Java Data Structures 1st edition Azevedo Test Bank

SKU: 9780357114841-TEST-BANK

Chapter 2 Test Bank
Java Data Structures

1. When is an array to sort already sorted?

a. When an array contains only one item
b. When an array consists of the same element repeated
c. When an array is already sorted
d. When an array has a size of zero

Analysis:

a. Correct. An array of one element is already sorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
b. Incorrect. An array of duplicate elements is simply a single element multiplied and replicated. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
c. Incorrect. A sorted array does not need to be sorted using a sorting algorithm. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
d. Incorrect. An array of size zero has no elements to be sorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

2. When is the bubble sort Big-O O(n) linear?

a. The bubble sort is linear for a sorted data list
b. The bubble sort is linear with a data size of zero elements
c. The bubble sort is linear when all the elements in the data list are the same
d. The bubble sort is linear when the data list is sorted in reverse

Analysis:

a. Correct. The bubble sort has O(n) performance when the data is already sorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
b. Incorrect. With a data list of zero elements, there is no need or data to sort. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
c. Incorrect. The bubble sort will be linear in this case, but having the same elements is not necessarily going to assure a linear performance of O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
d. Incorrect. In such a case, the bubble sort will check that the list is not sorted upwards and then fully sort the data list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

3. When is the bubble sort algorithm possible to use in practice?

a. When the data is partially sorted
b. When the data is completely sorted
c. When the data list has no duplicate elements
d. When the data list has 0 elements

Analysis:

a. Correct. When the data list is partially sorted, the bubble sort only needs to make one or two passes to sort the already partially sorted list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
b. Incorrect. If the data list is sorted, we don't need to use the bubble sort. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
c. Incorrect. The data list can have duplicates. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
d. Incorrect. In such a case, there would be no data to sort. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

4. How is partitioning important in the quick sort algorithm?

a. Partitioning divides the unsorted input data list into smaller unsorted data lists
b. Partitioning is the process of removing duplicates from an array
c. Partitioning is the process of walking through an array
d. Partitioning is the process of finding the splitting point of an array

Analysis:

a. Correct. Partitioning divides the input data list of elements into two sublists, and this process continues. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
b. Incorrect. Partitioning splits an array into two smaller arrays. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. Partitioning walks through an array, but in order to divide an array into two smaller arrays. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
d. Incorrect. The partition element is used to split an array. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

5. What is the stopping condition of the quick sort algorithm?

a. When no elements can be moved
b. When the data input list is sorted
c. When the end of the array is reached
d. When the array is of zero length

Analysis:

a. Correct. The quick sort stops recursively partitioning when the pivot element does not need to move. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
b. Incorrect. Any sorting algorithm will stop when data is sorted, but the quick sort does so when the pivot element is no longer moving data. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. The quick sort uses the pivot element to move data, and this continues on each smaller array recursively. Reaching the end of an array is not used to indicate the end of the quick sort. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
d. Incorrect. The quick sort stops recursively partitioning when the pivot element does not need to move. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

6. How is the worst case Big-O for the quick sort determined?

a. It is determined by the partitioning system used
b. The worst case for quick sort is always the same
c. It is determined by the input data list
d. It is when there are duplicate elements in the data list

Analysis:

a. Correct. The worst case Big-O for the quick sort is determined by the partitioning system used. For example, the Lomuto scheme has a worst case for when the data list is already sorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
b. Incorrect. The use of a different partitioning scheme can affect the performance of the quick sort. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. The quick sort's worst case is determined by how the quick sort recursively partitions and chooses a pivot element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
d. Incorrect. The existence of duplicate elements does not affect the scheme of choosing a pivot and partitioning in the quick sort algorithm. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

7. How can the quick sort avoid the worst possible Big-O complexity?

a. By checking if the array is already sorted
b. This is impossible
c. By removing duplicate elements
d. By not using a pivot element

Analysis:

a. Correct. Similar to a bubble sort, we can add an initial pass to check if the array is sorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
b. Incorrect. Checking if the data list is sorted makes this possible. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. Duplicate elements do not create the worst case Big-O complexity of O(n^2). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
d. Incorrect. A pivot element is how the quick sort operates. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

8. Can the quick sort use the merge sort to compute the midpoint of an array?

a. No
b. Yes
c. Yes, if the array is already sorted
d. Yes, for an array without duplicates

