Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 02

Lecture 09

Welcome to the ninth lecture of the second-level algorithms course. We have been studying Fibonacci heaps, and we have studied the insertion procedure and the extract-min procedure. So, now let us move on to the decrease-key procedure. Decrease key. So, as usual, before moving on to the formal pseudocode, let us first understand this decrease-key operation using some examples.

So, for that, let me take a Fibonacci heap, and we will perform multiple decrease-key operations. Suppose this is how the root list looks. So, I have taken a toy example of a Fibonacci heap, and let us perform multiple decrease-key operations. And where does $H.min$ point to? This 7.

Let us make this chain slightly longer. So, in this decrease-key operation, we are given a pointer to the node and the value of the new key. So, the value of the new key should be less than the value of the current key. So, if the value of the new key is more than the value of the current key, we will just display an error. So, let us assume that the first operation is decrease-key.

that this node this 20 this node the new value should be say 11. So, if it is in the root list, the key to be decreased, if it is in the root list then the process is very simple. You simply change the key to 11, the new key and you update the $h.min$ if necessary. So, what is the new list then? Okay.

And in this case, the new key is still more than the $h.min$. We do not need to update the $h.min$. So, had the decrease key operation made the key new key of the node to be say 5 or 1, then we just had to update the $h.min$. That is it. The more interesting case happens when the node to be decreased is not the root list, not in the root list.

So, suppose I now next want to decrease this node to say 2, I perform decrease key and make it say this node with key 70 actually it points the first parameter should be a pointer to the node not the key so that I do not have to search, but for this example let us write 70 and the new value I want to make it say 5. So, what should I do? The first step is I just make it delete 70 and make it 5. Now, if the new key is continued to more than the parent of the key, the parent of the node.

then the heap property is maintained and I do not need to change anything. So, for example, if I intended to change the key from 70 to say 47, then 47 continue to be more than 7. So, heap property is not violated anywhere and hence that we will stop the decrease criteria. We do not need to change anything else, we simply change the key from 70 to 47, but it can happen that by after decreasing the key the heap property is violated.

What is the heap property? Heap property says that for every node the key of the node should be greater than equal to the key of the parent node that is the heap property that we need to check after decreasing the key of any node whether it is violated or not. So, in this case that value of the new key is 5 and the parent is 7. So, the heap property is violated. So, what should we do?

We then cut this node from being the child of 7 and make this part of the root list. So, how does the new heap then look like? this was the original link list original root list and I insert the node whose value I decreased and that violated the heap property in the root list that 5. If 5 this node has some subtree then the whole subtree I would have moved here along with it.

So, basically, I cut this node from its parent, and that subtree rooted at that node—the entire subtree—I move it to the parent list. So, let us see how the heap looks after this decrease-key operation. So, this is the root list. And the parent of 5, because it is a node in the root list, is never marked because this node has not been made a child of any other node. In general, now let us see another decrease-key.

So, suppose I decrease this key to the value 65, say. So, if I decrease this key to the value 65, again, the heap property gets violated. The parent of 65 is 90, which violates the heap property. So, I cut down this subtree and make it part of the root list. But before that, here, $h.min$ also needs to be updated because it used to point to 7.

So, whenever you insert a new node in the root list, you always need to check whether $h.min$ needs to be updated. So, $h.min$ now points to 5. So, now, after performing this decrease-key, let us see how the heap looks. This is how the root list looks. I have inserted a new key in the root list.

So, I need to check whether I need to update the $h.min$. Answer is no because $h.min$ currently points to 5 and the key of the newly inserted node is 65. This is $h.min$. Now, here you see that 90 was suppose 90 was unmarked that means the mark field of that node was was false. So, now, it has lost a child whenever a non group node loses a child and if it is unmarked we will mark it.

So, we will mark 90. So, 90 is now marked. Now, suppose the next decrease key operation is on 120 and it wants to make it say 1. So, again I change 120 to 1 and we see that hip property is violated.

90 cannot be a parent of 1. So, I cut this node 1 and make it part of the root list again. So, here is the new Fibonacci hip. This is how it looks like. Whenever a new node is inserted in the root list, we need to check whether we need to update the $h.min$ or not.

In this case, again yes, we have to update the $h.min$ because $h.min$ was pointing to 5 and the newly inserted key is 1 which is less than 5. So, here is where $h.min$ points. Now here you see 90 is already marked. So that means 90 had already lost a child and it is now losing its second child. Now here we perform an important step that whenever a node loses its second child that node also we will make it part of root list.

