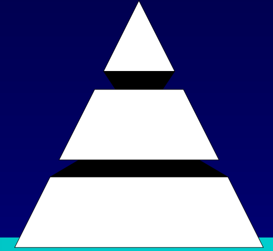
CSC212
Data Structure

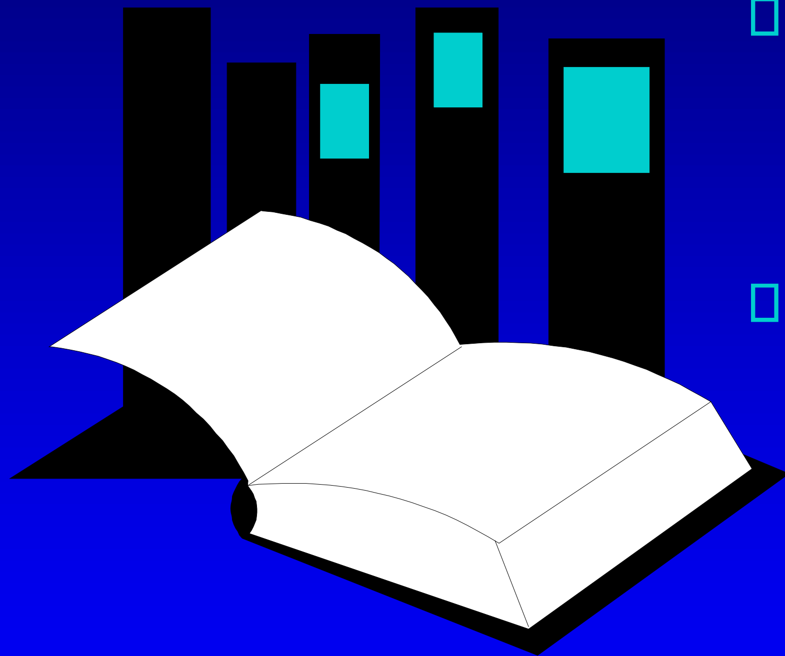
COMPUTER SCIENCE
CITY COLLEGE OF NEW YORK

Lecture 16
Heaps and Priority Queues

Instructor: George Wolberg
Department of Computer Science
City College of New York

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++

Topics

- Heap Definition
- Heap Applications
  - priority queues (chapter 8), sorting (chapter 13)
- Two Heap Operations – add, remove
  - reheapification upward and downward
  - why is a heap good for implementing a priority queue?
- Heap Implementation
  - using binary_tree_node class
  - using fixed size or dynamic arrays

Heaps Definition

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

Heaps

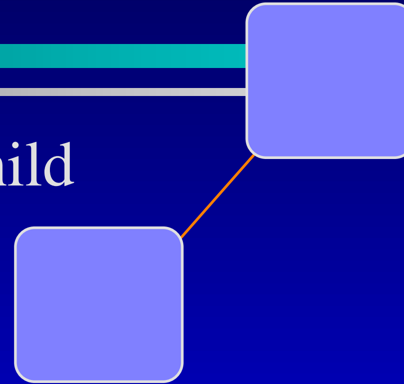
A **heap** is a certain kind of complete binary tree.

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

Heaps

Complete binary tree.

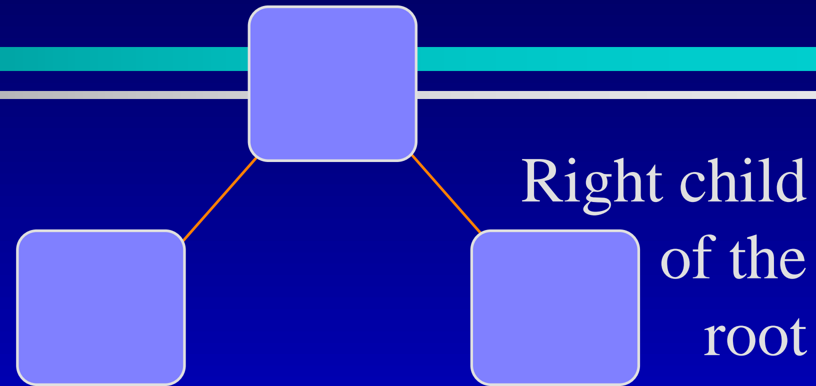
Left child of the root



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

Heaps

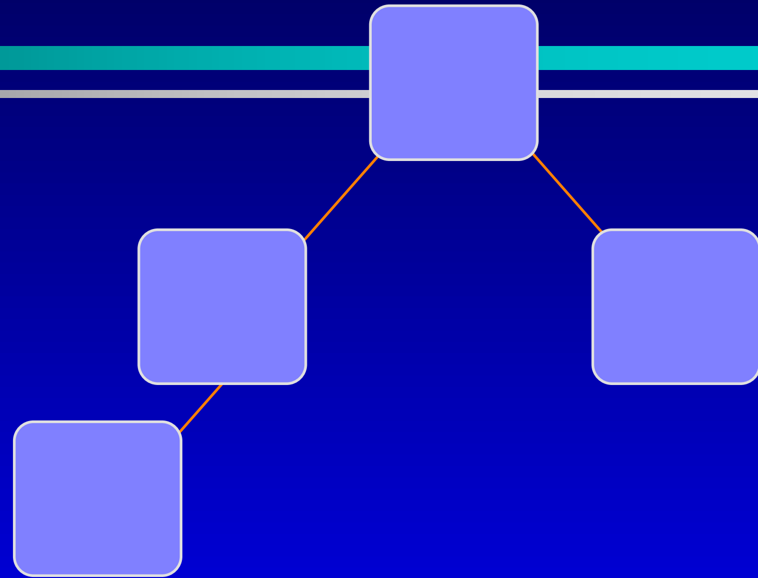
Complete binary tree.



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

Heaps

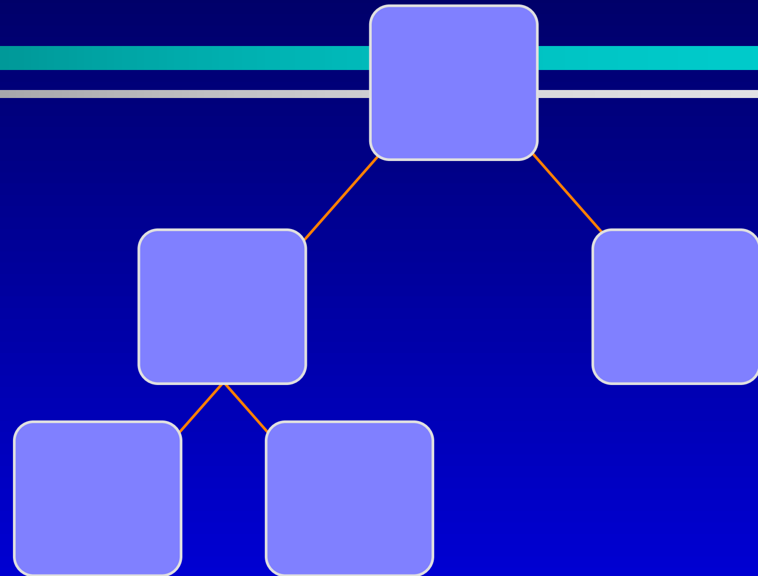
Complete binary tree.



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

Heaps

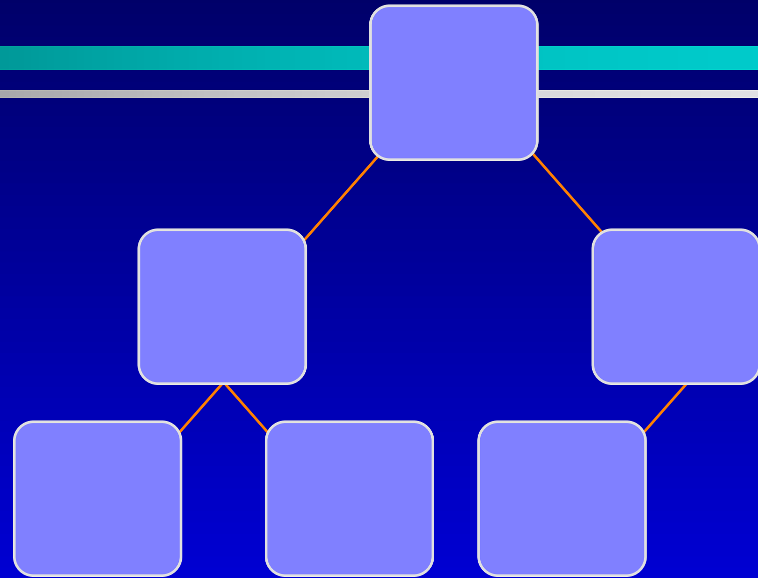
Complete binary tree.



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

Heaps

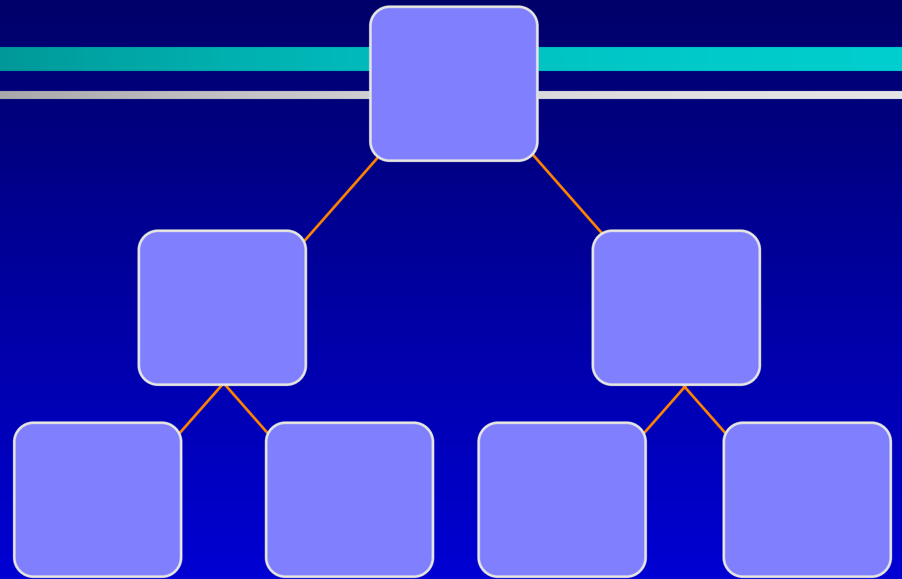
Complete binary tree.



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

Heaps

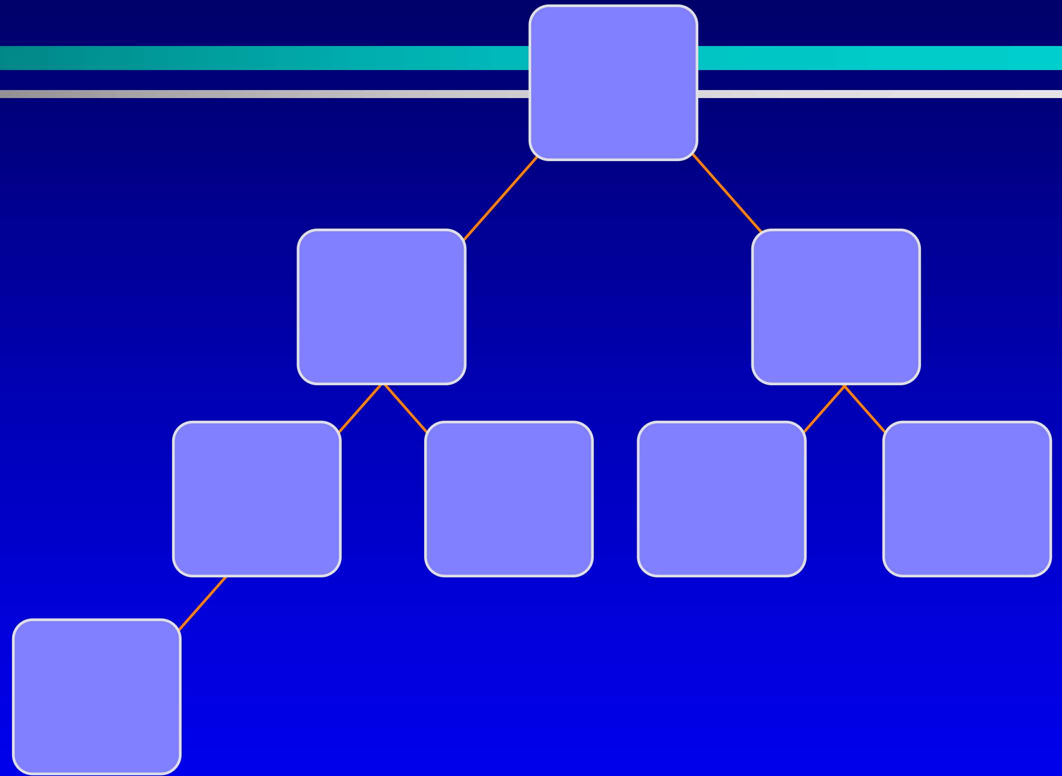
Complete binary tree.



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

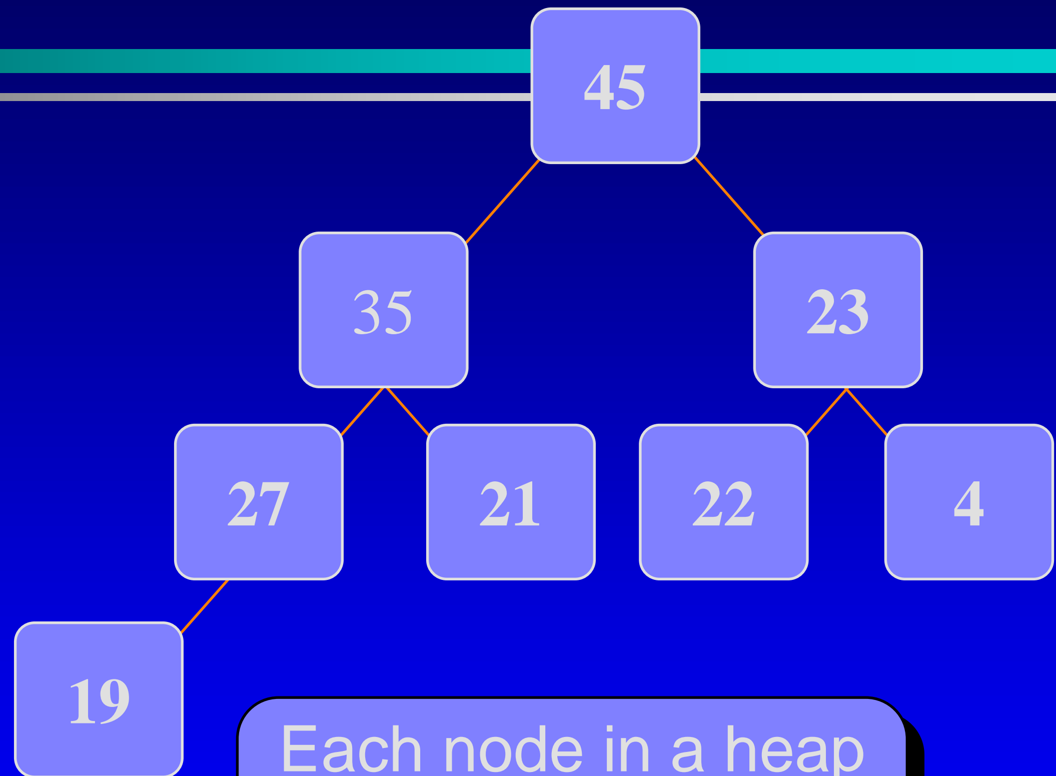
Heaps

Complete binary tree.



Heaps

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



Heaps

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

What it is not: It is not a BST

- In a binary search tree, the entries of the nodes can be compared with a strict weak ordering. Two rules are followed for every node n:
  - The entry in node n is NEVER *less than* an entry in its left subtree
  - The entry in the node n is *less than* every entry in its right subtree.
- BST is not necessarily a complete tree

What it is: Heap Definition

- A heap is a binary tree where the entries of the nodes can be compared with the *less than* operator of a strict weak ordering. In addition, two rules are followed:
  - The entry contained by the node is NEVER *less than* the entries of the node's children
  - The tree is a COMPLETE tree.
- Q: where is the largest entry? → for what....

Application: Priority Queues

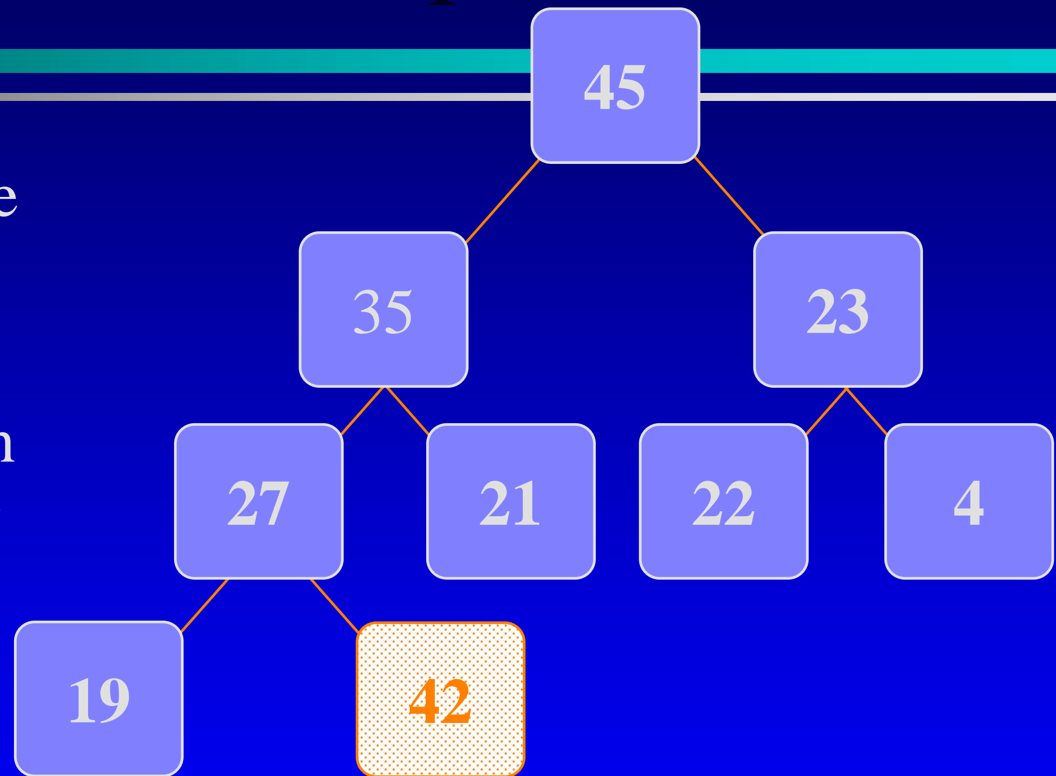
- A priority queue is a container class that allows entries to be retrieved according to some specific priority levels
  - The highest priority entry is removed first
  - If there are several entries with equally high priorities, then the priority queue's implementation determines which will come out first (e.g. FIFO)
- Heap is suitable for a priority queue

The Priority Queue ADT with Heaps

- The entry with the highest priority is always at the root node
- Focus on two priority queue operations
  - adding a new entry
  - remove the entry with the highest priority
- In both cases, we must ensure the tree structure remains to be a heap
  - we are going to work on a conceptual heap without worrying about the precise implementation
  - later I am going to show you how to implement...