Analysis:

a. Correct. The quick sort cannot use the merge sort to compute the midpoint of an array. The quick sort algorithm needs a pivot element in order to partition. Simply dividing the array is not partitioning. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
b. Incorrect. The quick sort requires a pivot element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. If the array is already sorted, we do not need to use a sorting algorithm. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
d. Incorrect. Duplication of data elements in a list is irrelevant. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

9. What is the Big-O of the partition function of the quick sort algorithm?

a. O(n)
b. O(log n)
c. O(n^2)
d. O(1)

Analysis:

a. Correct. The partition function traverses the array for the given size, and swaps elements in O(3). Thus, the Big-O is O(3n) + O(3) for the final swap, which equates to O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

b. Incorrect. The partition function has a Big-O of O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. The partition function has a Big-O of O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
d. Incorrect. The partition function has a Big-O of O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

10. What is the Big-O advantage of the merge sort algorithm?

a. The merge sort even in the worst case has a Big-O of O(n log n)
b. The merge sort has a Big-O of O(n)
c. The merge sort is much more memory efficient than the quick sort
d. The merge sort chooses a pivot element more efficiently

Analysis:

a. Correct. The merge sort avoids the problem of sorting on a sorted list by dividing the input data list by computing the split point and not using a pivot element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
b. Incorrect. The Big-O of merge sort is never O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
c. Incorrect. The merge sort has no great advantage related to memory use over the quick sort. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
d. Incorrect. The merge sort does not use a pivot element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.

11. How does the merge sort algorithm split an array of elements?

a. It uses the size of the array to divide the array into two smaller arrays
b. It uses a pivot element
c. It splits an array in an arbitrary fashion
d. It uses the average of the elements to compute the midpoint

Analysis:

a. Correct. The merge sort computes the mid-point of an input data array. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
b. Incorrect. The merge sort does not use a pivot element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
c. Incorrect. The merge sort does not split an array in an arbitrary fashion. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
d. Incorrect. The merge sort uses the array bounds to compute the midpoint of the array. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.

12. Can the merge sort use the partition function of the quick sort?

a. No
b. Yes
c. Yes, if the array does not have any duplicates
d. Yes, if the array is reverse sorted

Analysis:

a. Correct. The merge sort divides the array of elements, but does not move elements from each smaller array. Thus the merge sort cannot use the quick sort partition function. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
b. Incorrect. The merge sort requires a merge function. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
c. Incorrect. Having duplicate elements in the array is irrelevant. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
d. Incorrect. The starting configuration or ordering of the data elements in the list is irrelevant. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.

13. What is the Big-O performance of the merge function in the merge sort?

a. O(n)
b. O(log n)
c. O(n^2)
d. O(1)

Analysis:

a. Correct. The merge function copies the array into a temp array in O(n), merges the array in linear time O(n), and then copies from the temp array to the array. This is O(n) + O(n) + O(n) = O(3n) or O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
b. Incorrect. The merge function has a Big-O of O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
c. Incorrect. The merge function has a Big-O of O(n). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.
d. Incorrect. A temp array and the array are merged, but in linear O(n) time. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.

14. What is an advantage of a linked list data structure?

a. It is more efficient compared to others
b. The linked list uses links

c. The linked list uses an array
d. The linked list does not have duplicate elements

Analysis:

a. Correct. Linked lists are memory efficient as the data structure grows with new data. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. Links are used by a linked list to interconnect list elements, but that is neither an advantage nor a disadvantage. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The linked list does not use an array data structure. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. A linked list can contain duplicate elements. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

15. How does a search operation work in the linked list?

a. It requires traversing the entire list from the head to the tail of the list.
b. The linked list is searched starting from the head.
c. The linked list is searched starting from the tail.
d. The linked list stores the most recent element to search for at the front or head of the list.

Analysis:

a. Correct. The entire linked list must be traversed in order to find an element theoretically. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. The search operation in a linked list can start at the head or tail, or even from the middle of the list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The search operation in a linked list can start at the head or tail, or even from the middle of the list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. The search operation in a linked list can start at the head or tail, or even from the middle of the list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

16. Why does a linked list have so many operations to insert an element into the list?

a. The linked list must maintain the links for traversal through the list.
b. The linked list must avoid duplicate elements.
c. The linked list must perform a search operation when adding an element to the list.
d. The linked list must update the head and tail of the list.

Analysis:

a. Correct. The element is added and the links between the two nodes must be modified. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. The linked list does not avoid duplicate elements during the insertion operation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The linked list does not perform a search operation when inserting an element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. Unless an element is inserted at the head or tail, neither is modified by an insert operation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

17. How is a queue more efficient than a linked list data structure?

a. The queue data structure operations are all at the head and tail of the list.
b. A queue is more efficient than a linked list because it does not have duplicate elements.
c. A queue is more efficient because it uses a single linked list.
d. A queue sorts the elements so they can be easily found.

