Lecture 8 — Sets and Tables

Parallel and Sequential Data Structures and Algorithms, 15-210 (Qatar-Spring 2014)

Lectured by Kemal Oflazer — 4 February 2014

Today:

- Sets
- Tables
- Example of Tables for indexing the web
- Single Threaded Sequences

1 An Abstract Data Type for Sets

Sets undoubtedly play an important role in mathematics and are often needed in the implementation of various algorithms.

Question 1.1. *Can you think of a reason why you would prefer using a set data type instead of a sequence?*

Whereas a sequence is an ordered collection, a set is its *unordered* counterpart. In addition, a set has no duplicate element. In some algorithms and applications, ordering is not necessary and it may be important to make sure that there are no duplicates.

Specification of sets. We now define an abstract data type for sets. The definition follows the mathematical definition of sets from set theory and is purely functional. In particular, when updating a set (e.g. with insert or delete) it returns a new set rather than modifying the old set.

†Lecture notes by Umut Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, with additional edits by Kemal Oflazer

Definition 1.2. 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 $\mathbb{U}$) along with the following functions:

`empty`	:	$\mathbb{S}$	=	$\emptyset$
`size`(S)	:	$\mathbb{S} \to \mathbb{N}$	=	$\lvert S \rvert$
`singleton`(e)	:	$\mathbb{U} \to \mathbb{S}$	=	$\{e\}$
`filter`(f,S)	:	$((\mathbb{U} \to \{\mathrm{T},\mathrm{F}\}) \times \mathbb{S}) \to \mathbb{S}$	=	$\{s \in S \mid f(s)\}$
`find`(S,e)	:	$\mathbb{S} \times \mathbb{U} \to \{\mathrm{T},\mathrm{F}\}$	=	$\lvert\{s \in S \mid s = e\}\rvert = 1$
`insert`(S,e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$	=	$S \cup \{e\}$
`delete`(S,e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$	=	$S \setminus \{e\}$
`intersection`(S_1,S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cap S_2$
`union`(S_1,S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cup S_2$
`difference`(S_1,S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \setminus S_2$

where $\mathbb{N}$ are the natural numbers (non-negative integers).

Question 1.3. *Can you see a redundancy in this interface?*

Note that the *bulk operations*, `intersection, union, difference` can be implemented in terms of the operations `find, insert, delete`. Indeed, they are the bulk versions of these operations.

- `intersection` — search for multiple elements instead of one.
- `union` — insert multiple elements.
- `difference` — delete multiple elements.

Question 1.4. *Can you implement the non-bulk operations in terms of the bulk operations?*

We can implement `find`, `insert`, and `delete` in terms of the others.

$$\begin{aligned} \texttt{find}(S,e) &= \texttt{size}(\texttt{intersection}(S,\texttt{singleton}(e))) = 1 \\ \texttt{insert}(S,e) &= \texttt{union}(S,\texttt{singleton}(e)) \\ \texttt{delete}(S,e) &= \texttt{difference}(S,\texttt{singleton}(e)) \end{aligned}$$

Question 1.5. *Can you see a way to implement these bulk operations in terms of the non-bulk versions?*

One simple idea would be to perform the bulk operations one by one, one after the other.

Question 1.6. *Do you see a disadvantage to this approach?*

The problem is that it doesn't leave much opportunity for parallelism.

It turns out these operations can be performed in parallel more efficiently than with using the non-bulk versions. We will not talk about the parallel-efficient implementations of these operations in this class—but later we will. But we will talk about their cost specifications.

Remark 1.7. We write this definition to be generic and not specific to Standard ML. In our library, the type $\mathbb{S}$ is called `set` and the type $\mathbb{U}$ is called `key`, the arguments are not necessarily in the same order, and some of the functions are curried. For example, the interface for find is `find` : `set` → `key` → `bool`. Please refer to the documents for details. In the pseudocode, we will give in class and in the notes we will use standard set notation as in the right hand column of the table above.

Remark 1.8. 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 1.9. 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.

Cost specification for sets. So far, we have laid out a semantic interface, but before we can put it to use, we need to worry about the cost specification.

As we have discussed before, cost specifications depend on implementation.

Question 1.10. *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. We will cover how to implement these set operations when we talk about balanced trees later in the course. For now, a good intuition to have is that we use a comparison function to keep the elements in sorted order in a balanced tree.

