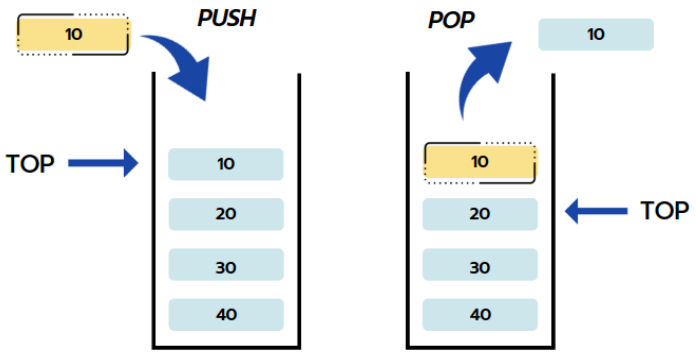
Chapter 3: Stack and Queue

3.1 Stack Introduction

There are some problems in which we want to store information, and we want to access the more recently stored information first. A data structure that allows us to access information in this manner is called a stack. It is so named because it resembles a stack of dishes. When you want a dish, you take it off the stack of dishes. When you clean a dish, you put it back on top of the stack of dishes. The same holds with a stack of data. When you store a new data item, you store it on top of the stack. When you want to retrieve a data item, you retrieve the top item from the stack. Note that this data item will be the one most recently stored on the stack.

Thus, a Stack is a restricted ordered sequence in which we can only add to and remove from one end — the **top** of the stack. Imagine stacking a set of books on top of each other — you can **push** a new book on **top** of the stack, and you can **pop** the book that is currently on the top of the stack. You are not, strictly speaking, allowed to add to the middle of the stack, nor are you allowed to remove a book from the middle of the stack. The only book that can be taken out of the stack is the **most recently added** one; a stack is thus a "last in, first out" (LIFO) data structure.

We use stacks everyday — from finding our way out of a maze, to evaluating postfix expressions, "undoing" the edits in a word-processor, and to implementing recursion in programming language runtime environments.



STACK DATA STRUCTURE

Three basic stack operations are:

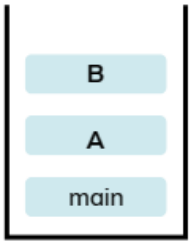
push(obj): adds obj to the top of the stack ("overflow" error if the stack has fixed capacity, and is full)
pop: removes and returns the item from the top of the stack ("underflow" error if the stack is empty)
peek: returns the item that is on the top of the stack, but does not remove it

("underflow" error if the stack is empty)

3.2 Stack Applications

A couple of examples where you want to access information in this manner in the real world are as follows:

3.2.1 Call Stacks: Whenever your C program calls a function, the C compiler creates a so-called *stack record* that contains important information about that function, such as storage for its parameters, its return value, and its local variables. The C compiler pushes this stack record onto something called a *call stack*. The call stack is organized so that the stack record of the most recently called function is on top of the stack, and the stack record of the least recently called function, which in the case of C is main, is at the bottom of the stack. For example, if the main calls A, which in turn calls B, then the call stack would be:



By convention stacks are shown growing upwards. While a function is executing, its stack record will be on top of the stack, and hence its information will be readily available. In the above case, B is currently executing and so we can readily access its parameters and local variables. If B calls a new function C, then a stack record for C is created and pushed onto the call stack. C now has access to its local variables and parameters. When C returns, we want B to resume executing, and thus we want access to B's stack record. By simply popping C's stack record off the call stack, we again have access to B's stack record. Hence a stack is the perfect data structure for implementing a call stack, because we always want access to the most recently called function, and when it returns, we want access to the function that called it.

