# Homework 3 Solution-Komal Wavhal

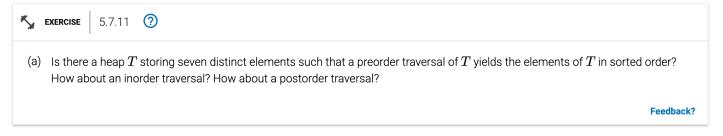
Exercises Chapter 5: 5.7.11, 5.7.24, 5.7.28 Exercises Chapter 6: 6.7.13, 6.7.17, 6.7.25

#### **Table of Contents**

| Homework 3 Solution-Komal Wavhal            |   |
|---|---|
| Exercises Chapter 6: 6.7.13, 6.7.17, 6.7.25 |   |
| Chapter 5                                   | 1 |
| 5.7.11                                      | 1 |
| 5.7.24                                      |   |
| 5.7.28                                      | 4 |
| Chapter 6                                   | 6 |
| 6.7.13                                      | 6 |
| 6.7.17                                      |   |
| 6.7.25                                      |   |

## Chapter 5

5.7.11

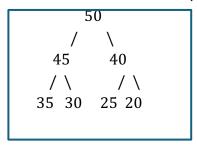


## **Solution**

#### **Preorder Traversal:**

The Preorder Traversal follows the sequence: Root, Left Child, Right Child.

• Preorder Traversal: 50, 45, 40, 35, 30, 25, 20



In this case, the Preorder Traversal starts at the root (50) and moves down the tree. Since the root is always the largest element in a Max-Heap, and the traversal visits the root before its children, the output is not in sorted order.

#### **Inorder Traversal:**

The Inorder Traversal follows the sequence: **Left Child, Root, Right Child.** 

• Inorder Traversal: 40, 45, 35, 50, 25, 30, 20

In a Max-Heap, the parent node is always greater than its children, so in the Inorder Traversal, the elements are not sorted. In fact, the result is a mixed order due to the structure of the heap. Hence, an Inorder Traversal does not produce a sorted sequence for a heap.

#### **Postorder Traversal:**

The Postorder Traversal follows the sequence: **Left Child, Right Child, Root**.

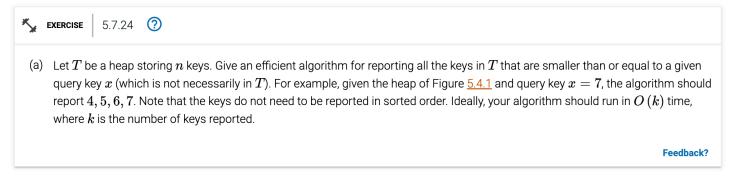
• Postorder Traversal: 40, 35, 45, 25, 20, 30, 50

In a Max-Heap, where the parent node is greater than its children, the Postorder Traversal also does not result in sorted order. The traversal goes down to the leaf nodes first, then visits the parents, but the heap property does not guarantee a sorted sequence.

#### **Conclusion:**

- Preorder Traversal: Does not yield sorted order.
- **Inorder Traversal**: Does not yield sorted order.
- Postorder Traversal: Does not yield sorted order.

#### 5.7.24



## **Solution**

#### **Problem Breakdown:**

- 1. **Heap Properties**: A binary heap is a complete binary tree where each parent node is less than or equal to its children (in a min-heap). The minimum element is always at the root.
- 2. **Efficient Reporting**: The algorithm should identify all nodes in the heap whose keys are less than or equal to the query key X. Once the root is greater than X, we know the entire subtree rooted at that node will contain only larger keys (in a min-heap), so we can skip that subtree entirely.

### Approach:

• **Start from the root**: Since the heap is a min-heap, the root always contains the smallest key.

- **Compare with the query key**: If the root's key is less than or equal to the query key, report it.
- **Recursively traverse**: Recursively traverse the left and right subtrees of the current node if the key is valid (less than or equal to the query key).
- **Terminate early**: If a node has a key greater than the query key, we can stop exploring its children since all keys in that subtree will be larger than the query key.

## Algorithm:

The Minimum key in the tree is the root. Every time, we compare the target and the root value. If the query key is larger than the key in root, we pop out and report the key in the root. The loop ends either when there are no more nodes in the tree or the key in root is larger than the query key. The heap will self-reconstruct and make sure the minimum value is in the root.

