Chapter 9 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Which of the following statements describes a benefit of using a `std::deque<>` over a `std::vector<>`?
   A) Deques store their internal data contiguously in memory, while vectors do not
   B) Using `operator[]` is typically faster on a deque than a vector
   C) Deques allow values to be efficiently added to both ends of the container, while vectors only support efficient insertion on one end
   D) The worst-time complexity of finding an arbitrary element is better with a deque than with a vector
   E) More than one of the above

2. Consider the following snippet of code:

```
1   int main () {
2     std::stack<int32_t> s;
3     s.push(12);
4     s.push(15);
5     s.push(18);
6     s.push(21);
7     s.push(24);
8     s.pop();
9     std::cout << s.top() << '\n';
10  } // main()
```

   What does this code output?
   A) `12`
   B) `15`
   C) `18`
   D) `21`
   E) `24`

3. Which of the following is **NOT** a disadvantage of using a linked list over an array to implement a stack?
   A) The time complexity of pushing in an element is $\Theta(1)$ for an array and $\Theta(n)$ for a linked list
   B) If you do not internally track the size of a linked list, the time complexity of finding the size of your stack becomes $\Theta(n)$
   C) Using a linked list is less efficient than using an array, as a linked list must allocate memory for each node individually
   D) Using a linked list results in higher memory overhead
   E) None of the above

4. Which of the following real-life situations is most similar to how a queue works?
   A) Looking for a book on a shelf in the library, where books are sorted alphabetically by title
   B) Selecting an open seat in the front of the classroom on the first day of class
   C) Searching for an exam room that is assigned by student ID
   D) Waiting in line to get seated at a restaurant
   E) Looking through the midterm exam and answering questions in order of increasing difficulty

5. What is the best possible *time* complexity of removing the most recently added element in a `std::queue<>` of size n and returning the queue with all of the other elements in their original order?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

6. What is the best possible *auxiliary space* complexity of removing the most recently added element in a `std::queue<>` of size n and returning the queue with all of the other elements in their original order?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

7. Suppose you are using a circular buffer with an array to implement a queue. In your implementation, you store two pointers: a front pointer that points to the first element in the queue and a back pointer that points one past the last element in the queue. These pointers wrap around the array as needed. Which of the following indicates that your circular buffer is full, and that you should reallocate your data to a larger array?
   A) Incrementing the front pointer causes it to point one past the end of the array
   B) Incrementing the back pointer causes it to point one past the end of the array
   C) Incrementing the front iterator causes it to point to the same location as the back pointer
   D) Incrementing the back iterator causes it to point to the same location as the front iterator
   E) More than one of the above

8. Consider the following snippet of code:

```
1  int main () {
2    MYSTERY_CONTAINER<int32_t> eecs;
3    eecs.push(203);
4    eecs.push(370);
5    eecs.push(281);
6    std::cout << eecs.top() << ", ";
7    eecs.pop();
8    eecs.push(280);
9    eecs.push(376);
10   eecs.push(183);
11   std::cout << eecs.top() << ", ";
12   eecs.pop();
13   std::cout << eecs.top();
14   eecs.pop();
15 } // main()
```

You are told that this code compiles. If the output of this code is `"281, 183, 376"`, what is a valid container type for `eecs`?

A) `std::deque<int32_t>`
B) `std::stack<int32_t>`
C) `std::queue<int32_t>`
D) `std::priority_queue<int32_t>`
E) None of the above

9. Two retail companies, Darget and Paolmart, keep track of their inventory in different ways. Darget tracks its inventory using the FIFO (first in, first out) method, assuming that the goods added to inventory first are also the first removed from inventory for sale. Paolmart tracks its inventory using the LIFO (last in, first out) method, assuming that the goods added to inventory last are the first removed from inventory for sale. The two companies want to use an STL data structure to efficiently keep track of their inventories, and they come to you for advice on which data structure to choose. What should you tell these two companies?

A) Darget should use a queue, and Paolmart should use either a stack or a vector
B) Darget should use a stack, and Paolmart should use a queue or a vector
C) Paolmart should use a queue, and Darget should use either a stack or a vector
D) Paolmart should use a stack, and Darget should use either a queue or a vector
E) Both Darget and Paolmart can efficiently use stacks, queues, and vectors to track their inventory

10. Which of the following data structures **CANNOT** be efficiently used as the underlying container for a queue?

I. `std::vector<>`
II. `std::list<>`
III. `std::deque<>`

A) I only
B) II only
C) III only
D) I and II only
E) I and III only

11. Which of the following statements is **TRUE** about container adaptors?

A) Container adaptors rely on the functionality of a standard container (vector, list, deque, etc.) to provide a specific interface
B) Stacks and queues are the only two container adaptors that exist in the STL library
C) Container adaptors can only be used on containers that provide constant time random access
D) Container adaptors will always support the full functionality of their underlying container
E) More than one of the above

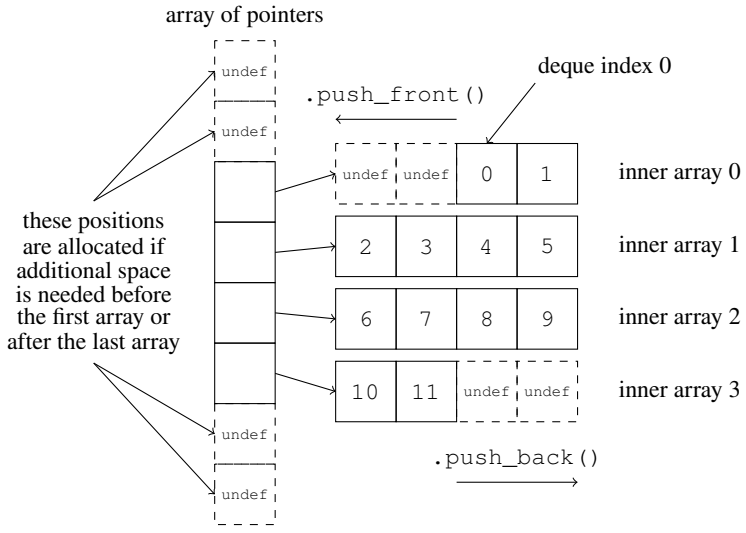
12. If you were using a deque in place of a queue in your program, which of the following statements is/are **TRUE**?

