# VE477 Lab4 Report

Student Name: Zhou Haoquan

Student ID: 519370910059

## 2. Represent the time complexity

To begin with the analysis of time complexity, we first focus on a function called potential function for a given Fibonacci Heap. The detailed representation can be written as

$$\Phi(H) = T(H) + 2 \times m(H)$$

The `T(H)` gives the number of root nodes in the Heap, or we can treat it as the number of trees in the forest. `m(H)` is the number of nodes that are marked in the Heap. The essential point is that, if we have more trees or more marked points, we tend to have the amortized time increasing. **So when we calculate the amortized time complexity of an operation, we should add the time cost for doing all the operation and the change of potential together.**

#### 2.1 `MakeHeap`

To make a Fibonacci Heap from a randomly generated array, we just need to insert all the elements at the root linked list with time complexity

$$\mathcal{O}(n)$$

The potential change is the same with this complexity. So it is the overall result.

#### 2.2 `Minimum`

Since we are maintaining a pointer pointing to the minimum element in the heap, visit it requires time complexity

$$\mathcal{O}(1)$$

#### 2.3 `Union`

This action is also easy, since we just need to link the root linked list of two heaps together, which requires time complexity

$$\mathcal{O}(1)$$

Note that the total tree roots before combination and after combination does not change. Neither does the number of marked point. So the potential function does not change. This means the time complexity above is the final answer.

#### 2.4 `ExtractMin`

We can access the minimum element and delete it in

$$\mathcal{O}(1)$$

time. But we need to maintain the children of this element. To maintain the minimum pointer, we will traverse through the root linked list, which takes

$$\mathcal{O}(rank(H)) + T(H)$$

time. Also, we should maintain the current Heap to ensure none of the root node has the same rank. It also takes

$$\mathcal{O}(rank(H)) + T(H)$$

time to traverse and do the combination. Actually, the two steps above can be done in one traversal

Now let check the change in potential function. The tree nodes will be at most `rank(H)+1` because we ensure that all the tree nodes have different rank. And the number of marked points will not change. So the change of potential function is

$$rank(H) + 1 - T(H)$$

Then we can calculate the time complexity to be

$$\mathcal{O}(rank(H)) + T(H) - rank(H) + 1 - T(H) = \mathcal{O}(rank(H)) = \mathcal{O}(\log n)$$

**2.5 `Insert`**

To insert an element, we just insert it in the root linked list, which takes

$$\mathcal{O}(1)$$

time. And the potential function only experiences a `+1`, so the time complexity remains constant.

**2.6 `DecreaseKey`**

It takes us

$$\mathcal{O}(1)$$

time to decrease the value of one node. If the invariant of `min-heap` does not change, then we are done. Otherwise we need to consider the following operations.

Cut the tree rooted at this node and melt it into the root linked list takes

$$\mathcal{O}(1)$$

time. And we need to check whether its parent is marked in

$$\mathcal{O}(1)$$

time. If the parent is not marked, we are done. If the parent is marked, we cut it done and melt it into the root linked list. We recursively do this until we meet a root or we have an unmarked parent. Assuming that we have done `c` cuts in total. Then the time complexity for each cut and melt is

$$\mathcal{O}(1)$$

The total time cost is

$$\mathcal{O}(c)$$

Now let's consider the change of the potential function. The tree roots after melting `T(H')` is `T(H)+c` because each time we cut a node and melt it into the root linked list, we increase the `T(H)` by one. Then number of marked points after all the `c` cuts is no larger than `m(H)-c+2`. The worst case happens when the point that we want to `DecreaseKey` and its root node are not marked and all its other ancestors are marked. So the change of potential is

$$c + 2 \times (-c + 2) = -c + 4$$

Note that we can actually make the unit of the potential function changed and then the overall time complexity can be

$$\mathcal{O}(c) + (-c + 4) \times t$$

where `t` can be any constant due to change of unit. We can make `t` suitable to cancel out the `O(c)`, so the overall time complexity can be

$$\mathcal{O}(1)$$

#### 2.7 `Delete`

The `Delete` operation is just a combination of `DecreaseKey` and `ExtractMin`. Namely, we first decrease the key number to be less than the current minimum and then we extract it out. The time complexity for this operation is

$$\mathcal{O}(1) + \mathcal{O}(\log n) = \mathcal{O}(\log n)$$

## 3. Pros and Cons

#### 3.1 Advantages

The advantage of using a Fibonacci Heap is that the time complexity all the operations are faster or as fast as the min heap. So it will provide a better performance compared with the min heap.

#### 3.2 Disadvantages

Although the theory proves that the time complexity of Fibonacci Heap is better than that of min heap, but the time complexity is amortized, which means that for a single operation, it is not as fast as what we expect. Also, the implementation for a Fibonacci Heap is more complicated than the min heap. It may take a lot of time to design a Fibonacci Heap, while the min heap is fast to implement.

## 4. When to use Fibonacci Heap

Take the implementation cost into consideration, the Fibonacci Heap should be used when the input data is large and the time difference between different heaps are too large to be ignore. Namely, the user wants fast operation over a huge data input, and the user may do a lot of operations such that we are safe to analyze the time complexity with the amortized time complexity. Then we can apply the Fibonacci Heap.