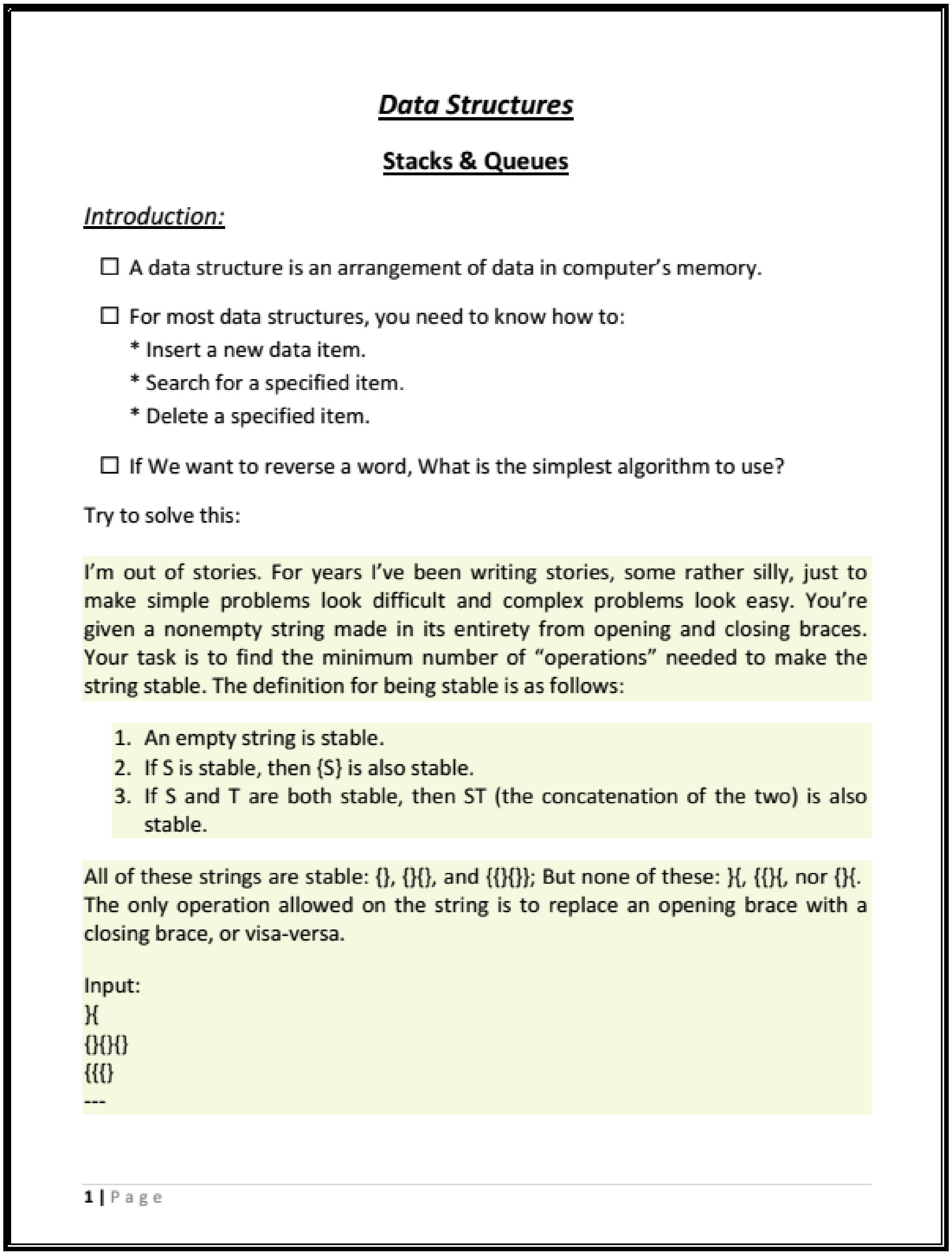


# ***Data Structures***

## **Stacks & Queues**

***Introduction:***

- ☐ A data structure is an arrangement of data in computer's memory.
- ☐ For most data structures, you need to know how to:
  * Insert a new data item.
  * Search for a specified item.
  * Delete a specified item.
- ☐ If We want to reverse a word, What is the simplest algorithm to use?

