1

Lecture #15

- Priority Queues
  - Heaps
- HeapSort

5

Priority Queues

A Priority Queue supports three operations:

- Insert a new item into the queue
- Get the value of the highest priority item
- Remove the highest priority item from the queue

When you define a Priority Queue, you must specify how to determine the priority of each item in the queue.

Priority = amount of blood lost + number of cuts

You must then design your PQ data structure/algorithms so you can efficiently retrieve the highest-priority item.

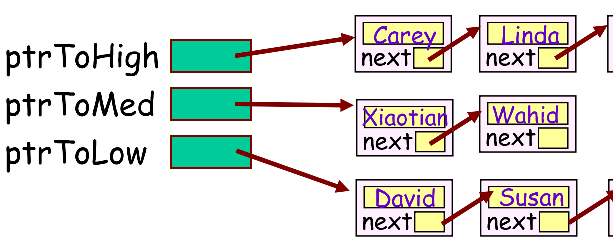
6

Priority Queues

Question: What data structures can we use to implement a priority queue? Hmmm…

Let's make it easier… What if we have just a limited set of priorities, e.g.: high, medium low?

Hint: Think of an airport ticket line with first, business and coach (cattle) class…



Right – we can use n linked lists, one for each priority level.

To obtain the highest-priority item, always take the first item from the highest priority, non-empty list.

7

Priority Queues

Question: Ok, but what data structure should we use if we have a huge number of priorities? Hmmm!

The HEAP data structure is one of the most efficient ones we can use to implement a Priority Queue.

The heap data structure uses a special type of binary tree to hold its data.

As we'll see, while a heap does use a binary tree to store its data, a heap is NOT a binary search tree.

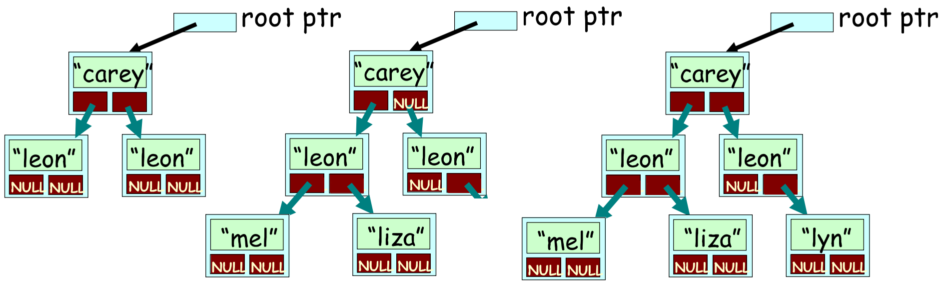
8

All Heaps Use a "Complete" Binary Tree

A complete binary tree is one in which:

- The top N-1 levels of the tree are completely filled with nodes
- All nodes on the bottom-most level must be as far left as possible (with no empty slots between nodes!)

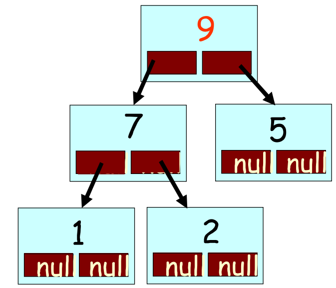
Is it complete?



14

Extracting the Biggest Item

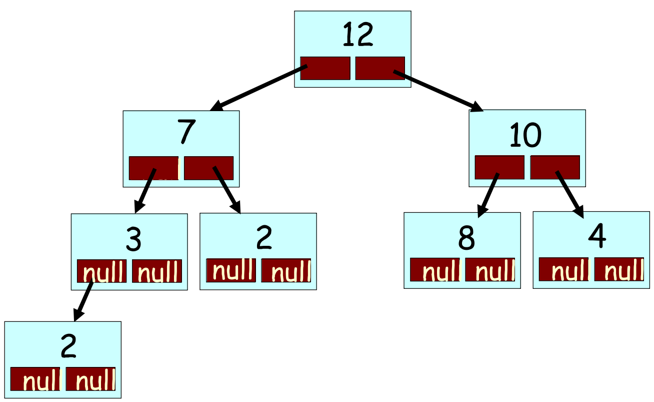
1. If the tree is empty, return error.
2. Otherwise, the top item in the tree is the biggest value. Remember it for later.
3. If the heap has only one node, then delete it and return the saved value.
4. Copy the value from the right-most node in the bottom-most row to the root node.
5. Delete the right-most node in the bottom-most row.
6. Repeatedly swap the just-moved value with the larger of its two children until the value is greater than or equal to both of its children. ("sifting DOWN")
7. Return the saved value to the user.



