COMP 3170 - Analysis of Algorithms & Data Structures

Shahin Kamali

Lecture 11 - Jan. 26, 2018
CLRS 14-1, 14-2
University of Manitoba
In practice, it often happens that you want an abstract data type to support additional queries.

To implement this, we need to augment the underlying data structure.

Augmentation often involves storing additional data which facilitates the query.
Augmented Data Structures

In practice, it often happens that you want an abstract data type to support additional queries.

- To implement this, we need to augment the underlying data structure.
- Augmentation often involves storing additional data which facilitates the query.

Consider AVL tree which supports search, insert, delete in $\Theta(\log n)$ time.

- What if your ‘boss’ asks you to additionally support minimum, maximum, rank, and select?
Augmented Data Structures

In practice, it often happens that you want an abstract data type to support additional queries.

- To implement this, we need to **augment** the underlying data structure.
- Augmentation often involves storing additional data which facilitates the query.

Consider AVL tree which supports search, insert, delete in $\Theta(\log n)$ time.

- What if your ‘boss’ asks you to **additionally** support minimum, maximum, rank, and select?
- Without augmentation, minimum and maximum take $\Theta(\log n)$ while rank and select require linear time (in-order traversal to retrieve the sorted list of keys).
- What if your angry boss wants them to be faster?
Augmented Data Structures

Augmenting AVL trees

- We can augment AVL trees to support *minimum*/*maximum* in $\Theta(1)$.
- Just add a pointer to the leftmost/rightmost leaf of the tree.
- After updating the tree by an insert/deleted, make sure that the pointer still points to the smallest/largest element.
Augmented Data Structures

Augmenting AVL trees

- After an insertion, update the minimum pointer
  - If the newly inserted key is less than minimum, update the minimum pointer to point to it (similar for maximum pointer).
Augmented Data Structures

Augmenting AVL trees

- After an insertion, update the minimum pointer
  - If the newly inserted key is less than minimum, update the minimum pointer to point to it (similar for maximum pointer).
Augmented AVL trees

- After an insertion, update the minimum pointer
  - If the newly inserted key is less than minimum, update the minimum pointer to point to it (similar for maximum pointer).
Augmented Data Structures

Augmenting AVL trees

- After an insertion, update the minimum pointer
  - If the newly inserted key is less than minimum, update the minimum pointer to point to it (similar for maximum pointer).

![Augmented AVL Tree Diagram](image-url)
Augmented Data Structures

Augmenting AVL trees

- After an insertion, update the minimum pointer
  - If the newly inserted key is less than minimum, update the minimum pointer to point to it (similar for maximum pointer).
  - It takes an additional time of $\Theta(1)$ (the insertion time is still $\Theta(\log n)$)
- Similar update for max pointer
Augmented Data Structures

**Augmenting AVL trees**

- After a deletion, update the maximum pointer
  - Check if the maximum element was deleted. If so, update the maximum pointer to the predecessor of the deleted element
  - Finding predecessor takes additional time of $\Theta(1)$
    - Let $x$ be the max element before deletion; there is nothing on the right of $x$.
    - The left subtree of $x$ has zero or one node (otherwise $x$ is unbalanced).
    - If there is an item $y$ on the left of $x$, then it is the successor of $x$
    - If $x$ is a leaf, then its parent is the successor

- Similar update for max pointer
Augmented Data Structures

Augmenting AVL trees

- After a deletion, update the maximum pointer
  - Check if the maximum element was deleted. If so, update the maximum pointer to the predecessor of the deleted element
  - Finding predecessor takes additional time of $\Theta(1)$
    - Let $x$ be the max element before deletion; there is nothing on the right of $x$.
    - The left subtree of $x$ has zero or one node (otherwise $x$ is unbalanced).
    - If there is an item $y$ on the left of $x$, then it is the successor of $x$.
    - If $x$ is a leaf, then its parent is the successor.