Try to solve this:

I'm out of stories. For years I've been writing stories, some rather silly, just to make simple problems look difficult and complex problems look easy. You're given a nonempty string made in its entirety from opening and closing braces. Your task is to find the minimum number of "operations" needed to make the string stable. The definition for being stable is as follows:

1. An empty string is stable.
2. If S is stable, then {S} is also stable.
3. If S and T are both stable, then ST (the concatenation of the two) is also stable.

All of these strings are stable: {}, {}{}, and {{}{}}; But none of these: }{, {{}{, nor {}{. The only operation allowed on the string is to replace an opening brace with a closing brace, or visa-versa.

Input:
}{
{}{}{}
{{{}
---

## ***Data Structures***

### **Stacks & Queues**

*Introduction:*

□ A data structure is an arrangement of data in computer’s memory.

□ For most data structures, you need to know how to: * Insert a new data item. * Search for a specified item. * Delete a specified item.

□ If We want to reverse a word, What is the simplest algorithm to use?

Try to solve this:

I’m out of stories. For years I’ve been writing stories, some rather silly, just to make simple problems look difficult and complex problems look easy. You’re given a nonempty string made in its entirety from opening and closing braces. Your task is to find the minimum number of “operations” needed to make the string stable. The definition for being stable is as follows:

1. An empty string is stable. 2. If S is stable, then {S} is also stable. 3. If S and T are both stable, then ST (the concatenation of the two) is also stable.

All of these strings are stable: {}, {}{}, and {{}{}}; But none of these: }{, {{}{, nor {}{. The only operation allowed on the string is to replace an opening brace with a closing brace, or visa-versa.

Input: }{ {}{}{} {{{} ---

```
Output:
1. 2
2. 0
3. 1
```

## *Stack*

- The Stack class represents a last-in-first-out (LIFO) stack of objects, because the last item inserted is the first one to be removed
- Placing a data on the top of the stack is called *pushing* it.
- Removing it from the top of the stack is called *popping* it.
- Stack is a class in Java, to use it you must make an object from this class.
- Stack <Type> st = new Stack<Type(optionally)>();
- In stack and all data structures type must be of wrapper class.
- Java.util.stack must be imported. "import java.util.stack();"

### *Methods of Stack:*

- **Push**
  public Object push(Object item)
  Pushes an item onto the top of this stack.
  Parameters: *item* - the item to be pushed onto this stack.
  Returns: the item argument.

- **Pop**
  public Object **pop**()
  Removes the object at the top of this stack and returns that object as the value of this function.
  Returns **:** The object at the top of this stack (the last item of the Vector object).

Output: 1. 2 2. 0 3. 1

## *Stack*

 The Stack class represents a last-in-first-out (LIFO) stack of objects, because

the last item inserted is the first one to be removed

 Placing a data on the top of the stack is called pushing it.

 Removing it from the top of the stack is called popping it.

 Stack is a class in Java, to use it you must make an object from this class.

 Stack <Type> st = new Stack<Type(optionally)>();

 In stack and all data structures type must be of wrapper class.

 Java.util.stack must be imported. “import java.util.stack();”