A) If you use `.push_front()` to add data to your queue, you should use `.pop_front()` to remove data from the queue
B) If you use `.push_back()` to add data to the queue, you should use `.pop_back()` to remove data from the queue
C) If you use `.push_back()` to add data to the queue, you should use `.pop_front()` to remove data from the queue
D) `.push_front()` should never be used on your deque if you want to emulate the behavior of a queue
E) More than one of the above

13. Which of the following operations **CANNOT** always be done on a deque in $\Theta(1)$ time?

A) Inserting an element at the front of the deque
B) Inserting an element in the middle of the deque
C) Inserting an element at the back of the deque
D) Removing an element at the front of the deque
E) Removing an element at the back of the deque

14. Which of the following is **NOT** a downside of using a doubly-linked list to implement a deque instead of the segmented fixed-size array-based approach discussed in section 9.6 (replicated below)?



A) Without a tail pointer, the linked list deque cannot be used as the underlying container for a stack that supports $\Theta(1)$ time push and pop
B) Without a tail pointer, the linked list deque cannot be used as the underlying container for a queue that supports $\Theta(1)$ time push and pop
C) The linked list deque cannot support $\Theta(1)$ time `operator[]`
D) The linked list deque may consume more memory due to the need to store two additional pointers for each element in the container
E) None of the above

15. If the integers 1, 2, 3, and 4 are pushed into a queue in this order and then popped out one at a time, in what order will they be removed?
A) 1, 2, 3, 4
B) 1, 2, 4, 3
C) 4, 3, 2, 1
D) 4, 3, 1, 2
E) None of the above

16. Which of the following is a computational application of a queue?
A) Queues can be used by a program to simulate recursive calls
B) Queues can be used by an internet browser to store webpage data for the back and forward buttons
C) Queues can be used by a compiler to check for matching braces in a program's code
D) Queues can be used by the operating system to allocate scarce resources among different running processes
E) None of the above

17. Consider the function:

```
T return_fifth_element(std::stack<T>& s);
```

which returns the fifth oldest element in a stack. For example, if a stack has five elements, the fifth oldest element is the element at the top of the stack. The implementation of this function **only** uses other public methods provided in the `std::stack<>` interface, and it does not access any of the stack's internal data structures directly. When the function returns, the stack *must contain the same elements as before, in the same order*. The fifth oldest element is **NOT** deleted after the function's completion; only its value is returned. If there are fewer than five elements, the newest element in the stack is returned. What is the worst-case time and auxiliary space complexities for the `return_fifth_element()` function, if the most efficient implementation is used? Let n represent the size of the input stack.
A) Runtime: $\Theta(1)$, Space: $\Theta(1)$
B) Runtime: $\Theta(1)$, Space: $\Theta(n)$
C) Runtime: $\Theta(n)$, Space: $\Theta(1)$
D) Runtime: $\Theta(n)$, Space: $\Theta(n)$
E) None of the above

18. Consider the following snippet of code:

```
1   struct MysteryContainer {
2     std::queue<int32_t> q;
3
4     void push(int32_t x) {
5       q.push(x);
6       for (int32_t i = 0; i < q.size() - 1; ++i) {
7         q.push(q.front());
8         q.pop();
9       } // for i
10    } // push()
11
12    void pop() { q.pop(); }
13    int32_t& top() { return q.front(); }
14    bool empty { return q.empty(); }
15  };
```

The above code uses a queue to implement the behavior of which data structure?

A) Queue
B) Stack
C) Priority Queue
D) Deque
E) None of the above

19. Consider the `MysteryContainer` implemented in the code from the previous question. Suppose you wanted to push n elements into the `MysteryContainer`. What is the time complexity of pushing these n elements into the `MysteryContainer`?

A) $\Theta(1)$
B) $\Theta(n)$
C) $\Theta(n^2)$
D) $\Theta(n^3)$
E) None of the above

20. Which one of the following statements is **TRUE**?

A) If a `std::deque<>` is used as the underlying container for a `std::stack<>`, then inserting an element into the middle of the stack takes $\Theta(1)$ time
B) If a `std::vector<>` is used as the underlying container for a `std::stack<>`, then inserting an element into the middle of the stack takes $\Theta(1)$ time
C) If a `std::vector<>` is used as the underlying container for a `std::queue<>`, then inserting an element into the middle of the queue takes $\Theta(1)$ time
D) If a `std::list<>` is used as the underlying container for a `std::stack<>`, then inserting an element into the middle of the stack takes $\Theta(1)$ time
E) None of the above

21. Consider the following code:

```
1   void mystery_function(std::queue<int32_t>& q) {
2     std::stack<int32_t> s;
3     while (!q.empty()) {
4       int32_t x = q.front();
5       q.pop();
6       s.push(x);
7     } // while
8     while (!s.empty()) {
9       int32_t y = s.top();
10      s.pop();
11      q.push(y);
12    } // while
13  } // mystery_function()
```

What does `mystery_function()` do?

