Homework 6

15-210: Parallel and Sequential Data Structures and Algorithms (Spring'12)

`tag` — Version 1.0 **due:** *tue, mar 27 @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this homework, you will be implementing a version of the `SEQUENCE` data structure using balanced binary trees.

1.1 Submission

This assignment is distributed in a number of files in our `git` repository. Instructions on how to access that repository can be found at `http://www.cs.cmu.edu/~15210/resources/git.pdf`. This is how assignments will be distributed in this course.

This assignment requires that you submit both code and written answers.

Submit your solutions by placing you solution files in your handin directory, located at

```
/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn6/
```

Name the files exactly as specified below. You can run the check script located at

```
/afs/andrew.cmu.edu/course/15/210/bin/check/06/check.pl
```

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw06.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful.

For the programming part, the only files you're handing in are

```
TreeSequence.sml
```

These files must contain all the code that you want to have graded for this assignment and must 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.

1.2 Naming Modules

The questions that follow ask you to organized your solutions in a number of modules. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

1.3 The SML/NJ Build System

This assignment includes a substantial amount of library code spread over several files. The compilation of this is orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found at on the website at `http://www.cs.cmu.edu/~15210/resources/cm.pdf`.

1.4 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

1. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments would convince a reader that your code is correct.
2. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so. We have provided a file, `Test.sml` which you should put all your test code in.
3. **Clearly indicate parallelism in your code.** Use the provided `par` and `par3` functions in the library structure `Primitives` for this purpose.

2 Balanced Trees

In this assignment, we will be using *balanced* trees as an underlying data structure. The particular binary trees we will be using ascribe to the `BINARY_TREE` signature given in your library code.

These particular binary trees are *full* binary trees, that is, every node either has two children or is a leaf, with values stored at the leaves. The trees are also *unordered*, unlike Treaps or other BSTs. Instead, the ordering of their elements is given by the way that you choose to construct the trees from the functions provided. For implementing sequences, we will be ordering the elements in the trees left to right, such that the first element in the sequence is the leftmost value in the tree, the second element is the second leftmost, etc.

Finally, these trees are (approximately) balanced. It is guaranteed that for any tree of n nodes, the depth of the tree is $O(\log n)$.

The following functions for building binary trees and examining their structure are given in the `BalancedTree` structure in `BalancedTree.sml`:

- `empty`: $\mathit{unit} \rightarrow \alpha\ \mathit{tree}$

  `empty ()` evaluates to the empty tree.

- `singleton`: $\alpha \rightarrow \alpha\ \mathit{tree}$

  If `x` is a value, then `(singleton x)` evaluates to the singleton tree, a leaf containing $\langle x \rangle$.

- `size`: $\alpha\ \mathit{tree} \rightarrow \mathit{int}$

  If `T` is a tree value, then `(size T)` evaluates to the number of leaves in `T`.

- `expose`: *α tree $\to$ α node*

  `(expose T)` evaluates to a node value `N` which represents the underlying structure of the tree. We can sum up the behavior as:

  - `(expose (empty ()))` evaluates to `EMPTY`.
  - `(expose (singleton v))` evaluates to `LEAF(v)`.
  - If `T` is a tree with `size(T)` ≥ 2, i.e. containing two branches, `L` and `R`, then `(expose T)` evaluates to `NODE(L,R)`.

- `join`: *α tree $\times$ α tree $\to$ α tree*

  If `T1` and `T2` are tree values then `(join (T1,T2))` is a tree containing all the elements in `T1` and `T2` (even duplicates), with all of the elements of `T1` being to the left of all `T2` elements in the tree ordering. Within trees the elements retain their order. Logically, you can think of this as the tree that would result if you wrote out the leaves of `T1` and `T2` as lists `L1` and `L2` in left to right order, appended them, and then turned the final list back in to a resulting tree.

  This function will also rebalance the tree if necessary, so that the balance invariant is preserved.

