

## Problem Set 4

Name: Subhankari Mishra

Assigned: June 17

Due: June 24

### PROBLEM 1

Modify the definition of a 2-3 tree so that it supports the following operations with the specified running times. You may assume that the reader understands the standard definition of a 2-3 (the one given in class, with all the values in the leaves); all you have to describe are the modifications that need to be made.

Note that we want a **single** (compound) data structure that supports **all** these operations, not different data structures for each operation.

- `add(x)` : Add element  $x$  to the set. Time:  $O(\log(n))$
- `delete(x)` : Delete element  $x$  from the set. Time:  $O(\log(n))$ .
- `element?(x)` : Is  $x$  in the set? Time:  $O(\log(n))$ .
- `index(i)` : Find the  $i$ th smallest element in the set. Time:  $O(\log(n))$ .
- `indexOf(x)` : Find the index of  $x$  in the set. Time:  $O(\log n)$ .
- `subrange(i, k)` : Return  $k$  elements in the set in sequence starting with the  $i$ th. For instance, `subrange(100,5)` should return a list of the 100th, 101st, 102nd, 103rd, and 104th smallest elements. Time:  $O(k + \log(n))$ .
- `min()` : Find the smallest element in the set. Time:  $O(1)$ .
- `max()` : Find the largest element in the set. Time:  $O(1)$ .
- `median()` : Find the median element in the set. For instance if there are 99 or 100 elements in the set, return the 50th. Time:  $O(1)$ .

Ans:

1. Augment the 2-3 tree to have an array or structure in the vertices which can the size of each of its subtree, then the leftmost and the rightmost leaf value the median and the smallest value of the second and third subtree.

Let the array be  $A[1..8]$

Where

- $A[1]$  = Size of the 1<sup>st</sup> subtree
- $A[2]$  = size of the 2<sup>nd</sup> subtree
- $A[3]$  = size of the 3<sup>rd</sup> subtree
- $A[4]$  = leftmost leaf value of the 1<sup>st</sup> subtree
- $A[5]$  = median i.e. the value of element at the position  $\text{ceil}(n/2)$
- $A[6]$  = rightmost leaf value of the 3<sup>rd</sup> subtree
- $A[7]$  = smallest value of the 2<sup>nd</sup> subtree
- $A[8]$  = smallest value of the 3<sup>rd</sup> subtree

Size of 1st subtree	Size of 2nd subtree	Size of 3rd subtree	Leftmost leaf value of the 1st subtree	Median	Rightmost leaf value of the 3rd subtree	Smallest value of the 2nd subtree	Smallest value of the 3rd subtree
---------------------	---------------------	---------------------	--	--------	---	-----------------------------------	-----------------------------------

2. Add(x): this follows the standard algorithm by comparing the smallest value of the 2nd subtree and the 3rd subtree if exists.
3. Delete(x): this also follows the standard algorithm by comparing the smallest value of the 2nd subtree and the 3rd subtree if exists
4. Element?(x): this also follows the standard algorithm by comparing the smallest value of the 2nd subtree and the 3rd subtree if exists
5. Index(i): Maintain a variable S to calculate the index. S is initialized to zero and follow the below algorithm.

Comparing the size of the subtree decide on which subtree to follow keeping a count of total nodes to the left to track the index number till we reach the leaf node.

```

Index(i){
    If ( i <= (root.A[1] + S))
        If ((i == (root.A[1] + S)) && (1st subtree is a leaf node))
            Return 1st subtree value.
        Endif.
        Tree = 1st subtree
        Index(i);
    Elseif(i <= (root.A[1] + root.A[2] + S))
        If ((i == (root.A[1] + root.A[2] + S)) && (2nd subtree is a leaf node))
            Return 2nd subtree value.
        Endif.
        Tree = 2nd subtree
        S = root.A[1] + S
        Index(i)
    Elseif((i <= (root.A[1] + root.A[2] + root.A[3] + S)) && (root.A[3] != 0))
        If((i == (root.A[1] + root.A[2] + root.A[3] + S)) && (3rd subtree is a
leaf node))
            Return 3rd subtree value.
        Endif.
        Tree = 3rd subtree
        S = root.A[1] + root.A[2] + S
        Index(i)
    Endif.
}

```

As at any time we are just traversing the height of the tree the worst case complexity would be  $O(\log n)$ .

6. Indexof (x): Maintain a variable S to calculate the index. S is initialized to zero and follow the below algorithm.

```

Indexof(x){
    If (x < root.A[7])
        If (1st subtree is a leaf node)
            Return S + 1;
        Endif.
        Tree = 1st subtree
        Indexof(x)
    Elseif ( x == root.A[7])
        S = S + root.A[1];
        Return S + 1;
    Elseif ( x < root.A[8])
        Tree = 2nd subtree
        S = S + root.A[1]
    Endif.
}

```

```

        Indexof(x)
    Elseif ( x == root.A[8])
        S = S + root.A[1] + root.A[2]
        Return S + 1;
    Else
        If (3rd subtree exists)
            Tree = 3rd subtree
            S = S + root.A[1] + root.A[2]
            Indexof(x)
        Else
            Return error.
        Endif.
    Endif
}

```

As at any time we are just traversing the height of the tree the worst case complexity would be  $O(\log n)$ .

