Alexandria University
Faculty of Engineering
Computer and Systems Engineering Department

Level 1 - Term 2
Data Structures 1
Sheet 3

Student Name: George Samy Wahba Beshay
Student Academic ID: 20010435

Question 1:
Suppose an initially empty stack S has performed a total of 25 push operations, 12 top operations and 10 pop operations. Three of the pop operations generated ”Empty Stack Exception”, which were caught and ignored.
1) What is the size of S after performing the operations described above?
2) If this stack is implemented as an array, what is the value of the ”top” data member of the Stack class?

1) **Based on that 3 pop operations generated an error >> That means that - *for example* - 7 push operations were made first then 10 pop operations were executed >> then (25 - 7) = 18 push operations executed after. Also the top operations were called in between them, Note that the top operation just shows the top most element without removing it from the stack.**
   Size of S = (25 - 7) = 18
2) **The top most element in the Array - Based Stack is considered to be the last element in the array.**
   So taking in consideration that the array indexing starts from 0 then
   Value of the “Top” Data Member = Size - 1 = 18 - 1 = 17

Question 2:

Describe the output of the following series of stack operations:

push(5), push(3), pop(), push(2), push(8), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().

NOTE: Element coloured in *blue* is the top most element in the stack that could be popped out.

Operation	Output	Stack Content
-	-	(-)
push(5)	-	(5)
push(3)	-	(5,3)
pop()	3	(5)
push(2)	-	(5,2)
push(8)	-	(5,2,8)
pop()	8	(5,2)
push(9)	-	(5,2,9)
push(1)	-	(5,2,9,1)
pop()	1	(5,2,9)
push(7)	-	(5,2,9,7)
push(6)	-	(5,2,9,7,6)
pop()	6	(5,2,9,7)
pop()	7	(5,2,9)
push(4)	-	(5,2,9,4)
pop()	4	(5,2,9)
pop()	9	(5,2)

Question 3:

Write an algorithm that returns the number of elements in a stack leaving it unchanged. (Assume that the stack Abstract Data Type provides only pop and push operations).

```
// We will keep tracking the pop operation until it
// generates an “Empty Stack Exception”.
// We will need the help of another stack - OR queue -
// to store in them the popped elements,
// so we can return them at the end to the same stack.
Algorithm getSize(){
      Input: Stack S
      Output: Int  Size
      int Size <- 0;
      TempStack <- new Stack();
      while(S is not Empty){
            Object temp <- S.pop();
            TempStack.push(temp);
            Size++;
      }
      while(TempStack is not Empty){
            Object temp <- TempStack.pop();
            S.push(temp);
      }
      Return Size;
}
```

Question 4:

Write an algorithm that uses a stack to determine if an HTML document is well-formed. (A well-formed HTML document should have all tags properly nested and all opened tags should have the corresponding closing tags).

```
// We will scan the text from the file line by line
// and whenever we detect an HTML Tag we will push it
// to the stack, and at the end, if the stack is empty
// then the HTML is well-formed, if not, or if an error
// of "Empty Stack Exception" has been generated then
// the HTML is not well-formed.
Algorithm HTML_Check(file HTML.txt){
      Input: HTML - Text Document File that will be scanned.
      Output: Boolean Value (True or False).
      stack TempStack <- new stack();
      Boolean GoodFlag <- True;
      OP_HTML_Tags <- [<body>,<center>,<h1>,<p>,<ol>,<li>]
      CL_HTML_Tag <-
      [</body>,</center>,</h1>,</p>,</ol>,</li>]
      Loop(InputScanner.hasNextLine() AND no exception
      Was detected){
            Object tempObj <- InputScanner.Next()
            if(tempObj in OP_HTML_Tags)
                  TempStack.push(tempObj);
            if(tempObj in CL_HTML_Tags){
                  if(TempStack.top == tempObj)
                        TempStack.pop();
                  Else
                        GoodFlag <- False;
                        Break Loop;
            }
      }
      Return GoodFlag;
}
```

