Assignment 4

CS 1027
Computer Science Fundamentals II

Due date: December 4, 2023 at 11:55 pm

Learning Outcomes

In this assignment, you will get practice with:

- Writing and using iterators
- Recursion

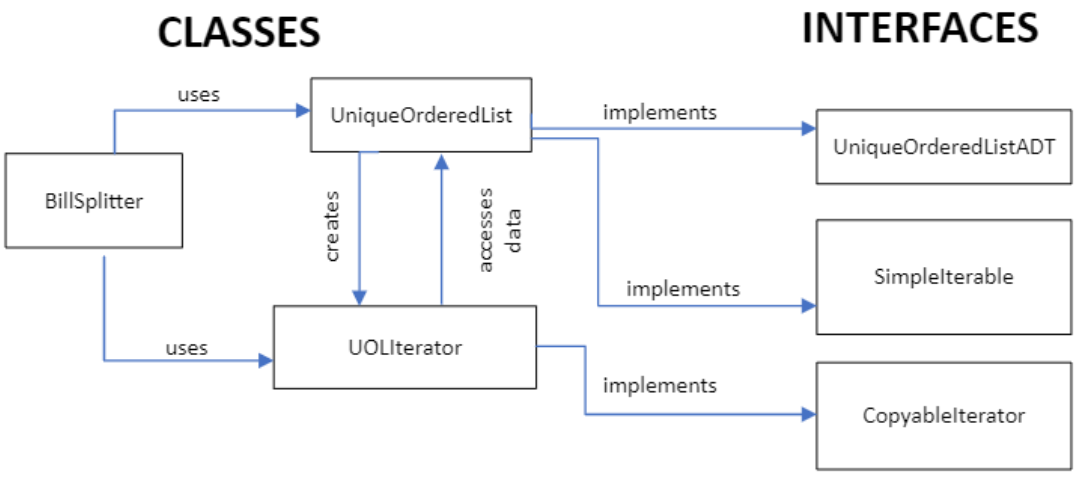
Introduction: Split the Bill

You and your friend have gone to a restaurant and ordered many dishes and drinks. Through the course of the night, you both have lost track of who has ordered what, who ate what and who drank what. You decide to just split the bill based on what each of you can pay (so you've agreed to something like "I will pay $80 and you will pay $20"), but the restaurant insists on each of you pay for entire items. So you have to find items that add up to your desired totals.

For example, suppose you had purchased items that cost $10, $15, $16, $20, $22 and $30, for a total bill of $113. And suppose that you agreed to split where you would pay $36 and your friend would pay $77. To accomplish this, you could pay for the $16 item and the $20 item, and your friend could pay for everything else. If you wanted to pay $24 and for your friend to pay $89, then this is impossible, since there are no items whose costs add to $24.

In this assignment, you will write a recursive algorithm to determine how to split a bill between you and your friend. The algorithm is described in Background 2 and Step 4. To manage the bill, you will use a data structure and develop an iterator in Steps 1-3.

You will build several classes and use several interfaces in this assignment. The relationship between these classes is shown in the figure below.



Assignment 4

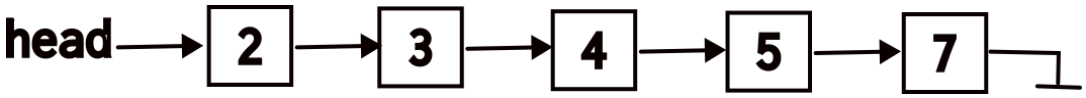
CS 1027
Computer Science Fundamentals II

Some of these files are provided to you (e.g., the interfaces). Others are provided and you will need to modify (e.g., UniqueOrderedList). All these classes are described in the sections below. (The LinearNode class is not shown in the figure.)

Background 1: `UniqueOrderedList`

A `UniqueOrderedList` is an ADT where the elements are in order but are also unique: if you try to insert the same element in the ADT twice, nothing happens the second time. At most one copy of each element is stored in the ADT. So if you inserted 3, 5, 7, 3, 2, 4, and 2 in your `UniqueOrderedList`, then the resulting contents would be 2, 3, 4, 5, 7, as the elements are stored in order (from smallest to largest in this assignment, unlike Assignment 2) without duplicates. **You are given this ADT in the UniqueOrderedList.java file**. **You are also provided with the LinearNode.java** file, which should not be modified in this assignment.

