Chapter 11

Sets and Tables

> "A *set* is a gathering together into a whole of definite, distinct objects of our perception or of our thought—which are called *elements* of the set."
>
> Georg Cantor,
> from "Contributions to the founding of the theory of transfinite numbers."

Set theory, founded by Georg Cantor in the second half of the nineteenth century, is one of the most important advances in Mathematics. From it came the notions of countably vs. uncountably infinite sets, and ultimately the theory of computational undecidability, i.e. that computational mechanisms such as the λ-calculus or Turing Machine cannot compute all functions. Set theory has also formed the foundations on which other branches of mathematics can be formalized. Early set theory, sometimes referred to as naïve set theory, allowed anything to be in a set. This lead to several paradoxes such Russell's famous paradox:

$$\text{let } R = \{x \mid x \notin x\}\,, \text{ then } R \in R \iff R \notin R\,.$$

Such paradoxes were resolved by the development of axiomatic set theory. Typically in such a theory, the universe of possible elements of a set needs to be built up from primitive notions, such as the integers, or reals, using various compositions rules, such as Cartesian products.

Our goals in this chapter are much more modest than trying to understand set theory and its many interesting implications. Here we simply recognize that in algorithm design sets are a very useful data type in their own right, but also in building up more complicated data types, such as graphs. Furthermore particular classes of sets, such as mappings, are themselves very useful, and hence deserve their own interface. In this book we refer to mappings as tables. In this chapter, we define the interface for sets and tables, specify their cost, and present some examples.

Other chapters cover data structures on binary search trees (Chapter 10) and hashing (Chapter **??**) that can be used for implementing the sets and tables interfaces. We also consider many applications of sets and tables. For example, sequences (Chapter 5) are a particular type of

table—one where the range of the tables are the integers from 0 to $n-1$. Other examples include graphs (Chapter 12).

11.1 Sets: Interface and Cost

Our ADT for sets is defined in ADT 11.1. It includes many of the functions one would expect on sets, such as insertion, intersection and size. It also includes functions to convert to and from another data type, sequences. As we will see, some of these functions should be considered primitives, others can be defined efficiently in terms of those primitives.

There are a few things to note about the interface. Firstly the sets are all subsets of some universe U, itself a set. If the universes are built up in some reasonable way this prevents paradoxes such as Russel's paradox. We also note that the elements of U must support the notion of equality. This might seem like a either a philosophical issue or perhaps an non-issue, but it has a very practical computational consequence. In particular, to implement a Set ADT, at minimum we need to supply an equality test for the elements of U. If U are the integers, then equality is probably integer equality, but in general equality could be more complicated. For implementing the Set ADT correctly equality on U is necessary and sufficient, but implementing sets efficiently requires some additional operations on U beyond equality. In particular an implementation based on balanced search trees requires a total ordering on the elements of U and hence a comparison. An implementation based on hashing does not require comparison, but requires the ability to hash the elements of U. These operations, equality, comparison, and hashing, are implicitly in the universe U and hence type $\mathbb{S}$. We note that when we defined sequences we did not need equality or any other operations on the elements of the sequence. This is because we need know nothing about the elements.

A second aspect to notice about the Set ADT is that although the universe U is potentially infinite (e.g. the integers), $\mathbb{S}$ only consists of finite sized subsets. Unfortunately this restriction means that the interface is not as powerful as general set theory, but it makes computation on sets feasible. A consequence of this requirement is that the interface does not include a function that takes the compliment of a set—such a function would generate an infinite sized set from a finite sized set (assuming the size of U is infinite).

Exercise 11.2. *Convince yourself there is no way to create an infinite sized set using the interface and with finite work.*

Finally, note that the definition of `toSeq` needs to place a total order on the elements of the set S. Such an order is ambiguous since there is no inherent ordering of the elements of a set. This means that `toSeq` is non-deterministic—it could possibly return different orderings in different implementations, or possibly even on different runs of the same implementation. We note that with `toSeq` we can easily implement `iter` and `reduce` on sets by just first converting them to a sequence and then applying the sequence versions of those functions.

Abstract Data Type 11.1 (Sets). *For a universe of elements $\mathbb{U}$ (e.g. the integers or strings), the* SET *abstract data type is a type $\mathbb{S}$ representing the power set of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) limited to sets of finite size, along with the functions below. In the specification, $\mathbb{N}$ is the natural numbers (non-negative integers) and $\mathbb{B} = \{\mathtt{T}, \mathtt{F}\}$; for a A of type $\mathbb{S}$ $[|A|]$ denotes the (mathematical) set of keys in the tree.*

`empty`	:	$\mathbb{S}$
`empty`	=	A *where* $[|A|] = \emptyset$
`size`	:	$\mathbb{S} \to \mathbb{N}$
`size(A)`	=	$\lvert A \rvert$
`singleton`	:	$\mathbb{U} \to \mathbb{S}$
`singleton`(e)	=	A *where* $[|A|] = \{e\}$
`filter`	:	$((\mathbb{U} \to \mathbb{B}) \times \mathbb{S}) \to \mathbb{S}$
`filter`(f, A)	=	B *where* $[|B|] = \{x \in A \mid f(x)\}$
`find`	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{B}$
`find`(A, e)	=	`true` *if and only if* $e \in [|A|]$
`insert`	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$
`insert`(A, e)	=	B *where* $[|B|] = [|A|] \cup \{e\}$
`delete`	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$
`delete`(A, e)	=	$A \setminus \{e\}$
`intersection`	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$
`intersection`(A_1, A_2)	=	B *where* $[|B|] = [|A_1|] \cap [|A_2|]$
`union`	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$
`union`(A_1, A_2)	=	B *where* $[|B|] = [|A_1|] \cup [|A_2|]$
`diff`	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$
`diff`(A_1, A_2)	=	B *where* $[|B|] = [|A_1|] \setminus [|A_2|]$
`fromSeq`	:	$Seq \to \mathbb{S}$
`fromSeq` S	=	B *where* $[|B|] = \mathit{range}(S)$
`toSeq` A	:	$\mathbb{S} \to Seq$
`toSeq` A	=	S *where* $S = \langle x_0, x_1, \ldots, x_{\lvert A \rvert - 1} \rangle$ *and* $[|A|] = \{x_0, x_1, \ldots, x_{\lvert A \rvert - 1}\}$.

