Brandeis University

Computer Science (1)21b (Spring Term, 2020)

Structure and Interpretation of Computer Programs

Problem Set 2: Binomial Queues

Due Monday, February 10

Reading Assignment: Chapter 2, Sections 2.1, 2.2.

1 Homework Exercises

Exercise 2.4: Representing pairs as procedures.

Exercise 2.22: Bad implementations of `square-list`.

Exercise 2.25: `car` and `cdr` finger-exercises.

Exercise 2.26: comparing `cons`, `list`, and `append`.

Exercise 2.27: `deep-reverse`.

Exercise 2.28: `fringe`.

2 Coding Assignment

In this problem set, you are asked to implement a *priority queue*, a data structure that stores a finite set of non-negative integers, supporting the following operations:

`(make-queue)` Create an empty queue.

`(insert x Q)` Insert integer x into the queue.

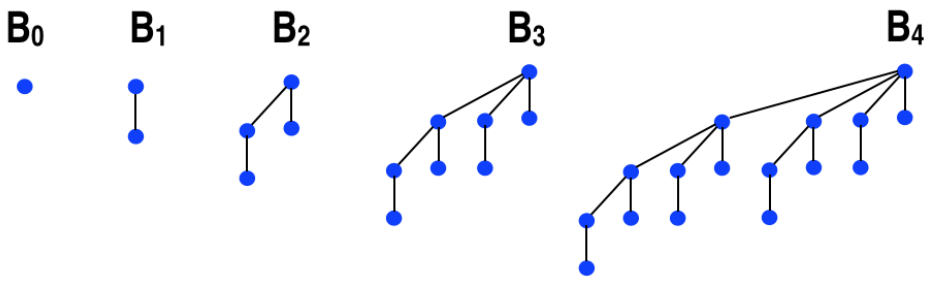
`(max-queue Q)` Return the largest element in the queue.

`(remove-max Q)` Remove the largest element in the queue.

`(merge Q1 Q2)` Merge two binomial queues into a single binomial queue.

This queue is called a priority queue because it can implement a queue of users who have a numeric priority; the user with the highest number gets priority and is served first, then removed from the queue. It's easy enough to implement these operations, using a list of integers, but the operations can take up to $O(n)$ (linear) time, where n is the number of integers in the queue. We want to implement all such operations in $O(\log n)$ time, where what we're really thinking of is the base 2 logarithm (even though logarithms with different bases vary only by a constant).

A *binomial queue* is a particular data structure that implements these priority queue operations in logarithmic time. It is made up of a forest of *binomial trees*, where each tree has a different *size*. What do binomial trees look like? First, we'll describe them *graphically* before we do so with list structures. Here are the first five such trees:



The *root* of tree B_k is said to be at *level 0*, followed by levels $1, 2, \ldots, k$. Observe that B_3, for example, has 1 node at level 0, 3 nodes at level 1, 3 nodes at level 2, and 1 node at level 3.

Exercise 0. Explain how B_{k+1} is made up of two B_k trees, and also trees $B_0, B_1, \ldots B_k$. In each case, explain how. How many subtrees are connected to the root of B_k? In the answer you give, pictures are OK.

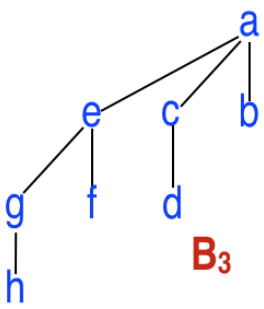
Exercise 1. Write $\left[\begin{smallmatrix} k \\ \ell \end{smallmatrix}\right]$ for the number of nodes at level ℓ of tree B_k. Use the pictures of the trees to explain why $\left[\begin{smallmatrix} k \\ \ell \end{smallmatrix}\right] = \left[\begin{smallmatrix} k-1 \\ \ell \end{smallmatrix}\right] + \left[\begin{smallmatrix} k-1 \\ \ell-1 \end{smallmatrix}\right]$, From that, explain why $\left[\begin{smallmatrix} k \\ \ell \end{smallmatrix}\right] = \frac{k!}{\ell!(k-\ell)!}$. *Hint:* try plugging the last equation into the previous one.

Exercise 2. Explain why $\Sigma_\ell \left[\begin{smallmatrix} k \\ \ell \end{smallmatrix}\right] = 2^k$. How many nodes does binomial tree B_k have?

