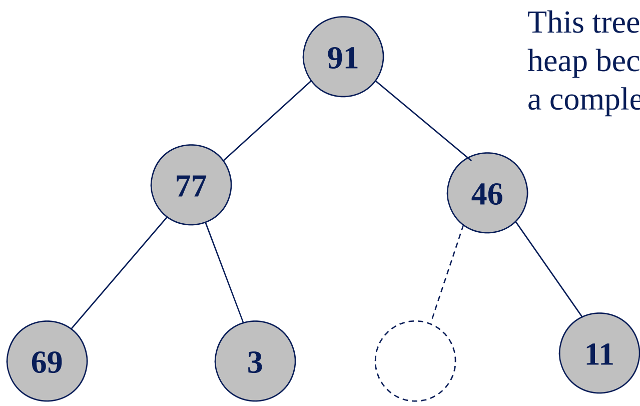
Heaps

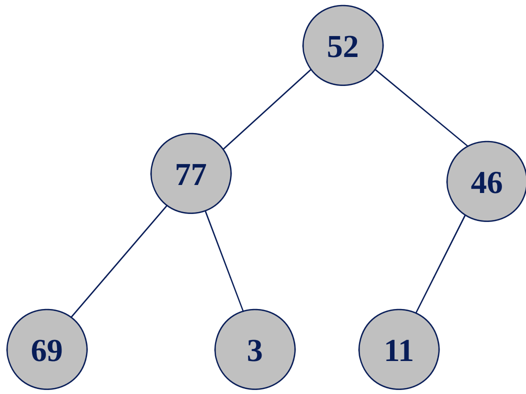
- Like BSTs, ***heap***s are special binary trees
  - They have special properties that can be exploited for performance improvements in certain applications
- (Corresponding to the 2 rules we've seen that make BSTs special are) ***two storage rules*** that make heaps special:
  - The item contained in a node is ***greater than or equal to*** the items of the node's children
    - A parent's item is never less than the item of any of its children
    - (enforcement of this requires that the node's items can be compared with the usual comparison operators that form a ***strict weak ordering***)
  - The tree is a ***complete binary tree***
    - Every level except the deepest must have as many nodes as possible
    - At the deepest level, all nodes are as far left as possible

Heaps



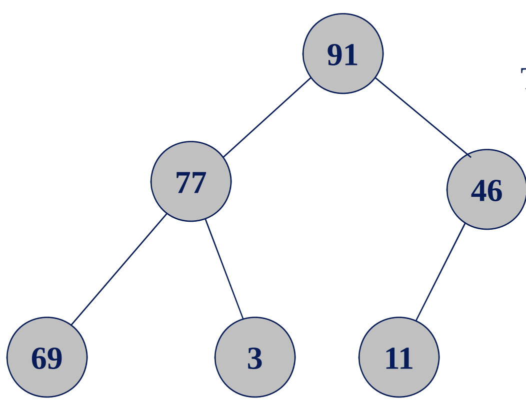
This tree is NOT a heap because it is not a complete binary tree

Heaps



This tree is NOT a heap because 77 is greater than 52

Heaps



This tree is a heap

Heaps — Implementation Using Array

- A heap is easily implemented with an array, since it is a complete binary tree
  - As we saw previously, a complete binary tree can be stored in an array with the root node in position 0 and all the children can be located mathematically
- If the size of the heap is uncertain, dynamic arrays can be used that grow and shrink as needed

Heaps — Implementation for Priority Queue

- A regular queue is a FIFO (first-in, first-out) data structure
- A priority queue is a queue in which each item has a priority assigned to it
  - Items with higher priority can cut in line
- Heaps offer an efficient implementation for priority queues

Heaps — Implementation for Priority Queue

- Each node of a heap contains one entry along with the entry's priority
- The tree is maintained so that it follows the heap storage rules
  - The entry contained by a node has a priority greater than or equal to the priorities of the entries of the node's children
  - The tree is a complete binary tree