A) It pops off the element at the back of the input queue
B) It pops off the element at the front of the input queue
C) It removes all elements in the input queue
D) It reverses the input queue
E) None of the above

22. Which one of the following statements is most accurate about the STL stack container?

A) The member function `std::stack::push()` will remove data off the top of the stack
B) Elements are added to and removed from a stack in first in, first out (FIFO) order
C) The member function `std::stack::pop()` can be used to directly retrieve data from a stack
D) The `std::stack::back()` member function allows a user to access the oldest element in the stack
E) Stacks can be implemented using either a vector or a linked list as its underlying container

23. If you want a container that only needs to support the functionality of a `std::stack<>` or `std::queue<>`, which of the following are valid reasons why it may be preferable to select one of these two containers instead of a `std::deque<>`?

    I. Stacks and queues typically perform faster than deques, since they are container adaptors that restrict the interface of their underlying container, and thus do not need to support other time consuming operations

    II. If you only need the behavior of a stack or queue, explicitly using a `std::stack<>` or `std::queue<>` instead of a `std::deque<>` better conveys your intent and may make your code easier to understand

    III. Because they are container adaptors that restrict the interface of their underlying container, stacks and queues may be safer to use since they are less prone to programming mistakes (such as popping or pushing from the wrong end)

    A) II only
    B) I and II
    C) I and III only
    D) II and III only
    E) I, II, and III

24. You want to reverse a singly-linked list. If you are only allowed to use one of these STL data structures in addition to the list you want to reverse, which of the following data structures is **least** useful?
    A) Stack
    B) Queue
    C) Deque
    D) Vector
    E) All of the above data structures are equally useful

25. The STL's queue supports the `std::queue::back()` member function, which returns a reference to the last element in the queue in $\Theta(1)$ time. Knowing this, what is the time complexity of removing the most recently added element in a `std::queue<>` of size n?
    A) $\Theta(1)$
    B) $\Theta(\log(n))$
    C) $\Theta(n)$
    D) $\Theta(n\log(n))$
    E) $\Theta(n^2)$

26. Suppose you need a container that optimizes for speed of sequential access over a fixed set of elements. Which of the following STL containers would be the best to use?
    A) `std::vector<>`
    B) `std::deque<>`
    C) `std::stack<>`
    D) `std::queue<>`
    E) `std::list<>`

27. Which of the following STL data structures can be used to efficiently simulate the behavior of a recursive algorithm without needing a function that makes a recursive call itself?

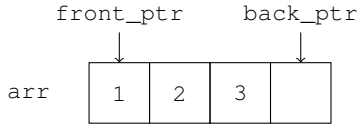
    I. Vector
    II. Queue
    III. Stack

    A) II only
    B) III only
    C) I and III only
    D) II and III only
    E) I, II, and III

28. Which of the following statements is **FALSE** regarding stacks and queues?
    A) The STL stack interface gives direct access to only the most recently inserted element, and not everything else inside the stack
    B) A vector can be efficiently used as an underlying container for implementing a stack
    C) A deque can be used to simulate the behavior of both stacks and queues
    D) If a singly-linked list is used to implement a queue, then the `.size()` member function would always take $\Theta(n)$ time
    E) None of the above (i.e., all of the above statements are true)

29. A queue is implemented using a non-circular singly-linked list that only supports a head and tail pointer. Let n be the number of nodes in the queue. The `.push()` operation is implemented by inserting a new node at the head of the list, and the `.pop()` operation is implemented by deleting the node at the tail of the list. What is the time complexity of `.push()` and `.pop()` for this queue, if the most efficient implementation is used?
    A) Push: $\Theta(1)$, Pop: $\Theta(1)$
    B) Push: $\Theta(1)$, Pop: $\Theta(n)$
    C) Push: $\Theta(n)$, Pop: $\Theta(1)$
    D) Push: $\Theta(n)$, Pop: $\Theta(n)$
    E) None of the above

30. Suppose you are implementing a queue using a circular buffer, using two pointers `front_idx` and `back_idx` (with the implementation as defined in section 9.3, and replicated below).



If the underlying array `arr` has a size of 16, which of the following configurations is **NOT** possible?

A) `front_idx` points to `arr[9]` and `back_idx` points to `arr[9]`
B) `front_idx` points to `arr[14]` and `back_idx` points to `arr[11]`
C) `front_idx` points to `arr[15]` and `back_idx` points to `arr[16]`
D) `front_idx` points to `arr[5]` and `back_idx` points to `arr[8]`
E) All of the above configurations are possible

31. Consider the map below:

	1	2	3	4	5	6	7
A	#	#	#	#	#	#	#
B	#			*z*	#	**H**	#
C	#		#	#	#		#
D	#			**S**		*w*	#
E	#		*y*		*x*		#
F	#	#	#	#	#	#	#

You want to find a path from your starting position (**S**) to your house (**H**). Cells marked with a # are walls that are inaccessible. Assume that you use a **stack**-based pathfinding scheme and add non-visited cells into a `std::stack<>` in the following direction order until you reach **H**: north, east, south, west. Which of the following correctly gives the order in which the cells labeled *w*, *x*, *y*, and *z* are encountered? For this problem, consider a cell as encountered when it is first pushed onto the pathfinding stack, and assume the search is terminated once **H** is encountered.

