## 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()

Sure! Here's the implementation of a stack interface, followed by implementations of a stack using an array and a stack using linked lists, along with the basic methods you mentioned.

1. Stack Interface:

java

Copy

public interface Stack<E> {

boolean isEmpty();

int size();

E top();

void push(E e);

E pop();

}

1. Stack Using Array:

java

Copy

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

private static final int DEFAULT\_CAPACITY = 10;

private E[] data;

private int top;

public ArrayStack() {

this(DEFAULT\_CAPACITY);

}

public ArrayStack(int capacity) {

data = (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 data[top];

}

public void push(E e) {

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

resize(2 \* data.length);

}

data[++top] = e;

}

public E pop() {

if (isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

E element = data[top];

data[top--] = null;

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

resize(data.length / 2);

}

return element;

}

private void resize(int capacity) {

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

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

newData[i] = data[i];

}

data = newData;

}

}

1. Stack Using Linked List:

java

Copy

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

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;

}

}

private Node<E> top;

private int size;

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 e) {

Node<E> newNode = new Node<>(e, 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;

}

}

The Stack interface defines the basic methods that a stack should have: isEmpty(), size(), top(), push(E e), and pop().

The ArrayStack class represents a stack implemented using an array. It keeps track of the top index of the stack and resizes the array if necessary. The resize method doubles the array size when it becomes full and halves it when it becomes one-quarter full.

The LinkedStack class represents a stack implemented using a linked list. It maintains a reference to the top node of the linked list.

Both implementations provide the required methods and handle edge cases such as empty stack checks and resizing the array in the ArrayStack implementation.

## 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.

To implement a method transfer(S, T) that transfers all elements from stack S onto stack T, preserving the order of elements, we can utilize the basic stack operations such as push() and pop().

Here's an example implementation in Java:

java

Copy

import java.util.Stack;

public class StackTransfer {

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

// Transfer elements from sourceStack to destinationStack

while (!sourceStack.isEmpty()) {

destinationStack.push(sourceStack.pop());

}

}

public static void main(String[] args) {

// Create two stacks

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

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

// Push elements onto stackS

stackS.push(1);

stackS.push(2);

stackS.push(3);

stackS.push(4);

stackS.push(5);

// Transfer elements from stackS to stackT

transfer(stackS, stackT);

// Print elements in stackT

while (!stackT.isEmpty()) {

System.out.println(stackT.pop());

}

}

}

In this example, the transfer method takes two parameters: sourceStack and destinationStack, both of type Stack<E>.

The implementation uses a while loop to transfer elements from the sourceStack to the destinationStack. In each iteration, the top element of the sourceStack is popped using pop() and then pushed onto the destinationStack using push(). This process continues until the sourceStack is empty.

In the main method, we create two stacks, stackS and stackT, and push some elements onto stackS. Then, we call the transfer method to transfer the elements from stackS to stackT. Finally, we print the elements in stackT, which will be in the desired order.

Note that the elements will be transferred in reverse order since popping from one stack and pushing onto another stack reverses their order.

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

To remove all the elements from a stack using a recursive method, we can use a recursive approach that repeatedly pops elements from the stack until it becomes empty.

Here's an example implementation in Java:

java

Copy

import java.util.Stack;

public class StackRemoveRecursive {

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

if (!stack.isEmpty()) {

stack.pop(); // Remove the top element

removeAll(stack); // Recursively remove the remaining elements

}

}

public static void main(String[] args) {

// Create a stack

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

// Push elements onto the stack

stack.push(1);

stack.push(2);

stack.push(3);

stack.push(4);

stack.push(5);

// Remove all elements from the stack recursively

removeAll(stack);

// Print the size of the stack (should be 0)

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

}

}

In this example, the removeAll method takes a single parameter stack of type Stack<E>. It uses a recursive approach to remove elements from the stack.

The implementation checks if the stack is not empty using the isEmpty() method. If the stack is not empty, it pops the top element from the stack using pop(). Then, it calls itself recursively to remove the remaining elements from the stack. This process continues until the stack becomes empty.

In the main method, we create a stack called stack and push some elements onto it. Then, we call the removeAll method to remove all the elements from the stack recursively. Finally, we print the size of the stack, which should be 0 indicating that all elements have been removed.

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.

To evaluate an expression in postfix notation, you can use a stack-based algorithm. The algorithm reads the postfix expression from left to right and performs operations based on the encountered operators and operands.

Here's a step-by-step description of a non-recursive way to evaluate an expression in postfix notation:

1. Create an empty stack to hold the operands.
2. Read the postfix expression from left to right.
3. For each element in the postfix expression:
   * If the element is an operand (number or variable), push it onto the stack.
   * If the element is an operator, pop the top two operands from the stack.
     + Perform the operation using the operator on the two operands.
     + Push the result back onto the stack.
4. After processing all elements in the postfix expression, the final result will be the only element left on the stack.
5. Pop the result from the stack and return it as the evaluated value of the postfix expression.

Here's an example implementation in Java:

java

Copy

import java.util.Stack;

public class PostfixEvaluator {

public static int evaluatePostfix(String postfixExpression) {

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

for (char c : postfixExpression.toCharArray()) {

if (Character.isDigit(c)) {

stack.push(Character.getNumericValue(c));

} else if (c == ' ') {

continue; // Skip whitespace

} else {

int operand2 = stack.pop();

int operand1 = stack.pop();

int result;

switch (c) {

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 operator: " + c);

}

stack.push(result);

}

}

return stack.pop();

}

public static void main(String[] args) {

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

int result = evaluatePostfix(postfixExpression);

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

}

}

In this example, the evaluatePostfix method takes a postfixExpression parameter, which is a string containing the postfix expression to be evaluated. The method utilizes a stack (Stack<Integer> stack) to hold the operands during the evaluation.

