Priority Queues

Axel Månson Lokrantz

Spring Fall 2022

Introduction

A priority queue is an abstract data-type much like a stack or a regular queue. It is often used in data structures such as linked lists or trees. A tree structure with a priority queue is called heap. Items with high priority are closer to the root and removal of elements should always return the element with the highest priority. The following program was written in Java together with Daniel Dahlberg at KTH.

Task 1: Priority queue in a linked list

Task one was to make two implementations of a priority queue in a singly linked list. The first implementation should have an add method with the time complexity of $O(1)$ and a remove method with time complexity of $O(n)$. The linked list should hold n number of integers where small numbers have high priority and be placed near the head.

```
public void add(int value){
          Node temp = new Node(value);

          if (head == null)
              head = temp;
          else{
              temp.next = head;
              head = temp;
          }
      }
```

Add method in a singly linked list with time complexity O(1).

Since we are adding nodes to the head there is no need to traverse the list and therefor the time complexity will be $O(1)$. As for our remove method it needs to return the node with the highest priority. To do this we mimic a doubly list, making use of a previous pointer.

```
  public void add(int value){
      public Node priorityRemove(){
            Node smallest = head;
            Node temp = head;
            Node previous = null;
            /*
             * Traverse the list. Find the smallest value, assign smallest to it.
             */
             while(temp != null){
                 if (temp.next != null && temp.next.value < smallest.value){
                     smallest = temp.next;
                     previous = temp;
             }
             temp = temp.next;
             }

             /*
             * If smallest element is not head, we remove smallest.
             * If smallest element is head, we remove head.
             */
            if(smallest != head)
            previous.next = smallest.next;
            else
            head = head.next;

            return smallest;
    }
```

To find the item with highest priority we need to traverse the list and do n number of comparisons. The time complexity will therefor be $O(n)$. In the second implementation the remove method should have the time complexity of $O(1)$ and the add method $O(n)$.

```
public Node remove(){

        if (head == null){
            return null;
        }
        Node temp = head;
        head = head.next;
        return temp;
    }
```

Similarly to the previous implementation we now remove from the head resulting in a time complexity of $O(1)$. The add method should insert items in priority order. Since we, in this program, know that all elements are in the correct priority order all we have to do is to compare the priority of the item we want to add with the rest of the elements. When an item is found with a lower priority than our item we insert the new item.

```
    public void priorityAdd(int value){

        Node temp = new Node (value);

        if (head == null || head.value >= value){
            temp.next = head;
            head = temp;
        }
        else{
            Node current = head;
            while(current.next != null && current.next.value < value){
                current = current.next;
            }
            temp.next = current.next;
            current.next = temp;
        }
    }
```

In a list where all elements are already sorted in priority order it is preferred to use the second implementation since it has better time complexities for best and average case, where the first implementation always performs $O(n)$. If you on the other hand is working with a list that will continuously grow and not much removal will be done then the first implementation is preferred.

n	priorityAdd	priorityRemove
10	50	60
100	250	150
1000	3900	1350
10000	74000	34500
100000	4600000	60000

Table 1: Measurements in ns, n refers to length of list.

Task 2: A Heap

There are two types of heaps, min heaps and max heaps. In this assignment we will work with a min heap. In a min heap, the root holds the item with the lowest value and highest priority, this is one of the strength of this datatype since finding it will have the time complexity of $O(1)$. Removing it however, requires some work. To keep our tree balanced we need to keep track of the total number of elements in each sub-tree and always add new elements to which ever branch that has the fewest.

Task 2.1: Add

```
public void add(Integer priority, Integer value){

            if(this.priority == priority){
                this.value = value;
                return;
            }
            if(priority < this.priority){
                Integer tempPriority = this.priority;
                Integer tempValue = this.value;
                this.priority = priority;
                this.value = value;
                priority = tempPriority;
                value = tempValue;
            }
            this.size++;
            if (this.left == null)
                this.left = new Node(priority, value, 1);
            else if(this.right == null)
                this.right = new Node(priority, value, 1);
            else if (this.left.size < this.right.size)
                this.left.add(priority, value);
            else
                this.right.add(priority, value);
        }
```

Lets call the element we want to add "a". If a has the same priority as an already existing element we simply update the elements value with a's value and return. If a's priority is less than the priority of the current element b, we swap a and b. Now we have to find a new place for b. First we increment the size of a since we are moving further down the tree. If there are no children to the left or right we make a new node and insert the values of b. If a has children either to the left or right we call the add method recursively and the same steps repeat until we find a leaf.