3.2.2 Reverse: The simplest application of a stack is to reverse a word. You push a given word to stack – letter by letter – and then pop letters from the stack. Here's the trivial algorithm to print a word in reverse:

```
1. begin with an empty stack and an input stream.
2. while there is more characters to read, do:
3.         read the next input character;
4.         push it onto the stack;
5. end while;
6. while the stack is not empty, do:
7.         c = pop the stack;
8.         print c;
9. end while;
```

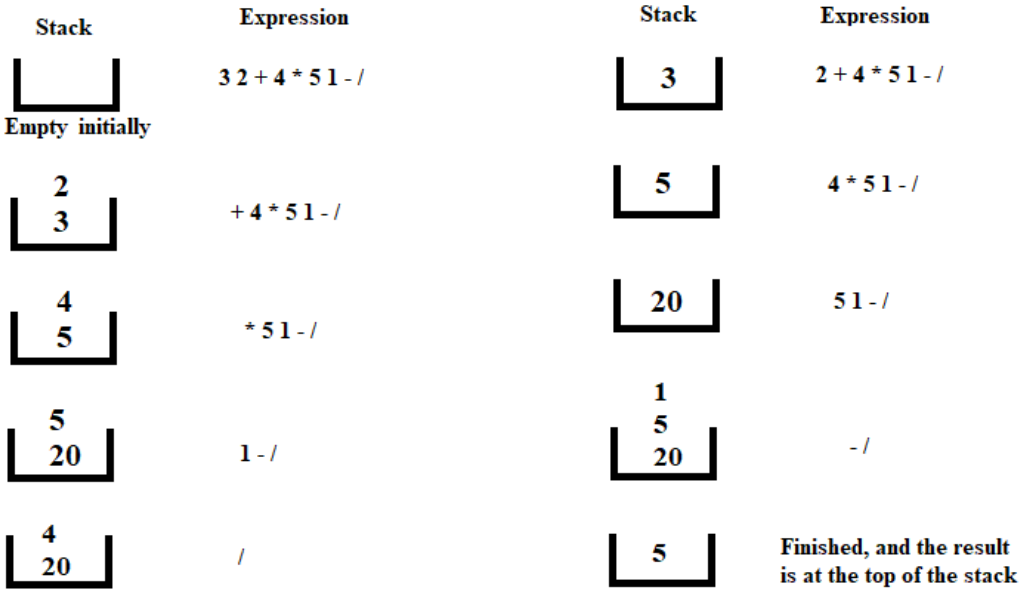
3.2.3 Undo: Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. Popping the stack is equivalent to "undoing" the last action. A very similar one is going back pages in a browser using the *back* button; this is accomplished by keeping the pages visited in a stack. Clicking on the *back* button is equivalent to going back to the most-recently visited page prior to the current

one.

3.2.4 Expression Evaluation: When an arithmetic expression is presented in the *postfix* form, you can use a stack to evaluate it to get the final value. For example: the expression 3 + 5 * 9 (which is in the usual *infix* form) can be written as 3 5 9 * + in the *postfix*. More interestingly, postfix form removes all parentheses and thus all implicit precedence rules; for example, the infix expression ((3 + 2) * 4) / (5 - 1) is written as the postfix 3 2 + 4 * 5 1 - /. You can now design a calculator for expressions in postfix form using a stack. The algorithm may be written as the following:

```
1.  begin with an empty stack and an input stream (for the expression).
2.  while there is more input to read, do:
3.          read the next input symbol;
4.          if it's an operand,
5.              then push it onto the stack;
6.          if it's an operator
7.              then pop two operands from the stack;
8.              perform the operation on the operands;
9.              push the result;
10. end while;
11. // the answer of the expression is waiting for you in the stack:
12. pop the answer;
```

Let's apply this algorithm to to evaluate the postfix expression 3 2 + 4 * 5 1 - / using a stack:



3.2.5 Parentheses Matching: In many editors, it is common to want to know which left parenthesis (or brace or bracket, etc) will be matched when we type a right parenthesis. Clearly we want the most recent left parenthesis. Therefore, the editor can keep track of parentheses by pushing them onto a stack, and then popping them off as each right parenthesis is encountered. We often have expressions involving "()[]{}" that require that the different types of parentheses are *balanced*. For example, the following are properly balanced: (a (b + c) + d)

[(a b) (c d)]

([a {x y} b])

