CSE 5311

Lecture 11 Augmenting Data Structures

Song Jiang, Ph.D.
Department of Computer Science and Engineering
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: \(\text{key} \text{ size}\)
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 \( size[\text{NIL}] = 0 \).

\[
\text{OS-SELECT}(x, i) \quad \triangleright \quad \text{\(i\)th smallest element in the subtree rooted at} \ x \\
\]

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

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

if \( i < k \)

then return \( \text{OS-SELECT}(left[x], i) \)

else return \( \text{OS-SELECT}(right[x], i - k) \)

(OS-RANK is in the textbook.)
Example

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

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

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

\textbf{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.

\[ i = [7, 10] \]

\[ \text{low}[i] = 7 \quad \text{high}[i] = 10 \]

**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 $int[x]$ 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:

```
6,20  30
11,15

6,20  30
30
11,15
30
```

Total **INSERT** time = $O(\log n)$; **DELETE** similar.
New operations

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

INTERVAL-SEARCH(i)

\[
\begin{align*}
x & \leftarrow \text{root} \\
\text{while } x \neq \text{NIL} \text{ and } (\text{low}[i] > \text{high}[\text{int}[x]]) \\
& \quad \text{or } \text{low}[\text{int}[x]] > \text{high}[i]) \\
\text{do } i \text{ and } \text{int}[x] \text{ don’t overlap} \\
\text{if } \text{left}[x] \neq \text{NIL} \text{ and } \text{low}[i] \leq m[\text{left}[x]] \\
\text{then } x \leftarrow \text{left}[x] \\
\text{else } x \leftarrow \text{right}[x] \\
\text{return } x
\end{align*}
\]
Example 1: \textsc{Interval-Search}([14,16])

\begin{itemize}
  \item $x \leftarrow \text{root}$
  \item $[14,16]$ and $[17,19]$ don’t overlap
  \item $14 \leq 18 \Rightarrow x \leftarrow \text{left}[x]$
\end{itemize}
Example 1: \textsc{Interval-Search}([14,16])

[14,16] and [5,11] don’t overlap

14 > 8 \Rightarrow x \leftarrow \text{right}[x]
Example 1: \textsc{Interval-Search}([14,16])

\[ [14,16] \text{ and } [15,18] \text{ overlap} \]

\textbf{return} \quad [15,18]
Example 2: \textsc{Interval-Search}([12,14])

\begin{itemize}
\item $[12,14]$ and $[17,19]$ don't overlap
\item $12 \leq 18 \Rightarrow x \leftarrow \text{left}[x]$
\end{itemize}
Example 2: \textsc{interval-search}([12,14])

\[
\begin{array}{c}
\text{[12,14] and [5,11] don't overlap} \\
12 > 8 \Rightarrow x \leftarrow \text{right}[x]
\end{array}
\]
Example 2: \texttt{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])

\begin{itemize}
\item \(5, 11\) \(\text{⇒}\) \(17, 19\)
\item \(22, 23\)
\item \(4, 8\) \(\text{⇒}\) \(5, 11\)
\item \(15, 18\)
\item \(7, 10\)
\item \(x = \text{NIL} \implies \text{no interval that overlaps } [12, 14] \text{ exists}\)
\end{itemize}
Analysis

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

List \textit{all} overlapping intervals:
\begin{itemize}
  \item Search, list, delete, repeat.
  \item Insert them all again at the end.
\end{itemize}

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

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

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

**Theorem.** Let $L$ be the set of intervals in the left subtree of node $x$, and let $R$ be the set of intervals in $x$’s right subtree.

- If the search goes right, then
  \[
  \{ i' \in L : i' \text{ overlaps } i \} = \emptyset.
  \]

- If the search goes left, then
  \[
  \{ i' \in L : i' \text{ overlaps } i \} = \emptyset \\
  \Rightarrow \{ i' \in R : i' \text{ overlaps } i \} = \emptyset.
  \]

*In other words, it’s always safe to take only 1 of the 2 children: we’ll either find something, or nothing was to be found.*
Correctness proof

Proof. Suppose first that the search goes right.

- If \( \text{left}[x] = \text{NIL} \), then we’re done, since \( L = \emptyset \).
- Otherwise, the code dictates that we must have \( \text{low}[i] > m[\text{left}[x]] \). The value \( m[\text{left}[x]] \) corresponds to the high endpoint of some interval \( j \in L \), and no other interval in \( L \) can have a larger high endpoint than \( \text{high}[j] \).

\[
\begin{array}{c}
\cdots \quad j \quad \quad \quad \quad i \\
\text{high}[j] = m[\text{left}[x]] & \quad \quad \quad \quad \text{low}(i)
\end{array}
\]

- Therefore, \( \{i' \in L : i' \text{ overlaps } i \} = \emptyset \).
Proof (continued)

Suppose that the search goes left, and assume that
\[ \{ i' \in L : i' \text{ overlaps } i \} = \emptyset. \]

- Then, the code dictates that \( \text{low}[i] \leq \text{m[left][x]} = \text{high}[j] \) for some \( j \in L \).
- Since \( j \in L \), it does not overlap \( i \), and hence \( \text{high}[i] < \text{low}[j] \).
- But, the binary-search-tree property implies that for all \( i' \in R \), we have \( \text{low}[j] \leq \text{low}[i'] \).
- But then \( \{ i' \in R : i' \text{ overlaps } i \} = \emptyset. \)