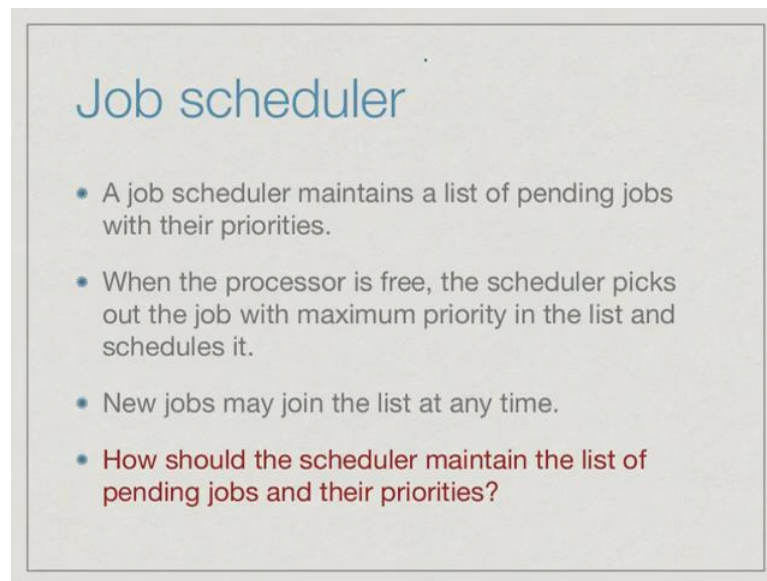


Programming, Data Structure and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 06
Lecture - 05
Priority queues and heaps

(Refer Slide Time: 00:02)



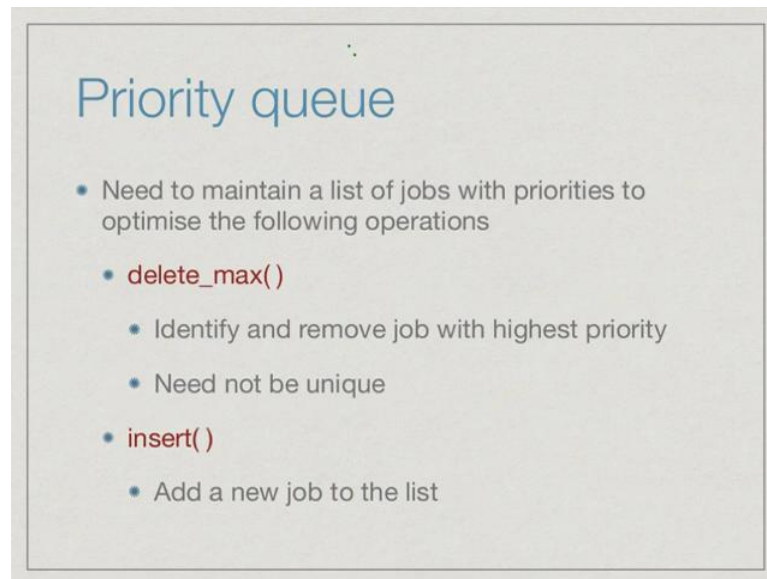
Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities.
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it.
- New jobs may join the list at any time.
- How should the scheduler maintain the list of pending jobs and their priorities?

Let us look at a data structure problem involving job schedulers. Job scheduler maintains a list of pending jobs with priorities.

Now, the job scheduler has to choose the next job to execute at any point. So, whenever the processor is free it picks the job, not the job which arrived earliest, but the one with maximum priority in the list and then schedules it. New jobs keep joining the list, each with its own priority and according to their priority they get promoted ahead of other jobs which may have joined earlier. So, our question is how should the scheduler maintain the list of pending jobs and their priorities? So that it can always pull out very quickly the one with the highest priority to schedule next.

(Refer Slide Time: 00:47)



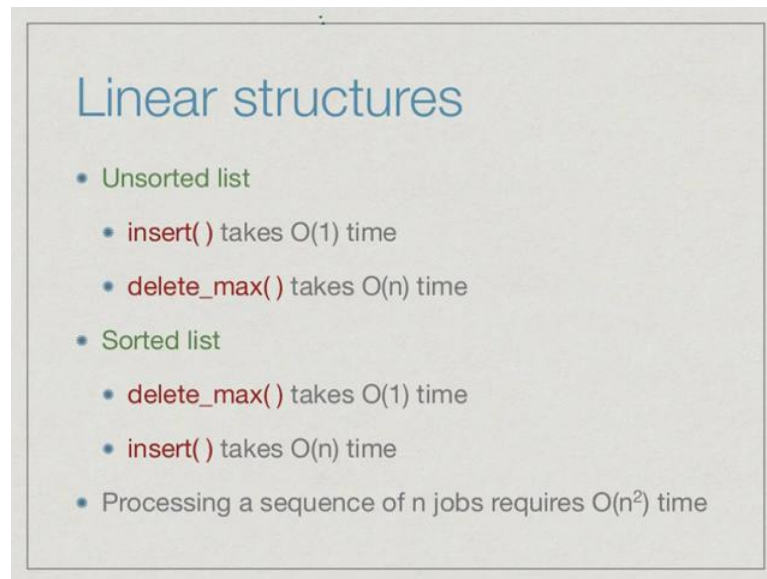
Priority queue

- Need to maintain a list of jobs with priorities to optimise the following operations
 - `delete_max()`
 - Identify and remove job with highest priority
 - Need not be unique
 - `insert()`
 - Add a new job to the list

This is like a queue, but a queue in which items have priority based on some other characteristic not on when they arrived. So, we saw a normal queue is a first-in-first-out object, the **ones that arrive** first leave first. In a priority queue, they leave according to their priority. There are two operations associated with the priority queue, one is delete max. In delete queue we just said we take the element at the head of the queue. In delete max we have to look through the queue and identify and remove the job with **the** highest priority.

Note of course, that **the priorities of two** different jobs may be the same in which case we can pick any one and the other operation is which we normally called add to the queue we will call insert and insert just adds a new job to the list and each job when it is added comes with its own priority.

