LECTURE 9

Balanced Search Trees
• Red-black trees
• Height of a red-black tree
• Rotations
• Insertion

Augmenting Data Structures
• Dynamic order statistics
• Methodology
• Interval trees

Introduction to Computer Programming
Balanced search trees

**Balanced search tree:** A search-tree data structure for which a height of $O(\lg n)$ is guaranteed when implementing a dynamic set of $n$ items.

**Examples:**
- AVL trees
- Red-black trees
Red-black trees

This data structure requires an extra one-bit color field in each node.

**Red-black properties:**

1. Every node is either red or black.
2. The root and leaves (NIL’s) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node $x$ to a descendant leaf have the same number of black nodes $= \text{black-height}(x)$. 

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson
Example of a red-black tree

$h = 4$

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson
Example of a red-black tree

1. Every node is either red or black.
Example of a red-black tree

2. The root and leaves (NIL’s) are black.
3. If a node is red, then its parent is black.
Example of a red-black tree

4. All simple paths from any node $x$ to a descendant leaf have the same number of black nodes = $black\text{-}height(x)$. 
Height of a red-black tree

**Theorem.** A red-black tree with $n$ keys has height $h \leq 2 \lg(n + 1)$.

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.
Theorem. A red-black tree with $n$ keys has height 
$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

Intuition:
• Merge red nodes into their black parents.
Height of a red-black tree

**Theorem.** A red-black tree with $n$ keys has height

$$h \leq 2 \log(n + 1).$$

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.
Height of a red-black tree

**Theorem.** A red-black tree with \( n \) keys has height
\[
h \leq 2 \lg(n + 1).
\]

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.
Height of a red-black tree

**Theorem.** A red-black tree with \( n \) keys has height
\[
h \leq 2 \lg(n + 1).
\]

**Proof.** (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.
Theorem. A red-black tree with $n$ keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

**Intuition:**
- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth $h'$ of leaves.
Proof (continued)

- We have $h' \geq h/2$, since at most half the leaves on any path are red.

- The number of leaves in each tree is $n + 1$
  \[n + 1 \geq 2^{h'}\]
  \[\Rightarrow \lg(n + 1) \geq h' \geq h/2\]
  \[\Rightarrow h \leq 2 \lg(n + 1).\]
Query operations

**Corollary.** The queries **SEARCH**, **MIN**, **MAX**, **SUCCESSOR**, and **PREDECESSOR** all run in $O(\lg n)$ time on a red-black tree with $n$ nodes.
Modifying operations

The operations **INSERT** and **DELETE** cause modifications to the red-black tree:

- the operation itself,
- color changes,
- restructuring the links of the tree via "rotations"
Rotations

Rotations maintain the inorder ordering of keys:
• \( a \in \alpha, \ b \in \beta, \ c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c. \)

A rotation can be performed in \( O(1) \) time.
**Insertion into a red-black tree**

**IDEA:** Insert $x$ in tree. Color $x$ red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

```
    7
   / \
  3   18
 /   /  \
8   10   22
       /    /
       11   26
```
Insertion into a red-black tree

**IDEA:** Insert $x$ in tree. Color $x$ red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**
- Insert $x = 15$.
- Recolor, moving the violation up the tree.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson
Insertion into a red-black tree

**Idea:** Insert $x$ in tree. Color $x$ red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**
- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- **Right-Rotate(18).**
**Insertion into a red-black tree**

**Idea:** Insert $x$ in tree. Color $x$ red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**
- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- **Right-Rotate(18).**
- **Left-Rotate(7)** and recolor.
Insertion into a red-black tree

**Idea:** Insert $x$ in tree. Color $x$ red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**
- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- **Right-Rotate(18).**
- **Left-Rotate(7)** and recolor.
Pseudocode

RB-INSERT($T, x$)

TREE-INSERT($T, x$)

color[$x$] ← RED  \(\triangleright\) only RB property 3 can be violated

while $x \neq \text{root}[T]$ and color[$p[x]$] = RED

do if $p[x] = \text{left}[p[p[x]]$

then $y ← \text{right}[p[p[x]]$  \(\triangleright\) $y = \text{aunt/uncle of } x$

if color[$y$] = RED

then ⟨Case 1⟩

else if $x = \text{right}[p[x]]$

then ⟨Case 2⟩ \(\triangleright\) Case 2 falls into Case 3

⟨Case 3⟩

else ⟨“then” clause with “left” and “right” swapped⟩

color[root[$T$]] ← BLACK
Graphical notation

Let \( \triangle \) denote a subtree with a black root.

All \( \triangle \)'s have the same black-height.
Case 1

(Or, children of $A$ are swapped.)

Push $C$’s black onto $A$ and $D$, and recurse, since $C$’s parent may be red.

Recolor

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson
Case 2

\[
\text{LEFT-ROTATE}(A)
\]

Transform to Case 3.
Case 3

RIGHT-ROTATE(C)

B

C

A

x

y

Done! No more violations of RB property 3 are possible.
Analysis

• Go up the tree performing Case 1, which only recolors nodes.

• If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

**Running time:** $O(\lg n)$ with $O(1)$ rotations.

**RB-DELETE** — same asymptotic running time and number of rotations as RB-INSERT (see textbook).
Dynamic order statistics

**OS-SELECT**(i, S): returns the i\(^{th}\) smallest element in the dynamic set S.

**OS-RANK**(x, S): returns the rank of \(x \in S\) in the sorted order of S’s elements.

**Idea**: Use a red-black tree for the set S, but keep subtree sizes in the nodes.

Notation for nodes: \[
\begin{array}{c}
\text{key} \\
\text{size}
\end{array}
\]
Example of an OS-tree

\[ \text{size}[x] = \text{size}[\text{left}[x]] + \text{size}[\text{right}[x]] + 1 \]
Selection

Implementation trick: Use a sentinel (dummy record) for NIL such that size[NIL] = 0.

OS-SELECT(x, i) \succ \text{i\textsuperscript{th} smallest element in the subtree rooted at } x

\[ k \leftarrow \text{size[left[x]]} + 1 \succ k = \text{rank}(x) \]

if \( i = k \) then return \( x \)

if \( i < k \)

then return OS-SELECT(left[x], i)

else return OS-SELECT(right[x], i - k)

(OS-RANK is in the textbook.)
Example

**OS-SELECT**(root, 5)

![Red-black tree diagram]

Running time = $O(h) = O(\lg n)$ for red-black trees.

*Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson*
Q. Why not keep the ranks themselves in the nodes instead of subtree sizes?

A. They are hard to maintain when the red-black tree is modified.

**Modifying operations:** INSERT and DELETE.

**Strategy:** Update subtree sizes when inserting or deleting.
Example of insertion

**INSERT(“K”)**
Handling rebalancing

Don’t forget that RB-INSERT and RB-DELETE may also need to modify the red-black tree in order to maintain balance.

- **Recolorings**: no effect on subtree sizes.
- **Rotations**: fix up subtree sizes in $O(1)$ time.

**Example:**

$\therefore$ RB-INSERT and RB-DELETE still run in $O(\lg n)$ time.
Data-structure augmentation

**Methodology:** (e.g., order-statistics trees)

1. Choose an underlying data structure (red-black trees).
2. Determine additional information to be stored in the data structure (subtree sizes).
3. Verify that this information can be maintained for modifying operations (RB-INSERT, RB-DELETE — don’t forget rotations).
4. Develop new dynamic-set operations that use the information (OS-SELECT and OS-RANK).

