Chapter 13

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 composition 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 12) and hashing (Chapter 20) that can be used for implementing the sets and tables interfaces. We also consider many applications of sets and tables. For example, sequences (Chapter 6) are a particular type of table—one where the domain of the tables are the integers from 0 to $n-1$. Other examples include graphs (Chapter 14).

Returning to the discussion in the first paragraph about paradoxes in set theory: in our presentation we define sets as elements from particular universes, so as long as these universes are built up in a certain way, we avoid such paradoxes. However, we put an additional restriction on sets (and tables). In particular, although we allow for infinite universes (countable or uncountable) we limit ourselves to finite sized sets. This means we certainly do not capture all set theory, but the assumption makes everything computationally feasible. Representation of infinite sets is an interesting research topic but beyond the scope of this book.

When using sets in SPARC pseudo-code, we adopt some special notation for sets to simplify their expressions in comprehension.

Syntax 13.1. [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. We will also use the following shorthands:

$$\bigcup_{s \in S} s \quad \equiv \texttt{reduce union } \emptyset \quad S$$

13.1 Sets: Interface and Specification

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 $\mathbb{B} = \{\texttt{T}, \texttt{F}\}$; for a set A of type $\mathbb{S}$, $[\![A]\!]$ denotes the (mathematical) set of keys in the set.

Recall that a ***set*** is a collection of distinct objects. The objects that are contained in a set, are called ***members*** or the ***elements*** of the set. The elements of a set must be distinct: a set may not contain the same element more than once. (For more background on sets, please see Section 2.1.) Based on these definitions, we define our ADT for sets (ADT 13.2) by assuming that elements of the sets are drawn from a universe $\mathbb{U}$ of elements that can be tested for equality.

The Set ADT consists of basic operations on sets. The operation `size` takes a set and returns the number of elements in the set. The operation `toSeq` converts a set to a sequence by ordering the elements of the set in an unspecified way. Since elements of a set may not accept an ordering relation, the resulting order is arbitrary. This means that `toSeq` is possibly non-deterministic—it could return different orderings in different implementations, or even on different runs of the same implementation. We specify `toSeq` as follows

```
toSeq ({ x_0, x_1, ..., x_n } : 𝕊): seq = ⟨ x_0, x_1, ..., x_n ⟩.
```

Several operations enable constructing sets. The function `empty` returns an empty set; we can specify it as follows.

Abstract Data Type 13.2. [Sets]

`size`	:	$\mathbb{S} \to \mathbb{N}$
`toSeq` A	:	$\mathbb{S} \to Seq$
`empty`	:	$\mathbb{S}$
`singleton`	:	$\mathbb{U} \to \mathbb{S}$
`fromSeq`	:	$Seq \to \mathbb{S}$
`filter`	:	$((\mathbb{U} \to \mathbb{B}) \to \mathbb{S}) \to \mathbb{S}$
`intersection`	:	$\mathbb{S} \to \mathbb{S} \to \mathbb{S}$
`difference`	:	$\mathbb{S} \to \mathbb{S} \to \mathbb{S}$
`union`	:	$\mathbb{S} \to \mathbb{S} \to \mathbb{S}$
`find`	:	$\mathbb{S} \to \mathbb{U} \to \mathbb{B}$
`delete`	:	$\mathbb{S} \to \mathbb{U} \to \mathbb{S}$
`insert`	:	$\mathbb{S} \to \mathbb{U} \to \mathbb{S}$
`iterate`	:	$(\alpha * \mathbb{U} \to \alpha) \to \alpha \to \mathbb{S} \to \alpha$
`reduce`	:	$(\mathbb{U} \times \mathbb{U} \to \mathbb{U}) \to \mathbb{U} \to \mathbb{S} \to \mathbb{U}$

```
empty : 𝕊 = ∅
```

The function `singleton` constructs a singleton set from a given element.

```
singleton (x : 𝕌) : 𝕊 = { x }
```

The operation `fromSeq` takes a sequence and returns a set consisting of the distinct elements of the sequence, eliminating duplicate elements. We can specify `fromSeq` as returning the range of the sequence A (recall that a sequence is a partial function mapping from natural numbers to elements of the sequence).

```
fromSeq (A: seq) : 𝕊 = range A
```

