Chapter 8 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. The largest Fibonacci number that can be represented as a 32-bit integer is the 47^{th} Fibonacci number. Suppose you store the first 47 Fibonacci numbers as `int32_t` values in both a doubly-linked list and an array. What is the difference in the number of bytes that these two data structures take up in memory? Assume that we are using a 64-bit system, where pointers take up 8 bytes.

```
1  // Linked list of 47 Fibonacci Numbers
2  struct Node {
3    Node* next;
4    Node* prev;
5    int32_t value;
6  };
7
8  // Array of 47 Fibonacci Numbers
9  int32_t arr[47];
```

   A) 376 bytes (47×8)
   B) 752 bytes (47×16)
   C) 1,128 bytes (47×24)
   D) 1,504 bytes (47×32)
   E) None of the above

2. Which of the following statements is **FALSE**?
   A) Traversing a doubly-linked list from beginning to end takes $\Theta(n)$ time
   B) Searching for a particular element in a singly-linked list can take $\Theta(n)$ time
   C) Inserting into a doubly-linked list takes $\Theta(1)$ time if you are given an iterator to the insertion point
   D) The last element in a singly-linked list with a tail pointer can be removed in $\Theta(1)$ time
   E) None of the above

3. What is the worst-case time complexity of appending an element to the back of an array vs. a singly-linked list with a head pointer but **NO** tail pointer? Assume the there is enough capacity in the array to store another element (so no reallocation is involved).
   A) Both the array and linked list are $\Theta(1)$
   B) The array is $\Theta(1)$ but the linked list is $\Theta(n)$
   C) The array is $\Theta(n)$ but the linked list is $\Theta(1)$
   D) Both the array and linked list are $\Theta(n)$
   E) None of the above

4. What is the worst-case time complexity of inserting an element after another element with a given value (i.e., the value of the element to insert after is known, but not the position) array vs. a singly-linked list with a head pointer but **NO** tail pointer? Assume the there is enough capacity in the array to store another element (so no reallocation is involved).
   A) Both the array and linked list are $\Theta(1)$
   B) The array is $\Theta(1)$ but the linked list is $\Theta(n)$
   C) The array is $\Theta(n)$ but the linked list is $\Theta(1)$
   D) Both the array and linked list are $\Theta(n)$
   E) None of the above

5. What is the worst-case time complexity of accessing an element at an arbitrary index in an array vs. a singly-linked list with a head pointer but **NO** tail pointer?
   A) Both the array and linked list are $\Theta(1)$
   B) The array is $\Theta(1)$ but the linked list is $\Theta(n)$
   C) The array is $\Theta(n)$ but the linked list is $\Theta(1)$
   D) Both the array and linked list are $\Theta(n)$
   E) None of the above

6. For which of the following containers is it possible to insert a new element *before* another existing element in constant time, provided that you are given a pointer to the existing object that you want to insert before?
   I. Vector
   II. Singly-linked list
   III. Doubly-linked list

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

7. Which of the following operations has a better time complexity when performed on a doubly-linked list instead of a singly-linked list? Assume that the lists are **NOT** implemented with a tail pointer.
   I. Inserting an element at the back of the list
   II. Inserting a node after another node, given the other node's pointer
   III. Erasing an element at the back of the list
   IV. Erasing an element when given its pointer

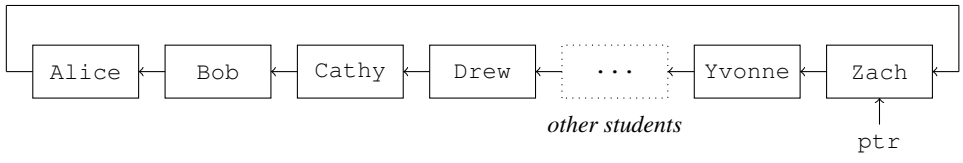
   A) IV only
   B) I and III only
   C) II and IV only
   D) III and IV only
   E) I, II, III, and IV

