CS 600 Advanced Algorithms

Sri Dhanush Reddy Kondapalli

CWID: 10476150

Homework 3

1 R-5.7.11 Is there a heap storing seven distinct elements such that a preorder traversal of yields the elements of in sorted order? How about an inorder traversal? How about a postorder traversal?

Consider a heap T with seven unique elements, 100, 200, 300, 400, 500, 600, 700.

When we do a preorder traversal on the heap T, the items of T are returned in sorted order, i.e., your return result for such a traversal will be in sorted order, 100, 200, 300, 400, 500, 600, 700.

For **in-order traversal**, in the order as mentioned → 400, 200, 600, 100, 200, 500, 700
For **post-order traversal**, in the order as mentioned → 700, 300, 600, 100, 200, 400, 500

2 C-5.7.24 Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T).

- The minimal key is found at the tree's root. That is, for each comparison of your desired value to the root value.
- When the target key $>$ the root key, we use pop() and report the key in the root.

- To end the loop, either confirm that there are no nodes or that root key > target key.
- During heap self-restructuring, it will keep the root value as low as possible.
- In an ideal world where every node has only one correct child, every pop operation will replace the root of the tree with the right child of the root. This occurs in time complexity O(1).

Algorithm & Pseudocode:

```
Arr - array for the heap
Ind - index of the element
q_Key - query key
Find_key(Arr, ind, q_Key):
    If ind not valid, then
        Return
    If Arr[ind] < q_Key, then
        Print(Arr[ind])

    Else:
        Find_key(Arr, leftChild(ind), q_Key)
        Find_key(Arr, rightChild(ind), q_Key)
    Return
```

3 A-5.7.28 Describe a way to support event processing in a discrete event simulation in $O(\log n)$ time, where n is the number of events in the system.

A heap-based priority queue is a data structure that enables this. This is because inserting into this data structure requires $O(\log n)$ time in the worst-case scenario. In this case, the heap keys will be represented by t_e, the time for each event. Because of the implementation's time of $O(\log n)$, new events are handled rapidly to allow the simulation to continue.

Begin with scheduling events into the future, t_e. We have a list of events for their future timings on here. At each level, we obtain the event with the lowest t_e and proceed with it, adding a finite number of events linked.

Finding the smallest t_e at each stage is one possible solution. Each step will have an O(n) operation cost.

Priority queues appear to be the best approach. Assume there is a minimum-priority queue for t_e times. A min priority queue is one in which the smallest element is at the top of the queue.

Analyzing the complexity:
For our priority queue:

- Insertion $\rightarrow O(\log n)$: Which means to add a new element to the queue, it would cost $\log(n)$ time.
- Accessing front element $\rightarrow$ O(1): Which means to access the first element, it would cost O(1) time.

At each level, the first element in the queue is removed, and new elements are added to the priority queue.
Minimum priority queues are simple to construct using minimum heap.

4 C-6.7.13 Dr. Wayne has a new way to do open addressing, where, for a key k, if the cell h(k) is occupied, then he suggests trying $(h(k) + i.f(k))modN$, for i = 1,2,3,..., until finding an empty cell, where f(k) is a random hash function returning values from 1 to N-1. Explain what can go wrong with Dr. Wayne's scheme if N is not prime.

The data in an open addressed hash table is kept in the table itself rather than in buckets. Different ways of probing the hash table are used to resolve Hash Collisions.
Dr. Wayne suggests (h(k) + i . f(k)) mod N to probe the table where f(k) is a random hash function which returns value ranging 1 to N-1.
Ascertain that N is a prime number, since this will reduce collisions caused by repetitive patterns in a set of hash values.
Let's assume uniform hashing. The number of positions we can expect to probe in our address hash table is $\frac{1}{\alpha-1}$, where α is a load factor, defined by

$$\alpha = \frac{numberofelementsinthetable(n)}{numberofpositionsinthetable(m)}$$

In this case, all keys in the hash function that store a common factor with the number of buckets(m) are hashed to a bucket that is a multiple of our specified factor. This is further enhanced when we select 'N' as a prime.

5 A-6.7.17 Describe a modification to this software that can allow both admissions and discharges to go fast. Characterize the running times of your solution for both admissions and discharges.

Inserting a new patient into a linked list takes just O(1), whereas deleting a patient takes O(n). As a result, instead of a linked list, a Hash Map may be used to solve this problem.
When using a Hash Map, the patient admittance number may be used as the key and the other patient data as the value. Because finding an element in a Hash Set takes O(1) time, this would allow us to scale admission and deletion to O(1) time.
Collisions can be addressed by linear probing.

6 A-6.7.25 You may assume that you have a parser that returns the n words in a given speech as a sequence of character strings in O(n) time. What is the running time of your method?

Let's create a hash map with a polynomial hashing function. Here,
key = word
value = occurrences of the word in speech
Our tasks would be to traverse our data in time O(n), which would be followed by hashing/retrieving/storing data, which would cost us O(1). As a result, counting each word O(n) times and storing them in a hash map O(1) times will take a total of O(n) time.
A map tracking word frequencies might be an effective way to solve this problem. The map's organizing scheme may employ each word as a key and the frequency of that term as a value. As the speech is repeated and each word is visited, the frequency of that word is increased by one. This makes use of the map's get(k) and put(k,v) functions, which each require O(1) time. After cataloging all terms and frequencies, the map may be iterated through in O(n) time to obtain the results. This results in an algorithm with the time complexity:

$$O(n)(fromthegivenwordparser) + O(1)O(1) + O(n) = O(n)$$