15-150 Spring 2012
Homework 04

Out: Tuesday, 7 February 2012
Due: Wednesday, 15 February 2012 at 09:00 EST

1 Introduction

This homework will focus on lists, trees, and work-span analysis.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

To submit your solutions, place your `hw04.pdf` and modified `hw04.sml` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw04/
```

Your files must be named exactly `hw04.pdf` and `hw04.sml`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/04/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Also remember that your written solutions must be submitted in PDF format—we do not accept MS Word files.

Your `hw04.sml` file must contain all the code that you want to have graded for this assignment and compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, you will lose points for omitting the purpose, examples, or tests even if the implementation of the function is correct.

1.4 Style

Starting with this assignment, we will begin grading your submissions based on your coding style. There are several ways that you can learn what is good style and what isn't:

- Your returned and graded homework submissions have been marked up as if we had graded them for style without actually deducting points for style.
- We have published solution code for the previous assignments, labs, and lectures.
- We have published a style guide at

  http://www.cs.cmu.edu/~15150/resources/style.pdf.

  There is also a copy in the `docs` subdirectory of your git clone.
- You can ask your TAs about specific examples, or post on Piazza asking general questions.

1.5 Due Date

This assignment is due on Wednesday, 15 February 2012 at 09:00 EST. Remember that this deadline is final and that we do not accept late submissions.

2 Types For This Assignment

2.1 `rel`

We define a type `rel` that's used to represent the position of elements in an ordering relative to each other. As always, you inspect values of type `rel` with case statements. `rel` is defined as

```
datatype rel = LT | GEQ
```

You can use the function

```
(* Purpose: intrelcmp(x,y) == LT if x < y
            intrelcmp(x,y) == GEQ if x >= y *)
val intrelcmp : int * int -> rel
```

provided in the starter code to obtain a `rel`.

2.2 `tree`

The type `tree` represents binary trees of integers with data stored only in the internal nodes. This type is defined as

```
datatype tree =
  Empty
| Node of tree * int * tree
```

- The tree `Empty` has *depth* 0. The tree `Node (l,x,r)` has *depth* d if and only if
  1. `l` has depth d_l
  2. `r` has depth d_r
  3. $d = \max(d_l, d_r) + 1$.
- The tree `Empty` has *size* 0. The tree `Node(l,x,r)` has *size* s if and only if
  1. `l` has size s_l
  2. `r` has size s_r
  3. $s = s_l + s_r + 1$
- The tree `Empty` is *balanced*. The tree `Node(l,x,r)` is *balanced* if and only if
  1. `l` is balanced
  2. `r` is balanced
  3. `l` has depth d_l, `r` has depth d_r and $|d_l - d_r| \leq 1$.

- The tree `Empty` is *sorted*. The tree `Node(l,x,r)` is *sorted* if and only if

  1. `l` is sorted
  2. `r` is sorted
  3. For every node `Node(ll,xl,rl)` in `l`, $xl < x$
  4. For every node `Node(lr,xr,rr)` in `r`, $xr \geq x$

These definitions are implemented in `lib.sml`, with a few other helper functions. You should feel free to write your tests in terms of these functions.

- `depth : tree -> int` computes the depth of its argument.
- `size : tree -> int` computes the size of its argument.
- `isbalanced : tree -> bool` evaluates to true if and only if its argument is balanced.
- `issorted : tree -> bool` evaluates to true if and only if its argument is sorted.
- `tolist : tree -> int list` computes a flattening of its argument into a list, as given in class.
- `treeeq : tree * tree -> bool` tests whether two trees are equal

3 Correctness of Insertion Sort

In the Lecture 6 notes, you can find a discussion of a simple sorting algorithm called insertion sort:

```
fun insert (n : int , l : int list) : int list =
    case l of
        [] => n :: []
      | (x :: xs) => (case n < x of
                          true => n :: (x :: xs)
                        | false => x :: (insert (n , xs)))

