1. Give a recursive method for removing all the elements from a stack.

**Algorithm.** The core idea is to remove one element of the stack and then recursively continue this process until the Stack is empty. This is shown in Algorithm 1.

**Algorithm 1 removeAll()**

<table>
<thead>
<tr>
<th>Input: Stack S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: if ¬S.isEmpty() then</td>
</tr>
<tr>
<td>2: S.pop()</td>
</tr>
<tr>
<td>3: removeAll(S)</td>
</tr>
</tbody>
</table>

**Time Complexity.**

**Theorem 1.** Algorithm 1 runs in \(O(n)\) time, where \(n\) is the number of elements in the input Stack.

*Proof.* Let the Stack be implemented using a linked-list. In this case, `isEmpty()` and `pop()` will run in \(O(1)\) time. In order to empty the entire stack, the algorithm must recurse \(n\) times until it is empty. Thus, in total the algorithm runs in \(O(n)\) time.

**Memory Complexity.**

**Theorem 2.** Algorithm 1 uses \(O(n)\) additional memory, where \(n\) is the number of elements in the input Stack.

*Proof.* As described in the proof of Theorem 1, Algorithm 1 will recurse \(O(n)\) times. It is known that every function call uses memory on a computer – in our case we need to store a reference to the Stack being manipulated. This is \(O(1)\) memory per function call, totaling \(O(n)\) additional memory.
2. Postfix notation is an unambiguous way of writing an arithmetic expression without parenthesis. It is defined so that if \( (\text{exp}_1) \text{op} (\text{exp}_2) \) is a normal fully parenthesized expression whose operation is \( \text{op} \), the postfix notation is \( \text{pexp}_1 \text{pexp}_2 \text{op} \), where \( \text{pexp}_1 \) is the postfix version of \( \text{exp}_1 \) and \( \text{pexp}_2 \) is the postfix version of \( \text{exp}_2 \). So, for example, the postfix version of \( ((5 + 2) \times (8 - 3))/4 \) is \( 5 \ 2 \ + \ 8 \ 3 \ - \ * \ / \).

Describe a nonrecursive way of evaluating an expression in postfix notation.

**Algorithm.** To evaluate a postfix expression, we first create an empty Stack \( S \). For each token of the postfix expression, we will manipulate \( S \). If the token is a number then we push onto the Stack, otherwise we need to evaluate the operator. Postfix notation will guarantee that the top two items on the Stack are numbers for evaluation. If this is not the case, we have detected an error – or could modify our algorithm to detect one. Once the expression has been completely evaluated, the single item on the top of the Stack is the answer to the expression. This process is shown in Algorithm 2.

**Algorithm 2** Evaluate Postfix Expression

**Input:** Postfix expression \( P \)

**Output:** Value of \( P \)

1: Stack \( S \leftarrow \emptyset \)
2: while \( P.\text{hasNext}() \) do
3: \( v \leftarrow P.\text{next}() \)
4: if \( v \) is a number then
5: \( S.\text{push}(v) \)
6: else \{ \( v \) is an operator \}
7: \( x_1 \leftarrow S.\text{pop}() \)
8: \( x_2 \leftarrow S.\text{pop}() \)
9: \( S.\text{push}(x_1 \ v \ x_2) \) \{Evaluate \( v \) with \( x_1 \) and \( x_2 \)\}
10: return \( S.\text{pop}() \)

**Time Complexity.**

**Theorem 3.** Algorithm \( 2 \) runs in \( O(n) \) time, where \( n \) is the number of tokens in the input expression.

**Proof.** Assume, we use a linked-list based implementation for the Stack. At each iteration of the loop in Algorithm \( 2 \), a constant amount of work is done because we only perform a set number of push() \( \), pop() \( \), and operator evaluations. The loop will run \( n \) iterations in order to evaluate the entire expression. This totals \( O(n) \) time.

**Memory Complexity.**

**Theorem 4.** Algorithm \( 2 \) uses \( O(1) \) additional memory.

**Proof.** Clearly the expression will use \( O(n) \) memory, where \( n \) is the number of input tokens. Assuming the expression moves the tokens to the Stack, and does not copy the elements, then the Stack will only take an additional \( O(1) \) memory for the algorithm. Additionally, a constant number of temporary variables are used in each iteration. In total, this is \( O(1) \) additional memory.

**Note:** \( O(n) \) additional memory is also an acceptable answer. It all depends if you copy portions of the expression into the stack or not. If you copy then you duplicate memory causing the additional overhead.
3. **Bonus.** 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.

**Algorithm.** The basic idea here is in the first phase pop all elements off of the Stack looking for $x$ while enqueueing them onto the Queue. In the next phase, return the elements to the stack (they are now in reverse order). Then, put all elements into the queue. Finally, store elements into the stack (back in correct order). The algorithm is shown in Algorithm 3.

**Algorithm 3** Stack Search

**Input:** Stack $S$, Queue $Q$, key $x$

**Output:** Boolean stating existence of $x$

1. $found \leftarrow false$
2. while $\neg S$.isEmpty() do {Search for element}
3. if $x = S$.top() then
4. $found \leftarrow true$
5. $Q$.enqueue($S$.pop())
6. while $\neg Q$.isEmpty() do {Restore elements}
7. $S$.push($Q$.dequeue())
8. while $\neg S$.isEmpty() do
9. $Q$.enqueue($S$.pop())
10. while $\neg Q$.isEmpty() do
11. $S$.push($Q$.dequeue())
12. return $found$

**Time Complexity.**

**Theorem 5.** Algorithm 3 runs in $O(n)$ time, where $n$ is the number of elements of the input Stack.

**Proof.** Assume the Stack and Queue are implemented well with linked-list structures such that all ADT operations take $O(1)$ time. So in the four phases, we clearly would transfer all elements from the Stack onto the Queue (or vice versa) taking $O(n)$ operations. Thus in total all phases take $4 \times O(n) = O(n)$. □

**Memory Complexity.**

**Theorem 6.** Algorithm 3 uses $O(1)$ additional memory.

**Proof.** The only complex operations involve transferring elements from the Stack to the Queue, one end of the Queue to the other end, or from the Queue to the Stack. In all of those portions of the algorithm the element is moved and not copied, thus taking $O(1)$ additional memory. Otherwise only a constant number of temporary variables are used taking $O(1)$ additional memory. □