COMP 212 Fall 2022
Homework 05

This homework will focus on lists, trees, and work-span analysis. You must write purposes and tests for all functions on this assignment.

1 Trees

The type `tree`, with constructors `Empty` and `Node(l,x,r)` represents binary trees of integers with data stored only in the internal nodes. Here are some definitions about trees:

- 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 `hw05-lib.sml`, with a few other helper functions. Download this file and put it in the same folder as your `hw05.sml` file. 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 Contains

Task 2.1 (7 pts). Write a function `contains : tree * int -> bool` such that for any sorted tree `t`, `contains(t,i)` returns `true` if `i` is an element of `t` and returns `false` if not. For full credit, your solution should have $O(\log n)$ work and span when t is a balanced tree of size n.

The `listToTree` function from lab is called `fromlist : int list -> tree` in the homework code, and can be used to write tests.

3 Tree Induction

Recall the `size` and `depth` functions on trees:

```
(* size t computes the natural number which
   represents how many Nodes are in a tree *)
fun size (t : tree) : int =
    case t of
        Empty => 0
      | Node (l, x, r) => 1 + size l + size r


(* depth t computes the natural number which
   represents how many levels are in a tree *)
fun depth (t : tree) : int =
    case t of
      Empty => 0
    | Node (l, x , r) => 1 + max (depth l, depth r)
```

Task 3.1 (15 pts). Prove the following relationship:

Theorem 1. *For all trees* `t`*,* `depth t` $\leq$ `size t`*.*

The function $\texttt{max}(x, y)$ returns the maximum of x and y; i.e. if x is bigger it returns x and if y is bigger it returns y. You may want to use some of the following properties of `max`:

- $x \leq \texttt{max}(x, y)$ and $y \leq \texttt{max}(x, y)$
- $\texttt{max}(x, y) \leq z$ if $x \leq z$ and $y \leq z$
- $\texttt{max}(x, y) \leq \texttt{max}(x', y')$ if $x \leq x'$ and $y \leq y'$

4 QuickSort

In last week's homework, you implemented QUICKSORT on lists. 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 implement QUICKSORT on trees. 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`.

4.1 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 we choose. This means the following code suffices:

```
fun combine (t1 : tree, t2 : tree) : tree =
    case t1 of
        Empty => t2
      | Node(l1,x1,r1) => Node(combine(l1,r1),x1,t2)
```

You may wish to draw `combine`'s output on some examples to understand how it works. More formally, this code 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.
- Depth: For the analysis of `quicksort`, we 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))` are both $O(d_1)$.

4.2 Filter

Task 4.1 (15 pts). Implement a tree analogue of `filter_less` and `filter_greatereq`:

```
filter_less : tree * int -> tree
filter_greatereq : tree * int -> tree
```

Your implementation must satisfy the following specs:

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

4.3 Quicksort

Task 4.2 (15 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 may want to use the `fromlist : int list -> tree` and `issorted` functions to test your implementation of `quicksort_t`.

5 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 5.1 (15 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; see the next problem for its big-O.

```
    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 5.2 (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 $c \log n$. This will be true when `rebalance` is called from `mergesort`, because the merging will only have unbalanced the tree by a known amount.

   Give a tight big-O bound for $W_{\texttt{rebalance}}(n)$. Show your work using a closed form and/or sum.

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 $c \log n$. This will be true when `rebalance` is called from `mergesort`, because the merging will only have unbalanced the tree by a known amount.

   Give a tight big-O bound for $S_{\texttt{rebalance}}(n)$. Show your work using a closed form and/or sum.

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:

$$\begin{array}{rcl} \log n + \log \frac{n}{2} + \log \frac{n}{4} + \log \frac{n}{8} + \ldots + 1 & \text{is} & O\left((\log n)^2\right) \\ \log n + 2\log \frac{n}{2} + 4\log \frac{n}{4} + 8\log \frac{n}{8} + \ldots \text{(with } \log n \text{ terms)} & \text{is} & O\left(n\right) \end{array}$$

6 NON-COLLABORATIVE CHALLENGE PROBLEM: Cargo

Remember that non-collaborative challenge problems are to be done independently. You are not allowed to communicate with anyone about the problems, except to ask the instructor or TAs clarification questions (not hints). Additionally, you are not allowed to search for help on the specific problem from any sources besides the course materials. You are free to ask clarification questions about the concepts involved.

You work for a shipping company, and are writing a program to help decide which cargo to load onto an airplane. Each shipping box has a weight. Each box also has a profit value, which is how much the shipping company makes by shipping that item on this plane. The airplane has a maximum weight limit that it can safely carry. Your function will be given a list of potential boxes for shipping, and must choose which boxes should be shipped on the airplane. The company would like you to select the boxes that generate the highest profit, subject to the constraint that chosen items cannot exceed the weight limit for the plane.

Task 6.1 (10 pts). Write a function `cargo` that, when given a list of shipping boxes and a maximum allowed weight, returns the best list of items to ship on the plane.

Task 6.2 (6 pts). Analyze the work and span of your `cargo` function.