Lecture 21 — Fun With Trees (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Nov 8, 2011

Today:
- Ordered Sets
- Augmenting Balanced Trees

1 Ordered Sets and Tables

The set and table interfaces described so far do not give any operations that make use of the ordering of the elements. This allows it to be defined on types that don't have a natural ordering. It is also well suited for an implementation based on hash tables. In many applications, however, it is useful to take advantage of the order of the keys. For example in a database one might want to find all the customers who spent between \$50 and \$100, all emails in the week of Aug 22, or the last stock transaction before noon on October 11th. Here we assume the data is organized by transaction value, date or any other ordered key.

For these purposes we can extend the operations on sets and tables with some additional operations that take advantage of ordering. Here we will just describe the operations on ordered sets. The operations on ordered tables are completely analogous.

Definition 1.1. For a totally ordered universe of elements $\mathbb{U}$ (e.g. the integers or strings), the *Ordered Set* abstract data type is a type $\mathbb{S}$ representing the powerset of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) along with the following functions:

all operations supported by the Set ADT, and				
$\texttt{last}(S)$	:	$\mathbb{S} \to \mathbb{U}$	=	$\max S$
$\texttt{first}(S)$	:	$\mathbb{S} \to \mathbb{U}$	=	$\min S$
$\texttt{split}(S, k)$	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S} \times \mathit{bool} \times \mathbb{S}$	=	as with trees
$\texttt{join}(S_1, S_2)$	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	as with trees
$\texttt{getRange}(S, k_1, k_2)$	:	$\mathbb{S} \times \mathbb{U} \times \mathbb{U} \to \mathbb{S}$	=	$\{k \in S \mid k_1 \leq k \leq k_2\}$

Note that *split* and *join* are the same as the operations we defined for binary search trees. Here, however, we are abstracting the notion to ordered sets.

If we implement using trees, then we can use the tree implementations of *split* and *join* directly. Implementing *first* is straightforward since it only requires traversing the tree down the left branches until a left branch is empty. Similarly *last* need only traverse right branches. The *getRange* operation can easily be implemented with two calls to *split*.

Here we consider a more involved example of the application of ordered sets. Recall that when we first discussed sets and tables we used an example of supporting an index for searching for documents based on the terms that appear in them. The motivation was to generate a search capability similar to what is supported by Bing, Google and other search engines. The interface we described was:

```
signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index
  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
end
```

We implemented this interface by using a table that mapped each word to the set of documents it appeared in. The *docList* type is therefore a *set* and the *index* type is a *set table*, where the sets are indexed by document and the table by words. We could then use *intersection*, *union*, and *difference* to implement *and*, *or* and *andNot* respectively. The interface and implementation make no use of any ordering of the documents.

Now lets say we want to augment the interface to support queries that restrict the search to certain domains, such as `cs.cmu.edu` or `cmu.edu` or even `.edu`. The function we want to add is

```
val inDomain : domain * docList -> docList
```

Given an index *idx* we could then do a search of the form

```
inDomain("cs.cmu.edu", and(find idx "cool", find idx "TAs"))
```

and it would return documents about all the cool TAs in CS.

Lets look at how to implement this with ordered sets. Our interface associates a *docId* with every document and let us assume these IDs are the URL of the document. Since URLs are strings, and strings have a total order, we can use an ordered set to implement the *docList* type. However instead of keeping the sets of URLs ordered "lexicographically", lets keep them ordered in backwards lexicographic order (i.e., the last character is the most significant). Given this order all documents within the same domain will be contiguous.

This suggests a simple implementation of *inDomain*. In particular we can use *getRange* to extract just the URLs in the *docList* that match the domain by "cutting out" the appropriate contiguous range. In particular we have:

```
1  fun inDomain(domain, L) =
2    getRange(L, domain, string.append(domain, "$"))
```

where \$ is a character greater than any character appearing in a URL. Note that this extracts precisely the documents that match the URL since strings between `domain` and `string.append(domain, "$")`) are precisely the strings that match the domain.

There are many other applications of ordered sets and tables including some which will be discussed in the following section.

2 Augmenting Balanced Trees

Often it is useful to include additional information beyond the key and associated value in a tree. In particular the additional information can help us efficiently implement additional operations. Here we will consider two examples: (1) locating positions within an ordered set or ordered table, and (2) keeping "reduced" values in an ordered or unordered table.

