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 the structure described in the question, we have used two arrays. One array will store the values of the elements in the set in a sorted manner (increasing order of value). We can always visualize a sorted array as a weight balanced BST, with the middle element as the root pointing to the middle element of the left half, and the middle element of the right half, and so on.

The second array will be used as a translation from the linear array into this structure. Each element will store the size of the subtree formed at that element *i.e.* the middle element will store n, the length of the array, the middle element of the left half will store $\approx n/2$, and so on.

Explanation

If we assume that we can succesfully simulate a weight Balanced Binary Tree using two arrays, and solve the problem using a single BST, the visualization becomes easier.

The basic idea is that we do not delete an element when called upon, but delete lazily. That is, we delete the elements all together, when the total number of elements deleted are half of that of the size of the BST.

Consider the idea of deleting in a normal Balanced BST, such as a Red Black Tree. The procedure is to first replace the element with a leaf node, that does not harm the sorted nature of the BST, and then delete the element. We follow the same procedure, *i.e.* We replace the element with a valid leaf node. However, instead of deleting the element, we simply let it be, however, we invalidate the value, *i.e.* we mark it as deleted.

Note We consider any element with only deleted elements as its child nodes as a leaf node too, however, the second array `sizes will not change on deletion of an element`

While finding the valid leaf node, we must make sure that the leaf node itself has not been deleted. For searching and predecessor problems, we function exactly as we would in a normal Binary Search Tree.

The idea is very similar to pruning a tree, but by marking the lower level nodes.

Algorithm

Pseudocode

Algorithm 1: Initialization of Arrays and Functions for Recreating / Reordering the Arrays

```
elements ← SortedArray(S)           ▷ Array of values of elements in sorted manner
sizes ← Array(0, 0 ... 0)          ▷ Array of 0s for size of subtrees at elements
computeSizes(sizes, sizes)
numElements ← size(S)

procedure computeSizes(sizes)
   len ← sizes.length
   if len ≤ 1 then
      sizes[0] = len
   end if
   sizes[len / 2] ← len
   computeSizes(sizes[0 ... len / 2 - 1], sizes)
   computeSizes(sizes[len / 2 + 1 ... len], sizes)
end procedure

procedure recomputeArrays(size)
   newElements ← Array(size)
   count ← 0
   for i in $\{0...(\text{elements.length}-1)\}$ do
      if elements[i] ≠ NULL then
         newElements[count] ← elements[i]
         count ← count + 1
      end if
   end for
   delete elements
   delete sizes
   elements ← newElements
   sizes ← Array($\{0,0...\ 0\}$, size)                          ▷ Array of ‘size’ 0s
   computeSizes(sizes)sizes
end procedure
```

Algorithm 2: Search and Predecessor Functions

```
procedure search(key)
   start ← 0,   end ← elements.length - 1
   while start ≠ end do
      mid ← (start + end) / 2                  ▷ Integer Division with Round Down
      if elements[mid] = NULL then
         return NULL
      end if
      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

procedure predecessor(key)
   pred ← NULL
   index ← elements.length / 2
   while elements[index] ≠ NULL and sizes[index] ≠ 1 do
      if elements[index] < key then
         pred ← index
         index ← index + (sizes[index] - 1) / 2 + 1
      else
         index ← index - sizes[index] / 2 - 1
      end if
   end while
   return pred
end procedure
```

Algorithm 3: Delete Function

```
procedure left(index)
   return index - (sizes[index] / 2 - 1) / 2 - 1
end procedure

procedure right(index)
   if sizes[index] $\leq$ 2 then
      return index
   end if
   return index + (sizes[index] - 1) / 4 + 1
end procedure

procedure delete(key)
   index $\leftarrow$ search(key)
   if index = NULL then
      return
   end if
   ## Standard Code for Replacing with a leaf node (except using arrays)
   replacement $\leftarrow$ NULL
   while left(index) $\neq$ index and elements[left(index)] $\neq$ NULL do
      while right[replacement] $\neq$ replacement do
         replacement $\leftarrow$ right[replacement]
      end while
      switchValues(index, replacement)            $\triangleright$ Exchange values at the indices
      index $\leftarrow$ replacement
   end while
   if replacement = NULL then
      while right(index) $\neq$ index and elements[right(index)] $\neq$ NULL do
         while left[replacement] $\neq$ replacement do
            replacement $\leftarrow$ left[replacement]
         end while
         switchValues(index, replacement)
         index $\leftarrow$ replacement
      end while
   end if
   elements[index] $\leftarrow$ NULL
   numElements $\leftarrow$ numElements - 1
   if numElements = elements.length / 2 then
      recomputeArrays(numElements)
      return
   end if
end procedure
```

Time Complexity Analysis

For SEARCH operation, we are using a simple binary search on the sorted array of elements which has worst case $\mathcal{O}(\mathit{log}\,(n))$ bound.

