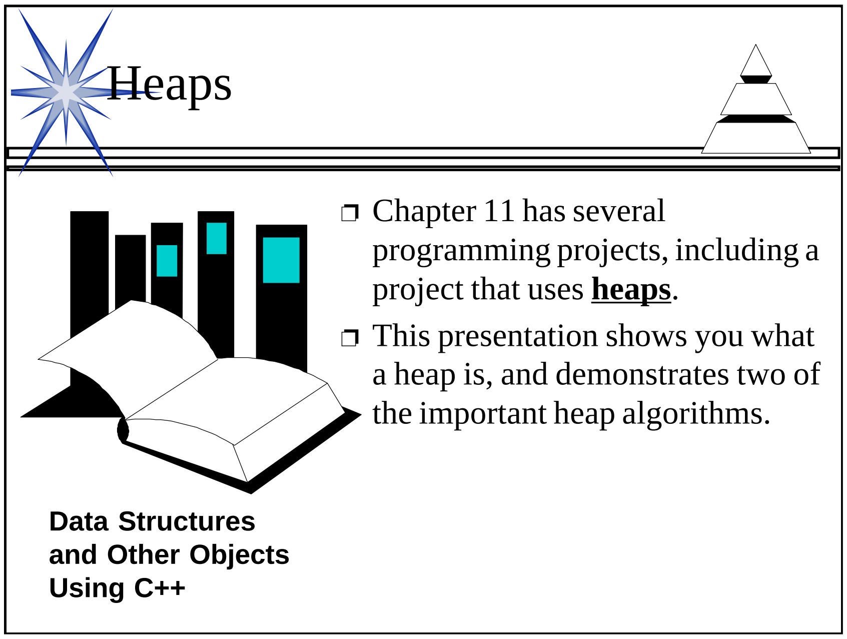


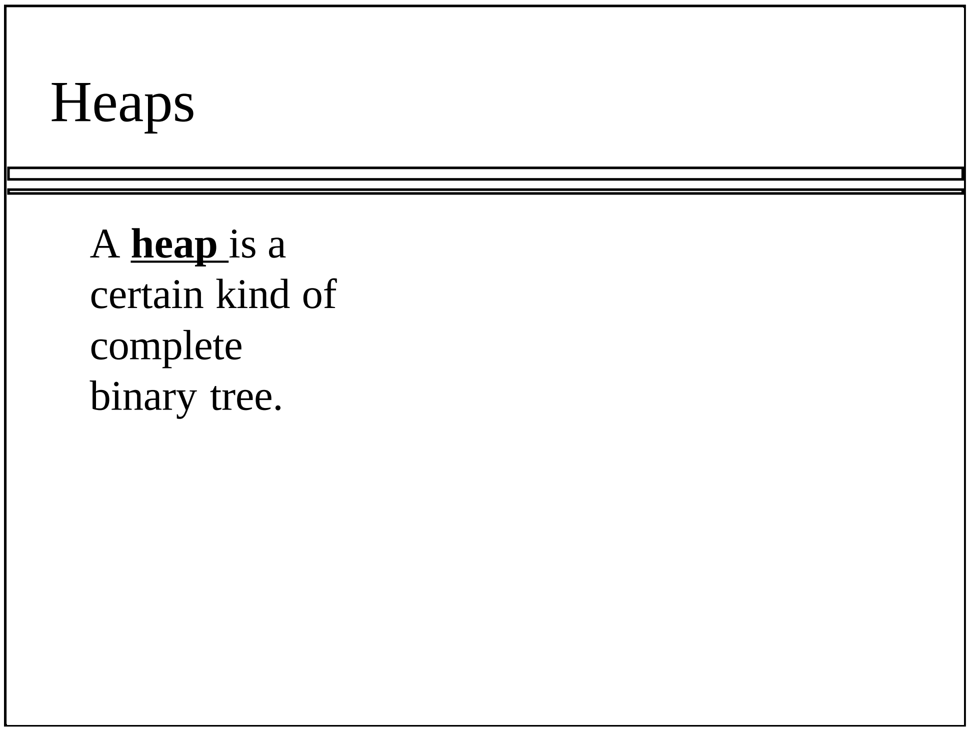
Heaps

- Chapter 11 has several programming projects, including a project that uses **heaps**.
- This presentation shows you what a heap is, and demonstrates two of the important heap algorithms.

Data Structures and Other Objects Using C++

This lecture introduces heaps, which are used in the Priority Queue project of Chapter 11. The lecture includes the algorithms for adding to a heap (including reheapification upward), removing the top of a heap (including reheapification downward), and implementing a heap in a partially-filled array.

Prior to this lecture, the students need a good understanding of complete binary trees. It would also help if they have seen binary search trees and the priority queue class.

