Homework 04

15-150 Fall 2024

Due: Mon 23rd Sept, 2024 at 8:00PM

LaTeX

The written part of your assignments must be written in LaTeX. To make your job easier, a LaTeX template is provided for each assignment where you simply need to fill in your solutions.

If you want to know what is the LaTeX name for a symbol, you can use Detexify. Also, Mathcha can help you type math. For constructs particular to this course (inductive definitions, proofs, etc), we provide a LaTeX guide with templates.

Finally, if you are just lost on how to type something or why your file is not compiling, reach out to the course staff.

Your submission will be penalized if it is not written using LaTeX. The applicable penalties are described in the style guide.

Code Structure

Check out the *basic requirements* for coding tasks in the style guide.

Proof Structure

Check out the *basic structure* for proofs in the style guide.

1 Insertion Sort

In this section, we will be adapting insertion sort to work on trees.

1.1 Insertion Sort on Lists

Insertion sort is another classical sorting algorithm on lists. It works by inserting elements in a sorted list in the "right" position. To sort a given list, insertion sort does this operation for each element in it starting from the empty (sorted) list. The SML implementation of insertion sort is very simple:

```
fun insert (x: int, []: int list): int list = [x]
  | insert (x, y::l) = if x <= y
                       then x::y::l
                       else y :: insert (x, l)

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

(The starter file `isort.sml` contains this code with its specs for reference.) It is easy to check that the work and span of `isort` are quadratic in the length of the input list. It is not a great sorting algorithm.

1.2 Sorting Trees by Insertion

Could we use some of the ideas behind insertion sort on lists to sort trees? To explore this, we will consider trees with data in their inner nodes:

```
datatype itree = empty
               | node of itree * int * itree
```

Inserting a node in a sorted tree in such a way that the resulting tree is sorted is fairly simple. It is achieved by the following function `Insert` (first letter capitalized) which does on trees what `insert` does on lists:

```
fun Insert (x: int, empty: itree): itree = node(empty, x, empty)
  | Insert (x, node(tL,y,tR)) =
      if x <= y
      then node (Insert (x, tL), y, tR)
      else node (tL, y, Insert (x, tR))
```

This function can be found in the starter file `isort.sml`, together with its specs.

Task 1.1 (6 points) Write the recurrence relation for the work of `Insert` in terms of the height of the input tree, deduce an upper-bound approximation in closed form, and determine a tight big-O class for it. Use this to identify the best and worst cases measured in terms of the number of nodes in the tree, and give their big-O classes. Do the same for its span.

One simple approach for sorting a tree is to read off its elements into a list (the familiar code for `inorder` is given in `isort.sml`) and then repeatedly use `Insert`. That pretty much follows the idea behind insertion sort on lists.

Coding Task 1.2 (5 points) Implement the SML function

```
ILsort: itree -> itree
```

that returns a sorted tree with the exact same elements as its input. Your code should use `Insert` and `inorder`.

Task 1.3 (10 points) Write the recurrence relation for the work of `ILsort` in terms of the number of nodes in the tree. For both the best case and the worst case, deduce an upper-bound approximation in closed form, and determine a tight big-O class for it. Do the same for its span.

Task 1.4 (2 points) Chances are that your code does not prevent the worst case scenario from happening. How would you modify it so that the best case is always realized. Explain why. [You are not required to modify your code in this way, although you are welcome to.]

1.3 Insertion Sort for Trees

The above approach to sorting a tree goes through a list, the inorder traversal of the input tree. Can we implement a more direct version of insertion sort on trees?

Coding Task 1.5 (5 points) Implement the SML function

```
Isort: itree -> itree
```

that returns a sorted tree with the exact same elements as its input. Your code should use `Insert`, but it is not allowed to use any list or operations on lists.

2 Balanced Trees

In many applications, it is convenient to work with trees that carry their data in the inner nodes rather than at the leaves. This is easily captured in SML by the following type declaration:

```
datatype tree = empty
              | node of tree * char * tree
```

The starter code for this homework implements for you the function `inorder: tree -> char list` that performs an in-order traversal of its argument — that is, the call `(inorder t)` returns the list of the characters in `t` such that the elements of the left subtree of any inner node occur before the element at the node which in turn occurs before the elements in its right subtree.

Chances are that some trees are imbalanced one way or another. When our trees get too imbalanced the computational benefits of working with trees disappear. The obvious remedy is to *rebalance* them from time to time, so that the heights of the left and right subtree of any inner node differ by at most one.

In this part of the exercise, we will implement a function that rebalances a tree, and then analyze its work and span. The starter code for this homework implements the following auxiliary functions for you:

- `size: tree -> int` computes the number of elements in its input tree.
- `height: tree -> int` returns the height of a tree, i.e., the length of the longest path to a leaf.
- `balanced: tree -> bool` returns `true` if and only if its argument is a balanced tree.

Feel free to use these functions. If you do, you may assume they have $O(1)$ work and span.[1]

Coding Task 2.1 (8 points) Define the SML function