But the following are not:

(a (b + c) + d

[(a b] (c d))

([a {x y) b])

The algorithm may be written as the following:

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
      read the next input character;
      if it's an opening parenthesis/brace/bracket ("(" or "{" or "[")
         then push it onto the stack;
      if it's a closing parenthesis/brace/bracket (")" or "}" or "]")
         then pop the opening symbol from stack;
         compare the closing with opening symbol;
         if it matches
            then continue with next input character;
         if it does not match
            then return false;
end while;
// all matched, so return true
return true;
```

3.2.6 Backtracking: This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

3.2.7 Language Processing:

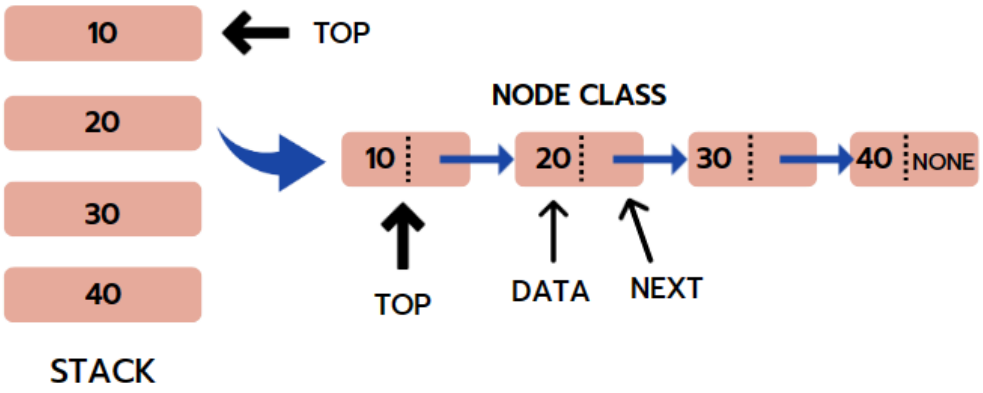
- space for parameters and local variables is created internally using a stack (*activation records*).
- The compiler's syntax check for matching braces is implemented by using stack.
- support for recursion

3.3 Stack Implementation

In the standard library of classes, the data type stack is an *adapter* class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list. Regardless of the type of the underlying data structure, a ***Stack*** must implement the same functionality. One requirement of a ***Stack*** implementation is that the ***push*** and ***pop*** operations run in *constant time*, that is, the time taken for stack operation is independent of how big or small the stack is.

3.3.1 Linked List-Based Implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



The following shows a partial head-referenced singly-linked based implementation of an *unbounded* stack. In a singly-linked list-based implementation we add the new item being pushed at the beginning of the array (why?), and consequently pop the item from the beginning of the list as well.

First, we create a class node. This is our Linked List node class which will have data in it and a node pointer to store the address of the next node element.

Then, we define the Stack class inside which we have the push(), pop() and peek() methods.

3.3.1.1 Push Operation

Adding a new node in the Stack is termed a push operation. Push operation on stack implementation using linked-list involves several steps:

- Create a node first and allocate memory to it.
- If the stack is empty, then the node is pushed as the first node of the linked list and defined as top. This operation assigns a value to the data part of the node and gives None to the address part of the node.
- If some nodes are already there in the linked list, then we have to add a new node at the beginning of the linked list so that we do not violate Stack's property. For this, assign the previous top to the next address field of the new node and the new node will be the starting node or the current top of the list.

The algorithm may be written as the following:

```
begin
   if stack is empty
      Create a node
      Assign the node to the top variable of the stack
   else
      Create a node and make the top of the stack its next node
      Assign the node to the top variable of the stack

end procedure
```

3.3.1.2 Pop Operation:

Deleting a node from the Stack is known as a pop operation. Pop operation involves the following steps:

- In Stack, the node is removed from the top of the linked list. Therefore, we must delete the value stored in the head/top pointer, and the node must get free. The following link node then becomes the head/top node.
- An underflow condition will occur when we try to pop a node when the Stack is empty.

The algorithm may be written as the following:

```
begin
   if stack is empty
      return underflow exception
   else
      Save the reference of the top of the stack node in a variable
      Make the top node's next node the top of the stack
      Make the previously top node's element and next null using the saved variable

end procedure
```

3.3.1.3 Peek Operation:

In the peek operation, we simply return the data stored in the head/top of the Linked List based Stack. An underflow condition will occur when we try to peek at a value when the Stack is empty.

The algorithm may be written as the following:

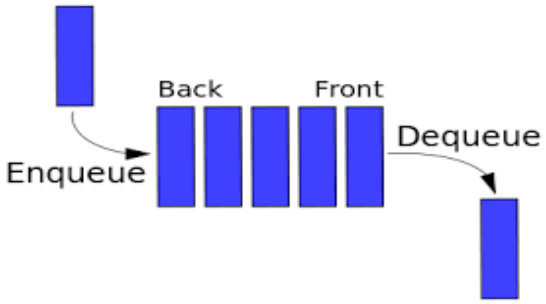
```
Pseudocode for peek:
begin
   if stack is empty
      return underflow exception
   else
      Return the reference of the top of the stack node
end procedure
```

3.4 Queue Introduction

We have started to learn secondary data structures made using primary data structures. The first secondary data structure we have learned is Stack. The idea of Stack was Last In First Out (LIFO) and we implemented that using a linked list. Stack helps us to solve multiple problems more efficiently. Similarly, we can create a new Data Structure that will follow different rules. One example of a new data structure is Queue

A queue is a new data structure that follows **First In First Out (FIFO)** while inserting and retrieving data. We are very familiar with the idea of queue in our daily lives. To illustrate, if you are waiting in line to buy a bus ticket, the counter will serve the customer who has come first and a new customer will wait behind the last person waiting in line. This idea is also important for our computers. One of the main uses of the queue in computers is resource management. For example, think of an office scenario where 10 people use the same printer. So if everyone requests to print simultaneously, the printer will print the first request and store the other requests in a queue. While executing any printing operation if a new print request comes it will store it in the same queue and execute the command in the First In First Out manner. In addition, when we execute 100 programs in our 8-core CPU, the processor also maintains a queue for command execution (Though the scenario is a bit more complex which you will learn in Operating Systems).

3.4.1 Queue Operations:



Queue

A queue has three main operations (like a stack). These operations are:

1. **Enqueue:** It stores a new object in the queue at the end. If the queue is full, this operation throws a QueueOverflow exception.
2. **Dequeue:** It removes the first object from the queue. If the queue is empty, this operation throws a QueueUnderflow exception.
3. **Peek**: This operation shows the first item in the queue. This operation does not remove any items from the queue. If the queue is empty, this operation throws a QueueUnderflow exception.

3.5 Queue Implementation

As we can see, to maintain the queue, we need two pointers, one for item retrieval and one for item insertion. As a result, we will maintain a front and a back for a queue. The idea is whenever we dequeue an item or peek at an item, we will use the front pointer and to enqueue an item we will use the back pointer. An example of a linked list-based queue is given below.

Algorithm for Queue using Linked List:

1. Create a class called `Node` with the following data members:
   - `elem`: to store the value of the node.
   - `next`: to point to the next node in the queue.

2. Initialize two variables: `front` and `rear`, both initially set to `NULL`.

3. Define the following operations:

   a. **Enqueue (Add an element to the queue):**
      - Create a new `Node` with data equal to the value to be enqueued.
      - If `rear` is `NULL`, set both `front` and `rear` to the new node.
      - Otherwise, set `rear->next` to the new node and update `rear` to the new node.

   b. **Dequeue (Remove an element from the queue):**
      - If `front` is `NULL`, return the underflow exception.
      - Store the current `front` in a temporary variable `temp`.
      - Move `front` to the next node (i.e., `front = front->next`).
      - If `front` becomes `NULL`, set `rear` to `NULL`.
      - Delete `temp` from memory.

   c. **Peek (Print all elements of the queue):**
      - If `front` is `NULL`, return the underflow exception.
      - Else return the front variable

4. End procedure.

