Lecture #15

- unordered_map
- Priority Queues
- Heaps
  - Using classic binary trees
  - Using arrays!
- HeapSort
- Review Challenge

The unordered_map: A hash-based version of a map

```
#include <unordered_map>
#include <iostream>
#include <string>
using namespace std::tr1; // required for a hash-based map
using namespace std;

int main( )
{
   unordered_map <string,int> hm;                 // define a new U_M
   unordered_map <string,int>::iterator iter; // define an iterator for a U_M

   hm["Carey"] = 10;                            // insert a new item into the U_M
   hm["David"] = 20;

   iter = hm.find("Carey");                     // find Carey in the hash map

   if (iter == hm.end())                        // did we find Carey or not?
      cout << "Carey was not found!";           // couldn't find "Carey" in the hash map
   else
   {
      cout << "When we look up " << iter->first;       // "When we look up Carey"
      cout << " we find " << iter->second;             // "we find 10"
   }
}
```

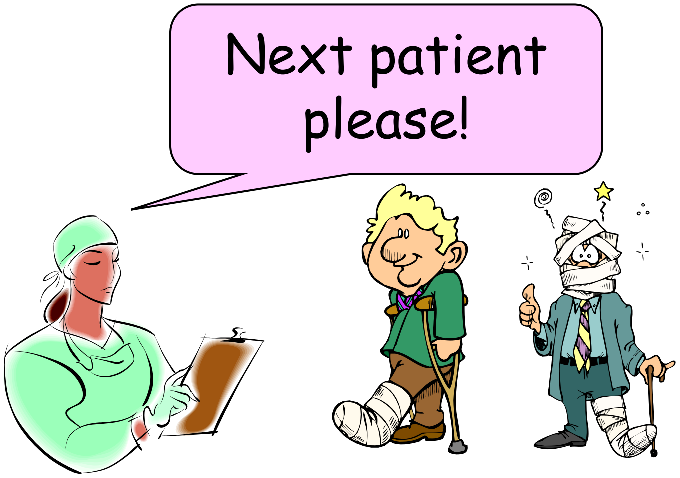
Priority Queues

A priority queue is a special type of queue that allows us to keep a prioritized list of items.

In a priority queue, each item you insert into the queue has a "priority rating" indicating how important it is.

Any time you dequeue an item from a priority queue, it always dequeues the item with the highest priority (instead of just the first item inserted).

Example: If I have a queue of patients in the emergency room, I don't just take the next patient in line, I take the one who has the most severe injuries.



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

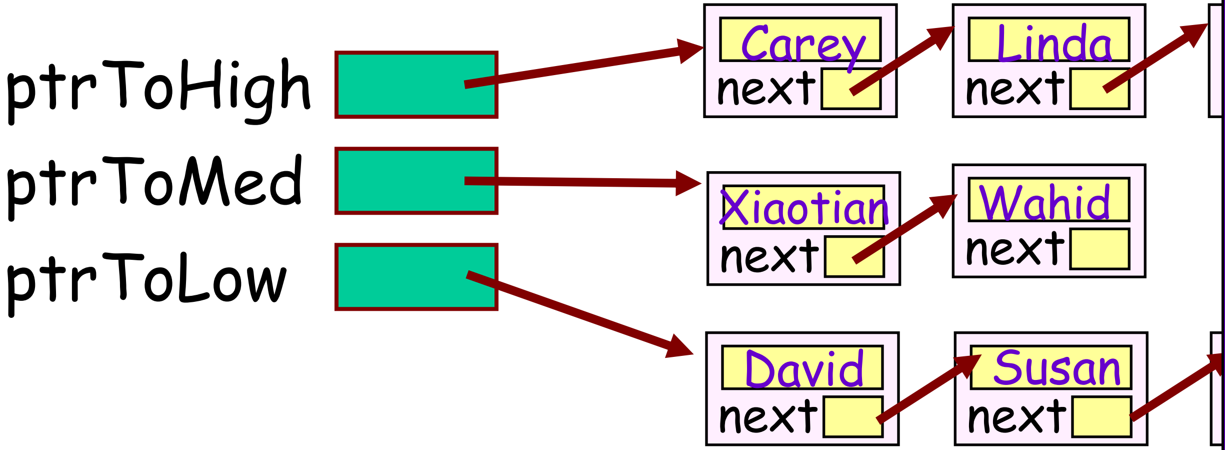
You must then design your PQ data structure/algorithms so you can efficiently retrieve the highest-priority item.

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.

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.

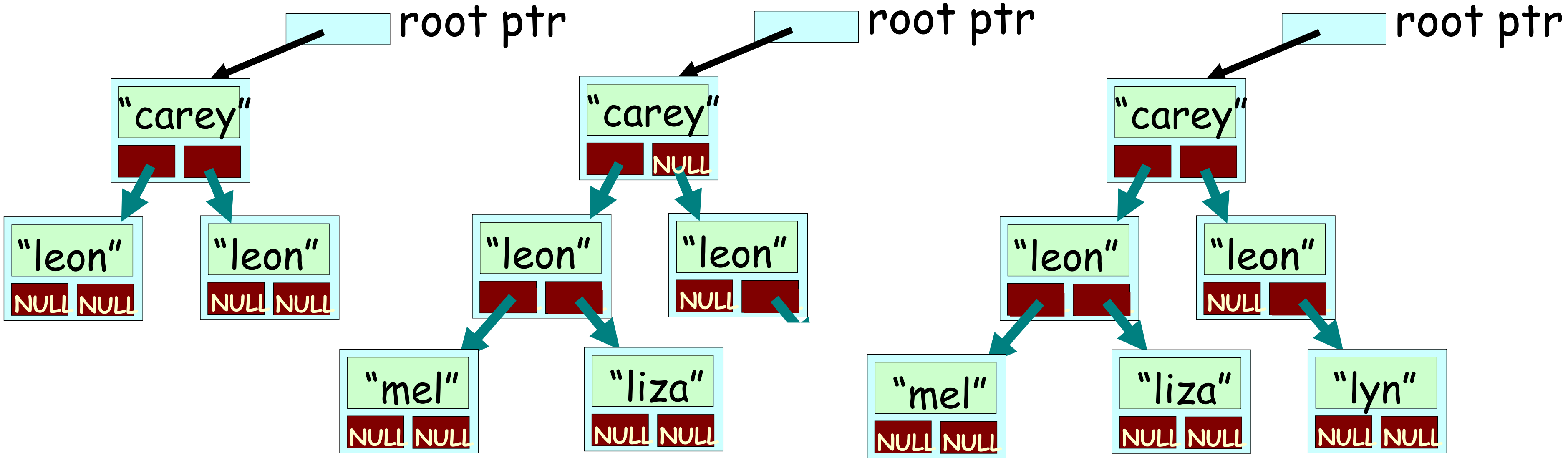
As we'll see, while a heap does use a binary tree to store its data, a heap is NOT a binary search tree.

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?



Heaps… of…

A heap is a special type of complete binary tree
(it's **not** a binary search tree).

There are two types of heaps, minheaps and maxheaps:

Maxheap:

1. Quickly insert a new item into the heap
2. Quickly retrieve the largest item from the heap

Minheap:

1. Quickly insert a new item into the heap
2. Quickly retrieve the smallest item from the heap

The Maxheap

A maxheap is a binary tree which follows these rules:

1. The value contained by a node is ALWAYS GREATER THAN OR EQUAL TO the values of the node's children.

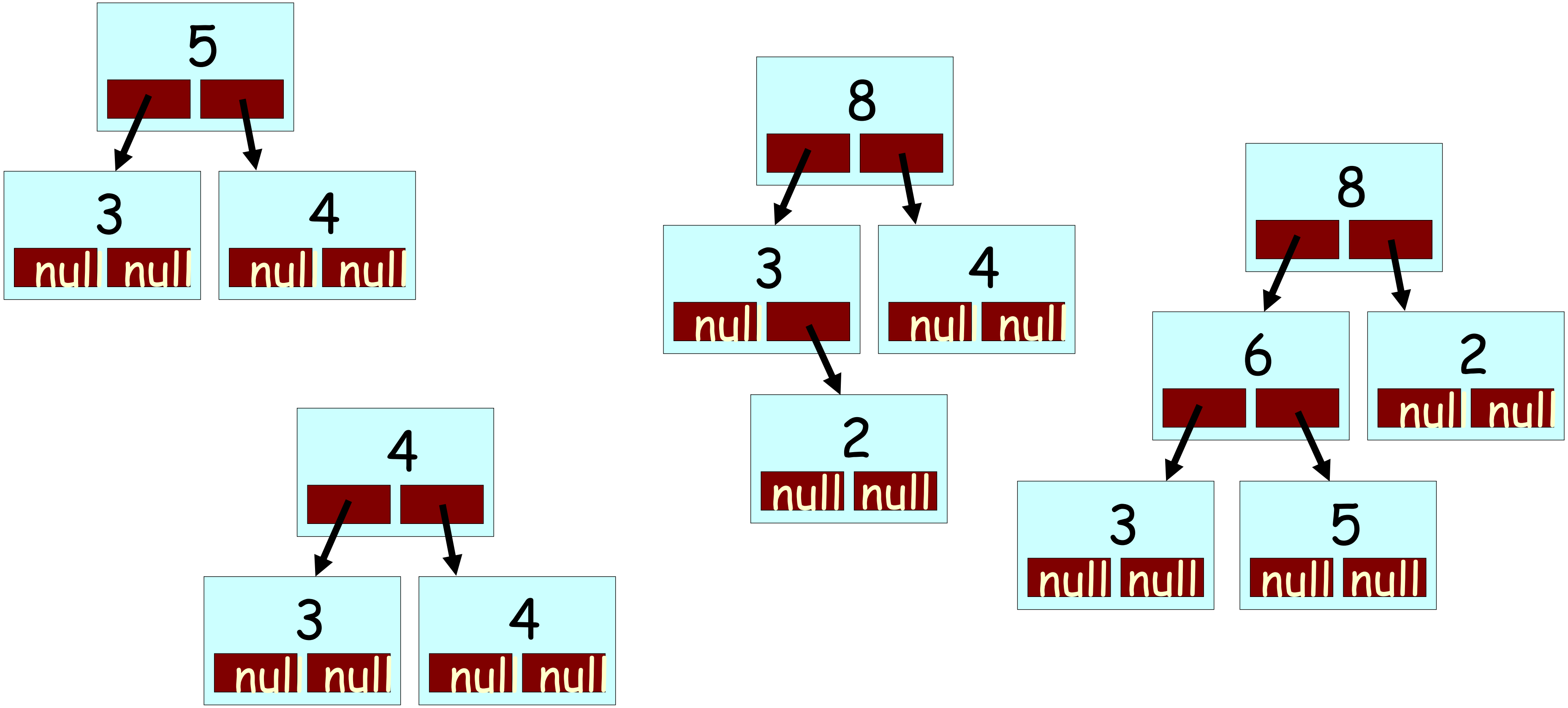
2. The tree is a COMPLETE binary tree.

Question: What are the rules for a minheap?

Complete Binary Tree:
Every level except the deepest level must contain as many nodes as possible, and the at the deepest level, all nodes are as far left as possible.

The Maxheap

Which of the following are valid maxheaps?



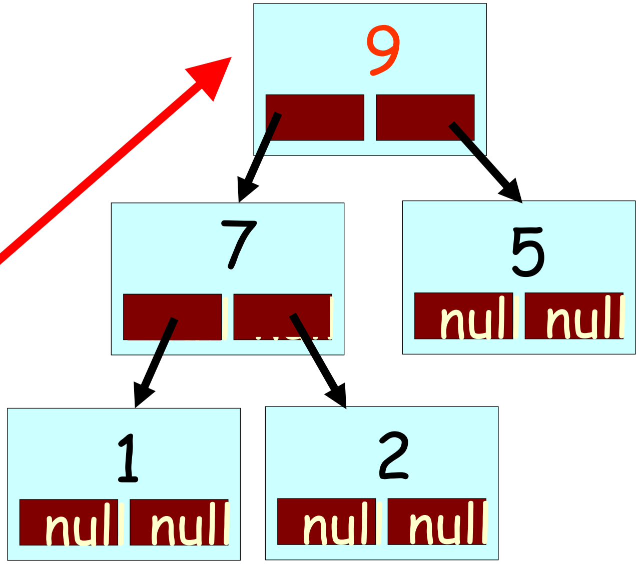
1. The value contained by a node is ALWAYS >= the values of the node’s children.
2. The tree is a COMPLETE binary tree.

