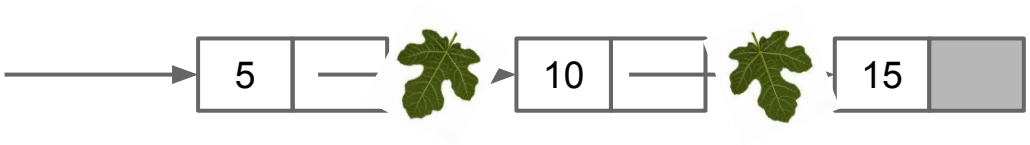
Discussion 3: Linked Lists, Arrays

Administrivia

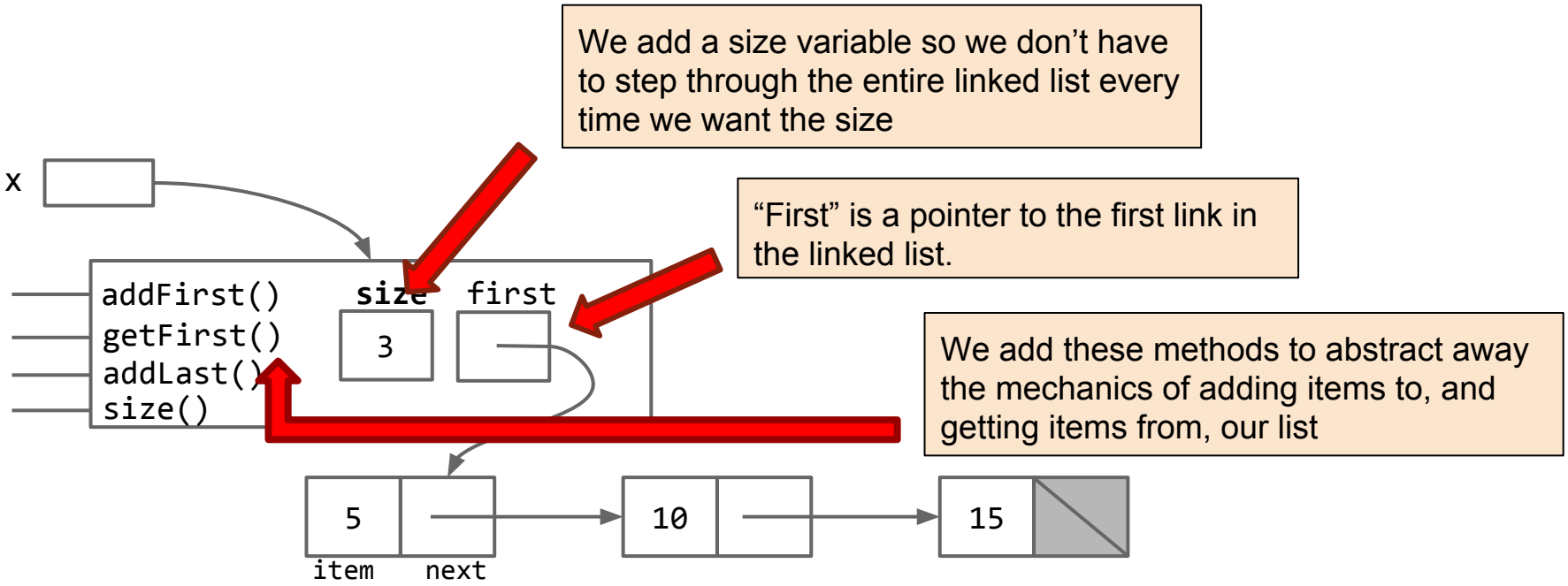
1. Project 1A due 2/8 at 11:59!
    a. You will be implementing a deque interface
    b. Most common piece of advice from TA's and past students: START THE PROJECTS EARLY!! So pls start :)
    c. Highly recommend to use Java Visualizer
2. Pre-semester advising available @414
3. Choose your mentor GSI here
4. Automated extension system available here
5. HKN provides past exams, hosts review sessions, and has drop-in tutoring services M-F, 11AM-5PM throughout the semester. See @574 for more information.
6. Weekly surveys will be announced on Piazza from now on so please check for those!



Review: SLList

- The SLList is a class that hides the internal workings of the Linked List using something called encapsulation.
- The raw linked list is a “naked structure”: you can see everything inside of it including first and rest. If someone were to use it, they would need to know what these are and also know how to step through the list.
- The SLList allows us to hide away the inner workings of the linked list and also allows us to store meta-information about the list as a whole such as the size.
- Analogy: You refer to a train as a whole, not as individual cars. In the same way, instead of just having the raw links with first and next, you refer to it as a whole.

