

Question 16–3: (Solution, p 2) Explain why it is a poor idea to use a Haskell list to represent the *Queue* abstract data type, where the front of the queue is the same as the front of the list.

Question 16–4: (Solution, p 2) Describe a data structure where we can support random-access `get` and `set` methods in Haskell, both taking $o(n)$ time on average. (If you haven't learned this, little-O means less-than, whereas big-O means less-than-or-equal. So I'm asking for a data structure where both `get` and `set` both take some amount of time *better* than $O(n)$.)

Question 17–1: (Solution, p 2) Complete the following class definition by writing the `>>=` function for the `Maybe` type.

```
data Maybe a = Just a | Nothing
instance Monad (Maybe a) where
    return x      = Just x
```

For this type, the expression `x >>= f` operator should yield `Just (f data)` if `x` is `Just data`, and it should yield `Nothing` if `x` is `Nothing`.

Question 17–2: (Solution, p 2) Convert the following Haskell expression using `do` notation to the equivalent using the *bind* Monad operator `>>=`.

```
do args <- getArgs
   fstr <- readFile (head args)
   putStrLn (show (length fstr))
```

Question 17–3: (Solution, p 2) Write a function `main` with type `IO ()` which reads two lines from the user and then prints them back in opposite order (the second line, then the first line).

Question 17–4: (Solution, p 2) Write a function `getLines` of type `IO [String]` that reads lines typed by the user until the user types “.” and then returns a list of the lines typed wrapped in an `IO`, not including the “.” line. You will use the `getLine` function (of type `IO String`) to read a single line.

Question 18–1: (Solution, p 2) What is the advantage of a tail-recursive function over other recursive functions?

Question 18–2: (Solution, p 2) Consider the following Haskell function.

```
pow2 0 = 1
pow2 n = 2 * pow2 (n - 1)
```

Convert it into an equivalent *tail-recursive* Haskell function.

Question Garbage–1: (Solution, p 3) Name and describe a disadvantage of using reference counting for garbage collection.

Question Garbage–2: (Solution, p 3) Describe the copy collection algorithm for memory reclamation.

Question Garbage–3: (Solution, p 3) When you think of the efficiency of explicit deallocation compared to that of garbage collection, you will naturally think that garbage collection must inherently takes more time, since the computer would spend time inferring what it would be told automatically in explicit deallocation. How is it possible that explicit deallocation can actually be *less* efficient than garbage collection?

Question Prolog–1: (Solution, p 3) Suppose we have a `parent` predicate defined in Prolog:`i/pre>`

```
parent(Cheri, Carl).
parent(Cheri, Hal).
parent(Chuck, Carl).
parent(Dorothy, Cheri).
parent(Dorothy, Vicki).
```

Define a Prolog predicate `sibling(A,B)` based on the `parent` predicate that represents whether A and B are siblings.

2 Solutions

Solution 16-3: (Question, p 1) Enqueueing data at the tail of a list takes $O(n)$ time, which is much more than we might hope. [Technically, since Haskell uses lazy lists, enqueueing takes $O(1)$ time, and it is *dequeuing* that must disentangle all the lazy conses, taking $O(n)$ time.]

Solution 16-4: (Question, p 1) We discussed two strategies in class; either is a good answer.

- We could use an array structure with a buffer listing up to \sqrt{n} recent modifications. When the buffer becomes full, we'll re-create the array and clear the buffer.

Re-creating the array takes $O(n)$ time, but that will occur only once every \sqrt{n} set operations. The other set operations take $O(1)$ time to simply buffer the change, for an average time of $O(\sqrt{n})$.

The get operation will involve searching the buffer and if the index is not found, going directly to the value in the array in $O(1)$ time. Searching the buffer takes $O(\sqrt{n})$ time.

- We could use a balanced tree indexed by element index. In this case, the depth of the tree will be $O(\log n)$, and so both get and set will take $O(\log n)$ time.

Solution 17-1: (Question, p 1)

```
data Maybe a = Just a | Nothing
instance Monad (Maybe a) where
    return x      = Just x

    Nothing >>= f = Nothing
    (Just x) >>= f = Just (f x)
```

Solution 17-2: (Question, p 1)

```
getArgs >>= (\args -> readFile (head args)
              >>= \fstr -> putStrLn (show (length fstr)))
```

Solution 17-3: (Question, p 1)

```
main = do first <- getLine
         second <- getLine
         putStrLn second
         putStrLn first
```

Solution 17-4: (Question, p 1)

```
readLines = do line <- getLine
               if line == "."
                 then return []
                 else do rest <- readLines
                          return line : rest
```

Solution 18-1: (Question, p 1) The optimizer can remove tail recursion in a tail-recursive function, eliminating the overhead involved in function calls that would otherwise be incurred with other recursive calls and removing the possibility of stack overflow with the function.

The optimization works by simply taking the parameter values of the recursive call and placing them in their appropriate locations for the current call. Then, the function can simply jump back to the beginning, thus surrendering control to the result of the recursive call.

Solution 18-2: (Question, p 1)

```
pow2 n = sub n 1
  where sub 0 ret = ret
        sub i ret = sub (i - 1) (ret * 2)
```

Solution Garbage–1: (Question, p 1) Garbage collection cannot identify unused objects involved in cyclic references. For example, even if two objects have instance variables referring to each other, and no other objects reference either of these objects, then the reference counts for both objects will be 1, and neither of them will be freed.

Solution Garbage–2: (Question, p 1) The heap is divided into two regions. Memory allocations are made progressively in the first region. When it is full, the copy collection algorithm takes over and identifies all the chunks in the first region that are useful, copying these over to successive locations in the second region. It must repair all pointers to refer to the updated locations.

After collection is complete, the program continues with memory allocation taking from successive locations in the second region. When that becomes full, the collector then switches to copying useful memory in the second region into the first region. The collector thus flips back and forth between the two regions with each collection.

Solution Garbage–3: (Question, p 1) There are three basic reasons that I can think of.

- When explicit deallocation is required, programmers for complex software must often resort to reference counting to track when a chunk of memory is no longer useful. Reference counting has a high overhead compared to other garbage collection techniques.
- Many garbage collection algorithms compact storage. While this is not in itself efficient, it leads to highly efficient memory allocation.
- Explicit deallocation often results in *memory leaks* where memory is allocated but not deallocated once it becomes unreferenced. Such memory leaks can impair a program's performance by chewing up memory. (More precisely, the program's memory could become scattered across a wide variety of pages, leading to a much heavier use of virtual memory than needed.)

[I want to add that the case for which is more efficient is not firmly established. (There's no question that garbage collection enables the programmer to be significantly more effective, though.) Probably explicit deallocation is faster for situations where reference counting does not turn out to be necessary. In fact, when people really want efficiency, they to opt for no dynamic memory allocation at all.]

Solution Prolog–1: (Question, p 1)

```
sibling(A, B) :- parent(X, A), parent(X, B).
```