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.
Heaps

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

Complete binary tree.

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.
Heaps

Complete binary tree.

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

Complete binary tree.

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

Complete binary tree.

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

Complete binary tree.

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

Complete binary tree.
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.
Heaps

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

The "heap property" requires that each node's key is $\geq$ the keys of its children.
Adding a Node to a Heap

1. Put the new node in the next available spot.
2. Push the new node upward, swapping with its parent until the new node 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.
Adding a Node to a Heap

1. Put the new node in the next available spot.
2. Push the new node upward, swapping with its parent until the new node reaches an acceptable location.
Adding a Node to a Heap

✔ The parent has a key that is $\geq$ new node, or
✔ The node reaches the root.
➚ The process of pushing the new node upward is called **reheapification upward**.
Removing the Top of a Heap

1. Move the last node onto the root.
Removing the Top of a Heap

1. Move the last node onto the root.
Removing the Top of a Heap

1. Move the last node onto the root.
2. Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.
Removing the Top of a Heap

1. Move the last node onto the root.
2. Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.
Removing the Top of a Heap

1. Move the last node onto the root.
2. Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.
Removing the Top of a Heap

✔ The children all have keys $\leq$ the out-of-place node, or
✔ The node reaches the leaf.

△ The process of pushing the new node downward is called **reheapification** downward.
Implementing a Heap

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

An array of data
Implementing a Heap

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

An array of data
Implementing a Heap

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

An array of data
Implementing a Heap

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

An array of data
Implementing a Heap

- Data from the next row goes in the next two array locations.
- We don't care what's in this part of the array.

An array of data
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
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.
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.
Presentation copyright 1997 Addison Wesley Longman,  
For use with *Data Structures and Other Objects Using C++*  
by Michael Main and Walter Savitch.  

Some artwork in the presentation is used with permission from Presentation Task Force  
(copyright New Vision Technologies Inc) and Corel Gallery Clipart Catalog (copyright  
Corel Corporation, 3G Graphics Inc, Archive Arts, Cartesia Software, Image Club  

Students and instructors who use *Data Structures and Other Objects Using C++* are welcome  
to use this presentation however they see fit, so long as this copyright notice remains  
intact.

THE END