Analysis:

a. Correct. A queue is more efficient because operations of enqueue and dequeue are at the head and tail of the list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. A queue can have duplicate elements. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. A queue more efficiently uses a double linked list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures
d. Incorrect. A queue does not sort the elements nor does it search. See Module 2: Sorting .Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

18. What is the advantage of modeling a queue or stack with an array?

a. An array implementation is more memory efficient and avoids the added cost of links in a linked list data structure.
b. An array can be searched.
c. An array has O(1) constant performance.
d. An array can be sorted.

Analysis:

a. Correct. Modeling a queue or stack with an array guarantees the data structure will grow to a fixed size, and is more efficient. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. Neither a queue nor a stack requires a search operation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. Each operation to access an element has O(1) performance. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. There is no need for sorting in a queue or stack data structure. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

19. Why is a one-way linked list used instead of a two-way linked list for a stack data structure?

a. Because a stack operates on only one end of the data structure.
b. Because a single linked list is easier to implement.
c. Because a stack has a FIFO ordering.
d. Because a stack has no duplicate elements.

Analysis:
a. Correct. Since a stack operates on one end of the data structure, a single linked list can be used. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. A double linked list could be used, but a single linked list is used as a stack has only one point of operation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The queue has a FIFO ordering. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. A stack can have duplicate elements. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

20. Why is a double linked list used for a queue data structure?

a. Because a queue enqueues and dequeues elements at both ends.
b. Because a double linked list is simpler to implement.
c. For a queue to have LIFO ordering.
d. To make a queue searchable.

Analysis:
a. Correct. A queue operates on both ends of the data structure, so a double linked list is most efficient. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

b. Incorrect. A double linked list is not simpler to implement than single linked list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. A stack has a LIFO ordering. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. A queue does not have search capability. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

21. Can a queue be implemented with a single linked list?

a. Yes
b. No
c. Yes, by not allowing duplicates in the single linked list queue
d. Yes, using an array along with a single linked list

Analysis:

a. Correct. A queue can be implemented with a single linked list by connecting the head and tail into a ring; the enqueue and dequeue operation is at the same point, with elements going around back to the start of the ring. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. A queue can be implemented with a single linked list by connecting the head and tail into a ring; the enqueue and dequeue operation is at the same point, with elements going around back to the start of the ring. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. A queue can have duplicate elements. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. A queue can be implemented with only a single linked list by connecting the head and tail into a ring; the enqueue and dequeue operation is at the same point, with elements going around back to the start of the ring. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

22. Can a stack be used to implement a queue data structure?

a. Yes.
b. Yes, if the stack is implemented using a double linked list.
c. No.
d. Yes, if the stack has an array.

Analysis:

a. Correct. Yes. A stack can implement a queue; the stack is "flipped" during operation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

b. Incorrect. It doesn't matter how the stack is implemented, so long as it can be "flipped." See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. A stack can implement a queue using the "flip" operation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. An array is unnecessary in this case. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

23. Which linked list is more memory efficient?

a. Single linked list.
b. Coupled linked list.
c. Double linked list.
d. Both single and double-linked lists are equally efficient.

Analysis:

a. Correct. The single linked list is more memory efficient because it only uses one link instead of two to connect with another data element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. There is no such thing as a coupled linked list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The double linked list uses twice the memory for two links. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. The single linked list is more memory efficient because it only uses one link instead of two to connect with another data element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

24. What sorting algorithm is implemented by the following function?

```
public class another {
   static void sort(int[] arr) {
      int n = arr.length;
      int temp = 0;
      for(int i=0; i < n; i++){
         for(int j=1; j < (n-i); j++){
               if(arr[j-1] > arr[j]){
               temp = arr[j-1];
               arr[j-1] = arr[j];
```

```
                arr[j] = temp;
            }
        }
    }
}
```

a. Bubble sort.
b. Quick sort.
c. Selection sort.
d. Merge sort.