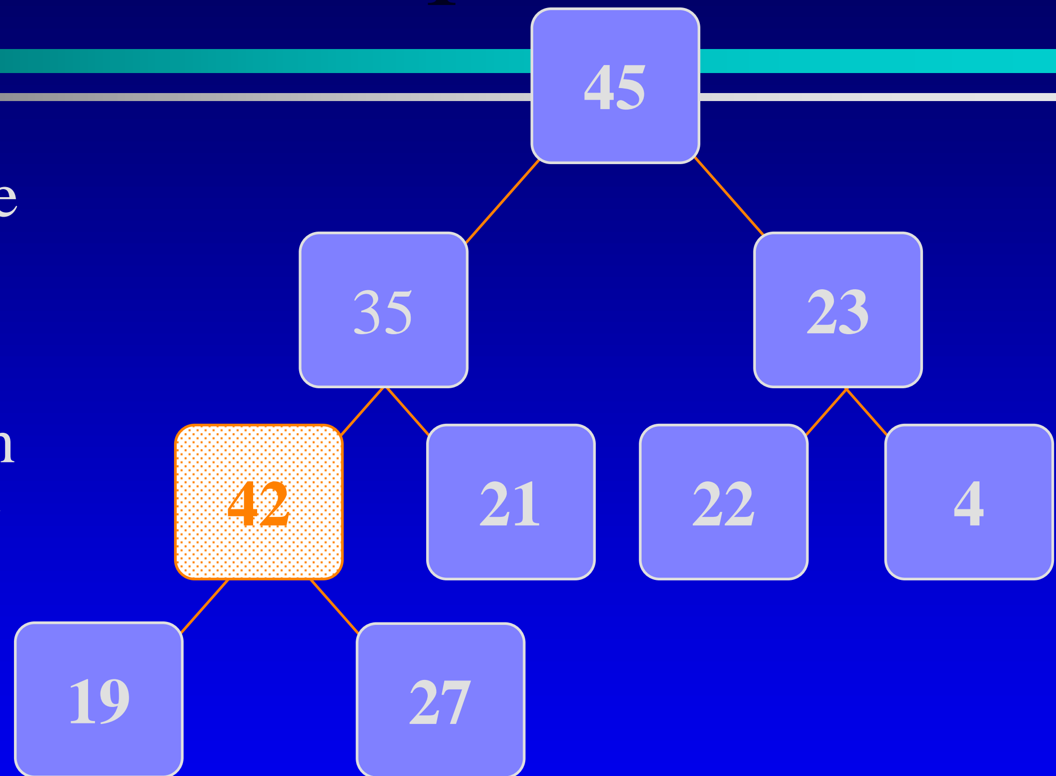
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



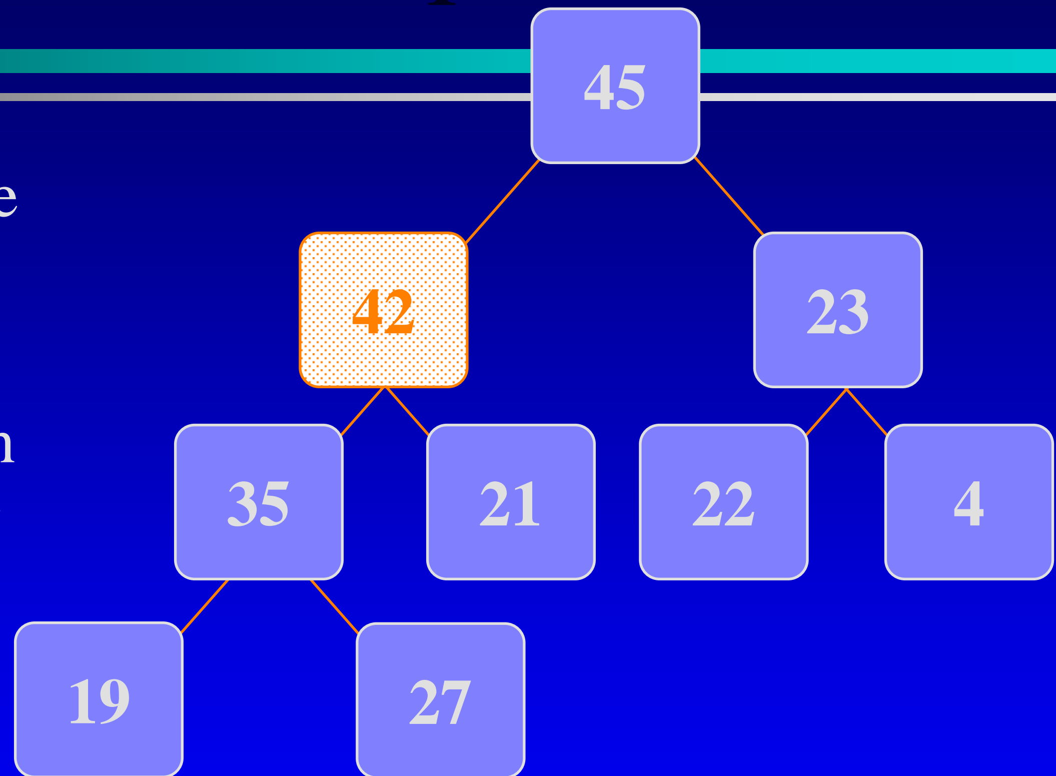
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.

