Lecture 11: Red-Black Trees

Red-Black Tree

Designed to represent 2-3-4 tree without the additional link overhead

A Red-Black tree is a binary search tree in which each node is colored red or black

Red nodes represent the extra keys in 3-nodes and 4-nodes

- 2-node = black node

Red-black trees are not unique, but the corresponding 2-3-4 tree is unique
**Red-Black Tree**

Red-black trees are widely used:
- C++ STL: `map`, `set`, `multimap`, `multiset`
- Java: `java.util.TreeMap`, `java.util.TreeSet`
- Linux kernel: `linux/rbtree.h`

RBT node representation:

```
+------+
|      |
| LEFT |
|      |
+------+
|      |
| PARENT | RIGHT |
+--------+
| KEY    |
+--------+
| COLOR  |
+--------+
```

- The root is **black**
- The children of red nodes are **both black**

**Red-Black Tree Rules and Properties**

- Every node is colored either **red** or **black**
- External nodes are **black**
- If a node is **red**, its children must be **black**

**A Red-Black Tree**

Every node is colored either red or black

The root is black

External nodes are black

If a node is red, its children must be black

**Black-height `bh(x)`**

Black-height of node `x` is the number of black nodes on the path from `x` to an external node (including the external node but not counting `x` itself)

Every node has a black-height, `bh(x)`

For all external nodes, `bh(x) = 0`

For root `x`, `bh(x) = bh(T)`
Every path from a node \( x \) to an external node must contain the same number of black nodes = \( \text{black-height}(x) \)

Red-Black Rules and Properties

1. Every node is either red or black
2. The root is black [root rule]
3. External nodes (nulls) are black
4. If a node is red, then both its children are black [red rule]
5. Every path from a node to a null must have the same number of black nodes (black height) [black-height rule]
   a. this is equivalent to a 2-3-4 tree being a perfect tree: all the leaf nodes of the 2-3-4 tree are at the same level (black-height=1)
   b. a black node corresponds to a level change in the corresponding 2-3-4 tree

Implications of the Rules

If a red node has any children, it must have two children and they must be black (why?)
- can’t have 2 consecutive reds (double red) on a path
- however, any number of black nodes may appear in a sequence

If a black node has only one child that child must be a red leaf (why?)

Red-Black Tree Height Bound

Red-black tree rules constrain the adjacency of node coloring, ensuring that no root-to-leaf path is more than twice as long as any other path, which limits how unbalanced a red-black tree may become

Theorem: The height of a red-black tree with \( n \) internal nodes is between \( \log_2(n+1) \) and \( 2\log_2(n+1) \)
Red-Black Tree Height Bound

Start with a red black tree with height $h$
(note: height here includes the external nodes)
Merge all red nodes into their black parents

Nodes in resulting tree have degrees between 2 and 4
All external nodes are at the same level

It’s the equivalent 2-3-4 tree to the red-black tree!
Height of the resulting tree is $h' = h/2$

Let $h' \geq h/2$ be the height of the collapsed tree

The tree is tallest if all internal nodes have degree 2, i.e., there were no red-node in the original red-black tree, $h' = h$, and number of internal nodes is $n = 2^{h'} - 1$ and $h' = 2 \log_2(n+1)$

The tree is shortest if all internal nodes have degree $> 2$, and $h' = h/2$; e.g., if all internal nodes have degree 4, the number of internal nodes is $n = 4^{h'} - 1$ and $h' = \log_2(n+1)$

In the mixed case, $\log_2(n+1) \leq h' \leq 2 \log_2(n+1)$

(Alternate Proof)

Prove: an $n$-internal node RB tree has height $h \leq 2 \log(n+1)$

Claim: A subtree rooted at a node $x$ contains at least $2^{bh(x)} - 1$ internal nodes
- proof by induction on height $h$
- base step: $x$ has height 0 (i.e., external node)
  - What is $bh(x)$?
  - $0$
  - So…subtree contains $2^{bh(x)} - 1$
    $= 2^0 - 1$
    $= 0$ internal nodes (claim is TRUE)
Red-Black Tree Height Bound
(Alternate Proof)