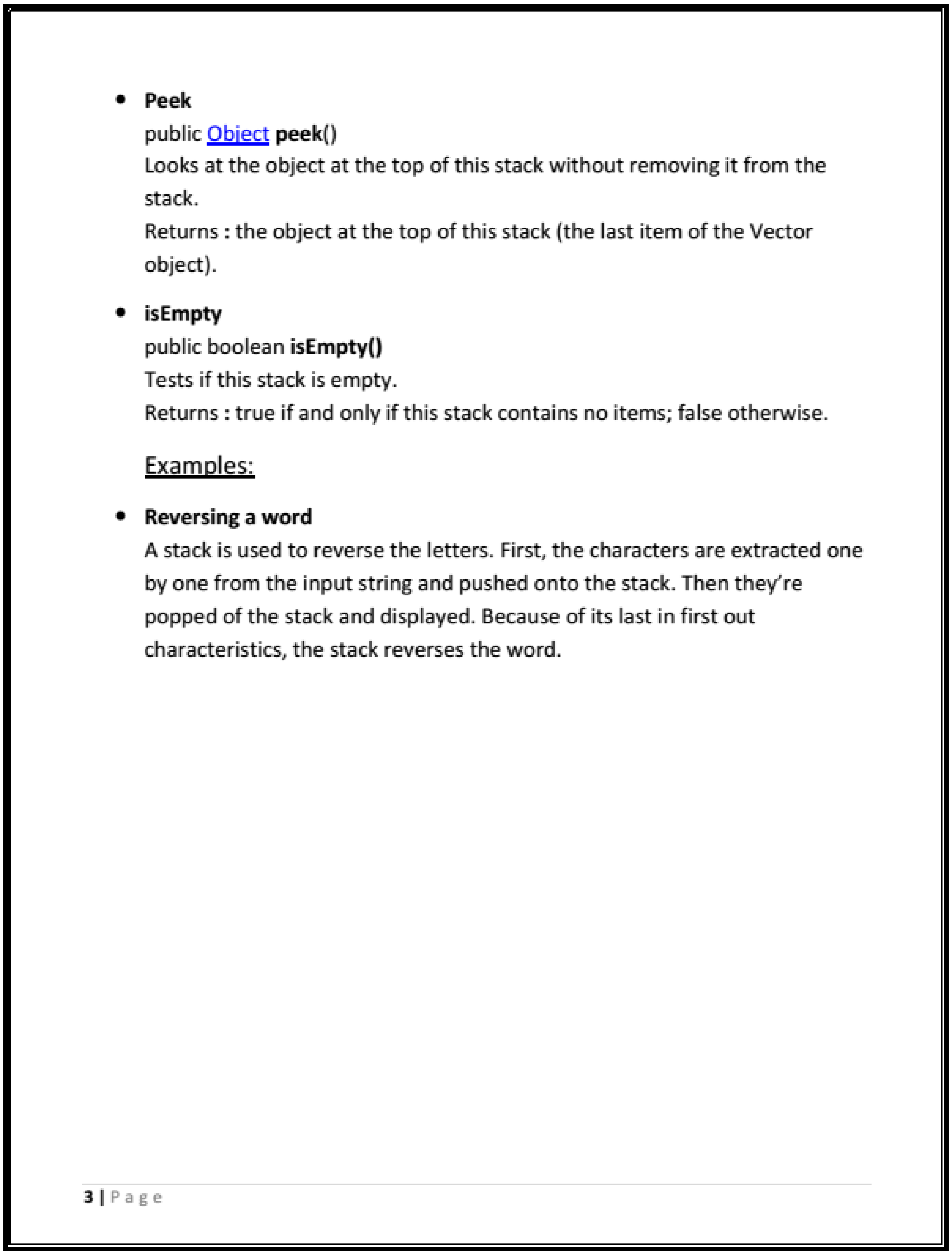
## *Methods of Stack:*

 **Push**

public Object push(Object item) Pushes an item onto the top of this stack. Parameters: item - the item to be pushed onto this stack. Returns: the item argument.

 **Pop**

public Object pop() Removes the object at the top of this stack and returns that object as the value of this function. Returns : The object at the top of this stack (the last item of the Vector object).



- **Peek**
  public Object **peek**()
  Looks at the object at the top of this stack without removing it from the stack.
  Returns **:** the object at the top of this stack (the last item of the Vector object).

- **isEmpty**
  public boolean **isEmpty()**
  Tests if this stack is empty.
  Returns **:** true if and only if this stack contains no items; false otherwise.

  Examples:

- **Reversing a word**
  A stack is used to reverse the letters. First, the characters are extracted one by one from the input string and pushed onto the stack. Then they're popped of the stack and displayed. Because of its last in first out characteristics, the stack reverses the word.