Here, the ideal case is that each node has only right child in the tree. Therefore, every time after t the key to the root pops, the right child of the root should be a new root in the tree, and this action takes O(1) time.

Since we assume that we have k report numbers, the running time should be O(k).

### **Time Complexity:**

• Worst-case time complexity: O(n) where n is the number of nodes in the heap. This is because in the worst case, we may have to explore all nodes in the heap to find those that satisfy the condition key≤z.

## **Space Complexity:**

• **Space complexity**: O(h) where h is the height of the tree, due to the recursive call stack.

```
print_at max(key z, tree_node_n)
Input: key z, tree_node_n(!)
Output: The keys of all nodes of the subtree rooted at n with keys at most z.

class Node
{
    int data;
    Node left, right;

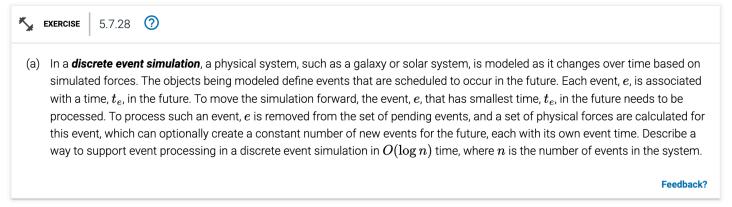
    public Node(int item)
    {
        data = item;
        left = right- null;
    }

    public class heap_sort_key
    {
        Node root;
        boolean isHeap (Node node, min_val, max val)
    }
}
```

```
if (node=NULL)
    return true;

else (n.get_Key()<=z)
{
    println(n.get_key();
    print_at_max(z, n.getLeftChild();
    print_at_max(z, n.getRightChild();
}
</pre>
```

### 5.7.28



## **Solution**

Here, we have a set of events for their future times 't<sub>e</sub>'. In each step, we need to extract the event with minimum 't<sub>e</sub>' and proceed with that event and add other finite numbers of events associated with our extracted event.

We can do this by finding the minimum 'te' event in each step which will be an O(n) operation in each step.

Here, an optimal solution will be to use priority queues. We make a min-priority queue of our event 'te' times. A min-priority queue is a queue in which the minimum element is at the front of the queue. In this way, we must just extract the first element of the queue in each step, and we are done.

Complexity and Runtime of the solution:

In a priority queue:

- Insertion = log(n)
   So, when a new element is added to the queue, it will take log(n) time to insert it.
- 2. Accessing the front element = O(1). It just need to access the first element of the queue which is O(1).

So, it maintains a min-priority queue. In each step, access the first element of the queue. Add the new elements in the priority queue.

Minimum priority queues can be easily implemented using minimum heap.

Pseudo Code:

```
#Function called by min-heap function to build the heap.
def min heapify (Arr[], i, N)
      left = 2*i;
      right = 2*i+1;
      smallest;
      if left <= N and Arr[left] < Arr[i]
         smallest = left
      else
        smallest = i
      END If
      if right <= N and Arr[right] < Arr[smallest]
        smallest = right
      END If
      if smallest != i
        swap (Arr[ i ], Arr[ smallest ])
        min heapify (Arr, smallest,N)
      END If
End
#function to build min-heap
def min-heap(Arr[], N)
      i = N/2
      for i : N/2 to 1 in steps of 1
        min heapify (Arr, i);
End
#function to extract minimum time
def min extract(A[])
return A [0]
END
#functions to insert new elements
def insertion (Arr[], value)
  length = length + 1
  Arr[length] = -1
  value increase (Arr, length, val)
def value_increase(Arr [ ], length, val)
      if val < Arr[i]
```

```
return
END If
Arr[ i ] = val;
while i > 1 and Arr[ i/2 ] < Arr[ i ]

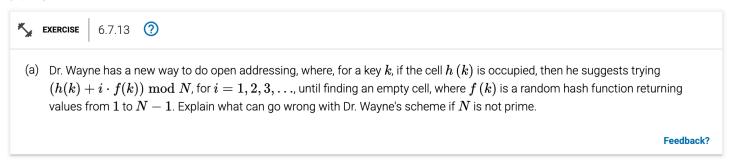
swap|(Arr[ i/2 ], Arr[ i ]);
i = i/2;
END While
```

#### **END**

Runtime of this algorithm is O(log n).

## Chapter 6

6.7.13



### **Solution**

In Dr. Wayne strategy of open addressing for a key k, if h(k) is occupied then try search  $(h(k) + i * f(k)) \mod N$  cell where i=1,2,3... and f(k) returns a random number from 1 to N-1.