After inserting the items listed above, the `UniqueOrderedList` would look like this:



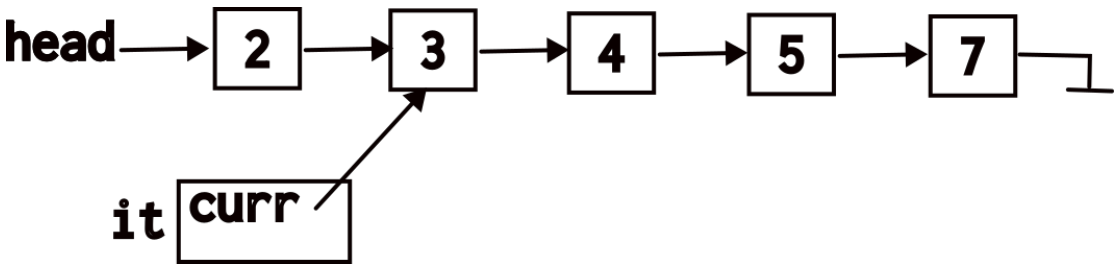
The `UniqueOrderedList` class implements the `UniqueOrderdListADT` interface:

```
public interface UniqueOrderedListADT<T> {
      public boolean contains(T element);
      public boolean add(T element);
      public int size();
}
```

You are provided with the UniqueOrderedListADT.java file. Note that the interface does not contain any way to retrieve the elements. **You will modify the** `UniqueOrderedList` class and **build an iterator class** to provide a way to retrieve the elements in order.

Step 1: Iterator for UniqueOrderedList

In this part, you will build an **iterator** for the `UniqueOrderedList` class. The iterator method should **NOT** copy the data into an array. Instead, the iterator should keep a pointer to the element in the list that is **next to be** returned by the iterator, as described in class. The elements should be returned in the order that they appear in the structure (i.e., in increasing order). Thus, if you had an iterator called `it` for the list shown earlier, and the iterator had called next once, then the iterator's status would be represented by a pointer to the node containing 3:



Assignment 4

CS 1027
Computer Science Fundamentals II

Create an iterator class called `UOLIterator`. The class must be generic. Your class must have the following **private** instance variable:

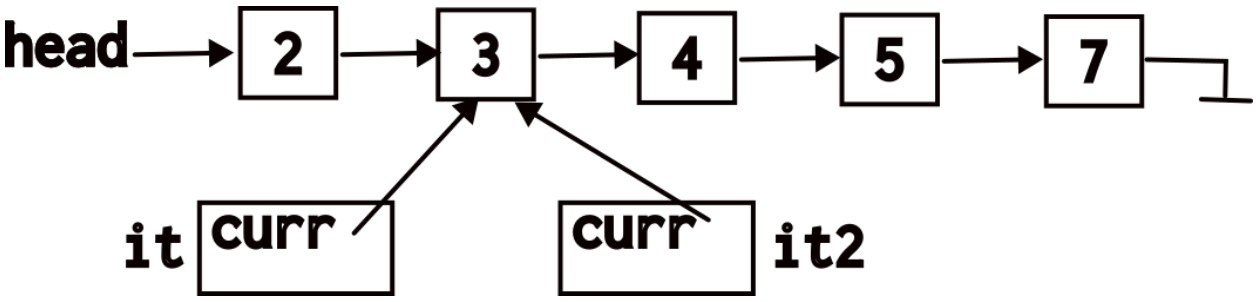
- A `LinearNode<T>` pointer called `curr` that points to the next item that should be returned by the iterator.

The iterator class must have the following **public** methods:

- A public constructor to the class that accepts a `LinearNode<T>` parameter that is a reference to the node in the list where you want to start your iterator (or null if the linked list is empty). ***Hint: Usually, the start point will be the first node in the `UniqueOrderedList`, except when you are copying (described in Step 2).***
- A `hasNext` method that takes no parameters and returns a `boolean` to determine if there are unvisited elements in the list, as described in the Iterators course material.
- A `next` method that takes no parameters and returns an element of type `T`, which is the **next** unvisited element in the list, as described in the Iterators course material. The method should throw the `NoSuchElementException` exception if there is no next element (i.e., when `hasNext` would return false). Make sure to import this exception as `java.util.NoSuchElementException`.

Step 2: Make the Iterator copyable.

Now, you should make your iterator ***copyable***. By copyable, we mean that you should be able to copy the iterator position to another, separate iterator. After copying, both iterators can progress through the collection from the same (copied) point. When copying an iterator, we copy the ***current status*** of the iterator: the copied iterator should be at the same position as the iterator being copied. So for instance, if the iterator `it` in the previous diagram was copied, then a second iterator `it2` would be created, pointing at the same location:



After this, `it` and `it2` start in the same spot, but they are ***independent*** iterators: advancing one does not advance the other.