Analysis:

a. Correct. This is an implementation of bubble sort (we are comparing values next to each other and changing their place if necessary). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
b. Incorrect. Quick sort is a divide and conquer algorithm. It first divides a large array into two smaller subarrays: the low elements and the high elements. It then recursively sorts the subarrays. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
c. Incorrect. Selection sort divides the input list into two parts: the sublist of items already sorted (built up from left to right at the front (left) of the list) and the sublist of items remaining to be sorted (that occupy the rest of the list). Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
d. Incorrect. Conceptually, a merge sort works as follows: it divided the unsorted list into n sublists, each containing one element (a list of one element is considered sorted). It repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining. This is the sorted list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

25. What sorting algorithm is implemented by the following function?

```
void sort(int arr[])
{
    int n = arr.length;
    for (int i = 0; i < n-1; i++)
    {
        int min_idx = i;
```

```
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

a. Selection sort.
b. Bubble Sort.
c. Quick sort.
d. Merge Sort.

Analysis:

a. Correct. Selection sort divides the input list into two parts: the sublist of items already sorted (built up from left to right at the front (left) of the list) and the sublist of items remaining to be sorted (that occupy the rest of the list). Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
b. Incorrect. In bubble sort, we compare values next to each other and change their place if necessary. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
c. Incorrect. Quick sort is a divide and conquer algorithm. It first divides a large array into two smaller subarrays: the low elements and the high elements. It then recursively sorts the subarrays. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
d. Incorrect. Conceptually, a merge sort works as follows: it divided the unsorted list into n sublists, each containing one element (a list of one element is considered sorted). It repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining. This is the sorted list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

26. What is the complexity of the bubble sort algorithm in the best case scenario?

a. O(n)
b. O(n^2)
c. O(1)
d. O(log(n))

Analysis:

a. Correct. We need to pass through all elements in the array to check if no change is necessary. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
b. Incorrect. We need to pass through all elements in the array to check if no change is necessary. No additional passes are necessary. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
c. Incorrect. We need to pass through all elements in the array to check if no change is necessary. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
d. Incorrect. We need to pass through all elements in the array to check if no change is necessary. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

27. What is the complexity of the bubble sort algorithm in the worst case scenario?

a. O(n^2)
b. O(n)
c. O(2^n)
d. O(log(n))

Analysis:

a. Correct. For an unsorted set of data, bubble sort is not an efficient solution. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
b. Incorrect. In the worst case scenario, bubble sort's complexity is O(n^2). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
c. Incorrect. In the worst case scenario, bubble sort's complexity is O(n^2). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
d. Incorrect. In the worst case scenario, bubble sort's complexity is O(n^2). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

28. What is the complexity of the selection sort algorithm in the best case scenario?

a. O(n^2)
b. O(1)
c. O(n)
d. O(log (n))

Analysis:

a. Correct. Selection sort has O(n^2) complexity; it doesn't matter if the input is sorted or unsorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

b. Incorrect. Selection sort has O(n^2) complexity; it doesn't matter if the input is sorted or unsorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
c. Incorrect. Selection sort has O(n^2) complexity; it doesn't matter if the input is sorted or unsorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.
d. Incorrect. Selection sort has O(n^2) complexity; it doesn't matter if the input is sorted or unsorted. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.1: Introducing Bubble Sorting.

29. What sorting algorithm is implemented by the following function?

```
public class another {
   public void sort(int arr[], int begin, int end) {
      if (begin < end) {
         int partitionIndex = partition(arr, begin, end);
         sort(arr, begin, partitionIndex-1);
         sort(arr, partitionIndex+1, end);
      }
   }
}
```

a. Quick sort
b. Bubble Sort
c. Merge sort
d. Selection sort

Analysis:

a. Correct. This is an implementation of the quick sort algorithm. Quick sort is a divide-and-conquer algorithm. It first divides a large array into two smaller subarrays: the low elements and the high elements. It then recursively sorts the subarrays. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
b. Incorrect. In bubble sort, we compare values next to each other and change their place if necessary. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. Conceptually, a merge sort works as follows: it divided the unsorted list into n sublists, each containing one element (a list of one element is considered sorted). It repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining. This is the sorted list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
d. Incorrect. Selection sort divides the input list into two parts: the sublist of items already sorted (built up from left to right at the front (left) of the list) and the sublist of items remaining to be sorted (that occupy the rest of the list). Initially, the sorted sublist is empty and the unsorted

sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

30. What is the complexity of the quick sort algorithm in the average case scenario?

a. O(n log(n))
b. O(n^2)
c. O(n)
d. O(log(n))

Analysis:

a. Correct. Mathematical analysis of quick sort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
b. Incorrect. Mathematical analysis of quick sort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. Mathematical analysis of quick sort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
d. Incorrect. Mathematical analysis of quick sort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

31. What is the complexity of the quick sort algorithm in the worst case scenario?

a. O(n^2)
b. O(n log(n))
c. O(n)
d. O(log(n))

Analysis:

a. Correct. Mathematical analysis of quicksort shows that, in the worst case scenario, it makes O(n^2) comparisons, though this behavior is rare. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
b. Incorrect. Mathematical analysis of quicksort shows that, in the worst case scenario, it makes O(n^2) comparisons, though this behavior is rare. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.
c. Incorrect. Mathematical analysis of quicksort shows that, in the worst case scenario, it makes O(n^2) comparisons, though this behavior is rare. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

d. Incorrect. Mathematical analysis of quicksort shows that, in the worst case scenario, it makes O(n^2) comparisons, though this behavior is rare. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.2: Understanding the Quick Sort.

32. What sorting algorithm is implemented by the following function?