Several operations operate on sets to produce new sets. The operation `filter` selects the elements of a sequence that satisfy a given Boolean function, i.e.,

```
filter (f : 𝕌 → 𝔹) (A : 𝕊) : 𝕊 = { x ∈ A | f(x) }.
```

The operations `intersection`, `difference`, and `union` perform the corresponding set operation on their arguments:

```
intersection (A : 𝕊) (B : 𝕊) : 𝕊 = A ∩ B
difference (A  : 𝕊) (B : 𝕊) : 𝕊 = A \ B.
union (A : 𝕊) (B : 𝕊) : 𝕊 = A ∪ B
```

We refer to the operations `intersection`, `difference`, and `union` as ***bulk updates***, because they allow updating with a large set of elements "in bulk."

The operations `find`, `insert`, and `delete` are singular versions of the bulk operations `intersection`, `union`, and `difference` respectively. The `find` operation checks whether an element is in a set—it is the basic membership test for sets.

$$\texttt{find}\ (A\ :\ \mathbb{S})\ (x\ :\ \mathbb{U})\ :\ \mathbb{B} = \begin{cases} \texttt{true} & \text{if } x \in A \\ \texttt{false} & \text{otherwise} \end{cases}$$

We can also specify the `find` operation is in terms of set intersection:

$$\texttt{find}\ (A\ :\ \mathbb{S})\ (x\ :\ \mathbb{U})\ :\ \mathbb{B}\ =\ |A \cap \{x\}| = 1\,.$$

The operations `delete` and `insert` delete an existing element from a set, and insert a new element into a set, respectively:

$$\texttt{delete}\ (A\ :\ \mathbb{S})\ (x\ :\ \mathbb{U})\ :\ \mathbb{S}\ =\ A \setminus \{x\}\,.$$
$$\texttt{insert}\ (A\ :\ \mathbb{S})\ (x\ :\ \mathbb{U})\ :\ \mathbb{S}\ =\ A \cup \{x\}$$

The next two operations enable aggregating over sets, via iteration and reduction. The operation `iterate` can be used to iterate over a set while accumulating information. The operation takes an initial value x, a function f, and a set A, and iterates over the elements of A in some unspecified, possibly non-deterministic, order, to produce a final value.

$$\begin{aligned} \texttt{iterate}\ \ & (f : \alpha * \mathbb{U} \to \alpha) \to (\text{x: } \alpha)(A : \mathbb{S}) : \alpha \\ & = \begin{cases} x & \text{if } |A| = 0 \\ \texttt{iterate}\ f\ (f(v, y))\ (A \setminus \{y\}) & \text{otherwise, where } y \in A. \end{cases} \end{aligned}$$

The function `iterate` computes its final result by computing a new state for each element of the set $A = \{a_0, \ldots a_n\}$

$$\begin{aligned} x_0 &= x \\ x_1 &= f(x_0, a_0) \\ x_2 &= f(x_1, a_1) \\ &\vdots \\ x_n &= f(x_{n-1}, a_n). \end{aligned}$$

To perform a reduction over a set, the ADT supplies the `reduce` operation. The operation takes associative binary operation f along with the left identity element for the operation id, and applies f to the elements in the set to produce the "sum" of the elements. The identity of f satisfies the constraint that $f(id, x) = x$ for all $x \in \mathbb{U}$. For a set $A = \{a_0, \ldots, a_n\}$, we can define the behavior of `reduce` inductively as follows.

$$\texttt{reduce}\ \ (f : \mathbb{U} * \mathbb{U} \to \mathbb{U})\ (id : \mathbb{U})\ (A : \mathbb{S}) : \mathbb{U}$$
$$= \begin{cases} id & \text{if } |A| = 0 \\ A[0] & \text{if } |A| = 1 \\ f(x, y) & \text{if } |A| = 2, \text{ and } A = \{x, y\} \\ f(\texttt{reduce}\ f\ id\ \{a_0, \ldots, a_m\} & \\ \quad \texttt{reduce}\ f\ id\ \{a_{m+1}, \ldots, a_n\} & \text{otherwise, where } m = \lceil n/2 \rceil. \end{cases}$$

Notice that in the Set ADT although the universe $\mathbb{U}$ 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 complement 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 13.3. Convince yourself that there is no way to create an infinite sized set using the interface and with finite work.

