Question 1

Question 3 Big O

Part A

In question 3 part A, the methods empty(), full(), size(), top(), push() and pop() were implemented into THFixedStack.

empty()

```java
public boolean empty() {
        return top < 0;
}
```

This method performs in constant time O(1). It performs a comparison, which is returned as a boolean.

full()

```java
public boolean full() {
        return size() >= arraySize;
}
```

This method performs in constant time O(1). It performs a comparison, which is returned as a boolean.

size()

```java
public int size() {
        return top + 1;
}
```

This method performs on constant time O(1), returning a value in one operation.

top()

```java
public T top() throws EmptyStackException {
        if (empty()) {
                throw new EmptyStackException();
        }
        return itemArray[top];
}
```

This method first calls empty(), which executes in constant time, then uses the result in a condition, leaving that line after two operations. If the condition is true, one operation is performed, else another operation is performed. There are three operations performed at constant time O(1).

push()

```java
public boolean push(T item) {
        if (++top >= arraySize) {
                throw new StackOverflowError();
        }
        itemArray[top] = item;
        return true;
}
```

This method performs a comparison and a condition in the first line, which is two operations. If the condition is true, an exception is thrown with a total of 3 operations. If not, two operations are performed for a total of 4 operations. This method performs in constant time O(1).

pop()

```java
public T pop() throws EmptyStackException {
        if (empty()) {
                throw new EmptyStackException();
        }
        return itemArray[top--];
}
```

This method performs 2 operations. The fist is the condition check and the second is either throwing an exception or returning the popped item. It performs in constant time O(1).

All methods of the THFixedStack perform in constant O(1) time.

empty(), O(1)
full(), O(1)
size(), O(1)
top(), O(1)
push(), O(1)
pop(), O(1)

Part B

In question 3 part B, the methods deque(), push() and pop() were implemented into THStueue.

push()

```
public boolean push(T item) {
        return stack1.push(item);  //1
}
```

The push() method uses the THFixedStack.push() method, which performed in constant time so this method also does so. O(1)

pop()

```
public T pop() {
        return stack1.pop();  //1
}
```

It uses the THFixedStack.pop() method which also performed in constant time, so the result here is also O(1).

deque()

```
public T deque() {
        T item;  //1
        while (!stack1.empty()) {  //N+1
                stack2.push(stack1.pop());  //N*2
        }
        item = stack2.pop();
        while (!stack2.empty()) {  //N+1
                stack1.push(stack2.pop());  //N*2
        }
        return item;  //1
}
```

This method has two loops, traversing each item in the stack. It uses methods of the THFixedStack that performed in constant time and will not affect the execution that much. The loops on the other hand will have to perform two operations for each item in the stack and this is also done twice. If there are N items in the stack there will be 2*N*2
operations from the loops giving this method linear growth of O(N).

pop(), O(1)
push(), O(1)
deque(), O(N)

Question 4 Big O

THCustomer

In the THCustomer class, everything is constant O(1).

```java
public String name() {
        return name;
}

public int customerId() {
        return customerId;
}

public int groceryAmount() {
        return groceryAmount;
}

@Override
public String toString() {
        return "THCustomer: " + customerId + ", " +
                        "name: " + name + ", " +
                        "amount of groceries: " + groceryAmount + ".";
}
```

customerId(), O(1)
groceryAmount(), O(1)
toString(), O(1)

THLinkedList

The only methods used in the THLinkedList class are: size(), removeFirst(), addLast() and isEmpty(). Therefore those are the only public methods I will evaluate for that class. There are however some private and public methods that are used by those methods that will be mentioned.

size()

```java
public int size() {
        return size;
}
```

Constant, O(1)

isEmpty()

```java
public boolean isEmpty() {
        return size == 0;
}
```

Constant, O(1)

private outOfBounds()

```java
private boolean outOfBounds(int index) {
        return index < 0 || index >= size;
}
```

Constant, O(1)

private getNode()

```java
private Node<T> getNode(int index) {
        if (outOfBounds(index)) {
                throw new IndexOutOfBoundsException();
        }
        Node<T> node = head.next;
        for (int i = 0; i < index; i++) {
                node = node.next;
        }
        return node;
}
```

If the condition in the if statement holds true, it will be constant O(1). Otherwise it will execute the for loop, which makes it linear O(N).

remove()

```java
public T remove(int index) {
        Node<T> prevNode;
        if (index == 0) {
                prevNode = head;
        } else {
                prevNode = getNode(index - 1);
        }
        T item = prevNode.next.item;
        prevNode.next = prevNode.next.next;
        size--;
        return item;
}
```