Now we want to describe these trees by list structures. Here is how we will do it:

- A binomial tree of order 0 is just an integer. In other words, a leaf.
- A binomial tree of order k is a list of length $k+1$, such that:
  - Its `car` is an integer, the root.
  - Its `cdr`, which has length k, is a list of its children. The ith child will have order i.

Here is an example of an order-three binomial tree, B_3:




```
(a b (c d) (e f (g h))
```

Notice that B_0 is just an integer (a single node), and B_k is coded as a list whose first element is the integer at its root, and whose subsequent elements are the codings of its children, *in the order of their size.*

Here are some useful procedures for the next exercises—use them. They help to make coding the solutions easier.

```
> (define (ints from to)
   (if (> from to)
       '()
       (cons from (ints (+ 1 from) to))))
> (define (tack x L)
   (if (null? L)
       (cons x L)
       (cons (car L) (tack x (cdr L)))))
> (ints 5 17)
(5 6 7 8 9 10 11 12 13 14 15 16 17)
> (tack 0 (ints 5 17))
(5 6 7 8 9 10 11 12 13 14 15 16 17 0)
```

Exercise 3. Code procedure `evensplit` that takes an even-length list of elements and returns a list whose elements are *combinations* of of consecutive pairs of input elements. By combinations we mean: combining two lists means appending them; combining two numbers means creating a list holding them. You do not need to handle odd-length input. The elements of the input list will either all be integers, or all be lists of integers.

```
> (evensplit (ints 1 32))
((1 2) (3 4) (5 6) (7 8) (9 10) (11 12) (13 14) (15 16)
```

```
 (17 18) (19 20) (21 22) (23 24) (25 26) (27 28) (29 30) (31 32))
> (evensplit (evensplit (ints 1 32)))
((1 2 3 4) (5 6 7 8) (9 10 11 12) (13 14 15 16)
 (17 18 19 20) (21 22 23 24) (25 26 27 28) (29 30 31 32))
```

Again, notice that when successive list elements are combined, they are with `list` if they are integers, but `append` if they are lists of integers. (You might want to define a procedure to do this.)

Exercise 4. Use `evensplit` to define a procedure `split` that will divide a list of integers into another list of of elements which *may* begin with one integer (if the input list has odd length), followed by lists whose length is either 0 or some power of 2. No two non-empty lists can have the same length.

```
> (split (ints 1 13))
(1 () (2 3 4 5) (6 7 8 9 10 11 12 13))
> (split (ints 1 21))
(1 () (2 3 4 5) () (6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21))
```

Notice how `'()` is inserted where there isn't an appropriate group of a power of 2, and how a list of length 2^i is produced if and only if the binary representation of the input length has a 1 in the ith bit from the right. A good way of thinking of what lists they are comes from thinking about the base 2 representation of the lists: for example, a list of $13_{10} = 1101_2$ integers will be divided up as a list of three lists: one of length 8, one of length 4, and one of length 1.

Your solution must run in linear time: the way we interpret "linear time" is that the number of applications of `car`, `cdr`, `cons`, and `tack` is linear in the length of the input list. Justify your answer. *Hint:* Recall $1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\cdots = 2$.

Exercise 5. Define a procedure `tree` that builds a binomial tree from a list of integers. You can assume the input has power-of-two length, since a binomial tree has power-of-two many elements.

Moreover, the tree should be *heap ordered*: The root of the tree should be larger than that of its children, the root of each child should be bigger than the roots of that child's children, and so on. To do this, you will likely need a helper procedure that looks like `evensplit`, but that combines trees. (*Hint:* building a heap-ordered B_{k+1} from two heap-ordered B_k trees is pretty simple.)

Your solution must run in linear time. Justify your answer.

```
> (tree (ints 1 16))
(16 15 (14 13) (12 11 (10 9)) (8 7 (6 5) (4 3 (2 1))))
```

Exercise 6. (Subtle, and a bit difficult!) Your solution to Exercise 5 probably used `tack`—if assuming each use of `tack` takes $O(n)$ steps (i.e., uses of `cons`, `car`, `cdr`), you probably get an overall $O(n \log n)$ cost on building a tree on n nodes. Can you show that the cost is actually $O(n)$? *Hint:* $(\frac{1}{1-z})^2 = \sum_k kz^k$.