Example 13.4. Some operations on sets:

$$\begin{aligned} |\{a, b, c\}| &= 3 \\ \{x \in \{4, 11, 2, 6\} \mid x < 7\} &= \{4, 2, 6\} \\ \texttt{find}\ \{6, 2, 9, 11, 8\}\ 4 &= \texttt{false} \\ \{2, 7, 8, 11\} \cup \{7, 9, 11, 14, 17\} &= \{2, 7, 8, 9, 11, 14, 17\} \\ \texttt{toSeq}\ \{2, 7, 8, 11\} &= \langle 8, 11, 2, 7 \rangle \\ \texttt{fromSeq}\ u\ \langle 2, 7, 2, 8, 11, 2 \rangle &= \{8, 2, 11, 7\} \end{aligned}$$

Remark 13.5. 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 13.6. 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.

13.2 Cost Specification for Sets

Sets can be implemented in several ways. The most common efficient ways use hashing or balanced trees. An interesting aspect of both of these implementations is that they require more that just equality on the element (or the universe $\mathbb{U}$): an implementation based on balanced search trees requires a total ordering on the elements of $\mathbb{U}$ and hence a comparison; an implementation based on hashing does not require comparison, but requires the ability to hash the elements of $\mathbb{U}$. For the purposed of the ADT, we consider the operations equality, comparison, and hashing, to be implicitly defined on the universe $\mathbb{U}$.

There are various cost tradeoffs in the implementations based on balanced binary search trees and hashing. We 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 the specification presented here Cost Specification 13.7, we assume that the comparison operation requires constant work and span.

Cost Specification 13.7. [Tree Sets]

The cost specification for tree-based implementation of sets follow. For the specification, we define $n = \max(|A_1|, |A_2|)$ and $m = \min(|A_1|, |A_2|)$.

	Work	*Span*
`size` A `singleton` x	1	1
`toSeq` A	$\sum_{x \in A} \lvert A \rvert$	$\lg \lvert A \rvert$
`filter` f A	$\sum_{x \in A} W(f(x))$	$\lg \lvert A \rvert + \max_{x \in A} S(f(x))$
`intersection` A_1 A_2 `difference` A_1 A_2 `union` A_1 A_2	$m \cdot \lg(1 + \frac{n}{m})$	$\lg(n + m)$
`find` A e `delete` A x `insert` A x	$\lg \lvert A \rvert$	$\lg \lvert A \rvert$

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 $O(\lg n)$ depth and each operation will require searching the tree from the root to some node. We cover such an implementation in Chapter 12.

The work bounds for the bulk functions (`intersection`, `difference`, and `union`) may

seem confusing, especially because of the expression inside the logarithm. 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 A 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 $n = |A|$, giving:

$$O\left(\lg\left(1 + \frac{|A|}{1}\right)\right) = O(|A|)\ .$$

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

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

We can implement `find`, `delete`, and `insert` in terms of the functions `intersection`, `difference`, and `union` (respectively) by making a singleton set out of the element that we are interested in. Such an implementation would be asymptotically efficient, giving us the work and span as the direct implementations. Conversely, we can also implement the bulk operations in terms of the singleton ones by iteration. Since it uses iteration, however, the resulting algorithms are sequential. Furthermore, they are also work inefficient. For example, if we implement `union` by inserting n elements into a second set of n elements, the cost would be $O(n \lg n)$. We would obtain a similar bound when implementing `difference` with `delete`, and `intersection` with `find` and `insert`. In designing parallel algorithms, we therefore prefer to use the bulk operations `intersection`, `difference`, and `union` instead of `find`, `delete`, and `insert` when possible.

