CS225 Assignment 3

Group 18: Xu Ke, Wang Dingkun, Qian shuaicun, Hua Bingshen

March 2021

1 Assignment 3 Exercise 1

1.1

When using comparison-based algorithm to determine the smallest of n elements, it is clear that we can begin with one arbitrarily chosen element and compare with any one from the rest and record the smaller one, and then consider it as the newly-chosen one to do another comparison until no element left. Thus, more specifically, for the situation with n elements, choose one and compare it with anyone from the rest (n-1) element. Then keep the smaller one and compare the number with the anyone from the rest (n-2) elements. Repeatedly, and at last we will finish comparison when only one element left which is the answer we want to get. During the whole process, we totally have n-1 comparisons.

Of course, with another method, we divide them into $\frac{n}{2}$ groups with 2 elements in a group to compare (if n is an even number, that is; otherwise, one group has only 1 element and itself is the small one) and then keep the $\frac{n}{2}$ smaller one and repeat until only one element left which is the answer that we want to get. In this way, we also have totally n-1 comparisons.

1.2

Using comparison-based algorithm to determine the smallest and second smallest elements of n elements, it is clear that we can divide them into $\frac{n}{2}$ groups with 2 elements in a group to compare (if n is an even number, that is; otherwise, one group has only 1 element and itself is the small one) and from the last question, we have already proved that in this way, comparison-based algorithm for determining the smallest of n elements requires n - 1 comparisons. And since n elements in this way will have a comparison depth of $logn$. And then we need to find the second smallest element. In order to find it, we just need to compare each element in the last-step comparison which the smallest one is also in (of course, the smallest one should be excluded). Considering the worst condition, that is the first smallest and second smallest elements are divided in the same group during the first step, we have to go back $logn$ times to find the second smallest element. Thus, totally, time consumption of (n-1+$log(n)$)

will be needed. Thus, we can conclude that any comparison-based algorithm for determining the smallest and second smallest elements of n elements requires at least (n-1+$log(n)$) comparisons.

1.3

It is clear that one way of using comparison-based algorithm to determine the smallest of n elements is to use the binary tree. Every leaf (external) node represents a element and internal node represents the smaller one. There will be n − 1 internal nodes in a binary tree with n leaf (external) nodes.

It is obvious that to select the smallest element among n nodes, (n − 1) elements to be eliminated, that is to say, we need minimum of (n − 1) comparisons. Mathematically, in a binary tree I = E − 1, where I is number of internal nodes and E is number of external nodes. It means to find minimum element of an array, we need n − 1 (internal nodes) comparisons.

As mentioned above, we need at least (n-1) comparisons to find the smallest element. Then what we still need to do is to find the second smallest element, and binary tree is still a way to deal with. Let's consider the condition with n leaf (external) nodes again. After finding the smallest elements, we just need to compare each element in the sub-tree of the smallest one (of course, the smallest one should be excluded). It's clear that a binary tree with n leaf (external) nodes has the deep of $log(n)$. Thus, another $logn$ will be needed to find the second smallest element. Thus, totally, (n-1+$log(n)$) is the least time consumption. Thus, we can conclude that any comparison-based algorithm for determining the smallest and second smallest elements of n elements requires at least (n-1+$log(n)$) comparisons.

And here is a **code example** as following:

```
#include <bits/stdc++.h>
using namespace std;

struct Node
{
    int idx;
    Node *left, *right;
};

// Utility function to create a tree node
Node *createNode(int idx)
{
    Node *t = new Node;
    t->left = t->right = NULL;
    t->idx = idx;
    return t;
}