- Similar update for max pointer
Augmented Data Structures

Augmenting AVL trees

- After a deletion, update the maximum pointer
  - Check if the maximum element was deleted. If so, update the maximum pointer to the predecessor of the deleted element
  - Finding predecessor takes additional time of $\Theta(1)$
    - Let $x$ be the max element before deletion; there is nothing on the right of $x$.
    - The left subtree of $x$ has zero or one node (otherwise $x$ is unbalanced).
    - If there is an item $y$ on the left of $x$, then it is the successor of $x$
    - If $x$ is a leaf, then its parent is the successor

- Similar update for max pointer
Augmented Data Structures

Augmenting AVL trees

Theorem

We can augment AVL trees by adding only two pointers ($\Theta(1)$) extra space to support minimum/maximum queries in $\Theta(1)$ and without changing time complexity of other queries (insertion, deletion, and search).
Augmented Data Structures

Augmenting AVL trees

Can we augment AVL trees to support rank/select operations in $O(\log n)$ time?

- $\text{rank}(x)$ reports the index of key $x$ in the sorted array of keys
- $\text{select}(i)$ returns the key with index $i$ in the sorted array of keys
Augmented Data Structures

Augmenting AVL trees

Can we augment AVL trees to support rank/select operations in $O(\log n)$ time?

- $\text{rank}(x)$ reports the index of key $x$ in the sorted array of keys
- $\text{select}(i)$ returns the key with index $i$ in the sorted array of keys

Idea 1: Store the rank of each node at that node.

- $O(\log n)$ rank and select are guaranteed (why?)
- Is it a good augment data structure?
Augmented Data Structures

Augmenting AVL trees

Can we augment AVL trees to support rank/select operations in $O(\log n)$ time?

- $rank(x)$ reports the index of key $x$ in the sorted array of keys
- $select(i)$ returns the key with index $i$ in the sorted array of keys

Idea 1: Store the rank of each node at that node.

- $O(\log n)$ rank and select are guaranteed (why?)
- Is it a good augment data structure? No because inserting an item (e.g., key 1 here) might require updating all stored ranks
  Insertion/deletion take $\Theta(n)$. Failed!

```
2 1
 7
 3 5
 11
 1
```

```
2 4 6
 11
 24
 17
```

```
3 5 8
 13
 15
```

```
7 3 9
 12
```

```
1 4
 2
```

```
1 3
```

```
1 2
```

COMP 3170 - Analysis of Algorithms & Data Structures
Augmented Data Structures

Augmenting AVL trees

- Idea 2: At each node, store the size (no. of nodes) of the subtree rooted at that node
  - The size of a node is the sum of the sizes of its two subtrees plus 1.
  - The size of an empty subtree is 0.

- The rank of a node $x$ in its own subtree is the size of its left subtree.
  - E.g., rank of root 12 is 6

![Augmented AVL Tree Diagram]
Augmented Data Structures

Augmenting AVL trees

- Selection on an AVL tree augmented with size data is similar to quickselect, where the root acts as a pivot.

  Select(i): compare $i$ with the rank of the root $r$ (size of left subarray).

    - If equal, return the root $r$
    - if $i < \text{rank}(\text{root})$, recursively find the same index $i$ in the left subtree
    - if $i > \text{rank}(\text{root})$, recursively find index $i - \text{rank}(\text{root}) - 1$ in the right subtree
Augmented Data Structures

Augmenting AVL trees

- Selection on an AVL tree augmented with size data is similar to quickselect, where the root acts as a pivot.
- Select(i): compare $i$ with the rank of the root $r$ (size of left subarray).
  - If equal, return the root $r$
  - if $i < \text{rank(root)}$, recursively find the same index $i$ in the left subtree
  - if $i > \text{rank(root)}$, recursively find index $i - \text{rank(root)} - 1$ in the right subtree

E.g., select(x,5) $\rightarrow$ select(y,5) $\rightarrow$ select(z,2) $\rightarrow$ select(w,1) $\rightarrow$ w (11) is returned
Augmented Data Structures

Augmenting AVL trees

\[ \text{select } (\text{node}, i) \]

- If \( \text{node} = \emptyset \) then return error (\( i \) exceeds number of nodes).
  [Could have checked this at the root: if \( i \geq \text{node.size} \)]

- If \( i = \text{node.left.size} \) then return \( \text{node.key} \).

- If \( i < \text{node.left.size} \) then return \( \text{select}(\text{node.left}, i) \).

- If \( i > \text{node.left.size} \) then return \( \text{select}(\text{node.right}, i - \text{node.left.size} - 1) \).