COMP251: Red-black trees

Jérôme Waldispühl
School of Computer Science
McGill University

Based on (Cormen et al., 2002)

Based on slides from D. Plaisted (UNC)
Definition: An AVL tree is a BST such that the heights of the two child subtrees of any node differ by at most one.

- AVL trees are self-balanced binary search trees.
- Insert, Delete & Search take \( O(\log n) \) in average and worst cases.
Rotations preserve the BST property.

**Proof:** elements in B are \( \geq x \) and \( \leq y \)...
Recap lecture 3

1. Diagram 1
2. Diagram 2
3. Diagram 3
4. Diagram 4
Insert in AVL trees

Right rotation at 27
Insert in AVL trees

Left rotation at 43

RotateLeft(T, 43)
Insert in AVL trees

Right rotation at 57

RotateRight(T,57)
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.
  – It is possible can use a single sentinel, nil, for all the leaves of red-black tree $T$, with $color[nil] = \text{black}$. The root’s parent is also $nil[T]$. 
Red-black 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 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).
Note: every internal node has two children, even though nil leaves are not usually shown.
Height of a Red-black Tree

- **Height of a node:**
  - \( h(x) = \) number of edges in a 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 Property 5, black height is well defined.
Height of a Red-black Tree

- 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: $bh(x) \leq h(x) \leq 2 \times bh(x)$
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 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 $h(x) = 0 \Rightarrow x$ is a leaf $\Rightarrow bh(x) = 0$. Subtree has $2^0 - 1 = 0$ nodes.

- **Induction Step:**
  - Each child of $x$ has height $h(x) - 1$ and black-height either $b(x)$ (child is red) or $b(x) - 1$ (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. (The +1 is for $x$ itself) ■
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 $n \geq 2^{h/2} - 1$.
- $\Rightarrow h \leq 2 \lg(n + 1)$. 
Insertion in RB Trees

• Insertion must preserve all red-black properties.
• Should an inserted node be colored Red? Black?
• Basic steps:
  – Use BST Tree-Insert to insert a node $x$ into $T$.
    • Procedure $\text{RB-Insert}(x)$.
  – Color the node $x$ red.
  – Fix the new tree by (1) re-coloring nodes, and (2) performing rotation to preserve RB tree property.
    • Procedure $\text{RB-Insert-Fixup}$.
**Insertion**

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

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

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

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

Regular BST insert + color assignment + fixup.
Insert RB Tree – Example
Insert RB Tree – Example

Insert(T,15)
Insert RB Tree – Example

Recolor 10, 8 & 11
Insert RB Tree – Example

Right rotate at 18
Insert RB Tree – Example

Right rotate at 18 (parent & child with conflict are aligned)
Insert RB Tree – Example

Left rotate at 7
Insert RB Tree – Example

Left rotate at 7
Insert RB Tree – Example

Recolor 10 & 7 (root must be black!)
Insertion – Fixup

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

1. **while** \(color[p[z]] = \text{RED}\)
2. **do** **if** \(p[z] = \text{left}[p[p[z]]]\)
3. **then** \(y \leftarrow \text{right}[p[p[z]]]\)
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.  \textbf{else if} $z = \text{right}[p[z]]$  // color[y] $\neq$ RED
10. \textbf{then} \hspace{1em} $z \leftarrow p[z]$  \hspace{1em} // Case 2
11. \hspace{1em} $\text{LEFT-ROTATE}(T, z)$  \hspace{1em} // Case 2
12. $\text{color}[p[z]] \leftarrow \text{BLACK}$  \hspace{1em} // Case 3
13. $\text{color}[p[p[z]]] \leftarrow \text{RED}$  \hspace{1em} // Case 3
14. $\text{RIGHT-ROTATE}(T, p[p[z]])$  \hspace{1em} // Case 3
15. \textbf{else} \hspace{1em} (if $p[z] = \text{right}[p[p[z]]]$)(same as 10-14
16. \hspace{1em} with “right” and “left” exchanged)
17. $\text{color}[\text{root}[T]] \leftarrow \text{BLACK}$
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).
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 on $p[p[z]]$. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.
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.
  
  – If \( p[z] \) is the root, then \( p[z] \) is black.
  
  – 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.
  - Let $y$ be $z$’s uncle ($p[z]$’s sibling).