A heap or priority queue

Fatima Mohammad Ali

Fall 14/10/2023

Introduction

The purpose of this assignment was to explore *priority queues* implemented in different structures. A priority queue stores items and their priorities in the queue with higher priorities being closer to the head. In this assignment, lower integers had higher priorities and the queue was implemented as a linked list, a linked tree structure (so-called *heap*), and an array heap. The representations were benchmarked and analysed mainly by their *add*, *remove*, and *push* methods.

A list of items

In this task, two implementations of the `add` and `remove` methods were done for a linked list. As usual, the list kept track of the `first` pointer and each node held a *next* pointer and an integer value `prio` which was used to compare priorities between the nodes.

The first implementation had an `add` ($O(1)$) and a `remove` ($O(n)$) method. Adding was done regularly by linking a new node to the beginning of the queue with the use of `first`. The `remove` method however, had to iterate till the end of the queue to find the node with lowest value (highest priority) and unlink it. This was done by utilizing an iterator `Node cur` (and its previous pointer `Node prev`) that traversed through the queue while comparing each value with a candidate, `Node temp`. Both pointers started at `first`. Each time a lower value was found, `temp` and its `tempPrev` pointer were updated to the current and previous nodes. Hence when the iteration was done, the node with the highest priority was saved in `temp` and unlinking it was done by utilizing the previous pointer.

The second implementation, with `add` ($O(n)$) and `remove` ($O(1)$), was done in opposite manner. The `remove` method unlinked a node from the beginning of the queue by utilizing the `first` pointer and returning its value (priority). The `void add(int prio)` inserted items according to their priority in the list. This was done by first checking if the queue was empty or if the to-be-added item had higher priority than `first`. If so, the new item was inserted as `first`. Otherwise, a `cur` iterator was initialized (started at *first*), and the queue was iterated as long as the next node had lower priority than the to-be-added node. When the correct position to make an insert was found, the given node was inserted in the queue by the use of the next pointers.

Benchmark results and analysis

The benchmark started by populating n elements to the queue (sequential order). The time of adding 100 additional random values (ranged [0,...,n], given in an array) was measured, and the time to remove 100 elements was measured. The queue was cleared before testing the second add and remove methods but the same 100 random keys were given.

n	add O(1)	remove O(n)	add O(n)	remove O(1)
200	8.0	150	49	0.7
400	1.0	280	82	1.4
800	1.0	590	140	0.4
1600	0.7	1100	240	0.4
3200	0.7	1300	520	0.4
6400	0.7	2300	950	0.3
12800	0.7	4400	1700	0.3

Table 1: Time measured in [µs]. Each timestamp is the minimum value of 100 tries.

As seen in table 1, the $O(n)$ methods seems to be linear considering both are approximately doubling in size since they have to iterate through the queue of n elements to perform their respective operation. The constant implementations are constant since they only need to use the first pointer.

Furthermore, it could be seen that the `remove` method of $O(n)$ performed worse than `add` O(n). This could be due the fact that `add` do not need to traverse till the end of the queue everytime to insert a certain value at the correct position (it breaks out of the loop). However, the remove method has to iterate through the entire list and determine which one has the highest priority everytime. Additonally, the remove method has to keep track of several extra nodes and constantly updates those (the previous and temporary pointers). If the list was doubly linked, perhaps the remove method would perform a little better.

If the queue has to be sorted from the start, the second implementation where add inserts values in sequential order (from highest to lowest priority) could be beneficial. The other add method solely adds to the beginning of the queue without sorting the queue, since its corresponding remove method would remove the elements from highest to lowest priority.

Linked Heap Implementation

In this task, a heap of linked tree structure was implemented. Each `node` held a *size*, a *prio* (int), and *left* and a *right* pointer. The tree kept track of the *root*.

The `add(int pr)` method was implemented by checking different cases to keep the tree complete and balanced. As a start, the *size* was incremented. Then the inserting procedure started. If the given priority (`pr`) was less than the current one (`prio`), the values were swapped. If the left branch was not empty, check if the left branch was not empty. If it was non-empty, add `pr` recursively to the branch with smaller size. Else, add to the empty branch and return. When calling this `add` function, it was first checked if the root was null (if so it was added directly as the root), or if it was not null then `add` was called.

