Q1:

الحالة الأولية: الستاك فارغة.

stack.push(new Character('A')): الستاك يحتوي الآن على ['A'].

stack.push(new Character('B')): الستاك يحتوي الآن على ['A', 'B'].

stack.push(new Character('C')): الستاك يحتوي الآن على ['A', 'B', 'C'].

stack.pop(): يتم إزالة العنصر الأعلى 'C' من الستاك. الستاك يحتوي الآن على ['A', 'B'].

stack.pop(): يتم إزالة العنصر الأعلى 'B' من الستاك. الستاك يحتوي الآن على ['A'].

stack.push(new Character('D')): الستاك يحتوي الآن على ['A', 'D'].

stack.push(new Character('E')): الستاك يحتوي الآن على ['A', 'D', 'E'].

stack.push(new Character('F')): الستاك يحتوي الآن على ['A', 'D', 'E', 'F'].

stack.pop(): يتم إزالة العنصر الأعلى 'F' من الستاك. الستاك يحتوي الآن على ['A', 'D', 'E'].

stack.push(new Character('G')): الستاك يحتوي الآن على ['A', 'D', 'E', 'G'].

stack.pop(): يتم إزالة العنصر الأعلى 'G' من الستاك. الستاك يحتوي الآن على ['A', 'D', 'E'].

stack.pop(): يتم إزالة العنصر الأعلى 'E' من الستاك. الستاك يحتوي الآن على ['A', 'D'].

stack.pop(): يتم إزالة العنصر الأعلى 'D' من الستاك. الستاك يحتوي الآن على ['A'].

stack.pop(): يتم إزالة العنصر الأعلى 'A' من الستاك. الستاك الآن فارغ.

نهاية تنفيذ الكود..

Q2:

في هذا السياق، نحن بصدد حساب حجم الستاك الحالي وقيمة المتغير العضوي `t` استنادًا إلى العمليات التي تم تنفيذها على الستاك.

المعطيات المتاحة:

- إجمالي عدد عمليات الإضافة (push): 25 عملية.

- إجمالي عدد عمليات استعراض العنصر الأعلى (top): 12 عملية.

- إجمالي عدد عمليات إزالة العنصر الأعلى (pop): 10 عمليات (3 منها أعادت قيمة فارغة للدلالة على ستاك فارغ).

لنحلل العمليات:

1. عمليات الإضافة (push): كل عملية إضافة تزيد حجم الستاك بمقدار 1.

إجمالي عمليات الإضافة = 25

حجم الستاك بعد عمليات الإضافة = 25

2. عمليات إزالة العنصر الأعلى (pop): كل عملية إزالة تقلص حجم الستاك بمقدار 1.

إجمالي عمليات الإزالة = 10

حجم الستاك بعد عمليات الإزالة = 25 - 10 = 15

3. عمليات استعراض العنصر الأعلى (top): عمليات الاستعراض لا تؤثر على حجم الستاك.

الآن، دعونا نحسب قيمة المتغير العضوي `t`:

- المتغير العضوي `t` يمثل مؤشر العنصر الأعلى في الستاك.

- نظرًا لأن الستاك يستخدم فهرسًا يبدأ من الصفر، فإن مؤشر العنصر الأعلى سيكون `الحجم - 1`.

بالتالي:

- حجم الستاك = 15

- قيمة المتغير العضوي `t` = 15 - 1 = 14

إذاً، حجم الستاك الحالي هو 15، وقيمة المتغير العضوي `t` هي 14.

Q3:

تقييم التعبيرات اللاحقة التالية (صح أم خطأ):

أ. 8 2 + 3 \* 16 4 / - =

ب. 12 2 5 5 1 / / \* 8 7 + - =

ج. 70 14 4 5 15 3 / \* - / 6 + =

د. 3 5 6 \* + 13 - 18 2 / + =

لحساب قيمة التعبيرات اللاحقة، يمكننا استخدام ستاك لتتبع النتائج الوسيطة. سنقوم بتحليل كل تعبير على حدة:

أ. 8 2 + 3 \* 16 4 / - =

- نبدأ بستاك فارغ.