A) w, x, y, z
B) x, y, w, z
C) x, w, y, z
D) y, z, x, w
E) Not all of *w*, *x*, *y*, and *z* are encountered before the destination (**H**) is found

32. Which of the following statements is/are **TRUE**?

I. By default, a `std::queue<>` uses a `std::deque<>` as its underlying container
II. By default, a `std::stack<>` uses a `std::vector<>` as its underlying container
III. One difference between a `std::queue<>` that uses a `std::list<>` as its underlying container versus one that uses a `std::vector<>` is that the list-based queue does not support $\Theta(1)$ time random access, but the vector-based queue does

A) I only
B) II only
C) III only
D) I and III only
E) II and III only

33. Given two data structures of the same type and size, we define *interleaving* the two data structures as the process of creating a third data structure of the *same type* with double the size, and where popping elements interleaves the result of popping from each of the two data structures individually. For example, if popping from container A until it's empty yields [4, 8, 10], and popping from container B until it's empty yields [5, 9, 3], then interleaving A and B would produce a container that yields [4, 5, 8, 9, 10, 3] when values are popped. If you are **NOT** allowed to push elements into the two containers you want to interleave (i.e., A and B), which of the following statements is **TRUE**?

I. Interleaving two `std::vector<>` containers of size n can be done in $\Theta(n)$ time
II. Interleaving two `std::stack<>` containers of size n can be done in $\Theta(n)$ time
III. Interleaving two `std::queue<>` containers of size n can be done in $\Theta(n)$ time

A) I only
B) I and II only
C) I and III only
D) II and III only
E) I, II, and III

34. Implement the following `QueueWithStacks` templated class, which represents a queue using two stacks. The following member functions should be implemented:
    - `.size()`: returns the size of the queue time
    - `.empty()`: returns true if the queue is empty and false otherwise
    - `.push()`: inserts an element at the back of the queue
    - `.pop()`: removes an element at the front of the queue

    You may use any `std::stack<>` methods, including `.size()`, `.empty()`, `.push()`, and `.pop()`. However, you are **NOT** allowed to use `std::queue<>` in any capacity.

```
template <typename T>
class QueueWithStacks {
private:
  // These two stacks will be used to simulate the behavior of a queue
  // You may add other member variables here
  std::stack<T> stack1;
  std::stack<T> stack2;

public:
  // These methods should be implemented (as efficiently as possible under the constraints)
  size_t size() const {

  }

  bool empty() const {

  }

  void push(const T& val) {

  }

  void pop() {

  }
};
```

35. Implement the following `BrowserTab` class, which simulates a browser tab. You will start on the given `homepage` (initialized in the constructor), and you can visit different `url` strings. You are also able to go back or move forward in the URL history a given number of `steps`. Feel free to add any member variables that you may find useful.

```
class BrowserTab {
private:
  // Add any member variables that you may find useful here

public:
  // This constructor inits the BrowserTab with the homepage of the browser
  BrowserTab(const std::string& homepage) {

  }

  // This function visits the given url from the current page
  // Calling this function clears up all the forward history.
  void visit(const std::string& url) {

  }

  // This function moves back in history a total of "steps" steps
  // (or as far back as possible if the number of steps specified
  // exceeds the history length) - return the url you end on after
  // moving back in history
  std::string back(uint32_t steps) {

  }

  // This function moves forward in history a total of "steps" steps
  // (or as far forward as possible if the number of steps specified
  // exceeds the history length) - return the url you end on after
  // moving forward in history
  std::string forward(uint32_t steps) {

  }
};
```

36. You are given a stack of integers. Implement the following `flip()` method, which flips the order of the elements in the input stack using *only one auxiliary stack* and no other STL or C-style container (although primitive variables are okay).

```
void flip(std::stack<int32_t>& input);
```

You should implement your solution in worst-case $\Theta(n^2)$ time and $\Theta(n)$ auxiliary space, where n is the size of the input stack.

37. Suppose you are given a vector of integers and a number k, and you consider k or more copies of the same value in a row to constitute a "group". You need to sum up any values that are left over after forming and removing groups; the values left over are known as the remnants. For example, given $k = 3$ and the following vector:

```
[3, 5, 5, 2, 2, 2, 5, 5, 5, 2, 3, 4, 4, 4, 3]
```

Starting from the left, there is only one copy of 3 (not a group since there are less than $k = 3$ copies of this value), followed by two 5's (also not a group). These are followed by three 2's, which do form a group, so they are removed from the list:

```
[3, 5, 5, 2, 2, 2, 5, 5, 5, 2, 3, 4, 4, 4, 3]
[3, 5, 5, 5, 5, 5, 2, 3, 4, 4, 4, 3]
```

Now, there are five copies of the value 5, which do form a group (since the number of 5's is greater than $k = 3$). Thus, these 5's are removed from the list, and the remaining values to be considered are:

```
[3, 5, 5, 5, 5, 5, 2, 3, 4, 4, 4, 3]
[3, 2, 3, 4, 4, 4, 3]
```

The cluster of 4's is a group, so removing these leaves us with:

```
[3, 2, 3, 4, 4, 4, 3]
[3, 2, 3, 3]
```

Even though there are k copies of the value 3 remaining, they are not consecutive and thus do not constitute as a group. These four remaining values are the remnants of the original input vector.

Implement the `sum_remnant_values()` function, which takes in a vector of integers `nums` and an integer k, and returns the sum of all remnant values, using the procedure illustrated above. For the given example, the function would return $3 + 2 + 3 + 3 = 11$.

```
int32_t sum_remnant_values(const std::vector<int32_t>& nums, int32_t k);
```

You may assume that `nums` is not empty and that $k > 1$. Note that you should only return the sum of the remnants; you should not physically remove values from the input vector itself. You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(n)$ auxiliary space, where n is the size of the input vector.

