Queues

Aditya Gupta

October 2022

1 Introduction

In this task, we implement a data structure that emulates a real-life queue but for items first using a dynamic node-based structure similar to how we implemented a dynamic stack but this time it uses the First in First out principle. We re-implement the Tree iterator but now use a queue instead of a stack by performing a breadth-first search. After this, we implement a static circular queue using an array and then make a dynamic implementation that increases the size of the array every time we run out of space.

2 Dynamic Queue

The easiest way one can implement the dynamic queue is by using a linked list. Every node has a value and a next pointer. We always add a node to the end of the queue and remove elements from the queue from the front, like in a real-life queue. The reason for this is that in a normal queue a person that has been in the queue the longest is the one that is removed and people that enter the queue are always added to the back of the queue. This is why a queue is called a FIFO data structure(First-In-First-Out).

2.1 enqueue():

```
public void enqueue(T item) {
    if(head == null){
        head = new Node(item);
        tail = head;
        //System.out.println("enqueued item: "+item);
        return;
    }
    else{
        if(tail != null){
            tail.next = new Node(item);
            tail = tail.next;
            //System.out.println("enqueued item: "+item);
            return;
    }
}
```

As explained before the items added to the queue are added as nodes to the tail of the queue. If the queue is originally empty then we set the head element as the node and set it as the tail node as well. Otherwise, we set the tail's next element as the new node to be added to the queue and then we reassign the tail node to the last added node.

2.2 dequeue():

```
public T dequeue(){
        if(head == null){
            return null;
        }
        else{
            Node temp = head;
            head = head.next;
            if(tail == head){
                tail = null;
            }
            //System.out.println("dequeued item: " + temp.item);
            return (T) temp.item;
        }
    }
```

If the queue is empty then we just return null. Otherwise, we create a temporary variable to store the head node which is later going to be returned. After this, we just reassign the head node to the next node that we have in the queue.

3 Tree Iterator using a Queue - Breadth First Traversal

In breadth first search we traverse the tree level by level and check each node and add its child nodes to our queue. Upon the initialization of the queue, we add the root node to the queue. After this whenever we call the next function to retrieve the next element in the tree, we do 2 things. First, we dequeue the first element in the queue and then we add its child nodes to the queue after which we return the dequeued node. We add the child nodes by calling upon the addChildNodes() function. This function takes the node whose child nodes need to be entered into the queue as an argument. After which it checks for the nullity of the argument followed by adding its left and right child nodes to the queue(if they are not null).

```
private void addChildNodes(QTreeNode current){
    if (current != null) {
        if (current.left != null){
            queue.enqueue(current.left);
        }
        if (current.right != null){
            queue.enqueue(current.right);
        }
    }
}

public Integer next() {
    this.next = queue.dequeue();
    addChildNodes(this.next);
    return this.next.value;
}
```

These functions are invoked when we try to iterate through the tree using an itemized loop. This approach iterating through the tree doesn't give us the values inside of the tree in a sorted order like in the case of a depth first search done using a stack as we go node by node in this approach.

4 Static Circular Queue

We can implement the queue data structure in a static manner as well that is by using an array of a fixed size. In the StaticQueue class we create a few variables that help us keep track of some of the properties of our queue. We create 4 integer variables: front, capacity, currentQueuePointer and dequeuCounter. The front keeps track of where the first element in the queue is, while the queue counter keeps track of the position after the last added element. The dequeueCounter is to check how many times elements have been removed from the queue.

4.1 `enqueue():`

For adding elements to the queue we have the enqueue() function. We check for the currentQueuePointer's current value and check if it's the same as the capacity of the queue while the value of the dequeue counter is non-zero. This means that we check if any elements from the front have been removed and if we have run out of space on the other side of the array. If this holds true then we can cycle over to the beginning of the array where there are new vacant spaces from the dequeuing of some elements.

