LECTURE 23
Augmenting Data Structures
• Dynamic order statistics
• Methodology
• Interval trees
Review Question

- Recall last lectures were on 2-3 trees
  - “Special” search trees that guarantee $O(\log n)$ depth of the tree
  - Two kinds of nodes: “2-nodes” and “3-nodes”
- Question: Suppose we had “4-nodes”
  - What fields would a 4-node have?
  - What ranges would the keys of subtrees lie in?
- “B-trees” consist of nodes with between $t-1$ and $2t-1$ keys (where $t$ is some parameter depending on the application). See CLRS, Chapter 18.
Dynamic order statistics

Abstract Data Type:

Dynamic set that supports standard Dictionary operations:

- **INSERT, DELETE, FIND,**

as well as order statistic queries:

- **SELECT**\((i, S)\): returns the \(i\)th smallest element in the dynamic set \(S\).
- **RANK**\((x, S)\): returns the rank of \(x \in S\) in the sorted order of \(S\)’s elements.
First Attempts

• Keep data in binary search tree
  √ To Select $i^{th}$ largest element, just start from left-most node in tree and skip to its successor $i-1$ times
  √ Running time: $\Theta(n)$ time in worst case
• We will see a data structure that implements all operations in $O(\log n)$ time
Augmenting a data structure

**Design Technique:** Start from standard data structure, and *augment* it with additional info.

For order stats: Use a binary search tree for the set $S$, but keep subtree sizes in the nodes.

**Notation for nodes:**

```
key
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 \( \text{size}[\text{NIL}] = 0 \).

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

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

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

if \( i < k \)

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

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

(OS-RANK is in the textbook.)

Running time: \( O(\text{height}) \)
Example

**OS-SELECT** \((\text{root, 5})\)

Running time \(= O(h)\)
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 tree is modified.

Modifying operations: INSERT and DELETE.

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

\textbf{INSERT(“K”)}
Handling 2-3 trees

- A similar idea can be used in balanced search trees such as 2-3 trees.
  - Tricky part is maintaining sizes correctly during merging and splitting
  - Can be done because $size[x]$ can be computed from information stored at children of $x$.
  - Example: splitting an internal node (sizes in red)
Data-structure augmentation

Methodology: (e.g., order-statistics trees)
1. Choose an underlying data structure (2-3 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 (Insert, Delete).
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.

\[
low[i] = 7 \rightarrow_1 10 = high[i]
\]

**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 $int[x]$ 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.
   - (For 2-3 trees: recalculate \( m[x] \) for affected nodes when a split is performed)

Total INSERT time = \( O(\lg n) \); DELETE similar.
New operations

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

**INTERVAL-SEARCH(i)**

\[ x \leftarrow root \]

\[ \text{while } x \neq \text{NIL and } (\text{low}[i] > \text{high}[\text{int}[x]]) \]
\[ \text{or } \text{low}[\text{int}[x]] > \text{high}[i]) \]

\[ \text{do } i \text{ and } \text{int}[x] \text{ don’t overlap} \]

\[ \text{if } \text{left}[x] \neq \text{NIL and } \text{low}[i] \leq m[\text{left}[x]] \]
\[ \text{then } x \leftarrow \text{left}[x] \]
\[ \text{else } x \leftarrow \text{right}[x] \]

**return** \( x \)
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])

\[ [14,16] \text{ and } [5,11] \text{ don’t overlap} \]

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

\begin{itemize}
\item [14,16] and [15,18] overlap
\item return [15,18]
\end{itemize}
Example 2: \textsc{interval-search}([12,14])

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

[12,14] and [17,19] don’t overlap

\[ 12 \leq 18 \implies x \leftarrow \text{left}[x] \]
Example 2: `INTERVAL-SEARCH([12,14])`

[12,14] and [5,11] don’t overlap

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

\[ 5,11 \quad 17,19 \quad 22,23 \]
\[ 4,8 \quad 15,18 \quad 23 \]
\[ 8 \quad 18 \quad 23 \]

\[ 7,10 \quad x \quad \]  
\[ 10 \]

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

\[
\begin{array}{c}
5,11 \\
18
\end{array}
\quad
\begin{array}{c}
15,18 \\
18
\end{array}
\quad
\begin{array}{c}
22,23 \\
23
\end{array}
\quad
\begin{array}{c}
17,19 \\
23
\end{array}
\quad
\begin{array}{c}
4,8 \\
8
\end{array}
\quad
\begin{array}{c}
7,10 \\
10
\end{array}
\quad
x
\]

\[x = \text{NIL} \Rightarrow \text{no interval that overlaps } [12,14] \text{ exists}\]
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 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 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
  \[ \{ i' \in L : i' \text{ overlaps } i \} = \emptyset \quad \Rightarrow \quad \{ i' \in R : i' \text{ overlaps } i \} = \emptyset. \]

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 \( \text{low}[i] > \text{m}[\text{left}[x]] \). The value \( \text{m}[\text{left}[x]] \) corresponds to the high endpoint of some interval \( j \in L \), and no other interval in \( L \) can have a larger high endpoint than \( \text{high}[j] \).

\[
\begin{align*}
\ldots & \quad j \quad i \\
\text{high}[j] &= \text{m}[\text{left}[x]] \\
\text{low}(i) &
\end{align*}
\]

- 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 \( \text{low}[i] \leq m[\text{left}[x]] = \text{high}[j] \) 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 \( \{i' \in R : i' \text{ overlaps } i \} = \emptyset \).