Queues Report.

Adrian Jonsson Sjödin

Fall 2022

1 Introduction

In this assignment we will look at how queues function. How to implement them and how they can be used in other programs

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.

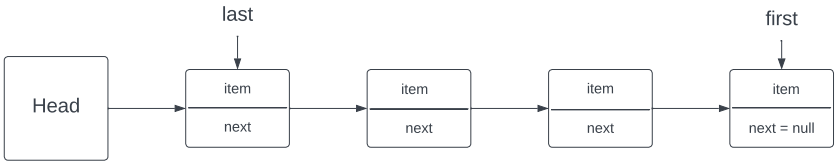

Figure 1: Linked List Queue with only one pointer

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.

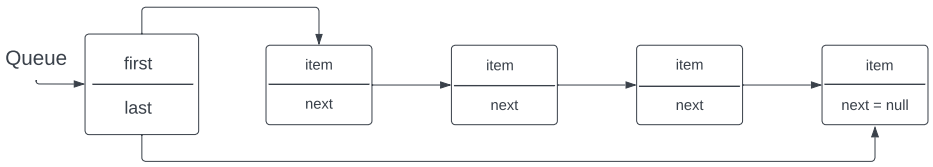

Figure 2: Linked List Queue with two pointers

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 3.

3.4 Dynamic Queue Using Arrays

Lastly we were supposed to implement a queue using arrays. This was by far the trickiest thing to implement since we needed to keep track of where in the array we where storing the elements.

The idea is that we create an empty array of a fixed size that can hold generic objects, and where each element is initially set to null. We can then add elements to the que by placing them in the array. To keep track of where the first element and the last element is located in the array we also need two pointers, `first` and `last`. These are simply storing the index of the array where the first and last element in the queue is stored.

When adding or removing an element to the queue, or resizing the array there is a lot of things to consider. Here I am going to go through them one after the other in the order that I implemented them, starting with the `enqueue` method.

3.4.1 `enqueue(Item item)`

First we need to check if the array is empty. This is done by having a private field `itemsInQueue` which tracks how many elements we have added to the queue. If the queue is empty we store the element at the index value given by `first` (`first` and `last` are set to 0 when queue is first created), and then increment `last` and `itemsInQueue` by one. This will make it so that `last` points to the next empty spot in the array.

If it is not empty then we do one or two things out of four things. First we check if `last -1 == size -1` (meaning is the last index in the array taken) and check if the array is not full. If this gives `true` then it means that we have dequeued elements at the front of the array and that there's free space we can use. We then set `last = 0` which will allow us to reuse that space. After this check we also check if the array is full and if it is we resize it. This will not happen in this case since we said the earlier check gave `true`. After this check we have yet another check and this one checks wether or not `last == 0`. This is to see if we should add the new element to the start of the array and reuse space or not. If it gives false we simply add it at the index value of `last` and then increment it and `itemsInQueue` with one. If it gives `true`, which it will in this case we do the same thing but the index value of `last` will now be 0.

As mentioned this was quite tricky and also hard to describe in text. To get a better overview of how this works I've included the code for this method in code overview 4.

3.4.2 `dequeue()`

The `dequeue` method in comparison is quite simple. Here we first check if the queue is empty and if it is we inform the user and return null. If it is not we do basically the same operation as in `dequeue` for the linked list. We start with retrieving the first item from the array by retrieving the element at the index value of `first` in the array. We then set that spot to null and increment `first` by one and decrement `itemsInQueue` by one. After this we check if `itemsInQueue <= size / 4`. If this gives `true` that means we have a lot of empty space in the array and we want to shrink it. To do this we call `resize()` which will shrink the array. Now lastly we return the retrieved item.

The code for the `dequeue` method can be seen in code overview 5.

3.5 `resize()`

The `resize` method is what makes this queue dynamic. It will expand it if it is full and shrink it if there's a lot of empty space. It will also place the elements in order with `first` on index zero the rest placed after it in order.

Before we do anything else we need to initialize some variables. We need two `int`'s, one for the new size of the array and one for iterating through the new array. We also need a temporary array for storing the elements from the queue array. After this is done we can check if we should expand or shrink the queue array. This is done using `if else` where if the array is full we expand it, else we shrink it.

If it is full we set the new size to be double that of the old size and use that size when allocating the temporary array. After that we need to set the iterator counter `j` to zero and loop through the queue array from index `first` to the end of the array and save it to the temporary array at index `j`, not forgetting to increment `j` with one in each loop. After this loop we need to loop through the queue array once more but this time from the start of the array to where index `first - 1` is at, and save each value to the temporary array at index `j`, and make sure to increment `j` with one once more. We will now have a copy of the queue array stored in the temporary array with `first` stored at index zero, `last` stored at index (`size -1`), `j` having the value of `size` and index at value `size` to (`newSize -1`) set to null.

Now we just need to let `queue` point at the temporary array, set `size = newSize`, set `first` to zero, `last` to `j` and `temporaryQueue = null`. This is done outside the `if` and `else` statement, at the end of the `resize` method. When we now call the `resize` method in `enqueue(Item item)`, `last` will point at the next empty space in the queue array that will now have doubled in size.

If the queue array is not full and the `resize` method is called then the

first if statement will give `false` and we want to shrink the array. We set `newSize` to be half of `size` and allocate a temporary array of that size. We also need to set the index counter `j` to zero. Then we can start looping through the queue array from `first` to the end of the array and saving it to the temporary array at index `j`. However we add an if statement in the loop before the copying step that checks wether or not the element at the current index in the queue array is null. If it is we break the for loop since we have then copied all the elements from the queue array. It is after this check that we do the copying to the temporary array as well as incrementing `j`.