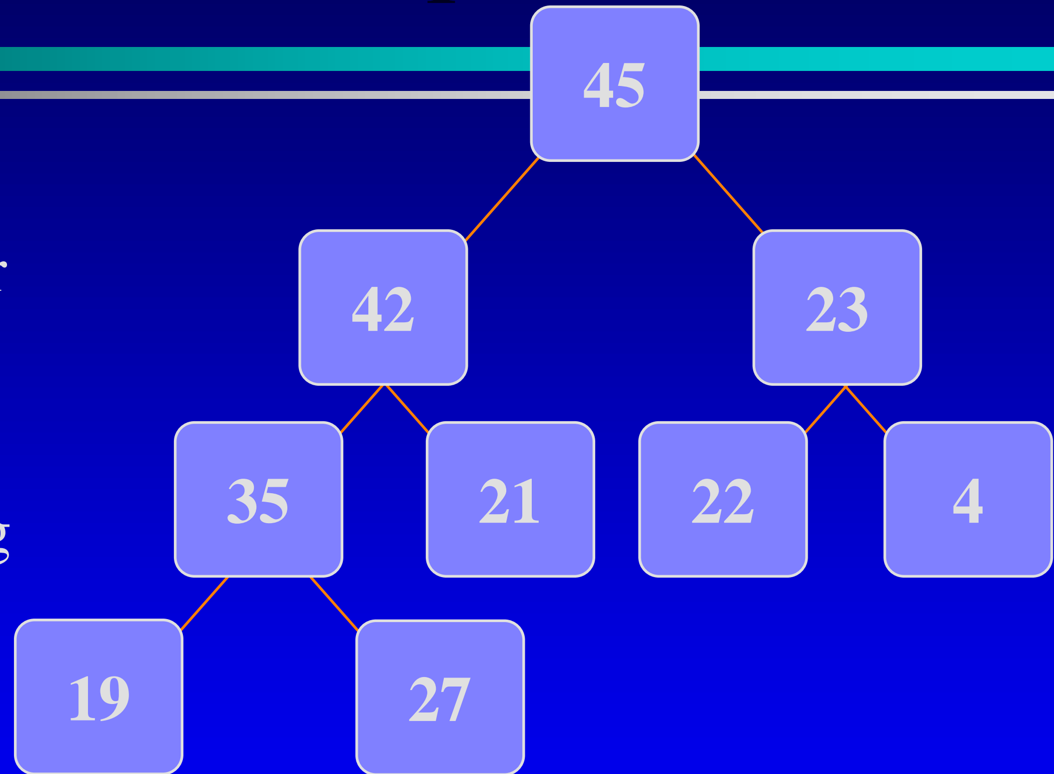


Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Adding a Node to a Heap

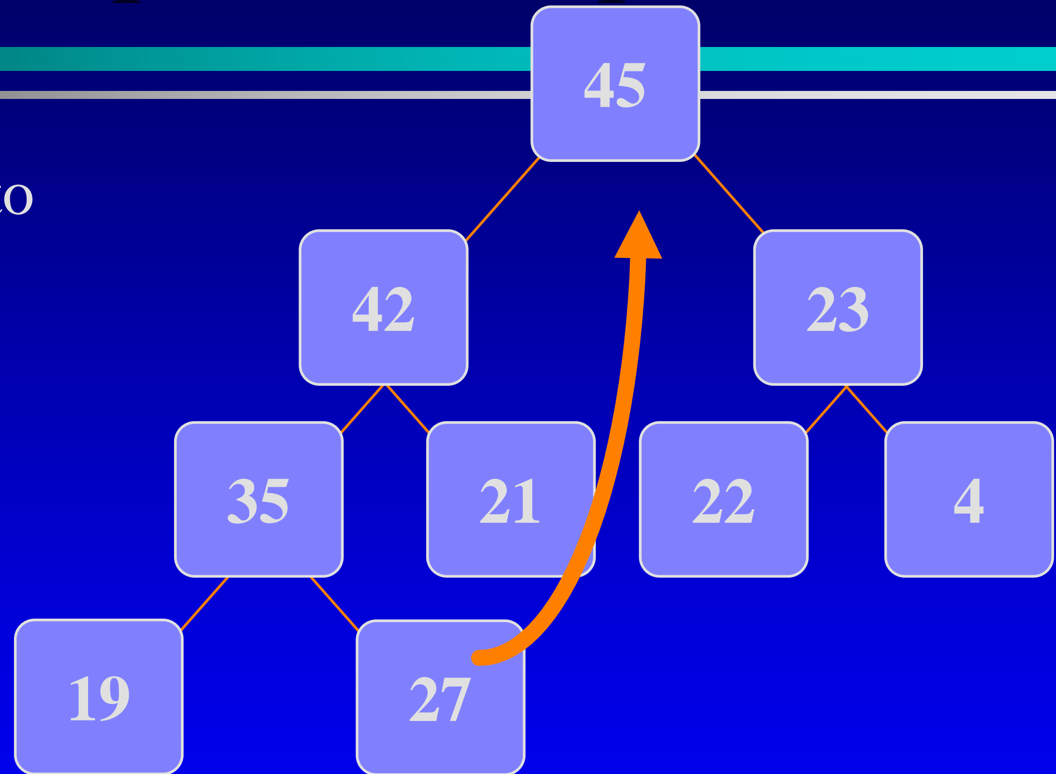


- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.

Note: we need to easily go from child to parent as well as parent to child.

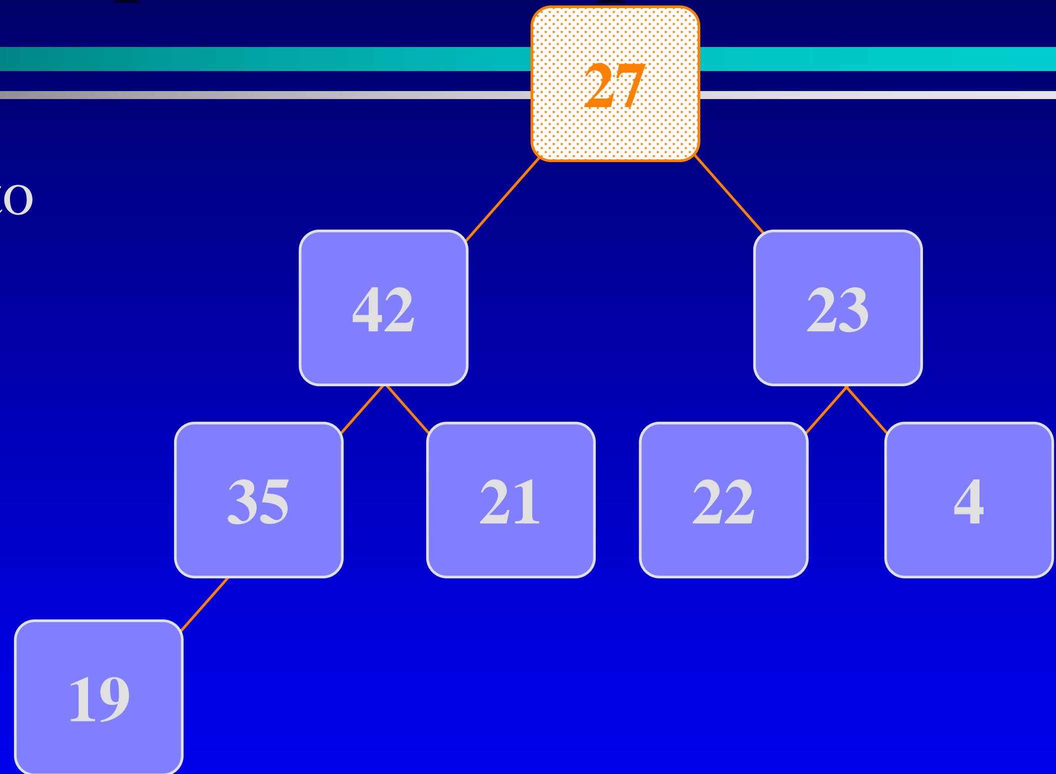
Removing the Top of a Heap

- Move the last node onto the root.



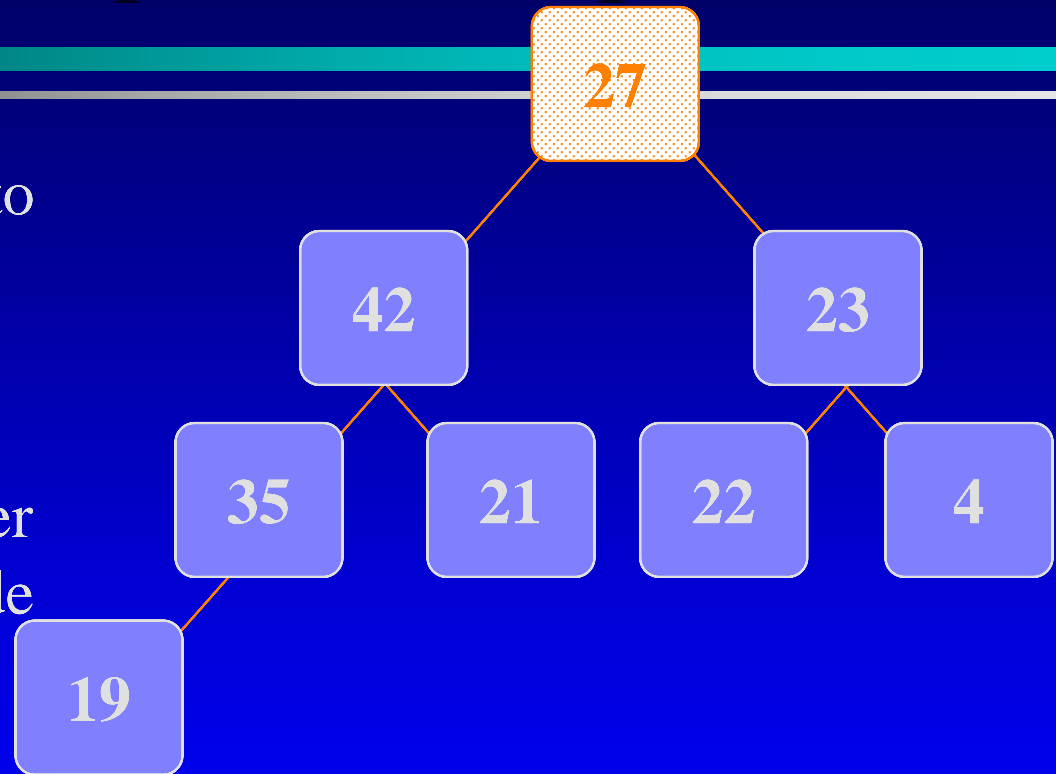
Removing the Top of a Heap

- Move the last node onto the root.



