9.1 Priority queue abstract data type (ADT)

Priority queue abstract data type

A ***priority queue*** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority. The priority queue ***enqueue*** operation inserts an item such that the item is closer to the front than all items of lower priority, and closer to the end than all items of equal or higher priority. The priority queue ***dequeue*** operation removes and returns the item at the front of the queue, which has the highest priority.

PARTICIPATION ACTIVITY 9.1.1: Priority queue enqueue and dequeue.

Animation content:

```
undefined
```

Animation captions:

1. Enqueueing a single item with priority 7 initializes the priority queue with 1 item.
2. If a lower numerical value indicates higher priority, enqueueing 11 adds the item to the end of the queue.
3. Since 5 < 7, enqueueing 5 puts the item at the priority queue's front.
4. When enqueueing items of equal priority, the first-in-first-out rules apply. The 2nd item with priority 7 comes after the first.
5. Dequeue removes from the front of the queue, which is always the highest priority item.

PARTICIPATION ACTIVITY 9.1.2: Priority queue enqueue and dequeue.

Assume that lower numbers have higher priority and that a priority queue currently holds items: 54, 71, 86 (front is 54).

1) Where would an item with priority 60 reside after being enqueued?

- ○ Before 54
- ○ After 54
- ○ After 86

2) Where would an additional item with priority 54 reside after being enqueued?

- Before the first 54
- After the first 54
- After 86

3) The dequeue operation would return which item?

- 54
- 71
- 86

Common priority queue operations

In addition to enqueue and dequeue, a priority queue usually supports peeking and length querying. A ***peek*** operation returns the highest priority item, without removing the item from the front of the queue.

Table 9.1.1: Common priority queue ADT operations.

Operation	Description	Example starting with priority queue: 42, 61, 98 (front is 42)
Enqueue(PQueue, x)	Inserts x after all equal or higher priority items	Enqueue(PQueue, 87). PQueue: 42, 61, 87, 98
Dequeue(PQueue)	Returns and removes the item at the front of PQueue	Dequeue(PQueue) returns 42. PQueue: 61, 98
Peek(PQueue)	Returns but does not remove the item at the front of PQueue	Peek(PQueue) returns 42. PQueue: 42, 61, 98
IsEmpty(PQueue)	Returns true if PQueue has no items	IsEmpty(PQueue) returns false.
GetLength(PQueue)	Returns the number of items in PQueue	GetLength(PQueue) returns 3.

PARTICIPATION ACTIVITY 9.1.3: Common priority queue ADT operations.

Assume servicePQueue is a priority queue with contents: 11, 22, 33, 44, 55.

1) What does GetLength(servicePQueue) return?
   - 5
   - 11
   - 55

2) What does Dequeue(servicePQueue) return?
   - 5
   - 11
   - 55

3) After dequeuing an item, what will Peek(servicePQueue) return?

- 11
- 22
- 33

4) After calling Dequeue(servicePQueue) a total of 5 times, what will GetLength(servicePQueue) return?

- -1
- 0
- Undefined

Enqueueing items with priority

A priority queue can be implemented such that each item's priority can be determined from the item itself. Ex: A customer object may contain information about a customer, including the customer's name and a service priority number. In this case, the priority resides within the object.

A priority queue may also be implemented such that all priorities are specified during a call to ***EnqueueWithPriority***: An enqueue operation that includes an argument for the enqueued item's priority.

PARTICIPATION ACTIVITY | 9.1.4: Priority queue EnqueueWithPriority operation.

Animation content:

`undefined`

Animation captions:

1. Calls to EnqueueWithPriority() enqueue objects A, B, and C into the priority queue with the specified priorities.
2. In this implementation, the objects enqueued into the queue do not have data members representing priority.
3. Priorities specified during each EnqueueWithPriority() call are stored alongside the queue's objects.

PARTICIPATION ACTIVITY | 9.1.5: EnqueueWithPriority operation.

1) A priority queue implementation that

requires objects to have a data member storing priority would implement the _____ function.

- Enqueue
- EnqueueWithPriority

2) A priority queue implementation that does not require objects to have a data member storing priority would implement the _____ function.

- Enqueue
- EnqueueWithPriority

Implementing priority queues with heaps

A priority queue is commonly implemented using a heap. A heap will keep the highest priority item in the root node and allow access in O(1) time. Adding and removing items from the queue will operate in worst-case O($logN$) time.

Table 9.1.2: Implementing priority queues with heaps.

Priority queue operation	Heap functionality used to implement operation	Worst-case runtime complexity
Enqueue	Insert	O($logN$)
Dequeue	Remove	O($logN$)
Peek	Return value in root node	O(1)
IsEmpty	Return true if no nodes in heap, false otherwise	O(1)
GetLength	Return number of nodes (expected to be stored in the heap's member data)	O(1)

PARTICIPATION ACTIVITY 9.1.6: Implementing priority queues with heaps.

1) The Dequeue and Peek operations

both return the value in the root, and therefore have the same worst-case runtime complexity.

- True
- False

2) When implementing a priority queue with a heap, no operation will have a runtime complexity worse than O($logN$).

- True
- False

3) If items in a priority queue with a lower numerical value have higher priority, then a max-heap should be used to implement the priority queue.

- True
- False

4) A priority queue is always implemented using a heap.

- True
- False

CHALLENGE ACTIVITY 9.1.1: Priority queue abstract data type.

422352.2723990.qx3zqy7

Start

Assume that lower numbers have higher priority and that a priority queue numPQueue cur holds items: 10, 29, 38, 73 (front is 10).

Where does Enqueue(numPQueue, 4) add an item?

After 10

Where does Enqueue(numPQueue, 10) add an item?

After 10

1 2 3 4 5

Check Next

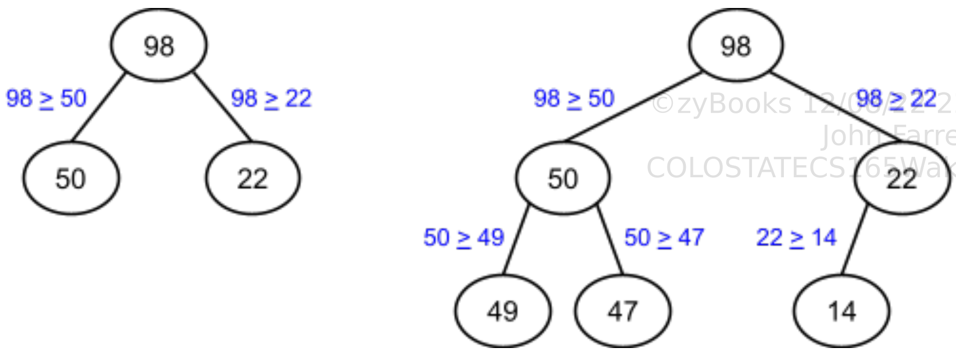
9.2 Heaps

Heap concept

Some applications require fast access to and removal of the maximum item in a changing set of items. For example, a computer may execute jobs one at a time; upon finishing a job, the computer executes the pending job having maximum priority. Ex: Four pending jobs have priorities 22, 14, 98, and 50; the computer should execute 98, then 50, then 22, and finally 14. New jobs may arrive at any time.