```
splitN: tree * int -> tree * tree
```

so that, given a non-negative number `i` and a tree `t` containing at least `i` elements, the call `splitN (t, i)` returns a pair of trees `(t1, t2)` such that `t1` contains the leftmost `i` elements of `t`, and `t2` contains the rest. Neither `t1` nor `t2` can be higher than `t`. Write the specification comments for your implementation using solely SML code — write a `_safe` variant of this function if you find it useful.

Coding Task 2.2 (2 points) Using `splitN`, define the SML function

```
leftmost: tree -> char * tree
```

so that, given a non-empty tree `t`, the call `leftmost t` returns a pair `(x,t')` where `x` is the element that is furthest to the left in the in-order traversal of `t` and `t'` is the rest of `t`. As usual by now, write code-based specs.

Coding Task 2.3 (4 points) Using what you have written so far, define the SML function

```
halves: tree -> tree * char * tree
```

so that, given a non-empty tree `t`, the call `halves t` returns the triple `(t1,x,t2)` so that the following code-based `ENSURES` condition is satisfied:

[1] These bounds are not achievable for the type `tree` used in this exercise. However, a slighlty more complex notion of tree can achieve them.

```
   * inorder t == (inorder t1) @ x :: (inorder t2)
   * size t1 = (size t) div 2
   * height t1 <= height t
   * height t2 <= height t
```

Coding Task 2.4 (4 points) Using what you have written so far, define the SML function

```
rebalance: tree -> tree
```

so that the call `rebalance t` returns a balanced tree containing the same elements as `t` and in the same order.

How long does it take for `rebalance` and the auxiliary functions it uses to run? Let's find out! The next two tasks ask you to analyze these functions. As you do so, you may use the following tight bounds as facts, but you should cite them when you use them:

$$B1: \sum_{i=0}^{\log n} 2^i \in O(n)$$

$$B2: \sum_{i=0}^{\log n} \log \frac{n}{2^i} \in O((\log n)^2)$$

$$B3: \sum_{i=0}^{\log n} 2^i \log \frac{n}{2^i} \in O(n)$$

In these tasks, you may 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 for the provided implementation. However, it's easy to make binary trees whose size can be computed in constant time by caching the size at each node.

Note: the recurrences for the later tasks should be defined in terms of the recurrences you wrote in earlier tasks for the helper functions.

Task 2.5 (2 points) Give a recurrence that describes the work $W_{\texttt{splitN}}(d)$ of `splitN` in terms of the *height* d of the input tree. Give a tight big-O bound for $W_{\texttt{splitN}}(d)$.

Task 2.6 (2 points) Give a recurrence that describes the work $W_{\texttt{leftmost}}(d)$ of `leftmost` in terms of the *height* d of the input tree. Give a tight big-O bound for $W_{\texttt{leftmost}}(d)$.

Task 2.7 (2 points) Give a recurrence that describes the work $W_{\texttt{halves}}(d)$ of `halves` in terms of the *height* of the input tree. Give a tight big-O bound for $W_{\texttt{halves}}(d)$.

Task 2.8 (2 points) Give a recurrence that describes the work $W_{\texttt{rebalance}}(n)$ of `rebalance` in terms of the *size* n of the input tree. Show how to obtain 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 $W_{\texttt{rebalance}}(n)$.

Task 2.9 (2 points) How does the work of `rebalance` change if you know that the input tree is *roughly balanced*? A tree is roughly balance if its height is $O(\log n)$. Redo the calculations for $W_{\texttt{rebalance}}(n)$ to incorporate this additional piece of information.

Task 2.10 (2 points) Give a recurrence that describes the span $S_{\texttt{splitN}}(d)$ of `splitN` in terms of the *height* d of the input tree. Give a tight big-O bound for $S_{\texttt{splitN}}(d)$.

Task 2.11 (2 points) Give a recurrence that describes the span $S_{\texttt{leftmost}}(d)$ of `leftmost`, in terms of the *height* d of the input tree. Give a tight big-O bound for $S_{\texttt{leftmost}}(d)$.

Task 2.12 (2 points) Give a recurrence that describes the span $S_{\texttt{halves}}(d)$ of `halves` in terms of the *height* of the input tree. Give a tight big-O bound for $S_{\texttt{halves}}(d)$.

Task 2.13 (2 points) Give a recurrence that describes the span $S_{\texttt{rebalance}}(n)$ of `rebalance` in terms of the *size* n of the input tree. Show how to obtain 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 $S_{\texttt{rebalance}}(n)$.

Task 2.14 (2 points) How does the span of `rebalance` change if you know that the input tree is roughly balanced? Redo the calculations for $S_{\texttt{rebalance}}(n)$ to incorporate this additional piece of information.