When we're done, the largest value is on the top again, and the heap is consistent.

15

Extraction Challenge!



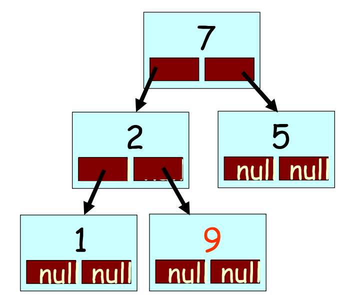
Show the resulting heap after extracting the largest item!

16

Adding a Node to a Maxheap

(Let's see how to add a value of 9)

1. If the tree is empty, create a new root node & return.
2. Otherwise, insert the new node in the bottom-most, left-most position of the tree (so it's still a complete tree).
3. Compare the new value with its parent's value.
4. If the new value is greater than its parent's value, then swap them.
5. Repeat steps 3-4 until the new value rises to its proper place.



This process is called "reheapification."

20

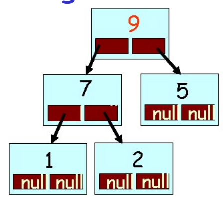
Implementing A Heap

Question:
What data structure can we use to implement a heap?

How about a classical binary tree node with links?

```
struct node
{
  int value;
  node *left, *right;
};
```

Hmmm… But this has some challenges. What are they?



1. It's not easy to locate the bottom-most, right-most node during extraction.
2. It's not easy to locate the bottom-most, left-most open spot to insert a new node during insertion!
3. It's not easy to locate a node's parent to do reheapification swaps.

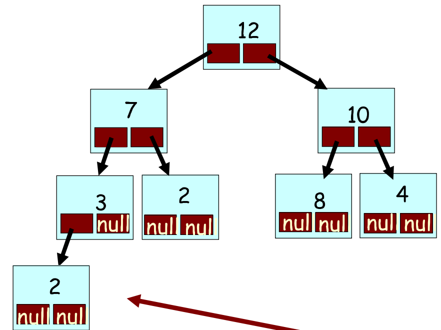
21

Implementing A Heap

Perhaps there's some better data structure we could use…

Hmmmm. What about an array?

Well, we know that each level of our tree has 2x the number of nodes of the previous level*.



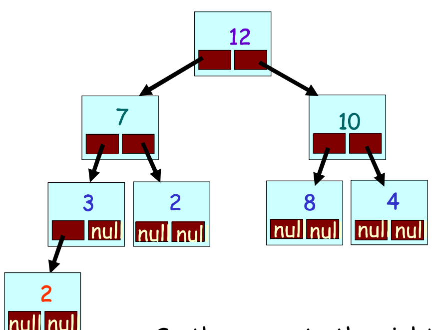
So what if we just copy our nodes a level at a time into an array???

* Except for the last level…

22

Implementing A Heap

So what if we just copy our nodes a level at a time into an array???



Let's see, we can put our root node value in heap[0]

And we can put the next two nodes' values into the next two available slots…

And then the next four values in the next four slots…

Finally, let's use a simple int variable to track how many items are in our heap!

So the array to the right now logically represents the tree on the left! And if we use the array, there's no need to use a node-based tree!

int count;

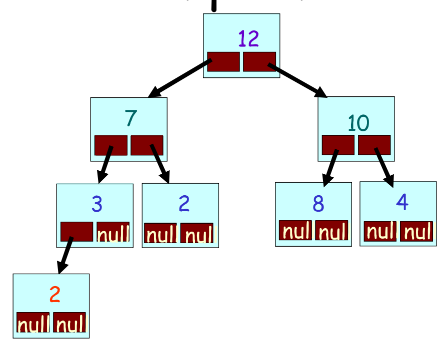
8

int heap[1000];

0	12
1	7
2	10
3	3
4	2
5	8
6	4
7	2
8	
9	
10	
11	
12	
13	
…	

23

Implementing A Heap



int count;

9

int heap[1000];

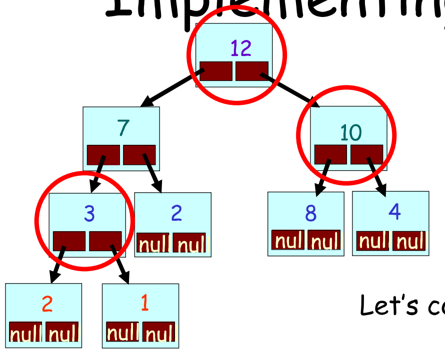
0	12
1	7
2	10
3	3
4	2
5	8
6	4
7	2
8	1
9	
10	
11	
12	
13	
…	

So what are the properties of our array-based tree?

