Exercise R-6.1

Problem:
Suppose an initially empty stack S has performed a total of 25 push operations, 12 top operations, and 10 pop operations, 3 of which returned null to indicate an empty stack. What is the current size of S?

Solution:
- Push adds 25 elements. - Pop removes elements. Out of 10 pops, 3 failed (null), so only 7 successful pops occurred. - Thus net size $= 25 - 7 = 18$. - Top operations do not change the size.
Answer: $\boxed{18}$

—

Exercise R-6.7

Problem:
Suppose an initially empty queue Q has performed a total of 32 enqueue operations, 10 first operations, and 15 dequeue operations, 5 of which returned null to indicate an empty queue. What is the current size of Q?

Solution:
- Enqueue adds 32 elements. - Dequeue removes elements. Out of 15 dequeues, 5 failed (null), so only 10 successful dequeues occurred. - Thus net size $= 32 - 10 = 22$. - First operations do not change the size.
Answer: $\boxed{22}$

—

Exercise R-6.9

Problem:
What values are returned during the following sequence of queue operations, if executed on an initially empty queue?

enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue().

Solution:
We trace the queue step by step:

- enqueue(5) $\rightarrow$ [5]
- enqueue(3) $\rightarrow$ [5, 3]
- dequeue() $\rightarrow$ returns 5, queue = [3]
- enqueue(2) $\rightarrow$ [3, 2]
- enqueue(8) $\rightarrow$ [3, 2, 8]
- dequeue() $\rightarrow$ returns 3, queue = [2, 8]
- dequeue() $\rightarrow$ returns 2, queue = [8]
- enqueue(9) $\rightarrow$ [8, 9]
- enqueue(1) $\rightarrow$ [8, 9, 1]
- dequeue() $\rightarrow$ returns 8, queue = [9, 1]
- enqueue(7) $\rightarrow$ [9, 1, 7]
- enqueue(6) $\rightarrow$ [9, 1, 7, 6]
- dequeue() $\rightarrow$ returns 9, queue = [1, 7, 6]
- dequeue() $\rightarrow$ returns 1, queue = [7, 6]
- enqueue(4) $\rightarrow$ [7, 6, 4]

- dequeue() $\rightarrow$ returns 7, queue = [6, 4]
- dequeue() $\rightarrow$ returns 6, queue = [4]

Answer: The values returned are:

$$\{5, 3, 2, 8, 9, 1, 7, 6\}.$$

—

Exercise C-6.19

Problem:
Postfix notation is a way of writing arithmetic expressions without parentheses. For example, $((5+2)\times(8-3))/4$ is written in postfix as `5 2 + 8 3 - * 4 /`. Describe a nonrecursive way of evaluating an expression in postfix notation.

Solution:
We use a stack to store operands and evaluate operators as they appear.

- Scan the postfix expression left to right.
- If the token is a number, push it onto the stack.
- If the token is an operator, pop the top two elements, apply the operator, and push the result back.
- At the end, the stack contains the final result.

Example:

$$5\,2\,+\,8\,3\,-\,*\,4\,/ \quad \Rightarrow \quad 8$$

Pseudocode:

```
function evaluatePostfix(expr):
    create empty stack S

    for each token t in expr:
        if t is a number:
            S.push(t)
        else if t is an operator:
            b = S.pop()
            a = S.pop()
            result = applyOperator(a, b, t)
            S.push(result)

    return S.pop()
```

Listing 1: Pseudocode for C-6.19

Java Implementation:

```
import java.util.*;

public class PostfixEvaluator {
    public static int evaluate(String expr) {
        Stack<Integer> S = new Stack<>();
        String[] tokens = expr.split("␣");

        for (String token : tokens) {
            if (token.matches("\\d+")) { // if number
                S.push(Integer.parseInt(token));
            } else { // operator
                int b = S.pop();
                int a = S.pop();
                int result = 0;

                switch (token) {
                    case "+": result = a + b; break;
                    case "-": result = a - b; break;
                    case "*": result = a * b; break;
                    case "/": result = a / b; break;
                }
                S.push(result);
            }
        }
        return S.pop();
    }

    public static void main(String[] args) {
        String expr = "5 2 + 8 3 - * 4 /";
        System.out.println("Result: " + evaluate(expr));
            // Output: 8
    }
}
```

Listing 2: Java code for C-6.19

Complexity: Each token is processed once, so the algorithm runs in $O(n)$ time with $O(n)$ space for the stack.