8. You are trying to implement a queue that keeps track of students who go to EECS 281 office hours. When a student arrives, they put their information into the program and get sent to the back of the line. When an instructor is available, they query the program for the student who has been waiting the longest. The student is then removed from the queue after they receive help. There is no limit on the number of students that end up waiting for office hours. If an efficient time complexity is the sole concern, and these are the only behaviors that need to be supported, which of the following data structures would be best for storing the `Student` objects for this program?
   A) A singly-linked list with both head and tail pointers
   B) A doubly-linked list with only a head pointer
   C) A fixed size array
   D) A dynamic array (i.e., vector)
   E) All of the above data structures are equally efficient

9. You currently have a singly-linked list that only has a head pointer. For which of the following operations would adding in a tail pointer *improve* the worst-case time complexity of that operation? Assume that the most efficient algorithm is used.
   A) Reversing the linked list
   B) Removing the last element in the list
   C) Finding an element in the list
   D) Removing the first element in the list
   E) None of the above

10. What are the time complexities of finding the 10^{th} element in a singly-linked list and finding the 10^{th} *to last* element in a singly-linked list? Let n be the number of nodes in the linked list. You may assume that the list contains more than 10 elements.
    A) $\Theta(1)$ and $\Theta(1)$
    B) $\Theta(1)$ and $\Theta(n)$
    C) $\Theta(n)$ and $\Theta(1)$
    D) $\Theta(n)$ and $\Theta(n)$
    E) None of the above

11. Consider the following circular singly-linked list of n students, which represents an office hours queue.



    The pointer `ptr` is the only way to access elements in the queue. As shown above, `ptr` is currently pointing to the student named Zach in the line. You are told that both adding and removing students from the queue always take $\Theta(1)$ time. Once a student is in the queue, they cannot leave until they receive help. Under this setup, which of the following students could potentially be the next in line to be helped?
    I. Alice
    II. Yvonne
    III. Zach

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

12. Suppose you have a doubly-linked list that supports a head and tail pointer. How many pointers are modified when an element is inserted into the middle of the list? Assume that there are elements on both sides of the insertion position.
    A) 0
    B) 1
    C) 2
    D) 4
    E) 6

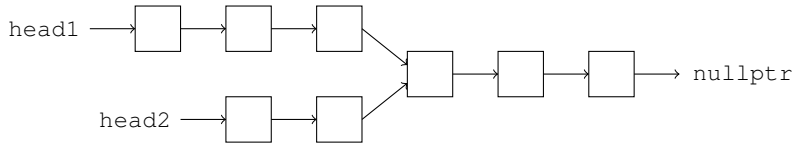
13. Which of the following **CANNOT** be done in $\Theta(1)$ time? Assume that the lists mentioned in the answers support both head and tail pointers.
    A) Inserting an element at the front of a singly-linked list
    B) Inserting an element at the back of a singly-linked list
    C) Deleting an element at the front of a singly-linked list
    D) Deleting an element at the back of a singly-linked list
    E) None of the above

14. Consider the following function, which detects whether a cycle exists in a singly-linked list and removes the cycle if there is one.

```
1   void remove_list_cycle(Node* head) {
2     if (head == nullptr || head->next == nullptr) {
3       return;
4     } // if
5
6     Node* slow = head->next;
7     Node* fast = head->next->next;
8     while (slow != nullptr && fast != nullptr) {
9       if (slow == fast) {
10        break;
11      } // if
12
13      slow = slow->next;
14      fast = fast->next->next;
15    } // while
16
17    if (slow == fast) {
18      slow = head;
19      while (slow->next != fast->next) {
20        slow = slow->next;
21        fast = fast->next;
22      } // while
23
24      fast->next = nullptr;
25    } // if
26  } // remove_list_cycle()
```

    The above implementation may have a bug. Which of the following linked lists, when passed into this function, would expose this bug?
    A) A singly-linked list that has a cycle with an even number of nodes
    B) A singly-linked list that has a cycle with an odd number of nodes
    C) A singly-linked list that has an even number of nodes, but no cycle
    D) A singly-linked list that has an odd number of nodes, but no cycle
    E) None of the above, there is actually no bug