```
public class another {
   public static void sort(int[] a, int n) {
      if (n < 2) {
         return;
      }
      int mid = n / 2;
      int[] l = new int[mid];
      int[] r = new int[n - mid];
      for (int i = 0; i < mid; i++) {
         l[i] = a[i];
      }
      for (int i = mid; i < n; i++) {
         r[i - mid] = a[i];
      }
      sort(l, mid);
      sort(r, n - mid);
      concat(a, l, r, mid, n - mid);
   }
}
```

a. Merge sort
b. Bubble sort
c. Quick sort
d. Selection sort

Analysis:

a. Correct. Conceptually, a merge sort works as follows: it divided the unsorted list into n sublists, each containing one element (a list of one element is considered sorted). It repeatedly merges sublists to produce new sorted sublists until there is only one sublist

remaining. This is the sorted list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.

b. Incorrect. In bubble sort, we compare values next to each other and change their place if necessary. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.

c. Incorrect. Quick sort is a divide and conquer algorithm. It first divides a large array into two smaller subarrays: the low elements and the high elements. It then recursively sorts the subarrays. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.

d. Incorrect. Selection sort divides the input list into two parts: the sublist of items already sorted (built up from left to right at the front (left) of the list) and the sublist of items remaining to be sorted (that occupy the rest of the list). Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.3: Using the Merge Sort.

33. What is the operational complexity of adding an element to a linked list?

a. O(n)
b. O(1)
c. O(log(n))
d. O(n^2)

Analysis:

a. Correct. We need to traverse the entire list (n elements) to add an element at the end. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

b. Incorrect. We need to traverse the entire list (n elements) to add an element at the end. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

c. Incorrect. We need to traverse the entire list (n elements) to add an element at the end. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

d. Incorrect. We need to traverse the entire list (n elements) to add an element at the end. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

34. What is the operational complexity of removing an element from a linked list?

a. O(1)
b. O(n)
c. O(log(n))
d. O(n^2)

Analysis:

a. Correct. In a linked list, we remove elements from the beginning, which takes O(1). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. In a linked list, we remove elements from the beginning, which takes O(1). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. In a linked list, we remove elements from the beginning, which takes O(1). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. In a linked list, we remove elements from the beginning, which takes O(1). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

35. What does the following function do for a linked list with the first node as head?