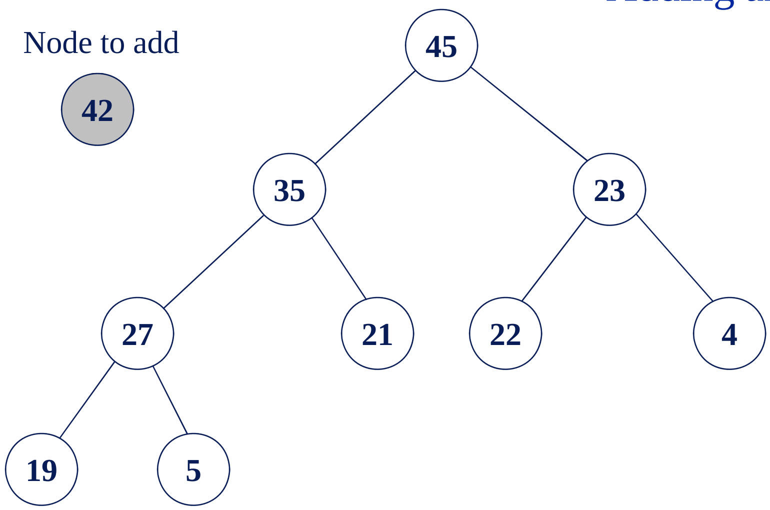
Heaps — Two Key Operations

- Two key operations for a heap are..
  - Adding a new entry
    - When applied to priority queue → *enqueue* an entry
  - Removing the entry with the highest priority
    - When applied to priority queue → *dequeue* an entry
- Both operations must ensure that the structure remains a heap when the operation concludes

Heaps

First of Two Key Operations
Adding an Entry

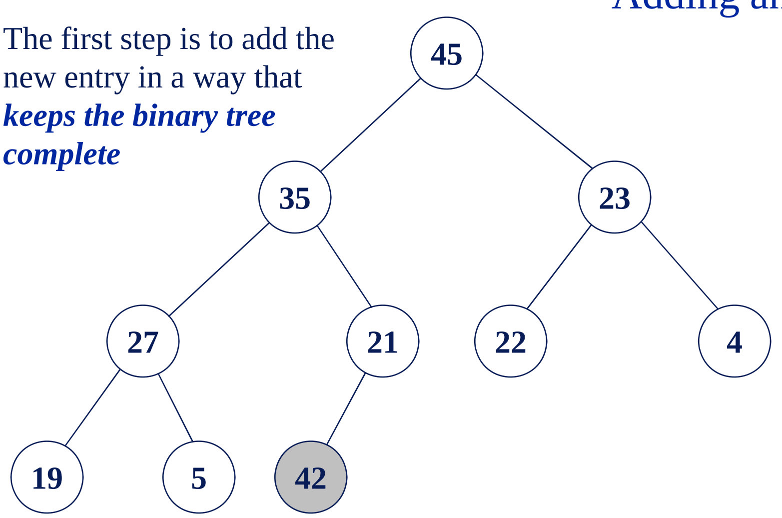
Node to add



Heaps

First of Two Key Operations
Adding an Entry

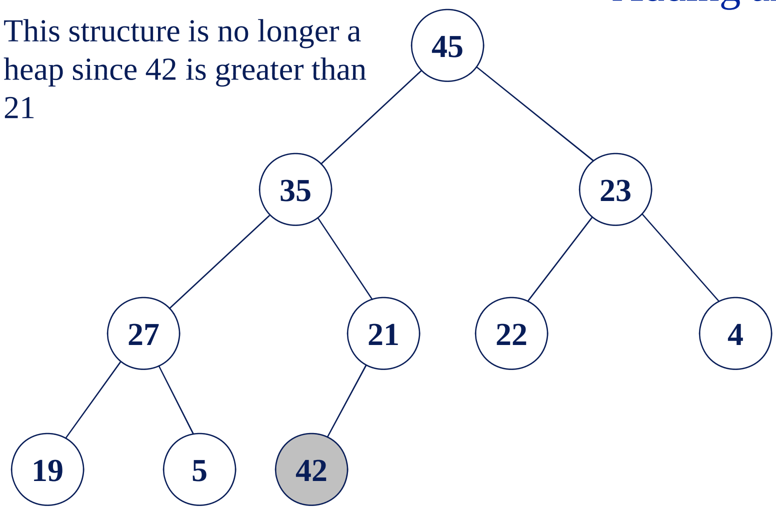
The first step is to add the new entry in a way that ***keeps the binary tree complete***