Modify your `UOLIterator` class to make it copyable. The class **must** implement the `CopyableIterator` interface which is given to you:

```
public interface CopyableIterator<T> extends Iterator<T> {
      public CopyableIterator<T> copy();
}
```

Note that this interface extends the (Java built-in) `Iterator` interface, so it ensures that any implementing class must also have `hasNext` and `next` methods. You must also add the following **public** method to the `UOLIterator` class:

- The copy method (as shown in the interface) accepts no parameters but returns a new iterator object that is at the same position as the current iterator (i.e., `this`). Note that this means that the copied iterator ***points to the same place in the*** `UniqueOrderedList` as the current iterator. *Hint*: in this case, the starting point of the iterator is not the first node in the list, so the parameters to your constructor call (when instantiating the copy of the iterator) should be the **current position**.

Step 3: Add the iterator method to the UniqueOrderedList class

You must now modify the `UniqueOrderedList` class to add the iterator method. Add the following public method to the `UniqueOrderedList` class:

- An iterator method that has no parameters and returns an object that implements the CopyableIterator<T> interface. This iterator method should instantiate and return a new iterator object from the `UOLIterator` class that you've defined. This iterator must start iteration at the ***beginning*** of the list. *Hint*: when creating this iterator, the constructor to the iterator class should accept a pointer to the ***first*** node of the linked list as a parameter.

You are provided with the following interface that defines that a class implements a copyable iterator:

```
public interface SimpleIterable<T> {
      public CopyableIterator<T> iterator();
}
```

You must modify the `UniqueOrderedList` class so that it implements the `SimpleIterable` interface (but make sure that the `UniqueOrderedList` still also implements the `UniqueOrderedListADT` interface -- `UniqueOrderedList` should implement both interfaces.).

Testing: Run the TestUOL.java file to test iterator copying and creation.

Background 2: Splitting the bill

We will use a recursive algorithm to determine a way to split the bill between you and your friend. You have agreed on an amount that you will pay (which we call the ***target***). Your friend has agreed to pay everything else.

Assignment 4

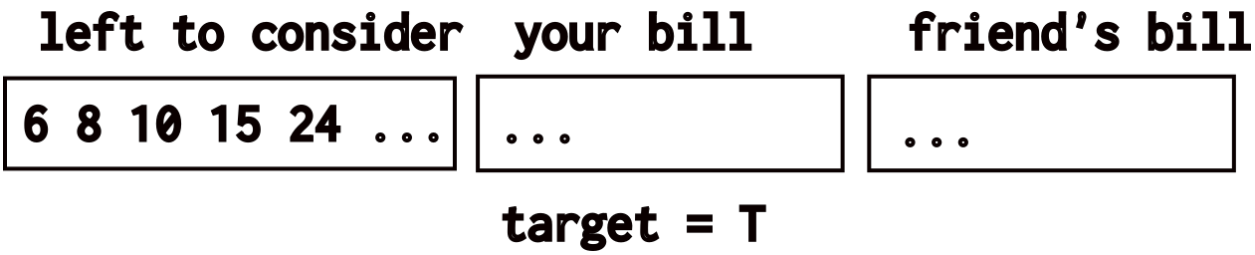
CS 1027
Computer Science Fundamentals II

You must find a way to assign each of the items to either you or your friend. To do so, you will use a recursive algorithm. The algorithm, which we call `split` takes two parameters:

1. A list of all the items left to consider (i.e., those that you haven't decided who will pay for yet).
2. The target of the total cost that **you** have to pay for.

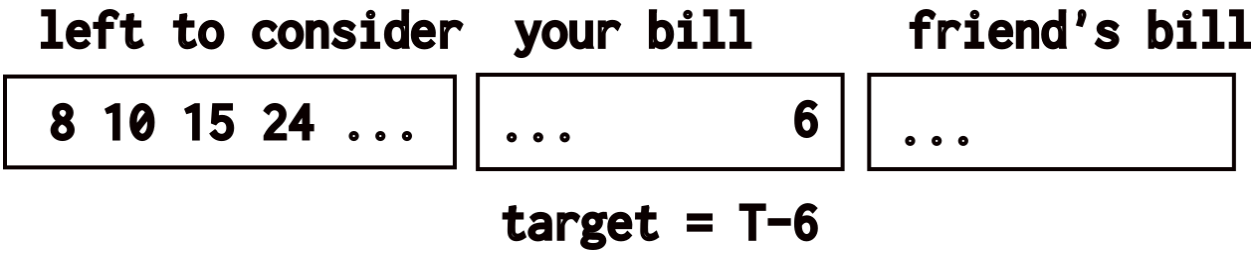
The method should **return** a list of all the costs of all the items that **you** have to pay for, if it exists. If it does not exist (i.e., there's no way to split up the costs of the remaining items so that it matches your desired costs exactly), the method should return nothing (**null)**.

