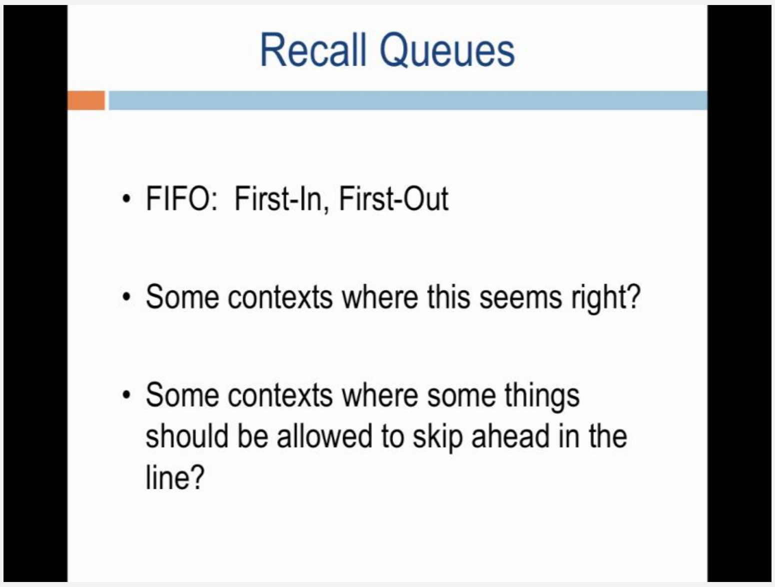
Programming and Data Structures

Prof. N. S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 18
Heaps

In today’s lecture, let us look at the second of the advanced data structures, non linear data structures that we studying this week which is the heap data structure. As you will see again like the binary search tree data structure, once you have a binary tree implementation, a heap data structure is actually fairly simple to implement. In particular, you will see that the heap data structure can be very easily implemented in an array and I encourage all of you to implement the heap data structure and all the methods that you see here. As usual as this is the theme for this week, the code will be within the presentation itself, I am not going to show you any C code or any execution or any compilation. So, I encourage you to implement the heap data structure, it is very easy to implement and it is a very powerful data structure.

(Refer Slide Time: 00:58)



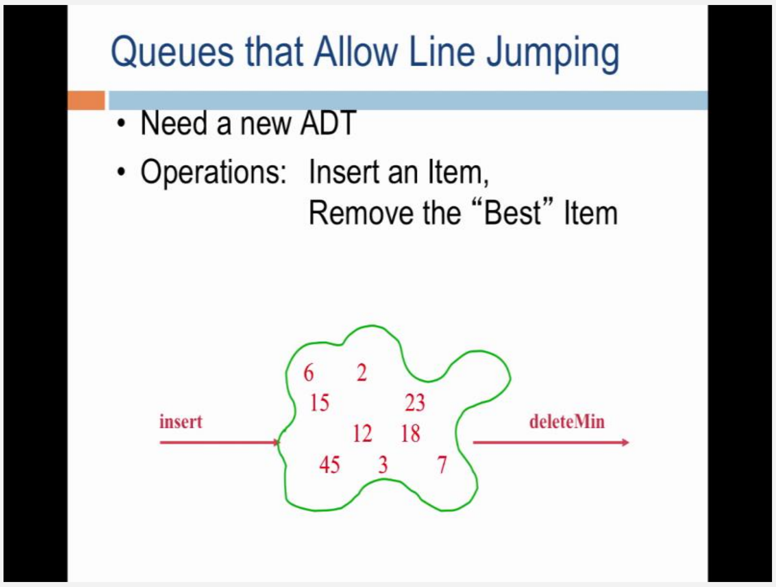
Recall Queues

- FIFO: First-In, First-Out
- Some contexts where this seems right?
- Some contexts where some things should be allowed to skip ahead in the line?

So, let us look at the heap abstract data type and it is implementation in today's lecture, as we will also recall quite of you things that we have seen, you recall that queue is a first in first out data structure or a data type, and it is very useful write a many contexts. For example, if you want to implement the traffic on a street with a single lane, then the queue is the best way, where at the signal only the vehicle at the front can exit.

And on the other hand, if you have two lanes, then you might want to have a special kind of a data structure which behaves like a queue, but has priorities. For example, an ambulance on a road with two lanes has high priorities and if all the vehicles are using one lane and the second one is an ambulance lane, the ambulance lane can go ahead whenever required.

(Refer Slide Time: 01:52)



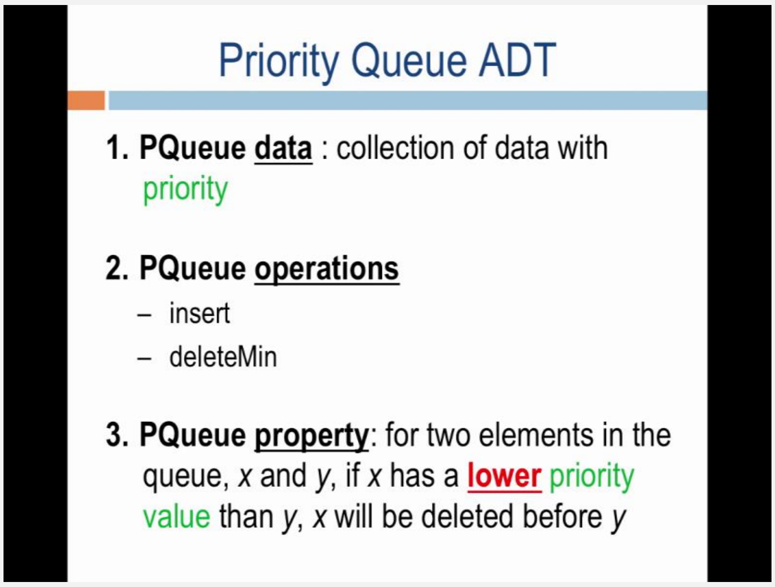
Queues that Allow Line Jumping

- Need a new ADT
- Operations: Insert an Item,
  Remove the “Best” Item

insert
6
2
15
23
12
18
45
3
7
deleteMin

So, that is we are going to talk about queues that allow you to jump the lines or go ahead. So, the basic operations in a queue of this form is that we will insert an element and the goal is always to remove the best or the most important or the element with the highest priority. So, here is a set of keys you want to insert the set of keys and at every step you want to quickly delete the one with the least key, delete men and delete men will model removing the best element or the one with the highest priority.