38. You are given a string `str` that consists of the characters `'('` and `')'`. Implement a function that returns the minimum number of parentheses (either `'('` or `')'`) that need to be added to the string so that the resulting string is balanced. For example, given the input string `"()))(("`, you would return 4, since a minimum of 4 new parentheses are need to balance the string: `"()()()(())"` .

```
int32_t min_add_to_make_string_valid(const std::string& str);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(n)$ auxiliary space, where n is the size of the input list.

39. Implement the following `MovingAverageCalculator` class, which can be used to calculate the average of the k most recent values that were added. For example, if a calculator is initialized with $k = 3$, and the values of 2 and 4 were added, then `.calculate_k_average()` would return a value of $(2 + 4) / 2 = 3$. If the value 6 were then added afterward, then `.calculate_k_average()` would return a value of $(2 + 4 + 6) / 3 = 4$. Then, if the value 8 were added, `.calculate_k_average()` would return $(4 + 6 + 8) / 3 = 6$ (note that the 2 is no longer considered since it is now the 4th most recently added value, which exceeds our limit of $k = 3$).

```
1  class MovingAverageCalculator {
2  private:
3    // Add any member variables that you may find useful here
4
5  public:
6    // This constructor inits the MovingAverageCalculator with a limit of k
7    MovingAverageCalculator(uint32_t k) {
8
9    }
10
11   // Adds a value to be considered
12   void add_value(double val) {
13
14   }
15
16   // Calculates the average of the k most recent values that were added
17   // If less than k values were added, return the average of all the values
18   double calculate_k_average() const {
19
20   }
21 };
```

Chapter 9 Exercise Solutions

1. **The correct answer is (C).** Deques allow values to be inserted efficiently at both ends of the container, while vectors only allow efficient insertions at the back. Option (A) is incorrect because vectors do store their data contiguously in memory. Option (B) is incorrect because the contiguity of elements in a vector allows `operator[]` to be implemented with simple pointer addition (deques also require arithmetic, but the more advanced internal structure of a deque results in slightly more complicated arithmetic compared to a vector). Option (D) is incorrect because finding an arbitrary element is worst-case $\Theta(n)$ regardless of which data structure you choose.

2. **The correct answer is (D).** Five elements are pushed onto the stack. The first `.pop()` on line 8 removes the most recently added element, or `24`. Then, the call to `.top()` on line 9 retrieves the most recent element currently on the stack, which is `21`.

3. **The correct answer is (A).** The time complexity of pushing an element into a stack implemented with a linked list is also $\Theta(1)$, since you can simply attach the element to the beginning of the list. All of the remaining options are valid disadvantages of a linked list implementation over an array one.

4. **The correct answer is (D).** The situation described in answer choice (D) is most similar to how a queue works, since those who are in line first are seated first.

5. **The correct answer is (C).** To access the most recently added element in a `std::queue<>`, you must remove the $n-1$ elements that are before it, which takes $\Theta(n)$ time.

6. **The correct answer is (A).** You do not need any additional space to complete this process: simply pop off the front element and readd it to the back of the queue. Continue doing this for $n-1$ elements before you reach the final element; at this point, you can take the element out and the other elements will still be in their original order.

7. **The correct answer is (D).** When you push back an element into the circular buffer, you increment the back iterator; if this back iterator ends up at the same position as the first element in the circular buffer (the front iterator), you would know that your circular buffer is full.

8. **The correct answer is (B).** When 203, 370, and 281 are pushed into the container, 281 is the first to be retrieved. When 280, 376, and 183 are added to the container, 183 is the first to be retrieved. After 183 is popped off, 376 is the next to be retrieved. Notice that the top value is always the most recent element inserted into the container; thus, of the options provided, the container must be a stack. Notice that a deque would not work here, since they do not support `.push()` and `.pop()` methods without specifying the end you want to push or pop from, so the code would not compile if `MYSTERY_CONTAINER` were a `std::deque`.

9. **The correct answer is (A).** Since Darget needs to insert inventory from one end of a container and remove it from the other, it should use a queue, which can be used to efficiently support FIFO behavior. On the other hand, Paolmart needs to insert and remove inventory from the same end of a container (since the newest element is removed first), which can be efficiently done using a stack or a vector.

10. **The correct answer is (A).** Queues need to support efficient insertion and removal from both ends of its underlying container, since the side in which you insert an element is not the side you should pop elements out of. This can be done with a `std::list<>` or a `std::deque<>`, which can support $\Theta(1)$ insertion and removal from both ends of the container, but not with a `std::vector<>`, which only supports $\Theta(1)$ insertion and removal from the back.

11. **The correct answer is (A).** Only option (A) is a true statement, since it describes the functionality of a container adaptor. Option (B) is false because stacks and queues are not the only container adaptors on the STL (priority queues are another one). Option (C) is false because random access is not a prerequisite for an underlying container: for containers like stacks and queues where random access is not necessary and you only need efficient access at the ends of the container, you can use a `std::list<>` as the underlying container. Option (D) is false, since container adaptors may restrict or modify the functionality of its underlying container to support the desired behavior.

12. **The correct answer is (C).** If you want to emulate the behavior of a queue using a deque, then you should insert and remove elements from opposite ends of the deque.

13. **The correct answer is (B).** Deques support $\Theta(1)$ insertions and removals from the ends of the container, but not the middle. Inserting an element into the middle of the deque may require elements to be shifted in memory, which could take $\Theta(n)$ time.

14. **The correct answer is (B).** Without a tail pointer, a linked list (regardless of whether it is doubly- or singly-linked) cannot support $\Theta(1)$ insertion or removal from the back. To implement the queue, you would need a way to insert and remove elements in $\Theta(1)$ from both ends of the container, which is not supported with a linked list deque with no tail pointer.

15. **The correct answer is (A).** The order in which elements are inserted into a queue is the same order in which they are removed.

16. **The correct answer is (D).** Queues are often used to allocate scarce resources among different running processes by providing them in first in, first out order for different processes (e.g., if two processes need a shared resource, and one gets there before the other, then the first process would get the resource first). The other three applications are better suited for a stack, which supports efficient insertions and removals from the same end.

17. **The correct answer is (D).** Since you are only allowed to use public methods provided in the `std::stack<>` interface, you would not be able to access the fifth oldest element without popping out all but five of the elements in the stack. Since the order of elements in the stack cannot be changed either, you would need to store the values you popped as well, in an external data structure. Because of this, the worst-case time and auxiliary space complexities of finding the fifth oldest element in a stack are both $\Theta(n)$.

18. **The correct answer is (B).** This class uses a queue to implement the behavior of a stack. This is because, after every insertion, all the previously existing elements in the queue are popped out and reinserted back in the queue. This ensures that the most recently insert element is always at the front of the queue, which essentially recreates the functionality of a stack.

19. **The correct answer is (C).** Each insertion takes $\Theta(n)$ time, since all the elements in the queue are removed and inserted back in. To insert n elements, the total time complexity would become $n \times \Theta(n) = \Theta(n^2)$.

20. **The correct answer is (E).** None of the statements are true. The STL stack and queue do not support constant-time insertions in the middle of the container, regardless of what the underlying container is.

21. **The correct answer is (D).** This function takes all the elements in a queue and inserts them into a stack. Then, it takes the elements in the stack and adds it back to the queue. Since stacks provide LIFO access, the last elements of the original queue are taken out of the stack and reinserted back into the queue first, essentially reversing the order of the queue's contents.

22. **The correct answer is (E).** The underlying container of a stack needs to support $\Theta(1)$ insertion and removal from the same end of the container, which works for both vectors and linked lists. Option (A) is false since pushing an element inserts it into the stack instead of removing it. Option (B) is false because elements are added to and removed from a stack in last in, first out (LIFO) order. Option (C) is false because popping an element out of an STL stack does not return a reference to the removed value. Option (D) is false because no such member exists to access the oldest element in the stack.

23. **The correct answer is (D).** If you only need the functionality of a `std::stack<>` or `std::queue<>`, it may be preferable to use one of these container adaptors rather than a `std::deque<>` since it makes clear in your code what the container is designed to be used for and also lowers the likelihood of programming mistakes (e.g., if you use a `std::queue<>`, you can just use `.push()` and `.pop()` and the container handles it properly for you; for a `std::deque<>`, you would need to keep track of which direction you are inserting and removing elements yourself). Statement I is false because stacks and queues are essentially the same as deques, just with a restricted interface, so there is no difference in performance.