2.1 Tracking Sizes and Locating Positions

Lets say that we are using binary search trees (BSTs) to implement ordered sets and that in addition to the operations already described, we also want to efficiently support the following operations:

$$\begin{array}{lcll} \texttt{rank}(S,k) & : & \mathbb{S}\times\mathbb{U}\to int & = & |\{k'\in S \mid k'<k\}| \\ \texttt{select}(S,i) & : & \mathbb{S}\times int\to\mathbb{U} & = & k \text{ such that } |\{k'\in S \mid k'<k\}| = i \\ \texttt{splitIdx}(S,i) & : & \mathbb{S}\times int\to\mathbb{S}\times\mathbb{S} & = & (\{k\in S \mid k<\texttt{select}(S,i)\}, \\ & & & & \ \{k\in S \mid k\geq \texttt{select}(S,i)\}) \end{array}$$

In the previous lectures the only things we stored at the nodes of a tree were the left and right children, the key and value, and perhaps some balance information. With just this information implementing the `select` and `splitIdx` operations requires visiting all nodes before the i^{th} location to count them up. There is no way to know the size of a subtree without visiting it. Similarly `rank` requires visiting all nodes before k. Therefore all these operations will take $\Theta(|S|)$ work. In fact even implementing $\texttt{size}(S)$ requires $\Theta(|S|)$ work.

To fix this problem we can add to each node an additional field that specifies the size of the subtree. Clearly this makes the `size` operation fast, but what about the other operations? Well it turns out it allows us to implement `select`, `rank`, and `splitIdx` all in $O(d)$ work assuming the tree has depth d. Therefore for balanced trees the work will be $O(\log|S|)$. Lets consider how to implement `select`:

```
1  fun select(T, i) =
2    case expose(T) of
3      NONE ⇒ raise Range
4    | SOME(L, R, k) ⇒
5        case compare(i, |L|) of
6          LESS ⇒ select(L, i)
7        | EQUAL ⇒ k
8        | GREATER ⇒ select(R, i − size(L) − 1)
```

To implement `rank` we could simply do a split and then check the size of the left tree. The implementation of `splitIdx` is similar to split except when deciding which branch to take, be base it on the sizes instead of the keys. In fact with `splitIdx` we don't even need select, we could implement it as a `splitIdx` followed by a `first` on the right tree.

We can implement sequences using a balanced tree using this approach. In fact you already saw this in 15-150 when you covered the tree implementation of sequences.

2.2 Ordered Tables with Reduced Values

A second application of augmenting trees is to dynamically maintain a sum (using an arbitrary associative operator f) over the values of a table while allowing for arbitrary updates, e.g. insert, delete, merge, extract, split, and join. It turns out that this ability is extremely useful for several applications. We will get back to the applications but first describe the interface and its implementation using binary search tres. Consider the following abstract data type that extends ordered tables with one additional operation.

Definition 2.1. Consider an ordered table that maps keys of type k to values of type v, along with a function $f : v \times v \to v$ and identity element I_f. An *orderes table with reduced values* of type $\mathbb{T}$ supports all operations on ordered tables, e.g.

- empty, singleton, map, filter, reduce, iter, insert, delete, find, merge, extract, erase
- split, join, first, last, previous, next

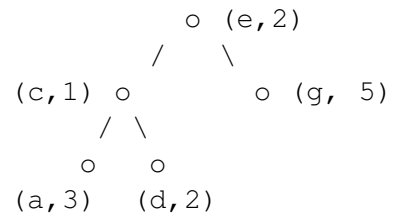
and in addition support the operation

$$\texttt{reduceVal}(T) : T \to v = \texttt{reduce}\ f\ I\ T$$

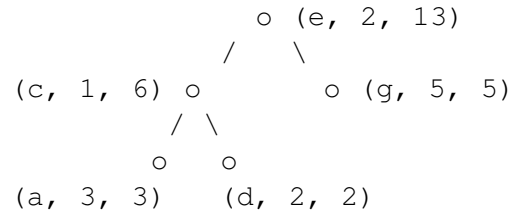
The $\texttt{reduceVal}(T)$ function just returns the sum of all values in T using the associative operator f that is part of the data type. It might seem redundant to support this function since it can be implemented by the existing `reduce` function, but the idea is that we will be able to implement it much more efficiently by including it in the interface. In particular our goal is to support all the table operations in the same asymptotic bounds as we have previously used for the binary search tree implementation of ordered tables, but also support the `reduceVal` in $O(1)$ work.

You might ask how can we possibly implement a `reduce` in $O(1)$ work on a table of arbitrary size? The trick is to make use of the fact that the function f over which we are reducing is fixed ahead of time. This means that we can maintain the reduced value and update it whenever we do other operations on the ordered table. That way whenever we ask for the value it has already been computed and we only have to look it up. Using this approach the challenge is not in computing `reduceVal`, it just needs to return a precomputed value, but instead how to maintain this value whenever we do other operations on the table.

We now discuss how to maintain the reduced values using binary search trees. The basic idea is simply to store with every node n of a binary search tree the reduced value of its subtree (i.e. the sum of all the values that are descendants of n as well as the value at n itself. For example lets assume our function f is addition, and we are given the following BST mapping character keys to integer values:



Now when we associate the reduced with each node we get



Note that the value at each node can simply be calculated by summing the reduced value in each of the two children and the value in the node. So, for example, the reduced value at the root is the sum of the left reduced value 6, the right reduced value 5, and the node value 2, giving 13. This means that we can maintain these reduced values by simply taking this "sum" of three values whenever creating a node. In turn this means that the only real change we have to make to existing code for implementing ordered tables with binary search trees is to do this sum whenever we make a node. If the work of f is constant, then this sum takes constant work.

