Queue

William Frid

Fall 2023

Introduction

When it comes to programming, queues aren't much different from the ones you might encounter at the grocery store. The difference though is that the types of queues a developer gets to work with come in different forms and structures, and hold data (obviously). They can for instance hold nodes that contain data such as data jobs that need to be queued to prevent incorrect order of work or simply a too-heavy workload.

This report will focus on exploring how queues containing nodes can be built using different structures as well as using queues alongside trees to make a breadth-first iterator. No benchmarks will be run, but some methods will be analyzed to determine their bi-O complexity and if/how they can be improved based on that.

A Basic Queue

The first queue that'll be focused on is one built using a singly linked list. This queue builds on four key components: `head`, Node class, `add()`, and `remove()`. The method `add()` adds a new node to the head if none exists, otherwise, it iterates through the whole queue to find the last position. This leads to its big-O complexity of $O(n)$. And `remove()` reads removes the first node from the queue and returns its value as well as updating the queues variable `head`.

This algorithm can however be significantly improved by adding an additional pointer apart from `head`, namely `tail`. This pointer will point to the last element in the queue, thus giving us an improved `add()` with big-O complexity $O(1)$ since the iteration is no longer needed. The final methods as visible in the snippet below.

```
public void add(Integer item) {
    Node n = new Node(item, null);
    if (head == null) {
        head = n;
```

```
            tail = n;
        } else {
            tail.next = n;
            tail = n;
        }
    }

    public Integer remove() {
        if (head == null)
            return null;
        Integer t = head.item;
        head = head.next;
        return t;
    }
```

Lastly, we could also use a doubly linked list instead of a singly. Doing so would allow us to unlink nodes from the queue without having to use a loop to locate the previous node of which we would update the variable `next`. This can be very useful in cases where there is a risk unlinking is needed. Even though it does require a bit more memory (because of the extra variable, let's call it `prev`), it would reduce the big-O complexity from $O(n)$ to $O(1)$.

Breadth-First Traversal

In case the programmer wishes to search a tree to find a certain node as close to the root node as possible, there's a great way called "breadth-first traversal". This form of algorithm is similar to iterating through a tree element by element with the keys in ascending order, but there's a key difference. Instead of following the keys, it simply iterates layer by layer, from left to right (where the left holds the smaller node key).

Looking at the snippet below, we see how the method `goToNext()` is different from the previous assignment called "Trees". This version of `goToNext()` pushed the left and right nodes to the stack which leads to this decided behavior.

```
private class TreeIterator implements Iterator<Integer> {

    private Node next;
    private QueueBinaryTree<Node> queue;

    TreeIterator() { ... }

    @Override public boolean hasNext() { ... }
```

```
    @Override public Integer next() { ... }

    private void goToNext() {

        if (next.left != null)
            queue.add(next.left);

        if (next.right != null)
            queue.add(next.right);

        next = queue.remove();

    }

    @Override public void remove() { ... }

}
```

This algorithm can as mentioned be great for searching through trees for nodes close to the root node or for handling trees that might include circular paths as this iterator wouldn't get stuck in an infinite loop and thus cause a stack overflow exception.

Array Queues

Queues can also be built using more simple data structures, such as an array. However keeping track of the array queue can be tricky due to three reasons: The fact that the array has to grow and shrink when needed, moving the queued nodes if needed, and lastly the usage of modulo to optimize the algorithm to now more or resize the array when there already is enough memory allocated.

To achieve this behavior the program has to keep track of where it currently is located in the array, and on what indexes there is accurate data to be used. Focusing on the `add()` and `remove()` methods (visible in the snippet below), we see that the logic needed for those is centered around calling a method for array size adjustment, and changing the variables `start` and `end` needed for locating the correct data.

```
public void add(Integer item) {
        if (start == (end  + 1) % arr.length && end + 1 > arr.length - 1)
                resizeArr(true);
        Node n = new Node(item);
        arr[++end % arr.length] = n;
}
```

```
public Integer remove() {
        if (start > end) // Empty queue.
                return null;
        if (end - start <= arr.length / 4 && arr.length > 4)
                resizeArr(false);
        Integer t = arr[start % arr.length].item;
        arr[start % arr.length] = null;
        start++;

        if (arr[start % arr.length] == null) {
                start = 0;
                end = -1;
        }

        return t;
}
```

It's clear that the logic behind removing a node from the queue is more complex than adding. Let's walk through the `remove()` method just to clarify how it works: The first step is to check if the starting point is greater and end, if so, it returns null (as the queue is empty). Next, we check the size to see if the array size has to be adjusted (more on this later). Next, we store the integer held by the node we wish to return in a temporary array so that we can set the queue slot to null. This is done for debugging purposes (as it's easier to understand that a slot is empty if it is equal to null). Finally, we increment start by one and then check if the queue is empty. If that is the case, we reset the `start` and `end` variables.

Next up, we look at the method `resizeArr()` which, as the implies, resizes the array/queue when needed. When it does this, it also rearranges the nodes into a natural order, so that the object first in the queue is at index zero as well as updating the variables `start` and `end`. How this is done is visible in the snippet below.

```
private void resizeArr(boolean grow) {
        int len;
        if (grow)
                len = arr.length * 2;
        else
                len = arr.length / 2;
        Node[] newArr = new Node[len];
        int newEnd = -1;
        while (start <= end) {
                newArr[++newEnd] = arr[start++ % arr.length];
```

```
        }
        arr = newArr;
        start = 0;
        end = newEnd;
}
```

Conclusion

Queues can have different structures based on the desired advantages. For instance, for instance, the array queue can read faster (using the method `remove()`) than the basic queue that builds on a singly linked list since that one's data is more likely to be spread about (physically).On the other hand, the basic queue is easier to work with and is more dynamic than the array one. Lastly, we see that queues have other use fields, such as acting as stacks for building iterators for breadth-first iterators.