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 1.11. *We have the following cost specification:*

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

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

Question 1.12. *Can you see the why the bulk operations are more work and span efficient?*

If we perform for example $|S_2|$ applications of each of these operations, the work and span would be $|S_2|\log|S_1|$ (ignoring the cost of the comparison). This is significantly worse that what we would obtain with the bulk operations, which has has a vastly improved span (by a linear factor).

The work of the bulk operations might be somewhat more difficult to see. The key point to realize is that the bound is linear in the smaller of the two sets and logarithmic in the ratio of the two sets.

This means that if the ratio is very uneven, then it behaves very much like the naive algorithm that performs the non-bulk version repeatedly. For example, when one of the sets is a singleton, then the work is $O(\log n)$.

However, if the ratio is close to even, then it can reduce the total work to linear. When $n = m$, the work is simply

$$O(C_w \cdot m \cdot \log(1+1)) = O(C_w \cdot n).$$

This should not be surprising, because it corresponds to the cost of merging two approximately equal length sequences (effectively what these operations have to do).

In fact, these bounds turn out to be both the theoretical upper and lower bounds for any comparison-based implementation of sets. We will get to this later.

Below is the plot for this function.

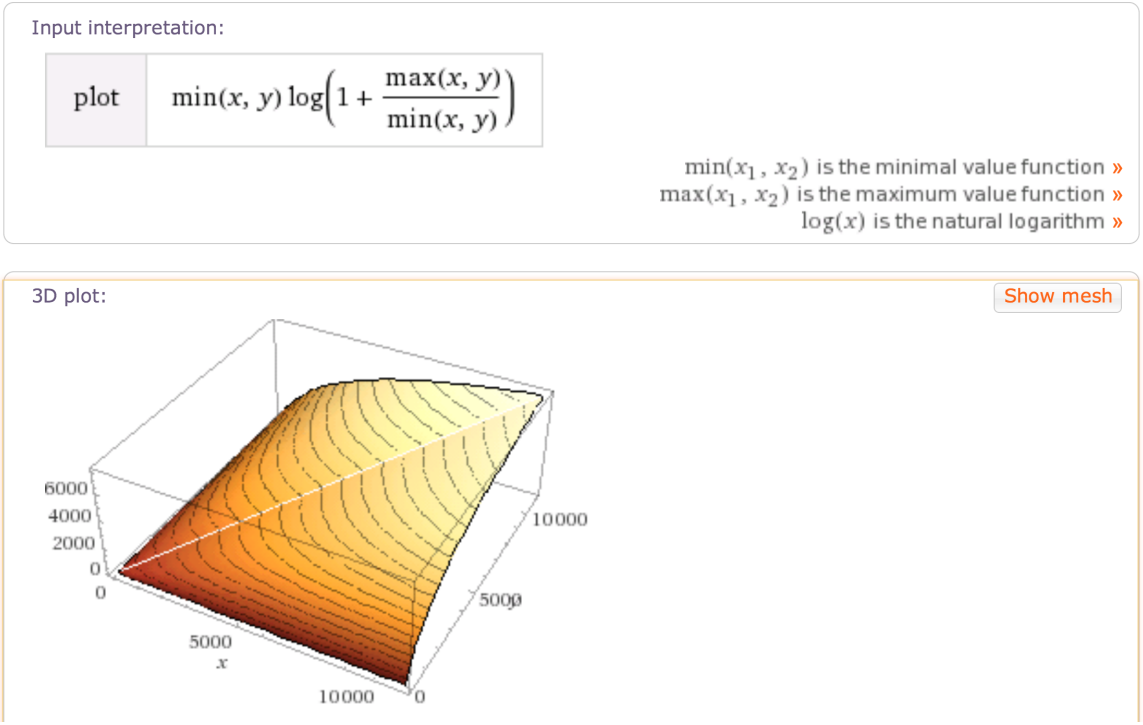


Figure 1: The work of the intersection, union, difference functions.

Exercise 1. *Draw a two dimensional version of this figure by assuming one of S_1 to be of fixed size and varying the size of S_2 (or vice versa).*

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

```
fun fromSeq(S) = Seq.iter (fn (S',e) => Set.insert e S') Set.empty
S
```

However, the above is sequential. To do it in parallel we could instead do

```
fun fromSeq(S) = Seq.reduce Set.union Set.empty (Seq.map
Set.singleton S)
```

Exercise 2. *What is the work and span of the first version of* `fromSeq`.