(Refer Slide Time: 01:39)



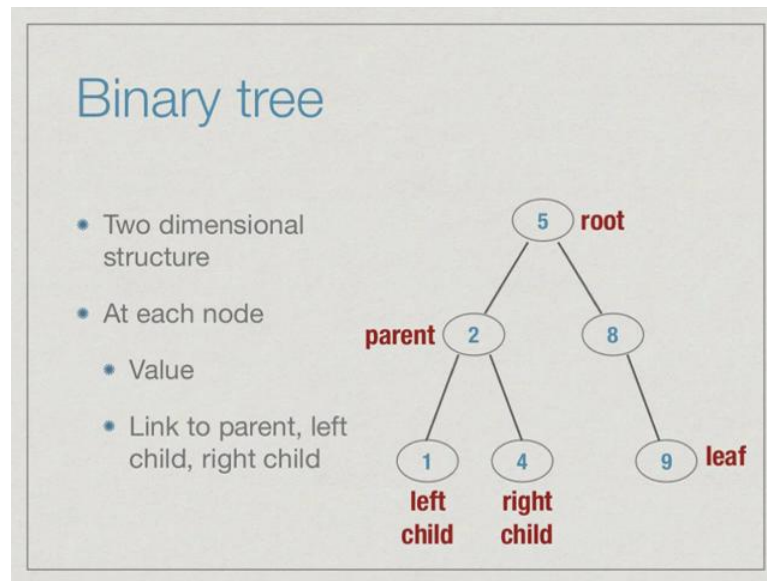
Linear structures

- Unsorted list
 - `insert()` takes $O(1)$ time
 - `delete_max()` takes $O(n)$ time
- Sorted list
 - `delete_max()` takes $O(1)$ time
 - `insert()` takes $O(n)$ time
- Processing a sequence of n jobs requires $O(n^2)$ time

Based on linear structures that we already studied, we can think of maintaining these jobs just as a list. Now, if it is an unsorted list when we add something to the queue we can just add it to the list append it in any position. This takes constant time; however, to do a delete max we have to scan through the list and search for the maximum element and as we have seen in an unsorted list, it will take us order n time to find the maximum element in list because we have to scan all the items.

The other option is to keep the list sorted. This helps to delete max, for instance, if we keep it sorted in descending order the first element of the list is always the largest element. However, the price we pay is for inserting because to maintain the sorted order when we insert the element we have to put it in the right position and as we saw in insertion sort, insert will take linear time. So, as a trade-off we either take linear time for delete max or linear time for insert. If we think of n jobs entering and leaving the queue one way or another we end up spending n squared time processing these n jobs.

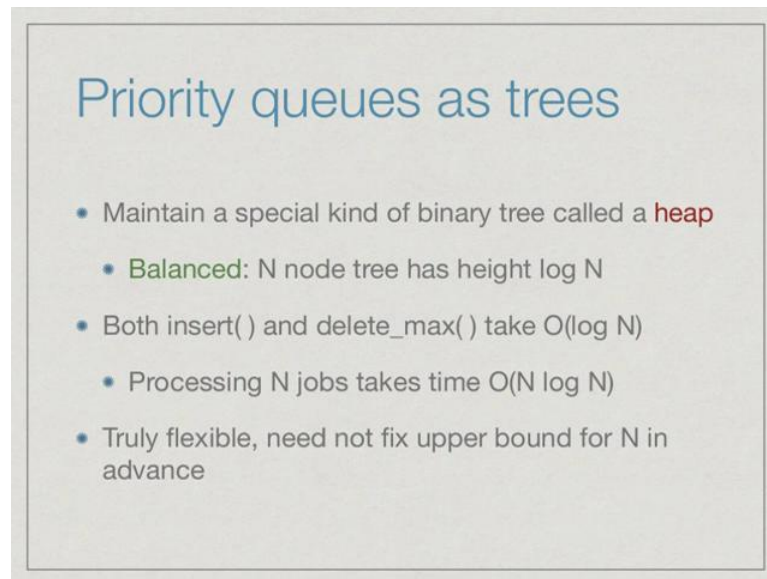
(Refer Slide Time: 02:55)



This is the fundamental limitation of keeping the data in a one dimensional structure.

Let us look at two dimensional structures; the most basic two dimensional structure that we can think of is a binary tree. A binary tree consists of nodes and each node has a value stored in it and it has possibly a left and a right child. We start at the root which is the top of the tree and then each node will have 1 or 2 children. A node which has no children is called a leaf and then with respect to a child we call the parent node, the node above it and we refer to the children as **the** left child and the right child.

(Refer Slide Time: 03:40)



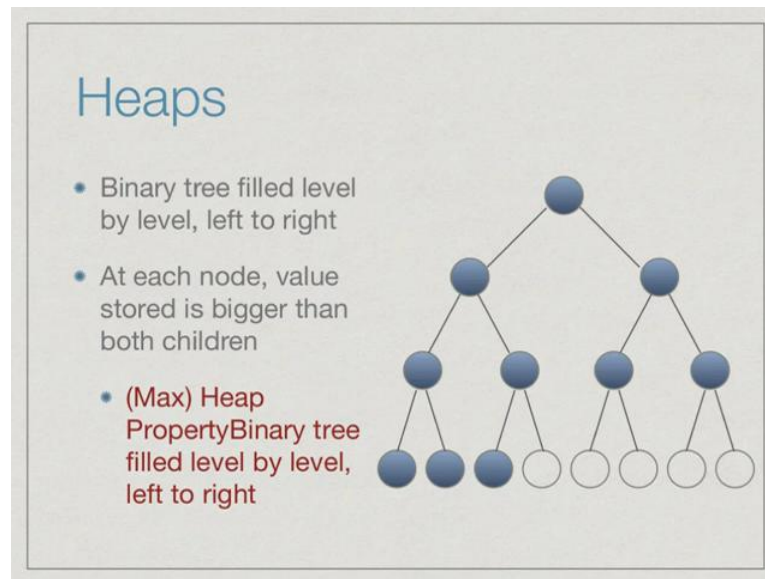
Priority queues as trees

- Maintain a special kind of binary tree called a **heap**
 - **Balanced**: N node tree has height $\log N$
- Both `insert()` and `delete_max()` take $O(\log N)$
- Processing N jobs takes time $O(N \log N)$
- Truly flexible, need not fix upper bound for N in advance

So, our goal **is** to maintain a priority queue as a special kind of binary tree which we will call a heap. This tree will be balanced. A balanced tree is one in which roughly speaking at each point the left and right sides are almost the same size. Because of this it turns out that in a balanced tree, if we have n nodes then the height of the tree will be logarithmic because remember that the height doubles with each level and as a result of which we can fit n nodes in $\log n$ levels provided it is balanced and because it is of height $\log n$, we will achieve both insert and delete max in order $\log n$ time.