15. You have the head pointers of two singly-linked lists that converge to become a single linked list. An illustration is shown below:



    The length of the first list is m, and the length of the second list is n. There is no relationship between m and n; that is, you cannot assume that m is larger than n or vice versa. What is the worst-case time complexity of finding the intersecting node between the two lists (i.e., the first node that is shared by both lists) if you use the most efficient algorithm?
    A) $\Theta(m+n)$
    B) $\Theta(n^2)$
    C) $\Theta(mn)$
    D) $\Theta(\min(m,n)))$
    E) $\Theta(\min(mn^2, m^2n)))$

16. Which of the following statements is **FALSE**?
    A) The nodes of a linked list are not necessarily stored sequentially in memory on the heap
    B) Singly- and doubly-linked lists that store n elements both have $\Theta(n)$ memory overhead
    C) Given a head pointer, identifying whether or not a linked list is circular takes $\Theta(n)$ time for both singly- and doubly-linked lists
    D) An element can be inserted at any position in a doubly-linked list in constant time, as long as the position to insert after is provided using a node pointer
    E) None of the above

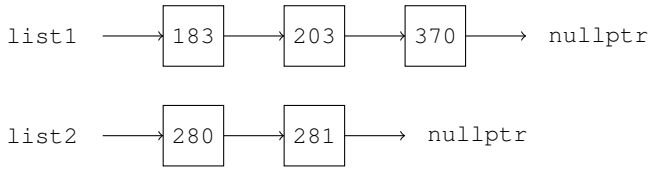
17. You are given two singly-linked lists of size n containing integers. Both lists are *sorted*. What is the worst-case time complexity of merging the two lists into a single, sorted linked list if you use the most efficient algorithm? Assume that both lists support a head and tail pointer.
    A) $\Theta(1)$
    B) $\Theta(\log(n))$
    C) $\Theta(n)$
    D) $\Theta(n\log(n))$
    E) $\Theta(n^2)$

18. For which of the following situations would a linked list be preferable to a vector?
    A) When lower memory overhead is needed
    B) When fast sequential traversal is needed
    C) When pointers and iterators to elements in the container cannot be invalidated
    D) When $\Theta(1)$ random access is needed
    E) None of the above

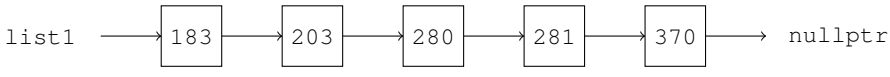
19. Suppose you want to a find an element given its value, either in an array or in a singly-linked list that supports a head pointer. Which of the following is **TRUE** about the worst-case time complexity of accomplishing this on these two container types?
    A) Finding this element takes $\Theta(1)$ for the array and $\Theta(1)$ time for the linked list
    B) Finding this element takes $\Theta(1)$ for the array and $\Theta(n)$ time for the linked list
    C) Finding this element takes $\Theta(n)$ for the array and $\Theta(1)$ time for the linked list
    D) Finding this element takes $\Theta(n)$ for the array and $\Theta(n)$ time for the linked list
    E) None of the above

20. Given a linked list with n nodes, which of the following statements is **TRUE**?
    A) The first element in a singly-linked list can be removed in $\Theta(1)$ time
    B) The last element in a singly-linked list can be removed in $\Theta(1)$ time
    C) Finding an element in a doubly-linked list takes worst-case $\Theta(\log(n))$ time if the list supports both head and tail pointers
    D) The time complexity of reversing a doubly-linked list with both head and tail pointers is better than the time complexity of reversing a doubly-linked with only a head pointer
    E) More than one of the above

21. Linked lists support a special operation known as *splicing*, which can be used to transfer elements from one list into another. For example, consider the following two lists, `list1` and `list2`:

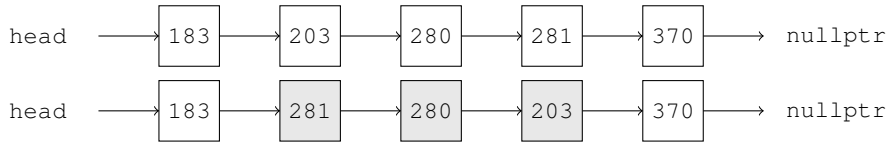


    We can splice the entirety of `list2` into `list1` by transferring all the elements in `list2` into `list1` at any given position, as shown. After this operation, `list2` would become empty.



