Queues

Ayda Ghalkhanbaz

Fall 2023

Introduction

In this assignment, a common data structure named queue is implemented. We will first look into the implementation details of this structure and discuss its time complexity for different approaches.

Queue

Queues are, as mentioned before, a very common data structure, whether in real life or in computer science. Queues follow a FIFO principle, meaning first-in-first-out, in contrast to stacks that use a LIFO (last-in-first-out) principle. Every queue implementation has two main methods: *enqueue()* and *dequeue()*. The method *enqueue()* adds a new item to the end of the queue and *dequeue()* removes one from the beginning. In this assignment we will implement queues using linked lists and arrays. Figure 1 illustrates a basic structure of a queue constructed with a linked list.

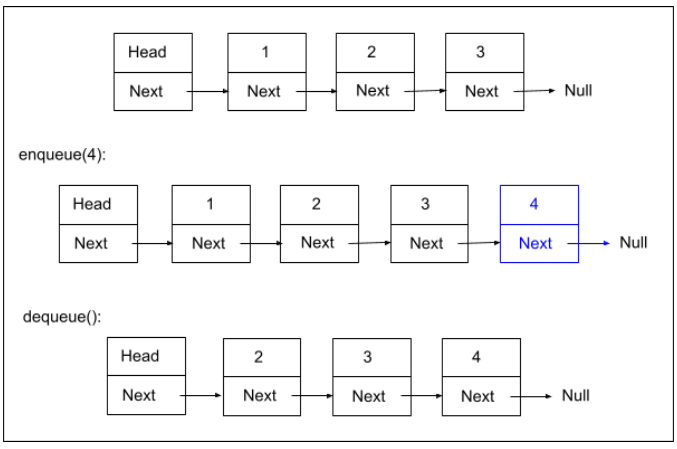

Figure 1: *A basic illustration of a queue and its enqueue() and dequeue() methods*

Linked List Implementation

The queue to be implemented in this task has the same structure as shown in figure 1, in other words it only has a pointer to the first node in the queue. The implementation of this queue included a constructor and three methods; *isEmpty()*, *enqueue()* and *dequeue()*.

The method *isEmpty()* returns *'true'* if the queue is empty, *'false'* otherwise. The methods *enqueue()* and *dequeue()* have the same functionality as briefly explained before. The *enqueue()* method accepts an element and adds it to the end of the queue and the *dequeue()* method removes the first element from the queue and returns the removed element.

The implementation of this queue was quite simple but not very efficient (we discuss the reason in the discussion section). So the second thing to do in this part, was to improve the implementation by adding a further pointer that keeps track of the last element in the queue. The structure remains the same as illustrated in figure 1, with this difference that there is another pointer *last* that points to the end of the queue. This allows a more efficient insertion operation at the end of the list. The code snippet below demonstrates the *enqueue()* method and how the last pointer is updated when adding a new element.

```
public void enqueue(T element){
        Node addedItem = new Node(element, null);
        // if the queue is empty
        if(head == null)
            // set head and last to the first element
            head = addedItem;
        else
            // next pointer of the last item should
            //  point to the new element
            last.next = addedItem;
        // set the new added item as the last element
        last = addedItem;
        size++;
    }
```

Breadth-First Traversal

In the previous assignment, we saw how a stack allowed us to have a depth-first traversal. By employing the characteristics of a queue, we can traverse a tree by starting from its root, exploring all of the nodes at the current depth before moving on to the next level of depth. Figure 2 illustrates a

breadth-first traversal in a binary tree.

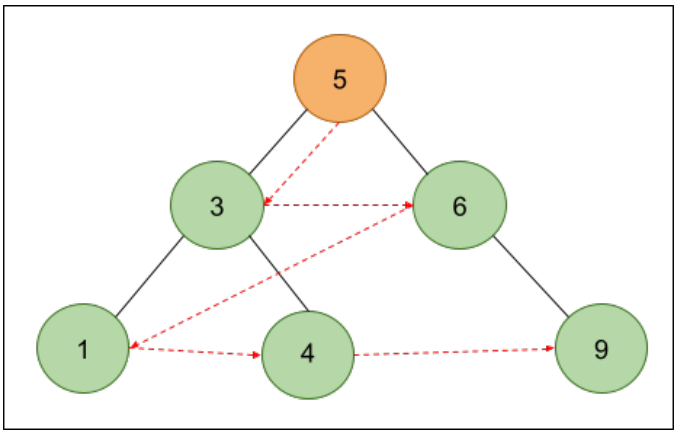

Figure 2: *A basic illustration of breadth-first traversal in a binary tree*

In the implementation of Breadth-first traversal (BFT), the same *TreeIterator()* used in the previous assignment has been employed. The first modification was that the *TreeITerator()* should have used a queue of *BinaryTree* nodes instead of a stack. Consequently, the constructor was adjusted to initialise the queue. The method *hasNext()* was unchanged since it only checks whether there is a next node in the tree or not and returns true/false. The *next()* method on the other hand, was completely modified. The following code snippet shows the implementation of *next()* method.

```
@Override
    public Integer next(){
        if(!hasNext())
            throw new NoSuchElementException();

        Node ret = next;
        if(next.left != null)
        // enqueue the left child
            queue.enqueue(next.left);
        if(next.right != null)
        // enqueue the right child
            queue.enqueue(next.right);

        next = queue.dequeue();
        return ret.key;
    }
```