We now look at an interesting relationship between the following pairs of functions:

$$\texttt{find} \leftrightarrow \texttt{intersection}$$
$$\texttt{insert} \leftrightarrow \texttt{union}$$
$$\texttt{delete} \leftrightarrow \texttt{difference}$$

It turns out that for each pair we can implement either one in terms of the other (almost). Lets refer to functions on the left (`find`, `insert`, and `delete`) as the *singleton* functions and the ones on right (`intersection`, `union`, and `difference`) as the *bulk* functions. Implementing the singleton functions in terms of the bulk functions we have:

fun $\texttt{find}(S, e) = |S \cap \{e\}| = 1$
fun $\texttt{insert}(S, e) = S \cup \{e\}$
fun $\texttt{delete}(S, e) = S \setminus \{e\}$

Indeed this is exactly the definitions given in the definition of sets. To implement the bulk functions in terms of the singleton ones we can basically apply the singleton version one by one on each element of one of the sets. This gives us:

```
fun intersection(S_1, S_2) =
  iter (fn (S, x) => if find(S_1, x) then insert(S, x) else S)
       ∅ S_2

fun union(S_1, S_2) = iter insert S_1 S_2

fun difference(S_1, S_2) = iter delete S_1 S_2
```

For intersection we used both `find` and `insert`, and hence the "almost" mentioned earlier.

Exercise 11.3. *Implement intersection with* `find` *and* `delete` *instead, still using an* `iter` *over* S_2.

Since the singleton versions of the functions are arguably simpler than the bulk versions, one might argue that the singleton functions should be primitives and the bulk functions should be implemented based on them, as shown above.

Question 11.4. *Can you see a problem with this approach?*

One problem with this approach is that the one-by-one approach forces the bulk versions to be fully sequential.

Question 11.5. *Why can we not make the implementations of the bulk functions parallel by replacing* `iter` *with* `reduce`.

Unfortunately we cannot implement a parallel version of the bulk functions by replacing `iter` with `reduce` since the combining functions being used are not associative.

When looking at the tree-based cost specification for the Set ADT momentarily we will see indeed that the bulk functions can be implemented in parallel with only logarithmic span, compared to at least linear for implementation based on the singleton functions. Perhaps, more surprisingly, we will see that the direct implementation of the bulk functions also do less asymptotic work than the versions implemented with the singleton functions. Finally we will see that the implementation of the singleton functions based on the bulk functions is has asymptotically the same performance as any direct implementation of the singleton functions.

In summary, one should view the bulk functions `intersection`, `union` and `difference` as primitives and the singleton functions as derivative. In fact, in parallel code one should try to avoid the singleton functions (especially `insert` and `delete`) as much as possible.

Remark 11.6. *You may notice that the interface does not contain a `map` function. If we try to generalize the notion of `map` from sequences, a `map` function does not make sense in the context of a set: if we interpret map to take in a collection, apply some function to each element and return a collection of the same structure. Consider a function that always returns* 0*. Mapping this over a set would return all zeros, which would then be collapsed into a singleton set, containing exactly* 0*. Therefore, such a `map` would allow reducing the set of arbitrary size to a singleton, which doesn't match the `map` paradigm (which traditionally preserves the structure and size).*

Remark 11.7. *Most programming languages either support sets directly (e.g., Python and Ruby) or have libraries that support them (e.g., in the C++ STL library and Java collections framework). They sometimes have more than one implementation of sets. For example, Java has sets based on hash tables and balanced trees. Unsurprisingly, the set interface in different libraries and languages differ in subtle ways. So, when using one of these interfaces you should always read the documentation carefully.*

Syntax for sets. As in mathematics, we adopt some special notation for sets to simplify their expressions in comprehension.

Syntax 11.8 (Sets). *We use the standard set notation* $\{e_o, e_1, \cdots, e_n\}$ *to indicate a set. The notation* $\emptyset$ *or* $\{\}$ *refers to an empty set. We also use the conventional mathematical syntax for set operations such as* $|S|$ *(size),* $\cup$ *(union),* $\cap$ *(intersection), and* $\setminus$ *(difference). In addition, we use set comprehensions for `filter` and for constructing sets from other sets. In addition We will also use the following shorthands:*

$$\begin{aligned} e \in^? S &\equiv \texttt{find}(S, e) \\ \bigcup_{s \in S} s &\equiv \overline{\sum}^{\texttt{union}\ \emptyset} S \end{aligned}$$

Example 11.9. *Some operations on sets:*

$$\begin{aligned} |\{a,b,c\}| &= 3 \\ \{x \in \{4,11,2,6\} \mid x < 7\} &= \{4,2,6\} \\ 4 \in^? \{6,2,9,11,8\} &= F \\ \{2,7,8,11\} \cup \{7,9,11,14,17\} &= \{2,7,8,9,11,14,17\} \\ \texttt{toSeq}(\{2,7,8,11\}) &= \langle 8,11,2,7 \rangle \\ \texttt{fromeq}(\langle 2,7,2,8,11,2 \rangle) &= \{8,2,11,7\} \end{aligned}$$

Cost specification for sets. Having laid out their interface, we can consider costs, which depends on implementation.

