**Chapter 6) – 4 Points**

**1,2 – 0.5 Points)**

**1) Suppose an initially empty stack S has performed a total of 25 push operations, 12 top operations, and 10 pop operations, 3 of which return null to indicate an empty stack. What is the current size of S?**

25 Push Operations – 25 Adittions

12 Top Operations – 12 Checks on top value – No effect on size

10 pop operations – 10 attempts at removing element – 3 Unnsuccessful

25 + - 10 + 3 = 18

Current size is 18.

**2) Had the stack of Exercise 6.4.1 been an instance of the ArrayStack class, from Code Frament 6.1.2, what would be the final value of the instance variable t?**

ArrayStack in example is not circular. Instance variable t = 17.

**3, 9, 12 – 0.5 Points)**

**3) What values are returning during the following series of stack operations, if executed upon an initially empty stack?**

push(5) - 5

push(3) – 5,3

pop() – 5, RETURN 3

push(2) – 5, 2

push(8) – 5, 2, 8

pop() – 5, 2 RETURN 8

pop() – 5 RETURN 2

push(9) – 5, 9

push(1) – 5, 9, 1

pop() – 5, 9 RETURN 1

push(7) – 5, 9, 7

push(6) – 5, 9 7, 6

pop() – 5, 9, 7 RETURN 6

pop() – 5, 9 RETURN 7

push(4) – 5, 9, 4

pop() – 5, 9 RETURN 4

pop().- 5, RETURN 9

*RETURNS = 3, 8, 2, 1, 6, 7, 4, 9*

**9) What values are returned during the following sequence of queue operations, if executed on an initially empty queue?**

enqueue(5) - 5

enqueue(3) – 3, 5

dequeue() – 3 RETURN 5

enqueue(2) – 2, 3

enqueue(8) – 8, 2, 3

dequeue() – 8, 2 RETURN 3

dequeue() – 8 RETURN 2

enqueue(9) – 9,8

enqueue(1) – 1, 9, 8

dequeue() – 1, 9 RETURN 8

enqueue(7) – 7, 1, 9

enqueue(6) – 6, 7, 1, 9

dequeue() – 6, 7, 1 RETURN 9

dequeue() – 6, 7, RETURN 1

enqueue(4) – 4, 6, 7

dequeue() – 4, 6 RETURN 7

dequeue() – 4 RETURN 6

*RETURNS = 5, 3, 2, 8, 9, 1, 7, 6*

**12) What values are returned during the following sequence of deque ADT operations, on an initially empty deque?**

addFirst(3) - 3

addLast(8) – 3, 8

addLast(9) – 3, 8, 9

addFirst(1) – 1, 3, 8, 9

last() – 1, 3, 8, 9 RETURN 9

isEmpty() – 1, 3, 8, 9 RETURN FALSE

addFirst(2) – 2, 1, 3, 8, 9

removeLast() – 2, 1, 3, 8 RETURN 9

addLast(7) – 2, 1, 3, 8, 7

first() – 2, 1, 3, 8, 7 RETURN 2

last() – 2, 1, 3, 8, 7 RETURN 7

addLast(4) – 2, 1, 3, 8, 7, 4

size() – 2, 1, 3, 8, 7, 4 RETURN 6

removeFirst() – 1, 3, 8, 7, 4 RETURN 2

removeFirst() – 3, 8, 7, 4 RETURN 1

*RETURNS – 9, false, 9, 2, 7, 6, 2, 1*

**6 – 0.5 Points) Give a precise and complete definition of the concept of matching for grouping symbols in an arithmetic expression. Your definition may be recursive.**

In arithmetic brackets, parenthesis, and braces are commonly used to group various parts of a problem that need to be evaluated together before moving outward.

In terms of matching, each opening bracket, brace, parenthesis needs to have a corresponding closing bracket. We cannot introduce a closing bracket without first introducing an opening one. Also, the most recent type of opening bracket, brace, parenthesis most also match the type of closing one. We cannot have an opening brace and a closing bracket.

Thus, each ( must have a )

Each [ must have a ]

And each { must have a }

When running into an opening bracket, brace, or parenthesis, we enter a smaller equation within the equation where this portion must be evaluated up to the matching closing brace before returning to the top most equation.