    Suppose you are given two *doubly-linked* lists that support tail pointers, one of length m and one of length n, and an iterator pointing to the element that the spliced elements should be inserted before (in the list of length n). What is the time complexity of splicing the entirety of the list of length m into the list of length n at the position before the given iterator, if you use the most efficient implementation strategy?
    A) $\Theta(1)$
    B) $\Theta(m)$
    C) $\Theta(n)$
    D) $\Theta(m+n)$
    E) $\Theta(mn)$

22. You are given the head of a singly-linked list and two valid indices `left` and `right` (1-indexed), where `left` $\leq$ `right`. Reverse the nodes of the list from position `left` to position `right`, inclusive, and return the reversed linked list. For example, given the following list and `left` = `2` and `right` = `4`, you would reverse the list as follows:

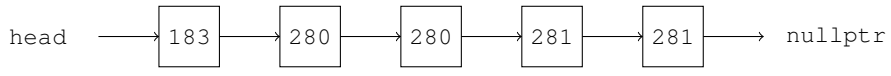


The function header is provided below:

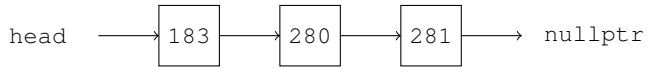
```
struct Node {
  int32_t val;
  Node* next;
  Node(int32_t val_in) : val{val_in}, next{nullptr} {}
};

Node* reverse_list(Node* head, uint32_t left, uint32_t right);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(1)$ auxiliary space, where n is the size of the list.

23. You are given the head of a *sorted* singly-linked list. Implement a function that deletes all duplicates in the sorted list so that each element only appears once, and then returns the sorted linked list without duplicates. For example, given the following list:



you would return the following list without any duplicates:

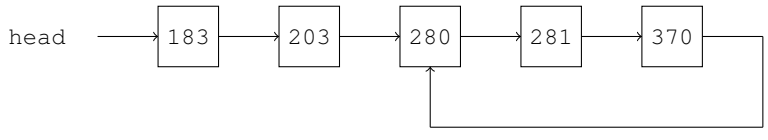


The function header is provided below:

```
struct Node {
  int32_t val;
  Node* next;
  Node(int32_t val_in) : val{val_in}, next{nullptr} {}
};

Node* remove_duplicates(Node* head);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(1)$ auxiliary space, where n is the size of the list.

24. You are given the head of a singly-linked list. Implement a function that returns the node where a cycle begins, or `nullptr` if there is no cycle. For example, given the following list, you would return the node that stores the value `280`.

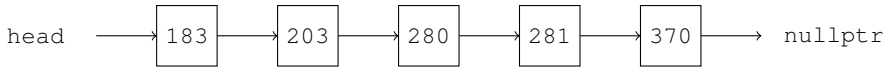


The function header is provided below:

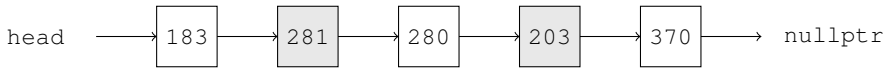
```
struct Node {
  int32_t val;
  Node* next;
  Node(int32_t val_in) : val{val_in}, next{nullptr} {}
};

Node* first_node_in_cycle(Node* head);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(1)$ auxiliary space, where n is the size of the list.

25. You are given the head of a singly-linked list and an integer k. Implement a function that swaps the value of the k^{th} node with the value of the k^{th} node *from the end*, and then returns the head of this modified list. For this problem, the list is 1-indexed. For example, if you are given the following list with $k = 2$:



you would swap the 2nd value in the list (203) with the 2nd to last value in the list (281), as shown:

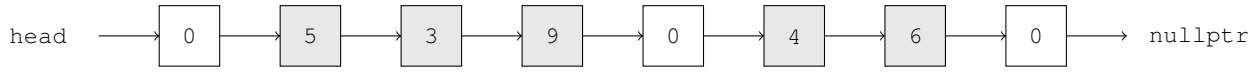


The function header is provided below:

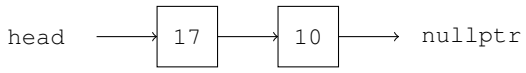
```
struct Node {
  int32_t val;
  Node* next;
  Node(int32_t val_in) : val{val_in}, next{nullptr} {}
};

