Chapter 16

Ordered Sets and Augmented BSTs

16.1 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. Abstract Data Type 16.1 defines the operations supported by ordered sets, which simply extend the operations on sets. The operations on **ordered tables** are completely analogous and extend the operations on tables. Note that `split` and `join` are the same as the operations we defined for binary search trees, but `join` does not take a middle element.

Abstract Data Type 16.1 (Ordered Sets). *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 (6.1), as well as*

$$
\begin{array}{lcll}
\texttt{first}(S) & : & \mathbb{S} \to (\mathbb{U} \cup \{\bot\}) & = \min S \\
\texttt{last}(S) & : & \mathbb{S} \to (\mathbb{U} \cup \{\bot\}) & = \max S \\
\texttt{previous}(S, k) & : & \mathbb{S} \times \mathbb{U} \to (\mathbb{U} \cup \{\bot\}) & = \max \{k' \in S \mid k' < k\} \\
\texttt{next}(S, k) & : & \mathbb{S} \times \mathbb{U} \to (\mathbb{U} \cup \{\bot\}) & = \min \{k' \in S \mid k' > k\} \\
\texttt{split}(S, k) & : & \mathbb{S} \times \mathbb{U} \to \mathbb{S} \times \mathbb{B} \times \mathbb{S} & \\
 & & \quad = (\{k' \in S \mid k' < k\}, k \overset{?}{\in} S, \{k' \in S \mid k' > k\}) & \\
\texttt{join}(S_1, S_2) & : & \mathbb{S} \times \mathbb{S} \to \mathbb{S} & = S_1 \cup S_2, \\
 & & & \textit{assuming } \max S_1 < \min S_2 \\
\texttt{getRange}(S, k_1, k_2) & : & \mathbb{S} \times \mathbb{U} \times \mathbb{U} \to \mathbb{S} & = \{k \in S \mid k_1 \leq k \leq k_2\} \\
\texttt{rank}(S, k) & : & \mathbb{S} \times \mathbb{U} \to \mathbb{N} & = |\{k' \in S \mid k' < k\}| \\
\texttt{select}(S, i) & : & \mathbb{S} \times \mathbb{N} \to (\mathbb{U} \cup \{\bot\}) & = k \in S \textit{ such that } \texttt{rank}(S, k) = i \\
 & & & \textit{or } \bot \textit{ if there is no such } k \\
\texttt{splitIdx}(S, i) & : & \mathbb{S} \times \mathbb{N} \to \mathbb{S} \times \mathbb{S} & = (\{k \in S \mid k < \texttt{select}(S, i)\}, \\
 & & & \quad \{k \in S \mid k \geq \texttt{select}(S, i)\})
\end{array}
$$

We assume max *or* min *of the empty set returns the special element $\bot$. $\mathbb{B}$ indicates a Boolean, and $\mathbb{N}$ indicates the natural numbers.*

Example 16.2. *Consider the following sequence ordered lexicographically:*

$$S = \{\text{"artie"}, \text{"burt"}, \text{"finn"}, \text{"mercedes"}, \text{"mike"}, \text{"rachel"}, \text{"sam"}, \text{"tina"}\}$$

- $\texttt{first}(S)$ *returns* $\texttt{SOME}(\text{"artie"})$
- $\texttt{next}(S, \text{"quinn"})$ *or* $\texttt{next}(S, \text{"mike"})$ *returns* $\texttt{SOME}(\text{"rachel"})$
- $\texttt{range}(S, \text{"blain"}, \text{"quinn"})$ *or* $\texttt{range}(S, \text{"burt"}, \text{"mike"})$ *returns*

  $$\{\text{"burt"}, \text{"finn"}, \text{"mercedes"}, \text{"mike"}\}$$

- $\texttt{rank}(S, \text{"rachel"})$ *or* $\texttt{rank}(S, \text{"quinn"})$ *returns the number of elements less than "rachel" or "quinn", which is* 5 *in both cases,*
- $\texttt{select}(S, 6)$ *returns the* 6^{th} *element (*0 *based) giving "sam", and*
- $\texttt{splitIndex}(S, 3)$ *splits S at location* 3 *returning*

  $$(\{\text{"artie"}, \text{"burt"}, \text{"finn"}\}, \{\text{"mercedes"}, \text{"mike"}, \text{"rachel"}, \text{"sam"}, \text{"tina"}\})$$

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`.

