1.1

Assume that the stack names is defined as in Figure 3.2(c) and perform the following sequence of operations. Indicate the result of each operation and show the new stack if it is changed.

names.push("Jane");

names.push("Joseph");

String top = names.pop();

String nextTop = names.peek();

|  |
| --- |
| Philip |
| Dustin |
| Robin |
| Debbie |
| Rich |

names.push("Jane");

|  |
| --- |
| Jane |
| Philip |
| Dustin |
| Robin |
| Debbie |
| Rich |

names.push("Joseph");

|  |
| --- |
| Joseph |
| Jane |
| Philip |
| Dustin |
| Robin |
| Debbie |
| Rich |

String top = names.pop();

|  |
| --- |
| Jane |
| Philip |
| Dustin |
| Robin |
| Debbie |
| Rich |

The String top contains "Joseph"

String nextTop = names.peek();

The String nextTop contains "Jane" and the stack remains unchanged.

1.3

What would be the effect of using peek instead of pop in Question 2?

The program would be in an infinite loop printing “Philip”.

2.1

The result returned by the palindrome finder depends on all characters in a string, including spaces and punctuation. Discuss how you would modify the palindrome finder so that only the letters in the input string were used to determine whether the input string was a palindrome. You should ignore any other characters.

The palindrome finder could be modified to only push letters onto the stack and to append only the letter characters to a new temporary string being formed. Then, after the reverse string is built by emptying the stack, it could be compared to the temporary string.

3.1

For the implementation of stack s using an ArrayList as the underlying data structure,

show how the underlying data structure changes after each statement below executes.

Assume the characters in "Happy" are already stored on the stack (H pushed on first).

s.push('i');

s.push('s');

char ch1 = s.pop(); s.pop();

s.push(' ');

char ch2 = s.peek();

The data field s.theData after each statement executes follows. Each element of this ArrayList is a reference to a Character object that wraps the letter shown.

s.theData

‘H’

‘a’

‘p’

‘p’

‘y’

After s.push(‘i’);

s.theData

‘H’

‘a’

‘p’

‘p’

‘y’

‘i’

After s.push(‘s’);

s.theData

‘H’

‘a’

‘p’

‘p’

‘y’

‘I’

‘s’

After char ch1 = s.pop(); s.pop();

ch1 stores ‘s’

s.theData

‘H’

‘a’

‘p’

‘p’

‘y’

After s.push(' ');

s.theData

‘H’

‘a’

‘p’

‘p’

‘y’

‘ ‘

After char ch2 = s.peek();

Ch2 stores ‘ ‘

s.theData is unchanged.

3.3

For the implementation of stack s using a linked list of nodes as the underlying data structure (see Figure 3.5), show how the underlying data structure changes after each statement in Question 1 executes. Assume the characters in "Happy" are stored on the stack (H pushed on first).

Note: Data type ArrayStack should be LinkedStack in the diagrams below.

Initial stack:

After executing s.push('i');

Question 3_3_3_b.eps

After executing s.push('s')

Question 3_3_3_c.eps

After executing char ch1 = s.pop(); Note that ch1=='s' The Node that references 's' is no longer referenced, so it will be garbage collected.

Question 3_3_3_d.eps

After executing s.pop(); The Node that references 's' is no longer referenced, so it will be garbage collected.

Question 3_3_3_e.eps

After executing s.push(' ');

Question 3_3_3_f.eps

After executing char ch2 = s.peek(); ch2 == ' ' and the stack is unchanged.

4.1

Trace the evaluation of the following expressions using class PostfixEvaluator. Show the operand stack each time it is modified.

13 2 \* 5 / 6 2 5 \* – +

5 4 \* 6 7 + 4 2 / - \*

| Expression | Action | Stack |
| --- | --- | --- |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Push 13 | |  | | --- | | 13 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Push 2 | |  | | --- | | 2  13 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Pop 2 and 13 Evavauate 13 \* 2 Push 26 | |  | | --- | | 26 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Push 5 | |  | | --- | | 5  26 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Pop 5 and 26 Evaluate 26/5 Push 5 | |  | | --- | | 5 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Push 6 | |  | | --- | | 6  5 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Push 2 | |  | | --- | | 2  6  5 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Push 5 | |  | | --- | | 5  2  6  5 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Pop 5 and 2 Evaluate 2 \* 5 Push 10 | |  | | --- | | 10  6  5 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Pop 10 and 6 Evaluate 6 - 10 Push -4 | |  | | --- | | -4  5 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Pop -4 and 5 Evaluate 5 + -4 Push 1 | |  | | --- | | 1 | |
| 13 2 \* 5 / 6 2 5 \* – +  ↑ | Pop 1 Stack is empty Result is 1 | |  | | --- | |  | |