1. We can always find the root value in heap[0]
2. We can always find bottom-most, right-most node in heap[count-1]
3. We can always find the bottom-most, left-most empty spot (to add a new value) in heap[count]
4. We can add or remove a node by simply setting heap[count] = value; and/or updating our count!

24

Implementing A Heap



int count;

9

int heap[1000];

0	12
1	7
2	10
3	3
4	2
5	8
6	4
7	2
8	1
9	
10	
11	
12	
13	
...	

Ok, in our array, how do we locate the left and right children of a node?

Let's consider some examples:

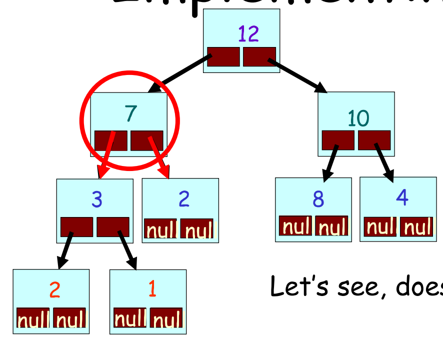
Parent Slot#	Left Child Slot#	Right Child Slot#
0	1	2
2	5	6
3	7	8

Challenge: Come up with a formula to locate a node's children

Hint: It's of the form leftChild(parent) = 2 * parent + 1
rightChild(parent) = 2 * parent + 2

25

Implementing A Heap



int count;

9

int heap[1000];

0	12
1	7
2	10
3	3
4	2
5	8
6	4
7	2
8	1
9	
10	
11	
12	
13	
...	

Let's see, does our formula work?

Consider this node, which is in slot #1 of our array?

leftChild(1) = 2 * 1 + 1 = slot 3

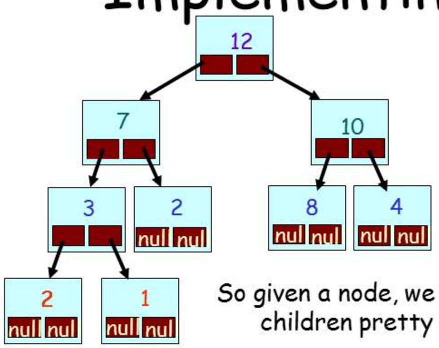
rightChild(1) = 2 * 1 + 2 = slot 4

Our formula appears to work!

Hint: It's of the form leftChild(parent) = 2 * parent + 1
rightChild(parent) = 2 * parent + 2

24

Implementing A Heap



int count;

9

int heap[1000];

0	12
1	7
2	10
10	
11	
12	
13	
...	

So given a node, we can find its two children pretty easily. Cool!

Question: So now how do we find the slot of the parent of some node in ou

Answer: Use simple algebra!

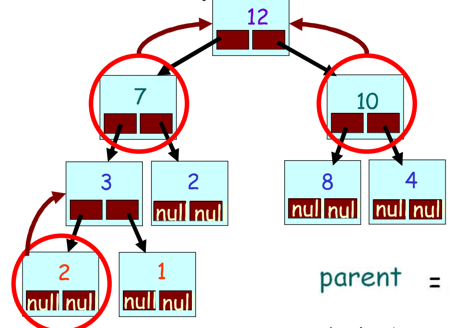
$$\frac{\text{child-1}}{2} = \text{parent}$$

And, due to a property of C++ integer division...

this formula works equally well for both left and right children!

27

Implementing A Heap



int count;

9

int heap[1000];

0	12
1	7
2	10
3	3
4	2
5	8
6	4
7	2
8	1
9	
10	
11	
12	
13	
...	

$$\text{parent} = \frac{\text{child-1}}{2}$$

Ok, let's verify that it works...

The parent of slot #1 is... (1-1)/2 = 0

The parent of slot #2 is... (2-1)/2 = 0

The parent of slot #7 is... (7-1)/2 = 3

Cool stool! So now we know how to locate the children of a node, find the parent of a node, and add and remove nodes!

28

Heap in an Array Summary

So, now we know how to store a heap in an array!

