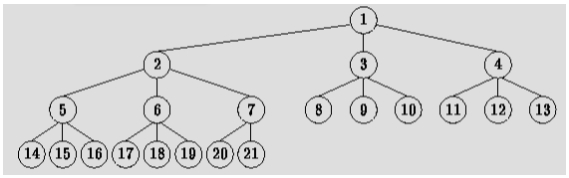
* **Due** Saturday by 11:59pm

**Overview**

A d-heap can be used as an alternative to 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.

A d-heap is a complete d-ary tree, where each node in the tree can have up to d children. This assignment will focus on building a heap from a complete ternary tree (3 child nodes). One such tree can be seen below:



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 percolateUp() and percolateDown() operations, it is imperative to know how to identify the position of a parent node or child node in the array.

For a binary heap and a node at index i, the following nodes can be identified:

**Parent**: i / 2  
**Left child**: 2\*i  
**Right child**: 2\*i + 1

A similar calculation can identify nodes in a ternary tree:

**Parent**: (i + 1) / 3  
**Left child**: 3i - 1  
**Middle child**: 3i  
**Right child**: 3i + 1

*Note: verify that this is true using the tree pictured above!!*

**Ternary Heap**

Build a priority queue using a ternary heap as described above. You will first need to download a starter file, [found hereView in a new window](https://egator.greenriver.edu/courses/1273210/files/63631542/download?wrap=1). Your class should support each of the operations given in the starter file. Also, your ternary heap should be stored in an array internally. The details for each operation are given below:

* add(element)
  + This is the insert() operation from the priority queue ADT.
  + If an element is added to an empty heap, then add your initial element to index one 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 percolateUp() on your new element. Recall that a parent index can be found at (i + 1) / 3, where i is the index of your new element.
    - Make sure to carefully debug your percolateUp() function!
* peek()
  + This is the findMin() operation from the priority queue ADT.
  + Returns the element at index one in your array, if present, otherwise peek() should return null if the heap is empty.
* remove():
  + This is the deleteMin() operation from the priority queue ADT.
  + This should return the element found at index one in your array.
  + This function should also take the highest index of an element in your array, move this element to the root of your tree (index one) and then percolateDown() with index one.
    - During percolateDown(), the operation will need to choose the smallest of three child nodes (as opposed to two nodes with a binary heap).
    - You will also need to be careful only to considered a child node if the index is a valid element in your tree. Notice how the node at index seven in the tree pictured above does not have a valid right child.
    - Make sure to carefully debug your percolateDown() 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!*

**Testing**

Create a test class and thoroughly test your methods above. Part of your grade will be determined by how you verify the functionality of your priority queue using tests. Your tests should include at least the following cases:

* Adding to an initially empty tree.
* Calling remove() on an empty tree.
* Calling remove() on a tree with one element.
* Writing a loop to remove all elements from the tree, by repeatedly calling remove() 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 n = 100, 1000, 10000, and 100000. How can you verify your results?

**Submission**

Submit the code for your priority queue to the dropbox on Canvas. Also, submit a few paragraphs that answer the following questions:

* What are the advantages/disadvantages of your design above?
* Would you use a ternary heap or binary heap in your own code? Why?
* What is the big-oh for each of the operations on a ternary heap? How is the big-oh estimation for these functions different from the estimates for a binary heap.