The code to extend treaps with this idea is the following:

```
1  datatype Treap = Leaf | Node of (Treap × Treap × key × data × data)
2  fun reduceVal(T) =
3    case T of
4      Leaf ⇒ Reduce.I
5    | Node(_, _, _, _, r) ⇒ r
6  fun makeNode(L, R, k, v) =
7    Node(L, R, k, v, Reduce.f(reduceVal(L), Reduce.f(v, reduceVal(R))))
8  fun join′(T1, T2) =
9    case (T1, T2) of
10     (Leaf, _) ⇒ T2
11   | (_, Leaf) ⇒ T1
12   | (Node(L1, R1, k1, v1, s1), Node(L2, R2, k2, v2, s2)) ⇒
13       if (priority(k1) > priority(k2)) then
14         makeNode(L1, join′(R1, T2), k1, v1)
15       else
16         makeNode(join′(T1, L2), R2, k2, v2)
```

Note that the only difference in the `join'` code is the use of `makeNode` instead of `Node`. Similar use of `makeNode` can be used in `split` and other operations on treaps. This idea can be used with any

binary search tree, not just treaps. In all cases one needs only to replace the function that creates a node with a version that also sums the reduced values from the children and the value from the node to create a reduced value for the new node.

We note that in an imperative implementation of binary search trees in which a child node can be side affected, then the reduced values need to be recomputed on all nodes in the path from the modified node to the root.

We now consider several applications of ordered tables with reduced values.

2.2.1 Analyzing Profits at Tramlaw

Lets say that based on your expertise in algorithms you are hired as a consultant by the giant retailer Tramlaw™. Tramlaw sells over 10 billion items per year across its 6000+ stores. As with all retailers, it keeps track of every sale it makes and analyzes these sales regularly to make business and marketing decisions. Lets say that the sale records it keeps consists of a timestamp when the sale was made, the amount of the sale and various other information associated with the sale.

Tramlaw wants to be able to quickly determine the total amount of sales within any range of time, e.g. between 5pm and 10pm last friday, or during the whole month of September, or during the halftime break of the last Steeler's football game, or for the week after its latest promo. It uses this sort of information, for example, to decide on staffing levels or marketing strategy. It needs to maintain the database so that it can be updated quickly, including not just adding new sales to the end, but merging in information from all its stores, and possibly modifying old data (e.g. if some item is returned or a record is found to be in error).

How would you do this? Well after thinking a few minutes you remember ordered tables with reduced values from 210. You realize that if you keep the sales information keyed based on timestamps, and maintain the sum of sales amounts as the reduced values then all the operations required are cheap. In particular the function f is simply addition. Now the following will extract the sum in any range:

$$\texttt{reduceVal}(\texttt{getRange}(T, t_1, t_2))$$

This will take $O(\log n)$ work, which is much cheaper than n. Now lets say Tramlaw wanted to do a somewhat more complicated query where they want to know the total sales between 5 and 7 pm on every day over the past year. You could do this by applying the query above once for each day. These can all be done in parallel and summed in parallel. The total work will be $365 \times O(\log n)$, which is still much cheaper than looking at all data over the past year.

2.2.2 Working for Qadsan

Now in your next consulting job Qadsan hires you to more efficiently support queries on the their stock exchange data. For each stock they keep track of the time and amount of every trade. This data is updated as new trades are made. As with Tramlaw tables might also need to be merged since they might come from different sources : e.g. the Tokyo stock exchange and the New York stock exchange. They want to efficiently support queries that return the maximum price of a trade during any time range (t_1, t_2) .

Well you can use an ordered table with reduced values but instead of using addition for f, you would use $\max$. The query now is exactly the same as with your consulting jig with Tramlaw and they will similarly run in $O(\log n)$ work.

Exercise 1. *Now lets say that Qadsan also wants to support queries that 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 function would you use for f to support such queries in $O(\log n)$ work.*

2.2.3 Interval Queries

After your two consulting jobs, you are taking 15-451, with professor Mulb. On a test he asks you to support an abstract data type iTable that supports the following operations on intervals where an interval is a region on the real number line starting at x_l and ending at x_r.

insert(T, I)	:	iTable * (Real * Real) $\rightarrow$ iTable	insert interval I into table T
delete(T, I)	:	iTable * (Real * Real) $\rightarrow$ iTable	delete interval I from table T
count(T, x)	:	iTable * Real $\rightarrow$ int	return the number of intervals crossing x in T

Exercise 2. *How would you implement this.*

2.2.4 Others

- By using the parenthesis matching code in recitation 1 we could maintain whether a sequence of parenthesis are matched while allowing updates. Each update will take $O(logn)$ work and verify whether updated.
- By using the maximum contiguous subsequence sum problem described in class you could maintain the sum while updating the sequence by for example inserting new elements, deleting elements, or even merging sequences.