Homework 10: ADTs
Due: 10:00 PM, Nov 14, 2017

Contents

1 Lists (Practice) 2

2 Dictionaries 3

2.1 Dictionary Signature ........................................... 3
2.2 List Dictionaries ............................................... 4
2.3 Tree Dictionaries ............................................... 4
2.3.1 Balanced Binary Search Trees (BSTs) ..................... 4
2.3.2 Sorting ..................................................... 5
2.3.3 Task ....................................................... 5
2.4 A Note on Modules ............................................. 6

3 Analysis 6

3.1 Big Theta ...................................................... 6
3.2 Keys Analysis .................................................. 6

Objectives

By the end of this homework you will be able to:

1. implement dictionaries as an abstract data type in OCaml
2. represent dictionaries in more than one concrete way
3. analyze procedures that recur on trees

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Dictionaries and Analysis questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem, as follows (e.g., sig_dictionary.ml corresponds to Dictionary Signature):
• README.txt
• CS17setup.ml
• sig_dictionary.ml
• list_dictionary.ml
• tree_dictionary.ml
• analysis.tex

For this assignment, you must hand in four OCaml files, which must end with extension .ml, and a \LaTeX-generated .tex file. If you are using a departmental Linux system, all your solution files should reside in your `~/course/cs0170/homeworks/hw10 directory.

For this and every assignment, you should also have a README.txt file whose first line contains only your CS-department email address, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:
 jfh@cs.brown.edu
There’s nothing to say except that I’m turning in these files plus this README the way the instructions say that I should.

To hand in your solutions to these problems, you must zip your hw10 directory into a file hw10.zip.

Hand in this zip file using the method you learned in the first lab: visit https://tinyurl.com/cs0170-handin to get started.

Set-Up

We have provided you with all of the files you will need for this homework assignment. The complete CS17setup file as well as outlines for the other files you will fill out for this assignment are in the hw10 src directory.

Before starting this assignment you’ll want to copy the contents of the hw10 src directory into your personal hw10 directory. On the department system, this would look something like this:

cp /course/cs0170/src/hw10/* ~/course/cs0170/homeworks/hw10

After you’ve transferred the files, begin filling in your solutions to the tasks on this homework. Make sure you do not modify any of the code we have written for you, especially not the #use statements. If you modify the template, there will be deductions.

Practice

1 Lists (Practice)

Here is a signature for lists.
module type LIST =
 sig
   type 'a list

   (* Constructs an empty list *)
   val empty : 'a list

   (* Determines whether a list is empty *)
   val is_empty : 'a list -> bool

   (* Constructs a new list from a datum and an old list *)
   val cons : 'a -> 'a list -> 'a list

   (* Determines whether a list is non-empty *)
   val is_cons : 'a list -> bool

   (* Deconstructs a list, and produces its head *)
   val head : 'a list -> 'a

   (* Deconstructs a list, and produces its tail *)
   val tail : 'a list -> 'a list

   (* Produces the nth element of a list *)
   val nth : 'a list -> int -> 'a

   (* Appends two lists together *)
   val append : 'a list -> 'a list -> 'a list

   (* Reverses a list *)
   val reverse : 'a list -> 'a list

   ...
 end

One possible representation of the type 'a list follows:

type 'a list =
 | Empty
 | Cons of 'a * 'a list

Task: Using this representation, implement the module type LIST.

Note: You can find this signature in /course/cs0170/src/hw10/sig_list.ml.

Problems

2 Dictionaries

Percy the Parrot, recognizing Circe’s algebra skills from Homework 8, has decided to make Circe a TA for his mimicry school! However, as Circe is a bulldog and is not skilled at mimicry, they have
been getting many grade complaints from the students. As Percy’s mimicry school does anonymous grading, Circe needs help looking up which student is associated with a given ID number.

To help Circe with their problem, we will be creating *dictionaries*, which will store each student’s anonymous ID (as an int) and each student’s name (as a string). A *dictionary* is a collection of key-value pairs. You will be implementing dictionaries in two ways: first with lists, and next using trees. For this homework, a dictionary is composed of (int * string) tuples.¹

### 2.1 Dictionary Signature

To start, we need to help Circe understand what functionality to expect from their dictionary.

**Task:** Write a module type that includes the following:

- A type `dict` for the dictionary.
- `dict_of_kvpair_list`, which constructs a dictionary from a list of (int * string) tuples. The tuples within this input list are guaranteed to never have the same key (that is, the ints cannot be the same).
- `containsP`, which takes in a dictionary and a int and returns a boolean: true if the int is a key in the dictionary, and false otherwise.
- `lookup`, which takes in a dictionary and a int and produces a string option: Some of the corresponding value, if the int is a key in the dictionary, and None otherwise.
- `keys`, which takes in a dictionary and returns a list of all the keys (which are ints) in the dictionary. Note that the list of all keys need not be in any particular order.

