Notes about Purely Functional Data Structures

Gene Michael Stover

created Sunday, 2005 November 27
updated Sunday, 2006 March 12
Contents

1 What is this? 5

2 Answers to Exercises in Chapter 2 7
   2.1 Exercise 2.1 ................................................. 8
   2.2 Exercise 2.2 .................................................. 9
   2.3 Exercise 2.3 .................................................. 9
   2.4 Exercise 2.4 .................................................. 10
   2.5 Exercise 2.5 .................................................. 11
   2.6 Exercise 2.6 .................................................. 13

3 Answers to Exercises in Chapter 3 15
   3.1 Exercise 3.1 .................................................. 15
   3.2 Exercise 3.2 .................................................. 17
   3.3 Exercise 3.3 .................................................. 18
       3.3.1 Discussion of the three solutions .................. 20
   3.4 Exercise 3.4 (Cho & Sahni) .............................. 22
       3.4.1 Part A .................................................... 22
       3.4.2 Part B .................................................... 22
       3.4.3 Part C .................................................... 23
       3.4.4 Part D .................................................... 24
   3.5 Exercise 3.5 .................................................. 25
   3.6 Exercise 3.6 .................................................. 27
   3.7 Exercise 3.7 .................................................. 29
   3.8 Exercise 3.8 .................................................. 31
   3.9 Exercise 3.9 .................................................. 31

4 Answers to Exercises in Chapter 4 33
   4.1 Exercise 4.1 .................................................. 33
   4.2 Exercise 4.2 .................................................. 34

5 Fundamentals of Amorization 37
CONTENTS

6 Amortized Queues Performance Comparison 41
   6.1 Interface to Queues ........................................ 41
   6.2 Implementions of Queues .................................... 41
      6.2.1 Naıve Queues ........................................... 42
      6.2.2 Batch Queues ........................................... 42
      6.2.3 Banker’s Queues ......................................... 42
      6.2.4 Physicist’s Queues ...................................... 42
   6.3 The Performance Results ...................................... 42
   6.4 Observations about Memoization & Memory Use ............ 44
   6.5 Analysis ...................................................... 45

7 Other Observations 47
   7.1 NIL & object-oriented programming ........................... 47
   7.2 Nil for empty & special cases ................................ 47
   7.3 Memoization & memory ......................................... 48
   7.4 Lisp & suspensions ............................................ 48

8 The Source Code 49

A Other File Formats 51

B Some comments about suspend & force 53

C Source Code (amo.lisp) 55

D Source Code (zstream.lisp) 63
Chapter 1

What is this?

This is still under construction as of 2006 January 10. I plan to finish it Some Time Soon.

These are my notes from Purely Functional Data Structures [2], by Chris Okasaki. It includes my answers to exercises & a performance comparison that was inspired by the book.
Chapter 2

Answers to Exercises in Chapter 2

Some of the exercises rely on the sets, binary search trees, & Unbalanced Sets that are presented in [2] so I had to implement those in Lisp.

For the Ordered interface, I created a CLOS protocol. I did not create a CLOS protocol for sets & binary search trees for these reasons:

1. Given: I wanted to support multiple, co-existent implementations of `member` & `insert` so I could run performance comparisons between them.

2. If your Lisp implementation of binary search trees is analogous to the Standard ML implementation in [2], which is what I wanted, you can’t specialize methods on it because all of your binary search trees will have the type `cons` as far as Lisp is concerned.

3. If you create types for your different implementations of binary search trees so you can use CLOS, but you retain similarity with the Standard ML implementations by using NIL as an empty tree, your MEMBER methods will work just fine by re-using a MEMBER that is specialized for NIL. In fact, that’s very similar to how it’s done in the book. Your INSERT methods cannot all use the same INSERT specialized for NIL because INSERT on NIL must create trees of a particular type, but your INSERT method specialized for NIL won’t know what that type is.

4. To overcome the problem in the previous item, you might use a flag in your classes to indicate a terminal node. Then your INSERT methods would work on empty trees because they would have full type information for the tree. This would work, I guess, but it’s ugly, so I’m not going to do it.

5. You could implement the “business end” of the data structure with lists, like in [2], then wrap it in an object whose type would give INSERT all the information it needed. I would probably do something like this in a
production application, but it creates complexity which might hide the point of the exercises from the book. So though I like this idea, I will not use it here.

The different insert & member functions in the book operate on the same binary search tree structures, so I chose to create a different function for each implementation of insert & member.

I created an interface for ORDERED because that’s the way it’s done in [2], but in a real application, I would prefer to give the comparator as a parameter to the tree, not as part of the objects held in the tree.

The source code for ORDEREDs is in ordered.lisp.
The source code for sets, & trees is in chapter02.lisp.

2.1 Exercise 2.1

Write a function suffixes of type alist → alistlist that takes a list xs & returns a list of all the suffixes of xs in decreasing order of length. For example,

suffixes [1, 2, 3, 4] = [[1, 2, 3, 4], [2, 3, 4], [3, 4], [4], []]

Show that the resulting list of suffixes can be generated in O(n) time & represented in O(n) space.

Here’s a suffixes function in Lisp.

(defun suffixes (lst)
  (if (endp lst)
      '(()
      (cons lst (suffixes (rest lst))))
=> SUFFIXES

(suffixes '(1 2 3 4))
=> ((1 2 3 4) (2 3 4) (3 4) (4) NIL)

(suffixes ())
=> (NIL)

Show time: To generate the list of suffixes, we walk the original list, one element at a time. If we call suffixes on an empty list, it does not recurse, so we call suffixes once. If we call suffixes on a list of 1 element, it recursively calls itself once, so we call suffixes twice. In general, we call suffixes \(N + 1\) times for a list of length \(N\), which is \(O(N)\).

Show space: Suffixes creates a list of \(N + 1\) elements. Each element’s cons cell has a car which points to an element in the original list, so only the cons cells of the new list are freshly allocated. There will be \(N + 1\) cons cells in the new list, so the space requirement is \(O(N)\).
2.2 Exercise 2.2

In the worst case, member performs approximately $2d$ comparisons, where $d$ is the depth of the tree. Rewrite member to take no more than $d + 1$ comparisons by keeping track of a candidate element that might be equal to the query element (say, the last element for which $<$ returned false or $\leq$ returned true) & checking for equality only when you hit the bottom of the tree.

The source code for my answer is in the file chapter02.lisp. Load it & search for “Exercise 2.2”. Here’s my discussion.

```lisp
(labels
  ((member3 (x tree candidate)
     (if (null tree)
         (xeq x candidate)
     (let ((value (bst-value tree)))
         (if (xlt x value)
             (member3 x (bst-left tree) candidate)
         ;; else
             (member3 x (bst-right tree) value))))))

(defun tree-member-2-2 (x tree)
  ;; This IF is not a performance optimization; it would be
  ;; dumb to use it as a performance optimization. Instead,
  ;; it’s the easiest way to keep NIL out of the candidate.
  (if tree
      (member3 x tree (bst-value tree))
  nil)))

I took the hint from the exercise & used it in an obvious way. Most of the work happens in the helper function, member3. One twist is in the tree-member-2-2 function, when we check that tree is not NIL. That is not a performance optimization. It keeps NIL out of the candidate argument for member3. We want to do that because member3 is simpler if it need not worry about a NIL candidate. Why? Because we don’t want to bother to specialize the comparison methods for NIL.

My answer, with test programs, is in chapter02.lisp.

2.3 Exercise 2.3

Inserting an existing element into a binary search tree copies the entire search path even though the copied nodes are indistinguishable from the originals. Rewrite insert using exceptions to avoid this copying. Establish only one handler per insertion rather than one handler per iteration.
It’s interesting that the exercise bothers to say that there should be only one active exception handler, not one per call.

Here’s my answer. This source code & some test programs are in chapter02.lisp.

```
(labels
  ((insert2 (x tree root)
    (if (null tree)
      (make-bst :value x :left nil :right nil)
      (let ((value (bst-value tree)))
        (cond ((xlt x value)
          (make-bst :value value
            :left (insert2 x (bst-left tree) root)
            :right (bst-right tree)))
          ((xgt x value)
            (make-bst :value value
              :left (bst-left tree)
              :right (insert2 x (bst-right tree) root)))
          (t (throw 'already-a-member root))))))
)

(defun insert-2-3 (x tree)
  (catch 'already-a-member
    (insert2 x tree tree)))
```

### 2.4 Exercise 2.4

*Combine the ideas of the previous two exercises to obtain a version of `insert` that performs no unnecessary copying & uses no more than \(d + 1\) comparisons.*

Here is my answer to Exercise 2.4. The source code & test programs are also in chapter02.lisp.

```
(labels
  ((insert4 (x tree candidate root)
    (cond ((equal x candidate)
      (throw 'already-a-member root))
      ((endp tree)
        (make-tree x))
      ((< x (tree-value tree))
        (insert4 x (tree-left tree) candidate root))
      (t
        (insert4 x (tree-right tree) (tree-value tree) root)))
    (defun tree-insert-2-4 (x tree)
      (catch 'already-a-member
        (insert2 x tree (tree-value tree))))
```

```
2.5 Exercise 2.5

Sharing can also be useful within a single object, not just between objects. For example, if the two subtrees of a given node are identical, then they can be represented by the same tree.

(a) Using this idea, write a function complete of type Elem × int → Tree where complete (x, d) creates a complete binary tree of depth d with x stored in every node. (Of course, this function makes no sense for the set abstraction, but it can be useful as an auxiliary function for other abstractions, such as bags.) This function should run in O(d) time.

(b) Extend this function to create balanced trees of arbitrary size. These trees will not always be complete binary trees, but should be as balanced as possible: for any given node, the two subtrees should differ in size by at most one. This function should run in O(log n) time. (Hint: use a helper function create2 that, given a size m, creates a pair of trees, one of size m and one of size m + 1.)

Here is my answer to Exercise 2.5.a. The source code & test programs are also in chapter02.lisp.

(defvar *complete-2-5-count* 0)

(defun complete-2-5 (x d)
    (incf *complete-2-5-count*)
    (if (zerop d)
        nil
        (let ((tree (complete-2-5 x (1- d))))
            (make-bst :value x :left tree :right tree))))

The print-table-2-5-a function creates a table of run-times & call counts for different values of N. The run-time increases so slowly with d that the times mostly show a bunch of zeros. So the most useful performance measurement is the call counter, which is in the third column. Table 2.1 shows the results from print-table-2-5-a.

Here is my answer for part B:

(defvar *balanced-2-5-count* 0)

(defun balanced-2-5 (x n)
    "Given a size, create a tree of size $N$ & populate it with $X$. The tree will be more-or-less balanced."
    (incf *balanced-2-5-count*)
    (cond ((zerop n) nil)
            ((eql n 1) (make-bst :value x :left nil :right nil))
            ((evenp (1- n))
                (let ((left (balanced-2-5 x (1- n))
                        (right (balanced-2-5 x (1- n))))
                    (make-bst :value x :left left :right right))))

Looks like the number of calls is $O(d + 1)$, which is linear as requested.

Here is my answer for part B:
Table 2.1: Performance measurements for function $\text{complete-2-5}$

<table>
<thead>
<tr>
<th>$n$</th>
<th>seconds</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00e+0</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0.00e+0</td>
<td>101</td>
</tr>
<tr>
<td>200</td>
<td>0.00e+0</td>
<td>201</td>
</tr>
<tr>
<td>300</td>
<td>0.00e+0</td>
<td>301</td>
</tr>
<tr>
<td>400</td>
<td>0.00e+0</td>
<td>401</td>
</tr>
<tr>
<td>500</td>
<td>0.00e+0</td>
<td>501</td>
</tr>
<tr>
<td>600</td>
<td>0.00e+0</td>
<td>601</td>
</tr>
<tr>
<td>700</td>
<td>0.00e+0</td>
<td>701</td>
</tr>
<tr>
<td>800</td>
<td>0.00e+0</td>
<td>801</td>
</tr>
<tr>
<td>900</td>
<td>0.00e+0</td>
<td>901</td>
</tr>
<tr>
<td>1000</td>
<td>0.00e+0</td>
<td>1001</td>
</tr>
</tbody>
</table>

Table 2.2: Performance measurements for function $\text{balanced-2-5}$

<table>
<thead>
<tr>
<th>$n$</th>
<th>seconds</th>
<th>calls</th>
<th>$\text{fraccalls}/\lg N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00e+0</td>
<td>3</td>
<td>3.00e+0</td>
</tr>
<tr>
<td>3002</td>
<td>0.00e+0</td>
<td>334</td>
<td>2.89e+1</td>
</tr>
<tr>
<td>6002</td>
<td>0.00e+0</td>
<td>523</td>
<td>4.17e+1</td>
</tr>
<tr>
<td>9002</td>
<td>0.00e+0</td>
<td>965</td>
<td>7.35e+1</td>
</tr>
<tr>
<td>12002</td>
<td>0.00e+0</td>
<td>1247</td>
<td>8.99e+1</td>
</tr>
<tr>
<td>15002</td>
<td>10.00e-3</td>
<td>1420</td>
<td>1.00e+2</td>
</tr>
<tr>
<td>18002</td>
<td>0.00e+0</td>
<td>891</td>
<td>6.21e+1</td>
</tr>
<tr>
<td>21002</td>
<td>10.00e-3</td>
<td>907</td>
<td>6.23e+1</td>
</tr>
<tr>
<td>24002</td>
<td>0.00e+0</td>
<td>1305</td>
<td>8.87e+1</td>
</tr>
</tbody>
</table>

It & its test programs are in chapter02.lisp.