```
public void fun1(Node head)
{
  if(head == null)
    return;
  fun1(head.next);
  System.out.println(head.data);
}
```

a. Print all elements in reverse order.
b. Print all elements in normal order.
c. Prints all elements in random order.
d. Prints the first element.

Analysis:

a. Correct. This is a recursive function, which prints data stored in a given node and then moves on to the next one. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. This is a recursive function, which prints data stored in a given node and then moves on to the next one. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. This is a recursive function, which prints data stored in a given node and then moves on to the next one. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. This is a recursive function, which prints data stored in a given node and then moves on to the next one. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

36. What is the complexity of concatenating two single linked lists?

a. O(n)
b. O(1)
c. O(n^2)
d. O(log(n))

Analysis:

a. Correct. We need to traverse the first list to find the last element and add a pointer to the second list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. We need to traverse the first list to find the last element and add a pointer to the second list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. We need to traverse the first list to find the last element and add a pointer to the second list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. We need to traverse the first list to find the last element and add a pointer to the second list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

37. What is the complexity of concatenating a single linked list of length n to a single linked list of length m (m > n)?

a. O(m)
b. O(n)
c. O(1)
d. O(n+m)

Analysis:

a. Correct. We need to traverse the first list (length m) to find the last element and add a pointer to the second list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. We need to traverse the first list (length m) to find the last element and add a pointer to the second list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. We need to traverse the first list (length m) to find the last element and add a pointer to the second list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. We need to traverse the first list (length m) to find the last element and add a pointer to the second list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

38. In a doubly linked list, what is the number of pointers affected by an insertion operation?

a. It can't be determined
b. 1
c. 0
d. 4

Analysis:

a. Correct. It depends on where we are going to insert the element (beginning, end, or middle). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. It depends on where we are going to insert the element (beginning, end, or middle). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. It depends on where we are going to insert the element (beginning, end, or middle). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. It depends on where we are going to insert the element (beginning, end, or middle). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

39. In a doubly linked list, what is the number of pointers affected by an insertion operation at the beginning of the list?

a. 1
b. 2
c. 0
d. 4

Analysis:

a. Correct. We need to change the 'previous' pointer of the first element; the rest remains the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. We need to change the “previous” pointer of the first element; the rest remains the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. We need to change the “previous” pointer of the first element; the rest remains the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. We need to change the “previous” pointer of the first element; the rest remains the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

40. In a doubly linked list, what is the number of pointers affected by an insertion operation at the end of the list?

a. 1
b. 2
c. 0
d. 4

Analysis:

a. Correct. We need to change the “next” pointer of the last element; the rest remains the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. We need to change the “next” pointer of the last element; the rest remains the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. We need to change the “next” pointer of the last element; the rest remains the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. We need to change the “next” pointer of the last element; the rest remains the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

41. In a doubly linked list, what is the number of pointers affected by an insertion operation in the middle of the list?

a. 2
b. 1
c. 0
d. 4

Analysis:

a. Correct. We need to change the “previous” pointer of the element next to the new one, and the “next” pointer of the preceding element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. We need to change the “previous” pointer of the element next to the new one, and the “next” pointer of the preceding element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. We need to change the “previous” pointer of the element next to the new one, and the “next” pointer of the preceding element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. We need to change the “previous” pointer of the element next to the new one, and the “next” pointer of the preceding element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

42. Which of the following statements is true for a doubly linked list?

a. It has at least two pointers in each node.
b. It has a backup of the data.
c. It stores the elements in a sorted manner.
d. It has twice the number of nodes as a normal linked list.

Analysis:

a. Correct. That is true. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. That is not true for a doubly linked list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. That is not true for a doubly linked list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. That is not true for a doubly linked list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

43. In a single linked list, what is the number of pointers affected by an insertion operation?

a. It can't be determined
b. 2
c. 1
d. 0

Analysis:

a. Correct. It depends on where we are going to insert the element (beginning, end, or middle). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. It depends on where we are going to insert the element (beginning, end, or middle). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. It depends on where we are going to insert the element (beginning, end, or middle). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. It depends on where we are going to insert the element (beginning, end, or middle). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

44. In a single linked list, what is the number of pointers affected by an insertion operation in the middle of the list?

a. 1
b. 0
c. 2

d. 4

Analysis:

a. Correct. We need to change the “next” pointer of the preceding node. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. We need to change the “next” pointer of the preceding node. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. We need to change the “next” pointer of the preceding node. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. We need to change the “next” pointer of the preceding node. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

45. In a single linked list, what is the number of pointers affected by an insertion operation at the end of the list?

a. 1
b. 0
c. 2
d. 4

Analysis:

a. Correct. We need to change the “next” pointer of the last element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. We need to change the “next” pointer of the last element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. We need to change the “next” pointer of the last element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. We need to change the “next” pointer of the last element. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