For example:

Let N = 10 and f(k) produces 5 each time, then according to these values it always shows values h(k) + 0 or h(k) + 5 cells.

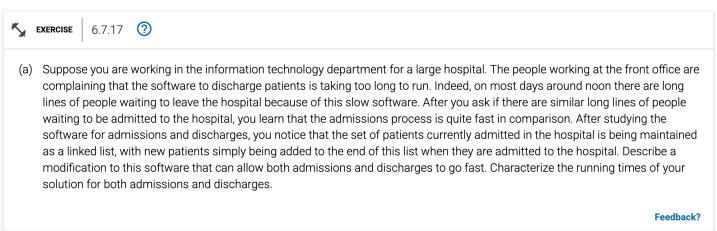
```
20 \mod 5 = 0
40 \mod 5 = 0
60 \mod 5 = 0
80 \mod 5 = 0
100 \mod 5 = 0
```

Even though 5 is a prime number, all the keys are multiples of 5 and thus the mod always be 0. Similarly, this will happen with any value which is a multiple of a number. This type of distribution is not good as it will form collision even after space is left in the bucket. So, to avoid such conditions we should use 'N' as a prime number (usually large numbers) to allow probing of all cells.

Prime numbers are used to neutralize the effect of patterns in the keys in the distribution of collisions of a hash function.

According to the question f(k) is random hash function, it will be a good hash function when it never evaluates to zero which can be possible by selecting prime numbers. And a common choice for f(k) can be q - (k mod q), for some prime number q < N.

#### 6.7.17



#### **Solution**

As per the question, the software computes the admission process of patients faster by using a linked list as the new patient are just being added to the end of the list which takes O(1) time. While discharging a patient takes longer as it traverses the complete list until it finds that specific patient, which will take O(n) time.

To solve this, we can use 'Hash Map' data structure which maps keys to values. We take 'm' as the map data structure, 'k' as the key and 'v' as the value mapped to the key. We can use the below methods for solving the problem:

- 1. addPatient(k, v):
  - a. Check if the patient exists on file by hash map lookup.
  - b. If the patient doesn't exist, insert a patient no. with value v and associate a key k with it, so that it can be added at the last index of array and added to the lookup table.
- 2. dischargePatient(k):
  - a. Check if the patient exists on file by hash map lookup.
  - b. If the patient exists in the hash map, remove value with key which matches in k. This way, the discharge of a patient will also remove the index key from array.
- 3. searchPatient(k):

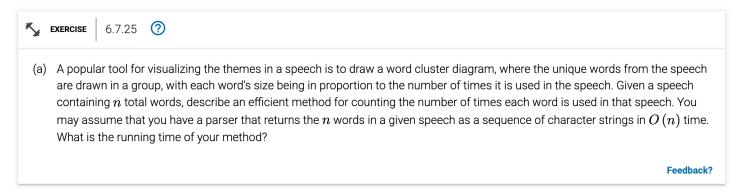
Check if patient exists on file by hash map lookup.

Time Complexity:

addPatient(k, v) and dischargePatient(k) operations can be done in O(1) time as in the lookup table of Hash Map each of the essential methods run in O(1) time only.

Rarely, if patients are hashed to the same key, rehash operation must be done. Rehash operation is of O(n) and will happen after n/2 operations and these are all assumption of O(1)

#### 6.7.25



Describe how to implement a stack using two queues. What is the running time of the push() and pop() methods in this case?

#### **Solution**

The efficient method for counting the number of times each word is used in the speech containing n total words is Cuckoo Hashing.

Cuckoo Hashing uses two lookups' tables  $T_0$  and  $T_1$ , each of size N, where N is greater than n. n is the number of items in the map. For any key k, there are two possible places where an item can be stored with key k,  $T_0$   $[h_0(k)]$ ,  $T_1$   $[h_1(k)]$ .

All insertion(put), removal(remove) and search(get) operations are done in O(1) time in worst case.

If collision occurs in the insertion operation, then evict the previous item in the cell and insert a new one in its place. Then evicted item go to its alternate location in other table and inserted there which may repeat the eviction process with another item and so on. But this may cause looping which can be overcome using rehash the keys in the table.

Words can be added in both the tables where words are the key, and their frequencies will be stored a value in hash table.

For counting total words n, when each word is used in speech, it will take O(n) time.