Note that here, we are maintaining a front (for dequeue and peek) and a back (for enqueue). As the size of the linked list is not fixed, we are not maintaining the QueueOverflow exception in the given code.

Array Based Queue: If we want to create a queue with fixed size then an array based queue is a good approach. Here we will also use a front index and a back index. Initially both will start from the same index. After that, after every enqueue the back index will move one index further. After every dequeue the front index will move one index further. To keep indexing convenient we will use a circular array to implement the queue. To determine the queue overflow exception, we will maintain a size variable. **Array based queues are also known as buffers.** Buffer is used in many applications such as web server request management.

Algorithm for **Array Based Queue**:

1. Initialize an array `queue` of size `n`.
2. Initialize two variables: `front` and `rear`, both set to 0 (indicating an empty queue).
3. Define the following operations:

   a. **Enqueue (Add an element to the queue):**
      - If `rear` is equal to `n`, print "Queue is full" (overflow condition).
      - Otherwise, store the new element at `queue[rear]` and increment `rear` by 1.
      - update rear = rear % length of array

   b. **Dequeue (Remove an element from the queue):**
      - If `front` is equal to `rear`, print "Queue is empty" (underflow condition).
      - Otherwise store the queue[front] in a variable temp.
      - Update the queue[front] index as empty value
      - update front = (front + 1) % len of the array.
      - Return the temp

   c. **Front (Get the front element from the queue):**
      - If `front` is equal to `rear`, print "Queue is empty."
      - Otherwise, print `queue[front]`.

   d. **Display (Print all elements of the queue):**
      - If `front` is equal to `rear`, print "Queue is empty."
      - Otherwise, traverse and print all elements from index `front` to `rear - 1`.

4. End procedure.

3.6 Queue Simulation

You need to do the following operations on a queue:

You need to perform the following operations on an array-based queue where the length of the array is 4 and the starting index of front and back is 3. [Blank means empty space].

enqueue a, enqueue b, dequeue, peek, enqueue c, peek, dequeue.

Index →	0	1	2	3	Front index	Back Index
Initial					3	3
enq (a)				a	3	0
enq (b)	b			a	3	1
deq	b				0	1
peek	b				0	1
enq (c)	b	c			0	2
peek	b	c			0	2
deq		c			1	2

Dequeued values: a, b

Peeked values: b, b

3.7 A Discussion on the Importance of Stacks and Queues

By now you well understand that larger complex data structures are made of smaller building block data structures. For example, multi-dimensional and circular arrays or vectors are made using linear arrays; lists are used to create sets and maps (which is called a dictionary in some languages and associated arrays in some others). However, as building block data structures, stacks and queues are unique in some sense.

For example, a set or a map that you create from a linked list provides additional features over their building block component that is the list. However, stacks and queues are strictly restricted forms of linked list (in rare cases, where you know the maximum number of elements you will put in them, you can use arrays for stacks and queues also). In other words, they do not provide any new features that a linked list does not already have. So it is a natural question why stacks and queues are considered fundamental data structures.

One reason for their importance is that scenarios where you need a linked list behave like a stack or queue is very frequent. For example, you need stacks for language parsing, expression evaluation, function calls, efficient graph traversals, etc. Queues are similarly important for resource scheduling, time sharing of CPU among candidate processes, message routing in network switches and routers, and efficient graph traversal.

Another central reason is that you often do not want to give access to the underlying list data structure that forms a stack or queue to the code that requested an insert or a removal from the stack/queue. If access to the list is provided then the code becomes more insecure. To consider this case, imagine direct access to the function call stack is possible. Then a program can manipulate the stack and jump from the current function to a much earlier function, instead of the immediate predecessor function that called it. Why might this be a problem? Well, this would not be a problem if all the functions in the stack belong to your own program. However, in real-world, library functions, operating system's service functions interleave with user program's functions in the stack. Unprotected jumps to those functions can crash the system.