(Refer Slide Time: 02:36)



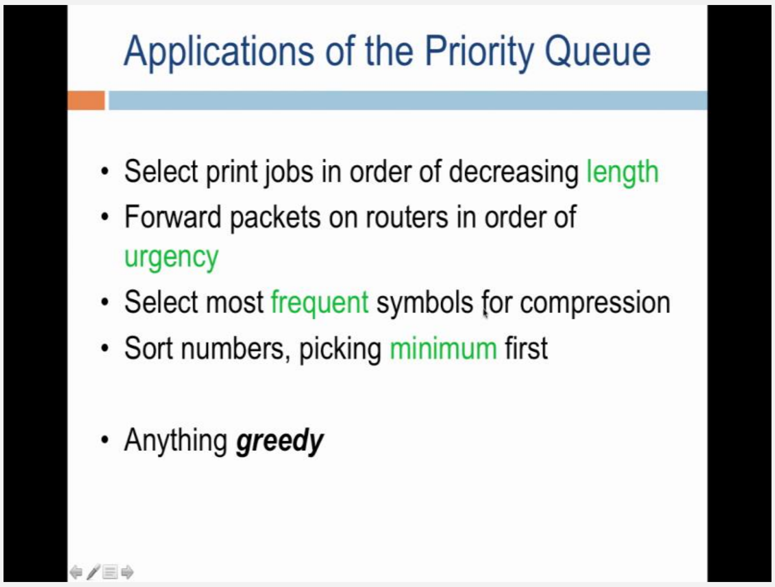
Priority Queue ADT

1. **PQueue <u>data</u>** : collection of data with priority
2. **PQueue <u>operations</u>**
   - insert
   - deleteMin
3. **PQueue <u>property</u>**: for two elements in the queue, *x* and *y*, if *x* has a **<u>lower</u>** priority value than *y*, *x* will be deleted before *y*

This is the priority queue abstract data type, we maintain data with a special value called the priority value, the priority queue data type has two operations. So, the data remember is data in which every element has a certain value, an integer value called the priority. We also implement the priority queue operation which are inserted into the priority queue and always delete the element with the minimum priority. Observe that the priority queue is a special kind of a queue in…

Observe that the priority queue is more general than a queue because in the queue, the priority is whoever came first has the higher priority. If therefore, the queue that we were studied is a special kind of a priority queue, where the priorities depend upon the order in which the data items arrive to enter into the queue. The most important property of the priority queue is that if two elements have… There are two elements x and y and x has a lower priority value than y, then x will be deleted before y. Now, observe that in English this can be a bit confusion that is why lower and priority value have been put in red and green. So, we are doing the delete min, therefore, we will always delete an element with a low priority value.

(Refer Slide Time: 04:07)



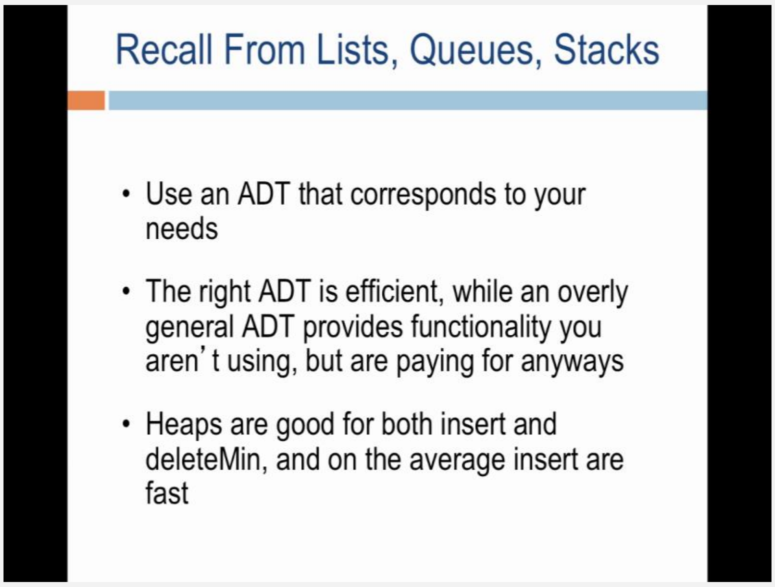
Applications of the Priority Queue

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Select most frequent symbols for compression
- Sort numbers, picking minimum first

- Anything ***greedy***

Now, the applications of the priority queue are many, if you wanted to print the elements of an order of decreasing length or if you wanted to identify the most frequent symbols for comparison or if you want to send packets based on their importance or if you wanted to sort the element, where picking the smallest value first, this is called a heap short algorithm. And it is repeatedly used in many greedy algorithms like if you look at the minimum spanning tree algorithms. Though you have not studied in this course, we study those algorithms the next value of the next smallest value is always extracted from a priority queue kind of a data structure.

(Refer Slide Time: 04:47)

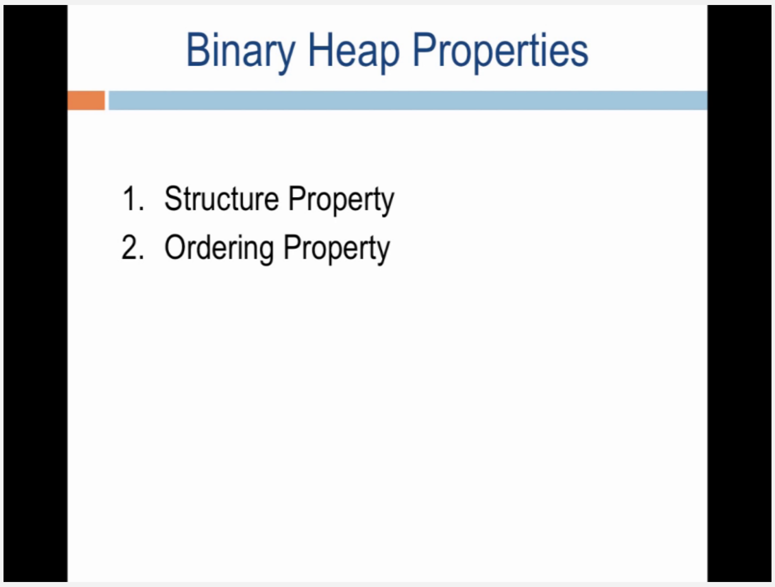


Recall From Lists, Queues, Stacks

- Use an ADT that corresponds to your needs
- The right ADT is efficient, while an overly general ADT provides functionality you aren't using, but are paying for anyways
- Heaps are good for both insert and deleteMin, and on the average insert are fast

So, we are going to talk about ADT's, so we are going to define the priority, so we have already defined the priority queue ADT, and then we will show an implementation of insert and deleteMin.

