Priority queues

Pontus Hector

Fall 2022

Introduction

This report is conducted as the eighth assignment in the course ID1021. In this report one explores different implementations of priority queues and comparing their benefits and drawbacks.

Priority queue using linked lists

In the first task of the assignment one is meant to implement two different queues using linked list in which the values stored are accessed in sequential order based on their numeric value, from smallest to largest. There is no way to implement a queue in which we want to retrieve the values based on a priority where both the add and remove functions are of constant time complexity, hence the two different implementations. In the first implementation the add function should be of constant time complexity and the remove function should be dependent on the amount of elements inside the queue. If one wants to implement an add function with constant time complexity, the method will have to add the elements at the first available slot. This implies that the stored elements will **not** be stored according to our priority and that the remove method need to contain an algorithm to remove the elements according to our priority.

```
public Integer remove() {
...
    while (temp != null) {
        if (temp.item < lowest_value) {
            lowest_value = temp.item;
            prev = previous;
            next = temp.tail;
        }
        previous = temp;
        temp = temp.tail;
    }
    if(prev == Head && Head.tail == null){
        Head = null;
    } else if(prev == Head && Head.tail != null){
        Head = Head.tail;
    } else if(next != null) {
        prev.tail = next;
    } else{
        prev.tail = null;
    }
    return lowest_value;
```

Iterations	**Nodes (n)**	**add**	**remove**
10000	100	2μs	15.4μs
10000	200	2.1μs	56.5μs
10000	400	2.54μs	169μs
10000	800	5.54μs	651μs
10000	1600	11.2μs	2660μs
10000	3200	25.6μs	10700μs

Table 1: Time to add/remove n amount of elements to/from the priority queue.

If one wants to implement a remove function with constant time complexity, the first available element of the queue must be the one of highest priority. This implies that the stored elements must be stored according to our priority and that the add method need to contain an algorithm to add the elements to the queue according to our priority.

```
public void add(Integer item) {
...
if(Head == null){
    Head = newNode;
} else if (item < Head.item){
    newNode.tail = Head;
    Head = newNode;
} else {
    while(temp != null && temp.item < item){
        if(temp.item > item){
            prev = previous;
            next = temp;
        }
        previous = temp;
        temp = temp.tail;
    }
    if(prev != null){
    prev.tail = newNode;
    newNode.tail = next;
```

```
        } else{
            previous.tail = newNode;
```

Iterations	Nodes (n)	add	remove
10000	100	4.25μs	0.24μs
10000	200	7.91μs	0.37μs
10000	400	15.2μs	0.87μs
10000	800	27μs	1.38μs
10000	1600	58.8μs	2.84μs
10000	3200	125μs	5.73μs

Table 2: Time to add/remove n amount of elements to/from the priority queue.

If one would need to add a considerable amount of elements to the priority queue and very seldomly remove any items from the queue the first implementation may be the optimal choice. However, for the general case of constructing a priority queue the second implementation is the one to prefer.

Implementation of heap using linked list

Another approach to implement priority queues is to make use of a binary tree. When implementing a priority queue using a binary tree one needs to consider the features of the priority queue since a binary tree in it self does not maintain the priority property of the stored items. This involves implementing around plenty of corner cases and changing the logical structure of adding and removing elements to the binary tree. If done properly one should achieve a priority queue in the form of a binary tree, well known as a heap.

Adding elements to the heap is done recursively (as in a binary tree) and if the heap is empty a root node is created with the given element. If the heap is not empty the given element needs to be injected in the correct spot and the elements further down the binary tree (from the injected node) needs to be advanced. To maintain the priority property the element to be added is compared to the root node and if the element to be added is of less priority than the root node they are swapped. The swapped value is then compared to the branches recursively up until a null node is within reach, where a new node is created with the least priority (according to the branches taken).

```
private Node addRecursive(Node current, Integer value) {
        if (current == null) {
            return new Node(value, 0);
        } else if (value < current.value) {
            temp = current.value;
            current.value = value;
            value = temp;
        }
        if (current.left == null) {
            current.subtree++;
            current.left = addRecursive(current.left, value);
            return current;
        } else if (current.right == null) {
            current.subtree++;
            current.right = addRecursive(current.right, value);
            return current;
        } else if (left == 1){
            current.subtree++;
            current.left = addRecursive(current.left, value);
            return current;
        } else if (left == 0){
            current.subtree++;
            current.right = addRecursive(current.right, value);
            return current;
        }
```