Exercise 3. *Show that on a sequence of length n the second version of* `fromSeq` *does $O(C_w n \log n)$ work and $O(\log^2 n)$ span.*

Summary 1.14. *We talked about sets.*

- *Unordered collection.*
- *Unique elements.*
- *Supports efficient, find, insert, delete, operations serially and in parallel (bulk).*

2 Tables: Assigning A Value to each Key

Since the elements of a set are unique (no duplicates), we can think of them as keys.

In many applications, it is important to be able to assign a value or data to each key. A *table* is an abstract data type that stores for each key data associated with it.

A table essentially stores *key-value* pairs in such a way that we can perform a range of operations quickly, e.g., finding the value for a key, inserting new key-value pairs, and deleting keys (and their values).

Question 2.1. *Have you seen similar data structures before? Do you recall what they are called?*

Tables are common data structures. They are also called dictionaries, associative arrays, maps, mappings, and functions (in set theory).

Most languages have tables 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). We note that the interfaces for these languages and libraries have common features but typically differ in some important ways, so be warned. Most do not support the "parallel" operations we discuss.

Here we will define tables mathematically in terms of set theory before committing to a particular language.

Specifying tables. The specification of tables is quite similar to sets.

Definition 2.2. A table is set of key-value pairs where each key appears only once in the set.

We will use the following notation for a table

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

where we have *keys* and *values*—and each key k_i mapped to a value v_i.

We choose this notation because that it makes clear that we are using tables rather than some other data structure such as sets.

Question 2.3. *Can you see how we might represent tables as sets?*

We can else represent a table as a set of key value pairs, e.g., $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$. Since keys are unique so are key-value pairs.

Question 2.4. *Does this way of viewing maps remind you of a mathematical object?*

Such sets are called *functions* in set theory since they map each key to a single value. We avoid this terminology so that we don't confuse it with functions in a programming language. However, note that the `(find T)` in the interface is precisely the "function" defined by the table `T`. In fact it is a *partial function* since the table might not contain all keys and therefore the function might not be defined on all inputs.

Here is the definition of a table.

Definition 2.5. 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:

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

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

Question 2.6. *Can you see some differences between sets and tables?*

There are several differences between sets and tables.

- The `find` function does not return a Boolean, but instead it returns the value associated with the key k. As it may not find the key in the table, its result may be bottom ($\bot$). For this reason, in the Table library, the interface for find is `find` : `'a table` → `key` → `'a option`, where `'a` is the type of the values.
- 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.

For this reason, the `insert` function takes a function f as an argument,

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

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.
- We also introduce new specification-language notation for `map` and `filter` on tables:

  $$\{k \mapsto f(v) : (k \mapsto v) \in T\}$$

  is equivalent to $\texttt{map}(f, T)$ and

  $$\{(k \mapsto v) \in T \mid p(k, v)\}$$

  is equivalent to $\texttt{filter}(p, T)$.

Specifying the cost of tables. The costs of the table operations are very similar to sets.

Cost Specification 2.7.

	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)	$O\Big(\sum_{(k\mapsto v)\in T} W(f(v))\Big)$	$O\Big(\max_{(k\mapsto v)\in T} S(f(v))\Big)$
`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 $n = \max(|T_1|,|T_2|)$ *and* $m = \min(|T_1|,|T_2|)$.

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 these three.

Remark 2.8. We note that, in the SML Table library we supply, the functions are polymorphic (accept any type) over the values but not the keys. In particular the signature starts as:

```
 1  signature TABLE =
 2  sig
 3    type 'a table
 4    type 'a t = 'a table
 5    structure Key : EQKEY
 6    type key = Key.t
 7    structure Seq : SEQUENCE
 8    type a seq = 'a Seq.seq
 9    type set = unit table
10  ...
11    val find : 'a table -> key -> 'a option
12  ...
```

The `'a` in `'a table` refers to the type of the value. The key type is fixed to be `key`. Therefore there are separate table structures for different keys (e.g. `IntTable`, `StringTable`). The reason to do this is because all the operations depend on the key type since keys need to be compared for a tree implementation, or hashed for a hash table implementation. Also note that the signature defines `set` to be a `unit table`. Indeed a set is just a special case of a table where there are no values.

