CSC 263 Tutorial Week 4

Robin Swanson (robin@cs.Toronto.edu)

Tutorial Overview

- Augmented AVL Tree Questions
  - Finding dynamic order statistics
  - Finding the average
- AVL Augmentation Theorem (14.1 in CLRS)
- Binary Search Tree Questions

AVL Dynamic order statistics

- Dynamic order statistics: The *i*th order statistic is the element of a set with the *i*th smallest key.

- Rank: The position of an element in the linearly ordered set

- Gives rise to two operations:
  - Select(*r*): Given a rank *r*, what is the key with that rank?
  - Rank(*k*): Given a key *k*, what is its rank?

AVL Dynamic order statistics

- Naïve method: Inorder traverse entire tree until rank or element is found
- Will this affect runtime of SEARCH/INSERT/DELETE?
- What is the runtime for a query?

AVL Dynamic order statistics

- Better method: Maintain rank information for each element
  - i.e., each node contains a rank variable
- Will this affect runtime of SEARCH/INSERT/DELETE?
- What is the runtime for a query?

AVL Dynamic order statistics

- Best method: Maintain size information for each element
  - i.e., each node contains the size of the tree below it (including itself)
  - X.size = x.left.size + x.right.size + 1
- How is this related to rank?

AVL: RANK

- We can define rank(x) = 1 + # keys that came before x
- We can also say that the relative rank of x (its rank at the subtree rooted at x) = 1 + x.left.size

AVL: RANK

- We can define rank(x) = 1 + # keys that came before x
- We can also say that the relative rank of x (its rank at the subtree rooted at x) = 1 + x.left.size
- Rank(k):
  - Search for value *k*
  - As you search, keep track of the current rank
  - i.e., Each time you descend a level, add the size of the subtrees you skip
  - Return total calculated rank

AVL: SELECT

- Select(r):
  - x = root
  - Return SELECT(x, r)
- Select(x, k)
  - RR = x.relative_rank = 1 + x.left.size
  - If k < RR, return SELECT(x.left, k)
  - If k > RR, return SELECT(x.right, k- RR)
  - Else return x

Augmented AVL

- What changes need to be made to INSERT(x)?
- What changes need to be made to DELETE(x)?
- Does this affect the running time of INSERT or DELETE?

Augmented AVL

- **What changes need to be made to INSERT(x)?**

1. Increment size of subtree rooted at every node examined as we traverse the tree.
2. Update balancing of ancestor nodes.
3. If we need to rotate, recompute the size values from their new children.

Augmented AVL

- **What changes need to be made to DELETE(x)?**

1. Decrease size value from root to leaf node removed.
2. If we need to rotate, recompute size from their children.

Augmented AVL

- **How does this affect running time?**

1. Still traverse height of tree (O(log*n*))
2. But we still only do constant work at each level
3. Therefore we're still O(log*n*) time

AVL: Average

- Consider the operation AVERAGE(v) which returns the average value of all keys stored in the subtree rooted at node v. Explain how to augment an AVL tree to support this operation in O(1) time without affecting the O(log$n$) time complexity of INSERT, and DELETE.
  a) What extra information needs to be stored at each node?
  b) Explain how to perform AVERAGE in O(1) time using the extra information from part (a).
  c) Explain why SEARCH, INSERT, and DELETE can still be performed in O(log$n$) time.

AVL: Average

a) **What extra information needs to be stored at each node?**

We augment the nodes with two values: **sum** (the total sum of the keys in the subtree) and **size** (the number of items in the subtree). These are local properties that can be computed from the node and its direct children in O(1)

x.sum = x.left.sum + x.key + x.right.sum

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

AVL: Average

b) Explain how to perform AVERAGE in O(1) time using the extra information from part (a).

AVERAGE simply returns **root.sum/root.size** which takes O(1) time

AVL: Average

c) **Explain why SEARCH, INSERT, and DELETE can still be performed in O(log*n*) time.**

Since these are local properties that for each node can be computed in O(1) time from the values for the node and its children, we can use the theorem about augmentation of AVL trees that states the augmented properties can be maintained without changing the running time of the original AVL tree operations, i.e., O(log*n*) time.