(Refer Slide Time: 05:00)

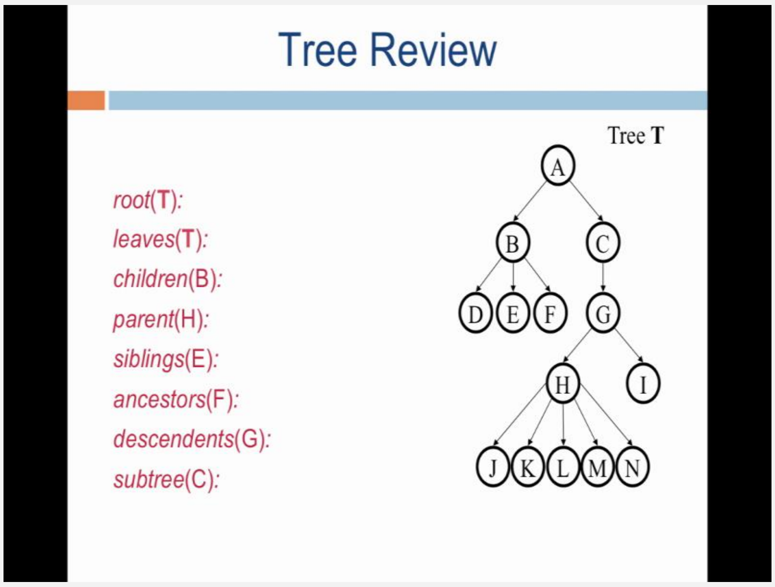


Binary Heap Properties

1. Structure Property
2. Ordering Property

So, let us look at the heap properties, first of all it is a binary heap and there is a structure property that is the underline tree, like it was there in the binary search trees and then there is an ordering property. Ordering property in a heap is different from the ordering property in a binary search tree, this is one very important thing to be kept in mind.

(Refer Slide Time: 05:24)



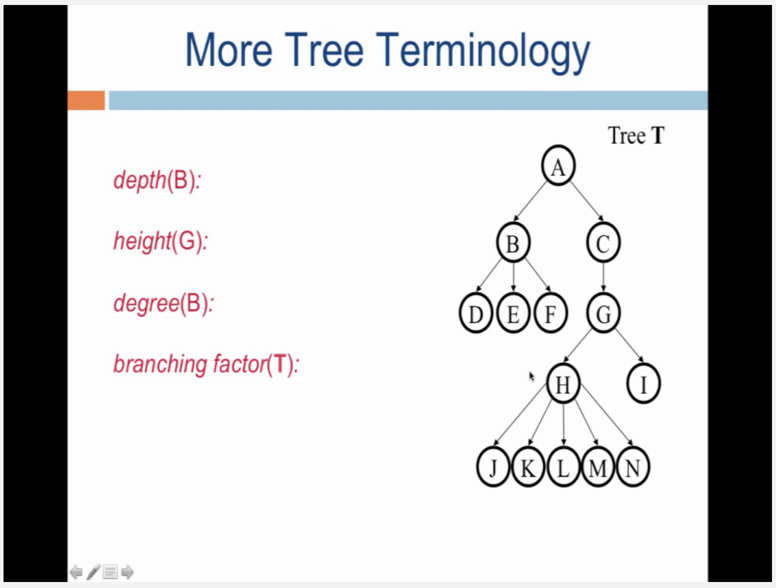
Tree Review

root(**T**):
leaves(**T**):
children(B):
parent(H):
siblings(E):
ancestors(F):
descendents(G):
subtree(C):

Let us quickly review the parameter associated with the trees, this is a tree as we saw in the earlier lectures, this node is the root of the tree, these are all the descendants of the particular node, this node is a left child of this node and this node is a right child, we are talking of binary trees. But, when you look here, all these nodes are of children of this particular node, so the notion of left child and right child would not exist and all the child nodes with share a common parent are called siblings.

And then there is a ancestor relationship between nodes, for example, C the node containing C is an ancestors of the node containing L and for L the ancestors are all these nodes right up to the root. Now, you can also see that the sub tree at this particular node is this whole tree for which this element is a root. Now, here is a numeric parameter associated with a tree those are all structural parameters. The relationship between them, the name relationships you can say something is a parent. For example, this node is a child of this node, this node containing B is the, parent of this node containing F. So, these are all structural relationships.

(Refer Slide Time: 06:39)



More Tree Terminology

depth(B):

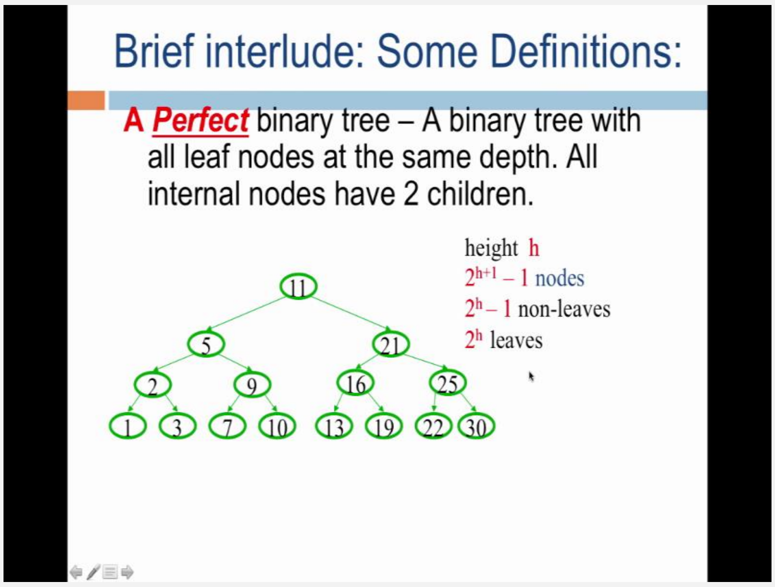
height(G):

degree(B):

branching factor(**T**):

Here are some numerical values associate with a tree, you can define the height of a particular node, you can define the height of G, which is from the leaf level the height is 2, the depth of this node is from the root the depth is 2, the degree of B, you can that B has 3 children, so the degree of B can be defined to be 3, the branching factor is the number of branches that the node has. So, the node with the largest number of children is defined to be the branching factor of the tree.

