Exercise 1:

(1) In a comparison-based algorithm for determining the smallest of n elements:

Initially, all the n elements can be the "smallest element" (call it MIN). Then we can delete **one and only one** element from the "candidates" of MIN from each time of comparison. As we need to find the smallest element, we have to delete totally (n-1) elements, and thus we need at least (n-1) comparisons.

(2) It is easy to prove that after **proper** (n+logn-1) comparisons, we can always find the smallest and second smallest elements. (Please refer to the method in question (3)).

Now it remains to prove that this is the smallest number of comparisons needed:

1. The first step, we need to find the smallest element, which needs (n-1) comparisons as shown in (1).
2. The second step, we need to find the second smallest element.

One useful fact here is that, the second smallest element must has been compared (to be larger, and then deleted from the "candidates") with the smallest element in the first step. Then we only have to find the smallest element within this kind of elements.

If we draw out the first step of comparisons as a binary tree in which parent element is the smaller one of its two children elements, then the tree will have (2n-1) nodes (with the n origin elements as leaf nodes). To save times of comparison, we should make the number of elements in each level of the tree to be as fewer as possible. In such a binary tree, each level has {k/2} elements, where k is the number of elements of the level below it and {a} means taking the smallest integer that is larger than a.

Then, the biggest length of a route that the smallest element "sifts up" from leaf node to the "top" node will contain {log((2n-1)+1)}={log(2n)}={logn+1} nodes of this element, indicating that {logn} elements have been compared with it. So within these elements, we need {logn-1} comparisons to find the second smallest element.

3. Totally, to find the smallest and second smallest elements, we need at least (n-1)+{log(n)-1}={n+logn-2} comparisons. As n may not be the power of 2, we need at least (n+logn-1) comparisons.

(3) An algorithm suiting both (1) and (2) is as follows:

1. For each element, distribute a set to store the elements that have been compared with it, which is initially empty for them all.
2. Divide all the n elements in the array in groups of 2. There may remain one element.
3. Do comparison for each group, and store all the smaller elements (call them "winners") as well as the possibly remaining element into a "new set". Also, for each winner, update their set of compared elements.
4. Solve the problem with the "new set" by repeating 1.2.3. again and again, until the "new set" has only one element. **That is the smallest element.**
5. Then, take the set storing all the compared elements with this smallest element. Solve the problem with this set and we could get the second smallest element.

Exercise 3:

(1) We can Use a structure (say, pq) to represent nodes, which include key (in int), previous node (in pq*) and next node (in pq*). Here, we can regard pointers of previous node and next node as handles of the item.

To realize the properties of addressable priority queue (assume the smallest key value is at the beginning), we need to firstly sort the doubly linked list by the key of each node. This can be done by ordinary sorting algorithms like bubble sort, with the addition of changing pointers.
Then with this doubly linked list: 1) We can do insert(k) by firstly insert one item to the last of the list as one new node, then continuously compare its value with its previous node and change position (can be done by "delete" and "insert" operations of doubly linked list) until finding one node with smaller key value than its. 2) We can do delete(h) by finding from the sentinel to the node with the corresponding handle and then do "delete" operation. 3) We can do decrease(h,k) by finding from the sentinel to the node with the corresponding handle and after that, do delete(h) and insert(k). 4) We can do delete_min() by doing "delete" for the first node after the sentinel.

(2) Insert: For the sorted list, the time complexity is O(n), as we need to firstly append it at last, and then find an appropriate place for it; for the unsorted list, the time complexity is O(1), as we only need to append it at last.

Delete: For the sorted list, the time complexity is O(n), as we need to find where the element we want to delete is and then do the deletion; for the unsorted list, this is also the case, so the time complexity is also O(n).

Decrease: For the sorted list, the time complexity is O(n), as we need to firstly find where the element we want to decrease is and then do the change, and after that we need to "sift up" to find the appropriate place of the new key value, so there is totally O(2n)=O(n); for the unsorted list, we only need the first step in the procedure described previously, but the time complexity of finding and changing is also O(n).

Delete_min: For the sorted list, the time complexity is O(1), as we simply need to delete the first element, which is already the smallest one; for the unsorted list, the time complexity is O(n), as we need to firstly find the smallest key value among all the n elements.

Exercise 4:

(1) Two obvious methods are shown here:

i) Use the operation "INSERT", which is introduced in the lecture, for k times. The time complexity of this method is O(k*logn).

ii) Tear down the origin heap and build a new heap with these (k+n) elements. The time complexity of this method is O(k+n).

[Note that both of the two methods have time complexity bigger than O(k+logn).]

(2) Here we provide one better algorithm:

1. Firstly, append the k new elements at the end of the heap. Here we define "H2_node" to represent the node that satisfy: (i) it is from the n nodes in the original max-heap; (ii) it has children which include at least one node from the origin n elements; (iii) its grandchildren, grand-grandchildren …are all from the newly appended k elements.

2. Then, for all "H2_nodes", build max-heap by regarding it as the root.

3. Finally, for all the nodes that didn't participate in the "building max-heap" procedure (this kind of node must come from the original n nodes), do sift-down for all of them from the bottom to the top to ensure the properties of max-heap.

Next we show that this algorithm has time complexity O(k+logn) by doing induction for the height h of the original n-node max-heap.

1。 For h<=2, n<=3, so we can build a new max-heap, as mentioned in the second step in the algorithm, with all these at most k+3 elements, whose time complexity is $O(k+1) \in O(k+\log 1)$, $O(k+2) \in O(k+\log 2)$, $O(k+3) \in O(k+\log 3)$. So the statement is true for h=1 and h=2.

2。Assume that when the statement is true when the height of the original heap is h0 (h0>=2). Now for

a heap with n nodes and height h0+1, we can consider its left and right subtree (w.r.t. the root node) respectively, both of which has height h0. Further assume that we have append k1 new nodes to the n1-node left subtree and k2 new nodes to the n2-node right one (here k1+k2=k, n1+n2+1=n). By our assumption, protect a max-heap for the left subtree has time complexity O(k1+logn1), and for the right subtree O(k2+logn2). After that, we only need to sift down the root to obtain a heap with (k+n) nodes, whose time complexity is O(log(k+n)). As log(n1)<log(n), log(n2)<log(n), we know O(k1+logn1) + O(k2+logn2)$\in$O(k+logn), and further we have O(log(k+n))$\in$O(k+logn). Totally, O(k1+logn1) + O(k2+logn2) + O(log(k+n))$\in$O(k+logn). Therefore, when the statement is true for original heaps with height h0, it is also true with original heaps with height h0+1.

By 1。2。and induction axiom, the conclusion is that we can insert k new elements into a max-heap with n elements with time complexity O(k+logn) by using the algorithm given.