- نضع العدد 8 في الستاك. الستاك: [8]

- نضع العدد 2 في الستاك. الستاك: [8, 2]

- نقوم بعملية الجمع: 8 + 2 = 10. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [10]

- نضع العدد 3 في الستاك. الستاك: [10, 3]

- نقوم بعملية الضرب: 10 \* 3 = 30. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [30]

- نضع العدد 16 في الستاك. الستاك: [30, 16]

- نضع العدد 4 في الستاك. الستاك: [30, 16, 4]

- نقوم بعملية القسمة: 16 / 4 = 4. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [30, 4]

- نقوم بعملية الطرح: 30 - 4 = 26. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [26]

- النتيجة النهائية هي 26.

ب. 12 2 5 5 1 / / \* 8 7 + - =

- نبدأ بستاك فارغ.

- نضع العدد 12 في الستاك. الستاك: [12]

- نضع العدد 2 في الستاك. الستاك: [12, 2]

- نضع العدد 5 في الستاك. الستاك: [12, 2, 5]

- نضع العدد 5 في الستاك. الستاك: [12, 2, 5, 5]

- نضع العدد 1 في الستاك. الستاك: [12, 2, 5, 5, 1]

- نقوم بعملية القسمة: 5 / 1 = 5. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [12, 2, 5, 5]

- نقوم بعملية القسمة: 5 / 5 = 1. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [12, 2, 1]

- نقوم بعملية الضرب: 2 \* 1 = 2. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [12, 2]

- نضع العدد 8 في الستاك. التقييم التعبيرات اللاحقة:

أ. 8 2 + 3 \* 16 4 / - =

- نبدأ بستاك فارغ.

- نضع العدد 8 في الستاك. الستاك: [8]

- نضع العدد 2 في الستاك. الستاك: [8, 2]

- نقوم بعملية الجمع: 8 + 2 = 10. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [10]

- نضع العدد 3 في الستاك. الستاك: [10, 3]

- نقوم بعملية الضرب: 10 \* 3 = 30. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [30]

- نضع العدد 16 في الستاك. الستاك: [30, 16]

- نضع العدد 4 في الستاك. الستاك: [30, 16, 4]

- نقوم بعملية القسمة: 16 / 4 = 4. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [30, 4]

- نقوم بعملية الطرح: 30 - 4 = 26. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [26]

- النتيجة النهائية هي 26.

ب. 12 2 5 5 1 / / \* 8 7 + - =

- نبدأ بستاك فارغ.

- نضع العدد 12 في الستاك. الستاك: [12]

- نضع العدد 2 في الستاك. الستاك: [12, 2]

- نضع العدد 5 في الستاك. الستاك: [12, 2, 5]

- نضع العدد 5 في الستاك. الستاك: [12, 2, 5, 5]

- نضع العدد 1 في الستاك. الستاك: [12, 2, 5, 5, 1]

- نقوم بعملية القسمة: 5 / 1 = 5. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [12, 2, 5, 5]

- نقوم بعملية القسمة: 5 / 5 = 1. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [12, 2, 1]

- نقوم بعملية الضرب: 2 \* 1 = 2. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [12, 2]

- نضع العدد 8 في الستاك. الستاك: [12, 2, 8]

- نضع العدد 7 في الستاك. الستاك: [12, 2, 8, 7]

- نقوم بعملية الجمع: 8 + 7 = 15. نستبدل أعلى عنصرين في الستاك بالنتيجة. الستاك: [12, 2, 15]

- نقوم بعملية الطرح: 12 - 15 = -3. نستبدل أعلى عنصرين في الستاك بالنتيجة.

