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
**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 = \( \text{black-height}[x] \)

\[ \rightarrow \text{Fails rule 5} \]
4) every red node has a black parent.

5) for any node \( x \): all paths down to leaves contain equal number of black nodes = 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 $\times 2$ 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 : \( n+1 = \text{size}(T) + 1 \)

#leaves in T' : same

height(T') \( \leq \log(n+1) \)  [higher degree \( \Rightarrow \) smaller height ; worst-case : binary]

Re-inserting red nodes : at most doubles height \( \rightarrow \) 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

$T_1$

- $A < B$
- $x \leq A \leq y \leq B \leq z$

$T_2$

- $A < B$
- $x \leq A \leq y \leq B \leq z$

$\text{right-rotate}(T_1, B) \quad \text{left-rotate}(T_2, A)$

$O(1)$ 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)

- Insert 15
- Recolor parent of 15 and create black-height problem
- Black height problem fixed, new consecutive-red problem

Problem transferred from new node to its grandparent
Insert 15

rotate-right(18)

rotate-left(7)

Done!
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
  
  \[ \begin{cases} 
  g(x) & \text{if } p(x) = \text{left}(g(x)) \\
  p(x) & \text{if } p(x) = x
  \end{cases} \]

\[ \text{don't care about colors} \]

\[ \text{why does } g(x) \text{ exist?} \]

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

\textbf{RB-insert(}x\textbf{)}
- initial step: regular insert & color \(x\) \textbf{red.}
- while \(x \neq \text{root}\) and \(p(x)\) is \textbf{red}
  \begin{itemize}
  \item \textbf{if} \(p(x) = \text{left}(g(x))\)
  \end{itemize}

\[ \begin{align*}
  \text{y} & \leftarrow \text{right}(g(x)) \\
  \text{don't care about colors} \end{align*} \]

why can we assume \(y\) exists?
It is at least a dummy leaf, in which case the subtree of \(g(x)\) is \(g(x)\) 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
  \[ \text{if } p(x) = \text{left}(g(x)) \quad \begin{cases} \text{if } y \text{ is red then run CASE 1} \\ \text{else if } x = \text{right}(p(x)) \text{ then run CASE 2.} \\ \text{Run CASE 3} \end{cases} \]
  else \( \backslash p(x) = \text{right}(g(x)) \)
  do as before but with "left" \& "right" switched \( \backslash \) symmetric
  color the root black.
if y is red then run CASE 1

Every △ contributes same to black-height.

Recolor y, p(x), g(x)
Let x ← g(x)

blackheight(p(x)) preserved

return to while loop.
(repeat if new p(x) is red)
else if \( x = \text{right}(p(x)) \) then run CASE 2.

\[
\begin{align*}
\text{rotate-left}(p(x)) & \quad \text{rotate-left}(p(x)) \\
\text{switch labels } x \leftrightarrow p(x) & \quad \text{switch labels } x \leftrightarrow p(x)
\end{align*}
\]

\[
\begin{align*}
\text{conditions of case 3} & \\
\end{align*}
\]

Run CASE 3

\[
\begin{align*}
\text{rotate-right}(g(x)) & \quad \text{rotate-right}(g(x)) \\
\text{re-color } p, g & \quad \text{re-color } p, g \\
\text{relabel } g & \quad \text{relabel } g
\end{align*}
\]

\[
\begin{align*}
\text{\( x \) moves up in tree} & \quad \text{\( x \) moves up in tree} \\
\text{Now } p(x) \text{ is black.} & \quad \text{Now } p(x) \text{ is black.} \\
\text{Exit while} & \quad \text{Exit while}
\end{align*}
\]
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations;

**RB-insert**(x)

- initial step: regular insert & color x red.
- while x ≠ root and p(x) is red
  - if p(x) = left(g(x))
    - y ← right(g(x))
    - if y is red then run CASE 1
    - else if x = right(p(x)) then run CASE 2.
    - Run CASE 3
  - else p(x) = right(g(x))
    - do as before but with "left" & "right" switched

color the root black