This means that if we have to add and remove n elements from the queue, overall we will go from n squared to $n \log n$ and another nice feature about a heap is that we do not have to fix n in advance this will work as the heap grows and shrinks so we do not need to know what n is.

(Refer Slide Time: 04:44)

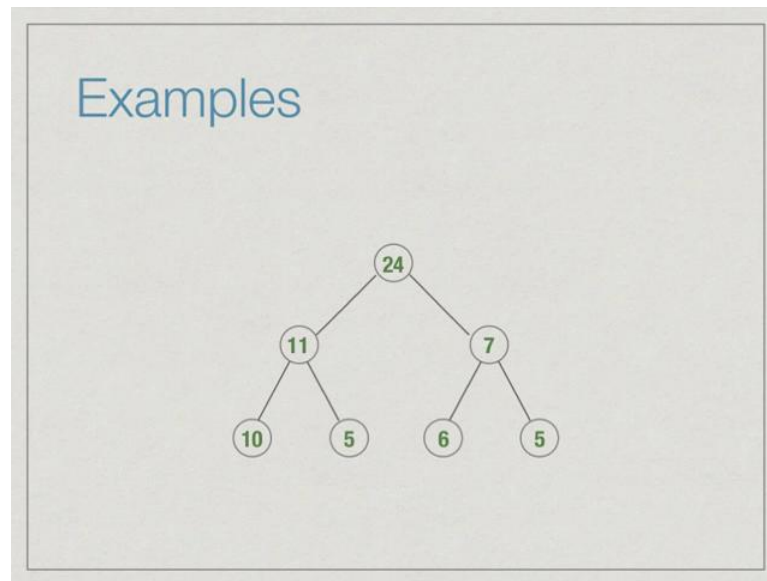


What is a heap? Heap is a binary tree with two properties, the first property is structural: which are the values which are stored in the tree, remember that leaves, nodes in a binary tree may have 0 children, 1 children or 2 children. So, we could have in general a very uneven structure. Heaps have a very regular structure when we have a heap we have a binary tree in which we fill each level from top to bottom, left to right.

In other words, at any point a heap consists of a number of filled levels and then in the bottom level we have nodes filled from left to right and then possibly some unfilled nodes. The other property with the heap, the first property is **structural, it** just tells us how the values look in the heap. In this node, for example, in this picture the blue nodes are those which have values and the empty nodes are indicated with open circles at the bottom.

The second property about the heap is the values themselves. So, the heap property in this case what we call the max heap property because we are interested in maximum values says that every value is bigger than the values of its 2 children. So, at every node if you look at the value and we look at the value in the left child and the right child then the parent will have a larger value than the both the children.

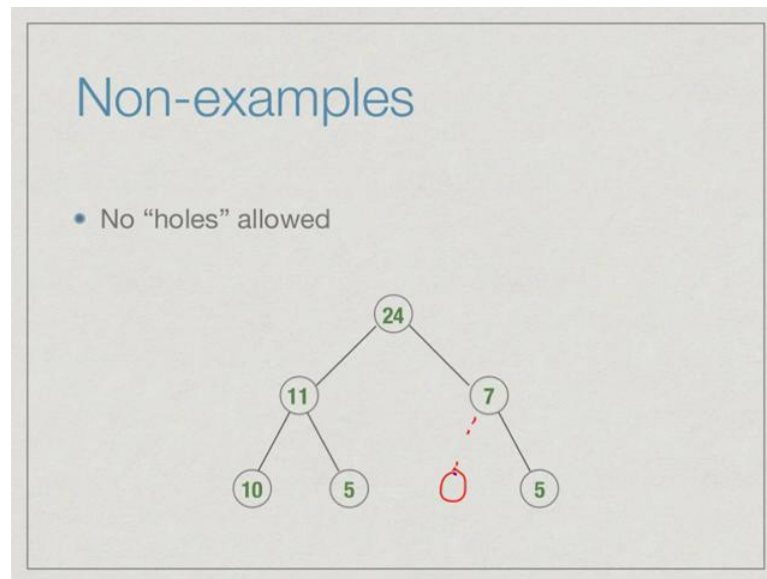
(Refer Slide Time: 06:06)



Let us see some examples. Here is a four node heap, because it has four nodes we fill the root in the first level and finally, in the second level we have only one node which is a left most and notice **that** the values are correctly ordered, 24 is bigger than 11 and 7, 11 is bigger than its only child 10, 7 has no children. So, there are no constraints.

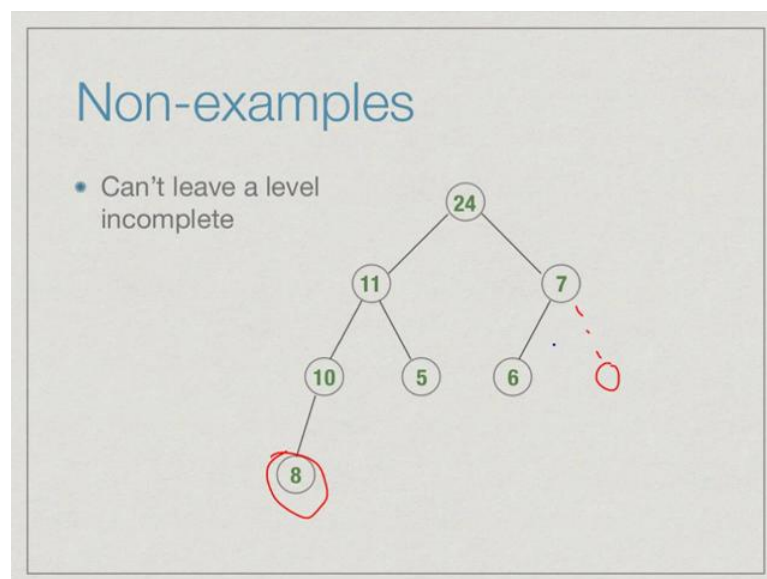
Here is another example where the bottom level is actually full, here we have 7 nodes and once again at every point we see that 11 is bigger than 10 and 5, 7 is bigger than 6 and 5. So, we have the value **property** - the max heap property along with the structural property.

