# CS-600-A Homework 3

Zijing Huang 10414952

**R-4.4 A certain Professor Amongus claims that the order in which a fixed set of elements is inserted into an AVL tree does not matter—the same tree results every time. Give a small example that proves Professor Amongus wrong.**

Hahaa, this Professor is so funny. He can just take a few minutes to go through a test case, and then he will find out the conclusion is incorrect. The test case could be as simple as {1, 2, 3, 4}. So I insert [1, 2, 3, 4] in the first time, the AVL tree is as following:



And then I reconstruct the AVL tree by the order [4, 3, 2, 1]:

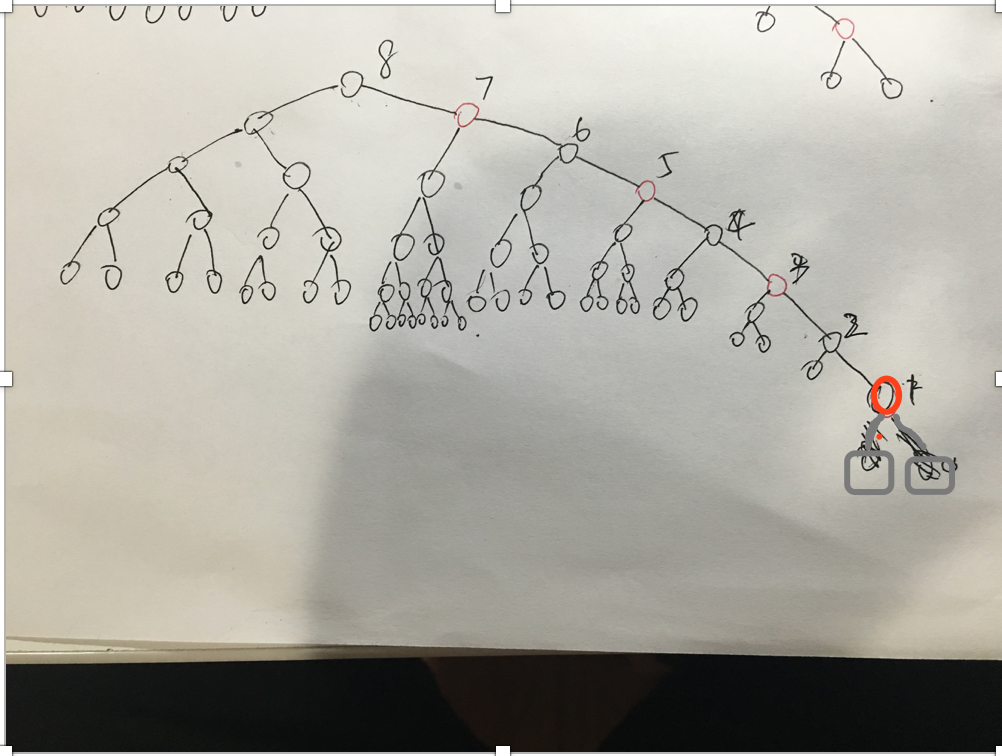


Obviously, they are not the same.

**R-4.7 What is the minimum number of nodes in a red-black tree of height 8?**

If we want to get the minimum number of nodes in a red-black tree of particular height, we would make the rightest path as many red nodes as possible so that each path will need as less black nodes and it reduces the total number of nodes.

So, the red-black tree can be built as following:



and the minimum number is 61.

**C-4.15 A *mergeable heap* supports operations insert(k,x), remove(k), unionWith(h), and min(), where the unionWith(h) operation performs a union of the mergeable heap h with the present one, destroying the old versions of both, and min() returns the element with minimum key. Describe a implementation of a mergeable heap that achieves O(log n) performance for all its operations. For simplicity, you may assume that all keys in existing mergeable heaps are distinct, although this is not strictly necessary.**

We can use Binomial heap as the data structure of mergeable heap.

### Insert

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes O(log *n*) time. However, across a series of *n* consecutive insertions, insert has an [*amortized* time](https://en.wikipedia.org/wiki/Amortized_time) of O(1) (i.e. constant).

### Delete

To delete an element from the heap, decrease its key to negative infinity (that is, some value lower than any element in the heap) and then delete the minimum in the heap.

