

1 Interweave

Implement `interweave`, which takes in an `IntList lst` and an integer `k`, and *destructively* interweaves `lst` into `k IntLists`, stored in an array of `IntLists`. Here, destructively means that instead of creating new `IntList` instances, you should focus on modifying the pointers in the existing `IntList lst`.

Specifically, we require:

- It is the **same** length as the other lists. You may assume the `IntList` is evenly divisible.
- The first element in `lst` is put in the first index of the array of `IntLists`. The second element is put in the second index. This goes on until the array is traversed, and then we wrap around to put elements in the first index of the array.
- Its ordering is consistent with the ordering of `lst`, i.e. items in earlier in `lst` must **precede** items that are later.

For instance, if `lst` contains the elements `[6, 5, 4, 3, 2, 1]`, and `k = 2`, then the method should return an array of `IntList`, `[6, 4, 2]` at index 0, and `[5, 3, 1]` at index 1.

In the beginning, we reversed the `IntList lst` destructively, because it's usually easier to build `IntList` backwards.

Hint: The elements in the array should track the head of the small `IntList` that they are building.

```
public static IntList[] interweave(IntList lst, int k) {
    IntList[] array = new IntList[k];
    int index = k - 1;
    IntList L = reverse(lst); // Assume reverse is implemented correctly

    while (L != null) {

        IntList prevAtIndex = array[index];

        IntList next = L.rest;

        array[index] = L;

        array[index].rest = prevAtIndex;

        L = next;
        index -= 1;

        if (index < 0) {

            index = k - 1;
        }
    }
    return array;
}
```

Here is a video walkthrough of the solution. We reverse our `IntList` so that we can build up each element of the `IntList[]` array backwards—in general, it is much easier to build an `IntList` backward than forward. The general idea is to initialize each element in the array to null, then put an element of `L` inside the correct index by assigning `array[index] = L`. Then, we get whatever we've built up so far (`prevAtIndex`) and add it to the end of our `rest` element so that we have the entire `IntList` again with one element at the front. Afterwards, we advance `L` to the next element and change the index. One thing to notice is that when jumping ahead in the linked list, one cannot directly jump via `L = L.rest` because the `rest` field of the current `L` is already modified. Instead, we store the next element in a temporary variable `next` and then update `L` to `next`.

2 Skippify

We have the following `IntList` class, as defined in lecture and lab, with an added `skippify` function. Suppose that we define two `IntLists` as follows.

```
IntList A = IntList.list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
IntList B = IntList.list(9, 8, 7, 6, 5, 4, 3, 2, 1);
```

Fill in the method `skippify` such that the result of calling `skippify` on A and B are as below:

- After calling `A.skippify()`, A: (1, 3, 6, 10)
- After calling `B.skippify()`, B: (9, 7, 4)

```
public class IntList {
    public int first;
    public IntList rest;

    @Override
    public boolean equals(Object o) { ... }
    public static IntList list(int... args) { ... }

    public void skippify() {
        IntList p = this;
        int n = 1;
        while (p != null) {

            IntList next = -----;

            for (-----) {

                if (-----) {

                    -----

                }

                -----

            }

            -----

            -----

            -----

        }
    }
}
```

Here is a video walkthrough of the solutions.

```
public class IntList {
    public int first;
    public IntList rest;

    @Override
    public boolean equals(Object o) { ... }
    public static IntList list(int... args) { ... }

    public void skipkify() {
        IntList p = this;
        int n = 1;
        while (p != null) {
            IntList next = p.rest;
            for (int i = 0; i < n; i += 1) {
                if (next == null) {
                    break;
                }
                next = next.rest;
            }
            p.rest = next;
            p = p.rest;
            n++;
        }
        ...
    }
}
```

Explanation:

Looking at **IntList A**, we only need to change the **rest** attribute of **IntList** instances 1, 3, and 6. To achieve this, we will use the **for** loop to find the new **rest** attribute (which we will store in **next**) of the current **IntList** instance (**p**). The outer **while** loop enables us to repeat these actions for, in our case, **IntList** instances 3 and 6. The **int n** will increment by one each iteration and gives us the number of iterations in the for loop, i.e. how many **IntList** instances to skip. Finally, the **if** check accounts allows us to exit the for loop early if we ever hit the end of the Linked List.

3 Gridify

- (a) Consider a circular sentinel implementation of an **SLList** of **Nodes**. For the first **rows** * **cols** **Nodes**, place the item of each **Node** into a 2D **rows** × **cols** array in row-major order. Elements are sequentially added filling up an entire row before moving onto the next row.

For example, if the **SLList** contains elements $5 \rightarrow 3 \rightarrow 7 \rightarrow 2 \rightarrow 8$ and **rows** = 2 and **cols** = 3, calling **gridify** on it should return this grid.

5	3	7
2	8	0

If the SLList contains fewer elements than the capacity of the 2D array, the remaining array elements should be 0; if it contains more elements, ignore the extra elements.

```
public class SLList {
    Node sentinel;

    public SLList() {
        this.sentinel = new Node();
    }

    private static class Node {
        int item;
        Node next;
    }

    public int[][] gridify(int rows, int cols) {
        int[][] grid = new int[rows][cols];

        gridifyHelper(grid, sentinel.next, 0);

        return grid;
    }

    private void gridifyHelper(int[][] grid, Node curr, int numFilled) {
        if (curr == sentinel || numFilled >= grid.length * grid[0].length) {
            return;
        }

        int row = numFilled / grid[0].length;

        int col = numFilled % grid[0].length;

        grid[row][col] = curr.item;

        gridifyHelper(grid, curr.next, numFilled + 1);
    }
}
```