CS345 : Theoretical Assignment 5

Pranjal Prasoon (150508)
Raktim Mitra (150562)

November 2017

1 Question 1 (Hard Version)

1.1 Algortihm and Idea (with inherent proofs of correctness)

The idea is pretty simple- we use the power of Binary Search Trees (BSTs) :we maintain two arrays, each of size S initially, described as follows:

- Array **TREE**, which as the name suggests, stores the elements in a sorted order and acts as a binary search tree with the median of the entire array acting as root with the median of the left half of the array as its left child and the median of the right half of the array as its right child. The children's children are defined in a similar way.

  Any invalid element has a value=**NULL**.

- Array **SIZE** , which stores the size of the sub-tree rooted at every corresponding index of the array TREE. This array can be intialized easily with the following function: Intializing the array SIZE FnFunction
  index$_{start} \leftarrow 0$
  $index_{end} \leftarrow |S|$
  $init - size$(index$_{start}$,$index_{end}$) $len \leftarrow$index$_{end} - index_{start} + 1$
  $len = 1$ or 0 SIZE[0] $\leftarrow 1$
  $return$;

  SIZE[len/2] $\leftarrow len$
  $init - size$(0,len/2-1)
  $init - size$(len/2+1,index$_{end}$)

  Now, it is easy to see that at an index= i in the array TREE,its left child is located at index= $i - (SIZE[i]/2 - 1)/2 - 1$ while the right child is at index= $i + (SIZE[i] - 1)/4 + 1$, that is, can be accessed in $O(1)$ time like in any normal BST.

Now, the algorithms for different operations are as follows:

- **Search(S,x):** We simply search for x like we search on search on a BST or a simple binary search on the sorted array (done in class; trivial).

- **Pred(S,x):** We use the generic predecessor finding algorithm that we use in any BST (done in class; trivial).
- **Delete(S,x):** Here, we follow the following procedure:
  - Find the index corresponding to x (return false if not found).

    Now, we intend to replace x by a leaf-node (a node with no children or no non-NULL children), such that the the in-order of the BST (or the validity of the BST) is preserved. For this, we invoke the following procedure:
    - Find the successor of x (using the generic successor finding algorithm).
    - If this successor has no right child (the successor can't ever have a non-NULL left child), then we simply swap this successor and x and get done. Else, we still swap the successor and x, but re-invoke the procedure for x again and keep on doing so, until x becomes a proper leaf node (note that the validity of the BST's structure isn't violated anywhere).
    - At the index j which finally contains x, we make $TREE[j] = NULL$ that is, we mark x as invalid.
  - All this while, we keep a counter called *count* for storing the number of "valid" (non-deleted) elements in the BST and upon deleting x, we reduce it by one.
  - If this counter becomes ¡= —Size of BST—, we invoke a re-intialization procedure for both the arrays, where we do the following:
    - Copy all 'valid' elements from the array $TREES$ to a new array $Temp_1$ of size= *count* , preserving the order. We dump the old array $TREE$ and make $TREE = Temp_1$.
    - We dump the old array $SIZE$ and create a new $SIZE$ of size= *count* and re-initialize it using the new $TREE$ array and the function described above.

    This re-initialization or re-structuring takes $O(n)$ time, as is evident.

1.2 Time Complexity Analysis

We have a BST being brought into action by a sorted array, making the height bounded by $O(lg(n))$ and an array storing the size of sub-tree at every node, thus making the accessing of left and right child of any node possible in constant time. Thus, for **SEARCH(S,x)** and **PRED(S,x)**, the time bound will be $O(lg(n))$. Now, we present the amortized cost analysis for the **DELETE(S,x)** function as follows:

1.2.1 Potential Function:

ϕ is defined as "a constant (say, multiplied by the number of invalid (NULL) elements in the array TREE". This is a valid potential function because initially, the number of invalid elements is zero (making the initial value of $\phi = 0$) and quite trivially, $\phi \geq 0$.

1.2.2 Amortized cost

Note: $|TREE| = n$

We deal via two cases formulated as below:

- **Case 1:** When number of valid elements in the array TREE is at least two more than $|n|/2$, it means that one more deletion won't require a restructuring. So, the actual cost will just be the cost of replacing the deleted element with a leaf node, which entails travelling down the height of the tree exactly once (bounded by $O(lg(n)$ while performing $O(1)$ operations, making the actual cost $= c * lg(n) + c_1$. In this case, the final potential function will be one (times a constant) greater than the initial potential function as the number of invalid elements has increased by 1 that is, $\Delta\phi = c_2 * 1)$

  Thus, total amortized cost= $c * lg(n) + c_1 - c_2 = O(lg(n))$.

- **Case 2:** When number of valid elements in the array TREE $= |n|/2 + 1$, that is one more deletion will force a restructuring. Now, the actual cost will be the actual cost of case 1 plus the additional linear time involved in restructuring, making it $= c * lg(n) + c_1 + c_3 * n$. The change in potential function, will however be $\Delta\phi = c_2 * (-n/2)$ because initially, there were $n/2$ invalid elements (making $\phi = c_2 * n/2$) while after restructuring, there are no invalid elements (making $\phi = 0$).