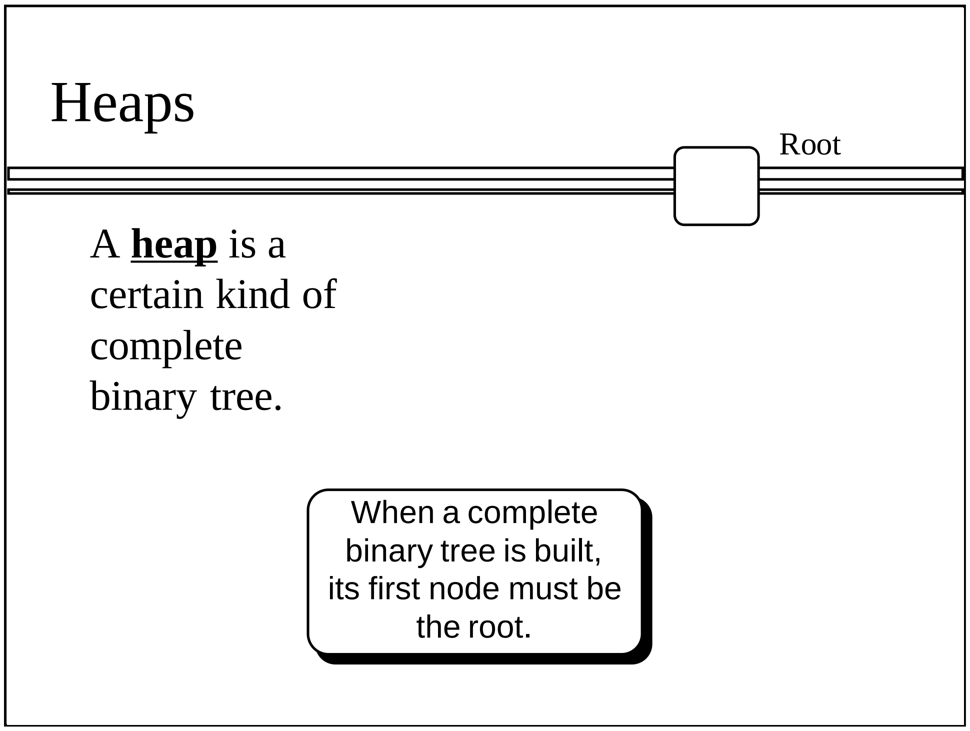


Heaps

A **heap** is a certain kind of complete binary tree.

A heap is a data structure with several applications, including a way to implement Priority Queues, as shown in Chapter 11. The definition of a heap is a special kind of complete binary tree.

You probably recall that a complete binary tree requires that its nodes are added in a particular order...



Heaps

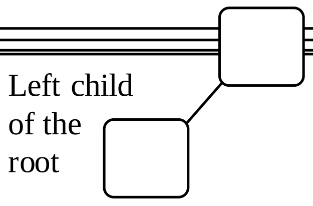
A **heap** is a certain kind of complete binary tree.

When a complete binary tree is built, its first node must be the root.

The first node of a complete binary tree is always the root...

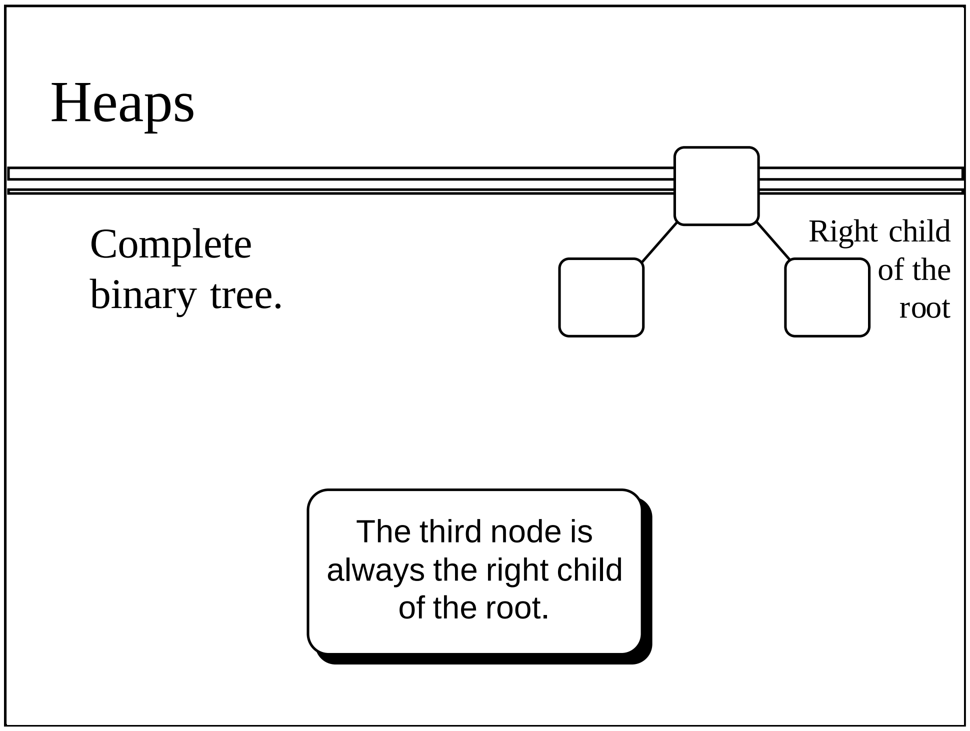
Heaps

Complete binary tree.



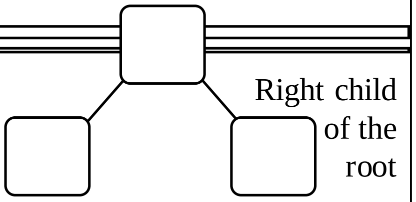
The second node is always the left child of the root.

...the second node is always the left child of the root...



Heaps

Complete binary tree.

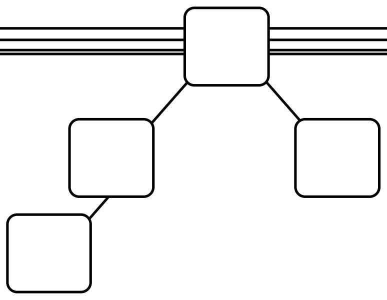


The third node is always the right child of the root.

...then the right child of the root...

Heaps

Complete
binary tree.

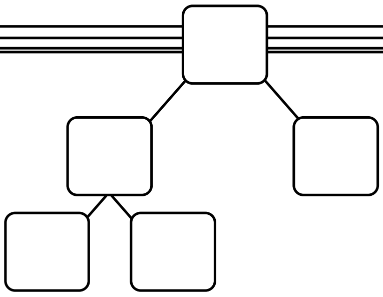


The next nodes always fill the next level from left-to-right.