Example 13.8. One way to convert a sequence to a set would be to insert the elements one by one, which can be coded as

```
fromSeq A = iterate Set.insert ∅ A
```

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

```
fromSeq A = reduce Set.union ∅ ⟨{x} : x ∈ A⟩
```

13.3 Tables: Interface

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

Abstract Data Type 13.9. [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 along with functions below. Throughout the set $\mathbb{S}$ denotes the powerset of the keys $\mathbb{K}$, $\mathbb{N}$ are the natural numbers (non-negative integers), and $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$.

$$
\begin{array}{ll}
\texttt{size} & : \mathbb{T} \to \mathbb{N} \\
\texttt{empty} & : \mathbb{T} \\
\texttt{singleton} & : \mathbb{K} \times \mathbb{V} \to \mathbb{T} \\
\texttt{tabulate} & : (\mathbb{K} \to \mathbb{V}) \to \mathbb{S} \to \mathbb{T} \\
\texttt{map} & : (\mathbb{V} \to \mathbb{V}) \to \mathbb{T} \to \mathbb{T} \\
\texttt{filter} & : (\mathbb{K} \times \mathbb{V} \to \mathbb{B}) \to \mathbb{T} \to \mathbb{T} \\
\texttt{intersection} & : (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \to \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\
\texttt{union} & : (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \to \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\
\texttt{difference} & : \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\
\texttt{find} & : \mathbb{T} \to \mathbb{K} \to (\mathbb{V} \cup \bot) \\
\texttt{delete} & : \mathbb{T} \to \mathbb{K} \to \mathbb{T} \\
\texttt{insert} & : (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \to \mathbb{T} \to (\mathbb{K} \times \mathbb{V}) \to \mathbb{T} \\
\texttt{restrict} & : \mathbb{T} \to \mathbb{S} \to \mathbb{T} \\
\texttt{subtract} & : \mathbb{T} \to \mathbb{S} \to \mathbb{T}
\end{array}
$$

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 an 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.

Our ADT for tables is defined in ADT 13.9. 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.

The operation `size` returns the size of the table, defined as the number of key-value pairs, i.e.,

$$\texttt{size}\ (A : \mathbb{T}) : \mathbb{N} = |A|.$$

The operation `empty` generates an empty table, i.e.,

$$\texttt{empty} : \mathbb{T} = \emptyset.$$

The operation `singleton` generates an table consisting of a single key-value pair, i.e.,

$$\texttt{singleton}\ (k : \mathbb{K}, v : \mathbb{V}) : \mathbb{T} = \{k \mapsto v\}\,.$$

Larger tables can be created by using the `tabulate` operation, which takes a function and a key set and creates a table by applying the function to each element of the set, i.e.,

$$\texttt{tabulate}\ (f : \mathbb{K} \to \mathbb{V})(A : \mathbb{S}) : \mathbb{T} = \{k \mapsto f(k) : k \in A\}\,.$$

The function `map` creates a table from another by mapping each key-value pair in a table to another by applying the specified function to the value while keeping the keys the same:

$$\texttt{map}\ (f : \mathbb{V} \to \mathbb{V})\ (A : \mathbb{T}) : \mathbb{T} = \{k \mapsto f(v) : k \mapsto v \in A\}\,.$$

The function `filter` selects the key-value pairs in a table that satisfy a given function:

$$\texttt{filter}\ (f : \mathbb{K} \times \mathbb{V} \to \mathbb{B})\ (A : \mathbb{T}) : \mathbb{T} = \{(k \mapsto v) \in A \mid f(k, v)\}\,.$$

The function `intersection` takes the intersection of two tables to generate another table. To handle the case for when the key is already present in the table, the operation takes a ***conflict-resolution*** function f of type $\mathbb{V} \times \mathbb{V} \to \mathbb{V}$ as an argument and uses it to assign a value to the key by passing to f the existing and the new value. We specify intersection as

$$\begin{aligned}\texttt{intersection}\ \ & (f : \mathbb{V} \times \mathbb{V} \to \mathbb{V})\ (A_1 : \mathbb{T})\ (A_2 : \mathbb{T}) : \mathbb{T} \\ & = \{k \mapsto f(v_1, v_2) : (k \mapsto v_1) \in A_1 \cap (k \mapsto v_2) \in A_2\}\,.\end{aligned}$$

The function `difference` substracts one table from another by throwing away all entries in the first table whose key appears in the second.

$$\texttt{difference}\ (A_1 : \mathbb{T})\ (A_2 : \mathbb{T}) : \mathbb{T} = \{k \mapsto v : (k \mapsto v_1) \in A_1 \text{ and } (k \mapsto v_2) \notin A_2\}\,.$$