fun isort (l : int list) : int list =
    case l of
        [] => []
      | (x :: xs) => insert (x , isort xs)
```

Recall the following definition of sortedness on lists:

- `[]` sorted
- `e::es` sorted **iff** `e` valuable and `es` sorted and $(\forall x \in es, e \leq x)$
- `e` sorted if `e` $\cong$ `e'` and `e'` sorted

Task 3.1 (10%). Prove the following spec for `insert`:

> For all values `n:int` and `l:int list`, if `l` sorted then `insert(n,l)` sorted

Task 3.2 (2%). Prove the following corollary:

> For all **valuable expressions** `e:int` and `es:int list`,
> if `es` sorted then `insert(e,es)` sorted

Your proof should not use induction.

Task 3.3 (8%). Prove the following spec for `isort`:

> For all values `l:int list`, `(isort l)` sorted

You may assume that `insert` and `isort` are total, and that `insert (x,l)` is a permutation of `x::l`. The proof of correctness of `mergesort` in the Lecture 6 notes may be a helpful reference.

4 QuickSort

QuickSort is a well-known sorting algorithm. In terms of lists, the idea is as follows:

1. The empty list is sorted.
2. Any singleton list is sorted.
3. If the list being sorted has more than one element:
   (a) Pick a pivot element from the list being sorted.
   (b) Divide the list being sorted into two lists: a list of elements less than the pivot and a list of elements greater than or equal to the pivot.
   (c) Call QuickSort recursively to sort both lists.
   (d) Append these lists together with the pivot in the appropriate order.

The main concern in making QuickSort a practical algorithm comes in choosing the pivot. A naïve answer is to always choose the first element of the list for your pivot, and this is what you will implement in this assignment. This simple policy gives QuickSort a worst case work in $O(n^2)$ on a list of n elements, rather than the $O(n \log n)$ we got for `mergesort`: if the list being sorted has no repeated elements, and happens to be in reverse-sorted order, then the list of elements greater than the pivot is empty, the list of elements equal to the pivot is just that contains only the pivot, and the list of elements less than the pivot has $(n-1)$ elements. However, this question will not deal with these issues. We are only concerned with implementing a naïve QuickSort on two different types.

4.1 QuickSort on Lists

Task 4.1 (5%). Implement a function

```
filter_l : int list * int * rel -> int list
```

`filter_l(l,p,r)` computes a list with only those elements of `l` that are in the appropriate relation to the pivot `p`, where "appropriate" is determined by `r`. More formally: For all values `l:int list`, `p:int`, and `r:rel`:

- If `r` is `LT`, then `filter_l (l,p,r)` contains all and only the elements of `l` that are less than `p`.
- If `r` is `GEQ`, then `filter_l (l,p,r)` contains all and only the elements of `l` that are greater than or equal to `p`.

Task 4.2 (5%). Implement QuickSort on lists in the function

```
quicksort_l : int list -> int list
```

For any list l, `(quicksort_l` l`)` evaluates to a permutation of l that is sorted in increasing order.

4.2 QuickSort on Trees

As we've discussed in lecture, there is not a lot to be gained by using parallel sorting algorithms on lists: there are dependencies inherent in the structure of a list that get in the way of real parallelism.

In that spirit, you'll now work up to an implementation of QuickSort on trees through a series of helper functions. We'll represent trees with the `tree` type defined at the beginning of this assignment. In specs, we will say that `x` is an element of a tree `t` when `Node(...,x,...)` appears somewhere in `t`.

4.2.1 Combine

Task 4.3 (5%). Implement a function

```
combine : tree * tree -> tree
```

that combines two trees into one. Unlike `merge` from lecture, you may **not** assume that the trees are sorted. Your implementation must satisfy the following specifications:

- Functionality: For all trees `T1` and `T2`, `combine (T1,T2)` is valuable, and contains every element of `T1` and every element of `T2` and no other elements.
- Depth: For the analysis of `quicksort`, you need the following bound on the depth of `combine`'s result:

  Lemma 1. *For all values* `t1 t2:tree`*,*
  `depth (combine(t1,t2))` $\leq 1 + max($`depth t1`$,$ `depth t2`*.*

- Running-time: Let d_1 be the depth of `T1`, d_2 be the depth of `T2`. The work and span of `(combine (T1,T2))` should be $O(d_1)$.

Task 4.4 (10%). Prove Lemma 1. Be sure to follow the template for structure induction on trees. You may assume that `combine` is total, but must carefully cite this fact when you use it.

Note: there is a simple recursive definition of `combine` that makes this theorem easy to prove. If you did not find this solution in the previous task, it may be easier to revise your code than to prove the spec for a more-complicated implementation. Hint: you won't need any helper functions.

4.2.2 Filter

Task 4.5 (12%). You'll also need a tree-analogue of `filter_l`:

```
filter : tree * int * rel -> tree
```

Your implementation must satisfy the following specs:

- Functionality: If `T` is a value of type `tree`, `p` is a value of type `int` and `r` is a value of type `rel`, then:
  - If `r` is `LT`, then `filter(T,p,r)` contains all and only the elements of `T` that are less than `p`.
  - If `r` is `GEQ`, then `filter(T,p,r)` contains all and only the elements of `T` that are greater than or equal to `p`.
- Depth: For all `T:tree`, `p:int` and `r:rel`, `depth (filter (T,p,r))` $\leq$ `depth T`.
- Running-time: If d is the depth of a tree `T`, your implementation of `(filter (T,p,ord))` should have $O(d^2)$ span. On a balanced tree, your implementation of `filter` should have $O(n)$ work.[1]

4.2.3 Quicksort

Task 4.6 (10%). Finally, put all the pieces together to write