Question 11.10. *Can you think of a way to implement sets?*

Sets can be implemented in several ways. The most common efficient ways used hashing or balanced trees. There are various tradeoffs in cost. For simplicity, we'll consider a cost model based on a balanced-tree implementation. Since a balanced tree implementation requires comparisons inside the various set operations, the cost of these comparisons affects the work and span. For this, we'll assume that compare has C_w work and C_s span.

Cost Specification 11.11 (Tree Sets)**.**

	Work	Span
$\texttt{size}(S)$ $\texttt{singleton}(e)$	$O(1)$	$O(1)$
$\texttt{filter}(f, S)$	$O\left(\sum_{e \in S} W(f(e))\right)$	$O\left(\log \lvert S\rvert + \max_{e \in S} S(f(e))\right)$
$\texttt{find}(S, e)$ $\texttt{insert}(S, e)$ $\texttt{delete}(S, e)$	$O(C_w \cdot \log \lvert S\rvert)$	$O(C_s \cdot \log \lvert S\rvert)$
$\texttt{intersection}(S_1, S_2)$ $\texttt{union}(S_1, S_2)$ $\texttt{difference}(S_1, S_2)$	$O\big(C_w \cdot m \cdot \log(1 + \frac{n}{m})\big)$	$O\big(C_s \cdot \log(n+m)\big)$

where $n = \max(|S_1|, |S_2|)$ *and* $m = \min(|S_1|, |S_2|)$.

Let us consider these cost specifications in some more detail. The cost for `filter` is effectively the same as for sequences, and therefore should not be surprising. It assumes the

function f is applied to the elements of the set in parallel. The cost for the singleton functions (`find`, `insert`, and `delete`) are what one might expect from a balanced binary tree implementation. Basically the tree will have depth around $O(\log n)$ and each of the operations will require searching the tree from the root to some node. We will cover such an implementation in Chapter 10.

The work bounds for the bulk functions (`intersection`, `union`, and `difference`) may seem confusing, especially because of the expression inside the $\log$. To shed some light on the cost, it is helpful to consider two cases, the first when one of the sets is a single element and the other when both sets are equal length. In the first case the bulk operations are doing the same thing as the singleton operations. Indeed if we implement the singleton operations on a set S using the bulk ones, as suggested a few pages ago, then we would like it to be the case that we get the same asymptotic performance. This is indeed the case since we have that $m = 1$ and $n = |S|$, giving:

$$O\left(C_w \cdot \log\left(1 + \frac{|S|}{1}\right)\right) = O(C_w \cdot |S|) \ .$$

Now lets consider the second case when both sets have equal length, say n. In this case we have $m = n$ giving

$$W(n) = O\left(C_w \cdot n \cdot \log\left(1 + \frac{n}{n}\right)\right) = O(C_w \cdot n).$$

Note that this is significantly more efficient than using the singleton operations one by one. For example if we implemented `union` by inserted n elements into a second set of n elements, the cost would be $O(n \log n)$. We would get the same bounds when implementing `difference` with `delete` or `intersection` with `find` and `insert`.

Exercise 11.12. *Show that the work of the bulk operations is $O(n + m)$ and hence is never more than the sum of the lengths of the inputs.*

In addition to doing less work than the singleton functions, the bulk functions also have much less span than multiple applications of the singleton functions. Consequently, in designing parallel algorithms it is good to think about how to use `intersection`, `union`, and `difference` instead of `find`, `insert`, and `delete` if possible.

Example 11.13. *One way to convert a sequence to a set would be to insert the elements one by one, which can be coded as*

$$\texttt{function fromSeq}S = \overline{\prod}^{\texttt{Set.insert}\ \emptyset} S$$

However, this implementation is sequential. We can write a parallel function as follows.

$$\texttt{function fromSeq}S = \overline{\sum}^{\texttt{Set.union}\ \emptyset} \langle \{x\} : x \in S \rangle$$

Exercise 11.14. *Assuming $C_w = C_s = 1$, what is the work and span of each of the implementations of* `fromSeq` *above.*

11.2 Tables: Interface and Cost

We now consider the table interface, which is an abstract data type for the mathematical notion of a mapping. Recall that in set theory a mapping (or function) is a set of key-value pairs in which each key only appears once in the set. Mappings allow us to associate data with keys, which is very useful in the design of algorithms. We already discussed how sequences are the special case of mapping where the keys are integers in the range $\{0, \ldots, n\}$ for some n.

Since we have a ADT for sets we could represent a table as a set of key-value pairs as in their mathematical definition, e.g., $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$. However, there are many unique features of tables that warrant having a separate interface. For example, we would like the interface to ensure that each key only appears once in a table, and we would like a method for efficiently looking up a value based on a key.

