# Second Level Algorithms

**Prof. Palash Dey**

**Department of Computer Science and Engineering**

**Indian Institute of Technology, Kharagpur**

**Week – 02**

**Lecture 07**

Welcome to the seventh lecture of the second-level algorithms course. So, till now, we have studied amortized time complexities. We have seen three methods for computing the amortized time complexity. They are aggregate analysis, accounting method and potential function. Then, we have seen the dynamic table data structure, which supports $O(1)$ amortized time complexity for both insertion and deletion. Today, we will start an important topic called Fibonacci Heap, and we have already seen that using Fibonacci Heap, we can make the Dijkstra's algorithm substantially faster, ok.

So, let us begin with Fibonacci Heap. So, it is the same. The set of operations supported in Fibonacci Heap is the same as the min-heap. So, let us list down the comparison of time complexity between min-heap and Fibonacci Heap. So, here are the procedures. Here is the min-heap. Worst-case time complexity. For min-heap, the worst-case time complexity of the operations is the same as the amortized time complexity of those operations. Here, we have Fibonacci.

Here, we are looking at amortized time complexity. The first operation is make-heap. It is just making an empty heap, initializing the heap. Both are $O(1)$ or $\Theta(1)$. Ok. Then, we have insert. Insert in min-heap takes theta of worst-case time complexity, which is $\Theta(\log n)$, whereas for Fibonacci Heap, insertion is $\Theta(1)$. Then, we have minimum.

it just returns the minimum key without deleting the minimum key from the heap both takes $\Theta(1)$. Then we have extractmin, this deletes the key the minimum key from the heap and returns extract means mean heap the worst case time complexity is theta of log n and here it is Fibonacci heap $O(\log n)$. That means, the amortized time complexity of extract mean of Fibonacci heap is at most some constant times $\log n$. We will use the charging method accounting method to prove this and we will see that sometimes we will charge less that is why I am using $O(\log n)$ where in the other notations it is $\Theta$ ok. Then decrease key.

So, we are given a pointer to the key and we have to decrease it and then rearrange the heap. So, that it satisfies the heap property. So, decrease key is the main thing if you recall during the implementation of Dijkstra's algorithm we the total number of decrease keys performed is the number of edges or big O of number of edges twice the number of m twice the number of edges. and for mean heap decrease key also takes $\log n$. On the other hand for Fibonacci heap decrease key takes $\Theta(1)$ and this is the reason why the Fibonacci heap gives much better Fibonacci heap implementation of Dijkstra's algorithm gives much better worst case time complexity than the mean heap based implementation of Dijkstra's algorithm.

Let us also write down the worst case time complexity of these operations for Fibonacci heap. So, worst case make heap again $\Theta(1)$. insert again $\Theta(1)$, min $\Theta(1)$, extract min is worst case is $\Theta(n)$ this could be quite bad in the worst case and for decrease key this also no the worst case could be as bad as $\Theta(n)$. So, now, let us go into further details. So, first let us see what does each node store in the Fibonacci heap.

In classical mean heap each node in the tree implementation each node stores the left pointer to left child pointer to right child and pointer to parent child. that is the typical node in in any binary tree stores, but recall the min heap because it is a complete binary tree the array implementation of the min heap is much more efficient both time and space wise than the pointer implementation. But here in Fibonacci heap we do not have that complete binary tree structure. So, we will use the tree structure. So, what does each node of the Fibonacci heap stored. first obviously, every node must store a key value, then a node stores a pointer to parent ok, then pointer to any one child then pointer to left sibling pointer to right sibling . and a special Boolean flag called marked which indicates whether. So, let me write.

So, a special or a Boolean variable called marked which is set to true if the node has lost any of its children. since the time it was made a child of another node. So, let us not worry about it we will this marked thing is required in the decrease key operation. So, we will come back and explain more what does this mean when we discuss the decrease key operation, but before that let us do the easy things or the fast operations.

So, other than these nodes, we also store two more pieces of information. Other than the nodes, we also store the following. A pointer called min to 0.2, the minimum to the node storing the minimum key of the heap, ok.

We maintain a global min pointer which points to the node that stores the minimum key of the heap. And second is a variable count to store the current number of elements, the number of keys. stored in the array in the heap, ok. So, now, with this, you see the min operation, this min operation which is supposed to

return only the minimum key without changing the data structure is obviously O(1) because we are explicitly storing a pointer to the node which stores the minimum key, ok. And we have I think that is enough. So, now, with this, let us discuss the insert procedure. The insert process is very simple.

