Foundations of Computer Science

Supervision 4 – Solutions

10. Queues and search strategies

10.1. Conceptual questions

1. Suppose that we have an implementation of queues, based on binary trees, such that each operation takes logarithmic time in the worst case. Outline the advantages and drawbacks of such an implementation compared with one presented in the notes.

    > Binary tree operations usually take logarithmic time due to the branching structure of trees. This is a constant cost: although each operation may take more or less time (depending, for example, on the location of the element that we are searching for), the worst-case time complexity for the operations is logarithmic. In contrast, the list implementation of queues is constant *amortised* cost: we have to perform linear-time normalisation in the worst case, but most of the time the constant-time consing and unconsing operations are sufficient. The difference is that the second implementation has two modes of operation, one of which is more expensive but occurs rarely; the first implementation has just normal binary tree lookup which might take longer or shorter depending not on the structure of the collection, but on the element we are looking for. The binary tree version is therefore logarithmic, while the list implementation is constant amortised: while we might need to perform expensive normalisation, this “evens out” over the lifetime of the collection.

2. The traditional way to implement queues uses a fixed-length array. Two indices into the array indicate the start and end of the queue, which wraps around from the end of the array to the start. How appropriate is such a data structure for implementing breadth-first search?

    > While arrays allow for constant-time access to any element (including the front and end of the queue), its length is fixed: we would have a limit on what size tree we can traverse, or need to initialise a very large array that might end up being unnecessary for our purposes.

3. Why is iterative deepening inappropriate if $b \approx 1$, where b is the branching factor? What search strategy would make more sense in this case?

    > A tree with branching factor $b \approx 1$ is almost like a list with one element at every depth. Given that iterative deepening is appropriate when the number of nodes at each level increases exponentially, it would be wasteful for a linear structure such as a list. Normal depth-first or breadth-first search is suitable.

4. An interesting variation on the data structures seen in the lectures is a *deque*, or *double-ended queue* [1]. A deque supports efficient addition and removal of elements on both ends (either its front or back). Suggest a suitable implementation of deques, justifying your decision.

[1] Deque is usually pronounced "deck", which is convenient because a deck of cards is a good example of a double-ended queue – computer scientists are often way too clever with naming things.

> Double-ended queues must support enqueuing and dequeuing operations on both ends, so we need to find an implementation that gives constant-time access to both the "front" and "back" of the deque. Conveniently, our double list implementation for queues would work nicely for deques as well, as we can cons to and uncons from both halves of the list. The only difference is the normalisation algorithm: as we can get elements from both ends of the deque, we cannot just move every element to the first list and leave the second one empty. The alternative is to evenly split the normalised list between the two halves, so we always have elements we can access in constant time on both ends.

10.2. Exercises

5. Write a version of the function `breadth` shown on Page 88 using a nested `let` construction rather than `case`.

> We replace the `case`-expression with pattern-matching in `let`-bindings. To avoid repeated evaluation of `deq q`, we can assign it to a name too.

```
let rec breadth q =
  if qnull q then [] else
  let q' = deq q in
  let next = function
         | Lf -> breadth q'
         | Br (v,t,u) -> v :: breadth (enq (enq q' t) u)
  in next (qhd q)
```

6. Mathematical sets can be treated as an abstract data type for an unordered collection of unique elements. One approach we may take is to represent a set as an *ordered list* without duplicates: $\{5,3,8,1,18,9\}$ would become the OCaml list `[1;3;5;8;9;18]`. Code the set operations of membership test, subset test, union and intersection using this ordered-list representation. Remember that you can assume the ordering invariant for the inputs (and thereby make your functions more efficient), and your output should maintain this invariant.