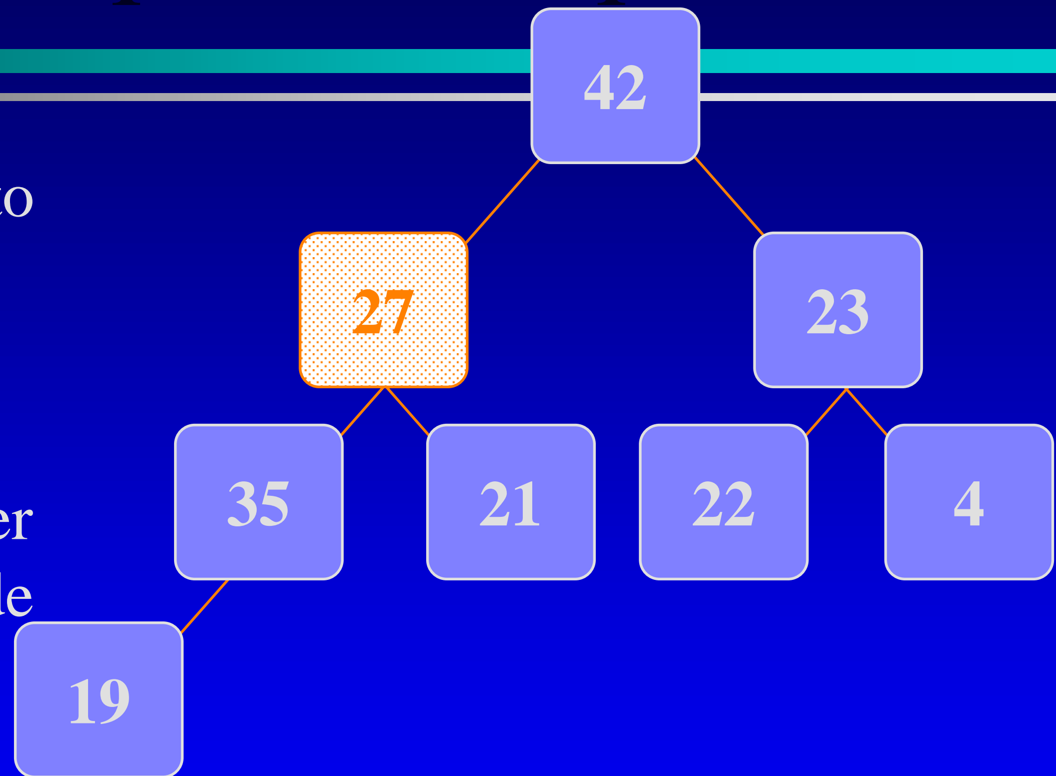
Removing the Top of a Heap

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



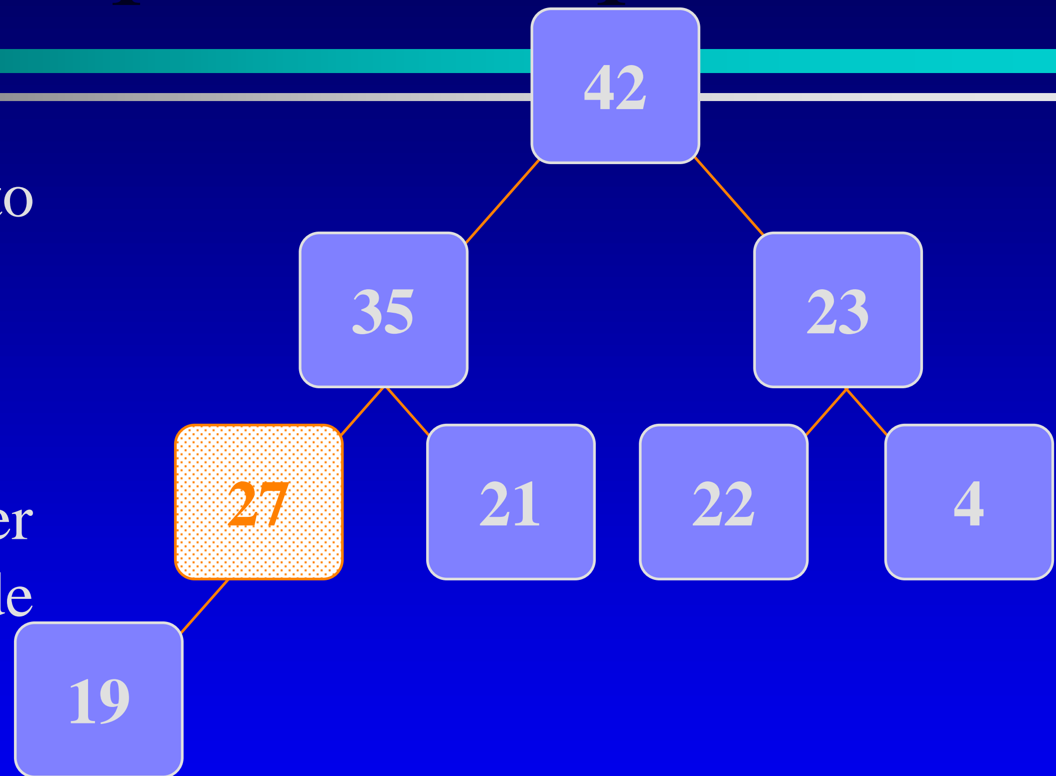
Removing the Top of a Heap

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



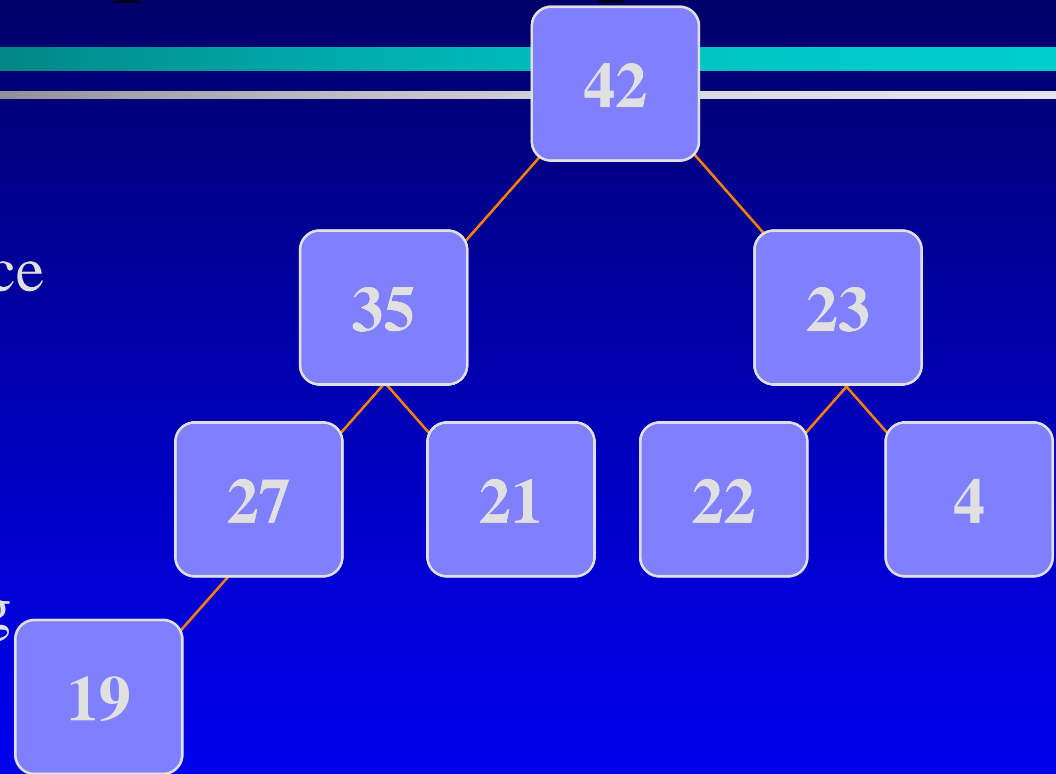
Removing the Top of a Heap

- Move the last node onto the root.
- 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 <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.

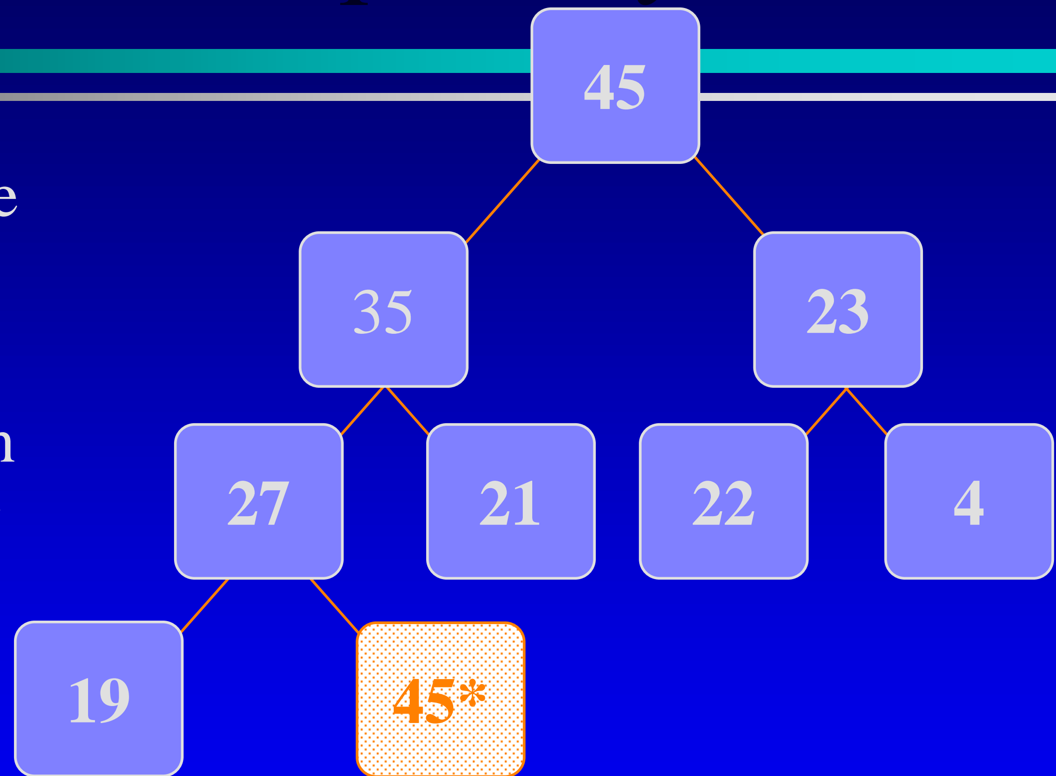


Priority Queues Revisited

- A priority queue is a container class that allows entries to be retrieved according to some specific priority levels
  - The highest priority entry is removed first
  - **If there are several entries with equally high priorities, then the priority queue's implementation determines which will come out first (e.g. FIFO)**
- Heap is suitable for a priority queue

Adding a Node: same priority

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.