46. In a single linked list, what is the number of pointers affected by an insertion operation at the beginning of the list?

a. 0
b. 1
c. 2

d. 4

Analysis:

a. Correct. We just point “next” of the new element to the beginning of the list. All of the other pointers in the list remain the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. We just point “next” of the new element to the beginning of the list. All of the other pointers in the list remain the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. We just point “next” of the new element to the beginning of the list. All of the other pointers in the list remain the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. We just point “next” of the new element to the beginning of the list. All of the other pointers in the list remain the same. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

47. Consider an implementation of a linked list sorted in an ascending manner. What is the complexity of retrieving the lowest value from the list?

a. O(1)
b. O(n)
c. O(log n)
d. It can't be determined

Analysis:

a. Correct. The lowest value is at the beginning of the list, so we need to return the head. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. The lowest value is at the beginning of the list, so we need to return the head. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The lowest value is at the beginning of the list, so we need to return the head. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. The lowest value is at the beginning of the list, so we need to return the head. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

48. Consider an implementation of a linked list sorted in an ascending manner. What is the complexity of retrieving the highest value from the list?

a. O(n)
b. O(1)
c. O(log n)

d. It can't be determined

Analysis:

a. Correct. The highest value is at the end of the list, so we need to traverse the entire list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. The highest value is at the end of the list, so we need to traverse the entire list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The highest value is at the end of the list, so we need to traverse the entire list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. The highest value is at the end of the list, so we need to traverse the entire list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

49. Consider an implementation of a linked list sorted in an ascending manner. What is the complexity of retrieving the median value from the list?

a. O(n)
b. O(1)
c. O(log n)
d. It can't be determined

Analysis:

a. Correct. As long as we don't know how long the list is, we need to traverse it entirely and then return the element in the middle. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. As long as we don't know how long the list is, we need to traverse it entirely and then return the element in the middle. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. As long as we don't know how long the list is, we need to traverse it entirely and then return the element in the middle. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. As long as we don't know how long the list is, we need to traverse it entirely and then return the element in the middle. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

50. What is the postfix representation of the following expression?

((A + B) * (C + E))

a. A B + C E + *
b. *+AB +CE

c. ABCE++*
d. ABCE*++

Analysis:

a. Correct. That is how the given expression would look in the postfix notation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. This is the prefix expression. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. This is a postfix expression, but it can't be evaluated to ((A + B) * (C + E)) in infix. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. This is a postfix expression, but it can't be evaluated to ((A + B) * (C + E)) in infix. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

51. What is a postfix notation?

a. It's a notation used to write arithmetic operations, where the operator is written after the operands.
b. It's a notation used to write arithmetic operations, where the operator is written before the operands.
c. It's a notation used to write arithmetic operations, where the operator is written between the operands.
d. It's a notation used to express algorithm complexity.

Analysis:

a. Correct. That is the definition of postfix notation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. That is the definition of prefix notation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. That is the definition of infix notation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. It is not related to algorithm complexity at all. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

52. What is the time required to pick an element from a queue built up on a linked list of length n?

a. O(n)
b. O(1)
c. O(log n)
d. It can't be determined

Analysis:

a. Correct. The element is at the end of the list, so we need to traverse it entirely. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. The element is at the end of the list, so we need to traverse it entirely. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The element is at the end of the list, so we need to traverse it entirely. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. The element is at the end of the list, so we need to traverse it entirely. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

53. What is the infix representation of the following expression?

AX B C * +

a. (AX + (B * C))
b. A*X +BC
c. A+XB*C
d. BC + A * X

Analysis:

a. Correct. This is what we will get after transition to infix notation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. After transition to infix notation, we will get: (AX + (B * C)). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. After transition to infix notation, we will get: (AX + (B * C)). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. After transition to infix notation, we will get: (AX + (B * C)). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

54. What is the postfix representation of the following expression?

(A + B) * C

a. A B + C *
b. ABC+*
c. ABC*+
d. AB*C+

Analysis:

a. Correct. This is the postfix representation of (A+B) * C. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. The postfix representation of (A+B) * C is different. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The postfix representation of (A+B) * C is different. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. The postfix representation of (A+B) * C is different. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

55. What is the infix representation of the following expression?

A B C * + D +

a. A + B * C + D
b. (A + B) * C + D
c. (A + B) * (C + D)
d. A * B + C + D

Analysis:

a. Correct. This is what we will get after transition to infix notation. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. After transition to infix notation, we will get: A + B * C + D. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. After transition to infix notation, we will get: A + B * C + D. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. After transition to infix notation, we will get: A + B * C + D. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

56. What does it mean when we say a sorting algorithm is “stable”?

a. It means the algorithm does not change the order of elements if they are equal (not needed to be replaced)
b. It means that the algorithm has the same complexity in all scenarios
c. It means that the algorithm always works
d. None of the above

Analysis:

a. Correct. That is the definition of ”stability.” See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. The term “stable” relates to the order of elements in the sorted output. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. An algorithm should work for any kind of data, by definition. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. There is a correct option here. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

