Duy Huynh
TCSS 342 – Spring ‘15
Assignment 1 – Burger Baron
Worst Case Run-Time Analysis of changePatties

1 Introduction

This is a worst case run-time analysis of the changePatties() method in my Burger Baron assignment. For convenience, the relevant code is provided below with numbered lines:

```
1 public void changePatties(final String pattyType) {
2
3     // Don't change patties if it's already of that type!
4     if (!this.pattyType.equals(pattyType)) {
5           final MyStack<Ingredient> tempStack = new MyStack<>();
6
7           // Change the patty that is underneath the cheeses
8
9           // 1. pop off and store the cheeses:
10          for (int i = 0;i < cheeseCount;i++) {
11                tempStack.push(botStack.pop());
12          }
13
14          // 2. Dispose the old patty
15          botStack.pop();
16
17          // 3. Push new patty:
19          botStack.push(Ingredient.getIngredient(pattyType));
20
21          // 4. Put back the cheeses
22          for (int i = 0;i < cheeseCount;i++) {
23                botStack.push(tempStack.pop());
24          }
25
26          // Work on the patties of the other stack
27
28          // 5. Pop off old patties, if any, from top stack
29          for (int i = 0;i < pattyCount - 1;i++) {
30                topStack.pop();
31          }
32
33          // 6. Push new patties, if any, onto top stack
34          for (int i = 0;i < pattyCount - 1;i++) {
35                topStack.push(Ingredient.getIngredient(pattyType));
36          }
37
38          // 7. Change patty type
39          this.pattyType = pattyType;
40    }
41 }

42 /**Create a Stack. */
43 public MyStack() {
44    top = null;
45    myPointer = 0;
46 }
```

Duy Huynh
TCSS 342 – Spring ‘15
Assignment 1 – Burger Baron
Worst Case Run-Time Analysis of changePatties

```
public void push(final T element) {

   if (top == null) {
         top = new Node(element, null);
   } else {
         top = new Node(element, top);
   }
   myPointer++;

}
```

```
public T pop() {

   T temp;
   if (top == null) {
         throw new NullPointerException("Nothing to pop off stack!");
   } else {
         temp = (T) top.getElement();
         top = top.myNextNode;
         myPointer--;
   }
    return temp;

}
```

```
public static Ingredient getIngredient(String ingredientName) {
   for (Ingredient i : values()) {
         if (i.getName().equalsIgnoreCase(ingredientName)) {
               return i;
         }
   }
   throw new IllegalArgumentException("Ingredient \"" + ingredientName
         + "\" not found!");
}
```

```
private T getElement() {
  return myElement_
}
```

```
private Node(final T theElement, final Node theNextNode) {
  this.myElement = theElement_
  this.myNextNode = theNextNode_
}
```

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

Duy Huynh
TCSS 342 – Spring ‘15
Assignment 1 – Burger Baron
Worst Case Run-Time Analysis of changePatties

2 Analysis

2.1 Line-by-line breakdown:

Let CHEESE_MAX = 3, PATTY_MAX = 2, INGREDIENTS = 17 (total values contained in Ingredients enum).
Let declarations, assignments, negations, checks, integer and Boolean arithmetic, and similar operations cost 1.

changePatties:

Line 4: check + negation + C_{equals} = 2 + C_{equals} = C_4

Line 5: declaration + assignment + $C_{MyStack}$ = 2 + $C_{MyStack}$ = 2 + 13 = 15

Line 10 & 11: declaration + $\sum_{n=1}^{CHEESE_{MAX}}(\text{check} + \text{increment} + C_{push} + C_{pop}) = 1 + CHEESE\_MAX * (2 + C_{push} + C_{pop})$

Line 15: C_{pop}

Line 19: C_{push} + $C_{getIngredient}$

Line 22 & 23: declaration + $\sum_{n=1}^{CHEESE_{MAX}}(\text{check} + \text{increment} + C_{push} + C_{pop}) = 1 + CHEESE\_MAX * (2 + C_{push} + C_{pop})$

Line 29 & 30: declaration + $\sum_{n=1}^{PATTY\_MAX-1}(\text{check} + \text{subtraction} + \text{increment} + C_{pop}) = 1 + (PATTY_{MAX} - 1) * (3 + C_{pop})$

Line 34 & 35: declaration + $\sum_{n=1}^{PATTY\_MAX-1}(C_{push} + \text{check} + \text{subtraction} + \text{increment} + C_{getIngredient}) = 1 + (PATTY\_MAX - 1) * (3 + C_{push} + C_{getIngredient})$

Line 39: assignment = 1

Total: $C_4 + 15 + 1 + 2 * CHEESE\_MAX * (2 + C_{push} + C_{pop}) + 1 + (PATTY\_MAX - 1) * (3 + C_{pop}) + 1 + PATTY\_MAX * (3 + C_{push} + C_{getIngredient})$

MyStack():

Line 44: assignment = 1

Line 45: assignment = 1

Total: 1 + 1 = 2 = $C_{MyStack}$

push():

Line 49: check + check = 2

Line 50: assignment + C_{Node} = 1 + C_{Node} (Node has 2 assignments, so it costs 2). = 3