As with the answer for part A, it runs so quickly that measuring the work in real-time is difficult, so I measured the work by counting the number of calls. Table 2.2 shows the performance results. They were generated by function $\text{print-table-2-5-b}$ which is also in chapter02.lisp.

Does it run in $O(\log N)$ time? I’m not sure. I guess it does, but I would feel more confident is the values in the $\frac{\text{calls}}{\lg N}$ column varied less.
Exercise 2.6

Adapt the UnbalancedSet functor to support finite maps rather than sets. Figure 2.10\(^1\) gives a minimal signature for finite maps. (Note that the NotFound exception is not predefined in Standard ML – you will have to define it yourself. Although this exception could be made part of the FiniteMap signature, with every implementation defining its own NotFound exception, it is convenient for all finite maps to use the same exception.)

I’m not going to change any of the Binary Search Tree code I’ve written. Instead, I’ll create a Finite Map Node class & comparison methods for it. Then I should be able to create Finite Maps by inserting Finite Map Nodes into binary search trees.

My answer to exercise 2.6 is in the file chapter02.lisp. Search for “Exercise 2.6”; that’s where the answer begins.

Here are some important parts of the solution:

```lisp
(defclass finite-map-node ()
  ((key :initarg :key :accessor finite-map-node-key)
   (value :initarg :value :accessor finite-map-node-value)))

(defmethod xeq ((x finite-map-node) (y finite-map-node))
  (xeq (finite-map-node-key x) (finite-map-node-key y)))

(defmethod xgeq ((x finite-map-node) (y finite-map-node))
  (xgeq (finite-map-node-key x) (finite-map-node-key y)))

(defmethod xgt ((x finite-map-node) (y finite-map-node))
  (xgt (finite-map-node-key x) (finite-map-node-key y)))

(defmethod xlt ((x finite-map-node) (y (eql nil)))
  nil)

(defmethod xleq ((x finite-map-node) (y finite-map-node))
  (xleq (finite-map-node-key x) (finite-map-node-key y)))

(defun bind (key value map)
  (insert (make-instance 'finite-map-node :key key :value value) map))

;; This LOOKUP function satisfied the interface described in
;; Okasaki except that it does not throw. When the key is
;; not found in the map, instead of tossing an exception, it
;; returns NIL. So a key bound to NIL is indistinguishable
;; from a key that is not in the table. I implemented it this
;; way because (a) it resembles the semantics of Common Lisp’s
;; hash tables & (b) I dislike exceptions.
```

\(^1\)That’s Figure 2.10 in [2], page 16.
(defun lookup (x map)
  (let ((node (xmember (make-instance 'finite-map-node
                                          :key x :value nil)
                          map)))
    (and node
         (finite-map-node-value node)))))
Chapter 3

Answers to Exercises in Chapter 3

Some of the code for Chapter 3 in [2] use an ordered interface. My Lisp implementation of that interface is in ordered.lisp.

3.1 Exercise 3.1

Prove that the right spine of a leftist heap of size \( n \) contains at most \( \lfloor \log(n + 1) \rfloor \) elements. (All logarithms in this book are base 2 unless otherwise indicated.)

1. Think about a leftist heap which has an added restriction of being “as balanced as possible”. Figure 3.1 shows such a tree with seven nodes. Notice that the tree is both a leftist heap & a balanced binary tree.

2. The tree in Figure 3.1 is leftist, balanced, & full. In this state, \( n = 2^{r_b} - 1 \), where \( n \) is the number of nodes & \( r_b \) is the height of the right subtree.

Figure 3.1: An example of a “balanced as possible” leftist heap of seven nodes
CHAPTER 3. ANSWERS TO EXERCISES IN CHAPTER 3

This rule applies to the tree in Figure 3.1 & to any other leftist, balanced, full tree.

3. If we add a node to the tree, the leftist property forces the new node to become a left child of the left-most node. After that addition, \( n \geq 2^{r_b} - 1 \).

4. As we add nodes one at a time, the leftist property causes the new nodes to fill open left slots before filling open right slots, but the “balanced as possible” property prevents the new nodes from starting a new level until the current level is full. Eventually, the lowest level fills & we again have a leftist, balanced, full tree. Until the tree is again full, it is always true that \( n \geq 2^{r_b} - 1 \). When the tree is again full, it is true that \( n = 2^{r_b} - 1 \), but it will still be true that \( n \geq 2^{r_b} - 1 \).

5. So in a “balanced as possible”, leftist heap/tree, it is always true that \( n \geq 2^{r_b} - 1 \).

6. Now consider a leftist heap that does not have the “balanced as possible” property. As we add nodes to it, the leftist property forces them to fill open left-side slots before filling right-side slots, but without a “balanced as possible” property, they may start a new level before filling the current lowest level. In other words, in a leftist heap, the left subtree may increase in depth more quickly than can the left subtree in a “balanced as possible” leftist heap. So in a leftist heap, the depth, \( r \), of the right subtree may be less than or equal to the depth, \( r_b \), of a “balanced as possible” leftist heap with the same number of nodes. So \( r \leq r_b \).

7. So \( 2^r - 1 \leq 2^{r_b} - 1 \leq n \).

8. \( 2^r - 1 \leq n \)

9. \( 2^r \leq n + 1 \)

10. \( \log 2^r \leq \log(n + 1) \)

11. \( r \leq \log(n + 1) \)

12. Because \( r \) is integral, \( r \leq \lfloor \log(n + 1) \rfloor \)

13. The depth, \( r \), of the right-most subtree is also the length of the right spine.

14. So the right spine contains \( r \) nodes, \( r \leq \lfloor \log(n + 1) \rfloor \).

It’s hardly the shortest proof in the world, but it’s all me. I didn’t look in any books to see how it was done until I had finished this proof.
3.2 Exercise 3.2

Define insert directly rather than via a call to merge.

My answer is in the file chapter03.lisp. To find it & its test programs, search for “defun insert-3-2”. Here is a copy of insert-3-2 function.

(defun insert-3-2 (x heap)
    "Insert item X into the HEAP, returning a new heap, & without calling XMERGE."
    (cond ((null heap)
        ;; If we insert an item into an empty heap, the new
        ;; heap has just one item -- our new item. Our
        ;; MAKET function does that.
        (maket x nil nil))
      ((xleq x (heap-x heap))
        ;; The new item X comes before the current node's item.
        ;; So we make a new heap with X as its item, & we push
        ;; the current node's item into one of the subtrees.
        (makeT x (heap-left heap) (insert-3-2 (heap-x heap)
            (heap-right heap))))
      (t
        ;; X comes after the current node, so we recursively insert
        ;; it into a subtree.
        (makeT (heap-x heap)
            (heap-left heap)
            (insert-3-2 x (heap-right heap))))))

Other than the trivial case of inserting an item into an empty heap, insert-3-2 works by walking the tree. At any node in the tree, there are two cases (besides the trivial case).

In the first case, \(x \leq \text{node's} x\). When that happens, we need to make a new node for \(x\) & then insert the current node’s \(x\) into one of the subtrees.

In the second case, \(x > \text{node's} x\). When that happens, we recursively insert \(x\) into one of the subtrees.

Which subtree do we use? It’s nice to make an effort to keep the tree somewhat balanced, so we should insert the item into the subtree with the smallest rank. We could check the ranks of the subtrees, but if we remember that the right subtree’s rank is never greater than the rank of the left subtree, we can skip the check. So we always insert into the right subtree. Our makeT helper function will ensure that the subtree with the largest rank becomes the new left subtree.

I’m curious to see why Okasaki gave this as an exercise. Is there an advantage to an insert function which does its own work instead of passing the buck to merge? Let’s do a performance comparison to see.
Table 3.1: Result of comparing the plain `INSERT` to my answer for Exercise 3.2

Table 3.1 shows the results of my performance comparison. I don’t see any benefit to the `INSERT` function which does all of its own work. I’m a little surprised by that.

Here is the Lisp expression I used to make the performance comparison.

```
* (in-package "CHAPTER-03")

* (let ((lst (loop for i from 1 to 10000
                    collect (random 100))))
  (labels
    ((test (insert)
            ;; Insert a bunch of items into an empty heap.
            (declare (type function insert))
            (do ((x lst (rest x))
                 (heap nil (funcall insert (first x) heap)))
                ((endp x) heap))
      (test-with-xmerge ()
        ;; Call TEST using INSERT
        (test #'insert))
    (test-3-2 ()
      ;; Call TEST with INSERT-3-2.
      (test #'insert-3-2)))
  (with-open-file (strm "tab-ex0302.tex" :direction :output
                        :if-exists :rename-and-delete)
    (cybertiggyr-test:ratetable
     (list (list "INSERT uses XMERGE" #'test-with-xmerge)
           (list "INSERT self-contained" #'test-3-2))
     strm))))

#<FILE-STREAM "tab-ex0302.tex">

3.3 Exercise 3.3

Implement a function `fromList` of type `Elem.T list → Heap` that produces a leftist heap from an unordered list of elements by first converting each element into a singleton heap & then merging the heaps until only one heap remains. Instead of merging the heaps in one right-to-left or left-to-right pass using `foldr` or `foldl`, merge

<table>
<thead>
<tr>
<th>function</th>
<th>count</th>
<th>seconds</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT uses XMERGE</td>
<td>57</td>
<td>3.04e+0</td>
<td>1.88e+1</td>
</tr>
<tr>
<td>INSERT self-contained</td>
<td>23</td>
<td>3.03e+0</td>
<td>7.59e+0</td>
</tr>
</tbody>
</table>
3.3. EXERCISE 3.3

the heaps in $\lceil \log n \rceil$ passes, where each pass merges adjacent pairs of heaps. Show that fromList takes only $O(n)$ time.

My answer is in the file chapter03.lisp. To find it & its test programs, search for “defun from-list-01”. It’s called FROM-LIST-01 because I wrote three solutions; FROM-LIST-01 is the fastest. Here is a copy of FROM-LIST-01 function:

(labels
 (m02 (lst)
   (do ((x lst (mapcar #'(lambda (lst2)
                 (apply #'xmerge lst2))
             (pairs x))
       ((<= (length x) 1)
            (first x))))

(defun from-list-02 (lst)
  (declare (type list lst))
  (m02
   ;; Convert each item in LST into a single-element heap.
   (mapcar #'(lambda (x) (makeT x nil nil)) lst))))

The FROM-LIST-02 function converts the list of items into a list of single-item leftist heaps, then passes the buck to its helper function, m02. m02 repeatedly splits the list into pairs, merges the heaps in each pair, & then repeats by splitting the new list into pairs.

Given a list of length $n$, m02 calls xmerge for each of the $\frac{n}{2}$ pairs, then performs same operation on the new list of $\frac{n}{2}$ items. So for a list of length $n$, the number of merges is $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} \cdots + 1 = n$. So from-list-01 runs in $O(N)$ time.

So much for the theory. Table 3.2 shows the results of an actual performance run.

Remember that the rate from the fourth column measures function calls per second, & the work of one of those function calls is proportional to $N$. So to convert the rate from calls-per-second to items-per-second, we must multiply by $N$.

The fifth column shows the rate times $N$. The values are very nearly the same. In fact, I’m impressed that they are so consistent. So they effectively fit a line, & that line is $O(N)$. So FROM-LIST-02 runs in $O(N)$ time both in theory & in practice.

Here is the expression I used for the performance run in Table 3.2. I added the fifth column by hand.

* (with-open-file (strm "tab-ex0303a.tex" :direction :output
 :if-exists :rename-and-delete)

(labels
 ((make-test (n)
     (declare (type integer n)))

*
3.3.1 Discussion of the three solutions

I wrote three solutions. They are called FROM-LIST-00, FROM-LIST-01, & FROM-LIST-02. All are in the file chapter03.lisp. All three solutions have the same form: A FROM-LIST-* function that calls a local helper function. Each FROM-LIST-* function applies makeT to each element in the list to obtain a list of single-item leftist heaps. Then the FROM-LIST-* function calls its helper function on that new list.

FROM-LIST-00 is, to my mind, the obvious solution. If the list is empty, it returns an empty list. If the list has one element (which is a heap, thanks to the enclosing FROM-LIST-00 function), it returns that element. If the list has two elements, it merges them & returns the result. Otherwise, it calls itself on each half of the list & merges the two results.

Table 3.2: Results of performance run of FROM-LIST-02

<table>
<thead>
<tr>
<th>function</th>
<th>count</th>
<th>seconds</th>
<th>rate</th>
<th>rate × n</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>352597</td>
<td>3.05e+0</td>
<td>1.16e+5</td>
<td>2.32e+5</td>
</tr>
<tr>
<td>4</td>
<td>202727</td>
<td>3.00e+0</td>
<td>6.76e+4</td>
<td>2.70e+5</td>
</tr>
<tr>
<td>8</td>
<td>105340</td>
<td>3.00e+0</td>
<td>3.51e+4</td>
<td>2.81e+5</td>
</tr>
<tr>
<td>16</td>
<td>52065</td>
<td>3.00e+0</td>
<td>1.74e+4</td>
<td>2.78e+5</td>
</tr>
<tr>
<td>32</td>
<td>26303</td>
<td>3.00e+0</td>
<td>8.77e+3</td>
<td>2.81e+5</td>
</tr>
<tr>
<td>64</td>
<td>12225</td>
<td>3.00e+0</td>
<td>4.07e+3</td>
<td>2.60e+5</td>
</tr>
<tr>
<td>128</td>
<td>6295</td>
<td>3.00e+0</td>
<td>2.10e+3</td>
<td>2.69e+5</td>
</tr>
<tr>
<td>256</td>
<td>3043</td>
<td>3.00e+0</td>
<td>1.01e+3</td>
<td>2.59e+5</td>
</tr>
<tr>
<td>512</td>
<td>1566</td>
<td>3.00e+0</td>
<td>5.22e+2</td>
<td>2.67e+5</td>
</tr>
<tr>
<td>1024</td>
<td>775</td>
<td>3.04e+0</td>
<td>2.55e+2</td>
<td>2.61e+5</td>
</tr>
<tr>
<td>2048</td>
<td>394</td>
<td>3.00e+0</td>
<td>1.31e+2</td>
<td>2.68e+5</td>
</tr>
<tr>
<td>4096</td>
<td>170</td>
<td>3.00e+0</td>
<td>5.66e+1</td>
<td>2.32e+5</td>
</tr>
<tr>
<td>8192</td>
<td>88</td>
<td>3.01e+0</td>
<td>2.93e+1</td>
<td>2.40e+5</td>
</tr>
<tr>
<td>16384</td>
<td>39</td>
<td>3.06e+0</td>
<td>1.28e+1</td>
<td>2.10e+5</td>
</tr>
</tbody>
</table>
3.3. EXERCISE 3.3

<table>
<thead>
<tr>
<th>function</th>
<th>count</th>
<th>seconds</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>from-list-00</td>
<td>30</td>
<td>3.02e+0</td>
<td>9.92e+0</td>
</tr>
<tr>
<td>from-list-01</td>
<td>70</td>
<td>3.02e+0</td>
<td>2.32e+1</td>
</tr>
<tr>
<td>from-list-02</td>
<td>70</td>
<td>3.03e+0</td>
<td>2.31e+1</td>
</tr>
</tbody>
</table>

Table 3.3: Results of performance comparison of the three FROM-LIST-* functions

My gut feeling said that splitting the list into two halves is inefficient, so I wrote FROM-LIST-01. Its helper function has the same three special cases, but in its general case, it splits the list into pairs, merges the two elements in each pair to get a shorter list of larger heaps, & then calls itself on the new list.

Since FROM-LIST-01 used simple tail recursion, I was curious to see the results of removing that recursion by hand, so I wrote FROM-LIST-02. It uses the same pair-making logic as FROM-LIST-01, but it does so in a loop, & I took advantage of the fact that FIRST NIL is NIL to collapse the case into a single expression.

I compared the performances of the three FROM-LIST-* functions, & the results are in Table 3.3. The table shows that FROM-LIST-01, the implementation that splits into pairs & uses a single, tail-recursive call, is the fastest by a hair's breadth. The iterative solution, FROM-LIST-02, is neck-&-neck with it. In fact, I ran the performance comparison many times, & FROM-LIST-02 was occasionally faster than FROM-LIST-01 by one or two iterations in the count column.

Even though it's not the fastest, I prefer FROM-LIST-02 because it's more understandable, in my opinion.

Here is the expression I used for the performance comparison:

```lisp
* (in-package "CHAPTER-03")

#<PACKAGE "CHAPTER-03">
* (let ((biglst (loop for i from 1 to 10000 collect (random 100))))
  (labels
    ((test (fn)
      "Call FN on BIGLST. Assume FN is one of the FROM-LIST-*
      functions."
      (declare (type function fn))
      (funcall fn biglst))
    (m00 () (test #'from-list-00))
    (m01 () (test #'from-list-01))
    (m02 () (test #'from-list-02))))

(with-open-file (strm "tab-ex0303b.tex" :direction :output
  :if-exists :rename-and-delete)
  (cybertiggyr-test:ratetable
    (list (list "from-list-00" #'m00)
          (list "from-list-01" #'m01)
          (list "from-list-02" #'m02)))
```

```
3.4 Exercise 3.4 (Cho & Sahni)

Weight-biased leftist heaps are an alternative to leftist heaps that replace the leftist property with the weight-biased leftist property; the size of any left child is at least as large as the size of its right sibling.

(a) Prove that the right spine of a weight-biased leftist heap contains at most \( \lfloor \log(n + 1) \rfloor \) elements.

(b) Modify the implementation in Figure 3.2\(^1\) to obtain weight-biased leftist heaps.

(c) Currently, \texttt{merge} operates in two passes: a top-down pass consisting of calls to \texttt{merge}, & a bottom-up pass consisting of calls to the helper function \texttt{makeT}. Modify \texttt{merge} for weight-biased leftist heaps to operate in a single, top-down pass.

(d) What advantages would the top-down version of \texttt{merge} have in a lazy environment? In a concurrent environment?

3.4.1 Part A

The proof from Exercise 3.1 (Section ??) applies to weight-biased leftist heaps if we change “leftist property” to “weight-biased leftist property”. Significantly, in the new version of step 3 of the proof, the weight-biased leftist property still forces new nodes to fill the left-most leaf nodes.

3.4.2 Part B

To make a weight-biased leftist heap system from the leftist heap system developed earlier in the chapter, the main change is to replace the \texttt{rank} function with a \texttt{weight} function. We also change the \texttt{makeT} function to call the new \texttt{weight} function instead of the \texttt{rank} function. I also created a new weight-biased leftist heap structure (called \texttt{wheap}), & copied all the leftist heap functions to weight-biased leftist heap functions & placed a “w” character at the beginning of their names.

The whole system & test programs are in the file chapter03.lisp; search for “defstruct wheap”. Here are the important parts of the change (the structure, the weight function, & the new \texttt{wmakeT} function).

\begin{verbatim}
(defstruct wheap weight x left right)

(defun weight (heap)
  (cl-floor (log (1+ (length heap)) 2))
)
\end{verbatim}

\(^1\)It means Figure 3.2 in [2], page 20.
"Return the weight of the heap. An empty heap has weight 0."
(if heap
  (wheap-weight heap)
0))

(defun wmaket (x a b)
  (if (>= (weight a) (weight b))
      (make-wheap :weight (+ (weight a) (weight b) 1)
                   :x x :left a :right b)
    ;; else
    (make-wheap :weight (+ (weight a) (weight b) 1)
                :x x :left b :right a)))

3.4.3 Part C

The exercise asks us to remove the makeT call that wraps merge’s recursive call. If we can place the makeT elsewhere so that it doesn’t wrap the recursive merge, we’ll have a tail-recursive merge.

Notice that when we call makeT, the order of the two child subtrees does not matter because makeT will ensure that the heavier subtree is on the left. We can use this detail to keep a stack of arguments for calling makeT. Each element of that stack has the X for a node & one of the subtrees for the node.

My answer is in the file chapter03.lisp; search for “defun wmerge-3-4c”. Here is a copy of that function:

(labels
  ((m00 (stack a b)
        (cond ((null a)
               ;; Use the stack to call WmakeT to build the heap.
               (reduce #’(lambda (&optional x y)
                             (cond ((null y) x)
                                   ((null x) y)
                                   ((typep x ’wheap)
                                    (WmakeT (first y) (second y) x))
                                   (t ; X is a list, Y is a heap
                                    (WmakeT (first x) (second x) b)))))
               stack
               :initial-value b))
       ((null b)
        ;; Like the previous case, but B is empty. Instead
        ;; of duplicating the code, we’ll swap A & B to use
        ;; the previous chunk of code.
        (m00 stack b a))
       (xleq (wheap-x a) (wheap-x b))
       ;; A’s X goes into a new node. A’s left will be
       ;; one of the subtrees. We’ll merge A’s right &
### 3.4.4 Part D

I don’t see any benefit to my `wmerge-3-4c` function in a lazy environment. It still does all the work that the original down-up merge did. I’m pretty sure it would not be any easier for the language system to memoize. So I don’t see any advantage.

Likewise, I see no benefit to my `wmerge-3-4c` function in a concurrent environment. Each call to `wmerge-3-4c` does all its work before passing

\[\text{Actually, this new merge function is a trivial wrapper around its helper function, which is tail-recursive.}\]
the buck to another instance of wmerge-3-4c. In fact, if you eliminated the recursion by hand, wmerge-3-4c could be implemented as a single loop that does all the work while accumulating a stack, then a second loop that does the reduce work on the stack. The first loop would not be naturally parallelizable, though maybe the reduce could operate by calling the function on pairs, in parallel, until only one element remained in the list.

As I said in Section 3.4.3, I suspect I did not solve the exercise as Okasaki planned.

### 3.5 Exercise 3.5

For this & the next few exercises, I translated Okasaki's implementation of binomial heaps ([2], page 24) to Lisp. It is in the file binomial-heap.lisp.

*Define findMin directly rather than via a call to removeMinTree.*

Here is my answer:

```lisp
(defun findMin-3-5 (heap)
  (assert (binomial-heap:is-heap heap))
  (let ((min0 (binomial-heap:root (first heap))))
    (if (= (length heap) 1)
      min0
      ;; else
      (let ((min1 (findMin-3-5 (rest heap))))
        (if (xleq min0 min1)
          min0
          ;; else
          min1)))))
```

My answer & test programs for it are also in the file chapter03.lisp. Search for “defun findMin-3-5”.

My guess is that the point of Exercise 3.5 was to improve the performance of findMin.

Before running a performance comparison, let’s predict its results. Simply removing findMin’s call to removeMinTree won’t change performance much; it’ll just remove one function call, which will have little effect. A bigger difference is that removeMinTree constructs lists as it returns, whereas the self-contained findMin only deconstructs heaps. I suspect this could have a noticeable effect on large heaps. So I predict that findMin-3-5 will be faster for large heaps.

Table 3.4 shows the results of the performance test. The rows are in pairs. The first two rows compare findMin & findMin-3-5 for heaps of 1,000 elements. The next pair of rows compare them for heaps of 10,000 elements.

My findMin-3-5 is faster, so I presume that was the point of the exercise.

Here is the Lisp expression I used for the comparison:
CHAPTER 3. ANSWERS TO EXERCISES IN CHAPTER 3

<table>
<thead>
<tr>
<th>function</th>
<th>count</th>
<th>seconds</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>findMin-3-5 1000</td>
<td>365778</td>
<td>3.04e+0</td>
<td>1.20e+5</td>
</tr>
<tr>
<td>findMin 1000</td>
<td>264264</td>
<td>3.00e+0</td>
<td>8.81e+4</td>
</tr>
<tr>
<td>findMin-3-5 10000</td>
<td>408572</td>
<td>3.00e+0</td>
<td>1.36e+5</td>
</tr>
<tr>
<td>findMin 10000</td>
<td>300420</td>
<td>3.00e+0</td>
<td>1.00e+5</td>
</tr>
<tr>
<td>findMin-3-5 50000</td>
<td>364576</td>
<td>3.00e+0</td>
<td>1.22e+5</td>
</tr>
<tr>
<td>findMin 50000</td>
<td>261948</td>
<td>3.00e+0</td>
<td>8.73e+4</td>
</tr>
</tbody>
</table>

Table 3.4: Performance comparison between findMin & findMin-3-5

(in-package "CHAPTER-03")
=> #<PACKAGE "CHAPTER-03”>

(labels
  ((init-heap (n))
   (declare (type integer n))
   (format t "~A: N is ~S" ’init-heap n)
   ;; Jezus H. Krist. If I don’t have this COERCE, SBCL’s
   ;; compiler (yes, compiler, not during eval) croaks
   ;; because it thinks N will be of type
   ;; (OR (INTEGER 10000 10000) (INTEGER 1000 1000)), which
   ;; isn’t an integer. Dumbshit wannabe optimizing compiler.
   (setq n (coerce n ’integer))
   (do ((heap nil (binomial-heap:insert (random n) heap))
         (i 0 (1+ i)))
       ((>= i n) heap))

(let ((heap1k (init-heap 1000))
       (heap10k (init-heap 10000))
       (heap50k (init-heap 50000)))
  (with-open-file (strm "tab-ex0305.tex" :direction :output
                      :if-exists :rename-and-delete)
    (cybertiggyr-test:ratetable
      (list
       (list "findMin-3-5 1000" #'(lambda () (findMin-3-5 heap1k)))))
    (list "findMin 1000" #'(lambda () (binomial-heap:findMin heap1k)))
    (list "findMin-3-5 10000" #'(lambda () (findMin-3-5 heap10k)))
    (list "findMin 10000" #'(lambda () (binomial-heap:findMin heap10k)))
    (list "findMin-3-5 50000" #'(lambda () (findMin-3-5 heap50k)))
    (list "findMin 50000" #'(lambda () (binomial-heap:findMin heap50k))))
  strm)))
3.6 Exercise 3.6

Most of the rank annotation in this representation of binomial heaps are redundant because we know that the children of a node of rank \( r \) have ranks \( r - 1, \ldots, 0 \). Thus, we can remove the rank annotations from each node and instead pair each tree at the top-level with its rank, i.e.,

\[
\text{datatype } \text{Tree} = \text{Node of Elem} \times \text{Tree list}
\]

\[
\text{type } \text{Heap} = (\text{int} \times \text{Tree}) \text{ list}
\]

Reimplement binomial heaps with this new representation.