So, let us see with an example suppose you do insert say 10 suppose we are the keys are integers. So, what we do initially the count was 0 and min was pointing to null. So, what we do we create a node with key 10 node means it has all the fields as described here keys pointed to parents pointed to any one child and so on. and so, the key very key variable stores the key it has no parent.

So, parent is null pointer to parent is null it has no child. So, pointer to child is null and it has no left sibling. So, if it has no left sibling the pointer to left sibling points to itself and pointer to right sibling points to itself and this is the only key. So, min now points to this node and count equal to 1 ok.

So, this is the insert procedure insert 10 in an empty list. What does the make a heap make heap process will do? Make heap process will simply set the mean pointer to null and the count to 0. So, it obviously, takes theta of 1 time. let us do more insert say insert 15.

So, what we will do we will simply. So, you see with pointer to left sibling and pointer to right sibling it looks like a doubly linked list doubly circular linked list. So, we will do simple insertion in this doubly circular linked list using the mean pointer. So, here is 10 I am inserting in doubly circular linked list.

and I check whether the newly entered key is less than the min, if it is not then min pointer also remains same and I simply increase the count. So, this is the insert procedure as simple as this. So, let us do one more insert. So, now I am now I am inserting 5, 5 is less than the minimum which is 10 in the Fibonacci heap. So, I insert it in this doubly circular linked list and update the mean pointer.

This is how it looks like 10, 5, Let us do another insert. again the newly inserted key 1 is less than the minimum. So, you update the min and count is 4 ok. So, clearly insert takes big O of 1 time or theta 1 time in the worst case.

So, Insert takes theta 1 time in the worst case, ok. And we have already argued that make heap which is initializing the empty heap, which is nothing but setting the min pointer to null and the count variable to 0. Make heap and min, which returns the minimum key without deleting it, take theta 1 time

in the worst case. So, if I keep doing insertions, suppose I do 100 insertions, then the Fibonacci heap will look like a 100-size tubular circular linked list. If I do 100 inserts on an initially empty Fibonacci So, this is how a Fibonacci heap after a lot more insertions will look like. So, the main work you see is that insertions are lazy; we are not doing any work. We are just inserting it, and even this thing is not sorted.

So, all this pending work is done during the extract min operation. So, how is the extract min operation performed? So, maybe for concreteness, let us take this. So, extract min. So, let us make this example a little more concrete, and we will see the extract min operation on this.

ok what we do we first delete the ah minimum node ok. If it has any child in this case it does not have any child, but if it has any child we insert those child also in this doubly circular linked list. So, we first delete it and then what we do we consolidate this list that means, what. So, we first we delete the minimum and consolidate the list.

You see after deleting minimum we are forced to work. Why? Because I need to find the new minimum. And, because we have just maintained a doubly circular linked list I have to traverse the whole list which can take a long time.

So, why so, I have I am forced to spend long time why not use this long time to make this list better. So, that subsequent operation is faster. Now, what do we mean by better that is consolidate. After consolidate after consolidation each node in the parent list.

So, this list is called parent list each node of the parent list has different degrees. You see we are keep doing insertion and by degree we mean the number of children that is the number of children. So, you see we are keep doing insertion and so many nodes have 0 degree the number of So, if in consolidation what we do after deleting it whenever I start with 0 and ask is there two node with degree 0.

So, there are two nodes. So, these are the two nodes with degree 0 there are many more, but pick any two node with degree 0 and consolidate or merge these two these two trees these are these are single node tree. So, what we what I do is I have 10 and 5 and it is a mean heap thing mean heap property will maintain that every node the key of every node is at least less than equal to the key of all its children. So, there is only one way to consolidate 5 and 10 it is make 5 the parent and 10 the child.

So, this way you consolidate this you reduce the number of node by 1. you have 2 nodes you have merge them and 5 is now the now the head and 10 is its child. Now, again you see is there any 2 nodes of degree 0 yes again say 12 and 50 you consolidate them again. So, in the next iteration the remaining nodes remain same. So, 12 50 then 70 and 7 . So, you get again 2 nodes with same degree you consolidate them 5 10 is only one way to consolidate 12 and 50 make 12 parent and 50 child 77. Does there exist 2 node in the root list with same degree? Again yes 5 and 12 consolidate them there is only one way make 5 the parent or root.

So, here you have 5 10 and make this tree 1250 a sub tree of 5 77 does there exist 2 node with same degree again yes 70 and 7 again consolidate them. So, here you have 5. 10 12 50 and there is only one way to consolidate 7 and 70 make 7 parent 70 child ok. This is

Fibonacci heap look like after this consolidation cause due to extract mean and then you traverse this root list and update the mean ok and also you update the count. So, in the next class we will start this formal pseudo code of this procedure and also analyze this procedure ok. So, let us stop here. Thank you very much.