Lecture 8 — Sets and Tables II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Umut Acar — 7 February 2013

Today:

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

1 Tables: Associating Each Element With A Value

Suppose we want to extend sets so that each element is associated with a payload. A table is an abstract data type that stores for each key data associated with it. The table ADT supplies operations for finding the value associated with a key, for inserting new key-value pairs, and for deleting keys and their associated value. Tables are also called dictionaries, associative arrays, maps, mappings, and functions (in set theory). Given our focus on parallelism, the interface we will discuss also supplies "parallel" operations that allow the user to insert multiple key-value pairs, to delete multiple keys, and to find the values associated with multiple keys.

In this class, the notation we are going to be using is

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

where we have *keys* and *values*—and each key k_i is associated with the value v_i. Mathematically, a table is simply a set of pairs and can therefore be written as a set of ordered pairs $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$. Our notation choice is largely to better identify when tables are being used.

As with sets, tables are commonly used in many applications. 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. Again, here we will define tables mathematically in terms of set theory before committing to a particular language.

Formally, a table is set of key-value pairs where each key appears only once in the set. Such sets are called *functions* in set theory since they map each key to a single value. We will 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.

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

Definition 1.1. 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:

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

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).

Distinct from sets, 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.

Unlike sets, when we insert an element, we can't simply ignore that element if it is already present—their values might be different. For this reason, the `insert` function takes a function $f : \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ as an argument. 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 `merge` function, which takes a similar function since it also has to consider the case that an element appears in both tables.

We also introduce new pseudocode notation for `map` and `filter` on tables:

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

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

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

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

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

	Work	*Span*
`size`(T) `singleton`(k,v)	$O(1)$	$O(1)$
`filter`(f,T)	$O\Big(\sum_{(k\mapsto v)\in T} W(f(v))\Big)$	$O\Big(\log\|T\| + \max_{(k\mapsto v)\in T} S(f(v))\Big)$
`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)	$O(C_w \log\|T\|)$	$O(C_s \log\|T\|)$
`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 the earlier three. 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 `merge` operation can add multiple values to a table in parallel by merging two tables. The `erase` operation can delete multiple values from a table in parallel.

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:

```
signature TABLE =
sig
  type 'a table
  type 'a t = 'a table
  structure Key : EQKEY
  type key = Key.t
  structure Seq : SEQUENCE
  type a seq = 'a Seq.seq
  type set = unit table
  ...
  val find : 'a table -> key -> 'a option
  ...
```

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.

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

Exercise 1. *Figure out what this code does.*

2 Example: Bingle® It

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®**. The idea is to support a function that makes the index from the documents, which is run just once (or once in a while) so it can take a while. On the other hand you want queries to the database to be fast since people will be running them all the time. The type of queries you 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.

OK, so maybe 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.

Let's imagine you want to support the following interface

```
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 -> docs
  val And : docs * docs -> docs
  val AndNot : docs * docs -> docs
  val Or : docs * docs -> docs
  val size : docs -> int
  val toSeq : docs -> 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. So for example we might want to index recent tweets, that might include the following "documents":

$T = \langle$ ("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.

The interface can be used to make an index of these tweets:

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

In addition to making the index, this partially applies `find` on it. We can then use this index for various queries. For example:

```
 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", and

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

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

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

```
1  function makeIndex(docs) =
2  let
3    function 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. For example, on the document

```
tagWords("jack", "chess club was fun")
⇒  ⟨ ("chess","jack"),("club", "jack"), ("was", "jack"), ("fun", "jack") ⟩
```

The function `tagWords` is then applied to all document and the result flattened so it is a single sequence. In our example the result would start as:

```
Pairs = ⟨("chess","jack"),("club", "jack"), ("was", "jack"),
         ("fun", "jack"), ("I", "mary"), ("had", "mary"), ("fun", "mary"), ...
```

The `Table.collect` then collects the entries by word creating a sequence of matching documents. In our example it would start:

```
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.

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. The rest of the interface can be implemented as follows:

function `find` T v = `Table.find` T v

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

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

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

function `size`$(s) = |s|$

function `toSeq`(s) = `Set.toSeq`(s)

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:

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

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.

3 Single-Threaded Array Sequences

In this course we will be using purely functional code because it is safe for parallelism and enables higher-order design of algorithms by use of higher-order functions. It is also easier to reason about formally, and is just 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 $\Theta(n)$ 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:

```
function 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) 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 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  function 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.

4 SML Code

4.1 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
```