```

```
// This function traverses tree across height to
// find second smallest element in tournament tree.
// Note that root is smallest element of tree.
void traverseHeight(Node *root, int arr[], int &res)
{
    // Base case
    if (root == NULL || (root->left == NULL &&
                         root->right == NULL))
        return;

    // If left child is smaller than current result,
    // update result and recur for left subarray.
    if (res > arr[root->left->idx] &&
       root->left->idx != root->idx)
    {
        res = arr[root->left->idx];
        traverseHeight(root->right, arr, res);
    }

    // If right child is smaller than current result,
    // update result and recur for left subarray.
    else if (res > arr[root->right->idx] &&
             root->right->idx != root->idx)
    {
        res = arr[root->right->idx];
        traverseHeight(root->left, arr, res);
    }
}

// Prints minimum and second minimum in arr[0..n-1]
void findSecondMin(int arr[], int n)
{
    // Create a list to store nodes of current
    // level
    list<Node *> li;

    Node *root = NULL;
    for (int i = 0; i < n; i += 2)
    {
        Node *t1 = createNode(i);
        Node *t2 = NULL;
        if (i + 1 < n)
        {
            // Make a node for next element
            t2 = createNode(i + 1);

            // Make smaller of two as root
            root = (arr[i] < arr[i + 1])? createNode(i) :
                                           createNode(i + 1);

            // Make two nodes as children of smaller
            root->left = t1;
            root->right = t2;

            // Add root
            li.push_back(root);
        }
        else
            li.push_back(t1);
    }

    int lsize = li.size();

    // Construct the complete tree from above
    // prepared list of smaller in first round.
    while (lsize != 1)
    {
        // Find index of last pair
        int last = (lsize & 1)? (lsize - 2) : (lsize - 1)
            ;

        // Process current list items in pair
        for (int i = 0; i < last; i += 2)
        {
            // Extract two nodes from list, make a new
            // node for smaller of two
            Node *f1 = li.front();
            li.pop_front();

            Node *f2 = li.front();
            li.pop_front();
            root = (arr[f1->idx] < arr[f2->idx])?
                createNode(f1->idx) : createNode(f2->idx)
                    ;

            // Make winner as parent of two
            root->left = f1;
            root->right = f2;

            // Add winner to list of next level
            li.push_back(root);
        }
        if (lsize & 1)
        {
            li.push_back(li.front());
            li.pop_front();
        }
        lsize = li.size();
    }

    // Traverse tree from root to find second minimum
    // Note that minimum is already known and root of
    // tournament tree.
    int res = INT_MAX;
    traverseHeight(root, arr, res);
    cout << "Minimum: " << arr[root->idx]
        << ", Second minimum: " << res << endl;
}

// Driver code
int main()
{
    int arr[] = {61, 6, 100, 9, 10, 12, 17};
    int n = sizeof(arr)/sizeof(arr[0]);
    findSecondMin(arr, n);
    return 0;
}
```

2 Assignment 3 Exercise 2

Please see the code attached in the file.

3 Assignment 3 Exercise 3

3.1

A doubly linked list can have two pointers. One points to the head of the list, another one points to the end of the list. And the addressable priority queue has one entrance and one exit which are corresponding to the head and the end of the list.

Insert(k) is the same as insert a element in a Dlist. If this element is the smallest, let the end pointer point to this element.

Delete(h) is also the same as delete a element in a Dlist. $Delete_min$ is just delete the smallest element and make the end pointer point to the new smallest element.

Decrease(h,k) is just change the key at h. If this element becomes the smallest one, let the end pointer point to this element.

3.2

For the sorted lists, we have:

Insert(k): Θ(n) we need to check the whole list to sort the elements.

Delete(k): Θ(n) we need to find k elements to delete.

$Delete_min(k)$: Θ(1) we just need to delete one element.

Decrease(h,k): Θ(n) we need to find h element to delete.

While for the unsorted lists, we have:

Insert(k): Θ(1) we just need to put the element at the end.

Delete(k): Θ(n) we need to find k elements to delete.

$Delete_min(k)$: Θ(n) we need to make n-1 times comparisons to find the smallest element.

Decrease(h,k): Θ(n) we need to find h element to delete.

4 Assignment 3 Exercise 4

4.1

We can define a function with a loop which add the elements one by one with the insert() function, then the new max-heap was finished, and the time complexity will be $O(klogn)$

4.2

Make a heap with these k elements, the time complexity will be $O(k)$, Take this heap as a node so the time complexity will be $O(logn)$ So the total time complexity of this algorithm will be $O(k + logn)$