(Refer Slide Time: 06:42)



Here is an example of something which is structurally not a heap because it is a heap we should have it filled from top to bottom, left to right. So, we should have a node here to the left of 7 before we add a right child therefore, this is not a heap.

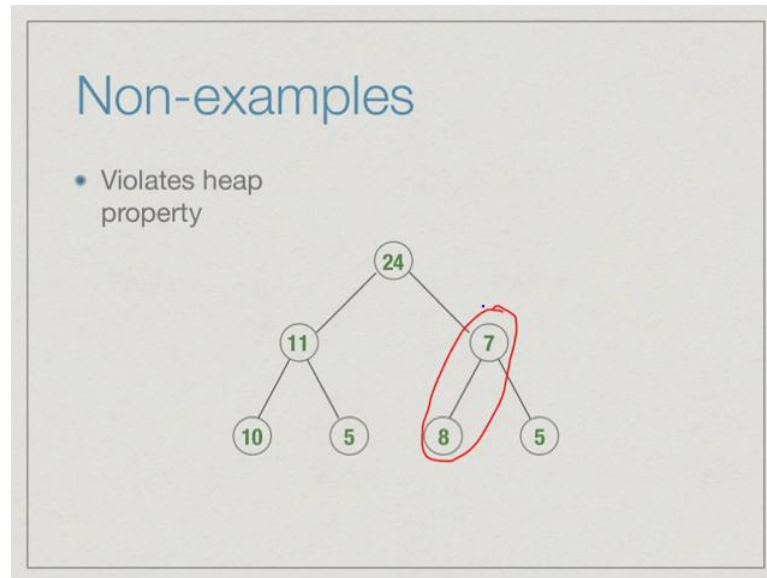
(Refer Slide Time: 06:59)



Similarly, here the node 8 could not have been added here before we filled in the right

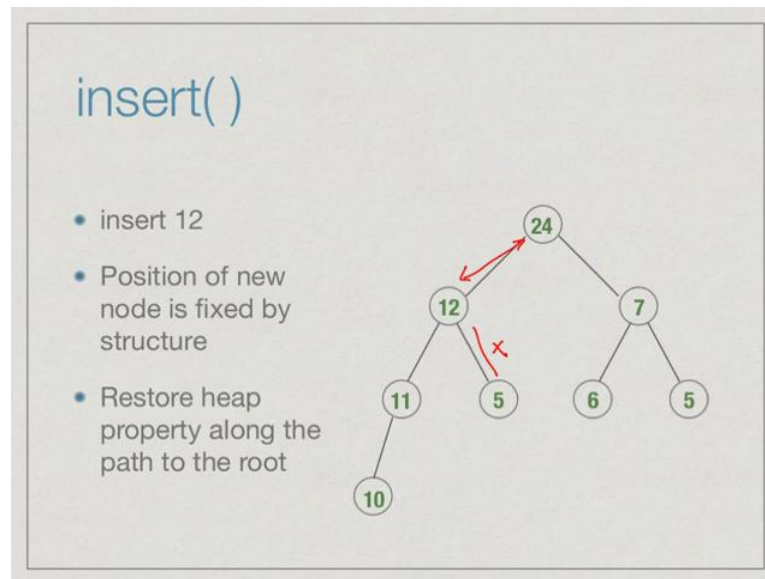
child of 7. So, once again this **has** not been filled correctly left to right, top to bottom and therefore, this is not a heap.

(Refer Slide Time: 07:14)



This particular tree satisfies the structural property of a heap in the sense that it is filled from top to bottom, but we have here a violation of the heap property because 7 has a child which has a larger value **than** it namely 8.

(Refer Slide Time: 07:32)

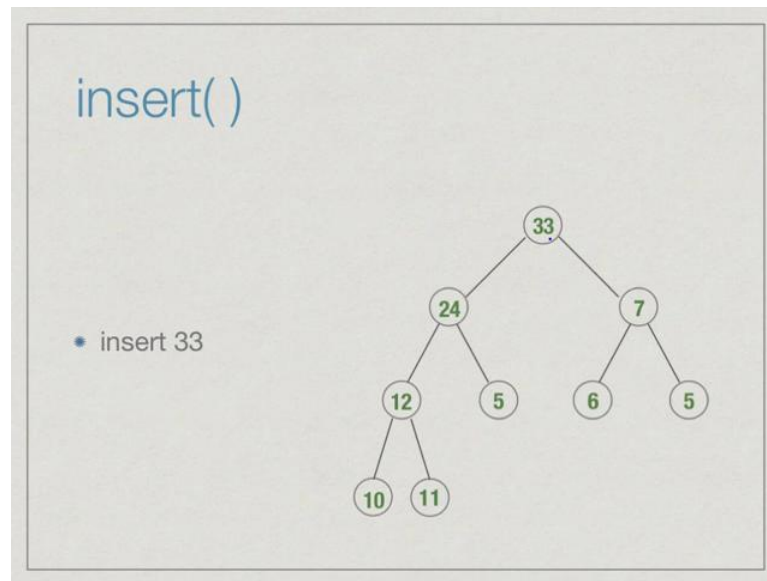


Our first job is to insert a value into a heap while maintaining the heap property. So, the first thing to note is that we have no choice about where to put the new node, remember that heap nodes are constructed top to bottom, left to right. If we want to insert 12 it must come below the 10 to the left because we have to start a new level, since the previous level is full. The problem is that this may not satisfy the heap property; in this case 12 is bigger than its parent 10.

Although this is now structurally correct, it does not have the right value distribution. So, we have to restore the heap property in some way. This is what we do we first create the node, we put the new value into that node and then we start looking at violations with respect to it is parent. We notice that 12 and 10 are not correctly ordered. So, we exchange them, right now this is a new node. We have to check whether it is correctly ordered with respect to **it's** current parent. So, we look and we find that it is not. So, again we exchange these.

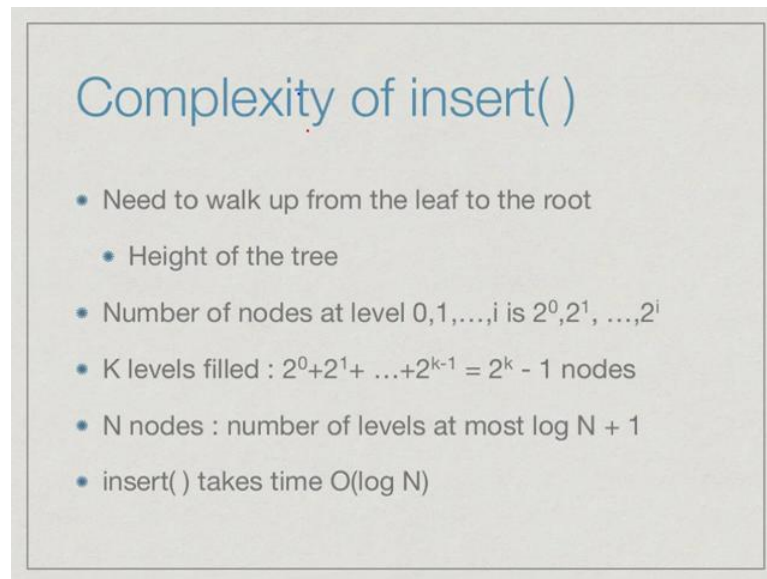
Now, notice that because 11 was already bigger than 5, 12 will remain bigger than 5. There is no need to check anything down from where we got, we only have to look up. Now, we have to check whether there is still a problem above. In this case, there is no problem 12 is smaller than 24. So, we stop.

(Refer Slide Time: 08:59)



Let us add another node. Supposing, we **add** 33 now to the heap that we just created. So, 33 again creates a new node at this point. Now, 33 being bigger than 11 we have to walk up and swap it then again we compare 33 and its parent 12 and we notice that 33 is bigger than 12. So, we swap it again then we look at the root, in this case 24 and we find that 33 is bigger than 24. So, we swap it again and now 33 **has** no parents and it is definitely bigger than it is 2 children. So, we can stop.

(Refer Slide Time: 09:35)



Complexity of insert()

- Need to walk up from the leaf to the root
 - Height of the tree
- Number of nodes at level $0, 1, \dots, i$ is $2^0, 2^1, \dots, 2^i$
- K levels filled : $2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$ nodes
- N nodes : number of levels at most $\log N + 1$
- insert() takes time $O(\log N)$

How much time does insert take? In each time we insert a node, we have to check with its parent, swap, check with its parent, swap and so on, but the good thing is we only walk up a path we never walk down a path. So, the number of steps you walk up will be bounded by the height of the tree.

Now, we argued before or we mentioned before that a balanced tree will have height $\log n$. So, we can actually measure it correctly by saying that the number of nodes at level i is 2 to the i . Initially, we have 1 node 2 to the 0 , then at the first level we have 2 nodes 2 to the 1 and second level we have 4 nodes 2 to the 2 and so on. If we do it this way then we find that when k levels are filled, we will have 2 to the k minus 1 nodes and therefore, turning this around we will find that if we have n nodes then the number of levels must be $\log n$. Therefore, insert walks up a path, the path is equal to the height of the tree, and the height of the tree is order of $\log n$. So, insert takes time order $\log n$.

(Refer Slide Time: 10:42)

delete_max()

- Maximum value is always at the root
- From heap property, by induction
- How do we remove this value efficiently?

```
graph TD; 33((33)) --- 24((24)); 33 --- 7((7)); 24 --- 12((12)); 24 --- 5_1((5)); 12 --- 10((10)); 12 --- 11((11)); 7 --- 6((6)); 7 --- 5_2((5));
```

The other operation we need to implement in a heap is delete max. Now, one thing about a heap is that the maximum value is always at the root this is because of the heap property you can inductively see that because each node is bigger than **it's** children the maximum value in the entire tree must be at the root. So, we know where the root is; now the question is how do we remove it efficiently?

(Refer Slide Time: 11:10)

delete_max()

- Removing maximum value creates a "hole" at the root
- Reducing one value requires deleting last node
- Move "homeless" value to root

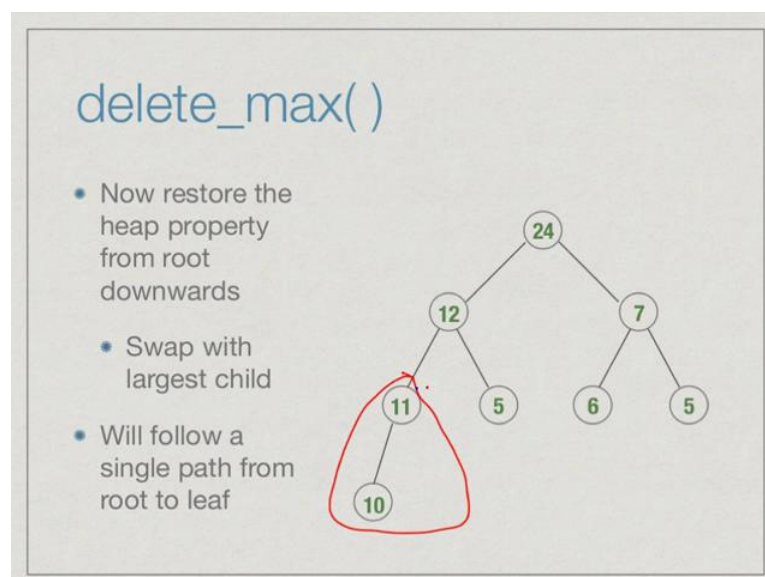
```
graph TD; 11((11)) --- 24((24)); 11 --- 7((7)); 24 --- 12((12)); 24 --- 5_1((5)); 12 --- 10((10)); 12 --- 11_1((11)); 7 --- 6((6)); 7 --- 5_2((5));
```

If we remove this node, first of all we cannot remove the node because it is a root. If you remove this value then we have to put some value there. On the other hand, the number of values in the node in the heap has now shrunk. So, this node at the bottom right must be deleted because the structural property of the heap says that we must fill the tree left to right, top to bottom. We are going top to bottom and we have run out of a value.