Heaps

First of Two Key Operations
Adding an Entry

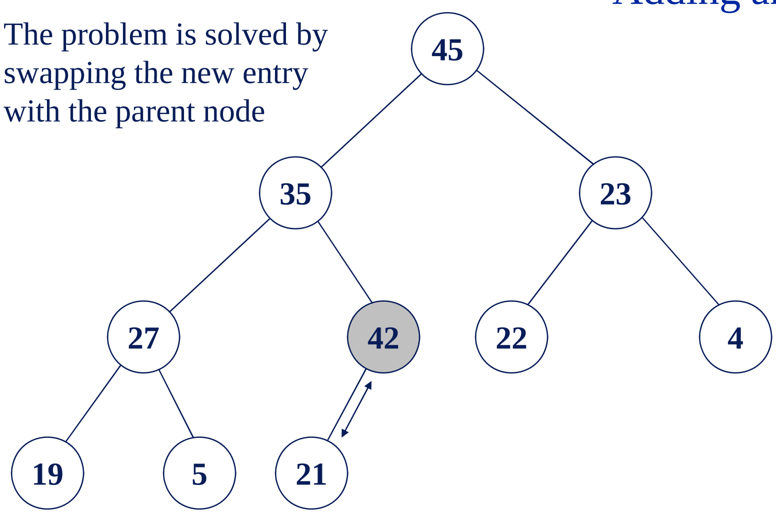
This structure is no longer a heap since 42 is greater than 21



Heaps

First of Two Key Operations
Adding an Entry

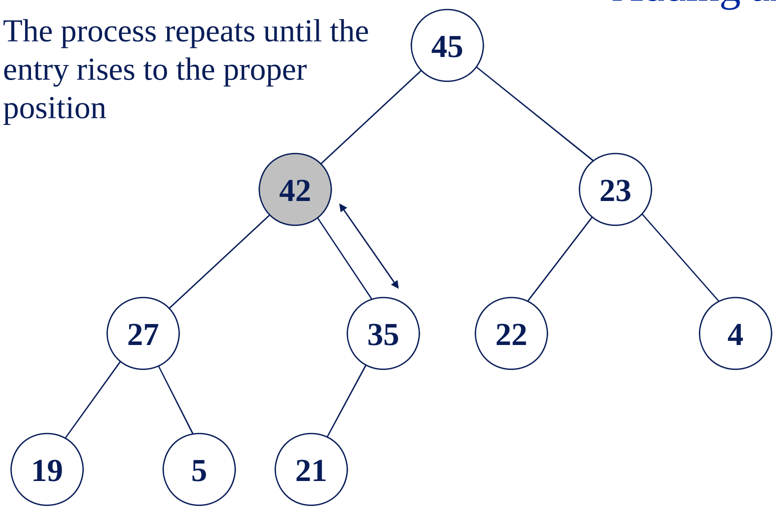
The problem is solved by swapping the new entry with the parent node



Heaps

First of Two Key Operations Adding an Entry

The process repeats until the entry rises to the proper position



Heaps

First of Two Key Operations Adding an Entry

Pseudocode

- Place the new entry in the heap in the first available location
  - This keeps the structure as a ***complete binary tree***, but it ***may no longer be a heap***
- While the new entry has a priority that is higher than its parent, swap the new entry with the parent
  - This is called ***reheapification upward***

Heaps

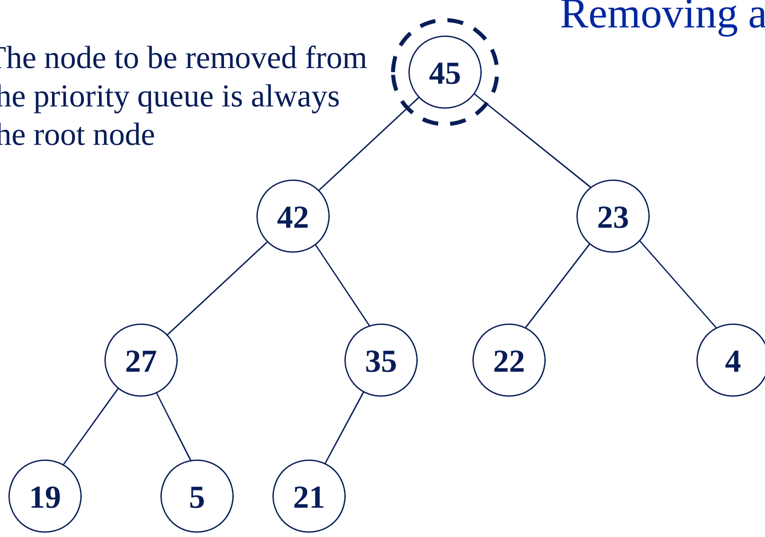
Second of Two Key Operations Removing an Entry