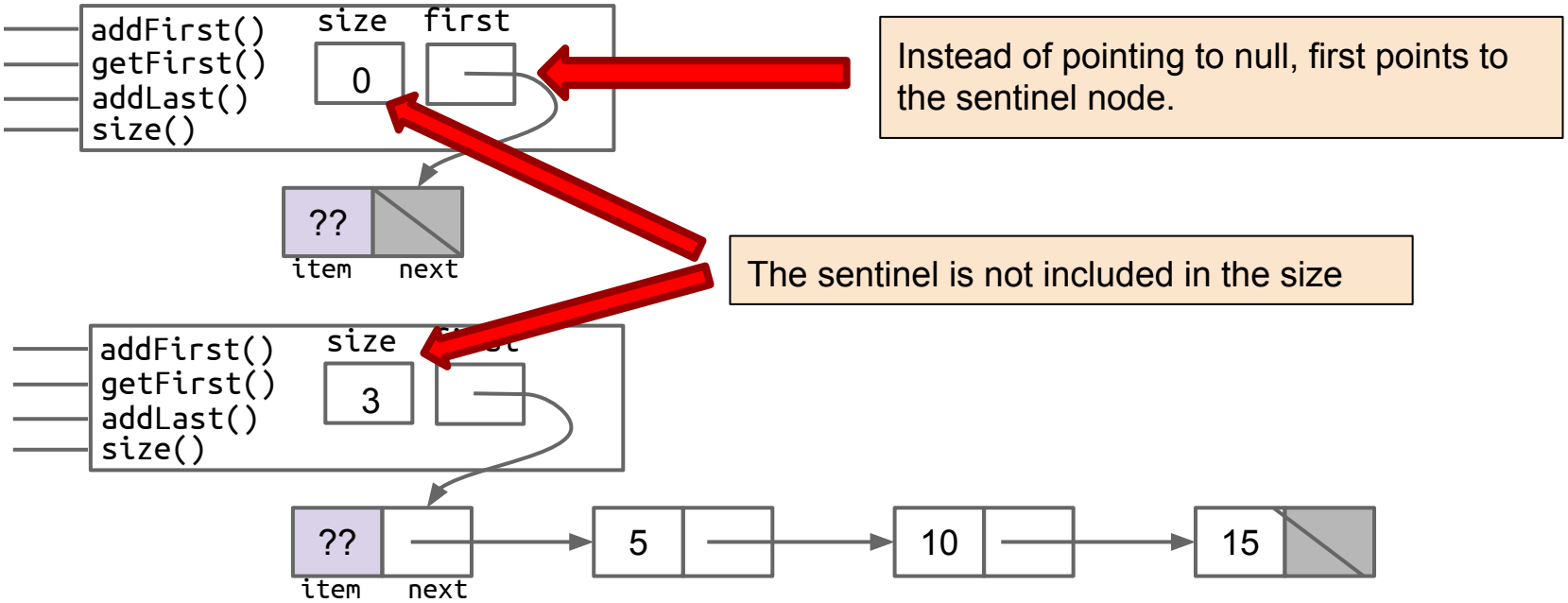
Review: SLList Diagram



Review: Sentinel Nodes

- Let's add some more useful stuff to our list class.
- **Question:** How do we represent an empty linked list? Set first to null?
- This leads to ugly code, as we will have to add a bunch of null checks to code that handles the list.
    - For example, addLast would need a special case for if the list is null → finding the old last is different if the list is null vs. if it has at least one item.
- Goal of sentinel: make code as simple and generic as possible so we don't have to have any special cases.
- A sentinel is a "dummy" node that doesn't actually hold any important information. It's a placeholder that exists solely so we can avoid those pesky null checks.

Review: Sentinel Node Diagram

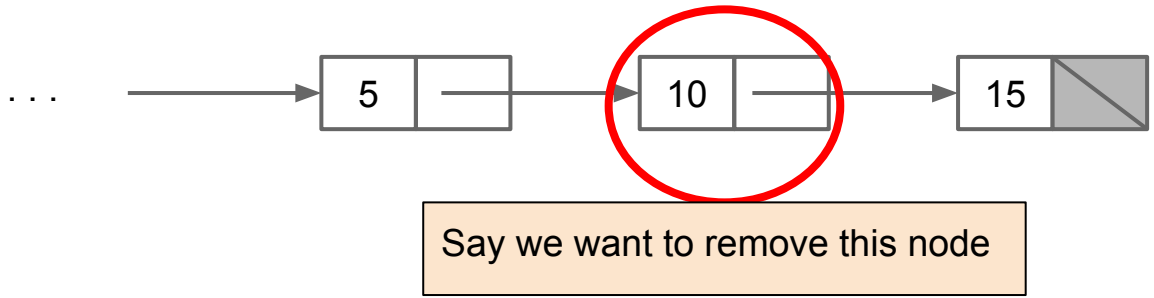


Review: DLLists

- One thing that a list should support is **remove(int i)** which removes the node whose item is i**.**
- With our brand new SLList with sentinel, what would we have to do to remove a link?

