York University EECS 4101 May 30, 2023

Homework Assignment #3
Due: June 9, 2023 at 10:00 p.m.

1. This question is a continuation of Assignment 2, in which we shall add support for deletions. Suppose we want to implement an ADT for a set of integers that supports the operations

- INSERT(x), which adds x to the set (assuming, as a precondition, that x is not already in the set),
- DELETE(x), which removes the item x from the set (assume that the argument actually indicates the location of x in the data structure), and
- SEARCH(x), which returns true if x is in the set and false otherwise.

Let n be the number of elements in the set. Once again, we store the elements in a linked list of arrays. Each element of the set appears in exactly one of the arrays. Each array is sorted in increasing order, but there is no constraint on how the elements of one array compare to the elements of another array. Each array has size 2^k for some non-negative integer k and no two arrays in the list have the same size. The arrays are ordered in the list by their size (in increasing order).

Various approaches could be used to delete elements. We shall use a simple technique of marking elements as deleted. Each array entry has an associated bit, where 0 indicates the item in the array entry is present in the set and 1 indicates that the item is marked as deleted (and is no longer in the set).

```
 1: function SEARCH($x$)
 2:     for each array $A$ in the list of arrays do
 3:         if $x$ is found by a binary search of $A$ and the entry is not marked as deleted then return true
 4:         end if
 5:     end for
 6:     return false
 7: end function
 8: function INSERT($x$)
 9:     precondition: $x$ is not already in the set
10:     create an array of size 1 containing $x$ and insert it at the beginning of the list
11:     $current \leftarrow$ first element in the list
12:     $done \leftarrow$ false
13:     loop
14:         exit when $done$ or $current.next$ is null                ▷ we reached end of list or did line 20
15:         if $current$ and $current.next$ are arrays of the same size then
16:             merge arrays $current$ and $current.next$ into a new array $new$
17:             replace $current$ and $current.next$ in the list by $new$
18:             $current \leftarrow new$                                 ▷ advance to new array for next iteration
19:         else
20:             $done \leftarrow$ true
21:         end if
22:     end loop
23:     $n \leftarrow n + 1$
24:     $space \leftarrow space + 1$
25: end function
26: function DELETE($A, i$)
27:     precondition: $A$ is an array in the linked list of arrays and $A[i]$ is an unmarked element of $A$
28:     set the mark bit of $A[i]$ to 1
29:     $n \leftarrow n - 1$
30:     if ______________________________ then REBUILD
31:     end if
32: end function
```

We have added two variables: n stores the number of elements currently in the set and *space* stores the total size of all the arrays used to represent the set. They are updated on lines 23, 24 and 29.

Remark: We maintain an invariant that if several copies of an element appear in an array, then only the leftmost one can be unmarked. This is maintained during merges: if the merge on line 16 compares two entries that have identical values and one is unmarked, it copies the unmarked entry into the merged array first. Thus, the binary search on line 3 should look for the leftmost copy of x.

[1] **(a)** Suppose the test on line 30 always evaluates to false. What problem(s) do you see with the performance of the data structure (in terms of time and/or space)?

[3] **(b)** We would like to improve the performance of the data structure by periodically rebuilding it using the REBUILD routine. The goal of this rebuild will be to discard all of the elements that are marked as deleted. Describe how you would implement the REBUILD function efficiently. Use Θ notation to describe the running time of your REBUILD in terms of n and/or *space*.

[2] **(c)** A REBUILD may be fairly expensive, so we do not want to do it too often. However, doing it too infrequently might not solve the problems you identified in part (a). Give a condition for line 30 that will ensure the amortized time per DELETE is reasonable, while ensuring that the amortized time per INSERT is still $O(\log n)$ and that every SEARCH still takes $O(\log^2 n)$ time.

[2] **(d)** Use Θ notation to give a good bound on the amount of space used as a function of n. Briefly justify your answer.

[5] **(e)** Consider a sequence of m operations (including insertions, deletions and searches). Let n_{max} be the maximum value of n at any time during the sequence. Do an amortized analysis of the new data structure, giving a good bound on the amortized time per DELETE while still maintaining an amortized time per INSERT of $O(\log n_{max})$.

You can use the accounting method or the potential method.

If you use the potential method, you can define the value of the potential to be

$$\Phi = \sum_{i=1}^{\ell} 2^{k_i}(\log_2 n_{max} - k_i) + \underline{\hspace{3cm}}$$

when the list contains arrays of sizes $2^{k_1}, 2^{k_2}, \ldots, 2^{k_\ell}$. (You have to figure out how to replace the blank to get a good bound for DELETE operations; if you do so correctly, most of the analysis of INSERT operations will be very similar to Assignment 2, but you must also consider how INSERT operations affect the value you write in the blank. The value you choose for the blank will also have to be useful for offsetting the expensive REBUILD operations.)

If you use the accounting method, you should clearly state your credit invariant, and prove it carefully.

Hint: It might be helpful to use the fact that, immediately after a rebuild, $\sum_{i=1}^{\ell} 2^{k_i}(\log_2 n_{max} - k_i) \leq n \log n_{max}$.