## Topics

1. Create Stack Interface

public interface Stack<T> {

void push(T item);

T pop();

T peek();

boolean isEmpty();

int size();

}

1. Create Stack Using Array

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

private static final int DEFAULT\_CAPACITY = 10;

private T[] stackArray;

private int top;

public ArrayStack() {

this(DEFAULT\_CAPACITY);

}

public ArrayStack(int initialCapacity) {

stackArray = (T[]) new Object[initialCapacity];

top = -1;

}

@Override

public void push(T item) {

if (top == stackArray.length - 1) {

expandCapacity();

}

top++;

stackArray[top] = item;

}

@Override

public T pop() {

if (isEmpty()) {

throw new EmptyStackException();

}

T item = stackArray[top];

stackArray[top] = null;

top--;

return item;

}

@Override

public T peek() {

if (isEmpty()) {

throw new EmptyStackException();

}

return stackArray[top];

}

@Override

public boolean isEmpty() {

return top == -1;

}

@Override

public int size() {

return top + 1;

}

private void

expandCapacity() {

int newCapacity = stackArray.length \* 2;

stackArray = Arrays.copyOf(stackArray, newCapacity);

}

1. Create Stack Using Linked Lists

public class LinkedListStack<T> implements Stack<T> {

private Node top;

private int size;

private class Node {

T data;

Node next;

public Node(T data) {

this.data = data;

this.next = null;

}

}

public LinkedListStack() {

top = null;

size = 0;

}

@Override

public void push(T item) {

Node newNode = new Node(item);

newNode.next = top;

top = newNode;

size++;

}

@Override

public T pop() {

if (isEmpty()) {

throw new EmptyStackException();

}

T item = top.data;

top = top.next;

size--;

return item;

}

@Override

public T peek() {

if (isEmpty()) {

throw new EmptyStackException();

}

return top.data;

}

@Override

public boolean isEmpty() {

return top == null;

}

@Override

public int size() {

return size;

}

}

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;

private static class Node<E> {

private E data;

private Node<E> next;

public Node(E data) {

this.data = data;

}

}

## 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.

public static <E> void transfer(Stack<E> S, Stack<E> T) {

while (!S.isEmpty()) {

T.push(S.pop());

}

}

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

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

if (!stack.isEmpty()) {

stack.pop();

removeElements(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.

public static double evaluatePostfixExpression(String expression) {

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

String[] tokens = expression.split(" ");

for (String token : tokens) {

if (isNumber(token)) {

stack.push(Double.parseDouble(token));

} else {

double operand2 = stack.pop();

double operand1 = stack.pop();

double result = evaluateOperator(token, operand1, operand2);

stack.push(result);

}

}

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

@SuppressWarnings("unchecked")

@Override

public ArrayStack<E> clone() throws

CloneNotSupportedException {

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

clone.data = data.clone();

return clone;

}

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

public static double evaluatePostfixExpression(Strin

g expression) {

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

String[] tokens = expression.split(" ");

for (String token : tokens) {

if (isNumber(token)) {

stack.push(Double.parseDouble(token));

} else {

double operand2 = stack.pop();

double operand1 =

stack.pop();

double result = evaluateOperator(token, operand1, operand2);

stack.push(result);

}

}

return stack.pop();

}

private static boolean isNumber(String token) {

return token.matches("-?\\d+(\\.\\d+)?");

}

private static double evaluateOperator(String operator, double operand1, double operand2) {

switch (operator) {

case "+":

return operand1 + operand2;

case "-":

return operand1 - operand2;

case "\*":

return operand1 \*

operand2;

case "/":

return operand1 / operand2;

default:

throw new IllegalArgumentException("Invalid operator: " + operator);

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter an expression in postfix notation: ");

String expression = scanner.nextLine();

double result = evaluatePostfixExpression(expression);

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

}