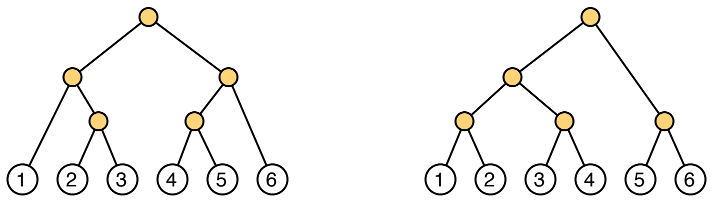
The work and span bounds for each of the above functions are as follows:

TreeSequence	*Work*	*Span*
`empty()` `singleton`(v) `size`(T) `expose`(T)	$O(1)$	$O(1)$
`join`(T_1, T_2)	$O\left(\log(\lvert T_1\rvert + \lvert T_2\rvert)\right)$	$O\left(\log(\lvert T_1\rvert + \lvert T_2\rvert)\right)$

3 Tree Sequences

Using the balanced tree data structure described above, you will give an implementation of the `SEQUENCE` interface you are familiar with from class. Your implementation should go in `TreeSequence.sml`. For brevity, you will only be implementing a subset of the core sequence functions: `empty`, `singleton`, `length`, `nth`, `append`, `take`, `drop`, `tabulate`, `map`, `filter` and `reduce`. Your implementation should behave identically to the implementation found in `ArraySequence` but will use balanced trees to store the sequence instead of an SML array. Your implementation must satisfy the cost bounds given in Section 3.1, which will be different from the cost bounds for `ArraySequence`.

The tree representation of a sequence is as follows: We will represent every value in the sequence as a leaf in the tree, and the order of the values in the sequence is given by the left-to-right ordering of the values in the tree. So, if a value v_1 is to the left of another value v_2 in the tree, that means that v_1 comes before v_2 in the sequence the tree represents. Also remember that, as per the `BalancedTree` implementation, the tree is full and is approximately balanced so that the depth is $O(\log n)$. Other than that, the structure is not fixed and your implementation cannot rely on a certain structure. For example, the following trees are both valid representations of the sequence $\langle 1, 2, 3, 4, 5, 6\rangle$:



For more insight into the tree representation of a sequence, you can look at the `fromList` and `toString` functions we have provided. Your task is to implement the basic sequence operations for this tree-based version of sequences. Detailed definitions of the functions you need to implement are as follows:

Task 3.1 (2%). `empty`: *unit* $\to$ *α seq*

`empty ()` evaluates to the empty sequence $\langle\rangle$.

Task 3.2 (2%). `singleton`: $\alpha \to \alpha$ *seq*

If `x` is a value, then `(singleton x)` evaluates to the singleton sequence $\langle x \rangle$.

Task 3.3 (2%). `length`: *α seq* $\to$ *int*

If `s` is a sequence value, then `(length s)` evaluates to $|s|$.

Task 3.4 (6%). `nth`: *α seq* $\to$ *int* $\to \alpha$

If `s` is a sequence value and `i` is an `int` value and i is a valid index into s, then `(nth s i)` evaluates to s_i. This application raises `Range` if i is not a valid index.

Task 3.5 (6%). `append`: *α seq* $\times$ *α seq* $\to$ *α seq*

If `s1` and `s2` are sequence values, then `(append (s1, s2))` evaluates to a sequence s with length $\left|s_1\right| + \left|s_2\right|$ such that the subsequence of s starting at index 0 with length $\left|s_1\right|$ is s_1 and the subsequence of s starting at index $\left|s_1\right|$ with length $\left|s_2\right|$ is s_2.

Task 3.6 (6%). `take`: *α seq* $\times$ *int* $\to$ *α seq*

If `s` is a sequence value and `n` is an integer, then `(take (s,n))` evaluates to the first subsequence of s of length n. This application will raise `Range` if $n > |s|$.

Task 3.7 (6%). `drop`: *α seq* $\times$ *int* $\to$ *α seq*

If `s` is a sequence value and `n` is an integer, then `(drop (s,n))` evaluates to the last subsequence of s of length $|s| - n$. This application will raise `Range` if $n > |s|$.