### Merge

the simplest and most important operation is the merging of two binomial trees of the same order within a binomial heap. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree becomes a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps.

### Find minimum

To find the minimum element of the heap, find the minimum among the roots of the binomial trees. This can again be done easily in *O*(log *n*) time, as there are just *O*(log *n*) trees and hence roots to examine.

By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to *O*(1). The pointer must be updated when performing any operation other than Find minimum. This can be done in *O*(log *n*) without raising the running time of any operation.

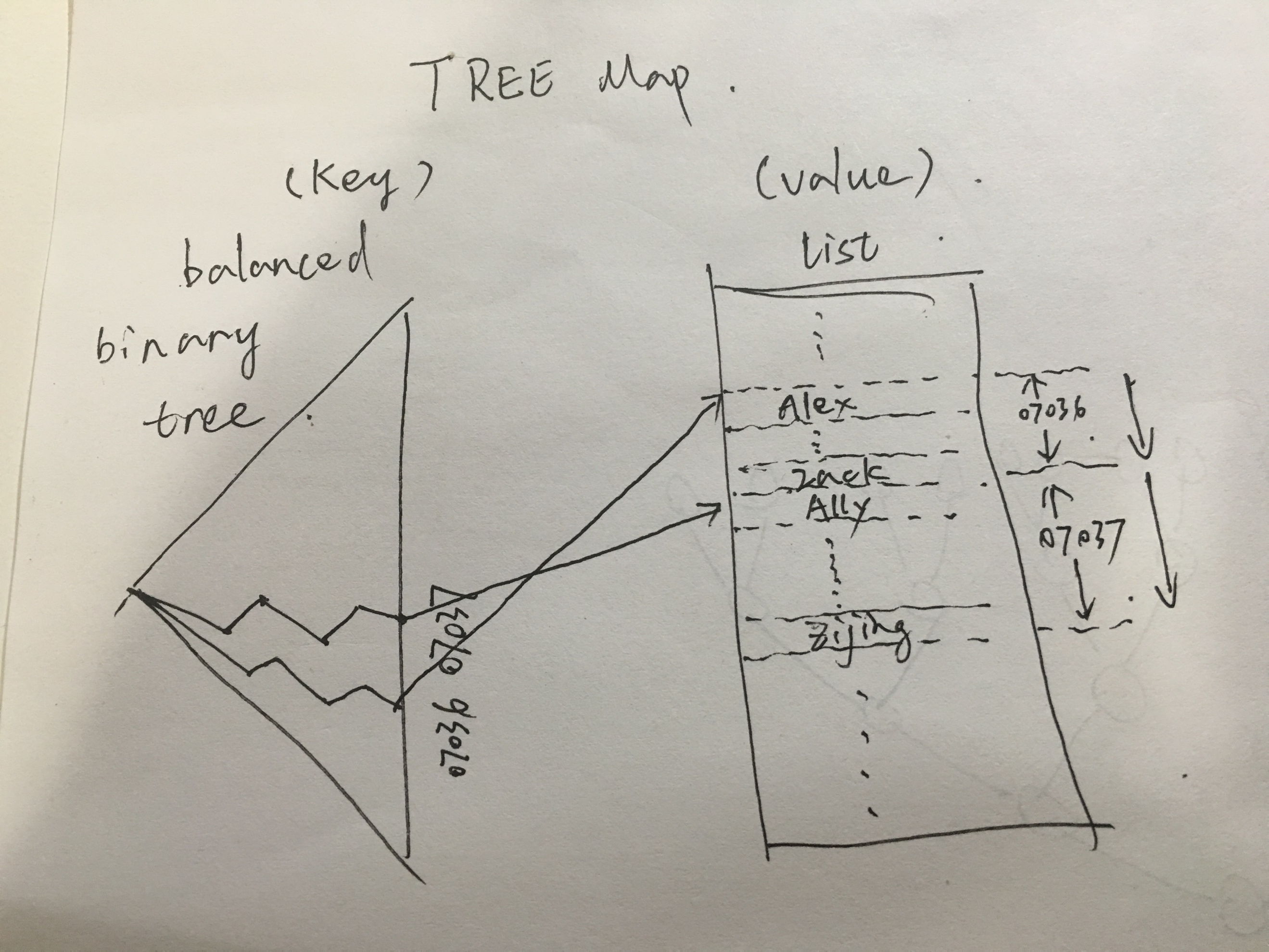
All the operation takes log(n) time complexity.

