Queues

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the seventh assignment "Queues" on the course "ID2021" TIDAB, KTH. This assignment is regarding program queues. The queues follow the principle of first in, first out meaning that the first item that enters the queue should be the first to leave as well this in contrast to the stack implementations we have used previously. We will in this assignment look at 5 different implementations of a queue in programming and discuss these:

1. Single linked list with just a pointer to the beginning of list.
2. Single linked list with a pointer to the start and the end of the list.
3. A Breadth first approach to the binary tree from the previous assignment where we use a queue instead of a stack to print the elements of the tree.
4. An array approach to the queue algorithm, that has a wrap around feature.
5. A similar array, but it is also dynamic i.e. it expands when the list is full.

1. Single linked list with 1 pointer

The single linked list queue was very similar to how we worked with linked lists in the previous assignments. This time the suggested structure was different from the previous assignment where the list and the node class were not separated. Changing the mindset to this new structure was the biggest challenge when implementing this queue. It works in a very straight forward manner where an element is added and then the following items are added as a tail. When we want to take an item we take the head of the list and set the next item in the queue as the new head. We solve the special

case of an empty list through checking if the head of the list is equal to null. The code implementation for the add and remove function is given below:

```
public void add(Integer item){
    Queue.Node next = head;
    Node new_node = new Node(item, null);
    if (head == null){
        head = new_node;
    }
    else{
        while(next.tail != null){
            next = next.tail;
        }
        next.tail = new_node;
    }
}
public Integer remove(){
    Integer temp = this.head.item;
    head = this.head.tail;
    return temp;
}
```

The problem with this implementation is that when we add an element to the queue we have to go through the whole list which gives us a linear time complexity of O(N). Removing an item from the queue however is a constant function for all list sizes since we just take the first item in the queue. To make this even better we add another pointer which is the next task.

2. Single linked list with 2 pointers

To improve on the linked list queue we add another pointer to the last element in the list, this way we can make both the add and remove function have a constant time complexity of O(1). This however adds another special case where the pointers can be on the same value. At first there was trouble separating these cases and sometimes elements were added twice. Implementing the add operation with first "if" and then "else if" solved this problem. The implementation of the updated add function is given in the code below:

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

```
            last = new_node;
        }
        else if (head == last){
            last = new_node;
            head.tail = last;
        }
        else{
            Queue2.Node temp = last;
            last = new_node;
            temp.tail = last;
        }
    }
```

The linked list implementations of a queue were easy to grasp and worked well. The next task is to use them on a binary tree.

3. Breadth first traversal

In the previous assignment we used depth first traversal of the binary tree. Now that we have a queue we can implement breadth first traversal where you traverse through the binary tree level by level. Firstly for this to be possible the queue was changed to hold a binary tree node. The iterator is then able to be changed into using a queue instead of a stack. For the breadth first iteration firstly the root is added to the queue then each node is taken from the queue in order and put into a pointer called next. We check if the node next has any element to its left and then right and then add these to the queue. By repeating this each element will have the pointer next pointing to it and we will iterate through the list level by level from left to right. The code for this iterator is given below:

```
    public TreeIterator(BinaryTree.Node root){
        next = root;
        queue = new Queue();
        queue.add(root);
    }
    @Override
    public boolean hasNext(){
        return queue.head != null;
    }
    @Override
    public Integer next(){
        if (!hasNext()){
            throw new NoSuchElementException();
        }
```

```
        next = queue.remove();
        if(next.left() != null){
            queue.add(next.left);
        }
        if(next.right() != null){
            queue.add(next.right);
        }
        return next.key;
    }
```

4. Array queue

The array queue if fast and it is easy to access any element in the array at all times, it however creates many special cases that need to be addressed. It took a lot of testing to find all cases, this array has been implemented with "wrap around" meaning that if elements at the start of the array queue are removed we can place new elements there. To do this we need to keep track of the order of which the elements entered the queue. This is done with a first and last pointer. To enqueue and element we put it where the last pointer is and increment it. In the case of reaching the final slot in the array we add a conditional statement that checks if the first slot of the array is empty, then we can wrap around and ad it there. The dequeue operation similarly checks if we have reached the end of the list and if so moves its pointer to the start of the array. To avoid overwriting, the program first checks if the two pointers are equal, if this is the case then we cannot add more elements and the queue is full. If they are equal when trying to dequeue something we return that the list is empty.

5. Dynamic array queue

The next step is to add functionality for dynamic adaptation of the queue if it becomes full. In the implementation two separate cases was added for copying over the elements to a new larger array. The array is doubled when full and this is done when we need cannot wrap around or if the pointers reach each other. When the list is doubled it is important to keep track of where the pointers are to copy them over to the new bigger array in the correct order. Especially if the pointer to the last element is behind the pointer to the first element. The code for adding and removing items to the queue is given below:

```
public void Enqueue(int data){
        if (queue[0] == null && last == size){
            last = 0;
```

```
        }
        if (queue[0] != null && last == size){
            System.out.println("Dubbel addera");
            fullDouble();
        }
        queue[last] = data;
        last++;
        if (first == last){
            wrapDouble();
        }
    }
    public void Dequeue(){
        if (first == last){
            System.out.println("Tom lista");
            return;
        }
        if(first != (size-1)){
            queue[first] = null;
            first++;
            if (first == last){
                wrapDouble();
            }
        }
        else{
            queue[first] = null;
            first = 0;
        }
    }
```

The code for doubling and creating a new array is given below, the second simpler one is for when the last pointer is at the end of the array and we cannot wrap, then the list is just copied over in order:

```
    public void wrapDouble(){
        size = size*2;
        Integer[] temp = new Integer[size];
        int a = 0;
        for(int i = first; i < queue.length; i++){
            temp[a] = queue[i];
            a++;
        }
        for(int j = 0; j < (last-1); j++){
            temp[a] = queue[j];
            a++;
```

```
        }
        first = 0;
        last = a;
        queue = temp;
    }
    public void fullDouble(){
        size = size*2;
        Integer[] temp = new Integer[size];
        int a = 0;
        for(int i = 0; i < queue.length; i++){
            temp[a] = queue[i];
            a++;
        }
        first = 0;
        last = a;
        queue = temp;
    }
```