CS345 Theoretical Assignment 5

Ayush Agarwal, 13180

M.Arunothia, 13378

Contents

1 Binary search under insertion

1.1 Data Structure Overview

The data structure consists of arrays of multiple arrays with size of the form 2^i. For searching, we search each of these arrays. To insert an element, we first insert it into the first array(of size 1). If it is full we try to insert them into array of size 2. If that is also full we try the same for array of size 4 and so on. This insertion operation is amortized $logn$.

- **array**[i] is the pointer to array of size 2^i
- **filled** number of arrays being used

```
1: procedure insertion(int newElement)
2:     i ← 0
3:     while !array[i].empty() do
4:         i++
5:     end while
6:     array[i] ← merge(array, i, newElement)
7:     if i >= filled then
8:         filled ← i
9:     end if
10: end procedure
11:
12: procedure merge(int *array[], int end, int newElement)
13:     Merge all the arrays with lengths starting from 1 to 2^end and insert the new element into that array.
14:
15:     i ← 0
16:     while i < end do
17:         free(array[i])
18:         i++
19:     end while
20:     return mergedArray
21: end procedure
22:
23: procedure search(int element)
24:     i ← 0
25:     while i <= filled do
26:         if binarySearch(array[i], element) then
27:             return True
28:         end if
29:         i ← i + 1
30:     end while
31: end procedure
```

1.2 Justification

At any point of time, any of the $array[i]$ will either be empty or completely filled. There can't be a case when any of the array is partially filled. Think of it like the binary number representation of number n. The indexes where n has value 1, those corresponding arrays are filled, rest are empty.

1.3 Time Complexity Analysis

- **search** Worst case time would be when every array from size 1 to size 2^{logn-1} would be filled. Without the loss of generality assume n = 2^k So search time,
  = log1 + log2 + ... logn/2
  = (1 + 2 + 3 ... k-1)
  = k(k-1)/2
  = O(log^2n)

- **insert**
  Let amortized function

$$\phi(i) = \sum_{e}^{allelements} depth(e) = \sum_{i}^{i \leq logn} 2^i(logn - i)$$

after the i^th insertion where $depth(e) = logn - arrayIndex(e)$.

Amortised Analysis			
Case	Actual Cost	$\Delta(\phi)$	Amortised Cost
Direct Insert(no merge)	1	logn	O(logn)
On merge(till array 2^k)	2^{k+1}	$-\sum_i^{i<k}(2^i(k-i))$	k+1 $\leq$ O(logn)

2 Binary search and predecessor/successor queries under deletions

2.1 Data Structure Overview

The data structure proposed is an array Z of size $|S|$. Each entry in the array has the following -

- Value x, where $x \in S$
- Flag which is marked valid at the start. Invalid flag implies that entry has been deleted.
- Index nextValid, that stores the next valid entry's index correctly for any valid entry.
- Index prevValid, that stores the previous valid entry's index correctly for any valid entry.
- For every index belonging to $\{logn, 2logn, ..\}$, a boolean array of size $O(logn)$ is stored. The boolean array of ith index will says whether there is any valid entry between i and each of $\{i+1, i+2, i+4, i+8, .., i-1, i-2, i-4, i-8, ..\}$.

It can be easily seen that the data structure is $O(n)$. Apart from this we maintain global variables $Size$ and $DelCounter$. $Size$ tracks the size of the set S whenever it is halved. $Size = n$ and $DelCounter = 0$ at the start.

2.2 Search(x, Z): search for element x in S

2.2.1 Algorithm and Correctness

The procedure is similar to a binary search with the only difference being that if the found entry is *invalid*, then we return $notFound$. The proof hence follows from the correctness of binary search. Note that we are given a set, implying there will be no repeating entries. This ensures that if at all an element existed in S, then our search will return its index correctly.

2.2.2 Time Complexity Analysis

Worst case - $O(log(n))$. Follows from the similarity with binary search.