Node* swap_nodes(Node* head, uint32_t k);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(1)$ auxiliary space, where n is the size of the list.

26. You are given the head of a singly-linked list, which contains a series of integers that separated by zeros. The first and last value in this list are guaranteed to be zero. Implement a function that, for every two consecutive zeros in the list, merges all the nodes in between the zeros into a single node whose value is the sum of the merged nodes. This new list is then returned. The modified list should not include any zeros; you may assume that the original list will not have two consecutive zeros. For example, given the following list:



you would return the following list (where the first node has a value of $5 + 3 + 9 = 17$, and the second node has a value of $4 + 6 = 10$):



The function header is provided below:

```
struct Node {
  int32_t val;
  Node* next;
  Node(int32_t val_in) : val{val_in}, next{nullptr} {}
};

Node* sum_nodes_between_zeros(Node* head);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(n)$ auxiliary space, where n is the size of the list. The solution list should be returned as a brand new list, so the original list passed into the input should not be modified.

Chapter 8 Exercise Solutions

1. **The correct answer is (B).** An `int32_t` has a size of 4 bytes. For a linked list of 47 integers, you have to store the Fibonacci number itself (with size 4 bytes) along with 2 pointers (8 bytes each). Hence, the total storage of 47 integers in a linked list is $47 \times (8 + 8 + 4)$, or 940 bytes. For an array, you only have to store the number and not the pointers, so an array of 47 numbers would take up 47×4, or 188 bytes. The difference is thus 940 - 177 = 752 bytes. You can also think of this in terms of the additional storage needed to store a value in a doubly-linked list compared to an array: in the linked list, each value would have to use 16 additional bytes of memory to store 2 pointers, so the extra memory required for the linked list of 47 values is 47×16 bytes.

2. **The correct answer is (D).** You cannot remove the last element in a singly-linked list in constant time, regardless of whether you have a tail pointer or not. This is because you have to set the `next` pointer of the node before the one being deleted to `nullptr`, which you cannot access in constant time if the list is singly-linked. Option (A) is true, because you need to visit every value in the list during a traversal, which takes $\Theta(n)$ since it takes constant time to visit each element. Option (B) is true if the element you want to find is the last one you encounter in the list (or if it does not exist in the list). Option (C) is correct because, if you are given a pointer to the element to insert at, you can set its internal pointers after the insertion without have to do a separate traversal.

3. **The correct answer is (B).** Appending an element to the back of an array, assuming no reallocation, takes constant time. Appending an element to a linked list takes constant time *only* if there is a tail pointer, which is not provided in this problem — without this tail pointer, you would have to perform a linear traversal to get to the point of insertion, which takes $\Theta(n)$ time.

4. **The correct answer is (D).** Without a tail pointer, insertion into the linked list still takes $\Theta(n)$ in the worst case, since you may traverse the entire list to find the point of insertion. However, since insertion is allowed at any point, the time complexity for the array also increases to $\Theta(n)$, not only to find the position of insertion, but to shift all subsequent elements to make space in the array.

5. **The correct answer is (D).** Since elements are stored contiguously in memory in an array, you can access an element at any index of an array in constant time using pointer arithmetic. This is not true for linked lists, which only support sequential access, so you would need to iterate to the index you want to access.

6. **The correct answer is (C).** The doubly-linked list is the only of the three containers that can support constant time insertion before any given object in the container. For the vector, you would have to shift elements over after the insertion, which could take linear time. For the singly-linked list, you would have to update the `next` pointer of the object before the point of insertion, which you cannot access in constant time from the given pointer (due to the absence of a `prev` pointer).

7. **The correct answer is (A).** Insertion at the back of a list takes $\Theta(n)$ time for both singly- and doubly-linked lists if no tail pointer is provided. Inserting an element after another node, when given the other node's pointer, takes $\Theta(1)$ time for both singly- and doubly-linked lists. Erasing an element at the back of the list takes $\Theta(n)$ time for both singly- and doubly-linked lists if no tail pointer is provided. Erasing an element when given its pointer takes $\Theta(1)$ time in a doubly-linked list but $\Theta(n)$ time in a singly-linked list, since there is no way to access the node before the one being deleted in constant time (you will need to update the `next` pointer of this node, but you do not have a `prev` pointer to get there in a singly-linked list).

8. **The correct answer is (A).** There are two main things that this container needs. First, you need a container that supports efficient insertion from one end and efficient removal from the other. Second, the size of the container can grow without bound, and thus cannot be fixed. From this, both the vector and fixed size array would not work, since removal from positions not at the end may take linear time. The doubly-linked list is also not ideal because it does not have a tail pointer, so it can only support constant time insertion and removal from the front. This leaves us with the singly-linked list with both head and tail pointers, which supports all the operations we need in constant time (we can append new students to the back and remove from the front, both of which take constant time with the tail pointer).