Task 3.8 (10%). `tabulate`: (*int* $\to \alpha$) $\to$ *int* $\to$ *α seq*

If `f` is a function and `n` is an `int` value, then `(tabulate f n)` evaluates to a sequence s such that $|s| = n$ and, for all valid indicies i into s, s_i is the result of evaluating `(f i)`.

Note that the evaluation of this application will only terminate if f terminates on all valid indices into the result sequence s.

Task 3.9 (10%). `map`: $(\alpha \to \beta) \to \alpha$ *seq* $\to \beta$ *seq*

If `f` is a function and `s` is a sequence value such that $|s| = n$, then `(map f s)` evaluates to the sequence r such that $|r| = n$ and, for all valid indicies i into s, r_i is the result of evaluating `(f` s_i`)`.

Note that the evaluation of this application will only terminate if `f` terminates on s_i for all valid indicies i.

Task 3.10 (10%). `filter`: $(\alpha \to bool) \to \alpha\ seq \to \alpha\ seq$

If p is a predicate and s is a sequence value, then `(filter p s)` evaluates to the longest subsequence s' of s such that p holds for every element of s'. The elements in subsequence s' are in the same order as in sequence s.

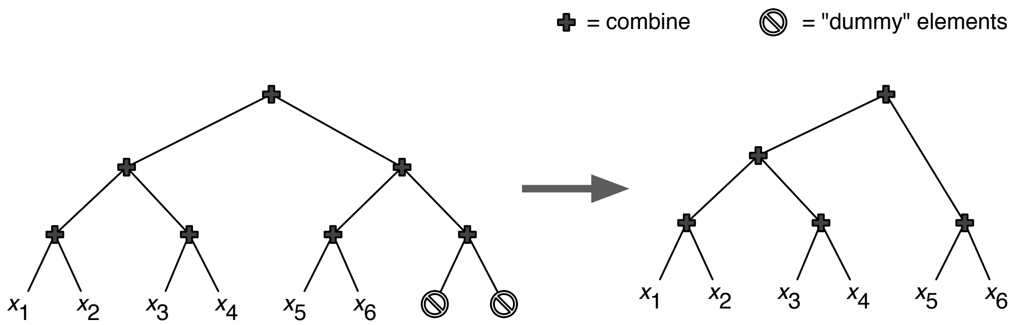
Task 3.11 (20%). `reduce`: $((\alpha \times \alpha) \to \alpha) \to \alpha \to \alpha\ seq \to \alpha$

Let f be a binary function represented as the infix operator $\oplus$, b be a value, and s a sequence value. Then, `(reduce f b s)` returns the result of computing

$$b \oplus s_0 \oplus s_1 \oplus s_2 \oplus s_3 \oplus \cdots \oplus s_{|s|-1}.$$

using a particular parenthesization. As a simple example, if f corresponds to the $+$ operation, `reduce f b s` corresponds to $b + \sum s_i$—and in this particular case, the specific parenthesization doesn't matter because the $+$ operator is associative.

In general, the critical detail of this computation is the order that the operations are applied in because the function f needs not be associative. Indeed, when the combining function is not associative, the value returned can be very different depending on the order the operations are performed in. We're interested in an implementation with deterministic behavior, one that matches the behavior specified in the `SEQUENCE` signature and also the implementation given in `ArraySequence`. Here, the order is defined by a specific combining tree. When $|s| = 2^k$, this combining tree is a *perfect* binary tree, where every node either has two children or is a leaf and all leaves are at the same depth. The sequence values are at the leaves, and the internal nodes apply f to their two children recursively. When $|s|$ is not a power of two, the combining tree is the same as if s has "dummy" values appended to it so that $|s|$ is the next power of two. Whenever there are children with dummy values, the combining f is not applied, as shown in the following figure.



Notice that for all nodes in the combining tree, its left subtree always as 2^k leaves for some k, and right subtree has no more leaves than the left subtree. **This reduction tree might differ substantially from the shape of the tree we use to represent the sequence.** And, once again, since we don't require `f` to be associative, reduction order matters.

