1- Let's trace the code and keep track of the contents of the stack after each invocation:

1. Initial stack: []

2. `stack.push(new Character('A'))`: ['A']

3. `stack.push(new Character('B'))`: ['A', 'B']

4. `stack.push(new Character('C'))`: ['A', 'B', 'C']

5. `stack.pop()`: ['A', 'B']

6. `stack.pop()`: ['A']

7. `stack.push(new Character('D'))`: ['A', 'D']

8. `stack.push(new Character('E'))`: ['A', 'D', 'E']

9. `stack.push(new Character('F'))`: ['A', 'D', 'E', 'F']

10. `stack.pop()`: ['A', 'D', 'E']

11. `stack.push(new Character('G'))`: ['A', 'D', 'E', 'G']

12. `stack.pop()`: ['A', 'D', 'E']

13. `stack.pop()`: ['A', 'D']

14. `stack.pop()`: ['A']

After the last `stack.pop()` call, the stack becomes empty.

2-

:

لتحديد الحجم الحالي لـ ArrayStack S وقيمة المتغير الفردي t، يمكننا تحليل عدد عمليات الإضافة والاستعلام والحذف وتأثيرها على الستاك.

المعطيات:

* إجمالي عمليات الإضافة: 25
* إجمالي عمليات الاستعلام: 12
* إجمالي عمليات الحذف: 10
* عودة قيمة فارغة من عمليات الحذف تشير إلى ستاك فارغ: 3

دعنا نحسب الحجم الحالي لـ S وقيمة المتغير الفردي t:

1. كل عملية إضافة تزيد من حجم الستاك بمقدار واحد.  
   إجمالي عمليات الإضافة: 25  
   حجم S بعد عمليات الإضافة: 25
2. كل عملية استعلام لا تؤثر على حجم الستاك.  
   إجمالي عمليات الاستعلام: 12
3. كل عملية حذف تقلل من حجم الستاك بمقدار واحد، ما لم تعود قيمة فارغة.  
   إجمالي عمليات الحذف: 10  
   عودة قيمة فارغة تشير إلى ستاك فارغ: 3  
   عمليات الحذف الفعالة: 10 - 3 = 7  
   حجم S بعد عمليات الحذف: 25 - 7 = 18

بالتالي، الحجم الحالي لـ S هو 18.

المتغير الفردي t يمثل مؤشر العنصر العلوي في الستاك. نظرًا لأن الستاك يستخدم فهرسة بدءًا من الصفر، سيكون مؤشر العنصر العلوي بمقدار أقل بواحد من حجم الستاك.

قيمة t = حجم S - 1 = 18 - 1 = 17

لذا، قيمة المتغير الفردي t هي 17

3-

Let's evaluate the postfix expressions and determine whether they are true or false:

a. 8 2 + 3 \* 16 4 / - =

Step 1: 8 + 2 = 10

Step 2: 10 \* 3 = 30

Step 3: 16 / 4 = 4

Step 4: 30 - 4 = 26

Final result: 26

Answer: True

b. 12 2 5 5 1 / / \* 8 7 + - =

Step 1: 5 / 1 = 5

Step 2: 5 / 5 = 1

Step 3: 2 \* 1 = 2

Step 4: 12 / 2 = 6

Step 5: 6 + 8 = 14

Step 6: 14 - 7 = 7

Final result: 7

Answer: False

c. 70 14 4 5 15 3 / \* - / 6 + =

Step 1: 15 / 3 = 5

Step 2: 5 \* 5 = 25

Step 3: 4 - 25 = -21

Step 4: 14 / -21 = 0 (integer division)

Step 5: 0 + 6 = 6

Final result: 6

Answer: True

d. 3 5 6 \* + 13 - 18 2 / + =

Step 1: 5 \* 6 = 30

Step 2: 3 + 30 = 33

Step 3: 33 - 13 = 20

Step 4: 18 / 2 = 9

Step 5: 20 + 9 = 29

Final result: 29

Answer: True

Therefore:

a. True

b. False

c. True

d. True

4-

import java.util.Stack;

public class تقييمالمعكوسةالبولندية {

public static int قيمة(String المعكوسة) {

Stack<Integer> ستاك = new Stack<>();

for (char حرف : المعكوسة.toCharArray()) {

if (Character.isLetterOrDigit(حرف)) {

ستاك.push(حرف - 'A'); // نفترض أن المشغلات هي حروف كبيرة مفردة

} else {

int المشغل2 = سعفوًا، واجهت بعض المشكلات في ترجمة إجابتي السابقة إلى العربية. ها هي الترجمة الصحيحة:

بالتأكيد! هنا ترجمة للإجابة:

لتحويل التعبيرات المعكوسة البولندية إلى صيغة معكوسة (Postfix) باستخدام عمليات الستاك، يمكننا استخدام القواعد التالية:

1. أنشئ ستاك فارغ وسلسلة ناتجة فارغة.

2. قم بفحص التعبير المعكوس من اليسار إلى اليمين.

3. إذا واجهت مشغل (متغير أو رقم)، أضفه إلى السلسلة الناتجة.

4. إذا واجهت فتح قوس، ضعه في الستاك.

5. إذا واجهت عملية، قم بإخراج المشغلات من الستاك وإضافتها إلى السلسلة الناتجة حتى يتحقق أحد الحالات التالية:

- الستاك فارغ.

- واجهت فتح قوس في الستاك.

- أولوية المشغل الحالي أقل من أولوية المشغل في أعلى الستاك.

ثم ضع المشغل الحالي في الستاك.

6. إذا واجهت إغلاق قوس، قم بإخراج المشغلات من الستاك وإضافتها إلى السلسلة الناتجة حتى تصل إلى فتح قوس. تجاهل القوسين الفتح والإغلاق.

7. بعد الانتهاء من فحص التعبير المعكوس، قم بإخراج أي مشغلات متبقية في الستاك وإضافتها إلى السلسلة الناتجة.

8. السلسلة الناتجة هي الصيغة المعكوسة للتعبير.

الآن دعنا نقوم بتحويل التعبيرات المعكوسة إلى صيغة معكوسة باستخدام عمليات الستاك:

أ. (A + B) \* (C + D) - E

الصيغة المعكوسة: AB+CD+\*E-

ب. A - (B + C) \* D + E / F

الصيغة المعكوسة: ABC+D\*-EF/+

ج. ((A + B) / (C - D) + E) \* F - G

الصيغة المعكوسة: AB+CD-/E+F\*G-

د. A + B \* (C + D) - E / F \* G + H

الصيغة المعكوسة: ABCD+\*+EF/G\*-H+

الآن دعنا نقوم بتحويل أول تعبيرين معكوسيين (أ وب) إلى رموز جافا باستخدام عمليات الستاك:

أ. رموز جافا للصيغة المعكوسة "AB+CD+\*E-":

```java

import java.util.Stack;

public class تقييمالصيغةالمعكوسة {

public static int تقييم(String المعكوسة) {

Stack<Integer> ستاك = new Stack<>();

for (char حرف : المعكوسة.toCharArray()) {

if (Character

5-

template<class T>

void linkedStack<T>::printListReverse() const {

std::stack<T> stack;

Node<T>\* currentNode = top;

// Push elements onto the stack

while (currentNode != nullptr) {

stack.push(currentNode->data);

currentNode = currentNode->next;

}

// Pop elements from the stack and print

while (!stack.empty()) {

std::cout << stack.top() << " ";

stack.pop();

}

std::cout << std::endl;

}

6-

public static <E> void reverse(ArrayStack<E> stack) {

ArrayStack<E> tempStack = new ArrayStack<>();

while (!stack.isEmpty()) {

E element = stack.pop();

tempStack.push(element);

}

while (!tempStack.isEmpty()) {

E element = tempStack.pop();

stack.push(element);

}

}

* - push(E element): Pushes an element onto the stack.
* top(): Returns the element at the top of the stack without removing it.
* pop(): Removes and returns the element at the top of the stack.
* isEmpty(): Returns true if the stack is empty, false otherwise.

7-

public static <E> E popBottom(LinkedStack<E> stack) {

LinkedStack<E> tempStack = new LinkedStack<>();

// Transfer elements from the original stack to the temporary stack

while (!stack.isEmpty()) {

E element = stack.pop();

tempStack.push(element);

}

// Remove and return the bottom element from the temporary stack

E bottomElement = tempStack.pop();

// Transfer elements back from the temporary stack to the original stack

while (!tempStack.isEmpty()) {

E element = tempStack.pop();

stack.push(element);

}

return bottomElement;

}

* - push(E element): Pushes an element onto the stack.
* top(): Returns the element at the top of the stack without removing it.
* pop(): Removes and returns the element at the top of the stack.
* isEmpty(): Returns true if the stack is empty, false otherwise.

8-

public class ArrayStack<E> {

private E[] elements;

private int top;

// Constructor and other methods...

public E topSecond() {

if (top < 1) {

throw new NoSuchElementException("Stack does not have a second element.");

}

return elements[top - 1];

}

}

9-

public class ArrayStack<E> {

private E[] elements;

private int top;

// Constructor and other methods...

public E popSecond() {

if (top < 1) {

throw new NoSuchElementException("Stack does not have a second element.");

}

E secondElement = elements[top - 1];

top--;

return secondElement;

}

}

1. We check if the top index is less than 1, indicating that the stack does not have a second element. In such a case, we throw a NoSuchElementException with an appropriate error message.
2. If the stack has at least two elements, we access the element at the index top - 1 and store it in the secondElement variable.
3. We decrement the top index to remove the second element from the stack.
4. Finally, we return the secondElement.

Note that this implementation assumes that the ArrayStack class has a member variable elements of type E[] that stores the elements, and an integer top that represents the index of the top element. Adjust the implementation as per your actual class structure.

Remember to import java.util.NoSuchElementException if it's not already imported in your code.

9-

public class ArrayStack<E> {

private E[] elements;

private int top;

// Constructor and other methods...

public E popSecond() {

if (top < 2) {

throw new NoSuchElementException("Stack does not have a second element.");

}

E secondElement = elements[top - 2];

top -= 2;

return secondElement;

}

}

In this implementation, the popSecond() method removes and returns the second element from the top of the stack. Here's a breakdown of the implementation:

1. We check if the top index is less than 2, indicating that the stack does not have a second element. In such a case, we throw a NoSuchElementException with an appropriate error message.
2. If the stack has at least two elements, we access the element at the index top - 2 and store it in the secondElement variable.
3. We decrement the top index by 2 to remove the second element from the stack.
4. Finally, we return the secondElement.

Note that this implementation assumes that the ArrayStack class has a member variable elements of type E[] that stores the elements, and an integer top that represents the index of the top element. Adjust the implementation as per your actual class structure.

Remember to import java.util.NoSuchElementException if it's not already imported in your code.

10-

public class LinkedStack<E> {

private Node<E> top;

// Constructor and other methods...

public E bottom() {

if (top == null) {

throw new NoSuchElementException("Stack is empty.");

}

Node<E> currentNode = top;

while (currentNode.next != null) {

currentNode = currentNode.next;

}

return currentNode.data;

}

private static class Node<E> {

E data;

Node<E> next;

Node(E data, Node<E> next) {

this.data = data;

this.next = next;

}

}

}

11-

public class ArrayStack<E> {

private E[] elements;

private int top;

// Constructor and other methods...

public E popBottom() {

if (top < 1) {

throw new NoSuchElementException("Stack is empty.");

}

E bottomElement = elements[0];

System.arraycopy(elements, 1, elements, 0, top - 1);

top--;

return bottomElement;

}

}

1. - We check if the top index is less than 1, indicating that the stack is empty. In such a case, we throw a NoSuchElementException with an appropriate error message.
2. If the stack is not empty, we store the element at index 0, which represents the bottom element, in the bottomElement variable.
3. We use System.arraycopy() to shift the remaining elements in the elements array to the left, effectively removing the bottom element.
4. We decrement the top index by 1 to reflect the removal of the bottom element.
5. Finally, we return the bottomElement.

Note that this implementation assumes that the ArrayStack class has a member variable elements of type E[] that stores the elements, and an integer top that represents the index of the top element. Adjust the implementation as per your actual class structure.

11-

public class ArrayStack<E> {

private E[] elements;

private int top;

// Constructor and other methods...

public E popBottom() {

if (top < 1) {

throw new NoSuchElementException("Stack is empty.");

}

E bottomElement = elements[top - 1];

top--;

return bottomElement;

}

}

- In this implementation, the popBottom() method removes and returns the bottom element of the stack. Here's a breakdown of the implementation:

1. We check if the top index is less than 1, indicating that the stack is empty. In such a case, we throw a NoSuchElementException with an appropriate error message.
2. If the stack is not empty, we access the element at index top - 1, which represents the bottom element, and store it in the bottomElement variable.
3. We decrement the top index by 1 to reflect the removal of the bottom element.
4. Finally, we return the bottomElement.

Note that this implementation assumes that the ArrayStack class has a member variable elements of type E[] that stores the elements, and an integer top that represents the index of the top element. Adjust the implementation as per your actual class structure.

12-

**Here's the analysis of the code and answers to your questions:**

**a) Contents of the stack after execution:**

* Stack: [2, 8, 14, 20]

**b) Values of count and top:**

* count: 4 (number of elements in the stack)
* top: 4 (index of the top element)

**c) Element returned by top():**

* The top() method would return 20.

**d) Is the stack full?**

* No, the stack is not full. It has a capacity of 10, and there are only 4 elements in it.

**e) Making the stack return to the empty state:**

Java

while (!stack.isEmpty()) {

stack.pop();

}

**Explanation:**

1. The stack is initialized with a capacity of 10, but it's initially empty (size = 0, top = 0).
2. The for loop iterates from 1 to 10:
   * If i % 3 != 0, the value i \* 2 is pushed onto the stack.
   * If i % 3 == 0, the top element is popped from the stack.
3. The values pushed onto the stack are: 2, 8, 14, 20.
4. The values popped from the stack are: 6 (when i = 3), 18 (when i = 6), 36 (when i = 9).
5. After the loop, the stack contains the elements that were pushed but not popped: 2, 8, 14, 20.
6. The count is 4 because there are 4 elements in the stack.
7. The top is 4 because the last element pushed (20) is at index 4.
8. The stack is not full because it has a capacity of 10, and only 4 elements are currently in it.
9. To make the stack empty, you can repeatedly pop elements until it's empty.