Here's a recap of what we just learned :

1. The root of the heap goes in array[0]
2. If the data for a node appears in array[i], its children, if they exist, are in these locations:
   Left child: array[2i+1]
   Right child: array[2i+2]
3. If the data for a non-root node is in array[i], then its parent is always at array[(i-1)/2] (Use integer division)

29

A Heap Helper Class

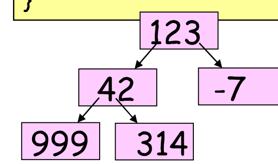
```
class HeapHelper
{
  HeapHelper()              { num = 0;  }
  int GetRootIndex()        { return(0); }
  int LeftChildLoc(int i)   { return(2*i+1); }
  int RightChildLoc(int i)  { return(2*i+2); }
  int ParentLoc(int i)      { return((i-1)/2); }
  int PrintVal(int i)       { cout << a[i]; }
  void AddNode(int v)       { a[num] = v; ++num;}
private:
  int a[MAX_ITEMS];
  int num;
};
```

```
main()
{
  HeapHelper a;

  a.AddNode(123);
  a.AddNode(42);
  a.AddNode(-7);
  a.AddNode(999);
  a.AddNode(314);

  int i = GetRootIndex();
  PrintVal(i);
  i = LeftChildLoc(i);
  PrintVal(i);
  i = RightChildLoc(i);
  PrintVal(i);
}
```

Output:

num 5

a[0]	123
[1]	42
[2]	-7
[3]	999
[4]	314



30

A Heap Helper Class

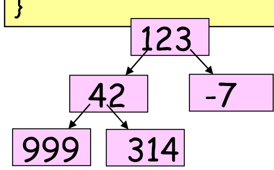
```
class HeapHelper
{
  HeapHelper()              { num = 0;  }
  int GetRootIndex()        { return(0); }
  int LeftChildLoc(int i)   { return(2*i+1); }
  int RightChildLoc(int i)  { return(2*i+2); }
  int ParentLoc(int i)      { return((i-1)/2); }
  int PrintVal(int i)       { cout << a[i]; }
  void AddNode(int v)       { a[num] = v; ++num;}
private:
  int a[MAX_ITEMS];
  int num;
};
```

```
main()
{
  HeapHelper a;

  a.AddNode(123);
  a.AddNode(42);
  a.AddNode(-7);
  a.AddNode(999);
  a.AddNode(314);

  int i = GetRootIndex();
  PrintVal(i);
  i = LeftChildLoc(i);
  PrintVal(i);
  i = RightChildLoc(i);
  PrintVal(i);
}
```

Output:

num 5

a[0]	123
[1]	42
[2]	-7
[3]	999
[4]	314



33

Extracting from a Maxheap – The Array Version!

1. If the count == 0 (it's an empty tree), return error.
2. Otherwise, heap[0] holds the biggest value. Remember it for later.
3. If the count == 1 (that was the only node) then set count=0 and return the saved value.
4. Copy the value from the right-most, bottom-most node to the root node: heap[0] = heap[count-1]
5. Delete the right-most node in the bottom-most row: count = count - 1
6. Repeatedly swap the just-moved value with the larger of its two children:
   Starting with i=0, compare and swap:
   heap[i] with heap[2*i+1] and heap[2*i+2]
7. Return the saved value to the user.

int count;

9

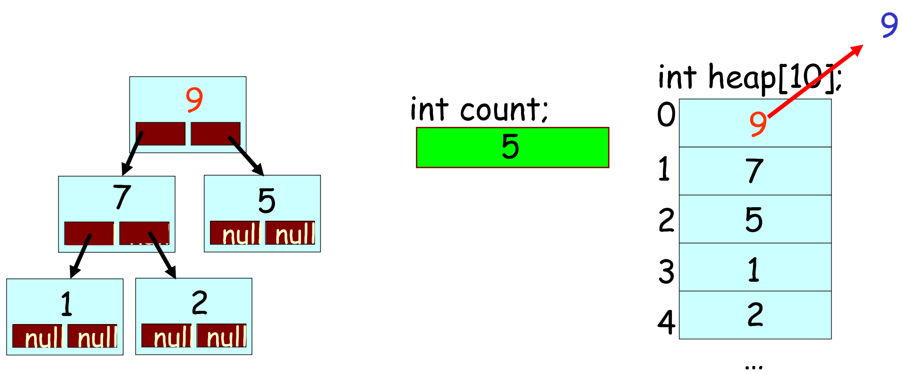
int heap[1000];

0	12
1	7
2	10
3	3
4	2
5	8
6	4
7	2
8	1
9	
10	
11	
12	
13	
...	

