Adam vanWestrienen
CSCI 313-36

Project 1 — Testing Priority Queues

Overview

In this project we analyze three different implementations of priority queues using a double linked list, a dynamic array, and a binary heap. Priority queues are a convenient abstract data structure in that they allow us to simulate the collection and processing of data much like we might find in the real world; by priority rather than next in line. For example, the way a hospital emergency room works. A bullet wound in the stomach is more life-threatening than a sore throat, and therefore the priority of whom the doctors see must be adjusted accordingly. Because the doctors must treat people by the severity of the emergency rather than first come/first serve, a priority queue would meet the requirements as an abstract data structure.

The idea for this project is to benchmark each implementation using the same interface and to test how quickly each handles three different operations: inserting a task, processing and returning the task with the highest priority, and then a combination of both.

Tasks for our analysis were randomly generated (with randomized priorities) and run at intervals of 10,000.

Complexity Analysis

Doubly-Linked List-based Priority Queue

Enqueue: Insertion using a doubly-linked list uses a head node to keep track of the maxPQ task. Therefore every new element that is inserted must see where it fits in the priority link meaning that if n items are inserted, i'm guessing the runtime should be O(n).

Dequeue: Implementing a maxPQ using the head as the largest priority task means that the runtime is just O(1), all the linked-list has to do is return.

Array-based Priority Queue

Enqueue: Inserting a task using a dynamic array (unordered) is O(1) because we are just pushing each task object onto the end of the array. Using a grow function to increase the capacity of the array is used when needed and does not execute every operation. Therefore, pushing n tasks should take O(n) time.

Dequeue: Finding the highest priority in the queue becomes problematic when we've just inserted all the items in an unordered fashion. What you've gained in speed from the insertion, you've lost on the removal as the array needs to be sorted before it can find the max. Here I'm guessing the runtime will be higher than O(n).

Enqueue With the binary heap implementation I'm expecting the runtime to be logarithmic because our array behaves as a tree, meaning we can keep track of our task priorities in more efficient way without having to run through the whole array over and over.

Dequeue: Same as above for removal. We don't have to compare every element, we only have to know where the children are and then act accordingly.

Code

PriorityQueue.java

```
public interface PriorityQueue<T> extends Comparable<T> {
	void enqueue(T item);
	T dequeue();
	T peek();
	int size();
	boolean isEmpty();
}
// end of PriorityQueue.java
```

Task.java

```
public class Task implements Comparable<Task> {
	public int id;
	public int priority;

	public Task(int id, int priority) {
		this.id = id;
		this.priority = priority;
	}
	@Override
	public int compareTo(Task other) {
		return this.priority - other.priority;
	}
	@Override
	public String toString() {
		return "Task [id:" + id + ", priority:" + priority + "]";
	}
	public int getId() {
		return id;
	}
	public int getPriority() {
		return priority;
	}
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + id;
		result = prime * result + priority;
```

```
                return result;
        }
        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                Task other = (Task) obj;
                if (id != other.id)
                        return false;
                if (priority != other.priority)
                        return false;
                return true;
        }
}
// end of Task.java
```

Node.java

```
public class Node<T> {
        private T item;
        private Node<T> next;
        private Node<T> prev;

        public Node(T value) {
                this.item = value;
        }
        public T getItem() {
                return item;
        }
        public void setItem(T item) {
                this.item = item;
        }

        public Node<T> getNext() {
                return next;
        }
        public void setNext(Node<T> next) {
                this.next = next;
        }
        public Node<T> getPrev() {
                return prev;
        }
        public void setPrev(Node<T> prev) {
                this.prev = prev;
        }
        @Override
        public String toString() {
```

```
                return "Node [item:" + item + ", next:" + next + ", prev:" + prev + "]";
        }
}
// end of Node.java
```