Remark 2.9. In the SML Table library, we supply 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. This is equivalent to using a sequence collect followed by a `Table.fromSeq`. Alternatively, it can be implemented as

```
1  fun collect(S) =
2  let
3      S' = ⟨{k ↦ ⟨v⟩} : (k, v) ∈ S⟩
4  in
5    Seq.reduce (Table.merge Seq.append) {} S'
6  end
```

Exercise 4. *Figure out what this code does.*

Remark 2.10. Tables are similar to sets: they extend sets so that each key now carries a value. Their cost specification and implementations are also similar. In fact, in 15210 SML library, tables are implemented simply as sets where each key carries unit (constant) value.

3 Example: Bingle® It

Question 3.1. *Can you think of some applications of sets and tables?*

There are many. Here we consider an application of sets and tables to searching a corpus of documents.

In particular lets say one night, late, while avoiding doing your 210 homework you come up with a great idea: provide a service that indexes all the pages on the web so that people can search them by keywords. You figure a good name for such a service would be **Bingle**®.

Question 3.2. *Can you think of a simple algorithm to solve the problem.*

Assuming that you have a copy of the Internet is some format, you can traverse it every time a query comes in.

Question 3.3. *Going back to one of lectures on designing algorithms, can you see the problem with that?*

The problem is redundancy. Every time a search comes is you have to traverse the whole data.

Instead, a better thing to do would be to build an *index* and use the index to avoid the redundant work. The idea would be for the index to organize the data in such a way to make searching efficient. You want queries to be fast since people will be running them all the time. On the other hand, the index can be slower to build, because you will run it every once in a while to keep your data up to date.

Question 3.4. *Can you thing of the types of queries that you would wish to provide for?*

You may want to support are logical queries on words involving `And`, `Or`, and `AndNot`. For example a query might look like

"CMU" `And` "fun" `And` ("courses" `Or` "clubs")

and it would return a list of web pages that match the query (*i.e.*, contain the words "CMU", "fun" and either "courses" or "clubs"). This list would include the 15-210 home page, of course.

Remark 3.5. This idea has been thought of before. Indeed these kinds of searchable indexes date back to the 1970s with systems such as Lexis for searching law documents. Today, beyond web searches, searchable indices are an integral part of most mailers and operating systems. The different indices support somewhat different types of queries. For example, by default Google supports queries with `And` and `adjacent to` but with their advanced search you can search with `Or`, `AndNot` as well as other types of searches.

Example 3.6. *As a simple set of documents, we can consider the tweets made by some of your friends yesterday.*

$T = \langle$ *("jack", "chess club was fun"),*
("mary", "I had a fun time in 210 class today"),
("nick", "food at the cafeteria sucks"),
("sue", "In 217 class today I had fun reading my email"),
("peter", "I had fun at nick's party"),
("john", "tiddlywinks club was no fun, but more fun than 218"),
$\rangle$

where the identifiers are the names, and the contents is the tweet.
On this set of documents, searching for "fun" and "club" would return "jack", "mary", "sue", "peter", and "john"; lots of fun. (Note that our search returned "john" as well, even though he wasn't having that much fun.)

You can imagine that you would want to support an interface such as the following.

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

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

The input to `makeIndex` is a sequence of pairs each consisting of a document identifier (e.g. the URL) and the contents of the document as a single text string.

Example 3.7. *Continuing on our example, we can use the interface to make an index of these tweets:*

$$f = (\texttt{find (makeIndex}(T))) : \texttt{word} \to \texttt{docs}$$

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 "class", f "club")))
⇒ ⟨"jack", "mary", "sue", "john"⟩
```

returns all the documents (tweets) that contain "fun" and either "class" or "club".
The code,

```
size(AndNot(f "fun", f "tiddlywinks"))
⇒ 4
```

returns the number of documents that contain "fun" and not "tiddlywinks".

Question 3.8. *Can you think of a way to implement this interface? Let's start with the* `makeIndex` *function.*

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

```
1  fun makeIndex(docs) =
2  let
3    fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
4      Pairs = flatten⟨ tagWords(d) : d ∈ docs ⟩
5      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 3.9. *Here is an example of how* `tagWords` *works:*

`tagWords`(*"jack", "chess club was fun"*)
$\Rightarrow$ $\langle$(*"chess","jack"*),(*"club", "jack"*), (*"was", "jack"*), (*"fun", "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:

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

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

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

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 3.10. *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:

fun `find` $T\ v = $ `Table.find` $T\ v$

fun `And`$(s_1, s_2) = s_1 \cap s_2$

fun `Or`$(s_1, s_2) = s_1 \cup s_2$

fun `AndNot`$(s_1, s_2) = s_1 \setminus s_2$

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 `makeIndex` 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:

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

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.

