Dynamic \textit{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[\( x \)]

\[ \text{→ Fails rule 5} \]
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[$x$]

$\rightarrow$ Fails rule 5 $\Rightarrow$ 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 $\geq 2$ 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: $\approx 2\log n$.

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     y
```

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

$T_2$:

```
  ?
 /  \
 /    \
A      B
    /  \
   /    \
  x     y
```

$O(1)$ time

left-rotate($T_2$, A)

right-rotate($T_1$, B)
**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)

problem transferred from new node to its grandparent
Insert 15:

1. Rotate right at 18:
   - New root: 18
   - 7 becomes left child
   - 10 becomes right child

2. Rotate left at 7:
   - New root: 7
   - 18 becomes right child
   - 10 becomes left child

Done!
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

\texttt{RB-insert}(x)
- initial step: regular insert & color \( x \) red.
- while \( x \neq \text{root} \) and \( p(x) \) is red
  \hspace{1em} if \( p(x) = \text{left}(g(x)) \)
  \hspace{1em} \{ \begin{align*}
    & \begin{array}{c}
      g(x) \\
      p(x) \\
      x
    \end{array} \\
    & \begin{array}{c}
      \text{or} \\
      x
    \end{array}
  \end{align*} \}
  \text{don't care about colors}

\hspace{1em} \text{why does } g(x) \text{ exist?}

\hspace{1em} p(x) = \text{red} \text{ so it isn't the root.}
The algorithm is basically: re-color 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
  - if \(p(x) = \text{left}(g(x))\)
    \[
    \begin{align*}
    y &= \text{right}(g(x)) \\
    \end{align*}
    \]
- \(x \neq \text{root} \) and \(p(x)\) is red

\(\{ \begin{array}{c}
g(x) \\
p(x) \\
\end{array} \} \) or \(\{ \begin{array}{c}
\text{ or } \\
\text{ or } \\
\end{array} \} \)

don't care about colors

\begin{itemize}
  \item why can we assume \(y\) exists?
  \item It is at least a dummy leaf.
  \item in which case the subtree of \(g(x)\)
  \item is \(g(x)\) before inserting \(x\).
\end{itemize}
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 \textit{CASE 1}
    else if x = right(p(x)) then run \textit{CASE 2}.

    Run \textit{CASE 3}

  else \quad p(x) = right(g(x))

  do as before but with "left" & "right" switched \textit{\indrome{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)
If $x = \text{right}(p(x))$ then run CASE 2.

Run CASE 3

$\text{rotate-right}(g(x))$

$\text{rotate-left}(p(x))$

Switch labels $x \leftrightarrow p(x)$

$x$ moves up in tree

Now $p(x)$ is black.

Exit while
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

\[
\text{if } p(x) = \text{left}(g(x)) \quad \left\{ \begin{array}{l}
g(x) \\
p(x) \\
x \quad y \quad x
\end{array} \right.
\]

\[
\text{if } y \text{ is red then run CASE 1}
\]
\[
\text{else if } x = \text{right}(p(x)) \text{ then run CASE 2.}
\]

Run CASE 3

\[
\text{else } \quad \text{do as before but with "left" & "right" switched \& symmetric color the root black}
\]

\textbf{rotate, exit WHILE}

\textbf{recolor, repeat WHILE}
CASE 1

Insert 15

rotate-right(18)

CASE 2

rotate-left(7)

CASE 3

Done!