Indian Institute of Technology Kanpur
CS345: Algorithms II, 17–18

Student Name: Gurpreet Singh and Nikita Awasthi
Roll Number: 150259 and 150453
Date: 15 November, 2017

ASSIGNMENT

5

Question 1

For implementing this structure, we can have two (sorted) arrays, with space utilization $\geq \frac{1}{4}$. One array will store the the values of the elements in the set, whereas the other will store the count / validity of the element. Since we are not taking insertion into consideration, we need not worry about the sorted structure of the array.

Algorithm

Explanation

The only difference from a normal sorted array implementation is that we are lazily deleting the elements *i.e.* we are deleting (rather reconstructing) when we there have been $\frac{n}{2}$ deletions, where n is the size when the array is full (no element has been deleted, either after reconstruction or initially).

The `elements` array stores the element values in a sorted manner. This allows us to search through the elements, even if a few have been deleted, without reconstructing the array after every deletion. The validity of the element, *i.e.* checking whether the element has been previously deleted is supported by the additional array (here `pointers`).

The pointer array stores -1, at index i if the element in the `elements` array at index i has not been deleted, *i.e.* is valid.

If a series of elements have been deleted, say from index i to j, then the `pointers` will have the values j at index i and value i at the index j. This allows us to know the length of the series that has been deleted from either the starting index or the last index of the series. For the indices between i and j (exclusive), the value can be anything, but ≥ 0, therfore valid.

This series is also easy to update if any element after or before this series gets deleted, we can just merge this into this series (considering the concatention if there is one element between two separate series). Since we do not care about the values in between unless they are not -1, we can do this in constant time. This is crucial for the predecessor function, as otherwise, we would have to iterate throught the array to find the predecessor (since $\mathcal{O}(n)$) elements can get deleted.

Algorithm 1: Initialization of Arrays and Function for Recreating / Reordering Elements Array

```
elements ← SortedArray(S)            ▷ Array of values of elements in sorted manner
pointers ← Array(-1, -1 ...  -1) ▷ Array of -1s representing validity of elements
numElements ← size(S)

procedure reorder(size)
   newElements ← Array(size)

   count ← 0
   for i in {0...(elements.length − 1)} do
      if pointers[i] = -1 then
         newElements[count] ← elements[i]
         count ← count + 1
      end if
   end for

   elements ← newElements
   pointers ← Array({−1, −1 ... − 1}, size)                    ▷ Array of 'size' -1s
end procedure
```

Algorithm 2: Functions for Finding Index of Element

```
procedure find(key)
   start ← 0,   end ← elements.length - 1

   while start ≠ end do
      mid ← (start + end) / 2                          ▷ Integer Division with Round Down

      if elements[mid] > key then
         end ← mid
      else if elements[mid] < key then
         start ← mid + 1
      else
         return mid
      end if
   end while

   return start
end procedure
```

Algorithm 3: Functions for Search, Predecessor and Delete

```
procedure search(key)
   index ← find(key)

   if elements[index] = key and pointers[index] = -1 then
      return TRUE
   else
      return FALSE
   end if
end procedure

procedure predecessor(key)
   index ← find(key) - 1

   if index ≥ 0 then
      if pointers[index] = -1 then
         return elements[index]
      else if pointers[index] > 0 then
         return elements[pointers[index] - 1]
      end if
   end if

   return NULL
end procedure

procedure delete(key)
   index ← find(key)

   if elements[index] ≠ key or pointers[index] ≠ -1 then
      return FALSE
   end if

   start ← index,   end ← index
   if index > 0 and pointers[index - 1] ≠ -1 then
      start ← pointers[index - 1]
   end if
   if index < elements.length - 1 and pointers[index + 1] ≠ -1 then
      end ← pointers[index - 1]
   end if

   pointers[start] ← end,   pointers[end] ← start

   numElements ← numElements - 1
   if numElements ≤ elements.length / 2 then
      reorder( )
   end if

   return TRUE
end procedure
```

Time Complexity Analysis

For SEARCH and PREDECESSOR functions, it can be observed that we are simply using Binary Search (in the FIND function), and hence the worst case time in these cases is $\mathcal{O}(log\,(n))$

For the case of deletion, since we are reordering when the number of elements deleted is equal to the total size of the array, the worst case analysis is a verly loose bound. Hence, we use amortized analysis in this case.

Amortized Analysis of Deletion Operation

The potential function we will use for this case is the difference between the number of valid elements in the elements array and the total size of the elements array. In terms of the pseudocode, this can be defined as $\phi = c_3\,(elements.length - numElements)$.

NOTE c is just a positive constant multiplied to allow the proper use of the potential function

NOTE Valid element is an element that has not been deleted, since all the elements deleted are not instantly removed from the array

CLAIM *ϕ is a valid potential function*

PROOF

> Initially, the size of the elements array is $|S|$ and so is the number of valid elements. Hence, initially, $numElements = elements.length \implies \phi = 0$
>
> At any time, the length of the array is greater than the number of valid elements as we are never inserting elements. Therefore, $elements.length \geq numElements \implies \phi \geq 0$.
>
> Therefore, our potential function satisfies the properties of potential functions, and is thus valid.