4 Single-Threaded Array Sequences

In this course we will be using purely functional code because it is safe for parallelism, easier to reason about formally, and because its cool. For many algorithms using the purely functional version makes no difference in the asymptotic work bounds—for example quickSort and mergeSort use $\Theta(n \log n)$ work (expected case for quickSort) whether purely functional or imperative. However, in some cases purely functional implementations lead to up to a $O(\log n)$ factor of additional work. To avoid this we will slightly cheat in this class and allow for benign "effect" under the hood in exactly one ADT, described in this section. These effects do not affect the observable values (you can't observe them by looking at results), but they do affect cost analysis—and if you sneak a peak at our implementation, you will see some side effects.

The issue has to do with updating positions in a sequence. In an imperative language updating a single position can be done in "constant time". In the functional setting we are not allowed to change the existing sequence, everything is persistent. This means that for a sequence of length n an update can either be done in $O(n)$ work with an arraySequence (the whole sequence has to be copied before the update) or $O(\log n)$ work with a treeSequence (an update involves traversing the path of a tree to a leaf). In fact you might have noticed that our sequence interface does not even supply a function for updating a single position. The reason is both to discourage sequential computation, but also because it would be expensive.

Consider a function `update` (i, v) S that updates sequence S at location i with value v returning the new sequence. This function would have cost $O(|S|)$ in the arraySequence cost specification. Someone might be tempted to write a sequential loop using this function. For example for a function $f : \alpha \to \alpha$, a `map` function can be implemented as follows:

```
fun map f S =
  iter (fn ((i, S'), v) ⇒ (i + 1, update (i, f(v)) S'))
       (0, S)
       S
```

This code iterates over S with i going from 0 to $n-1$ and at each position i updates the value S_i with $f(S_i)$. The problem with this code is that even if f has constant work, with an `arraySequence` this will do $O(n^2)$ total work since every update will do $O(n)$ work. By using a `treeSequence` implementation we can reduce the work to $O(n \log n)$ but that is still a factor of $O(\log n)$ off of what we would like.

In the class we sometimes do need to update either a single element or a small number of elements of a sequence. We therefore introduce an ADT we refer to as a *Single Threaded Sequence* (`stseq`).

Although the interface for this ADT is quite straightforward, the cost specification is somewhat tricky. To define the cost specification we need to distinguish between the latest "copy" of an instance of an `stseq`, and earlier copies. Basically whenever we update a sequence we create a new "copy", and the old "copy" is still around due to the persistence in functional languages. The cost specification is going to give different costs for updating the latest copy and old copies. Here we will only define the cost for updating the latest copy, since this is the only way we will be using an `stseq`. The interface and costs are as follows:

	Work	Span
`fromSeq(S) : α seq → α stseq` Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq(ST) : α stseq → α seq` Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth ST` i `: α stseq → int → α` Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` (i,v) S `: (int × α) → α stseq → α stseq` Replaces the i^{th} element of S with v.	$O(1)$	$O(1)$
`inject` I S `: (int × α) seq → α stseq → α stseq` For each $(i,v) \in I$ replaces the i^{th} element of S with v.	$O(\|I\|)$	$O(1)$

An `stseq` is basically a sequence but with very little functionality. Other than converting to and from sequences, the only functions are to read from a position of the sequence (`nth`), update a position of the sequence (`update`) or update multiple positions in the sequence (`inject`). To use other functions from the sequence library, one needs to convert an `stseq` back to a sequence (using `toSeq`).

In the cost specification, the work for both `nth` and `update` is $O(1)$, which is about as good as we can get. Again, however, this is only when S is the latest version of a sequence (i.e., no one else has updated it). The work for `inject` is proportional to the number of updates. It can be viewed as a parallel version of `update`.