—

Exercise C-6.23

Problem:
Show how to use a stack S and a queue Q to generate all possible subsets of an n-element set T nonrecursively.

Solution:
We can generate subsets iteratively using a queue to expand subsets level-by-level (similar to BFS). The stack S is used to store generated subsets.

- Start by enqueuing the empty set into Q.
- While Q is not empty:
  - Dequeue a subset *curr*, push it onto S.
  - For each element $e \in T$ that comes after the last element in *curr*, form a new subset $curr \cup \{e\}$ and enqueue it into Q.
- When done, S contains all 2^n subsets of T.

Java Implementation:

```java
import java.util.*;

public class SubsetGenerator {
    public static void generateSubsets(List<Integer> T) {
        Queue<List<Integer>> Q = new LinkedList<>();
        Stack<List<Integer>> S = new Stack<>();

        // Start with the empty subset
        Q.add(new ArrayList<>());

        while (!Q.isEmpty()) {
            // Dequeue current subset
            List<Integer> curr = Q.poll();

            // Push subset onto stack (stores generated subsets)
            S.push(curr);

            // Expand: add each possible next element
            int startIndex = curr.isEmpty() ? 0 : T.indexOf(curr.get(curr.size() - 1)) + 1;
            for (int i = startIndex; i < T.size(); i++) {
                List<Integer> newSubset = new ArrayList<>(curr);
                newSubset.add(T.get(i));
                Q.add(newSubset); // enqueue new subset
            }
        }

        // Print all subsets from the stack
        while (!S.isEmpty()) {
            System.out.println(S.pop());
        }
    }

    public static void main(String[] args) {
        List<Integer> T = Arrays.asList(1, 2, 3);
        generateSubsets(T);  // prints all subsets of
```

```
            {1,2,3}
    }
}
```

Listing 3: Java code for C-6.23 with comments

Complexity: There are 2^n subsets, each requiring $O(n)$ to construct, so the runtime is $O(n \cdot 2^n)$. Space usage is $O(2^n)$ to hold subsets.

—

Exercise C-6.24

Problem:
Suppose you have a stack S containing n elements and a queue Q that is initially empty. Describe how you can use Q to scan S to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements back to S in their original order. You may only use S, Q, and a constant number of other primitive variables.

Solution:
We scan by transferring all elements from S to Q, checking each element against x. This empties S and reverses the order once elements are returned. To restore the original order, repeat the transfer $S \to Q \to S$.

- Step 1: Initialize `found = false`.
- Step 2: Pop each element from S, check if it equals x, then enqueue it into Q.
- Step 3: Dequeue elements from Q and push them back into S (stack reversed).
- Step 4: Transfer everything $S \to Q \to S$ again to restore original order.
- Step 5: Return `found`.

Pseudocode:

```
function contains(S, Q, x):
    found     false

    # First pass: move from S to Q, check x
    while not S.isEmpty():
        e     S.pop()
        if e == x:
            found     true
        Q.enqueue(e)

    # Second pass: move from Q to S (stack reversed)
    while not Q.isEmpty():
        e     Q.dequeue()
        S.push(e)

    # Third + fourth pass: restore original order
    while not S.isEmpty():
        Q.enqueue(S.pop())
    while not Q.isEmpty():
        S.push(Q.dequeue())

    return found
```

Listing 4: Pseudocode for C-6.24

Complexity: Each element is moved a constant number of times, so the algorithm runs in $O(n)$ time with $O(1)$ extra space beyond S and Q.