LinkedList.java

```
public class LinkedList<T> implements PriorityQueue<T> {

        private Node<T> head;
        private int size;
        public LinkedList() {
                head = null;
                size = 0;
        }
        @SuppressWarnings("unchecked")
        public void enqueue(T item) {
                Node<T> newNode = new Node<T>(item);
                // If list is empty, just add the node as the head.
                if (size == 0) {
                        head = newNode;
                        size++;
                        return;
                }
                // Find the location to add the node.
                else {
                        Node<T> current = head;
                        // The new node is lower priority than the current node.
                        int diff = ((Comparable<T>) newNode.getItem()).compareTo((T)current.getItem());
                        if (diff < 0) {
                                // While there is a next node in the list, and the new node is less than the
                                next node, keep searching.
                                // If the next node is null, then add the new node to the end of the list.
                                // If the new node is greater than the next, then insert here.
                                while ((current.getNext() != null) && (((Comparable<T>)
                                newNode.getItem()).compareTo((T) current.getNext().getItem())< 0)) {
                                        current = current.getNext();
                                }
                                Node<T> next = current.getNext();
                                // If we reached the end of the list, then add the new node to the end.
                                if ( next == null ) {
                                        current.setNext(newNode);
                                        newNode.setPrev(current);
                                }
                                // Insert after the current.
                                else {
                                        newNode.setPrev(current);
                                        newNode.setNext(next);
                                        next.setPrev(newNode);
                                        current.setNext(newNode);
                                }
                        }
```

```
                        // The new node has a higher priority than the current/head node.
                        else if (diff >= 0) {
                                newNode.setNext(current);
                                current.setPrev(newNode);
                                head = newNode;
                        }
                        size++;
                }
        }

        public T dequeue() {
                // If the there are elements in the list, the head is the highest priority.
                if (head != null) {
                        Node<T> highest = head;
                        head = head.getNext();
                        size--;
                        return highest.getItem();
                }
                else {
                        return null;
                }
        }

        public T peek() {
                return head.getItem();
        }
        public int size() {
                return size;
        }
        public boolean isEmpty() {
                return size > 0;
        }
        @Override
        public int compareTo(T o) {
                return 0;
        }
}
// end of LinkedList.java
```

DynamicArray.java

```
import java.lang.reflect.Array;
import java.util.Arrays;

public class DynamicArray<T> implements PriorityQueue<T> {
        private T[] pQueue;
        private int size;
        private int next;
        private boolean dirty;
        private Class<T> classType;
```

```
@SuppressWarnings("unchecked")
public DynamicArray(Class<T> classType, int capacity) {
	pQueue = (T[])Array.newInstance(classType, capacity);
	this.classType = classType;
	this.size = capacity;
	this.next = 0;
}

@SuppressWarnings("unchecked")
public void grow() {
	// Create a new array and copy contents of old array into it.
	T[] newPQ = (T[])Array.newInstance(classType, size * 2);
	for (int i = 0; i < size; i++) {
		newPQ[i] = pQueue[i];
	}
	pQueue = newPQ;
	this.size = this.size * 2;
}

public void enqueue(T item) {
	// Check if there is room. If not create a bigger array and add element to end.
	if (this.next >= pQueue.length - 1) grow();
	pQueue[this.next++] = item;
	// Note the array is not sorted now.
	dirty = true;
}

public T dequeue() {
	T max = null;
	// Sort the array if you've added elements.
	if (dirty) {
		Arrays.sort(pQueue, 0, next);
	}
	if ( next > 0 ) {
		max = pQueue[next - 1];
		next--;
	}

	return max;
}
public void printQueue() {
	System.out.println("Queue: ");
	for ( int i = 0; i < next; i++ ) {
		System.out.println(pQueue[i]);
	}
}
@Override
public int compareTo(T o) {
	return 0;
}
```

```
	public T[] getpQueue() {
		return pQueue;
	}

	public int getSize() {
		return size;
	}

	public boolean isDirty() {
		return dirty;
	}

	public Class<T> getClassType() {
		return classType;
	}

	@Override
	public T peek() {
		return null;
	}

	@Override
	public int size() {
		return 0;
	}

	@Override
	public boolean isEmpty() {
		return false;
	}
}
// end of DynamicArray.java
```