| Expression | Action | Stack |
| --- | --- | --- |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Push 5 | |  | | --- | | 5 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Push 4 | |  | | --- | | 4  5 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Pop 4 and 5 Evavauate 5 \* 4 Push 20 | |  | | --- | | 20 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Push 6 | |  | | --- | | 6  20 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Push 7 | |  | | --- | | 7  5  20 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Pop 7 and 5 Evaluate 5 + 7 Push 12 | |  | | --- | | 12  20 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Push 4 | |  | | --- | | 4  12  20 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Push 2 | |  | | --- | | 2  4  12  20 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Pop 2 and 4 Evaluate 4 /2 Push 2 | |  | | --- | | 2  12  20 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Pop 2 and 12 Evaluate 12 - 2 Push 10 | |  | | --- | | 10  20 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Pop 10 and 20 Evaluate 20 \* 10 Push 200 | |  | | --- | | 200 | |
| 5 4 \* 6 7 + 4 2 / - \*  ↑ | Pop 200 Stack is empty Result is 200 | |  | | --- | |  | |

5.1

Draw the queue in Figure 4.6 as it will appear after the insertion of customer Harris and the removal of one customer from the queue. Which customer is removed? How many customers are left?

Original Queue (Fig 4.6)

|  |
| --- |
| Thome |
| Abreu |
| Jones |

After insertion of Harris

|  |
| --- |
| Thome |
| Abreu |
| Jones |
| Harris |

After removal of customer Thome.

|  |
| --- |
| Abreu |
| Jones |
| Harris |

6.1

Write an algorithm to display all the elements in a queue using just the queue operations. How would your algorithm change the queue?

declare an integer x

declare an integer y

declare an object temp

set x = queue.getSize()

set y = 0

loop while y < x

set temp = queue.remove()

display temp

queue.add(temp)

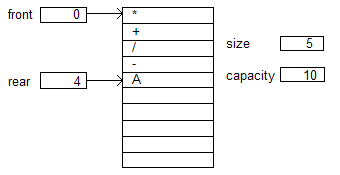
increment y by 1

next loop

This algorithm would rotate the entire queue once, but the queue would essentially be the same in the end, with the same size and order of objects.

7.1

Show the new array for the queue in Figure 4.10 after the array size is doubled.



7.3

Redraw the queue in Figure 4.10 so that rear references the list head and front references the list tail. Show the queue after an element is inserted and an element is removed. Explain why the approach used in the book is better.

QC6p3p3

While inserting a new item at the rear is easy, to remove an item from the front you need to walk along the linked list until you reach the next to last node, so that you have a reference to the new front, this is an O(n) operation. By having the front at the head of the linked list and the rear at the tail, both insertion and removal are constant time operations.

8.1

For object stackOfStrings declared above, replace each stack operation with the appropriate Deque method and explain the effect of each statement in the following fragment.

stackOfStrings.push("Hello");

String one = stackOfStrings.pop();

if (!stackOfStrings.isEmpty())

System.out.println(stackOfStrings.peek());

stackOfStrings.push("Good bye");

for (String two : stackOfStrings)

System.out.println(two);

Re-written using deque operations

stackOfStrings.addFirst("Hello"); // Makes "Hello" the first item on the stack

String one = stackOfStrings.removeFirst(); // Removes "Hello" and sets one to refer to it.

if (!stackOfStrings.isEmpty()) // Tests to see if the stack is empty

System.out.println(stackOfStrings.peekFirst()); // Does not execute since stack is empty

stackOfStrings.addFirst("Good bye"); // Makes "Good bye" the first item on the stack

for (String two : stackOfStrings) // Starts a loop through the stack contents

System.out.println(two); // Prints "Good bye"

8.3

Would the following statements execute without error? If your answer is “yes,” what would their effect be? If “no,” why not?

stackOfStrings.offer("away");

String three = stackOfStrings.remove();

These statement would execute without error. After offer("away") the stack would contain:

|  |
| --- |
| Good bye |
| away |

After the remove, the stack would conatin:

|  |
| --- |
| away |