All that is left is to handle the inital value passed in to reduce. If `f` is a function, `b` a value, and `s` a sequence value, there are two cases:

- If $|s| = 0$ then `(reduce f b s)` evaluates to `b`.
- If $|s| > 0$, and `(reduce f b s)` evaluates to some value `v` using the reduction order described above, then `(reduce f b s)` evaluates to `(f (b, v))`.

Using the infix notation for `f` again, the combining tree corresponds to the following way to parenthesize the expression given before:

$$b \oplus \Big(\ldots ((s_0 \oplus s_1) \oplus (s_2 \oplus s_3)) \oplus \cdots \oplus s_{|s|-1}) \ldots \Big).$$

In getting started, you can use the `ArraySequence` implementation in the library to further explore the behavior of reduce and what the proper computation order for non-associative functions is. (Hint: you can come up with a function for f that will get the existing implementation to print the order of computation for you for any given sequence.)

Task 3.12 (10%). Prove that your reduce implementation meets the work and span bounds given below.

3.1 Work and Span Bounds

The functions you are required to implement above should have the work and span bounds given in the following table. In some cases, you can actually do better, but these are the worst that your functions should perform.

TreeSequence	*Work*	*Span*
`empty()` `singleton`(v) `length`(S)	$O(1)$	$O(1)$
`nth` $S\ i$	$O\left(\log \lvert S \rvert\right)$	$O\left(\log \lvert S \rvert\right)$
`append`(S_1, S_2)	$O\left(\log(\lvert S_1 \rvert + \lvert S_2 \rvert)\right)$	$O\left(\log(\lvert S_1 \rvert + \lvert S_2 \rvert)\right)$
`take`(S, n)	$O\left(\log^2 \lvert S \rvert\right)$	$O\left(\log^2 \lvert S \rvert\right)$
`drop`(S, n)	$O\left(\log^2 \lvert S \rvert\right)$	$O\left(\log^2 \lvert S \rvert\right)$
`tabulate` $f\ n$	$O\left(\sum_{i=0}^{n-1} W(f(i))\right)$	$O\left(\log^2 \lvert S \rvert + \max_{i=0}^{n-1} S(f(i))\right)$
`map` $f\ S$	$O\left(\sum_{s \in S} W(f(s))\right)$	$O\left(\log^2 \lvert S \rvert + \max_{s \in S} S(f(s))\right)$
`filter` $f\ S$	$O\left(\sum_{s \in S} W(f(s))\right)$	$O\left(\log^2 \lvert S \rvert + \max_{s \in S} S(f(s))\right)$
`reduce` $f\ i\ S$	$O\left(\sum_{f(a,b) \in \mathcal{O}_r(f,i,S)} W(f(a,b))\right)$	$O\left(\log^3 \lvert S \rvert \max_{f(a,b) \in \mathcal{O}_r(f,i,S)} S(f(a,b))\right)$

where $\mathcal{O}_r(f, i, S)$ represents the set of applications of f for `reduce` as defined in the documentation.

4 Treaps

In our analysis of the expected depth of a key in a treap, we made use of the following indicator random variable

$$A_i^j = \begin{cases} 1 & j \text{ is an ancestor of } i \\ 0 & \text{otherwise} \end{cases}$$

where i and j refer to the i^{th} and $j^{th)}$ largest keys in the treap, respectively.

Task 4.1 (10%). For a treap of size n, let S_i be the size of the subtree rooted at key i. Write an expression for S_i in terms of these indicator random variables. Then derive a closed-form expression for $\mathbf{E}\left[S_i\right]$ in terms of harmonic numbers H_n and in big-O notation.

5 Testing

We provide a separate structure `Test` in file `test.sml` for your testing. You shouldn't put any testing code in your `TreeSequence` implementation, as it can make it difficult for us to test your code when you turn it in.

It's up to you how to write your tests, but we provide an example test of some of the functions we have provided to show you one way that might work well. You will probably find comparing against `ArraySequence` useful, especially for testing that your implementation meets the `SEQUENCE` specifications. You can run the example tests using `Test.run()` after running `CM.make` in the REPL.