A third reason to have stack and queues as restricted forms of linked lists (or arrays) is that their restricted functions provide smaller code that can fit in very short memory or can even be implemented inside hardware devices directly. This feature is particularly important when you need fast response, for example, in routers and switches. In fact, array based stack and queue implementations are most suited in cases like this.

Exercises

3.1: Implement the push, pop and peek functions using an array.

3.2: Implement the push, pop and peek functions using a linked list.

3.3: Reverse a string using stack.

Example:
Input: ‘CSE220’
Output: ‘022ESC”

3.4: Check whether a string is palindrome or not using stack. If it is palindrome, print True, otherwise, print False.

Example:
Input: “MADAM”
Output: True

Input: “CSE220”
Output: False

3.5: The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a **stack**. At each step:

- If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.
- Otherwise, they will leave it and go to the queue's end.

This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays students and sandwiches where sandwiches[i] is the type of the ith sandwich in the stack (i = 0 is the top of the stack) and students[j] is the preference of the jth student in the initial queue (j = 0 is the front of the queue). Return the number of students that are unable to eat.

Example 1:

Input: students = [1,1,0,0], sandwiches = [0,1,0,1]
Output: 0

Explanation:
- Front student leaves the top sandwich and returns to the end of the line making students = [1,0,0,1].

- Front student leaves the top sandwich and returns to the end of the line making students = [0,0,1,1].
- Front student takes the top sandwich and leaves the line making students = [0,1,1] and sandwiches = [1,0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [1,1,0].
- Front student takes the top sandwich and leaves the line making students = [1,0] and sandwiches = [0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [0,1].
- Front student takes the top sandwich and leaves the line making students = [1] and sandwiches = [1].
- Front student takes the top sandwich and leaves the line making students = [] and sandwiches = [].

Hence all students are able to eat.

Example 2:
Input: students = [1,1,1,0,0,1], sandwiches = [1,0,0,0,1,1]
Output: 3

3.6: Convert the following infix notation to postfix using stack following the given precedence of the operators. You must show the workings. You do not need to write code.

Consider the following precedence (decreases down the order)
1. * , /, //
2. %
3. + , -
4. == , <=, >=
5. &&

Now convert the following infix to its postfix notations:

1. A + B * C
2. A * B + C
3. A + B * C - D
4. A + B * (C - D / E)
5. A * (B + C) * D
6. A * B + C * D
7. A - B + C - D * E
8. (A - B + C) * (D + E * F)
9. A * (B + C - (D + E / F))
10. ((A + B) - C * (D / E)) + F

3.7: Convert the following postfix notations to its infix using stack. You must show the workings. You do not need to write code.

1. A B -
2. A B + C D + *
3. A B C * + D +
4. A B * C D * +
5. A B C + * D *

6. A B * C D * +
7. A B - C + D E * -
8. A B - C + D E F * + *
9. A B + C D E / * - F +
10. A B C D E / - * +

3.8: Evaluate the postfix expressions using stack. You must show the workings. You do not need to write code.

1. 3 4 * 2 5 * +
2. 2 3 - 4 + 5 6 7 * + *

3.9: Implement the enqueue, dequeue and peek functions using an array.

3.10: Implement the enqueue, dequeue and peek functions using a linked list.

3.11: Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.

Example:
Input: s = "programmingisfun"
Output: 0

Input: s = "pythonispopular"
Output: 1

Input: s = "aabb"
Output: -1

3.12: There are n people in a line queuing to buy tickets, where the 0th person is at the front of the line and the (n - 1)th person is at the back of the line.

You are given a 0-indexed integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) in order to buy more tickets. If a person does not have any tickets left to buy, the person will leave the line.

