Priority Queues Part #1

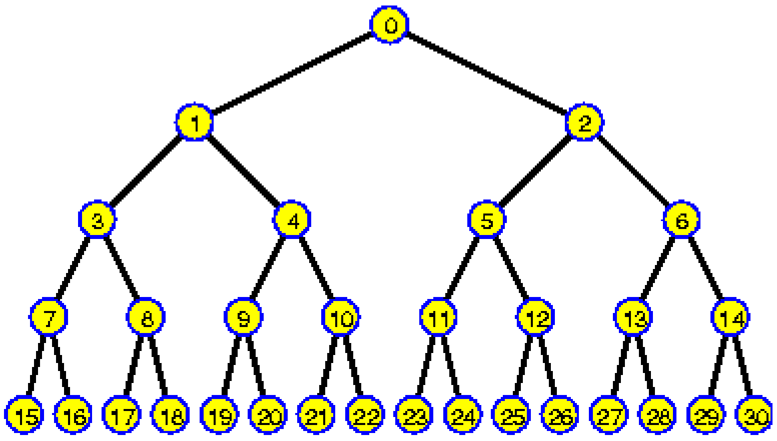
Re-submit Assignment

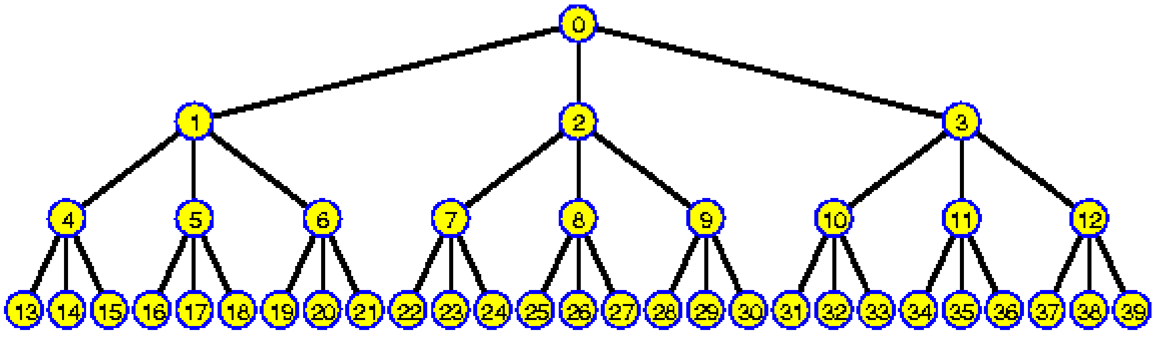
Due May 27, 2017 by 11:59pm **Points** 100 **Submitting** a file upload
File Types zip

Overview

We have explored how to use a binary heap for implementing the priority queue ADT. Recall that a priority queue is a smart queue that allows for retrieval of elements from a queue based on the importance (priority) of an element.

An alternative design would be to use a tree with up to m children, called an m-ary tree. A heap based on this tree would be a complete m-ary tree, where each node in the tree can have up to m children. Note that this is just an extension of the binary heap we have discussed, where m = 2. Below you can see two heaps, one of which has m = 2 and the other has m = 3.





This assignment will focus on building a heap based on an m-ary tree. Each student will be assigned a different value of m from 3-5. This will allow the class to collectively build several variations of this data structure. Your "m" value can be seen below:

Name	M value
ANTONYUK SOFIYA	4
BACK JAMES	4
BENNER ELIZABETH	5
BOURQUE MICHELINE	3
BRADEN KYLE	4
CALLAHAN PATRICK	5
CHURCH JEFFREY	4
GINN THEODORE	3
HASCUP NATHANAEL	4
HAWKS JOSHUA	5
LEANO MARLENE	5
LOCKE KEVIN	5
MANALO JEREMY-RYAN	3
MCCOY JONNATHON	4
MOON SSONNI	5
NGUYEN DUC	5
NGUYEN KEVIN	4
NOBLE NEAL	3
OSTRANDER CALEB	5
PADILLA ELENA	4
PHAM CYNTHIA	4
PRATT JEFFREY	3
ROUSH TIMOTHY	3
SAYLOR BRIAN	4
STRAND NATHAN	3
TAYLOR BRENT	4

Note: Any student submitting a solution with the incorrect value of m will receive a zero on the assignment. Double check your value of m before starting on the assignment!