Question 11.15. *How would you look up a value based on a key using the set ADT?*

Our ADT for tables is defined in ADT 11.16. The function `find` returns the value associated with the key k. As it may not find the key in the table, its result may be bottom ($\bot$). The function `insert` takes a function f as an argument,

$$f : \mathbb{V} \times \mathbb{V} \to \mathbb{V}.$$

When we insert a key-value pair, we can't simply ignore it if the key is already present, because the values might be different. The purpose of f is to specify what to do if the key being inserted already exists in the table; f is applied to the two values. This function might simply return either its first or second argument, or it can be used, for example, to add the new value to the old one. The parallel counterpart of `find` is the `extract` function. The `extract` operation can be used to find a set of values in a table, returning just the table entries corresponding to elements in the set. The parallel counterpart of `insert` is the `merge` function, which takes a similar function to `insert` since it also has to consider the case that an element appears in both tables. The `merge` operation can add multiple values to a table in parallel by merging two tables. The parallel counterpart of `delete` is the `erase` function, The `erase` operation can delete multiple values from a table in parallel.

In addition to these operations, we can also provide a *`collect`* operation that takes a sequence of key-value pairs and produces a table that maps every key in S to all the values associated with it in S, gathering all the values with the same key together in a sequence. Such a function can be implemented in several ways. For example, we can use the *`collect`* operation that operate on sequences (Chapter 5) as discussed previously and then use tabulate to make a table out of this sequence. We can also implement it more directly using as follows.

Abstract Data Type 11.16 (Tables). *For a universe of keys $\mathbb{K}$, and a universe of values $\mathbb{V}$, the* TABLE *abstract data type is a type $\mathbb{T}$ representing the power set of $\mathbb{K} \times \mathbb{V}$ restricted so that each key appears at most once (i.e., any set of key-value pairs where a key appears just once) along with the following functions:*

$$
\begin{array}{lcl}
\texttt{empty} & : & \mathbb{T} \\
\texttt{empty} & = & \emptyset \\
\texttt{size} & : & \mathbb{T} \to \mathbb{N} \\
\texttt{size}(T) & = & |T| \\
\\
\texttt{singleton} & : & \mathbb{K} \times \mathbb{V} \to \mathbb{T} \\
\texttt{singleton}(k,v) & = & \{k \mapsto v\} \\
\\
\texttt{filter} & : & (\mathbb{K} \times \mathbb{V} \to \mathbb{B}) \times \mathbb{T} \to \mathbb{T} \\
\texttt{filter}(p,T) & = & \{(k \mapsto v) \in T \mid p(k,v)\} \\
\\
\texttt{map} & : & (\mathbb{V} \to \mathbb{V}) \times \mathbb{T} \to \mathbb{T} \\
\texttt{map}(f,T) & = & \{k \mapsto f(v) : (k \mapsto v) \in T\} \\
\\
\texttt{tabulate} & : & (\mathbb{K} \to \mathbb{V}) \times \mathbb{S} \to \mathbb{T} \\
\texttt{tabulate}(f,S) & = & \{k \mapsto f(k) : k \in S\} \\
\\
\\
\texttt{find} & : & \mathbb{T} \times \mathbb{K} \to (\mathbb{V} \cup \bot) \\
\texttt{find}(T,k) & = & \begin{cases} v & (k \mapsto v) \in T \\ \bot & \textit{otherwise} \end{cases} \\
\texttt{insert} & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} \times (\mathbb{K} \times \mathbb{V}) \to \mathbb{T} \\
\texttt{insert}(f,T,(k,v)) & = & \texttt{merge} \\
\\
\texttt{delete} & : & \mathbb{T} \times \mathbb{K} \to \mathbb{T} \\
\texttt{delete}(T,k) & = & \{(k' \mapsto v') \in T \mid k \neq k'\} \\
\\
\\
\texttt{extract} & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} \\
\texttt{extract}(T,S) & = & \{(k \mapsto v) \in T \mid k \in S\} \\
\texttt{merge} & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} \times \mathbb{T} \to \mathbb{T} \\
\texttt{merge}(f,T_1,T_2) & = & \forall k \in \mathbb{K}, \begin{cases} k \mapsto f(v_1,v_2) & (k \mapsto v_1) \in T_1 \\ & \quad \wedge (k \mapsto v_2) \in T_2 \\ k \mapsto v_1 & (k \mapsto v_1) \in T_1 \\ k \mapsto v_2 & (k \mapsto v_2) \in T_2 \end{cases} \\
\texttt{erase}(T,S) & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} \\
\texttt{erase}(T,S) & = & \{(k \mapsto v) \in T \mid k \notin S\}
\end{array}
$$

where $\mathbb{S}$ is the power set of $\mathbb{K}$ (i.e., any set of keys), $\mathbb{N}$ are the natural numbers (non-negative integers), and $\mathbb{B} = \{\texttt{T}, \texttt{F}\}$.

Algorithm 11.17 (`collect` on Tables)**.**

function $\texttt{collect}(S) = \overline{\sum}^{Table.merge\ \{\}} \langle \{k \mapsto \langle v \rangle\} : (k, v) \in S \rangle$

Syntax. For brevity and clarity, we use a separate notation for tables: we write

$$\{(k_1 \mapsto v_1), (k_2 \mapsto v_2), \ldots\}$$

for a table that maps k_i to v_i instead of $\{(k_1, v_1), (k_2, v_2), \ldots\}$ since it allows us to distinguish between tables and a set of pairs.