If I translate the suggested data structure from the Standard ML of Exercise 3.6 to Lisp, I get this:

\[(\text{defstruct } \text{xtree } x (c \text{ nil}))\]

\[(\text{defstruct } \text{heapnode } r \text{ tree})\]

;; A HEAP is a list of HEAPNODE.

When we re-write some of the binomial heap functions to work with the new types, a general rule will be that functions which access the rank of a tree must operate on HEAPNODES. The other functions which operate on trees can continue to operate on trees. The functions which operate on heaps continue to operate on heaps, though they may need changes in how they call the tree-handling functions because now there is a difference between operating on a tree & operating on a HEAPNODE.

Tree-handling functions which become HEAPNODE-handling functions are rank & insTree.

The unchanged tree-handling function are root.

I still need a rank function which computes the rank of a tree instead of just returning the rank that is stored in a HEAPNODE. I would be very interested in a solution which did not recompute the rank of any trees.

The link function becomes a HEAPNODE function, but the easiest way to implement the recursive part is with a tree-handling function. So link splits into link-nodes, which operates on heap nodes, & link-trees, which operates on trees.

It looks like all the other functions, which operate on heaps, will need changes because, where they used to call tree-handling & other heap-handling functions, they now also sometimes call HEAPNODE-handling functions.

My answer is in the file binomial-heap2.lisp.

If we assume that pointers, fixnums, & many other data types all occupy the same amount of space (a word), then the original implementation of binomial heaps stores the rank of a tree in each node. So each tree node requires space for the payload & for the rank; that’s two words. This new implementation uses
Table 3.5: Comparison of the space requirements of the two binomial heap implementations

<table>
<thead>
<tr>
<th>n</th>
<th>trees</th>
<th>BH space</th>
<th>BH2 space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>30</td>
<td>19</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>62</td>
<td>36</td>
</tr>
<tr>
<td>1023</td>
<td>10</td>
<td>2046</td>
<td>1033</td>
</tr>
</tbody>
</table>

Table 3.6: Performance comparison between original binomial heap implementation & the new one from Exercise 3.6

<table>
<thead>
<tr>
<th>function</th>
<th>count</th>
<th>seconds</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH 100</td>
<td>4284</td>
<td>3.00e+0</td>
<td>1.43e+3</td>
</tr>
<tr>
<td>BH2 100</td>
<td>616</td>
<td>3.00e+0</td>
<td>2.05e+2</td>
</tr>
<tr>
<td>BH 500</td>
<td>786</td>
<td>3.00e+0</td>
<td>2.62e+2</td>
</tr>
<tr>
<td>BH2 500</td>
<td>47</td>
<td>3.04e+0</td>
<td>1.55e+1</td>
</tr>
<tr>
<td>BH 1000</td>
<td>378</td>
<td>3.00e+0</td>
<td>1.26e+2</td>
</tr>
<tr>
<td>BH2 1000</td>
<td>14</td>
<td>3.11e+0</td>
<td>4.50e+0</td>
</tr>
</tbody>
</table>

one word for each tree node & one word for each tree. Table 3.5 compares the space requirements for the two implementations of binomial heaps.

The space savings for this new implementation of binomial heap could be considerable.

Table 3.6 compares run-time performance for the two techniques.

So the first implementation of binomial heap, with the rank in each tree node, is about twenty times faster than the second implementation, which saves space by storing the rank per tree. I'd bet that the time difference has to do with computing the rank of the trees, which the second implementation does when it merges.

This is just one more data point in support of the heuristic that you can always trade space for time. Nevertheless, I'm still open to the possibility of a binomial heap implementation that stores the rank for each tree & never computes the rank of the trees. If there is such an implementation, then I failed Exercise 3.6.

(in-package "COMMON-LISP-USER")

=> #<PACKAGE "COMMON-LISP-USER"> 

(labels
  ((sort1 (lst)
      (utils:heapsort lst nil #'binomial-heap:insert
        #'binomial-heap:findMin
        #'binomial-heap:deleteMin))
   (sort2 (lst))

)
3.7. EXERCISE 3.7

One clear advantage of leftist heaps over binomial heaps is that \texttt{findMin} takes only $O(1)$ time, rather than $O(\log n)$ time. The following functor skeleton improves the running time of \texttt{findMin} to $O(1)$ by storing the minimum element separately from the rest of the heap.

\begin{verbatim}
functor ExplicitMin (H : Heap) : Heap =
  struct
    structure Elem = H.Elem
    datatype Heap = E or NE of Elem.T × H.Heap
  end
end
\end{verbatim}

Note that this functor is not specific to binomial heaps, but rather takes any implementation of heaps as a parameter. Complete this functor so that \texttt{findMin} takes $O(1)$ time, \& \texttt{insert}, \texttt{merge}, \& \texttt{deleteMin} take $O(\log n)$ time (assuming that all four take $O(\log n)$ time or better for the underlying implementation $H$.)

The past few exercises have made me wish I had defined an interface for heaps before doing all these exercises. I chose not to do that because, since all these heap implementations us \texttt{nil} to indicate an empty heap, it's not just a case of creating an interface in CLOS. So I stand by my choices, but this Exercise 3.7 sure makes me reconsider it. (See Section 7.1.)

Here's an answer to Exercise 3.7 in pseudocode:
1. We store the minimum element along with the heap.

2. `findMin` just returns that element, so it runs in $O(1)$ time.

3. `insert` compares the new element to the stored minimum element. If the new element is less, it stores the new element as the minimum element. This extra work is $O(1)$, which is probably less than H’s `insert`, so ExplicitMin’s `insert` will run in the same time at H’s `insert`.

4. `deleteMin` must find the next minimum by using H’s `findMin`. This is a bummer, but I think it’s necessary. If H’s `deleteMin` & `findMin` both run in $O(\log n)$, then ExplicitMin’s `deleteMin` will run in $O(2 \times \log n)$, which is still $O(\log n)$ as required by Exercise 3.7.

Let’s do it with a Lisp macro. My answer is in the file chapter03.lisp. Search for “Exercise 3.7”.

```lisp
(defun def-explicit-min (basename insert findMin deleteMin)
  (let ((xmaker (intern (format nil "~A-~A" 'make ,basename)))
        (xmin (intern (format nil "~A-~A" ,basename 'min)))
        (xheap (intern (format nil "~A-~A" ,basename 'heap)))
        (xinsert (intern (format nil "~A-~A" basename 'insert)))
        (xfindMin (intern (format nil "~A-~A" basename 'findMin)))
        (xdeleteMin (intern (format nil "~A-~A" basename 'deleteMin)))
        (arg-x (gensym))
        (arg-heap (gensym))
        (tmp (gensym)))
    `(progn
      (defstruct ,basename min heap)
      (defun ,xinsert (,arg-x ,arg-heap)
        (if ,arg-heap
            ,(xmaker :min (if (xleq ,arg-x ,xmin ,arg-heap)
                            ,arg-x
                            (,xmin ,arg-heap))
                  :heap (,insert ,arg-x (,xheap ,arg-heap)))
            ,(xmaker :min ,arg-x :heap (,insert ,arg-x nil))))
      (defun ,xfindMin (,arg-heap)
        (declare (type ,basename ,arg-heap))
        ,(xmin ,arg-heap))
      (defun ,xdeleteMin (,arg-heap)
        (declare (type ,basename ,arg-heap))
        (let ((,tmp (,deleteMin (,xheap ,arg-heap))))
          ,(xmaker :min (,findMin ,tmp) :heap ,tmp))
        (list ,xmaker ,xmin ,xheap ,xinsert ,xfindMin ,xdeleteMin ,arg-x ,arg-heap ,tmp)))))
```
3.8 Exercise 3.8

Prove that the maximum depth of a node in a red-black tree of size n is at most $2\lfloor \log(n+1) \rfloor$.

The proof is algebraic.

1. Let $S$ be the length of the shortest path to a leaf in the tree.
2. Then $2^S \leq N - 1$, the number of nodes in the tree.
3. $S \leq \lfloor \log(N - 1) \rfloor$
4. Let $M$ be the length of the longest path to a leaf in the tree.
5. $M \leq 2 \cdot S$, according to the rules of red-black trees.
6. $\frac{M}{2} \leq S$
7. $\frac{M}{2} \leq S \leq \lfloor \log(N - 1) \rfloor$
8. $M \leq 2 \cdot \lfloor \log N - 1 \rfloor$

3.9 Exercise 3.9

Write a function $\text{fromOrdList} : \text{Elem list} \rightarrow \text{Tree}$ that converts a sorted list with no duplicates into a red-black tree. Your function should run in $O(n)$ time.

To test my answer to this problem, I had to implement red-black trees in Lisp. That implementation is in file red-black-set.lisp. (fixme: As of 2006 January 1, it’s not finished yet.)

Wait. That’s not true. I couldn’t implement the balance function in a way that wasn’t ugly. Okasaki’s implementation in Standard ML is nice, but I’ve never seen another red-black tree implementation I liked, & I couldn’t figure it out in Lisp. So screw it. I reject red-black trees! I renounce them!

So now I turn to the next chapter.
Chapter 4

Answers to Exercises in Chapter 4

A part of Chapter 4 of [2] is an implementation of a delayed-evaluation streams package. I implemented that in Lisp in the file zstream.lisp, which is in Appendix D & is online at http://cybertigyr.com/gene/amo/zstream.lisp.

4.1 Exercise 4.1

Use the fact that force ($e$) is equivalent to $e$ to show that these two definitions of drop are equivalent.

The two implementations of drop, which I will rename to dropA & dropB, are:

fun lazy dropA (0, s) = s
  | dropA (n, $Nil) = $Nil
  | dropA (n, $Cons (x, s)) = dropA (n - 1, s)

fun lazy dropB (n, s) =
  let fun drop' (0, s) = s
      | drop' (n, $Nil) = $Nil
      | drop' (n, $Cons (x, s)) = drop' (n - 1, s)
  in drop' (n, s) end

By comparing dropA to the drop' which is within dropB, anyone can see that dropA is equivalent to drop'. So the task is to show that dropB's use of drop' is equivalent to the stand-alone function dropA.

We'll use the rule from page 33 in [2] which says that

fun lazy f p = e
is equivalent to

\[
\text{fun } f \ x = \text{case } x \ of \ p \Rightarrow \text{force } e
\]

By applying that rule to \( \text{dropB} \), we get

\[
\text{fun } \text{dropB} \ (x, y) = \text{case } (x, y) \ of \ (n, s) = \\
\text{force (let fun drop'} (0, s) = s \\
| \text{drop'} (n, \Nil) = \Nil \\
| \text{drop'} (n, \text{Cons } (x, s)) = \text{drop'} (n - 1, s) \\
in \text{drop'} (n, s) \text{ end)}
\]

Now we employ the fact that \( \text{force } (\text{s}) \) is equivalent to \( e \). So \( \text{dropB } (n, s) \) is equivalent to

\[
\text{fun } \text{drop'} (0, s) = s \\
| \text{drop'} (n, \Nil) = \Nil \\
| \text{drop'} (n, \text{Cons } (x, s)) = \text{drop'} (n - 1, s)
\]

And, like I already said, it’s obvious that \( \text{drop'} \) is equivalent to \( \text{dropA} \).

### 4.2 Exercise 4.2

Implement insertion sort on streams. Show that extracting the first \( k \) elements of \( \text{sort } xs \) takes only \( O(n \cdot k) \) time, where \( n \) is the length of \( xs \), rather than \( O(N^2) \) time, as might be expected of insertion sort.

Notice that an obvious implementation of a delayed selection sort has this same \( O(n \cdot k) \) property, but for me, an implementation of insertion sort with this property is not obvious. In fact, I wasn’t convinced it was possible until I did it.