The Maxheap

One thing you'll notice about a maxheap is…

that, by definition, the biggest (highest priority) item

is always at the *root* (top) of the tree!



This means its easy to always find the biggest (highest priority) item in just a single step!

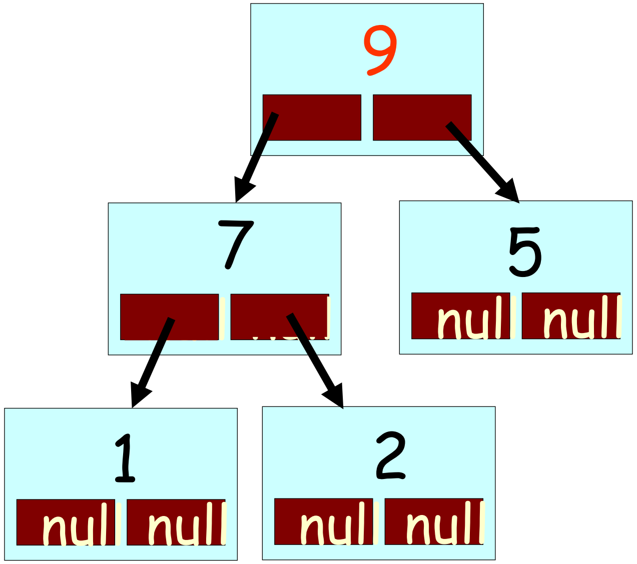
1. The value contained by a node is ALWAYS >= the values of the node's children.
2. The tree is a COMPLETE binary tree.

Operations on a Maxheap

Allright, now let's see how to extract an item from a heap and to add a new item to a heap!

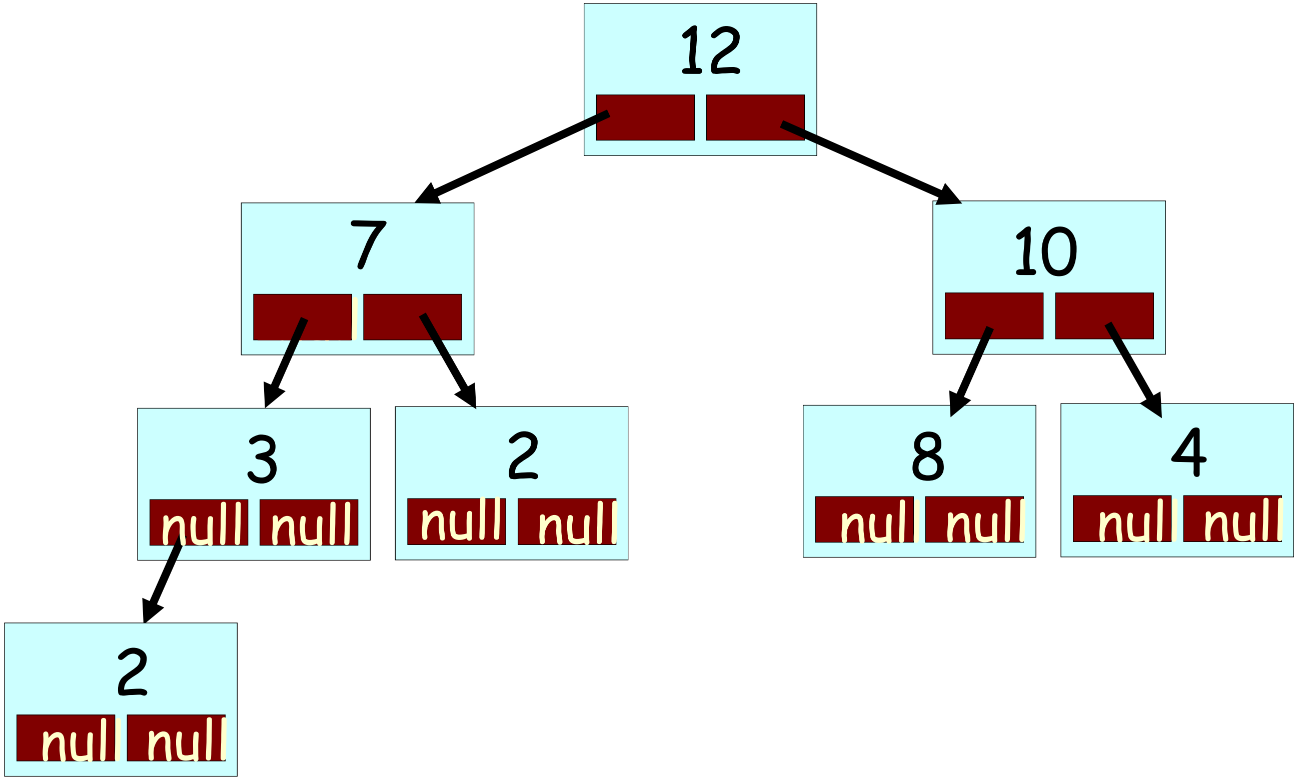
Extracting the Biggest Item

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.

Extraction Challenge!

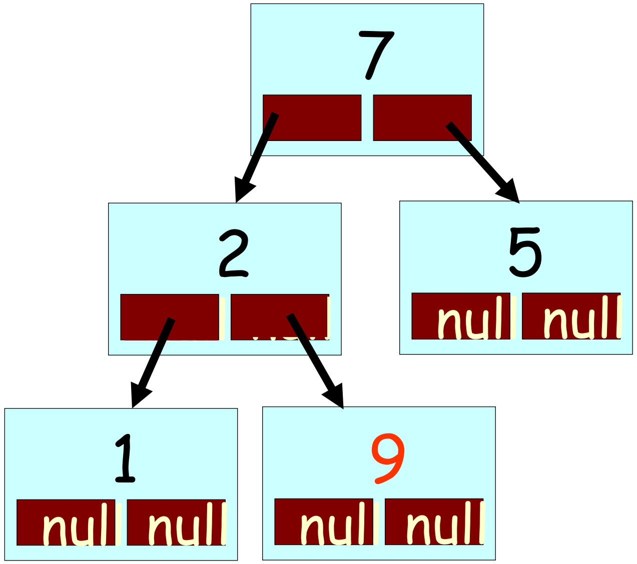


Show the resulting heap after extracting the largest item!

Adding a Node to a Maxheap

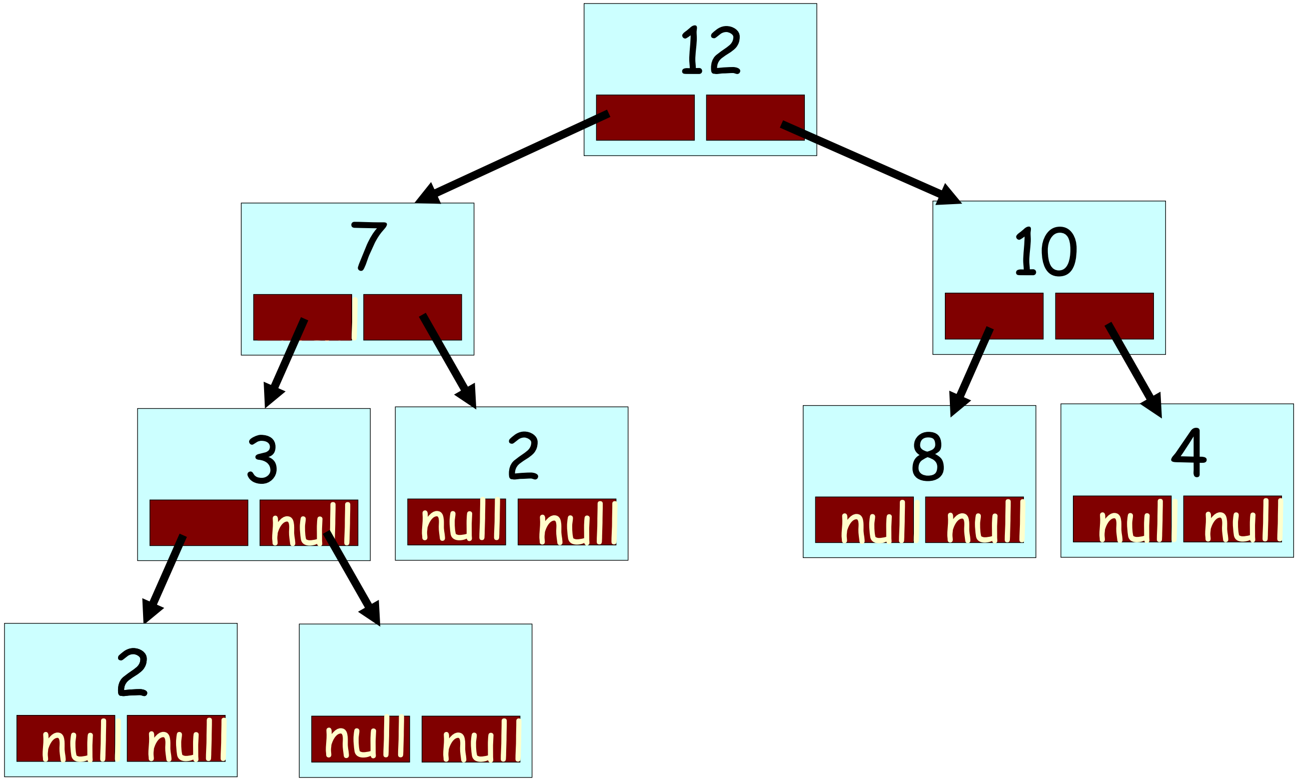
(Let's see how to add a value of 9)

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.