...and so on. The nodes always fill each level from left-to-right...

Heaps

Complete binary tree.

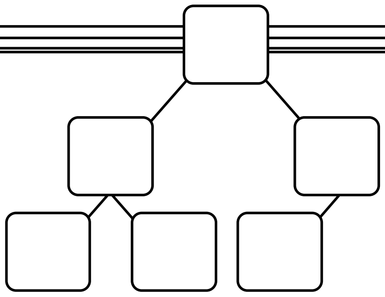


The next nodes always fill the next level from left-to-right.

...from left-to-right...

Heaps

Complete
binary tree.

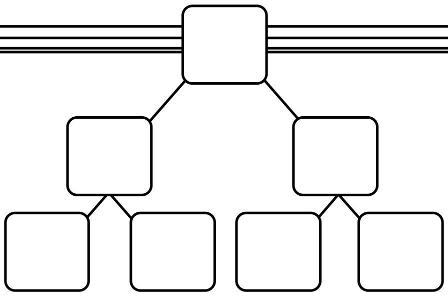


The next nodes always fill the next level from left-to-right.

...from left-to-right...

Heaps

Complete binary tree.

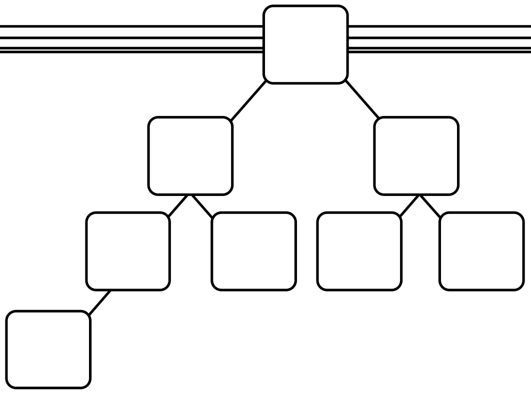


The next nodes always fill the next level from left-to-right.

...from left-to-right...

Heaps

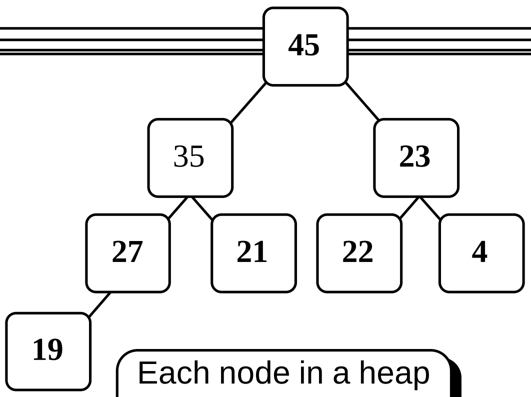
Complete
binary tree.



...and when a level is filled you start the next level at the left.

Heaps

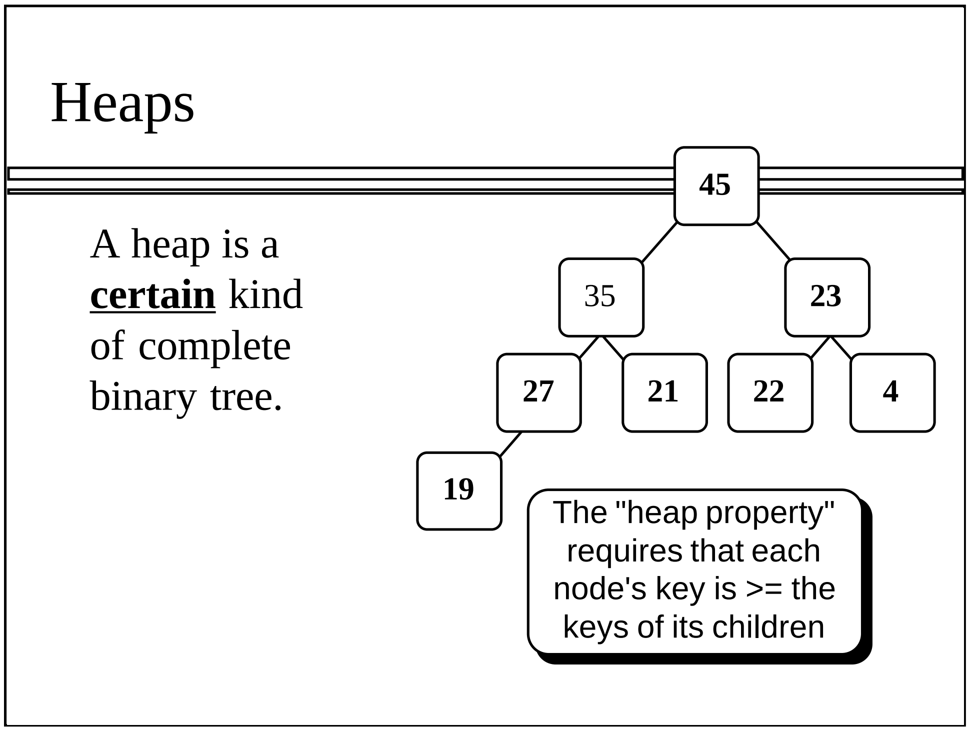
A heap is a **certain** kind of complete binary tree.



Each node in a heap contains a key that can be compared to other nodes' keys.

So, a heap is a complete binary tree. Each node in a heap contains a key, and these keys must be organized in a particular manner. Notice that this is not a binary search tree, but the keys do follow some semblance of order.

Can you see what rule is being enforced here?



Heaps

A heap is a **certain** kind of complete binary tree.

