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.

**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[\( x \)]
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[\( x \)]

\[ \rightarrow \text{fails rule 5} \Rightarrow \text{fix by making some nodes red.} \]
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
No hope to recolor too unbalanced

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.
Proof that RB trees have height $< 2\log n$
Any root→leaf path of size $k$? Rule 4

must have $\geq \frac{k}{2}$ black nodes.

So if any path is $\geq 2$ times longer than another, we can't make it RB.

Proof that RB trees have height $< 2\log n$

1) Fact: If a tree is perfectly balanced the height is $\log n$
Proof that RB trees have height \(< 2 \log n\)

1) Fact: If a tree is perfectly balanced the height is \(\log n\)

2) If a tree is not perfectly balanced, there is a node, \(x\), at depth \(d < \log n\)

Any root-to-leaf path of size \(k\)

\[ \Rightarrow \frac{k}{2} \text{ black nodes}. \]

So if any path is \(>2\) times longer than another, we can’t make it RB.
Proof that RB trees have height < 2logn

1) Fact: If a tree is perfectly balanced the height is logn

2) If a tree is not perfectly balanced, there is a node, x, at depth d < logn

3) By our claim above there can’t be any other node at depth > 2d
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.

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

Every non-leaf (still) has at least 2 children.
If root(T) has same black-height on all paths then height(T') is perfectly balanced

#leaves in T : $n+1 = \text{size}(T) + 1$
#leaves in T' : same

Every non-leaf (still) has at least 2 children
If \( \text{root}(T) \) has same black-height on all paths then \( \text{height}(T') \) is perfectly balanced.

- \#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]

Every non-leaf (still) has at least 2 children.
If root(T) has same black-height on all paths then height(T') is perfectly balanced.

# 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)

Every non-leaf (still) has at least 2 children
We have seen that RB trees are reasonably balanced: $\sim 2 \log n$

$\mathcal{O}$ 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: $\sim 2\log n$

$\rightarrow$ 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

$x \leq a \leq y \leq b \leq z$

right-rotate($T_1, B$)

$x \leq a \leq y \leq b \leq z$
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

\[ T_1 \]

\[ \begin{array}{c}
A \\
\downarrow \\
B \\
\downarrow \\
x \\
\end{array} \quad \begin{array}{c}
y \\
\downarrow \\
z \\
\end{array} \quad \begin{array}{c}
x \\
\end{array} \]

\( x \leq A \leq y \leq B \leq z \)

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

\[ \text{left-rotate}(T_2, A) \]

\( O(1) \) time

\[ T_2 \]

\[ \begin{array}{c}
A \\
\downarrow \\
B \\
\downarrow \\
x \\
\end{array} \quad \begin{array}{c}
y \\
\downarrow \\
z \\
\end{array} \]

\( x \leq A \leq y \leq B \leq z \)
INSERTION
IN A R-B TREE
INSERTION
IN A R-B TREE

Regular insertion as RED leaf
INSERTION
IN A R-B TREE

Regular insertion as RED leaf

No black height violation introduced.
Regular insertion as RED leaf

- No black height violation introduced.
- Might have 1 red-red violation.
Regular insertion as RED leaf

- No black height violation introduced.
- Might have 1 red-red violation.

- If so, start fixing: locally & upward
fixing: locally & upward
fixing: locally & upward

Loop invariants:
- No black height violation.
- 1 red-red violation, at positions (P, X)
fixing: locally & upward

Loop invariants:
- No black height violation.
- 1 red-red violation, at positions \( PIx \)

Loop outcomes:
- Obtain R-B tree
  OR
- Redefine ancestor as \( x \)
Redefine ancestor as \( x = \text{fix upward} \)

OR

Obtain R-B tree

Loop outcomes:

- No black height violation.
- I red-red violation, at positions \( x - P \)

Loop invariants:

fixing: locally & upward
fixing: **locally & upward**

**Loop invariants:**
- No black height violation.
- 1 red-red violation, at positions $P, I, X$

**Loop outcomes:**
- Obtain R-B tree
- OR
- Redefine ancestor as $x = \text{fix upward}$

Local positions considered in loop
fixing: locally & upward

Loop invariants:
- No black height violation.
- 1 red-red violation, at positions $P, I, X$

Loop outcomes:
- Obtain R-B tree
- OR
- Redefine ancestor as $x = \text{fix upward}$

Local positions considered in loop
x & p : both red if we needed to continue loop

g: grandparent of x

y: "uncle" of x

Diagram:
x & p: both red if we needed to continue loop

g: grandparent of x

y: "uncle" of x

2 possible shapes...
x & p: both red if we needed to continue loop

g: grandparent of x

y: "uncle" of x

2 possible shapes...

...and their mirror images
\(x \& p\): both red if we needed to continue loop

\(g\): grandparent of \(x\)

\(y\): "uncle" of \(x\)

---

\(g\) must exist because \(p\) can't be root

\(y\) must exist (could be a fake leaf)
x & p: both red if we needed to continue loop

g: grandparent of x

y: "uncle" of x

2 possible shapes...

...and their mirror images

\[
\begin{align*}
\text{g must exist: \textbf{must be black}} & \quad \text{because p can't be root} \\
\text{y must exist} & \quad (\text{could be a fake leaf})
\end{align*}
\]
x & p: both red if we needed to continue loop

g: grandparent of x

y: "uncle" of x

We will handle the 2 main shapes:

2 possible shapes...