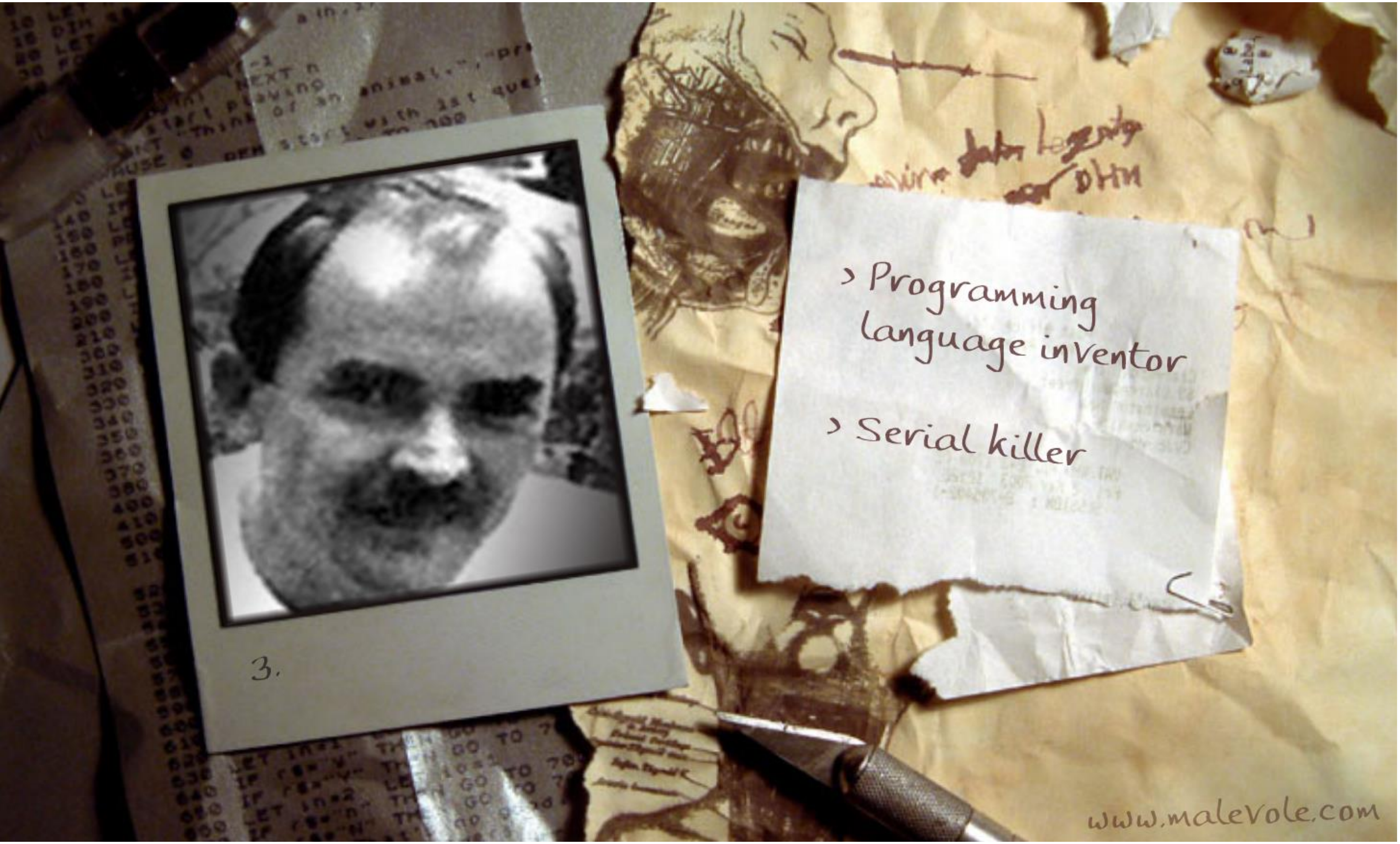
This process is called "reheapification."

Insertion Challenge!



Show the resulting heap after inserting a value of 9!

And now, time for your favorite game!!!!



Implementing A Heap

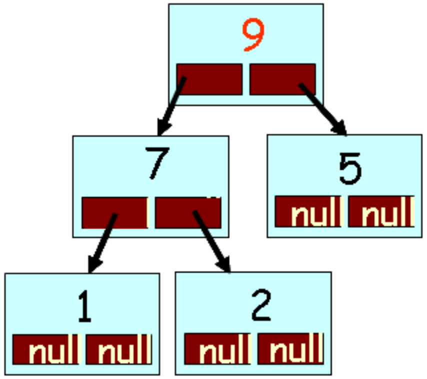
Question:

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.

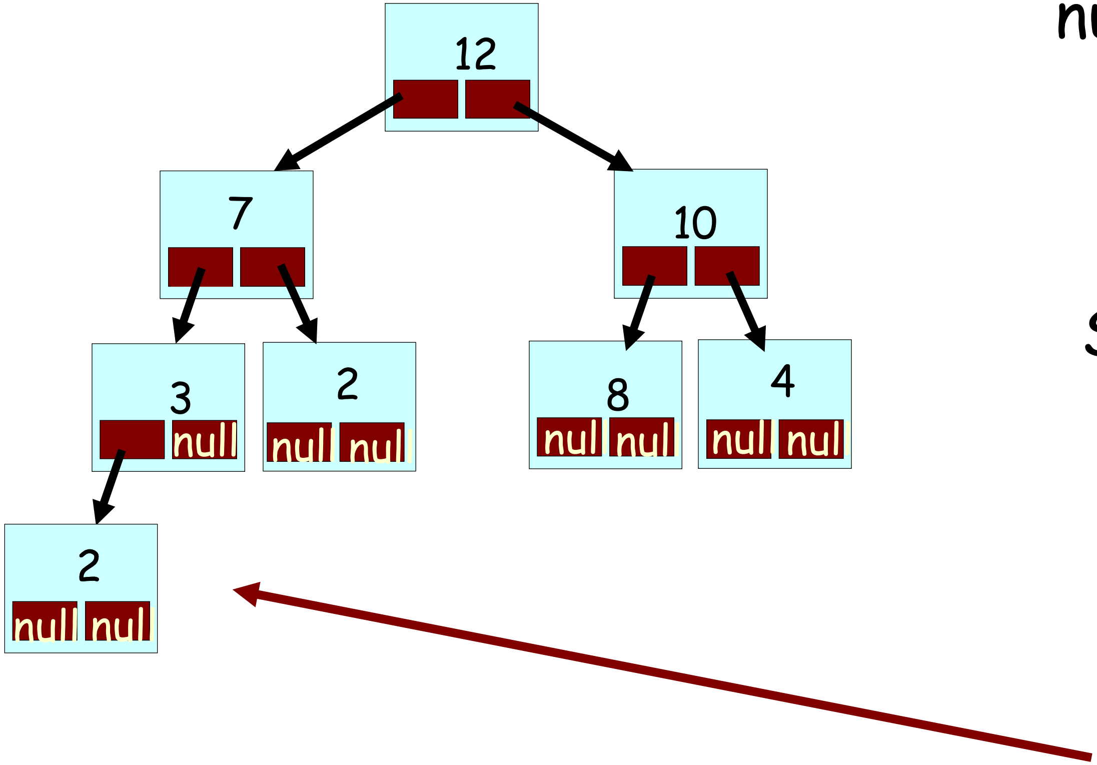
Implementing A Heap

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…

Implementing A Heap

int count;

8

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

int heap[1000];

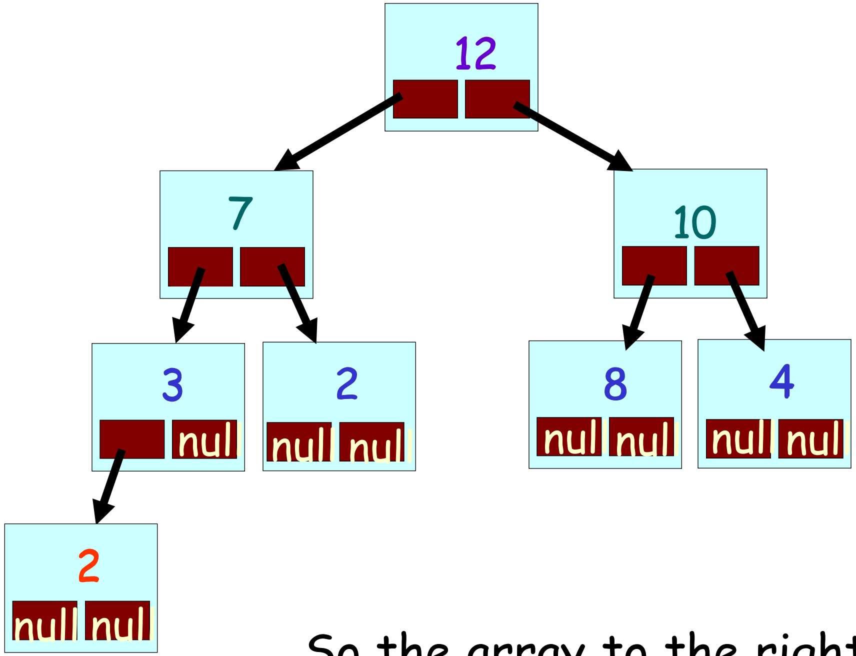
0	12
1	7
2	10
3	3
4	2
5	8
6	4
7	2
8	
9	
10	
11	
12	
13	
…	

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…

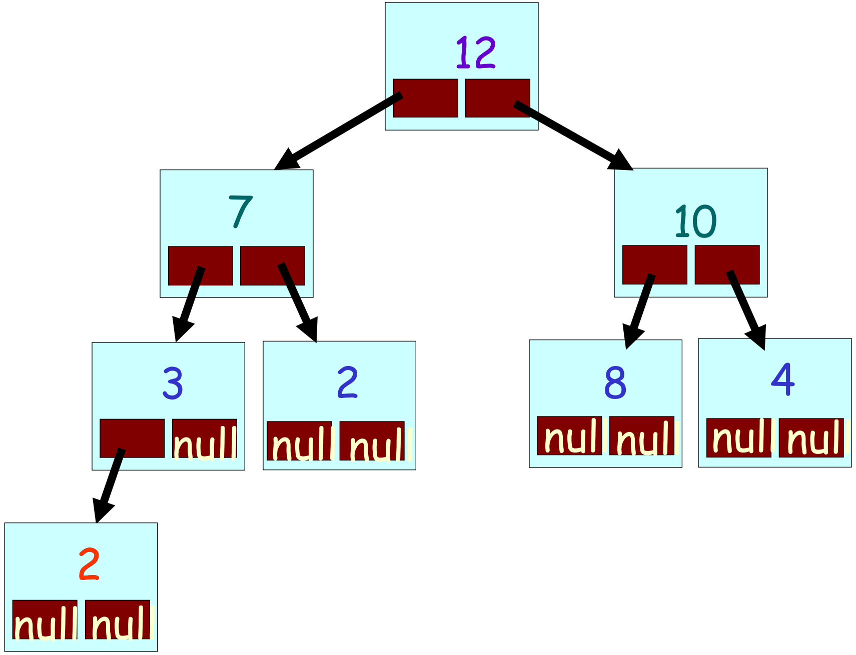
And then the next four values in the next four slots…

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!

Implementing A Heap



int count;

9

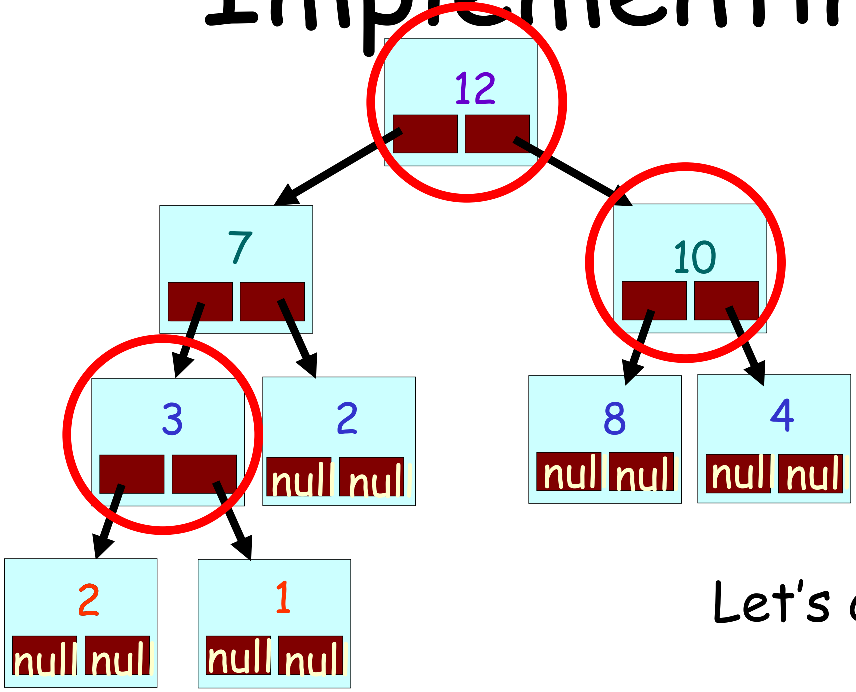
int heap[1000];

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!