Now we can build a binomial queue from a list of n elements:

```
> (define (forest L) (map tree (split L)))
> (forest (ints 1 21))
(1
 ()
 (5 4 (3 2))
 ()
 (21 20 (19 18) (17 16 (15 14)) (13 12 (11 10) (9 8 (7 6)))))
```

This binomial forest on 21 nodes has a B_0, a B_2, and a B_4. Note $21_{10} = 10101_2 = 2^0 + 2^2 + 2^4$. Moreover, the root of each tree is the maximum among the nodes in it—so the maximum element in the queue is among the roots of the trees.

Exercise 7. Code `max-queue`, which should return the largest element in your binomial queue:

```
> (max-queue (forest (ints 1 21)))
21
```

Your implementation should run in time *logarithmic* in the number of integers in the queue–equivalently, in the number of trees in the queue. Justify this time bound for your solution.

Exercise 8. Define a procedure `insert` that will insert an integer into the queue:

```
> (define Q (forest (ints 1 19)))
> Q
(1 (3 2) () () (19 18 (17 16) (15 14 (13 12)) (11 10 (9 8) (7 6 (5 4)))))
> (insert 0 Q)
(() () (3 2 (1 0)) () (19 18 (17 16) (15 14 (13 12)) (11 10 (9 8) (7 6 (5 4)))))
```

The analogous problem you want to think about when doing this exercise is: how do you add 1 to a binary number? Your implementation should run in time *logarithmic* in the number of integers in the queue. Justify this time bound for your solution.

Exercise 9. This is a more difficult problem: define a procedure `merge` that will combine two binomial queues into a single queue:

```
> (define Q1 (forest (ints 1 9)))
> (define Q2 (forest (ints 21 43)))
> Q1
(1 () () (9 8 (7 6) (5 4 (3 2))))
> Q2
(21
 (23 22)
 (27 26 (25 24))
 ()
 (43 42 (41 40) (39 38 (37 36)) (35 34 (33 32) (31 30 (29 28)))))
> (merge Q1 Q2)
(() () () () ()
 (43
  42
  (41 40)
  (39 38 (37 36))
  (35 34 (33 32) (31 30 (29 28)))
  (27 26 (25 24) (23 22 (21 1)) (9 8 (7 6) (5 4 (3 2))))))
```

Notice `Q1` has 9 elements, `Q2` has 23, so the merged queue has 32 elements—and the forest for 32 elements has one B_5 tree. The analogous problem you want to think about solving this problem is: how do you add two binary numbers?

Your implementation should run in time *logarithmic* in the number of integers in the queue—or equivalently, linear in the number of trees in the queue. Justify this time bound for your solution.

Exercise 10. The final, and most difficult exercise in this problem set is to return the queue resulting from removal of its maximal element. To solve this exercise, you need to find the maximum element, remove the tree containing it (at its root) from the queue, and merge the subtrees of that tree (which form a queue) back into the queue with the tree removed.

```
> (define Q (forest (ints 1 21)))
> Q
(1 () (5 4 (3 2)) () (21 20 (19 18) (17 16 (15 14)) (13 12 (11 10) (9 8 (7 6)))))
> (remove-max Q)
(() () (20 1 (19 18)) () (17 16 (15 14) (5 4 (3 2)) (13 12 (11 10) (9 8 (7 6)))))
> (remove-max (remove-max Q))
(1 (19 18) () (17 16 (15 14) (5 4 (3 2)) (13 12 (11 10) (9 8 (7 6)))))
```

In solving this exercise, you will also probably need to code a procedure `shorten` that removes *unnecessary* `'()` elements from the queue, for example:

```
> (define Q (append (forest (ints 1 13)) '(() () () () ())))
> Q
(1 () (5 4 (3 2)) (13 12 (11 10) (9 8 (7 6))) () () () () ())
> (shorten Q)
(1 () (5 4 (3 2)) (13 12 (11 10) (9 8 (7 6))))
```

The analogy here is removing leading zeroes from a binary number: 00001101 is just 1101. Your implementation should run in time *logarithmic* in the number of integers in the queue—or equivalently, linear in the number of trees in the queue. Justify this time bound for your solution.