Syntax 11.18 (Tables). *In the book we will use the notation*

$$\{(k_1 \mapsto v_1), (k_2 \mapsto v_2), \ldots, (k_n \mapsto v_n)\},$$

to indicate a table in which each key k_i is mapped to a value v_i. We will also use the following shorthands:

$$\begin{array}{ll}
T[k] & \equiv \texttt{find}(T, k) \\
\{k \mapsto f(x) : k \mapsto x \in T\} & \equiv \texttt{map}(f, T) \\
\{k \mapsto f(x) : k \mapsto x \in S\} & \equiv \texttt{tabulate}(f, S) \\
\{(k \mapsto v) \in T \mid p(k, v)\} & \equiv \texttt{filter}(p, T) \\
T \setminus S & \equiv \texttt{erase}(T, S) \\
T_1 \cup T_2 & \equiv \texttt{merge second}\ (T_1, T_2) \\
\bigcup_{t \in S} t & \equiv \texttt{reduce}\ (\texttt{merge second})\ \emptyset\ S
\end{array}$$

where `fun second(a,b) = b`.

Example 11.19. *Define tables T_1 and T_2 and set S as follows.*

$$\begin{array}{rcl}
T_1 & = & \{a \mapsto 4, b \mapsto 11, c \mapsto 2\} \\
T_2 & = & \{b \mapsto 3, d \mapsto 5\} \\
S & = & \{3, 5, 7\}.
\end{array}$$

The examples below show some operations, also using our syntax.

$$\begin{array}{lrcl}
\texttt{find:} & T_1[b] & = & 11 \\
\texttt{filter:} & \{k \mapsto x \in T_1 \mid x < 7\} & = & \{a \mapsto 4, c \mapsto 2\} \\
\texttt{map:} & \{k \mapsto 3 \times v : k \mapsto v \in T_2\} & = & \{b \mapsto 9, d \mapsto 15\} \\
\texttt{tabulate:} & \{k \mapsto k^2 : k \in S\} & = & \{3 \mapsto 9, 5 \mapsto 25, 9 \mapsto 81\} \\
\texttt{merge:} & T_1 \cup T_2 & = & \{a \mapsto 4, b \mapsto 3, c \mapsto 2, d \mapsto 5\} \\
\texttt{merge:} & \texttt{merge} + (T_1, T_2) & = & \{a \mapsto 4, b \mapsto 14, c \mapsto 2, d \mapsto 5\} \\
\texttt{erase:} & T_1 \setminus \{b, d, e\} & = & \{a \mapsto 4, c \mapsto 2\}
\end{array}$$

Cost specification for tables. The costs of the table operations are very similar to sets.

Cost Specification 11.20 (Tables).

	Work	Span
`size`(T) `singleton`(k, v)	$O(1)$	$O(1)$
`filter`(p, T)	$O\Big(\sum_{(k \mapsto v) \in T} W(p(k, v))\Big)$	$O\Big(\log \lvert T\rvert + \max_{(k \mapsto v) \in T} S(f(k, v))\Big)$
`map`(f, T)	$O\Big(\sum_{(k \mapsto v) \in T} W(f(v))\Big)$	$O\Big(\log \lvert T\rvert + \max_{(k \mapsto v) \in T} S(f(v))\Big)$
`find`(S, k) `insert`$(T, (k, v))$ `delete`(T, k)	$O(C_w \log \lvert T\rvert)$	$O(C_s \log \lvert T\rvert)$
`extract`(T_1, T_2) `merge`(T_1, T_2) `erase`(T_1, T_2)	$O\big(C_w m \log(1 + \frac{n}{m})\big)$	$O\big(C_s \log(n + m)\big)$

where $n = \max(\lvert T_1\rvert, \lvert T_2\rvert)$ *and* $m = \min(\lvert T_1\rvert, \lvert T_2\rvert)$.

As with sets there is a symmetry between the three operations `extract`, `merge`, and `erase`, and the three operations `find`, `insert`, and `delete`, respectively, where the prior three are effectively "parallel" versions of the earlier three.

Remark 11.21. *Abstract data types that support mappings of some form are referred to by various names including mappings, maps, tables, dictionaries, and associative arrays. They are perhaps the most studied of any data type. Most programming languages have some form of mappings either built in (e.g. dictionaries in Python, Perl, and Ruby), or have libraries to support them (e.g. map in the C++ STL library and the Java collections framework).*

Remark 11.22. *Tables are similar to sets: they extend sets so that each key now carries a value. Their cost specification and implementations are also similar.*

11.3 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.

For these purposes we can extend the operations on sets and tables with some additional operations that take advantage of ordering. ADT 11.23 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 essentially the same as the operations we defined for binary search trees (Chapter 10).

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

To implement efficiently `rank`, `select` and `splitRank`, we can augment the underlying binary search tree implementation with sizes as described in (Chapter 10).

