Lab 10 A heap or priority queue

Weichen Chai

Fall 2022

Introduction

This lab report seeks to answer the questions presented in "Heaps or priority queue" for ID1021. In this assignment, the goal is to analyse the advantage of disadvantages priority queues of different variations.

Background

In a priority queue, the elements are placed according to priority instead of when the item was added.

Tasks

Linked list

In this section, we will be bench-marking the adding and removal of elements in a priority queue list, as stated in the task, we will be making an add function of complexity O(1), and remove function of complexity O(n), Then compare it with a function when add function is O(n) and the remove function is O(1). The priority of the smaller elements will be greater than the larger ones.

```
    public Integer removeFirst() { // only removes the element with the highest pri
        int s = head.item;
        head = head.next;
        return s;
    }

    public void add(Integer item) {
...
    Node c = head;

    if (item < c.item) { //replaces if added item has lower priority
```

```
            Node n = new Node(item, c);
            head = n;
    } else {
        while (c.next.item < item && c.next != null) {
            c = c.next;
        }
...
```

Delete finds the first element of the list, which contains the highest priority, then removes it. add traverses through the function to find a position to insert the node.

```
 public void addLast(Integer item) {
        Node n = new Node(item, null);
        if (head == null)
            head = n;
        if (tail != null)
            head.next = n;
...


  public Integer remove() {
...
        while (c.next != null) { //loops with complexity O(n)
            if (c.next.item < min && c.item > c.next.item) {
                p = list
                min = c.next;
   ...
```

Size	Remove O(1) Add O(n)	Remove O(n) Add O(1)
100	171	151
200	275	253
300	472	478
400	591	586
500	702	715
1000	1402	1393
1500	2059	2063

Its obvious from the table shown above that the two variations of add and remove does not serve a fundamental difference in the final result in time. Their time complexity are both O(n), as adding and removing in both cases add up to this complexity.. The situations where we prefer one over

the other can be when we are asked to mainly remove or add an item, if we remove a lot of elements, then using the remove function with complexity O(1) would be better, and vice versa.

Heap

In this task, we will implement heap as a linked binary tree, this method is slightly faster than linked list as the time complexity should be O(log n). The requirement is for the node in the tree to hold element, branches and even the number of elements in the sub trees. When adding an element, we go down the branch with fewest elements. The add function for priority queue with heaps are implemented by following the instructions given by the assignment.

```
Integer temp = null;
            while(current != null){
                current.size = current.size + 1;
                if (current.data > value) {
                    temp = current.data;
                    current.data = value;
                    temp = value;
                }
                if (current.left == null) {
                    current.left = new Node(temp);
                }else if (current.right == null) {
                    current.right = new Node(temp);
            }
```

Remove function on the other hand, is also done by following the instructions given by the assignment. When left branch is empty then right node is promoted. This process means the remove only deals with root values. We begin by promoting the right value if left is empty and vice versa. Following the guide, we have...

```
            if (current.left == null) {
                current.data = current.right.data;
                current.right = null;
                current.size = current.size - 1;
                return current;
            }
...
            if (current.left.data < current.right.data) {
                current.data = current.left.data;
                current.size = current.size - 1;
```

```
            if (current.size == 1) {
                current.left = null;
            } else {
                current.left = current.left.remove();
            }
            return current;
```

.
.
.

Increment in a heap can be implemented through a push method, which is quite frequently used. When we perform operations on elements, such as giving it a new priority and returning it to the queue, it will only be slightly less than its previous priority, so it only needs to be pushed a few levels down. Its a cheaper alternative to removing and adding again. Its used by looping until reaching a value that is larger than it, then

```
public void push(Integer incr) {
Integer temp = null;
incr += current.data;
if (current.left != null && current.left.data < current.data) {
        if (current.right.data < current.left.data && current.right != null
            swap(current.right.data , current.data)
```

...
//Mirror the above

In the benchmark, we only look at the depth in which push operation and the add operation arrives at. According to the task, we add 64 elements with random elements to a heap ranging form 0 to 100, then a series of push operations which ranges from 10 to 20. We will have 6 layers in this tree, and the benchmark shows the depth of push and add method respectively.

push	**Depth**	**add**	**Depth**
15	4	14	6
17	5	16	5
12	3	17	4
14	3	17	6
16	5	19	5
16	4	18	6
15	3	14	4
13	4	16	6

Array implementation

At last, we will implement a heap with an array. First we look at adding an element of bubble method, we compare the newly added node to if it has a lower value compared to its parent. if its lower we swap them. The only two possible outcomes when the parent is not greater than the added, or we reach the root of the tree, in both cases, we achieved our goal.

```
    public void add(Integer value) {
    ...
    while (arr[parent] > arr[child]) { // if its lower than parent, swap

            int temp = arr[child];
            arr[child] = arr[parent];
            arr[parent] = temp;
   ...
```

Sink method remove, we replace the removed item with last value in the array. Then compare downwards and swap with small branches until we reach a leaf or branch that has higher values.

```
Integer parent = 0;
int child1 = 2 * parent + 1;
int child2 = 2 * parent + 2;
        while (k >= parent) {
        ...
            if (arr[child1] == null) {
                                Arrswap(parent, child2)
                                        parent = child2;

                        }else if (arr[child2] == null){
// Mirror the above
                        }else if (arr[child2] <= arr[child1]) {
                                        if (arr[child2] < arr[parent]) {
                                            Arrswap(parent, child2)
                                                parent = child2;
...
}
// Mirror the above
```

The benchmark compares the efficiency of binary tree heap to array based heap.

As shown above, they have the time complexity of O(log n).

Size	**AddList**	**AddArray**	**RemoveList**	**RemoveArray**
100	68	40	81	71
200	79	42	89	73
300	84	45	96	79
400	88	46	101	81
500	92	48	106	86
1000	102	51	116	91
1500	108	53	120	96