Inductive proof that subtree at node \( x \) contains at least \( 2^{bh(x)} - 1 \) internal nodes

- inductive step: \( x \) has positive height and 2 children
  - each child has black-height of \( bh(x) \) or \( bh(x) - 1 \) (Why?)
  - the height of a child = \( \text{height of } x - 1 \)
  - so the subtrees rooted at each child contain at least \( 2^{bh(x)} - 1 \) internal nodes by induction hypothesis
  - thus subtree at \( x \) contains
    \[
    (2^{bh(x)} - 1) + (2^{bh(x)} - 1) + 1 = 2 \times 2^{bh(x)} - 1 = 2^{bh(x)} - 1 \text{ nodes}
    \]

Thus at the root of the red-black tree:
\[
\begin{align*}
    n &\geq 2^{bh(root)} - 1 \\
    n &\geq 2^{bh(root)} - 1 \geq 2^{h/2} - 1 \\
    \log(n+1) &\geq h/2 \\
    h &\leq 2 \log(n+1)
\end{align*}
\]

Thus \( h = O(\log_2 n) \) By the black-height rule, the additional nodes in paths longer than the black height of the tree can consist only of red nodes

By the red rule, at least 1/2 of the nodes on any path from root to an external node are black

Since the longest path of the tree is \( h \), the black-height of the root must be at least \( h/2 \)
Inserting into a 3-node: Two Cases

1. Inserting node $b$ to a black parent that is part of a 3-node, creating a 4-node, done
   \[ \Rightarrow \text{inserting a new node to a black parent is always simple} \]

2. Inserting node $b$ to a red parent that is part of a 3-node, creating double red
   \[ \Rightarrow \text{how to recognize that parent and grandparent are part of a 3-node?} \]
   \[ \text{parent is red, grandparent and uncle ($w$) are black} \]
   \[ \Rightarrow \text{need to rotate to create a new 4-node} \]

3-Node, Red-Parent

1. Make the new node ($b$) along with parent ($c$) and grandparent ($a$) a 4-node
   \[ \text{Rotate to make parent ($c$) the middle value of the 4-node} \]

   There are four possible combinations of $a$, $b$, and $c$ corresponding to LL, RR, LR, RL rotations (see next slide)

   As the middle value of a 4-node, parent ($c$) will be black, and the two outer nodes ($a$) and ($b$) will be red

3-Node, Red-Parent Rotations

- Single rotation
- Double rotation

Inserting into a 3-Node

Insert 2

3-node

4-node
**Inserting into a 3-Node**

Insert 14

![Node Diagram]

1. **Red-Black Insert**
   - as with BST, insert new node as leaf, must be **red**
     - can’t be black or will violate black-height rule
     - therefore the new leaf must be **red**
   - insert new node, if inserting into a 2-node representation (black parent), done
   - if inserting into a 3-node, **could** result in double red
     - need to rotate and recolor nodes to represent a 4-node, with a black parent
   - if inserting into a 4-node, “split” 4-node → recolor children black, parent red, and “promote” parent
   - maintain root as black node

**Inserting into a 4-node**

Inserting node $d$ causes double red, and $d$’s parent has **red** sibling $w$
  → parent, aunt, and grandparent are part of a 4-node
  → need to recolor, to split the 4-node and “promote” grandparent
  parent and aunt become black
  grandfather becomes **red**

If grandparent is root, change it back to black
Otherwise, insert grandparent to great-grandparent, applying the same insertion rules as before depending on whether great-grandparent is a 2-node, 3-node, or 4-node
**Inserting into a 4-node**

Grandparent is root: recolor the two children black

Insert red grandparent into a 2-node great-grandparent:

Inserting into a 4-node

After inserting 55, promote red grandparent to a 3-node, black great-grandparent:

**Inserting into a 4-node**

Inserting into a 4-node

Promoting red grandparent to a 3-node, red-great grandparent:

Four cases:
- **RR**: requiring a single left rotation, e.g.,

**3-Node, Red-Great Grandparent**

- **LL**: requiring a single right rotation, e.g.,
3-NODE, RED-GREAT GRANDPARENT

- **LR**: requiring a double left-right rotation, e.g.:

- **RL**: requiring a double right-left rotation, e.g.,

---

**RBT Insertion Examples**

Insert 10 – root, must be black

Equivalent 2-3-4 Tree:

---

**RBT Insertion Examples**

Insert 85 (root is now a 3-node)

---

**RBT Insertion Examples**

Insert 15

double red!
RBT Insertion Examples

Equivalent 2-3-4 tree:

Rotate – Recolor (root becomes a 4-node)

10 15 85

Insert 70 (split the 4-node)

10 15 85 70

double red!

Recolor (root must be black)

10 15 85 70

double red!

Insert 20 (sibling of parent is black, a 3-node)

10 15 85 20

double red!

[Rosenfeld]
RBT Insertion Examples

Rotate (becomes a 4-node)

[Diagram]

Equivalent 2-3-4 tree:

[Diagram]

Insert 60 (sibling of parent is red, a 4-node, need to split)

[Diagram]

double red!

RBT Insertion Examples

Ricolor (promote middle value, 70)

[Diagram]

Equivalent 2-3-4 tree:

[Diagram]

Insert 30 (sibling of parent is black, a 3-node)

[Diagram]
RBT Insertion Examples

Equivalent 2-3-4 tree:

Rotate (made a 4-node)

Insert (sibling of parent?)

RBT Insertion Examples

Insert (promote middle value, 30, causing another double red; sibling of 30’s parent, 70, is black, → 70 is in a 3-node; with 30 it becomes a 4-node and needs to be rotated)
Red-Black Tree Removal

Observations:
- if we delete a red node, tree is still a red-black tree
- a red node is either a leaf node or must have two children

Rules:
1. if node to be deleted is a red leaf, remove leaf, done
2. if it is a single-child parent, it must be black (why?): replace with its child (must be red) and recolor child black
3. if it has two internal node children, swap node to be deleted with its in-order successor
   - if in-order successor is red (must be a leaf, why?), remove leaf, done
   - if in-order successor is a single child parent, apply second rule
In both cases the resulting tree is a legit red-black tree (we haven’t changed the number of black nodes in paths)
4. if in-order successor is a black leaf, or if the node to be deleted is itself a black leaf, things get complicated . . .
Black-Leaf Removal

We want to remove $v$, which is a black leaf
Replace $v$ with external node $u$, color $u$ **double black**

To eliminate **double black** edges, idea:
- find a red edge nearby, and change the pair (red, double black) into (black, black)
- as with insertion, we recolor and/or rotate
- rotation resolves the problem locally, whereas recoloring may propagate it two levels up
- slightly more complicated than insertion

Red Sibling

If sibling is red, rotate such that a black node becomes the new sibling, then treat it as a black-sibling case (next slides)

Black Sibling and Nephew/Niece

If sibling and its children are black, recolor sibling and parent
If parent becomes double black, percolate up

Black Sibling but Red Nephew

If sibling is black and one of its children is red, rotate and recolor red nephew involved in rotation
Red-Black Tree Removal Example

Remove 9:

sibling and its children are black, recolor sibling and parent

Red-Black Tree Removal Example

Remove 8:

not a black leaf, no double black

Red-Black Tree Removal Example

Remove 7:

sibling is black and one of its children is red, rotate and recolor red nephew involved in rotation

Efficiency of Red Black Trees

Insertions and removals require additional time due to requirements to recolor and rotate

Most insertions require on average a single rotation: still $O(\log_2 n)$ time but a bit slower empirically than in ordinary BST