Advanced Algorithms and Data Structures

Lab 2: Stack & Queue ADT

This lab will give you practice about stack and queue ADT. You are to download the supporting files provided in the archive lab2.zip (see at the bottom).

Part 1: stack with findMin

In this part you have to first implement an array-based stack class and then using that class, you have to implement another kind of stack (a stack with one more operation called *findMin*).

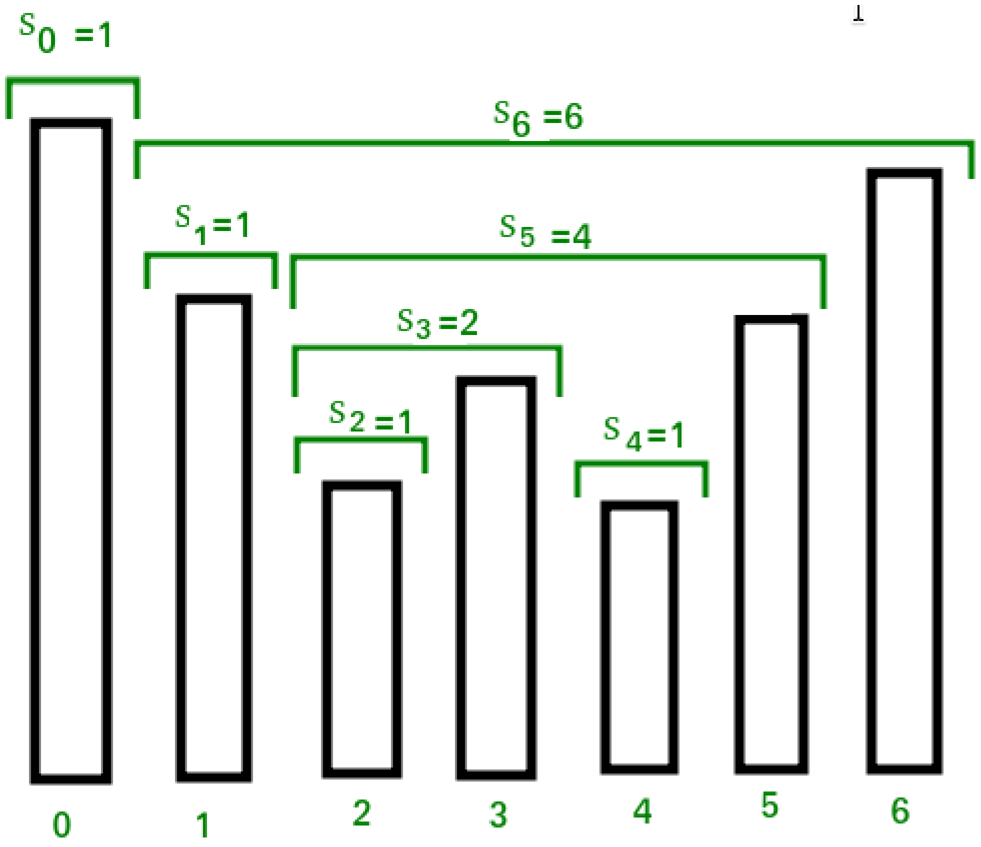
- Write the class ArrayStack which implements an array-based stack by using the provided class template ArrayStack.java. The methods must match the given complexity
- Using the previous class, write the class StackMin which implements a stack with the extra operation *findMin*. A stack of type StackMin can only handle *Comparable* object. Such a stack supports the same methods as a normal stack plus the method *findMin*: if s is a non empty StackMin, *s.findMin()* returns the minimum value currently in the stack. The complexity of *findMin* **must** be Θ(1) (just like all the other methods). You must use the provided class template StackMin.java
- In term of memory usage what is the worst case and when does it occur?

Supporting files:

- StackInterface.java
- ArrayStack.java
- StackMin.java
- EmptyStackException.java
- TestArrayStack.java
- TestStackMin.java

Part 2: stock span

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days. The span Si of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day. For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for the corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}:



Write the simple static method *computeSpan* from the class *StockSpan* which Implements a naive algorithm for computing the span of prices in a given array P of prices. The method should return an array S of integers such that S[i] is the span of the price P[i]. Give the running time complexity for both the best and the worst case.

Now, the input is no longer an array but an arbitrary long stream of prices for consecutive days, and the output is an arbitrary *PrintStream*. The input stream could be delivering real time data, so we cannot longer store the data inside a collection. To simulate a real time stream, we use a text file we can read with a *Scanner*. Using a **stack**, write the method *computeSpan* from the class *StockSpan* which Implements a smart algorithm for computing the span of prices in a given a (long) list of prices. This method should have a running time complexity of Θ(n) where n is the size of the input price list.

Supporting file:

- StockSpan.java
- stock.txt

Part 3: pairing

In this part you have first to implement a list-based queue class and then using that class, you have to solve a pairing problem.

- Write the class ListQueue which implements a list-based queue, by using the provided class template ListQueue.java. The methods must match the given complexity
- Using the previous class, complete the class Pairing to solve the following problem: given an entirely increasing sequence of integers stored in a file and a constant integer N, find and display all the pairs (x,y) from the sequence such that y = x + N. For example, if N = 3 and the file contains the numbers 1 3 5 6 9 10 11 12 14 16, then the matching pairs are (3,6), (6,9), (9,12) and (11,14). You must use the provided class template Pairing.java.
- What is the running time complexity of your algorithm?
- In term of memory usage what is the worst case and when does it occur?