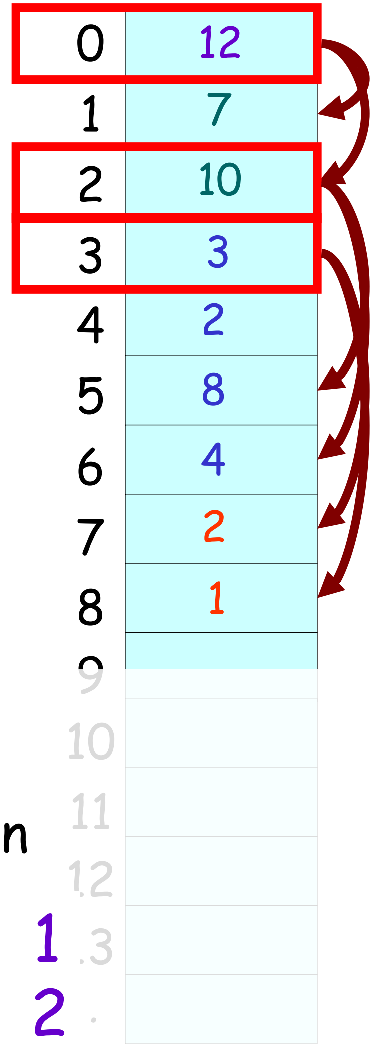
Implementing A Heap



int count;

9

int heap[1000];



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

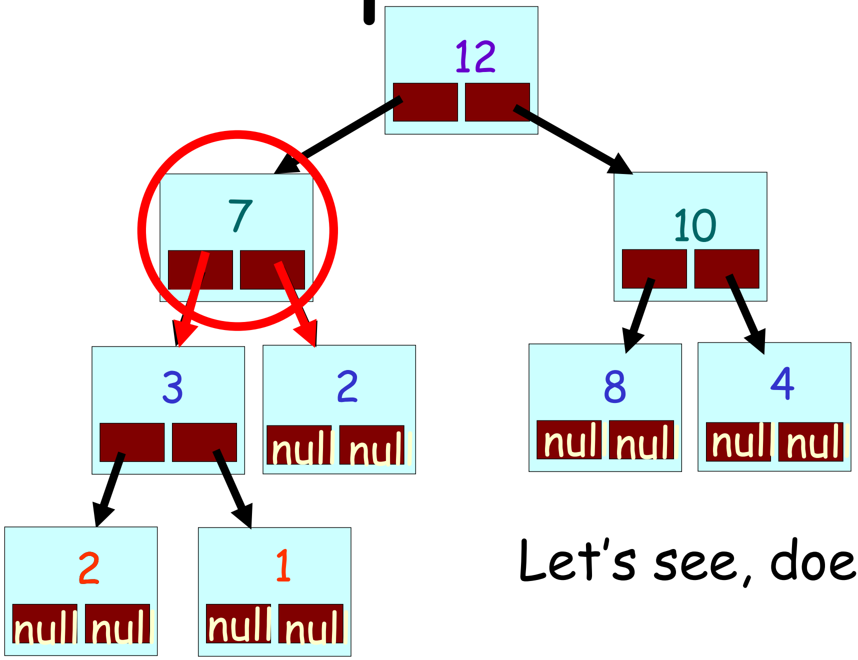
Let's consider some examples:

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

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?

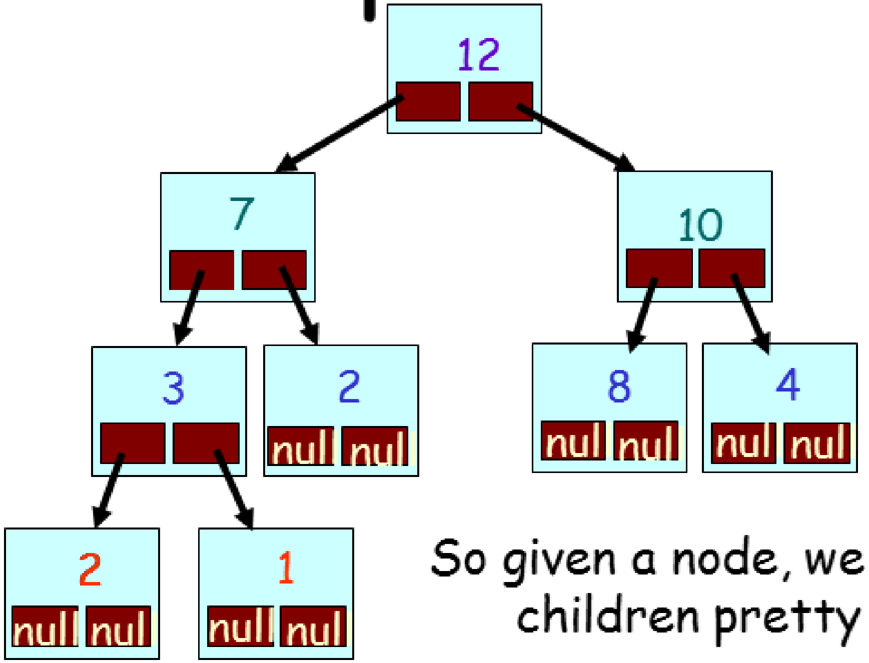
leftChild(1) = 2 * 1 + 1 = slot 3

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

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

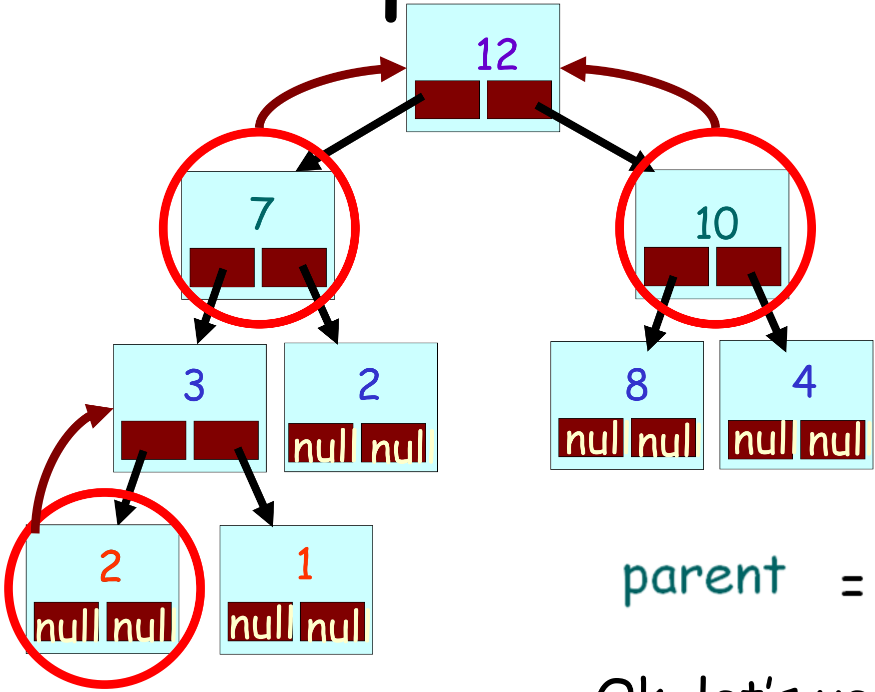
Answer: Use simple algebra!

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

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

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

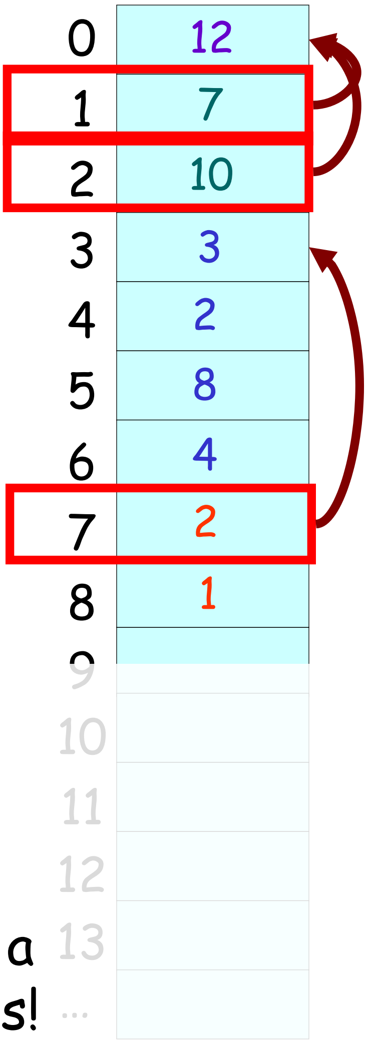
Implementing A Heap



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!

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)

A Heap Helper Class

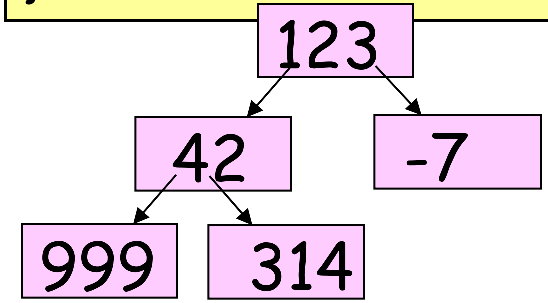
```
class HeapHelper
{
   HeapHelper()              { num = 0;  }
   int GetRootIndex()        { return(0); }
   int LeftChildLoc(int i)   { return(2*i+1); }
   int RightChildLoc(int i)  { return(2*i+2); }
   int ParentLoc(int i)      { return((i-1)/2); }
   int PrintVal(int i)       { cout << a[i];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

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

Output:

num	5



A Heap Helper Class

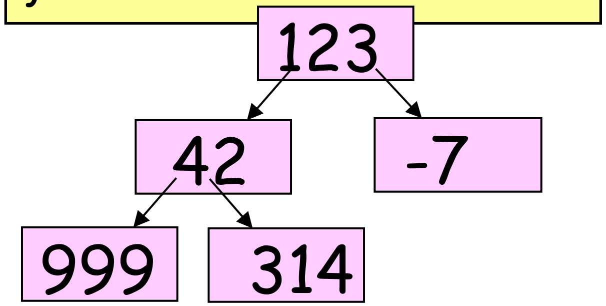
```
class HeapHelper
{
   HeapHelper()             { num = 0;  }
   int GetRootIndex()       { return(0); }
   int LeftChildLoc(int i)  { return(2*i+1); }
   int RightChildLoc(int i) { return(2*i+2); }
   int ParentLoc(int i)     { return((i-1)/2); }
   int PrintVal(int i)      { cout << a[i]; }
   void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

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

num 5

Output:



Using an Array to Implement a Heap

Ok, so now let's see how to use a simple array to implement a maxheap.

To do so, we'll need to be able to easily do the following operations:

Locate the root node of the tree... ☑

Locate (and delete) the bottom-most, right-most node in the tree... ☑

Add a new node in the bottom-most, left-most empty position in the tree... ☑

Easily locate the parent and children of any node in the tree... ☑

Using an Array to Implement a Heap

Ok, so now let's see how to use a simple array to implement a maxheap.

