Algorithms II, CS 502
Augmenting Data Structures

Ugur Dogrusoz
Computer Eng Dept, Bilkent Univ
Augmenting

Often times “textbook” data structures (DS) are sufficient

- need to modify for real-life usage of course (we rarely sort “just integers” but rather “objects based on a unique field which is an integer”)

Frequently, will suffice to augment a textbook DS by storing additional info in it

- to perform additional operations on the DS
Example: Dynamic order statistics

- Order statistic (OS) trees augment red-black trees:
  - Associate a size field with each node in the tree
  - $x\cdot\text{size}$ keeps size of subtree rooted at $x$ (including $x$)
  - $x\cdot\text{size} = x\cdot\text{left}\cdot\text{size} + x\cdot\text{right}\cdot\text{size} + 1$
Selecting $i$\textsuperscript{th} element

We can use this new field to select $i$\textsuperscript{th} element in $O(lg n)$ time

```
OS-SELECT(x, i)
1   r = x.left.size + 1
2   if i == r
3       return x
4   elseif i < r
5       return OS-SELECT(x.left, i)
6   else return OS-SELECT(x.right, i - r)
```
Selecting $i^{th}$ element

- $\text{OS-Select(T.root,17)}$
Selecting $i^{th}$ element

- $\text{OS-Select}(T.\text{root}, 17)$

$\text{rank}(T.\text{root}) = 12 + 1 = 13 < 17^{th}$
Selecting $i^{th}$ element

- OS-Select(x,4)
Selecting $i^{\text{th}}$ element

- OS-Select$(x, 4)$

```
rank = 5 + 1 = 6 > 4^{\text{th}}
```
Selecting $i^{th}$ element

- **OS-Select($x, 4$)**
Selecting $i^{th}$ element

- OS-Select$(x, 4)$

rank = 1 + 1 = 2 < $4^{th}$
Selecting $i^{th}$ element

- OS-Select($x$, 2)

$4^{th} - 2 = 2^{nd}$ in
Selecting $i^{th}$ element

- **OS-Select**($x, 2$)

  - rank = $1 + 1 = 2 = 2^{nd}$
  - Bingo! 17$^{th}$ smallest is 38
Calculating rank

We can use this new field to calculate rank in $O(\lg n)$ time

```
OS-RANK(T, x)
1  r = x.left.size + 1
2  y = x
3  while y \neq T.root
4    if y == y.p.right
5      r = r + y.p.left.size + 1
6    y = y.p
7  return r
```
Calculating rank

- **OS-Rank** \((T, y)\)

<table>
<thead>
<tr>
<th>iteration</th>
<th>(y . key)</th>
<th>(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>41</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>17</td>
</tr>
</tbody>
</table>
Maintaining subtree sizes: insertion

First phase: go down tree to insert new node as a child of an existing node
- Increment \( x.size \) for each node on simple path traversed from the root downward leaves

Second phase: go up tree changing colors and rotating to maintain tree properties
- Rotations only locally affect size attribute
- For LEFT-ROTATE\((T,x)\) add:
  - \( y.size = x.size \)
  - \( x.size = x.left.size + x.right.size + 1 \)
Maintaining subtree sizes: *insertion*

- Example update on rotations

- Additional work:
  - First phase: $O(lg n)$
  - Second phase: $O(1)$

- Overall $O(lg n)$ is preserved
Maintaining subtree sizes: deletion

- **First phase:** only operates on the search tree
  - Either removes one node $y$ from tree or moves upward it within the tree
  - Simply traverse a simple path from node $y$ up to root, decrementing size attribute of each node on the path
  - Additional cost of $O(\lg n)$

- **Second phase:** causes at most 3 rotations (no other structural changes)
  - Handle similar to insertion
  - Additional cost of $O(1)$

- Overall $O(\lg n)$ is preserved
How to augment a data structure

1. Choose an underlying data structure
   - Red-black trees

2. Determine additional information to maintain
   - Add the size attribute

3. Verify additional information can be maintained for basic modifying operations
   - insert and delete still in $O(\lg n)$