24. **The correct answer is (B).** To build a singly-linked list in reverse order, one common strategy is need to start from the last node and build the list by inserting nodes from the back to the front. This can be done using a container that supports efficient last in, first out (LIFO) access (e.g., push all the nodes into the container, take them out one at a time in LIFO order, and the attach them to the reversed list). Of the four containers provided, all support efficient LIFO access except the queue, which only supports efficient first in, first out (FIFO) access.

25. **The correct answer is (C).** The time complexity it takes to *access* the most recently added element is irrelevant here, since you are not changing the contents of the queue. To *remove* the most recently added element from a `std::queue<>`, you have no choice but to first remove and reinsert all the elements before it, which would take $\Theta(n)$ time.

26. **The correct answer is (A).** Sequential access is fastest if the elements to access are stored contiguously in memory. Stacks and queues do not support sequential access at all. Of the remaining three containers, only the `std::vector<>` guarantees that all of its elements are stored contiguously in memory.

27. **The correct answer is (C).** To emulate the stack frames of a recursive call without performing any recursion, you will need a container that supports efficient LIFO access (since with recursion, the item you put on the stack frame most recently is the one you want to retrieve first). This can only be done with a vector and a stack, and not a queue.

28. **The correct answer is (D).** There is nothing that prevents the queue from keeping track of its size internally, even if it uses a singly-linked list as its underlying container. In this case, the time complexity of accessing the queue's size would be $\Theta(1)$, not $\Theta(n)$.

29. **The correct answer is (B).** Inserting an element at the front of a singly-linked list takes $\Theta(1)$ time, and removing an element at the back of a singly-linked list takes $\Theta(n)$ time, regardless of whether a tail pointer is supported. Since push acts upon the front and pop acts upon the back, the time complexity of push would be $\Theta(1)$, and the time complexity of pop would be $\Theta(n)$.

30. **The correct answer is (C).** Only configuration (C) is impossible, since the back pointer points past the end of the array. In a circular buffer, the back pointer would circle back to the front if it ever moves past the end, so if the last element is at the last index of the array, then `back_idx` would end up at index 0, and not 16.

31. **The correct answer is (D).** During the first iteration, D5, E4, and D3 are inserted into the stack, in this order.

	1	2	3	4	5	6	7
A	#	#	#	#	#	#	#
B	#			*z*	#	**H**	#
C	#		#	#	#		#
D	#			**S**		*w*	#
E	#		*y*		*x*		#
F	#	#	#	#	#	#	#

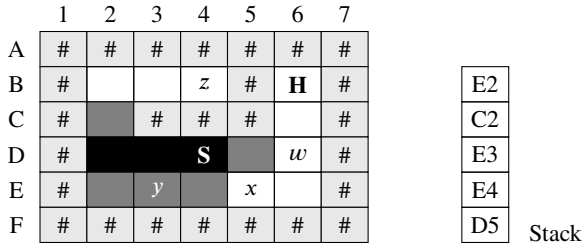
Stack
D3
E4
D5

D3 is now at the top of the stack, so on the next iteration, we take it out and add the unvisited cells of E3 (*y*) and D2 into the stack.

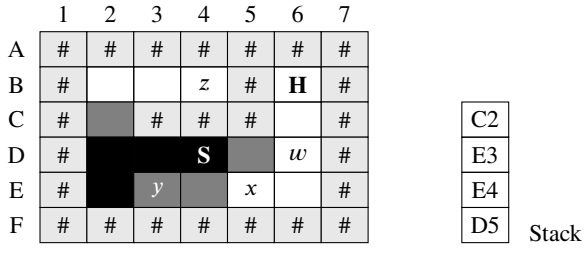
	1	2	3	4	5	6	7
A	#	#	#	#	#	#	#
B	#			*z*	#	**H**	#
C	#		#	#	#		#
D	#			**S**		*w*	#
E	#		*y*		*x*		#
F	#	#	#	#	#	#	#

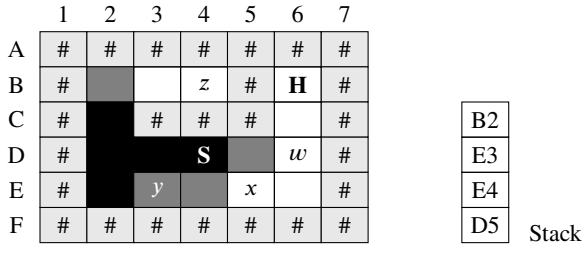
Stack
D2
E3
E4
D5

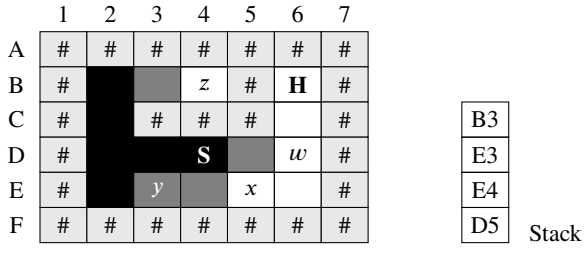
D2 is now at the top of the stack, so on the next iteration, we take it out and add the unvisited cells of C2 and E2 into the stack.

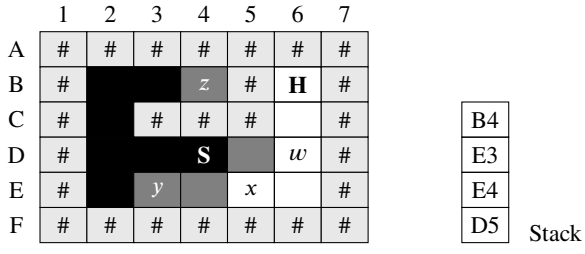


Continuing this process, we eventually encounter cell z next.

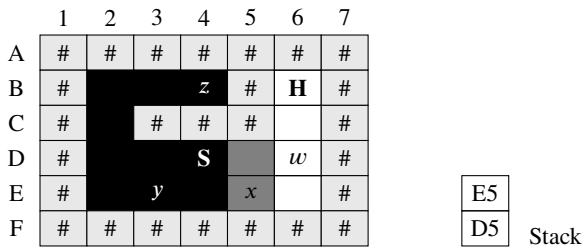








The next cell we encounter is x.



Lastly, while not explicitly shown, we encounter cell w before reaching our destination **H**.

32. **The correct answer is (A).** Both `std::stack<>` and `std::queue<>` use a `std::deque<>` as its default underlying container. Statement III is false because a `std::queue<>` does not support random access

33. **The correct answer is (E).** All three containers can be interleaved in $\Theta(n)$. For the vector, you can just iterate over the two input vectors simultaneously and alternate pushing their values into the interleaved vector. For the queue, you can alternate popping out elements from the two input queues and push them into the interleaved queue. The stack is a bit more tricky since you only have access to the most recently added element, so the items at the top of the two stacks must be pushed into the interleaved stack last. This requires the use an additional

stack; items are popped out of the two input stacks and alternately pushed into this additional stack first, and then these elements are popped back out and pushed into the interleaved stack (which reverses the order of elements to support the stack's LIFO processing order).

34. To solve this problem, we can just use the strategy discussed in example 9.1. We will treat one stack as the front of our queue, and the other stack as the back of our queue. If pop is ever invoked while the front stack is empty, we would move all the elements from the back stack into the front stack. An implementation is shown below:

```
template <typename T>
class QueueWithStacks {
private:
  std::stack<T> stack1;  // front stack
  std::stack<T> stack2;  // back stack
public:
  size_t size() const {
    return stack1.size() + stack2.size();
  } // size()

  bool empty() const {
    return stack1.empty() && stack2.empty();
  } // empty()

  void push(const T& val) {
    stack2.push(val);
  } // push()

  void pop() {
    if (stack1.empty()) {
      while (!stack2.empty()) {
        stack1.push(stack2.top());
        stack2.pop();
      } // while
    } // if
    stack1.pop();
  } // pop()
};
```

35. One way to solve this problem is to use two stacks, one storing back pages and the other storing forward pages. Whenever you visit a page, add it to the back stack and clear out the forward stack. Whenever you want to go back by k pages, pop k pages out of the back stack and into the forward stack, and then set the current page to the one at the top of the back stack. Whenever you want to go forward by k pages, pop k pages out of the forward stack and into the back stack, and then set the current page to the one at the top of the forward stack. One implementation is shown below:

```
class BrowserTab {
private:
  std::stack<std::string> pages_back, pages_forward;
  std::string current_page;
public:
  BrowserTab(const std::string& homepage) {
    current = homepage;
  } // BrowserTab()

  void visit(const std::string& url) {
    pages_forward = std::stack<std::string>();
    pages_back.push(current);
    current = url;
  } // visit()

  std::string back(uint32_t steps) {
    while (--steps >= 0 && !pages_back.empty()) {
      pages_forward.push(current);
      current = pages_back.top();
      pages_back.pop();
    } // while
    return current;
  } // back()

  std::string forward(uint32_t steps) {
    while (--steps >= 0 && !pages_forward.empty()) {
      pages_back.push(current);
      current = pages_forward.top();
      pages_forward.pop();
    } // while
    return current;
  } // forward()
};
```

36. One unique challenge of this problem is that you are restricted to only a single auxiliary stack; it is trivial to reverse the order of the elements in another stack (by just popping the elements out of the input stack and pushing it into the other stack), but ensuring that the elements are reversed in the input stack itself is trickier. A general solution to this problem is to repeat the following process from $k = 0$ to $k = n - 1$:
    1. Pop off the top of the input stack and store it in a local variable.
    2. Pop $n - k - 1$ elements from the input stack and push it into the auxiliary stack.
    3. Push the element that was popped off in step 1 back onto the input stack.
    4. Pop all elements in the auxiliary stack back into the input stack.

    This solution is implemented in code as follows:

```
void flip(std::stack<int32_t>& input) {
  std::stack<int32_t> aux;
  size_t n = input.size();

  for (size_t k = 0; k < n; ++k) {
    int32_t next = input.top();
    input.pop();

    for (size_t j = 0; j < (n - k - 1); ++j) {
      aux.push(input.top());
      input.pop();
    } // for j

    input.push(next);

    while (!aux.empty()) {
      input.push(aux.top());
      aux.pop();
    } // while
  } // for k
} // flip()
```

37. You can use a LIFO container such as a stack or vector to keep track of groups. To do so, iterate over the input values and check the most recently added element to see if it is equal to the new value to be added; if it is, increment a counter associated with that element. Once the counter exceeds k, then you have a group that can be removed — by removing groups while iterating over the values, you are able to handle groups larger than k that are only formed after removing other groups in the input. One implementation is shown below:

```
int32_t sum_remnant_values(const std::vector<int32_t>& nums, int32_t k) {
  struct Counter {
    int32_t value{};
    int32_t count{};
  };

  std::stack<Counter> s;
  int32_t sum = 0;

  for (int32_t num : nums) {
    sum += num;
    if (s.empty()) {
      s.push({ num, 1 });
    } // if
    else if (s.top().value == num) {
      ++s.top().count;
    } // else if
    else {
      if (s.top().count >= k) {
        sum -= s.top().value * s.top().count;
        s.pop();
      } // if
      if (s.empty() || s.top().value != num) {
        s.push({ num, 1 });
      } // if
      else {
        ++s.top().count;
      } // else
    } // else
  } // for

  // check one more time in case last element caused group to exceed size of k
  if (s.top().count >= k) {
    sum -= s.top().value * s.top().count;
  } // if

  return sum;
} // sum_remnant_values()
```

38. One way to solve this problem is to keep counters for the number of left and right parentheses that need to be added for the string to be balanced. We will iterate over the string and count the number of left and right parentheses that are required using the following procedure:
    - If a left parenthesis is encountered, increment the number of right parenthesis that are needed (since one will be needed to match this left parenthesis).
    - If a right parenthesis is encountered:
      - If the right parenthesis counter is greater than zero, decrement it.
      - Otherwise, increment the left parenthesis counter, since a new left parenthesis will be needed to match this right parenthesis.

    An implementation of this solution is shown below:

```
int32_t min_add_to_make_string_valid(const std::string& str) {
  int32_t left = 0, right = 0;
  for (char c : str) {
    if (c == '(') {
      ++right;
    } // if
    else if (right > 0) {
      --right;
    } // else if
    else {
      ++left;
    } // else
  } // for c

  return left + right;
} // min_add_to_make_string_valid()
```

    This can also be done using a stack. Whenever we encounter a left parenthesis, we push it onto the stack. Whenever we encounter a right parenthesis, we pop it from the stack (or, if the stack is empty, then the right parenthesis has no matching left parenthesis, so we will increment a separate counter). This solution is shown below:

```
int32_t min_add_to_make_string_valid(const std::string& str) {
  int32_t left = 0;
  std::stack<char> s;
  for (char c : str) {
    if (c == '(') {
      s.push(c);
    } // if
    else {
      if (!s.empty()) {
        s.pop();
      } // if
      else {
        ++left;
      } // else
    } // else
  } // for c

  return left + s.size();
} // min_add_to_make_string_valid()
```

39. This problem can be solved using a queue that stores the elements in the window. Each new element is pushed into the back of the queue, and whenever the capacity exceeds k, we pop the oldest element out from the front of the queue. An implementation is shown below:

```
class MovingAverageCalculator {
private:
  std::queue<double> window;
  double sum{};
  uint32_t window_size{};
public:
  MovingAverageCalculator(uint32_t k) {
    window_size = k;
  } // MovingAverageCalculator()

  void add_value(double val) {
    if (window.size() == window_size) {
      sum -= window.front();
      window.pop();
    } // if
    sum += val;
    window.push(val);
  } // add_value()

  double calculate_k_average() const {
    return sum / window.size();
  } // calculate_k_average()
};
```