Q4:

a. Infix expression: (A + B) \* (C + D) - E

Postfix notation: AB+CD+\*E-

Java code using stack operations:

```java

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String postfix) {

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < postfix.length(); i++) {

char ch = postfix.charAt(i);

if (Character.isLetterOrDigit(ch)) {

stack.push(ch - 'A'); // Assuming operands are single letters from A to Z

} else {

int operand2 = stack.pop();

int operand1 = stack.pop();

switch(ch) {

case '+':

stack.push(operand1 + operand2);

break;

case '-':

stack.push(operand1 - operand2);

break;

case '\*':

stack.push(operand1 \* operand2);

break;

case '/':

stack.push(operand1 / operand2);

break;

}

}

}

return stack.pop();

}

public static void main(String[] args) {

String postfix = "AB+CD+\*E-";

int result = evaluatePostfix(postfix);

System.out.println("Result: " + result);

}

}

```

b. Infix expression: A - (B + C) \* D + E / F

Postfix notation: ABC+D\*-EF/+

Java code using stack operations:

```java

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String postfix) {

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < postfix.length(); i++) {

char ch = postfix.charAt(i);

if (Character.isLetterOrDigit(ch)) {

stack.push(ch - 'A'); // Assuming operands are single letters from A to Z

} else {

int operand2 = stack.pop();

int operand1 = stack.pop();

switch(ch) {

case '+':

stack.push(operand1 + operand2);

break;

case '-':

stack.push(operand1 - operand2);

break;

case '\*':

stack.push(operand1 \* operand2);

break;

case '/':

stack.push(operand1 / operand2);

break;

}

}

}

return stack.pop();

}

public static void main(String[] args) {

String postfix = "ABC+D\*-EF/+";

int result = evaluatePostfix(postfix);

System.out.println("Result: " + result);

}

}

```

Note: The Java code assumes that the operands are single letters from A to Z. You may need to modify it according to your specific requirements.

c. Postfix notation: ((A + B) / (C - D) + E) \* F - G

Java code using stack operations:

```java

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String postfix) {

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < postfix.length(); i++) {

char ch = postfix.charAt(i);

if (Character.isLetterOrDigit(ch)) {

stack.push(ch - 'A'); // Assuming operands are single letters from A to Z

} else {

int operand2 = stack.pop();

int operand1 = stack.pop();

switch(ch) {

case '+':

stack.push(operand1 + operand2);

break;

case '-':

stack.push(operand1 - operand2);

break;

case '\*':

stack.push(operand1 \* operand2);

break;

case '/':

stack.push(operand1 / operand2);

break;

}

}

}

return stack.pop();

}

public static void main(String[] args) {

String postfix = "AB+CD-/E+F\*G-";

int result = evaluatePostfix(postfix);

System.out.println("Result: " + result);

}

}

```

d. Postfix notation: A B C D + \* + E F / G \* - H +

Java code using stack operations:

```java

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String postfix) {

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < postfix.length(); i++) {

char ch = postfix.charAt(i);

if (Character.isLetterOrDigit(ch)) {

stack.push(ch - 'A'); // Assuming operands are single letters from A to Z

} else {

int operand2 = stack.pop();

int operand1 = stack.pop();

switch(ch) {

case '+':

stack.push(operand1I apologize for the incomplete Java code in the previous response. Here's the corrected version for the fourth expression:

d. Postfix notation: A B C D + \* + E F / G \* - H +

Java code using stack operations:

```java

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String postfix) {

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < postfix.length(); i++) {

char ch = postfix.charAt(i);

if (Character.isLetterOrDigit(ch)) {

stack.push(ch - 'A'); // Assuming operands are single letters from A to Z

} else {

int operand2 = stack.pop();

int operand1 = stack.pop();

switch(ch) {

case '+':

stack.push(operand1 + operand2);

break;

case '-':

stack.push(operand1 - operand2);

break;

case '\*':

stack.push(operand1 \* operand2);

break;

case '/':

stack.push(operand1 / operand2);

break;

}

}

}

return stack.pop();