Now with an `stseq` we can implement our map as follows:

```
1  fun map f S = let
2      S' = StSeq.fromSeq(S)
3      R = iter (fn ((i,S''),v) ⇒ (i+1, StSeq.update (i,f(v)) S''))
4               (0,S')
5               S'
6  in
7    StSeq.toSeq(R)
8  end
```

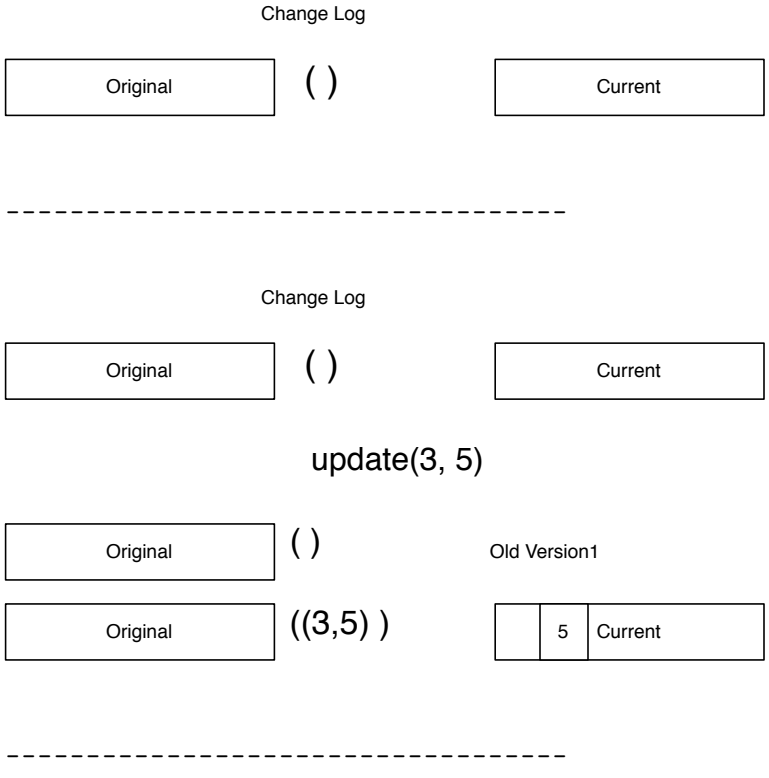
This implementation first converts the input sequence to an `stseq`, then updates each element of the `stseq`, and finally converts back to a sequence. Since each update takes constant work, and assuming

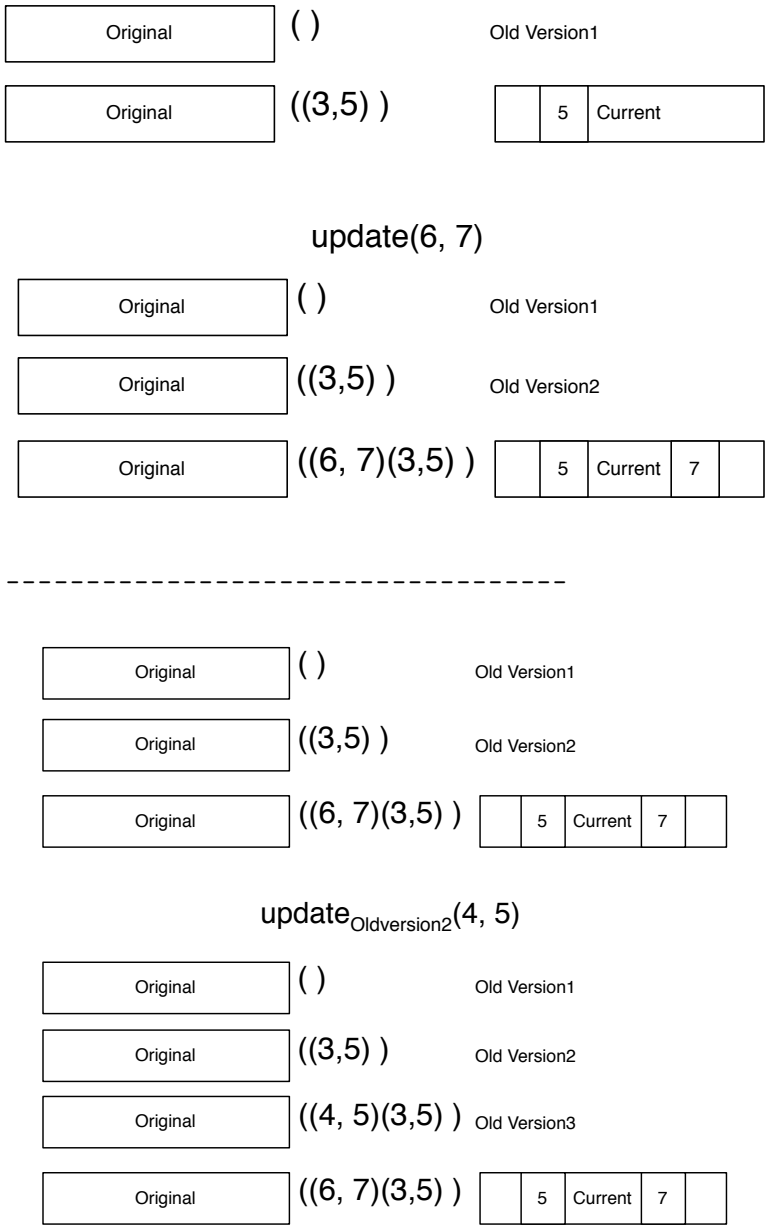
the function f takes constant work, the overall work is $O(n)$. The span is also $O(n)$ since `iter` is completely sequential. This is therefore not a good way to implement `map` but it does illustrate that the work of multiple updates can be reduced from $O(n^2)$ on array sequences or $O(n \log n)$ on tree sequences to $O(n)$ using an `stseq`.

