Queues

Laasyasree Desu

Fall 2023

1 Introduction

This assignment involved the implementation of queues in two different ways - using linked lists as well as arrays. For the linked list a method for traversal of the queue was created and for the array a dynamic queue was also created. The two implementations were then compared.

2 Queue

A queue is a fundamental data structure that can be described with a real life example of standing in line, where the last person join stands at the back of the line, and the first person is the first to be served. Similarly, a queue as a data structure is an ordered collection of elements that follows the first-in, first-out principle, also known as FIFO. This means that the element added to the queue first will be the one to be removed first. Queues can be used for various different applications and have various implementations as well such as linked lists and arrays.

3 Linked list implementation

The implementation of a queue using linked lists was first done by making the linked list through initialising two nodes to act as pointers. One named first and one last, to keep track of the first and last nodes of the queue. The two pointers were set initially set to null. Additionally, two different methods were created - enqueue and dequeue. The two methods functions involved adding an element two the queue and removing an element respectively.

```
public Integer element;
public Node next

public Node(Integer element, Node next) {
    this.element = element;
    this.next = next;
}
```

3.1 Enqueue

The enqueue method involved adding an element or node to the list. This was implemented by checking the "status" of both the pointers. An if - statement would first check the "first pointer" was pointing to null, in which the queue is empty. The "first" pointer would then be updated to point to the new node, and the "last" pointer would also be updated to the new node. Otherwise, another if statement would check if the last pointer was not pointing to null, in which case the new node would be added there by updating the pointer to point to it. The "last" would then be updated to the new node.

```
    if (first==null) {
        first = newNode;
        }
    if (last != null){
        last.next = newNode;
    last = newNode;
```

3.2 Dequeue

The dequeue method handled the removal of the first node in the queue. This was implemented in three steps - the first step checked if the queue was empty or not. This was done through an if-statement that checked weather the "first" pointer pointed to null. If it did then the method would return null. Otherwise the "first" pointer would be updated to the next node and the original first element would be returned. But before returning, the "last" pointer was also pointing to the node being removed (the queue only had one element), the pointer would be updated accordingly as well.

```
if (first == null)
    return null;
Integer firlement = first.element;
 first = first.next;
 return firlement;
```

---

4 Traversal

This method for traversing a binary tree (iterator) was implemented queue, as opposed to a linked list for a previous assignment. The implementation was relatively similar, except this the traversal was breadth first, where all the nodes on the same level would be iterated through first. Similar to before this implementation has a ”hasNext” method and a ”next” method that handle verifying that there is a node after the current one and dequeing and enqueing of upcoming nodes respectively. The breadth first traversal happens as the nodes are traversed and processed in the order they appear on the tree instead starting from the far left.

```
public boolean hasNext() {
        return !queue.isEmpty();
    }
```

5 Array implementation

The third task for this assignment involved implementing the queue using an array instead. This was done similar to the linked list, by first creating the array and then creating to methods to enqueue and dequeue. The creation of the array was simply done by initialising the array and allocating the necessary space (size of the array). Additionally, similar to before to pointer were also created and set to 0. Additionally, other conditions were set in place to resize the array when necessary, for example by doubling the arrays size when the queue was full. This is to make space allocation more efficient. Apart from this an element counter was also added to keep track of the number of elements in the queue.

5.1 Enqueue

The implementation enqueue to add an element was relatively more complicated than the linked list implementation. This is because the condition of the array being had to be considered. Therefore, the first step was to check this, which was done by an if-statement that would check if the elements counter was equal to allocated size of the array. This would mean that the array would have to be resized, which was done allocating space for an array twice as big and copying all existing elements onto it. The ”last” and ”first” pointers are reset accordingly and the the new queue is returned. In the case that the queue is not full, or after the new array is returned, the ”last” pointer is updated accordingly and the new element is added to the queue. Lastly the elements counter is also incremented.

```
if (last == first) {
    Integer [] new = Integer [capacity*2];
    int j = 0;
    for (int i = first, i< capacity; i++){
    new[j] =arr[i];
    j++;
    }
}
```

5.2 Dequeue

The dequeue method was implemented similar to the linked list implementation. The method started off by first verifying if the queue was empty. This was done with an if statement that check if the elements counter was equal to 0, in which case the method would throw an exception. Otherwise the element that the ”front” pointer pointed to was taken into a new variable (to later be returned). The front pointer would then be updated accordingly and the element counter would be decremented.

```
public Integer dequeue() {
    if (counter==0) {
    throw new IllegalStateException("queue is empty")

}
```

6 Summary

The assignment involved implementing queues through linked lists and arrays. The implementation through linked lists was fairly simple, adding and removing elements only involved changing the references/pointers, making it a constant time operation. The implementation was also used to traverse a binary tree, this time breadth first. The implementation through was slightly ore complicated as it had handle resizing to be able to accommodate more elements.