**3 |** Page

---

□ **Peek**

public Object peek() Looks at the object at the top of this stack without removing it from the stack. Returns : the object at the top of this stack (the last item of the Vector object).

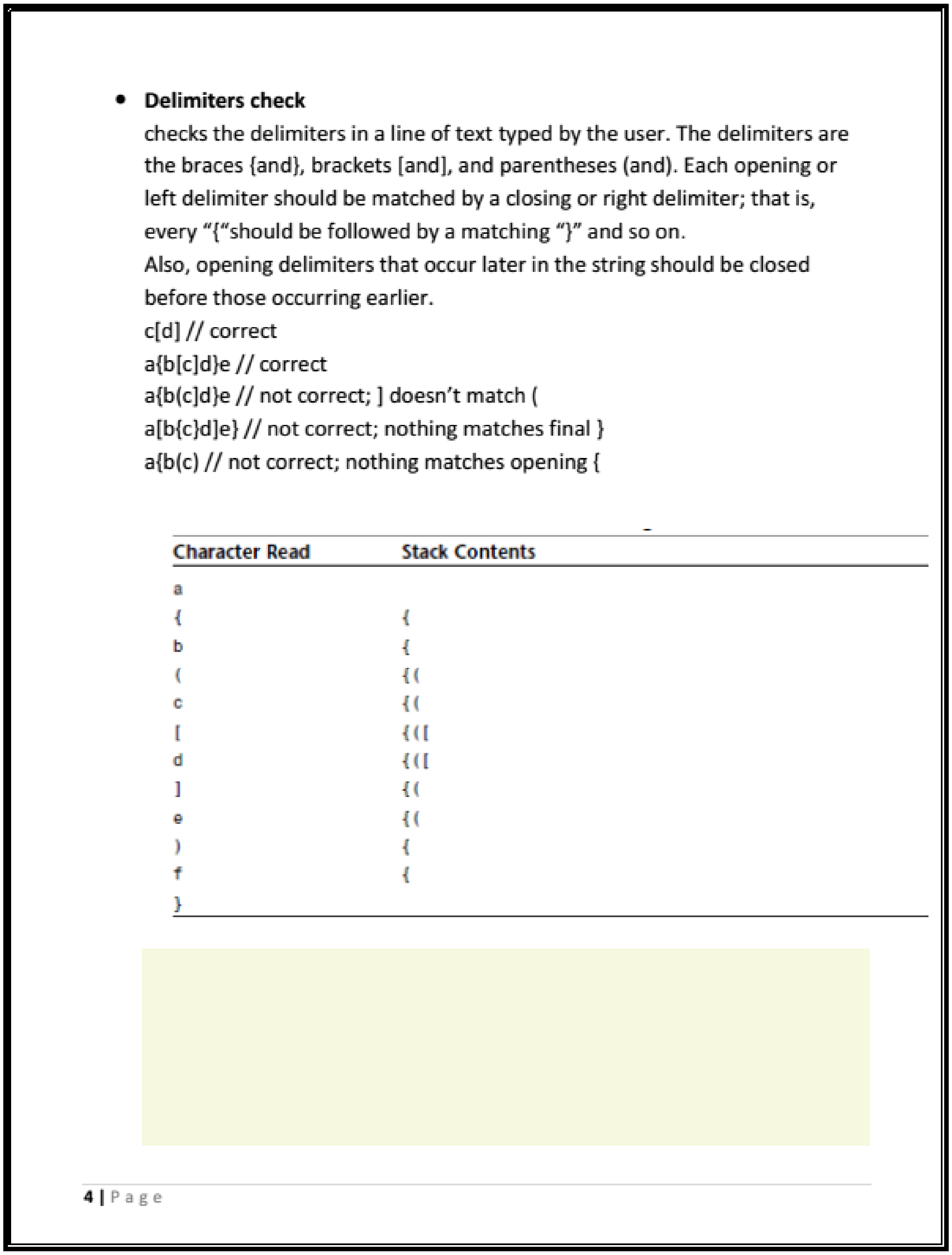
 **isEmpty**

public boolean isEmpty() Tests if this stack is empty. Returns : true if and only if this stack contains no items; false otherwise.