Heap.java

```
import java.lang.reflect.Array;

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

	private T[] arr;

	private int size;

	private Class<T> classType;


	@SuppressWarnings("unchecked")

	public Heap(Class<T> classType, int capacity) {

		this.arr = (T[])Array.newInstance(classType, capacity + 1);

		this.size = 0;

		this.classType = classType;

	}
```

```
@SuppressWarnings("unchecked")
private void grow() {
	// Create a new array of double cap and copy the contents into it.
	T[] temp = (T[])Array.newInstance(classType, this.arr.length * 2);
	for ( int i = 1; i < this.arr.length; i++ ) {
		temp[i] = arr[i];
	}
	this.arr = temp;
}

public int leftChild(int current) {
	return current*2;
}

public int rightChild(int current) {
	return (current*2) + 1;
}

public int parentIdx(int current) {
	return current/2;
}

public boolean hasLeftChild(int current) {
	return leftChild(current) <= size;
}

public boolean hasRightChild(int current) {
	return rightChild(current) <= size;
}

public boolean hasParent(int current) {
	return current > 1;
}
```

```
public void swap(int i1, int i2) {
        // Exchange the elements
        T temp = arr[i1];
        arr[i1] = arr[i2];
        arr[i2] = temp;
}
public void enqueue(T item) {
        // Resize if out of room.
        if (size == arr.length - 1) {
                grow();
        }
        // Add to the end of the array and bubbleUp.
        arr[++size] = item;
        bubbleUp(size);
}
public T dequeue() {
        // Set the max to the front of the array
        T max = peek();
        if ( size > 1) {
                // Put element at back in front and decrease size.
                arr[1] = arr[size];
                size--;
        }
        // Rearrange the elements
        bubbleDown();
        return max;
}
@SuppressWarnings("unchecked")
public void bubbleUp(int index) {
        index = size;
        while (hasParent(index) && (((Comparable<T>) arr[parentIdx(index)]).compareTo(arr[index]) < 0))
        {
                swap(index, parentIdx(index));
                index = parentIdx(index);
        }
}

@SuppressWarnings("unchecked")
```

```
public void bubbleDown() {
        int index = 1;
        while (hasLeftChild(index)) {
                // See which of children is smaller.
                int child = leftChild(index);
                // Bubble with the smaller child, if I have a smaller child
                if (hasRightChild(index)&& ((Comparable<T>)
                    arr[leftChild(index)]).compareTo(arr[rightChild(index)]) < 0) {
                        child = rightChild(index);
                }
                if (((Comparable<T>) arr[index]).compareTo(arr[child]) < 0) {
                        swap(index, child);
                } else {
                        break;
                }
                // Update loop counter/index of where last element is put.
                index = child;
        }
}
public void printList() {
        System.out.println("\nArray size = " + size);
    for (int i = 1; i < size + 1; i++ ) {
        System.out.println("Heap: " + arr[i]);
    }
  }
@Override
public T peek() {
        return arr[1];
}

@Override
public boolean isEmpty() {
        return size == 0;
}
@Override
public int compareTo(T o) {
        return 0;
}
```

```
	@Override
	public int size() {
		return size;
	}
	@Override
	public int size() {
		return size;
	}
}
// end of Heap.Java
```