These steps are guidelines, not rigid rules.
Interval trees

**Goal:** To maintain a dynamic set of intervals, such as time intervals.

\[ i = [7, 10] \]

\[ low[i] = 7 \quad 10 = high[i] \]

**Query:** For a given query interval \( i \), find an interval in the set that overlaps \( i \).
Following the methodology

1. *Choose an underlying data structure.*
   • Red-black tree keyed on low (left) endpoint.

2. *Determine additional information to be stored in the data structure.*
   • Store in each node \( x \) the largest value \( m[x] \) in the subtree rooted at \( x \), as well as the interval \( \text{int}[x] \) corresponding to the key.
Example interval tree

\[ m[x] = \max \left\{ \text{high}[\text{int}[x]], \text{m}[\text{left}[x]], \text{m}[\text{right}[x]] \right\} \]
Modifying operations

3. Verify that this information can be maintained for modifying operations.

- **INSERT**: Fix $m$’s on the way down.
- Rotations — Fixup = $O(1)$ time per rotation:

```
Total INSERT time = $O(\lg n)$; DELETE similar.
```

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson
New operations

4. Develop new dynamic-set operations that use the information.

INTERVAL-SEARCH\((i)\)

\[
x \leftarrow \text{root} \\
\text{while } x \neq \text{NIL} \text{ and } (\text{low}[i] > \text{high}[\text{int}[x]]) \text{ or } \text{low}[\text{int}[x]] > \text{high}[i]) \text{ do } \triangleright i \text{ and } \text{int}[x] \text{ don’t overlap} \\
\quad \text{if } \text{left}[x] \neq \text{NIL} \text{ and } \text{low}[i] \leq \text{m}[\text{left}[x]] \\
\qquad \text{then } x \leftarrow \text{left}[x] \\
\qquad \text{else } x \leftarrow \text{right}[x] \\
\text{return } x
\]
Example 1: \textsc{interval-search}([14,16])

\[
\begin{array}{c}
\text{x} \\
\hline
17,19 \\
\hline
23
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\hline
5,11 \\
\hline
18
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\hline
4,8 \\
\hline
8
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\hline
15,18 \\
\hline
18
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\hline
7,10 \\
\hline
10
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\hline
22,23 \\
\hline
23
\end{array}
\]

\[
\text{x} \leftarrow \text{root}
\]

[14,16] and [17,19] don’t overlap

\[
14 \leq 18 \Rightarrow x \leftarrow \text{left}[x]
\]
Example 1: \textsc{Interval-Search}([14,16])

\begin{itemize}
  \item [14,16] and [5,11] don’t overlap
  \item 14 > 8 \Rightarrow x \leftarrow \text{right}[x]
\end{itemize}
Example 1: \textsc{Interval-Search}(\([14,16]\))

\[14,16\] and \([15,18]\) overlap

\textbf{return} \([15,18]\)
Example 2: INTERVAL-SEARCH([12,14])

\[ x \leftarrow \text{root} \]

[12,14] and [17,19] don’t overlap

\[ 12 \leq 18 \Rightarrow x \leftarrow \text{left}[x] \]
Example 2: \textsc{Interval-Search}([12,14])

[12,14] and [5,11] don’t overlap

12 > 8 ⇒ x ← right[x]
Example 2: \textsc{Interval-Search}([12,14])

[12,14] and [15,18] don’t overlap
12 > 10 \Rightarrow x \leftarrow \text{right}[x]
Example 2: \textsc{Interval-Search}([12, 14])

\[ x = \text{NIL} \Rightarrow \text{no interval that overlaps } [12, 14] \text{ exists} \]
Analysis

Time = \( O(h) = O(lg \ n) \), since INTERVAL-SEARCH does constant work at each level as it follows a simple path down the tree.

List \emph{all} overlapping intervals:

- Search, list, delete, repeat.
- Insert them all again at the end.

Time = \( O(k \ lg \ n) \), where \( k \) is the total number of overlapping intervals.

This is an \emph{output-sensitive} bound.

Best algorithm to date: \( O(k + lg \ n) \).