## Examples:

 **Reversing a word**

A stack is used to reverse the letters. First, the characters are extracted one by one from the input string and pushed onto the stack. Then they're popped of the stack and displayed. Because of its last in first out characteristics, the stack reverses the word.

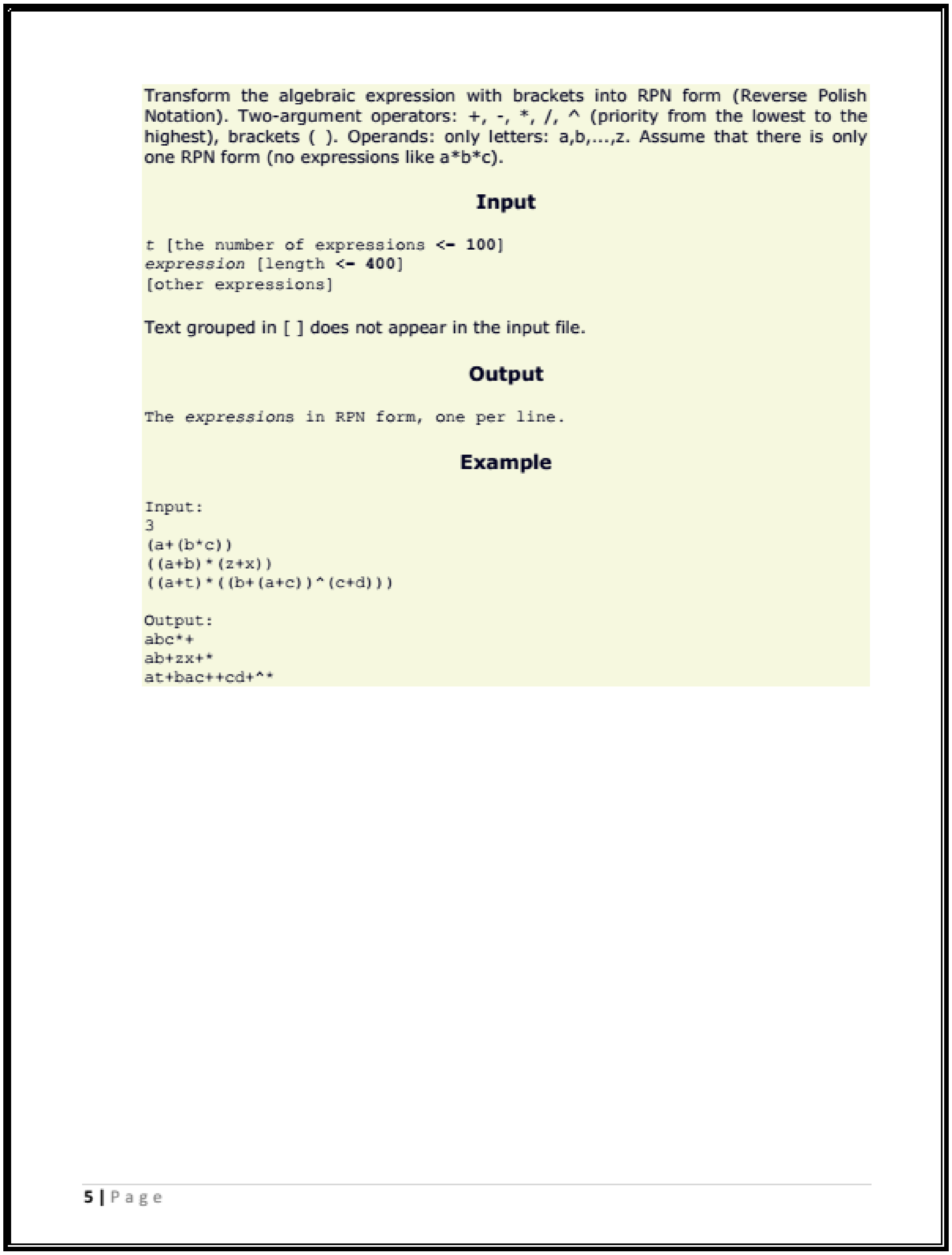


- **Delimiters check**
  checks the delimiters in a line of text typed by the user. The delimiters are the braces {and}, brackets [and], and parentheses (and). Each opening or left delimiter should be matched by a closing or right delimiter; that is, every "{"should be followed by a matching "}" and so on.
  Also, opening delimiters that occur later in the string should be closed before those occurring earlier.
  c[d] // correct
  a{b[c]d}e // correct
  a{b(c]d}e // not correct; ] doesn't match (
  a[b{c}d]e} // not correct; nothing matches final }
  a{b(c) // not correct; nothing matches opening {

| Character Read | Stack Contents |
|---|---|
| a | |
| { | { |
| b | { |
| ( | {( |
| c | {( |
| [ | {([ |
| d | {([ |
| ] | {( |
| e | {( |
| ) | { |
| f | { |
| } | |

□ **Delimiters check**

checks the delimiters in a line of text typed by the user. The delimiters are the braces {and}, brackets [and], and parentheses (and). Each opening or left delimiter should be matched by a closing or right delimiter; that is, every “{“should be