Why to augment data structures?

Motivation

- Real situations require more than a "textbook" data structure!
Why to augment data structures?

Motivation

▶ Real situations require more than a "textbook" data structure!
▶ Create your own data structure to support more functionalities.
Why to augment data structures?

Motivation

▶ Real situations require more than a "textbook" data structure!
▶ Create your own data structure to support more functionalities.
▶ In rare situations, you need to create something totally new!
▶ Most of times, create something new out of our textbook :)
How to augment data structures?

- Know what you want to achieve!!
  - Choose an underlying data structure.
  - Determine additional information to maintain in the underlying data structure.
  - Verify that we can maintain the additional information for the basic modifying operations on the underlying data structure.
  - Develop new operations.
  - Alert: keep efficiency/complexity in mind all the time. Try simple implementation first!!
How to augment data structures?

Know what you want to achieve !!

- Choose an underlying data structure.
- Determine additional information to maintain in the underlying data structure.
How to augment data structures?

Know what you want to achieve !!

- Choose an underlying data structure.
- Determine additional information to maintain in the underlying data structure.
- Verify that we can maintain the additional information for the basic modifying operations on the underlying data structure.
How to augment data structures?

Know what you want to achieve !!

- Choose an underlying data structure.
- Determine additional information to maintain in the underlying data structure.
- Verify that we can maintain the additional information for the basic modifying operations on the underlying data structure.
- Develop new operations.

Alert: keep efficiency/complexity in mind all the time. try simple implementation first!!
How to augment data structures?

Know what you want to achieve !!

- Choose an underlying data structure.
- Determine additional information to maintain in the underlying data structure.
- Verify that we can maintain the additional information for the basic modifying operations on the underlying data structure.
- Develop new operations.

Alert: keep efficiency/complexity in mind all the time. try simple implementation first !!
Example: Order-Statistic Tree

Functionality

- Dynamic data structures: updates - insert, delete. $O(\log(n))$ time.
- Find any order statistic in $O(\log(n))$ time.
Example: Order-Statistic Tree

Functionality

- Dynamic data structures: updates - insert, delete. $O(\log(n))$ time.
- Find any order statistic in $O(\log(n))$ time.
- What is a good underlying data structure?
Example: Order-Statistic Tree

Functionality

- Dynamic data structures: updates - insert, delete. $O(\log(n))$ time.
- Find any order statistic in $O(\log(n))$ time.
- What is a good underlying data structure?
- What additional information do we need?
Figure 14.1 shows a data structure that can support fast order-statistic operations. An order-statistic tree $T$ is simply a red-black tree with additional information stored in each node. Besides the usual red-black tree attributes $x$: key, $x$: color, $x$: $p$, and $x$: left, and $x$: right in a node $x$, we have another attribute, $x$: size. This attribute contains the number of (internal) nodes in the subtree rooted at $x$ (including $x$ itself), that is, the size of the subtree. If we define the sentinel's size to be 0—that is, we set $T$: nil: size to be 0—then we have the identity $x$: size $= x$: left: size + $x$: right: size + 1.

Retrieving an element with a given rank

Before we show how to maintain this size information during insertion and deletion, let us examine the implementation of two order-statistic queries that use this additional information. We begin with an operation that retrieves an element with a given rank. The procedure $OSS$-ELECT. $x; i$ returns a pointer to the node containing the $i$th smallest key in the subtree rooted at $x$. To find the $i$th smallest key in an order-statistic tree $T$, we call $OSS$-SELECT. $T$: root; $i$.
OS-SELECT\( (x, i) \)
\begin{align*}
1 \quad & r = x.\text{left}.\text{size} + 1 \\
2 \quad & \text{if } i == r \\
3 \quad & \quad \text{return } x \\
4 \quad & \text{elseif } i < r \\
5 \quad & \quad \text{return } \text{OS-SELECT}(x.\text{left}, i) \\
6 \quad & \text{else return } \text{OS-SELECT}(x.\text{right}, i - r)
\end{align*}
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)

**Time:** $O(\log(n))$. Why?
Figure 14.1 shows a data structure that can support fast order-statistic operations. An order-statistic tree $T$ is simply a red-black tree with additional information stored in each node. Besides the usual red-black tree attributes $x$: key, $x$: color, $x$: parent, and $x$: left, and $x$: right in a node $x$, we have another attribute, $x$: size. This attribute contains the number of (internal) nodes in the subtree rooted at $x$ (including $x$ itself), that is, the size of the subtree. If we define the sentinel's size to be 0—that is, we set $T$: nil: size to be 0—then we have the identity

$$x: size = x: left: size + x: right: size + 1.$$ 

We do not require keys to be distinct in an order-statistic tree. (For example, the tree in Figure 14.1 has two keys with value 14 and two keys with value 21.) In the presence of equal keys, the above notion of rank is not well defined. We remove this ambiguity for an order-statistic tree by defining the rank of an element as the position at which it would be printed in an inorder walk of the tree. In Figure 14.1, for example, the key 14 stored in a black node has rank 5, and the key 14 stored in a red node has rank 6.

Retrieving an element with a given rank

Before we show how to maintain this size information during insertion and deletion, let us examine the implementation of two order-statistic queries that use this additional information. We begin with an operation that retrieves an element with a given rank. The procedure $\text{OSS-ELECT}.x; i/$ returns a pointer to the node containing the $i$th smallest key in the subtree rooted at $x$. To find the $i$th smallest key in an order-statistic tree $T$, we call $\text{OSS-ELECT}.T: root; i/$.

Find rank 15th element?
Order-Statistic Tree: Find the rank of a given element \( i \)

**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** \( r \)
Order-Statistic Tree: Find the rank of a given element $i$

**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** $r$

**Time:** $O(\log(n))$. Why?
Order-Statistic Tree: Find the rank of a given element $i$

Loop invariant
At the start of each iteration of the while loop, $r$ is the rank of $x.key$ in the subtree rooted at node $y$.

**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 $r$
Maintain subtree sizes

What are the problems?

- Recall Red-Black Trees updates.
- When do you need to change the size of subtrees?
Maintain subtree sizes

What are the problems?

- Recall Red-Black Trees updates.
- When do you need to change the size of subtrees?
- How does the size of subtrees change then?
Maintain subtree sizes

Left-Rotation \( (T, x) \)

\[
\begin{align*}
y \cdot size &= \hspace{1.25in} x \cdot size \\
x \cdot size &= \hspace{1.25in} x \cdot left \cdot size + x \cdot right \cdot size + 1
\end{align*}
\]
Augmenting Red-black trees

Theorem 14.1 (Augmenting a red-black tree)
Let $f$ be an attribute that augments a red-black tree $T$ of $n$ nodes, and suppose that the value of $f$ for each node $x$ depends on only the information in nodes $x$, $x.left$, and $x.right$, possibly including $x.left.f$ and $x.right.f$. Then, we can maintain the values of $f$ in all nodes of $T$ during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance of these operations.