Assignment 7: Queues

Mahad Ahmed

Fall 2022

Introduction

In this assignment we implemented a queue data structure. Queues is something we are pretty familiar with as we are in queues in the stores and pretty much everywhere in everyday life. The point of having a queue is to organize and be fair. FIFO, stands for *first in first out* is a fair system that we are familiar with. Or maybe we call it first come first serve. The queue data structure is pretty much the same concept.

Queues

A queue is as I mentioned a linear data structure that is open on both ends. It uses FIFO order. A queue is defined as a list where we only add at one end and delete on the other. So if we take a look at figure 1, we have a *rear* which is were we add or called in the figure *Enqueue*. Then we have the front where we delete or *Dequeue*. These should be interpreted as a linked list and not arrays. This is the simplest way to implement a queue. Array implementation will come further down in the report.

First implementation of queue using linked list

This first implementation goes as follows:

- Set a pointer to the first node which is called head
- When our list is empty we need to create a new node which is called the head
- Then to add we we keep the head pointer pointing to the first node but say the next null position is where we will put our new node. In this case it will be node.next
- Then point the head node to the one in after

To remove lets consider we have an linked list with some nodes:

- We follow the FIFO order so we already have our pointer pointed at our first node so we just return it's value
- Then update our head to the node after

Drawbacks of the first implementation

To remove a node in our queue is constant as you just return the value of the head and point the head pointer at the next node. However when adding we don't have the address of the last node so we have to traverse through the entire list to find it. Then add the new node at the tail. Which makes this cost very expensive. Linearly expensive. We can do better.

A second implementation of a queue using linked list

The implementation goes as follows:

- We start as the implementation before but this time we have a pointer also to the last node in our list
- If we look at figure 2, we can see that our front pointer is pointing to the head and the head in turn points to the next item. Not the other way around as the before gone implementation.

To add/Enqueue:

- Basically the same as before but now the current rear will point at the new node we are adding and rear pointer will be updated to the new node

To remove/Dequeue:

- They only thing we need to do is return front node's value and update our front pointer to the next node

Why is this implementation better?

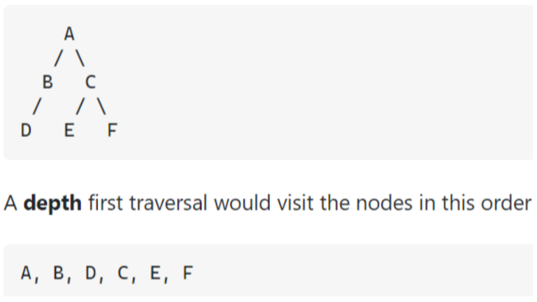
This implementation is better because of how we are implementing our *enqueue* method. In this implementation we have a rear pointer which will help us keep track of the last node in our list. Now when we want to add a node to our list we don't have to traverse through the entire list because we already have the address of the last node. We just take our last node then point it to the new node and update the rear pointer to the new node's address. This implementation will be in constant time, *O(1).* Instead of linear time.

The dequeue method will still be the same which had the time complexity of constant time, *O(1).*

For these above mentioned reasons is this implementation better than the one before.

Depth first traversal

Depth first traversal or for short DFT is a search method. We implemented it when doing assignment 6 with the binary tree. So we go depth first i.e down levels. It kinda acts like it want to go as far from the starting point asap. This method also uses a stack to remember where it shall go when it hits a dead end. Depth first is mainly used when we know if something is deep within a graph or a tree.



```
    A
   / \
  B   C
 /   / \
D   E   F
```

A **depth** first traversal would visit the nodes in this order

```
A, B, D, C, E, F
```

Figure 1: Example output of DFT

Breadth first traversal

Breadth first traversal is a new method we implemented using queue instead of a stack. I like to think this method like a big landscape with high an low points. Think of the traversal as a flood. First it will flood the low points then go on to the higher and higher points. By this I mean if we have a binary tree and we wanted to implement BFT then we would start by looking at the root. Then look at BOTH it's children not only the left one as DFT. We look at both (or more) child nodes and traverse through everything level to level without missing anything. Using BFT is better when you know if the will be close to the root/vertex.

