CS157 Homework 6

BY TQIAN AND SILAO_XU

March 9, 2013

Problem 1

(30 points) You can allocate a block of n memory locations on your computer in constant time, however the contents of the memory in the block may be arbitrary. Typically, you will initialize these memory locations before you use them, by setting them all to a special symbol Empty, which takes $O(n)$ time.

The object of this problem is to create a new data structure that mimics the properties of an array, while being much faster to initialize, but while still ensuring that any values returned by this data structure are meaningful, and not uninitialized garbage.

You need to come up with a data structure that behaves like a 0-indexed array A of n elements. The following operations must take a constant time:

- set(index, value): Assign the value to A[index].
- get(index): Return the value from A[index]. If no value has yet been assigned, return Empty.
- initialize(n): Initialize the data structure so that it will mimic an array of size n, and where any **get** call returns **Empty**.

Warning: Keep in mind that, initially, the entries in memory can be arbitrary and may imitate valid parts of whatever data structure you design—your data structure should work *no matter what* is in memory initially.

Hints:

- Use more than n storage (but you do not need to use more than $O(n)$ storage).
- Most memory locations may be garbage, but think about how you can be sure some memory locations are meaningful.
- Because all operations in this data structure must take constant (worst case) time, you cannot use anything fancy: no hash tables, no binary search trees, no heaps, etc.

Important: If you are using extra space, please explain in sentences how they are used. As always, you need to communicate clearly that your proposed data structure works correctly, and that the running time for each operation, including initialization, is constant.

- initialize(n): Initialize the data structure so that it will mimic an array of size n, and where any **get** call returns **Empty**.

  We need another two n-element arrays and a counter.

  Counter is used for recording how many items are assigned values so far. It starts from 0 and ends with n (when all entries are filled).

Array F is used for recording the ith element's assignment order, *e.g.*, in the first time we assign value to $A[1]$, $F[1]$ should be 0 and in the second time we assign value to $A[8]$, $F[8]$ should be 1.

Array T is used as asserting whether entry i has been assigned value.

```
initialize(n):
1    A ← new array of length n
2    F ← new array of length n /* for recording assignment sequence */
3    T ← new array of length n /* for asserting entry i in A has been assigned */
4    count ← 0 /* for counting how many entries have been assigned so far */
```

- get(index): Return the value from A[index]. If no value has yet been assigned, return *Empty*.

  Initially, the condition in line 1 of the following pseudocode (F[index] $<$ count) would never be true because after filtering out all the valid assigning order number (F[index] $\geqslant$ 0), there would be no entry in F could be smaller than count-0. *Empty* would be returned in this case.

```
get(index):
1    if F[index] ⩾ 0 and F[index] < count and T[F[index]] = index then do:
2        return A[index]
3    else:
4        return Empty
```

  Firstly, we need to judge the index i we want to retrieve is in our assigning order. If not, the value has yet been assigned. If so, since there's still a possibility that garbage value could be stored in $F[i]$ and the value is by accidence smaller than count, we need another assurance, $T[F[i]]$, such that index i points to the only really assigned entries.

- *set* operation is based on *get* operation when the same entry in A is going to be modified. Otherwise, we need to set up condition telling that the index has already been assigned (see line 4 to 7).

```
set(index, value):
1    if get(index) ≠ Empty then do:
2        A[index] ← value
3    else:
4        A[index] ← value
5        F[index] ← count
6        T[top] ← index
7        count ← count + 1
```