The last node that we added was the one at the right most end of the bottom row and that must go. So, we have a value which is missing at the top and we have a value at the bottom namely 11 whose node is going to be deleted. So, the strategy now is to move this value to 11 and then fix things, right. So, we first remove the 33 from the root, we remove the node containing 11 and we move the 11 to the position of the root.

Now, the problem with this is we have moved an arbitrary value not the maximum value to the top. Obviously, there is going to be some problem with respect to its children. So, here it turns out that 11 is bigger than 7 which is correct, but unfortunately it is smaller than 24.

(Refer Slide Time: 12:19)

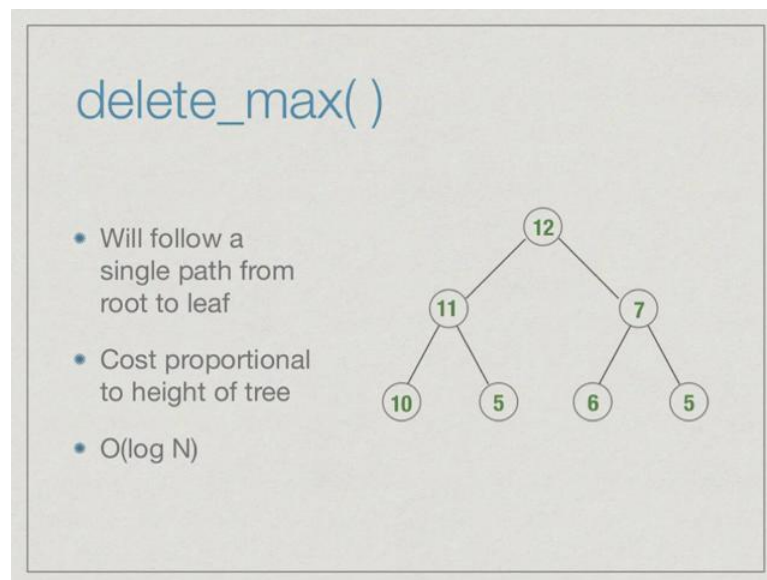


To restore the property what we do is, we look at both directions right and we exchange it with the largest child. Suppose, this had been 17 here then we could swapped 11 with

17 here or 11 with 24, both violations are there, but if you move this smaller child up then 17 will not be bigger than 24, so we move the largest one.

In this case we move 24 up right and now we have 11 again and now we have to again check whether it is correct with respect to its 2 children, again it is not. So, we move the largest one up namely 12 and then we see now whether it is correct with respect to its children. At this point 11 is bigger than 10. So, we stop.

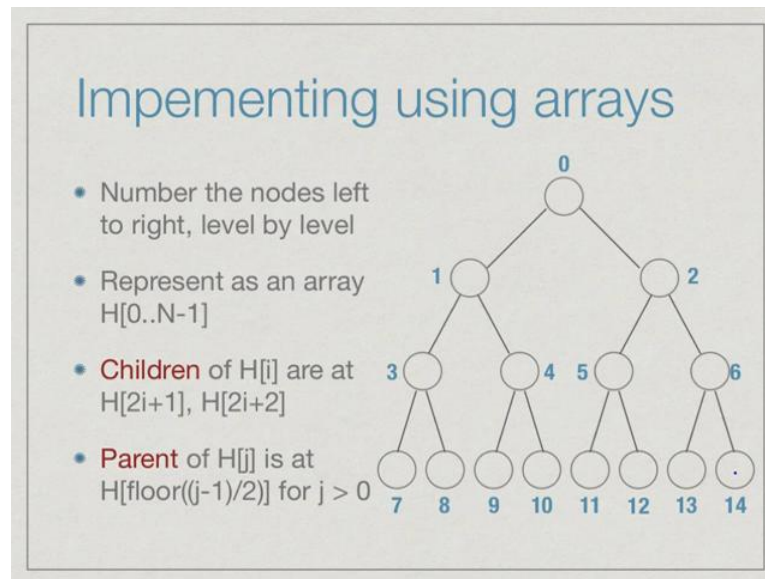
(Refer Slide Time: 13:03)



Just as insert followed a single path from the new node at the leaf up to the root, delete max will follow a single path from the root down to a leaf.

Once again the cost of delete max will be proportional to the height of the tree which as we said earlier is $\log n$. Let us do another delete; we delete, in this case 24, now we remove the node for 10, 10 goes to the root. We compare 10 with its 2 children, 12 and 7 and find that it is not satisfying heap property. So, we move the larger of the two up namely 12. Now, we look at its children, new children here 11 and 5 and again we see it is not satisfying the property. So, the larger one moves up and once it reaches the leaf there are no properties to be satisfied anymore. So, we stop.

