Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 03

Lecture 11

Welcome to the eleventh lecture of the second-level algorithms course. We have been studying Fibonacci heaps for the last couple of lectures. So, today also we will continue that. In the last lecture, we have seen the amortized time complexity of insertion and extract min. We have seen that extract min takes O(log n) time amortized and insertion takes O(1) time amortized.

Today in this lecture, we will see that decrease key takes O(1) time amortized, ok? So, let us begin. So today, we will prove that the amortized time complexity of decrease key is O(1). Recall we have started taking an arbitrary sequence of n operations where each operation is either an insert, extract min, or decrease key, and we have maintained the loop invariant or the invariant that after the i-th operation, all the nodes in the root list carry 1 rupee.

And all the nodes that are marked, that means the nodes who have lost one child since the time they have been made a child of another node, those nodes also carry one rupee. So that was our invariant. Every node in the root list carries one rupee and every marked node carries 1 rupee. So, we will change this loop invariant slightly and make it so every marked node carries, instead of 1 rupee, 2 rupees.

This is important for proving this result. Notice that this change does not invalidate the amortized time complexity of insertion and extract min because those operations do not change the set of bar nodes. Also, we decided that we will charge decrease key by 1 rupee. So, this also will change: instead of 1 rupee, we will say 2 rupees, and this also will change: instead of 2 rupees, we will charge them 3 rupees.

We will and recall that 1 rupee can pay for any constant-time operation. So, now, suppose the ith operation is a decrease key. So, we will make a couple of cases depending on the

node whose key is decreased. Case one is the node whose key is decreased is part of the root list.

In this case, the actual cost is O(1). This cost we pay by the 3 rupees we have charged for this operation. Although 1 rupee is enough, so we are overpaying. Case 2: the node whose key is decreased is not part of the root list, but decreasing the key does not violate the min-heap property. So, there was no cut operation.

In this case, also, the actual cost is O(1). We cover up this cost by rupees 3 charged for this operation. You see, for both case 1 and case 2, the set of nodes or the root list does not change. So, the invariant because it was. True before the ith operation, it continues to hold after the i-th operation, and also the set of marked nodes does not change for both case 1 and case 2.

So, because the invariant held or the invariant was true before the ith operation, the invariant continues to hold true after the i-th operation. This is the most interesting one. In the third case, the decrease key results in one cut and maybe 0 or more cascading cuts. So, let K be the number of cascading cuts. So, now you observe that we, or the actual cost, the actual cost of

the non-cascading cut performed on the key on the node whose key is decreased is O(1). So, we spend 1 rupee from 3 rupees charged for the operation, ok. So, you are left with 2 more rupees.

so we store 2 rupees with the newly marked node if any. You observe that when you perform a cut the cascading cut traverses upward and this will stop either at a root node or at some node which was not marked before and that node will be marked. So, suppose this is the root node, this is the part of a root list and we are performing a decrease key. on this particular node.

Here we are performing decrease key and suppose this decrease key results in violation of mean heap property. So, we cut the link between this node and the parent node and if the parent node is already marked then I have to cut the link between parent node and grandparent node and I continue in this fashion when will the process will stop? It can stop in two ways. One way is that this process reaches the root node in that case the process stops there or it can stop when this reaches to some node which is not merged and after this operation we mark this node. So, in every decrease key operation at most one node may be marked it can it we may not have a newly marked node, but if there is a

node which is marked in this operation that will be at most one that is why you say if any. So, if there is any newly marked node with because of the invariant we need to store 2 rupees with the newly marked node. So, we store the remaining 2 rupees with that newly marked node, but then who will pay for the cascading cuts? You see each cascading cut takes O(1) time and there are 2 rupees stored with the node which are marked each node which is cut because of cascading cut they are already marked. So, this is the first observation. Observe that every node that are cut during cascading cut operation. must already be marked and thus contain 2 rupees with them. Cutting such a node has an actual cost O(1) which we pay using rupees 1 out of 2 rupees stored in the node that got cut during cascading cut. and the remaining 1 rupee is stored with them because they are now the part of the root list and because of loop invariant that is needed.

The remaining 1 rupee is stored with them who are now part of root list as required. by the invariant condition. So, we are maintaining this invariant condition which there is at most one node which is marred in this operation and we are storing 2 rupees and all the nodes which are cut because of cascading cut that they are cut operation takes O(1) time they have 2 rupees with them. So, we spend 1 rupee to pay for the O(1) time operation for their cut and the remaining 1 rupee is stored with them.

which are now part of root list. So, invariant condition holds true except for one node which node? The node which is cut. So, you see this node on which the decrease key operation is performed, this node if it was marked before then it is fine because if it is marked before it has two rupees with it and now it is part of the root list it needs one rupee with it it has more so that's fine but if it was not marked node then there was no money with it and

we are paying for the operation for cutting this node to make this node part of root list, but we are not storing the 1 rupee that needs to be stored with this node because of loop invariant. To tackle this thing the fix is very simple instead of charging 3 rupees for the Decrease key operation, you charge 4 rupees. Every mark note carries not 1, 2 rupees. Okay, so now you charge 4 rupees.

Now you see case 1, case 2 is still fine for case 3. you spend 1 rupees out of 4 rupees. Now, so you have 3 rupees left. So, you store 2 rupees with the newly marked note. So, you still have 1 rupee left that 1 rupee you store with this.

Node on which the decrease operation is performed is now part of the root list. The remaining 1 rupee is stored. With the node on which the decrease key operation was performed. And that resulted in cutting this node and making it Part of the root list.

So, what do we have? After the ith operation, the newly marked node If any, stores 2 rupees with it and every node That was not part of the root list before the ith operation. But has become part of the root list after the ith operation

Stores 1 rupee with it. This proves that the invariant Invariant holds true after the ith operation given It was true before the ith operation. Hence, all the operations, all the actual costs of the operations are paid.

By the amount charged. So, this proves that the amortized time complexity of insertion is O(1), extract-min is O(log n), assuming that the maximum degree of any n-node Fibonacci heap is O(log n), and the amortized cost of decrease-key is O(1). So, let us stop here. In the next class, we will prove the missing piece, which is proving that the maximum degree of any n-node Fibonacci heap is O(log n), and there we will see the use of Fibonacci numbers, thereby justifying the name of the data structure. OK? So, let us stop here. Thank you.