Dynamic \textbf{BALANCED SEARCH TREES} \textcolor{magenta}{(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.
4) every red node has a black parent.
5) for any node $x$: all paths down to leaves contain equal number of black nodes = $\text{black-height}[x]$, not including $x$. 

**The 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[\( x \)]

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

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

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

black-height difference: 6 vs 3
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 = size(T)+1
#leaves in T': same
height(T') ≤ log(n+1)  [higher degree ⇒ smaller height; worst-case: binary]
Re-inserting red nodes: at most doubles height → height(T) ≤ 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

$T_1$

$A$ \(\rightarrow\) $B$

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

$T_2$

$A$ \(\leftarrow\) $B$

left-rotate($T_2$, $A$)

$X \leq A \leq Y \leq B \leq Z$

$O(1)$ time

$X \leq A \leq Y \leq B \leq Z$
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

recolored parent of 15

creates new black-height problem

black height problem fixed

creates new consecutive-red 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
  - if $p(x) = \text{left}(g(x))$
    - $g(x)$ or \\
    - don't care about colors

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 \( p(x) = \text{left}(g(x)) \)
    \[ \begin{cases} 
    g(x) \\
    p(x)
    \end{cases} \]
    \[ x \quad y \quad \text{or} \quad \text{don't care about colors} \]
    \[ y \leftarrow \text{right}(g(x)) \]

\[ \text{why can we assume} \ y \text{ exists?} \]
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

**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 \ symmetric

color the root black
if \( y \) is red then run **CASE 1**

Every \( \triangle \) contributes same to black-height.

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

Let \( x \leftarrow g(x) \)

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.

\[
\begin{align*}
\text{rotate-left}(p(x)) & \quad \rightarrow \\
\text{switch labels } x & \leftrightarrow p(x) & \rightarrow \\
\text{conditions of case 3} & \\
\end{align*}
\]

Run CASE 3

\[
\begin{align*}
\text{rotate-right}(g(x)) & \quad \rightarrow \\
\text{re-color } \text{p}, g & \rightarrow \\
\text{relabel } g & \rightarrow \\
\text{x moves up in tree} & \\
\text{Now } p(x) & \text{ is black.} & \quad \text{Exit while}
\end{align*}
\]
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
  \[
  \text{if } p(x) = \text{left}(g(x)) \quad \begin{cases} 
    y \leftarrow \text{right}(g(x)) \\
    \text{if } y \text{ is red then run CASE 1} \\
    \text{else if } x = \text{right}(p(x)) \text{ then run CASE 2.}
  \end{cases}
  \]
  - Run CASE 3
  \[
  \text{else } \backslash p(x) = \text{right}(g(x)) \quad \text{do as before but with } \text{"left" } \& \text{"right" switched}.
  \]
\text{color the root black}
CASE 1
Insert 15

CASE 2
rotate-right(18)

CASE 3
rotate-left(7)

Done!