To do so, we'll need to be able to easily do the following operations:

Locate the root node of the tree... ☑

Locate (and delete) the bottom-most, right-most node in the tree... ☑

Add a new node in the bottom-most, left-most empty position in the tree... ☑

Easily locate the parent and children of any node in the tree... ☑

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;

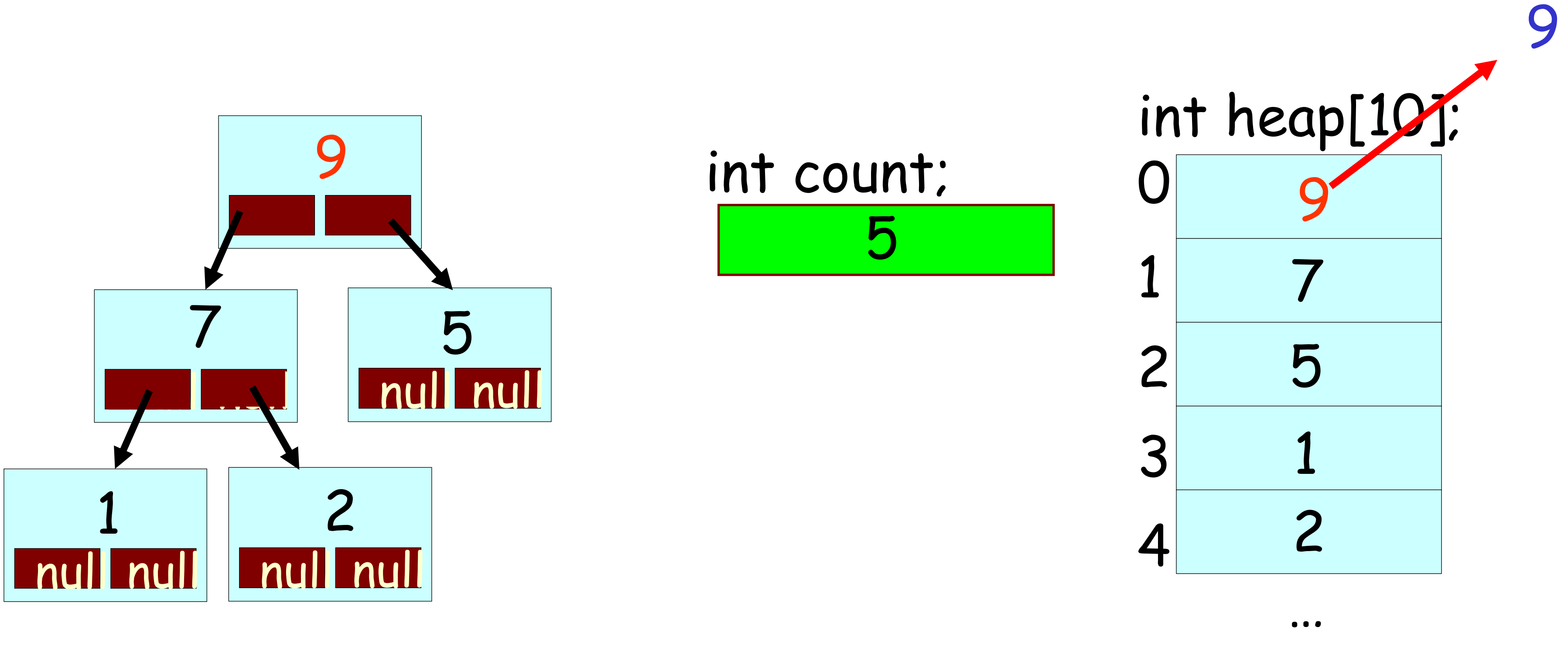
9

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	
…	

Implementing A Heap

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



int count;

5

int heap[10];

0	9
1	7
2	5
3	1
4	2

...

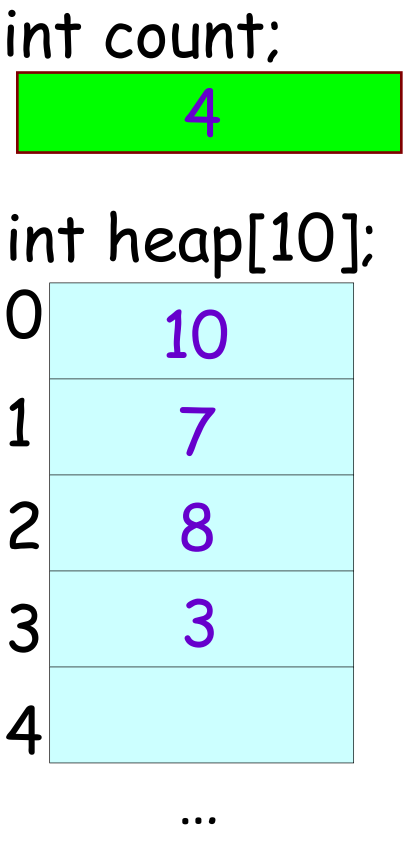
Adding a Node to a Maxheap – The Array Version

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.



int count;

4

int heap[10];

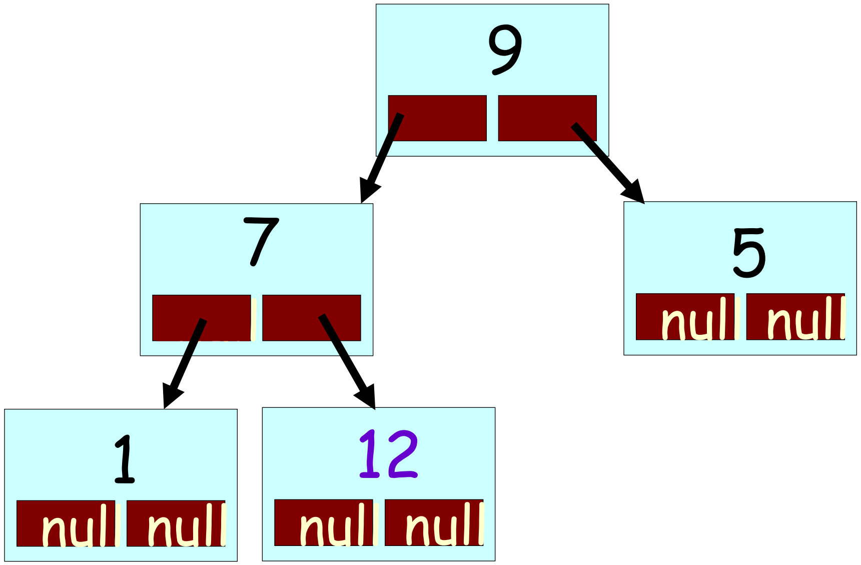
0	10
1	7
2	8
3	3
4	

…

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.



int count;

4

int heap[10];

0	9
1	7
2	5
3	1
4	12

...

Class Challenge

Show a maxheap and its array after inserting each of the following numbers:

1, 6, 4, 5, 0, 8, 3, 12

1. The root of the binary tree goes in array[0]
2. If a node appears in array[i], its children 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)

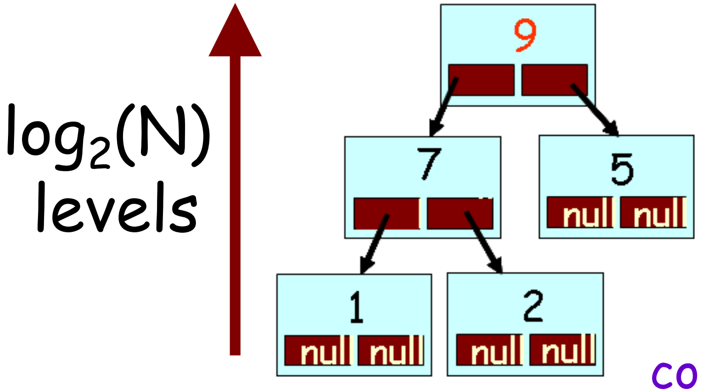
Class Challenge #2

Now show the maxheap and its array after removing the biggest 2 numbers:

1. The root of the binary tree goes in array[0]
2. If a node appears in array[i], its children 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)

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!)

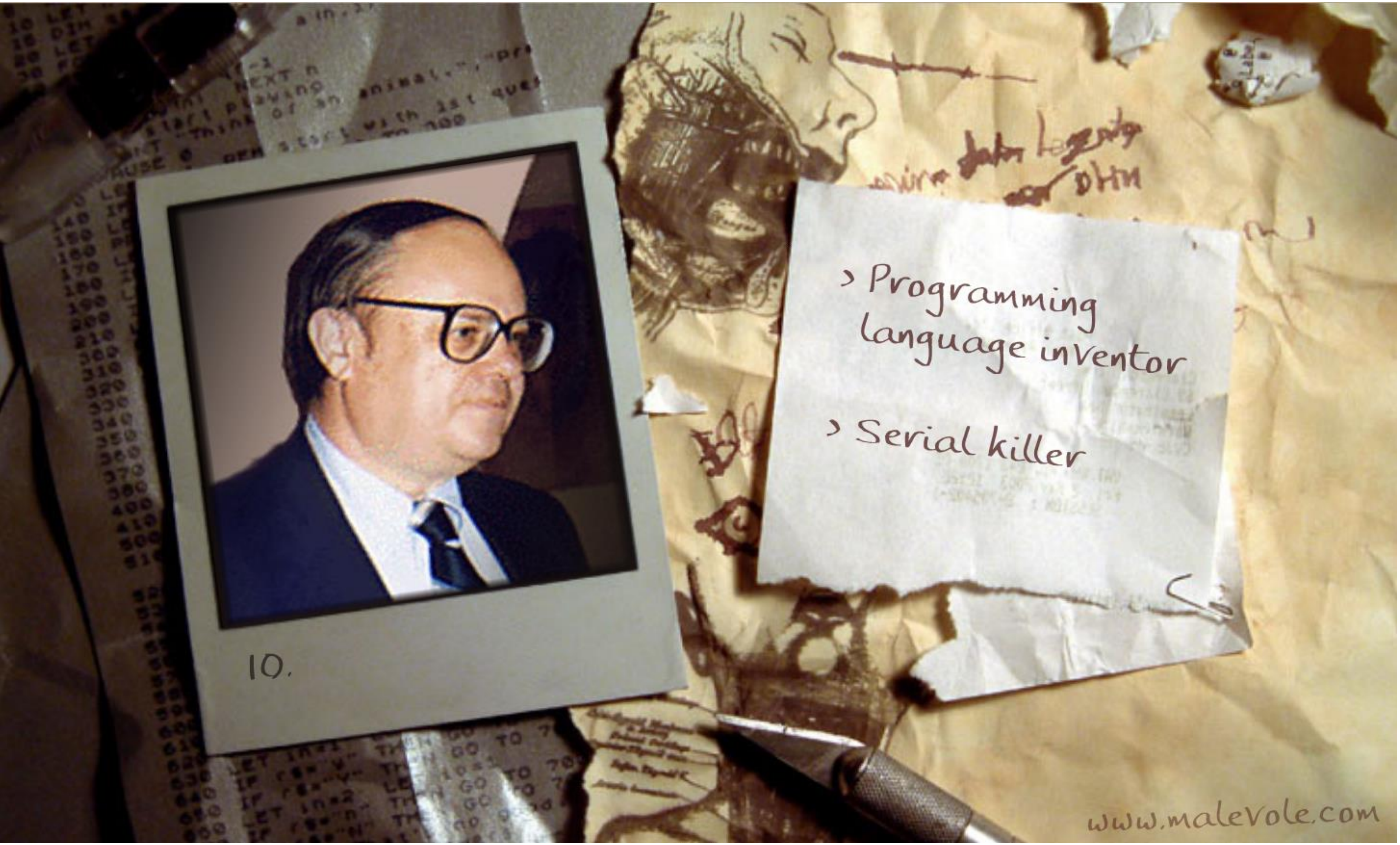
Question: What is the big-oh cost of extracting the maximum/minimum item from a heap?

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 $O(\log_2(n))$

Alright, one last time today…



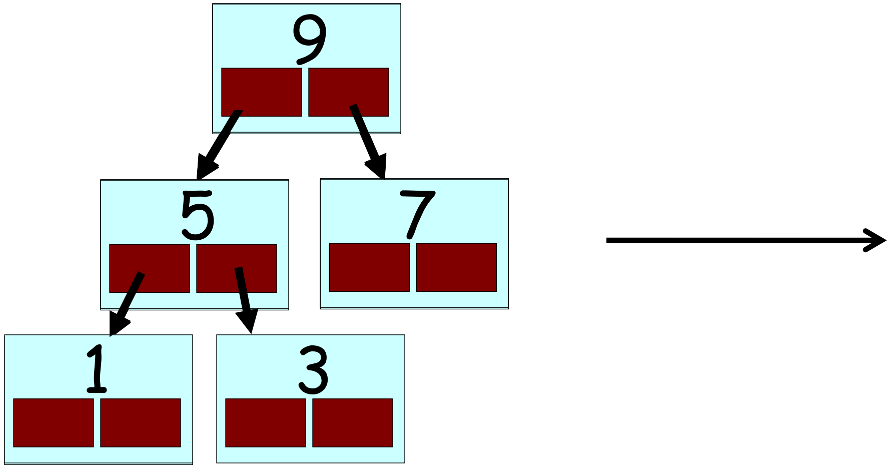
The Heapsort

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 Naïve Heapsort

Question:
What's the complexity of our simple version of heapsort?