Main.java

```
import java.util.ArrayList;
import java.io.PrintWriter;
import java.util.List;
import java.util.Random;
import java.io.FileNotFoundException;

public class Main {
	private static final int taskCnt = 90000;
	private static List<Task> tasks = new ArrayList<Task>();
	private static Random ran = new Random();
	private static long llEnTime;
	private static long arrEnTime;
	private static long heapEnTime;
	private static long llDeqTime;
	private static long arrDeqTime;
	private static long heapDeqTime;
	private static long llEnDeqTime;
	private static long arrEnDeqTime;
	private static long heapEnDeqTime;

	public static void main(String[] args) {

		long llEnTotal = 0;
		long llDeqTotal = 0;
		long llEnDeqTotal = 0;
		long arrEnTotal = 0;
```

```
long arrDeqTotal = 0;
long arrEnDeqTotal = 0;
long heapEnTotal = 0;
long heapDeqTotal = 0;
long heapEnDeqTotal = 0;

int testRun = 1;
while (testRun <= 10) {
        System.out.println("TestRun: " + testRun);

for (int i = 0; i < taskCnt; i++) {
        tasks.add(new Task(i, ran.nextInt(10000000)));
}

System.out.println("\nLinkedList: ");
testSorting(new LinkedList<Task>(), tasks, 20);
System.out.println("\nDynamicArray: ");
testSorting(new DynamicArray<Task>(Task.class, 25), tasks, 20);
System.out.println("\nHeapQueue: ");
testSorting(new Heap<Task>(Task.class, 25), tasks, 20);
System.out.println("\n ");


timeLinkedList(tasks);
llEnTotal += llEnTime;
llDeqTotal += llDeqTime;
llEnDeqTotal += llEnDeqTime;

timeDynamicArray(tasks);
arrEnTotal += arrEnTime;
arrDeqTotal += arrDeqTime;
arrEnDeqTotal += arrEnDeqTime;

timeHeapQueue(tasks);
heapEnTotal += heapEnTime;
heapDeqTotal += heapDeqTime;
heapEnDeqTotal += heapEnDeqTime;
```

```
		testRun++;
		}
		System.out.println("LinkList Averages for " + taskCnt + " items");
		System.out.println("---------------------------------------");
		System.out.println("Avg. Enqueue time: " + averageTime(llEnTotal, 10));
		System.out.println("Avg. Dequeue time: " + averageTime(llDeqTotal, 10));
		System.out.println("Avg. Enqueue/Dequeue time: " + averageTime(llEnDeqTotal, 10));
		System.out.println(" ");
		System.out.println("DynamicArray Averages for " + taskCnt + " items");
		System.out.println("---------------------------------------");
		System.out.println("Avg. Enqueue time: " + averageTime(arrEnTotal, 10));
		System.out.println("Avg. Dequeue time: " + averageTime(arrDeqTotal, 10));
		System.out.println("Avg. Enqueue/Dequeue time: " + averageTime(arrEnDeqTotal, 10));
		System.out.println(" ");
		System.out.println("Heap Averages for " + taskCnt + " items");
		System.out.println("---------------------------------------");
		System.out.println("Avg. Enqueue time: " + averageTime(heapEnTotal, 10));
		System.out.println("Avg. Dequeue time: " + averageTime(heapDeqTotal, 10));
		System.out.println("Avg. Enqueue/Dequeue time: " + averageTime(heapEnDeqTotal, 10));
		System.out.println(" ");

}

@SuppressWarnings("unchecked")
private static <T> void testSorting(PriorityQueue<T> queue, List<Task> tasks, int cnt) {
		for (int i = 0; i < cnt; i++ ) {
			queue.enqueue((T) tasks.get(i));
		}
		for (int i = 0; i < cnt; i++ ) {
			System.out.println(queue.dequeue());
		}
}

private static void timeLinkedList(List<Task> tasks) {
		System.out.println("Test LinkedList");
		LinkedList<Task> linkedList = new LinkedList<>();
		long t = System.currentTimeMillis();
		for (int i = 0; i < taskCnt; i++) {
```

```
                linkedList.enqueue(tasks.get(i));
        }
        llEnTime = System.currentTimeMillis() - t;
        System.out.println("Enqueue " + taskCnt + " items: " + llEnTime + " milliseconds");

        t = System.currentTimeMillis();
        for (int i = 0; i < taskCnt; i++) {
                linkedList.dequeue();
        }
        llDeqTime = System.currentTimeMillis() - t;
        System.out.println("Dequeue " + taskCnt + " items: " + llDeqTime + " milliseconds");

        linkedList = new LinkedList<>();
        t = System.currentTimeMillis();
        int j = 0;
        while ( j < tasks.size() ) {
                if ( ran.nextInt() % 2 == 0) {
                        linkedList.enqueue(tasks.get(j));
                        j++;
                }
                else {
                        linkedList.dequeue();
                }
        }
        llEnDeqTime = System.currentTimeMillis() - t;
        System.out.println("Enqueue/Dequeue " + taskCnt + " items: " + llEnDeqTime + " milliseconds\n");
}

private static void timeDynamicArray(List<Task> tasks) {
        System.out.println("Test DynamicArray");
        DynamicArray<Task> dynArray =  new DynamicArray<>(Task.class, tasks.size() + 1);
        long t = System.currentTimeMillis();
        for (int i = 0; i < taskCnt; i++) {
                dynArray.enqueue(tasks.get(i));
        }
        arrEnTime = System.currentTimeMillis() - t;
        System.out.println("Enqueue " + taskCnt + " items: " + arrEnTime + " milliseconds");
```

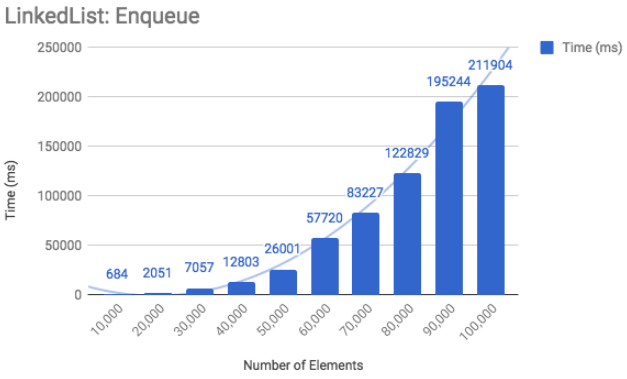
```
		for (int i = 0; i < taskCnt; i++) {
			dynArray.dequeue();
		}
		arrDeqTime = System.currentTimeMillis() - t;
		System.out.println("Dequeue " + taskCnt + " items: " + arrDeqTime + " milliseconds");

		dynArray = new DynamicArray<>(Task.class, tasks.size() + 1);
		t = System.currentTimeMillis();
		int j = 0;
		while ( j < tasks.size() ) {
			if ( ran.nextInt() % 2 == 0) {
				dynArray.enqueue(tasks.get(j));
				j++;
			}
			else {
				dynArray.dequeue();
			}
		}
		arrEnDeqTime = System.currentTimeMillis() - t;
		System.out.println("Enqueue/Dequeue " + taskCnt + " items: " + arrEnDeqTime + " milliseconds\n");

}

private static void timeHeapQueue(List<Task> tasks) {
		System.out.println("Test HeapQueue");
		Heap<Task> heap =  new Heap<>(Task.class, tasks.size() + 1);
		long t = System.currentTimeMillis();
		for (int i = 0; i < taskCnt; i++) {
			heap.enqueue(tasks.get(i));
		}
		heapEnTime = System.currentTimeMillis() - t;
		System.out.println("Enqueue " + taskCnt + " items: " + heapEnTime + " milliseconds");

		t = System.currentTimeMillis();
		for (int i = 0; i < taskCnt; i++) {
			heap.dequeue();
		}
		heapDeqTime = System.currentTimeMillis() - t;
```

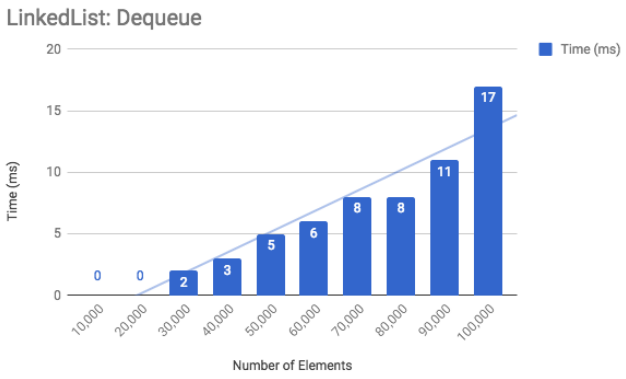
```
            System.out.println("Dequeue " + taskCnt + " items: " + heapDeqTime + " milliseconds");


            heap =  new Heap<>(Task.class, tasks.size() + 1);
            t = System.currentTimeMillis();
            int j = 0;
            while ( j < tasks.size() ) {
                  if ( ran.nextInt() % 2 == 0) {
                        heap.enqueue(tasks.get(j));
                        j++;
                  }
                  else {
                        heap.dequeue();
                  }
            }
            heapEnDeqTime = System.currentTimeMillis() - t;
            System.out.println("Enqueue/Dequeue " + taskCnt + " items: " + heapEnDeqTime + " milliseconds\n");


      }
      public static int averageTime(long totalTime, int numRuns) {
            int average = (int)totalTime/numRuns;
            return average;
      }
```

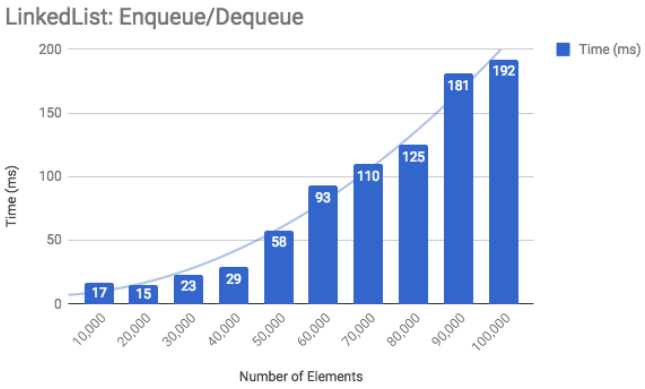
Benchmark Results

LinkedList

Looking at the performance charts below we can see that the linked list implementation really excels on the removal (dequeue) of our tasks. This is what we expected as we already know where the max priority is. As you can see, however, all the work is done on the insertion (enqueue). It's painfully slow compared to the dynamic array and heap implementations as it finds where each task needs to be inserted. Therefore we have a polynomial or $O(n^2)$ for the enqueue and linear O(n) for the dequeue. Enqueue/Dequeue was also O(n).

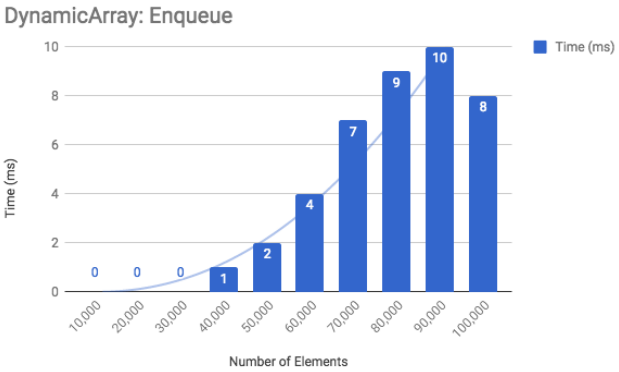


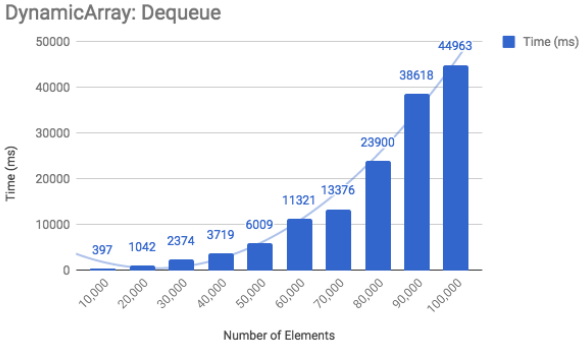


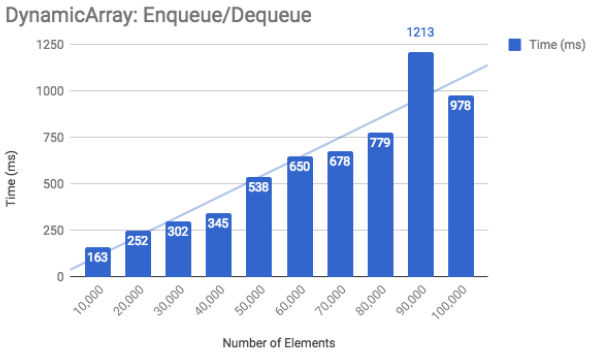


Dynamic Array

When we look at the dynamic array's performance we get the direct opposite of what we see with the linked list. The array implementation is quick to add (enqueue) tasks since we're just loading them on the end without concern for the priority rank of each task. Here we see O(1) for single insertion and O(n) otherwise. We get the contrary when we dequeue items due to the need to sort all the elements before removing the highest priority. Like enqueue for linked list, our array dequeue has a polynomial $O(n^2)$ trend line. Here, like the linked list Enqueue/Dequeue performed linearly like the linked list.

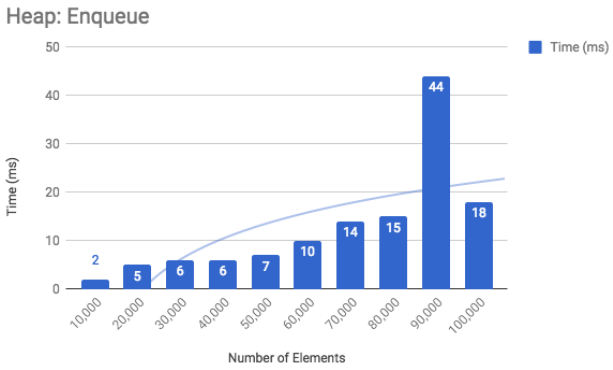


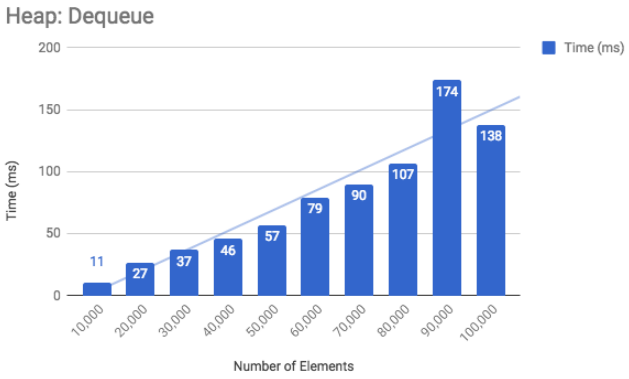


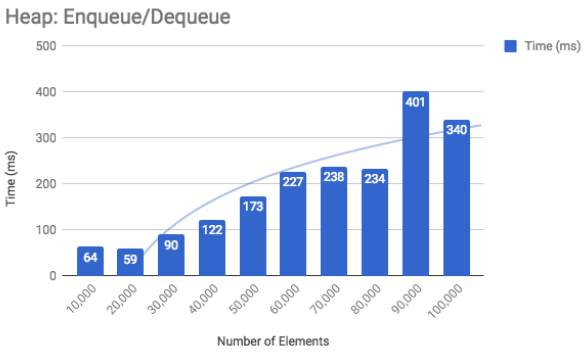


Heap

As we can see from the graph analysis, the heap performed much better on average than the linked list and array at log(n) (or close to it) across the three methods. There seems to be one anomaly at the ninth data point (90,000 tasks). Otherwise the results look consistent throughout.







Conclusion

Looking at the runtime analysis as a whole, we see that the data for all implementations is consistent, meaning we can see clear trend lines and there is not any drastic fluctuations. We also have an overall undisputed “winner” with the binary heap as it performed really well across all methods, whereas the linked list and dynamic array had obvious drawbacks with inserting and removing elements respectively.