Task 2.2: Remove

```
public Node remove(){
    if(this.left == null){
        this.priority = this.right.priority;
        this.right = null;
        this.size--;
        return this;
    }
    if(this.right == null){
        this.priority = this.left.priority;
        this.left = null;
        this.size--;
        return this;
    }
    if (this.right.priority < this.left.priority){
        this.priority = this.right.priority;
        this.size--;

        if(this.right.size == 1)
            this.right = null;
        else
            this.right = this.right.remove();
            return this;
            }
    else{
        this.priority = this.left.priority;
        this.size--;
            if(this.left.size == 1)
                this.left = null;
            else
                this.left = this.left.remove();
        }
        return this;
}
```

We first check for children either to the left or right. If the node only has one child remove that child and promote it. If the node has two children we compare the priority of the two children, which ever has the highest priority we promote. This operation is done recursively until we reach a leaf.

Task 2.3: Increment

Sometimes when working with a heap you want to access the item with highest priority, perform some operation and then reinsert it with a lower priority. Of course this could be done with the above remove / add operations, but it is more time efficient to use a method that increments the priority rather than always inserting it at the bottom of the tree.

```
// Overrider push method with different param.
    public int push(int increment){
        this.root.priority += increment;
        Node temp = this.root;
        return push(this.root.priority, temp, 0);
    }

    private int push (Integer newPriority, Node temp, int depth){
        if (temp.left == null){
            if (newPriority < temp.right.priority)
                return depth;
            else {
                depth++;
                swap(temp.right, temp);
            }

            if (temp.right.size != 1)
                depth = push(newPriority, temp.right, depth);
        }

        if (temp.right == null){
            if(newPriority < temp.left.priority)
                return depth;
            else{
                depth++;
                swap(temp.left, temp);
            }
            if (temp.left.size != 1)
                depth = push(newPriority, temp.left, depth);
        }
        if (temp.right != null && temp.left.priority > temp.right.priority){
            if (temp.right.priority > newPriority)
                return depth;
            else {
                depth++;
                swap(temp.right, temp);
            }
            if (temp.right.size != 1)
                depth = push(newPriority, temp.right, depth);
        }
        else {
            if(temp.left.priority > newPriority)
                return depth;
            else{
                depth++;
                swap(temp.left, temp);
            }
            if (temp.left.size != 1)
                depth = push(newPriority, temp.left, depth);
        }
        return depth;
    }
```

The method takes an argument of the new priority and pushes the root through swap operations either to the left or the right heap. For bench marking we have to return the depth of which we travel, therefor a new variable is declared named depth. Every time we call the swap method we make sure to increment it.

Task 2.4: Benchmark

To compare the push and add methods we implemented a benchmark. The benchmark compares the depth of which the node travel during execution. For a tree structure with 64 elements with random values ranging up to 100 we got an average depth level of 4 for push and 8 for add. There are basically three scenarios for our push method, either we push somewhere at the top, middle or bottom. We know that most of the time we are going to end up with the average case which is depth 4 while add always has to traverse all the way down to depth 8 to reach a leaf.

Task 3.1: An Array Implementation

More commonly heaps are represented in arrays when working with priority queues. If a node is stored at index k, then its left child is stored at index $2k+1$, and its right child at index $2k+2$. The root is always stored at index 0. For even indexes the parent node can be found at index $k-2/2$ and odd indexes at $k-1/2$.

Task 3.2: Bubble

```
private void bubble(int priority) {
        queue[k++] = priority;
        int pivot = k-1;

        while(queue[pivot] < parent(pivot)) {
          swap(pivot, parentIndex(pivot));
          pivot = parentIndex(pivot);
        }
      }
```

To maintain the correct order of our heap we have to consider the priority when adding new elements. We set our pivot to k, k in this program is a pointer to the last element of the array. We then check if the priority of the leaf is higher than its parent. If this is the case we swap the child with its parent, and keep doing so until the node is at the correct index.

Task 3.2: Sink

Just as in the tree implementation removal is done at the root which corresponds to index 0. To rearrange the array we insert the element with the lowest priority (position k-1) at the root and let it sink to its right position.

```
public int remove() {
        int removedElement = queue[0];
        queue[0] = queue[--k];
        sink(0);
        return removedElement;
      }

      private void sink (int pivot) {
        if(!isLeaf(pivot)) {
          if(queue[pivot] > leftChild(pivot) ||
            queue[pivot] > rightChild(pivot)){

            if(leftChild(pivot) < rightChild(pivot)) {
              swap(pivot, leftChildIndex(pivot));
              sink(leftChildIndex(pivot));
            } else {
              swap(pivot, rightChildIndex(pivot));
              sink(rightChildIndex(pivot));
            }
          }
        }
```

As long as we have not reached a leaf we check for nodes with higher priorities than our pivot element. Any element with a higher priority is then swapped recursively until the array is rearranged into the desired order.

Task 3.3: Benchmark

n	Heap (Linked)	Heap (Array)
25	1200	1200
50	3000	2500
100	7000	6000
200	19000	16000
400	47000	30000

Table 2: Benchmark for a heap in a linked structure and an array. Measurements in ns, n refers to length of list.

When we insert or delete a node in our heap it is required that we restore the heap property by swapping a node with its parents recursively. The number of required operations depends on the number of levels the new element must sink. Therfore the time complexity becomes $(log(n))$.