We now have the second loop and this one only runs if `last < first`. This means the the queue we want to shrink have elements that have wrapped around and that `last` comes before `first` in the array. If this is `true` we loop through the queue array from index zero to index `last -1` and save each element in the temporary array at index `j`, not forgetting to increment `j` in each loop. After this is done the same step as described above will be executed.

The code for `resize()` can be seen in code overview 6.

4 Discussion

As mentioned in subsection 3.1 the time complexity for adding objects to the queue is $\mathcal{O}(n)$, there n is the number of objects in the queue, and the time complexity to retrieve objects from the queue is $\mathcal{O}(1)$. So while it is an easy queue to implement the cost of adding elements are a serious drawback when n becomes really large. This is why the second implementation of a linked list queue discussed in subsection 3.2 is to prefer. In it the both the cost of adding and retrieving objects are of time complexity $\mathcal{O}(1)$. This is due to the fact that the queue is using two pointer here, one to the first object in the queue (that object has a pointer to the next object in the queue) and one to the last object in the queue (that object has a null pointer). This way retrieving the first object is simply retrieving what the first pointer is pointing at and then set first to what that object was pointing at. This is a constant time operation. The same goes for adding new objects to the queue. Then we only need to look at what the last pointer is pointing at and set that objects null pointer to point at the new object and then re-point the last pointer to point to that object. This as well is a constant time operation.

Implementing an iterator that traverses the binary tree breath first was quite easy using a queue data structure. The queue made it possible to just simply look at the current node in the tree and then push the node to the left and the node to the right of the current node to the stack. When we then pop the queue the left node will be the first to be popped and then its left and right node will be added to the queue, then will the previous

added right node be popped and its left and right node will be added to the queue. Now the next time the queue is popped that previous popped left's node will be the one retrieved and the process will be repeated. This way starting from the `root` we will be able to traverse the tree from top down, left to right.

About the dynamic stack array. This was by far the hardest to implement and I had to make heavy use of the debugging tool to make it work. The reason why this was so hard was because of the need to keep track of where `first` and `last` was located in the array and when we could wrap around in the array as opposed to just make the array bigger. Then we also had to take into account how to expand and shrink the array when we had wrapped around objects in the array.

The time complexity for adding objects to this array queue is $\mathcal{O}(1)$ until we reach the point where the array is full, then we need to expand the array which will means that the time complexity to add an object then is $\mathcal{O}(n)$, where n is the size of the queue before it is expanded. This however will only happens ones every $n-1$ add operation (assuming we have no dequeue operations in there) so the amortized time complexity of adding objects to the queue is $\mathcal{O}(1)$. The same goes for retrieving objects from the queue. To retrieve one object has a time complexity of $\mathcal{O}(1)$, but if the object we want to retrieve makes it so that there is only $k/4$, there k is the number of objects in the queue, objects left in the queue we will have a time complexity of $\mathcal{O}(k)$ to retrieve said object. This is because we then want to shrink the array and then need to copy all the objects. However this as well will not trigger every time we want to retrieve an object so the amortize cost will be $\mathcal{O}(1)$.

While there is a bit more overhead involved when adding objects to an array queue, I still believe this will be faster than adding an object in the linked list queue. This because there we need to make a new node and then link it, and memory allocations does generally takes more time than just writing something in an already allocated array where everything is next to each other in the memory. The same goes for retrieving objects from the queue. However the fact that we sometimes need to expand and shrink the array might make up for this and actually make the linked list queue faster on average. But until this happens the array queue is probably faster.

Code

All the code can be found here: GitHub

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

```
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

```
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

```
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: Array implementation of `enqueue(Item item)`

```
public void enqueue(Item item) {
    if (isEmpty()) {
        this.queue[this.first] = item;
        this.itemsInQueue++;
        this.last++;
        return;
    }
    if (!isEmpty()) {
        if ((this.last - 1) == (this.size - 1) && !isFull()) {
            this.last = 0;
        }
        if (isFull()) {
            resize();
        }
        if (this.last == 0) {
            this.queue[this.last] = item;
            this.last++;
            this.itemsInQueue++;
        } else {
            this.queue[this.last] = item;
            this.last++;
            this.itemsInQueue++;
        }
    }
}
```

Code Overview 5: Array implementation of `dequeue()`

```
public Item dequeue() {
    if (isEmpty()) {
        System.out.println("dequeue(): queue is empty");
        return null;
    }
    Item returnedItem = this.queue[this.first];
    this.queue[this.first] = null;
    this.first++;
    this.itemsInQueue--;
    if (this.itemsInQueue <= (this.size / 4)) {
        resize();
    }
    return returnedItem;
}
```

Code Overview 6: The `resize` method

```
public void resize() {
    int newSize;
    int j;
    Item[] newQueue;
    if (isFull()) {
        newSize = this.size * 2;
        newQueue = (Item[]) new Object[newSize];
        j = 0;
        for (int i = this.first; i < this.size; i++) {
            newQueue[j] = this.queue[i];
            j++;
        }
        for (int i = 0; i < this.first; i++) {
            newQueue[j] = this.queue[i];
            j++;
        }
    } else {
        newSize = this.size / 2;
        newQueue = (Item[]) new Object[newSize];
        j = 0;
        for (int i = this.first; i < this.size; i++) {
            if (this.queue[i] == null)
                break;
            newQueue[j] = this.queue[i];
```

```
            j++;
        }
        if (this.last < this.first) {
            for (int i = 0; i < this.last; i++) {
                newQueue[j] = this.queue[i];
                j++;
            }
        }
    }
    this.queue = newQueue;
    this.size = newSize;
    this.first = 0;
    this.last = j;
    newQueue = null;
}
```