### 2.2 List Dictionaries

**Task:** Implement your module using lists of (int * string) tuples to represent dictionaries. Specifically, use this type:

```ocaml
type dict = (int * string) list
```

In testing your `TestListDictionary`, you’ll need to test its `keys` procedure. What if the keys are 1 and 4? Then the keys procedure could return either [1; 4] or 4; 1]. What should you test against? Answer: if you *sort* the keys, you can just compare against a single standard answer, namely [1; 4]. OCaml includes a sorting procedure in its List module, but it requires, as a first argument, a function of two arguments *x* and *y* that produces −1 if *x* < *y*, 0 if *x* = *y*, and +1 if *x* > *y*. Fortunately, the designers of OCaml have included a function of that sort that works on ints, called `compare`. So to sort a list of ints, you write

```ocaml
# List.sort compare [4; 1];;
- : int list = [1; 4]
```

¹This can be generalized to 'a * 'b tuples but we won’t do that in this homework.

²So that Circe can keep track of who needs to be graded without seeing the students’ names!
That means that to test your keys procedure, you might write something like this:

```ocaml
check_expect (List.sort compare (TestListDict.keys myDict)) [1; 3; 4; 6];;
```

All the other tests should be straightforward.

### 2.3 Tree Dictionaries

#### 2.3.1 Balanced Binary Search Trees (BSTs)

Although list-based dictionaries will meet Circe’s dictionary specification, they are not necessarily the most efficient choice. In the list implementation, in order to look up an individual key in the tree, Circe might have to go through every other key in the dictionary, taking linear time. Another way to represent a dictionary is to use balanced binary search trees, which would make this procedure run in \(\log n\) time.

We went over balanced BSTs in [Lecture 23](#). To review, if a binary tree is a leaf, then it is automatically a BST. If the tree is a node, then the following invariants must hold:

- Every value in the nodes left subtree must be less than the nodes value; and
- Every value in the nodes right subtree must be greater than the nodes value.

To make an effective search tree, it is important to make the tree balanced. In other words, the left subtree and right subtree of a node should be roughly the same size. So, for a balanced BST of the integers \([1 ; 2 ; 3 ; 4 ; 5]\), you would choose 3 to be your top node so that 1 and 2 go into the left subtree and 4 and 5 go into the right subtree, thus keeping both sides with the same size.

#### 2.3.2 Sorting

Since our values here are tuples, not integers as from lecture 23, you might be wondering what it means for a tuple to be less or greater than another tuple. For the purposes of our dictionary, we will sort based on the key, which is an int. We have provided a method in the `tree_dictionary.ml` file that compares two `(int * string)` tuples using their key and returns a value useful for sorting the list.

You can use this ‘comparator’ (a term you will learn in this week’s lab) to sort a `(int * string)` list. You should again use the List module’s `List.sort` procedure. For example:

```ocaml
let lst : (int * string) list = [(3, "Percy") ;
   (1, "Circe") ;
   (2, "Rupert")]
check_expect (List.sort comparator lst)
   [(1, "Circe") ; (2, "Rupert") ; (3, "Percy")];;
```
2.3.3 Task

Task: Implement your module using BSTs of int * string tuples to represent dictionaries. Specifically, use this type:

```plaintext
type dict =
| Leaf
| Node of ((int * string) * dict * dict)
```

Make sure to take advantage of the binary search tree structure of your dictionary in your containsP and lookup procedures.

Note: There is an implementation of keys that has a runtime linear to the number of nodes in the tree. Implementing such a procedure is known as tree traversal and was a problem on last year’s final. For tree dictionaries, you need not implement keys in linear-time. Of course, we won’t penalize if you do produce a linear time solution.

Note: There are a few different ways to implement these procedures. As long as a) your tree maintains the invariants above, b) the number of nodes in a search tree’s subtrees are roughly the same, and c) the runtime of your procedure is reasonable, you can receive full credit.

2.4 A Note on Modules

While writing these procedures, you may notice that equivalent procedures between implementations have different runtimes. Part of the reason we implemented dictionaries using BST’s in the first place was to get the faster log \( n \) lookup time, as opposed to the linear lookup time from the list implementation. However, in getting this faster runtime, the code in the tree implementation is much more complicated than in the list implementation. As the number of bugs is proportional to the number of lines of code, this makes the tree version more difficult to maintain and debug.

A part of almost every problem in computer science is deciding what data structure to use, while considering these tradeoffs.

For example, in an application where many small dictionaries are being created, it might be more efficient to use the list implementation. In an application where you are looking up keys in an enormous dictionary, a tree implementation is the way to go. Knowing which data structures to use in specific scenarios comes with practice and knowledge of how long the relevant operations take in those data structures.

Modules allow us to seamlessly switch between different implementations, without changing how users interact with our code. This allows us the ability to create multiple implementations, each of which might be useful in a different situation.

3 Analysis

3.1 Big Theta

Task: Look back at Lecture 21, and then prove the following (which is more or less just an exercise in following the definitions):
Theorem 1. If $f, g : \mathbb{N} \to \mathbb{R}$ are real-valued functions on the natural numbers, and $f \in \Theta(g)$, then $g \in \Theta(f)$.

Task: Now prove (again, following the definitions) that if $f, h, : \mathbb{N} \to \mathbb{N}$ satisfy

$$h(n) \leq f(n) \leq 2 \cdot h(n)$$

for all $n \in \mathbb{N}$, then $f \in \Theta(h)$.

To prove that $h \in O(f)$, for example, you may want to exhibit positive numbers $M$ and $c$ with the property that for every $n > M$, it’s true that $h(n) \leq c \cdot f(n)$. To help you out slightly, in all such proofs for this problem, $M = 1$ will work fine. So you only need to find a workable value of $c$.

3.2 Keys Analysis

Consider a non-linear, tree-based implementation of keys from the previous problem (or the Mergesort algorithm we discussed in Lecture 29). The operation-counting function $U$ for this procedure, operating on a tree with $n$ nodes (where $n$ is at least 1) and a depth $k$, might satisfy a recurrence of the following form:

$$U(1) = C$$

$$U(n) \leq An + B + 2U(\lfloor \frac{n}{2} \rfloor)$$

for $n > 1$,

where $A, B, C > 0$.

(For this to be the recurrence for Mergesort, we have to alter the algorithm very slightly to be sure to get “floor” instead of “ceiling” in the recurrence.)

Plug-n-chug on cases where $n$ is a power of two, so that the floor of $n/2$ is just $n/2$, says this:

$$U(1) = C$$

$$U(2) \leq 2 \cdot A + B + C$$

$$U(4) \leq 4 \cdot A + B + 2 \cdot (2 \cdot A + B + C)$$

$$= 4 \cdot A + (B + 2B) + 2 \cdot 2 \cdot A + 2^2 C$$

$$= 4 \cdot A + 4 \cdot A + (B + 2B) + 2^2 C$$

$$= 8 \cdot A + 8 \cdot A + 8 \cdot A + (B + 2B + 4B) + 2^3 C$$

$$U(8) \leq 8 \cdot A + B + 2(4 \cdot A + 4 \cdot A + (B + 2B) + 2^2 C)$$

$$= 8 \cdot A + 8 \cdot A + 8 \cdot A + (B + 2B + 4B) + 2^3 C$$

$$U(16) \leq 16 \cdot A + B + 2(...)$$

$$= 4 \cdot (16 \cdot A) + (1 + 2 + 4 + 8)B + 2^4 C$$

$$U(32) \leq 32 \cdot A + B + 2(...)$$

$$= 5 \cdot (32 \cdot A) + (1 + 2 + 4 + 8 + 16)B + 2^4 C$$

$$U(2^k) \leq k \cdot (2^k \cdot A) + (1 + 2 + \ldots + 2^{k-1})B + 2^k C$$

$$= k \cdot (2^k \cdot A) + (2^k - 1)B + 2^k C$$

$$= k \cdot (2^k \cdot A) + 2^k (B + C) - B.$$
That lets us (correctly) conjecture that the function $U$ satisfies

$$U(n) \leq A \cdot n(\log n) + (B + C) \cdot n - B$$

for all integers $n$ that are greater than or equal to 1.

**Task:** Prove this claim. In other words, prove that if $U$ is a function on the positive natural numbers and $U$ satisfies the recurrence

$$U(1) = C$$

$$U(n) \leq An + B + 2U(\lfloor \frac{n}{2} \rfloor) \quad \text{for } n > 1$$

(R1) (R2)

then for all positive natural numbers $n$, we have

$$U(n) \leq A \cdot n(\log n) + (B + C) \cdot n - B$$

(*)

Your proof will look just like all the other proofs of claims like this, except that:

- Instead of letting $S$ be the set of all natural numbers for which statement (*) is false, you’ll let $S$ be the set of all positive natural numbers for which it’s false.

- Rather than proving that $0 \notin S$, you’ll prove that $1 \notin S$.

- Rather than considering the statement (*) for $n - 1$, you’ll consider it for $\lfloor \frac{n}{2} \rfloor$; this number will be at least 1 because . . . well, you’ll have to provide a reason.

Finally, remember that $\log \frac{n}{2} = \log(n) - 1$.

**Note:** Make your life (and the grader’s life) easier by using a two-column proof, please. A template for this can be found on Piazza.

Make sure to use LaTeX for this proof!

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)