7. Subrange(i,k):
  - a. Find the index(i) as per the above mentioned algorithm. ---  $O(\log n)$
  - b. Add the element at i and the next k elements to the right to a list and return the list. ---  $O(k)$
8. Min(): return the min value from the array at the root vertex i.e. root.A[4].
9. Max (): return the max value from the array at the root vertex i.e. root.A[6].
10. Median(): return the median from the array at the root vertex i.e. root.A[5].

## PROBLEM 2

Suppose that we have a hash table of size 23 and we insert the keys 10, 39, 4, 27, 17, 1, 62, 33, 48. Show the final state of the hash table, assuming we use:

- A. Chaining, with the hash function  $h(k) = k \bmod 23$ .
- B. Linear probing, with hash function  $h(k, i) = (k + i) \bmod 23$
- C. Double hashing, with hash function  $(k + i * (1 + k \bmod 17)) \bmod 23$




(Assume 0-based indexing in the hash table, and assume that i is initially 0 on the first probe and increases by 1 afterward.)

Ans: Chaining:

```

h(k) = k mod 23
h(10) = 10 mod 23 = 10
h(39) = 39 mod 23 = 16
h(4) = 4 mod 23 = 4
h(27) = 27 mod 23 = 4
h(17) = 17 mod 23 = 17
h(1) = 1 mod 23 = 1
h(62) = 62 mod 23 = 16
h(33) = 33 mod 23 = 10
h(48) = 48 mod 23 = 2

```

0		
1	1	
2	48	
3		
4	4	 27
5		
6		
7		
8		
9		
10	10	 33
11		
12		
13		
14		
15		
16	39	 62
17	17	
18		
19		
20		
21		
22		

Linear Probing:

$$h(k, i) = (k + i) \bmod 23$$

$$h(10, 0) = (10 + 0) \bmod 23 = 10$$

$$h(39, 0) = (39 + 0) \bmod 23 = 16$$

$$h(4, 0) = (4 + 0) \bmod 23 = 4$$

$$h(27, 0) = (27 + 0) \bmod 23 = 4$$

$$h(27, 1) = (27 + 1) \bmod 23 = 5$$

$$h(17, 0) = (17 + 0) \bmod 23 = 17$$

$$h(1, 0) = (1 + 0) \bmod 23 = 1$$

$$h(62, 0) = (62 + 0) \bmod 23 = 16$$

$$h(62, 1) = (62 + 1) \bmod 23 = 17$$

$$h(62, 2) = (62 + 2) \bmod 23 = 18$$

$$h(33, 0) = (33 + 0) \bmod 23 = 10$$

$$h(33, 1) = (33 + 1) \bmod 23 = 11$$

$$h(48, 0) = (48 + 0) \bmod 23 = 2$$

0	
1	1
2	48
3	
4	4
5	27
6	
7	
8	
9	
10	10
11	33
12	
13	
14	
15	
16	39
17	17
18	62
19	
20	
21	
22	

Double Hashing:

$$h(k, i) = (k + i * (1 + k \bmod 17)) \bmod 23$$

$$h(10, 0) = (10 + 0 * (1 + 10 \bmod 17)) \bmod 23 = 10$$

$$h(39, 0) = (39 + 0 * (1 + 39 \bmod 17)) \bmod 23 = 16$$

$$h(4, 0) = (4 + 0 * (1 + 4 \bmod 17)) \bmod 23 = 4$$

$$h(27, 0) = (27 + 0 * (1 + 27 \bmod 17)) \bmod 23 = 4$$

$$h(27, 1) = (27 + 1 * (1 + 27 \bmod 17)) \bmod 23 = 38 \bmod 23 = 15$$

$$h(17, 0) = (17 + 0 * (1 + 17 \bmod 17)) \bmod 23 = 17$$

$$h(1, 0) = (1 + 0 * (1 + 1 \bmod 17)) \bmod 23 = 1$$

$$h(62, 0) = (62 + 0 * (1 + 62 \bmod 17)) \bmod 23 = 16$$

$$h(62, 1) = (62 + 1 * (1 + 62 \bmod 17)) \bmod 23 = 74 \bmod 23 = 5$$

$$h(33, 0) = (33 + 0 * (1 + 33 \bmod 17)) \bmod 23 = 10$$

$$h(33, 1) = (33 + 1 * (1 + 33 \bmod 17)) \bmod 23 = 50 \bmod 23 = 4$$

$$h(33, 2) = (33 + 2 * (1 + 33 \bmod 17)) \bmod 23 = 67 \bmod 23 = 21$$

$$h(48, 0) = (48 + 0 * (1 + 62 \bmod 17)) \bmod 23 = 2$$

0	
1	1
2	48
3	
4	4
5	62
6	
7	
8	
9	
10	10
11	
12	
13	
14	
15	27
16	39
17	17
18	
19	
20	
21	33
22	

### PROBLEM 3

Suppose that you have a set of  $n$  large, orderable, objects, each of size  $q$ , so that it requires time  $\Theta(q)$  to time to compute a hash function  $h(x)$  for any object. Describe a compound data structure, built out of a heap and a hash table, that supports the following operations with the specified run times.