Theorem 14.1

- Let *f* be an attribute that augments an AVL tree *T* of *n* nodes. Suppose that the value of *f* for each node *x* depends on only the information in nodes *x, x.left, 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 $O(\log n)$ performance of these operations.

Theorem 14.1

- Main idea: A change in attribute *f* in a node *x* propagates only to ancestors of *x* in the tree.
- i.e., Updating *x* may depend on its parent, and its parent parent, etc. but we always finish once we arrive at the root.
- During insertion and deletion we may rotate a constant number of nodes, but those will require updating at most O(log*n*) updates

BST: 12.2-6

- Consider a binary search tree *T* whose keys are distinct. Show that if the right subtree of a node *x* in *T* is empty and *x* has a successor *y*, then *y* is the lowest ancestor of *x* whose left child is also an ancestor of *x*. (Recall that every node is its own ancestor)

BST: 12.2-6

Consider a binary search tree *T* whose keys are distinct. Show that if the right subtree of a node *x* in *T* is empty and *x* has a successor *y*, then *y* is the lowest ancestor of *x* whose left child is also an ancestor of *x*. (Recall that every node is its own ancestor)

First, we establish that *y* must be an ancestor of *x*. If *y* weren't an ancestor of *x*, then let *z* denote the first common ancestor of *x* and *y*. By the BST property, $x < z < y$, so *y* cannot be the successor of *x*.

Next, observe that *y.left* must be an ancestor of *x* because if it weren't, then *y.right* would be an ancestor of *x*, implying that $x > y$. Finally, suppose that *y*is not the lowest ancestor of *x* whose left child is also an ancestor of *x*. Let *z* denote this lowest ancestor. Then *z* must be in the left subtree of *y*, which implies $z < y$, contradicting the fact that *y* is the successor of x.

BST: 12.2-7

- An alternative method of performing an inorder tree walk of an *n*-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making *n-1* calls to TREE-SUCCESSOR. Prove that this algorithm runs in Θ(n) time.

```
TREE-MINIMUM(x)
1.  While x.left != NIL
2.          x = x.left
3.  Return x
```

```
TREE-SUCCESSOR(x)
1.  If x.right != NIL
2.          return TREE-MINIMUM(x.right)
3.  y = x.p
4.  While y != NIL and x == y.right
5.          x = y
6.          y = y.p
7.  Return y
```

BST: 12.2-7

An alternative method of performing an inorder tree walk of an *n*-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making *n-1* calls to TREE-SUCCESSOR. Prove that this algorithm runs in Θ(n) time.

To show this bound on the runtime, we will show that using this procedure we traverse each edge twice. This will suffice because the number of edges in a tree is one less than the number of vertices.

Consider a vertex *x*. Then we have the edge between *x.p* and *x* gets used when successor is called on *x.p* and gets used again when it is called on the largest element in the subtree rooted at *x.* Since these are the only two times that the edge can be used, apart from the initial finding of the tree min. We have that the runtime is O(*n*). We trivially get the runtime is Ω(*n*) because that is the size of the input.

BST: 12.2-8

- Prove that no matter what node we start at in a height-*h* binary search tree, *k* successive calls to TREE-SUCCESSOR takes O(*k* + *h*) time.

BST: 12.2-8

Prove that no matter what node we start at in a height-*h* binary search tree, *k* successive calls to TREE-SUCCESSOR takes O(*k* + *h*) time.

