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

Δ always maintain $\Theta(\log n)$ height & update 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

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.
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.

**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]`

*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[\(x\)]

\[\text{\textsc{x} \quad \Rightarrow \quad \text{Fails rule 5}}\]
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
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. Rule 4

So if any path is $>2$ times longer than another, we can't make it RB.
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.
If $\text{root}(T)$ has same black-height on all paths then $\text{height}(T')$ is perfectly balanced.
If $\text{root}(T)$ has same black-height on all paths then $\text{height}(T')$ is perfectly balanced.

Q: what is max degree of $T'$?
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)
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 = size(T)+1
#leaves in T' : same
height(T') \leq \log(n+1) \quad [\text{higher degree} \Rightarrow \text{smaller height}; \text{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 = size(T) + 1
#leaves in T' : same
height(T') \leq \log(n+1) \quad [\text{higher degree } \Rightarrow \text{smaller height}; \text{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: $\sim 2\log n$.

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

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

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

$T_1$ → $T_2$

```
T_1

A
  /    /
B   ?
  /    /
A   Z
  /    /
X   Y
```

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

```
T_2

A
  /    /
B   ?
  /    /
X   Y
  /    /
Z
```

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

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

- \( \text{right-rotate}(T_1, B) \)
- \( \text{left-rotate}(T_2, A) \)

\( O(1) \) time

\[ T_2 \]

```
A
 /   \
B     
 \       X
   \   /   /   \   /   \
  X   Y Z     X     Y Z
```

\( X \leq A \leq Y \leq B \leq Z \)
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:
\[\begin{align*}
1) & \text{ insert as in any BST} \\
2) & \text{ color new node red (keeps black-height ok)} \\
3) & \text{ if parent is also red (violate parent rule 4)}
\end{align*}\]
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)
example of $O(1)$-time error-corrections (per level)

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

Problem transferred from new node to its grandparent
insert 15

rotate-right(18)
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 \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
  - if \( p(x) = \text{left}(g(x)) \)  
    - \( g(x) \) or \( x \)
      - 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 \( p(x) = \text{left}(g(x)) \)

\[ \begin{cases} 
  g(x) & \text{or} \\
  p(x) \\
  x
\end{cases} \]

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

\( p(x) = \text{red} \) 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))

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 \( x \neq \text{root} \) and \( p(x) \) is red:
  - If \( p(x) = \text{left}(g(x)) \):
    - y ← right(g(x))

\[
\begin{cases}
  g(x) \\
p(x) \\
x
\end{cases}
\]

\[
\begin{cases}
  g(x) \\
p(x) \\
x
\end{cases}
\]

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) = 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.

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

- initial step: regular insert & color \( x \) red.
- while \( x \neq \text{root} \) and \( p(x) \) is red
  - if \( p(x) = \text{left}(g(x)) \)
    \[
    \begin{cases}
      \text{Run CASE 1} & \text{if } y \text{ is red} \\
      \text{or} & \text{else if } x = \text{right}(p(x)) \text{ then run CASE 2.}
    \end{cases}
    \]
    \[
    \begin{cases}
      \text{Run CASE 3} & \text{(else)} \\
      \text{or} & \text{?}
    \end{cases}
    \]
  - else if \( x = \text{right}(p(x)) \) then run CASE 2.
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))
    \begin{align*}
    \text{y} &\leftarrow \text{right}(g(x)) \\
    \text{if } \text{y} \text{ is red then run CASE 1} \\
    \text{else if } x = \text{right}(p(x)) \text{ then run CASE 2.}
    \end{align*}
  
  Run CASE 3
  
  else if \ p(x) = \text{right}(g(x))
    \text{do as before but with “left” & “right” switched, \ symmetric
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
  \[
  \begin{align*}
  \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} \\
  \text{Run CASE 3} \quad \begin{cases} 
  \text{or} \\
  \text{?}
  \end{cases} \\
  \text{else } \downarrow p(x) = \text{right}(g(x)) \quad \text{do as before but with "left" & "right" switched} \quad \text{\( \downarrow \text{symmetric} \)}
  \end{align*}
\]
- color the root black.
if $y$ is red then run CASE 1

\begin{align*}
\text{or } \quad x & \quad y \\
\text{or } \quad x & \quad y
\end{align*}
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

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

Every contributes same to black-height.

Recolor $y, p(x), g(x)$
Let $x \leftarrow g(x)$
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)
else if \( x = \text{right}(p(x)) \) then run CASE 2.

\[ (y \text{ is black}) \]
else if $x = \text{right}(p_{\text{wx}})$ then run \text{CASE 2.} \{ \begin{align*}
\text{rotate-left}(p_{\text{wx}})
\end{align*} \}
else if \( x = \text{right}(p(x)) \) then run CASE 2. 

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

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

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

Run CASE 3

\begin{align*}
\text{rotate-right}(g(x))
\end{align*}
else if $x = \text{right}(p(x))$ then run \text{CASE 2}.

{ \text{rotate-left}(p(x)) \rightarrow \text{switch labels} \quad x \leftrightarrow p(x) \rightarrow \text{conditions of case 3} }

Run \text{CASE 3}

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

conditions of case 3

Run CASE 3

rotate-right(g(x))

re-color $p,g$

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

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

Run **CASE 3**

$$
\begin{align*}
\text{rotate-right}(g(x)) & \quad \rightarrow \\
\text{re-color} \ p(g) & \\
\text{relabel} \ g & \\
\end{align*}
$$

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

conditions of case 3

Run CASE 3

rotate-right(g(x))

re-color p,g

relabel g

\( x \) 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 \( p(x) = \text{right}(g(x)) \)
  - do as before but with "left" & "right" switched \( \backslash \text{symmetric} \)
  - color the root black

These will move x up (redefine)