CS1102: Data Structures and Algorithms

Tutorial 4 Stacks and Queues (Solutions)

Week of 22nd Feb 2010

1. The following two classes are available to you. The `Stack` class implements a stack ADT, while the `Queue` class implements a queue ADT.

```
class Stack<E> {
  // Data members are private and are not shown

  public boolean isEmpty() { ... }
  public int size() { ... }
  public void push{E item) { ... }
  public E pop() { ... }
  public E peek() { ... }
}

class Queue<E> {
  // Data members are private and are not shown

  public boolean isEmpty() { ... }
  public int size() { ... }
  public void enqueue(E item) { ... }
  public E dequeue() { ... }
  public E getFront() { ... }
}
```

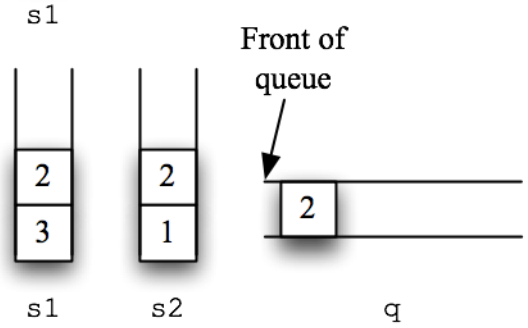
   Draw diagrams representing the contents of stack `s1`, stack `s2` and queue `q` at the end of the following program:

```
Queue<Integer> q = new Queue<Integer>();
Stack<Integer> s1 = new Stack<Integer>();
Stack<Integer> s2 = new Stack<Integer>();

s1.push(new Integer(3));
s1.push(new Integer(2));
s1.push(new Integer(1));

while(!s1.isEmpty()) {
  s2.push(s1.pop());
  if(!s1.isEmpty()) s2.push(s1.peek());
  q.enqueue(s2.peek());
}

s1.push(q.dequeue());
```

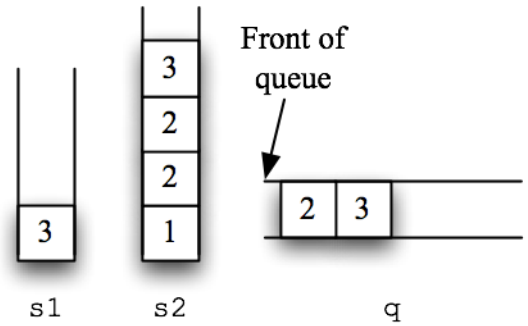
Answer:

Step 1:

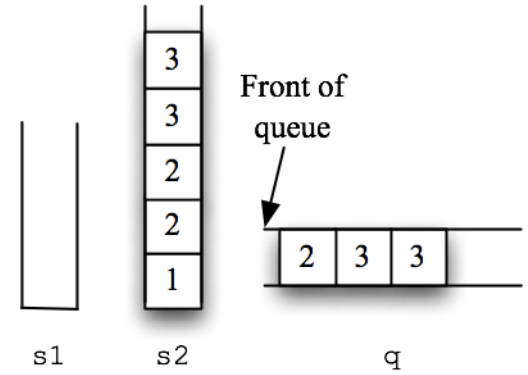
Step 2:



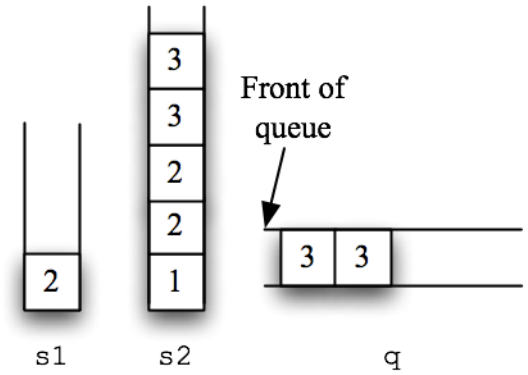
Step 3:



Step 4:



Step 5:



2. Given a 2D array of integers (`data`) with m rows and n columns, explain how you would check whether there is a path of adjacent 1's from the first element (`data[0][0]`) to the last element (`data[m-1][n-1]`). We consider two 1's to be adjacent if they are next to each other in the same column or in the same row. You should use stack(s) in your solution. If a path exists, you should also explain how to output the path.

