DSA Tutorial Week 1 - Collections

Find All Source Code on the DSA Tutorial Github: [DSA-Tutorial-Exercises]

Problem 0 - Preparation *!examine before the tutorial!*

Download the accompanying code for this week's tutorial. Open the LinkedList.java file. This is an incomplete implementation of a LinkedList. If you feel unsure of how a LinkedList is supposed to be implemented, then follow the steps below in order to finish the unfinished version provided.

1. **Complete the add Method:**
   Implement the logic to create a new Node<E> with the provided generic data and attach it to the end of the list.

   Steps:
   A. Check if the list is empty (i.e., **head** is **null**). If it is, create a new **Node<E>** with the given data and set it as the head.
   B. If the list is not empty, define a temporary **Node<E>** variable named current and initialize it with the head of the list.
   C. Traverse the list until you find the last node (when **current.next** is null).
   D. Create a new **Node<E>** with the provided data.
   E. Set the **next pointer** of the current last node to this new node.

2. **Implement the remove Method:**
   Implement logic to traverse the list and find the node with the specified generic data. Consider edge cases as mentioned previously.

   Steps:
   A. Check if the list is empty. If it is, there's nothing to remove.
   B. If the data to be removed is at the head (i.e., **head.data.equals(data)**), update the head to be **head.next**.
   C. If the data is not at the head, define two temporary variables: **current** and **previous**. Initialize **current** with the head and **previous** with null.
   D. Traverse the list until you find the node containing the data. (Perhaps you can take a look at the **find** method to see how to do this.
   E. In each iteration, before moving **current** to **current.next**, set **previous** to **current**.
   F. When the node with the data is found, set the **next** pointer of **previous** to **current.next**.
   G. Consider the case when the data is not found in the list.

3. **Implement the printList Method:**
   Implement logic to traverse the list and print the data of each node.

   Steps:
   A. Check if the list is empty. If it is, print a message like "List is empty."
   B. If the list is not empty, create a temporary **Node<E>** variable named **current** and initialize it with the **head** of the list.
   C. Traverse the list while **current** is not **null**.
      a. In each iteration, print **current.data** followed by a space, comma or newline (**\n**).
      b. Move to the **next** node by setting current to **current.next**.

4. **Implement the size Method:**
   Implement logic to count the number of nodes in the list using the steps below (OR)
   an alternative approach is to keep track of the number of nodes in the list and return this in the size method.
   Steps:

   A. Define an integer variable **count** and initialize it to 0.
   B. Check if the list is empty. If it is, return **count**.
   C. If the list is not empty, create a temporary **Node<E>** variable named current and initialize it with the head of the list.
   D. Traverse the list, incrementing **count** in each iteration until the end of the list is reached.
   E. Return the **count** variable.

Optional challenges

If you feel that you still do not grasp the idea behind linking or linked lists, or you'd like an extra challenge, consider these optional tasks:

- Enhance the **add** Method: Implement the ability to add a node at a specific **index**.
- Write a **get(int index)** Method: Implement the ability to return data stored at a specific **index**.
- Using the **Queue** ADT below implement each method of this ADT *using* the LinkedList as underlying storage data structure.

```
public interface Queue<E> {
    void enqueue(E element);
    E dequeue();
    E front();
    int size();
    boolean isEmpty();
}
```

  Although this may seem like a big task, it actually only requires a few lines of code to implement.
- ***Implement a Circular Linked List:*** Unlike a regular linked list, a circular linked list has a *tail node* connected back to the head node, creating a circular loop. Implement a circular linked list class with basic operations. This implementation is useful to use as a Queue because we don't need to maintain two pointers for front and rear if we use a circular linked list. We can maintain a pointer to the last inserted node and the front can always be obtained by referring to the next node from the tail.

Reflection:

1. **Understanding next in LinkedList:**
   - Reflect on the role of the next attribute in the Node class. How does it contribute to the structure of a LinkedList?
   - Consider the implications of setting the next pointer of a node. How does this action link nodes together in a list?
2. **Generics in LinkedList (<E>):**
   - Explore the significance of <E> in the context of Node<E> and LinkedList<E>. What does it represent?