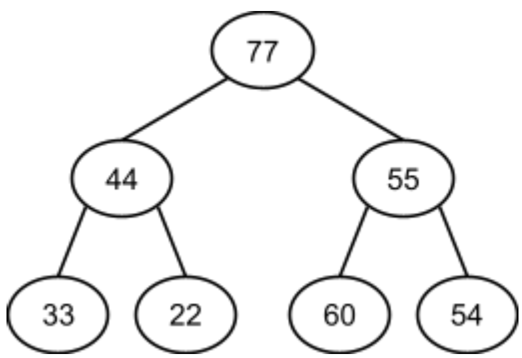
Maintaining jobs in fully-sorted order requires more operations than necessary, since only the maximum item is needed. A ***max-heap*** is a complete binary tree that maintains the simple property that a node's key is greater than or equal to the node's children's keys. (Actually, a max-heap may be any tree, but is commonly a binary tree). Because $x \geq y$ and $y \geq z$ implies $x \geq z$, the property results in a node's key being greater than or equal to all the node's descendants' keys. Therefore, *a max-heap's root always has the maximum key in the entire tree.*

Figure 9.2.1: Max-heap property: A node's key is greater than or equal to the node's children's keys.



PARTICIPATION ACTIVITY 9.2.1: Max-heap property.

Consider this binary tree:



1) 33 violates the max-heap property due to being greater than 22.

- True
- False

2) 54 violates the max-heap property due to being greater than 44.

- True
- False

3) 60 violates the max-heap property due to being greater than 55.

- True
- False

4) A max-heap's root must have the maximum key.

- True
- False

Max-heap insert and remove operations

An ***insert*** into a max-heap starts by inserting the node in the tree's last level, and then swapping the node with its parent until no max-heap property violation occurs. Inserts fill a level (left-to-right) before adding another level, so the tree's height is always the minimum possible. The upward movement of a node in a max-heap is called ***percolating***.

A ***remove*** from a max-heap is always a removal of the root, and is done by replacing the root with the last level's last node, and swapping that node with its greatest child until no max-heap property violation occurs. Because upon completion that node will occupy another node's location (which was swapped upwards), the tree height remains the minimum possible.

PARTICIPATION ACTIVITY | 9.2.2: Max-heap insert and remove operations.

Animation captions:

1. This tree is a max-heap. A new node gets initially inserted in the last level...
2. ...and then percolate node up until the max-heap property isn't violated.
3. Removing a node (always the root): Replace with last node, then percolate node down.

PARTICIPATION ACTIVITY | 9.2.3: Max-heap inserts and deletes.

1) Given N nodes, what is the height of a max-heap?

- $\lfloor logN \rfloor$
- N
- Depends on the keys

2) Given a max-heap with levels 0, 1, 2, and 3, with the last level not full, after inserting a new node, what is the maximum possible swaps needed?

- 1
- 2
- 3

3) Given a max-heap with N nodes, what is the worst-case complexity of an insert, assuming an insert is dominated by the swaps?

- O(N)
- O($logN$)

4) Given a max-heap with N nodes, what is the complexity for removing the root?

- O(N)
- O($logN$)

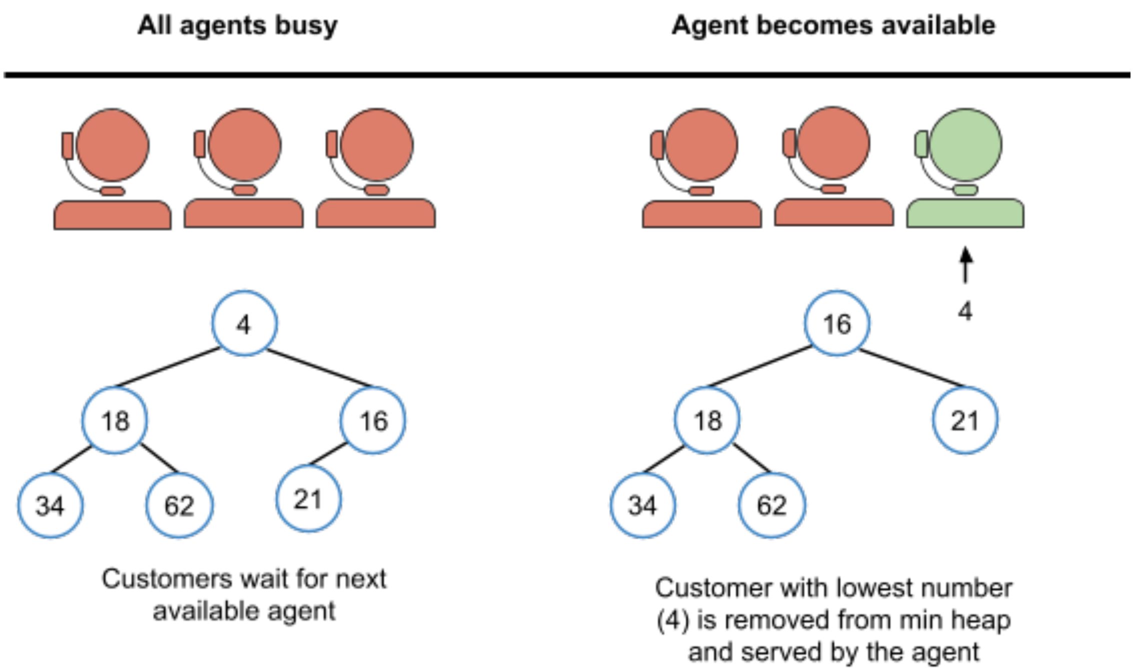
Min-heap

A ***min-heap*** is similar to a max-heap, but a node's key is less than or equal to its children's keys.

Example 9.2.1: Online tech support waiting lines commonly use min-heaps.

Many companies have online technical support that lets a customer chat with a support agent. If the number of customers seeking support is greater than the number of available agents, customers enter a virtual waiting line. Each customer has a priority that determines their place in line. The customer with the highest priority is served by the next available agent.

A min-heap is commonly used to manage prioritized queues of customers awaiting support. Customers that entered the line earlier and/or have a more urgent issue get assigned a lower number, which corresponds to a higher priority. When an agent becomes available, the customer with the lowest number is removed from the heap and served by the agent.



PARTICIPATION ACTIVITY 9.2.4: Min-heaps and customer support.

1) A customer with a higher priority has a lower numerical value in the min-heap.

- True
- False

2) If 2,000 customers are waiting for technical support, removing a customer from the min-heap requires about 2,000 operations.