After much thinking & experimentation, here is the \textsc{insertion-sort} function I wrote. It’s also in Appendix D.

\[
\text{(defun zinsert } (x z \text{ lessp)} \\
\text{"Insert } X \text{ into } Z \text{ in order, returning a new ZSTREAM."} \\
\text{(declare } (\text{type function lessp})) \\
\text{(let } ((z0 (force z))) \\
\text{(symbol-macrolet } ((\text{head } (\text{car } z0)) \\
(\text{tail } (\text{cdr } z0))) \\
(\text{cond } ((\text{null } z0) (zpush } x z)) \\
((\text{funcall } \text{lessp } x \text{ head}) (zpush } x z)) \\
(t (zpush head \\
;; \text{This suspension is the critical part.} \\
\text{(suspend } \text{zinsert-0} \\
\text{(force } (\text{zinsert } x \text{ tail } \text{lessp}))))))))
\]
### 4.2. EXERCISE 4.2

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>early count</th>
<th>later count</th>
<th>rel. $O(N \cdot K)$</th>
<th>rel. $O(N^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>1</td>
<td>255</td>
<td>255</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>255</td>
<td>24385</td>
<td>0.74</td>
<td>0.37</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>255</td>
<td>31360</td>
<td>0.48</td>
<td>0.48</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>511</td>
<td>511</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>511</td>
<td>97479</td>
<td>0.74</td>
<td>0.37</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>511</td>
<td>128785</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>1023</td>
<td>1023</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1024</td>
<td>512</td>
<td>1023</td>
<td>386882</td>
<td>0.74</td>
<td>0.37</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1023</td>
<td>517716</td>
<td>0.49</td>
<td>0.49</td>
</tr>
</tbody>
</table>

Table 4.1: Some performance measurements of my lazy insertion sort implementation

```lisp
(defun insertion-sort (z lessp)
  "Return a new ZTREAM which is a sorted version of Z. If you remove the first K elements from the new ZTREAM, the cost will be $O(K \cdot N)$, where N is the length of Z and also the length of the new ZTREAM."
  (declare (type function lessp))
  (let ((z0 (force z)))
    (if (null z0)
      ;; Empty ZTREAM is already sorted.
      z
      ;; Else
      (let ((head (car z0))
        (tail (insertion-sort (cdr z0) lessp)))
        (zinsert head tail lessp))))
```

To verify that my lazy insertion sort behaves as Exercise 4.2 requires, I wrote a function called `PERF-INSERTION-SORT`, which is in Appendix D. Table 4.1 shows the output of `PERF-INSERTION-SORT`.

In Table 4.1, the N column shows the length of the zstream which we’ll sort. The K column shows the number of items we’ll extract from the sorted zstream. We always use three values of K; they are 1, $\frac{N}{2}$ & N.

We generate a new zstream of N random numbers for every combination of N & K. Alternatively, we could have generated a new zstream for every N, independent of K.

The early count column shows the number of comparisons the sorting algorithm performed before we extract any items from the sorted zstream. The later count column shows the number of comparisons that have been performed after extracting K items.

The rel. $O(N \cdot K)$ column shows how much work (i.e., comparisons) my lazy insertion sort performed relative to the $O(N \cdot K)$ target value claimed by
Exercise 4.2.

The rel. $O(N^2)$ column shows how much work my lazy insertion sort performed relative to the $O(N^2)$ that a monolithic insertion sort would theoretically have done.\(^1\)

You might notice that the only suspension in my lazy insertion sort is in \texttt{zinsert}. The suspension in \texttt{zinsert} is \textit{critical} to achieve the behaviour that Exercise 4.2 requires, but a suspension around the recursive call in \texttt{INSERTION-SORT} is optional.

I generated Table 4.1 without the suspension. We can see from the table that \texttt{INSERTION-SORT} does $O(N)$ comparisons before it returns \& that extracting the first item from the new \texttt{ZTREAM} is immediate. With the suspension in \texttt{INSERTION-SORT}, \texttt{INSERTION-SORT} does no comparisons, but extracting the first element from the sorted \texttt{ZTREAM} does $O(N)$ comparisons. In other words, with the suspension, Table 4.1 would show the same amount of work except that the “early count” column would \textit{always} be zero.

Here is the \texttt{INSERTION-SORT} function with the suspension:

\begin{verbatim}
(defun insertion-sort (z lessp)
  "Return a new ZTREAM which is a sorted version of Z. If you remove the first K elements from the new ZTREAM, the cost will be $O(K \times N)$, where $N$ is the length of Z and also the length of the new ZTREAM."
  (declare (type function lessp))
  (let ((z0 (force z)))
    (if (null z0)
      ;; Empty ZTREAM is already sorted.
      z
      ;; Else
      (suspend insertion-sort-0
       (let ((head (car z0))
          (tail (insertion-sort (cdr z0) lessp)))
          (force (zinsert head tail lessp)))))))
\end{verbatim}

If you embrace suspensions fully, you’d probably use the version with the suspension, but since suspensions have an overhead cost; since it seems likely that if you sort a collection, you’ll want at least the first element from it; \& since the cost of fetching that first element is $O(N)$, I choose to forgo the suspension in \texttt{INSERTION-SORT}.

Remember that the suspension within \texttt{ZINSERT} is \textit{critical!} You can’t do without it.

---

\(^1\)When $K = N$, this final column is roughly $\frac{1}{2}$ because insertion sort on randomized lists, which is how I generated the data to sort, is close to $O\left(\frac{N^2}{2}\right)$.\)
Chapter 5

Fundamentals of Amortization

According to Okasaki, the basic way to make a persistent queue in a functional language is with two lists: the front & the rear. Insert into the queue by pushing onto the rear. Delete from the queue by removing from the front. We must sometimes reverse the rear & use it as the new front.

I implemented these queues as class “batchq” in the file amo.lisp. The source code is in Appendix C.

The worst-case cost of tail is \( O(n) \), but Okasaki proves that the amortized cost of tail is \( O(1) \).

Recall the second equation from page 40 in [2]; it says \( \sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} t_i \). If I understand amortized cost correctly (& I’m not sure I do), the (theoretical) amortized performance is an upper bound on the actual performance. Since the amortized cost for each of snoc, head, & tail is \( O(1) \), the amortized cost of any sequence of those operations will be \( O(k \cdot m) \), where \( m \) is the length of the sequence & \( k \) is some constant. Since the amortized cost is an upper bound, the actual performance should be no worse than \( O(k \cdot m) \).

Let’s do some measurements.

We’ll perform a lengthy sequence of operations randomly selected from (snoc, head, & tail). When the queue is empty, head will return nil instead of signalling an error. There is also a risk of overflowing memory with a large queue, but we’ll take that risk.

Periodically during the sequence (like every 1,000 operations), we’ll print the theoretical performance (which sill be the number of operations so far), the actual performance between the two (measured in real time), & the radio between the two. If the amortized performance, which is \( O(k \cdot m) \), is an upper bound on the actual performance, then the ratio will approximate \( k \) or it will decrease. The ratio definitely should not increase in the long run.

Here is a function which performs this performance test & prints a plain text table of the results. Table 5.1 shows the results of “(batchq-perf 10000000)”.

37
CHAPTER 5. FUNDAMENTALS OF AMORIZATION

<table>
<thead>
<tr>
<th>operations</th>
<th>amo’d cost</th>
<th>real cost</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>1,000,000.00</td>
<td>2.20</td>
<td>2.20e-6</td>
</tr>
<tr>
<td>2,000,000</td>
<td>2,000,000.00</td>
<td>4.43</td>
<td>2.21e-6</td>
</tr>
<tr>
<td>3,000,000</td>
<td>3,000,000.00</td>
<td>6.68</td>
<td>2.23e-6</td>
</tr>
<tr>
<td>4,000,000</td>
<td>4,000,000.00</td>
<td>8.89</td>
<td>2.22e-6</td>
</tr>
<tr>
<td>5,000,000</td>
<td>5,000,000.00</td>
<td>11.30</td>
<td>2.26e-6</td>
</tr>
<tr>
<td>6,000,000</td>
<td>6,000,000.00</td>
<td>13.49</td>
<td>2.25e-6</td>
</tr>
<tr>
<td>7,000,000</td>
<td>7,000,000.00</td>
<td>15.70</td>
<td>2.24e-6</td>
</tr>
<tr>
<td>8,000,000</td>
<td>8,000,000.00</td>
<td>17.84</td>
<td>2.23e-6</td>
</tr>
<tr>
<td>9,000,000</td>
<td>9,000,000.00</td>
<td>20.07</td>
<td>2.23e-6</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000.00</td>
<td>22.34</td>
<td>2.23e-6</td>
</tr>
</tbody>
</table>

Table 5.1: The results of “(batchq-perf 10000000)”

From those results, I’d say that the real performance is about $O(k \cdot n)$, where $k = 2.23 \times 10^{-6}$ second.

;; You must load "loadall.lisp" first.

(in-package "AMO")

(labels
  ((make-q ()
    "Make a big queue."
    (let ((q (make-batchq)))
      (dotimes (i 10000) (setq q (snoc q (random 100))))
      q))
  (random-op q)
  (case (random 3)
    (0 (snoc q (random 100)))
    (1 (head q) q)
    (2 (tail q))))

(amo-cost (i)
  "Return amortized cost. Because the three operations we're using all have an amortized cost of $O(1)$, the total amortized cost is just the number of operations. That's I."
  i)

(real-cost (start)
  "Return the real cost. The real cost is the amount of real time we have been running."
  (/ (- (get-internal-real-time) start)
      internal-time-units-per-second))

(defun batchq-perf (&optional (m 100000))
"M is the number of operations to perform."
(do ((q (make-q) (random-op q))
    (start (get-internal-real-time))
    (i 1 (1+ i)))
  ((> i m) q)
;; Every once in a while, print a line of
;; the table.
(when (zerop (mod i (/ m 10)))
  (format t "&9D{ ~9,2F} ,2E" i
    (list (amo-cost i) (real-cost start))
    (/ (real-cost start) (amo-cost i))))))
Chapter 6

Amortized Queues
Performance Comparison

Okasaki spends a lot of time discussing queues in [2], especially amortized queues
in Chapter 6. I like doing performance comparisons, so let’s compare the per-
formances of those queues & some others.

6.1 Interface to Queues

Each type of queue supports this interface, which I have borrowed from [2]:

\textbf{snoc} (q x) Insert item \(x\) into the queue. Return the new queue.

\textbf{head} (q) Return the item which is in the front of the queue. Return \texttt{NIL} for
an empty queue.

\textbf{tail} (q) Return a new queue which is equivalent to \(q\) with its head cut off.

\textbf{isEmpty} (q) Return \texttt{true} if & only if \(q\) is empty.

I chose to express this interface using \texttt{CLOS} because I want a single interface
with multiple implementations.

6.2 Implementations of Queues

The implementations of queues are:

\begin{itemize}
  \item naïve
  \item batch
\end{itemize}

\footnote{\textsf{"True" is any non-NIL value.}}
• amortized via the banker’s algorithm

• amortized via the physicist’s algorithm

All of the implementations are in `amo.lisp`.

6.2.1 Naïve Queues

A naïve queue is implemented as a single list. Retrieving or removing the front element is quick; it is the CAR of the list, but inserting a new element is expensive. To insert a new element, we must APPEND to the list, creating an entirely new queue each time.

I call this type of queue naïve because the implementation is obvious & far less efficient than other possibilities.

I included naïve queues as a sort of control group, a baseline, & out of curiosity.

6.2.2 Batch Queues

A batch queue is implemented with two lists. The first list is the front, & the second list is the rear. To insert into the queue, push the item onto the rear. The head of the queue is the FIRST of the front list if that list is not empty; otherwise, it’s the FIRST of the reverse of the rear list.

With batch queues, insertion is O(1). Accessing the front element is O(1) if the front list is non-empty; otherwise, it’s O(n).

The functions for “two lists” queues are in amo.lisp. All of their names begin with “listsq”. All of those functions are memoized.

Okasaki mentions that this implementation of queues is common & reasonable in functional languages.

6.2.3 Banker’s Queues

Okasaki derives this implementation of queues using the Banker’s Method of amortized analysis in Section 6.3.2 of [2].

6.2.4 Physicist’s Queues

Okasaki derives this implementation of queues using the Physicist’s Method of amortized analysis in Section 6.4.2 of [2].

