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.

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 = \text{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[\( x \)]

\[
\begin{array}{c}
\text{Fails rule 5}
\end{array}
\]
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
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 : \( n+1 = \text{size}(T)+1 \)
\#leaves in T' : same

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

Re-inserting \textcolor{red}{\text{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

\( T_1 \)

- \( X \leq A \leq Y \leq B \leq Z \)

- Right-rotate(\( T_1, B \))

\( T_2 \)

- \( X \leq A \leq Y \leq B \leq Z \)

- 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)

problem transferred from new node to its grandparent
Done!

rotate-right(18)

rotate-left(7)
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations.

\[ RB\text{-}insert(x) \]
- initial step: regular insert & color \( x \) red.
- while \( x \neq \text{root} \) and \( p(x) \) is red
  - if \( p(x) = \text{left}(g(x)) \)
    - \[ \begin{tikzpicture}
      \node (x) at (0,0) [circle, fill=red] {x};
      \node (p) at (1,0) [circle, fill=blue] {p(x)};
      \node (g) at (2,1) [circle, fill=blue] {g(x)};
      \draw (x) -- (p);
      \draw (p) -- (g);
    \end{tikzpicture} \]
  - or \[ \begin{tikzpicture}
      \node (x) at (0,0) [circle, fill=red] {x};
      \node (p) at (0,1) [circle, fill=blue] {p(x)};
      \node (g) at (0,2) [circle, fill=blue] {g(x)};
      \draw (x) -- (p);
      \draw (p) -- (g);
    \end{tikzpicture} \]
  - don't care about colors

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

**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))
  - don't care about colors

**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

**RB-insert(x)**
- initial step: regular insert & color x red.
- while x ≠ root and p(x) is red
  
  if p(x) = \text{left}(g(x))

  \begin{align*}
  &y \leftarrow \text{right}(g(x)) \\
  &\text{if } y \text{ is red then run } \text{CASE 1} \\
  &\text{else if } x = \text{right}(p(x)) \text{ then run } \text{CASE 2.}
  \end{align*}

  Run CASE 3

  else \ p(x) = \text{right}(g(x))

  do as before but with “left” & “right” switched \ symmetric

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

\[
\begin{array}{c}
\text{CASE 1:} \\
\quad \text{if } y \text{ is red then run CASE 1:} \\
\quad \quad \text{or} \\
\quad \quad \text{or}
\end{array}
\]

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.

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

\[
\begin{array}{c}
\text{rotate-left}(p(x)) \\
\text{switch labels } x \leftrightarrow p(x)
\end{array}
\]

conditions of case 3

Run CASE 3

\[
\begin{array}{c}
\text{rotate-right}(g(x)) \\
\text{re-color } p, g \\
\text{relabel } g
\end{array}
\]

\[x \text{ moves up in tree}
\]

Now $p(x)$ is black. Exit while
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 \(p(x) = \text{left}(g(x))\)
    - \(y \leftarrow \text{right}(g(x))\)
    - if \(y\) is red then run case 1
      - else if \(x = \text{right}(p(x))\) then run case 2.
        - Run case 3
  - else \(p(x) = \text{right}(g(x))\)
    - do as before but with "left" & "right" switched
    - color the root black