Supporting files:

- QueueInterface.java
- ListQueue.java
- Pairing.java
- EmptyQueueException.java
- TestQueue.java
- big-file.txt

Part 4: queues with stacks

In this part you have to implement a queue class using stacks. You are to design the class StackQueue which has the same public methods as the previous ListQueue.

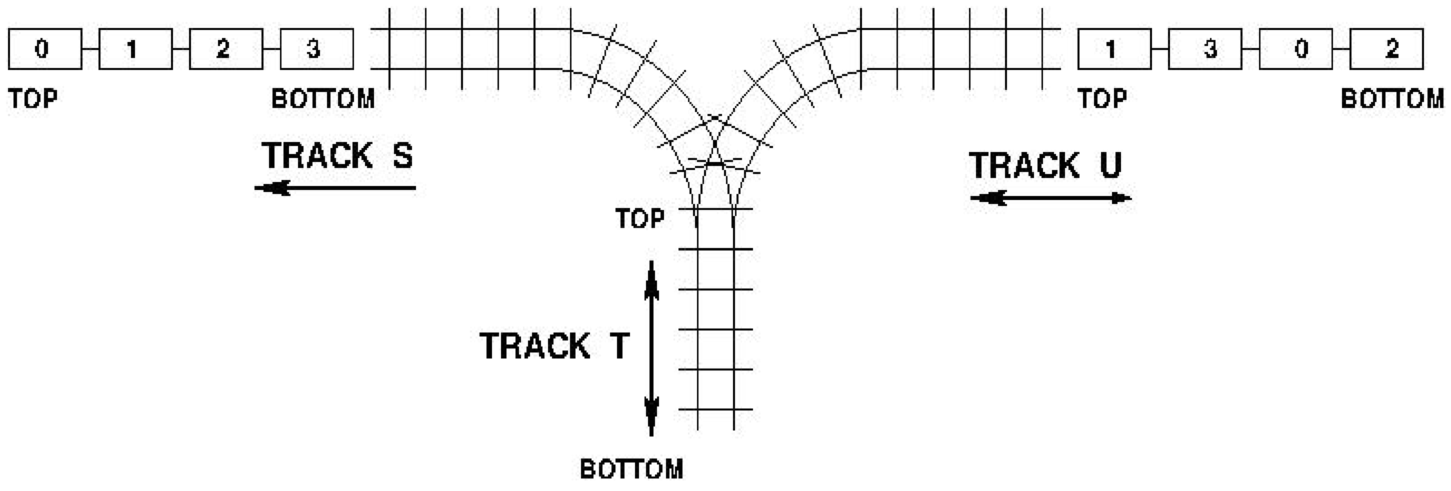
- write the class StackQueue which implements a queue based on stacks, by using the provided class template StackQueue.java . The methods must match the given complexity
- the methods *isEmpty*, *size* and *offer* must have a complexity of Θ(1)
- the methods *peek* and *poll* have now an **amortized** complexity of O(1)

Supporting files:

- StackQueue.java

Part 5: train composition

You are to implement a robot to arrange train cars in a railway station. The cars are initially on track U(nsorted). Using the track T(emporary), the robot must move all the cars from track U to track S(orted) by reordering them according to a given order. In the following example, the N cars making a train are numbered from 0 to N-1, but in your code, the N cars are labeled with strings.The robot can move only one car at a time from track U to track T, or from track T to track U or S. For example, in the example below, the cars are initially on track U in the order 1, 3, 0, 2 and they must be moved and reordered on track S in the order 0, 1, 2, 3:



To do so, the robot must output the following basic commands:

- move car 1 from track U to track T
- move car 3 from track U to track T
- move car 0 from track U to track T
- move car 0 from track T to track S
- move car 3 from track T to track U
- move car 1 from track T to track S
- move car 3 from track U to track T
- move car 2 from track U to track T
- move car 2 from track T to track S
- move car 3 from track T to track S

We can solve this problem using 3 stacks U, T and S. The stack U holds the cars on the track U such that the leftmost car is on top of the stack. The stack T holds the cars moving on track T. The stack S holds the cars in the final order (actually this stack is an *input* of the algorithm).

- Complete the class TrainManagement such that the method *arrange* can output the basic commands needed to move and rearrange the cars in the suitable ordering. You must use the provided class template TrainManagement.java.
- Ensure that both your algorithm and the sequence of basic commands produced by the *arrange* method are optimal (i.e. neither your algorithm or the robot are doing unnecessary move).
- Give the best and worst cases running time complexity for your algorithm and give example inputs when those cases occur

Supporting file:

- TrainManagement.java

Supporting files

lab2.zip	23 September 2024, 12:46 PM

Submission status

Submission status	No attempt
Grading status	Not graded
Due date	Monday, 23 September 2024, 4:30 PM
Time remaining	2 hours 8 mins
Last modified	-
Submission comments	Comments (0)

Add submission

You have not made a submission yet

◄ Lesson 6: ADT Implementation

Return to: Part 2: Abstrac...