Recursively speaking, our base case would be an equation without any of the previously stated matching symbols. As we process the part within the parentheses evaluating the smaller parts of the equation and return them to the top equation without said grouping symbols, we reach our base case and are able to finally evaluate the whole equation.

**13, 14 – 0.5 Points)**

**13) Suppose you have a deque D containing the numbers (1,2,3,4,5,6,7,8), in this order. Suppose further that you have an initially empty queue Q. Give a code fragment that uses only D and Q (and no other variables) and results in D storing the elements in the order (1, 2, 3, 5, 4, 6, 7, 8)**

For (int I = 0; I < 3; i++)

Q.enqueue(D.removeLast());// Q = 6, 7, 8, D = 1,2,3,4,5

D.addFirst(D.removeLast())); // Q = 6, 7, 8 D = 5, 1, 2, 3, 4

Q.enqueue(D.removeLast()); // Q = 4, 6, 7, 8 D = 5, 1, 2, 3

Q.enqueue(D.removeFirst()); // Q = 5, 4, 6, 7, 8 D = 1, 2, 3

While(!D.isEmpty())

Q.enqueue(D.removeLast(); // Q = 1, 2, 3, 5, 4, 6, 7, 8 D = Empty

While(!Q.isEmpty())

D.addFirst(Q.dequeue); // D = 1,2,3,4,5,6,7,8 Q = Empty

**14) Repeat Exercise 6.4.13 using the deque D and an initially empty stack S.**

For(int I = 0; I < 5; i++)

S.push(D.removeLast()); // S = 4, 5, 6, 7, 8 D = 1, 2, 3

D.addFirst(S.pop()); // S = 5, 6, 7, 8 D = 4, 1, 2, 3

D.addLast(S.pop()); // S = 6, 7, 8 D = 4, 1, 2, 3, 5

S.push(D.removeFirst()); // S = 4, 6, 7, 8 D = 1, 2, 3, 5

While(!S.isEmpty());

D.addLast(S.pop()); // S = empty D = 1, 2, 3, 5, 4, 6, 7, 8

**24 – 1 Point) Suppose you have a stack S containing n elements and a queue Q that is initially empty. Describe how you can use Q to scan S to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements back to S in their original order. You may only use S, Q, and a constant number of other primitive variables.**

Intialize two variables

Boolean elementIsPresent

Int(or whatever type element we’re looking for) checkVal

For loop that will run exactly 2 times

While S is not empty

Pop element into variable checkVal

Check if checkVal is what we are looking for

If true, set boolean elementIsPresent to true

Enqueue checkVal into Q

While Q is not empty

Push dequeued elements back onto stack

First iteration will put elements back into stack reverse. Second run will return it in the correct order. Boolean elementIsPresent can be returned as true or false depending on if it’s present

**31 - 1 Point) Describe how to implement the deque ADT using two stacks as the only instance variables. What are the running times of the methods?**

The bottom of S1 / Top of S2 will be defined as the front. Conversely, the Top of S1 and Bottom of S2 will be defined as the back.

Functions:

addFirst – If S1 is empty, push element onto S1

- If S1 is not empty, pop all elements and push them onto S2 then push element onto S2

First – If S1 is empty, check what element is at the top of S2 if it has elements

- If S1 is not empty, pop all elements and push onto S2. Then check top S2 element

removeFirst – If S1 has size equal to 1, pop the S1 element

- If S1 is empty, pop the element from S2

- If S1 size is greater than 1, pop all values and push onto S2 until size is equal to 1

- Pop and return last element. Do not input into S2

addLast – If S2 is empty, push element onto S2

- If S2 is not empty, pop all elements and push them onto S1 then push element onto S2

Last – If S2 is empty, check what element is at the top of S1 if it has elements

- If S2 is not empty, pop all elements and push onto S1. Then check top S1 element

removeLast – If S2 has size equal to 1, pop the S2 element

- If S2 is empty, pop the element from S1

- If S2 size is greater than 1, pop all elements and push onto S1 until size equals 1

- Pop and return the last element. Do not input into S1

Size – Add size of S1 and S2

isEmpty – Check if size is equal to 0

Runtime:

addFirst. first, removeFirst, addLast, last, removeLast are all at worst O(n).

Size and isEmpty should operate at O(1) time.