The "heap property" requires that each node's key is >= the keys of its children

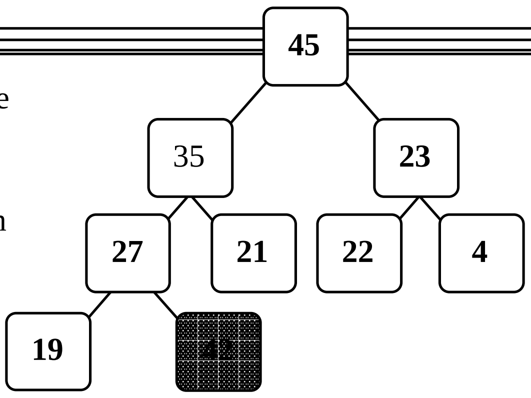
The heap property requires that each node's key is >= to the keys of its children.

This is a handy property because the biggest node is always at the top. Because of this, a heap can easily implement a priority queue (where we need quick access to the highest priority item).

Adding a Node to a Heap

❶ Put the new node in the next available spot.

❷ Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



We can add new elements to a heap whenever we like. Because the heap is a complete binary search tree, we must add the new element at the next available location, filling in the levels from left-to-right.

In this example, I have just added the new element with a key of 42.

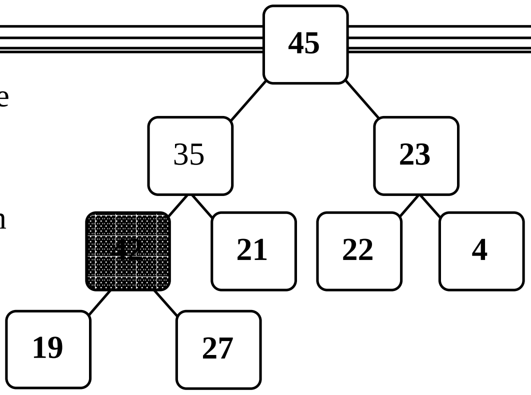
Of course, we now have a problem: The heap property is no longer valid. The 42 is bigger than its parent 27.

To fix the problem, we will push the new node upwards until it reaches an acceptable location.

Adding a Node to a Heap

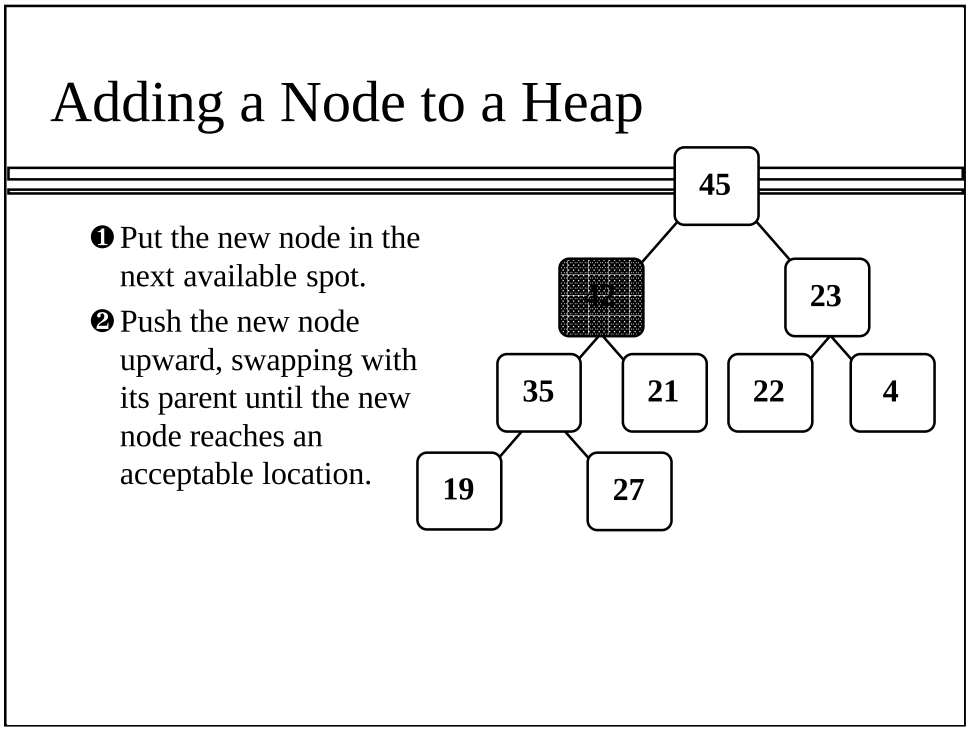
❶ Put the new node in the next available spot.

❷ Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Here we have pushed the 42 upward one level, swapping it with its smaller parent 27.

We can't stop here though, because the parent 35 is still smaller than the new node 42.



Adding a Node to a Heap

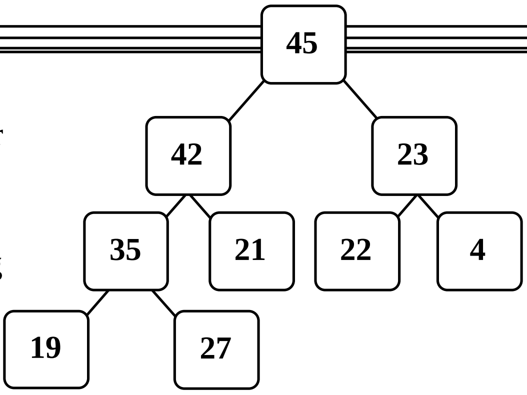
❶ Put the new node in the next available spot.

❷ Push the new node upward, swapping with its parent until the new node reaches an acceptable location.

Can we stop now? Yes, because the 42 is less than or equal to its parent.

