AD1 – Algorithms and Data Structures I (course 1DL210)

Assignment 2: Flexible Arrays

Due by 23:59:59 on Friday 25 April, 2008

Flexible Arrays

The array (`http://www.itu.dk/people/sestoft/mosmllib/Array.html`) is a very useful data structure, as one can access or update any element in constant time, provided its index is known. For example, if one has an array `a` of type `int array` then one can use the expressions

```
Array.sub(a,997)
Array.update(a,3,i)
```

in order to access the value of element `a[997]` and update the value of element `a[3]` to the value of `i`, respectively. However, arrays require that the memory needed to store all the elements is allocated contiguously and at once, as well as (for static arrays) that the maximum array size is known in advance. A rather important consequence thereof is that if for some reason one uses only a few elements in a large array, then one wastes a lot of memory. For example, assume one declares an array of a million elements, but for some reason only the values at the indices 3 and 997 are used in a given run; the declaration

```
val a = Array.array(1000000,0)
```

then allocates contiguous memory for a zero-based array `a` of a million integers and initialises them all to the default value zero, but only a very tiny percentage of these are used, which is very wasteful.

This is where the concept of flexible array comes in. A *flexible array* is like a normal zero-based array as far as how one uses it is concerned: one can access and update the element at a given index as easily as in a normal array. *Functionally*, there is thus no difference, but *computationally* there are differences in resource consumption. The first difference is that if one needs an array of a million elements, say, but really needs to use only a small number thereof, then the flexible array requires a much smaller amount of memory. This flexibility comes at a price though: the second difference is that the access to an element of known index is not constant-time any more.

Representing Flexible Arrays

The maximum size of a flexible array is not fixed in advance and can be arbitrary. One can represent a flexible array essentially as a list of chunks. Each *chunk*

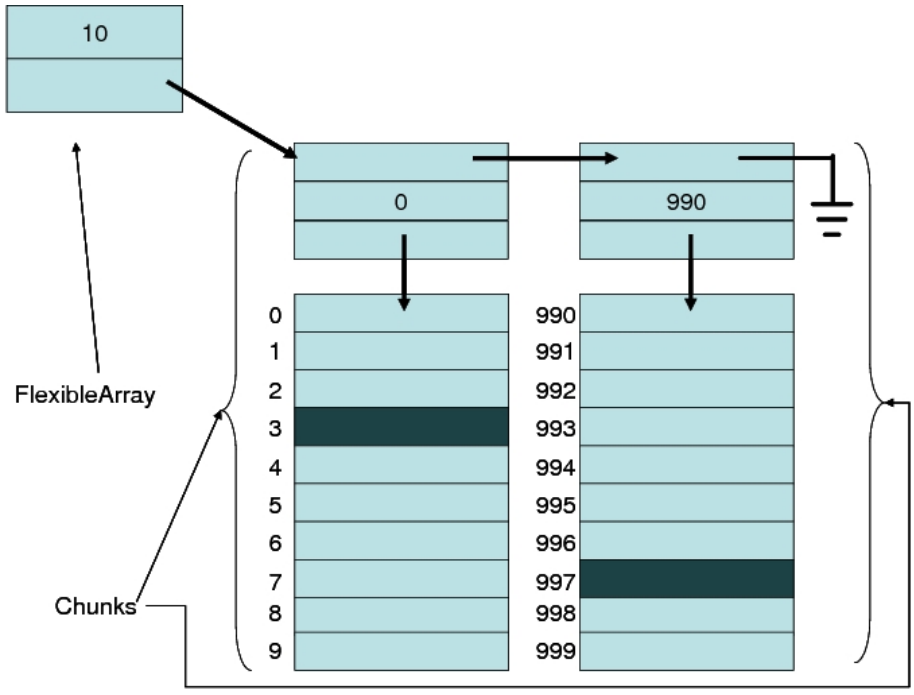
has a small fixed-size zero-based normal array of the same type of objects as the flexible array, plus the starting index within the flexible array of the chunk. The polymorphic `flexArray` abstract datatype has the following definition:

```
abstype 'a flexArray = Flex of {
  chunkSize  : int,
  defaultVal : 'a,
  chunks     : 'a chunk list ref
} with ... end
```

where `chunkSize` is the size of chunks and `defaultVal` is the default value for chunk elements when creating a new chunk. The polymorphic `chunk` type has the following definition:

```
type 'a chunk = {
  beginsAt : int,
  elements : 'a array
}
```

For example, a flexible array `a` with two chunks of 10 elements and with only the elements at indices 3 and 997 being used can be depicted as follows (careful, the default value is not shown):



Note that we have only allocated memory for 20 elements of the flexible array although it seems to have $1,000$ elements. If we need to store a value at index 6,

then we first go to the initial chunk (which holds the elements at indices 0 to 9) and then store that value at index 6 in that chunk. If we need to update the value at index 997, then we first find the chunk `c` where `c.beginsAt` $\leq 997 <$ `c.beginsAt` + `a.chunkSize` and then store the new value at index 7 in that chunk. In this case, we have such a chunk, namely the last one. If, on the other hand, we need to store a value at index 556, then we first create a new chunk, link it in after the first chunk, and then store that value at index 6 in that new chunk. Chunks in the list must be ordered increasingly by their `beginsAt` fields.

Work To Be Done

Implement the following functions:

- `array(c,d)` returns the empty flexible array of chunk size `c` and default value `d`; raises exception `Size` if `c` ≤ 0;
- `sub(a,i)` returns `a[i]`; raises exception `Subscript` if `i` < 0 or `i` $\geq$ `c`.`beginsAt` + `a`.`chunkSize`, where `c` is the last chunk of `a`;
- `update(a,i,x)` destructively replaces `a[i]` by `x` and returns `()`; raises exception `Subscript` if `i` < 0.

Give, in comments within the program, your explicit reasoning establishing the average-case and worst-case runtime complexities of your functions.

Grading

Your solution is graded in the following way:

- If your program was submitted before the deadline turns hard, loads under Moscow ML version 2.01, and is a serious attempt at implementing and commenting (under at least the coding convention) all the requested functions, then you get 30 points (before any penalty deductions for being late compared to the soft deadline); otherwise, you get 0 points.
- Your program is run on t orthogonal tests, checking also boundary conditions and error conditions. Each test is a flexible-array creation followed by a sequence of element accesses and updates. For each fully correct test result, you get $50/t$ points. We reserve the right to run these tests automatically, so be careful with names and argument orders.
- Your program is graded on style and comments (including specifications, representation conventions and invariants, and recursion variants), provided it does not fail on all the tests we perform. This covers 10 points.
- Your complexity analysis is graded for correctness of results and explicitness of reasoning. This covers 10 points.

Have fun!