COMP 212 Spring 2015
Homework 04

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

1 Types For This Assignment

1.1 `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

> A `rel` is either `LT` or `GEQ`, and that's it!

After Lecture 8, you will understand the SML datatype notation used to declare this:

```
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`.

1.2 `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`, $xr \geq x$

  An expression e : `tree` is sorted iff it is equivalent to a value that is sorted.

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 lab.
- `fromlist : int list -> tree` computes a balanced tree from a list—very useful for testing, but do **not** use it in any of your solutions, or they will not have the right span.
- `treeeq : tree * tree -> bool` tests whether two trees are equal

2 Correctness of Merge

Recall the `merge` function on trees from lecture 7:

```
fun merge (t1 : tree , t2 : tree) : tree =
 case t1 of
     Empty => t2
   | Node (l1 , x , r1) =>
      let val (l2 , r2) = splitAt (t2 , x)
      in
          Node (merge (l1 , l2) ,
                x,
                merge (r1 , r2))
      end
```

See the definition of sorted trees in Section 1 above.

Task 2.1 (13 pts). Prove the following:

Theorem 1. *For all values `l:tree`, for all values `r:tree`, if `l` is sorted and `r` is sorted then `merge(l,r)` is sorted.*

Follow the template for *structural induction on trees*, doing induction on `l`. Because we are proving the property "for all values `r:tree`, if `l` is sorted and `r` is sorted then `merge(l,r)` is sorted" by induction on `l`, each of your inductive hypotheses should be of the form "for all values `r':tree`, if `?` is sorted and `r'` is sorted then `merge(?,r')` is sorted" (where you have to figure out the `?`). That is, each inductive hypothesis works for *all* values of the second argument to the function.

You may use the following:

Lemma 1. *If `t` is sorted, then there exist `l` and `r` such that `split(t,bound)` $\cong$ `(l,r)` where*

- *`l` is sorted*
- *`r` is sorted*
- *every number in `l` is less than or equal to `bound`*
- *every number in `r` is greater than the `bound`*

Lemma 2. *Every number in `merge(l,r)` is either in `l` or in `r`.*

3 QuickSort

In this problem, you will implement QuickSort on lists and on trees. 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 just pick the pivot to be the first element.

3.1 QuickSort on Lists

Task 3.1 (5 pts). 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 3.2 (5 pts). Implement QuickSort on lists in the function

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

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

3.2 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. Assuming the pivots yield subproblems of equal size (which can be achieved using randomness), this algorithm will have $O(n \log n)$ work and $O((\log n)^3)$ span. The logarithmic span means significant speedups can be gained by running the algorithm in parallel.

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`.

3.2.1 Combine

First, we will need a function to combine two trees into one. Unlike `merge` from lecture, we will not require that the inputs are sorted, but we will also not ensure that the outputs are sorted, or that the outputs are in the same order as in the original trees. Because this function will be used prior to sorting, the elements can be in any order you choose.

Task 3.3 (8 pts). Write a function

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

that meets the following specification:

- 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. Hint: You can use `tolist` and `fromlist` to easily test this property.
- Depth: For the analysis of `quicksort`, you need the following bound on the depth of `combine`'s result:

  Lemma 3. *For all values* `t1 t2:tree`,
  $\texttt{depth (combine(t1,t2))} \leq 1 + max(\texttt{depth t1}, \texttt{depth t2})$.

  Hint: use the provided `depth` function to test this property.
- 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))` are both $O(d_1)$.

3.2.2 Filter

Task 3.4 (8 pts). 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, where n is the size of the tree.[1]

3.2.3 Quicksort

Task 3.5 (8 pts). 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 balanced tree with size n, `(quicksort_t T)` should have $O(n \log n)$ work and $O((\log n)^3)$ span, assuming the pivots yield balanced subproblems.

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

4 Balancing

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

Task 4.1 (10 pts). Implement the function

[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 $O(n)$.

```
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 $0 \leq$ `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.

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

Hint: use the `splitAt` function from mergesorting trees as a model; the difference is that instead of splitting based on the values stored in the tree, here we are splitting based on the number of elements in the tree.

Task 4.2 (18 pts). 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. Argue that $W_{\texttt{takeanddrop}}(d)$ is $O(d)$.

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. Argue that $S_{\texttt{takeanddrop}}(d)$ is $O(d)$

Note: the remaining tasks will be graded assuming that $W_{\texttt{takeanddrop}}(d)$ and $S_{\texttt{takeanddrop}}(d)$ are $O(d)$, even if that is not true for your code, or if your recurrence above is incorrect.

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, but you should cite them when you use them:

$$\sum_{i=0}^{\log n} \log\left(\frac{n}{2^i}\right) \text{ is } O\left((\log n)^2\right)$$

$$\sum_{i=0}^{\log n} 2^i \log\left(\frac{n}{2^i}\right) \text{ is } O\left(n\right)$$