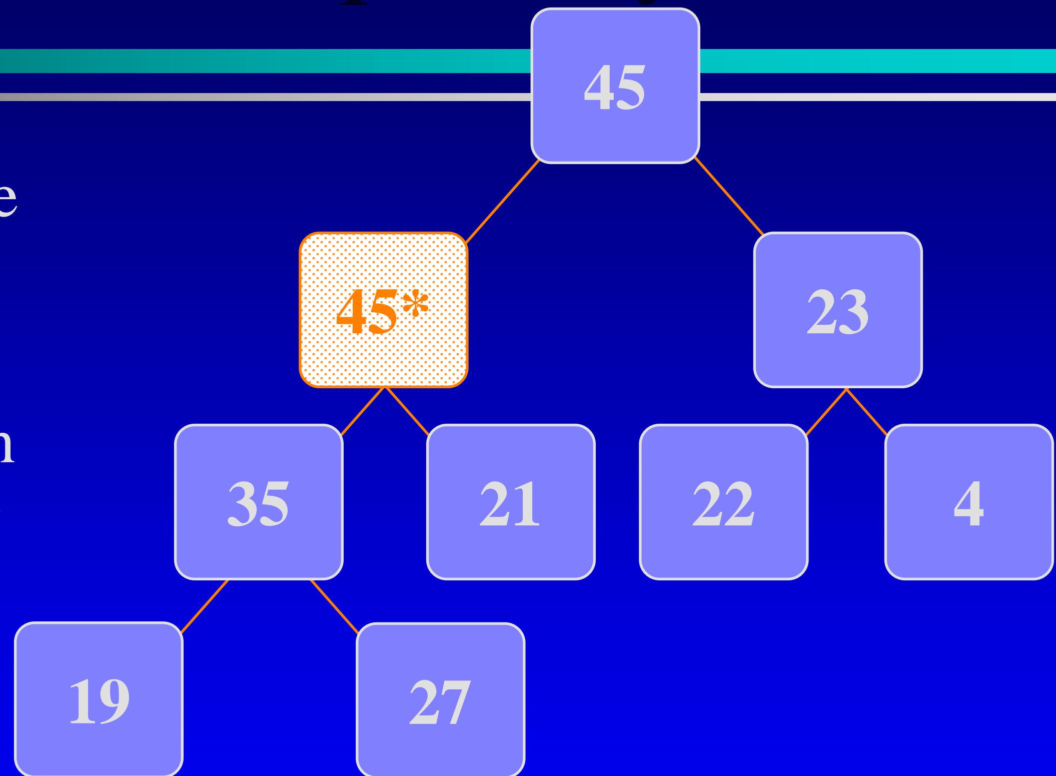


Adding a Node: same priority

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.

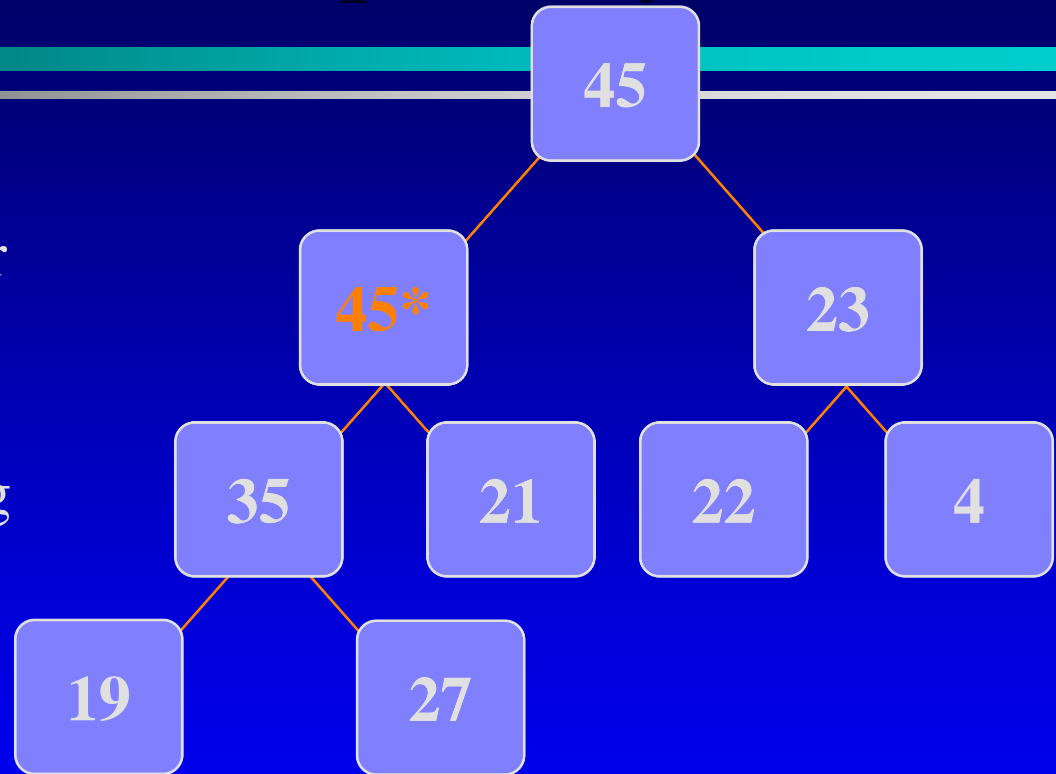
Adding a Node: same priority

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Adding a Node: same priority

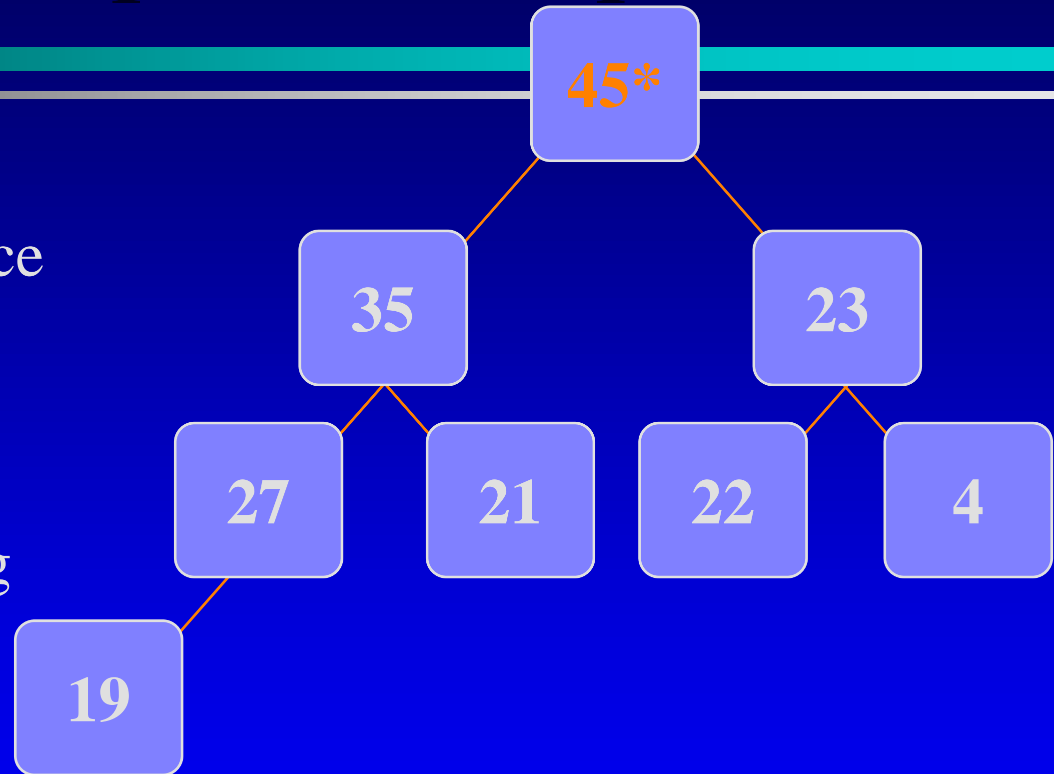
- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



Note: Implementation determines which 45 will be in the root, and will come out first when popping.

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**.



Note: Implementation determines which 45 will be in the root, and will come out first when popping.

