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

```
procedure insertion(INT NEWELEMENT)
    i ← 0
    while !array[i].empty() do
        i++
    end while
    array[i] ← merge(array, i, newElement)
    if i >= filled then
        filled ← i
    end if
end procedure

procedure merge(INT *ARRAY[], INT END, INT NEWELEMENT)
    Merge all the arrays with lengths starting from 1 to 2^end and insert the new element into that array.

    i ← 0
    while i < end do
        free(array[i])
        i++
    end while
    return mergedArray
end procedure

procedure search(INT ELEMENT)
    i ← 0
    while i <= filled do
        if binarySearch(array[i], element) then
            return True
        end if
        i ← i + 1
    end while
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

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

(where i is all the nonzero indices in the binary representation of n) 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}	$-\sum_{i}^{i<k}(2^i(k-i)) = -(2^{k+1} - k - 2)$	k+2 $\leq$ O(logn)

2 Binary search and predecessor/successor queries under deletions

2.1 Data Structure Overview

3 Extension of the problem of the mid-semester exam

3.1 Pseudo Code

```
procedure findSubgraph(V,E)
    E_s ← φ
    while V! = φ do
        Pick any vertex v from G
        if degree(v) <= n^{1/k} then
            Add all edges incident on v to E_s
            Remove v and all edges incident on v from G
        else
            Do a BFS from v in graph G till a depth of k − 1.
            Remove all the non-tree edges (in the formed k − 1 levels) from G
            Call this formed tree of depth k − 1 as cluster.
            For all v ∉ cluster retain just one edge with the cluster and remove all other edges from G
            Add all the tree(cluster) edges along with the edges incident on the cluster to E_s
            Remove cluster and all edges incident on cluster from G
        end if
    end while
    return Es
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 $O($ *vertices being removed* + *vertices not being removed* $)$ edges to E_s for a *cluster* of atleast $n^{1/k}$ vertices.
  - The number of edges that belong to the *cluster* is of $O($ *vertices being removed* $)$ as it is a tree.
  - The number of edges that are incident on the cluster is $O($ *vertices not being removed* $)$ as proved from justification (1).
  - Hence, the given order.
- Worst case $|E_s| = n * n^{1/k} + \sum_{i=0}^{k} n^{1-i/k} = O(n^{1+1/k})$

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 $O(m + n)$.