}

public static void main(String[] args) {

String postfix = "ABCD+\*+EF/G\*-H+";

int result = evaluatePostfix(postfix);

System.out.println("Result: " + result);

}

}

```

Please note that the above Java code assumes that the operands are single letters from A to Z. You may need to modify it according to your specific requirements.

Q5:

إليك تعريف لقالب الدالة `printListReverse` التي تستخدم الستاك لطباعة قائمة مرتبطة بترتيب عكسي. يفترض أن هذه الدالة هي عضو في فئة `linkedStack`:

```cpp

#include <iostream>

#include <stack>

template <typename T>

class linkedStack {

private:

struct Node {

T data;

Node\* next;

};

Node\* top;

public:

// البناء

linkedStack() : top(nullptr) {}

// الهدم

~linkedStack() {

while (!isEmpty()) {

pop();

}

}

// دفع عنصر إلى الستاك

void push(const T& element) {

Node\* newNode = new Node;

newNode->data = element;

newNode->next = top;

top = newNode;

}

// إخراج العنصر الأعلى من الستاك

void pop() {

if (isEmpty()) {

std::cout << "الستاك فارغ. لا يمكن إخراج عنصر." << std::endl;

return;

}

Node\* temp = top;

top = top->next;

delete temp;

}

// التحقق مما إذا كان الستاك فارغًا

bool isEmpty() const {

return top == nullptr;

}

// طباعة القائمة المرتبطة بترتيب عكسي باستخدام الستاك

void printListReverse() const {

if (isEmpty()) {

std::cout << "القائمة المرتبطة فارغة." << std::endl;

return;

}

std::stack<T> stack;

// دفع العناصر إلى الستاك

Node\* current = top;

while (current != nullptr) {

stack.push(current->data);

current = current->next;

}

// إخراج وطباعة العناصر من الستاك

while (!stack.empty()) {

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

stack.pop();

}

std::cout << std::endl;

}

};

```

في الشفرة أعلاه، فئة `linkedStack` هي تنفيذ لستاك باستخدام قائمة مرتبطة. الدالة `printListReverse` تستخدم الستاك لعكس ترتيب العناصر في القائمة المرتبطة وطباعتها بترتيب عكسي. تقوم الدالة أولاً بدفع جميع عناصر القائمة المرتبطة إلى الستاك، ثم تقوم بإخراج وطباعة العناصر من الستاك. تتم طباعة العناصر بترتيب عكسي نظرًا لأن الستاك يتبع مبدأ "الدخول الأخير - الخروج الأول" (LIFO).

Q6:

Here's the implementation of the `reverse` method using only the `push()`, `top()`, `pop()`, and `isEmpty()` methods:

```java

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

if (stack.isEmpty()) {

return; // Stack is already empty, nothing to reverse

}

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

// Push all elements from the original stack to the temporary stack

while (!stack.isEmpty()) {

tempStack.push(stack.top());

stack.pop();

}

// Push all elements from the temporary stack back to the original stack

while (!tempStack.isEmpty()) {

stack.push(tempStack.top());

tempStack.pop();

}

}

```

في الشفرة أعلاه، الدالة `reverse` تأخذ `ArrayStack` كمعامل وتقوم بعكس محتواها. تقوم الدالة بإنشاء `tempStack` كستاك مؤقت لاحتواء العناصر أثناء عملية العكس.

أولاً، تتحقق الدالة مما إذا كان الستاك الأصلي فارغًا بالفعل. إذا كان كذلك، فلا يوجد شيء يحتاج لعكسه، لذا تعود الدالة مباشرةً.

ثم تدخل في حلقة حيث تقوم بإخراج العناصر من الستاك الأصلي ووضعها في الستاك المؤقت حتى يصبح الستاك الأصلي فارغًا. هذا يعكس ترتيب العناصر.

أخيرًا، تدخل في حلقة أخرى حيث تقوم بإخراج العناصر من الستاك المؤقت ووضعها في الستاك الأصلي، مع استعادة ترتيب العناصر الأصلي.

بعد اكتمال الدالة، سيحتوي الستاك الأصلي على العناصر في ترتيب عكسي.

Q7:

Here's the implementation of the `popBottom` method using only the `push()`, `top()`, `pop()`, and `isEmpty()` methods:

```java

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

// Check if the stack is empty

if (stack.isEmpty()) {

throw new EmptyStackException();

}

// Create a temporary stack to hold the elements

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

// Move all elements from the original stack to the temporary stack

while (!stack.isEmpty()) {

tempStack.push(stack.top());

stack.pop();

}

// Pop the bottom element from the temporary stack

E bottomElement = tempStack.pop();

// Move the remaining elements back to the original stack

while (!tempStack.isEmpty()) {

stack.push(tempStack.top());

tempStack.pop();

}

return bottomElement;

}

```

In the above code, the `popBottom` method takes a `LinkedStack` as a parameter and removes and returns the bottom element of the stack. It uses a temporary stack (`tempStack`) to hold the elements during the process.

First, it checks if the original stack is empty. If it is, an `EmptyStackException` is thrown.

Then, it enters a loop where it moves all elements from the original stack to the temporary stack by popping them from the original stack and pushing them onto the temporary stack. This effectively reverses the order of elements.

Next, it pops the bottom element from the temporary stack and stores it in a variable `bottomElement`.

Finally, it enters another loop where it moves the remaining elements from the temporary stack back to the original stack by popping them from the temporary stack and pushing them onto the original stack.

After the method completes, the bottom element is removed from the original stack, and it is returned as the result of the method.

Q8:

```java

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

// التحقق مما إذا كان الستاك فارغًا

if (stack.isEmpty()) {

throw new EmptyStackException();

}

// إنشاء ستاك مؤقت لاحتواء العناصر

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

// نقل جميع العناصر من الستاك الأصلي إلى الستاك المؤقت

while (!stack.isEmpty()) {

tempStack.push(stack.top());

stack.pop();

}

// إخراج العنصر الأسفل من الستاك المؤقت

E bottomElement = tempStack.pop();

// نقل العناصر المتبقية إلى الستاك الأصلي

while (!tempStack.isEmpty()) {

stack.push(tempStack.top());

tempStack.pop();

}

return bottomElement;

}

```

في الشفرة أعلاه، الدالة `popBottom` تأخذ `LinkedStack` كمعامل وتقوم بإزالة وإرجاع العنصر الأسفل من الستاك. تستخدم الدالة ستاك مؤقت (`tempStack`) لاحتواء العناصر أثناء العملية.

أولاً، تتحقق الدالة مما إذا كان الستاك الأصلي فارغًا. إذا كان كذلك، فإنها تقوم برمي استثناء `EmptyStackException`.

ثم تدخل في حلقة حيث تقوم بنقل جميع العناصر من الستاك الأصلي إلى الستاك المؤقت عن طريق إخراجها من الستاك الأصلي ووضعها في الستاك المؤقت. هذا يعكس ترتيب العناصر.

بعد ذلك، تقوم بإخراج العنصر الأسفل من الستاك المؤقت وتخزينه في متغير `bottomElement`.

أخيرًا، تدخل في حلقة أخرى حيث تقوم بنقل العناصر المتبقية من الستاك المؤقت إلى الستاك الأصلي عن طريق إخراجها من الستاك المؤقت ووضعها في الستاك الأصلي.

بعد اكتمال الدالة، يتم إزالة العنصر الأسفل من الستاك الأصلي ويتم إرجاعه كنتيجة للدالة.

Q9:

لإضافة الدالة `popSecond()` إلى صنف `ArrayStack`، يمكنك تعديل الصنف كما يلي:

```java

public class ArrayStack<E> {

private E[] elements;

private int size;

private static final int DEFAULT\_CAPACITY = 10;

public ArrayStack() {

elements = (E[]) new Object[DEFAULT\_CAPACITY];

size = 0;