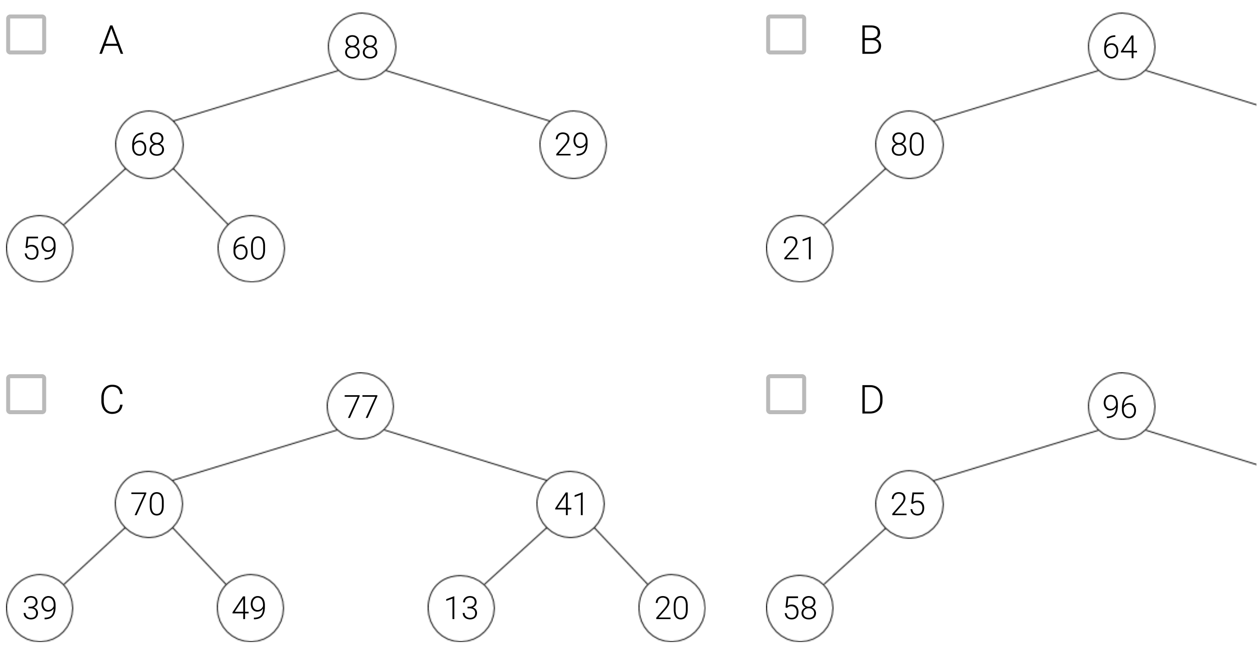
- True
- False

CHALLENGE ACTIVITY | 9.2.1: Heaps.

422352.2723990.qx3zqy7

Start

Select all valid max-heaps.



1 | 2 | 3 | 4

Check Next

9.3 Heaps using arrays

Heap storage

Heaps are typically stored using arrays. Given a tree representation of a heap, the heap's array form is produced by traversing the tree's levels from left to right and top to bottom. The root node is always the entry at index 0 in the array, the root's left child is the entry at index 1, the root's right child is the entry at index 2, and so on.

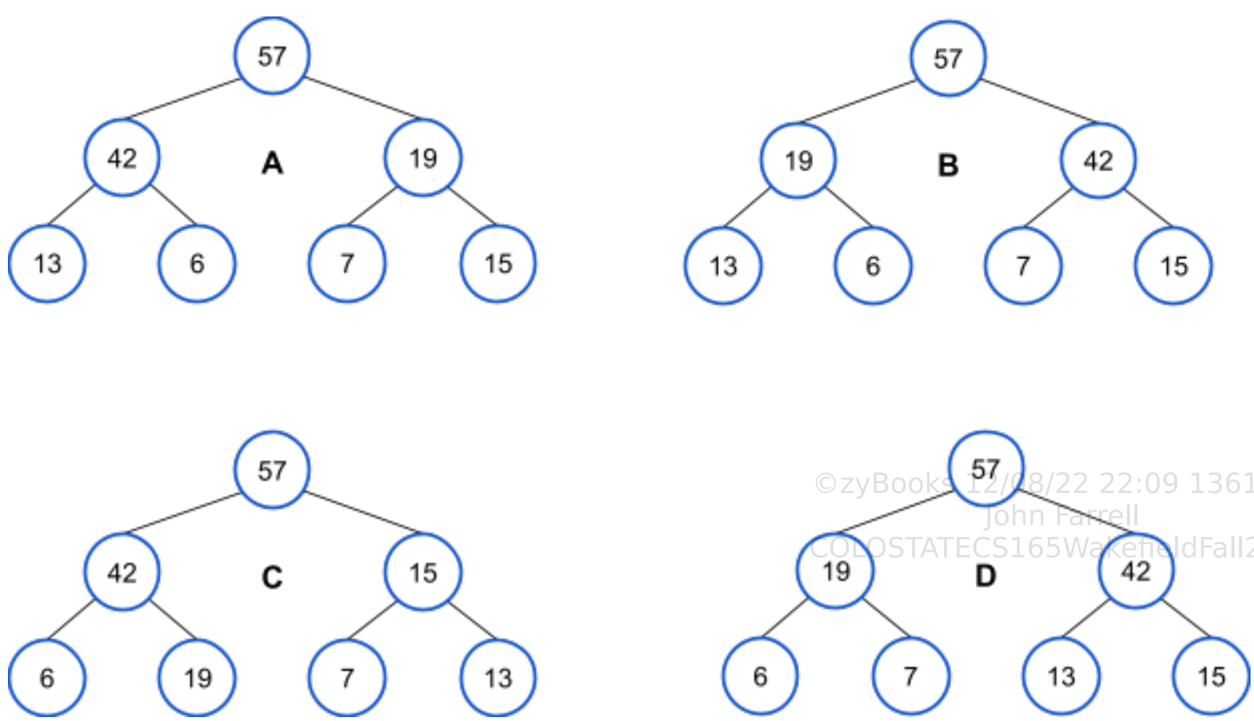
PARTICIPATION ACTIVITY | 9.3.1: Max-heap stored using an array.

Animation captions:

1. The max-heap's array form is produced by traversing levels left to right and top to bottom.
2. When 63 is inserted, the percolate-up operation happens within the array.

PARTICIPATION ACTIVITY | 9.3.2: Heap storage.

Match each max-heap to the corresponding storage array.



©zyBooks 12/08/22 22:09 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

If unable to drag and drop, refresh the page.

Heap B Heap D Heap A Heap C

57	19	42	13	6	7	15

57	42	15	6	19	7	13

57	42	19	13	6	7	15

57	19	42	6	7	13	15

Reset

Parent and child indices

Because heaps are not implemented with node structures and parent/child pointers, traversing from a node to parent or child nodes requires referring to nodes by index. The table below shows parent and child index formulas for a heap.

Table 9.3.1: Parent and child indices for a heap.

Node index	Parent index	Child indices
0	N/A	1, 2
1	0	3, 4
2	0	5, 6
3	1	7, 8
4	1	9, 10
5	2	11, 12
...	...	...
i	$\lfloor (i-1)/2 \rfloor$	2 * i + 1, 2 * i + 2

PARTICIPATION ACTIVITY 9.3.3: Heap parent and child indices.

1) What is the parent index for a node at index 12?

- 3
- 4
- 5
- 6

2) What are the child indices for a node at index 6?

- 7 and 8
- 12 and 13
- 13 and 14
- 12 and 24

3) The formula for computing parent node index should not be used on the root node.

- True
- False

4) The formula for computing child node indices does not work on the root node.

- True
- False

5) The formula for computing a child index evaluates to -1 if the parent is a leaf node.

- True
- False

Percolate algorithm

Following is the pseudocode for the array-based percolate-up and percolate-down functions. The functions operate on an array that represents a max-heap and refer to nodes by array index.

Figure 9.3.1: Max-heap percolate up algorithm.

```
MaxHeapPercolateUp(nodeIndex, heapArray) {
   while (nodeIndex > 0) {
      parentIndex = (nodeIndex - 1) / 2
      if (heapArray[nodeIndex] <= heapArray[parentIndex])
         return
      else {
         swap heapArray[nodeIndex] and
heapArray[parentIndex]
         nodeIndex = parentIndex
      }
   }
}
```

Figure 9.3.2: Max-heap percolate down algorithm.

```
MaxHeapPercolateDown(nodeIndex, heapArray, arraySize) {
   childIndex = 2 * nodeIndex + 1
   value = heapArray[nodeIndex]

   while (childIndex < arraySize) {
      // Find the max among the node and all the node's
children
      maxValue = value
      maxIndex = -1
      for (i = 0; i < 2 && i + childIndex < arraySize; i++)
{
         if (heapArray[i + childIndex] > maxValue) {
            maxValue = heapArray[i + childIndex]
            maxIndex = i + childIndex
         }
      }

      if (maxValue == value) {
         return
      }
      else {
         swap heapArray[nodeIndex] and heapArray[maxIndex]
         nodeIndex = maxIndex
         childIndex = 2 * nodeIndex + 1
      }
   }
}
```


PARTICIPATION ACTIVITY 9.3.4: Percolate algorithm.

