# CS 61BL Summer 2026

# Arrays, Linked Lists, and ADTs

Recitation 01: July 6th, 2026

## 1 Static Electricity

Consider the class `Pokemon`, implemented as shown below:

```
public class Pokemon {
    public String name;
    public int level;
    public static String trainer = "Ash";
    public static int partySize = 0;

    public Pokemon(String name, int level) {
        this.name = name;
        this.level = level;
        this.partySize += 1;
    }

    public static void main(String[] args) {
        Pokemon p = new Pokemon("Pikachu", 17);
        Pokemon j = new Pokemon("Jolteon", 99);
        System.out.println("Party size: " + Pokemon.partySize);
        p.printStats();
        int level = 18;
        Pokemon.change(p, level);
        p.printStats();
        Pokemon.trainer = "Ash";
        j.trainer = "Cynthia";
        p.printStats();
    }

    public static void change(Pokemon poke, int level) {
        poke.level = level;
        level = 50;
        poke = new Pokemon("Luxray", 1);
        poke.trainer = "Team Rocket";
    }

    public void printStats() {
        System.out.println(name + " " + level + " " + trainer);
    }
}
```

(a) Write what would be printed after the `main` method is executed.

```
Party Size: 2
Pikachu 17 Ash
Pikachu 18 Team Rocket
Pikachu 18 Cynthia
```

For a step-by-step walkthrough of how the variables change with each line, see here

(b) On line 28, we set `level` equal to `50`. What `level` do we mean?

A. An instance variable of the `Pokemon` object
B. The local variable containing the parameter to the `change` method
C. The local variable in the `main`
D. Something else (explain below)

B: It is the local variable in the `change` method and does not have any effect on the other variables of the same name in the `Pokemon` class or the `main` method.

(c) If we were to call `Pokemon.printStats()` at the end of our main method, what would happen?

If we were to add this line to our main method, it would error. In the class, `printStats()` is an instance method. What would it mean to print the name and level of the class `Pokemon`, as opposed to a specific `Pokemon`'s name? It doesn't really make sense. So when we try to run this method on our class, it errors.

One more thing to note is the method `change` is declared static itself. Static methods can be called using the name of the class, as in line 19, whereas non-static methods cannot. The golden rule for static methods to know is that **static methods can only modify non-static variables if they are provided a pointer to an object.**.

## 2 Interweave

Implement **interweave**, which takes in an **IntList lst** and an integer **k**, and *destructively* interweaves **lst** into **k IntList**s, stored in an array of **IntList**s. 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 **IntList**s. 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.

# 3 OHQueue

Dawn is designing the new 61B Office Hours Queue. The code below for OHRequest represents a single request.

```
public class OHRequest {
    public String description;
    public String name;
    public boolean isSetup;
    public OHRequest next;
    public OHRequest(String description, String name, boolean isSetup, OHRequest next) {
        /* Omitted: The constructor sets the instance variables that
        have the corresponding names. */
    }
}
```

Dawn would like to find a way to prioritize setup tickets on the queue so that they appear at the top. She wants to implement this based on the isSetup field of each OHRequest, but sometimes students forget to set it to true, so she decides to use description as backup to break ties.

(a) Fill in the compare method of OHRequestComparator below. First, if one but not both of the OHRequests have their isSetup set to true, the one with isSetup set to true should take priority (ie. earlier on the queue). If both or neither of the OHRequests have their isSetup set to true, tiebreak with the description: the description has to exactly match "setup" in order to be counted as a setup issue. If both requests have such descriptions, it's a true tie and return 0.