}

// باقي الدوال لصنف ArrayStack...

public E popSecond() {

if (size < 2) {

throw new NoSuchElementException("الستاك لا يحتوي على عنصر ثاني");

}

E secondElement = elements[size - 2];

size--;

return secondElement;

}

}

```

في الشفرة أعلاه، تمت إضافة الدالة `popSecond()` إلى صنف `ArrayStack`. تقوم الدالة بإزالة وإرجاع العنصر الثاني في الستاك.

تقوم الدالة أولاً بالتحقق مما إذا كان الستاك يحتوي على على الأقل عنصرين (`size >= 2`). إذا لم يكن الأمر كذلك، فإنها تقذف استثناء `NoSuchElementException` مع رسالة مناسبة.

إذا كان الستاك يحتوي على على الأقل عنصرين، تقوم الدالة بالحصول على العنصر الثاني عن طريق الوصول إلى `elements[size - 2]`. الفهرس `size - 2` يشير إلى العنصر الثاني لأن الفهارس تبدأ من الصفر.

بعد ذلك، تقوم الدالة بتنقيص قيمة المتغير `size` لتعكس إزالة العنصر الثاني.

أخيرًا، تقوم الدالة بإرجاع العنصر الثاني الذي تم إزالته.

ملاحظة: تأكد من تنفيذ باقي الدوال الضرورية لصنف `ArrayStack` إذا لم يتم تنفيذها بالفعل.

Q10:

لإضافة الطريقة `bottom()` إلى صنف `LinkedStack`، يمكنك تعديل الصنف على النحو التالي:

```java

public class LinkedStack<E> {

private Node<E> top;

private int size;

private static class Node<E> {

private E element;

private Node<E> next;

public Node(E element, Node<E> next) {

this.element = element;

this.next = next;

}

}

public LinkedStack() {

top = null;

size = 0;

}

// باقي الطرق لصنف LinkedStack...

public E bottom() {

if (isEmpty()) {

throw new NoSuchElementException("الستاك فارغ");

}

Node<E> current = top;

while (current.next != null) {

current = current.next;

}

return current.element;

}

}

```

في الشيفرة أعلاه، تم إضافة الطريقة `bottom()` إلى صنف `LinkedStack`. تقوم الطريقة بإعادة العنصر الأسفل في الستاك.

تقوم الطريقة أولاً بالتحقق مما إذا كان الستاك فارغًا باستخدام الطريقة `isEmpty()`، إذا كان الستاك فارغًا، فإنها تقذف استثناء `NoSuchElementException` مع رسالة مناسبة.

إذا لم يكن الستاك فارغًا، تبدأ الطريقة في عبور قائمة الارتباط من العقدة الأعلى `top` حتى تصل إلى العقدة الأخيرة. العقدة الأخيرة ستكون العنصر الأسفل في الستاك.

أخيرًا، تقوم الطريقة بإرجاع العنصر الموجود في العقدة الأخيرة، والذي يمثل العنصر الأسفل في الستاك.

ملاحظة: تأكد من تنفيذ الطرق الأخرى اللازمة لصنف `LinkedStack` إذا لم يتم تنفيذها بالفعل.

Q11:

لإضافة الطريقة `popbottom()` إلى صنف `ArrayStack`، يمكنك تعديل الصنف على النحو التالي:

```java

public class ArrayStack<E> {

private E[] elements;

private int size;

private int capacity;

public ArrayStack(int initialCapacity) {

elements = (E[]) new Object[initialCapacity];

size = 0;

capacity = initialCapacity;

}

// باقي الطرق لصنف ArrayStack...

public E popbottom() {

if (isEmpty()) {

throw new NoSuchElementException("الستاك فارغ");

}

E bottomElement = elements[0];

// نقل العناصر نحو بداية المصفوفة

for (int i = 0; i < size - 1; i++) {

elements[i] = elements[i + 1];