Implementing Single Threaded Sequences. You might be curious about how single threaded sequences can be implemented so they act purely functional but match the cost specification. Here we will just briefly outline the idea.

The trick is to keep two copies of the sequence (the original and the current copy) and additionally to keep a "change log". The change log is a linked list storing all the updates made to the original sequence. When converting from a sequence to an `stseq` the sequence is copied to make a second identical copy (the current copy), and an empty change log is created. A different representation is now used for the latest version and old versions of an `stseq`. In the latest version we keep both copies (original and current) as well as the change log. In the old versions we only keep the original copy and the change log. Let's consider what is needed to update either the current or an old version. To update the current version we modify the current copy in place with a side effect (non functionally), and add the change to the change log. We also take the previous version and mark it as an old version removing its current copy. When updating an old version we just add the update to its change log. Updating the current version requires side effects since it needs to update the current copy in place, and also has to modify the old version to mark it as old and remove its current copy.

Here is a sequence of figure that describes how updates work on single threaded sequences. Note that throughout there really is only one copy of the *Original* and all point to that copy.





Either updating the current version or an old version takes constant work. The problem is the cost of `nth`. When operating on the current version we can just look up the value in the current copy, which is up to date. When operating on an old version, however, we have to go back to the original copy and then check all the changes in the change log to see if any have modified the location we are asking about. This can be expensive. This is why updating and reading the current version is cheap ($O(1)$ work) while working with an old version is expensive.

In this course we will use `stseqs` for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.

The issue has to do with updating positions in a sequence. In an imperative language updating a single position can be done in "constant time". In the functional setting we are not allowed to change

the existing sequence, everything is persistent. This means that for a sequence of length n an update can either be done in $\Theta(n)$ work with an arraySequence (the whole sequence has to be copied before the update) or $\Theta(\log n)$ work with a treeSequence (an update involves traversing the path of a tree to a leaf). In fact you might have noticed that our sequence interface does not even supply a function for updating a single position. The reason is both to discourage sequential computation, but also because it would be expensive.

Consider a function `update` (i, v) S that updates sequence S at location i with value v returning the new sequence. This function would have cost $\Theta(|S|)$ in the arraySequence cost specification. Someone might be tempted to write a sequential loop using this function. For example for a function $f : \alpha -> \alpha$, a `map` function can be implemented as follows:

```
fun map f S =
  iter (fn ((i,S'),v) => (i+1, update (i, f(v)) S'))
       (0,S)
       S
```

This code iterates over S with i going from 0 to $n-1$ and at each position i updates the value S_i with $f(S_i)$. The problem with this code is that even if f has constant work, with an `arraySequence` this will do $\Theta(|S|^2)$ total work since every update will do $\Theta(|S|)$ work. By using a `treeSequence` implementation we can reduce the work to $\Theta(|S| \log |S|)$ but that is still a factor of $\Theta(\log |S|)$ off of what we would like.

