Ex.1

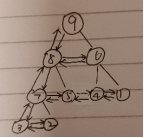
1. If we take a random element as the smallest element at the beginning, making it compared with another element will update the smallest element. Every element should be compared with the latest smallest element. Therefore, (n-1) comparisons are required.
2. To know the second smallest element, we can make pair-wise comparisons from a set A whose elements having compared with the smallest element. The task is to minimize the size of set A. Note that the second largest element must among the one which is compared with the smallest element, as it could not have been compared with any other element. Therefore, we can minimize the size of set to be log(n).
3. We build a binary tree: n leaf nodes represent n elements. Every parent is the smaller element of its children. Take those having compared with the smallest element as a new array and make pair-wise comparison. (Obviously, the number of those elements is the degree of the binary tree) Therefore, it requires other log(n) comparison.

Ex2.

Only codes

Ex3.

1. The priority queues using doubly linked lists can be represented like this.



The parent node is connected to its left child node, and the child nodes in the same

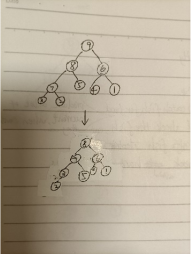
layer are connected with double linked list.

each list item represents an element in the queue obviously.

And a handle is a handle of a list item. The index started from 0, in the tree, it will increase from up to down, left to right.

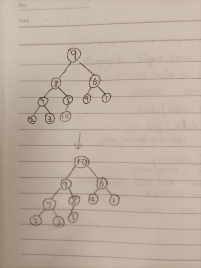
1. For unsorted lists, the push and pop operation should be like this process.

Pop front:



delete the top of the queue and put the bottom to the top, and then compare each node of children node and parent node to adjust the queue. Repeating that process recursively

Push back`:



Add the new element to the bottom, and then compare each node of children node and parents node to adjust the queue.

Both the operation needs time O(logn)

And for sorted list, it is just like an array, we can use the push back and pop front to add or delete elements. Their time complexity are O(1)

Ex4.

4.1

template<class T> void MaxHeap<T>::insert\_k(int k,T arr[])

{

    int i;

    int start\_point = this->getlength();

    for(i=0;i<k;i++)

    {

        this->append(arr[i]);

    }

    int end\_point = this->getlength()-1;

    for(i=end\_point;i>=start\_point;i--)

    {

        adjust\_heap(i);

    }

}

template<class T> void MaxHeap<T>::adjust\_heap(int position)

{

    int current=position;

    int parent;

    T parent\_value,current\_value;

    while(current>0)

    {

        parent=(current-1)/2;

        parent\_value=this->getitem(parent);

        current\_value=this->getitem(current);

        if(parent\_value<current\_value)

        {

            swap(parent,current);

            current=parent;

        }

        else

        {

            break;

        }

    }

}

Thinking:

Firstly, we append the added array at the end of the heap we already sorted. Then, for all the elements we added to the heap, we do sift-up operation to adjust the heap. After that, we will finish adjustment and get the new max heap.

4.2

Algorithm:

We firstly build the heap of added k elements using slit-up. Then, we merge the k\_heap and original n\_heap to insert the k elements into the heap.

Let the depth of the heap be h, for the k elements, we have the relationship . When building the heap, in the depth d, we have totally nodes, and for every node it needs to do sift-down operation for times. So, for the total time complexity , we have the relation: . We can solve this using , finally we have , we replace with , we will have , Therefore, the complexity of building a heap with k elements is .

Then we merge the two heaps. Since the merge operation is actually the same as heapify (sift-down) operation for the root of the heap. Therefore, we have the complexity .

In conclusion, the total complexity is