- $\text{elt}(x)$  — Is  $x$  an element of the set? Expected run time  $O(q)$ .
- $\text{add}(x)$  — Add  $x$  to the set. Expected run time  $O(q + \log n)$ .
- $\text{delete}(x)$  — Delete  $x$  from the set. Expected run time  $O(q + \log n)$ . Here, note that, in a heap, when you replace  $x$  by the last element  $y$ ,  $x$  may be either greater than  $y$  or less than  $y$  and your algorithm has to consider both cases.
- $\text{min}(x)$  — Return the minimum element in the set. Worst case run time  $O(1)$ .

Ans:

1. Store the objects in MINHEAP implemented as an array (let the array be  $A$ ) and then store the index of the array in a hash table (let the table be  $B$ ) as value and the object as key at index calculated by the hash function over the object.
2.  $\text{elt}(x)$  : Try to find the position of the object in the hash table, if the a key value same as

the object is found in the hash table at the identified index then take the value for the key from the table and return the value as  $A[B(h(x)).value]$ . Else return element not found if a null is found at the identified index or once the entire hash table is traversed using linear probing or double hashing and the object is not found. This procedure in worst case consumes time  $O(q)$  to calculate the hash functions on the object until the object or a null is found. Accessing element from the array is  $O(1)$ . So it is  $O(1 + q) = O(q)$ .

```

Elt(x){
    Do{
        I = h(x)
    }
    while((B[I].key != NULL) && (B[I].key != x)

    If(B[I].key == x)
        Return A[B[I].value]
    Else
        Return Not found
}

```

3. add(x):
  - a. Check if the element already exists. -----  $O(q)$
  - b. if not found, add the element to the heap -----  $O(\log n)$
  - c. Update the hash table with the index of the object in the array of heap. -----  $O(q)$
  - d. Total worst case time complexity would be  $O(q + \log n + q) = O(q + \log n)$

```

Add(x){
    Elt(x)
    If (Not found)
        ADDHEAP x to A
        Do{
            I = h(x)
        }
        While(B[I] != NULL){ }
        B[I].key = x
        B[I].value = index of x in A
    }
}

```

4. Delete(x):
  - a. Check if the element already exists. -----  $O(q)$
  - b. if found, delete the element from the heap. -----  $O(\log n)$ 
    - i. if the element has no children delete it.
    - ii. If the element has one child replace the element with its child
      - Update the index in the hash table. ---  $\Theta(q)$
    - iii. If the element has two children replace the element with the smaller child.
      - Update the index in the hash table ----  $\Theta(q)$
    - iv. Continue the above process going down the tree if the children also have more children till the tree is in MINHEAP form.
  - c. Calculate the hash function for x and set the key to NULL. -----  $O(q)$
  - d. Total worst case time complexity would be  $O(q + \log n + q) = O(q + \log n)$

```

Delete(x){
    Elt(x)
    If(found)
        DeleteHeap x from A
        Do{
            I = h(x)
        }while(B[I].key != x)
}

```

```

        B[I].key = NULL
    }

```

```

5. min():
    min(){
        return A[1];
    }

```

Total worst case time complexity would be  $O(1)$ .

#### PROBLEM 4

(Modified version of problem set 2, problem 3). Consider the following problem. The input is a sequence of  $n$  records with an integer key  $x$ .key. You wish to sort the records in increasing order by key. The value of the key has many duplicates, so that the number  $w$  of distinct integers among the keys is much smaller than  $n$ . In fact, assume that  $w \log w \ll n$ .

A. Describe an algorithm to sort the sequence that runs in worst case time  $O(n \log w)$ .

Ans:

1. Iterate through the elements and store the distinct elements in a B-Tree with two values in the leaf key and value. Key being the element's value and value being the frequency of the element in the input sequence. This can be done in  $O(n \log w)$ . As worst case insertion to B-Tree would be  $O(\log w)$ .
2. As the leaf node of B-Tree are already sorted we would extract the leaf nodes and expand it in an array using the values representing the frequency of each element. -----  $O(n + w \log w)$ .
3. Total worst case time complexity would be  $O(n \log w + n + w \log w) = O(n \log w)$

B. Describe an algorithm to sort the sequence that runs in expected time  $O(n)$ .

Ans: Use an algorithm similar to bin sort.

1. Traverse through all the elements.
2. Create a list (bucket) for each distinct element and add the subsequent duplicates to the end of the corresponding list. –  $O(n)$
3. Reorder the buckets in increasing order of its value. We can use merge sort for this. ---  $O(w \log w + n)$
4. Traverse the element from the start of the bucket with the smallest value to end of the bucket with the largest value. ----  $O(n)$

Total worst case time complexity would be  $O(n + w \log w + n + n) = O(n)$ .