(Refer Slide Time: 13:56)

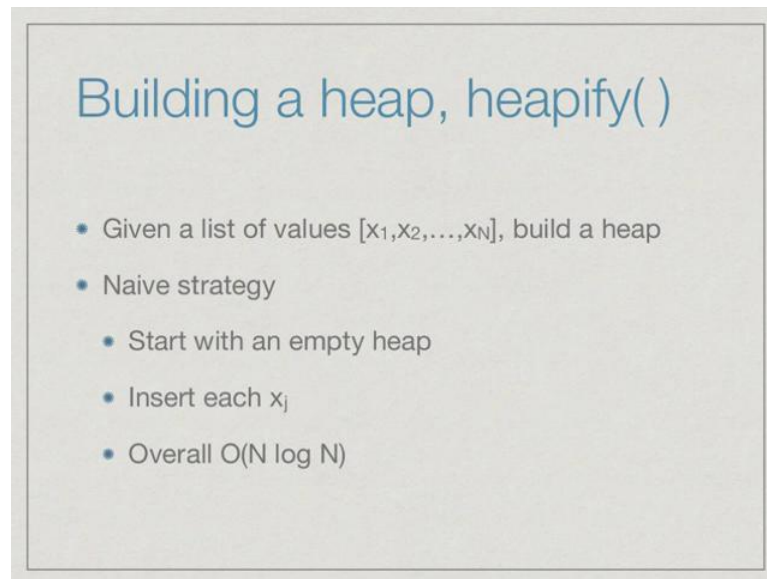


One very attractive feature of heaps is that we can implement this tree directly in a list or in an array. So, we have an n node heap, we can represent it as a list or an array with position 0 to n minus 1. The position 0 represents a root then in order 1 and 2 represent the children, then 3, 4, 5, 6, 7 nodes are the next level and so on. So, just as we said we filled up this heap left to right, top to bottom right. In the same way, we number the nodes also top to bottom, left to right. So, we start with 0 at the root, then 1 on the left, 2 on the right then 3, 4, 5, 6, 7, 8, 9, 10 and so on.

From this you can see that, if I have a position labeled i then the two children are $2i + 1$ and $2i + 2$ right. So, the children of 1 are $2 \times 1 + 1$, which is 3 and $2 \times 1 + 2$, which is 4. Similarly, children of 5 are $2 \times 5 + 1$, which is 11 and $2 \times 5 + 2$ which is 12. So, just by doing index calculations for a position in the heap we can figure out where its children are and by reversing this calculation we can also find the index of the parent, the parent of j is that j minus 1 by 2. Now, j minus 1 by 2 may not be an integer. So, we take the floor.

If we take 11, for example, 11 minus 1 is 10, 10 by 2 is 5. If we take 14, for example, 14 minus 1 is 13, 13 by 2 is 6.5 we take the floor and we get 6.

(Refer Slide Time: 15:32)



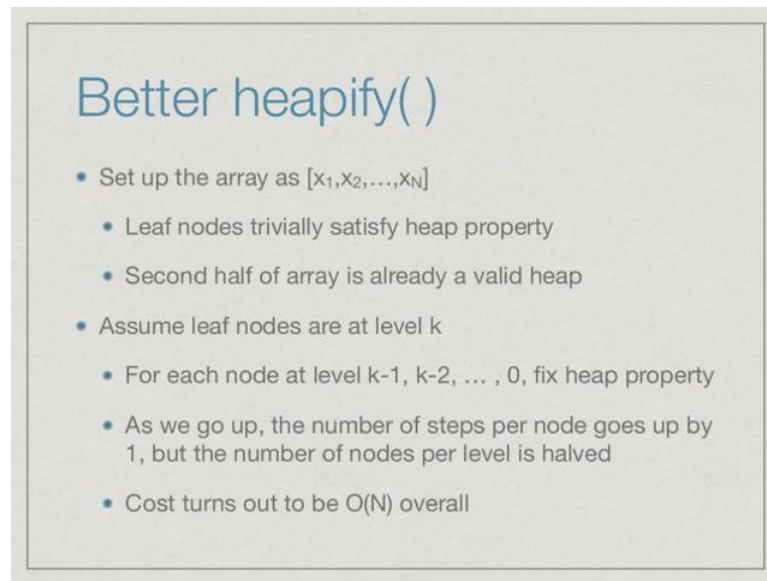
Building a heap, heapify()

- Given a list of values $[x_1, x_2, \dots, x_N]$, build a heap
- Naive strategy
 - Start with an empty heap
 - Insert each x_j
 - Overall $O(N \log N)$

This allows us to manipulate parent and children nodes by just doing index arithmetic we go from i to $2i + 1$, $2i + 2$ to go to the children and we go from j to $j - 1$ by 2 floor to go to the parent. How do we build a heap. A naive way to build a heap is just to take a list of values and insert them one by one using the heap operation into the heap.

So, we start with an empty heap, we insert x_1 , create a new heap containing x_1 , we insert x_2 , creating a heap of x_1, x_2 and so on. Each operation takes $\log n$ time of course, n will be growing, but it does not matter if we take the final n as an upper bound we do n inserts each just $\log n$ and we can build this heap in order $n \log n$ time.

(Refer Slide Time: 16:23)



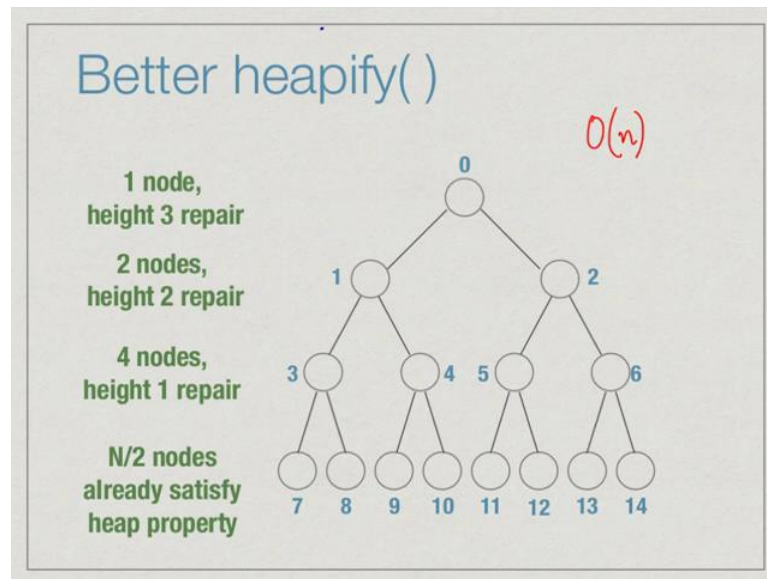
Better heapify()

- Set up the array as $[x_1, x_2, \dots, x_N]$
 - Leaf nodes trivially satisfy heap property
 - Second half of array is already a valid heap
- Assume leaf nodes are at level k
 - For each node at level $k-1, k-2, \dots, 0$, fix heap property
 - As we go up, the number of steps per node goes up by 1, but the number of nodes per level is halved
 - Cost turns out to be $O(N)$ overall

There is a better way to do this heap building if we have the array as x_1 to x_n then the last half of the nodes correspond to the leaves of the tree. Now, a leaf node has no properties to satisfy because it has no children. We do not need to do anything we can just leave the leaves as they are.