Exercise 16.3. *Describe how to implement `previous` and `next` using the other ordered set functions.*

We describe how to implement `rank`, `select` and `splitIndex` in the next section.

Cost Specification 16.4 (Tree-based ordered sets and tables). *The cost for the ordered set and ordered table functions is the same as for tree-based sets (Cost Spec 6.11) and tables (Cost Spec 6.20) for the operations supported by sets and tables. The work and span for all the operations in Data Type 16.1 is $O(\log n)$, where n is the size of the input set or table, or in the case of `join` it is the sum of the sizes of the two inputs.*

16.2 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.

16.2.1 Tracking Sizes and Locating Positions

We now consider how to efficiently implement the `rank`, `select` and `splitIndex` on a binary search tree. In Chapter 15 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 `size`(S) requires $\Theta(|S|)$ work.

Question 16.5. *How do we avoid visiting most of the tree to implement `rank`, `select`, and `splitIdx`?*

To avoid visiting all subtrees to calculate their size we can add to each node an additional field that gives the size of the subtree.

Algorithm 16.7 (Select)**.**

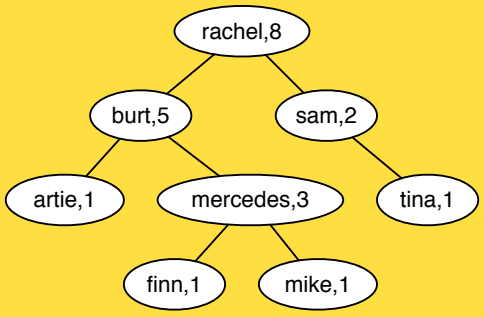
```
 1 function select(T, i) =
 2   case expose(T) of
 3     NONE ⇒ raise Range
 4   | SOME(L, k, R) ⇒
 5     case compare(i, |L|) of
 6       Less ⇒ select(L, i)
 7     | Equal ⇒ k
 8     | Greater ⇒ select(R, i − |L| − 1)

 9 function rank(T, k) =
10   case expose(T)
11     NONE ⇒ 0
12   | SOME(L, k′, R) ⇒
13     case compare(k, k′) of
14       Less ⇒ rank(L, k)
15     | Equal ⇒ |L|
16     | Greater ⇒ |L| + 1+ rank(R, k)
```

Example 16.6. *An example BST for the ordered set*

$$S = \{\text{"artie"}, \text{"burt"}, \text{"finn"}, \text{"mercedes"}, \text{"mike"}, \text{"rachel"}, \text{"sam"}, \text{"tina"}\}$$

augmented with the sizes of subtrees.



Storing the sizes of subtrees makes the `size` operation trivial, since we can just look it up at the root. It also makes `rank`, `select`, and `splitidx` efficient. In particular all three operations can be implemented in work proportional to the depth of the tree. Therefore for balanced trees the work will be $O(\log |S|)$. The algorithms for `select` and `rank` are given in Algorithm 16.7.

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.

16.2.2 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 very useful for several applications. We will get back to the applications but first describe the interface and its implementation using binary search trees. Consider the following abstract data type that extends ordered tables with one additional operation.

Definition 16.8 (Ordered table with reduced value ADT)**.** *For a specified associative function $f : v \times v \to v$ and identity element I_f, an* ordered table with reduced values *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, splitIdx, rank, select*

and in addition supports the operation

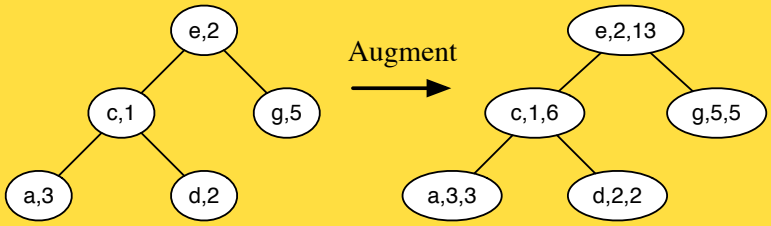
$$\texttt{reduceVal}(A) : \mathbb{T} \to v = \texttt{reduce}\ f\ I_f\ \ A$$

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 on the binary search trees. The basic idea is simply to store with every node v of a binary search tree the reduced value of its subtree (i.e. the sum of all the values that are descendants of v as well as the value at v itself).