Cost Specification 11.25 (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 Specification 11.11) and tables (Cost Specification 11.20) for the operations supported by sets and tables. The work and span for all the operations in ADT 11.23 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.*

Abstract Data Type 11.23 (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 functions below. In the specification,* $\mathbb{N}$ *is the natural numbers (non-negative integers) and* $\mathbb{B} = \{\mathtt{T}, \mathtt{F}\}$*; for a* A *of type* $\mathbb{S}$ $[|A|]$ *denotes the (mathematical) set of keys in the tree. We assume* max *or* min *of the empty set returns the special element* $\perp$*.*

Include Set ADT (ADT 11.1)

`first`	:	$\mathbb{S} \to (\mathbb{U} \cup \{\perp\})$
`first`(A)	=	$\min [|A|]$
`last`	:	$\mathbb{S} \to (\mathbb{U} \cup \{\perp\})$
`last`(A)	=	$\max [|A|]$
`previous`	:	$\mathbb{S} \times \mathbb{U} \to (\mathbb{U} \cup \{\perp\})$
`previous`(A, k)	=	$\max \{k' \in [|A|] \mid k' < k\}$
`next`	:	$\mathbb{S} \times \mathbb{U} \to (\mathbb{U} \cup \{\perp\})$
`next`(A, k)	=	$\min \{k' \in [|A|] \mid k' > k\}$
`split`	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S} \times \mathbb{B} \times \mathbb{S}$
`split`(A, k)	=	$\left(\{k' \in [|A|] \mid k' < k\}, k \stackrel{?}{\in} S, \{k' \in [|A|] \mid k' > k\}\right)$
`join`	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$
`join`(A_1, A_2)	=	$A_1 \cup A_2$ *assuming* $\max [|A_1|] < \min [|A_2|]$
`getRange`	:	$\mathbb{S} \times \mathbb{U} \times \mathbb{U} \to \mathbb{S}$
`getRange`(A, k_1, k_2)	=	$\{k \in [|A|] \mid k_1 \leq k \leq k_2\}$
`rank`	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{N}$
`rank`(A, k)	=	$\lvert \{k' \in A \mid k' < k\} \rvert$
`select`(A, i)	:	$\mathbb{S} \times \mathbb{N} \to (\mathbb{U} \cup \{\perp\})$
`select`(A, i)	=	$k \in [|A|]$ *such that* `rank`$([|A|], k) = i$ *or* $\perp$ *if there is no such* k
`splitRank`	:	$\mathbb{S} \times \mathbb{N} \to \mathbb{S} \times \mathbb{S}$
`splitRank`(A, i)	=	$(\{k \in [|A|] \mid k < \mathtt{select}(A, i)\}, \{k \in [|A|] \mid k \geq \mathtt{select}(S, i)\})$

Example 11.26. *Consider the following sequence ordered lexicographically:*

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

- `first`(S) *returns "artie".*
- `next`$(S,$ *"quinn"*$)$ *or* `next`$(S,$ *"mike"*$)$ *returns "rachel".*
- `range`$(S,$ *"blain","quinn"*$)$ *or* `range`$(S,$ *"burt","mike"*$)$ *returns*

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

.

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

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

11.4 Ordered Tables with Augmented with Reducers

An interesting extension to ordered tables (and perhaps tables more generally) is to augment the table with a reducer function. We shall see some applications of such augmented tables but let's first describe the interface and its cost specification.

Definition 11.27 (Reducer-Augmented Ordered Table ADT)**.** *For a specified reducer, i.e., an associative function on values* $f : \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ *and identity element* I_f*, a reducer-augmented ordered table supports all operations on ordered tables as specified in ADT 11.23 and the following operation*

$$\begin{array}{lcl} \texttt{reduceVal} & : & \mathbb{T} \to \mathbb{V} \\ \texttt{reduceVal}(T) & = & \texttt{Table.reduce}\ f\ I_f\ A \end{array}$$

The `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.

Question 11.28. *Why do we need to augment, can't we just use* `Table.reduce` *as needed?*

But, by augmenting the table with a reducer, we can do reductions much more efficiently. In particular, by using augmented binary search trees, we can implement `reduceVal` in $O(1)$ work and span.

We consider several applications of reducer-augmented ordered tables.

Example 11.29 (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 11.30 (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.*

11.5 Example: Indexing a Document Collection

As an interesting application of the sets and tables ADT's that we have discussed, let's consider the problem of indexing a set of documents to provide fast and efficient search operations on documents. Concretely suppose that you are given a collection of documents where each document has a unique identifier assumed to be a string and a contents, which is again a string and you want to support a range of operations including

- **word search:** find the documents that contain a given word,
- **logical-and search:** find the documents that contain a given word and another,
- **logical-or search:** find the documents that contain a given word or another,
- **logical-and-not search:** find the documents that contain a given word but not another,
- **domain-search** given a set of document and a given range on their identities, return those whose range falls within the given range.

This kind of interface is exactly what we use when we seach documents on the web. For example, seach engines from companies such as Google and Microsoft allow you to do all of these searches. When searching the web, we can think of the url of a page on the web as its identifier and its contents, as the text (source) of the page. When your search term is two words such as "parallel algorithms", the term is treated as a logical-and search. This is the common search from but search-engines allow you to specify other kinds of queries described above (typically in a separate interface). Perhaps the least common form, the domain search, can (at the time of writing) be invoked by prefixing the search term with "site". For example, the seach "site:cs.cmu.edu parallel algorithms" searches all pages that are in the "cs.cmu.edu" domain. Note that such a search can be specified as a range on identifers by lexicographically ordering the url's in reverse order and specifying a minimum and a maximum character.

Example 11.31. *As a simple document collection, we can consider the tweets made by some of your friends yesterday.*

$$
\begin{aligned}
T = \langle\ & (\text{“jack”}, \text{“Chess club was fun”}),\\
& (\text{“mary”}, \text{“I had fun in dance club today”}),\\
& (\text{“nick”}, \text{“Food at the cafeteria sucks”}),\\
& (\text{“melissa”}, \text{“Dude, I went rock climbing last weekend. It was a blast.”}),\\
& (\text{“peter”}, \text{“I had fun at nick’s party”})\\
& \rangle
\end{aligned}
$$

where the identifiers are the names, and the contents is the tweet.
On this set of documents, searching for “fun” would return “jack”, “mary”, “peter”. Searching for “club” would return “jack” and “mary”.