9. **The correct answer is (E).** None of the operations would be improved with the presence of a tail pointer. Reversing the linked list can be done in $\Theta(n)$ without the tail pointer (see section 8.4), and there is no way to improve this time complexity. Removing an element at the back takes linear time for a singly-linked list regardless of whether a tail pointer is supported or not, because you will need to update the `next` pointer of the node preceding the one that is deleted (which you cannot access in constant time for a singly-linked list). Finding an element in the list takes $\Theta(n)$ in the worst-case regardless of whether a tail pointer is present, since you might need to iterate over all the elements (which does not depend on the presence of a tail pointer). Removing the first element in the list takes $\Theta(1)$ time even without a tail pointer present.

10. **The correct answer is (B).** Finding the 10^{th} element in the singly-linked list requires iterating 9 positions from the head node, which takes constant time. Finding the 10^{th} to last element requires iterating $n-9$ positions from the head node, which takes linear time.

11. **The correct answer is (B).** Since `ptr` is the only way to access the queue, all insertions and removals from the queue must take place between Yvonne and Zach. Alice and Zach cannot be removed from the singly-linked list in constant time (to remove Zach, you would need to access Alice's `next` pointer, which requires a linear traversal of all the students in the queue). Thus, the only student that can be removed in constant time in Yvonne, so she must be the next in the queue if removing and adding students always takes constant time. This means that Zach is the last student in the queue. In fact, using this circular linked list setup, `ptr` points to the last student in the queue, since it takes constant time to insert a new student directly after this position.

12. **The correct answer is (D).** Four pointers are modified: `prev` of the new node is set to the previous node before the insertion position, `next` of the new node is set to the next node after the insertion position, `next` of the previous node is set to the new node, and `prev` of the next node is set to the new node.

13. **The correct answer is (D).** Deleting an element at the back of a singly-linked list takes $\Theta(n)$ time even if a tail pointer is present, since you need to update the `next` pointer of the node before the deleted node, which you cannot access in constant time.

14. **The correct answer is (D).** If the provided list has odd length but no cycle, then `fast` would end up referencing the last node in the list within the `while` loop on line 8. This causes `fast->next->next` on line 14 to produce undefined behavior, since `fast->next` is a `nullptr` that is dereferenced.

15. **The correct answer is (A).** This takes $\Theta(m+n)$ time and uses $\Theta(1)$ auxiliary space in the worst case. First, traverse the two linked lists to find the values of m and n. Then, go back to the heads of the linked list, and traverse $|m-n|$ nodes on the longer list. After this, iterate over the remaining nodes of the lists in lock step and compare the nodes until you find the first shared node.

16. **The correct answer is (C).** The time complexity of determining whether a doubly-linked list is circular is $\Theta(1)$, since you can just check if the `prev` of the head node is the last node of the list.

17. **The correct answer is (C).** Since the two lists are sorted, you can solve this problem in linear time by initializing two pointers to the beginning of the two lists, appending the smaller element of the two pointers to a new, merged list, and incrementing the pointer of the list whose element was just copied over. This is repeated until both pointers reach the end of their corresponding list.

18. **The correct answer is (C).** Unlike data in vectors, data in a linked list do not get reallocated as the container grows in size. This can be useful if you want to ensure the validity of pointers and references to objects in the list throughout the lifetime of the container. However, this comes at the cost of the other three options: linked lists take up more memory to store the same amount of data, does not provide fast sequential access (due to the inability to take advantage of caching, which allows contiguously memory to be accessed faster sequentially), and also does not provide $\Theta(1)$ random access.

19. **The correct answer is (D).** In the worst-case, you would have to visit all the values in the container before you find the one you want. This does not matter if the container is an array or singly-linked list: the traversal would take $\Theta(n)$ time.

20. **The correct answer is (A).** Only option (A) is true. Option (B) is false because deleting the last element in a singly-linked list takes $\Theta(n)$ time. Option (C) is false because finding an element takes worst-case $\Theta(n)$ time. Option (D) is false because reversing a doubly-linked list can be done in $\Theta(n)$ time regardless of whether a tail pointer is present.

21. **The correct answer is (A).** To splice one list into another, you just need to move some pointers around so that the nodes before and after the point of insertion (along with the end nodes of the list being spliced) are updated correctly. Since we have a doubly-linked list with tail pointers, accessing these nodes to update takes constant time, without needing to iterate over the remaining nodes of the two lists. Because of this, the splicing operation can be done in $\Theta(1)$ time, irrespective of the lengths of the two lists involved in the splice.

22. This problem is very similar to the original problem of reversing the entire linked list that we discussed in section 8.4. The only difference is that we only want to reverse a subsection of the list rather than the entire list. To accomplish this, we will follow the same logic as the optimized implementation in section 8.4.2, with the following additional steps:
    1. Before reversing, we will first traverse over the first `left` nodes. This can be done using a counter that keeps track of the number of positions we have iterated. In the following solution, we will use the values of `left` and `right` themselves as our counter variables.
    2. After reversing the nodes between `left` and `right`, we will attach the nodes afterward the new node at the position of `right`, which should be the node that was originally at the position of `left`.

    One implementation of this solution is shown below:

```
1   Node* reverse_list(Node* head, uint32_t left, uint32_t right) {
2     // null check
3     if (head == nullptr) {
4       return nullptr;
5     } // if
6
7     Node* prev = nullptr;
8     Node* curr = head;
9
10    // iterate forward "left" positions to get to the position of the first node to reverse
11    while (left > 1) {
12      prev = curr;
13      curr = curr->next;
14      --left;
15      --right;
16    } // while
17
18    // store node directly before the first position reversed
19    Node* node_before_reverse = prev;
20
21    // store first node to be reversed, will be final node in modified range after reversing
22    Node* first_node_reversed = curr;
23
24    // start reversing the linked list up to "right"
25    while (right > 0) {
26      Node* next = curr->next;
27      curr->next = prev;
28      prev = curr;
29      curr = next;
30      --right;
31    } // while
32
33    // update node before reversal position to point to first node of the reversed section
34    // if this node is nullptr, set head of entire reversed list to last node after reversing
35    if (node_before_reverse != nullptr) {
36      node_before_reverse->next = prev;
37    } // if
38    else {
39      head = prev;
40    } // else
41
42    // set the next of the last node of the reversed section to the remaining nodes of the list
43    first_node_reversed->next = curr;
44    return head;
45  } // reverse_list
```

23. Since the list is sorted, we can simply iterate over the list and check if there are two adjacent nodes that share the same value. If there are, simply point the next pointer of the first node to the next pointer of the second node (this essentially removes the second duplicate value from the list). One implementation of this solution is shown below:

```
Node* remove_duplicates(Node* head) {
  Node* curr = head;
  while (curr && curr->next) {
    // two duplicate values, remove the second duplicate from the list to return
    if (curr->val == curr->next->val) {
      curr->next = curr->next->next;
      continue;
    } // if
    // move forward with iterating over the list
    curr = curr->next;
  }

  return head;
} // reverse_list
```

24. The is the same as the linked-list cycle problem we solved in example 8.5 using Floyd's cycle-finding algorithm, with the added complexity of returning the node at which the cycle begins. To solve this problem, we can use the same implementation we had before. However, once the slow and fast pointers meet (which indicates the existence of a cycle), we will reset the slow pointer at the head of the original list and then increment both the slow and fast pointers at the same speed until they meet again. The node they meet at must be the first node in the cycle. An implementation is shown below:

```
Node* first_node_in_cycle(Node* head) {
  // use two pointers, where one moves faster than the other;
  // if the fast pointer catches up to the slow pointer.
  // then there exists a cycle
  Node* fast = head;
  Node* slow = head;
  while (fast && fast->next) {
    slow = slow->next;
    fast = fast->next->next;
    if (fast == slow) {
      slow = head;
      while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
      } // while
      return slow;
    } // if
  } // while

  // fast pointer reached end without meeting slow, so no cycle
  return nullptr;
} // first_node_in_cycle ()
```

25. There are two ways to approach this problem, both of which share a $\Theta(n)$ time complexity. The first approach is the simplest, as it traverses the list twice: during the first pass, we find the k^{th} node, and on the second pass, we find the k^{th} node from the end. An implementation of this simple solution is shown below:

```
Node* swap_nodes(Node* head, uint32_t k) {
  Node* curr = head;
  Node* kth = head;
  Node* kth_to_last = head;
  int32_t count = 0;
  while (curr != nullptr) {
    ++count;
    if (count == k) {
      kth = curr;
    } // if
    curr = curr->next;
  } // while

  curr = head;
  for (int32_t i = 0; i < count; ++i) {
    if (i == count - k) {
      kth_to_last = curr;
      break;
    } // if
    curr = curr->next;
  } // for i

  std::swap(kth->val, kth_to_last->val);
  return head;
} // swap_nodes()
```

Although it meets the time complexity requirements, this is actually not the most efficient solution. It turns out that you can solve this problem in only a single pass using the two pointer technique. To do so, we will keep track of a slow and fast pointer. The fast pointer will first be moved $k-1$ positions forward to reach the k^{th} node. We will keep track of this node. Then, we will follow the strategy described in example 8.3, incrementing both the slow and fast pointer in tandem until the fast pointer reaches the end. At this point, the slow pointer will be pointing the k^{th} to last node. Using the k^{th} node that we stored earlier, we can switch the two values and return the modified list. An implementation is shown below:

```
Node* swap_nodes(Node* head, uint32_t k) {
  Node* slow = head;
  Node* fast = head;
  for (int32_t i = 0; i < k - 1; ++i) {
    fast = fast->next;
  } // for i

  Node* kth = fast;
  while (fast->next) {
    slow = slow->next;
    fast = fast->next;
  } // while

  std::swap(kth->val, slow->val);
  return head;
} // swap_nodes()
```

26. To solve this problem, one solution is to traverse the list with a running counter that tracks the sum encountered so far in between zeros. Whenever a zero is encountered, the counter is reset, and the previous sum is appended to the list. One implementation is shown below:

```
Node* sum_nodes_between_zeros(Node* head) {
  Node* summed_list = nullptr;
  Node* new_head = nullptr;

  int32_t sum = 0;
  while (head != nullptr) {
    if (head->val == 0) {
      if (sum != 0) {
        ListNode* next_node = new ListNode{sum};
        // is the first node of the merged list
        if (summed_list == nullptr) {
          new_head = summed_list = next_node;
        } // if
        else {
          summed_list->next = next_node;
          summed_list = summed_list->next;
        } // else
      }
      // reset counter
      sum = 0;
    } // if
    else {
      sum += head->val;
    } // else

    head = head->next;
  } // while

  return new_head;
} // swap_nodes()
```