Queues

Pontus Hector

Fall 2022

Introduction

This report is conducted as the seventh assignment in the course ID1021. In this report one explores different implementations of queues, their benefits and drawbacks and the difference between iterating breadth first and depth first through a binary tree.

Queues using linked lists

The implementation of queues using linked list mainly consists of two functions, an add function and a remove function. The add function adds objects sequentially and places each new object at the end of the linked list which altogether forms the queue. Adding a new element to the list with no further implementations renders the method somewhat inefficient since the data structure of linked lists does not naturally allow one to add an element at the end of the list. This implies that one needs to iterate through the entire list to add a new element to the queue. Therefore a new node pointer *Next* is implemented to keep track of the last element of the list.

```
public void add(Integer item) {
            Node newNode = new Node(item, null);

            if(Head == null){
                Head = newNode;
                Next = newNode;
            }
            else {
                Next.tail = newNode;
                Next = newNode;
            }
        }
```

This operation is linearly dependent on the amount of elements added to the queue. However, the operation itself is now constant due to the added pointer.

The remove function returns the first object (the object which the *Head* pointer is pointing to) of the queue and re-links the head pointer to point to the next sequential object of the queue.

```
public Integer remove() {
    Node returner = Head;
    Head = Head.tail;

    if (Head == null) {
        Next = null;
    }
    return returner.item;
}
```

This operation is also linearly dependent on the amount of elements removed from the queue while the operation itself is constant.

Breadth first versus depth first

Implementing an iterator using a queue instead of a stack enables breadth first travel of a binary tree. Using breadth first travel the iterator wanders between each node sequentially from the root node through all nodes at the present depth before moving on to the nodes at the next depth level. In practice this leads to that breadth first traversal stores the entire current level of the node tree in the queue at the same time before moving on to the next depth level which, for the general binary tree, makes it less memory efficient than depth first and especially for large binary trees. In the benchmark below breadth first traversal seem to be a slightly faster traversal method when searching for a certain node in the node tree than depth first travel, but this is heavily dependent on whether the target node is closer to the root node or if its deep down the node tree.

Iterations	**Nodes (n)**	**Breadth first**	**Depth first**
1000000	100	2.07μs	2.44μs
1000000	200	3.34μs	4.07μs
1000000	400	6.72μs	8μs
1000000	800	13.2μs	16.7μs
1000000	1600	26.4μs	33.6μs
1000000	3200	54.6μs	72.9μs

Table 1: Time to iterate through the node tree to a randomly selected node.

Breadth first traversal is a considerably more efficient traversal method when the target node is relatively close to the root node of the binary tree **but** if the target node is far down the tree and/or if the binary tree is completely balanced the depth first traversal most certainly performs better.

Implementation of a queue using an array

The implementation of a queue using arrays contains the same methods as in the previous queue, but reminds more of the implementation of a stack using arrays. Here the add function is changed to operate on the contains of the nodes instead of actual nodes and to utilize the full potential of storing the elements in an array, we add values to every single empty spot of the array before expanding the array since this is a costly operation.

```
if (indexNext < capacity - 1) {
    queue[indexNext] = element;
} else if (indexNext == (capacity - 1)) {
    indexNext = 0;
}
if (indexNext == indexFirst) {
    expandArray();
} else {
    queue[indexNext] = element;
}
indexNext++;
```

The expansion of the array is a new method implemented which allocates more memory if the current array is completely full of elements. The method is implemented since one only have a predetermined amount of memory to store elements in, in contrast to the linked list implementation, and hence need a way of expanding that memory.

```
int curr_size = capacity;
Integer[] new_array = new Integer[curr_size * 2];
int i = 0;
for (int j = indexFirst; j < curr_size; j++) {
    new_array[i] = queue[j];
    i++;
}
for (int j = 0; j < indexFirst; j++) {
    new_array[i] = queue[j];
    i++;
}
indexFirst = 0;
queue = new_array;
capacity = new_array.length;
```

Dequeueing elements from the queue is simply done by extracting the first item of the queue, clearing the element at that index and then setting the first element pointer to point at the next sequential element of the queue. If the queue is empty the method shall return null and if the pointer of the first index runs out of bounds the index should be set to zero, where the next sequential element of the queue will be found.

```
if (indexFirst == (capacity)) {
    indexFirst = 0;
}
if (queue[indexFirst] == null) {
    return null;
} else {
    Integer returner = queue[indexFirst];
    queue[indexFirst] = null;
    indexFirst++;
    return returner;
}
```

The implementation of a queue using arrays is less efficient on enqueuing and dequeueing than the linked list implementation. This is since even though query is more efficient when operating on arrays than operating on linked lists this factor does not outweigh the time costly operation of dynamically expanding the array and the many new implemented if-statements, implemented to catch the edge cases in the array implementation of a queue.

Iterations	**Elements (n)**	**Array**	**Linked List**
1000000	100	2.9μs	1.3μs
1000000	200	4.9μs	2.7μs
1000000	400	8.8μs	4.7μs
1000000	800	15.2μs	8.3μs
1000000	1600	30.3μs	13.3μs
1000000	3200	62.7μs	26.1μs

Table 2: Time to add (n) amount of elements to the different types of queues.