I will cut down the link between that node and its parent. And the entire subtree rooted at that node, if it had some subtree here along with 90 and this subtree would go to the root list again. So the root list will also have 90. and again a node has been inserted in the root list, we need to check whether we update the $h.min$ or not. In this case, we do not need to update $h.min$.

So, this is how the fibonacci heap decrease key operation is performed. So, let us summarize the process. If the node whose value is decreased is a root node, then I decrease the key and update $h.min$ if needed. If it is a non root node, I update the key of that node and if it does not violate the heap property the main heap property then that is it we will not need to perform anything else.

If it violates the main heap property that means the value of the node value of the key in that node is less than the value of the parent node then that means it violates the main heap property in that case we cut the link between the node and the parent node and the subtree rooted at the node where we have performed decrease key that entire subtree becomes part of the root list and the parent of the node whose value we decreased. that parent node we mark it if it is not already marked and if it is marked then we cut the parent the link between parent and grandparent and make the parent part of root list and again here you see the grandparent now loses the child so we mark the grandparent also Now in this case the grandparent was unmarked but if the grandparent was also marked then I would have cut the link between grandparent and its parent and this process will continue. So you see that we can make many cuts because

of this mark node and this is why we have stored the information whether a node has lost any children any child since the time it has been made the child of another node. So, recall in the whenever we merge two trees in the Consolidation procedure in extract means the node which is made a child of the root node, then we reset the mark field, we make the mark false. So, with this now let us go to the pseudo code of decrease key operation. So, let us call this procedure fib hip decrease key h is the hip, x is the node whose key needs to be decreased and k is the value of the new key.

So, first you check if the value of the new key is less than the key stored in x. If it is not, then we throw an error message. okay so after this we can assume without loss of generality that k is less than equal to x dot key so in this case first i update the key field of x and at y_i store the parent pointer right and if the parent is null then that is fine I do not need to do anything I later I will update the $h.min$ if needed. So, if it is not null then I have to trace back to the root list or to an unmarked node. So, if

First I check whether the heap property is valid or not. So, here is x and its parent is y. So, if it has a parent that means y not equal to null and x dot key is less than $y.p$. Then the mean heap property is violated. Then what do we do? We cut this

This process cuts the link between X and Y and make X, the subtree rooted at X, the part of the parent list. But that is not enough. We have seen that we may need more cuts and that is handled in another procedure. Let us call it cascading cut. h y. So, from y to the first unmarked node the cut will continue and then I check I update the minimum.

So, if h dot min You see that the nodes separated in cascading card, they cannot be the new minimum node that you see that is easy to prove. So, all I need to check whether $h.min$ is less than $x.key$ or not. So, if this is the case, then I do not need to update the $h.min$. So, I should check the other way.

if $h.min.key$ is greater than $x.key$ then $h.min$ is equal to x. So, this is how the decrease key operation is processed and now we see these two procedures which is the first one is cut and the second one is cascading cut. So, let us begin with cut h x y. So, remove x from the child list of y, decrease $y.degree$ by 1. okay and then add x to the root list of H. We need to update the parent pointer of x. So, $x.p$ is null.

and the root list the nodes in the root list they are because they are not child of any other node they are always unmarked. So, $x.mark$ is false this is cut and now let us see cascading h, y. So, let us first get hold of the parent of y and store it in z. So, if z is null that means y is a root node and it does not need to be cut. So, we continue if z is not equal to null. So, in this case, we see if y is marked or not.

If y is not marked, that means if $y.marked$ is false, then we simply mark y. and we are done else if y is already marked we need to cut y from z and make it part of root list else we do what we cut y from z okay and again check from z whether do we need to make a cascading card or not. This is the end of else, end of if, end of cascading card. So, this is the pseudo code of cascading cart.

So, in the next class, we will see the analysis of this data structure. Till now, we have seen all the basic operations, that means build heap, which is just an empty heap. Then insertion, extract min, decrease key, and deletion. Deletion is basically decrease key—you decrease the key to, say, minus infinity, or you first find out the min. Whichever node you want to decrease or delete, you perform a decrease key operation with a value less than the current minimum, and then you perform extract min. So, delete key can be performed by decrease key followed by extract key. So, in the next class, we will analyze the amortized time complexity of all these three procedures.

So, let us stop here. Thank you.