34

Implementing A Heap

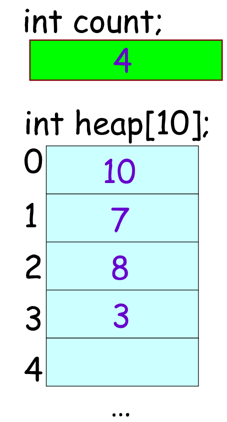
Ok, so now let's see how to extract the biggest item from an array-based max-heap!



35

Adding a Node to a Maxheap – The Array Version

1. Insert a new node in the bottom-most, left-most open slot:
   heap[count] = value
   count = count + 1;
2. Compare the new value heap[i] with its parent's value: heap[(i-1)/2]
3. If the new value is greater than its parent's value, then swap them.
4. Repeat steps 2-3 until the new value rises to its proper place or we reach the top of the array.

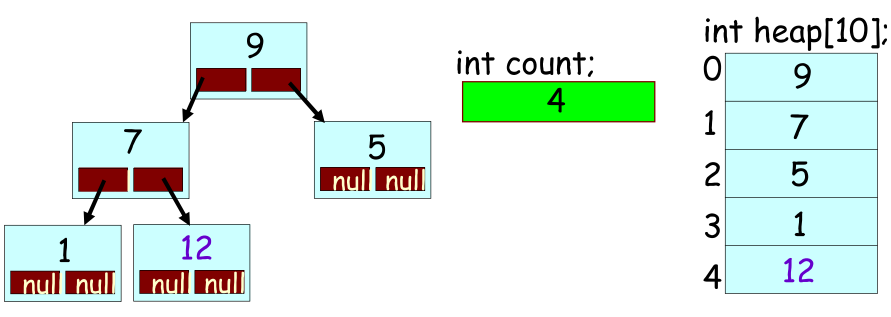


36

Heap Insertion Challenge

Now let's work through the insertion of a value into our array-based heap.

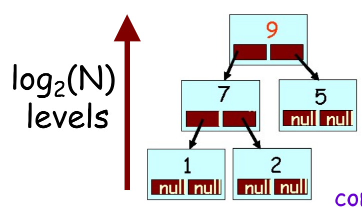
Let's add 12 to our heap.



39

Complexity of the Heap

Question: What is the big-oh cost of inserting a new item into a heap?



Every time we insert a new item, we need to keep comparing it with its parent until it reaches the right spot…

Since our tree is a COMPLETE binary tree, if it has N entries, it's guaranteed to be exactly $log_2(N)$ levels deep.

So in the worst case, we'll have to do $log_2(N)$ comparisons and swaps of our new value. (This is true whether or not our heap is stored in an array!)

Question: What is the big-oh cost of extracting the maximum/minimum item from a heap?

Just as with heap insertion, when we extract a value we need to bubble an item from the root down the tree.

Since the maximum number of levels in our tree is $log_2(N)$, the worst case that this requires $log_2(N)$ swaps.

So inserting and extracting from a heap is $O(log_2(n))$

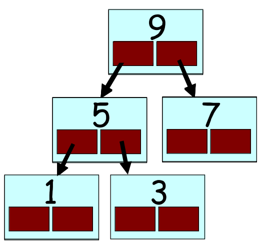
The Heapsort

Question:
How can we use a heap to sort a bunch of items?

Answer:
Here's a "naïve" way to do it…

Given an array of N numbers that we want to sort:

1. Insert all N numbers into a new maxheap
2. While there are numbers left in the heap:
   A. Remove the biggest value from the heap
   B. Place it in the last open slot of the array



And viola!
Our array is sorted!

7	3	9	1	5

The Efficient (Official) Heapsort

Let's update our original inefficient algorithm to turn it into the efficient (official) version.

Given an array of N numbers that we want to sort:

1. Convert our input array into a maxheap
2. While there are numbers left in the heap:
   A. Remove the biggest value from the heap
   B. Place it in the last open slot of the array

Wow! It's that simple? Yup!

We're just going to just shuffle the values around in our input array so they become a maxheap.