1) MaxHeapPercolateUp works for a node index of 0.

- True
- False

2) MaxHeapPercolateDown has a precondition that nodeIndex is < arraySize.

- True
- False

3) MaxHeapPercolateDown checks the node's left child first, and immediately

swaps the nodes if the left child has a greater key.

- True
- False

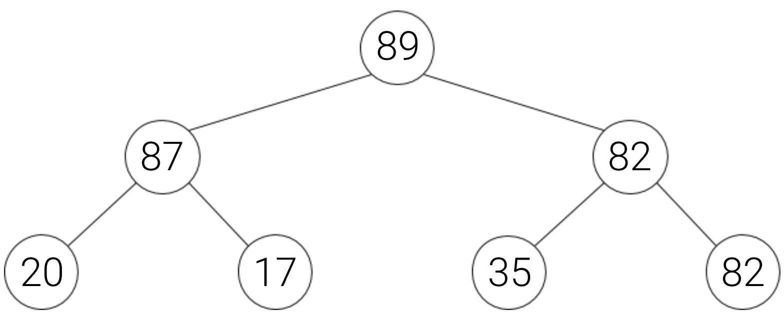
4) In MaxHeapPercolateUp, the while loop's condition nodeIndex > 0 guarantees that parentIndex is >= 0.

- True
- False

CHALLENGE ACTIVITY | 9.3.1: Heaps using arrays.

422352.2723990.qx3zqy7

Start



What is the above max-heap's array form?

Ex: 86, 75, 30 (comma between values)

1	2	3	4	5

Check Next

9.4 Heap sort

Heapify operation

Heapsort is a sorting algorithm that takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted array in reverse order. An array of unsorted values must first be converted into a heap. The ***heapify*** operation is used to turn an array into a heap. Since leaf nodes already satisfy the max heap property, heapifying to build a max-heap is achieved by percolating down on every non-leaf node in reverse order.

PARTICIPATION ACTIVITY 9.4.1: Heapify operation.

Animation captions:

1. If the original array is represented in tree form, the tree is not a valid max-heap.
2. Leaf nodes always satisfy the max heap property, since no child nodes exist that can contain larger keys. Heapification will start on node 92.
3. 92 is greater than 24 and 42, so percolating 92 down ends immediately.
4. Percolating 55 down results in a swap with 98.
5. Percolating 77 down involves a swap with 98. The resulting array is a valid max-heap.

The heapify operation starts on the internal node with the largest index and continues down to, and including, the root node at index 0. Given a binary tree with N nodes, the largest internal node index is $\lfloor N/2 \rfloor$ - 1.

Table 9.4.1: Max-heap largest internal node index.

Number of nodes in binary heap	Largest internal node index
1	-1 (no internal nodes)
2	0
3	0
4	1
5	1
6	2
7	2
...	...
N	$\lfloor N/2 \rfloor$ - 1

PARTICIPATION ACTIVITY 9.4.2: Heapify operation.

1) For an array with 7 nodes, how many percolate-down operations are necessary to heapify the array?

Check Show answer

2) For an array with 10 nodes, how many percolate-down operations are necessary to heapify the array?

Check Show answer

PARTICIPATION ACTIVITY | 9.4.3: Heapify operation - critical thinking.

1) An array sorted in ascending order is already a valid max-heap.

- True
- False

2) Which array could be heapified with the fewest number of operations, including all swaps used for percolating?

- (10, 20, 30, 40)
- (30, 20, 40, 10)
- (10, 10, 10, 10)

Heapsort overview

Heapsort begins by heapifying the array into a max-heap and initializing an end index value to the size of the array minus 1. Heapsort repeatedly removes the maximum value, stores that value at the end index, and decrements the end index. The removal loop repeats until the end index is 0.

PARTICIPATION ACTIVITY | 9.4.4: Heapsort.

Animation captions:

1. The array is heapified first. Each internal node is percolated down, from highest node index to lowest.
2. The end index is initialized to 6, to refer to the last item. 94's "removal" starts by swapping with 68.
3. Removing from a heap means that the rightmost node on the lowest level disappears before the percolate down. End index is decremented after percolating.
4. 88 is swapped with 49, the last node disappears, and 49 is percolated down.
5. The process continues until end index is 0.
6. The array is sorted.

PARTICIPATION ACTIVITY | 9.4.5: Heapsort.

Suppose the original array to be heapified is (11, 21, 12, 13, 19, 15).

1) The percolate down operation must be performed on which nodes?

- 15, 19, and 13
- 12, 21, and 11
- All nodes in the heap

2) What are the first 2 elements swapped?

- 11 and 21
- 21 and 13
- 12 and 15

3) What are the last 2 elements swapped?

- 11 and 19
- 11 and 21
- 19 and 21

4) What is the heapified array?

- (11, 21, 12, 13, 19, 15)
- (21, 19, 15, 13, 12, 11)
- (21, 19, 15, 13, 11, 12)

Heapsort algorithm

Heapsort uses 2 loops to sort an array. The first loop heapifies the array using MaxHeapPercolateDown. The second loop removes the maximum value, stores that value at the end index, and decrements the end index, until the end index is 0.

Figure 9.4.1: Heap sort.

```
Heapsort(numbers, numbersSize) {
   // Heapify numbers array
   for (i = numbersSize / 2 - 1; i >= 0; i--) {
      MaxHeapPercolateDown(i, numbers,
numbersSize)
   }

   for (i = numbersSize - 1; i > 0; i--) {
      Swap numbers[0] and numbers[i]
      MaxHeapPercolateDown(0, numbers, i)
   }
}
```


PARTICIPATION ACTIVITY 9.4.6: Heapsort algorithm.

1) How many times will MaxHeapPercolateDown be called by Heapsort when sorting an array with 10 elements?

- 5
- 10
- 14
- 20

2) Calling Heapsort on an array with 1 element will cause an out of bounds array access.

- True
- False

3) Heapsort's worst-case runtime is O(N log N).

- True
- False

4) Heapsort uses recursion.

True

CHALLENGE ACTIVITY | 9.4.1: Heap sort.

422352.2723990.qx3zqy7

Start

Given the array:

89	38	40	26	29	67	96

Heapify into a max-heap.

Ex: 86, 75, 30

1 2 3

Check Next

9.5 Lab 15 - Priority Queue - Array with Insertion Sort

Module 8: Lab 15 - Priority Queue - Array with Insertion Sort

Cutting in Line

This lab only have one .java file, and you can download it below or copy and paste it into your preferred IDE.

Queues are a nice way to represent a collection of things that eventually need to be processed. But what if some things are more important than others, and we should be getting to them faster? What if we ran the world's worst ice cream shop and let people cut in line based on how much they donated to the owners? We'd need a *priority queue* rather than a regular queue. In fact, that unscrupulous ice cream shop is exactly what we're going to be making a reality in this lab.

By this point, you know the drill. We're going to be implementing a data structure using an array. Much of what you've already done in previous labs has been taken care of for you - the main problem in this lab is getting the queue to automatically sort its elements such that, whenever you do a pop or peek, the **highest priority** customer comes out. There are a few catches with this lab in particular, though, so be sure to read on.

The Customer is Always Right

The PriorityQueue you're going to be working with in this lab is not generic. It's meant to represent a queue of customers at a shop, so it only stores Customers, a special class we've made and filled out for you. But remember, our shop is rigged! The position people have in the queue is based on how much they've donated to the shop, and not necessarily the order they arrived at the queue. You will use a compareTo method to put them in order so that the person who has donated the most is in the front of the queue, the highest priority.