Line 51: assignment + C_{Node} = 1 + C_{Node} = 3

Line 54: increment = 1

Total: $7 + 2 * C_{Node} = 7 + 2 * 3 = 13 = C_{push}$

pop():

Line 59: declaration = 1

Line 60: check + check = 2

Line 63: assignment + C_{cast} + $C_{getElement}$ = 1 + C_{cast} + $C_{getElement}$ (getElement() has one return statement = 1)

Line 67: return statement = 1

Total: 4 + C_{cast} + $C_{getElement}$ = 5 + C_{cast} = C_{pop}

getIngredient():

Line 71: declaration + C_{values} = 1 + C_{values}

Line 72 & 73: $\sum_{n=1}^{INGREDIENTS}(\text{check} + C_{getName} + C_{equalsIgnoreCase}) = INGREDIENTS * (1 + C_{getName} + C_{equalsIgnoreCase})$

Total: 1 + C_{values} + $C_{getName}$ + $C_{equalsIgnoreCase}$ = $C_{getIngredient}$

Duy Huynh
TCSS 342 – Spring '15
Assignment 1 – Burger Baron
Worst Case Run-Time Analysis of changePatties

2.2 For-loop breakdowns:

In changePatties(), there are 4 for-loops.

2.2.1: The for-loop on line 10 is responsible for taking off the cheeses that are present in the current Burger. This loop has a worst case run-time scenario when cheeseCount is at the maximum, which is 3 pieces of cheese:

$$\sum_{n=1}^{CHEESE_{MAX}}\left(\text{check } + \text{ increment} + \mathrm{C}_{push} + \mathrm{C}_{pop}\right) = CHEESE\_MAX * (2 + C_{push} + C_{pop})$$

2.2.2: The for-loop on line 22 is responsible for putting back on the cheeses after the new patty has been put on. This loop, similar to the previous, has worst case run-time scenario when cheeseCount is at the maximum, which is 3:

$$\sum_{n=1}^{CHEESE_{MAX}}\left(\text{check } + \text{ increment} + \mathrm{C}_{push} + \mathrm{C}_{pop}\right) = CHEESE\_MAX * (2 + C_{push} + C_{pop})$$

2.2.3: The for-loop on line 29 is responsible for removing any old existing patties that sit on top of the cheeses. This loop has a worst case run-time scenario when the patty count that sit on top of the cheeses is maxed out at 2 patties:

$$\sum_{n=1}^{PATTY\_MAX-1}\left(\text{check } + \text{ subtraction } + \text{ increment} + \mathrm{C}_{pop}\right) = (PATTY\_MAX - 1) * (3 + \mathrm{C}_{pop})$$

2.2.4: The for-loop on line 34 is responsible for pushing new patties onto the stack to match the new patty type. This loop has a worst case run-time scenario when there is a change of 2 patties max that sit on top of the cheeses. It is also interesting to note that the getIngredient() method called in this loop is also a for-loop that searches through all the values of my Ingredient enumeration which has 17 values:

$$\sum_{n=1}^{INGREDIENTS}\left(\text{check } + \ C_{getName} + \mathrm{C}_{equalsIgnoreCase}\right) = INGREDIENTS * (1 + C_{getName} + \mathrm{C}_{equalsIgnoreCase})$$

2.3 Total method cost:

The total summation of the line-by-line cost *c(n)* for changePatties() is:

$$c(n) = C_4 + 15 + 1 + 2 * CHEESE\_MAX * (2 + C_{push} + C_{pop}) + \ 1 + (PATTY\_MAX - 1) * (3 + \mathrm{C}_{pop}) + 1 + PATTY\_MAX * (3 + \mathrm{C}_{push} + \mathrm{C}_{getIngredient})$$

$$c(n) = C_4 + 18 + 2 * CHEESE\_MAX * (2 + C_{push} + C_{pop}) + \ (PATTY\_MAX - 1) * (6 + \mathrm{C}_{pop} + \mathrm{C}_{push} + \mathrm{C}_{getIngredient})$$

$$c(n) = C_4 + 18 + 2 * 3 * (2 + C_{push} + C_{pop}) + \ 2 * (6 + \mathrm{C}_{pop} + \mathrm{C}_{push} + \mathrm{C}_{getIngredient})$$

$$c(n) = C_4 + 18 + 6 * (2 + C_{push} + C_{pop}) + \ 2 * (6 + \mathrm{C}_{pop} + \mathrm{C}_{push} + \mathrm{C}_{getIngredient})$$

$$c(n) = C_4 + 18 + 12 + 6 * C_{push} + 6 * C_{pop} + 12 + 2 * \mathrm{C}_{pop} + 2 * \mathrm{C}_{push} + 2 * \mathrm{C}_{getIngredient}$$

$$c(n) = C_4 + 42 + 8 * C_{push} + 8 * C_{pop} + 2 * \mathrm{C}_{getIngredient}$$

$$c(n) = C_4 + 8 * (C_{push} + C_{pop}) + 2 * \mathrm{C}_{getIngredient} + 42 = C_{totalSum}$$

Conclusion: $c(n) \in O(n)$