## Topics

1. Create Stack Interface
2. Create Stack Using Array
3. Create Stack Using Linked Lists
4. Implement Basic Methods of Stack

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

## 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 void transfer(Stack<Integer> S, Stack<Integer> T) {

// While S is not empty, pop elements from S and push them onto T

while (!S.isEmpty()) {

T.push(S.pop());

}

}

public static void main(String[] args) {

// Example usage

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

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

// Fill stack S with elements

S.push(1);

S.push(2);

S.push(3);

S.push(4);

S.push(5); // Top of the stack

System.out.println("Stack S before transfer: " + S);

System.out.println("Stack T before transfer: " + T);

// Perform transfer

transfer(S, T);

System.out.println("Stack S after transfer: " + S);

System.out.println("Stack T after transfer: " + T);

}

}

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

import java.util.Stack;

public class RecursiveStackClear {

// Recursive method to remove all elements from the stack

public static void clearStack(Stack<Integer> stack) {

// Base case: If the stack is empty, stop recursion

if (stack.isEmpty()) {

return;

}

// Remove the top element

stack.pop();

// Recursive call to remove the next element

clearStack(stack);

}

public static void main(String[] args) {

// Example usage

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

// Fill the stack with some elements

stack.push(1);

stack.push(2);

stack.push(3);

stack.push(4);

stack.push(5); // Top of the stack

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

// Clear the stack

clearStack(stack);

System.out.println("Stack after clearing: " + 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.

### Algorithm

1. **Initialize a Stack**: Create an empty stack to hold numbers during the evaluation.
2. **Scan the Expression**: Read the postfix expression from left to right.
3. **Process Each Token**:
   * **If the token is a number**: Push it onto the stack.
   * **If the token is an operator**:
     1. Pop the top two numbers from the stack.
     2. Apply the operator to these two numbers (in the correct order: the second popped number is the left operand, and the first popped number is the right operand).
     3. Push the result back onto the stack.
4. **Final Result**: After processing all tokens, the final result will be the only number left in the stack.

### Example Walkthrough

Evaluate the postfix expression:  
5 2 + 8 3 − \* 4 /

* **Step 1**: Initialize an empty stack.
* **Step 2**: Process tokens from left to right:
  1. Token 5: Push → Stack: [5]
  2. Token 2: Push → Stack: [5, 2]
  3. Token +: Pop 5 and 2, compute 5 + 2 = 7, Push 7 → Stack: [7]
  4. Token 8: Push → Stack: [7, 8]
  5. Token 3: Push → Stack: [7, 8, 3]
  6. Token −: Pop 8 and 3, compute 8 − 3 = 5, Push 5 → Stack: [7, 5]
  7. Token \*: Pop 7 and 5, compute 7 \* 5 = 35, Push 35 → Stack: [35]
  8. Token 4: Push → Stack: [35, 4]
  9. Token /: Pop 35 and 4, compute 35 / 4 = 8.75, Push 8.75 → Stack: [8.75]
* **Final Result**: The stack contains [8.75], which is the result.

import java.util.Stack;

public class PostfixEvaluator {

public static double evaluatePostfix(String expression) {

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

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

for (String token : tokens) {

if (isNumeric(token)) {

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

} else {

// It's an operator, pop two elements from the stack

double b = stack.pop();

double a = stack.pop();

double result = applyOperator(a, b, token);

stack.push(result);

}

}

// The final result will be the only element left in the stack

return stack.pop();

}

private static boolean isNumeric(String token) {

try {

Double.parseDouble(token);

return true;

} catch (NumberFormatException e) {

return false;

}

}

private static double applyOperator(double a, double b, String operator) {

switch (operator) {

case "+":

return a + b;

case "-":

return a - b;

case "\*":

return a \* b;

case "/":

return a / b;

default:

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

}

}

public static void main(String[] args) {

// Example usage

String expression = "5 2 + 8 3 - \* 4 /";

double result = evaluatePostfix(expression);

System.out.println("Result: " + result); // Output: 8.75

}

}

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

import java.util.Arrays;

public class ArrayStack<E> implements Cloneable {

private static final int DEFAULT\_CAPACITY = 10;

private E[] data;

private int size = 0;

// Constructor

public ArrayStack() {

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

}

// Push an element onto the stack

public void push(E element) {

if (size == data.length) {

resize(2 \* data.length); // Resize if array is full

}

data[size++] = element;

}

// Pop an element from the stack

public E pop() {

if (isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

E result = data[--size];

data[size] = null; // Help garbage collection

return result;

}

// Peek at the top element

public E peek() {

if (isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

return data[size - 1];

}

// Check if the stack is empty

public boolean isEmpty() {

return size == 0;

}

// Get the size of the stack

public int size() {

return size;

}

// Resize the internal array

private void resize(int newCapacity) {

data = Arrays.copyOf(data, newCapacity);

}

// Implement the clone method

@Override

public ArrayStack<E> clone() {

try {

// Perform a shallow copy

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

// Deep copy the internal array

cloned.data = Arrays.copyOf(this.data, this.data.length);

return cloned;

} catch (CloneNotSupportedException e) {

throw new AssertionError("Clone not supported", e);

}

}

// Main method for testing

public static void main(String[] args) {

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

stack.push(10);

stack.push(20);

stack.push(30);

System.out.println("Original stack size: " + stack.size());

System.out.println("Top of original stack: " + stack.peek());

// Clone the stack

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

System.out.println("Cloned stack size: " + clonedStack.size());

System.out.println("Top of cloned stack: " + clonedStack.peek());

// Modify original stack

stack.pop();

System.out.println("After popping original stack:");

System.out.println("Original stack size: " + stack.size());

System.out.println("Cloned stack size: " + clonedStack.size());

}

}

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 {

// Method to evaluate a postfix expression

public static double evaluatePostfix(String expression) {

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

// Split the input expression into tokens

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

// Process each token

for (String token : tokens) {

if (isNumeric(token)) { // If the token is a number

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

} else if (isOperator(token)) { // If the token is an operator

// Pop the top two elements from the stack

double b = stack.pop();

double a = stack.pop();

// Apply the operator and push the result back onto the stack

stack.push(applyOperator(token, a, b));

} else {

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

}

}

// The final result should be the only element left in the stack

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

throw new IllegalArgumentException("Invalid postfix expression.");

}

return stack.pop();

}

// Method to check if a string is numeric

private static boolean isNumeric(String str) {

try {

Double.parseDouble(str);

return true;

} catch (NumberFormatException e) {

return false;

}

}

// Method to check if a string is a valid operator

private static boolean isOperator(String token) {

return token.equals("+") || token.equals("-") || token.equals("\*") || token.equals("/") || token.equals("^");

}

// Method to apply an operator to two operands

private static double applyOperator(String operator, double a, double b) {

switch (operator) {

case "+":

return a + b;

case "-":

return a - b;

case "\*":

return a \* b;

case "/":

if (b == 0) {

throw new ArithmeticException("Division by zero");

}

return a / b;

case "^":

return Math.pow(a, b);

default:

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

}

}

// Main method for testing

public static void main(String[] args) {

java.util.Scanner scanner = new java.util.Scanner(System.in);

System.out.print("Enter a postfix expression: ");

String expression = scanner.nextLine();

try {

double result = evaluatePostfix(expression);

System.out.println("The result of the postfix expression '" + expression + "' is: " + result);

} catch (Exception e) {

System.out.println("Error: " + e.getMessage());

} finally {

scanner.close();

}

}

}