- How does using <E> allow the LinkedList to be versatile in terms of the data types it can handle?
- Reflect on how you can specify the type for E when creating an instance of LinkedList in your code.

3. **Efficiency of find(E data) Method:**
    - Discuss your understanding of the find(E data) method. How does it traverse the LinkedList to find an element?
    - Consider the worst-case scenario where the element is not present. What does this imply about the method's performance, particularly in terms of time complexity?
4. **Comparing find Method in LinkedList and Array-based List:**
    - How would the process of finding an element in an array-based list differ from that in a LinkedList?
    - Consider the underlying structures of both types of lists and how they impact the implementation and performance of a find method.
5. **get(int index) Method in LinkedList vs Array-based List:**
    - Discuss how retrieving an element at a specific index (get(int index)) differs between a LinkedList and an array-based list.
    - Reflect on how the data structure's design affects the efficiency and implementation of this operation.
6. **Choosing Between LinkedList and Array-based List:**
    - Can you identify scenarios or use cases where a LinkedList might be more suitable than an array-based list, or vice versa?
    - Consider factors like memory allocation, insertion/deletion operations, and random access when making this comparison.
7. **Efficiency of size Method:**
    - Evaluate the two approaches for implementing a size method: calculating size on-demand vs. maintaining a size counter.
    - Discuss which approach might be preferable and under what circumstances one might be chosen over the other.
8. **Capacity Considerations in LinkedList:**
    - Reflect on whether a LinkedList can ever be "full". What factors might limit the size of a LinkedList? Are elements in a linked list stored in the program’s stack or heap?
    - Compare this with the concept of capacity in an array-based list and how it handles size limitations.

Problem 1 - Jen & Berry's[1]

Jen drives her ice cream truck to her local elementary school at recess. All the kids rush to line up in front of her truck. Jen is overwhelmed with the number of students (there are *2n* of them, i.e. some even number), so she calls up her associate, Berry, to bring his ice cream truck to help her out. Berry soon arrives and parks ***at the other end of the line of students***. He offers to sell to the last student in line, but the other students revolt in protest: "The last student was last! This is unfair!"

```
Jen's Truck
    |
    V
[Kid1]-> [Kid2]-> [...]-> [KidN]-> [KidN+1]-> [...]-> [Kid2N]--> Berry's Truck
```

Initially, all kids are in a single line that starts at Jen's truck and ends at Berry's truck. `Kid1` is at the front of the line near Jen's truck, and `Kid2N` is at the back, right in front of Berry's truck.

The students decide that the fairest way to remedy the situation would be to have the back half of the line (the n kids furthest from Jen) reverse their order and queue up at Berry's truck, so that the last kid in the original line becomes the last kid in Berry's line, with the $(n+1)^{st}$ kid in the original line becoming Berry's ***first customer***.

```
Jen's Truck
    |
    V
[Kid1]-> [Kid2]-> [...]-> [KidN]-> [Kid2N]-> [...]-> [KidN+1]--> Berry's Truck
```

After the reordering:

- The first half of the line (from `Kid1` to `KidN`) remains at Jen's truck.
- The second half of the line (from `KidN+1` to `Kid2N`) is reversed and now queues up at Berry's truck in reverse order.

---

a. Given a ***linked list*** containing the names of the *2n* kids, in order of the original line formed in front of Jen's truck (where the first node contains the name of the first kid in line), describe an algorithm to modify the linked list to reverse the order of the last half of the list. Your algorithm should not make any new linked list nodes or instantiate any new non-constant-sized data structures during its operation.

   Pointers:
   - How is the initial lineup of kids described in terms of a linked list?
   - Recognize that the problem involves manipulating a linked list and focus on the specific part of the list to be modified .
   - Consider dividing the problem into smaller steps or stages.
   - Determine how to find the midpoint of the list (n^{th} node) efficiently.
     Hint: [redacted]
   - Think about how you can reverse the second half of the list without creating new nodes or data structures. Is there a way to "swap" elements?

[1] Source: https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/mit6_006s20_prob1sol/

b. Write the **reorder_students()** algorithm in Java. This method should be added to the **LinkedList** class that was developed in the Preparation section. If you didn't complete this step or need a reference, you can use the provided **LinkedList** class, which already implements all the necessary components.