Implementation

Recall that a complete tree can be stored efficiently in an array using the indices displayed in the tree above. When traversing a heap with the swim() and sink() operations, it is imperative to know how to identify the position of a parent node or child node in the array.

Assume that nodes are stored at indices beginning with 0 (zero). For a m-ary heap and a node at index i, the following nodes can be identified:

Parent: (i - 1) / m
Children: i*m + 1, i*m + 2, i*m + 3, ... , i*m + m

Note: verify that this is the case by drawing an m-ary tree with your given m-value and practicing these calculations.

The m-ary Heap Operations

Build a priority queue using an m-ary heap as described above. You will first need to download a starter file, found here. Your class should support each of the operations given in the starter file. Also, your heap should be stored in an array internally. Anyone submitting a solution without this design will be forced to resubmit their work.

The details for each operation are given below:

- insert(element)
  - If an element is added to an empty heap, then add your initial element to index zero in your array.
  - If an element is added to a non-empty heap, then add your new element to the next unused index in your array and call swim() on your new element. Recall that a parent index can be found at (i - 1) / m, where i is the index of your new element.
    - Make sure to carefully debug your swim() function!
- peek()
  - Returns the element at index zero in your array, if present, otherwise peek() should return null if the heap is empty.
- delMin():
  - This should return the element found at index zero in your array.
  - This function should also take the highest used index of an element in your array, move this element to the root of your tree (index zero) and then call sink() on index zero.
    - During sink(), the operation will need to choose the smallest of m child nodes (as opposed to two nodes with a binary heap). This will require a loop to view each child index: i*m + 1, i*m + 2, i*m + 3, ... , i*m + m
    - You will also need to be careful only to consider a child node if the index is a valid element in your tree. i.e. if the index above is larger than the size() - 1 you will have a null value at that position in your array.
    - Make sure to carefully debug your sink() function!
- size(), isEmpty()
  - Behaves similarly to other data structures we have covered in class.
- clear()
  - This should efficiently remove all elements from your heap.
  - *Note: This can be done in one line of code!*
- contains(element)
  - Returns true if the element is found in the heap, otherwise the method returns false.

Testing

I have provided a file for you to unit test your structure above. Part of your grade will be determined by how you verify the functionality of your priority queue using these tests. Your tests should at least include the following cases:

- Calling insert() on an initially empty tree.
- Calling insert() on a non-empty tree.
- Calling delMin() on an empty tree.
- Calling delMin() on a non-empty tree
- Verifies that size(), isEmpty() and clear() behave predictably.
- Verifies that elements can located through contains().
- Writing a loop to remove all elements from the tree, by repeatedly calling delMin() on your queue.
- Adding n random values to the queue and retrieving them in sorted order from the queue. Your elements should be retrieved in ascending order.
    - Test your heap for large n. i.e. n should be close to 10,000, 100,000 or 1,000,000 elements.
    - How can you verify your results?

Submission

Submit the code for your priority queue to the dropbox on Canvas. Come to class ready to discuss the differences between your designs and the standard binary heap.

Priority Queues Part #1 Rubric

Criteria	Ratings		Pts
insert() correctly places an element into the heap and reorders the elements in the heap.	15.0 pts Full Marks	0.0 pts No Marks	15.0 pts
delMin() correctly removes the minimum element from the heap and reorders the elements in the heap.	15.0 pts Full Marks	0.0 pts No Marks	15.0 pts
size() and isEmpty() correctly track the number of elements in the heap.	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
clear() removes all elements from the heap.	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
contains() correctly identifies elements in the heap.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
Unit tests verify insert() on empty and non-empty trees. Unit tests verify delMin() on empty and non-empty trees.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
Unit tests verify size(), isEmpty() and clear() according to the description above.	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Unit tests verifies that a loop can remove all elements from the heap.	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Unit tests verify that the heap returns elements in sorted order. The heap has been tested with large numbers of inputs, according to the description above.	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Styles: naming conventions, brace placement, spacing, commented code, redundancy, packages and magic numbers.	15.0 pts Full Marks	0.0 pts No Marks	15.0 pts
Formal documentation: full Javadocs & comment block header.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
			Total Points: 100.0