Reference by WIKI:

https://en.wikipedia.org/wiki/Binomial\_heap

**A-4.4 Suppose you are working for a victim-support group to build a website for maintaining a set, S, containing the names of all the registered sex offenders in a given area. The system should be able to list out the names of the people in S ordered by their Zip codes, and, within each Zip code, ordered alphabetically. It should also be able to list out the names of the people in S just for a given Zip code. The running time for a full listing should be O(n), where n is the number of people in S, and the running time for a listing for a given Zip code should be O(log n + s), where s is the number of names returned. Insertions and removals from S should run in O(log n) time. Describe a scheme for achieving these bounds.**

We can use a tree map implemented by a red-black tree or AVL tree as the diagram below:



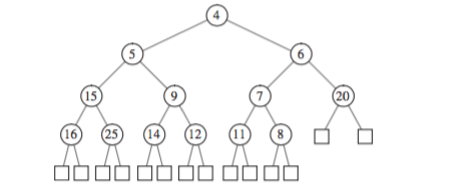
Base on the nature of red-black tree, the insertion and removal time should be O(logn), the running time for a full listing should be O(n) and the running time for a given Zip code should be O(logn + s), where s is the number of names returned. (We use logn time to get a particular leaf which is the search key. And then we do a constant search from where the search key is pointing to. )

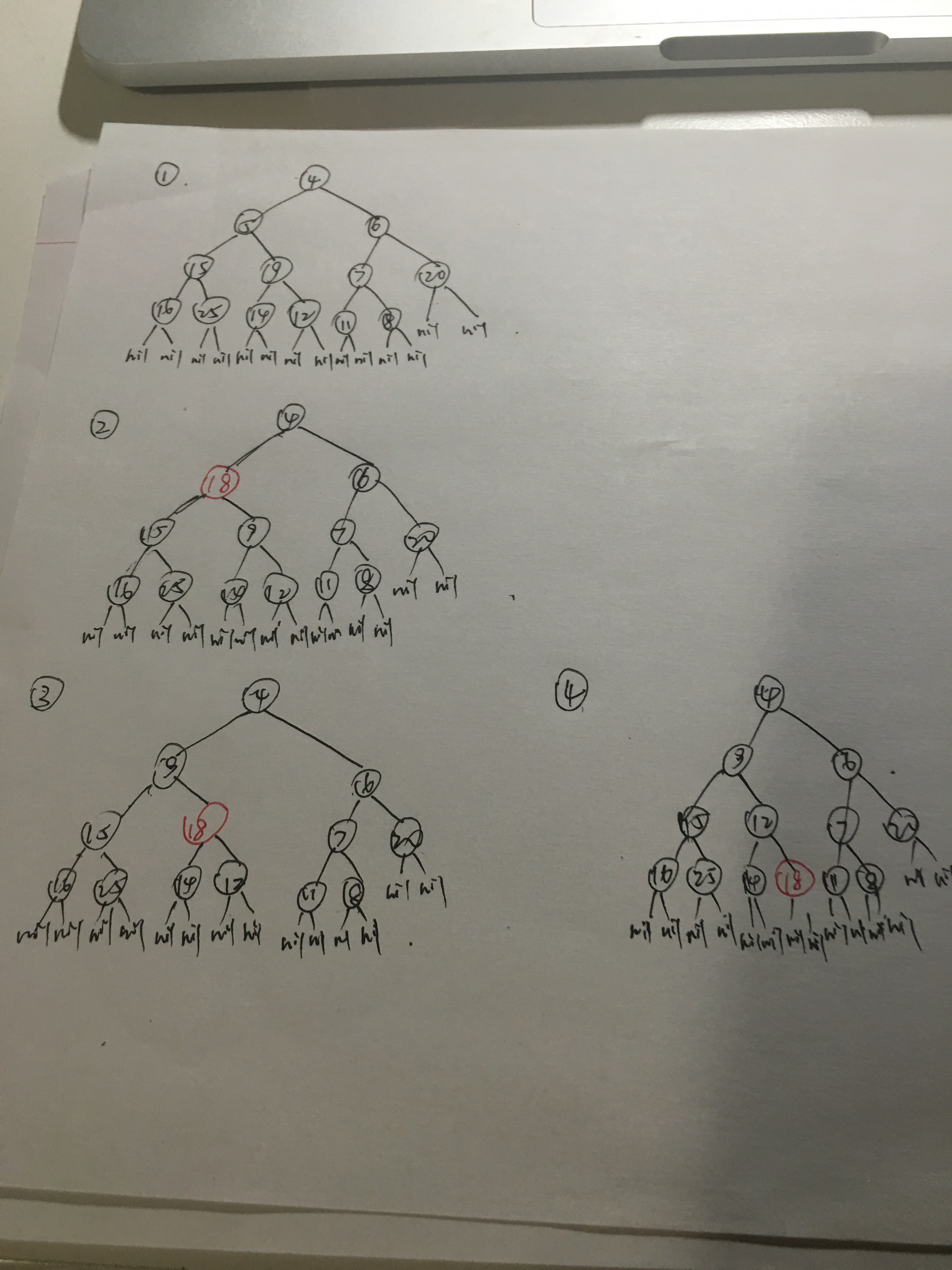
**R-5.6 Give an example of a worst-case list with n elements for insertion-sort, and show that insertion-sort runs in Ω(n2) time on such a list.**

