Ques Report.

Adrian Jonsson Sjödin

Fall 2022

1 Introduction

2 Task

Implement two different FIFO queues. One utilizing arrays and one utilizing a linked list structure.

For the linked list queue, first implement a linked list queue structure that only has one property, the *head*, and new elements are simply added to the end of this list. What are the drawbacks to this implementation? What are the cost of removing the next element and adding a new element?

Change the linked list queue so that it holds a pointer to to the first element of the queue (the *head*), but also a pointer to the last element of the queue. This should allow one to add a new node at the end directly without first having to traverse the list to find the last node.

Now use the linked list queue to create a new iterator for the binary tree from last weeks assignment that traverses the tree breath first.

The array implementation of the queue should be dynamic and be able to increase in size, but also optionally shrink in size.

3 Method & Theory

A queue is a data structure that is built on the FIFO principle of *first* element *in* is *first* element *out*. This structure is useful in real world application such as printers where we want to print a document but at the same time send more documents to it to be printed after, in the order we sent them.

3.1 Linked List Queue Without Second Pointer

The first queue to be implemented was the linked list queue that did not have a pointer to the last element in the queue. The task said to have a pointer to the head and add all new elements to the end of the list. I however went with the opposite and made the *head* pointer point to the last added element and that element in turn point to the old head. This way I can create a

linked list that looks like in figure 1. This shouldn't make a difference in the overall time it takes to add and remove elements to the queue. The way the task wanted us to do would mean a cost of $\mathcal{O}(n)$, there n is the size of the queue, for adding and $\mathcal{O}(1)$ for removing. My implementation would simply reverse these costs.
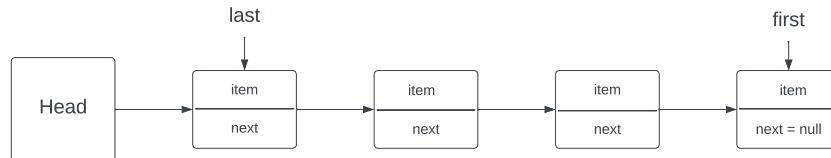

Figure 1: Linked List Queue with only one pointer

Adding an element to this queue is then just this one line of code `this.queue = new Node(item, this.queue);`, in which we set the *head* (here called `queue`) to be the new node and let that node point to the old *head*, which depending on if the queue is empty or not before the add operation, will have a pointer to an even older *head*.

Removing elements from the queue is a bit more code but still quite straightforward. We first need to see if the queue is empty or not, but assuming that it is not we then need to traverse the whole queue till we reach the element with a null pointer. This will be the first element added to the queue. Having found it we can simply make the pointer for the previous element point to null instead and then return the found first element.

The code for `add()` and `remove()` can be seen in code overview 1.

3.2   Linked List Queue With Pointer To First & Last Element

The second linked list queue is an improvement over the the first one in that it has two pointers. One to the first element in the queue and one to the last. This should allow us to make enqueue and dequeue operations at at constant time since we can directly add the new node to the end of the list without first having to go through the whole list and find the last element, and we can also immediately retrieve the first element of the list.

The queue itself is a built up as depicted in figure 2. So when we initialize the queue with the constructor we first set `first` and `last` to null. This way we now have an empty queue. We can then add elements to the queue using the `enqueue` method. When adding elements we first need to check if the queue is empty or not. That is done by looking at what `first` is pointing at. If it is null the queue is empty and we can simply create a new node and let both `first` and `last` point to that node. If `first` is not null then we have something in the queue and `first` and `last` is already pointing at something. We then create a new node and set `last.next` to

point at the new node to make sure every element in the queue is linked to the next element. We then finally set `last` to also point to the new node.
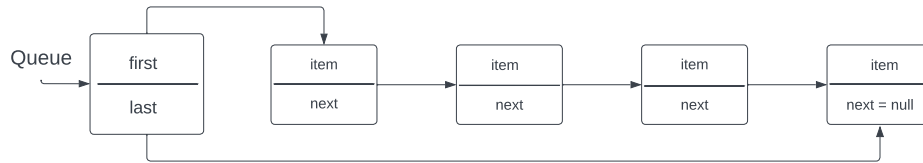

Figure 2: Linked List Queue with two pointers

The `dequeue` method is quite similar in that we first check to see if the queue is empty or not. If it is we print a message telling us that and return null. However if it is not empty then we start by retrieving the first element's item, then set `first` to point at `first.next` instead. We also need to check to make sure that the retrieved element wasn't also the only element in the queue. This is done by checking wether or not the retrieve element is the same element as the one `last` is pointing at. If it was we also put `last` to be null (`first` would already have been set to null in the previous step in this case) before returning the first element's item.

The code for `enqueue()` and `dequeue()` can be seen in code overview 2.

3.3   Breath First Traversal of Binary Tree

Implementing a iterator for the binary tree created in last week's assignment required just some slight modifications to the stack iterator we created. First instead of creating a stack in the `TreeIterator` we create our queue that can accept nodes. In the constructor we then initialize the queue and add the `root` of the binary tree to the queue using `enqueue()`.

The `hasNext` method is left the same but calling the queue's `isEmpty` method instead.

The `next` method had to be completely altered. We still check wether or not we have a next node in the tree using the `hasNext` method, but then assuming we have one we pop the queue and set it as the current node. We then first check if current has a pointer down left and if it has we push that to the queue. Then we check if it has a pointer down right and also push that to the queue if there is one. After this we simply return the value of the current node.

This way next time `next` is called the node that was down left will be popped and set as current, and if that was null then it will not have been added and instead the node down right will be the node popped. This will make it so that we traverse the binary tree from left to right, top to bottom, i.e. breath first traversal.

The code for `next` can be seen in code overview 4.

3.4 Dynamic Queue Using Arrays

4 Result

5 Discussion

Code

All the code can be found here:

Code Overview 1: Add and remove in first linked list queue

```java
public void add(Integer item) {
    this.queue = new Node(item, this.queue);
}
public Integer retrieve() {
    Node prev = null;
    Node current = this.queue;
    if (isEmpty()) {
        System.out.println("queue is empty");
        return null;
    }
    while (current.next != null) {
        prev = current;
        current = current.next;
    }
    this.queue = prev;
    return current.item;
}
```

Code Overview 2: Enqueue and dequeue in second linked list queue

```java
public void enqueue(Item item) {
    Node newNode = new Node(item, null);
    if (this.first == null)
        this.first = newNode;
    if (this.last != null)
        this.last.next = newNode;
    this.last = newNode;
}
public Item dequeue() {
    if (this.first == null) {
        System.out.println("dequeue(): queue is empty");
        return null;
    }
    Item returnedItem = this.first.item;
    Node retrievedNode = this.first;
    this.first = this.first.next;
    if (retrievedNode == this.last)
        this.last = null;
```

```
        return returnedItem;
}
```

Code Overview 3: The `next()` method for the iterator

```java
public Integer next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    Node current = queue.dequeue();
    if (current.left != null)
        queue.enqueue(current.left);
    if (current.right != null)
        queue.enqueue(current.right);
    return current.value;
}
```

Code Overview 4: Next() and moveLeft(Node current)