   Hint: You will need to use one stack to keep track of the indices of the adjacent 1's (for backtracking) and another stack for storing the path that has been traced so far.

m = 5, n = 7

	0	1	2	3	4	5	6
0	1	1	1	0	1	1	1
1	1	0	1	1	1	0	1
2	0	0	0	0	0	1	1
3	0	1	1	1	0	1	0
4	1	1	0	0	0	1	1

path of 1's exists

	0	1	2	3	4	5	6
0	1	1	0	1	1	1	1
1	0	1	1	1	0	0	1
2	1	1	0	1	1	1	1
3	0	1	1	1	1	1	0
4	1	1	0	0	0	0	1

path of 1's does not exist

Answer:
The main problem is that the path we are tracing may branch out into two or three possible ways. For example, if we arrived at the current element from the left and there are adjacent 1's on the top, right, and bottom, then we need to check all three directions (top, right, and bottom) for a possible path to the last element. The second problem is the danger of getting stuck in a loop as we trace our path.

One solution is to use a stack to hold the indices of the elements that we have yet to visit. In the example above, if from the first element (`data[0][0]`) we choose to visit its bottom element (`data[1][0]`), we can push the indices of its right element (`data[0][1]`) onto the stack. Since the bottom element (`data[1][0]`) is a dead end, we just pop the stack to see which element to consider next. In this case, the indices for `data[0][1]` would be considered. In other words, the stack allows us to backtrack to all the elements that we have yet to visit. If the stack is empty when we backtrack, then we can be sure that there is no path from the first to the last element. If we reach the last element, then there is indeed a path from the first to the last element.

To avoid tracing a loop, we need to keep track of which elements have been visited. This can be done by keeping a linear array of visited elements (their indices). Another way is to have a second 2D array of m rows and n columns, all initialized to `false`. Every time we visit an element, we change its corresponding element in the second 2D array to `true`. Although this uses more space, checking if an element has been visited can be done in a single step (constant time) as opposed to searching the entire linear array in the first method.

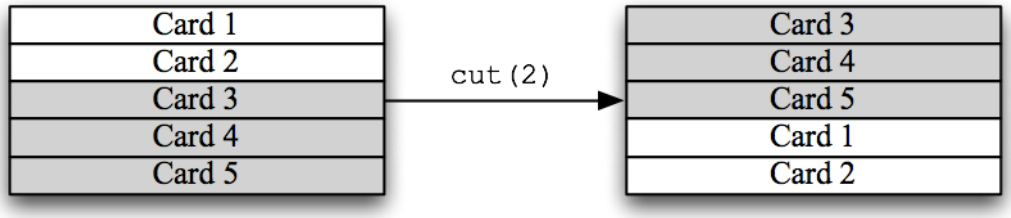
To output the path, we need to keep another stack. Every time we visit an element, we push its indices into the stack. Every time we backtrack, we pop the indices out of the stack. This way, when we reach the last element. The path from the first to the last element is given by the indices in the stack from the bottom to the top of the stack. To print these out, we just have to print the contents of the stack in reverse, which can be done by reversing the stack (pop and then push the contents into a temporary stack).

```
Psudocode:
pathfinder(Array[n][n] data):
   stack path
   stack next
   list visited
   path.push(data[0][0])
   for i in data[0][0].ajacent:
         next.push(i)
   do:
         if next.peek is adjacent to path.peek:
               path.push(next.pop)
               for i in path.peek.ajacent:
                     if i is not in visited:
                           next.push(i)
         else:
               visited.add(path.pop)
         if path is empty:
               print no solution
   until path.peek is data[n-1][n-1]
   path.reverse()
   do:
         print path.pop()
   while path is not empty
```

3. The following classes are available to you. The `Card` class represents a single playing card, while the `Deck` class represents a complete deck of 52 cards.

