DT183G - Datastrukturer och Algoritmer

Laboration 2 - Q1

MITTUNIVERSITETET

Institutionen för kommunikation, kvalitetsteknik och informationssystem (KKI) Östersund

Author	Samuel Thand	sath2102@student.miun.se
Supervisor	Raja-Khurram Shahzad	Raja-Khurram.Shahzad@miun.se
Examiner	Raja-Khurram Shahzad	Raja-Khurram.Shahzad@miun.se
Program	Programvaruteknik, 180HP	
Course	DT183G, Datastrukturer och Algoritmer	
Field	Computer Engineering	
Semester	VT 2023	

Table of Contents

1. Big O of Q3

SamuelFixedSizeStack.java

```java
package SamuelDatastructures;

import java.util.EmptyStackException;

public class SamuelFixedSizeStack<T> {

   T[] items;
   int top;

   @SuppressWarnings("unchecked")
   public SamuelFixedSizeStack(int size) {
       items = (T[]) new Object[size];
       top = 0;
   }

   public void push(T item) throws StackOverflowError { // O(1)
       if (top >= items.length) {
           System.out.println(items.length);
           throw new StackOverflowError("The stack is full");
       }
       else
           items[top] = item;
           top++;
   }

   public T pop() throws EmptyStackException { // O(1)
       if (top <= 0)
           throw new EmptyStackException();
       else
           top--;
           T item = items[top];
           items[top] = null;
           return item;
   }
```

```
    public T peek() { // O(1)
        if (top-1 <= 0)
            throw new EmptyStackException();
        else
            return items[top-1];
    }

    public int size() { // O(1)
        return top;
    }

    public boolean isEmpty() { // O(1)
        return top <= 0;
    }
}
```

All operations in this stack data structure have a O(1) time complexity, which is typical for stacks. No operation scale from the size of the input. Pushes, pops, peeks, size, and isEmpty calls take the same time to execute regardless of the size of the stack.

Q3Stueue.java

```
package SamuelDatastructures;

import java.util.EmptyStackException;

public class Q3Stueue<T> {

   private SamuelFixedSizeStack<T> stack1;
   private SamuelFixedSizeStack<T> stack2;

   public Q3Stueue(int size) {
       stack1 = new SamuelFixedSizeStack<>(size);
       stack2 = new SamuelFixedSizeStack<>(size);
   }

   public void push(T item) { // O(1)
       stack1.push(item);
   }

   public T pop() { // O(1) or O(N)
       if (!stack1.isEmpty()) {
           return stack1.pop();
       } else if (!stack2.isEmpty()) {
           while (!stack2.isEmpty()) {
               stack1.push(stack2.pop());
           }
           return stack1.pop();
       } else {
           throw new EmptyStackException();
       }

   }

   public T deque() { // O(1) or O(N)
       if (stack2.isEmpty()) {
           while (!stack1.isEmpty()) {
               stack2.push(stack1.pop());
           }
       }
       return stack2.pop();
```

```
    }
}
```

The push operation has a time complexity of O(1) since it is a normal stack push operation, which does not scale with the size of the input or the size of any stack.

The pop and deque operations have a time complexity of O(1)/O(N) - constant or linear time complexity. If stack1 is not empty, the pop operation is O(1) - since it can return the top item on stack1 without doing any other operations. If stack1 is empty, and stack2 is not empty, the operation iterates over each item of stack2 and executes a single operation for each item - which results in an O(N) time complexity. If stack2 is not empty, the deque operation is O(1) - since it can return the top item on stack2 without doing any other operations. If stack2 is empty, the operation iterates over each item of stack1 and executes a single operation for each item - which results in an O(N) time complexity.

