dynamic BALANCED SEARCH TREES (Non-Random)

Objectives: search, insert, delete in $O(\log n)$ time
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
RED-BLACK trees

Structure: 1) nodes are colored red or black.
**RED-BLACK trees**

**Structure:**
1) nodes are colored red or black.
2) root is always black.
**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.
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.
**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 = 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[$x$]

→ 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[$x$]

→ Fails rule 5 ⇒ 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.

Rule 4
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.
No hope to recolor \( \ldots \) too unbalanced

Any root→leaf path of size \( k \) must have \( \geq \frac{k}{2} \) black nodes.

So if any path is greater than another, we can't make it RB.

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

Q: what is max. degree of T'?
If $\text{root}(T)$ has same black-height on all paths then $\text{height}(T')$ is perfectly balanced.

Q: What is max. degree of $T'$? (2-3-4)
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
If root($T$) has same black-height on all paths then \( \text{height}(T') \) is perfectly balanced.

Q: what is \text{max} degree of $T'$? (2-3-4)

- \#leaves in $T$: \( n + 1 = \text{size}(T) + 1 \)
- \#leaves in $T'$: same
- \text{height}($T'$) $\leq \log(n+1)$  
  [higher degree $\Rightarrow$ smaller height; worst-case: binary]
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)
We have seen that RB trees are reasonably balanced: \( \sim O(\log n) \) time.

operations: \( \text{search, min, max, next, prev} \)

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

\[ T_1 \]

\[ X \leq A \leq Y \leq B \leq Z \]
We have seen that RB trees are reasonably balanced: $\sim 2\log n$

$\rightarrow$ search, min, max, next, prev: $O(\log n)$ time.

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

**ROTATIONS** in arbitrary BSTs

$T_1$  

```
    ?
   / \   
  B   A   ?
 / \   / \ 
X   Y 2  ?
```

$x \leq A \leq Y \leq B \leq z$

$T_2$

```
   ?
  /   
 A   ?
 /   /  
X   B   ?
 /   /  
Y   2  ?
```

$x \leq A \leq Y \leq B \leq z$
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**

$T_1$

$A \rightarrow B$

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

$\text{right-rotate}(T_1, B)$

$O(1)$ time

$T_2$

$A \rightarrow B$

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

$\text{left-rotate}(T_2, A)$
Insert in Red-Black Trees

greedy (optimistic) start: insert as in any BST
Greedy (optimistic) start:
1) Insert as in any BST
2) Color new node red (keeps black-height ok)
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)
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
**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 $O(1)$ recolorings and rotations per level.
**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)
example of $O(1)$-time error-corrections (per level)

insert 15

recolor parent of 15 & create black-height problem
example of $O(1)$-time error-corrections (per level)

1. Insert 15
2. Recolor parent of 15
3. Create black-height problem
4. New consecutive-red problem
5. Black height problem fixed.
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
Insert 15
Insert 15

rotate-right(18)

(doesn't affect BH in this case)
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

\text{RB-insert}(x)

- initial step: regular insert & color x red. If \( x = \text{root} \), trivial: \( x \rightarrow \text{black} \)
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

\[ \rightarrow \text{implies } x \text{ will be propagating up} \]
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

\[ \text{if } p(x) = \text{left}(g(x)) \quad \begin{cases} g(x) \quad \text{or} \quad \{ x \} \\ x \end{cases} \]

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

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

Why does g(x) exist?

p(x) = 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
  
  \text{if } p(x) = \text{left}(g(x)) \quad \begin{cases} g(x) \quad \text{or} \quad x \quad \text{y} \quad \text{y} \quad \text{\{don't care about colors}\} \\
  \quad y \leftarrow \text{right}(g(x)) \quad \text{why can we assume y exists?}\end{cases}
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 \( p(x) = \text{left}(g(x)) \)
    \[
    y \leftarrow \text{right}(g(x))
    \]