   Example Implementation in the LinkedList Class

```
public class LinkedList<E> implements List<E> {
    // Node and LinkedList implementation as per the Preparation section

    public void reorder_students() {
        // Method implementation here
        // 1. Find the middle of the list
        // 2. Reverse the second half
        // 3. Reconnect the halves
    }

    // Other LinkedList methods...
}
```

   An example solution in java can be found in the "*answers*" directory. Though try to implement it yourself first.

Reflections

- How does the efficiency of your solution scale with the number of students?
- Why is a linked list suitable for this problem? Could other data structures (like arrays or stacks) be used, and how would they compare in terms of efficiency and complexity?
- Are there any parts of your algorithm that could be made more efficient?
- How did you test and validate your solution? Did you consider edge cases and various test scenarios?
- If the number of students (nodes) were to significantly increase, would your solution still hold up effectively?

Problem 2 - Algorithms - Make It Go Fast!

a. Here is an algorithm that for any given array calculates the sum of its unique elements:

```
// Finds the sum of all unique elements in an array
public static int sumOfUniqueElements(int[] array) {
    int sum = 0;
    for (int i = 0; i < array.length; i++) {
        boolean isUnique = true;
        for (int j = 0; j < array.length; j++) {
            if (i != j && array[i] == array[j]) {
                isUnique = false;
                break;
            }
        }
        if (isUnique) {
            sum += array[i];
        }
    }
    return sum;
}
```

Input: An array of integers (array), e.g.: {-1, 2, -1, 2, 3}
Output: The sum of all unique elements in the array, e.g.: 3 (only 3 is unique)
Process: The algorithm iterates through each element of the array (i). For each element, it checks whether this element appears elsewhere in the array (j). If a duplicate is found, the element is not considered unique. If no duplicates are found, the element is added to the sum.

Given this algorithm, discuss the following:

- Identify the number of times each loop runs in relation to the size of the input array (n). See the below explanation.
- Find inefficiencies in the provided algorithm, if you can find them.

To count the operations, follow these steps:

1. **Identify the Basic Operations**
   First, determine what constitutes a basic operation in the context of your algorithm. This could be an arithmetic operation, a comparison, an assignment, or accessing an array element. The choice of basic operation might depend on what you are analyzing. For example, in sorting algorithms, comparisons are often considered the basic operation.
2. **Count Operations in Each Line of Code**
   Go through the algorithm line by line and count the number of basic operations. For simple lines of code, this might be straightforward. For instance, sum += array[i] might count as one operation (an addition and an assignment).
3. **Consider the Loops**
   Loops can significantly add to the operation count. Analyze how many times each loop runs.
   - For a for-loop, this is often directly related to the loop bounds.
   - For while-loops, you'll need to understand the logic to determine how many times the loop runs.

- Nested loops multiply the number of operations, so it's important to understand their relationship.

4. **Account for Conditional Statements**
   If your algorithm contains if-else statements or other conditional logic, remember that the operations in these blocks may not always be executed. Estimate how often these blocks are run, which might be based on the data or the structure of the algorithm.
5. **Summarize the Operations**
   Add up the operations for each part of the algorithm. In many cases, you'll express this sum as a function of n, where n is the size of the input (like the number of elements in an array).

   Example Analysis
   Consider a simple linear search algorithm:

```
for (int i = 0; i < n; i++) {
    if (array[i] == target) {
        return i;
    }
}
```

   1. The loop runs ***n*** times. (note that ***n*** appears to be the size of `array`)
   2. Each loop iteration performs ***1*** comparison (***i < n***), ***1*** comparison in the if-statement, and **1** array access. That's ***3*** operations per iteration.
   3. Total operations: ***3 * n***.

b. Discuss the following:
   - Propose and implement an optimization to reduce the number of operations, especially for large arrays. Ensure the optimized algorithm still correctly calculates the sum of unique elements.
   - Explain your optimization and why it improves the algorithm.
   - Discuss the number of steps of your optimized algorithm compared to the original.
   - If you want to you can implement your optimization, if you do not have time, skip to c.

