CS345 Theoretical Assignment 5

Ayush Agarwal, 13180

M.Arunothia, 13378

Contents

1 Binary search and predecessor/successor queries under deletions

1.1 Data Structure Overview

The data structure proposed is an Zay $Z[0,..,n-1]$ containing elements of type *struct node*. The *struct node* has the following elements in it -

- *int item* - where $item \in S$
- *bool flag* - where $flag$ is *valid* for an existing element and is *invalid* for a deleted element
- *int nextValid*() - that returns the index of the next valid entry in the Zay
- *int grParent* - that stores the index of the invalid entry that also has the same nextValid (to enable computation of nextValid)
- *int sz* - that stores the number of elements present in the same invalid group (i.e) those invalid entries that have the same nextValid entry (relevant only for the head of the group)

The Zay Z is sorted based on the entry value *item* at the start. This completes the description of the data structure Z that is built from the given set S. 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.

procedure nextValid()
    $i \leftarrow this$
    while $Z[i]! = Z[i].grParent$ **do**
        $i \leftarrow Z[i].grParent$
    end while
    return i
end procedure

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

1.2.1 Pseudo-Code

procedure Search(x,Z)
    $first \leftarrow 0$
    $last \leftarrow n-1$
    while $last >= first$ **do**
        $mid \leftarrow (last + first)/2$
        if $Z[mid].flag == valid$ **then**
            $pos \leftarrow mid$
        else
            $pos \leftarrow Z[mid].nextValid()$
        end if
        if $Z[pos].item > x$ **then**
            $last \leftarrow pos - 1$
        else if $Z[pos].item < x$ **then**
            $first \leftarrow pos + 1$
        else
            return pos
        end if
    end while
    return $notFound$
end procedure

1.2.2 Poof of Correctness

The procedure is similar to a binary search. Whenever an entry has its $flag$ to be *invalid*, we refer to the closest *valid* entry for the comparison. The proof hence follows from the correctness of binary search.

1.2.3 Time Complexity Analysis

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

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

1.3.1 Pseudo-Code

```
procedure Predecessor(x,Z)
    first ← 0
    last ← n − 1
    while last >= first do
        mid ← (last + first)/2
        if Z[mid].flag == valid then
            pos ← mid
        else
            pos ← Z[mid].nextValid()
        end if
        if Z[pos].item >= x then
            last ← pos − 1
        elseZ[pos].item < x
            if Z[pos + 1].flag == valid then
                nextValue ← Z[pos + 1].item
            else
                nextValue ← Z[Z[pos + 1].nextValid].item
            end if
            if nextValue < x then
                first ← pos + 1
            else
                return pos
            end if
        end if
    end while
    return notFound
end procedure
```

1.3.2 Poof of Correctness

The procedure is similar to a binary search. Whenever an entry has its $flag$ to be $invalid$, we refer to the closest $valid$ entry for the comparison. The find condition for predecessor is that **the element is valid and is less than x and also the next valid element is not less than x**. The proof hence follows from the correctness of binary search.

1.3.3 Time Complexity Analysis

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

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

1.4.1 Pseudo-Code

```
procedure MergeGroups(i,j)
    k ← Z[i].nextValid()
    l ← Z[j].nextValid()
    if Z[k].sz < Z[l].sz then
        Z[k].grParent ← l
        Z[l].sz ← Z[k].sz + Z[l].sz
        Z[k].sz ← 0
    else
        Z[l].grParent ← k
        Z[k].sz ← Z[k].sz + Z[l].sz
        Z[l].sz ← 0
    end if
end procedure
```

Delete Function does an $O(log(n))$ modification usually and does an $O(nlog(n))$ only when size of the original array has halved. There are four cases that can happen when x is getting deleted.

- Both Previous and next entries of x are valid.
- Previous entry of x is valid and next entry of x is invalid.
- Previous entry of x is invalid and next entry of x is valid.
- Both Previous and next entries of x are invalid.

```
procedure Delete(x,Z)
    pos ← Search(x, Z)
    if pos == notFound then
        return notFound
    else
        if DelCounter < Size/2 then
            if Z[pos − 1].flag == valid then
                if Z[pos + 1].flag == valid then
                    Z[pos].flag = invalid
                    Z[pos].grParent = pos
                    Z[pos].sz = 1
                else
                    Z[pos].flag = invalid
                    Z[pos].grParent = pos + 1
                    k ← Z[pos + 1].nextValid()
                    Z[k].sz ← Z[k].sz + 1
                    Z[pos].sz = 0
                end if
            else
                if Z[pos + 1].flag == valid then
                    Z[pos].flag = invalid
                    k ← Z[pos − 1].nextValid()
                    Z[pos].grParent = pos
                    Z[pos].sz ← Z[k].sz + 1
                    Z[k].grParent = pos
                    Z[k].sz ← 0
                else
                    Z[pos].flag = invalid
                    Z[pos].grParent = pos + 1
                    MergeGroups(pos − 1, pos + 1)
                    k ← Z[pos + 1].nextValid()
                    Z[k].sz ← Z[k].sz + 1
                    Z[pos].sz = 0
                end if
            end if
            DelCounter ← DelCounter + 1
        else
            Remove all entries whose flag = invalid
            DelCounter ← 0
            Size ← Size/2
        end if
    end if
end procedure
```

1.4.2 Poof of Correctness

1.4.3 Time Complexity Analysis

We use amortised analysis to analyse the Time Complexity of the delete function. We define our potential function as $\phi(X) = 2k * (X.Size)$. 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)	0	clog(Size)
$DelCounter = Size/2$	clog(Size) + k*Size	-k*Size	clog(Size)