The operation `union` unions the key value pairs in two tables into a single table. As with `intersection`, the operation takes a conflict-resolution function to determine the ultimate value of a key that appears is both tables. We specify `union` in terms of the `intersection` and `difference` operations.

$$\begin{aligned}\texttt{union}\ \ & (A_1 : \mathbb{T})\ (A_2 : \mathbb{T}) : \mathbb{T} \\ = \ & (\texttt{intersection}\ f\ A_1\ A_2) \cup \\ & (\texttt{difference}\ A_1\ A_2) \cup \\ & (\texttt{difference}\ A_2\ A_1)\end{aligned}$$

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 ($\perp$).

$$\texttt{find}\ (A : \mathbb{T})\ (k : \mathbb{K}) : \mathbb{B} = \begin{cases} v & \text{if } k \mapsto v \in A \\ \perp & \text{otherwise} \end{cases}$$

Given a table, the function `delete` deletes a the key-value pair for a specified key from the table:

$$\texttt{delete}\ (A : \mathbb{T})\ (k : \mathbb{K}) = \left\{(k' \mapsto v') \in A \mid k \neq k'\right\}.$$

The operation `insert` inserts a key-value pair into a given table. It can be thought as a singleton version of `union` and specified as such:

$$\texttt{insert}\ (f : \mathbb{V} * \mathbb{V} \to \mathbb{V})\ (A : \mathbb{T})\ (k : \mathbb{K}, v : \mathbb{V}) : \mathbb{T} = \texttt{union}\ f\ A\ (\texttt{singleton}\ (k, v)).$$

The function `restrict` restricts the domain of the table to a given set:

$$\texttt{restrict}\ (A : \mathbb{T})\ (B : set) : \mathbb{T} = \{k \mapsto v \in A \mid k \in B\}.$$

A closely related function `subtract` deletes from a table the entries that belong a specified set:

$$\texttt{subtract}\ (A : \mathbb{T})\ (B : set) : \mathbb{T} = \{(k \mapsto v) \in A \mid k \notin B\}.$$

The functions `intersection` and `resrict` can be viewed as bulk versions of the function `find`. Similarly, the function `difference` and `subtract` can be viewed as the bulk versons of the function `delete`.

In addition to these operations, we can also provide a `collect` operation that takes a sequence A of key-value pairs and produces a table that maps every key in A to all the values associated with it in A, 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 6) 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.

Algorithm 13.10. [`collect` on Tables]

$$\begin{aligned} &\texttt{collect}\ A\ =\ \texttt{Sequence.reduce}\ \ (\ \texttt{Table.union}\ \texttt{Sequence.append}\ ) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle\, \{k \mapsto \langle\, v \,\rangle\} : (k, v) \in A \,\rangle \end{aligned}$$

Syntax 13.11. [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:

$$\begin{array}{ll}
A[k] & \equiv \texttt{find}(A, k) \\
\{k \mapsto f(x) : k \mapsto x \in A\} & \equiv \texttt{map}\ \ f\ \ A \\
\{k \mapsto f(x) : k \in S\} & \equiv \texttt{tabulate}\ \ f\ \ S \\
\{(k \mapsto v) \in A \,|\, p(k, v)\} & \equiv \texttt{filter}\ \ p\ \ A \\
A \setminus C & \equiv \texttt{subtract}\ \ A\ \ C \\
A_1 \cup A_2 & \equiv \texttt{union second}\ \ A_1\ \ A_2 \\
\bigcup_{t \in S} t & \equiv \texttt{reduce (union second)}\ \ \emptyset\ \ S
\end{array}$$

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

Example 13.12. Define tables A_1 and A_2 and set S as follows.

$$\begin{array}{rcl}
A_1 & = & \{a \mapsto 4, b \mapsto 11, c \mapsto 2\} \\
A_2 & = & \{b \mapsto 3, d \mapsto 5\} \\
C & = & \{3, 5, 7\}.
\end{array}$$