Heap Implementation

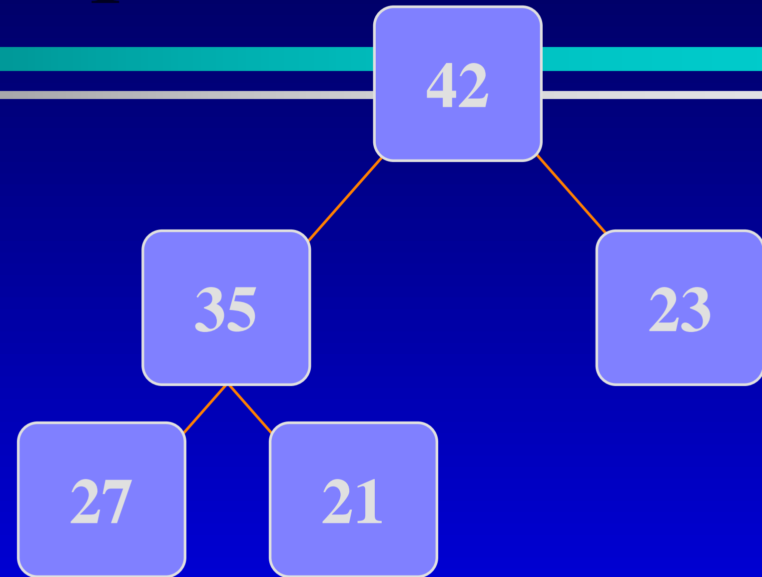
- Use binary_tree_node class
  - node implementation is for a general binary tree
  - but we may need to have doubly linked node
- Use arrays (page 475)
  - A heap is a complete binary tree
  - which can be implemented more easily with an array than with the node class
  - and do two-way links

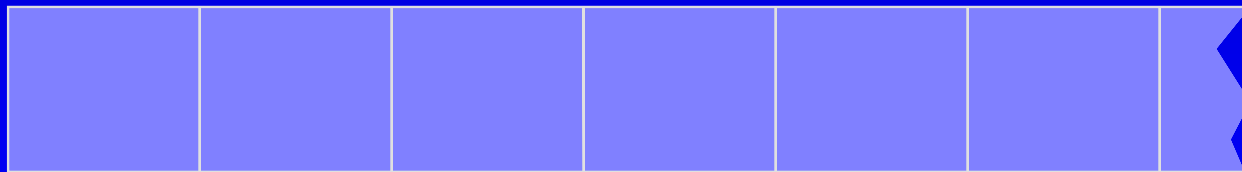
Formulas for location children and parents in an array representation

- Root at location [0]
- Parent of the node in [i] is at [(i-1)/2]
- Children of the node in [i] (if exist) is at [2i+1] and [2i+2]
- Test:
  - complete tree of 10, 000 nodes
  - parent of 4999 is at (4999-1)/2 = 2499
  - children of 4999 is at 9999 (V) and 10,000 (X)