Adding a Node to a Heap

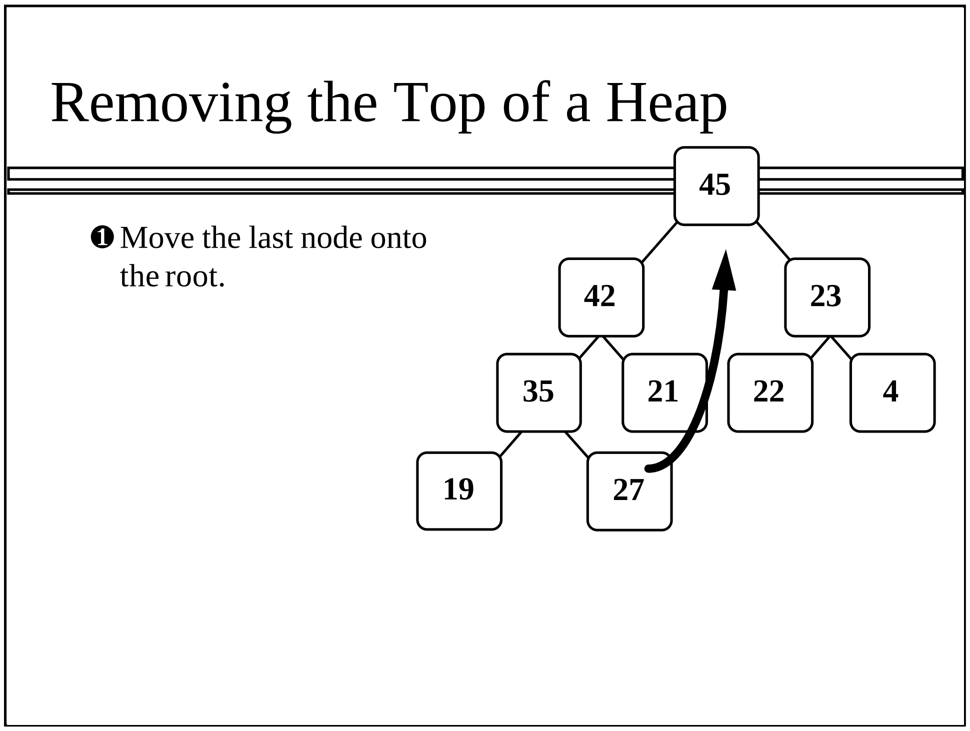
- ✔ The parent has a key that is >= new node, or
- ✔ The node reaches the root.
- ➚ The process of pushing the new node upward is called **<u>reheapification upward</u>**.



In general, there are two conditions that can stop the pushing upward:

1. We reach a spot where the parent is >= the new node, or
2. We reach the root.

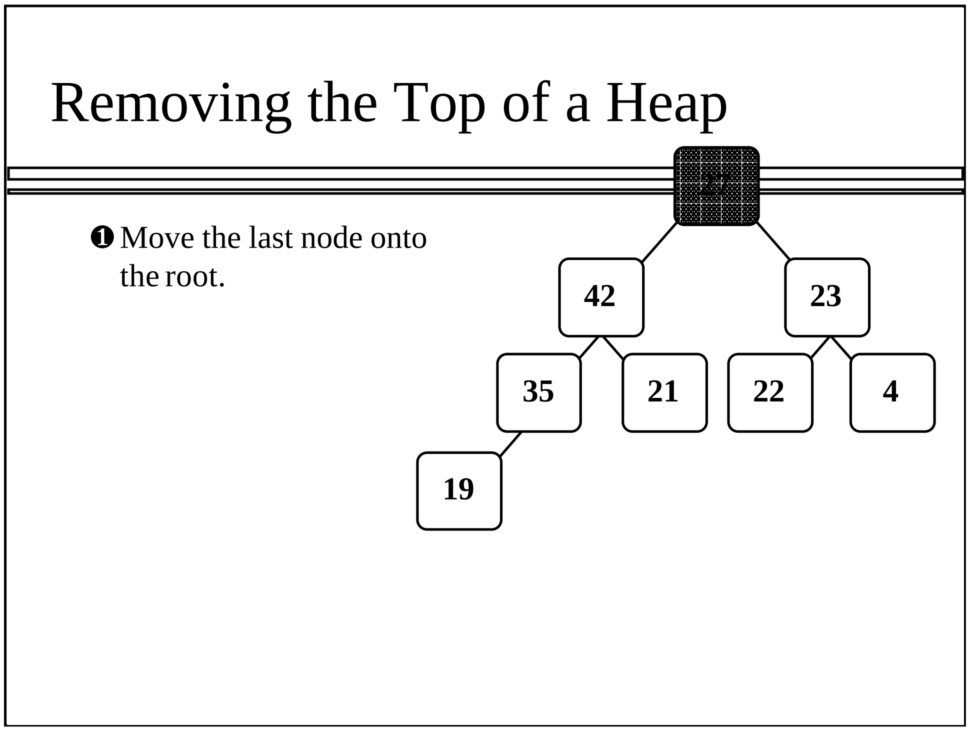
This process is called <u>reheapification upward</u> (I didn't just make up that name, really).



Removing the Top of a Heap

❶ Move the last node onto the root.

We can also remove the top node from a heap. The first step of the removal is to move the last node of the tree onto the root. In this example we move the 27 onto the root.



Removing the Top of a Heap

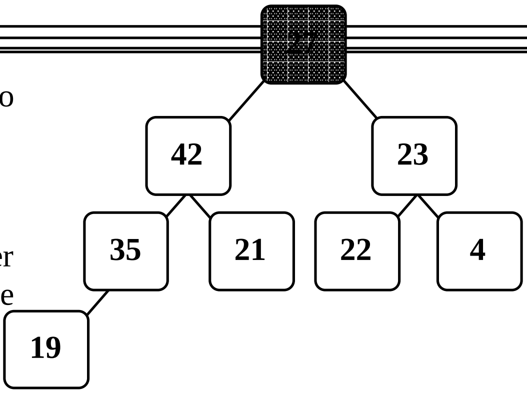
❶ Move the last node onto the root.

