CS2040 Tutorial 3 Suggested Solution

Week 5, starting 5 Sep 2022

Q1 Array- vs Reference-based Queue/Stack?

Does Java have an array-based queue, array-based stack, reference-based queue and reference-based stack? For each of them that exists, which methods does the Java API recommend to be used for stack/queue operations?

Answer

java.util.Queue<E> is an interface. A referenced-based implementation is (indirectly) java.util.LinkedList<E>
An array-based queue and stack implementation is java.util.ArrayDeque<E>

java.util.Stack<E> is the most direct implementation of an array-based stack, but java API suggests using stack methods in the java.util.Deque<E> interface, of which a referenced-based implementation is java.util.LinkedList<E> and an array-based implementation is ArrayDeque<E>

java.util.Deque<E> has both stack, queue and deque methods, so you have to decide what data structure you want before using the relevant methods

Stack methods: push(), pop(), peek()
Queue methods: offer()/add(), poll()/remove(), peek()/element()

Q2 Queue Application – Digging for Gold

There is a long rectangular stretch of land that can be divided into **N** x 1 side-by-side squares. Each square contains 0 or more gold bars and 0 or more rocks. You are allowed to dig up ONE contiguous piece of land in the stretch that has at most **k** (non-negative) rocks (or none if not possible). A square will either be untouched or completely dug up

```
int findMostGoldBars(int[] golds, int[] rocks, int k)
```

Design an algorithm to find the highest number of gold bars that can be dug up, and implement the method above. The algorithm should run in O(**N**) time

e.g. golds = {3, 5, 2, 4, 1}, rocks = {1, 3, 2, 0, 1}
when **k** = 3, the answer is 7
when **k** = 1, the answer is 5

Answer

The highest number of gold bars can be found in a window which has some starting (left) and ending (right) index. Instead of brute-forcing O($\mathbf{N}^2$) of such windows, find the maximum gold in windows that are as long as possible while meeting the constraint of not having more than **k** rocks

As the window keeps expanding and contracting from left to right but never moves back to the left, this is known as a sliding-window algorithm

```
int findMostGoldBars(int[] golds, int[] rocks, int k) {
   Queue<Integer> goldsWindow = new LinkedList<>(),
      rocksWindow = new LinkedList<>();
   int windowGold = 0, windowRocks = 0, maxGold = 0;
   for (int idx = 0; idx < golds.length; idx++) {
      goldsWindow.offer(golds[idx]); windowGold += golds[idx];
      rocksWindow.offer(rocks[idx]); windowRocks += rocks[idx];
      while (windowRocks > k) { // ensure window is valid
         windowGold -= goldsWindow.poll();
         windowRocks -= rocksWindow.poll();
      }
      maxGold = Math.max(maxGold, windowGold);
   }
   // fill ls
   return maxGold;
}
```

Q3 Stack Application – Expression Evaluation

In the Lisp programming language, each of the four basic arithmetic operators appears before an arbitrary number of operands, which are separated by spaces. The resulting expressions are enclosed in parentheses. There is only one operator in a pair of parentheses. The operators behave as follows:

- `( + a b c )` returns the sum of all the operands, and `( + )` returns 0.
- `( - a b c )` returns a - b - c - … and `( - a )` returns 0 - a.
  The minus operator must have at least one operand.
- `( * a b c )` returns the product of all the operands, and `( * )` returns 1.
- `( / a b c )` returns a / b / c / … and `( / a )` returns 1 / a, using **double division**
  The divide operator must have at least one operand.

You can form larger arithmetic expressions by combining these basic expressions using a fully parenthesized prefix notation. For example, the following is a valid Lisp expression:

```
( + ( - 6 ) ( * 2 3 4 ) )
```

The expression is evaluated successively as follows:

```
( + -6.0 ( * 2.0 3.0 4.0 ) )
( + -6.0 24.0 )
18.0
```

Design and implement an algorithm that uses up to 2 stacks to efficiently evaluate a legal Lisp expression composed of the four basic operators, integer operands, and parentheses. The expression is well formed (i.e. no syntax error), there will always be a space between 2 tokens, and we will not divide by zero.

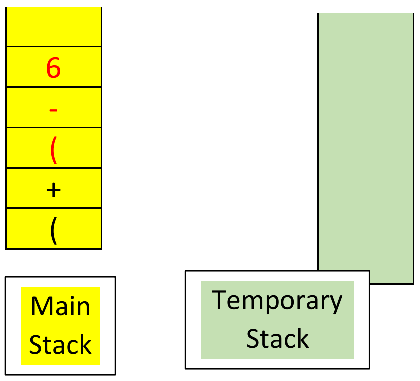
Output the result, which will be one double value

Answer

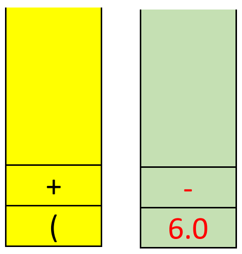
One algorithm uses two stacks. The first is used to store all tokens read from the expression one by one until the operator ")". The second stack is used to perform a simple operation on the operands in the innermost expression already in the first stack.

The tokens are pushed into the second stack in reverse order. Therefore, tokens from the second stack are popped in the order of input. The calculated result is then pushed back into the first stack.