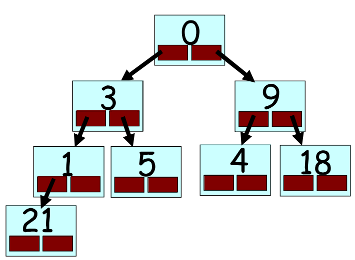
Then we'll use that maxheap as if it were a separate array like before. Only now everything's in just one array.

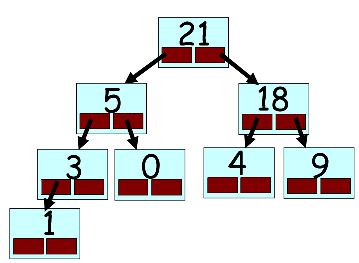
Step #1:

Convert your randomly-arranged input array into a maxheap by cleverly shuffling around the values in the array.

0	3	9	1	5	4	18	21

21	5	18	3	0	4	9	1



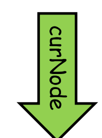


Let's learn the algorithm to perform this shuffling.

Step #1: Convert Your Input Array into a MaxHeap

Let's start by visualizing our array as a tree.

By last node, we mean the bottom-most, right-most node in the tree. This corresponds to the last element in the array.



0	3	9	1	5	4	18	21

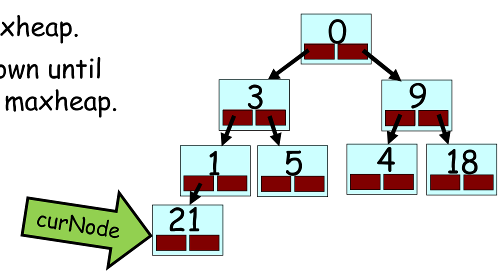
Ok, now here's the algorithm:

for (curNode = lastNode thru rootNode):

Focus on the subtree rooted at curNode.

Think of this subtree as a maxheap.

Keep shifting the top value down until your subtree becomes a valid maxheap.



49

Step #1: Convert Your Input Array into a MaxHeap

Let's start by visualizing our array as a tree.

curNode

0	3	9	1	5	4	18	21

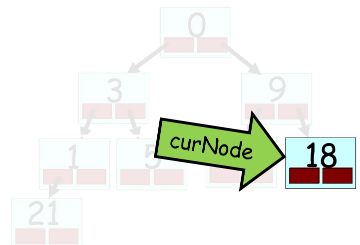
Ok, now here's the algorithm:

for (curNode = lastNode thru rootNode):

Focus on the subtree rooted at curNode.

Think of this subtree as a maxheap.

Keep shifting the top value down until your subtree becomes a valid maxheap.



56

Step #1: Convert Your Input Array into a MaxHeap

Let's start by visualizing o[ur] array as a tree.

curNode

0	21	18	3	5	4	9	1

Ok, now here's the algorithm:

for (curNode = lastNode t[hru rootNode]):

Focus on the subtree roo[ted at curNode.]

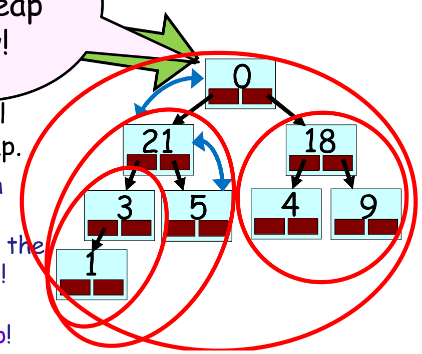
Think of this subtree as a m[axheap.]

Keep shifting the top value down until your subtree becomes a valid maxheap.

Essentially what we've done is heapify each sub-tree from the bottom-up.

As we heapify higher sub-trees, they rely upon the lower sub-trees that were heapified earlier!

Once we've finished heapifying from our root node, our entire array will hold a valid maxheap!



57

Step #1: Convert Your Input Array into a MaxHeap

There's one more thing to consider!

curNode

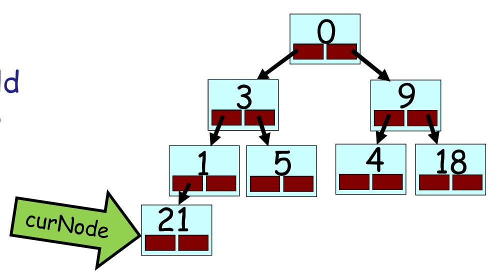
0	3	9	1	5	4	18	21

If you noticed, we wasted a bunch of time looking at single-node sub-trees.