Hints:
What is the cost of inserting an item into a maxheap?
What is the cost of extracting an item from a maxheap?
How many items must be inserted and then extracted?

Insertions: N items * $\log_2(N)$ steps per item → N $\log_2(N)$ steps

Extractions: N items * $\log_2(N)$ steps per item → N $\log_2(N)$ steps

That comes to 2 * N $\log_2(N)$ total steps, or O(N $\log_2(N)$).

Not bad! But in fact there's an even faster way to use a heap to sort an array… Let's see it!

The Efficient Heapsort

In our naïve algorithm, we took every item from the array and inserted it into a separate, brand-new maxheap.

So first we built a separate maxheap from scratch, copying every one of our items over!

And then we had to remove each one from our new heap and stick them back into our original array. So SLOW!

Question:
Could we have avoided creating a whole new maxheap and moving our numbers back and forth?

Answer:
Yes! That's the way the "official" Heapsort works! Let's see!

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. ~~Insert all N numbers into a new~~ 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.

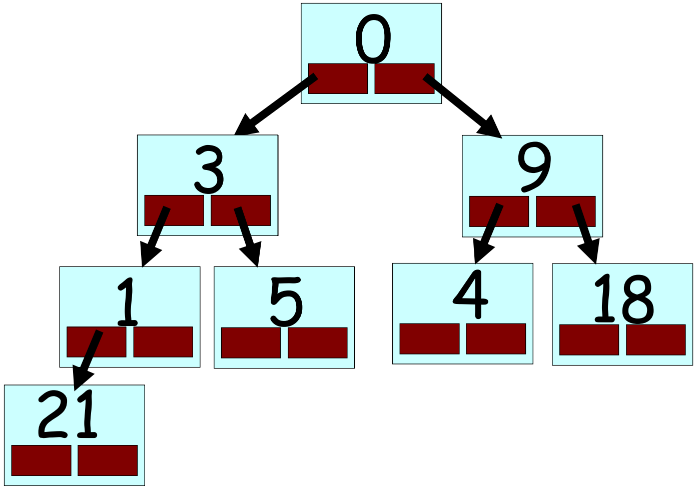
Then we'll use that maxheap as if it were a separate array like before. Only now everything's in just one array.

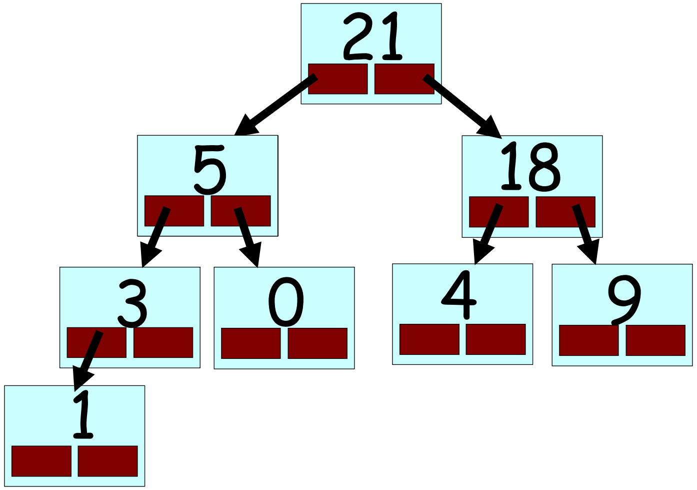
Step #1:

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.

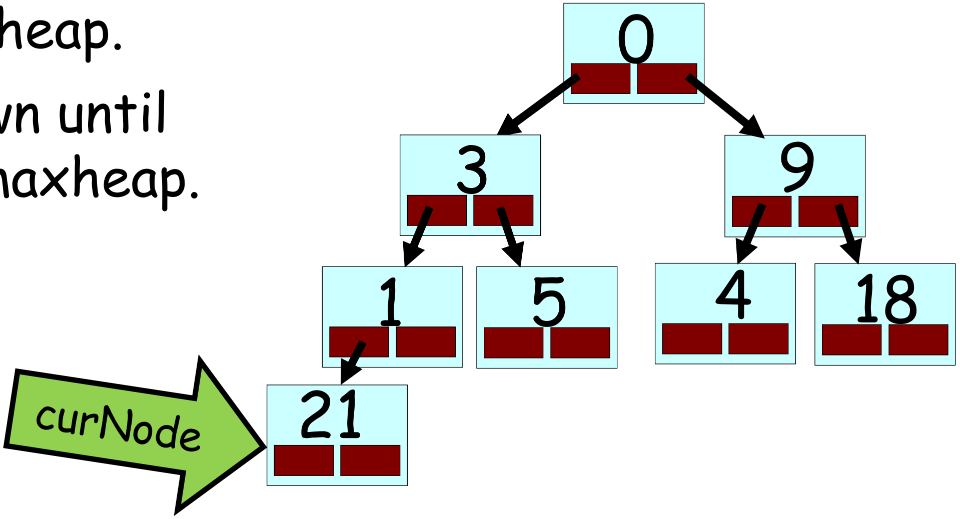
By last node, we mean the bottom-most, right-most node in the tree. This corresponds to the last element in the array.

curNode

0	3	9	1	5	4	18	21

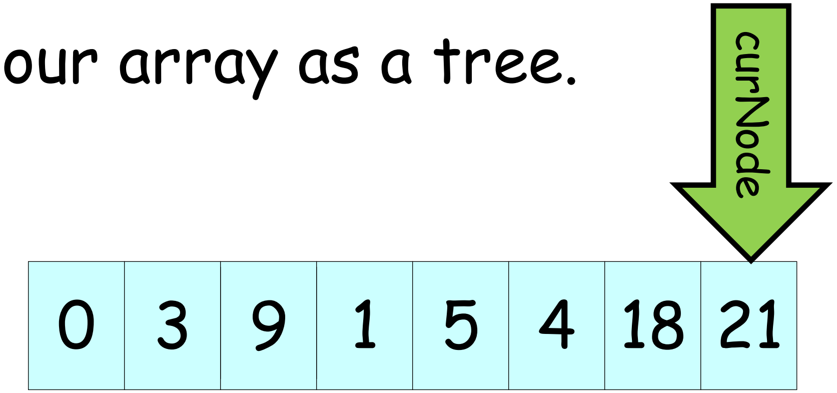
Ok, now here's the algorithm:

```
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.
```



Step #1: Convert Your Input Array into a MaxHeap

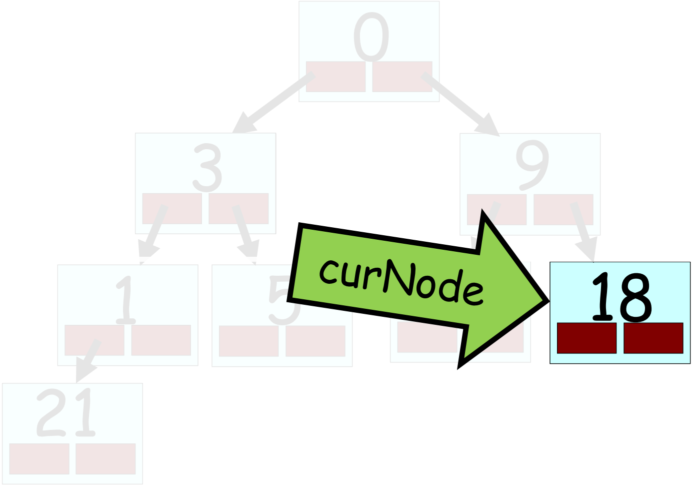
Let's start by visualizing our array as a tree.



0	3	9	1	5	4	18	21

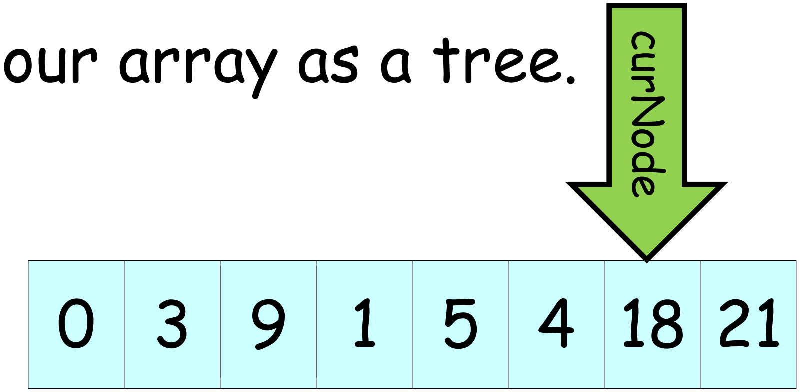
Ok, now here's the algorithm:

```
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.
```



Step #1: Convert Your Input Array into a MaxHeap

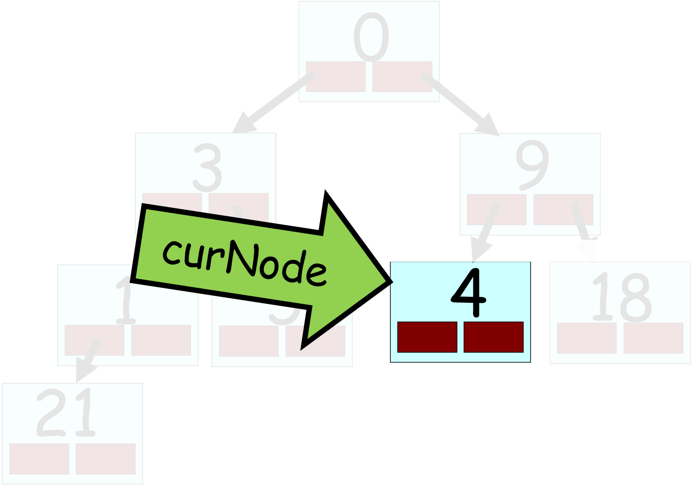
Let's start by visualizing our array as a tree.



0	3	9	1	5	4	18	21

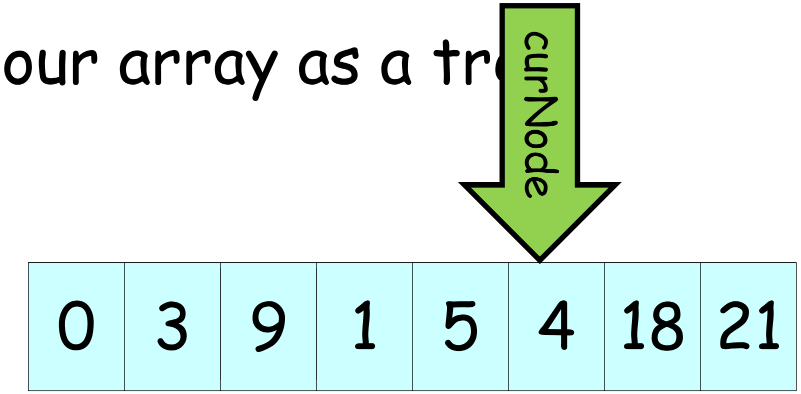
Ok, now here's the algorithm:

```
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.
```



Step #1: Convert Your Input Array into a MaxHeap

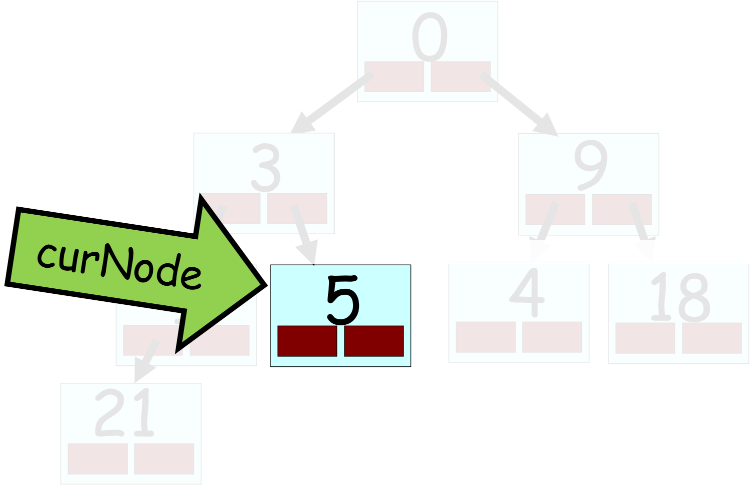
Let's start by visualizing our array as a tr