Example 16.9. *The following diagram shows a tree with key-value pairs on the left, and the augmented tree on the right, where each node additionally maintains the sum (the function f is addition) of it subtree.*

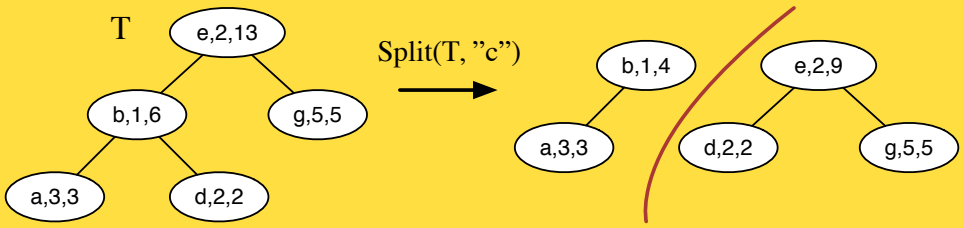


The sum at the root (13) is the sum of all values in the tree ($3+1+2+2+5$). It is also the sum of the reduced values of its two children (6 and 5) and its own value 2.

The value of each reduced value in a tree can be calculated as the sum of its two children plus the value stored at the node. 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 at each node.

Figure 16.10 describes the implementation of `join` on treaps extended to work with reduced values. In each `Node` the first `data` entry is the data stored at the node and the second is the reduced value. The only difference in the `join` code from Chapter 15 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.

Example 16.11. *The following diagram shows an example of splitting an augmented tree.*



The tree is split by the key c, and the reduced values on the internal nodes need to be updated. This only needs to happen along the path that created the split, which in this case is e, b, and d. The node for d does not have to be updated since it is a leaf. The `makeNode` *for e and b are what will update the reduced values for those nodes.*

Data Structure 16.10 (Treaps with reduced values)**.**

```
1  type rval                                  % type of the reduced value
2  val f : rval × rval → rval                 % associative combining function
3  val I_f : rval                             % identify for the reduced value
4  datatype Treap =
5      Leaf
6    | Node of (Treap × (key × rval × rval) × Treap)
7  function reduceVal(T) =
8    case T of
9      Leaf ⇒ I_f
10   | Node(_,(_,_,r),_) ⇒ r
11 function makeNode(L,(k,v),R) =
12   node(L,
13        (k, v, f(reduceVal(L), f(v, reduceVal(R)))),
14        R)
15 function join(T_1,T_2) =
16   case(T_1,T_2) of
17     (Leaf, _) ⇒ T_2
18   | (_, Leaf) ⇒ T_1
19   | (Node(L_1,(k_1,v_1,s_1),R_1),Node(L_2,(k_2,v_2,s_2),R_2)) ⇒
20       if (priority(k_1) > priority(k_2)) then
21         makeNode(L_1,(k_1,v_1),join(R_1,T_2))
22       else
23         makeNode(join(T_1,L_2),(k_2,v_2),R_2)
```

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.

Example: Analyzing Profits at TRAM⋆LAW®

Lets say that based on your expertise in algorithms you are hired as a consultant by the giant retailer **TRAM⋆LAW®**. Tramlaw sells over 10 billion items per year across its 8000+ stores. As with all major 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.

Example: A Jig with 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. Qasdan wants to efficiently support queries that return the maximum price of a trade during any time range (t_1, t_2) .

You tell them that they can use an ordered table with reduced values using $\max$ as the combining function f. The query will be exactly the same as with your consulting jig with Tramlaw, `getRange` followed by `reduceVal`, and it will similarly run in $O(\log n)$ work.

Exercise 16.12. *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.*

Example: Interval Queries

After your two consulting jobs, you are taking 15-451, with Professor Mulb. On a test he asks you to describe a data structure for representing an abstract data type called interval tables. An

interval is a region on the real number line starting at x_l and ending at x_r, and an interval table supports the following operations on intervals:

$\texttt{insert}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \to \mathbb{T}$	*insert interval I into table A*
$\texttt{delete}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \to \mathbb{T}$	*delete interval I from table A*
$\texttt{count}(A, x)$	:	$\mathbb{T} \times \texttt{real} \to \texttt{int}$	*return the number of intervals crossing x in A*

Exercise 16.13. *How would you implement this.*

Other Applications of Reduced Values

- 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 $O(\log n)$ work. In fact we could append two strings of parenthesis and check whether together they are matched in $O(\log n)$ work.
- 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.

.