2.3 Predecessor(x, Z): report the largest elements in S which is smaller than x

2.3.1 Algorithm

- We use binary search procedure to find the predecessor of x in S. If the returned entry is *valid*, we are done. If not consider this returned index as i.
- Now we should find the *valid* entry that is just before i. We first traverse linearly from i to the first index of the form $klogn$, all the while checking for any valid entry. If valid entry found we are done. Otherwise, go to next step.
- We use the boolean array of the $klogn$th entry to find l such that the first valid entry before index i lies in the range $(klogn - 2^{l-1}, klogn - 2^l)$
- Let us say that the first *valid* entry actually lies at $klogn - m$. We have to find m. Look at the binary representation of m. It has $l - 1$ digits. with first digit as 1. Now each of the entry from left to right can be filled in $O(1)$ time by discarding half of the potential search space using the boolean arrays stored at the $jlogn$ indices.
- return $klogn - m$

2.3.2 Time Complexity Analysis

Worst case - $O(log(n))$. Each step mentioned in the algorithm is $O(logn)$

2.4 Delete(x, Z): Delete element x from S

2.4.1 Algorithm

- We use binary search procedure to find the entry to be deleted, say it is i
- If $DelCounter < Size/2$, we do the following
  - Increment $DelCounter$
  - $i.flag = invalid$
  - $i.prevValid.nextValid = i.nextValid$
  - $i.nextValid.prevValid = i.prevValid$
  - For all entries of the form $jlogn$ between $i.prevValid$ and $i.nextValid$, we modify their boolean arrays accordingly so as to maintain the definition of these boolean values. (i.e)., if this was the last valid element in any range that has to be reflected in those boolean values.
- Else
  - Remove all invalid elements from the array and rebuilt the whole array that has now reduced to half its size.
  - $DelCounter = 0$
  - $Size = Size/2$

2.4.2 Time Complexity Analysis

We use amortised analysis to analyse the Time Complexity of the delete function. We define our potential function as $\phi(X) = 2k * (X.Size)+$ Total number of $TRUE$ stored in the boolean arrays of the $jlogn$ indices.
Let $H(X)$ =Number of boolean entries that has been turned $FALSE$ from being $TRUE$ in this turn.
This implies that $\Delta(\phi)$ will be 0 in the first case, as there is no change here and will be $2k*((Size/2)-Size) = -k*Size$ in the second case.

Amortised Analysis			
Case	Actual Cost	$\Delta(\phi)$	Amortised Cost
$DelCounter < Size/2$	clog(Size) + H(X)	0	clog(Size)
$DelCounter = Size/2$	clog(Size) + k*Size + H(X)	-k*Size	clog(Size)

Hence, deletion is done in amortised $logn$.

3 Extension of the problem of the mid-semester exam

3.1 Pseudo Code

```
1: procedure findSubgraph(V,E)
2:     E_s ← φ
3:     while V! = φ do
4:         Pick any vertex v from G
5:         if degree(v) <= n^{1/k} then
6:             Add all edges incident on v to E_s
7:             Remove v and all edges incident on v from G
8:         else
9:             Do a BFS from v in graph G till a depth of k − 1.
10:            Remove all the non-tree edges (in the formed k − 1 levels) from G
11:            Call this formed tree of depth k − 1 as cluster.
12:            For all v ∉ cluster retain just one edge with the cluster and remove all other edges from G
13:            Add all the tree(cluster) edges along with the edges incident on the cluster to E_s
14:            Remove cluster and all edges incident on cluster from G
15:        end if
16:    end while
17:    return Es
18: end procedure
```

3.2 Justifications

3.2.1 Retaining just one edge with the cluster for any $v \notin cluster$ is sufficient

Let v be the root of the *cluster* being discussed. Let $u \notin cluster$ be the vertex outside cluster who has edges to both $x \in cluster$ and $y \in cluster$. Let us consider what happens when we remove say the edge (u, y). We know v being the root of the BFS tree, is connected to both x and y. As the depth of the BFS tree being considered is $k-1$, the maximum path length between v and x(or y) is $k-1$. This means there is a path between x and y via v that has a maximum length of $2(k-1)$. Though we have removed the edge (u, y), u and y are still connected by the path (u, x), then x to y via v. The maximum length of this path between u and y is hence, $2k-1$, satisfying the requirement asked for in the question. This explains why retaining just one edge with the cluster for any $v \notin cluster$ is sufficient.

3.2.2 Non-Tree edges need not be retained

As within the tree any two vertices are always connected via a path whose length $\leq 2k-1$, there is no need to retain non-tree edges.

3.3 Proving $|E_s| = O(n^{1+1/k})$

- If $degree(v) <= n^{1/k}$ then, we add atmost $n^{1/k}$ edges to E_s for the single vertex v.
- If $degree(v) > n^{1/k}$ then, we add atmost $O(n)$ edges to E_s for a *cluster* of atleast $n^{1/k}$ vertices.
- Worst case $|E_s| = O(n^{1+1/k})$

3.4 Time Complexity

The overall algorithm accesses an edge exactly for O(1) times, because in any iteration of the while-loop the edge getting accessed is being removed off from G and hence, it is guaranteed that no edge is being accessed in two different iterations. Hence, the time complexity is $O(m+n)$.