followed by a matching “}” and so on. Also, opening delimiters that occur later in the string should be closed before those occurring earlier. c[d] // correct a{b[c]d}e // correct a{b(c]d}e // not correct; ] doesn’t match ( a[b{c}d]e} // not correct; nothing matches final } a{b(c) // not correct; nothing matches opening {



Transform the algebraic expression with brackets into RPN form (Reverse Polish Notation). Two-argument operators: +, -, *, /, ^ (priority from the lowest to the highest), brackets ( ). Operands: only letters: a,b,...,z. Assume that there is only one RPN form (no expressions like a*b*c).

## Input

```
t [the number of expressions <= 100]
expression [length <= 400]
[other expressions]
```

Text grouped in [ ] does not appear in the input file.

## Output

```
The expressions in RPN form, one per line.
```

## Example

```
Input:
3
(a+(b*c))
((a+b)*(z+x))
((a+t)*((b+(a+c))^(c+d)))

Output:
abc*+
ab+zx+*
at+bac++cd+^*
```

Transform the algebraic expression with brackets into RPN form (Reverse Polish Notation). Two-argument operators: +, -, *, /, ^ (priority from the lowest to the highest), brackets ( ). Operands: only letters: a,b,...,z. Assume that there is only one RPN form (no expressions like a*b*c).

## Input

t [the number of expressions <= 100] expression [length <= 400] [other expressions]

Text grouped in [ ] does not appear in the input file.