The recursive algorithm works as follows: consider the situation where you have already assigned some items to you or your friend:



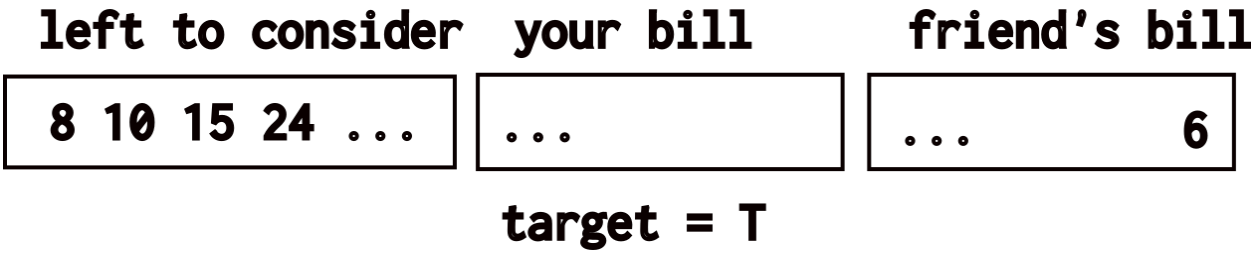
The target T is the total dollar amount that you must be assigned from the remaining items. Consider the first item (with cost $6) and assume that your target is at least $6. Then there are two possibilities for this item: you pay for it, or your friend pays for it.

In the **first case**, where you pay for it, the situation changes to this:



The $6 item is given to you, and now the new target is T-6.

But in the **second case,** when we assign the $6 item to you friend, the situation looks like this:



Notice that your target (the amount you want to pay from the remaining items) is still T because the $6 item was assigned to your friend.

The key to the recursive algorithm is to consider BOTH these possibilities recursively. If **either** of them (recursively) returns a valid solution, then we accept that as a valid way to split the bill. (If both do, then either is a valid way to assign the items and we pick the first one we find, which is the first option). When we recursively consider the first option to try and find a solution, it will create two more possibilities (i.e., what to do with the $8 item) and so on. In this way, if any assignment of items equals our original target, our recursive solution will eventually find it.

The base case of the recursive algorithm is, as usual, simple: if there are no items left, then we've found a valid way to split the bill if the target is 0. Otherwise, this option is not valid (and so we should return **null**). To start the algorithm, we use the full list of all items on the bill, as well as the agreed-to target for your portion of the bill. For example, when starting, the split method would get a list of items like [8,9,13,25] and the target for your portion of the bill, like $22. The algorithm is given by the following pseudocode:

```
1   split(items_remaining, target)
2     if (there are no items remaining):
3       if the target is zero, return an empty list.
4       else return null
5     else:  // there are more items remaining.
6       let curr be the first item in the remaining items.
7       if (curr <= target):
8         soln = split(items_remaining without curr, target-curr) // assign to you.
9         if soln is a possible solution, return the solution of (soln + curr) (*)
10
11      soln = split(items_remaining without curr, target) // assign to your friend.
12      return soln
```

(*) "soln + curr" means add the element curr to soln (which is a list) and then return it. See the image of the first case above for the representation of this case.

As an example, consider a bill of three items of value $2, $10, $13, and a target of $15. Clearly, we can see that we want to report that you can choose the $2 and $13 item in this small case. How is this found by the split algorithm?

- Original call **#0** is split([2,10,13],15). The recursive case (the first else on line 5) is executed (there are still items left).
- curr = 2 on line 6. curr is less than or equal to the target = 15. Recursive call to split is made.
    - Recursive call **#1** is split([10,13], 13) on line 8. The recursive case is executed on line 5.
    - curr = 10 on line 6. curr is less than or equal to the target = 13. Recursive call to split is made on line 8.
        - Recursive call **#2** is split([13],3) on line 8. [*Notice here that we know that this call must eventually return null: there's no way to split a bill with one $13 item in it such that you are assigned $3.*] The recursive case is executed on line 5.
        - curr = 13 on line 6, curr is **NOT** less than or equal to the target 10. The if statement is false, so we go to line 11.
            - Recursive call **#3** is split([],3) on line 11. The base case is executed, and this call returns null on line 4.