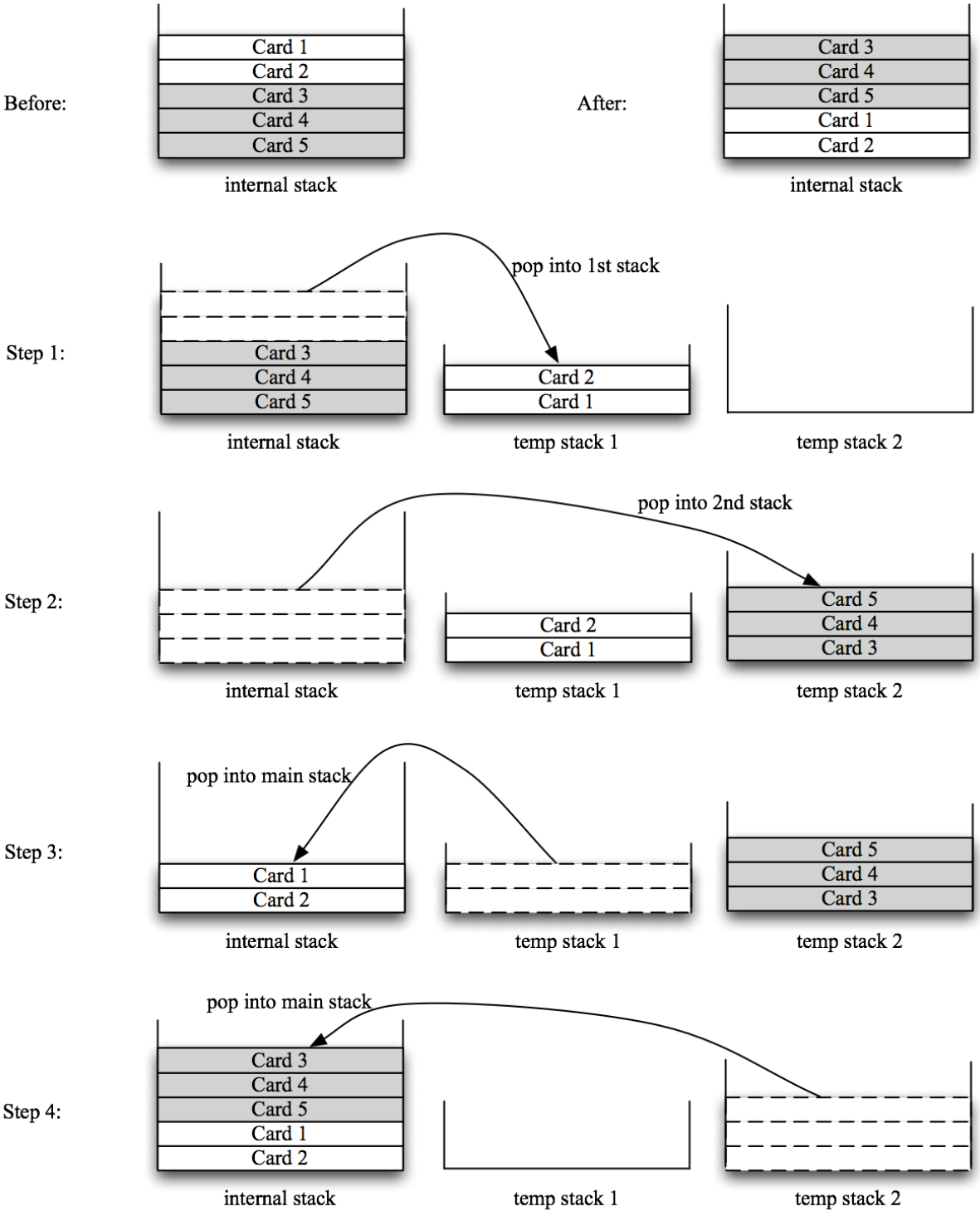
```
class Card {
  private String suit;
  private String denomination;

  public Card(String suit, String denomination) { ... }
  public String getSuit() { ... }
  public String getDenomination() { ... }
  public void setSuit(String suit) { ... }
  public void setDenomination(String denomination) { ... }
}

class Deck {
  public static final int NUM_OF_CARDS = 52;

  private Stack<Card> cards; // cards represent a deck of cards
                             // stacked face down on the table.
                             // In other words, cards.pop() will
                             // remove the top most card from the
                             // deck.

  public Deck() { ... } // The constructor initializes the stack
                        // with all 52 cards, but in some
                        // random order.
}
```

a) Write an instance method `public void cut(int numOfCutCards)` for the `Deck` class that simulates cutting the deck of cards. Cutting a deck of cards is done by taking the top `numOfCutCards` cards from the deck, placing them on the table, then placing the remaining bottom portion of the deck on top of the `numOfCutCards` cards. You may not use arrays in your method. You can assume that the deck will always have 52 cards and `numOfCutCards` will always be less than 52. The following diagram illustrates calling `cut(2)` on a stack of 5 cards:



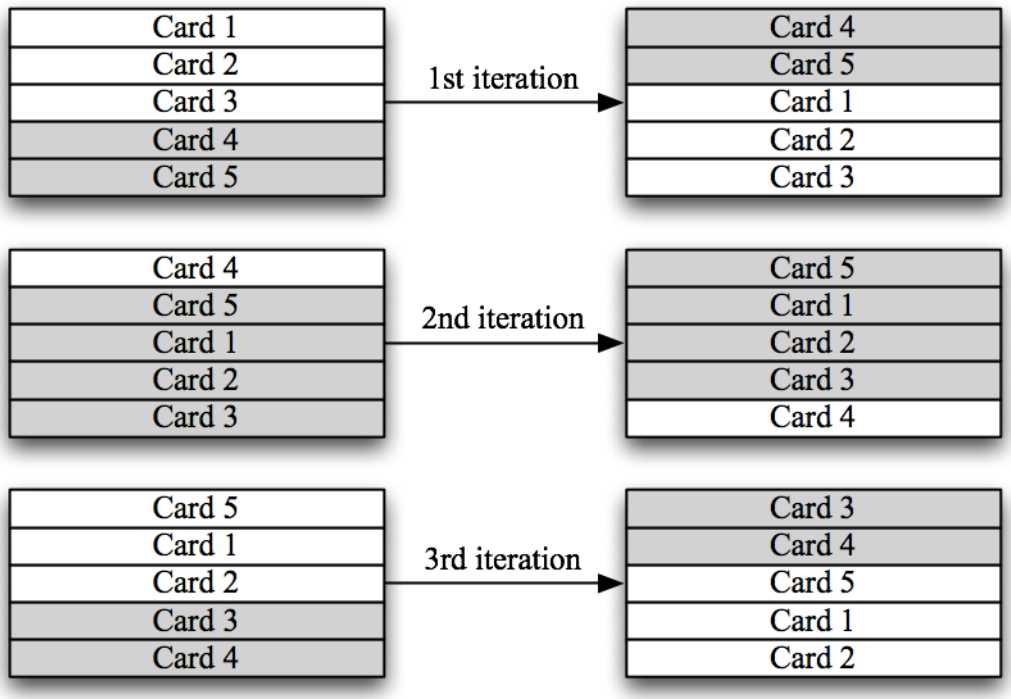
Answer:
One possible solution is to use two temporary stacks. Using the example provided in the question, we pop `numOfCutCards` cards (cards 1 and 2) and push them into the first temporary stack. Then the remaining cards (cards 3 to 5) are also popped and then pushed into the second temporary stack. To finish the cut, we simply put all the cards back but from the first temporary stack first. We pop the contents of the first temporary stack and push them back into our main stack. Then, we pop the contents of the second temporary stack and also push them back into our main stack.

Illustrating diagrams are on the following page.



```
public void cut(int numOfCutCards) {
  Stack<Card> stack1 = new Stack<Card>();
  Stack<Card> stack2 = new Stack<Card>();

  // Pop numOfCutCards cards and push them
  // into the first temporary stack
  while(cards.size() != NUM_OF_CARDS - numOfCutCards) {
    stack1.push(cards.pop());
  }

  // Pop the remaining cards and push them
  // into the second temporary stack
  while(!cards.isEmpty()) {
    stack2.push(cards.pop());
  }

  // Pop the cards from the first temporary stack and
  // push them back into our main stack
  while(!stack1.isEmpty()) {
    cards.push(stack1.pop());
  }

  // Pop the cards from the second temporary stack and
  // push them back into our main stack
  while(!stack2.isEmpty()) {
    cards.push(stack2.pop());
  }

  // Done!
}
```

b) Write another instance method `public void shuffle(int numOfIterations)` for the `Deck` class that simulates shuffling the deck of cards. You should assume that shuffling is done by moving a *random*-sized stack of cards from the bottom of the deck to the top and and repeating the process for a total of `numOfIterations` times. The following diagram illustrates calling `shuffle(3)` on a stack of 5 cards:



Answer:

The process of shuffling described here is actually very similar to the cutting process in (a). This is clear if we observe the diagrams provided in the question. So all we need to do is to call `cut()` `numOfIterations` times, each with a random number of `numOfCutCards` between 1 and 51 inclusive. This random number can be obtained by calling `Math.random()` which returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. We multiple this double value with 50 (`Deck.NUM_OF_CARDS - 2`) and add 1, making the range of our random number between 1 and 51 inclusive.

```
public void shuffle(int numOfIterations) {
  for(int i = 0; i < numOfIterations; ++i) {
    // Get a random int from 1 to 51 inclusive
    int rand = (int) Math.round(
      (Deck.NUM_OF_CARDS - 2) * Math.random() + 1);

    // Cut the cards
    cut(rand);
  }
}
```

c) Another way of shuffling is called the riffle or dovetail shuffle. In this shuffle, half of the deck is held in each hand with the thumbs inward, and then the thumbs release the cards so that they fall to the table interleaved (http://en.wikipedia.org/wiki/Shuffling#Riffle). Briefly explain how you would implement such a shuffle for the `Deck` class.