For the actual cost, we will have to find the index of the element we require to delete. This is $\mathcal{O}(log\,(n))$. In case we need to reorder, we can do so in $\mathcal{O}(n)$

NOTE We set $c = 2c_3$

Case	**Actual Cost**	$\mathbf{\Delta\phi}$	**Amortized Cost**
When number of valid elements is greater than half the length of the elements array	$c_1\,log\,(n) + c_2$	$c\,(-1)$	$c_1\,log\,(n) + c_2 - 2c_3$
When number of valid elements is equal to one more than the length of the elements array	$c_1\,log\,(n) + c_3'n + c_4'$	$c\left(-\frac{n}{2}\right)$	$c_1\,log\,(n) + c_4$

Hence, the amortized cost for the deletion operation is $\mathcal{O}(log\,(n))$. Therefore, our algorithm works in the required constraints.

Question 2

Algorithm

Algorithm 4: Completed Update-R Function

```
procedure Update-R(i, j)
   if R[i] = true and R[j] = false then
      R[j] ← true
      for neighbours of j as q do
         Update-R(j, q)
      end for
   end if
end procedure
```

Amortized Analysis

Notations

We will call each vertex that is unreachable from the source vertex as marked. Therefore, in every insertion of any edge, it is possible that some of the vertices get unmarked *i.e.* for these vertices, $R[i]$ is now 1.

Also, for any edge $e = (u, v)$, we say that the edge e is marked if u is marked *i.e.* if $R[u] = 0$.

Potential Function

We define the potential function to be $\phi = k_1\,(\#(\text{marked edges}))+k_2\,(\#(\text{number of edges}) - \#(\text{unmarked nodes}))$ where k_1 and k_2 are positive constants.

Note For simplicity, we ignore the vertex s when counting the unmarked nodes

Claim *ϕ is a valid potential function*

Proof

> Initially, when no edge has been inserted, the number of marked edges and total edges is 0. Also, since no vertex is reachable now (as we are excluding the vertex s), therefore, the initial value of the potential function is 0, as required.
>
> Now, for the positivity of the potential function. Clearly, the number of marked edges can be only non-negative. Also, since for any vertex to be unmarked (excluding s), there must be at least one incident edge on this vertex. Hence, the number of edges is always greater than or equal to the number of unmarked nodes. Therefore, the potential function is always non-negative.

Since ϕ satisfies the properties of a valid potential function, it is a valid potential function.

Actual Cost

There are two cases, when the edge, say (u, v) is inserted, either v is marked or v is unmarked.

Case 1 *When the vertex u is marked i.e. $R[u] = 0$*

> If the vertex u is marked, from the algorithm, we can say, that the algorithm will stop at the first call, as it will not satisfy the outer 'if' condition will return $false$. Hence this case will take only constant time, say c_1

Case 2 *When the vertex u is unmarked i.e $R[u] = 1$*

> We will enter a recursion and will unmark all nodes which have not been unmarked yet and are reachable from this vertex. This is exactly doing DFS traversal over the nodes.
>
> From this analogy, we can say that the time taken will be $\mathcal{O}(n+m)$. We can define n to be the number of reachable and unmarked nodes from this vertex, and m to be the number of edges in this subgraph and the number of out-edges in this subgraph to already unmarked nodes. This is essentially the sum of the outdegrees of all the nodes in this subgraph.
>
> Since we are marking all visited nodes, we can reduce n to be the number of nodes unmarked in this recursion. Therefore, $n = \#(\textit{nodes unmarked in this insertion})$.
>
> Also, since m is the sum of outdegrees. For each edge included in this, it was a marked edge, which is now unmarked, as the vertex from which this is outgoing is now unmarked. Hence, we can define $m = \#(\textit{edges unmarked in this insertion})$
>
> Hence, the total complexity of this insertion is therefore $\mathcal{O}(\#(\text{nodes unmarked in this insertion}) + \#(\text{edges unmarked in this insertion}))$

From the algorithm, we can see that the complexity of Case 1 will be the coefficient of the second term in the complexity of the second case. If in the number of edges unmarked also includes an edge that has been inserted and unmarked immediately, we can say the complexity of both cases will be $c_2\,\#(\textit{nodes unmarked in this insertion}) + c_1\,\#(\textit{edges unmarked in this insertion})$
Therefore, our actual cost is

$$c_2\,\#(\textit{nodes unmarked in this insertion}) + c_1\,\#(\textit{edges unmarked in this insertion})$$

Amortized Cost

We can easily analyse the amortized cost of two cases. Here, we assume that the edge is inserted from u to v

We can define the difference in the potential function

$$\Delta\phi = k_2 - (k_1\,\#(\textit{edges unmarked in this insertion}) + k_2\,\#(\textit{nodes unmarked in this iteration}))$$

Case	Actual Cost	$\Delta\phi$	Amortized Cost
When the vertex u is initially unmarked	c_1	k_2	$c_1 + c_2$
When the vertex u is initially marked	$c_1m + c_2n$	$k_2 - (k_1m + k_2n)$	c_2

Table 1: Amortized Cost Table

Note For the following discussion, assume $n = \#(\textit{nodes unmarked in this insertion})$ and $m = \#(\textit{edges unmarked in this insertion})$

Note We have taken $k_1 = c_1$ and $k_2 = c_2$

Therefore, it is clear that the amortized cost if $\leq c_1 + c_2$. Hence the amortized cost is $\mathcal{O}(1)$, which implies that the time for n edge insertions is $\mathcal{O}(n)$.

Question 3

<++>