The worst-case is when the list is in reverse(descending) order. For example, we are given a list: {5, 4, 3, 2, 1}. Because in each round, we fetch an element from the list, compare the element with each element in the new list, and put it in the end of it in this case. So, the running time f(n) would be 1+2+3….+n = (n2+n)/4. We choose c=1/4 and n0>=0 such that f(n) >= 1/4 \* n2. Therefore, f(n) is Ω(n2).

**R-5.14 Show the steps for replacing 5 with 18 in the heap of Figure 5.6.**

Figure 5.6 is as following:





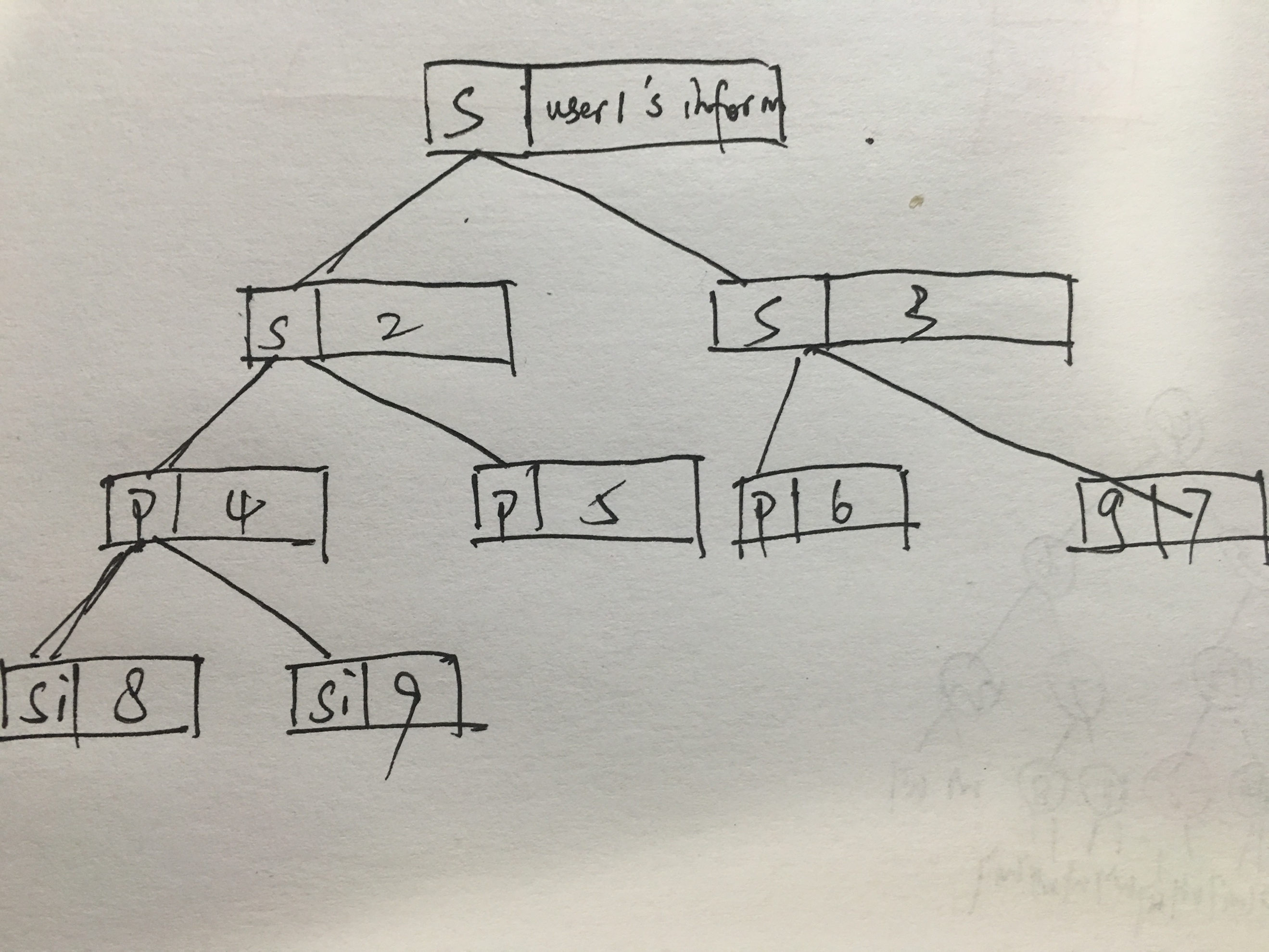
First of all, we replace 5 with 18. And then repeatedly, we replace 18 with the smaller child of its children nodes. Finally, we can get the new heap like figure 4.

**C-5.9 Let T be a heap storing n keys. 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 ). For example, given the heap of Figure 5.6 and query key x = 7, the algorithm should report 4, 5, 6, 7. Note that the keys do not need to be reported in sorted order. Ideally, your algorithm should run in O(k) time, where k is the number of keys reported.**

Since the minimum key in the tree is the key in the root. Every time we compare the target value and root value. If the query key is larger than the key in root, we pop out and report the key in the root. We end the loop either there are no more nodes in the tree or the key in root is larger than the query key. The heap will self-reconstruct that make sure the minimum value is in the root. The ideal case is that each node has only right child in the tree. Therefore every time after I pop the key of the root, 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).