(Refer Slide Time: 07:14)



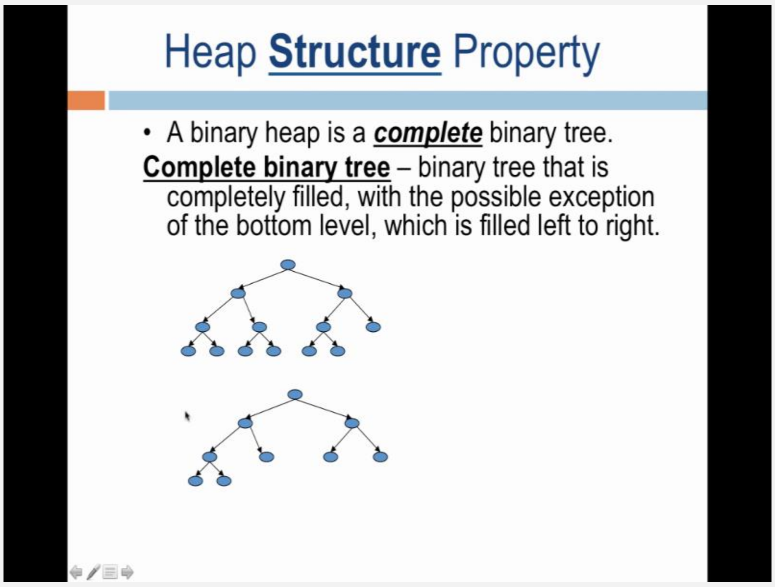
Brief interlude: Some Definitions:

A ***Perfect*** binary tree – A binary tree with all leaf nodes at the same depth. All internal nodes have 2 children.

height h
$2^{h+1} - 1$ nodes
$2^h - 1$ non-leaves
2^h leaves
11
5
21
2
9
16
25
1
3
7
10
13
19
22
30

So, before we looked at heap exactly as we did when we studied binary search trees, let us look at some structural special kinds of trees. Here is a perfect binary tree, in other words it is a full binary tree, all the nodes are there at height h there are 2 power h leaves 0, 1, 2 and 3. As you can see there are 8 leaves and there are 7 internal nodes or 7 non leaves, the total number of nodes is 16 minus 1. So, the height of tree is 3, the root is at level 0, these nodes are at 1, these nodes are at 2 and these nodes are at 3.

(Refer Slide Time: 08:00)



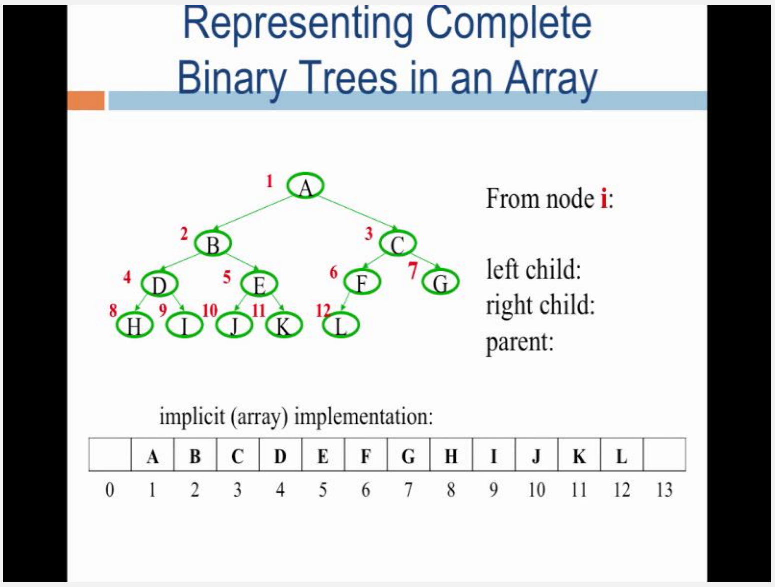
Heap **Structure** Property

- A binary heap is a ***complete*** binary tree.

Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

We saw the complete binary tree concept, when we looked at the binary search tree. We saw the best structure for binary search tree to maintain is a complete binary tree that controls the fine time of the binary search tree. These are two examples of complete binary search trees. The tree is completely filled except that the bottom most level where it is filled from the left to the right.

(Refer Slide Time: 08:29)



Representing Complete Binary Trees in an Array

From node **i**:

left child:
right child:
parent:

implicit (array) implementation:

	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

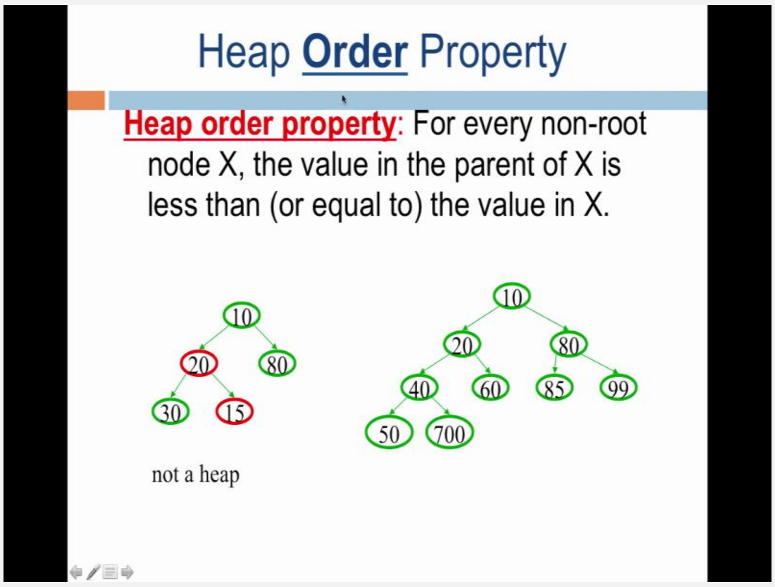
A complete binary tree can be very easily implemented and this is the central concept in the implementation of a heap. So, the structural data of a heap is the following. If you think of a complete binary tree, so whenever you think heap, you think a complete binary tree and whenever you think a complete binary tree, you think of the following implementations. You think of an array, the array in programming language is like C allows you to be index at 0.

Whereas, when you implement a heap, when you implement a complete binary tree, you always start off with a data items to be stored at the array index 1 and the rule is very simple. The left child of a node is always at two times the parent and the right child is two times the parent plus 1. For example, A is the root, the left child is at location 2, index 2 and it is B, the right child is 3 and the right child is the value C and it is at index 3 in the array, these are the left child and right child.

Let us look at B, twice the index is 4 which is the left child of B and twice the index plus 1, which is 5 is the right child of B containing the value E. So, let us see this E is a 5 and D is at 4. Let us look at 3, it is 6 and 7 and let us look at the left and right child of this particular node, it is at 10 and 11. So, E is here, that is node at element 5 and the node of element 10 and 11 J and K are the left and right child of this particular node. This is a very, very important concept. As you can see, this is very easy to implement, there is no implementation to be done at all.

Once, you define an array and if you put the values in to the array, you are essentially created a complete binary tree. This is something, but you should keep that in mind. Now, you should ask, what is a difference between a complete binary tree and a heap? The answer is organization of the data. So, recall that there is a structure property and there is an order property. Once you enforce the correct order property on a complete binary tree, one gets a heap.