- Since the heap represents a priority queue, the entry with the highest priority must always be removed first
- Since the heap is built by comparing priorities, the ***highest priority item is always on top***
  - It should also be the first item of that priority to enter the queue

Heaps

Second of Two Key Operations Removing an Entry

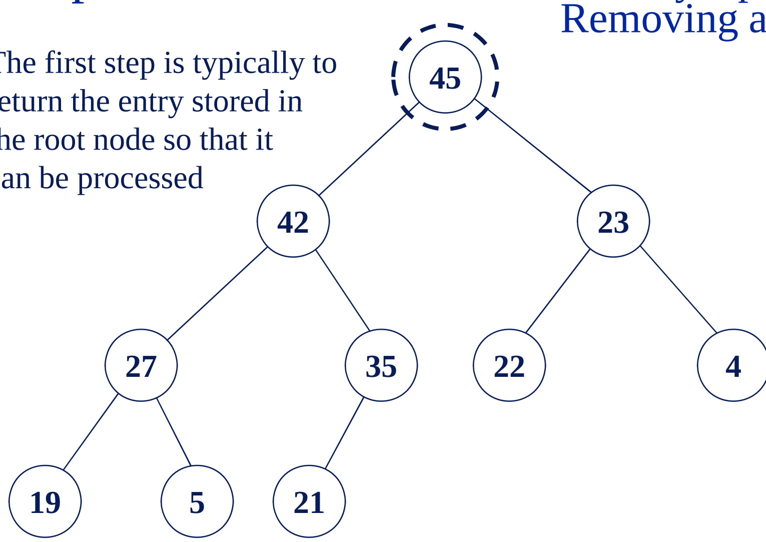
The node to be removed from the priority queue is always the root node



Heaps

Second of Two Key Operations
Removing an Entry

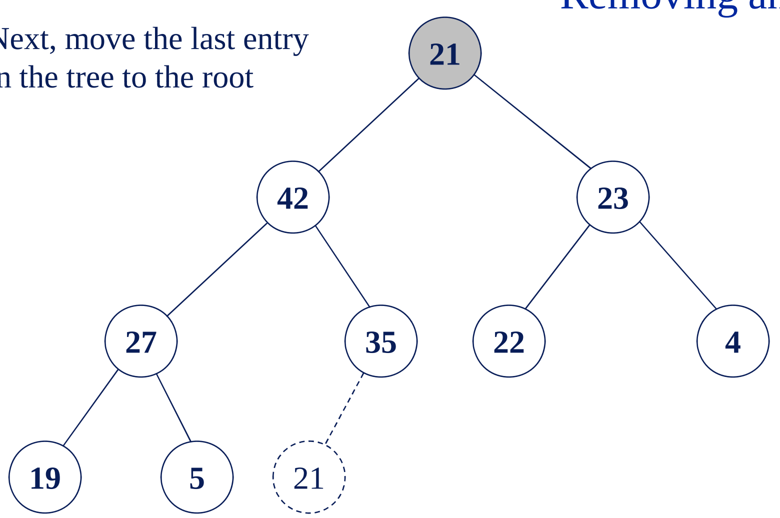
The first step is typically to return the entry stored in the root node so that it can be processed



Heaps

Second of Two Key Operations
Removing an Entry

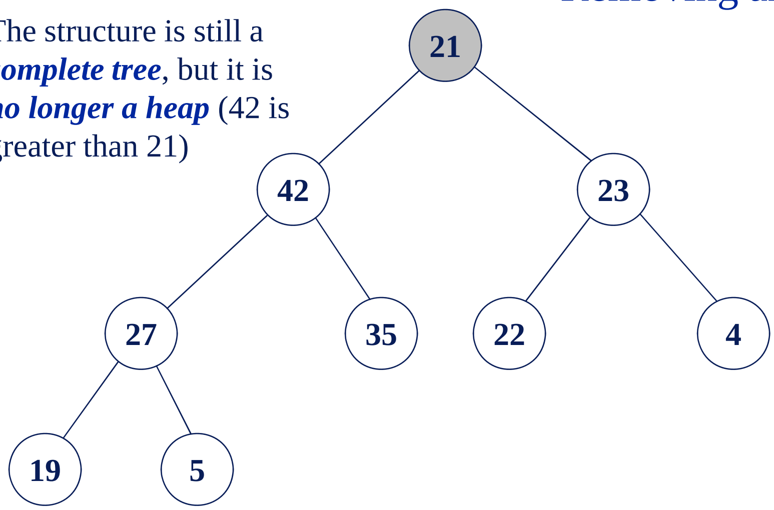
Next, move the last entry in the tree to the root