6.3 The Performance Results

The performance test works like this:

1. For each size \( N = 2^k, k = 1, 2, \ldots \), then for each queue type T,

   (a) Create an empty queue of type T & place N items in it.
6.3. THE PERFORMANCE RESULTS

<table>
<thead>
<tr>
<th>N</th>
<th>NAIVEQ</th>
<th>BATCHQ</th>
<th>BANKQ</th>
<th>PHYSQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.85 \times 10^4</td>
<td>2.97 \times 10^4</td>
<td>2.92 \times 10^5</td>
<td>3.00 \times 10^5</td>
</tr>
<tr>
<td>2</td>
<td>1.97 \times 10^4</td>
<td>1.98 \times 10^4</td>
<td>2.83 \times 10^5</td>
<td>2.79 \times 10^5</td>
</tr>
<tr>
<td>4</td>
<td>1.77 \times 10^4</td>
<td>2.08 \times 10^4</td>
<td>1.71 \times 10^5</td>
<td>2.44 \times 10^5</td>
</tr>
<tr>
<td>8</td>
<td>5.78 \times 10^4</td>
<td>2.74 \times 10^4</td>
<td>2.75 \times 10^5</td>
<td>2.86 \times 10^5</td>
</tr>
<tr>
<td>16</td>
<td>6.66 \times 10^4</td>
<td>1.97 \times 10^4</td>
<td>1.51 \times 10^5</td>
<td>2.15 \times 10^5</td>
</tr>
<tr>
<td>32</td>
<td>9.32 \times 10^4</td>
<td>1.99 \times 10^5</td>
<td>2.21 \times 10^5</td>
<td>2.48 \times 10^5</td>
</tr>
<tr>
<td>64</td>
<td>4.92 \times 10^4</td>
<td>1.71 \times 10^5</td>
<td>1.45 \times 10^5</td>
<td>2.21 \times 10^5</td>
</tr>
<tr>
<td>128</td>
<td>1.63 \times 10^4</td>
<td>2.21 \times 10^5</td>
<td>1.89 \times 10^5</td>
<td>2.21 \times 10^5</td>
</tr>
<tr>
<td>256</td>
<td>2.72 \times 10^4</td>
<td>1.21 \times 10^5</td>
<td>1.66 \times 10^5</td>
<td>2.33 \times 10^5</td>
</tr>
<tr>
<td>512</td>
<td>2.10 \times 10^4</td>
<td>2.16 \times 10^5</td>
<td>2.05 \times 10^5</td>
<td>2.36 \times 10^5</td>
</tr>
<tr>
<td>1024</td>
<td>3.78 \times 10^4</td>
<td>7.48 \times 10^4</td>
<td>1.30 \times 10^5</td>
<td>1.95 \times 10^5</td>
</tr>
<tr>
<td>2048</td>
<td>2.46 \times 10^5</td>
<td>3.21 \times 10^4</td>
<td>2.36 \times 10^5</td>
<td>1.95 \times 10^5</td>
</tr>
</tbody>
</table>

Table 6.1: The performance results with memoization

(b) Do a bunch of insert/delete operations on the queue. We always do a pair of operations: One insert, then one delete. Immediately before the insert, there are \( N \) items in the queue. Immediately after the delete, there are again \( N \) items in the queue.

(c) Report the number of insert/delete operation pairs per second for queue type \( T \) of size \( N \).

Table 6.1 shows the performance results with memoization. I obtained this table by loading amo.lisp into sbcl, & then evaluating these expression:

```lisp
(in-package "AMO")
=> #<PACKAGE "AMO”>
(setq *time-test2-max-expt* 11)
=> 11
(make-memos)
=> (MAPPEND blah blah blah ...)
(time-test2 "memo.tex")
```

See Section 6.4 for a discussion of why the performance of naïve queues dropped suddenly at \( N = 2,048 \) & why I didn’t run the comparison for larger values of \( N \).

In spite of the performance analyses in Okasaki’s [2], I see little difference in the performances. I presume this is the benefit of memoization. Let’s run the performance tests without memoization to test that idea.

To run the performance comparison without memoization, I started sbcl & evaluated these expressions:

```lisp
(in-package "AMO")
=> #<PACKAGE "AMO”>
```
<table>
<thead>
<tr>
<th>N</th>
<th>NAIVEQ</th>
<th>BATCHQ</th>
<th>BANKQ</th>
<th>PHYSQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.93 × 10³</td>
<td>1.73 × 10³</td>
<td>6.36 × 10⁴</td>
<td>8.79 × 10⁴</td>
</tr>
<tr>
<td>2</td>
<td>1.84 × 10⁴</td>
<td>1.79 × 10⁴</td>
<td>5.70 × 10⁴</td>
<td>1.10 × 10⁵</td>
</tr>
<tr>
<td>4</td>
<td>1.69 × 10⁵</td>
<td>1.81 × 10⁵</td>
<td>5.07 × 10⁴</td>
<td>1.17 × 10⁵</td>
</tr>
<tr>
<td>8</td>
<td>1.49 × 10⁶</td>
<td>1.84 × 10⁶</td>
<td>4.60 × 10⁴</td>
<td>1.23 × 10⁵</td>
</tr>
<tr>
<td>16</td>
<td>1.19 × 10⁷</td>
<td>1.91 × 10⁷</td>
<td>4.35 × 10⁴</td>
<td>1.23 × 10⁵</td>
</tr>
<tr>
<td>32</td>
<td>8.90 × 10⁷</td>
<td>1.94 × 10⁷</td>
<td>4.31 × 10⁴</td>
<td>1.25 × 10⁵</td>
</tr>
<tr>
<td>64</td>
<td>6.11 × 10⁸</td>
<td>2.06 × 10⁸</td>
<td>4.22 × 10⁴</td>
<td>1.26 × 10⁵</td>
</tr>
<tr>
<td>128</td>
<td>3.61 × 10⁹</td>
<td>2.03 × 10⁹</td>
<td>4.18 × 10⁴</td>
<td>1.25 × 10⁵</td>
</tr>
<tr>
<td>256</td>
<td>1.95 × 10⁹</td>
<td>2.08 × 10⁹</td>
<td>4.13 × 10⁴</td>
<td>1.26 × 10⁵</td>
</tr>
<tr>
<td>512</td>
<td>1.01 × 10¹⁰</td>
<td>2.07 × 10⁹</td>
<td>4.05 × 10⁴</td>
<td>1.24 × 10⁵</td>
</tr>
<tr>
<td>1024</td>
<td>5.05 × 10⁹</td>
<td>2.05 × 10⁹</td>
<td>4.04 × 10⁴</td>
<td>1.27 × 10⁵</td>
</tr>
<tr>
<td>2048</td>
<td>2.66 × 10⁹</td>
<td>2.09 × 10⁹</td>
<td>3.74 × 10⁴</td>
<td>1.27 × 10⁵</td>
</tr>
<tr>
<td>4096</td>
<td>1.30 × 10⁹</td>
<td>1.99 × 10⁹</td>
<td>2.95 × 10⁴</td>
<td>1.16 × 10⁵</td>
</tr>
<tr>
<td>8192</td>
<td>6.43 × 10⁹</td>
<td>1.99 × 10⁹</td>
<td>8.65 × 10⁴</td>
<td>1.15 × 10⁵</td>
</tr>
<tr>
<td>16384</td>
<td>2.96 × 10¹⁰</td>
<td>2.07 × 10⁹</td>
<td>5.53 × 10⁴</td>
<td>1.28 × 10⁵</td>
</tr>
<tr>
<td>32768</td>
<td>1.36 × 10¹⁰</td>
<td>1.89 × 10⁹</td>
<td>1.02 × 10⁹</td>
<td>1.10 × 10⁵</td>
</tr>
<tr>
<td>65536</td>
<td>5.89 × 10¹⁰</td>
<td>1.74 × 10⁹</td>
<td>3.40 × 10⁴</td>
<td>1.25 × 10⁵</td>
</tr>
</tbody>
</table>

Table 6.2: The performance results without memoization

;;; I did not do (setq *time-test2-max-expt* 12)
;;; I did not do (make-memos)
(time-test2 "nomemo.tex")

The results are in Table 6.2.

### 6.4 Observations about Memoization & Memory Use

I wanted to run the test for larger queues, but the memoized queues exhausted memory with anything larger. It turned out that the naïve queues were the memory-consuming culprit because each memoized item contains a fresh queue; there is no re-use between the queues. If each queue element requires 1 word for the CONS’s CAR, 1 word for the CONS’s CDR, & one word for the payload (& it probably requires more, for type information at least), then each list element requires 3 words, & a list of \(N\) elements requires \(3N\) words. If a word is 4 octets, then a list of 4,096 elements requires \(3 \times 4096 \times 4 = 49152\) bytes, & the memoization cache stores all lists from length 1 to length 4,096. So to memoize lists of length 4,096, we memoize \(\sum_{N=1}^{4096} N \rightarrow 8,390,656\) elements. If each element is 3 words of 4 octets each... Yeah, that’s why it used too much memory.
6.5 Analysis

I’m kind of stunned. Let’s look at the memoized results first (Table 6.2).

If you accept the memory size explanation for the abysmal performance of naïve queues, then the performances were roughly constant as $N$ increased. Batch queues had a few inefficient moments, but for the most part, batch, banker’s, & physicist queues had constant & equivalent performances. It’s possible that batch queues suffer from a minor version of the memory use problem that afflicts naïve queues. My main conclusion from the memoized results is that, if you have enough memory to cache all results, the implementation doesn’t make much difference if it makes any difference at all.\footnote{For what it’s worth, the same memoized test on clisp on Microsloth Winders showed even closer & more consistent performances.}

In the non-memoized test, naïve queues showed $O(N)$ performance, batch queues showed $O(\log N)$, banker’s queues showed roughly $O(N)$, & physicist’s queues showed roughly $O(\log \log N)$\footnote{$\log (\log N)$?! Am I sure about this? fixme: What is the predicted performance?} performance. Of those, the fastest was batch queues, then physicist’s queues, then naïve queues, then banker’s queues.

Banker’s queues were even slower than naïve queues. How could this be? Some explanations include:

- When I translated Okasaki’s banker’s queues from Standard ML to Common Lisp by hand, I made a mistake & broke the algorithm.

- Banker’s queues had the best performance in the memoized tests. If the difference between memoized banker’s queues & the second-fastest memoized queues was not noise, then it’s possible that banker’s queues are faster in the presence of memoization by reducing the number of distinct queues, thereby keeping the cache’s hash table size small & it’s lookup performance high.

- The test I used shows banker’s queues in a bad case. If so, this was unintentional on my part.

- It’s okay if the overall performance of banker’s queues is worse than a bad algorithm’s performance because amortized cost isn’t the same as worst-case cost. fixme: I must re-study the idea of amortized analysis to see if this is possible.

Table 6.3 shows the Big-O performance I measured & the theoretical performance from [2]. (The theoretical performance for naïve queues is my own S.W.A.G., not from [2].) The theory column is the maximum of the work for $\text{snoc}$ & the work for $\text{tail}$ because my performance test counted pairs of $\text{snoc}/\text{tail}$ operations.

The costs measured by my performance tests aren’t exactly the same as the theoretical cost. Explanations for this discrepancy include:
CHAPTER 6. AMORTIZED QUEUES PERFORMANCE COMPARISON

<table>
<thead>
<tr>
<th>implementation</th>
<th>theory</th>
<th>measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>batch</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>banker’s</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>physicist’s</td>
<td>log N</td>
<td>log log N</td>
</tr>
</tbody>
</table>

Table 6.3: The Big-O performance I measured & the theoretical performance from [2]

- My performance tests counted iterations over *clock time* whereas theoretical performance analysis depends on *steps*.
- The performance tests are unfair to some of the queue implementations, or they measured the wrong thing.
- I misunderstood the meaning of amortized cost & analysis.
- I implemented the queues incorrectly.

The actual speed of batch queues (1.74 × 10^5 SNOCTAIL pairs per second at 2^{16} elements) in the non-memoized run, which was the fastest among the queue implementations, supports Okasaki’s claim on page 44 that “these queues cannot be beat for applications that do not require persistence & for which amortized bounds are acceptable”.
Chapter 7

Other Observations

7.1 NIL & object-oriented programming

If you use NIL as an empty collection, then you can’t use a CLOS interface for multiple implementations of that interface. Here’s an example that shows why:

1. Think about an interface for heaps (which Okasaki’s Chapter 3 discusses in depth). It might have INSERT, FINDMIN, & DELETEMIN in its interface.

2. Assume we want NIL to be an empty heap.

3. Imagine that you implement one kind of heap. With CLOS, it’s easy to specialize methods on types & on values, so you can implement the methods to use NIL for empty heaps. It works fine.

4. Now imagine that you want to implement another kind of heap using the same interface. Your new FINDMIN & DELETEMIN methods are probably fine because they assume the heap is not empty, but what about INSERT? When you insert into a non-empty heap, you can specialize an INSERT method on the type, but INSERT is already specialized for NIL on the previous kind of heap.

7.2 Nil for empty & special cases

It seems that using NIL for empty collections makes it easy to use the collection in general & easy to implement the collection from within, but collections which wrap other collections must treat NIL as a special case.

This idea is from Exercise 3.7 (Section 3.7).
7.3 Memoization & memory

If you memoize, memoize everything. Otherwise, you eat too much memory because the low-level functions, such as CONS & APPEND, are not memoized.

7.4 Lisp & suspensions

You can implement suspensions in Lisp, but I would not want to design amortized algorithms using suspensions in Lisp. With suspensions in Lisp, the language seems to get in the way.

So where suspensions are concerned, Lisp is to a language with suspensions as an Algol-descendant language is to Lisp where lists (& code-as-data & macros & higher-order functions) are concerned. In other words, sure, you can implement suspensions in Lisp, but the syntax gets in the way of the algorithms, just like how you can implement lists (higher-order functions) in C++, but then the syntax gets in the way of the algorithms. So if you are writing an algorithm that uses higher-order functions, you are better off writing it in Lisp & then translating to C++. If you are writing an amortized algorithm, maybe you are better off doing it in Okasaki’s Standard ML with Suspensions, then translating to Lisp.
Chapter 8

The Source Code

- amo.lisp
- auxfns.lisp
- binomial-heap.lisp
- bst.lisp
- chapter02.lisp
- chapter03.lisp
- lazy.lisp
- loadall.lisp
- ordered.lisp
- red-black-set.lisp
- utils.lisp
- zstream.lisp

It requires the memoization functions from Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp [1], by Peter Norvig. They are available elsewhere, or you can use local copy at CyberTiggy.COM.
Appendix A

Other File Formats

- This document is available in multi-file HTML format at http://cybertiggyr.com/gene/amo/.