(Refer Slide Time: 10:51)



Heap Order Property

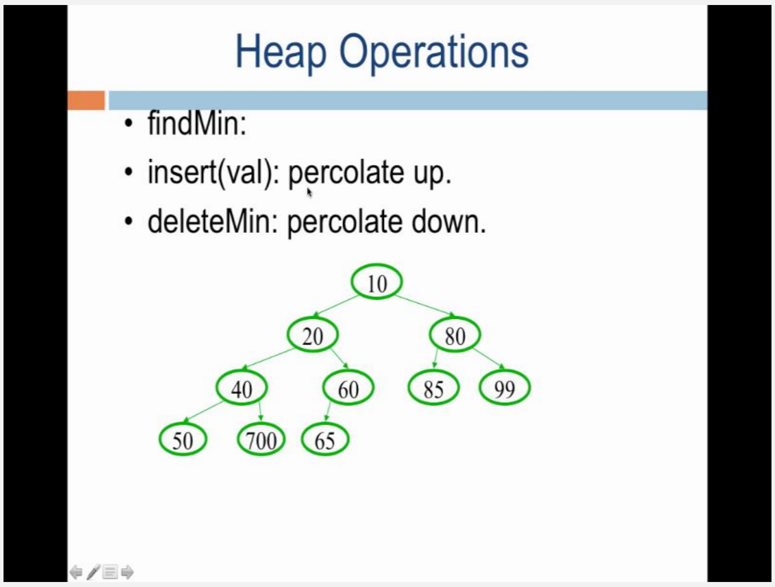
Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.

10
20
80
30
15
not a heap
10
20
80
40
60
85
99
50
700

So, here is a heap order property. So, what do you need know is a structure property of heap is complete binary tree, what is a heap order property, for every non root node X, the value in the parent of X is less than or equal to the value n X. So, this is the heap order property, let us look at this is a heap. So, let us look at this, the parent of this node is less than or equal to the value at this particular node.

Let us take any node; the value at this node is more than the value at it is parent. Therefore, this is a heap; first of all, it is the complete binary tree and it also has a heap order property. This is also complete binary tree, but not a heap, why is it not a heap, because if you look at 15, it is parent has a value which is more than in it. Therefore, this is not a heap; it does not a respect the heap order property.

(Refer Slide Time: 11:54)

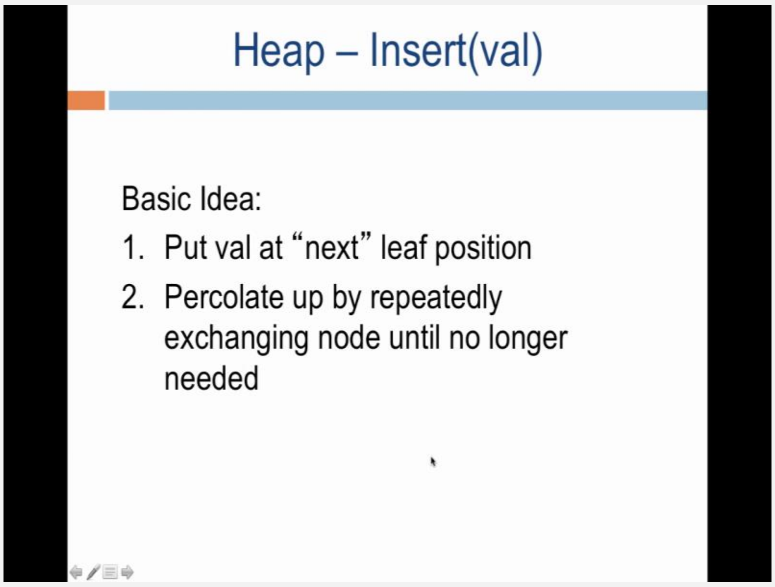


Heap Operations

- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.

So, we have quickly moved. So, basically since we understand non-linear data structure and binary tree is very well, we have quickly move into this sophisticated data structure. So, there are two kinds of heap operations, one is an insert the value, the other one is a delete men. So, therefore, whenever you want to create a heap, you repeatedly insert values into the heap and we will see how to do it and whenever a query to delete men is made heap is modified and the minimum element is deleted from the heap.

(Refer Slide Time: 12:32)



Heap – Insert(val)

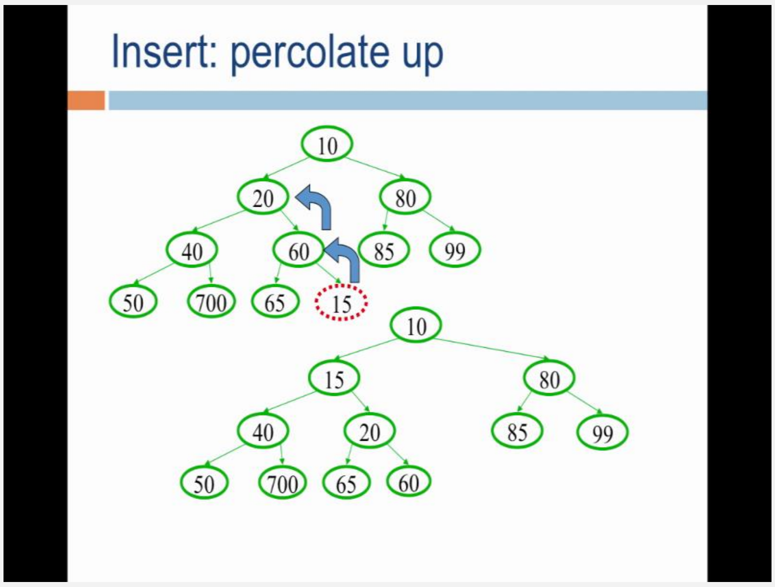
Basic Idea:

1. Put val at "next" leaf position
2. Percolate up by repeatedly exchanging node until no longer needed

So, let us see how to implement these heap operations, let as look at inserting a heap. So, now, let me ask this question, so that you can answer this, when you see heap what you think off, the answer is, you think of array. Now, plus the heap order property, so the array index at 1 up to the largest index are contains a value the heap, gives you complete binary tree.

Along with the order property, this complete binary tree is visualize by the person as a heap, in a program the heap is implemented always inside an array, you can choose implemented using some other methods, but it is very easily implementable as an array. Now, let say I want to implement is insert function and the argument to the insert function is a value. So, what is a basic idea, the basic idea is take the value there and put it at the next leaf position, what do mean by putting it to the next leaf position, simply inserted into the array at the last index or the first non empty index, whose value is more than 0. Now, you move this value repeatedly by exchanging the node until no longer needed, we will just see this.