57. What does it mean when we say a sorting algorithm is performed “in-place”?

a. It doesn't need any additional memory to perform operations
b. It needs additional memory to perform operations
c. It means that the complexity of the algorithm is O(1)
d. It means that the algorithm has a “sibling” that has exactly the same complexity

Analysis:

a. Correct. We only need the memory allocated for the data structure passed to the function. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. This term means something opposite. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. It would be great to have such a sorting algorithm; unfortunately it has not been yet invented. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. There is no such thing as a “sibling algorithm” and “in-place” means something different. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

58. Which of the following algorithms is the best in the worst case scenario?

a. Merge sort
b. Insertion sort
c. Quick sort
d. Bubble sort

Analysis:

a. Correct. Complexity is O(n log(n)) for best and worst case scenario. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

b. Incorrect. Complexity is O(n^2) in worst case scenario and O(n) in best case scenario. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. Complexity is O(n log(n)) in best case scenario and O(n^2) in worst case scenario. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. Complexity is O(n^2) in worst case scenario and O(n) in best case scenario. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

59. Which of the following is the correct output for the expression: 1 2 + 3 * ?

a. 9
b. 7
c. 5
d. 6

Analysis:

a. Correct. 1 2 + 3 * can be evaluated to (1+2)*3 that equals 9. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. 1 2 + 3 * can be evaluated to (1+2)*3 that equals 9. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. 1 2 + 3 * can be evaluated to (1+2)*3 that equals 9. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. 1 2 + 3 * can be evaluated to (1+2)*3 that equals 9. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

60. What is a data structure?

a. It is an organization of data elements and collection of functions that can be applied on data.
b. It's a way to store data on the hard drive.
c. It's all the data to be loaded into a program.
d. It's a way to feed a program with data.

Analysis:

a. Correct. This is the correct definition of a data structure. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. A data structure is not related to data storing. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

c. Incorrect. A data structure is a way of organizing data in a program. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. A data structure is not related to the manner of data input. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

61. What are the functions that can be applied on the common data structures?

a. add, delete, update, search, and size
b. sort and random
c. add
d. enqueue and dequeue

Analysis:

a. Correct. These are the basic functions implemented by the common data structures. Sometimes, they are called differently (for example, enqueue and dequeue instead add and delete). See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. Why would we need the sort and random methods if we are not able to put anything into our data structure? See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. A data structure in which you can only store data and not retrieve; it wouldn't be very useful. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. These methods are specific for queues. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

62. What is a “tail” in a linked list?

a. It is the last node in a data structure (which doesn't point to “next”)
b. It is the node with the lowest value in the list
c. It is the node with the highest value in the list
d. It is the first node in a data structure

Analysis:

a. Correct. This is exactly what we call “tail”. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. A “tail” is, in other words, the end of the list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. A “tail” is, in other words, the end of the list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. A “tail” is, in other words, the end of the list. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

63. What is a queue?

a. It's an abstract data structure meant to emulate a real life queue.
b. It's an abstract data structure that does not accept duplicated values.
c. It's a data structure meant to emulate a real life stack (last in first out).
d. None of the above.

Analysis:

a. Correct. That is the queue definition. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. That's the definition of a set. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. That's the definition of a stack. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. There is a correct answer here. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

64. Which of the following is an advantage of implementing a queue using an array?

a. It is more efficient if the exact size of the input is known upfront.
b. It prioritizes the elements in the queue.
c. It does not have to keep a head and a tail pointer.
d. It is not restricted by a static size.

Analysis:

a. Correct. That is an advantage. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. That is not an advantage. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. That is not an advantage. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. That is not an advantage. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

65. What is FIFO?

a. First In First Out – queue input/output behavior.
b. Such a term doesn't exist in programming.
c. First In First Out – stack input/output behavior.
d. It's a way how tasks are queued to be picked up by the processor.

Analysis:

a. Correct. That's exactly what FIFO is. Regardless of what was added at the end, we always pick up values that were added first. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. FIFO is the name of an often-used data structure mechanism. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The abbreviation is correct, but it's not related to stack behavior. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. Most processors use a prioritized queue for queueing tasks. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

66. Bubble sort is generally faster than quick sort.

a. FALSE
b. TRUE
c. They have equal complexity
d. It can't be determined

Analysis:

a. Correct. Quick sort is a faster algorithm. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
b. Incorrect. Quick sort is a faster algorithm. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. Quick sort is more efficient. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. Complexity of both these algorithms can be determined. Quick sort is more efficient. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

67. What is the expression "1 2 3 * +" an example of?

a. Postfix expression
b. Infix expression
c. Prefix expression
d. None of the above

Analysis:

a. Correct. This is a postfix notation example, where the operators are written after the operands. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.

b. Incorrect. The infix notation is what we use to write mathematical expressions, for example, 1+2*3. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
c. Incorrect. The prefix notation requires that the operators are written before the operands. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.
d. Incorrect. There is a correct answer here. See Module 2: Sorting Algorithms and Fundamental Data Structures, Lesson 2.4: Getting Started with Fundamental Data Structures.