This policy uses FIFO instead of LIFO (last in first out). The same example as figure 3 but the output would be, *A, B, C, D, E, F.*

Queue using arrays

As before when we implemented a stack using both linked list and arrays we can now implement a queue using an array. The idea is pretty straight forward but there are some corner cases I had to cover.

The basics

To start we need to first create an array with a size n. Then take two variables lets call them *first* and *rear* which will be 0 which means the queue is empty. *first* is the index of the first element and *rear* is the index of the next available spot.

Enqueue

A simple add operation. First will check if the queue is full or not. Then our rear pointer will gives us the index of the next available spot. After we add the element we need to increment our rear pointer. If the rear is equal to our capacity then we will simple throw an exception saying the array is full.

Dequeue

To remove something we need to first check if the there is something in the queue. When that is fulfilled then our first pointer gives us the index of the first element which can now be removed. Then when the first is removed we need to shift all elements one step to the left. This will be very costly but will do for a first implementation. But we can do it better. We can just increment the first pointer instead of shifting every element to the left. But what to do with the empty spaces when we remove elements at the front? Can we fill up those spaces? For example when first and rear pointer are equal, what shall we do? Let's wrap our head around this problem.

Wrap around

The problem I had to solve was what to do when our first pointer was equal to our rear pointer. A solution could be to allocate a new larger capacity array and dump our elements in there. But what if we instead just add the new element to index 0? Then just carry on as we did before? This is what I have implemented a so called circular queue. When we are adding and the next available pointer has reached the capacity of our array we simply reset it. I also implemented a boolean which tells us when our queue is empty. By doing this we can use up all available space in our array.

Some corner cases to consider

- If *first* is equal to *rear* the queue is empty. The initial queue will have both set to 0. Trying to remove an item from an empty queue will simply return null.
- When increment *rear* (after adding a new item) it should be set to 0 (not to n) if *rear* is equal to *n - 1*.
- If *rear* is equal to *first* after being incremented the queue is full and you have problem.
- When increment *first* (after remove) it should be set to 0 (not to *n*) if *first* is equal to *n - 1*.

Dynamic queue

In this section we solved the case when our queue is full. The solution is rather trivial. Let say we have our rear pointer incremented and realize that rear is equal to first pointer. Then we just allocate a new array twice a large and copy over all our values. The copying will be done in the order of the queue, FIFO.

Some code

Here is how I implemented the different tasks. It's not everything but some of the implementation.

Enqueue

```
//Enqueue

public void add(int newInQueue){
    array[nextAvailableSpot] = newInQueue;
    nextAvailableSpot++;
    queueIsEmpty = false;
    if (nextAvailableSpot == capacity)
        nextAvailableSpot = 0;
    if (nextAvailableSpot == firstInQueue){
        int[] newArray = new int[capacity*2];
        for (int i = 0; i < capacity; i++){
            newArray[i] = array[firstInQueue++];
            if (firstInQueue == capacity)
                firstInQueue = 0;
        }
        firstInQueue = 0;
        nextAvailableSpot = capacity;
        capacity = capacity * 2;
        array = newArray;
    }
}
```

Dequeue

```
  //Dequeue
public int remove(){
    if(queueIsEmpty)
        return 0;

    int temp = array[firstInQueue];
    array[firstInQueue] = 0;
    firstInQueue++;

    if (firstInQueue == capacity)
        firstInQueue = 0;
    if (nextAvailableSpot == firstInQueue)
        queueIsEmpty = true;
    return temp;
}
```

Breadth first traversal

```
      public Integer next() {
       if (hasNext()) {
           if (stack.head.binNode.left != null)
               stack.add(stack.head.binNode.left);
           if (stack.head.binNode.right != null)
               stack.add(stack.head.binNode.right);
       }
       return stack.remove().key;
   }
```