(Refer Slide Time: 13:52)

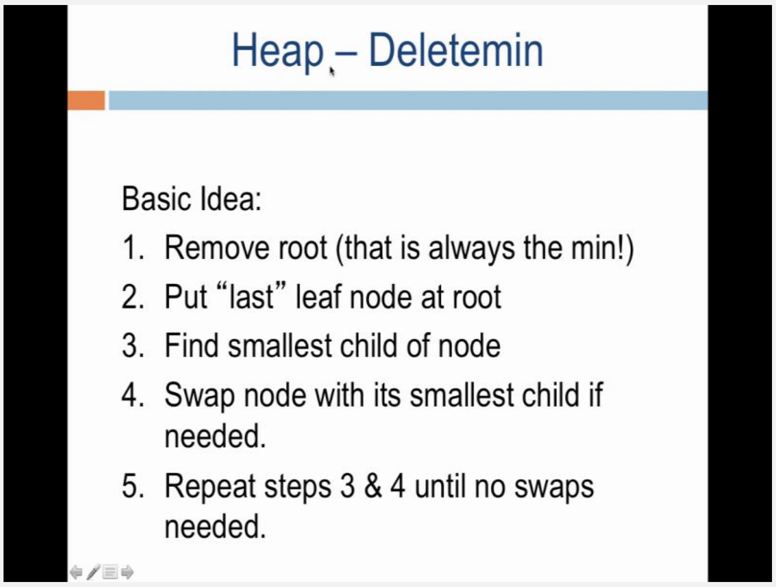


Insert: percolate up

So, let as assume this is the heap. So, this is an array, what are the order of the keys in the array 10, 20, 80, 40, 60, 85, 99, 50, 700, 65 is the heap, it is a complete binary tree. You can verify for every value, it is parent value smaller; therefore, we call it a heap. Now, you look at the value that I want to insert, I insert 15, compare it with a parent and the exchange when necessary, after that compare it again with a parent, then exchange when necessary and this guaranties that the heap property is indeed satisfy.

So, this require a bit of analysis and a cleanup step, for example, let me ask, so if I insert it 45 here, so let me just take a go back again, if want to insert 15, you exchange 15 with 60 and then exchange 15 with 20 again and 1 get's the heap property. So, this is the operation call percolation. So, it could also do this exercise of insert 45 and let as see how for 45 would go. So, 45 would get exchange with 60 and would stay there because it has found a parent which is definitely smaller than at and therefore, the heap property is always maintain by the percolation or the percolate of approach.

(Refer Slide Time: 15:35)



Heap – Deletemin

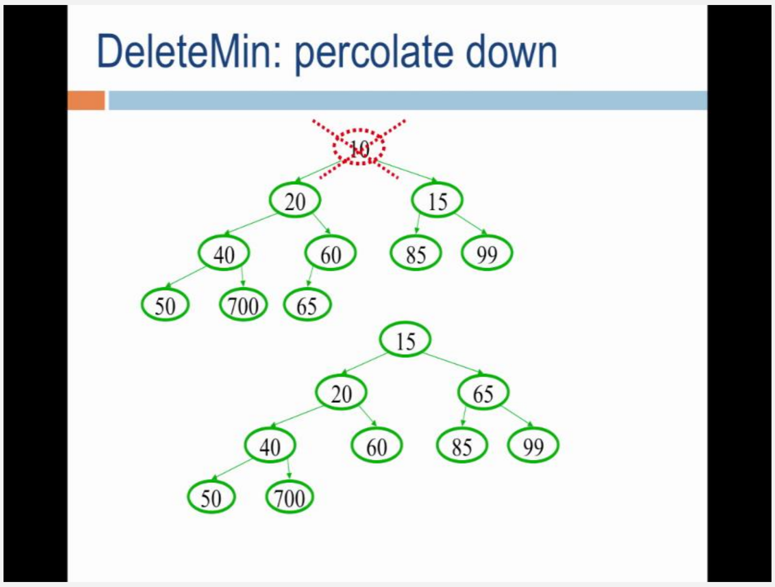
Basic Idea:

1. Remove root (that is always the min!)
2. Put "last" leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

The idea of the delete men is very simple to remove the element, which is the smallest one has the remove the value at the root. Now, how does when remove the root, what we do is, we take the root value, exchange it or rather take the last value from the array and placed into the root. And now what we do is, we find the smallest child of the node and perform the exchange, utilities it is no further necessary, this is the percolate down.

So, let as is understand this, you want to perform the delete men, let as look at the basic idea, first of all you remove the root value, take the last value, the last leaf node, that is the last location of array, take the value an copied into the root. Now, you ensure that the heap property is maintain by taking the root value, exchanging it with is smallest child of the root node and exchange it if necessary. If nodes change as necessary then you are done, if a change as necessary effect the change and keep repeating the steps, until no exchanges or further necessary and then we can stop, this is the delete men approach, this is the percolate down.

(Refer Slide Time: 16:52)

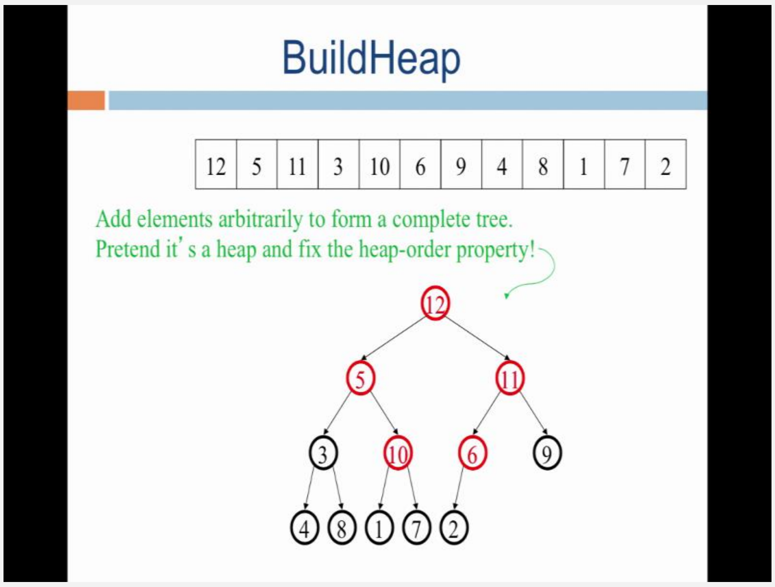


DeleteMin: percolate down

So, let us just see this, so I want a delete 10, so the operation that is perform is take 65, copied into the location word 10 was present. Now, observe that 65 is now exchange to with the smallest of the two children, which is 15 and 65, so 15 comes here and 65 comes here and the heap property is no respected queues' stock. Observe, there it is very important to move of the smallest to the root, when you perform the exchange, that is you take 65, put into the root location, that is a first location.