Heaps

Second of Two Key Operations
Removing an Entry

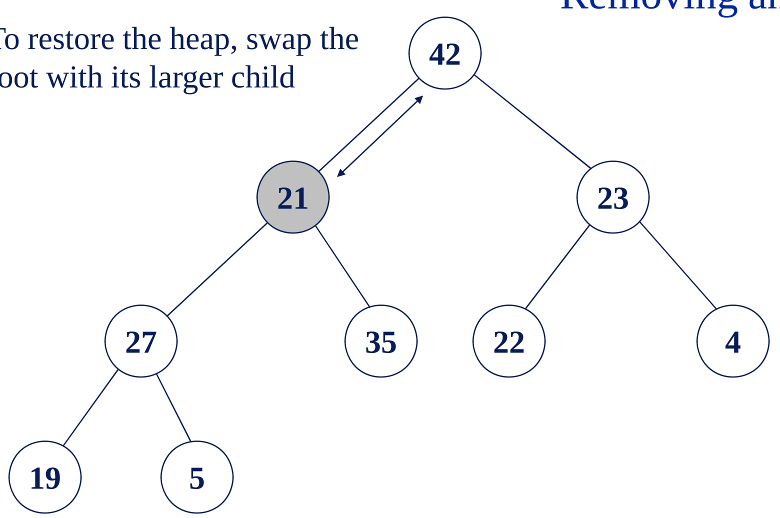
The structure is still a ***complete tree***, but it is ***no longer a heap*** (42 is greater than 21)



Heaps

Second of Two Key Operations
Removing an Entry

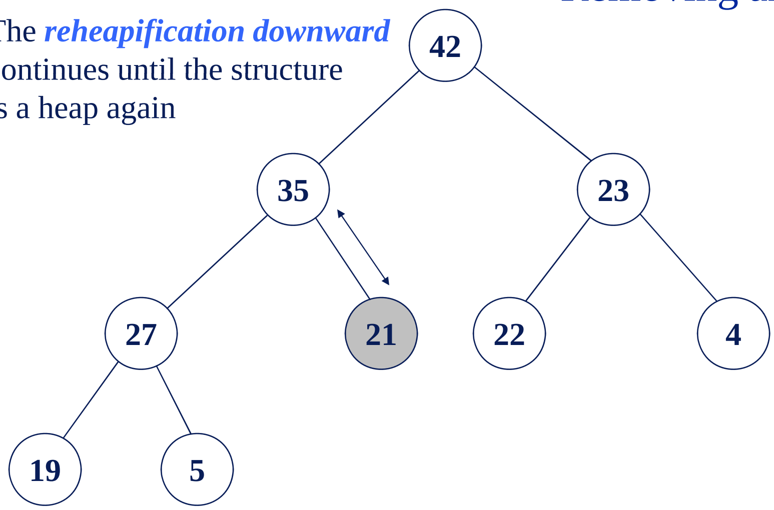
To restore the heap, swap the root with its larger child



Heaps

Second of Two Key Operations
Removing an Entry

The ***reheapification downward*** continues until the structure is a heap again



Heaps

Second of Two Key Operations
Removing an Entry

Pseudocode

- Copy the entry at the root of the heap to a variable that is used to save the return value
- Copy the last entry in the deepest level to the root and take the node (that has just been copied) out of the tree
- While the out-of-place entry has a priority lower than one of its children, swap the out-of-place entry with its highest-priority child
- Return the value that was saved in the first step

Heaps

2 Ways to Build a Heap from a Given Set of (Node) Values

- First ("brute-force") way:
  - Begin with an empty heap
  - Keep inserting new nodes (one node at a time) until all given values have been inserted
- Second (more elegant/efficient) way:
  - (to be discussed/illustrated in class)
  - (also described in a latter lecture note that discusses *heap sort*)

Textbook Readings

- Chapter 11
  - Section 11.1