Coming Through!

It's not bad enough that some customers get preferential treatment - it's also possible for them to **change their priority while in the queue**. The bump() method takes in the index of a customer in the queue adds a certain amount of dollars to their donation, letting them potentially cut in line!

Comparables

You'll be making extensive use of compareTo in this lab. The Comparable interface specifies that compareTo should work in a certain way that can be a little hard to remember.

So here's a pro tip for remembering how to use compareTo: imagine you want to do the comparison

```
if (apple < orange)
```

All you need to do is replace the comparison operator with a call to compareTo, and then use the same comparison operator to compare the result of the compareTo to a 0. It looks like this:


```
if (apple.compareTo(orange) < 0)
```

As another example, `apple >= orange` would turn into `apple.compareTo(orange) >= 0`.

The PriorityQueue

Our priority queue stores two things: an array of customers (our main data array), and a count of how many customers are in the queue. The constructors have been handled for you.

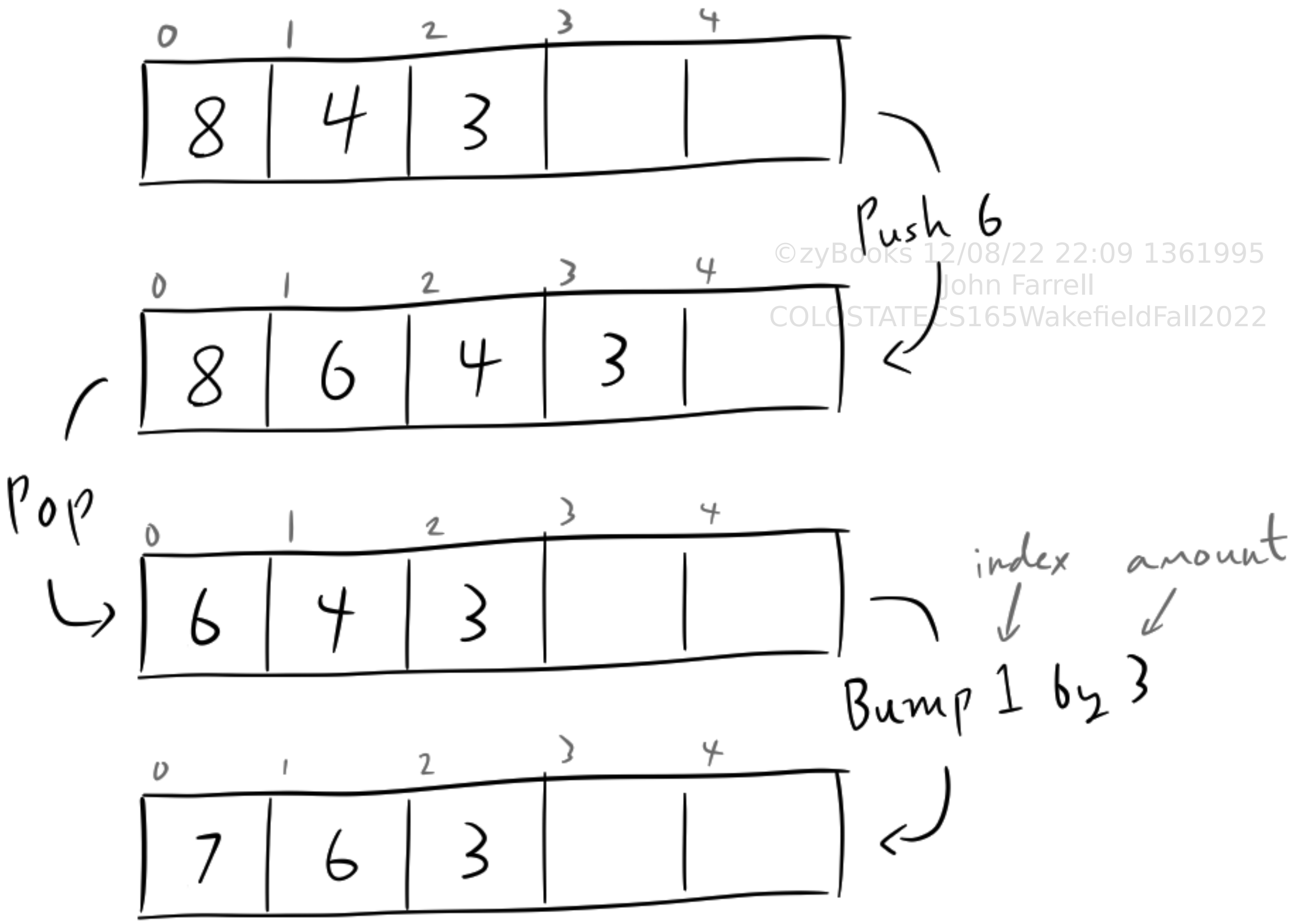
We can use our array somewhat like a stack to implement the behavior we want. The array should **stay in sorted order**, keeping the highest priority customer at the beginning of the array and the lowest at the end of the array.

The queue needs to support the following operations:

- push(): Add a customer to the queue, putting it in the proper sorted place.
- pop(): Remove the highest priority customer from the queue.
- peek(): Return, but do not remove, the highest priority customer.
- bump(): Increase the donation amount of the customer at a given index by a given amount of money, moving the customer in the queue if necessary

Make sure your operations pay attention to how many customers are in the queue! Hint: exceptions need to be thrown if certain operations are called with no customers or if you try to add too many customers.

Here are few examples of these operations; note how the array **always stays sorted**, and the leftmost item (at index 0) is always what is popped off:



Exceptions

You will need to use exceptions in this lab on some of these methods. For example, if you try to insert an item when the capacity is full or when you try to take out an item when the queue is empty. *Hint: we'll be testing these two scenarios.* It's better to be specific when throwing exceptions, so you should use an `IllegalStateException` in the former case and a `NoSuchElementException` in the latter.

Wrapping Up

As is tradition, the main method is full of code you can use to test your implementation. Run it when you're done writing methods to see if you've done it right. If you get the same output as this:

```
Testing push
Line should be:
[$10.00, $5.00, $1.00, null, null, null]
[$10.00, $5.00, $1.00, null, null, null]
Line size should be 3 is: 3
```

```
Testing pop
$10.00
$5.00

Testing bump
$61.00
$45.00
$45.00
$20.00
$2.00

Line should be:
[$10.00, $9.00, $8.00, $7.00, $7.00, null]
[$10.00, $9.00, $8.00, $7.00, $7.00, null]
```


Then you've got it!

Submission

In Submit mode, select "Submit for grading" when you are ready to turn in your assignment.

