Recitation 12 — Augmented Trees, Ordered Sets and Midterm Review

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

November 13, 2013

1 Announcements

- The second midterm is tomorrow. It will cover all of the material through last Thursday's lecture (which will be reviewed today.)
- After going over the last of the lecture material, we will spend the rest of today's recitation reviewing for the midterm.

2 Augmented Trees

Recall that an augmented tree is a balanced binary tree (such as a BST or treap) with a designated (associative) operation $f : \alpha \times \alpha \to \alpha$ and an operation `reduceVal`: $T \to \alpha$ which is equivalent to `reduce` f I_f T where I_f is an identity of f. However, unlike `reduce`, `reduceVal` takes $O(1)$ work and span. This is possible because we know the function ahead of time, and so whenever a tree is altered, we can efficiently compute the reduced value of the tree and store it at the root with no additional asymptotic cost over the operation that altered the tree. When the reduced value is requested, it can be returned without visiting any nodes other than the root.

Augmented trees can be implemented fairly easily. We modify the `Node` constructor to have another field for the reduced value. Instead of calling the constructor directly, the functions `split` and `join` (and any others that construct trees) should call the function `makeNode`, which computes the reduced value using the values already computed for the subtrees and constructs the node.

```
1  function makeNode(L, R, k, v) =
2    Node(L, R, k, v, f(reduceVal(L), f(v, reduceVal(R))))
```

`reduceVal` on a node simply returns the last field of the tuple, which contains the reduced value. `reduceVal` on a leaf would return the identity I_f. In either case, `reduceVal` takes $O(1)$ work. Assuming f is also $O(1)$, the work of `makeNode` is still constant and augmenting the trees doesn't change the complexity of the tree functions.

2.1 Example: Stock Market

Q: You're working as a consultant for the famous QADSAN market, which wants to support the following query: given a time range, return the maximum increase in stock value during that range (i.e. the largest difference between two trades in which the larger amount comes after the smaller amount). What keys, elements and function f would you use in an augmented tree to support such queries? (**Hint:** the reduced value, on which f operates, may include more information than the answer to the query.)

A: We want to query by time and get prices, so use time as the key and price as the value. When viewing a node, there are three possibilities: the maximum increase could be entirely in the left child, entirely in the right child, or span the two. To support all three cases, the reduced value needs to maintain the minimum and maximum (after the minimum) values and not just the increase. When we insert a value v, to make this work, we'll need to insert it as $(v, v, 0)$.

```
1  function f((a1, b1, i1), (a2, b2, i2)) =
2    (min(a1, a2), max(b1, b2),
3      max(i1, max(i2, b2 − a1)))
```

Q: Calling `reduceVal` and taking the difference between the max and min will only give the maximum increase across an entire (sub)tree. How can we make this query for a specific range (t_1, t_2) in $O(\lg n)$ work?

A: Split on t_1 and then split the right tree on t_2 to pull out a tree that has the right range, and call `reduceVal` on it.

3 Ordered Sets

Ordered sets are a data structure that make it easy to perform queries like the one above (get the subset of key/value pairs with keys in a given range.) The basic idea is that, as we saw last week, our implementation of sets contains additional information not present in the signature: an ordering on keys. Ordered sets take advantage of this information to implement additional features, listed in the table below (note that, with an ordering on keys, it now makes sense to expose the `split` and `join` operations as part of the ordered set ADT).

`last`(S)	:	$\mathbb{S} \to \mathbb{U}$	=	$\max\ S$
`first`(S)	:	$\mathbb{S} \to \mathbb{U}$	=	$\min\ S$
`split`(S,k)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S} \times bool \times \mathbb{S}$	=	$(\{k' \in S \mid k' < k\}, k \stackrel{?}{\in} S, \{k' \in S \mid k' > k\})$
`join`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cup S_2$, assuming $\max S_1 < \min S_2$
`getRange`(S, k_1, k_2)	:	$\mathbb{S} \times \mathbb{U} \times \mathbb{U} \to \mathbb{S}$	=	$\{k \in S \mid k_1 \leq k \leq k_2\}$
`rank`(S,k)	:	$\mathbb{S} \times \mathbb{U} \to int$	=	$\lvert\{k' \in S \mid k' < k\}\rvert$
`select`(S,i)	:	$\mathbb{S} \times int \to \mathbb{U}$	=	k such that $\lvert\{k' \in S \mid k' < k\}\rvert = i$
`splitIdx`(S,i)	:	$\mathbb{S} \times int \to \mathbb{S} \times \mathbb{S}$	=	$(\{k \in S \mid k <$ `select`$(S,i)\}$, $\{k \in S \mid k \geq$ `select`$(S,i)\})$

We saw above how to implement `getRange`. Implementing the last 3 functions efficiently requires a trick from augmented trees. Let's take a look at `rank`, which returns the numerical index of a key in a set according to the ordering. We can find the key in $O(\lg n)$ work, but, naively, finding the number of keys to the left of it would take $O(n)$ work, since we would need to count them all. The way around this is to store the size of each subtree at its root, so we can find the number of keys to the left of a given key by splitting on it and taking (now in $O(1)$) the size of the left tree. The code for `select` is in the lecture notes, and `splitIdx` is similar. As with augmented trees, trees must be constructed with a `makeNode` function that augments new nodes with the sum of the sizes of the two subtrees plus one.