Then, you look at the elements at look at location 2 and 3, take the smallest of 2, it is 15, we do not exchange a with 20, we exchange to this mallets of the two children and exchange 15 with 65, that is what mix you get this particular heap and this is a percolate down procedure. Therefore, insertion in a heap is how percolate up procedure and insert deletion of the smallest element from the heap that is deleting the root node is a percolate down procedure conceptual. As you can see while the pictures drown here are completely binary trees, the implementation is an array and this can be done very, very efficiently.

(Refer Slide Time: 18:16)

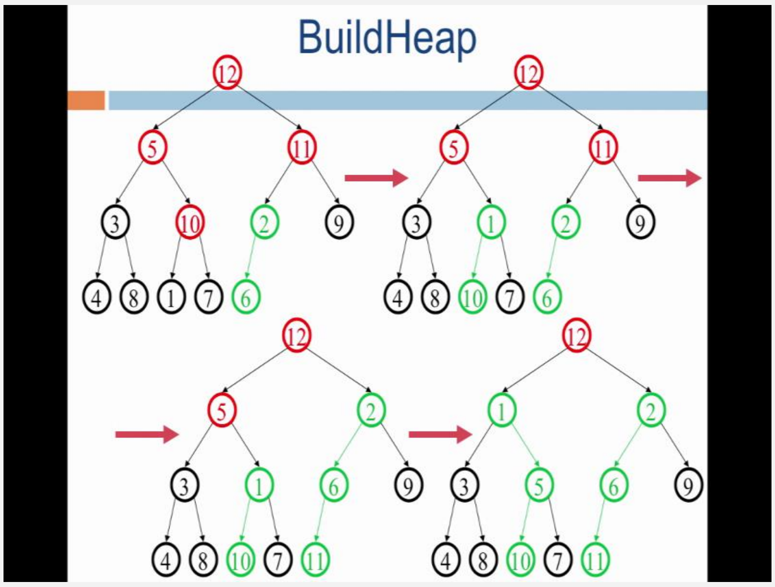

BuildHeap

12	5	11	3	10	6	9	4	8	1	7	2

Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!

Let us see how to do the build heap, here we are not worry about efficiency, we only will worry about the correctness. So, take the elements that need to be insert into be heap and arbitrary put it into the array. So, 12, 5, 11, 3, 10, 6, 9, 4, 8, 1, 7, 2 and this immediately one visualize the complete binary tree. Now, what we do we take this complete binary tree and insert do a sequence of inserts, so there it becomes a heap.

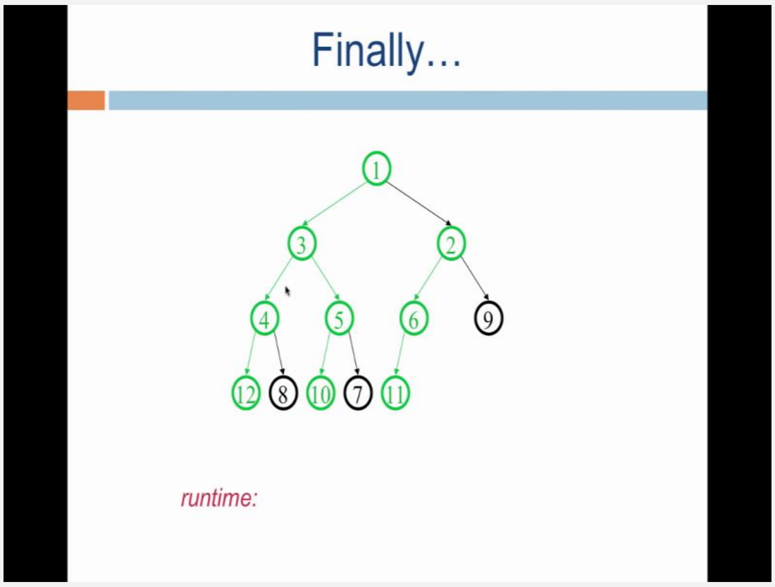
(Refer Slide Time: 18:56)


BuildHeap

Let us a start off with the simplest of insert that we will do. So, iteratively we short of with this particular complete binary tree, which is represented by the program in memory is an array and let us see if it is heap. Now, let us look at this node 2, which is parent is more then it, therefore, it is not a heap. So, therefore, I will exchange 6 and 2, now then 2 come here and notice that, which parent is larger. So, then it terms out that it is not a heap, so exchange it with a 2, then 2 comes here and then 2 comes all the way up to 12.

So, we may repeatedly have to do this till it becomes a heap. So, conceptually it is clear that by repeated exchange as with the parent, whenever a heap violation, heap structure violation is identify, you can finally make the complete binary tree into the heap. Let us just do this, we start here, now let as is observe that what I have done, I just shown in the you can do this on an a arbitrary element. So, let as look at the key 1, now 1 compare with is parent 10 is smaller, so I exchange now 1 compare with as parent 5 is smaller now this get's exchanged.

(Refer Slide Time: 20:36)



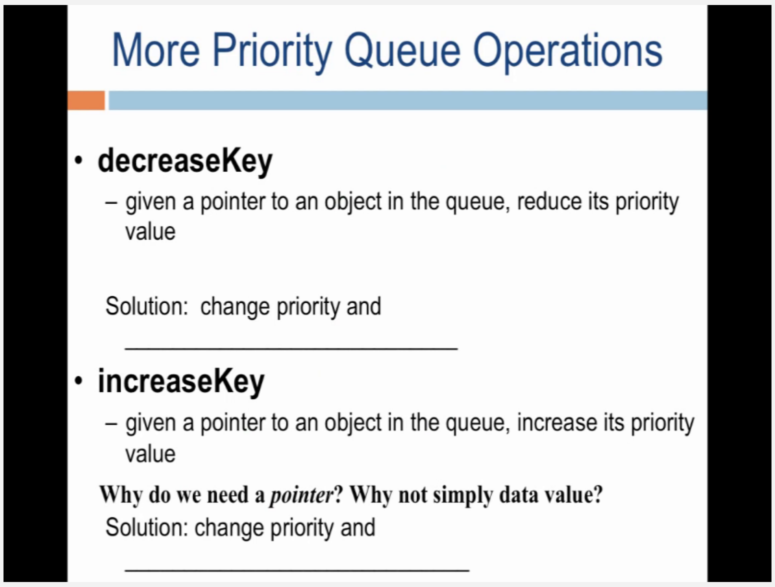
Finally…

runtime:

Now 5 is compared with 12, so this exchange brings 1 appear and now 5 is compare do this. So, eventually if you repeatedly do this, one can see that, one will get a heap, I have not completed this, but you can calculated the run time of this particular approach. So, the idea is sue repeatedly look at every node, compare it with is parent and effect in exchange, whenever necessary. Because, lesser then is a strict relationship between two keys, this procedure will finally terminate very clearly and it will leave you with a heap in the array, it will be a complete binary tree fine.

