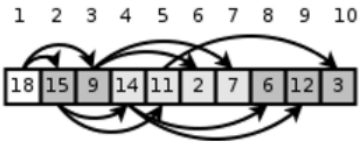
Min Heap



A heap is a ***complete*** binary tree, where all levels are completely filled (except possibly the last level – the last level will have all values as "left" as possible). Examples of valid heaps:

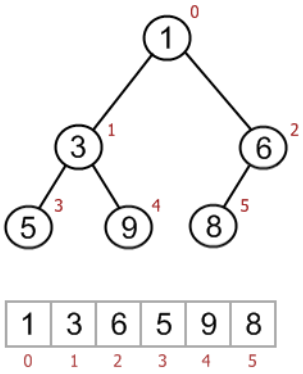
```
          10                          10
        /    \                      /    \
      20      100                 15      30
     /                           /  \    /  \
    30                          40   50  100  40
```

This property offers a variety of benefits, one of which is the ability to vary the number of elements in a heap quite easily. On the other hand, each node stores two auxiliary links, which requires additional memory costs.

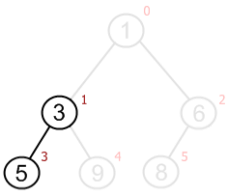
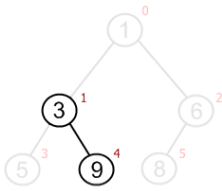
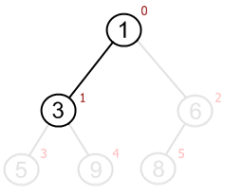
As mentioned previously, a heap is (by definition) a complete binary tree, which leads to the idea of storing it using an array. By utilizing an array-based representation, we can reduce memory costs while tree navigation remains quite simple.

Mapping a heap to an array

A heap can be stored in an array level by level. The top-most level contains the root only and is mapped to the first element of an array (index 0). The root's children are mapped to the second and third elements and so on. As a heap is a complete binary tree, this guarantees that a heap's nodes take up places in the array compactly, making the mapping quite efficient. Example:



This mapping scheme follows the convenient formulas seen below:

Left(i) = 2 * i + 1	Right(i) = 2 * i + 2	Parent(i) = (i - 1) / 2
		

You can see now that navigating a heap, mapped to an array, is actually very easy.

In this lab, you will implement an array-based "min heap" class that preferentially stores minimum values. In other words, the highest-priority element (in the case of a min heap, the smallest value) will always be stored at the root. In a min-heap, the data contained in each node is less than (or equal to) the data in that node's children.

Note: you can (and probably should, to be consistent with the powerpoints / runner code / animation) also leave index 0 empty and use index 1 as the first index to make calculations slightly "rounder":

- Store the root in position 1.
- For any node in position `i`,
  - its left child (if any) is in position `2 * i`
  - its right child (if any) is in position `2 * i + 1`
  - its parent (if any) is in position `i / 2` (use integer division)

The min heap animation found here (also linked in Canvas) may be useful later, for understanding how insert and pop min work. Implement the MinHeap class as follows:

Variables

- `Integer[] heap` – the array backing the heap
  - Use `Integer[]` rather than `int[]` to avoid confusion on whether a 0 in the array is actually a 0 added to the heap or a default array element value. Integer objects can be `null`.
- `int size` – NOT the capacity of the array, the number of actual nodes added to the heap (i.e. logical size)
- `j`

Methods

- Default and parameterized constructors (user specified initial size and not), chained with `this()`
- `int getSize()` – `size` getter
- `boolean isEmpty()` – returns true if the heap's size is 0
- `int peekMinimum()` – returns the min value of this heap
- `int getLeftChildIndex(int index)`
- `int getRightChildIndex(int index)` – returns the index of the left and right child, given a particular parent node (root) `index`
- `int getParentIndex(int index)` – gets the index of `index`'s parent node
- `private void doubleCapacity()` – doubles the array's capacity, when adding an element would exceed current capacity (used by the `insert` method, below)

- `void insert(int value)` – inserts a value into the last position in the heap, then calls the `bubbleUp` method to restructure the tree such that the inserted element is in the proper position.
  - Should supply an `index` value corresponding to the next available position to `bubbleUp`, such that the method starts at the "end" of the heap.

- `private void bubbleUp(int index)` – recursive helper method that allows a value added to the heap to "bubble up" to its correct position.

- `int popMinimum()` – removes and returns the min value of this heap. Calls `siftDown` to restructure the tree and to check that the tree remains a heap

- `private void siftDown(int index)` – recursive helper method that allows a value added to the head to "sift down" (bubble down) to its correct position in the heap.

- Copy/paste the following into your class (used for testing):

```
@Override
public String toString()
{
	String output = "";

	for (int i = 1; i <= getSize(); i++)
		output += heap[i] + ", ";

	return output.substring(0, output.lastIndexOf(",")); //lazily truncate last comma
}

/** method borrowed with minor modifications from the internet somewhere, for printing */
public void display() {
	int nBlanks = 32, itemsPerRow = 1, column = 0, j = 1;
	String dots = "...............................";
	System.out.println(dots + dots);
	while (j <= this.getSize())
	{
		if (column == 0)
			for (int k = 0; k < nBlanks; k++)
				System.out.print(' ');

		System.out.print((heap[j] == null)? "" : heap[j]);

		if (++column == itemsPerRow) {
			nBlanks /= 2;
			itemsPerRow *= 2;
			column = 0;
			System.out.println();
		}
		else
			for (int k = 0; k < nBlanks * 2 - 2; k++)
				System.out.print(' ');

		j++;
	}
	System.out.println("\n" + dots + dots);
}
```

Test your code by running the **Runner.java** file (provided, in the lab folder). Your output should match the contents of the "**output.txt**" file (also in the lab folder).

Next, add a varargs constructor (e.g. `MinHeap(Integer... nums)`) that will build a heap when some initial elements are known in advance. Initially test with the elements 23, 5, 2, 7, 456, 6, 88 but it should work for any test case with varying number of elements and values.

Rather than just doing N inserts (which could require N log N operations), write a method `buildHeap` that will build the heap with N initial elements in O(N) time (see the powerpoint for more info). Test that your `buildHeap` method is faster than simply calling `insert` for a large number of items. **Note:** The varargs parameter is (or is converted to) an array, so an array can be supplied as well.

Finally, make a copy of your MinHeap class, call it MinHeapGeneric. Make the changes to allow your heap to store any type of Comparable data (rather than just `int`s). Use the following code to make a "generic array":

```
public class MinHeapGeneric<T extends Comparable<T>> //must have Comparable elements
{
	private T[] heap;

	//other variables not shown

	public MinHeapGeneric()
	{
		//cast required, can't make a T[] directly
		//  as it's not known at
		this.heap = (T[]) new Comparable[DEFAULT_CAPACITY];
	}
}
```

(Advanced) Min-Max heap

You've seen heaps that can be min *or* max heaps.

Design a data structure that supports min *and* max in constant time, and insert, delete min, and delete max in logarithmic time (O(log n)) by putting the items in a single array of size *n* with the following properties:

- The array represents a complete binary tree.
- The key in a node at an even level is less than (or equal to) the keys in its subtree; the key in a node at an odd level is greater than (or equal to) the keys in its subtree.

Note that the maximum value is stored at the root and the minimum value is stored at one of the root's children.

(Over 9000) Fibonacci heap

There are actually quite a few heap implementations used for priority queues. One very interesting implementation is called a Fibonacci heap, which offers a handful of benefits over regular array-based heaps. Research and implement a Fibonacci heap class. A tutorial, for the recklessly curious:

cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf