Priority Queues (Heaps)

Dhruba Das

Spring Fall 2023

Introduction

This assignment was a bit more challenging than I expected. There were 3 main problems to solve: implementing and benchmarking two types of linked list heaps, implementing and benchmarking a binary tree heap, and implementing and benchmarking an array heap. In this assignment, we assume that smaller numbers have higher priority.

HeapsList1 vs HeapsList2

Here, HeapsList1 has methods add with O(n) and remove with O(1). HeapsList2 has the same methods, but the time complexities for add and remove are swapped. I will only be highlighting the methods in this report, as the constructors are not as important.

Add and Remove for HeapsList1

The add method adds an item in the list according to it's priority; the list is created in an ascending order, with the number with the highest priority at the beginning. Below is the code snippet:

```
public void add(int value) {
        Node newNode = new Node(value);
        if (head == null || value < head.prio) {
            newNode.next = head;
            head = newNode;
        } else {
            Node cur = head;
            while (cur.next != null && cur.next.prio < value) {
                cur = cur.next;
            }
            newNode.next = cur.next;
            cur.next = newNode;
```

```
        }
    }
```

The remove method simply removes and returns the item with the highest priority. Below is the code snippet:

```
public int remove() {
        if (head == null) {
            System.out.println("Heap is empty");
            return -1;
        } else {
            int removedValue = head.prio;
            head = head.next;
            return removedValue;
        }
    }
```

Add and remove for HeapsList2

Here, we assume that the list is unsorted. So the add method just adds an item to the end of the list.

```
public void add(int value) {
        Node newNode = new Node(value);
        newNode.next = head;
        head = newNode;
    }
```

The remove method goes through the entire list to find the item with the highest priority, and then removes and returns that item. Below is the code snippet:

```
public int remove() {
        if (head == null) {
            System.out.println("Heap is empty");
            return -1;
        } else {
            Node highestPriorityNode = findHighestPriority();
            removeNode(highestPriorityNode); // O(n) time complexity
            return highestPriorityNode.prio;
        }
    }

public Node findHighestPriority() {
        Node highestPriorityNode = head;
```

```
        Node current = head;

        while (current != null) {
            if (current.prio < highestPriorityNode.prio) {
                highestPriorityNode = current;
            }
            current = current.next;
        }

        return highestPriorityNode;
    }
```

Benchmarks

Below is the difference in performance between the two implementations, represented in a table:

Size(n)	**HeapsList1 (ms)**	**HeapsList2 (ms)**
1000	1	1
2000	2	3
3000	4	7
4000	7	14
5000	11	21
6000	18	31
7000	26	42
8000	39	54
9000	55	70
10000	73	87

Table 1: (For each size, a thousand tries are done; min. time taken; time in milliseconds)

As we can see, HeapsList1 is faster than HeapsList2, but not by much.

Binary Tree Heap

This was not too difficult to implement, as we don't really care much about anything other than the root of the tree, which contains the item with the highest priority. Below I will be highlighting all implemented methods.

Add and enqueue

This method is similar to the add method in the linked list implementation, in the sense that it also adds an item to the tree according to it's priority.

The add method is inside the "Node" class.

```
private void add(int pr){
            if(pr < prio){
                Integer temp = prio;
                prio = pr;
                pr = temp;
            }
            if(right != null){
              if(left != null){
                if(left.size < right.size)
                  left.add(pr);
                else
                  right.add(pr);
              }
              else
                left = new Node(pr);
            }
            else
              right = new Node(pr);
        }
....
public void enqueue(int p){
        if(root == null)
          root = new Node(p);
        else
          root.add(p);
    }
```

Remove and dequeue

The remove and the dequeue method, together, remove the item with the highest priority, and returns the removed item. Below is the code snippet:

```
private Node remove(){
            if(right == null)
              return left;
            if(left == null)
              return right;
            if(left.prio < right.prio){
              prio = left.prio;
              left = left.remove();
            } else {
                prio = right.prio;
```

```
                right = right.remove();
            }
            return this;
        }
....
public int dequeue(){
        if(root == null)
          return -1;
        int p = root.prio;
        root = root.remove();
        return p;
    }
```

Push method

This method increments the root element by *incr* and then pushes it (by swapping values) either to the left or right branch. Below is the code snippet:

```
private int push(int incr) {
          int depth = 0; // What we want to return
          prio = prio + incr;

          if (left != null && prio > left.prio) {
              swap(this, left);
              depth++;
              depth += left.push(0);
          } else if (right != null && prio > right.prio) {
              swap(this, right);
              depth++;
              depth += right.push(0);
          }

          return depth;
        }
....
public int push(int incr) {
      int depth = 0;
      if (root != null) {
        depth = root.push(incr);
      }
      return depth;
    }
```

Benchmarks

Below are the benchmarks for comparison between removing and adding and pushing items on the tree.

Increment by:	Depth
10	53
20	66
40	96
80	130
100	140

Table 2: Depth describes how deep we need to traverse down the tree in order to push the item

We can see, as the incr in the push method increases, the depth also increases linearly.

Next, we compare the speed of Removing and Adding, against Pushing. Size represents the size of the binary tree heap.

Size(n)	Push (ms)	Remove and Add (ms)
100	9	16
200	18	37
400	45	62
800	110	137
1600	290	333

Table 3: (For each size, a thousand tries are done; min. time taken; time in microseconds)

Array Heap

Below I have described the methods sink and bubble used in the array implementation of the binary tree heap.

Bubble method

This method handles the adding of an item to the tree as a leaf; it then "bubbles" up the item to the correct depth, according to its priority. The resize method doubles the size of the array if the array is full.

```
public void bubble(int item) {
        if (size == heap.length) {
```

```
            resizeArray();
        }

        int currentIndex = size;
        heap[currentIndex] = item;
        size++;

        while (currentIndex > 0) {
            int parentIndex = (currentIndex - 1) / 2;
            if (heap[currentIndex] < heap[parentIndex]) {
                swap(currentIndex, parentIndex);
                currentIndex = parentIndex;
            } else {
                break;
            }
        }
    }
```

Sink method

This method handles the removing of the item with the highest priority, replacing the item with one of those with lowest priority (i.e a leaf) and then sinking it down to the right depth, recursively.

```
public int sink() {
        if (size == 0) {
            throw new IllegalStateException("Heap is empty");
        }

        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;

        sinkDown(0);

        return root;
    }
....
private void sinkDown(int index) {
        int left = 2 * index + 1;
        int right = 2 * index + 2;
        int smallest = index;

        if (left < size && heap[left] < heap[smallest]) {
```

```
            smallest = left;
           }

        if (right < size && heap[right] < heap[smallest]) {
            smallest = right;
        }

        if (smallest != index) {
            swap(index, smallest);
            sinkDown(smallest);
        }
    }
```

Benchmarks

Here, we compare the linked list implementation of the binary heap to that of the array implementation of the binary heap. We compare the time needed to bubble and sink, against the time needed to enqueue(Add) and dequeue(Remove).

Size(n)	**Bubble and Sink (ms)**
100	10
200	25
400	50
800	123
1600	277

Table 4: (For each size, a thousand tries are done; min. time taken; time in microseconds)

And as we can see, the array implementation is slightly faster than the linked list implementation. But both have the same time complexities, as both are binary tree heaps.