        - The recursive call **#3** returned null and so return null on line 12 [*as we expected*].
    - The recursive call **#2** returned null, and the if statement on line 9 is not executed.
    - The method then goes to line 11. A recursive call to split is made on line 11.
        - Recursive call **#4** is split([13],13). The recursive case is executed on line 5.
        - curr = 13 on line 6. curr is less than or equal to the target = 13. Recursive call to split is made on line 8.
            - Recursive call **#5** is split([],0) on line 8. The **base case** is executed and the empty list is returned on line 3.
        - Recursive call **#5** returned the empty list, so soln = [] on line 8.
        - On line 9, return soln = [13], since curr is 13.
    - Recursive call **#4** returns soln = [13].
    - Return soln = [13] on line 12.
- Recursive call **#1** returns soln = [13] on line 8.
- On line 9, Orignal call **#0** returns soln = [2,13] since curr = 2.
- The final solution is [2,13], as we expected.

Step 4: Splitting the Bill

Implement the split algorithm from the previous background section using the `UniqueOrderedList` class.

Write a class called BillSplitter.java. The class should have one **public static** method:

- `public static UniqueOrderedList<Integer> split (UniqueOrderedList<Integer> in, int target)`: This method will call the next recursive method.

You will also implement one **private static** method:

- `private static UniqueOrderedList<Integer> “yourSplit” (CopyableIterator<Integer> it, int target).`

This method can be called whatever you want (because it’s private). The private method will implement the recursive solution provided in the pseudocode in the previous background section. The public method will call the private method to find the split. Notice that the private method should accept a **copyable iterator** (which represents all the items remaining). This iterator comes from the `UniqueOrderedList` object that is a parameter to the public `split` method. The iterator will need to be **copied**: in implementing lines 8 and 11 of the pseudocode, they should receive **separate copies of the iterator** that each act independently.

Testing: Run the TestSplit.java file to test the split method.

Provided Files

Assignment 4

CS 1027
Computer Science Fundamentals II

You are provided with the following files:

- LinearNode.java
- UniqueOrderedList.java
- UniqueOrderedListADT.java
- CopyableIterator.java
- SimpleIterable.java
- TestUOL.java
- TestSplit.java

The last two files are tester files to help check if your classes are implemented correctly. Similar files will be incorporated into Gradescope's auto-grader. Additional tests will be run that will be hidden from you. Passing all the tests within the provided files does not necessarily mean that your code is correct in all cases.

Marking Notes

Functional Specifications

- Does the program behave according to specifications?
- Does it produce the correct output and pass all tests?
- Are the classes implemented properly?
- Does the code run properly on Gradescope (even if it runs on Eclipse, it is **up to you** to ensure it works on Gradescope to get the test marks)
- Does the program produce compilation or run-time errors on Gradescope?
- Does the program fail to follow the instructions (i.e. changing variable types, etc.)

Non-Functional Specifications

- Are there comments throughout the code (Javadocs or other comments)?
- Are the variables and methods given appropriate, meaningful names?
- Is the code clean and readable with proper indenting and white-space?
- Is the code consistent regarding formatting and naming conventions?
- Submission errors (i.e. missing files, too many files, etc.) will receive a penalty.
- Including a "package" line at the top of a file will receive a penalty.

Remember **you must do** all the work on your own. **Do not copy** or even look at the work of another student. All submitted code will be run through similarity-detection software.

Submission (due December 4 at 11:55 pm)

Assignment 4

CS 1027
Computer Science Fundamentals II

Assignments must be submitted to Gradescope, not on OWL. If you are new to this platform, see these instructions on submitting on Gradescope.

Rules

- Please only submit the files specified below.
- Do not attach other files even if they were part of the assignment.
- Do not upload the .class files! Penalties will be applied for this.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope. **If your code runs on Eclipse but not on Gradescope, you will NOT get the marks! Make sure it works on Gradescope to get these marks.**
- You are expected to perform additional testing (create your own tester class to do this) to ensure that your code works for a variety of cases. We are providing you with some tests but we may use additional tests that you haven't seen for marking.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code as many times as you wish, however, re-submissions after the assignment deadline will receive a late penalty.

Files to submit

- UniqueOrderedList.java
- UOLIterator.java
- BillSplitter.java

Grading Criteria

Total Marks: 20

15 marks	Autograder Tests (some tests are hidden from you)
5 marks	Non-functional Specifications (code readability, comments, correct variables and functions, etc.)