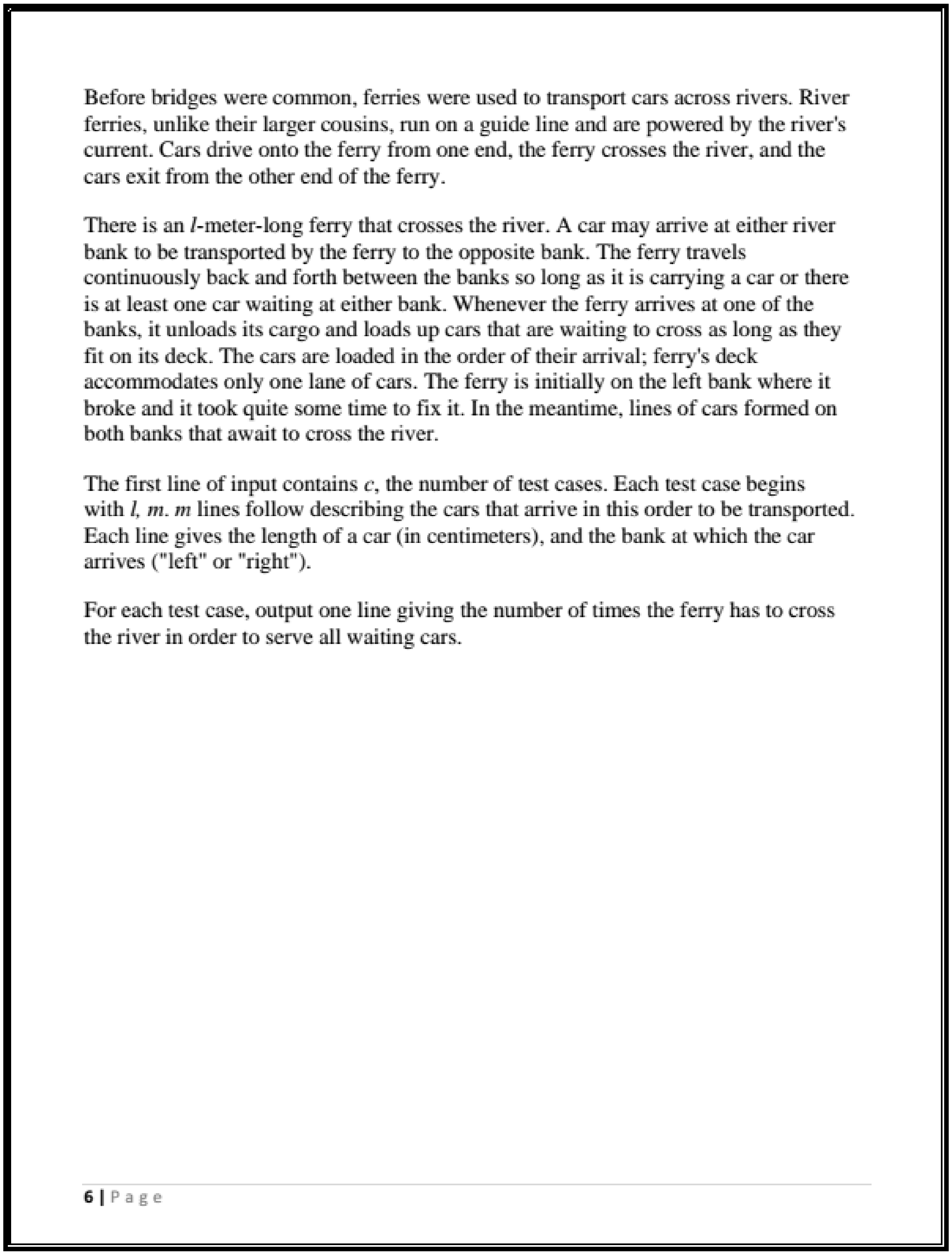
## Output

The expressions in RPN form, one per line.

## Example

Input: 3 (a+(b*c)) ((a+b)*(z+x)) ((a+t)*((b+(a+c))^(c+d)))

Output: abc*+ ab+zx+* at+bac++cd+^*



Before bridges were common, ferries were used to transport cars across rivers. River ferries, unlike their larger cousins, run on a guide line and are powered by the river's current. Cars drive onto the ferry from one end, the ferry crosses the river, and the cars exit from the other end of the ferry.

There is an *l*-meter-long ferry that crosses the river. A car may arrive at either river bank to be transported by the ferry to the opposite bank. The ferry travels continuously back and forth between the banks so long as it is carrying a car or there is at least one car waiting at either bank. Whenever the ferry arrives at one of the banks, it unloads its cargo and loads up cars that are waiting to cross as long as they fit on its deck. The cars are loaded in the order of their arrival; ferry's deck accommodates only one lane of cars. The ferry is initially on the left bank where it broke and it took quite some time to fix it. In the meantime, lines of cars formed on both banks that await to cross the river.

The first line of input contains *c*, the number of test cases. Each test case begins with *l, m*. *m* lines follow describing the cars that arrive in this order to be transported. Each line gives the length of a car (in centimeters), and the bank at which the car arrives ("left" or "right").

For each test case, output one line giving the number of times the ferry has to cross the river in order to serve all waiting cars.

