G52ADS 2014-15. Coursework THREE: "Implementation and Analysis of Heaps"
Part A. Algorithm and Implementation.

D-ary heap with d = 4, accounting for variable ROOT_INEX

```
private int d = 4;

// -------------- HEAP D: INSERT AND UPHEAP -- START ----------
this.heap[size + ROOT_INDEX] = value;
int currentNode = ++size;
while(ROOT_INDEX < currentNode) {
      int parentNode = getParentIndex(currentNode);
      if(heap[currentNode] < heap[parentNode]) {
            int temp = heap[currentNode];  // swap
            heap[currentNode] = heap[parentNode];
            heap[parentNode] = temp;
            currentNode = parentNode; // move to the parent node
      }
      else { //heap is already sorted, and can finish
            break;
      }
}
// -------------- HEAP D: INSERT AND UPHEAP -- END ----------

// -------------- HEAP D: REMOVE AND DOWNHEAP -- START ----------
heap[ROOT_INDEX] = heap[ROOT_INDEX + --size]; // effectively removes
'm' the entry that was swapped to the root
int currentNode = ROOT_INDEX; // starting position
while(kthChild(currentNode, 1) <= size) {
      //get the smallest child
      int smallestChild = minChild(currentNode);
      //swap if smaller than current
      if(heap[currentNode] > heap[smallestChild]) {
            int temp = heap[currentNode];
            heap[currentNode] = heap[smallestChild];
            heap[smallestChild] = temp;
            currentNode = smallestChild;
      }
      else { //already sorted and can finish
            break;
      }
}
// -------------- HEAP D: REMOVE AND DOWNHEAP -- END ----------

private int getParentIndex(int i) {
      return (i - 1 - ROOT_INDEX) / d + ROOT_INDEX;
}
private int kthChild(int i, int k) {
      return d * (i - ROOT_INDEX) + k + ROOT_INDEX;
}
private int minChild(int ind) {
      int bestChild = kthChild(ind, 1);
      int k = 2;
      int pos = kthChild(ind, k);
      while ((k <= d) && (pos < size + ROOT_INDEX)) {
            if (heap[pos] < heap[bestChild])
                  bestChild = pos;
            pos = kthChild(ind, ++k);
      }
      return bestChild;
}
```

Explanations and comments

Initializing d = 4

Add new value as the last element of the heap; set currentNode to this element; do ordering until we get to the root node or the heap is properly ordered (break); if the parent node's value is bigger, swap them.

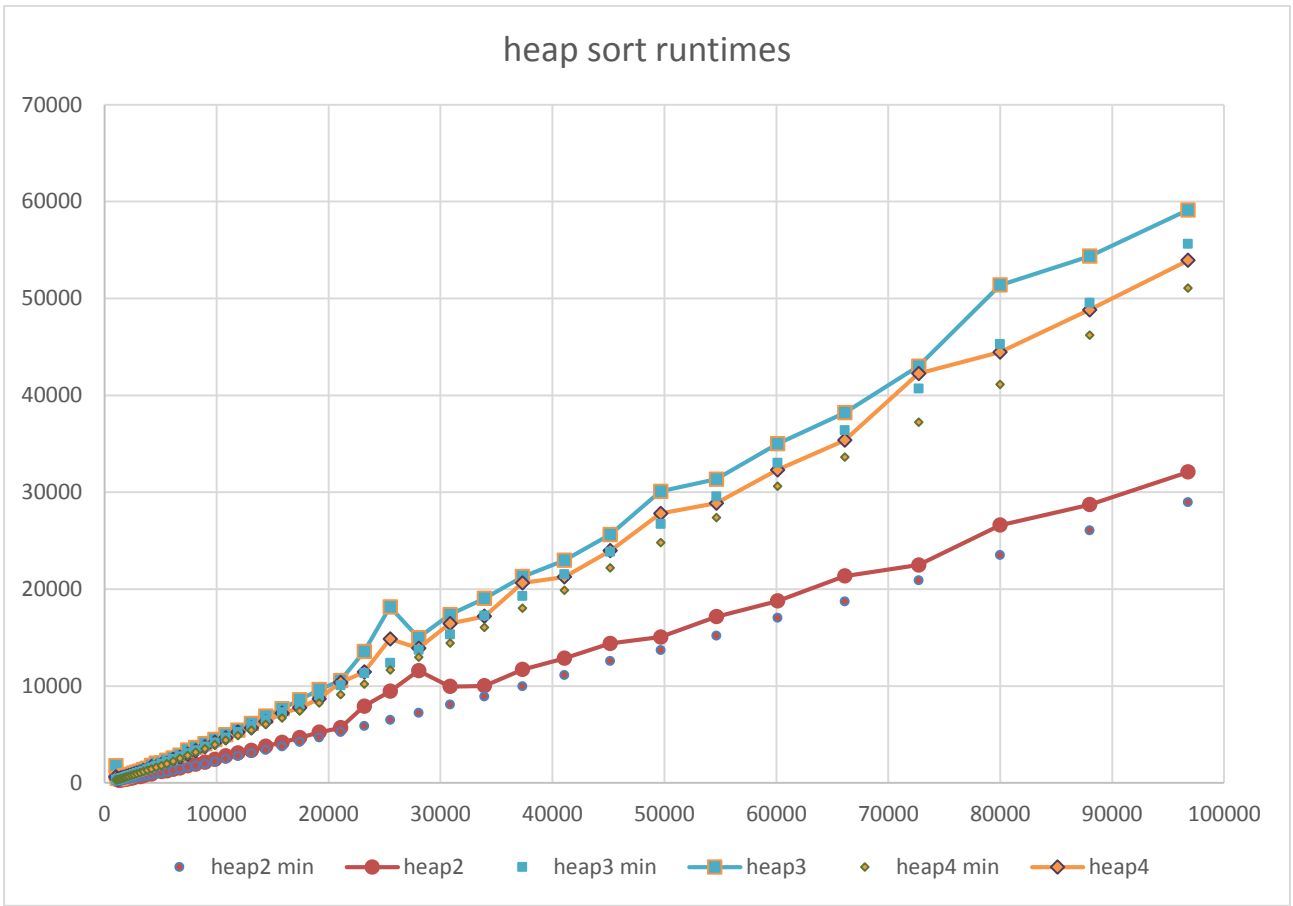
Swap the root node and the last one; set currentNode as the root node; start ordering if it has a child node; swap the two nodes if the smallest child is smaller than the current one.

