
Bound on RB Tree Height

Lemma 1: Any node $x$ with height $h(x)$ has a black-height $bh(x) \geq h(x)/2$.

Proof: By RB property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. ■
Bound on RB Tree Height

Lemma 2: The subtree rooted at any node $x$ contains $\geq 2^{bh(x)} - 1$ internal nodes.

Proof: By induction on height of $x$.

- **Base Case:** Height $h(x) = 0 \implies x$ is a leaf $\implies bh(x) = 0$. Subtree has $\geq 2^0 - 1 = 0$ nodes.

- **Induction Step:**
  - Each child of $x$ has height $h(x) - 1$ and black-height either $bh(x)$ (child is red) or $bh(x) - 1$ (child is black).
  - By ind. hyp., each child has $\geq 2^{bh(x) - 1} - 1$ internal nodes.
  - Subtree rooted at $x$ has $\geq 2 \cdot (2^{bh(x) - 1} - 1) + 1$
    - $= 2^{bh(x)} - 1$ internal nodes. ■
Bound on RB Tree Height

**Lemma 1:** Any node \( x \) with height \( h(x) \) has a black-height \( bh(x) \geq h(x)/2 \).

**Lemma 2:** The subtree rooted at any node \( x \) has \( \geq 2^{bh(x)} – 1 \) internal nodes.

**Lemma 3:** A red-black tree with \( n \) internal nodes has height at most \( 2 \lg(n+1) \).

**Proof:**
- By lemma 2, \( n \geq 2^{bh} – 1 \),
- By lemma 1, \( bh \geq h/2 \), thus \( n \geq 2^{h/2} – 1 \).
- \( \Rightarrow h \leq 2 \lg(n + 1) \).

Thus, a RB tree is balanced!
Insertion in RB Trees

• Insertion must preserve all red-black properties.
• Should an inserted node be colored Red? Black?
• Basic steps:

1. Use BST Tree-Insert to insert a node $x$ into $T$.
   • Procedure $\text{RB-Insert} (x)$.
2. Color the node $x$ in red.
3. Use (1) node re-coloring, and (2) rotations to restore RB tree property.
   • Procedure $\text{RB-Insert-Fixup}$.  

After step 2, some RB properties may be violated. Which ones?
**Insertion**

**Insertion**

**RB-Insert**(*T*, *z*)

1. \(y \leftarrow \text{nil}[T]\)
2. \(x \leftarrow \text{root}[T]\)
3. while \(x \neq \text{nil}[T]\)
4. do \(y \leftarrow x\)
5. if \(\text{key}[z] < \text{key}[x]\)
6. then \(x \leftarrow \text{left}[x]\)
7. else \(x \leftarrow \text{right}[x]\)
8. \(p[z] \leftarrow y\)
9. if \(y = \text{nil}[T]\)
10. then \(\text{root}[T] \leftarrow z\)
11. else if \(\text{key}[z] < \text{key}[y]\)
12. then \(\text{left}[y] \leftarrow z\)
13. else \(\text{right}[y] \leftarrow z\)

**RB-Insert**(*T*, *z*) Contd.

14. \(\text{left}[z] \leftarrow \text{nil}[T]\)
15. \(\text{right}[z] \leftarrow \text{nil}[T]\)
16. \(\text{color}[z] \leftarrow \text{RED}\)
17. **RB-Insert-Fixup**(*T*, *z*)

**Principles:**
- Regular BST insert (line 1-15)
- Color assignment (line 16)
- Fixup (line 17).

What is happening in the fixup is obviously the more sophisticated procedure.
Insert RB Tree – Example
In fact, the BST insertion inserts as internal node. We must always keep all leaves as Nil (no key).
Insert RB Tree – Example

Recolor 10, 8 & 11
Insert RB Tree – Example

Rotations will not fix the conflict but allow to adjust the structure of the RB tree to fix the conflict after.

Right rotate at 18
Parent & child with conflict are now aligned with the root.
Insert RB Tree – Example

Left rotate at 7
The conflict moved one level up!
Insert RB Tree – Example

Recolor 10 & 7 (root must be black!)

The conflict is now with the root, only re-coloring will help.
Case 1 – uncle $y$ is red

- $p[p[z]]$ ($z$’s grandparent) must be black, since $z$ and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and $y$ black $\Rightarrow$ now $z$ and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red $\Rightarrow$ restores property 5.
- The next iteration has $p[p[z]]$ as the new $z$ (i.e., $z$ moves up 2 levels).

$z$ is a right child here. Similar steps if $z$ is a left child.
Case 2 – $y$ is black, $z$ is a right child

- Left rotate around $p[z]$, $p[z]$ and $z$ switch roles $\Rightarrow$ now $z$ is a left child, and both $z$ and $p[z]$ are red.
- Takes us immediately to case 3.
Case 3 – $y$ is black, $z$ is a left child

- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate right on $p[p[z]]$ (in order to maintain property 4).
- No longer have 2 reds in a row.
- $p[z]$ is now black $\implies$ no more iterations.

The rotation enables us to restore the black-height property lost with re-coloring.
RB-Insert-Fixup $(T, z)$
1. while $color[p[z]] = \text{RED}$
2. do if $p[z] = left[p[p[z]]]$
3. then $y \leftarrow right[p[p[z]]]$
4. if $color[y] = \text{RED}$
5. then $color[p[z]] \leftarrow \text{BLACK} \quad // \text{Case 1}$
6. $color[y] \leftarrow \text{BLACK} \quad // \text{Case 1}$
7. $color[p[p[z]]] \leftarrow \text{RED} \quad // \text{Case 1}$
8. $z \leftarrow p[p[z]] \quad // \text{Case 1}$
Insertion – Fixup

RB-Insert-Fixup(T, z) (Contd.)

9.   else if \( z = right[p[z]] \)  // color[y] \( \neq \) RED
10.   then \( z \leftarrow p[z] \)  // Case 2
11.   LEFT-ROTATE(T, z)  // Case 2
12.   color[p[z]] \leftarrow \) BLACK  // Case 3
13.   color[p[p[z]]] \leftarrow \) RED  // Case 3
14.   RIGHT-ROTATE(T, p[p[z]])  // Case 3
15.   else (if \( p[z] = right[p[p[z]]] \))(same as 10-14
16. with “right” and “left” exchanged)
17.   color[root[T ]] \leftarrow \) BLACK
Algorithm Analysis

• $O(\lg n)$ time to get through RB-Insert up to the call of RB-Insert-Fixup.

• Within RB-Insert-Fixup:
  – Each iteration takes $O(1)$ time.
  – Each iteration but the last moves $z$ up 2 levels.
  – $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
  – Thus, insertion in a red-black tree takes $O(\lg n)$ time.
  – Note: there are at most 2 rotations overall.
Correctness

Loop invariant:

• At the start of each iteration of the while loop,
  – z is red.
  – There is at most one red-black violation:
    • Property 2: z is a red root, or
    • Property 4: z and p[z] are both red.
Correctness – Contd.

• **Initialization:** ✓

• **Termination:** The loop terminates only if \( p[z] \) is black. Hence, property 4 is OK.

  The last line ensures property 2 always holds.

• **Maintenance:** We drop out when \( z \) is the root (since then \( p[z] \) is sentinel \( nil[T] \), which is black). When we start the loop body, the only violation is of property 4.

  – There are 6 cases, 3 of which are symmetric to the other 3. We consider cases in which \( p[z] \) is a left child.

  – See cases 1, 2, and 3 described above.
AVL vs. Red-Black Trees

- AVL trees are more strictly balanced ⇒ faster search
- Red Black Trees have less constraints and insert/remove operations require less rotations ⇒ faster insertion and removal
- AVL trees store balance factors or heights with each node
- Red Black Tree requires only 1 bit of information per node
Further Readings


See Chapter 13 for the complete proofs & deletion