If the index of the item to remove is 0, it will be constant O(1), otherwise it will call getNode() which is linear O(N).

removeFirst()

```java
public T removeFirst() {
        return remove(0);
}
```

This method calls remove() which will perform in constant time if the index 0 is passed. Therefore, this method performs in constant time O(1).

addLast()

```
public void addLast(T item) {
        Node<T> node = new Node<>(null, null);
        tail.item = item;
        tail.next = node;
        tail = node;
        size++;
}
```

Constant O(1).

toString()

```
@Override
public String toString() {
        StringBuilder sb = new StringBuilder();
        THLinkedListIterator it = new THLinkedListIterator();
        int index = 0;
        while (it.hasNext()) {
                sb.append(index++).append(":
").append(it.next().toString()).append("\n");
        }
        return sb.toString();
}
```

This method will iterate through the entire list of N items. It will grow linearly with the size of the list, O(N).

size(), O(1)
isEmpty(), O(1)
outOfBounds(), O(1)
getNode(), O(N)
remove(), O(N)
removeFirst(), O(1)
addLast(), O(1)
toString(), O(N)

THCustomerQueue

The methods that are used by the program are addCustomer(), removeCustomer(), migrateCustomer() and toString().

addCustomer()

```java
public void addCustomer(THCustomer customer) {
        String queueType;
        int queueLength;

        if (customer.groceryAmount() <= EXPRESS_LIMIT) {
                expressQueue.addLast(customer);
                queueType = "express";
                queueLength = expressQueue.size();
        } else {
                normalQueue.addLast(customer);
                queueType = "normal";
                queueLength = normalQueue.size();
        }
        System.out.printf("Customer entered in %s queue and it grew to a length of
%d.%n", queueType, queueLength);
}
```

This method is constant, O(1). There are no loops and the method calls are also constant.

removeCustomer()

```java
public THCustomer removeCustomer() {
        THCustomer customer;
        String queueType;

        if (!expressQueue.isEmpty()) {
                customer = expressQueue.removeFirst();
                queueType = "express";
        } else if (!normalQueue.isEmpty()) {
                customer = normalQueue.removeFirst();
                queueType = "normal";
        } else {
                System.out.println("No customers in queue.");
                return null;
        }
        System.out.printf("Customer %d %s with %d items left the %s queue.%n",
        customer.customerId(), customer.name(), customer.groceryAmount(),
queueType);
        System.out.printf("Express queue: %d customers, Normal queue: %d
customers.%n", expressQueue.size(), normalQueue.size());

        // Migrate customer between queues if one queue is empty
        if (expressQueue.isEmpty() && !normalQueue.isEmpty()) {
                migrateCustomer(normalQueue, expressQueue);
        } else if (normalQueue.isEmpty() && !expressQueue.isEmpty()) {
                migrateCustomer(expressQueue, normalQueue);
        }
        return customer;
}
```

The time complexity of this method is constant and so are all the function calls. O(1)

migrateCustomer()

```
        private void migrateCustomer(THLinkedList<THCustomer> from,
THLinkedList<THCustomer> to) {
        THCustomer customer = from.removeFirst();
        to.addLast(customer);
                System.out.printf("Customer %d %s with %d items migrated between
queues.%n",  customer.customerId(), customer.name(), customer.groceryAmount());
}
```

Constant O(1).

toString()

```
public String toString() {
        return "\nTHCustomerQueue contains "
        + expressQueue.size() + " express customers and "
        + normalQueue.size() + " normal customers.\n"
        + "Express queue: \n" + expressQueue
        + "Normal queue: \n" + normalQueue;
}
```

This method will call toString() on both lists. If the expressQueue have N customers and the normalQueue have M customers, the time complexity of this method will be O(NM).

addCustomer(), O(1)
removeCustomer(), O(1)
migrateCustomer(), O(1)
toString(), O(NM)

Question 6 Big O

As each part of the hierarchy is stored in arrays and indexing is used to receive the correct item from each array, the time complexity to get an egg is constant, O(1). However, to initialize it all, each item must be created and put into the array. This results in a massive time penalty as there are 540 000 000 eggs, 5 400 000 trays, 180 000 stacks, 30 000 boxes, 300 trucks and 20 warehouses.
If there are E eggs, T trays, S stacks, B boxes, R trucks and W warehouses, the time complexity to initialize everything is O(ETSBRW).