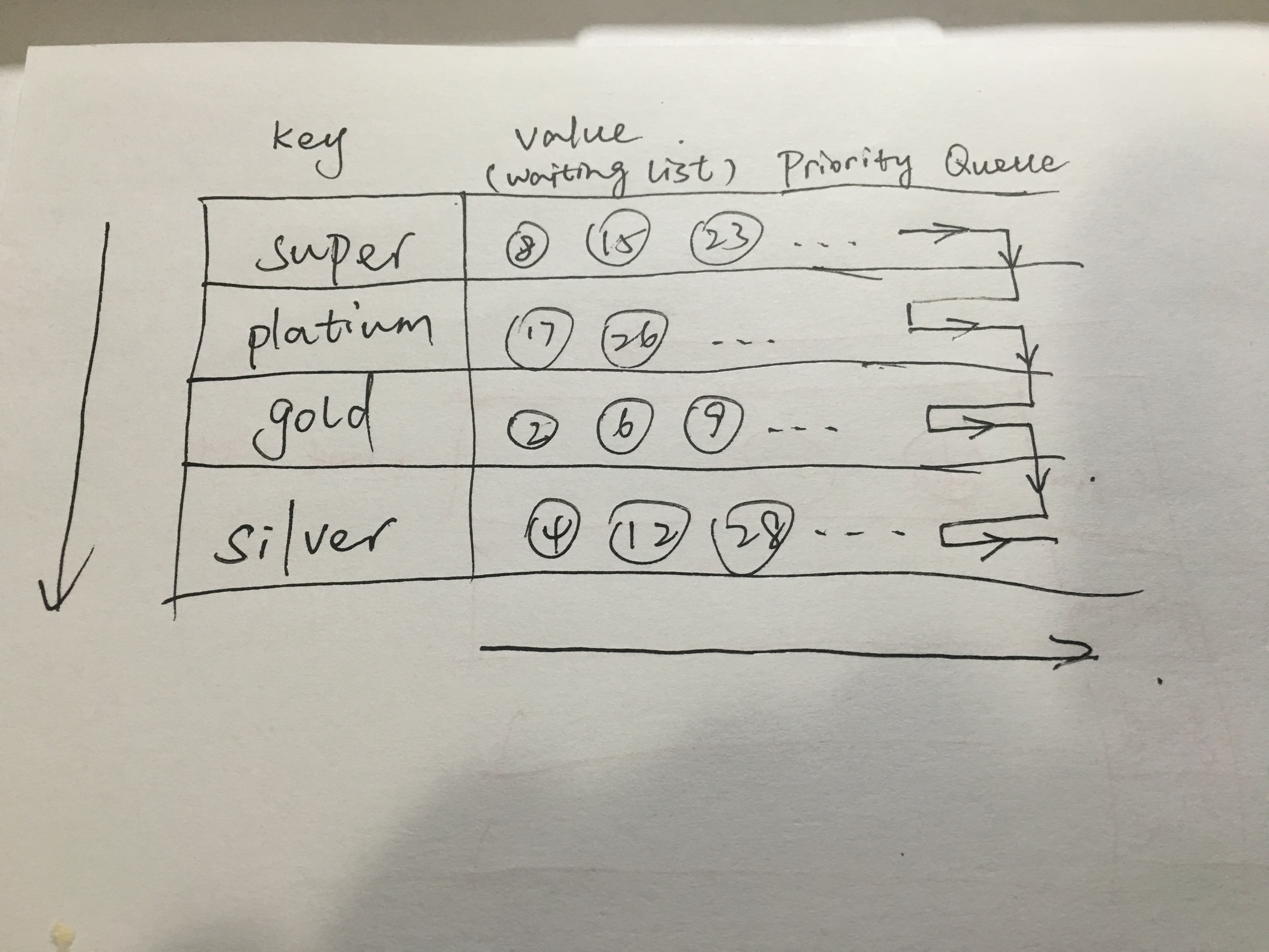
**A-5.3 Suppose you work for a major airline and are given the job of writing the algorithm for processing upgrades into first class on various flights. Any frequent flyer can request an upgrade for his or her upcoming flight using this online system. Frequent flyers have different priorities, which are determined first by frequent flyer status (which can be, in order, silver, gold, platinum, and super) and then, if there are ties, by length of time in the waiting list. In addition, at any time prior to the flight, a frequent flyer can cancel his or her upgrade request (for instance, if he or she wants to take a different flight), using a confirmation code they got when he or she made his or her upgrade request. When it is time to determine upgrades for a flight that is about to depart, the gate agents inform the system of the number, k, of seats available in first class, and it needs to match those seats with the k highest-priority passengers on the waiting list. Describe a system that can process upgrade requests and cancellations in O(log n) time and can determine the k highest-priority flyers on the waiting list in O(k log n) time, where n is the number of frequent flyers on the waiting list.**

We can use a Priory Queue implemented by a max-heap as following:



Each self-update(an upgrade request, cancellation or pop the maximum) of the PQ will cost O(logn) time. Therefore, we will spend O(klogn) to determin the k highest-priority.

In addition, to improve this algorithm, we can have a mapping data structure, such that a hashmap, which stores flyer status as keys and priority queues(waiting list) as values.



An upgrade request or cancellation just cost O(1) time because we just manipulate the particular bucket, either add an element to PQ or remove one from PQ. Also determine the k highest-priority would be O(1) as well. We just look up “super” first, if the number of request in PQ of super can satisfy the need, then return. Otherwise go through other status to find enough requests. The space complexity would be O(n).