Now the 27 is on top of the heap, and the original root (45) is no longer around. But the heap property is once again violated.

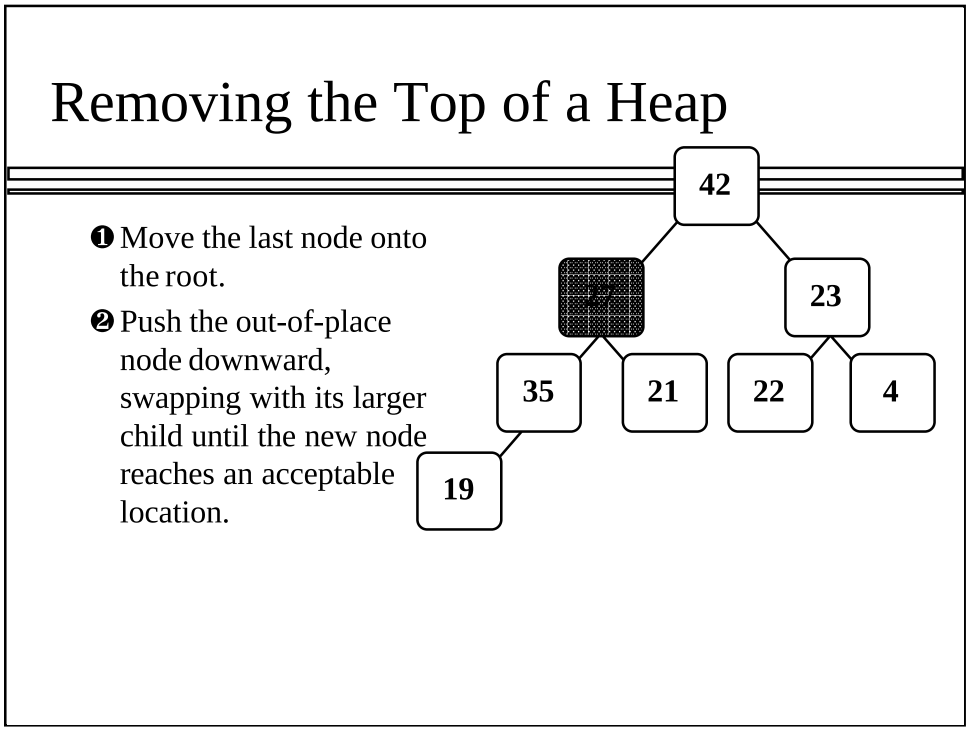
Removing the Top of a Heap

❶ Move the last node onto the root.

❷ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



We'll fix the problem by pushing the out-of-place node downward. Perhaps you can guess what the downward pushing is called....reheapification downward.



Removing the Top of a Heap

❶ Move the last node onto the root.

❷ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.

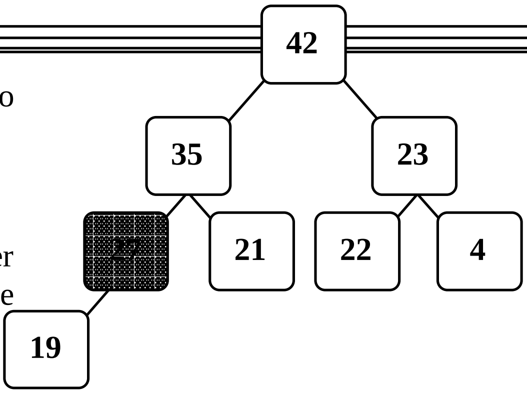
When we push a node downward it is important to swap it with its largest child. (Otherwise we are creating extra problems by placing the smaller child on top of the larger child.) This is what the tree looks like after one swap.

Should I continue with the reheapification downward?

Removing the Top of a Heap

❶ Move the last node onto the root.

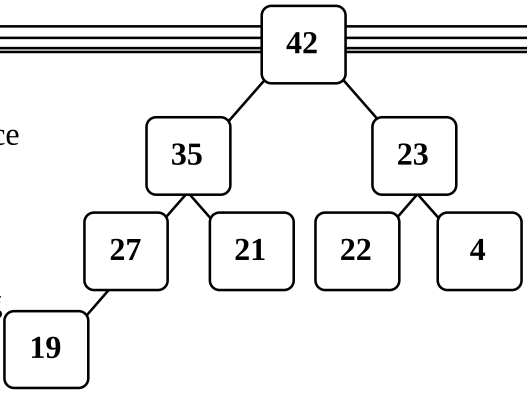
❷ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



Yes, I swap again, and now the 27 is in an acceptable location.

Removing the Top of a Heap

- ✔ The children all have keys <= the out-of-place node, or
- ✔ The node reaches the leaf.
- ➘ The process of pushing the new node downward is called **reheapification downward**.

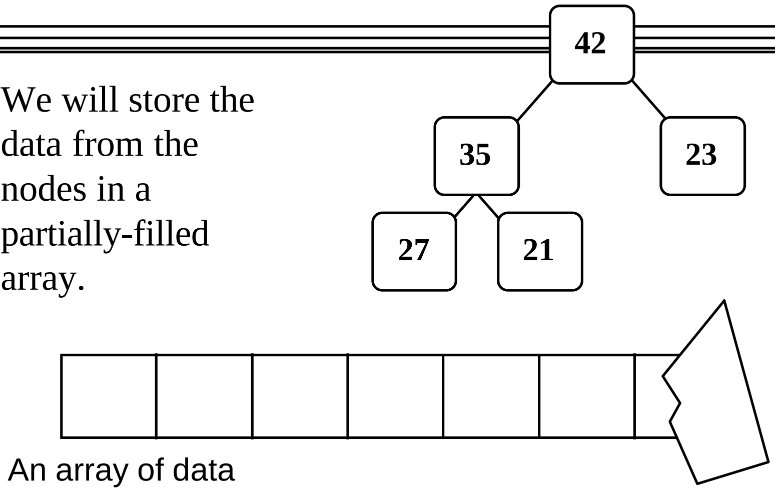


Reheapification downward can stop under two circumstances:

