## Topics

1. Create Stack Interface

public interface Stack<E> {

/\*\*

\* Pushes an element onto the top of the stack.

\*

\* @param element the element to be pushed onto the stack

\*/

void push(E element);

/\*\*

\* Removes and returns the element at the top of the stack.

\*

\* @return the element at the top of the stack

\* @throws EmptyStackException if the stack is empty

\*/

E pop();

/\*\*

\* Returns the element at the top of the stack without removing it.

\*

\* @return the element at the top of the stack

\* @throws EmptyStackException if the stack is empty

\*/

E peek();

/\*\*

\* Checks if the stack is empty.

\*

\* @return true if the stack is empty, false otherwise

\*/

boolean isEmpty();

/\*\*

\* Returns the number of elements in the stack.

\*

\* @return the number of elements in the stack

\*/

int size();

}

1. Create Stack Using Array

import java.util.EmptyStackException;

public class ArrayStack<E> implements Stack<E> {

private static final int DEFAULT\_CAPACITY = 10;

private E[] array;

private int size;

public ArrayStack() {

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

size = 0;

}

@Override

public void push(E element) {

if (size == array.length) {

resizeArray(array.length \* 2);

}

array[size++] = element;

}

@Override

public E pop() {

if (isEmpty()) { throw new EmptyStackException();

}

E element = array[--size];

array[size] = null;

if (size > 0 && size == array.length / 4) {

resizeArray(array.length / 2);

}

return element;

}

@Override

public E peek() {

if (isEmpty()) {

throw new EmptyStackException();

}

return array[size - 1];

}

@Override

public boolean isEmpty() {

return size == 0;

}

@Override

public int size() {

return size;

}

private void resizeArray(int newCapacity) {

E[] newArray = (E[]) new Object[newCapacity];

System.arraycopy(array, 0, newArray, 0, size);

array = newArray;

}

}

1. Create Stack Using Linked Lists

import java.util.EmptyStackException;

public class LinkedStack<E> implements Stack<E> {

private Node<E> top;

private int size;

public LinkedStack() {

top = null;

size = 0;

}

@Override

public void push(E element) {

Node<E> newNode = new Node<>(element, top);

top = newNode;

size++;

}

@Override

public E pop() {

if (isEmpty()) {

throw new EmptyStackException();

}

E element = top.getElement();

top = top.getNext();

size--;

return element;

}

@Override

public E peek() {

if (isEmpty()) {

throw new EmptyStackException();

}

return top.getElement();

}

@Override

public boolean isEmpty() {

return size == 0;

}

@Override

public int size() {

return 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 E getElement() {

return element;

}

public Node<E> getNext() {

return next;

}

}

}

1. Implement Basic Methods of Stack

* isEmpty()
* size()
* top()
* push(E e)
* pop()

import java.util.EmptyStackException;

public class Stack<E> {

private Node<E> top;

private int size;

public Stack() {

top = null;

size = 0;

}

/\*\*

\* Checks if the stack is empty.

\*

\* @return true if the stack is empty, false otherwise

\*/

public boolean isEmpty() {

return size == 0;

}

/\*\*

\* Returns the number of elements in the stack.

\*

\* @return the number of elements in the stack

\*/

public int size() {

return size;

}

/\*\*

\* Returns the element at the top of the stack without removing it.

\*

\* @return the element at the top of the stack

\* @throws EmptyStackException if the stack is empty

\*/

public E top() {

if (isEmpty()) {

throw new EmptyStackException();

}

return top.getElement();

}

/\*\*

\* Pushes an element onto the top of the stack.

\*

\* @param element the element to be pushed onto the stack

\*/

public void push(E element) {

Node<E> newNode = new Node<>(element, top);

top = newNode;

size++;

}

/\*\*

\* Removes and returns the element at the top of the stack.

\*

\* @return the element at the top of the stack

\* @throws EmptyStackException if the stack is empty

\*/

public E pop() {

if (isEmpty()) {

throw new EmptyStackException();

}

E element = top.getElement();

top = top.getNext();

size--;

return element;

}

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 E getElement() {

return element;

}

public Node<E> getNext() {

return next;

}

}

}

## Homework

1. Implement a method with signature transfer(S, T) that transfers all elements from stack S onto stack T, so that the element that starts at the top of S is the first to be inserted onto T, and the element at the bottom of S ends up at the top of T.

import java.util.Stack;

public class StackTransfer {

public static <E> void transfer(Stack<E> sourceStack, Stack<E> targetStack) {

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

// Transfer elements from sourceStack to tempStack

while (!sourceStack.isEmpty()) {

tempStack.push(sourceStack.pop());

}

// Transfer elements from tempStack to targetStack

while (!tempStack.isEmpty()) {

targetStack.push(tempStack.pop());

}

}

public static void main(String[] args) {

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

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

sourceStack.push(1);

sourceStack.push(2);

sourceStack.push(3);

System.out.println("Source Stack: " + sourceStack);

System.out.println("Target Stack: " + targetStack);

transfer(sourceStack, targetStack);

System.out.println("Source Stack: " + sourceStack);

System.out.println("Target Stack: " + targetStack);

