Problem Set 1

计83 李天勤 2018080106

维护列表1

根据题目建议，有6个不通思路完成这个题目。 我试了几个方法。

The difference between options 1 and 2 is when the data structure is sorted, and with what method the data structure is sorted with. The data structure I used for this problem was vector. In order to implement the first method, I used `push_back` every time a new number was added, then used a bubble sort to sort the array. Bubble sorts have better performance when sorting an already ordered array (except for the last term). In order to implement the second method, I used a common quicksort algorithm that would sort the array only when we needed to inquire a number. Both these two methods were able to acquire a half the non-invisible points. The main problem with method 1 is that bubble sorts still have to iterate through the entire array. Thus, this method becomes increasingly time inefficient as the data sets gets much larger because we have to iterate through larger and larger arrays.

Then, I looked to method 4, which is in theory an insertion sort for a list ordered from greatest to least, with each new inserted number being inserted in the correct position (from highest to lowest). Using a vector's `insert` function, it would be easy to implement this functionality. And due to the rule that `因为某种特殊性质，每次插入的数一定是从大到小排序后在前 m 的数`, the insertion algorithm most likely will not have to parse through the entire array, thus making this method efficient when m is small. This was enough to get both the non-invisible test cases accepted.

维护列表2

This requirements for this lab is similar to the first one. But instead of returning the value of a integer at a certain position in the list (from greatest to least), this lab wanted us to return the rank of the inserted value in a list ordered from big to small. Using method 4, and returning the position that the value is inserted in, the program was able to achieve `accepted` status on the first two cases, but achieved `time limit exceeded` status on the last two non-invisible test cases.

Due to a time limit, I did not attempt another method to improve the performance of the program. However, I believe performance can be enhanced by taking into consideration of the fact that `因为某种特殊性质，除第一个数以外，新加的数一定和现有的数都不一样，并且与上一个数相比差不会超过 [-m..m]。` Instead of iterating through the beggining of the array,

Stack

This lab required us to track the current maximum element in a stack data structure. In this lab, there are two functions that needed to be implemented

1. add a new element to the list
2. remove the element added latest to the bag

Both of which can be easy implemented using `std::stack` and its `pop` and `push` function. Given the stack, we must able to track the maximum value of it. The algorithm is as described below:

1. push first to the stack

2. compare the next element that is to be added to the stack with the top element of the stack. If it is greater, push that new element, if it is not, push the current top element of the stack
3. when we remove an element from the top, we simply pop the top element of the stack
4. to compute the maximum, simply print the top element of the stack

example

```
push 1, push 3, push 5, push 2, pop + aquire max, pop + aquire max

push 1 : stack : 1
push 3 : stack : 1 3
push 5 : stack : 1 3 5
push 2 : stack : 1 3 5 5
pop + aquire max : stack : 1 3 5, max = 5
pop + aquire max : stack : 1 3 , max = 3
```

Q

This lab is similar to the one above, but instead of removing the latest candy added into the bag, the earliest candy added is to be removed. This is more complicated than the stack lab because the maximum element may be the first element, and if that element is removed, we would needed to update the whole stack.

On the side note, one naive method that could be attempted is simply searching for the largest value in the array everytime that it is required. But the performance of this metod decreases as the data structure grows larger.

In order to implement this algorithm, I used two stacks, one that would model the functionality of enqueue and one that would model the functionality of dequeue, and each of them tracked the element's value, and the current maximum. I used two `std::stack` and `std::pair` .The algorithm is as described:

1. if enqueue is empty, add the first element and its value to the enqueue stack `stack.push({x, x})`
2. when adding a new element, (and enqueue is not empty) compare the value of the new element to the current maximum of the top element of the enqueue stack. If the new element is larger than the current maximum, push `{new_element, new_element}` ,if it is not, push `{new_element, enque_stack.currrent_max}`.
3. when removing a element, and if the dequeue stack is empty, we pop the elements of enqueue one by one and pushing them into the dequeue stack, while recalculating the maximum of the stack using algorithm explained in step 2. Then we pop the element at the top of dequeue.
    1. If enqueue is not empty, the current maximum is calcualted by taking the larger of the two pairs at the top of both stacks. Thus, if enqueue is empty, maximum is simply the top of the dequeue stack.
4. whenever both stacks are empty and pop is called, return -1
5. if both stocks are empty after a pop method, return -1

example

```
push 233, pop, push 2333, push 6666, push 555, pop, pop

1. push 233
```

```
enqueue_stack : {233, 233}
dequeue_stack :
curr_max = 233
2 pop
enqueue_stack :
dequeue_stack :
curr_max = -1
3 push 233, push 6666, push 555
enqueue_stack : {233, 233} {6666, 6666} {555, 6666}
dequeue_stack :
curr_max = 6666
4 pop
enqueue_stack : {}
dequeue_stack : {555, 555} {6666, 6666}
curr_max = 666
5 pop
enqueue_stack : {}
dequeue_stack : {555, 555}
curr_max = 555
```

实验总结

Other than 维护列表2, I did not have much difficulty completing these practice assignments. The only problem that I encountered, and was able to solve, thanks to the 微信群, was the use of `long int` instead of `int` and the use of `scanf/printf` instead of `cin/cout` for all these assignments.