```
quicksort_t: tree -> tree
```

which implements QuickSort values of type `tree`.

- Functionality: `quicksort_t T` is sorted and contains all and only the elements of `T`.
- Running-time: If `T` is a tree with depth d and size n, `(quicksort_t T)` should have $O(n \log n)$ work and $O(d^3)$ span, assuming the pivots yield balanced subproblems.

You should use `issorted` to test your implementation of `quicksort_t`.

5 Balancing

In `mergesort` in lecture, we saw that we needed to *rebalance* a tree after manipulating it. Rebalancing takes a tree that is not necessarily balanced, and computes a balanced tree with the same elements.

We have provided almost all of an implementation of a simple rebalancing algorithm in the handout. The key helper function is unimplemented. You will implement this helper and then analyze the complexity of `rebalance`.

In all of the tasks, you should assume that the function `size : tree -> int`, which computes the size of a tree, runs in constant time on all inputs. This happens to be obviously

[1] If you use the tree method to try to prove this, you will run into a sum that we have not yet seen in the course. A recurrence of the form $T(n) = k \log n + T(n/2)$ is $O(n)$.

false. However, it's easy to make binary trees whose size can be computed in constant time by caching the size at each node—so this is a relatively harmless lie.

Task 5.1 (15%). Implement the function

```
takeanddrop : tree * int -> tree * tree
```

`takeanddrop(T,i)` separates a tree `T` into "left" and "right" subtrees, `T1` and `T2` respectively. `T1` contains the leftmost `i` elements of `T`, in their original order, and `T2` the remaining elements, also in their original order. For example, if we define

```
val test =
  Node
    (Node (Node (Empty,1,Empty),
           2,
           Node (Empty,3,Empty)),
     4,
     Node (Node (Empty,5,Empty),
           6,
           Empty))
```

then we have

```
takeanddrop (test,3) ==
  (Node (Node (Empty,1,Empty),2,Node (Empty,3,Empty)),
   Node (Empty,4,Node (Node (Empty,5,Empty),6,Empty)))
```

More formally, suppose `T` is any `tree`, and `i` $\leq$ `size T`. Then `takeanddrop (T,i)` evaluates to a pair of trees `(T1,T2)` such that

- $max($`depth T1`$,$ `depth T2`$) \leq$ `depth T`
- `size T1` $\cong$ `i`
- `(tolist T1) @ (tolist T2)` $\cong$ `(tolist T)`

This last condition ensures that `T1` contains the leftmost elements, and that the elements of `T1` and `T2` are in the appropriate order.

`takeanddrop` should raise `Fail` with "not enough elements" if and only if $i > n$.

If d is the depth of `T` then your implementation of `(takeanddrop (T,i)` must have $O(d)$ work and span.

Task 5.2 (18%). Your implementation of `takeanddrop` is necessary for the helper function `halves`, which is used by `rebalance`; see the starter code. The following tasks ask you to analyze these functions:

1. Give a recurrence that describes the work of your implementation of `takeanddrop`, $W_{\texttt{takeanddrop}}(d)$, in terms of the **depth** d of the input tree. Give a tight big-O bound for $W_{\texttt{takeanddrop}}(d)$.

   Note that if your `takeanddrop` meets the above spec, this will be in $O(d)$, but you should argue why your code does actually meet this spec.

2. Give a recurrence that describes the span of your implementation `takeanddrop`, $S_{\texttt{takeanddrop}}(d)$, in terms of the **depth** d of the input tree. Give a tight big-O bound for $S_{\texttt{takeanddrop}}(d)$.

   Note that if your `takeanddrop` meets the above spec, this will be in $O(d)$, but you should argue why your code does actually meet this spec.

3. Give a recurrence that describes the work of `halves`, $W_{\texttt{halves}}(d)$, in terms of the **depth** of the input tree. Give a tight big-O bound for $W_{\texttt{halves}}(d)$.

4. Give a recurrence that describes the span of `halves`, $S_{\texttt{halves}}(d)$, in terms of the **depth** of the input tree. Give a tight big-O bound for $S_{\texttt{halves}}(d)$.

5. Give a recurrence that describes the work of `rebalance`, $W_{\texttt{rebalance}}(n)$, in terms of the **size** n of the input tree. You should assume that the input is roughly balanced—that is to say, there exists some constant c such that the depth of the input is $c \log n$.

   Use the tree method to give a closed form for this recurrence; your closed form may involve a sum. Use this closed form to give a tight big-O bound for $W_{\texttt{rebalance}}(n)$.

6. Give a recurrence that describes the span of `rebalance`, $S_{\texttt{rebalance}}(n)$, in terms of the **size** of the input tree. You should assume that the input is roughly balanced—that is to say, there exists some constant c such that the depth of the input is $c \log n$.

   Use the tree method to give a closed form for this recurrence; your closed form may involve a sum. Use this closed form to give a tight big-O bound for $S_{\texttt{rebalance}}(n)$.

The recurrences for the later tasks should be defined in terms of the recurrences defined in the earlier tasks for the helper functions.

You may use the following tight bounds as facts, but you should cite them when you use them:

$$\sum_{i=1}^{\log n} \log\left(\frac{n}{2^i}\right) \text{ is } O\left((\log n)^2\right)$$

$$\sum_{i=1}^{\log n} 2^i \log\left(\frac{n}{2^i}\right) \text{ is } O\left(n\right)$$