For the PREDECESSOR function, since we are maintaining a balanced binary search tree using a sorted array of elements and an array maintaining the size of the subtree rooted atsevery node which gives us an $\mathcal{O}(1)$ worst case bound to access the left and right child of any given element. The height of the simulated tree therefore sis $\mathcal{O}(\mathit{log}\,(n))$. Since the height bound and $\mathcal{O}(1)$ access to left and right child ensures simulation of balanced search tree, we get predecessor of an element in $\mathcal{O}(\mathit{log}\,(n))$.

For the case of deletion, since we are reordering when the number of elements deleted is equal to atmost half the number of elements present in the array. Therefore on reordering the height of the tree is bounded by $\mathcal{O}(\mathit{log}\left(\frac{n}{2}\right))$ which is $\mathcal{O}(\mathit{log}\,(n))$. Also, since we have simulated a balanced binary search tree with access to left and right children through LEFT and RIGHT functions. Therefore, we can replace a deleted node with a leaf node in $\mathcal{O}(\mathit{log}\left(\frac{n}{2}\right))$ as the height of the tree is $\mathcal{O}(\mathit{log}\,(n))$.

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 $\phi = c_3\,(\mathit{elements.length} - \mathit{numElements})$.

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, $\mathit{numElements} = \mathit{elements.length} \implies \phi = 0$

At any time, the length of the array is greater than the number of valid elements as we are never inserting elements. Therefore, $\mathit{elements.length} \geq \mathit{numElements} \implies \phi \geq 0$.

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 $\mathcal{O}(\mathit{log}\,(n))$. When the number of deletions are less than half the number of elements present, the node is marked as NULL which is $\mathcal{O}(1)$ and then it is replaced with a leaf node.

Since the height of the simulated tree is bounded by $\mathcal{O}(\mathit{log}\,(n))$, the length of the path from the node to the leaf node would be atmost $\mathit{log}\,(n)$. Therefore, this part would take $\mathcal{O}(\mathit{log}\,(n))$

Note We set $c = 2c_3$

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

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, $n = \#(nodes\ unmarked\ in\ this\ insertion)$.
>
> 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 $m = \#(edges\ unmarked\ in\ this\ insertion)$
>
> 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 $c_2\,\#(nodes\ unmarked\ in\ this\ insertion) + c_1\,\#(edges\ unmarked\ in\ this\ insertion)$
Therefore, our actual cost is

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

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

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

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	$c_1 m + c_2 n$	$k_2 - (k_1 m + k_2 n)$	c_2

Table 1: Amortized Cost Table

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

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

Given a Fibonacci heap, the most fundamental property is that for a node x of degree k, size of the tree rooted at x is always $\Omega(a^k)$ or in other words the maximum degree of a tree in Fibonacci heap of size n is $(\,log\,(n))$.

All the bounds discussed in class were based on this property of Fibonacci Heap. All the bounds in the original discussion would therefore hold if we prove that this property holds for the modified problem as well.

Claim *Given a Fibonacci heap, if the subtree rooted at marked node v is cut from its parent and added to the root list only when it loses its third child, the bound on $size(x)$ for a node of degree k is $\Omega(a^k)$ always*

Proof

Let us consider the children of the node x, $(v_1, v_2, v_3, \ldots, v_k)$ such that they are ordered in the increasing order of time of becoming children of the npde x. At the time when a child v_i is added, for $i \geq 2$, $deg(v_i) \geq i-1$ as $deg(x) \geq i-1$. At the given instant, since v_i is still a child of the node x, therefore it could have lost atmost 2 children. Therefore, currently $deg(v_i) \geq i-3$

$$\begin{aligned}
s_k &= \text{minimum size of tree rooted at node of degree k}\\
s_0 &= 1\\
s_k &\geq 1 + 1 + 1 + \sum_{i=3}^{k} s_{i-3}\\
&\geq 3 + \sum_{i=0}^{k-3} s_i\\
&\geq 3 + \sum_{i=0}^{k-4} s_i + s_{k-3}\\
&\geq s_{k-1} + s_{k-3}
\end{aligned}$$

Now we try to get a bound on s_k by trying to get a bound on the following recurrence

$$f_n = f_{n-1} + f_{n-3} \tag{1}$$

We now try to prove that f_n is bounded by $(\sqrt{2})^n$.

Claim *The recurrence given by (1) is bounded such that $f_n \geq c^n$ for $n \geq 2$ where $c = \sqrt{2}$*

Proof

We prove this using induction

Note We only need to find one c

Base Case: For $f_1 = 1$, $f_2 = 2$ and $f_3 = 3$. Therefore, the bound is true trivially for $n = 2, 3$.

Induction Hypothesis: If the bound holds true for all i such that $i \leq n$, then it holds true for $n + 1$ as well for $n \geq 2$.

Induction step:

$$\begin{aligned} f_{n+1} &= f_n + f_{n-2} \\ &\geq c^n + c^{n-2} \\ &= (c)^{n-2} \cdot (1 + c^2) \\ &\geq c^{n+1} \quad = \quad (\sqrt{2})^{n+1} \end{aligned}$$

Therefore, the induction step holds and hence the bound holds.

Also given that $s_k \geq f_k \geq 2^k$, therefore our original claim holds. This proves that the bounds persist even in the modified form of decrease-key and other operations.

Hence, we can say that the fibonacci heap will still hold.