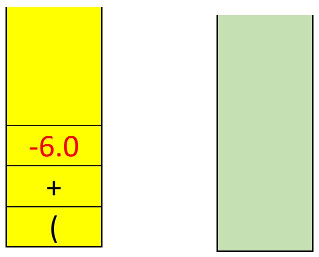
1. The main stack pushes the tokens one by one until it reads “)”



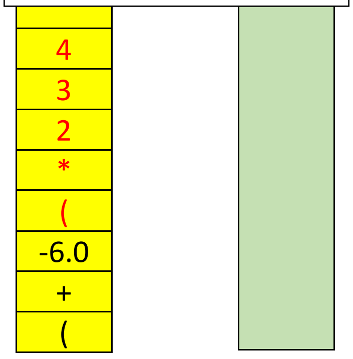
2. The main stack transfers its tokens to the temporary, stack for evaluation



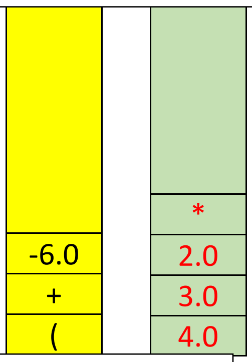
3. The temporary stack pushes back the result after performing subtraction



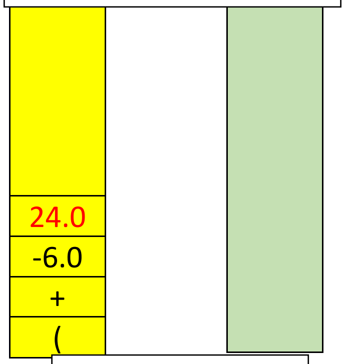
4. Main stack continues to push tokens until it reads “)”



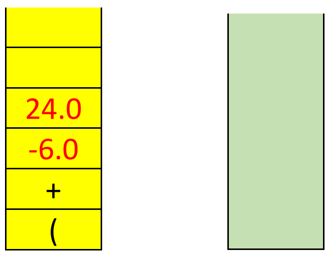
5. Main stack transfers tokens to temporary stack one by one



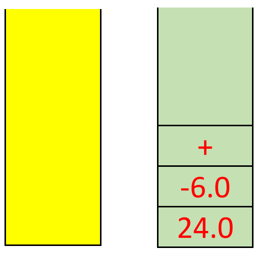
6. Temporary stack pushes back the result after calculation



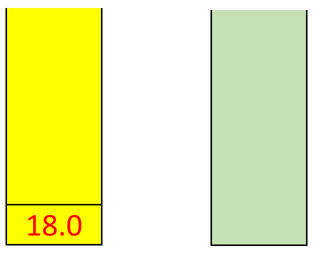
7. Main stack pushes until “)”



8. Main stack transfers to temporary stack



9. Temporary stack pushes back final result



```
static double performOp(Stack<Double> operands, char operator) {
   double result = 0.0;
   switch (operator){
      case '+': // 0.0 + opr1 + opr2 ...
         result = 0.0;
         while (!operands.empty()) result += operands.pop();
         return result;
      case '-':
         if (operands.size() == 1) return -operands.pop(); // -opr1
         result = operands.pop();
         while (!operands.empty()) result -= operands.pop();
         return result; // opr1 - opr2 - opr3 ...
      case '*':
         result = 1.0;
         while (!operands.empty()) result *= operands.pop();
         return result;
      case '/': // floating point division
         if (operands.size() == 1) return 1 / operands.pop(); // 1.0/opr1
         result = operands.pop();
         while (!operands.empty()) result /= operands.pop();
         return result; // opr1 / opr2 / opr3 ...
      default:
         throw new RuntimeException("Invalid operator");
   }
}

public static void main(String[] args) {
   Stack<String> allTokens = new Stack<String>(); // outer stack
   Kattio io = new Kattio(System.in);

   while (io.hasMoreTokens()) {
      String currentToken = io.getWord();
      if (currentToken.equals(")")) {
         Stack<Double> operands = new Stack<Double>(); // inner stack
         while ("+-*/".indexOf(allTokens.peek()) == -1) // while operand
            operands.push(Double.parseDouble(allTokens.pop()));
         char operator = allTokens.pop().charAt(0);
         allTokens.pop(); // remove "("
         allTokens.push("" + performOp(operands, operator));
      } else {
         allTokens.push(currentToken);
      }
   }
   io.println(allTokens.pop());
   io.close();
}
```

Question 4 (Online Discussion) – Queue, Stack or Deque?

You are given an array of **N** integers. For every **k** (k > 0) consecutive numbers in the array, report:

a) The leftmost non-zero number out of the **k** numbers

b) The rightmost non-zero number out of the **k** numbers

c) The maximum of the **k** numbers

Solve each part independent of the other parts, in O(**N**) time

e.g. if the numbers are {0, 1, 0, 0, 2, 0, 4, 0, 1, 0, 3, 1, 0, 2} and **k** = 4, then outputs are:

a) 1 1 2 2 2 4 4 1 1 3 3

b) 1 2 2 4 4 1 1 3 1 1 2

c) 1 2 2 4 4 4 4 3 3 3 3

How do you tell whether a Queue, Stack or Deque can help you in each part?