   Before continuing to c. try at least to find ***1*** way to optimize the algorithm.

c. Here's an example of a *potentially* more optimized version of the `sumOfUniqueElements` algorithm:

```
import java.util.HashSet;
import java.util.Set;

public static int sumOfUniqueElementsOptimized(int[] array) {
    Set<Integer> uniqueElements = new HashSet<>();
    Set<Integer> duplicates = new HashSet<>();

    for (int value : array) {
        // If the value is already in uniqueElements, it's a duplicate
        if (!uniqueElements.add(value)) {
            duplicates.add(value);
        }
    }
```

```
    int sum = 0;
    // Sum the elements that are unique
    for (int value : uniqueElements) {
        if (!duplicates.contains(value)) {
            sum += value;
        }
    }

    return sum;
}
```

Analyze the Optimization:

- Describe how the optimized version differs from the original.
- Explain why these changes could improve the algorithm's efficiency.
- How do you think a `HashSet` works?
    - Do a search online or ask a teacher (or AI friend) to explain it.
    - If you do not get it right now, do not worry, it will be further explained next week.
- Discuss the number of steps (or operations) in the optimized algorithm compared to the original for an array of size **n**. Assume that `.add` and `.contains` can be done in a single step.

Benchmarking:

- Use the provided Benchmark.java file to run and compare both versions of the algorithm. (You can find the implementations in the same file, so you do not have to copy them).
- Try to understand the benchmarking code and how it tests the performance of both algorithms.
- Execute the benchmark and observe which algorithm runs faster.

Analysis:

- Reflect on the benchmark results. Were they as expected? Why or why not?
- Consider the factors that might affect the performance of each version (e.g., array size, number of unique vs. duplicate elements).
- Try to benchmark with different parameters to find out whether your results differ.

Reflections

- Reflect on how the original algorithm's nested loops affect its performance. Why does this approach lead to a higher number of operations?
- Consider why the optimized algorithm with HashSet is more efficient. How does using a HashSet reduce the number of operations needed?
- Why is it important to have efficient algorithms in practice?
- Reflect on the benchmarking results. Were there any surprises in how the two algorithms performed? Why might these results vary under different conditions, such as array size or the proportion of unique vs. duplicate elements?
- Are there any other approaches or data structures that could further improve the algorithm's efficiency?

Problem 3 - Balancing Parentheses

Write an algorithm to check if an expression containing different types of brackets (parentheses ***()***, square brackets ***[]***, and curly braces ***{})*** is balanced. In a balanced expression, every opening bracket (including ***(, [, {***) is matched with its closing counterpart (respectively ***),], }***) in the correct order.

Input:

A string containing an expression with various types of brackets.
For example: "**`{[(a + b) * c] - d}`**"

Output:

A boolean value: **true** if the brackets in the expression are balanced, **false** otherwise.
For the above example: **`True`**

Pointers:

- This algorithm can be efficiently written using one of this week's data structures.
- Use the `.toCharArray()` method to loop over characters in a String as if it were an array.
- You should ignore non-bracket characters while ensuring the sequence and type of brackets are properly balanced.
- Consider edge cases such as unmatched brackets (e.g., "a) b", "{a + b)"), improperly nested brackets (e.g., "([ )]"), and empty strings.
- It's crucial to check not just the correct quantity of opening and closing brackets, but also their correct types and order.

Write the algorithm in **pseudocode** first, after that implement the algorithm in **Java**. Use ***`BalancedParenthesesChecker.java`*** as a starting point.

An example implementation is given in the "answers" directory. Though try to implement the algorithm on your own first.

Reflections

- Reflect on why a [redacted] was the appropriate choice for this problem. How does its functionality align with the requirements of checking balanced parentheses?
- Consider how your algorithm handles various test cases, especially edge cases like unbalanced parentheses or expressions with no parentheses.
- Reflect on the simplicity of your solution. Could the algorithm be simplified further while still being effective?
- Think about the steps you took to solve this problem. How did breaking down the problem into smaller parts help in finding a solution?
- Discuss where else in computer science this type of problem-solving might be applicable. Can you think of other scenarios where matching pairs in the correct order is crucial?

An example Java implementation is given in the "answers" directory. Though try to implement the algorithm on your own first.