Yash Avinash Patole

Course : CS600

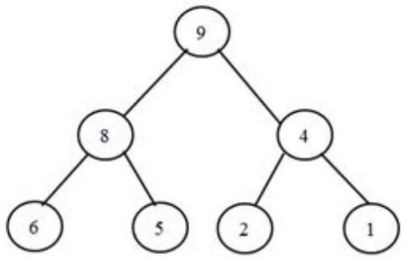
Homework 1

CWID: 10460520

Chapter 5

Exercise 5.7.11

Answer:

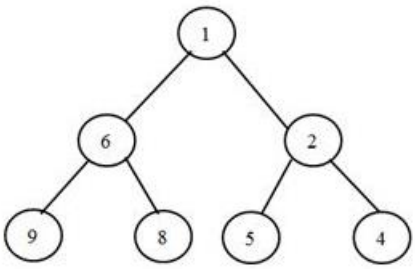


Heap T with 7 distinct elements

Pre-order traversal of T:
In a pre-order traversal, we visit the root, the left subtree, and then the right subtree in order.
So, it is possible to have an instance of the heap T.
Here the pre-order traversal of T yields the elements of T in sorted order.
Pre-order traversal of T: 9,8,6,5,4,2,1

In-order traversal of T:
In this traversal method, the left left-subtree is visited first, then root and then the right sub-tree.
In-order traversal of heap T: 6,8,5,9,2,4,1.

In a post-order traversal of T:
In a post-order traversal of T, we visit the left and right subtrees before visiting the root.
Post-order traversal of T: 6,5,8.2,4,1,9.



Exercise 5.7.24

Answer:

The minimal key in this problem is the root of the tree, thus each time we must compare our desired or goal value to the root value.
We do the pop out operation and report the key in the root if the desired key is greater than the key in the root.
We must verify that there are either no nodes at all or that the target key is greater in the root key to break the loop.
The heap will be able to self-restructure, allowing the root value to always be the lowest.
Every action that involves the root's correct child is always successful in the ideal scenario where each node in the tree has only the right children.
For any ideal case like, each node is having only right child in the tree, every opp out operation right child
of the root is always the new root of the tree. This gives us a 0(1) time.
We assumed that we must report k numbers, hence the running time should be O(k).
Algorithm 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.
PSEUDOCODE:

```
Class Node
{
int data;
Node 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 .getLeftChiId());
print _ at _max(z , n .getRightChiId()) ;
}
}
}
```

O(K) is the time for this algorithm, the reason for this is there is no node in tree which stores which is a key larger than z that has a descendent storing a key less than z.
For example:
sample mean (z)=15.11
hypothesized mean (uo) = 15
sample size (n) = 23
standard deviation(s) = 0.23
Therefore, t = (z-uo) * sqrt(n)/s=2.293658555.

Exercise 5.7.28

Answer:

Given Scenario:
Here, we have a sequence of occurrences in times 'te' in the future. Each step requires that we extract the event with the lowest "te" value, proceed with that event, and then add further events in a finite number that are related to our extracted event.
To do this, one strategy may be to identify the smallest "te" event in each phase. However, each step will be an O(n) operation.
Using priority queues is the best course of action. Let's create a min-priority queue using the "te" times for our events. The optimal solution is to use priority queues. Let us 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 have to just extract the first element of the queue in each step, and we are done.
Let us analyze the complexity 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-
Accessing the front element -> 0(1).
it just needs to access the first element of the queue which is 0(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)

END
```

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

Chapter 6

Exercise 6.7.13

Answer:

- Open addressed hash table stores data in the table itself instead of in buckets. These collisions are needed to be resolved.

Such Collisions are resolved by various methods of probing the table.

- Here Dr. Wayne Suggests that h(k)+i.f(k)mod N to prove the table where f(k) is a random hash f(n) returning the values from 1 to N-1
- The value of 'N' must be prime in order to reduce the number of collisions caused by repeated patterns in a set of hash values. Assuming uniform hashing, the number of positions we can expect to prove in an open-addressed hash table is 1/1-α where α is lead factor, defined by α = n/m
    - Where n ➔ number of elements in the table
    - M ➔ number of positions in the table

Therefore, every key is hash function that stores a common factor with the number of Buckets(m) will be hashed to a bucket that is multiple of this factor. This can be achieved when we choose 'N' to be a print number.

Exercise 6.7.17

Answer:

Currently, linked lists are being used to implement software: Because each item is added at the end of the list, hospital patient admission is quick. and upon discharging, it must locate the specific patient (in this case, item) and navigate through the list to locate the patient's information. Which task takes the longest to complete Due to the need to search through the entire list or until the specific patient's information is found That is the cause of the lengthy process.

The question iterates the software uses linked lists, which is the problem statement for the person discharging patients from the hospital.

Here is the problem statement.

1. Admission of the patients is fast because each entry is made at the end of the list.
2. Discharging however gives problem because he must find each patient in the list and needs to traverse the list to find the details.

This problem can be solved using heap function in the following way.

I - Admit Patient (Patient)

Step I: Check if the Patient exist by hash map lookup.

Step-2: if the Patient does not exist add the patient at the last index of the array.

Step 3: Add into hash table also, Patient is added as a Key and last index of Array as index.

2- Discharge (Patient)

Step-I: Check if Patient by using hash map lookup

Step-2: if Patient exist, get the index, and remove that index from the HashMap

Step-3: Swap the current patient Item with last patient item and remove the last Patient item- this is because it takes 0(1) time to remove the last item.

Step 4: update the index of last patient in the HashMap.

3- Search (Patient)

Lookup for the patient in HashMap

Time Complexity:

HashTab1e time complexity is O(1) in average and amortized time complexity. because it is very rare case that many patients will be hashed to the same key and Rehash operation Rehash operation is of O(n) and will happen after n/2 operations and these are all assumption of O(1).

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

Exercise 6.7.25

Answer:

Running time of following method => O(n)
because we are traversing each word from speech once and maintaining its count value.
Here, we want to find the frequency of each word in the speech. The suitable concept to use here is Python's Dictionaries ‘or associative array or hash since we need key-value pairs: where key is the word, and the value represents the frequency words appeared in the speech.
Assuming we have declared an empty dictionary frequency = { }
now we traverse through all n words: as we are travelling linearly so runtime is O(n)

```
import re
import string
frequency = {}
words = //given by parser that returns the n words in a given speech as a sequence of character strings
for word in words:
        count = frequency. get(wordß)
        frequency[word] = count + 1
frequency_list = frequency.keys()

for words in frequency_list:
        print words, frequency[words]
```