Homework 3

Atiq Patel CWID: 10432883

Screen%20Shot%202017-09-19%20at%201.24.33%20AM.png

**Solution:** Insertion sorts works in a way such that it uses comparison of the current element to the largest element in the sorted area of the list. If the largest element in the list is larger than the current element then it remains the way it is. Otherwise, the element e inserted in the place and the list gets reshuffled until we have the sorted list.

The worst case for insertion sort would be if the elements are in the reverse order.

Example: Given a list to be sorted in the increasing order: - 99,57,34,21,12,10,7,3.

In the above array, Each and every element would be moved at the first position and then back inside the array where it belongs and it goes on processing. There each element inside the array is being moved “n” times, if there are “n” number of elements placed in the decreasing order then it will that

“n2” times. Therefore, **the running time for insertion sort would be Ω (n2) for the worst cases.**

Screen%20Shot%202017-09-19%20at%201.24.42%20AM.png

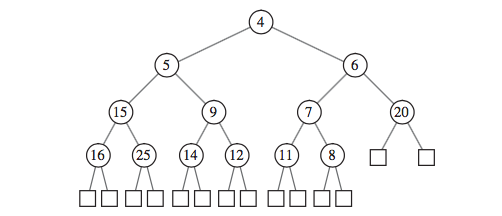


Figure 5.6

**Solution:** The steps for replacing 5 with 18 are as follows: -

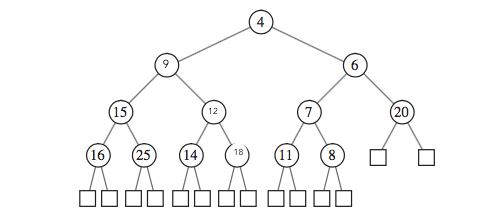
Step 1: Replace the node 5 with node 18.

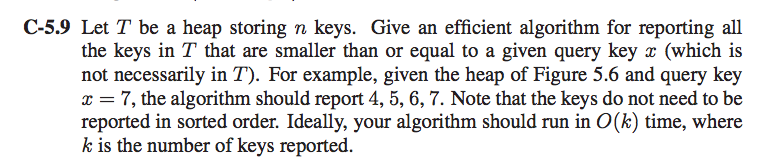
Step 2: Check the conditions for heap order property and restore it if not in order.

Step 3: In our case we will have to replace node 9 with node 18 as node 9 is the smallest children of node 18. Therefore node 9 becomes the parent of node 15 and node 18. We check it again to make sure the heap order satisfies.

Step 4: We can see that node 12 is the smallest children among node 14 and parent node 18 so we swap them again. Now we check again and confirm that node 18 is the smallest root among its children.

Thus, the heap order property exists and it is now a heap tree.





**Solution:**

In a heap tree, we know that the parent node is the smallest node. So we are only interested to find the node which are smaller than x and disregard the nodes that are larger than x as they do not meet the requirements.

Algorithm FindMax(T[]):

Input: The key element x and the heap T.

Output: The nodes with the key values less than x.

While(T[key]>=x) **do**

Display(T[key])

FindMax(T[key.left])

FindMax(A[key.right])

Return;

This algorithm will run recursively:

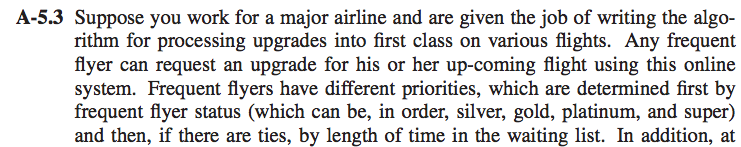
Step 1: Check whether the key is greater than equal to x if not then return.

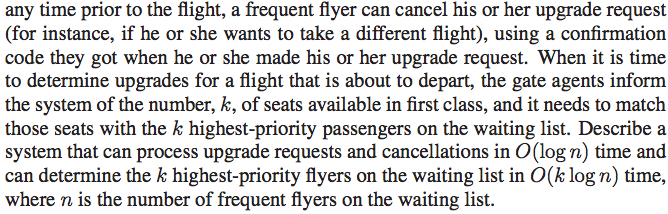
Step 2: Display the value of this node.

Step 3: Search the left children of the node.

Step 4: Search the right children of the node.

The algorithm runs only for the nodes which are only less than key x.It only runs into keys k number of searches, Thus the running time for the algorithm is O(k), because none of node which is inside T has a key that is bigger than x.





**Solution:** One of the ways we can implement this is by using a Priority Queue as it will be able to handle upgrades and cancellations in O(log n) and can determine k highest-priority passengers on the waiting list in O(log n) run time where n is the number of frequent flyers that on the waiting list. It can be implemented using the Binary Heap which uses insert(k,v), removemin(),min() in O(log n) time.