Answer:
Again, we can use two temporary stacks to simulate this shuffle. We pop half of our main stack to the first temporary stack and the remaining half to the other temporary stack. Then we start with the first temporary stack and randomly decide whether we should pop a card from that temporary stack into our main stack. The random decision can be done by calling `Math.random()` and assigning values greater than 0.5 as yes and otherwise no. We do the same for the second temporary stack by calling `Math.random()` again. This process is repeated for the first and second temporary stacks in turn until either of the temporary stacks is empty. At this point, we just pop the remaining cards and then push them into our main stack.
The code can also be written in java as follows.

```
public void riffle(){
   Stack<Card> tempStack1 = new Stack<Card>();
   Stack<Card> tempStack2 = new Stack<Card>();
   s=cards.size()

   while(cards.size()>s/2){
         stack1.push(cards.pop());
   }
   while(cards.size()>=0){
         stack2.push(cards.pop());
   }

   while(stack1.size()>=0 or stack2.size()>=0){
         if(Math.random()>0.5){
               stack.push(stack1.pop());
         }
         if(Math.random()>0.5){
               stack.push(stack2.pop());
         }
   }
   while (stack1.size()>0){
         stack.push(stack1.pop());
   }
   while (stack2.size()>0){
         stack.push(stack2.pop());
   }
}
```

Note: Question 3 using queue

Question 3 can also be solved using queue structure. It is necessary to change the class Deck.

```
class Deck {
  public static final int NUM_OF_CARDS = 52;

  private Queue<Card> cards; // cards represent a deck of cards
                             // stacked face down on the table.
                             // In other words, cards.pop() will
                             // remove the top most card from the
                             // deck.

  public Deck() { ... } // The constructor initializes the stack
                        // with all 52 cards, but in some
                        // random order.
}
```

For question (a), the cut process will be largely simplified using queue.
The algorithm of using queue structure for solving (a), (b) and (c) is presented as below:
(a)

```
public void cut(int numOfCutCards) {
    for(int i=0 ; i<numOfCutCards ; i++)
        cards.enqueue(cards.dequeue());
}
```

(b) The same as using stack for cards.

(c)

```
publics void riffle(){
   Queue<Card> queue1 = new Queue<Card>();
   Queue<Card> queue2 = new Queue<Card>();
   s=cards.size()

   while(cards.size()>s/2){
        queue1.enqueue(cards.dequeue());
   }
   while(cards.size()>=0){
        queue2. enqueue(cards.dequeue());
   }

   while(queue1.size()>=0 or queue2.size()>=0){
        if(Math.random()>0.5){
             cards.enqueue(queue1.dequeue());
        }
        if(Math.random()>0.5){
             cards.enqueue(queue2.dequeue());
        }
   }
   while (queue1.size()>0){
        cards.push(queue1.pop());
   }
   while (queue2.size()>0){
        cards.push(queue2.pop());
   }
}
```

4. The language Lisp, 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. The operators behave as follows:
    a) (+ a b c …) returns the sum of all the operands, and (+) returns 0.
    b) (- a b c …) returns a-b-c-… and (-a) returns –a. The minus operator must have at least one operand.
    c) (* a b c …) returns the product of all the operands, and (*) returns 1.
    d) (/ a b c …) returns a/b/c/… and (/a) returns 1/a. The divide operator must have at least one operand.

You can form larger arithmetic expressions by combining these basic expression 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 (*2 3 4))
(+ -6 24)
18

Design and implement an algorithm that uses a stack to evaluate a legal Lisp expression composed of the four basic operators and integer values.

Answer:

In a Lisp expression, a token can be a parenthesis, a number, or an operator. One of the solution can be designed using two stacks, one stack is used to store the tokens read from the expression one by one until the operator “)”. The other temporary stack is used to do simple Lisp calculation on the tokens pushed in from the first stack.