1. The children all have keys that are <= the out-of-place node.
2. The out-of-place node reaches a leaf.

Implementing a Heap

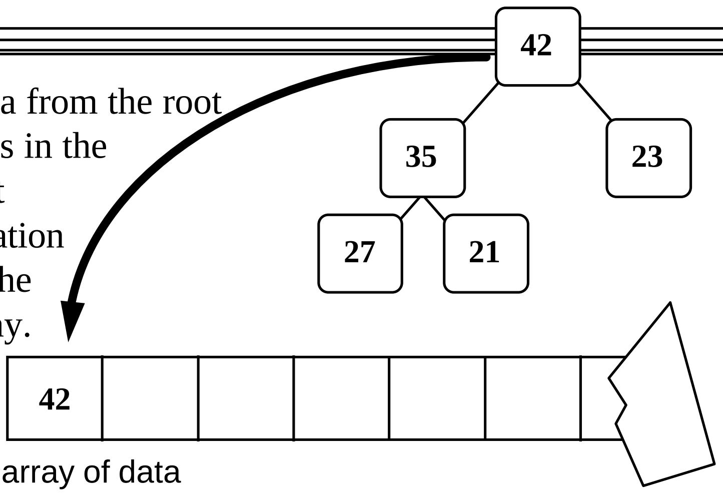
- We will store the data from the nodes in a partially-filled array.



This slide shows the typical way that a heap is implemented. For the most part, there is nothing new here, because you already know how to implement a complete binary tree using a partially-filled array. That is what we are doing with the heap.

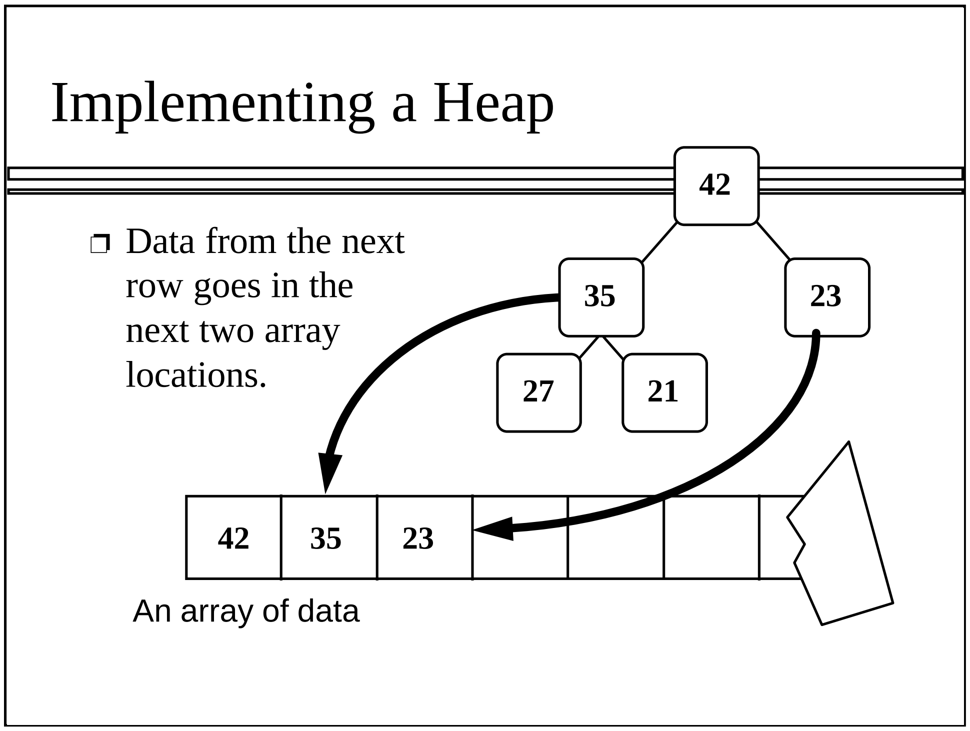
Implementing a Heap

- Data from the root goes in the first location of the array.



An array of data

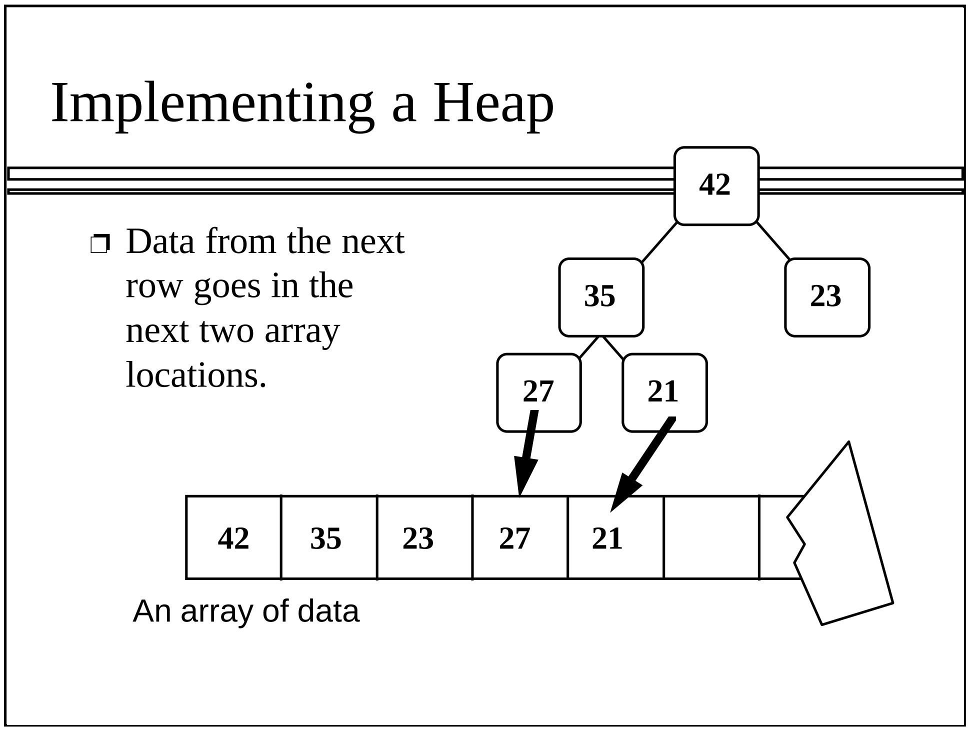
Following the usual technique for implementing a complete binary tree, the data from the root is stored in the first entry of the array.