As we can see in the code, we first check if there are any nodes in the tree and if it is empty an exception is thrown. Secondly we presume there is node to be dequeued from the queue and save it in the *ret* node. After that, we check if the node has any left and right children, in such a case, we add them to the queue and lastly dequeue the *ret* node and return it.

Array Implementation

As we implemented stacks using both linked lists and arrays, the next part of this assignment involved applying the same approach to queues. We've already covered how to implement queues using linked lists in the previous section. Now in this part of the assignment, we will look into how queues can be constructed by arrays.

Initially, an array with a fixed size n was implemented and later improved by using a dynamic array instead. Since both implementations share the same structure, we will only look into the implementation details of the dynamic arrays which is an enhanced version of the queue using a fixed-size array.

Dynamic Queue

The implemented dynamic queue includes a constructor, three main methods; *isEmpty()*, *enqueue()* and *dequeue()*, along with two helper methods; *expand()* and *shrink()*. In addition to maintaining two pointers first and last,the constructor also keeps track of the count of elements in the queue.

The method *isEmpty()* returns true if there is no element in the queue and false otherwise.

The *enqueue()* method which adds an item to the end of the queue is demonstrated in the code snippet below.

```
public void enqueue(Integer item){
    // if queue is full
    if(elementCounter == this.size)
        // doubling the size of array
        expand();

    queue[last] = item; // item added to the end
    // moving the last pointer(wrap arround)
    last = last + 1 % this.size;
    elementCounter++;
}
```

As it is evident in the code above, if the array has no empty spots then it calls the method *expand()*. The *expand()* method will then copy all the elements to a new array with a doubled size. After this process, the new item can now be enqueued to the array. The tricky part of this task was to update the *last* pointer and make it wrap around to the beginning of the array when it reaches the end. The modulo operator allows us to maintain this circular nature of the queue. In other words the modulo operation helps us to keep the pointer in the valid index range.

The *dequeue()* method returns 'null' if we try to remove an item from an empty queue. Otherwise it saves the item in a return variable, sets its position to 'null', and increments the *first* pointer to point to the next element in the queue. Incrementing the *first* pointer utilises the modulo operation as well in order to keep the pointer within the valid range. Lastly the method checks if the number of elements have decreased to a quarter of the array size. If so, the array size is halved by calling the *shrink()* method and the dequeued element is returned. The provided code snippet below illustrates the implementation details.

```
public Integer dequeue(){
    // queue is empty
    if(isEmpty())
        return null;

    Integer ret = queue[first];
    queue[first] = null;
    //moving the first pointer to next item
    first = (first + 1) % size;
    elementCounter--;
    // if queue is quite empty
    if(elementCounter <= (this.size/4)){
        shrink();
    }
    return ret;
}
```

The *shrink()* method has a similar structure to the *expand()*. It copies all the elements to a new array with a halved size and resets the *first* and *last* pointers so they point to the first and last item in the queue.

Discussion

In the primary implementation of the queue with linked lists, where we only had a head pointer, the time complexity of adding a node to the queue

is $\mathcal{O}(n)$, where n is the size of the queue. The reason is that we need to traverse the whole queue to reach the end and then add the new node at that position. However, adding a new node in the second implementation (with two pointers pointing to the first and last nodes) takes a constant time and has a time complexity of $\mathcal{O}(1)$. This is because we can access the last node directly, without traversing the queue. On the other hand, removing a node from the queue is a constant operation because we have access to the first node in both implementations. So the time complexity of dequeuing a node is $\mathcal{O}(1)$.

In the dynamic array implementation of queues, the time complexity of both enqueuing and dequeuing is typically $\mathcal{O}(1)$ but it may involve expanding or shrinking the array. The resize operations are costly due to copying the elements and writing them to the new arrays, but they occur infrequently. That's why we consider the amortised time complexity which is $\mathcal{O}(1)$.

Overall, the choice between the two implementations depends on several factors such as, memory usage, ease of implementations, the size of data etc. For instance, linked lists are usually preferred when the size of the queue is not predictable and it may need to be resized frequently.