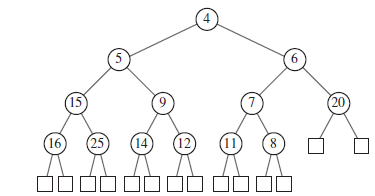
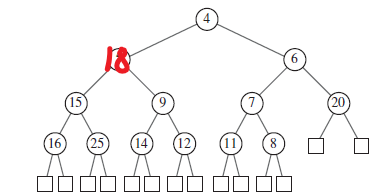
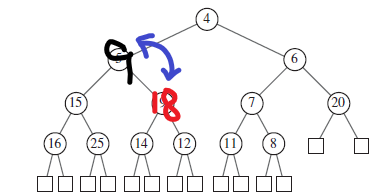
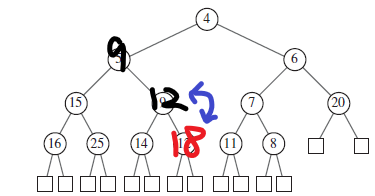
**R-5.6**

The worst case would be a list in descending order like [5,4,3,2,1]. Every time we insert an element, we need to get one from the list first ,then comparing it with each element in the new list, and finally put it to the end of the new list. The running time should be the sum of 1 to n = n \* ( n + 1) / 2 = (n^2 + n)/2 >= (n^2)/2. Thus, the running time is in Ω(n^2 ).

**R-5.14**



**C-5.9**

Using a recursive function to search the Node (thisNode, recFun(leftNode), recFun(rightNode)) starting from the root, and comparing the current node’s value with the key, returning the value if smaller or equal to key, and stops when tree is empty or node’s value is larger. Since a heap tree is already sorted we would search exactly O(k) time, where k is the number of keys returned.

**A-5.3**

We can use a double priority queue. The first one is sorted with silver, gold, platinum and super, and its value stores another priority queue sorted with waiting time, and each time corresponding to a name as such below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Q1(key) | Q1(value) -> Q2(key) which is time, Q(value) is names | | | | | |
| Super | 2(Ada) | 3(Dude 1) | 4(Dude 2) | 77() | 9() | 。。。 |
| Platinum | 22(Lady n) | 33(Lady x) | 44() | 。。。 |  |  |
| Gold | 1() | 2() | 3() | 。。。 |  |  |
| Silver | 5() | 6() | 88() | 99() | 。。。 |  |

Since priority is already sorted, an upgrade or cancellation would just cost Q(1) time, moreover, the highest priority person would be found in O(1) with looking from Super to Silver then return the end of the Q2 would return the highest class person with longest waiting time and thus has the highest priority. The k highest priority people would also cost O(k) with each person’s return time is O(1).

To make the upgrades and cancellations O(log n) and search k highest priority O(k log n), we can store time and name in a map or array with size 2. Then use class as key and time as value to form a tree in such we make sure super class is above all and so does the rest. Within Super class we then use heapsort to sort the time. Thus the time to get to each node would cost O(log n) for heap tree. Therefore the k highest priority would cost k times O(log n) which is O(k log n).

**C-6.6**

We could use a hashmap with key and value, where value is implemented as a linked list, an arraylist or just an array depending on the needs. To put an element we find the key in map and then insert value into a say arraylist with O(1). Similarly the findAll(k) need O(1) to return the address of list then it takes O(s) time to return the s elements within the list. Therefore the FindAll takes O(1+s) running time.

**A-6.4**

To make sure the efficiency is high enough we need to protect out search time to O(1), thus, we need cuckoo hashmap to store pairs. Cuckoo hashmap is not good at saving data in limited memory space but with only O(1) search time. Thus when we search the data and see how many score matches the current pair it would take least amount of time. For this case in cuckoo hashmap we store our current half-time score into the map as a key, if there is one existing key the we do value ++, else we create a new entry can save value = 1. After a search for pair we would be returned immediately with the value for that pair, and thus fulfill the need.

**A-6.5**

We use a polynomial hash function to process the data stored in a hash table. To maintain a length of m we can use a map D1 = (d1, d2, … dm) then for next element D1’(d2,d3,…dm+1) we can use f(D1’) = f(D1)\*c – d1\*c^(m-1) + dm+1 where c is constant in polynomial fun f. For each document we only need to go through once in O(n) time with an extra time for set D O(m). Thus the final time should be O(n+m). For each suspect of plagiarism we need extra O(m) time to see if it’s a case. Note that we only need to find one set of copy to determine plagiarism, so the running time is still O(n+m).