- This document is available in Pointless Document Format (PDF) at http://cybertiggyr.com/gene/amo/amo.pdf.
Appendix B

Some comments about suspend & force

Here is part of an e-mail message I wrote to a friend who asked about suspend & force in Lisp.

Date: Sunday, 2006 March 12
Suspend & Force aren’t part of Common Lisp. They are techniques that you can implement in Lisp. They are well-known; for example, they are discussed in Norvig’s “Paradigms of AI Programming: Case Studies in Common Lisp” [1].

My implementation of them contains some “counters” to help estimate performance by letting you track how many suspensions were created & how many were evaluated.

SUSPEND takes a chunk of code & gives you a function which returns two values:

1. The value from evaluating the suspended chunk of code, &
2. Another function which is the suspension of the next chunk of code.

FORCE just evaluates the chunk of code. In Lisp, you could just FUCNCALL the suspension, but I (& other programmers) like to have a function called FORCE to make it easier to see that the function you are funcalling is supposed to be a suspension. (Basically, a small aide to self-documentation.)

Suspensions are an important technique when you write algorithms which are lazy (i.e., which amortize their costs). There seem to be two types of benefits from suspensions, depending on how you use them or how you look at them:

1. Traditional (i.e., “monolithic”) algorithms can be implemented with suspensions, usually with trivial effort, & the resulting algorithm will do less work if you do not need all of its results. 53
2. Algorithms which must be incomprehensively complicated in monolithic form to be efficient can be refreshingly simple if you implement them with suspensions.

In the “Purely Functional Data Structures” book [2], Okasaki spends about 90 percent of the space in discussion of techniques which use suspensions & in analysis of amortized algorithms. It’s really cool stuff!

Interestingly, though Lisp does not have suspensions, it’s easy to implement them in Lisp. I’ve heard that Scheme does have suspensions. Standard ML (which Okasaki uses in the book), does not, but he adds it to Standard ML with a simple extension of the syntax.

I notice that it is easier to analyze the amortized algorithms in Okasaki’s extended Standard ML than in Lisp with suspend & force. Does this suggest that fully functional languages (such as Standard ML) are to Lisp as Lisp is to Algol-descendant imperative languages (such as Java, C++, C#, C, Pascal)???
Appendix C

Source Code (amo.lisp)

This source code is also online at http://cybertiggyr.com/gene/amo/amo.lisp.

;;; -*- Mode: Lisp -*-
;;; Copyright (c) 2005 Gene Michael Stover. All rights reserved.
;;; This program is free software; you can redistribute it and/or modify
;;; it under the terms of the GNU Lesser General Public License as
;;; published by the Free Software Foundation; either version 2 of the
;;; License, or (at your option) any later version.
;;; This program is distributed in the hope that it will be useful,
;;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
;;; GNU Lesser General Public License for more details.
;;; You should have received a copy of the GNU Lesser General Public
;;; License along with this program; if not, write to the Free Software
;;; Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301
;;; USA
;;;
;; (defpackage "AMO"
  (:use "COMMON-LISP")
)(in-package "AMO")

;;;
;; Wrappers around some Common Lisp functions so I can memoize
;; these wrappers, not the Common Lisp functions.
;;
(defun mappend (1st x) (append 1st x))
(defun mcons (x y) (cons x y))
(defun mfirst (cons) (first cons))
(defun mlist (&rest rest) (apply #'list rest))
(defun mrest (cons) (rest cons))
(defun mreverse (lst) (reverse lst))

(defun make-memos ()
  (mapc #'common-lisp-user::memoize
    '(mappend mcons mfirst mlist mrest mreverse
      xsnoc xtail
      make-naiveq
      make-batchq batchq-check
      make-bankq bankq-check
      make-physq physq-checkw physq-check
      lazy:suspend lazy:force)))

;;; INTERFACE TO QUEUES
;;;
;;; Every queue supports functions to do these things:
;;;
;;; make () : Return an empty, queue.
;;; insert (x q) : Insert item X into the queue. Return the
;;; new queue.
;;; front (q) : Return the item which is in the front of the queue.
;;; Return NIL for an empty queue.
;;; delete (q) : Delete the item which is in the front of the queue.
;;; Return the new queue.
;;; isEmpty? (q) : Return NIL if the queue is not empty. Return
;;; true (i.e., non-NIL) if the queue is empty.

(defun isEmpty (q)
  (:documentation "Return true if & only if Q is empty."))

(defun snoc (q x)
  (:documentation "Append item X to the queue. Return a new queue."))

(defun head (q)
  (:documentation "Return the first item in the queue. If the queue is
empty, you get NIL.")

(defun tail (q)
  (:documentation "Return a new queue which is like Q with its head
cut off. If Q is empty, you get an empty queue.")

(defun xtail (q) (tail q))
(defun xsnoc (q x) (snoc q x))

;;; NAIVE QUEUE
;;;
;;; This queue is a single list. To insert, we APPEND. To DELETE,
;;; we return REST. The FRONT is the FIRST.
;;;
;;; (defclass naiveq ()
;;;   ((lst :initarg :lst :accessor naiveq-lst)))

(defun make-naiveq (&optional (lst nil))
  (make-instance 'naiveq :lst lst))

(defun isEmpty ((q naiveq))
  (endp (naiveq-lst q)))

(defun snoc ((q naiveq) x)
  (make-naiveq (mappend (naiveq-lst q) (mlist x))))

(defun head ((q naiveq))
  (mfirst (naiveq-lst q)))

(defun tail ((q naiveq))
  (make-naiveq (mrest (naiveq-lst q))))

;;;
;;; LISTS QUEUE
;;;
;;; According to Okasaki in "Purely Functional Data Structures",
;;; the basic way to make a persistent queue in a functional
;;; language is with two lists: the front & the rear. Insert into
;;; the queue by pushing onto the rear. Delete from the queue by
;;; removing from the front. We must sometimes reverse the rear &
;;; use it as the new front.
;;;
;;; (defclass batchq ()
;;;   ((front :initarg :front :accessor batchq-front)
;;;    (rear :initarg :rear :accessor batchq-rear))))

(defun make-batchq (&optional (front nil) (rear nil))
  (make-instance 'batchq :front front :rear rear))

(defun batchq-check (q)
  (if (endp (batchq-front q))
    (make-batchq (mreverse (batchq-rear q)) ()
                     q))
  (defmethod snoc ((q batchq) x)
APPENDIX C. SOURCE CODE (AMO.LISP)

(make-batchq (batchq-front q) (mcons x (batchq-rear q)))

(defun head ((q batchq))
  (mfirst (batchq-front (batchq-check q))))

(defun tail ((q batchq))
  (setq q (batchq-check q))
  (make-batchq (mrest (batchq-front q)) (batchq-rear q)))

;;; BANKER'S QUEUES : Amortized queues using the banker’s method
;;; from Okasaki, page 65
;;; A banker's queue is a list of 4 things.
;;; FIRST is the length of the FRONT.
;;; SECOND is a suspension for the FRONT. Evaluate it to get the front.
;;; THIRD is the length of the REAR.
;;; FOURTH is the suspension for the REAR.

(defun make-bankq (&optional (lenf 0) (f (ztream:make-ztream))
  (lenr 0) (r (ztream:make-ztream)))
  (make-instance 'bankq :lenf lenf :f f :lenr lenr :r r))

(defun bankq-check (lenf f lenr r)
  (if (< lenf lenr)
      (make-bankq lenf f lenr r)
    ;; Else, the rear is too big, so we reverse it & append
    ;; to the front -- lazily.
    (make-bankq (+ lenf lenr) (ztream:zappend f (ztream:zreverse r))
      0 (ztream:make-ztream))))

(defun isEmpty ((q bankq))
  (and (zerop (bankq-lenf q))
    (zerop (bankq-lenr q))))

(defun snoc ((q bankq) x)
  (bankq-check (bankq-lenf q) (bankq-f q)
    (1+ (bankq-lenr q)) (ztream:zpush x (bankq-r q))))

(defun head ((q bankq))
  (ztream:head (bankq-f q)))

(defun tail ((q bankq)))
(if (ztream:isEmpty (bankq-f q))
  ;; If you delete the front of an empty q,
  ;; you get an empty q. The q we have is
  ;; already empty, so we'll return that.
  q
  (bankq-check (1- (bankq-lenf q)) (ztream:drop 1 (bankq-f q))
    (bankq-lenr q) (bankq-r q))))

;;;
;;; PHYSICIST'S QUEUE
;;; from Okisawa, page 73
;;;

(defclass physq ()
  ((w :initarg :w :accessor physq-w)
   (lenf :initarg :lenf :accessor physq-lenf)
   (f :initarg :f :accessor physq-f)
   (lenr :initarg :lenr :accessor physq-lenr)
   (r :initarg :r :accessor physq-r)))

(defun make-physq (&optional (w nil) (lenf 0) (f (lazy:suspend physq-make-0 nil))
  (lenr 0) (r nil))
  (make-instance 'physq :w w :lenf lenf :f f :lenr lenr :r r))

(defun physq-checkw (w lenf f lenr r)
  (make-physq (if (endp w) (lazy:force f) w)
    lenf f
    lenr r))

(defun physq-check (w lenf f lenr r)
  (physq-checkw w lenf f lenr r)
  ;; Else
  (let ((f-prime (lazy:force f)))
    (physq-checkw f-prime (+ lenf lenr)
      (lazy:suspend phys-normalize-0
        (mappend f-prime (mreverse r)))
      0 nil)))))

(defun snoc ((q physq) x)
  (physq-check (physq-w q)
    (physq-lenf q) (physq-f q)
    (1+ (physq-lenr q)) (mcons x (physq-r q))))

(defun head ((q physq))
  (mfirst (physq-w q)))

(defun tail ((q physq))
  (physq-check (mrest (physq-w q)))
(1- (physq-lenf q)) (let ((f-prime (lazy:force (physq-f q)))
  (lazy:suspend physq-delete-0
    (mrest f-prime)))
  (physq-lenr q) (physq-r q)))

;;;; TIME TESTS
;;;
;;; Functions & data definitions for timing things.
;;;;

(defun time-test (fn &key (mincount 3) (minseconds 3))
  (declare (type function fn))
  (do ((start-time (get-internal-real-time))
       (count 0 (1+ count)))
      ((and (<= mincount count)
            (<= (* minseconds internal-time-units-per-second)
                (- (get-internal-real-time) start-time)))
    (let* ((clocks (- (get-internal-real-time) start-time))
           (seconds (/ clocks internal-time-units-per-second))
           (rate (/ count seconds)))
      ;; Return a new TIME-RESULT datum
      (make-time-result :rate rate :clocks clocks :seconds seconds
                        :count count)))
  (funcall fn)))

(defun fill-queue (n q)
  (dotimes (i n) (setq q (xsnoc q 42)))
  q)

(defun do-snoc-tail (q)
  (xtail (xsnoc q 42)))

(defun time-test0 (n make)
  "Run a time test on the technique using a queue of size N."
  (declare (type integer n) (type symbol make))
  (let ((q (funcall make)))
    (print (list 'time-test0 n (class-name (class-of q))))
    (force-output)
    (setq q (fill-queue n q))
    (list n
      (class-name (class-of q))
      (time-test #'(lambda () (setq q (do-snoc-tail q)))))))
(defun time-test1 (n)
  "Run TIME-TEST0 on each technique."
  (mapcar #'(lambda (make) (time-test0 n make))
    ;; Must use the symbols for the functions unless we
    ;; memoized them before now, but I don't want to do
    ;; that because I want to do non-memoized test, then
    ;; a memoized test. So we use symbols.
    '(make-naiveq make-batchq make-bankq make-physq)))

(defvar *time-test2-max-expt* 16)

(defun sci (n)
  "Return a string which encodes the number N in scientific
  notation for LaTeX."
  (let* ((x (floor (log n 10.0)))
         (m (/ n (expt 10.0 x)))
         (format nil "$~,2F \times 10^{~D}$" m x))
    (format nil "\$~,2F \times 10^{~D}$" m x))

(defun time-test2-headers (strm)
  (let ((columns (mapcar #'second (time-test1 10))))
    (format strm "&\begin{tabular}{|r| ~A|} \hline"
      (mapcar (constantly #\r) columns))
    (format strm "{\bf N} ~{ & {\bf ~A} } \\ \hline" columns)))

(defun time-test2 (pn)
  (with-open-file (strm pn :direction :output :if-exists :rename-and-delete)
    (time-test2-headers strm)
    (dolist (n (loop for i from 0 to *time-test2-max-expt* collect (expt 2 i)))
      (format strm "&^\text{D}n\)  \\ \hline"
        (time-test1 n))
      (format strm "\begin{tabular} &\text{\textbf{A}}
        (time-test1 n))
      (format strm "\text{\textbf{A}} &\text{\end{tabular}}")
    (truename pn)))

;;; --- end of file ---
Appendix D

Source Code (ztream.lisp)

This source code is also online at http://cybertiggyr.com/gene/amo/ztream.lisp.