}

size--;

elements[size] = null; // تعيين العنصر الأخير إلى قيمة null

return bottomElement;

}

}

```

في الشيفرة أعلاه، تم إضافة الطريقة `popbottom()` إلى صنف `ArrayStack`. تقوم الطريقة بإزالة وإرجاع العنصر الأسفل في الستاك.

تقوم الطريقة أولاً بالتحقق مما إذا كان الستاك فارغًا باستخدام الطريقة `isEmpty()`، إذا كان الستاك فارغًا، فإنها تقذف استثناء `NoSuchElementException` مع رسالة مناسبة.

إذا لم يكن الستاك فارغًا، يتم استرداد العنصر الأسفل، والذي هو العنصر في الفهرس 0 في مصفوفة `elements`.

ثم، تقوم الطريقة بنقل جميع العناصر نحو بداية المصفوفة عن طريق التكرار من الفهرس 0 إلى `size - 1` وتعيين كل عنصر إلى الفهرس السابق.

بعد نقل العناصر، تقوم الطريقة بتنقيص قيمة المتغير `size` وتعيين العنصر الأخير (`elements[size]`) إلى `null` لإزالة الإشارة إلى العنصر الأسفل السابق.

أخيرًا، تقوم الطريقة بإرجاع العنصر الأسفل الذي تمت إزالته.

ملاحظة: تأكد من تنفيذ الطرق الأخرى اللازمة لصنف `ArrayStack` إذا لم يتم تنفيذها بالفعل.

Q12:

وفقًا لقطعة الشيفرة المقدمة، دعنا نحلل التنفيذ خطوة بخطوة:

```java

public static void main(String[] args) {

Stack<Integer> stack = new ArrayStack<>(10);

for (int i = 1; i <= 10; i++) {

if (i % 3 != 0) {

stack.push(i \* 2);

} else {

stack.pop();

}

}

}

```

أ) محتويات (العناصر) في الستاك بعد تنفيذ الشيفرة ستكون `[2, 4, 8, 10]`. ذلك لأن الحلقة تتكرر من `i = 1` إلى `i = 10`، ولكل `i` الذي ليس قابلًا للقسمة على 3، يتم إضافة `i \* 2` إلى الستاك. عندما يكون `i` قابلاً للقسمة على 3، يتم تنفيذ عملية `pop()`، مما يزيل العنصر الأعلى في الستاك.

ب) بعد تنفيذ الشيفرة، قيمة المتغير `size` ستكون `4`، مشيرة إلى أن هناك أربعة عناصر في الستاك. قيمة المتغير `top` ستكون `4`، والذي يمثل فهرس العنصر الأعلى في الستاك (بناءً على فرضية أن الفهرس يستخدم نظام ترقيم يبدأ من 0).

ج) طريقة `top()` ستعيد القيمة `10`، حيث تعيد العنصر الموجود في الأعلى من الستاك، وبعد التعامل مع الشيفرة، `10` هو العنصر الأعلى.

د) لا، الستاك ليس ممتلئًا. الستاك لديه سعة `10`، ولكن تم إضافة فقط `4` عناصر إليه، مما يترك `6` فتحات فارغة.

هـ) لجعل الستاك يعود إلى الحالة الفارغة، يمكنك استدعاء طريقة `clear()` (بشرط أن تكون مطبقة في واجهة `Stack` أو الصنف الخاص به). طريقة `clear()` تزيل جميع العناصر من الستاك، وتعيد حجمه إلى `0`، مما يجعله فارغًا مرة أخرى.

ملاحظة: في الشيفرة المقدمة، يبدو أن هناك اختلافًا بين تعريف المتغير `stack` (`Stack<int> stack`) وتكوينه (`new ArrayStack(10)`). بناءً على افتراض أن `ArrayStack` تنفذ واجهة `Stack`، يجب تكوينها كـ `new ArrayStack<Integer>(10)` لتتوافق مع النوع المعلن.