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

G 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 = $\text{black-height}[x]$, 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[\( x \)]

\[ \rightarrow \text{Fails rule 5} \]
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
No hope to recolor ...
   too unbalanced

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

Rule 4
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.
Any root to leaf path of size $k$ must have $\geq \frac{k}{2}$ black nodes. So if any path is more than twice as long as another, we can't make it RB.
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 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 $T$: $n+1 = \text{size}(T)+1$

#leaves in $T'$: same
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]
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)

$\#\text{leaves in } T : n+1 = \text{size}(T)+1$
$\#\text{leaves in } T' : \text{same}$
$\text{height}(T') \leq \log(n+1)$ [higher degree $\Rightarrow$ smaller height; worst-case: binary]

Re-inserting 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)
We have seen that RB trees are reasonably balanced: $\sim 2 \log n$

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

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: $\sim 2 \log n$.

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

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

**Rotations** in arbitrary BSTs

$T_1$ to $T_2$:

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

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

$O(1)$ time
Insert in Red Black Trees

Greedy (optimistic) start: insert as in any BST
Insert in RED BLACK TREES

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 REDBLACK 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 (keeps black-height ok)

2) color new node red

3) if parent is also red (violate parent rule 4)
   then color parent black and
   look for problems further up trail up to root

begin an error-correcting trail 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-correction trail up to root, involving $O(1)$ recolorings and rotations per level

$O(\log n)$ time
example of $O(1)$-time error-corrections (per level)

```
    7
   / \      
  3   18
 /     \    
8      22
 /  \
11  26
```

`insert 15`
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
5. New consecutive-red problem
example of $O(1)$-time error-corrections (per level)

- Insert 15
- Recolor parent of 15
- Image: Red nodes represent the problem, with the new nodes having a new consecutive red problem

Problem transferred from new node to its grandparent
Insert 15

rotate-right(18)

doesn't affect BH in this case
rotate-right(18)

Insert 15
Insert 15

rotate-right(18)

rotate-left(7)
The process of balancing a binary search tree.

1. **Insert 15**: After inserting 15, the tree becomes unbalanced.
2. **Rotate Right (18)**: To balance the tree, we perform a right rotation at node 18.
3. **Insert 10**: After inserting 10, the tree becomes unbalanced again.
4. **Rotate Left (7)**: To balance the tree, we perform a left rotation at node 7.

The final balanced tree is achieved through these rotations.

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

```
RB-insert(x)
- initial step: regular insert & color x red. If x = root, trivial: x \rightarrow 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

$\Rightarrow$ implies $x$ 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
  - if \( p(x) = \text{left}(g(x)) \)
    \[
    \begin{cases} 
      g(x) \\
      p(x) \\
      x
    \end{cases}
    \]
    
  \begin{cases} 
    g(x) \\
    p(x) \\
    x
  \end{cases}
  \]
  don't care about colors

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

\[ \text{RB-insert}(x) \]

- initial step: regular insert & color \( x \) red.
- while \( x \neq \text{root} \) and \( p(x) \) is red
  - if \( p(x) = \text{left}(g(x)) \)
    - \( y \leftarrow \text{right}(g(x)) \)
  \[ \text{don't care about colors} \]

why can we assume \( y \) exists?
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))

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.

```
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
```

- g(x) is black
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
  \begin{itemize}
  \item if \( p(x) = \text{left}(g(x)) \)
    \begin{itemize}
    \item \( x \) or \( y \)
    \end{itemize}
  \item \( y \leftarrow \text{right}(g(x)) \)
    \begin{itemize}
    \item if \( y \) is red then run CASE 1
    \item else if \( x = \text{right}(p(x)) \) then run CASE 2.
    \end{itemize}
  \end{itemize}
- Run CASE 3

\text{or \quad or \quad ?}
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
  
  \[ \text{if } p(x) = \text{left}(g(x)) \]
  
  \[ \begin{cases} 
  \text{if } y \text{ is red then run } \text{CASE 1} & \text{or} \\
  \text{else if } x = \text{right}(p(x)) \text{ then run } \text{CASE 2}. & \text{or} \\
  \text{Run CASE 3} & \text{or} \\
  \end{cases} \]

\[ \text{else } \backslash \backslash \, p(x) = \text{right}(g(x)) \]
  
  do as before but with “left” & “right” switched \( \backslash \text{symmetric} \)
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

**RB-insert** \((x)\)
- initial step: regular insert & color \(x\) red.
- while \(x \neq \text{root}\) 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
if $y$ is red then run \textit{CASE 1} \begin{cases} \begin{aligned} & x \quad \text{or} \quad y \\ \end{aligned} \end{cases} \begin{cases} \begin{aligned} & x \\ \end{aligned} \end{cases}
if $y$ is red then run \textit{CASE 1} \quad \begin{cases} \quad x \quad \text{or} \quad y \end{cases}

Every \quad \text{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

Every $\triangleright$ contributes same to black-height.
if \( y \) is red then run \( \text{CASE 1} \)  

\[
\begin{array}{c}
\text{Recolor } y, p(x), g(x) \\
\text{Let } x \leftarrow g(x)
\end{array}
\]

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

\[
\begin{align*}
\text{Recolor } y, \ p(x), \ g(x) \\
\text{Let } x \leftarrow g(x)
\end{align*}
\]

\[
\begin{align*}
\text{blackheight}(p(x)) \text{ preserved} \\
\text{return to while loop.} \\
\text{(repeat if new } p(x) \text{ is red)}
\end{align*}
\]

Every \( \Delta \) contributes same to black-height.
else if \( x = \text{right}(p(x)) \) then run CASE 2. 

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

\begin{align*}
\text{rotate-left}(p_x)
\end{align*}
else if $x = \text{right}(p(x))$ then run \textsc{Case 2}.

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

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

\[
\begin{array}{c}
\text{conditions of case 3} \\
\end{array}
\]
else if $x = \text{right}(p(x))$ then run CASE 2.

conditions of case 3

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

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

\{ \text{conditions of case 3} \}

Run \text{CASE 3} \quad \begin{aligned} 
&\text{rotate-right}(g(x)) \\
\end{aligned}
else if $x = \text{right}(p(x))$ then run CASE 2.

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

conditions of case 3

Run CASE 3

\[
\begin{align*}
\text{rotate-right}(g(x)) & \quad \text{re-color}
p \quad \text{pig}
\end{align*}
\]
else if $x = \text{right}(p(x))$ then run CASE 2.

Run CASE 3

Conditions of case 3

rotate-left($p(x)$)

switch labels $x \leftrightarrow p(x)$

rotate-right(g(x))

re-color p,g

relabel g
else if $x = \text{right}(p(x))$ then run CASE 2.

Run CASE 3

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

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

\text{conditions of case 3}

Run \textit{CASE 3}

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

\( x \) moves up in tree

\text{Now } p(x) \text{ is black. Exit while}
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations;

1. **RB-insert** \( x \)
   - initial step: regular insert \& color \( x \) red.
   - while \( x \neq \text{root} \) and \( p(x) \) is red
     
     1. If \( p(x) = \text{left}(g(x)) \)
        
        - \( y \leftarrow \text{right}(g(x)) \)
        
        - 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}

     2. Else \( p(x) = \text{right}(g(x)) \)
       
       do as before but with "left" \& "right" switched \( \text{\backslash symmetric} \)

   - color the root black

   - recolor, repeat \text{WHILE}

   - rotate, exit \text{WHILE}
CASE 1

CASE 2
rotate-right(18)

CASE 3
rotate-left(7)