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

$$\begin{array}{lrcl}
\texttt{find:} & A_1[b] & = & 11 \\
\texttt{filter:} & \{k \mapsto x \in A_1 \mid x < 7\} & = & \{a \mapsto 4, c \mapsto 2\} \\
\texttt{map:} & \{k \mapsto 3 \times v : k \mapsto v \in A_2\} & = & \{b \mapsto 9, d \mapsto 15\} \\
\texttt{tabulate:} & \{k \mapsto k^2 : k \in C\} & = & \{3 \mapsto 9, 5 \mapsto 25, 9 \mapsto 81\} \\
\texttt{union:} & A_1 \cup A_2 & = & \{a \mapsto 4, b \mapsto 3, c \mapsto 2, d \mapsto 5\} \\
\texttt{union:} & \texttt{union} + (A_1, A_2) & = & \{a \mapsto 4, b \mapsto 14, c \mapsto 2, d \mapsto 5\} \\
\texttt{subtract:} & A_1 \setminus \{b, d, e\} & = & \{a \mapsto 4, c \mapsto 2\}
\end{array}$$

13.4 Cost Specification for Tables

The costs of the table operations are very similar to sets. As with sets there is a symmetry between the three operations `restrict`, `union`, and `subtract`, and the three operations `find`, `insert`, and `delete`, respectively, where the prior three are effectively "parallel" versions of the earlier three.

Cost Specification 13.13. [Tables]

	Work	*Span*
`size` A `singleton` (k, v)	1	1
`filter` p A	$\sum_{(k \mapsto v) \in A} W(p(k, v))$	$\lg \lvert A \rvert + \max_{(k \mapsto v) \in A} S(f(k, v))$
`map` f A	$\sum_{(k \mapsto v) \in A} W(f(v))$	$\lg \lvert A \rvert + \max_{(k \mapsto v) \in A} S(f(v))$
`find` A k `delete` A k `insert` f A (k, v)	$\lg \lvert A \rvert$	$\lg \lvert A \rvert$
`intersection` f A_1 A_2 `difference` A_1 A_2 `union` f A_1 A_2 `restrict` A C `subtract` A C	$m \lg(1 + \frac{n}{m})$	$\lg(n + m)$

where $n = \max(|A_1|, |A_2|)$ and $m = \min(|A_1|, |A_2|)$ or $n = \max(|A|, |C|)$ and $m = \min(|A|, |C|)$ as applicable. For `union` and `intersection`, we assume that $W(f(\cdot, \cdot)) = S(f(\cdot, \cdot)) = O(1)$.

Remark 13.14. 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 13.15. Tables are similar to sets: they extend sets so that each key now carries a value. Their cost specification and implementations are also similar.

13.5 Example: Indexing and Searching a 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,

This kind of interface is exactly what we use when we search documents on the web. For example, search 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).

Example 13.16. As a simple document collection, we can consider the tweets made by some of your friends yesterday.

```
T = ⟨("jack", "chess␣is␣fun"),
     ("mary", "I␣had␣fun␣in␣dance␣club␣today"),
     ("nick", "food␣at␣the␣cafeteria␣sucks"),
     ("melissa'',␣"rock climbing was a blast.''),
     ("peter'',␣"I had fun at nick's party'')
     ⟩
```

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 `"mary"`.

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. Since in this problem, we are only interested in querying a static or unchanging collection of documents, we can ***stage*** the algorithm: first, we organize the documents into an *index* and then we use the index to answer search queries. Since, we build the index only once, we can afford to perform significant work to do so. Based on this observation, we can adopt the following ADT for indexing and searching our document collection.

Abstract Data Type 13.17. [Document Index]

```
type word      = string
type id        = string
type contents  = string
type docs
type index
mkIndex        : id → contents sequence → index
find           : index → word → docs
lAnd           : docs → docs → docs
lAndNot        : docs → docs → docs
or             : docs → docs → docs
size           : docs → ℕ
toSeq          : docs → docs sequence
```

Example 13.18. For our collection of tweets, we can use ADT 13.17 to make an index of these tweets and define a function to find a word in this index as follows.

```
fw : word → docs = find (mkIndex T)
```

We build the index and then partially apply `find` on the index. This way, we have staged the computation so that the index is built only once; the subsequent searches will use the index.