\( \exists y \) why can we assume \( y \) exists?
It is at least a dummy leaf.
in which case the subtree of \( g(x) \) 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
  - if \( p(x) = \text{left}(g(x)) \)
    \[
    \begin{cases}
    y \leftarrow \text{right}(g(x)) & \text{if } y \text{ is red then run CASE 1} \\
    \end{cases}
    \]
  - \( g(x) \) is black
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

**RB-insert(x)**

- Initial step: regular insert & color x red.
- While \(x \neq \text{root} \) and \(p(x)\) is red
  - If \(p(x) = \text{left}(g(x))\)
    - \(y = \text{right}(g(x))\)
      - If \(y\) is red then run **CASE 1**
      - Else if \(x = \text{right}(p(x))\) then run **CASE 2**.
    - Run **CASE 3**
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 ≠ root and p(x) is red
  - if \( p(x) = \text{left}(g(x)) \)
    - y ← \text{right}(g(x))
      - if y is red then run \text{CASE 1}
      - else if \( x = \text{right}(p(x)) \) then run \text{CASE 2}
        - Run \text{CASE 3}
  - else \( p(x) = \text{right}(g(x)) \)
    - do as before but with "left" & "right" switched \text{symmetric}
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 \( p(x) = \text{right}(g(x)) \)
    - do as before but with “left” & “right” switched
- color the root black
if $y$ is red then run CASE 1
if $y$ is red then run CASE 1

Every ▲ contributes same to black-height.
if $y$ is red then run CASE 1

Every ▲ contributes same to black-height.
if \( y \) is red then run CASE 1

Every \( \triangle \) contributes same to black-height.
if y is red then run CASE 1 \[ \begin{cases} 
  \text{Recolor } y, p(x), g(x) \\
  \text{Let } x \leftarrow g(x) 
\end{cases} \]

Every contributes same to black-height.
if $y$ is red then run **CASE 1**

- Recolor $y$, $p(x)$, $g(x)$
- Let $x \leftarrow g(x)$

Every △ contributes same to black-height.

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.

(y is black)
else if $x = \text{right}(p(x))$ then run CASE 2.

rotate-left(p(x))
else if \( x = \text{right}(p(x)) \) then run \text{CASE 2.} \{ \}

\[
\begin{align*}
\text{rotate-left}(p(x)) & \rightarrow \\
\text{switch labels } x & \leftrightarrow p(x) \\
\text{conditions of case 3}
\end{align*}
\]
else if $x = \text{right}(p(x))$ then run CASE 2. 

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

Run CASE 3
else if $x = \text{right}(p(x))$ then run \textbf{CASE 2}. 

-solve the case \textbf{Case 2}:

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

-switch labels $x \leftrightarrow p(x)$

-conditions of case 3

Run \textbf{CASE 3}

-rotate-right($g(x)$)
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 pig
else if $x = \text{right}(p(x))$ then run \textit{CASE 2}. \\
\{ \\
\text{rotate-left}(p(x)) \rightarrow \text{switch labels } x \leftrightarrow p(x) \\
\} \text{ conditions of case 3} \\

Run \textit{CASE 3} \\
\{ \\
\text{rotate-right}(g(x)) \rightarrow \text{re-color } p,g \rightarrow \text{relabel } g \\
\}
else if $x = \text{right}(p(x))$ then run CASE 2.

Conditions of case 3:

Run CASE 3:

$x$ moves up in tree.
else if \( x = \text{right}(p(x)) \) then run CASE 2.

Run CASE 3

\[ \text{rotate-left}(p(x)) \]

\[ \text{switch labels } x \leftrightarrow p(x) \]

conditions of case 3

\[ \text{rotate-right}(g(x)) \]

\[ \text{re-color } p,g \]

\[ \text{relabel } g \]

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

Now \( p(x) \) is black.

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

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

These will move x up (redefine)
(and it will be red)
CASE 1

Insert 15

rotate-right(18)

CASE 2

rotate-left(7)

CASE 3

Done!