Note: In this question, **we define s1 being "less than" s2 as s1 being prioritized over s2.** Consider it equivalent to comparing the two requests' position in the queue (1 vs 3, for instance).

```
public class OHRequestComparator implements Comparator<OHRequest> {
    @Override
    public int compare(OHRequest s1, OHRequest s2) {

        boolean s1DescMatchesSetup = s1.description.equals("setup");

        boolean s2DescMatchesSetup = s2.description.equals("setup");

        if (s1.isSetup && !s2.isSetup) {
            return -1;
        } else if (!s1.isSetup && s2.isSetup) {
            return 1;
        } else if (s1DescMatchesSetup && !s2DescMatchesSetup) {
            return -1;
        } else if (!s1DescMatchesSetup && s2DescMatchesSetup) {
            return 1;
        }
        return 0;
    }
}
```

(b) Create a class **OHIterator** that implements an **Iterator** over **OHRequests** and only returns requests with good descriptions (using the **isGood** function). Our **OHIterator**'s constructor takes in an **OHRequest** that represents the first **OHRequest** on the queue. If we run out of requests when we try to get the next one, throw a **NoSuchElementException** using **throw new NoSuchElementException();**

```
public class OHIterator implements Iterator<OHRequest> {
  OHRequest curr;
  public OHIterator(OHRequest queue) {

    curr = queue;
  }
  public static boolean isGood(String description) {
    return description.length() >= 5;
  }
  @Override
  public boolean hasNext() {

    while (curr != null && !isGood(curr.Description)) {

      curr = curr.next;
    }
    if (curr == null) {
      return false;
    }
    return true;
  }
  @Override
  public OHRequest next() {
    if (!hasNext()) {

      throw new NoSuchElementException();
    }

    OHRequest currRequest = curr;

    curr = curr.next;

    return currRequest;
  }
}
```

# 4 Gridify

(a) Consider a circular sentinel implementation of an **SLList** of **Node**s. For the first **rows** * **cols** **Node**s, 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 $5 \to 3 \to 7 \to 2 \to 8$ 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);

  }
}
```

# 5 In Circles 2: More Circles

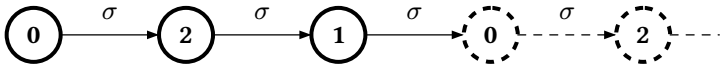
In abstract algebra, an **orbit** is defined as a closed path of elements generated by a permutation of a set. A permutation is a function that takes a set and **rearranges some or all of its elements**.

For example, given the following permutation $\sigma$ of the set of integers in the interval $[0, 5]$,

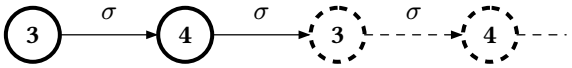
$$\sigma = \{0, 1, 2, 3, 4, 5\} \longmapsto \{2, 0, 1, 4, 3, 5\}$$

This permutation maps **0 to 2, 1 to 0, 2 to 1, 3 to 4, 4 to 3, and 5 to 5.**

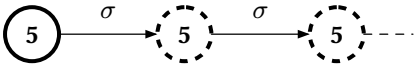
To find the orbits, let's try repeatedly applying $\sigma$ to each element of the set, starting at 0.




Notice how 3, 4, and 5 are completely unreachable, as we've encountered a repeating cycle. Let's try starting from 3 instead.




We've encountered another repeating cycle! That leaves just 5 left to check.




As we can see, there are three distinct orbits in $\sigma$. **They are $\{0, 1, 2\}$, $\{3, 4\}$, and $\{5\}$.**

Write a function that takes in an **`int[]` of length $N$ containing the integers $[0, N)$** representing a permutation, and returns the number of orbits in the permutation.

*Example: $\sigma$ would be represented in Java by the array* `[2, 0, 1, 4, 3, 5]`

```
public static int countOrbits(int[] permutation) {
  int count = 0;
  Set<Integer> seen = new TreeSet<>();

  for(int i = 0, i < permutation.length; i++) {
    int currValue = i;
    if (!seen.contains(currValue)) {
      while(!seen.contains(currValue)) {
        seen.add(currValue);
        currValue = permutation[j];
      }
      count++;
    }
  }

  return count;
}
```