Question 11.32. *Design a simple, possibly inefficient algorithm to solve the problem.*

We can solve such a search problem by employing a brute-force algorithm that traverses the document collection to answer a search query. Such an algorithm would require at least linear work in the number of the documents, which is unacceptable for large collections, especially when interactive searches are desirable. Instead of using a brute-force algorithm, we can organize the data into an *index* in some way to improve efficency. Since in this problem, we are only interested in querying a static or unchanging collection of documents, such an index can be built once and for all to be used for answering search queries. Since it is built once and for all, the index can be slow to build. Based on this observation, we can adopt the following ADT for indexing and searching our document collection.

Abstract Data Type 11.33 (Document Index).

$$
\begin{array}{lcl}
\mathit{word} & = & \mathit{string}\\
\mathit{id} & = & \mathit{string}\\
\mathit{contents} & = & \mathit{string}\\
\mathit{docs} & & \\
\mathit{index} & & \\
\mathit{mkIndex} & : & \mathit{id} * \mathit{contents\ sequence} \to \mathit{index}\\
\mathit{find} & : & \mathit{index} * \mathit{word} \to \mathit{docs}\\
\mathit{and} & : & \mathit{docs} * \mathit{docs} \to \mathit{docs}\\
\mathit{andNot} & : & \mathit{docs} * \mathit{docs} \to \mathit{docs}\\
\mathit{or} & : & \mathit{docs} * \mathit{docs} \to \mathit{docs}\\
\mathit{inDomain} & : & \mathit{docs} * (\mathit{id} * \mathit{id}) \to \mathit{docs}\\
\mathit{size} & : & \mathit{docs} \to \mathbb{N}\\
\mathit{toSeq} & : & \mathit{docs} \to \mathit{docs\ sequence}
\end{array}
$$

Example 11.34. *As a simple document collection, we can consider the tweets made by some of your friends yesterday.*

$$T = \langle \text{(“jack”, “Chess club was fun”)}, \\ \text{(“mary”, “I had fun in dance club today”)}, \\ \text{(“nick”, “Food at the cafeteria sucks”)}, \\ \text{(“melissa”, “Dude, I went rock climbing last weekend. It was a blast.”)}, \\ \text{(“peter”, “I had fun at nick’s party”)} \\ \rangle$$

where the identifiers are the names, and the contents is the tweet.
On this set of documents, searching for “fun” would return “jack”, “mary”, “peter”. Searching for “club” would return “jack” and “mary”.
To implement such operations, we can use ADT 11.33 to make an index of these tweets:

```
1 f : word → docs = find (mkIndex T)
```

In addition to making the index, we partially apply `find` *on the index. This makes it possible to use* `find` *with the index.*
For example, the code,

```
toSeq (and(f ''fun'', or(f ''food'', f ''rock'')))
```

returns all the documents (tweets) that contain “fun” and either “club” or “rock”, which are ⟨ `"jack", "mary"` ⟩*. The code,*

```
size(andnot(f ''fun'', f ''climbing''))
```

returns the number of documents that contain “fun” and not “club”, which is 1.

Question 11.35. *How can we implement this ADT? Let’s start with the* `mkIndex` *function.*

We can implement this interface using sets and tables. The `mkIndex` function can be implemented as follows.

Algorithm 11.36.

```
1 function mkIndex(docs) =
2 let
3   fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
4   val Pairs = flatten ⟨tagWords(d) : d ∈ docs⟩
5   val Words = Table.collect(Pairs)
6 in
7   {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
8 end
```

The `tagWords` function takes a document as a pair consisting of the document identifier and contents, breaks the string into tokens (words) and tags each token with the identifier returning a sequence of these pairs.

Example 11.37. *Here is an example of how* `tagWords` *works:*

```
tagWords(''jack'', ''chess club was fun'')
```

returns

$$\langle (\text{“Chess”}, \text{“jack”}), (\text{“club”}, \text{“jack”}), (\text{“was”}, \text{“jack”}), (\text{“fun”}, \text{“jack”}) \rangle$$

To build the index, we apply `tagWords` *to all documents, and flatten the result to a single sequence.*
In our example the result would start as:

$$\begin{aligned} \texttt{Pairs} = \langle\ & (\text{“chess”}, \text{“jack”}), (\text{“club”}, \text{“jack”}), (\text{“was”}, \text{“jack”}), \\ & (\text{“fun”}, \text{“jack”}), (\text{“I”}, \text{“mary”}), (\text{“had”}, \text{“mary”}), (\text{“fun”}, \text{“mary”}), \ldots \end{aligned}$$

Using `Table.collect`*, we then collect the entries by word creating a sequence of matching documents.*
In our example it would start:

$$\begin{aligned} \texttt{Words} = \{ & (\text{“a”} \mapsto \langle\ \text{“mary”}\ \rangle), \\ & (\text{“at”} \mapsto \langle\ \text{“mary”}, \text{“peter”}\ \rangle), \\ & \ldots \\ & (\text{“fun”} \mapsto \langle\ \text{“jack”}, \text{“mary”}, \text{“sue”}, \text{“peter”}, \text{“john”}\ \rangle), \\ & \ldots \end{aligned}$$

Finally, for each word the sequences of document identifiers is converted to a set. Note the notation that is used to express a map over the elements of a table.

