Dynamic BALANCED SEARCH TREES (Non-Random)

Objectives: search, insert, delete in $O(\log n)$ time

- always maintain $\Theta(\log n)$ height & update in $O(\log n)$ time

Variations: AVL, 2-3, 2-3-4, B-trees, red-black, skip lists, treaps today
**RED-BLACK** trees

**Structure:**
1) nodes are colored red or black.
2) root is always black.
3) add black "dummy" leaves so every "real" node has 2 children.

**Important Rules:**
4) every red node has a black parent.
5) for any node \( x \): all paths down to leaves contain equal number of black nodes = black-height[\(x\)] not including \( x \)
4) every red node has a black parent.

5) for any node x: all paths down to leaves contain equal number of black nodes = black-height[x]

→ Fails rule 5
4) every red node has a black parent.

5) for any node \( x \): all paths down to leaves contain equal number of black nodes = \( \text{black-height}[x] \)

\[ \rightarrow \text{fails rule 5} \Rightarrow \text{fix by making some nodes red.} \]

black-height difference : 6 vs 3
No hope
to recolor
...
too unbalanced

Any root→leaf path of size $k$ must have $\geq \frac{k}{2}$ black nodes. So if any path is $>2$ times longer than another, we can’t make it RB.

CONTRACTION
Black nodes are fully balanced.
If root(T) has same black-height on all paths then height(T') is perfectly balanced

Q: what is max.degree of T'? (2-3-4)

#leaves in T : n+1 = size(T)+1
#leaves in T' : same
height(T') \leq \log(n+1) \quad [\text{higher degree} \Rightarrow \text{smaller height}; \text{worst-case: binary}]

Re-inserting red nodes: at most doubles height \rightarrow \text{height}(T) \leq 2 \log(n+1)
We have seen that RB trees are reasonably balanced: \( \sim 2 \log n \).

Search, min, max, next, prev: \( O(\log n) \) time.

Next: how to update RB trees (insert, delete).

**Rotations in arbitrary BSTs**

1. **Right-rotate** \((T_1, B)\):
   - \( T_1 \):
     - `A`
     - `B`
     - `X`
     - `Y`

   \[ X \leq A \leq Y \leq B \leq Z \]

2. **Left-rotate** \((T_2, A)\):
   - \( T_2 \):
     - `A`
     - `B`
     - `X`
     - `Y`

   \[ X \leq A \leq Y \leq B \leq Z \]

\( O(1) \) time.
**Insert in Red-Black Trees**

**Greedy (optimistic) start:**

1) Insert as in any BST
2) Color new node red (keeps black-height ok)
3) If parent is also red (violate parent rule 4)
   - Then color parent black and
   - Look for problems further up

--

Begins an error-correcting trail up to root, involving $O(i)$ recolorings and rotations per level

---

$O(\log n)$ time
example of $O(1)$-time error-corrections (per level)

1. Insert 15
2. Recolor parent of 15
3. Create black-height problem

Problem transferred from new node to its grandparent.
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations.

\[ \text{RB-insert}(x) \]
- initial step: regular insert & color \( x \) red.
- while \( x \neq \text{root} \) and \( p(x) \) is red
  \[
  \begin{cases}
  g(x) & \text{if } p(x) = \text{left}(g(x)) \\
  p(x) & \text{or } \begin{array}{c}
  \text{or } \\
  x
  \end{array}
  \end{cases}
  \]

why does \( g(x) \) exist?

\( p(x) = \text{red} \) so it isn't the root.
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations.

\[ \text{RB-insert}(x) \]
- Initial step: regular insert & color \( x \) red.
- While \( x \neq \text{root} \) and \( p(x) \) is red
  - If \( p(x) = \text{left}(g(x)) \)
    - \( y \leftarrow \text{right}(g(x)) \)

\[ \begin{cases} \text{if } p(x) = \text{left}(g(x)) \text{ \{ \text{don't care about colors} \}} \end{cases} \]

why can we assume \( y \) exists?

It is at least a dummy leaf.

in which case the subtree of \( g(x) \) is before inserting \( x \).
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations.

\textbf{RB-insert} \(x\):
- initial step: regular insert & color \(x\) red.
- while \(x \neq \text{root}\) and \(p(x)\) is red
  \begin{enumerate}
  \item if \(p(x) = \text{left}(g(x))\)
    \begin{enumerate}
    \item y ← \text{right}(g(x))
    \item if y is red then run CASE 1
    \item else if \(x = \text{right}(p(x))\) then run CASE 2.
    \end{enumerate}
  \end{enumerate}
  Run CASE 3
  \begin{enumerate}
  \item else \(p(x) = \text{right}(g(x))\)
    \begin{enumerate}
    \item do as before but with "left" & "right" switched
    \item \text{symmetric}
    \end{enumerate}
  \end{enumerate}
- color the root black
if $y$ is red then run **CASE 1**

Every $\blacktriangle$ contributes same to black-height.

Recolor $y, p(x), g(x)$

Let $x \leftarrow g(x)$

blackheight($p(x)$) preserved

$(repeat$ if new $p(x)$ is red)$)
else if $x = \text{right}(p(x))$ then run **CASE 2**. 

- **rotate-left**($p(x)$) 
- switch labels $x \leftrightarrow p(x)$ 

**conditions of case 3**

Run **CASE 3**

- **rotate-right**($g(x)$) 
- re-color $p, g$ 
- relabel $g$ 

$x$ moves up in tree

Now $p(x)$ is black.

Exit while
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations.

**RB-insert**($x$)
- initial step: regular insert & color $x$ red.
- while $x \neq$ root and $p(x)$ is red
  - if $p(x) = \text{left}(g(x))$
    - $y \leftarrow \text{right}(g(x))$
    - if $y$ is red then run **CASE 1**
    - else if $x = \text{right}(p(x))$ then run **CASE 2**
      - Run **CASE 3**
  - else if $p(x) = \text{right}(g(x))$
    - do as before but with "left" & "right" switched

color the root black