  Thus, total amortized cost= $c * lg(n) + c_1 + c_3 * n + c_2 * (-n/2) = O(lg(n))$, by choosing $c_3 = c_2/2$.

Thus, we see that total amortized cost bound for deletion is $O(lg(n))$ in all cases.

2 Question 2 (Hard Version)

Recursive Procedure

FnFunction

Update-R(i,j)

$R[i] \neq 0$ and $R[j] = 0$ R[j] $\leftarrow true$

each neighbour q of jUpdate-R(j,q)

2.1 Potential Function

The potential function $\phi(i, j)$ after insertion of edge (i,j) is defined as:

$$\phi(i, j) = c * |E'| \tag{1}$$

Where E' =the set of edges of the form (x, y) in the graph G_{ij} obtained after insertion of the edge (i, j) such that $R[x] = false$.

In the beginning, no edge is there and $\phi = 0$ and as the number of edges ≥ 0, hence $\phi \geq 0$ and thus, it is a valid potential function.

2.2 Actual Cost

Note: For the our method approaching at this concise definition of Actual Cost, refer to the Amortized analysis below

Actual cost, $A(i, j)$ of inserting the edge (i, j) is defined as:

$$A(i, j) = c * |E''| \tag{2}$$

Where, E'' = the set of edges of the form (x, y) such that $R[x] = false$ and both x and y are reachable from i.

2.3 Amortized analysis

To analyse the cost of inserting the edge (i, j), we deal with two cases:

- **Case 1: R[i]=false** : In this case, **the actual cost** of inserting the edge is a constant (say, C_1 (as is easily seen from the pseudo code above). For the amortized cost, we take note of the fact that the number of edges of the form (x, y) in the graph G_{ij} obtained after insertion of the edge (i, j) such that $R[x] = false$ will have **increased by 1** ($\Delta\phi = 1$) from the previous graph because the R-value of all other vertices which are reachable from i remains unchanged (again visible easily from the pseudo code).

  Hence, the amortized cost= $C_1 + 1 = C$, which is a constant ($O(1)$).

- **Case 2: R[i]=true** :

  - **Actual Cost:** In this case, the inner loop will get executed for all js such that $R[j] = false$ and recursively, it will be executed for all neighbours of all such js reachable from i. Hence, the total number of times the loop runs is just the sum of 'out degrees' of all such js.

    Therefore, the actual cost=$\sum_k out - degree(k)$, where k is a vertex reachable from i with $R[k] = false$.

  - $\Delta\phi$**:** To find $\Delta\phi$, we note that the number of edges of the form (x, y) in the graph G_{ij} obtained after insertion of the edge (i, j) such that $R[x] = false$ will have decreased by the same amount, that is $\sum_k out - degree(k)$, where k is a vertex reachable from i with $R[k] = false$, as every such k undergoes a change to $R[k] = true$ (as

is seen in the pseudo code) and hence, all the outgoing edges from thes ks will undergo the change as well (no other edges change, as is evident from the code).

Hence, $\Delta\phi = -\sum_k out - degree(k) = -$Acutal Cost, where k is a vertex reachable from i with $R[k] = false$.

Thus,the amortized cost in this case = 0.

Thus, as the amortized cost in both the possible cases is a constant ($O(1)$), hence, the total amortized cost for processing m insertions is $O(m)$.

3 Question 3 (Hard Version)

Let the vertex v have children $(u_1, u_2, ..., u_k)$ such that if $j > i$, u_i became attached to v before u_j.

Now, consider u_i. When it was attached to v, v already had a degree $= i-1$ and hence, initially, degree of $u_i \geq i-1$. Now, till the time u_i was a child of v, it must have had a degree $/geqi - 3$ as it could have lost at most two children till then.

Now, the size of v (which has a degree k) (s_k) is given as : $s_k = \sum_{i=1}^{k} size(u_i) + 1$
$\geq \sum_{i=3}^{k} s_{i-3} + 3$
$= \sum_{i=0}^{k-3} s_i + 3$
$= \sum_{i=0}^{k-4} s_i + 3 + s_{k-3}$
$\geq s_{k-1} + s_{k-3}$
Note: We have used size¿1 and uniqueness of sizes in a Fib. heap repeatedly in the above derivations.

Now, this recurrence, $s_k \geq s_{k-1} + s_{k-3}$ is lower bounded by the recurrence $f_k \geq f_{k-1} + f_{k-3}$. Taking $f_1 = 1$ and $f_2 = 2$, we claim that $f_n \geq (7/5)^n$ for all $n > 1$.

For $n = 2, 3$ it holds. Now, we assume it holds for all $n \leq m$. Now, $f_{m+1} = fm + fm - 2$
$\geq (7/5)^m + (7/5)^{m-2}$
$\geq (7/5)^{m-2}(49/25 + 1)$
$\geq (7/5)^{m+1}$ This establishes induction and our claim holds.

Thus, $s_k \geq f_k \geq (7/5)^k$ and hence, the degree of any node will be again bounded by $O(lg(n))$, making our analysis done in the class hold.