;;; -*- Mode: Lisp -*-
;;; Copyright (c) 2005 Gene Michael Stover. All rights reserved.
;;; This program is free software; you can redistribute it and/or modify
;;; it under the terms of the GNU Lesser General Public License as
;;; published by the Free Software Foundation; either version 2 of the
;;; License, or (at your option) any later version.
;;; This program is distributed in the hope that it will be useful,
;;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
;;; GNU Lesser General Public License for more details.
;;; You should have received a copy of the GNU Lesser General Public
;;; License along with this program; if not, write to the Free Software
;;; Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301
;;; USA

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)

(defun drop)
(export 'head)
(export 'isEmpty)
(export 'make-ztream)
(export 'tail)
(export 'take)
(export 'zappend)
(export 'zpush)
(export 'zreverse)

(defun make-ztream ()
  "Return an empty ZTREAM."
  (suspend make-ztream nil))

;;
;; Here are three accessors which are not in Okasaki. I use them
;; here for convenience, especially convenience in testing.
;;
(defun ztream-p (x)
  "Return true if & only if X is a ZTREAM. Actually, it's just an
  approximation because not all functions are ZTREAMs."
  (functionp x))
(defun isEmpty (ztream)
  "Return true if & only if the ZTREAM is empty."
  (null (force ztream)))
(defun head (ztream) "Return first item from the ZTREAM"
  (car (force ztream)))
(defun tail (ztream)
  "Return the rest of the ZTREAM."
  (cdr (force ztream)))
(defun zpush (x z)
  (assert (ztream-p z))
  (suspend zpush-0
    (cons x z)))

(defun zappend (a b)
  "Return a new ZTREAM which is the concatenation of A & B."
  (assert (ztream-p a))
  (assert (ztream-p b))
  (if (isEmpty a)
      b
    (let ((afirst (head a))
          (atail (tail a)))
      (suspend zappend-2
        (cons afirst (zappend atail b)))))

(defun take (n z)
  (declare (type integer n))
  (assert (ztream-p z))
  (cond ((zerop n) (suspend take-0 nil))
        ((isEmpty z) (suspend take-1 nil))
        (t (let ((tbody (tail z))
                   (nt (force (take (1- n) z))))
             (assert (ztream-p nt))
             (assert (ztream-p tbody))
             (suspend take-2
               (cons (force (head nt))
                     (take n tbody)))))))
(t
  ;; We evaluate Z now to reduce stack depth. Clisp seems
  ;; to have a very limited stack. This might defeat alter
  ;; of the performance characteristics of ZTREAMs from
  ;; the implementation given in Okasaki.
  (let ((zhead (head z))
        (ztail (tail z)))
    (suspend take-2
      (cons zhead (take (1- n) ztail))))))

(defun drop (n z)
  (declare (type integer n))
  (assert (ztream-p z))
  (cond ((zerop n) z)
        ((isEmpty z) (suspend drop-0 nil))
        (t (drop (1- n) (tail z))))))

(defun zreverse (z)
  (labels
    ((zreverse2 (z2 r)
      (if (isEmpty z2)
          r
          (let ((z2head (head z2))
                 (z2tail (tail z2)))
            (zreverse2 z2tail
               (suspend zreverse-0
                 (cons z2head r)))))))
    (zreverse2 z (suspend zreverse-1 nil))))

;;;
;;; For the insertion sort to perform minimal work each
;;; time you extract an element, it is CRITICAL that
;;; ZINSERT do minimal work. (It’s, like, "duh", but
;;; it took a long time to remember to implement
;;; ZINSERT that way, & until I did, I had an insertion
;;; sort which did all of its work the first time you
;;; evaluated the sorted ZTREAM.
;;;
(defun zinsert (x z lessp)
  "Insert X into Z in order, returning a new ZTREAM."
  (declare (type function lessp))
  (let ((z0 (force z)))
    (symbol-macrolet ((head (car z0))
                      (tail (cdr z0)))
      (cond ((null z0) (zppush x z))
            ((funcall lessp x head) (zppush x z))
            (t (zppush head
                ;; This suspension is the critical part.
                (suspend zinsert-0
                  (force (zinsert x tail lessp))))))))
(defun insertion-sort (z lessp)
"Return a new ZTREAM which is a sorted version of Z. If you remove the first K elements from the new ZTREAM, the cost will be O(K * N), where N is the length of Z and also the length of the new ZTREAM."
(declare (type function lessp))
(let ((z0 (force z)))
  (if (null z0)
      ;; Empty ZTREAM is already sorted.
      z
      ;; Else
      (let ((head (car z0))
             (tail (insertion-sort (cdr z0) lessp)))
        (zinsert head tail lessp))))

;; TESTS
;;
(defun test0000 ()
  "Null test. Always succeeds."
  'test0000)

(defun test0001 ()
  "Verify that MAKE-ZTREAM does not crash."
  (make-ztream)
  'test0001)

(defun test0002 ()
  "Verify that MAKE-ZTREAM returns an empty ZTREAM."
  (isEmpty (make-ztream)))
(defun test0003 ()
  "Verify that the HEAD of an empty ZTREAM is NIL."
  (null (head (make-ztream))))

(defun test0004 ()
  "Verify that the TAIL of an empty ZTREAM is NIL."
  (null (tail (make-ztream))))

(defun test0010 ()
  "Verify that the HEAD of a ZTREAM on which we just ZPUSHed a thing is the thing."
  (eq (head (zpush 'thing (make-ztream))) 'thing))

(defun test0015 ()
  "Verify that the TAIL of a ZTREAM containing one thing is empty."
  (isEmpty (tail (zpush 'thing (make-ztream)))))

(defun test0100 ()
  "Test that if we have a ZTREAM of 1 thing, then we TAKE 1 thing from it, we get a ZTREAM whose HEAD is the thing."
  (eq (head (take 1 (zpush 'thing (make-ztream)))) 'thing))

(defun test0110 ()
  "Test that if we have a ZTREAM of 1 thing, then we DROP 1 thing from it, we get a ZTREAM that is empty."
  (isEmpty (drop 1 (zpush 'thing (make-ztream)))))

(defun test0200 ()
  "Test that we can REVERSE an empty ZTREAM & get an empty ZTREAM in return."
  (isEmpty (zreverse (make-ztream))))

(defun test0210 ()
  "Test that we can REVERSE a ZTREAM of one item, & the new ZTREAM's HEAD is the item."
  (eq (head (zreverse (zpush 'thing (make-ztream)))) 'thing))

(deftest test0230 ()
  "Test ZINSERT by inserting an item into an empty ZTREAM."
  (let ((z (zinsert 1 (make-ztream) #'<)))
    (and (eql (head z) 1)
         (eql (head (zinsert 1 (zpush 'thing (make-ztream)))) #'<)
         (isEmpty (tail z)))))

(deftest test0232 ()
  "Test ZINSERT by inserting an item into a ZTREAM containing
one item. The new item will go in front of the ZTREAM.

(let ((z (zinsert 1 (zpush 2 (make-ztream)) #'<)))
  (and (eql (head z) 1)
       (eql (head (tail z)) 2)
       (isEmpty (tail (tail z))))))

(deftest test0234 ()
  "Test ZINSERT by inserting an item into a ZTREAM containing one item. The new item will go behind the item that's already in the ZTREAM."
  (let ((z (zinsert 3 (zpush 2 (make-ztream)) #'<)))
    (and (eql (head z) 2)
         (eql (head (tail z)) 3)
         (isEmpty (tail (tail z))))))

(deftest test0236 ()
  "Test ZINSERT by inserting an item into a ZTREAM containing two items. The new item will be the second item of the new ZTREAM."
  (let (z)
    ;; I broke the initialization steps into separate lines
    ;; for readability.
    ;; Start with a two-item ZTREAM. Its elements will be (2 4).
    (setq z (zpush 2 (zpush 4 (make-ztream))))
    ;; Insert 3. We should get (2 3 4) in a ZTREAM.
    (setq z (zinsert 3 z #'<))
    ;; Check the results.
    (and (eql (head z) 2)
         (eql (head (tail z)) 3)
         (eql (head (tail (tail z))) 4)
         (isEmpty (tail (tail (tail z))))))

(deftest test0238 ()
  "Test ZINSERT by inserting an item into a ZTREAM containing two items. The new item will be the last item of the new ZTREAM."
  (let (z)
    ;; I broke the initialization steps into separate lines
    ;; for readability.
    ;; Start with a two-item ZTREAM. Its elements will be (2 3).
    (setq z (zpush 2 (zpush 3 (make-ztream))))
    ;; Insert 4. We should get (2 3 4) in a ZTREAM.
    (setq z (zinsert 4 z #'<))
    ;; Check the results.
    (and (eql (head z) 2)
         (eql (head (tail z)) 3)
         (eql (head (tail (tail z))) 4)
         (isEmpty (tail (tail (tail z))))))
(deftest test0250 ()
  "Test that INSERTION-SORT properly sorts an empty ZTREAM. A properly sorted empty ZTREAM is an empty ZTREAM."
  (isEmpty (insertion-sort (make-ztream) #'<)))

(deftest test0252 ()
  "Test that INSERTION-SORT properly sorts a ZTREAM which contains one element. A properly ZTREAM of length 1 is another ZTREAM of length 1 & containing that same element."
  (let* ((z0 (zpush 1 (make-ztream)))
         (z1 (insertion-sort z0 #'<)))
      (and (eql (head z1) 1)
           (isEmpty (tail z1)))))

(deftest test0254 ()
  "Test that INSERTION-SORT properly sorts a ZTREAM which contains two element, 1 & 2. A properly sorted version of that ZTREAM is 1, then 2."
  (let ((z (insertion-sort (zpush 1 (zpush 2 (make-ztream))) #'<)))
      (and (eql (head z) 1)
           (eql (head (tail z)) 2)
           (isEmpty (tail (tail z))))))

(deftest test0256 ()
  "Like TEST0254 except that the initial order of the elements is 2, then 1. The proper sorted order is 1, then 2."
  (let ((z (insertion-sort (zpush 2 (zpush 1 (make-ztream))) #'<)))
      (and (eql (head z) 1)
           (eql (head (tail z)) 2)
           (isEmpty (tail (tail z))))))

(deftest test0258 ()
  "Like TEST0256 except that the ZTREAM contains 3 elements."
  (let ((z (insertion-sort (zpush 3 (zpush 2 (zpush 1 (make-ztream)))) #'<)))
      (and (eql (head z) 1)
           (eql (head (tail z)) 2)
           (eql (head (tail (tail z))) 3)
           (isEmpty (tail (tail (tail z))))))

(deftest test0260 (&optional (n 10))
  "Create a ZTREAM of 10 randomly selected numbers, sort it, & verify that it is correctly sorted."
  (let ((z (make-ztream)))
      ;; Insert 10 random numbers into Z.
      (dotimes (i n) (setq z (zpush (random 100) z)))
      ;; Sort it.
      (setq z (insertion-sort z #'<))
      ;; Verify that it's sorted)
(do ((i 0 (1+ i))
     (j -100 (head z0))
     (z0 z (tail z0)))
   ((or (>= i n)
        (isEmpty z0)
        (< (head z0) j))
    ;; We have success if we inspected N items, &
    ;; there are no more items.
    (and (= i n) (isEmpty z0))))))

(deftest test0262 ()
 "Uses TEST0260 but for a ZTREAM of length 17."
 (test0260 17))

(defun perf-insertion-sort ()
 "Print a table of list lengths & number of comparisons
 for INSERTION-SORT."
 (format t "& N K early count later count")
 (format t "&----- ---- --------- ----------")
 (loop for n in '(256 512 1024) do
   (loop for k in (list 1 (floor (/ n 2)) n) do
     (format t "~4D ~4D" n k)
     (let ((z (make-ztream)) (cmp-count 0))
       ;; Fill the ZTREAM
       (dotimes (j n) (setq z (zpush (random 100) z)))
       ;; Sort the ZTREAM, counting the comparisons.
       (setq z (insertion-sort z #'(lambda (a b)
                                     (incf cmp-count)
                                     (< a b)))))
       (format t " ~11D cmp-count")
       ;; Remove K items from the ZTREAM. Due to delayed
       ;; evaluation, this may update the comparison counter.
       (dotimes (i k) (setq z (tail z)))
       (format t " ~11D cmp-count")
       ;; The ratio of the actual count & the theoretical
       ;; count.
       (format t " ~7,2F (/ cmp-count (* n k))")
       ;; The ratio of the actual count & the worse case
       ;; count.
       (format t " ~7,2F (/ cmp-count (* n n))")
     )))

;;;; --- end of file ---
Bibliography