Implementing a Heap

- Data from the next row goes in the next two array locations.

The next two nodes go in the next two locations of the array.



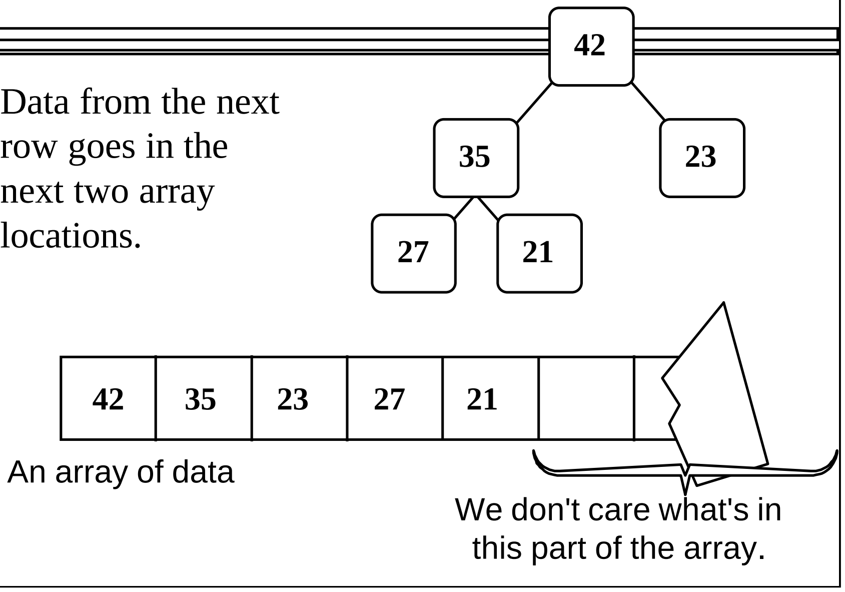
Implementing a Heap

- Data from the next row goes in the next two array locations.

and so on.

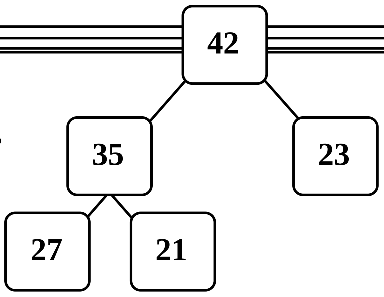
Implementing a Heap

❐ Data from the next row goes in the next two array locations.

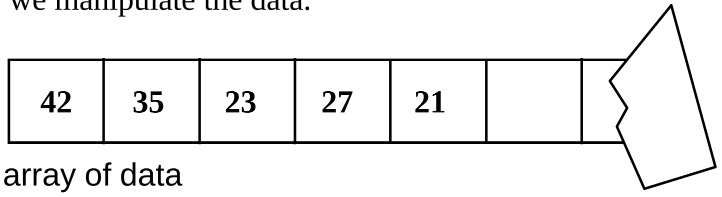


As with any partially-filled array, we are only concerned with the front part of the array. If the tree has five nodes, then we are only concerned with the entries in the first five components of the array.

Important Points about the Implementation



- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

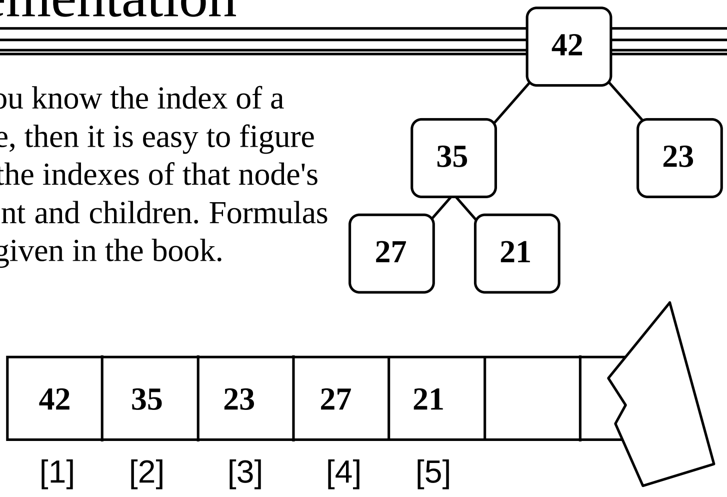


An array of data

With this implementation of a heap, there are no pointers. The only way that we know that the array is a heap is the manner in which we manipulate it.

Important Points about the Implementation

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children. Formulas are given in the book.

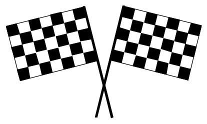


The manipulations are the same manipulations that you've used for a complete binary tree, making it easy to compute the index where various nodes are stored.

Summary

- A heap is a complete binary tree, where the entry at each node is greater than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.

A quick summary . . .



THE END

Feel free to send your ideas to:

Michael Main
main@colorado.edu