422352.2723990.qx3zqy7

LAB ACTIVITY | 9.5.1: Lab 15 - Priority Queue - Array with Insertion Sort | 0 / 7

Downloadable files

PriorityQueue.java Download

PriorityQueue.java Load default template...

```java
import java.util.Arrays;

public class PriorityQueue {

    /* This class is finished for you.
     */
    public static class Customer implements Comparable<Customer> {
        private double donation;

        public Customer(double donation) {
            this.donation = donation;
        }

        public double getDonation() {
            return donation;
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **PriorityQueue.java** (Your program) →

Program output displayed here

Coding trail of your work   What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

9.6 Lab 16 - Priority Queue - Heap

Module 8: Lab 16 - Priority Queue - Heap

What a Heap of Trash

Note: **This lab has a single .java file, which you can download below or copy and paste into your preferred IDE.**

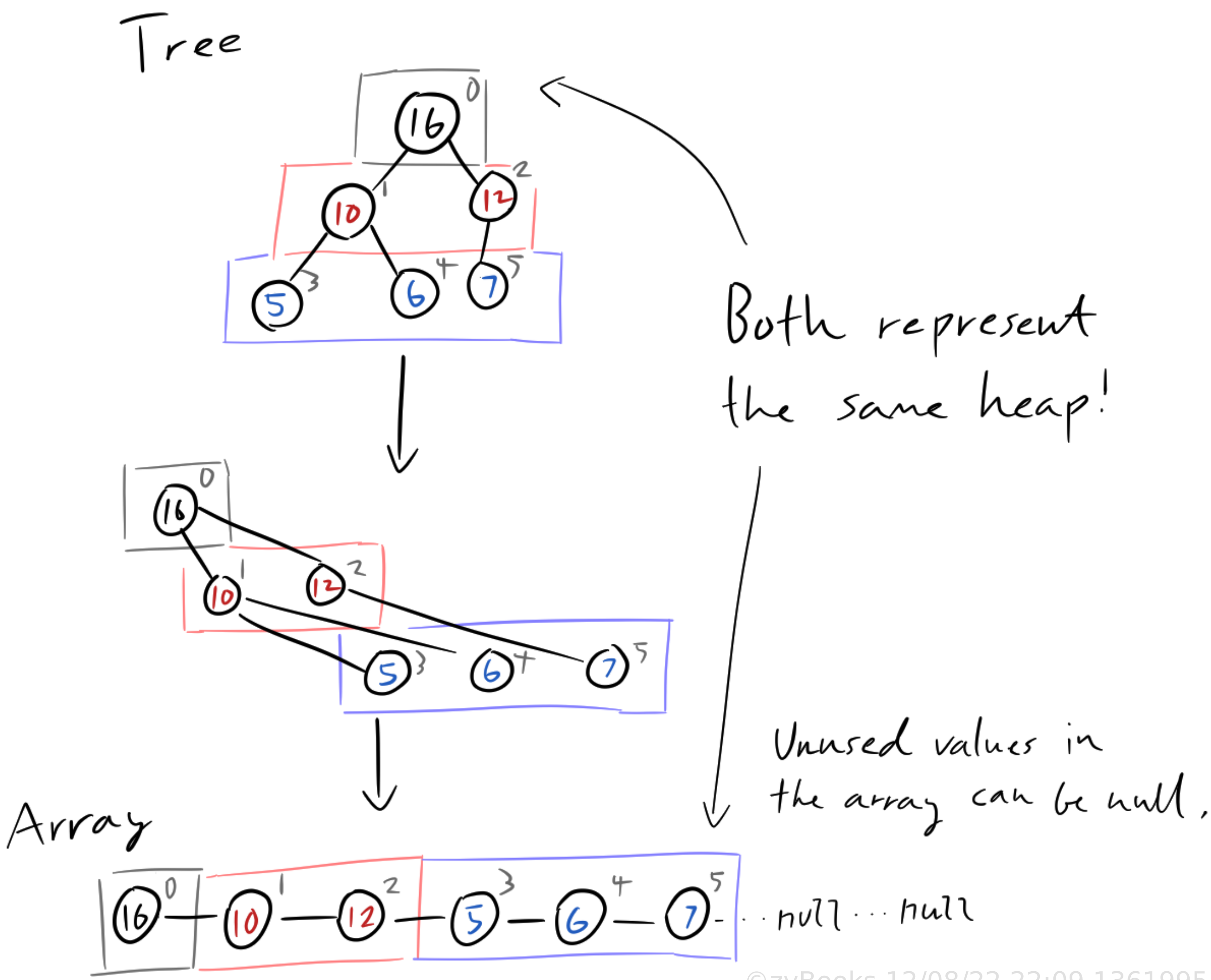
Using a sorted array to create a priority queue is fairly simple, but it's not very fast. A better (and more importantly, *more clever*) way to use arrays for a priority queue is to implement a heap!

Unlike the previous lab, we will be working with a generic data structure. We are also not implementing the bump() operation from last time - just the basic push, pop, and peek. I think you'll find this is plenty enough work, however.

The Heap Structure

Understanding how to represent a heap with an array is incredibly important for this lab, so we'll go over it again here.

Recall that a heap is complete binary tree - binary meaning each node has at most two children, and complete meaning that the tree fills itself out from top to bottom and left to right, like words on the pages of a book. There are no "gaps" in the tree, which means we can represent it with a contiguous array or list (but we prefer arrays, since they are faster).



You can think of the array representation of a heap as sort of like a folded version of the tree, where each layer of the tree are put one-after-another in a line, rather than being stacked on top of each other.

A very important property of a heap is that the **parent is always higher (or lower) priority than the children**. If you are working with a max-heap, the parent must always be higher priority - if you are working with a min-heap, the parent must always be a lower priority. This lab uses a **max-heap**, so you must make sure that every node in your parent is higher priority than its children! This parent-child relationship is so important for heaps that it is often called the **heap property**.

In the array representation, it's not visually obvious what the parent of a given node is, or the children. Thankfully, some very simple formulas can be used to get that information:

- the parent of the node at index **i** is at `floor((i-1)/2)`
- the left child of the node at index **i** is at `2i+1`
- the right child of the node at index **i** is at `2(i+1)`

The Heap Class

In the lab code, our heap class contains a fixed-size array called `heap` and a count of how many elements are in this array. The class implements a simple priority queue interface, so it supports the push(), pop(), and peek() operations. Remember what this means: push() adds to the queue, pop() gets and removes the highest priority element (which is always the first element in the array!), and peek gets the highest priority element without removing it.

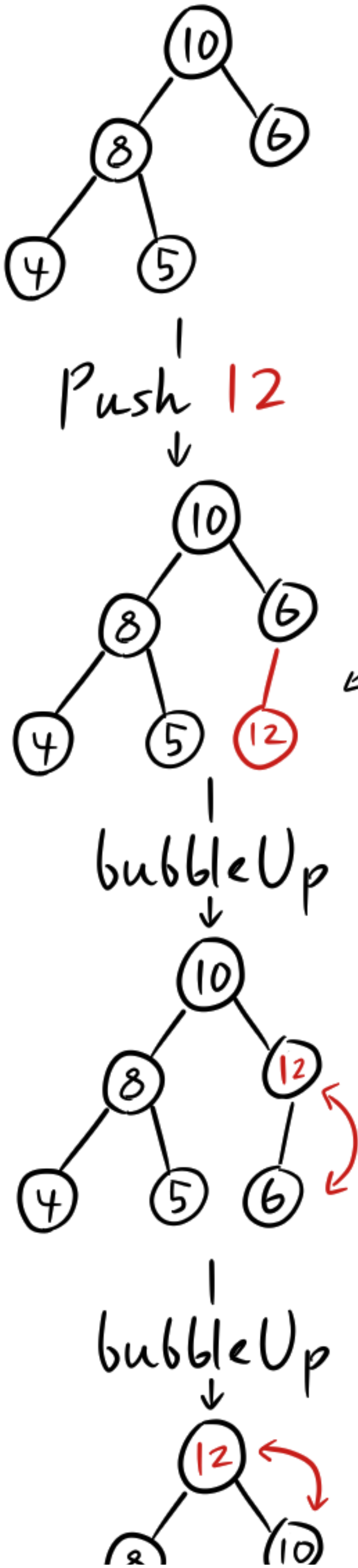
To help you complete these, there are quite a few helper methods you need to complete first.