}

}

1. Give a recursive method for removing all the elements from a stack.

import java.util.Stack;

public class StackUtils {

public static <E> void removeAll(Stack<E> stack) {

if (!stack.isEmpty()) {

stack.pop();

removeAll(stack);

}

}

public static void main(String[] args) {

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

stack.push(1);

stack.push(2);

stack.push(3);

System.out.println("Stack before removal: " + stack);

removeAll(stack);

System.out.println("Stack after removal: " + stack);

}

}

1. Postfix notation is an unambiguous way of writing an arithmetic expression without parentheses. It is defined so that if “(exp1)op(exp2)” is a normal fully parenthesized expression whose operation is op, the postfix version of this is “pexp1 pexp2 op”, where pexp1 is the postfix version of exp1 and pexp2 is the postfix version of exp2. The postfix version of a single number or variable is just that number or variable. So, for example, the postfix version of “((5 + 2) ∗ (8 − 3))/4” is “5 2 + 8 3 − ∗ 4 /”. Describe a nonrecursive way of evaluating an expression in postfix notation.

import java.util.Stack;

public class PostfixConverter {

public static String infixToPostfix(String infixExpression) {

StringBuilder postfixExpression = new StringBuilder();

Stack<Character> operatorStack = new Stack<>();

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

char ch = infixExpression.charAt(i);

if (Character.isLetterOrDigit(ch)) {

postfixExpression.append(ch);

} else if (ch == '(') {

operatorStack.push(ch);

} else if (ch == ')') {

while (!operatorStack.isEmpty() && operatorStack.peek() != '(') {

postfixExpression.append(operatorStack.pop());

}

if (!operatorStack.isEmpty() && operatorStack.peek() != '(') {

throw new IllegalArgumentException("Invalid infix expression");

}

operatorStack.pop(); // Remove '(' from stack

} else {

while (!operatorStack.isEmpty() && precedence(ch) <= precedence(operatorStack.peek())) {

postfixExpression.append(operatorStack.pop());

}

operatorStack.push(ch);

}

}

while (!operatorStack.isEmpty()) {

if (operatorStack.peek() == '(') {

throw new IllegalArgumentException("Invalid infix expression");

}

postfixExpression.append(operatorStack.pop());

}

return postfixExpression.toString();

}

private static int precedence(char operator) {

switch (operator) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

default:

return -1;

}

}

public static void main(String[] args) {

String infixExpression = "((A+B)\*C)-D";

String postfixExpression = infixToPostfix(infixExpression);

System.out.println("Infix Expression: " + infixExpression);

System.out.println("Postfix Expression: " + postfixExpression);

}

}

1. Implement the clone( ) method for the ArrayStack class.

import java.util.Arrays;

public class ArrayStack<E> implements Cloneable {

private Object[] elements;

private int size;

private static final int DEFAULT\_CAPACITY = 10;

public ArrayStack() {

elements = new Object[DEFAULT\_CAPACITY];

size = 0;

}

public void push(E element) {

if (size == elements.length) {

ensureCapacity(size \* 2);

}

elements[size++] = element;

}

public E pop() {

if (isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

@SuppressWarnings("unchecked")

E element = (E) elements[--size];

elements[size] = null; // Dereference the popped element

return element;

}

public boolean isEmpty() {

return size == 0;

}

private void ensureCapacity(int capacity) {

elements = Arrays.copyOf(elements, capacity);

}

@Override

public ArrayStack<E> clone() {

try {

@SuppressWarnings("unchecked")

ArrayStack<E> clonedStack = (ArrayStack<E>) super.clone();

clonedStack.elements = Arrays.copyOf(elements, size);

return clonedStack;

} catch (CloneNotSupportedException e) {

throw new InternalError(e);

}

}

public static void main(String[] args) {

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

stack.push(1);

stack.push(2);

stack.push(3);

ArrayStack<Integer> clonedStack = stack.clone();

System.out.println("Original Stack: " + stack.pop() + ", " + stack.pop() + ", " + stack.pop());

System.out.println("Cloned Stack: " + clonedStack.pop() + ", " + clonedStack.pop() + ", " + clonedStack.pop());

}

}

1. Implement a program that can input an expression in postfix notation (see Exercise C-6.19) and output its value

import java.util.Stack;

public class PostfixEvaluator {

public static int evaluatePostfix(String postfixExpression) {

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

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

char ch = postfixExpression.charAt(i);

if (Character.isDigit(ch)) {

operandStack.push(ch - '0');

} else if (ch == ' ') {

continue;

} else {

int operand2 = operandStack.pop();

int operand1 = operandStack.pop();

int result;

switch (ch) {

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

result = operand1 / operand2;

break;

default:

throw new IllegalArgumentException("Invalid postfix expression");

}

operandStack.push(result);

}

}

if (operandStack.size() != 1) {

throw new IllegalArgumentException("Invalid postfix expression");

}

return operandStack.pop();

}

public static void main(String[] args) {

String postfixExpression = "82 3 2 \* + 7 -";

int result = evaluatePostfix(postfixExpression);

System.out.println("Postfix Expression: " + postfixExpression);

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

}

}