Queues

Nariman Haidar

Fall term 2022

1 Introduction:

In this assignment we are going to learn about queues. First we are going to implement a queue using two different approaches, using an array and using a linked list. And then we are going to compare queue with a stack that we learned about it earlier in previous lessons.

2 Method:

A linked list:

A queue is a linear data structure that are also called FIFO, first-in-first-out, that describes the functionality. In the queue there is two main operations: en-queue and de-queue.

We can implement the queue using an array or linked list. In singly linked list implementation, we have two pointers in linked list front and rear. The front points to the first item of the queue and rear points to the last item.

In queue we have two main operations method en-Queue and this operation is going to add a new node after rear and moves rear to the next node.

The other method is de-Queue and this operation is going to remove the front node and moves front to the next node.

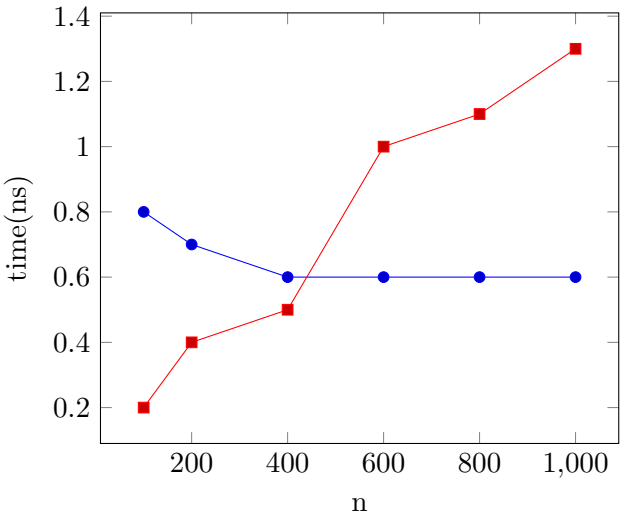
To implement the Queue first I create a class node with data members integer data and next. Then I write the constructor that takes an int x as a parameter and then I set data to x and next to null. Then I create a class Queue node front and rear. Then I write the constructor Queue sets front and rear as null. Then I write the method en-Queue with parameter x Then create an object new node and then If rear is null then set front and rear to new node and return. Else set rear‘s next to new node and then move rear to new node. Then I write the method de-Queue. There is two different ways to write de-queue method. The first one is you need to run to the first element to delete it, but the second method we can use a pointer that

points directly to the first element and then we can easily delete it without having to run through the entire list. To implement the first one so we do that:if front is set to null just return.Then create a new node and then set this new node to front and set front to its next. If front is null then set rear to null then delete this node. The other method de-queue we can just make a prev node and make it to pointer to the previous node and set it to the null and then make a current node and set it to the front, then while current next is not null so set the prev to the current and current pointer to the next. And if the previous element is null so the front is null or previous next is null so return the item. The time complexity to en queue an element is constant O(1). The time complexity to de queue an element is constant O(1). But if we wanted to remove the tail, so the time complexity would be O(n), because we are going to go through the whole list to get the tail so it takes o(n) and that is also because we need to find a new tail to this list since the old tail doesn't have access to the previous element in a list. In our situation the en-queue method takes time complexity constant o(1). In the first de-queue that I wrote in this code takes o(n). In the other method de-queue that has pointers so it takes o(1). And we can see the results in the graph below.

code: in this code we can see the en-queue and de-queue methods.

```
public void queue(T item) {
    Node newNode = new Node(item, null);
        if (this.head == null) {
            this.head = newNode;
        }
       if(this.last != null){
           this.last.next = newNode;
       }
        this.last= newNode;
    }
public T deQueue() {
   if(head == null){
       System.out.println("\n the queue is empty");
    }
    T temp= null;
   if(this.head != null){
       temp = this.head.item;
       this.head = this.head.next;
    }
    return temp;
}
```

Graph:de-queue



Breath first traversal:

In the previous assignment we made a binary tree with stack and we made a depth first traversal in the stack so it was going through all elements to left and then to the right. But this time we are going to make a binary tree with queue and we are going to go breath first since we traverse all nodes of one level before going to the next level that is deeper. In this algorithm it works in this order first it going to the first level and visit all nodes then to the second level and visits all nodes until the last level, so it visited first level and that is root and then the children that are in the second level then the grand children that are in the third level until the last level.

To implement this algorithm through the queue so we first We are going to create a queue and then the queue is going to save all elements width of first traverse pass through. Then de-queue the head of the queue, then en-queue all of its nodes that are in the same rad from left to right. We are going to do that until the queue is empty.

It is going to start with point root into the traversal queue. Then its going to traversal queue all children of the root from left to right, then all nodes in the second level until the last level in the tree.So now we can see all the nodes in the queue in breadth first order.

Time complexity of algorithm is O(n).

An array implementation:

A queue can also implemented using array. Now I am going to explain how we can implement a queue using a circular array. First we are going to write a class that is circular array queue and I make it generic. In this class initialize the size to 10 and we set the front and the rear to the array and object array. Then write constructor and then write the method is full and that is when the front equal to zero and the rear is -1. Then the method if the array is empty and that is when front equal to -1. Then write the method add, first we are going to check if the array is full so display that else check if the front equal to -1 so increment the front and make it zero then increment then assigning a element to the queue at the rear position using modulo. Then the de-queue method first I check if the array is empty so display, else if the front and rare is equal so set the front and rear to -1 else reset the queue after deleting it and return the element. In this circular array en-queue we make if we delete the element from the front so the index 0 is going to be used another time and with using the modulo operation must be applied to the pointer pointing to the last element. In the "dequeue" method, the calculation operation will be modulo the same operation that we used in the en-queue method. Time complexity in the best case to en-queue is o(1). Time complexity in the worst case to en-queue is o(n). Time complexity in the best case to de-queue is o(1). Time complexity in the worst case to en-queue is o(n).

dynamic queue:

To write dynamic queue first I write the method that is going to resize the size of array and double it if the array is full and we wanted to en queue a new element. In this method that take a new size like a parameter I initialize the old size to the array size that we have and then the new size is the double of the old size. Then I write the queue method that first check if the size is the same size to our array so resize the array and double it, else increment the rear and then set the rear to the data. If the front is equal to -1 so set the front to the zero. Then I write the method de-queue, and first it check if the array is empty so display it, else I write an item out it takes array front then increment the front with one then return the element that will be out.