> The membership test is similar to the usual `member` function for lists, except we can "short-circuit" if the first element is less than the head of the list. Given that the lists are ordered (here we assume an increasing order), the head of the list must be its smallest element; if we ask if an element smaller than the smallest element is in the list, we can say "no" right away. Hence we can add the `y < x` constraint as a conjunct to the recursive call, so we stop as soon as an element is found or if the element cannot be in the list.

```
let rec member x = function
  | []    -> false
  | y::ys -> (x = y) || (y < x && member x ys)
```

An empty set is a subset of anything, and nothing is a subset of the empty set. The general case checks the heads of the lists and makes the appropriate recursive call. We make use of the ordering constraint by comparing the heads: if `x < y`, the first list contains an element that cannot be in the second list, so we can short-circuit.

```
let rec subset xs ys = match xs, ys with
  | [],     _     -> true
  | _ ,     []    -> false
  | x::xs, y::ys -> (y <= x) && if x = y then subset xs ys
                                          else subset (x::xs) ys
```

Set union on ordered lists eliminates all the membership tests, as we can determine the elements contained by the union by simple comparison. The same can be done with set intersection as well.

```
let rec union xs ys = match xs, ys with
  | [],     ys    -> ys
  | xs,     []    -> xs
  | x::xs, y::ys ->
          if x < y then x :: union xs (y::ys)
      else if x > y then y :: union (x::xs) ys
      else               x :: union xs ys

let rec inter xs ys = match xs, ys with
  | x::xs, y::ys ->
          if x < y then inter xs (y::ys)
      else if x > y then inter (x::xs) ys
      else               x :: inter xs ys
  | _, _ -> []
```

10.3. Optional questions

7. Implement deques as an abstract data type in OCaml (including a type declaration and suitable operations). Estimate the amortised complexity of your solution.

As explained before, we can use the same double list implementation we gave for queues, with similar operations. The only function that needs changing is normalisation, as we want to evenly divide the elements between the two lists when either of the lists becomes empty. To accomplish this, we need to find the halfway point of the list where we can perform a split, which is a linear operation. One slight efficiency improvement we can make is a common trick when dealing with sized collections: instead of recalculating the length every time we need it, we keep track of it in the data structure itself as a separate field.

Then, a `length` operation simply returns this known value, and we don't need to find the length of the list in the normalisation function. There might be some extra bookkeeping required in the definitions below, but the point of abstract data types is that the actual implementation is hidden from view.

We make use of the helper function `split` which splits a list into two at a given index. Here we give a definition from scratch, but the result should have the property that `split l i = (take l i, drop l i)`. In `norm` we have two cases which need to be handled: when either the first or second list is empty. In those cases, we split the other list in half, and reverse one half based on which end of the deque we are looking at. The other operations are all straightforward, though we need to increment or decrement the length if we enqueue or dequeue and element.

```
let rec split xs n = match xs, n with
  | [],     _ -> ([], [])
  | x::xs, 0 -> ([], x::xs)
  | x::xs, i -> match split xs (i-1) with
                  (ys, zs) -> (x::ys, zs)

type 'a deque = D of 'a list * 'a list * int

let norm = function
  | D ([], tls, l) -> (match split tls (l / 2) with
                        (h1, h2) -> D (List.rev h2, h1, l))
  | D (hds, [], l) -> (match split hds (l / 2) with
                        (h1, h2) -> D (h1, List.rev h2, l))
  | d -> d

let dnull = function (D ([],[],0)) -> true | _ -> false
let enqFront (D (hds, tls, l), x) = norm (D (x::hds, tls, l+1))
let enqEnd   (D (hds, tls, l), x) = norm (D (hds, x::tls, l+1))
let deqFront (D (x::hds, tls, l)) = norm (D (hds, tls, l-1))
let deqEnd   (D (hds, x::tls, l)) = norm (D (hds, tls, l-1))
let length   (D (_, _, l)) = l
```

11. Elements of procedural programming

11.1. Conceptual questions

1. Comment, with examples, on the differences between an `int ref list` and an `int list ref`. How would you convert between the two?