The implementation iterates over each character in the postfixExpression. If the character is a digit, it is converted to an integer and pushed onto the stack. If the character is an operator (+, -, \*, or /), the top two operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack.

After processing all the characters in the postfixExpression, the final result is the only element left on the stack. It is popped from the stack and returned as the evaluated value of the postfix expression.

In the main method, we provide an example postfix expression ("5 2 + 8 3 - \* 4 /") to the evaluatePostfix method and print the result. In this case, the result should be 6.

1. Implement the clone( ) method for the ArrayStack class.
2. To implement the clone() method for the ArrayStack class, you need to create a deep copy of the stack. This means that a new ArrayStack object should be created, and the elements of the original stack should be copied to the new stack.
3. Here's an example implementation in Java:
4. java
5. Copy
6. import java.util.Arrays;
7. public class ArrayStack<E> implements Cloneable {
8. private static final int DEFAULT\_CAPACITY = 10;
9. private E[] elements;
10. private int size;
11. public ArrayStack() {
12. elements = (E[]) new Object[DEFAULT\_CAPACITY];
13. }
14. // Other methods of the ArrayStack class
15. @Override
16. public ArrayStack<E> clone() {
17. try {
18. ArrayStack<E> clonedStack = (ArrayStack<E>) super.clone();
19. clonedStack.elements = Arrays.copyOf(elements, size);
20. return clonedStack;
21. } catch (CloneNotSupportedException e) {
22. throw new InternalError(e);
23. }
24. }
25. public static void main(String[] args) {
26. ArrayStack<Integer> stack = new ArrayStack<>();
27. stack.push(1);
28. stack.push(2);
29. stack.push(3);
30. ArrayStack<Integer> clonedStack = stack.clone();
31. // Print the elements of the cloned stack
32. while (!clonedStack.isEmpty()) {
33. System.out.println(clonedStack.pop());
34. }
35. }
36. }
37. In this example, the ArrayStack class is implemented as a generic class that represents a stack using an array. The class implements the Cloneable interface to enable cloning.
38. The clone() method is overridden to create a deep copy of the stack. Inside the clone() method, we call super.clone() to create a shallow copy of the ArrayStack object. Then, we use Arrays.copyOf() to create a new array (elements) and copy the elements from the original stack to the new array. The size field is used to determine the number of elements to be copied.
39. Note that the clone() method is declared to return an ArrayStack<E> object. This allows for chaining of method calls, such as stack.clone().pop(), to operate on the cloned stack.
40. In the main method, we create an instance of ArrayStack called stack and push some elements onto it. Then, we call clone() on the stack object to create a clone of the stack, stored in the clonedStack variable. Finally, we print the elements of the cloned stack by repeatedly using the pop() method until the stack becomes empty.
41. Implement a program that can input an expression in postfix notation (see Exercise C-6.19) and output its value
42. To implement a program that can input an expression in postfix notation and output its value, you can use the algorithm described in the previous response for evaluating postfix expressions. The program will read the input expression, evaluate it using the postfix evaluation algorithm, and then output the resulting value.
43. Here's an example implementation in Java:
44. java
45. Copy
46. import java.util.Scanner;
47. import java.util.Stack;
48. public class PostfixEvaluator {
49. public static int evaluatePostfix(String postfixExpression) {
50. Stack<Integer> stack = new Stack<>();
51. for (char c : postfixExpression.toCharArray()) {
52. if (Character.isDigit(c)) {
53. stack.push(Character.getNumericValue(c));
54. } else if (c == ' ') {
55. continue; // Skip whitespace
56. } else {
57. int operand2 = stack.pop();
58. int operand1 = stack.pop();
59. int result;
60. switch (c) {
61. case '+':
62. result = operand1 + operand2;
63. break;
64. case '-':
65. result = operand1 - operand2;
66. break;
67. case '\*':
68. result = operand1 \* operand2;
69. break;
70. case '/':
71. result = operand1 / operand2;
72. break;
73. default:
74. throw new IllegalArgumentException("Invalid operator: " + c);
75. }
76. stack.push(result);
77. }
78. }
79. return stack.pop();
80. }
81. public static void main(String[] args) {
82. Scanner scanner = new Scanner(System.in);
83. System.out.print("Enter the postfix expression: ");
84. String postfixExpression = scanner.nextLine();
85. int result = evaluatePostfix(postfixExpression);
86. System.out.println("Result: " + result);
87. }
88. }
89. In this example, the evaluatePostfix method is the same as in the previous response. It takes a postfixExpression parameter, evaluates it using the postfix evaluation algorithm, and returns the resulting value.
90. In the main method, we use a Scanner object to read the postfix expression from the user. The expression is stored in the postfixExpression variable. Then, we call the evaluatePostfix method with the postfixExpression and store the resulting value in the result variable. Finally, we output the result to the console.
91. To use this program, you can run it and enter a postfix expression when prompted. For example, you can enter "5 2 + 8 3 - \* 4 /". The program will evaluate the expression and output the result, which in this case should be 6.