**Algorithm Upgrade(k):**

**Input:** Frequent Flyer number k and a binary Heap A.

**Output:** Seat upgrade of the passenger k.

**while** k > 1 **&** A[k/2] > A[k] **do**

Swap A[k/2] and A[k]

k🡨[k/2]

**Algorithm Cancellation():**

**Input:** None.

**Output:** A Binary Heap A with updated information of k passengers.

cancelledp 🡨 A[1]

A[1]🡨 A[k]

k🡨k-1

i🡨1

**while** i<k **do**

**if** 2i+1 <=k **then**

**if** A[i] <= A[2i] & A[2i+1] **then**

return cancelledp

**else**

Let j be the index of the smaller of A[2i] and A[2i+1]

Swap A[i] and A[j]

I🡨j

**else**

**if** 2i<=k & **then**

**if** A[i]> A[2i] **then**

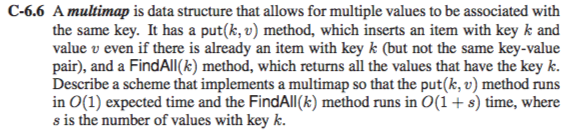
Swap A[i] and A[2i]

Return cancelledp

Return cancelledp

The run time for both the algorithms are **O(log n)**.

**Searching:** A heap search takes O(log n) time where n are the number of Frequent flyer passengers that are present on the waiting list. K list of passengers can be located in at the most worst case time would be O(k log n).



**Solution:** A multimap data allows for multiple values to be associated with the same key.

The two methods are FindAll(k) and put(k,v)

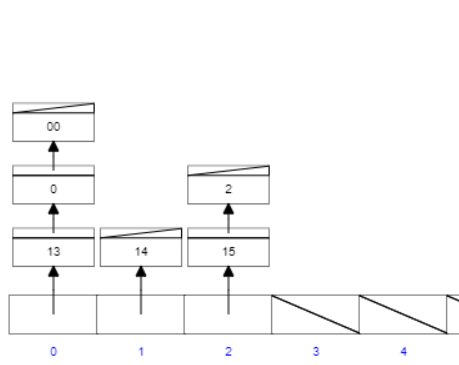
Put(k,v): It inserts the value v inside the key k.

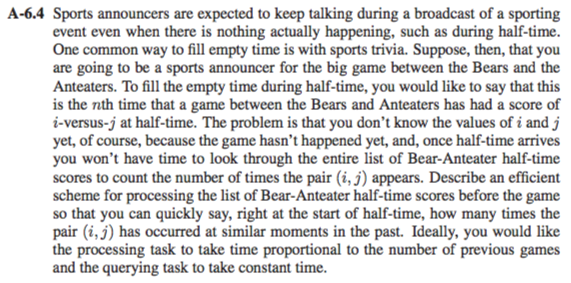
FindAll(k): Returns all the values of key k.

This can be accomplished using Hash Table.

In a Hash Table, all the data is put in such a way that the elements having same key k yet different values would be clubbed together as seen in the diagram.

As we can see from the diagram below, that all the values are hashed using a single value inside the table. So if we wish to use put(k,v), the run time would be O(1) insertion in the table where k is hashed and the FindAll(k) method would run in O(1+s) time where s is the number of values with key k.





**Solution:** Let I vs j represent the score between the two during half time.

We can solve this problem by using Hash Table with slight modifications. The key instead of a regular number would be carrying scores of both the teams and the value v would indicate the number of occurrences of the score during the half between these two teams.

Here we can make use of the hash function put(k,v) where k = (i,j) which basically represents the scores between these two team wherein “i” is the score of Bears and and “j” being the score of Anteater and v being the number of occurrence of the score during the half between two teams.

Owing up to these, there can be two conditions,

When there is a new score, then we insert the score i,j as key k =(i-j) and value being 1.

When the score has occurred in the past, which means that there is already a same key present in the table which leads to collision, we handle that by increasing the value of the same key by one.

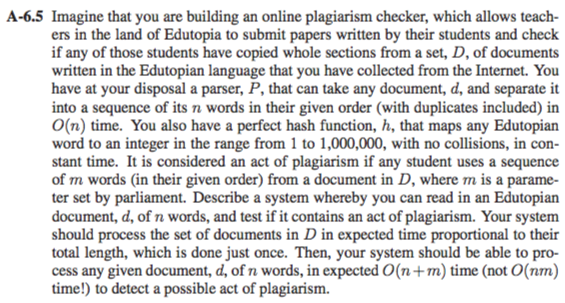
Example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Match Number | 1 | 2 | 3 | 4 | 5 | 6 |
| Key(i-j): | (1-3) | (2-4) | (2-0) | (2-3) | (1-1) | (0-0) |
| Value(Occurrence): | 2 | 4 | 2 | 1 | 3 | 5 |

So, for example Match 7 is taking place and the score is 0-0 then we will check if there already 0-0 taken place between the two. We can clearly see that it has (Match 6), so we increment the value of match 5 to 6.

The announcer can simple retrieve the list by comparing todays with their older games, if half time has taken place and the score between the two teams hasn’t ever been taken place so the person will insert the value into the table and put the value as 1 or if the score was already present, then the announcer would increment it. **The run time for this particular method is O(1).**

The time taken for processing the hash table which is basically parsing every single element would be O(n).



**Solution:** This problem can be solved using of a Hash Table of at least size N which double if the capacity reaches. We have a perfect hash function h that maps any word to an integer from 1 to 10,000,000 with no collisions and constant time. We start by mapping set of documents d of n words in the hash table which will take O(n) time because the parser must parse through every word. In case of same words, which is basically same key different value will be clubbed together like how “13,0 and 00” are clubbed in the diagram below using chaining which takes O(1) time. This would happen for all the words that are be repeated with same key and bounded together. When we see that the threshold “m” is achieved, the student has plagiarized. Since there “n” words and minimum sequence “m”. The parser P has to parse through n+m sequence which gives us the run time of **O(n+m)**.