> The first one is a list of references, e.g. `[ref 1, ref 2, ref 3] : int ref list`. Each reference can be individually assigned, but the list itself is immutable. The second one is a reference to a list, e.g. `ref [1,2,3] : int list ref`. We cannot change the elements individually, but we can change the whole list.
>
> Converting from a list of references to a reference to a list would involve dereferencing the list elements, and creating a reference to the resulting list. Since `(!) : 'a ref -> 'a` is just a function, we can map it over elements of a list:
>
> ```
> let reflistToListref rs = ref (List.map (!) rs)
> ```
>
> The other way is similar, except we dereference first and map `ref` after:
>
> ```
> let listrefToReflist lr = List.map ref (!lr)
> ```
>
> Of course, working with state and mutability means that the behaviour of these functions may not always be predictable.

2. What is the effect of `while (C1; B) do C2 done`? Where would such a formulation be useful?

> The sequencing operator `;` executes the commands left-to-right, and ignores the return value of all statements apart from the last one, which becomes the return value of the whole sequencing. Hence `C1` is executed on every iteration, but its value is replaced by the value of the Boolean condition `B`. The body of the while loop, `C2`, is executed on every iteration as well. In addition, `C1` is executed before the Boolean condition, so its side-effects are performed even if `B` evaluates to `false` right away. Therefore the expression is equivalent to `C1; while B do (C2; C1) done`. This formulation would be useful whenever we want to repeat instructions at least once; for example, making `C2` the empty instruction would result in a construct equivalent to a do-while loop, which checks the Boolean condition after the first iteration.

11.2. Exercises

3. Write a version of function power (Page 10) using `while` instead of recursion.

> Instead of tail-recursion, we use imperative iteration with a `while`-loop. We initialise an accumulator reference `accr` and references for our two inputs to allow mutability. What was an argument to the recursive call before becomes a reference assignment.
>
> ```
> let power x n =
>   let accr = ref 1.0
>   and xr = ref x
>   and nr = ref n
>    in while !nr <> 1 do
>         if !nr mod 2 = 0
> ```

```
          then (xr := !xr *. !xr; nr := !nr / 2)
          else (accr := !accr *. !xr;
                xr := !xr *. !xr; nr := !nr / 2)
        done;
        !accr *. !xr
```

4. Write a function to exchange the values of two references, `xr` and `yr`.

> While there are some type-specific hacks to perform the exchange “in-place”, we generally need a temporary reference to avoid overwriting a value that we need to use later.
>
> ```
> let swap xr yr = let v = !xr in xr := !yr; yr := v
> ```

5. Arrays of multiple dimensions are represented in OCaml by arrays of arrays. Write functions to (a) create an $n \times n$ identity matrix, given n, and (b) to transpose an $m \times n$ matrix. Identity matrices have the following form:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

> In both cases we make use of the `Array.init : int -> (int -> 'a) -> 'a array` function, which initialises an array with values calculated from the array index using a higher-order function argument. Matrices can be created by nesting these functions together, and using the current matrix index in the function argument of the inner `init` function. For example, an element of an identity matrix is 1 if its row and column index is equal, and 0 otherwise.
>
> ```
> let ident n = Array.init n (fun i ->
>               Array.init n (fun j ->
>                 if i = j then 1 else 0))
> ```
>
> There are several approaches to transpose a matrix: we can use references to perform the transposition in-place, but `Array.init` allows us to use a more elegant, functional solution. Matrices are represented as an array of rows, so getting a single row of the matrix is just indexing into the outer matrix. Selecting an individual column c is harder, since we need to return the c^{th} element of each of the rows. However, if we have a function that can do that, we can transpose the matrix by selecting the columns of the matrix and turning them into rows (by simply putting them in an array).
>
> The helper function below uses `init` to return a given column of the input matrix. Given an $n \times m$ matrix, it returns an array of size n (which is just the length of the outer array),

with its elements being the c^{th} element of the individual rows. “Looping” through the rows is done using `init` and `get`.

```
let getCol c arr = Array.(init (length arr)
                               (fun i -> get (get arr i) c))
```

The actual `transpose` function now uses `getCol` to create a new array of columns. The size of the outer array is the length of any of the inner arrays, and its r^{th} element is just the r^{th} column of the input matrix.

```
let transpose arr = Array.(init (length (get arr 0))
                                (fun r -> getCol r arr))
```