...and their mirror images

The others can be done by symmetry:
x & p: both red if we needed to continue loop

g: grandparent of x

y: "uncle" of x

We will handle the 2 main shapes:

Case 1: y is red

Cases 2 & 3: y is black
Case 1:  y is red
Case 1: \( y \) is red

\[ \rightarrow \text{Recolor } p, g, y \]
Case 1: y is red

→ Recolor p, g, y

- Preserve black-height invariant.
Case 1: \( y \) is red

1. Recolor \( p, g, y \)

- Preserve black-height invariant.
  (total 8 nodes on any path
  from global root down to leaves)
Case 1: \( y \) is red

- Recolor \( p, g, y \)

- Preserve black-height invariant.
  (total \( B \) nodes on any path from global root down to leaves)

- Eliminate \( P_X \) violation.
Case 1: \( y \) is red

\[ \rightarrow \text{Recolor } p, g, y \]

- Preserve black-height invariant.
  (total \( B \) nodes on any path
  from global root down to leaves)
- Eliminate \( P_x \) violation.
- If no new violation \( P_g \)
  then DONE
  (note: if \( g = \text{root} \)
  color \( g \) black)

or
Case 1: \( y \) is red

1. Recolor \( p, g, y \)

- Preserve black-height invariant.
  (total B nodes on any path from global root down to leaves)
- Eliminate \( P_{\Box} \) violation.
- Might cause violation

(otherwise DONE)
Case 1: \( y \) is red

\[\rightarrow \text{Recolor } p, g, y\]

- Preserve black-height invariant.
  (total \( B \) nodes on any path from global root down to leaves)
- Eliminate \( P \times X \) violation.
- Might cause violation otherwise DONE

\[\text{fix upward}\]

\[? \uparrow g\]

\(? \text{ becomes } p\]

\( g \text{ becomes } x\]
Case 1: \( y \) is red

\[ \rightarrow \text{Recolor } p, g, y \quad O(1) \]

- Preserve black-height invariant.
  (total B nodes on any path from global root down to leaves)

- Eliminate \( P_x \) violation.

- Might cause violation \( \uparrow \downarrow \) otherwise DONE

  \[ \text{fix upward} \]

\[ ? \to p \quad g \to x \]

\[ ? \to p \quad g \to x \]

\[ \text{Invariants restored} \]
Cases 2 & 3: \( y \) is black
Cases 1 & 3: $y$ is black

Case 2

Case 3

Diagram: A tree structure with nodes labeled $g$, $p$, $o$, $y$, and $x$. The nodes are connected in a specific order.
Cases 2 & 3: \( y \) is black

Case 2

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

\[
\begin{array}{c}
p
\xrightarrow{\text{Left-rotate}(p)}
\end{array}
\]
Cases 2 & 3: y is black

Case 2

Left-rotate(p)
Cases 2 & 3: \( y \) is black

- Preserve black-height invariant.

**Case 2**

Left-rotate(\( p \))
Cases 2 & 3: \( y \) is black

- Preserve black-height invariant.
- Swap labels \( x \leftrightarrow p \)
- No new violation

Case 2

Left-rotate(\( p \))
Cases 2 & 3: $y$ is black

Case 2

Case 3
Cases 2 & 3: $y$ is black

Case 2:

[Diagram showing a tree with nodes labeled 'p', 'y', 'x', 'g', and an arrow indicating 'Left-rotate(p)' and 'swap(p, x)'.]

Case 3:

[Diagram showing a tree with nodes labeled 'p', 'y', 'x', 'g'.]

Handling Case 3:
Cases 2 & 3: \( y \) is black

Case 2

```
           g
          / \
         p   y
         /   \
        X    
```

Left-rotate(p) & swap(p,x)

Case 3

```
           g
          / \
         p   y
         /   \
        X    
```

Handling Case 3:

```
           g
          / \
         p   y
         /   \
        X    
```

Right-rotate(g)
Cases 2 & 3: $y$ is black

Case 2

Case 3

Handling Case 3:
Cases 2 & 3: $y$ is black

- Preserve black-height invariant.

Case 2:

Left-rotate($p$)

& swap($p$, $x$)

Case 3:

Right-rotate($g$)

Handling Case 3:
Cases 2 & 3: \( y \) is black

- Preserve black-height invariant.

Case 2

Case 3

Left-rotate(\( p \))
& swap(\( p, x \))

Handling Case 3:

Right-rotate(\( g \))
Recolor \( p \) & \( g \)
Cases 2 & 3: $y$ is black

- Preserve black-height invariant.
- Eliminate $P_x$ violation in case 3.

Handling Case 3:

**Case 2**

$P \leftarrow g \leftarrow y$

Left-rotate($p$) & swap($p, x$)

**Case 3**

$P \leftarrow g \leftarrow y$

Right-rotate($g$) Recolor $p \& g$
Cases 2 & 3: \( y \) is black

- Preserve black-height invariant.
- Eliminate \( P_x^\perp \) violation in case 3.
- No new violation

Handling Case 3:

**Case 2**

- Left-rotate \( (p) \)
- & swap \( (p, x) \)

**Case 3**

- Right-rotate \( (g) \)
- Recolor \( p \) & \( g \)
Cases 1 & 3: $y$ is black

- Preserve black-height invariant.
- Eliminate $P_x$ violation in case 3.
- No new violation

Handling Case 3:

Case 2

Left-rotate($p$)
& swap($p, x$)

Case 3

Right-rotate($g$)
Recolor $p$ & $g$
Each case takes $O(1)$ time

All together $O(\log n)$ time
Each case takes $O(1)$ time

All together $O(\log n)$ time

Possibly lots of recoloring (Case 1)

followed by Case 3 OR Case 2 $\rightarrow$ 3 (1 or 2 rotations, total)

* useful property
Case 1

Insert 15
Case 1

Insert 15
Case 1

Case 2

Insert 15
Insert 15

Case 1

Case 2

rotate-right(18)
Case 1

Case 2

rotate-right(18)