## Topics

1. Create Stack Interface

public interface Stack<E> {

boolean isEmpty();

int size();

E top();

void push(E element);

E pop();

}

1. Create Stack Using Array

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

private static final int DEFAULT\_CAPACITY = 10;

private E[] stackArray;

private int top;

public ArrayStack() {

this(DEFAULT\_CAPACITY);

}

public ArrayStack(int capacity) {

if (capacity <= 0) {

throw new IllegalArgumentException("Capacity must be positive");

}

stackArray = (E[]) new Object[capacity];

top = -1;

}

public boolean isEmpty() {

return top == -1;

}

public int size() {

return top + 1;

}

public E top() {

if (isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

return stackArray[top];

}

public void push(E element) {

if (size() == stackArray.length) {

resize(2 \* stackArray.length);

}

stackArray[++top] = element;

}

public E pop() {

if (isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

E element = stackArray[top];

stackArray[top--] = null;

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

resize(stackArray.length / 2);

}

return element;

}

private void resize(int capacity) {

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

for (int i = 0; i <= top; i++) {

newArray[i] = stackArray[i];

}

stackArray = newArray;

}

}

1. Create Stack Using Linked Lists

public class LinkedStack<E> implements Stack<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;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public E top() {

if (isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

return top.element;

}

public void push(E element) {

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

top = newNode;

size++;

}

public E pop() {

if (isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

E element = top.element;

top = top.next;

size--;

return element;

}

}

1. Implement Basic Methods of Stack

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

public class Main {

public static void main(String[] args) {

Stack<Integer> arrayStack = new ArrayStack<>();

arrayStack.push(1);

arrayStack.push(2);

arrayStack.push(3);

System.out.println(arrayStack.size());

System.out.println(arrayStack.top());

arrayStack.pop();

System.out.println(arrayStack.size());

System.out.println(arrayStack.top());

Stack<String> linkedStack = new LinkedStack<>();

linkedStack.push("Hello");

linkedStack.push("World");

System.out.println(linkedStack.size());

System.out.println(linkedStack.top());

linkedStack.pop();

System.out.println(linkedStack.size());

System.out.println(linkedStack.top());

}

}

## 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.

def transfer(S, T):

aux\_stack = []

while not S.is\_empty():

aux\_stack.append(S.pop())

while aux\_stack:

T.push(aux\_stack.pop())

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

def remove\_all(stack):

if not stack.is\_empty():

stack.pop( remove\_all(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.

def evaluate\_postfix(expression):

stack = []

operators = set(['+', '-', '\*', '/'])

for token in expression.split():

if token not in operators:

stack.append(token)

else:

operand2 = stack.pop()

operand1 = stack.pop()

result = perform\_operation(operand1, operand2, token)

stack.append(result)

return stack.pop()

def perform\_operation(operand1, operand2, operator):

operand1 = float(operand1)

operand2 = float(operand2)

if operator == '+':

return operand1 + operand2

elif operator == '-':

return operand1 - operand2

elif operator == '\*':

return operand1 \* operand2

elif operator == '/':

return operand1 / operand2

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

class ArrayStack:

def \_\_init\_\_(self):

self.\_data = []

def is\_empty(self):

return len(self.\_data) == 0

def push(self, item):

self.\_data.append(item)

def pop(self):

if self.is\_empty():

raise IndexError("Stack is empty")

return self.\_data.pop()

def top(self):

if self.is\_empty():

raise IndexError("Stack is empty")

return self.\_data[-1]

def size(self):

return len(self.\_data)

def clone(self):

new\_stack = ArrayStack()

new\_stack.\_data = self.\_data.copy()

return new\_stack

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

def evaluate\_postfix(expression):

stack = []

operators = set(['+', '-', '\*', '/'])

for token in expression.split():

if token not in operators:

stack.append(token)

else:

operand2 = stack.pop()

operand1 = stack.pop()

result = perform\_operation(operand1, operand2, token)

stack.append(result)

return float(stack.pop())

def perform\_operation(operand1, operand2, operator):

operand1 = float(operand1)

operand2 = float(operand2)

if operator == '+':

return operand1 + operand2

elif operator == '-':

return operand1 - operand2

elif operator == '\*':

return operand1 \* operand2

elif operator == '/':

return operand1 / operand2

expression = input("Enter an expression in postfix notation: ")

result = evaluate\_postfix(expression)

print("Result:", result)