0	3	9	1	5	4	18	21

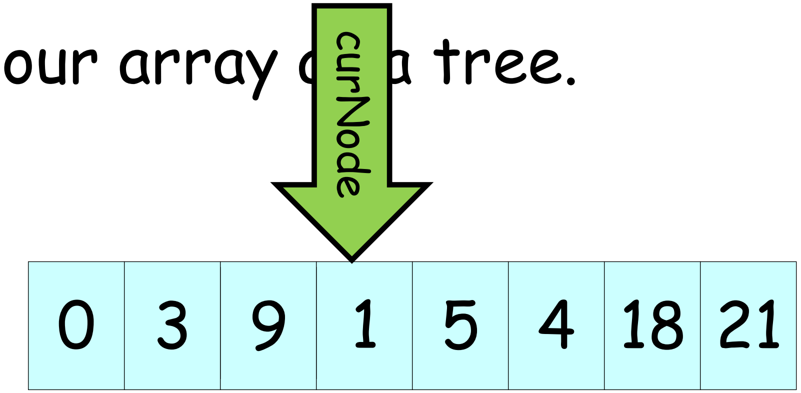
Ok, now here's the algorithm:

```
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.
```



Step #1: Convert Your Input Array into a MaxHeap

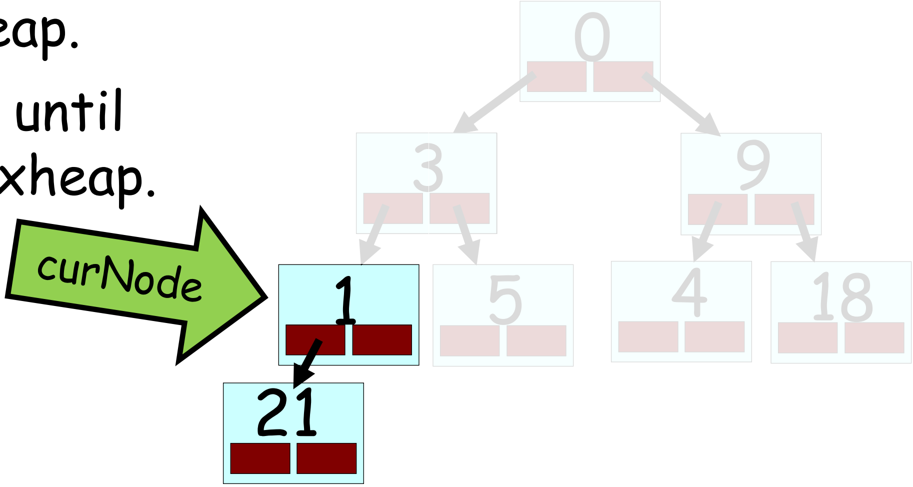
Let's start by visualizing our array as a tree.



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.
```



Step o a MaxHeap

L c a tree.

Of course, our actual algorithm would do these swaps in the array…

In this case, we'd swap array[curNode] with array[2*curNode+1]

Ok

fo Node)

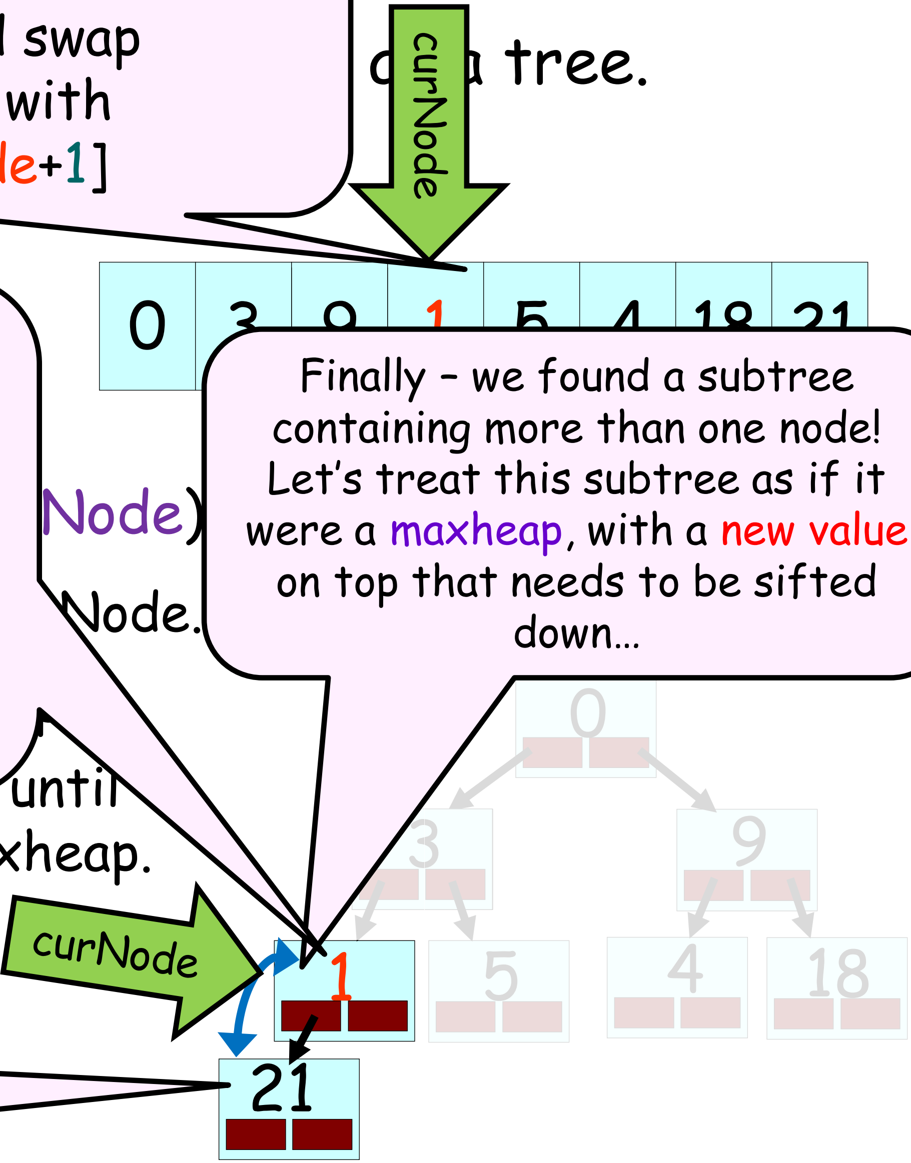
Node.

Now we simply use our normal heapification algorithm to turn this subtree into a maxheap.

We'll keep swapping our root value down with its larger child until it's bigger than both of its children (or hits a leaf)!

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

Finally – we found a subtree containing more than one node! Let's treat this subtree as if it were a maxheap, with a new value on top that needs to be sifted down…



Once we finish, our subtree has been converted into a valid maxheap.

Step #1: Conv… a MaxHeap

Let's star… as a tree.

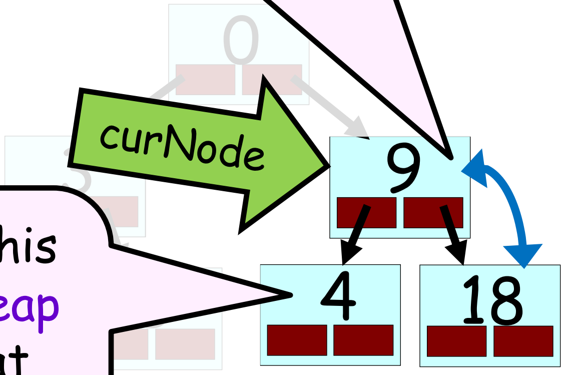
Of course, our actual algorithm would do these swaps in the array… ☺

0	3	9	21	5	4	18	1

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.
```

Excellent! Now this subtree is a valid maxheap too.



Again, let's treat this subtree as a maxheap with a new value at the top that needs to be sifted down…

Step #1: Conv o a MaxHeap

Let's star as a tree.

Of course, our actual algorithm would do these swaps in the array… ☺

Ok, no

0	3	18	21	5	4	9	1

Now this subtree is also a valid maxheap!

Notice how it actually built upon our earlier maxheap that we previously heapified!

for (cu de):

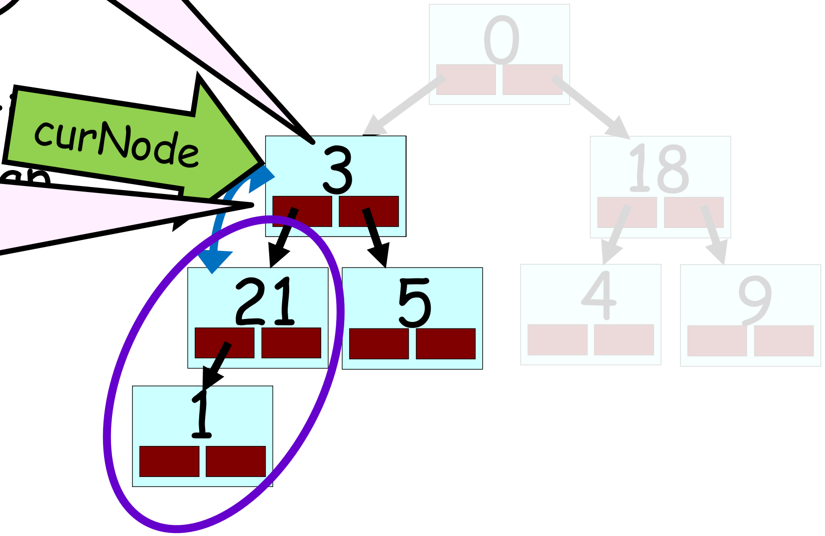
Focu

Think of this subtree as a maxheap.

Keep s own unti

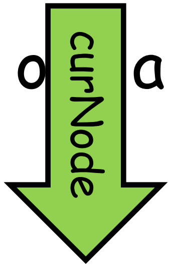
your s heap

Ok, you know the drill. Let's treat this subtree as if it's a maxheap and sift the top value down appropriately.



Step #1: Convert Your Input Array into a MaxHeap

Let’s start by visualizing o array as a tree.