Get parent index accounting to variable ROOT_INDEX.
Get the k-th child of a node, k = 1 being the leftmost.
Get the smallest child of a node; initialize bestChild with the leftmost child, and pos with the second; then iterate through all the children to find the one with the smallest value.

In my implementation I chose to use inheritance, as in the past 3 semesters we were taught to reuse code as much as possible. I also chose to adopt the code to account for a variable ROOT_INDEX, although this wasn't required it seemed to me that otherwise it would give the false sense that it is variable. Also in many places in the given code it wasn't clear that the calculation was different because the indexing starts at 1 – especially in the index getter methods – which could lead to unforeseen bugs. This might not be the best choice for performance, as adding and subtracting ROOT_INDEX many times is computationally expensive, but it gives a more dynamic, reusable and clear code.

The heap property only requires that all the parent nodes are smaller than the child and all children are in the leftmost position. Insertion to heap is done by inserting the element to the end of the heap and then doing an "upheap", such that it restores the heap property that might have been broken by inserting this element. This works in a way that the element is swapped with its parent until the parent is smaller than the child. This is correct because the rest of the heap wasn't touched so that part of it is correct and since the element is inserted in a leftmost position the latter requirement holds too. Removing works in a way that the first element is replaced by the last in the heap – effectively removing it – and then doing a "downheap" which restores the broken heap property. It is done by swapping the new root node with the smallest child of it until the swapped node is the smaller than all of its children. This is correct for it follows the definition of a heap – that all children must be bigger than their parent.

Part B. Main Graph



Part C. Analysis and Interpretation

On the graph I only added the averages and minimums for each heap sort, the reason why I omitted the maximum runtimes is because it wildly varies for all implementations, therefore making it useless and it would only make the graph hard to read.

By only taking a glimpse at the graph we can conclude that the most efficient algorithm is sorting with a binary heap. This has many reasons. First and foremost the given implementation is highly optimized as the index getters use bit shifts instead of costly divides and multiplies. Secondly when getting the smallest child in binary heaps it only has to do two comparison instead of three and four for 3-ary and 4-ary heaps respectively.

It is apparent too that heap4 is faster than heap3, as the minimum time for heap3 in many cases is worse than the average for heap4. I believe that this is caused by the getParentIndex() method as it does a division, for 3 it's an integer division which takes more than one primitive operation, but for 4 it's implicitly converted to a bit shift at compile time.

Also heap 4 is slower than the provided heap 2, as it is not well optimized for a specific heap size. Also it has to do four comparisons in the worst case when finding the smallest child for downheap, compared to two in binary heaps.

Other than these the graph has some anomalies too. As the data is the same for each run I believe that these are either caused by the JVM – making computation longer when the garbage collector kicks in – or by other background processes taking up processor time on my machine.

For testing I just modified the provided JUnit tests for heap3 and 4. Although I'm pretty sure it's not an exhaustive test suite I couldn't think of any other test cases. I also implemented a toString() method that pretty prints the heap making it easier to visualize the heap.

In the end we can say that all of the algorithms run close to linear time but in reality it is linearithmic time. Reason being is that inserting to a heap and doing an upheap is done in O(log n), then the smallest element can be found in O(1), which is followed by a downheap with a runtime of O(log(n)) and getting the smallest node is done n times adding up to O(n log(n)). This is pretty good for a sorting algorithm as it's a lot better than $O(n^2)$ that is the worst case for many algorithms – like bubble or insertion sort. Also this algorithm doesn't have a bottleneck like a basic quick sort when it runs in $O(n^2)$, for inserting to a heap has the same O(log(n)) runtime on arbitrary data. But this also means that it doesn't have a best case where it could run in O(n) time making it Ω(n log(n)). But since O(n) is close to O(n log(n)) and in some cases this outperforms O(n), we can conclude that heapsort is an efficient algorithm bound by O(n log(n)) and Ω(n log(n)) therefore making it run approximately in Θ(n log(n)). Also this algorithm is very compelling for uses where memory is a top priority as it can be implemented so that it does an in-place sorting taking up O(1) space.