This means we get the cycled back index by getting the modulus value of the currentQueuePointer with respect to the capacity. Then we check if this position is vacant after which we add a new element at that position and also increment the currentQueuePointer position. When we cycle back to the front we can only add the same number of elements as the number of elements that have been removed from the queue. This is because we have finite space inside of the internal queue array and as we have already run out of space on one end of the array it's not possible to add any more elements to the queue. The currentQueuePointer's value has to always be <= capacity of the internal array for us to be able to add elements to the queue.

However, if this was a dynamic queue then we could continue to add more elements indefinitely as we can always copy all the elements from our original queue array to a larger array that can replace the original queue array. This is the case we will discuss in more detail in the next section.

We check for the other case where the currentQueuePointer happens to be greater than the capacity which means that we have run out of space inside of the queue and thus no more elements can be added to it, thus we return at that point. Otherwise, we just add the new element at the currentQueuePointer position.

```
public void enqueue(Integer item){
    if(capacity == currentQueuePointer && dequeueCounter > 0){
        int t = currentQueuePointer%capacity;
        if (queue[t] == null){
            queue[t] = item;
            currentQueuePointer++;
        }
    }
    else if (currentQueuePointer>capacity) {
        return;
    }
    else {
        queue[currentQueuePointer++] = item;
    }
}
```

4.2 `dequeue():`

We remove elements from the queue by using the dequeue() function. Here we check if the value of the currentQueuePointer is the same as the value for the front. This is to check if the queue is empty or not. If the queue is already empty then there is nothing to remove from the queue and we just return from there. Otherwise, we check for the position of the first non-null element in the queue. We do this by iterating through the internal queue array until we encounter a non-null item in the queue upon which we break from the loop. After this, we set the element at this position to null. We increment the class member variable dequeueCounter to keep track of the number of elements removed.

```
public void dequeue(){
    if(front == currentQueuePointer){
        System.out.println("The Queue is empty");
        return;
    }
    else{
        int firstExistingElement = 0;
        for (int i = 0; i < capacity ; i++) {
            if (queue[i] != null){
                break;
            }
            ++firstExistingElement;
        }
        queue[firstExistingElement] = null;
        dequeueCounter++;
    }
}
```

5 Dynamic Circular Queue

The core implementation of a dynamic circular queue doesn't differ so much from our static circular queue implementation. The only difference is that rather than not doing anything and just returning when we run out of space we can expand the queue. We can also shrink the queue to save memory when the queue space doesn't see a lot of usages.

We do the following tasks by calling the functions queueExpander() function to increase the size of the internal queue array when we ran out of space and shrinkQueue() to reduce its size to save memory.

5.1 queueExpander():

We use this function to increase the size of the internal queue array by an array of twice the size which contains all the elements inside the original queue array. We increase the size by twice as we don't want to do the queue-expanding operation very often as it's very costly.

```
public void queueExpander(){
    Integer[] temp = new Integer[capacity*2];
    int index = 0;

    for(int i = front; i < capacity; i++)
        temp[index++] = queue[i];

    for(int i = 0; i < currentQueuePointer; i++)
        temp[index++] = queue[i];
    front = 0;
    currentQueuePointer = capacity;
    capacity = capacity*2;
    queue = temp;
}
```

5.2 shrinkArray()

As we don't want to waste memory space that can be used for more important tasks by the computer, we choose to reduce the size of the internal queue array by half whenever the number of elements in the queue is less than or equal to a fourth of its capacity. We do this using this function. We do this by copying all the values from the original internal queues array and putting all the values inside of it into an array of half the size. We end up with 2 cases based on the value of the currentQueuePointer.

```
public void shrinkQueue(){
    Integer[] temp = new Integer[capacity/2];
    int index = 0;

    if(front < currentQueuePointer) {
        for(int i = front; i < currentQueuePointer; i++) {
            temp[index++] = queue[i];
        }
    }

    if(currentQueuePointer < front) {
        for(int i = front; i < capacity; i++)
            temp[index++] = queue[i];

        for(int i = 0; i < currentQueuePointer; i++)
            temp[index++] = queue[i];
    }

    front = 0;
    currentQueuePointer = index;
    capacity = capacity/2;
    queue = temp;
}
```