1. parent(), rchild(), and lchild() are just implementations of the above formulas. They take in an index and tell you the index of the parent, right child, and left child respectively. Remember that, if you divide two integers in Java, a floor is automatically done for you, as the decimal part of the number is dropped.
2. hasLeftChild() and hasRightChild() take in an index and tell if the index has a valid left child or right child. If the child index falls outside the array, or outside the section of your array that represents valid data, then there is no valid child.
3. swap() is just a utility function that swaps the contents of two spaces in the array. You'll be using it a lot in other functions.
4. bubbleDown() is where things get interesting! This operation **restores the heap property** at a given index by moving it downwards through the heap if it is too small. First, it checks if the element at the given index is smaller than either of its children. If it is, it swaps the element with its higher priority child. Then, the process **recursively** repeats on the index we just swapped the element to. After this function is over, the element should be in its proper place in the heap.
5. bubbleUp() is just like bubbleDown(), except rather than moving an element *down* to a proper position, it moves the element *up* to a proper position. It checks to see if the element at the given index is greater than its parent, and if it is, it swaps itself with its parent, and then repeats the process on the index it was swapped to. After this function is done, the element should be in its proper place in the heap.

We can use bubbleUp and bubbleDown to help implement push and pop. When we push an element onto the queue, we place it **at the end of the array**, which is at the bottom of the heap. Of course, this may break the heap property - what if that thing we just added is a very large element, and it needs to be higher up in the heap? So, we call bubbleUp on the index we placed it in! This moves it up to its proper position in the heap, and ensures the heap property is maintained.

Note: The following examples illustrate a heap holding Integers, but the heap really could store any type where objects of that type can be compared to one another. Strings, for example, would be stored in the heap alphabetically, with the "larger" strings being ones that start with later letters. Keep this in mind, since the test code for this lab uses a String heap rather than an Integer one.

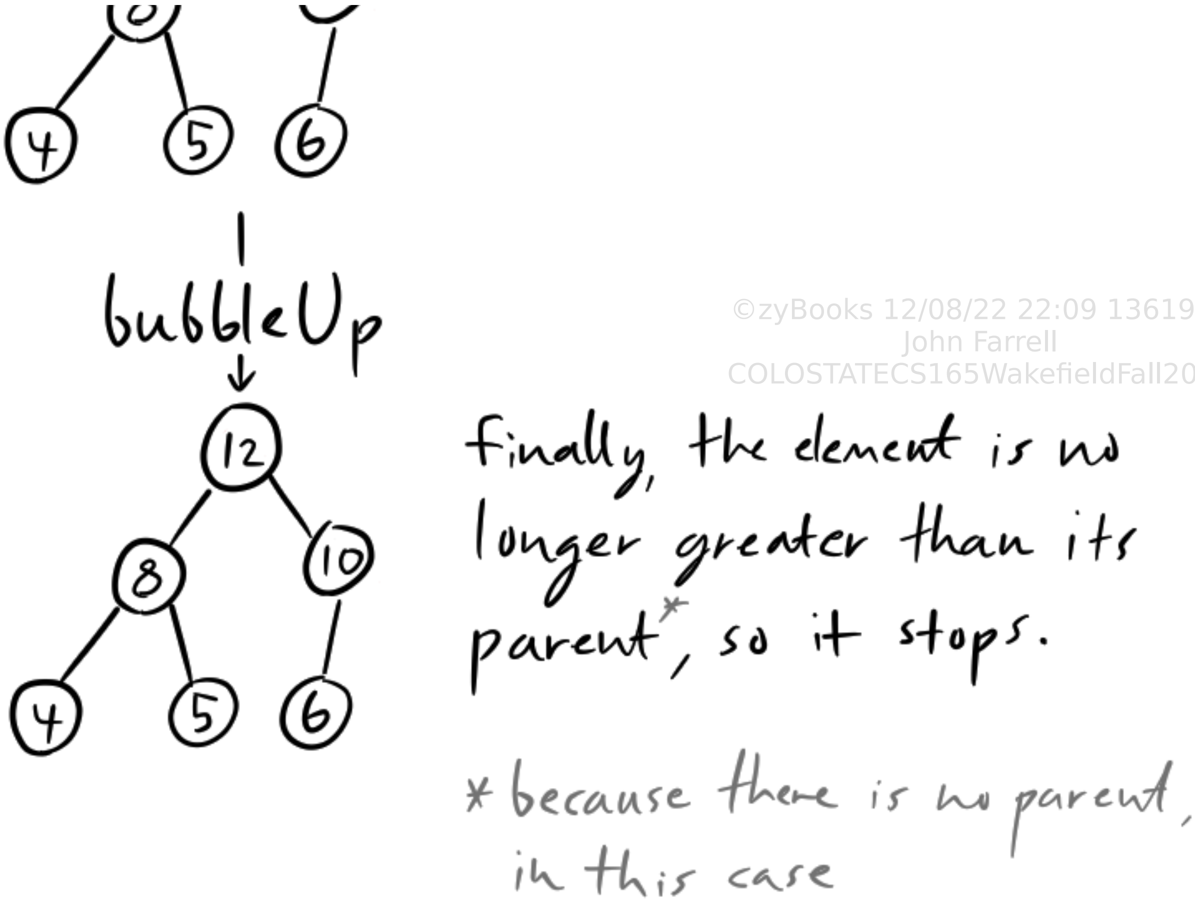
©zyBooks 12/08/22 22:09 1361995
John Farrell
COLOSTATECS165WakefieldFall2022



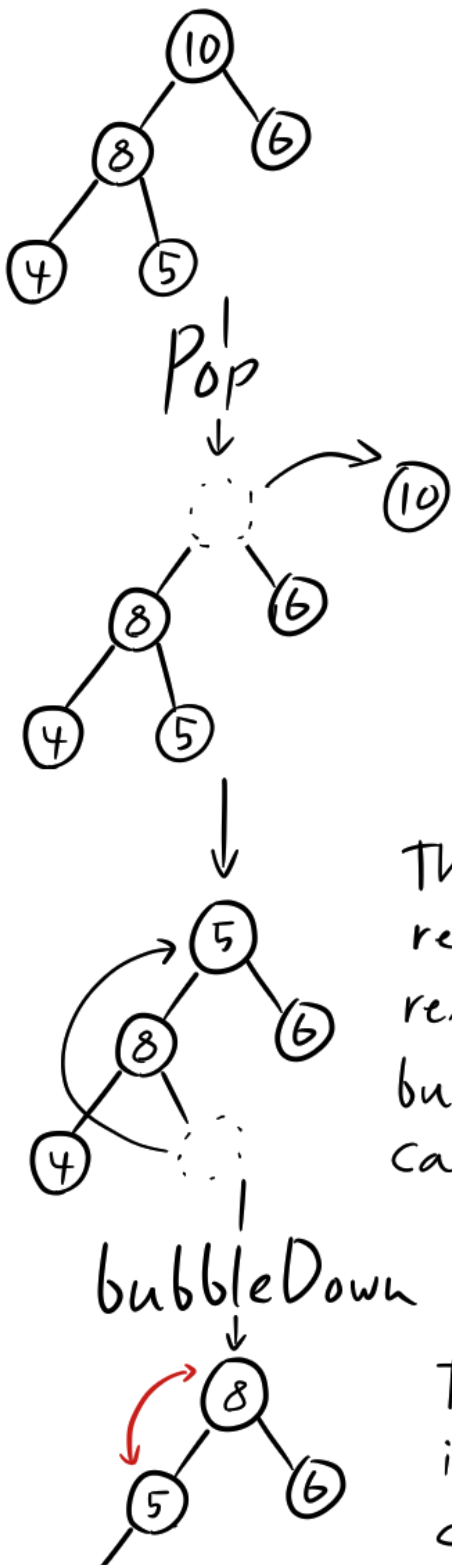
The new element goes on the end of the array

We call bubbleUp on the new element's index –

It finds the element is larger than its parent, so it swaps and continues.



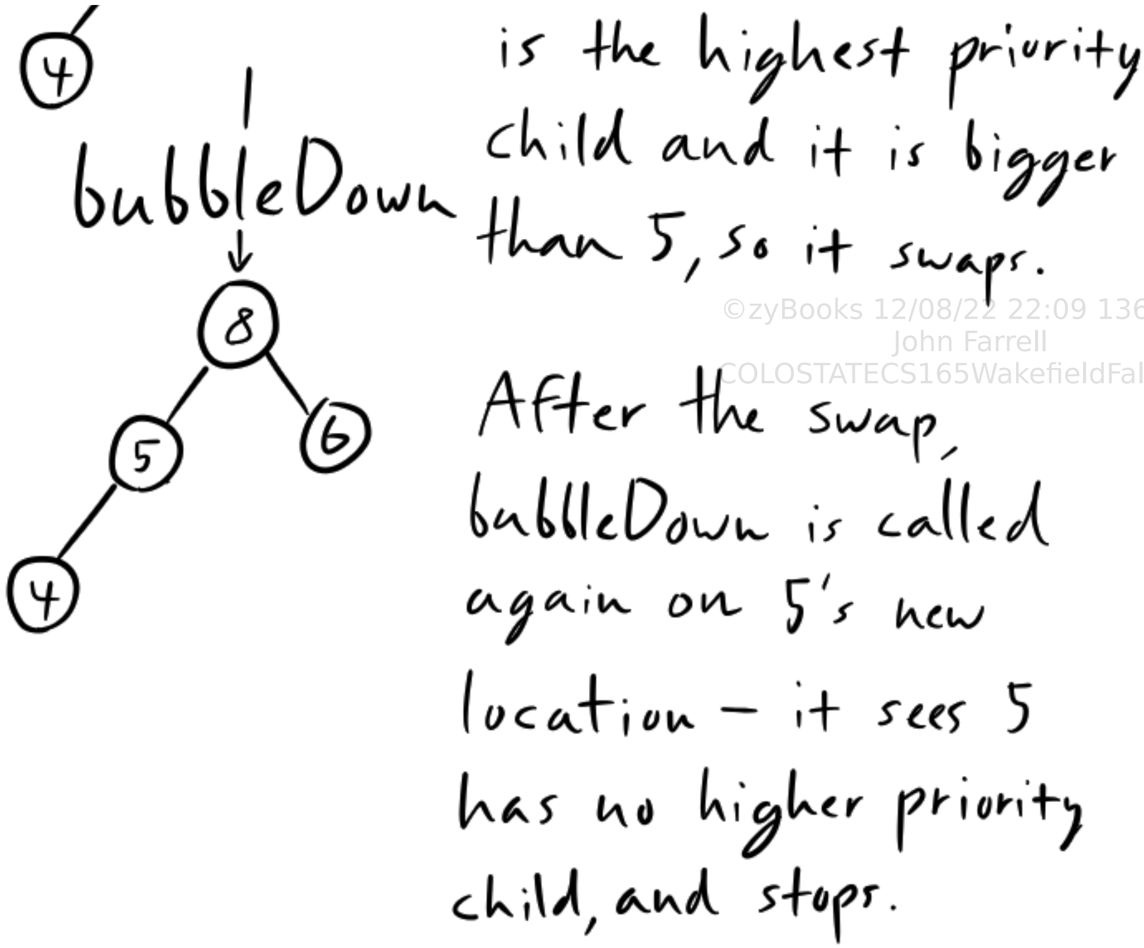
Popping is a little more interesting. When we pop an element, the top-most element (first in the array) is removed; because of the heap property, the top-most element is guaranteed to be the highest priority when we remove it. To fill the empty space created by removing that element, we move the bottom-most element (last in the array) to the very top. Of course, this will probably break the heap property, so we call a bubbleDown on this new top-most element, ensuring it is placed in its proper spot.



The top-most element is removed, to be returned later.

The last element replaces what was removed–
bubbleDown is then called on index 0.

The new top element is compared with its children – it sees 8



Wrapping Up

The main method tests your heap by making a String heap and doing a series of pushes, pops, and peeks. For Strings, they are in lexicographical order. In other words, y is greater than a. The output should look something like this:

```
push lime       -> [lime, null, null, null, null, null]
push fuchsia    -> [lime, fuchsia, null, null, null, null]
push cyan       -> [lime, fuchsia, cyan, null, null, null]
push yellow     -> [yellow, lime, cyan, fuchsia, null, null]
push maroon     -> [yellow, maroon, cyan, fuchsia, lime, null]

pop yellow      <- [maroon, lime, cyan, fuchsia, yellow, null]
pop maroon       <-[lime, fuchsia cyan, maroon, yellow, null]
pop lime       <-[fuchsia, cyan, lime, maroon, yellow, null]


peek fuchsia    <- [fuchsia, cyan, lime, maroon, yellow, null]
peek fuchsia    <- [fuchsia, cyan, lime, maroon, yellow, null]
```

```
push olive      -> [olive, cyan, fuchsia, maroon, yellow, null]
push icterine   -> [olive, icterine, fuchsia, cyan, yellow, null]
push sienna     -> [sienna, olive, fuchsia, cyan, icterine, null]
push silver     -> [silver, olive, sienna, cyan, icterine, fuchsia]
push teal       -> [silver, olive, sienna, cyan, icterine, fuchsia]
pop silver      <- [silver, olive, sienna, cyan, icterine, fuchsia]
pop sienna      <- [sienna, olive, fuchsia, cyan, icterine, silver]
push slate      -> [slate, olive, fuchsia, cyan, icterine, silver]
pop slate       <- [slate, olive, fuchsia, cyan, icterine, silver]
peek olive      <- [olive, icterine, fuchsia, cyan, slate, silver]
pop olive       <- [olive, icterine, fuchsia, cyan, slate, silver]
peek icterine   <- [icterine, cyan, fuchsia, olive, slate, silver]
pop icterine    <- [icterine, cyan, fuchsia, olive, slate, silver]
pop fuchsia     <- [fuchsia, cyan, icterine, olive, slate, silver]
pop cyan        <- [cyan, fuchsia, icterine, olive, slate, silver]
pop null        <- [cyan, fuchsia, icterine, olive, slate, silver]
peek null       <- [cyan, fuchsia, icterine, olive, slate, silver]
```


The values off to the left describe what is happening to the tree, including the results of any pop() and peek() operations - **these should be the same in your output**. The lists off to the right show what your internal heap array looks like at each step, and are mostly for your own debugging benefit so you can see your heap in action. The lists don't have to be the same in your own output - in fact, depending on your implementation, they may be a bit different (this implementation "removes" elements by pushing them to the back and ignoring them, while you may decide to replace them with `null`, for instance)

Once you get the values to come out properly, you've got it! Be patient and stick at it; you may find this lab harder than most others. Please do not be afraid to ask your TAs for help if you do not understand something or get stuck!

422352.2723990.qx3zqy7

LAB ACTIVITY | 9.6.1: Lab 16 - Priority Queue - Heap | 0 / 7

Heap.java

Load default template...


```
import java.util.Arrays;

interface PriorityQueue<T extends Comparable<T>> {
    public void push(T item);
    public T pop();
    public T peek();
}

public class Heap<T extends Comparable<T>> implements PriorityQueue<T> {

```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **Heap.java** (Your program) → 0

Program output displayed here

Coding trail of your work What is this?

```
10/27 R----------------------0 min:67
```