2. Big O of Q4

SamuelLinkedList.java

```java
package SamuelDatastructures;

import SamuelDatastructures.Node;

import java.util.Iterator;

public class SamuelLinkedList<T> implements Iterable<T> {

   private class SamuelLinkedListIterator implements Iterator<T> {
       private Node<T> current = head;

       @Override
       public boolean hasNext() { // O(1)
           return current != null;
       }

       @Override
       public T next() { // O(1)
           T item = current.getItem();
           current = current.getNextNode();
           return item;
       }
   }

   private Node<T> head;

   public SamuelLinkedList() {
       this.head = null;
   }

   @Override
   public Iterator<T> iterator() { // O(1)
       return new SamuelLinkedListIterator();
   }
```

```
 public void add(T item) { // O(1) or O(N)
    var node = new Node<>(item);

    if (isEmpty()) {
        this.head = node;
    } else {
        var currentNode = this.head;
        while (currentNode.getNextNode() != null) {
            currentNode = currentNode.getNextNode();
        }

        currentNode.setNextNode(node);
    }
}

public T remove(T item) { // O(1) or O(N)
    if (isEmpty()) {
        return null;
    } else if (head.getItem().equals(item)) {
        var removedItem = this.head.getItem();
        this.head = head.getNextNode();
        return removedItem;
    } else {
        var nextNode = head.getNextNode();
        var currentNode = head;
        while (nextNode != null) {
            if (nextNode.getItem().equals(item)) {
                var removedItem = nextNode.getItem();
                currentNode.setNextNode(nextNode.getNextNode());
                return removedItem;
            }
            currentNode = nextNode;
            nextNode = nextNode.getNextNode();
        }

        return null;
    }
}

public T removeHead() { // O(1)
    if (isEmpty()) {
        return null;
```

```
        } else {
            var item = head.getItem();
            head = head.getNextNode();
            return item;
        }
    }

    public int size() { // O(N)
        int count = 0;
        Node<T> currentNode = head;
        while (currentNode != null) {
            count++;
            currentNode = currentNode.getNextNode();
        }
        return count;
    }

    public boolean isEmpty() { // O(1)
        return this.head == null;
    }

}
```

The hasNext, next, iterator, removeHead, and isEmpty methods all have O(1) time complexity since the operations do not scale based on the input or the size of the data structure.

The add method has a time complexity of O(1) if the list is empty and O(N) if not - since it iterates to the end of the list and inserts items at the end. This is a subpar performance for a typical linked list since inserts should be O(1) for normal inserts, and the structure should be optimized to reach this time complexity. Inserts at the end could be accomplished in constant time by keeping track of the end of the list.

The remove method has a time complexity of O(1) if the head equals the requested item for deletion, O(N) if not - since it makes use of searching to find the requested item, which has a worst-case time complexity of O(N) if the item is the last one. This performance is expected for a typical linked list since the removal of the head is usually O(1) while the removal of another item is normally O(N) because of iterative search.

The size method is O(N) since it iterates through the entire list and counts items. Getting the size could be optimized to O(1) by keeping track of this data in the structure instead of computing it for each call.

Q4CustomerQueue.java

```
package SamuelDatastructures;

public class Q4CustomerQueue {

   private final SamuelLinkedList<Q4Customer> regularQueue;
   private final SamuelLinkedList<Q4Customer> expressQueue;

   public Q4CustomerQueue() {
       regularQueue = new SamuelLinkedList<>();
       expressQueue = new SamuelLinkedList<>();
   }

   public void addCustomer(Q4Customer customer) { // O(N)
       boolean isExpress = customer.itemsPurchased() <= 5;
       if (isExpress) {
           expressQueue.add(customer);
           System.out.printf("Customer %d: %s with %d items entered the express
queue as number %d.\n",
                   customer.customerID(),
                   customer.name(),
                   customer.itemsPurchased(),
                   expressQueue.size());
       } else {
           regularQueue.add(customer);
           System.out.printf("Customer %d: %s with %d items entered the regular
queue as number %d.\n",
                   customer.customerID(),
                   customer.name(),
                   customer.itemsPurchased(),
                   regularQueue.size());
       }
   }


    public Q4Customer removeCustomer() { // O(N)
       Q4Customer customerRemoved = null;
       if (!expressQueue.isEmpty()) {
           customerRemoved = expressQueue.removeHead();
           System.out.printf("Customer %d: %s with %d items left the express
```

```
queue, %d customers left in this queue.\n",
                    customerRemoved.customerID(),
                    customerRemoved.name(),
                    customerRemoved.itemsPurchased(),
                    expressQueue.size());
        }
        else if (!regularQueue.isEmpty()) {
            customerRemoved = regularQueue.removeHead();
            System.out.printf("Customer %d: %s with %d items left the regular
queue, %d customers left in this queue.\n",
                    customerRemoved.customerID(),
                    customerRemoved.name(),
                    customerRemoved.itemsPurchased(),
                    regularQueue.size());
        }

        if (customerRemoved != null) {
            if (expressQueue.isEmpty()) {
                reassignCustomer(expressQueue, regularQueue);
            } else if (regularQueue.isEmpty()) {
                reassignCustomer(regularQueue, expressQueue);
            }
        }

        return customerRemoved;
    }

    private void reassignCustomer(SamuelLinkedList<Q4Customer> newQueue,
SamuelLinkedList<Q4Customer> oldQueue) { // O(1) or O(N)
        newQueue.add(oldQueue.removeHead());
    }




    public void printCustomersInfo() { // O(N)
        System.out.println("Express queue:");
        for (Q4Customer customer : expressQueue) {
            System.out.printf("%s: %s - %d items\n", customer.customerID(),
customer.name(), customer.itemsPurchased());
```

```
        }
        System.out.println("Regular queue:");
        for (Q4Customer customer : regularQueue) {
            System.out.printf("%s: %s - %d items\n", customer.customerID(),
customer.name(), customer.itemsPurchased());
        }
    }
}
```