In the class we sometimes do need to update either a single element or a small number of elements of a sequence. We therefore introduce an ADT we refer to as a *Single Threaded Sequence* (`stseq`). Although the interface for this ADT is quite straightforward, the cost specification is somewhat tricky. To define the cost specification we need to distinguish between the latest "copy" of an instance of an `stseq`, and earlier copies. Basically whenever we update a sequence we create a new "copy", and the old "copy" is still around due to the persistence in functional languages. The cost specification is going to give different costs for updating the latest copy and old copies. Here we will only define the cost for updating the latest copy, since this is the only way we will be using an `stseq`. The interface and costs is as follows:

	Work	Span
`fromSeq(S) : α seq → α stseq` Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq(ST) : α stseq → α seq` Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth ST i : α stseq → int → α` Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update (i,v) ST : (int × α) → α stseq → α stseq` Replaces the i^{th} element of ST with v.	$O(1)$	$O(1)$
`inject I ST : (int × α) seq → α stseq → α stseq` For each $(i, v) \in I$ replaces the i^{th} element of ST with v.	$O(\|I\|)$	$O(1)$

An `stseq` is basically a sequence but with very little functionality. Other than converting to and from sequences, the only functions are to read from a position of the sequence (`nth`), update a position of the sequence (`update`) or update multiple positions in the sequence (`inject`). To use other functions from the sequence library, one needs to covert an `stseq` back to a sequence (using `toSeq`).

In the cost specification the work for both `nth` and `update` is $O(1)$, which is about as good as we can get. Again, however, this is only when S is the latest version of a sequence (i.e. noone else has updated it). The work for `inject` is proportional to the number of updates. It can be viewed as a parallel version of `update`.

Now with an `stseq` we can implement our map as follows:

```
1  fun map f S = let
2      S' = StSeq.fromSeq(S)
3      R = iter (fn ((i,S''),v) ⇒ (i+1, StSeq.update (i,f(v)) S''))
4               (0,S')
5               S
6  in
7    StSeq.toSeq(R)
8  end
```

This implementation first converts the input sequence to an `stseq`, then updates each element of the `stseq`, and finally converts back to a sequence. Since each update takes constant work, and assuming the function f takes constant work, the overall work is $O(n)$. The span is also $O(n)$ since `iter` is completely sequential. This is therefore not a good way to implement `map` but it does illustrate that the work of multiple updates can be reduced from $\Theta(n^2)$ on array sequences or $O(n \log n)$ on tree sequences to $O(n)$ using an `stseq`.

Implementing Single Threaded Sequences. You might be curious about how single threaded sequences can be implemented so they act purely functional but match the cost specification. Here we will just briefly outline the idea.

The trick is to keep two copies of the sequence (the original and the current copy) and additionally to keep a "change log". The change log is a linked list storing all the updates made to the original sequence. When converting from a sequence to an `stseq` the sequence is copied to make a second identical copy (the current copy), and an empty change log is created. A different representation is now used for the latest version and old versions of an `stseq`. In the latest version we keep both copies (original and current) as well as the change log. In the old versions we only keep the original copy and the change log. Lets consider what is needed to update either the current or an old version. To update the current version we modify the current copy in place with a side effect (non functionally), and add the change to the change log. We also take the previous version and mark it as an old version removing its current copy. When updating an old version we just add the update to its change log. Updating the current version requires side effects since it needs to update the current copy in place, and also has to modify the old version to mark it as old and remove its current copy.

Either updating the current version or an old version takes constant work. The problem is the cost of `nth`. When operating on the current version we can just look up the value in the current copy, which is up to date. When operating on an old version, however, we have to go back to the original

copy and then check all the changes in the change log to see if any have modified the location we are asking about. This can be expensive. This is why updating and reading the current version is cheap ($O(1)$ work) while working with an old version is expensive.

In this course we will use `stseq`s for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.

5 SML Code

5.1 Indexes

```
functor TableIndex(Table : TABLE where type Key.t = string) : INDEX =
struct

structure Seq = Table.Seq
structure Set = Table.Set

type word = string
type docId = string
type 'a seq = 'a Seq.seq
type docList = Table.set
type index = docList Table.table

fun makeIndex docs =
let
  fun toWords str = Seq.tokens (fn c => not (Char.isAlphaNum c)) str

  fun tagWords(docId,str) = Seq.map (fn t => (t, docId)) (toWords str)

  (* generate all word-documentid pairs *)
  val allPairs = Seq.flatten (Seq.map tagWords docs)

  (* collect them by word *)
  val wordTable = Table.collect allPairs

in
  (* convert the sequence of documents for each word into a set
     which removes duplicates*)
  Table.map Set.fromSeq wordTable
end

fun find Idx w =
   case (Table.find Idx w) of
      NONE => Set.empty
    | SOME(s) => s

val And = Set.intersection
val AndNot = Set.difference
val Or = Set.union
val size = Set.size
```

```
val toSeq = Set.toSeq

end
```