So, what have we done we have essentially, so how to work with a heap, we shown how to delete min from a heap and maintain the heap structure, we shown how to insert into a heap and maintain the heap structure, we seen how to create a heap, these are the three most important methods all these are implemented in an array and there very easy to implement.

(Refer Slide Time: 21:42)



More Priority Queue Operations

- **decreaseKey**
  - given a pointer to an object in the queue, reduce its priority value

  Solution: change priority and ____________________

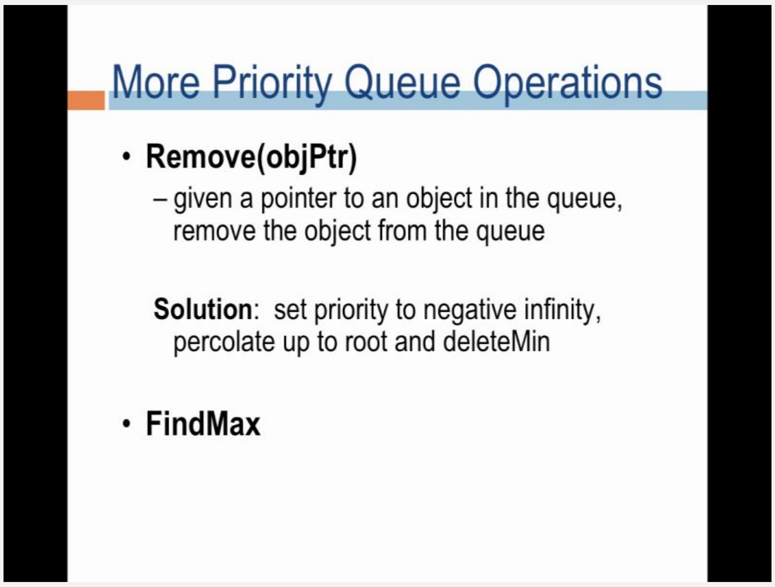
- **increaseKey**
  - given a pointer to an object in the queue, increase its priority value

  Why do we need a *pointer*? Why not simply data value?

  Solution: change priority and ____________________

Let as look at other priority queue operation, especially in operating system increasing the priority value or decreasing the priority value is a basic operation done by any model operation system. We need to implement is we are not going to do this, but I am just pointing of that these are the other operation at you may want to perform on the heap.

(Refer Slide Time: 22:02)



More Priority Queue Operations

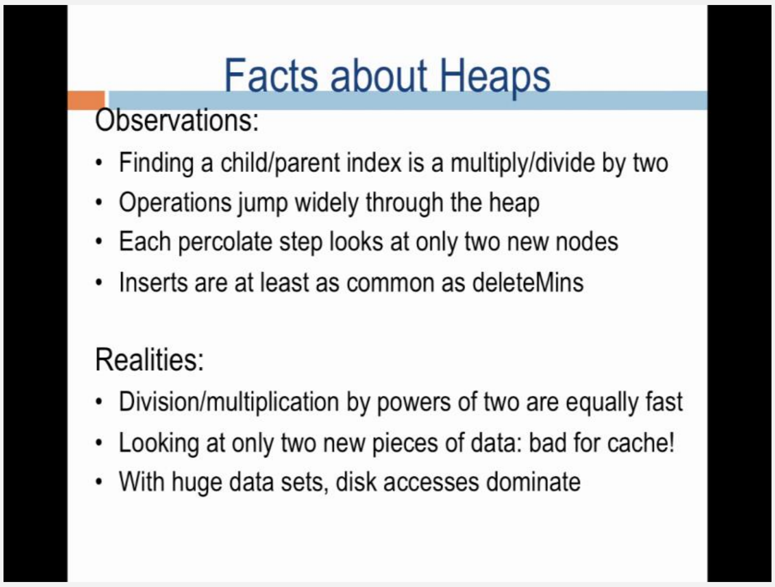
- **Remove(objPtr)**
  - given a pointer to an object in the queue, remove the object from the queue

  Solution: set priority to negative infinity, percolate up to root and deleteMin

- **FindMax**

You might want to remove an element from the heap itself, for example, if you are maintaining a heap of process is that have to be schedule on a processor once the process has been executed on the process, it has be removed from the heap.

(Refer Slide Time: 22:16)



Facts about Heaps

Observations:

- Finding a child/parent index is a multiply/divide by two
- Operations jump widely through the heap
- Each percolate step looks at only two new nodes
- Inserts are at least as common as deleteMins

Realities:

- Division/multiplication by powers of two are equally fast
- Looking at only two new pieces of data: bad for cache!
- With huge data sets, disk accesses dominate

Here at the nice implementation details about a heap, observe that all these implementation can be done an array. So, finding a child or parent is a multiplication division by 2 and it can be done very fast. Now, the other negative think is that, it is not possible to understand as structure of traverse is remember that, if you searching for key a binary search tree, the pointer traversal very clear.

Therefore, if you are doing a program analysis of the behavior of a binary search tree implementation, then there is a very clear understanding of how the pointer and traverse by the search based on a key. In a heap, this is not very clear at all, then every percolate step always looks only a two nodes and this is a very useful think, but it may also be very bad for implementation on fast memory is like causes.

Now, insert are also at is as common as delete men, therefore, when you implement a keep the memory access while performing the keep operations are not very clearly visible from the implementation in depends very much on the data that you are dealing with, however, address arithmetic is very fast. So, this brings to an end are lecture on the heap data structures which has very nice property's as we saw we could implement priority queues, priority queues a very important data structure in operating system, specifically for process scheduling and so on.

And we have seen implementation now, of course, I am not shown your C program that you can see; however and shown you almost all the code that has been done in binary such show to a C code, whereas when I showed you the keep implementation I showed you the Pseudo code, it there is the motivation for doing this. If you understand the Pseudo code by looking at a binary tree implementation, you should be able to implement your heap quickly by implementing the heap structure carefully. So, that more or less brings to in the sequence of lecture, it a programming a data structures codes and you must an already seen the programming assignment for implementing complete binary trees.

Thank you very much for your attention, and congrats on completing the sequence of lecture.