We go one level above and then we can fix all heap errors at one level above right and then again we move one level above and so on. So, we do the kind of top to bottom heap fixing that we did with the delete max, while we are building the heap. So, as we are going up the number of steps that we need to propagate this error goes higher and higher because we need to start at a higher point on the other hand the number of nodes for which this happens is smaller.

(Refer Slide Time: 17:19)

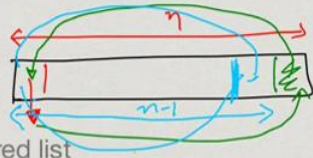


Let us look at this here. What we are saying is that if we start with the original list of say elements 0 to 14, then the numbers 7 to 14 already satisfy the heap property. Whatever values there are, we do not need to worry, then we go up and we may have to swap this with its children, we may have to swap this with its children and so on. For 4 nodes we have to do one level of shifting perhaps to repair the heap property then we go up now 1 and 2 are the original values, they may be wrong. So, again we may have to shift it down one value and then another value.

Now, we need two levels of shifting, but we have only two nodes for this. The number of nodes for which this is required is shrinking, it's halving actually and the number of steps for which we have to do it is increasing by 1. We will not do a careful calculation here, but it turns out that as a result of this, in this particular way of doing the heapify by starting from the bottom of the heap and working upwards rather than inserting one at a time into an empty heap actually takes us only linear time order n .

(Refer Slide Time: 18:30)

Heap sort



- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$
- After each `delete_max()`, heap shrinks by 1
 - Store maximum value at the end of current heap
 - In place $O(n \log n)$ sort

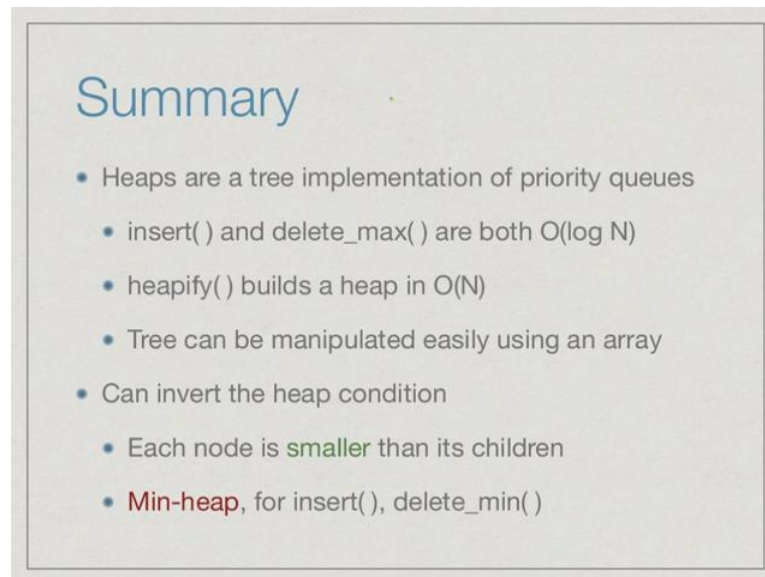
A final use of heap is to actually sort, we are taking out one element at a time starting with maximum one. It is natural that **if** we start with a list, build a heap and then do n times delete max we will get the list of values in descending order. We build a heap in order n time, **call** delete max n times and extract the elements in descending order. So, we get an order $n \log n$ algorithm for heap.

Now, the question is where do we keep these values. Well, remember that a heap is an array. Initially, we have a heap which has n elements. So, we build this heap now we said that the delete max will remove the element **at** the top because that is the root, but it will also create a vacancy here, this is the value that will go to the top, this is the last leaf which will go the top when we fix the delete max.

Since there is a vacancy here we can actually move this to this position, the maximum value will now go to the end of the heap, but the next time we process the heap there will be only n minus one values. So, we will not look at that value we will just use from 0 to n minus 2. Again this value will come here this value will go there and now we will have two elements fixed and so on. So, one by one the maximum value, second maximum value and so on will get into the same list or array in which we are storing the heap and eventually the heap will come out in ascending order.

This is actually an $n \log n$ sort. It has same as asymptotic complexity as merge sort we saw before and unlike merge sort which forced us to create a new array everytime we merge two lists, this is actually creating a list in place.

(Refer Slide Time: 20:20)



The slide is titled "Summary" in a blue font. It contains a list of seven bullet points:

- Heaps are a tree implementation of priority queues
- `insert()` and `delete_max()` are both $O(\log N)$
- `heapify()` builds a heap in $O(N)$
- Tree can be manipulated easily using an array
- Can invert the heap condition
- Each node is smaller than its children
- **Min-heap**, for `insert()`, `delete_min()`

To summarize heaps are a tree based implementation of priority queues in which both insert and delete max can be done in $\log n$ time. We can do a bottom up heapify to build a heap in order n time and these are trees, but they can be manipulated very easily using an array. Now, in this case we were looking at max heaps; we can also do a dual construction where we change the heap condition to say that each element must be smaller than its children, in which case we have what is called a min-heap and then instead of delete max, the operation we perform is delete min. Everything is exactly symmetric to the other case.

When we move something up we have to move it down according to the min condition and when we insert something at the bottom we have to move it up right. So, the insert and delete min work exactly like insert and delete max, except that the comparisons are reversed because we now have a min condition rather than the max condition locally and finally, we saw that with a heap we can do sorting in order $n \log$ time in place.