Implementing a Heap

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

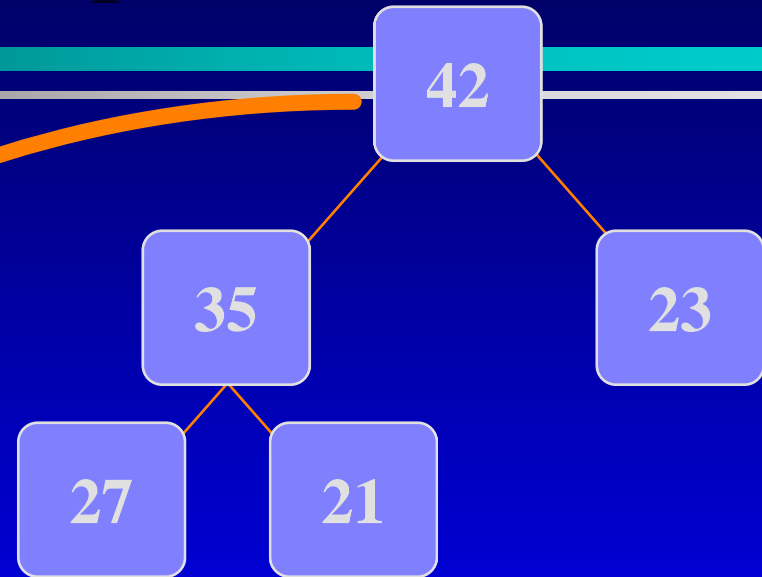


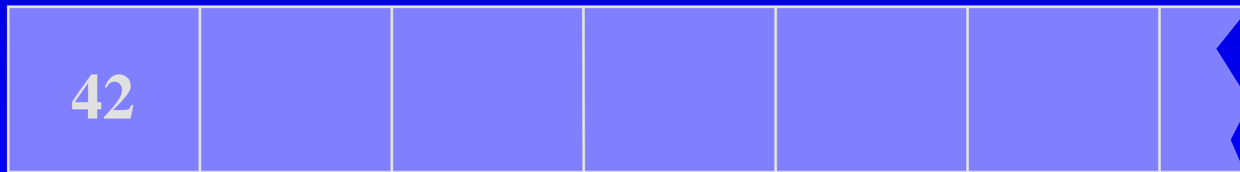


An array of data

Implementing a Heap

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

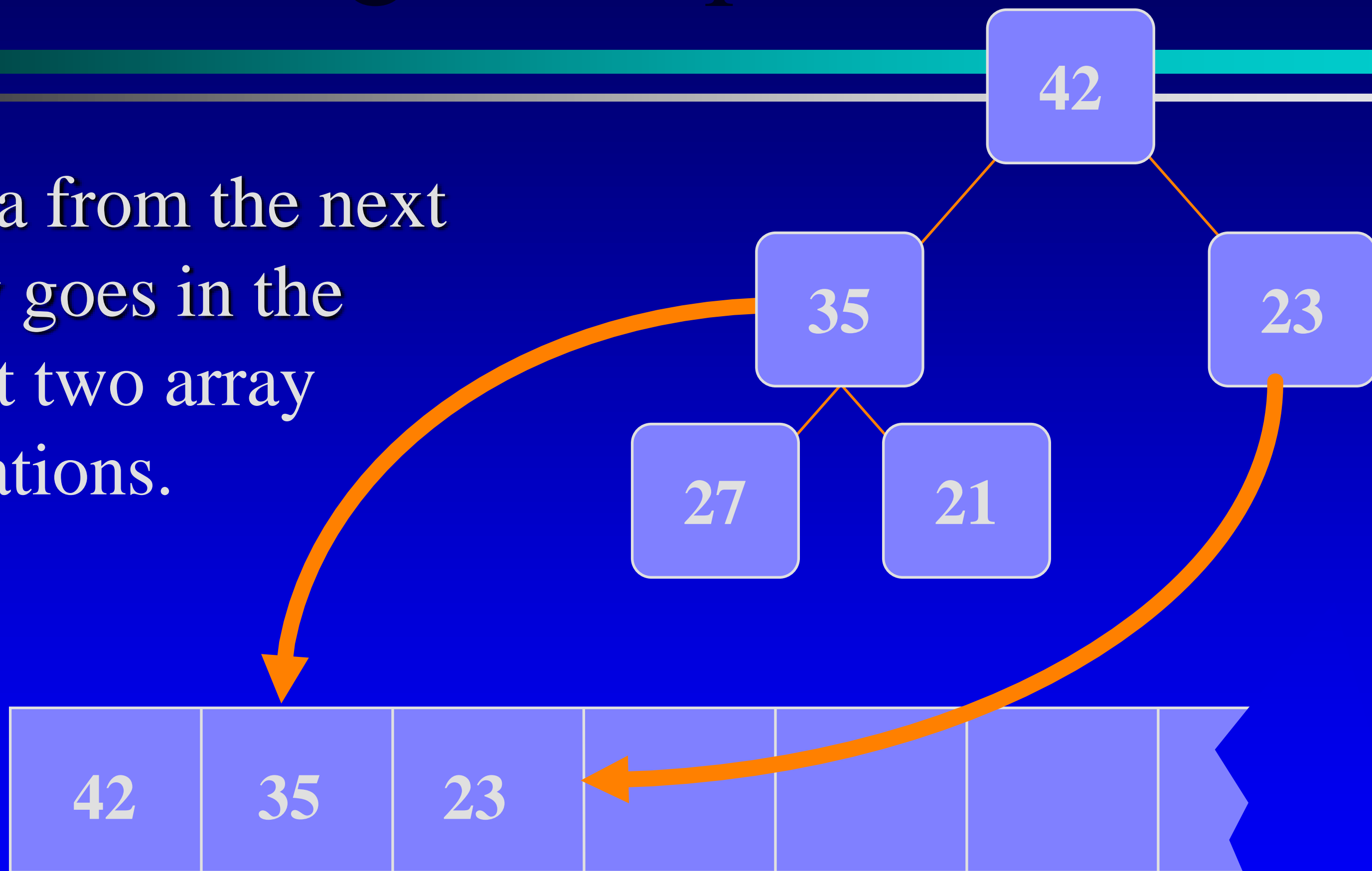




An array of data

Implementing a Heap

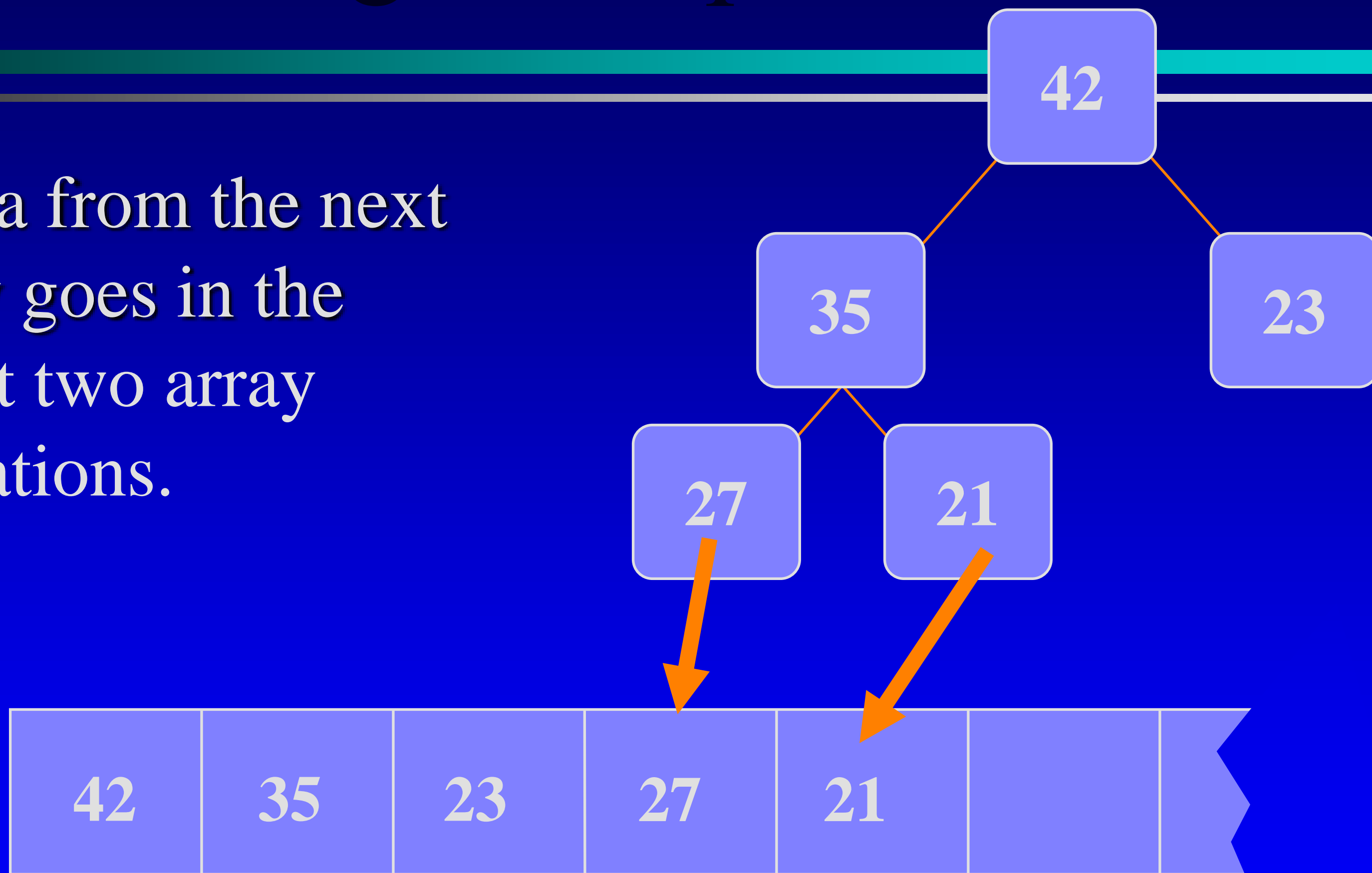
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

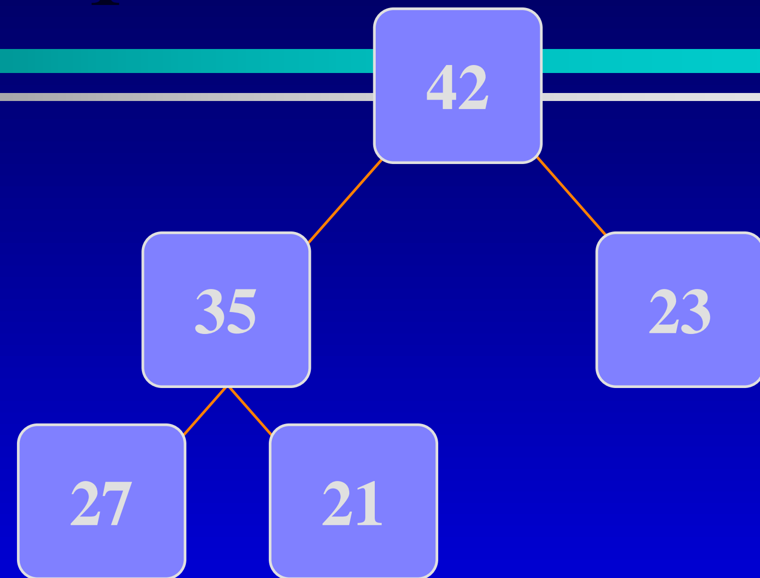
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

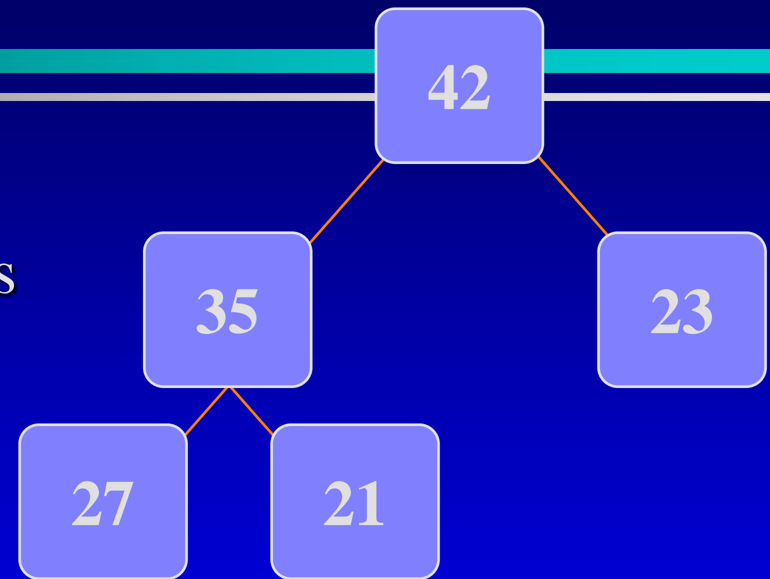
- Data from the next row goes in the next two array locations.



An array of data

We don't care what's in this part of the array.

Important Points about the Implementation

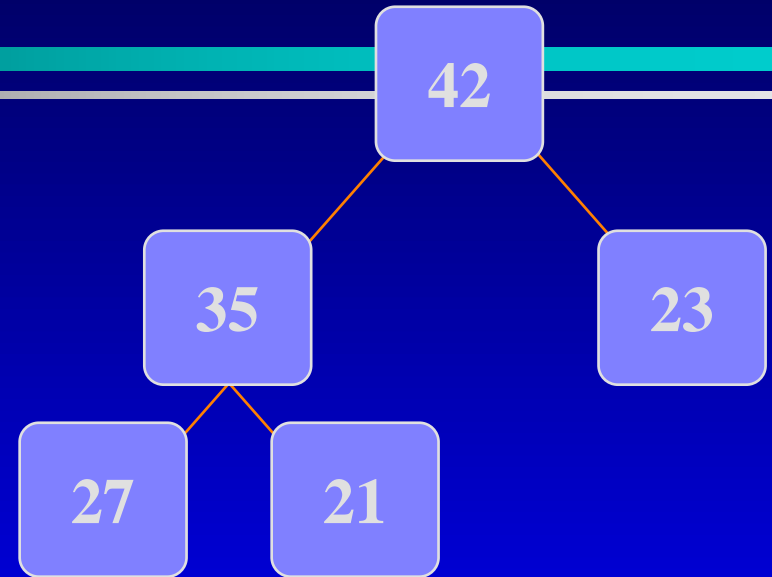


- The links between the tree's nodes are **not** actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

An array of data

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.



Formulas for location children and parents in an array representation

- Root at location [0]
- Parent of the node in [i] is at [(i-1)/2]
- Children of the node in [i] (if exist) is at [2i+1] and [2i+2]
- Test:
  - complete tree of 10, 000 nodes
  - parent of 4999 is at (4999-1)/2 = 2499
  - children of 4999 is at 9999 (V) and 10,000 (X)

Wrap Up...

```
class heap
{
 public:
        ....
        void push(const Item& entry); // add
        Item& pop(); // remove the highest
 private:
        Item data[CAPACITY];
        size_type used;
}
```

- Can you implement t[illegible] with the knowledge c[illegible]
- Add
  - put the new entry in the last location
  - Push the new node upward, swapping with its parent until the new node reaches an acceptable location
- Remove
  - move the last node to the root
  - Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location

Wrap Up...

- Can you imple
  with the know
- Add
  - put the new entry i
  - Push the new node
    node reaches an ac
- Remove
  - move the last node to the root
  - Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location

```
template <class Item>
class heap
{
 public:
        heap () { used = 0;}
        void push(const Item& entry); // add
        Item& pop(); // remove the highest
        size_t parent (size_t k) const { return (k-1)/2;}
        size_t l_child (size_t k) const { return 2*k+1;}
        size_t r_child (size_t k) const { return 2*k+2;}
 private:
        Item data[CAPACITY];
        size_type used;
}
```

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.

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 Graphics Inc, One Mile Up Inc, TechPool Studios, Totem Graphics Inc).

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.