Question 5:

A palindrome is a word or a phrase that is the same when spelled from the front or the back. For example reviver and able was I ere I saw elba are both palindromes. Write an algorithm that uses a stack to determine if a word or a phrase is a palindrome.

```
// We will Create a Stack and push in it all the string
// - that to be checked - characters, and then pop out
// all the characters from it and store it in another String
// and then compare the Main String to the temp String.
Algorithm CheckPalindrome(){
      Input: String S
      Output: Boolean
      Boolean GoodFlag <- False;
      TempStack <- new Stack();
      TempString <- “” ;          // Empty String.
      Loop(int i = 0 ; i < S.length ; i++){
      TempStack.push(S[i]);
      }
      Loop(int i = 0 ; i < S.length ; i++){
      TempString[i] <- TempStack.pop();
      }
      If (TempString == S)
            GoodFlag <- True;
      Return GoodFlag;
}
```

Question 6:

Write an algorithm that doing the following on the stack leaving it unchanged:

1. Return an identical copy of the Stack.
2. Return a reversed copy of the Stack.
3. Return a sorted copy of the Stack in descending order.

(Assume that the stack Abstract Data Type provides only pop, push, peak, and isEmpty operations)

```
1) && 2)
// We will Use 2 TempStacks, 1st will be pushed to it
// the main stack elements, so it will be in a reverse
// order (2nd Required), To Overcome this problem we will
// pop out the elements from it to another stack
// so the order will be correct at the end.
Algorithm Copy_Stack() && Reverse_Stack(){
      Input: Stack S to be copied - OR Reversed -.
      Output: Copied - OR Reversed - Stack.
      Stack Reversed <- new Stack;
      Stack Copied <- new Stack;
      Loop(While S !isEmpty){
      Reversed.push(S.pop());
      }
Break Point Here → For 2nd Required, we will
return Reversed;
      Loop(While Reversed !isEmpty){
      Copied.push(Reversed.pop());
      }
Break Point Here → For 1st Required, we will
return Copied;
}
```

```
3)
// We will use a temp stack - Empty at initialization -
// and check the top element in the main stack
// if it is smaller than the top element in the temp stack,
// then store the top element of the main in a tempVar,
// and push the top element from the temp stack to the main
// one. And so on after. return the tempVar value and check
// it again.
//                    Main Stack      Temp Stack
//                    (2,3,0,1,5,4,9,8) ()              →
//                    (2,3,0,1,5,4,9) (8)               →
//                    (2,3,0,1,5,4,8) (9)               →
//                    (2,3,0,1,5,4)   (9,8)             →
//                    (2,3,0,1,5)     (9,8,4)           →
//                    (2,3,0,1,4)     (9,8,5)           →
//                    (2,3,0,1)       (9,8,5,4)         →
//                    (2,3,0)         (9,8,5,4,1)       →
//                    (2,3)           (9,8,5,4,1,0)     →
//                    (2,0,1)         (9,8,5,4,3)       →
//                    (2,0)           (9,8,5,4,3,1)     →
//                    (2)             (9,8,5,4,3,1,0)   →
//                    (0,1)           (9,8,5,4,3,2)     →
//                    (0)             (9,8,5,4,3,2,1)   →
//                    ()              (9,8,5,4,3,2,1,0) →
Algorithm Sort_Stack(){
      Input: Stack “S” to be Sorted.
      Output: Stack “Sorted”.
      Sorted <- new Stack();
      CopiedTemp <- Copy_Stack(S); // To Return the main
      // stack to its initial values as it will be empty at
      // end of our sorting operation.
      Sorted.push(S.pop())
      Loop(while S !isEmpty()){
            int tempVar <- S.pop();
            Loop(while Sorted.peak() < tempVar && Sorted
            !isEmpty()){
                  S.push(Sorted.pop());
            }
            Sorted.push(tempVar);
      }
      S <- CopiedStack;
      return Sorted;
}
```

End of Sheet 3