For example, the code,

```
toSeq (lAnd (fw "fun") (lOr (fw "food")  (fw "rock")))
```

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

```
size (lAndNot (fw "fun") (fw "climbing"))
```

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

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

Algorithm 13.19.

```
mkIndex docs =
let
  tagWords id doc = ⟨ (w, id) : w \in tokens doc ⟩
  pairs = flatten ⟨ tagWords id d | (id, d) ∈ docs ⟩
  words = Table.collect pairs
in
  { w ↦ Set.fromSeq d | w ↦ d ∈ words }
end
```

The `tagWords` function takes a document as a pair consisting of the document identifier and contents, breaks the document into tokens (words) and tags each token with the identifier returning a sequence of these pairs. Using this function, we construct a sequence of identifier-word pairs. We then use `Table.collect` to construct a table that maps each identifier to the sequence of words. Finally, for each table entry, we convert the sequence to a set so that we can perform set operations on them.

Example 13.20.
Here is an example of how `mkIndex` works. We start by tagging the words with their document it, using `tagWords`.

```
tagWords ("jack", "chess is fun")
```

returns

```
⟨ ("chess","jack"), ("is","jack"), ("fun","jack") ⟩
```

To build the index, we apply `tagWords` to all documents, and flatten the result to a single sequence. Using `Table.collect`, we then collect the entries by word creating a sequence of matching documents. In our example, the resulting table has the form:

```
words = {
          ("a" ↦ ⟨ "mary" ⟩),
          ("at" ↦ ⟨ "mary", "peter" ⟩),
          ...
          ("fun" ↦ ⟨ "jack", "mary", "sue", "peter", "john" ⟩),
          ...
        }
```

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.

The rest of the interface can be implemented as follows:

Algorithm 13.21.

```
find T v = Table.find T v
lAnd A B = Set.intersection A B
lOr  A B = Set.union A B
lAndNot A B = Set.difference A B
size A = Set.size A
toSeq A = Set.toSeq A
```

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 perform a `size(fw "red")` the cost is only $O(\log n)$ work and span. It just involves a search and then a length.

If we perform `lAnd (fw "fun") (lOr (fw "courses") (fw "classes"))` the worst case work lAnd span are at most:

$$\begin{aligned} W &= O\left(|\texttt{fw "fun"}| + |\texttt{fw "courses"}| + |\texttt{fw "classes"}|\right) \\ S &= O\left(\log |index|\right) \end{aligned}$$

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

13.6 Ordered Sets and Tables

The set and table interfaces described so far do not include any operations that use the ordering of the elements. This allows the interfaces to be defined on types that don't have a natural ordering, which makes the interfaces to be well-suited for an implementation based on hash tables. In many applications, however, it is useful to order the keys and use various ordering operations. 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 August 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 13.22 defines the operations supported by ordered sets, which simply extend the operations on sets. The operations on ***ordered tables*** are completely analogous: they extend the operations on tables in a similar way. Note that `split` and `join` are essentially the same as the operations we defined for binary search trees (Chapter 12).

Abstract Data Type 13.22. [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 13.2)

$$
\begin{array}{lcl}
\texttt{first} & : & \mathbb{S} \to (\mathbb{U} \cup \{\perp\}) \\
\texttt{first}(A) & = & \min [\![A]\!] \\
\texttt{last} & : & \mathbb{S} \to (\mathbb{U} \cup \{\perp\}) \\
\texttt{last}(A) & = & \max [\![A]\!] \\
\texttt{previous} & : & \mathbb{S} \to \mathbb{U} \to (\mathbb{U} \cup \{\perp\}) \\
\texttt{previous}\ A\ k & = & \max \{k' \in [\![A]\!] \mid k' < k\} \\
\texttt{next} & : & \mathbb{S} \to \mathbb{U} \to (\mathbb{U} \cup \{\perp\}) \\
\texttt{next}\ A\ k & = & \min \{k' \in [\![A]\!] \mid k' > k\} \\
\texttt{split} & : & \mathbb{S} \to \mathbb{U} \to \mathbb{S} \times \mathbb{B} \times \mathbb{S} \\
\texttt{split}\ A\ k & = & \left(\{k' \in [\![A]\!] \mid k' < k\}, k \stackrel{?}{\in} S, \{k' \in [\![A]\!] \mid k' > k\}\right) \\
\texttt{join} & : & \mathbb{S} \to \mathbb{S} \to \mathbb{S} \\
\texttt{join}\ A_1\ A_2 & = & A_1 \cup A_2 \text{ assuming } \max [\![A_1]\!] < \min [\![A_2]\!] \\
\texttt{getRange} & : & \mathbb{S} \to \mathbb{U} \times \mathbb{U} \to \mathbb{S} \\
\texttt{getRange}\ A\ k_1\ k_2 & = & \{k \in [\![A]\!] \mid k_1 \le k \le k_2\} \\
\texttt{rank} & : & \mathbb{S} \to \mathbb{U} \to \mathbb{N} \\
\texttt{rank}\ A\ k & = & |\{k' \in A \mid k' < k\}| \\
\texttt{select}\ A\ i & : & \mathbb{S} \to \mathbb{N} \to (\mathbb{U} \cup \{\perp\}) \\
\texttt{select}\ A\ i & = & k \in [\![A]\!] \text{ such that } \texttt{rank}([\![A]\!], k) = i \\
 & & \text{or } \perp \text{ if there is no such } k \\
\texttt{splitRank} & : & \mathbb{S} \to \mathbb{N} \to \mathbb{S} \times \mathbb{S} \\
\texttt{splitRank}\ A\ i & = & (\{k \in [\![A]\!] \mid k < \texttt{select}(S, i)\}, \\
 & & \{k \in [\![A]\!] \mid k \ge \texttt{select}(S, i)\})
\end{array}
$$