4. Develop new operations
   - $O(\lg n)$ operations OS-Select and OS-Rank
Example: Dynamic set of intervals

- Interval $[t_1, t_2]$ represents the set $\{t \in \mathbb{R} \mid t_1 \leq t \leq t_2\}$
- Any two intervals $i$ and $i'$ satisfy interval trichotomy:
  - $i$ and $i'$ overlap,
  - $i$ is to the left of $i'$ ($i.\text{high} < i.\text{low}$),
  - $i$ is to the right of $i'$ ($i.\text{high} < i.\text{low}$).
Interval trees

- A red-black tree that maintains a dynamic set of intervals, with each element $x$ containing an interval $x.int$ supporting
  - $\text{Interval-Insert}(T,x)$ adds element $x$, whose $int$ attribute contains an interval, to interval tree $T$
  - $\text{Interval-Delete}(T,x)$ removes element $x$ from interval tree $T$
  - $\text{Interval-Search}(T,i)$ returns a pointer to an element $x$ in interval tree $T$ such that $x.int$ overlaps interval $i$, $T.nil$ otherwise
Interval trees
Interval trees

1. Choose an underlying data structure
   - Red-black trees with key of node \( x \) being \( x.int.low \)

2. Determine additional information to maintain
   - \( x.max \): max value of any interval endpoint stored in subtree rooted at \( x \)

3. Verify additional information can be maintained for basic modifying operations
   - insert and delete still in \( O(lg n) \)
Interval trees

Theorem 14.1 (Augmenting a red-black tree):
If the value of an augmenting field $f$ for each node $x$ depends on only the information in nodes $x$, $x.left$, and $x.right$, then we can maintain values of $f$ in all nodes of $T$ during insertion and deletion without asymptotically affecting $O(lg n)$ performance of these operations.

Determine $\text{max}$ value of a node as follows:

$$x.max = \max(x.int.high, x.left.max, x.right.max)$$

In fact, rotations only take $O(1)$ additional time.
4. Develop new operations

- $O(lg n)$ operation $\text{Interval-Search}(T,i)$ returns a node in tree $T$ whose interval overlaps $i$; $T.nil$ otherwise.

```
$\text{Interval-Search}(T,i)$
1 $x = T.root$
2 while $x \neq T.nil$ and $i$ does not overlap $x.int$
3     if $x.left \neq T.nil$ and $x.left.max \geq i.low$
4         $x = x.left$
5     else $x = x.right$
6 return $x$
```
Interval trees

**Interval-Search**\( (T,i=[22,25]) \)
Interval trees

- **Interval-Search**($T, i=[22,25]$)
Interval trees

- \textbf{Interval-Search}(T, i=[22, 25])
**Interval trees**

- **Interval-Search** \((T, i=[11,14])\)
Interval trees

Theorem 14.2 (Interval-Search works correctly):
Any execution of \textsc{Interval-Search}\( (T,i) \) either returns a node whose interval overlaps \( i \), or it returns \( T.nil \) and the tree \( T \) contains no node whose interval overlaps \( i \).

Proof: \textit{Invariant}: If tree \( T \) contains an interval that overlaps \( i \), then the subtree rooted at \( x \) contains such an interval.
- Initialization (line 1), Maintenance (line 4 or 5), Termination (line 2)

```
\textsc{Interval-Search}(T, i)
1 \quad x = T.root
2 \quad \textbf{while} x \neq T.nil \textbf{ and } i \textbf{ does not overlap } x.\text{int} \\
3 \quad \quad \textbf{if} x.\text{left} \neq T.nil \textbf{ and } x.\text{left}.\text{max} \geq i.\text{low} \\
4 \quad \quad \quad x = x.\text{left} \\
5 \quad \quad \textbf{else} \quad x = x.\text{right} \\
6 \quad \textbf{return} \ x
```