Lab 9 Queues

Weichen Chai

Fall 2022

Introduction

This lab report seeks to answer the questions presented in ”Queues” for ID1021. In this assignment, the goal is to analyse the advantage of disadvantages of the linked list, static array and dynamic arrays technique for queues.

Background

Queues are somewhat similar to stack, which follows the LIFO (last in first out) method, but in the case of the queue, it follows the FIFO(First in first out) method, where the first element inserted will be the first element to be removed.

Tasks

Linked list

One of the simplest ways to implement a queue is the usage of linked lists. In order to analyse the cost of removing and adding elements, we must first implement those functions. When implementing add function with one pointer, it will need to traverse the entire list to find the last node, the draw back of this method is that it is very inefficient and expensive, as this causes the time complexity to be O(n).

In order to avoid and optimize this, we use two pointers, one at the head and one at the tail. the add function adds at the tail of the queue. In one scenario, the element will be inserted into an empty queue, in this case, head is equal to null and we initialize both head and tails. In the case when head=null is false, the pointer for tail will be moved to the new tail node.

The add function is as follows.

```
public void add(int value) {
  Node n = new Node(value,null);
```

```
        if (head == null) {
            head = n;
            tail = head;
        }
        tail.next = n;
        tail = n;
        }
```

The add function inserts an element into the end of the queue, and therefore, it will have a constant time complexity of O(1).

The remove function throws exception the queue is empty. If the queue is not empty, then delete the element the head pointer is indicating. The remove function in both cases with one pointer and two pointers will always have a constant time complexity, as it only uses the head pointer. This indicates that the remove function also has a time complexity of O(1), as it only removes the first element in the queue.

```
 public Integer remove() {
        Integer re;
        if (head == null) {
            throw new NoSuchElementException();
        }
        re = head.item;
        head = head.next;

        return re;

    }
```

Breadth first traversal

In this task we will implement breath first traversal method where we traverse through one level before going to the level that is deeper than the current one, this is a bit similar to the depth first traversal used in trees, which focuses on depth before continuing on the same level.

Breadth first traversal is often used then attempting to find nodes that's closes to the root. The iterator for this process will have a next function which will queue to the left, then we check if it has a right branch, and return the current node. Shown in the following.

```
if(queue.head.item.left != null){
 queue.add(queue.head.item.left);
}
if(queue.head.item.right != null){
```

```
queue.add(queue.head.item.right);
}
```

Queues array

To implement a static array queue, we must use pointers k and i, where k is the first free slot, where we add items, and first element is i. In the add function, we have when the available slot in the array is not null and when i is equal to k to be when the array is full, as another case when i and k would be equal is when the array is empty. When increments k, it is set to 0 when k is equal to is array length - 1,

```
  public void add(Integer value) {
        if(array[k] != null&& i==k )
            System.out.println("Full");
        else{
            array[k] = value;
            if (k == array.length - 1) {
                k = 0;
            }
            else k++;
        }

    }
 public Integer remove() {
        Integer data;
        if(i == k) {
            System.out.printf("Empty");
        }
        if(i == length - 1) {
            data = array[i];
            array[i] = null;
            i = 0;
        } else {
            data = array[i];
            array[i++] = null;
        }
        return data;
    }
```

Finally we attempt to implement queues with the array method, however. This will only require more lines of code under add, as remove will be the same if not worked with in a dynamic manner. In order to implement a dynamic array queue, we can simply add expandsize() under the add function, in the location of the print ”full” line of code for static array queue. The code is the following.

```
private void expandsize() {
      Integer[] Big = new Integer[2*length];
      for (int j = i; j < length; j++) {
          Big[j - i] = array[j];
      }
      for (int j = 0; j < i; j++) {
          Big[j + length - i] = array[j];
      }
      array = Big;
      k = length;
      length = 2 * length;
      i = 0;
  }
```

The time complexity for static array is the same as when using a linked list, however, during the resizing of the array, the time complexity will become O(n), but will quickly return to O(1), the time complexity for deletion process will be the same as linked list queue of O(1). Even though the time complexities for linked list and array queues are somewhat similar, there will differences in time for execution. Normally, arrays takes longer for both add and delete, and linked list is much faster than arrays, so this would probably be the same case here, even though linked list might take extra time for creating a new node when adding to the list.