But we only had to reheapify once we reached a sub-tree with at least two nodes.

Wouldn't it be great if we could jump straight to this node to save time?

We can – here's how!



58

[Step #1: Convert Your] Input Array into a MaxHeap

This locates the lowest, right-most node in the tree that has at least one child.

Allowing us to skip all of the single-element trees – that's roughly 50% of all the subtrees!

OK, let's see:

startNode = 8/2 – 1

That means startNode = 3.

Let's update our shuffling algorithm slightly!

N=8

startNode

0	1	2	3	4	5	6	7
0	3	9	1	5	4	18	21

startNode = N/2 - 1

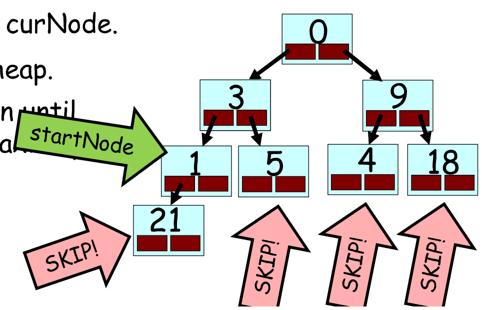
for (curNode = startNode thru rootNode):

Focus on the subtree rooted at curNode.

Think of this subtree as a maxheap.

Keep shifting the top value down until your subtree becomes a valid maxheap.

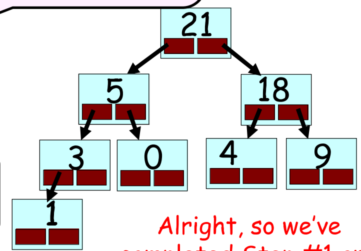
This is the complete version of the efficient shuffling algorithm!



But once we remove the top item, we have to reheapify our heap!

Note – this frees up the last slot in the array! Our heap now only occupies the first N-1 slots of the array!

21	5	18	3	0	4	9	1



When we finish reheapifying, our first N-1 slots hold a valid maxheap AND the last slot of the array is empty!

21

Alright, so we've completed Step #1 and our input array now holds a valid maxheap. On to Step #2!

Reheapification Algorithm (same as before)

1. Copy the value from the right-most node in the bottom-most row to the root node.
2. Delete the right-most node in the bottom-most row.
3. Repeatedly swap the just-moved value with the larger of its two children until the value is greater than or equal to both of its children.

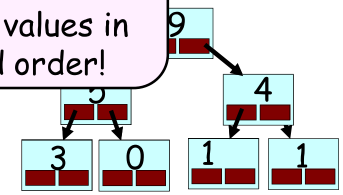
Guess what!

Step #2 of our efficient heapsort is virtually identical to Step #2 of naïve heapsort!

Efficient H…2

Now the last two slots of the array hold the two biggest values in sorted order!

9	5	4	3	0	1	18	21



While there are numbers left in the heap:

1. Extract the biggest value from the maxheap and re-heapify (just as we learned about 20 slides ago)

   This frees up the second-to-last slot in the array (since the heap now has 1 fewer value in it)

3. Now put the extracted value into this freed-up slot of the array.

…Heapsort: Step #2

Now the last three slots of the array hold the three biggest values in sorted order!

5	3	4	1	0	9	18	21

If you keep repeating this extraction/reheapification /insertion process the array will be completely sorted!

While there are numbers left in the heap:

1. Extract the biggest value from the maxheap and re-heapify (just as we learned about 20 slides ago)

   This frees up the third-to-last slot in the array (since the heap now has 1 fewer value in it)

3. Now put the extracted value into this freed-up slot of the array.

Big-O of Heapsort!

Step #1:

First we take our N-item array (shown here as a tree) and convert it into a maxheap.

We do this from the bottom up by converting successively larger subtrees into maxheaps until the entire tree has been converted.

Step #2:

Then we repeatedly extract the j^{th} largest item from the maxheap and place that item back into the array, j slots from the end.

If you think it should be O(N log2N), I did too. ☺ Come to office hours for an explanation.

Step #1 has a Big-O of O(N).

In other words, we can convert a random array into a maxheap in just O(N) steps!

+

Step #2 has a Big-O of O(N $\log_2$N).

Why? Each time we remove an item from the maxheap, it takes $\log_2$N steps. We perform this extraction operation N times to sort the entire array.

=

Therefore, Heapsort is O(N + N $\log_2$N), which as you know, is just O(N $\log_2$N).