Before bridges were common, ferries were used to transport cars across

rivers. River ferries, unlike their larger cousins, run on a guide line and are powered by the river's current. Cars drive onto the ferry from one end, the ferry crosses the river, and the cars exit from the other end of the ferry.

There is an l-meter-long ferry that crosses the river. A car may arrive at either river bank to be transported by the ferry to the opposite bank. The ferry travels continuously back and forth between the banks so long as it is carrying a car or there is at least one car waiting at either bank. Whenever the ferry arrives at one of the banks, it unloads its cargo and loads up cars that are waiting to cross as long as they fit on its deck. The cars are loaded in the order of their arrival; ferry's deck accommodates only one lane of cars. The ferry is initially on the left bank where it broke and it took quite some time to fix it. In the meantime, lines of cars formed on both banks that await to cross the river.

The first line of input contains c, the number of test cases. Each test case begins with l, m. m lines follow describing the cars that arrive in this order to be transported. Each line gives the length of a car (in centimeters), and the bank at which the car arrives ("left" or "right").

For each test case, output one line giving the number of times the ferry has to cross the river in order to serve all waiting cars.

## Queues

- In terms of data structure, the queue is a stored as a continuous list of elements.
- The first element is referred to as the **Front**(or **head**)
- The last element is referred to as the **Rear**(or **tail**)
- The first person that enters the queue gets served first.
- The Queue data structure is similar to the Stack data structure, except it's (**FIFO**) First in first out.
- Queues are very useful in a computer:
  1.Printer queue
  2.Keyboard buffer
  3.Network buffer

### Methods:

- **Add**
  public boolean **add**(E e)
  Inserts the specified element into this queue if it is possible, returning true upon success and throwing an IllegalStateException if no space is currently available.
  **Parameters:** e - the element to add
  **Returns:** true if the item can be inserted
- **Poll**
  E poll()
  Retrieves and removes the head of this queue, or null if this queue is empty.
  **Returns :** the head of this queue, or null if this queue is empty.
- **Peek**
  E peek()
  Retrieves, but does not remove, the head of this queue, returning

## Queues

 In terms of data structure, the queue is a stored as a continuous list of elements.

 The first element is referred to as the Front(or head)

 The last element is referred to as the Rear(or tail)

 The first person that enters the queue gets served first.

 The Queue data structure is similar to the Stack data structure, except it's

(FIFO) First in first out.

 Queues are very useful in a computer:

1.Printer queue 2.Keyboard buffer 3.Network buffer

## Methods:

• Add public boolean add(E e) Inserts the specified element into this queue if it is possible, returning true upon success and throwing an IllegalStateException if no space is currently available. Parameters: e - the element to add Returns: true if the item can be inserted

• Poll E poll() Retrieves and removes the head of this queue, or null if this queue is empty. Returns : the head of this queue, or null if this queue is empty.

• Peek E peek() Retrieves, but does not remove, the head of this queue, returning

null if this queue is empty.
**Returns:** the head of this queue, or null if this queue is empty.

- **isEmpty**
  public boolean **isEmpty**()
  Returns true if this collection contains no elements.
  **Returns:** true if this collection contains no elements

## <u>Another Example:</u>

- A computer processor is given N tasks to perform. The i- th task requires $T_i$ seconds of processing time. The processor runs the tasks as follows: each task is run in order, from 1 to N, for 1 second, and then the processor repeats this again starting from task 1. Once a task has been completed, it will not be run in later iterations. Determine, for each task, the total running time elapsed once the task has been completed.

null if this queue is empty. Returns: the head of this queue, or null if this queue is empty.

• isEmpty public boolean isEmpty() Returns true if this collection contains no elements. Returns: true if this collection contains no elements

## Another Example:

- A computer processor is given N tasks to perform. The i- th task requires $T_i$ seconds of processing time. The processor runs the tasks as follows: each task is run in order, from 1 to N, for 1 second, and then the processor repeats this again starting from task 1. Once a task has been completed, it will not be run in later iterations. Determine, for each task, the total running time elapsed once the task has been completed.