Question 11.38. *Do you see how we might implement the rest of the interface, which includes functionality for performing searches?*

The rest of the interface can be implemented as follows:

Algorithm 11.39.

```
fun find T v = Table.find T v
fun And(s1, s2) = s1 ∩ s2
fun Or(s1, s2) = s1 ∪ s2
fun AndNot(s1, s2) = s1 \ s2
fun size(s) = |s|
fun toSeq(s) = Set.ToSeq(s)
```

Cost. Assuming that all tokens have a length upper bounded by a constant, the cost of `mkIndex` is dominated by the collect, which is basically a sort. The work is therefore $O(n \log n)$ and the span is $O(\log^2 n)$, assuming the words have constant length.

Note that if we do a `size(f "red")` the cost is only $O(\log n)$ work and span. It just involves a search and then a length.

If we do `And(f "fun", Or(f "courses", f "classes"))` the worst case work and span are at most:

$$\begin{aligned} W &= O(|f(\text{"}\textit{fun}\text{"})| + |f(\text{"}\textit{courses}\text{"})| + |f(\text{"}\textit{classes}\text{"})|) \\ S &= O(\log |\textit{index}|) \end{aligned}$$

The sum of sizes is to account for the cost of the `And` and `Or`. The actual cost could be significantly less especially if one of the sets is very small.

11.6 Problems

11-1 Union on Different Sizes
Based on the cost specification for Sets, what is the assymptotic work and span for taking the union of two sets one of size n and the other of size $\sqrt{n}$? You can assume the comparison for sets takes constant work. Please simplify as much as possible.

11-2 Cost of Table Collect
The following code implements `Table.collect`(S).

```
Seq.reduce (Table.merge Seq.append) ⟨ ⟩ {k ↦ ⟨ v ⟩ : v ∈ S}
```

Derive (tight) asymptotic upper bounds on the work and span for the code in terms of $n = |S|$. You can assume the comparison function used by the table takes constant work, and that the Sequence is a array sequence. You don't need a proof.

11-3 Identifying Repeats
Implement a function `identifyRepeats` that given a sequence of integer returns a table that maps each unique element to either `ONE` or `MULTI`. For example

$$\texttt{identifyRepeats}(\langle 7, 2, 4, 2, 3, 2, 1, 3 \rangle)$$

would return $\{1 \mapsto \texttt{ONE}, 2 \mapsto \texttt{MULTI}, 3 \mapsto \texttt{MULTI}, 4 \mapsto \texttt{ONE}, 7 \mapsto \texttt{ONE}\}$. What is the (tight) assymptotic upper bound for work and span for your implementation in terms of $n = |S|$.

11-4 Union on Multiple Sets
The following code takes the union of a sequence of sets:

```
fun UnionSets(S) = Seq.reduce Set.Union} {} S
```

For example

`UnionSets`$(\langle \{5,7\}, \{3,1,5,2\}, \{2,5,8\} \rangle)$

would return the set $\{1,2,3,5,7,8\}$. Derive (tight) asymptotic upper bounds for the work and span for `UnionSets` in terms of $n = |S|$ and $m = \sum_{x \in S} |x|$. You can assume the comparison used for the sets takes constant work. Please keep your analysis to under a page. We do not need a formal proof. An explanation based on the tree-method is sufficient.

Exercise 11.40. *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.*

Exercise 11.41. *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:*

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

How would you implement this.

.

11.7 Problems

11-5 Union on Different Sizes
Based on the cost specification for Sets, what is the assymptotic work and span for taking the union of two sets one of size n and the other of size $\sqrt{n}$? You can assume the comparison for sets takes constant work. Please simplify as much as possible.

11-6 Cost of Table Collect
The following code implements `Table.collect`(S).

```
Seq.reduce (Table.merge Seq.append) ⟨ ⟩ {k ↦ ⟨ v ⟩ : v ∈ S}
```

Derive (tight) asymptotic upper bounds on the work and span for the code in terms of $n = |S|$. You can assume the comparison function used by the table takes constant work, and that the Sequence is a array sequence. You don't need a proof.

11-7 Identifying Repeats
Implement a function `identifyRepeats` that given a sequence of integer returns a table that maps each unique element to either `ONE` or `MULTI`. For example

$$\texttt{identifyRepeats}(\langle 7, 2, 4, 2, 3, 2, 1, 3 \rangle)$$

would return $\{1 \mapsto \texttt{ONE}, 2 \mapsto \texttt{MULTI}, 3 \mapsto \texttt{MULTI}, 4 \mapsto \texttt{ONE}, 7 \mapsto \texttt{ONE}\}$. What is the (tight) assymptotic upper bound for work and span for your implementation in terms of $n = |S|$.

11-8 Union on Multiple Sets
The following code takes the union of a sequence of sets:

```
fun UnionSets(S) = Seq.reduce Set.Union} {} S
```

For example

$$\texttt{UnionSets}(\langle \{5, 7\}, \{3, 1, 5, 2\}, \{2, 5, 8\} \rangle)$$

would return the set $\{1, 2, 3, 5, 7, 8\}$. Derive (tight) asymptotic upper bounds for the work and span for `UnionSets` in terms of $n = |S|$ and $m = \sum_{x \in S} |x|$. You can assume the comparison used for the sets takes constant work. Please keep your analysis to under a page. We do not need a formal proof. An explanation based on the tree-method is sufficient.

Exercise 11.42. *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.*

Exercise 11.43. *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*

How would you implement this.

.