Return the time taken for the person at position k (0-indexed) to finish buying tickets.

Example 1:
Input: tickets = [2,3,2], k = 2
Output: 6
Explanation:
- In the first pass, everyone in the line buys a ticket and the line becomes [1, 2, 1].
- In the second pass, everyone in the line buys a ticket and the line becomes [0, 1, 0].
The person at position 2 has successfully bought 2 tickets and it took 3 + 3 = 6 seconds.

Example 2:

Input: tickets = [5,1,1,1], k = 0
Output: 8
Explanation:
- In the first pass, everyone in the line buys a ticket and the line becomes [4, 0, 0, 0].
- In the next 4 passes, only the person in position 0 is buying tickets.
The person at position 0 has successfully bought 5 tickets and it took 4 + 1 + 1 + 1 + 1 = 8 seconds.

3.13: At a lemonade stand, each lemonade costs $5. Customers are standing in a queue to buy from you and order one at a time (in the order specified by bills). Each customer will only buy one lemonade and pay with either a $5, $10, or $20 bill. You must provide the correct change to each customer so that the net transaction is that the customer pays $5.

Note that you do not have any change in hand at first.

Given an integer array bills where bills[i] is the bill the ith customer pays, return true if you can provide every customer with the correct change, or false otherwise.

Example 1:
Input: bills = [5,5,5,10,20]
Output: true
Explanation:
From the first 3 customers, we collect three $5 bills in order.
From the fourth customer, we collect a $10 bill and give back a $5.
From the fifth customer, we give a $10 bill and a $5 bill.
Since all customers got correct change, we output true.

Example 2:
Input: bills = [5,5,10,10,20]
Output: false
Explanation:
From the first two customers in order, we collect two $5 bills.
For the next two customers in order, we collect a $10 bill and give back a $5 bill.
For the last customer, we can not give the change of $15 back because we only have two $10 bills.
Since not every customer received the correct change, the answer is false.

3.14: Given two strings s and goal, return true if you can swap two letters in s so the result is equal to goal, otherwise, return false.

Swapping letters is defined as taking two indices i and j (0-indexed) such that i != j and swapping the characters at s[i] and s[j]. For example, swapping at indices 0 and 2 in "abcd" results in "cbad".

Example 1:
Input: s = "ab", goal = "ba"
Output: true
Explanation: You can swap s[0] = 'a' and s[1] = 'b' to get "ba", which is equal to goal.

Example 2:
Input: s = "ab", goal = "ab"
Output: false
Explanation: The only letters you can swap are s[0] = 'a' and s[1] = 'b', which results in "ba" != goal.

Example 3:
Input: s = "aa", goal = "aa"
Output: true
Explanation: You can swap s[0] = 'a' and s[1] = 'a' to get "aa", which is equal to goal.

3.15: You are given an integer array deck. There is a deck of cards where every card has a unique integer. The integer on the ith card is deck[i]. You can order the deck in any order you want. Initially, all the cards start face down (unrevealed) in one deck. You will do the following steps repeatedly until all cards are revealed:

Take the top card of the deck, reveal it, and take it out of the deck.
If there are still cards in the deck then put the next top card of the deck at the bottom of the deck.
If there are still unrevealed cards, go back to step 1. Otherwise, stop.
Return an ordering of the deck that would reveal the cards in increasing order.

Note that the first entry in the answer is considered to be the top of the deck.

Example 1:
Input: deck = [17,13,11,2,3,5,7]
Output: [2,13,3,11,5,17,7]
Explanation:
We get the deck in the order [17,13,11,2,3,5,7] (this order does not matter), and reorder it.
After reordering, the deck starts as [2,13,3,11,5,17,7], where 2 is the top of the deck.
We reveal 2, and move 13 to the bottom. The deck is now [3,11,5,17,7,13].
We reveal 3, and move 11 to the bottom. The deck is now [5,17,7,13,11].
We reveal 5, and move 17 to the bottom. The deck is now [7,13,11,17].
We reveal 7, and move 13 to the bottom. The deck is now [11,17,13].
We reveal 11, and move 17 to the bottom. The deck is now [13,17].
We reveal 13, and move 17 to the bottom. The deck is now [17].
We reveal 17.
Since all the cards revealed are in increasing order, the answer is correct.