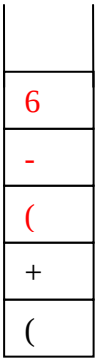
Tokens are pushed into the first stack one by one until it reads the first token “)” from the expression. When “)” is read, the first stack will pop its tokens one by one to the second stack until the peek value is “(”. The first stack pops out the “(“.

The temporary stack will do simple calculation on the values between the parentheses. The tokens are stored in reverse order in the temporary stack. Their order is reversed back when they are popped out from the temporary stack. The calculated result is pushed back to the first stack.

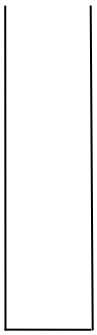
Below is an example of the algorithm with two stacks.

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

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

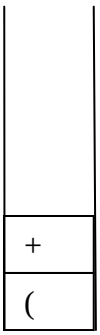


The main stack

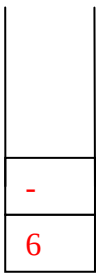


The temporary stack for simple calculation

2. The main stack pops out the tokens and push them one by one to the temporary stack

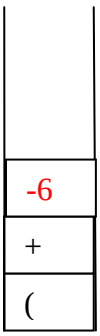


The main stack

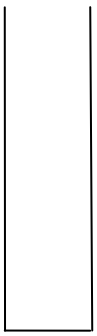


The temporary stack for simple calculation

3. The temporary stack pushes back the result after calculation



The main stack

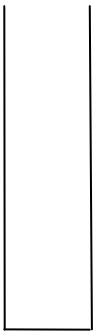


The temporary stack for simple calculation

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

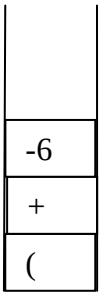
4. The main stack continues to push in the tokens from the expression until it reads ")"

The main stack

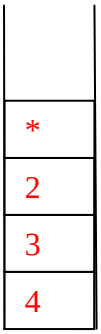


The temporary stack for simple calculation

5. The main stack pops out the tokens and push them one by one to the temporary stack

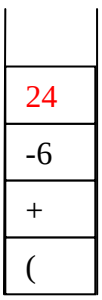


The main stack

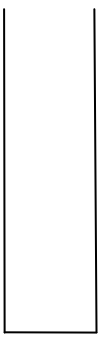


The temporary stack for simple calculation

6. The temporary stack pushes back the result after calculation.



The main stack

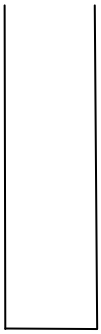


The temporary stack for simple calculation

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

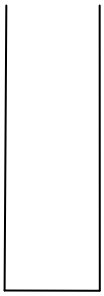
7. The main stack continues to push in the tokens from the expression until it reads ")"

The main stack

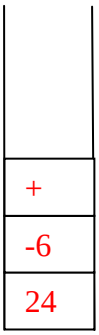


The temporary stack for simple calculation

8. The main stack pops out the tokens and push them one by one to the temporary stack

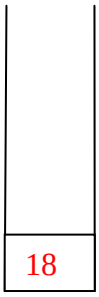


The main stack

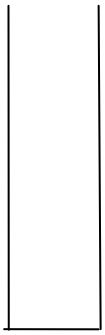


The temporary stack for simple calculation

9. The temporary stack pushes back the result after calculation. When it is at the end of expression, the final result is stored in the main stack.



The main stack



The temporary stack for simple calculation

Assume that the expression contains no error, so the program doesn't need to throw any exception. The pseudo-code answer can be written as follows:

```
Method SimpleExpressionCalculator(Stack t):
      operator = t.pop()
      switch operator:
            case “+”:
                  result=0
                  while t is not empty:
                        result=result+t.pop()
                  return result
            case “-”:
                  if t.size == 1:
                        return 1-t.pop()
                  else:
                        result=t.pop()
                        while t is not empty:
                        result=result-t.pop()
                  return result
            case “*”:
                  result=1
                  while t is not empty:
                        result=result*t.pop()
                  return result
            case “/”:
                  if t.size == 1:
                        return 1/t.pop()
                  else:
                        result=t.pop()
                        while t is not empty:
                              result=result/t.pop()
                  return result

Method LispCalculator (String expression):
      Stack s    //stack to store calculation results
      Stack t    //stack to store simple expression
      for i in expression.nextToken:
            if i equals “)”:
                  while s.peek() is not “(“:
                        t.push(s.pop)
                  s.pop()    //remove “(“
                  s.push(SimpleExpressionCalculator(t))
            else:
                  s.push(i)
      return s.peek()
```