Example 13.23. Consider the following sequence ordered lexicographically:

$$A = \{\texttt{"artie"}, \texttt{"burt"}, \texttt{"finn"}, \texttt{"mike"}, \texttt{"rachel"}, \texttt{"sam"}, \texttt{"tina"}\}$$

- `first` A returns "artie".
- `next` A `"quinn"` or `next` S `"mike"` returns `"rachel"`.
- `getRange` A `"blain","quinn"` or `getRange` A `"burt","mike"` returns

$$\{\texttt{"burt"}, \texttt{"finn"}, \texttt{"mike"}\}.$$

- `rank` A `"rachel"` or `rank` A `"quinn"` returns the number of elements less than

Cost specification. We can implement ordered sets and tables using binary search trees. 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 to traverse right branches.

Exercise 13.24. Describe how to implement `previous` and `next` using the other ordered set functions.

To implement `split` and `join`, we can use the same operations as supplied by binary search trees. The `getRange` operation can easily be implemented with two calls to `split` To implement efficiently `rank`, `select` and `splitRank`, we can augment the underlying binary search tree implementation with sizes as described in (Chapter 12).

Cost Specification 13.25. [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 13.7) and tables (Cost Specification 13.13) for the operations supported by sets and tables. The work and span for all the operations in ADT 13.22 is $O(\lg n)$, 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.

13.7 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 13.26. [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 13.22 and the following operation

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

The `reduceVal` A function just returns the sum of all values in A 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, 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.

Example 13.27. [Analyzing Profits at **TRAM⋆LAW®**]

Let's 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. The sale record 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 restrict the sum in any range:

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

This will take $O(\lg n)$ work, which is much cheaper than n. Now let's 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 $365 \times O(\lg n)$, which is still much cheaper than looking at all data over the past year.

Example 13.28. [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 unioned 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 $O(\lg n)$ work.

13.8 Problems

13-1 Cost of bulk operations
Show that the work of the bulk operations, `intersection`, `difference`, and `union` is $O(n+m)$, where n and m are the sum of the sizes of the inputs.

13-2 Implementing `fromSeq`
What is the work and span of each of the implementations of `fromSeq` in Example 13.8

13-3 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.

13-4 Cost of Table Collect
The following code implements `Table.collect`(S).

```
Seq.reduce (Table.union 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.

13-5 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

```
identifyRepeats ⟨7, 2, 4, 2, 3, 2, 1, 3⟩
```

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

13-6 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 ⟨{5, 7}, {3, 1, 5, 2}, {2, 5, 8}⟩
```

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.

13-7 Range query
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.

13-8 Intervals

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?

13-9 Domain search

Continuing on our document indexing example discussed in Section 13.5, suppose that you wish to extend the kind of queries to include domain search. A domain search specifies a "domain" that identifies a subset of the documents and requires performing all the queries with respect to that domain. For example, the domain search "site:cs.cmu.edu parallel algorithms" on a search engine would search all pages that are in the "cs.cmu.edu" domain for the keywords "parallel algorithms".

.