Similarly, `Node remove()` method was implemented. First off, the *size* was decremented. Then it was checked if the *left* branch was null, then the *right* branch would be *promoted* and returned, vice versa. Otherwise, if the left branch has higher priority than the right (smaller value than the right), then the left branch adopted the current node and its size was decremented then it was removed by calling `remove`. Same procedure for the opposite case. At last, `this` was returned. If the root was null when calling `remove`, -1 was returned (throwing an exception could have worked). Otherwise the root (current value), which holds the highest priority, was removed.

To increment the root and insert it back in the heap, a `push(Integer incr)` was used. As a start, the root was incremented accordingly and a `cur = root` was declared. To decide which branch to push towards, a temporary current node (`tempcur = null`) was declared. Several cases were checked then while traversing through the heap. If the current left branch was not null *and* -the right was null *or* the left branch had higher priority (smaller value) than the right branch-, then `tempcur` was promoted as the current left branch. If the current right branch was not null, then `tempcur` was promoted as the current right branch. To swap the current node with the temporary node, another condition was checked. If `tempcur != null` and `tempcur.prio < incrementedRootValue`, then the current and temporary nodes were swapped and the depth was incremented. Else, break out of the loop and return the depth.

Array Heap Implementation

In this task, a heap in array representation was implemented instead. The array heap initialized an `int[] heap`, kept track of its *size*, and a *free* index (lastly added). The heap was given a fixed size during run-time, so it was not dynamic. Adding to the end of the queue was simply done by adding at index free, then *bubbling* it up the heap. This was done in `void bubble(int indx)` where the parent is detected (with `parent = (indx-1)/2`), then it was checked if the child at `indx` was smaller than its parent. If it was, then it had higher priority, and it was swapped with the parent. This was done as long as the child did not reach the root (indx == 0) (bubble called itself recursively).

Similarly, when removing an item at the root (at index 0) from the heap, the following sink method was used to maintain heap property. Also a `push(int incr)` method that increments and pushes the root value was implemented for heap array.

```
public int sink(int indx, int depth){
    if(free == 1)   //
        return depth;
    //left and right children
    int left = indx*2 + 1;
    int right = indx*2 + 2;

    int smallest = indx; //the index that will be sinked
    //check if left child is smaller than its parent
    if(left < free && heap[left] < heap[smallest])
        smallest = left;
    if(right < free && heap[right] < heap[smallest]) //same with right child
        smallest = right;
    //If smallest is not the parent, swap(go down a level)
    if(smallest != indx){
        swap(indx, smallest);
        depth++;    //increment depth
        depth = sink(smallest, depth);
    }
    return depth;
}

 public int push(int incr){
    int depth = 0;
    heap[0] += incr;    //increment root value
    depth = sink(0, depth);
    return depth;
}
```

Heap benchmark

*Manually checking the push methods was done by populating respective heap with $(2^10) - 1 = 1023$ random values, incrementing the root by a random value ranged [0,...,100] (done 100 times wiht each call of `push(int incr)`), and printing the depth. The returned values seemed to be correct where the depths varied from 0 to maximum 6.

n	**linked deq/enq**	**l-heap push**	**arr-heap deq/enq**	**arr-heap push**
2000	110	59	74	52
4000	130	62	86	53
8000	150	69	87	60
16000	160	79	81	60
32000	160	73	84	59
64000	190	92	86	64
128000	210	86	94	65

Table 2: Removing then inserting the incremented root vs push operation for *linked heap* vs *array heap.* Each timestamp is the minimum value of 1000 tries and given in [µs].

Heap analysis

In a heap, the node with highest priority is found at the root in constant $O(1)$ time, but removing it requires replacing it with another node while also maintaining heap property. The sinking process is done at $O(log(n))$ time which is proportional to the height of the tree (depths). Inserting in a heap is in $O(log(n))$ time.

The push operation takes the root value, increments it, and then pushes it down the heap, which is done in $O(log(n))$ time. Removing and adding an incremented root instead, would require twice as much work as the push operation, which is more costly. In table 2, it can be seen that while all columns exhibit logarithmic behaviour, both push operations are faster than their counterpart.

Additionally, the array heap performed better than the linked heap. This could be due several factors, one being the memory access in arrays is done more efficiently than in linked structures.

*Since the heap only has 1023 values, there could be only 9 levels to traverse down to. Since the incremented values are not large enough, it would be difficult to reach further down so the furthest level would was 5-6. Doing the math would confirm this further but I have no energy nor mental capacity to do so right now.