Removing an element from the heap is also done in a recursive way but rather unlike the add method. The root element (the element with the highest priority) is first retrieved from the node tree and then we recursively promote the branch with the higher priority up until a rearranged node tree has formed, with the same priority and the same elements **excluding** the element retrieved.

```
public Node removeRecursive(Node current){
        current.subtree--;
        if(current == null){
            return null;
        } else if(current.left == null){
            current = current.right;
            return current;
        } else if(current.right == null){
            current = current.left;
            return current;
        } else if(current.left.value > current.right.value){
            //promote right
            current.value = current.right.value;
            current.right = removeRecursive(current.right);
            return current;
        } else if(current.left.value < current.right.value){
            //promote left
            current.value = current.left.value;
            current.left = removeRecursive(current.left);
            return current;
        }
        return current;
```

Iterations	Nodes (n)	add	remove
1000000	100	10.3μs	1.17μs
1000000	200	29.9μs	2.09μs
1000000	400	95μs	3.73μs
1000000	800	383μs	8.87μs
1000000	1600	1947μs	19.1μs
1000000	3200	8900μs	40.3μs

Table 3: Time to add and remove n amount of elements to/from the heap.

A method to increment the element of highest priority to a lower priority is implemented as well since removing and then adding a value is a costly operation that could be worked around. This method is called **push** and starts off by retrieving the element of highest priority and increment it by a given value. This element is then injected at the root node and recursively compared to its branches and swapped with if any contain a value of higher priority, which is done up until its rightful position has been reached and the priority property of the heap has been restored.

Implementation of a heap using array

An alternatively, and perhaps a more efficient approach to a heap would be to implement it using an array instead of linked list. To maintain the data structure of a heap, all stored values have their branches at an offset from the current index at index *((index * 2) + 1)* and index *((index * 2) + 2)*. Added elements are first placed at the end of the heap and then continuously compared to the value of the parent index. If the compared value is of less priority than the value of its parent index, the values are swapped, which reoccurs up until the added element is in its correct position (according to the priority) and the priority property of the list has been restored.

```
public void add(Integer element){
        Heap[indexNext] = element;
        if(indexNext % 2 == 0) {
            indexPrev = (indexNext - 2)/2;
        } else{
            indexPrev = (indexNext - 1)/2;
        }
        if (indexNext != 0) {
            while (indexPrev >= 0 && Heap[indexNext1] < Heap[indexPrev]) {
                temp = Heap[indexNext1];
                Heap[indexNext1] = Heap[indexPrev];
                Heap[indexPrev] = temp;
                indexNext1 = indexPrev;
                if (indexPrev % 2 == 0) {
```

```
                indexPrev = (indexPrev - 2) / 2;
            } else {
                indexPrev = (indexPrev - 1) / 2;
            }
        }
    }
    indexNext++;
```

The remove method is implemented in a similar fashion where the value of highest priority (the value at index 0) is extracted and then the priority property of the heap is restored. To restore the priority of the heap a recursive method is used which compares the branches at each level from the root to a leaf, and recursively promoting the branch in which the value of higher priority can be found.

```
private void restoreHeap(int prev) {
    if ((rightBranch(prev) < indexNext && leftBranch(prev) < indexNext)) {
        if(Heap[prev] > Heap[leftBranch(prev)] || Heap[prev] > Heap[rightBranch(prev)]) {
            if (Heap[leftBranch(prev)] < Heap[rightBranch(prev)]) {
                temp = Heap[prev];
                Heap[prev] = Heap[leftBranch(prev)];
                Heap[leftBranch(prev)] = temp;
                restoreHeap(leftBranch(prev));
            } else {
                temp = Heap[prev];
                Heap[prev] = Heap[rightBranch(prev)];
                Heap[rightBranch(prev)] = temp;
                restoreHeap(rightBranch(prev));
            }
```

Iterations	**Elements (n)**	**add**	**remove**
1000000	100	3.98μs	5.85μs
1000000	200	8.67μs	12.6μs
1000000	400	15.8μs	27.9μs
1000000	800	27.4μs	62.2μs
1000000	1600	50.5μs	130μs
1000000	3200	101μs	298μs

Table 4: Time to add/remove n amount of elements to/from the heap.

If one would need to remove a considerable amount of elements from the priority queue and very seldomly add any items from the queue the first implementation may be the optimal choice. However, for the general case of the array implementation is evidently the better performing implementation.