All methods in this data structure have worst-case estimated time complexities of O(N), which is not good. The cause of this is the underlying data structure SamuelLinkedList.java which is badly optimized, and the effects of this cascade to all parts of the program using the data structure. This underlines the importance of studying time complexity for the data structures algorithms and being aware of the worst-case estimates.

The addCustomer method is O(N) because it both uses the O(N) add method of the underlying queues and the O(N) size method of the underlying queues. The removeCustomer method is O(N) because of the O(N) size method of the underlying queues. The reassignCustomer method is also generally O(N) because of its use of the O(N) add method of the underlying queues. The printCustomersInfo is also O(N) because of iterating over all items in the queues.

Q4Customer.java

```
package SamuelDatastructures;

public record Q4Customer(String name, int customerID, int itemsPurchased) {}
```

This data structure has O(1) time complexity since the only methods it has are the built-in getter methods which do not scale based on input or the size of the data.

3. Big O of Q6

Q6Inomhus.java

```
public class Q6Inomhus {

   Q6Warehouse[] warehouses = new Q6Warehouse[20];

   public Q6Inomhus() {

       for (int i = 0; i < 20; i++) {
           warehouses[i] = new Q6Warehouse();
       }


   }

   public int getEggId( // O(1)
           int warehouseNumber,
           int truckNumber,
           int boxNumber,
           int stackNumber,
           int trayNumber,
           int trayRow,
           int trayColumn
   ) {
       return warehouses[warehouseNumber]
               .getTrucks()[truckNumber]
               .getBoxes()[boxNumber]
               .getTrayStacks()[stackNumber]
               .getTrays()[trayNumber]
               .getEggs()[trayRow][trayColumn];
   }

}
```

The getEggId method has a time complexity of O(1) since all underlying data structures are based on arrays, which have O(1) access.

Q6Warehouse.java

```
public class Q6Warehouse {

   Q6Truck[] trucks = new Q6Truck[15];

   public Q6Warehouse() {
       for (int i = 0; i < 15; i++) {
           trucks[i] = new Q6Truck(100);
       }
   }

   public Q6Truck[] getTrucks() {
       return trucks;
   }
}
```

Array-based data structure which allows for O(1) access.

Q6Truck.java

```
package Q6;

public class Q6Truck {

   Q6Box[] boxes = new Q6Box[100];

   public Q6Truck(int boxes) {
       for (int i = 0; i < boxes; i++) {
           this.boxes[i] = new Q6Box();
       }
   }

   public Q6Box[] getBoxes() {
       return boxes;
   }
}
```

Array-based data structure which allows for O(1) access.

Q6Box.java

```
package Q6;

public class Q6Box {

   Q6TrayStack[] trayStacks = new Q6TrayStack[6];

   public Q6Box() {
       for (int i = 0; i < 6; i++) {
           trayStacks[i] = new Q6TrayStack(30);
       }
   }

   public Q6TrayStack[] getTrayStacks() {
       return trayStacks;
   }
}
```

Array-based data structure which allows for O(1) access.

Q6TrayStack.java

```
package Q6;

public class Q6TrayStack {

   Q6Tray[] Trays = new Q6Tray[30];

   public Q6TrayStack(int stacks) {
       for (int i = 0; i < stacks; i++) {
           Trays[i] = new Q6Tray();
       }
   }

   public Q6Tray[] getTrays() {
       return Trays;
   }
}
```

Array-based data structure which allows for O(1) access.

Q6Tray.java

```
package Q6;

public class Q6Tray {

   int[][] eggs = new int[10][10];

   public Q6Tray() {
       var idGenerator = new Q6IDGenerator();
       for (int i = 0; i < 10; i++) {
           for (int j = 0; j < 10; j++) {
               eggs[i][j] = idGenerator.getId();
           }
       }
   }

   public int[][] getEggs() {
       return eggs;
   }
}
```

Array-based matrix data structure which allows for O(1) access.