Ok, now here’s the algorithm:

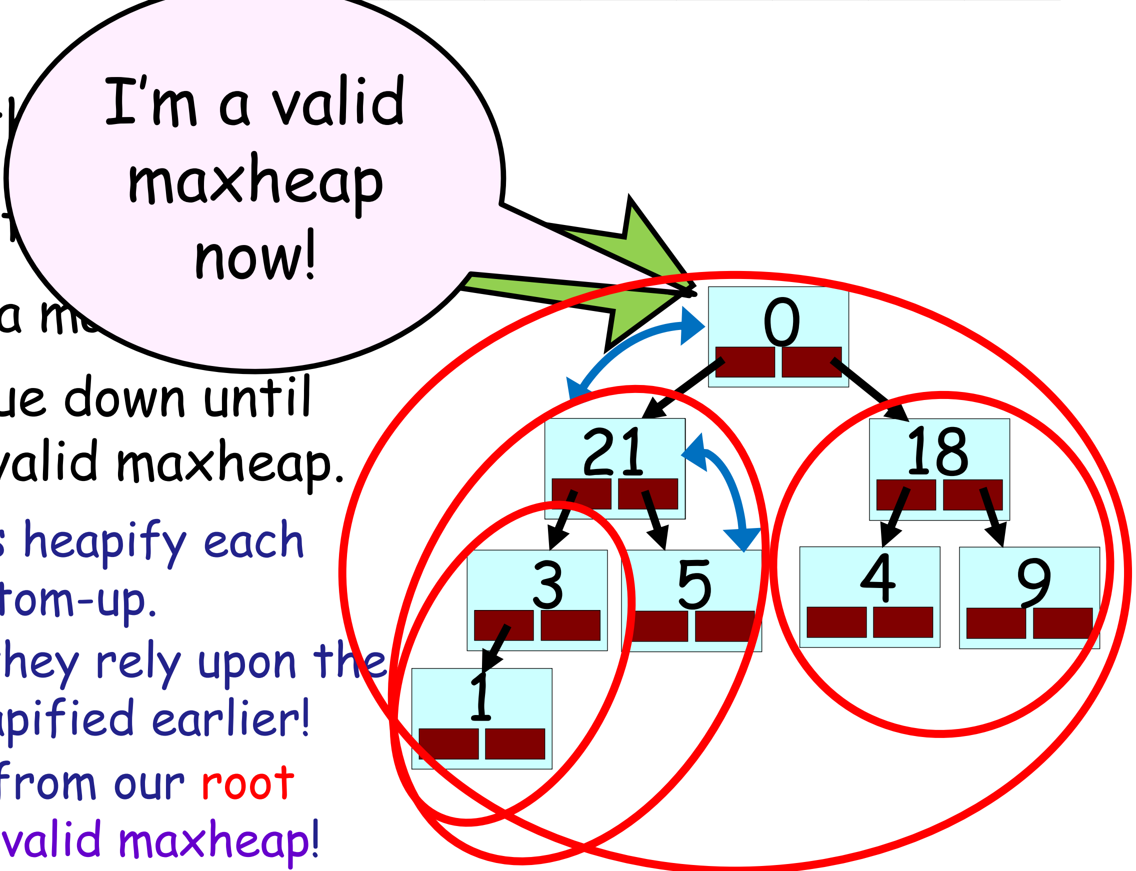
0	21	18	3	5	4	9	1

for (curNode = lastNode th

Focus on the subtree roo

Think of this subtree as a ma

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!

Step #1: Convert Your Input Array into a MaxHeap

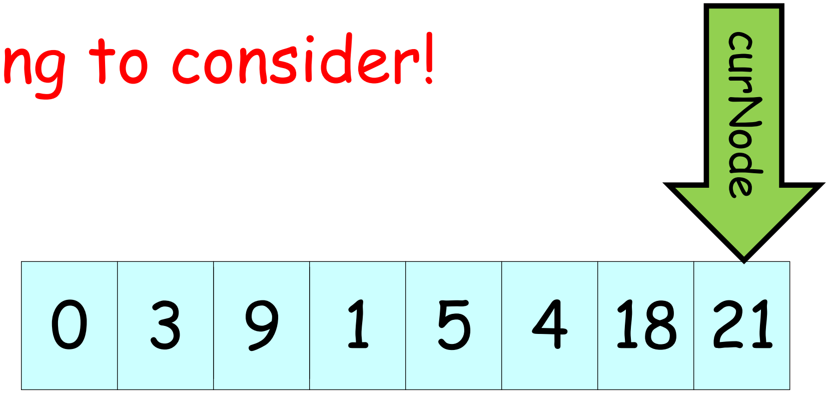
There's one more thing to consider!

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

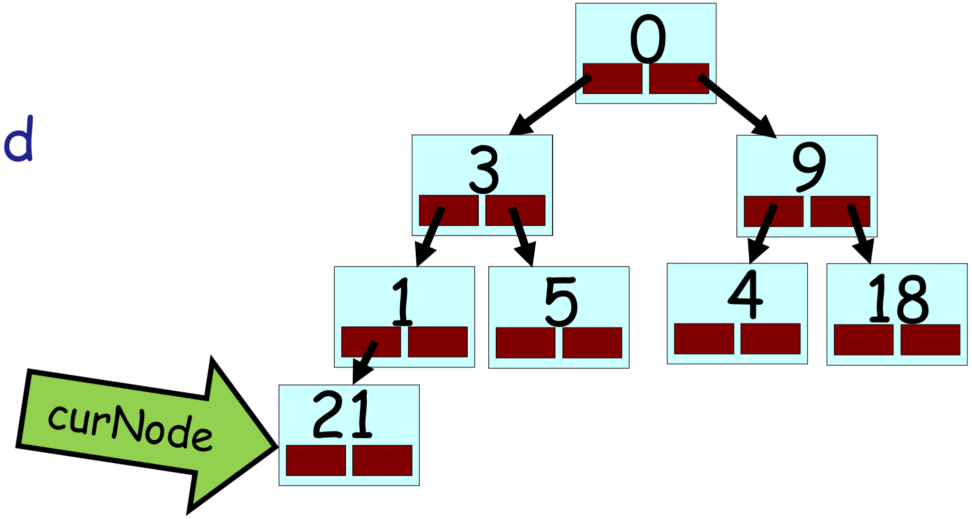
But we only had to reheapify once we reached a sub-tree with at least two nodes.

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

We can – here's how!



0	3	9	1	5	4	18	21



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!

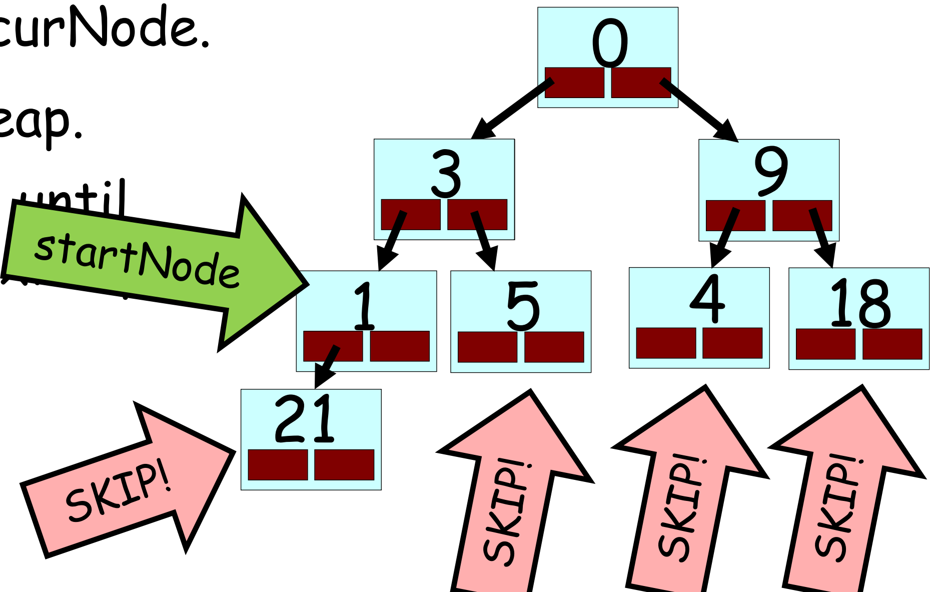
N=8

startNode

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

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 ma
```



This is the complete version of the efficient shuffling algorithm!

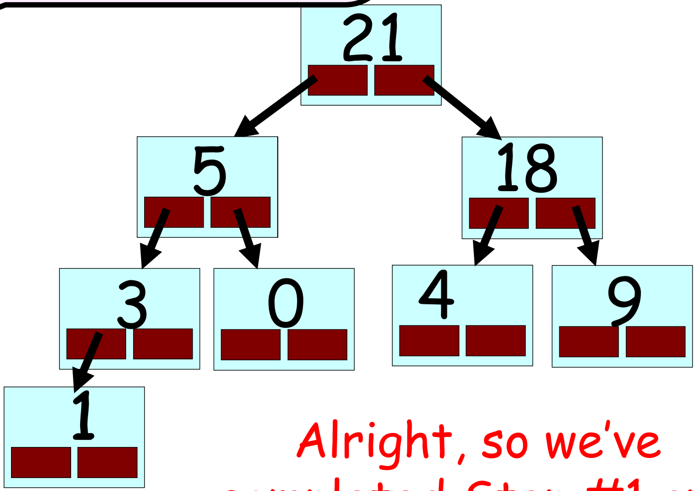
But once we remove the top item, we have to reheapify our heap!

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.

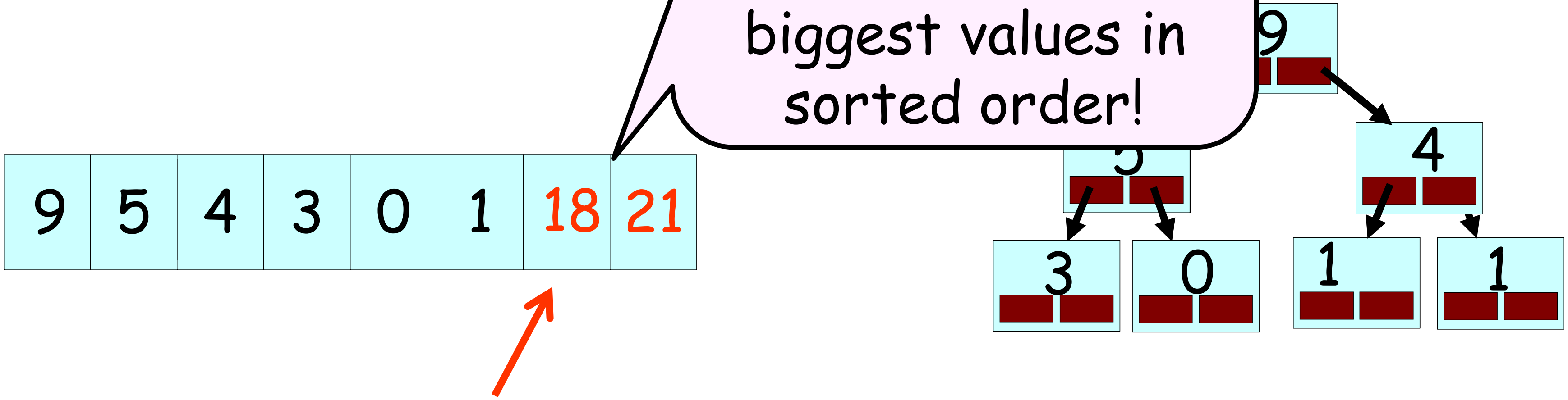
3. Repeatedly swap the just-moved value with the larger of its two children until the value is greater than or equal to both of its children.

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:

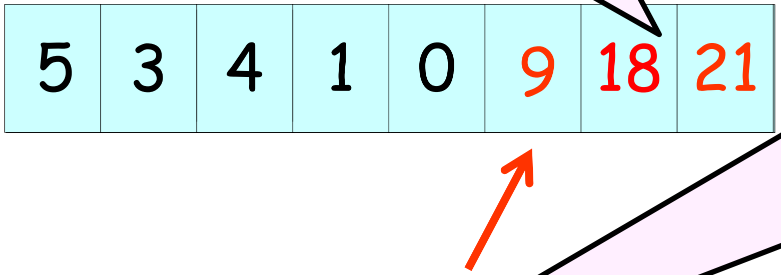
1. Extract the biggest value from the maxheap and re-heapify (just as we learned about 20 slides ago)

   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!

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

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.

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 $O(N \log_2 N)$.

HeapSort Challenge

Challenge: Show how to do an in-place heap-sort with the following array of numbers.

5	3	9	6	15	4	11	16

Step 1: Show the array after each non-leaf is "sifted down" in the array until a valid heap is formed.

Step 2: Show the array after the first 3 items have been removed from the heap and inserted at the end of the array.

(Remember: Sift from j=N/2-1 down-to j=0)

Challenges

Question: Consider a tri-nary tree, where each node has three children, represented by an array. Given a node *i* , where can you find its 3 children in the array? Where can you find its parent?

Question: You need to do a fast table search and you also need to print out records in alphabetical order. Which table ADT will you use?

Question: What is the big-oh of traversing a binary search tree?