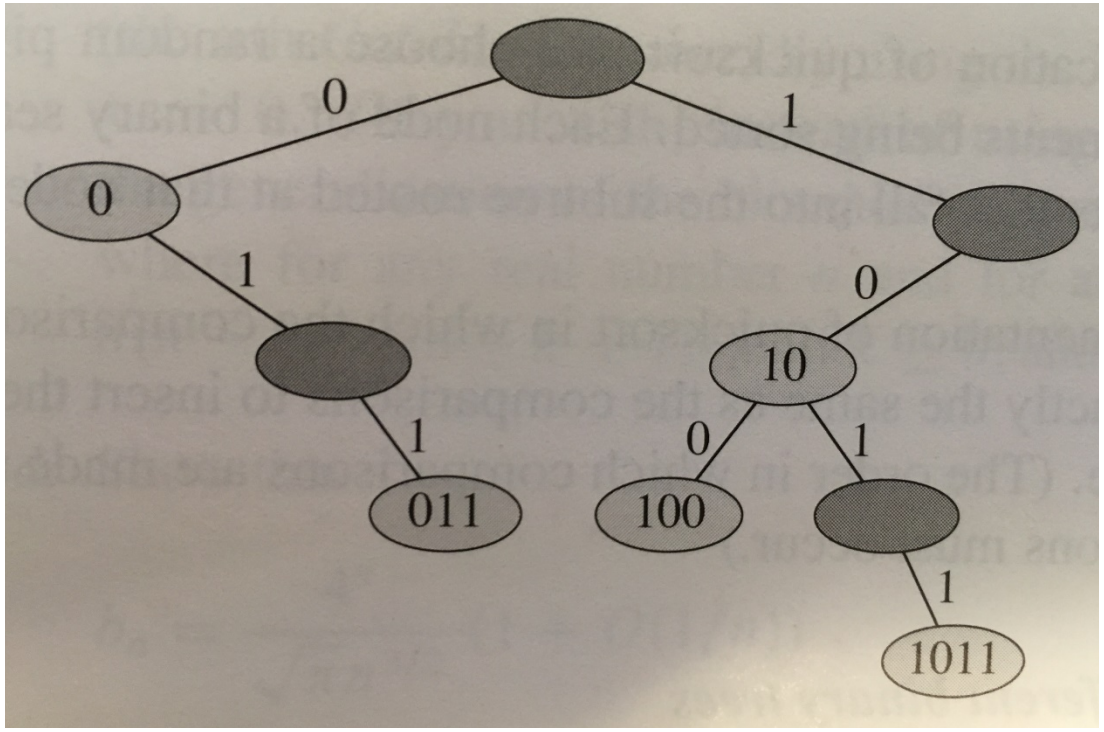
Let *x* be the node on which we have called TREE-SUCCESSOR and *y* be the *kth* successor of *x*. Let *z* be the lowest common ancestor of *z* and *y*. Successive calls will never traverse a single edge more than twice since TREE-SUCCESSOR acts like a tree traversal, so we will never examine a single vertex more than three times. Moreover, any vertex whose key value isn't between *x* and *y* will be examined at most once, and it will occur on a simple path from *x* to *z* or *y* to *z*. Since the lengths of these paths are bounded by *h*, the running time can be bounded by $3k+2h=O(k+h)$

Radix Sort: 12-2

- Given two strings $a=a\downarrow 0\ a\downarrow 1\ \ldots a\downarrow n$ and $b=b\downarrow 0\ b\downarrow 1\ \ldots b\downarrow n$, where each a_i and b_j is in some ordered set of characters, we say that string *a* is ***lexicographically less than*** string *b* if either:
  1. There exists an integer *j*, where 0 <= *j* <= *min(p, q)*, such that $a_i = b_i$ for all $i=0, 1, \ldots, j-1$ and $a_j < b_j$
  2. $p < q$ and $a_i = b_i$ for all $i=0, 1, \ldots, p$
- For example if *a* and *b* are bit strings, then 10100 < 10110 by rule 1 (letting j = 3) and 10100 < 101000 by rule 2. This ordering is similar to that used in English-language dictionaries.

Radix Sort: 12-2 (Continued …)

- The ***radix-tree*** shown here stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key *a* we go left at a node of depth *i* if $a_i = 0$ and right if $a_i = 1$. Let *S* be a set of distinct bit strings whose lengths sum to *n*. Show how to use a radix tree to sort *S* lexicographically in Θ(n) time. For the example here, the output of the sort should be the sequence 0, 011, 10, 100, 1011.



Radix Sort: 12-2

Show how to use a radix tree to sort *S* lexicographically in Θ(n) time.

The word at the root of the tree is necessarily before any word on its left or right subtree because it is both shorter, and the prefix of, every word in each of these trees. Moreover, every word in the left subtree comes before every word in the right subtree, so we need only perform a preorder traversal. This can be done recursively (see 12.1-4):

```
PREORDER-TREE-WALK(x)
    if x != NIL then:
            print(x)
            PREORDER-TREE-WALK(x.left)
            PREORDER-TREE-WALK(x.right)
    endif
```