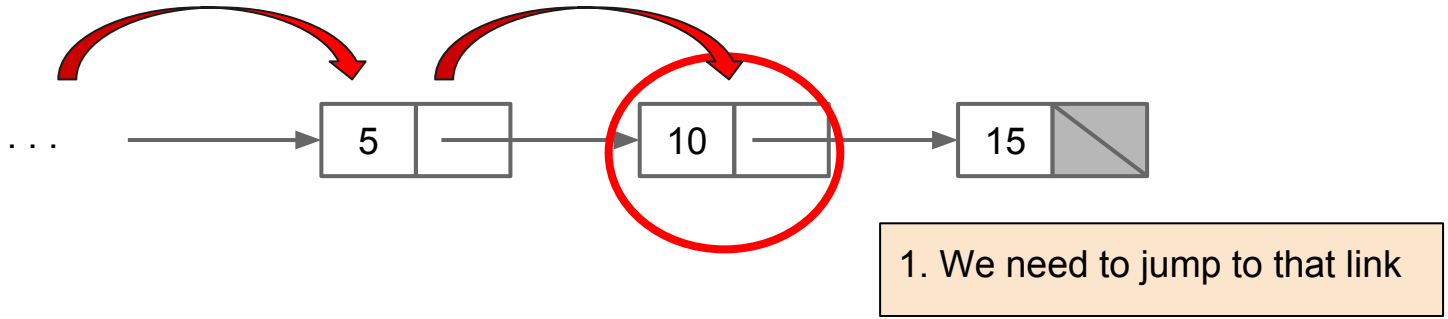
Review: DLLists

- One thing that a list should support is **remove(int i)** which removes the node whose item is i.
- With our brand new SLList with sentinel, what would we have to do to remove a link?



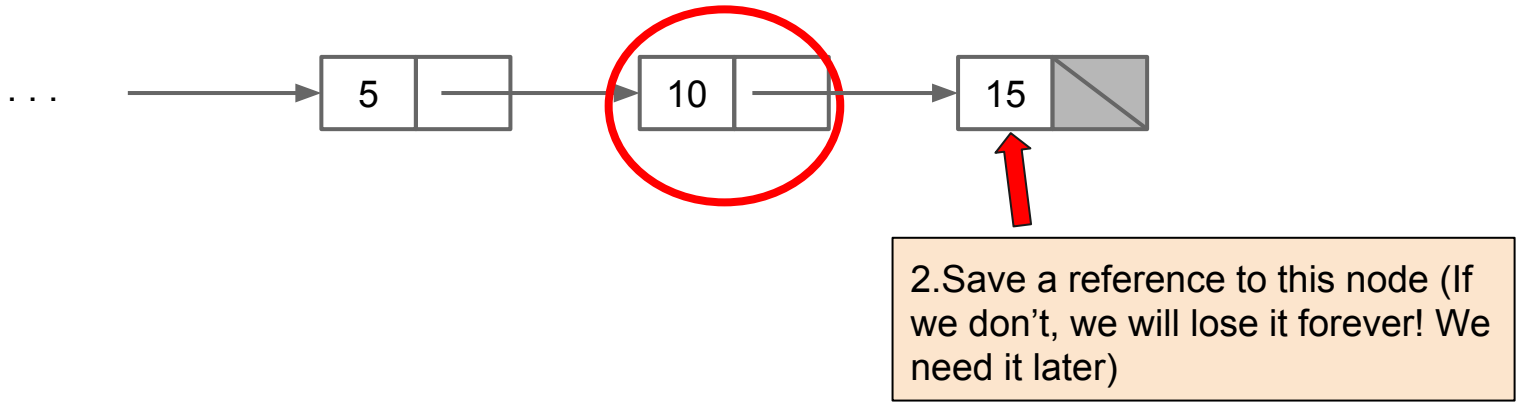
Review: DLLists

- One thing that a list should support is **remove(int i)** which removes the node whose item is i.
- With our brand new SLList with sentinel, what would we have to do to remove a link?



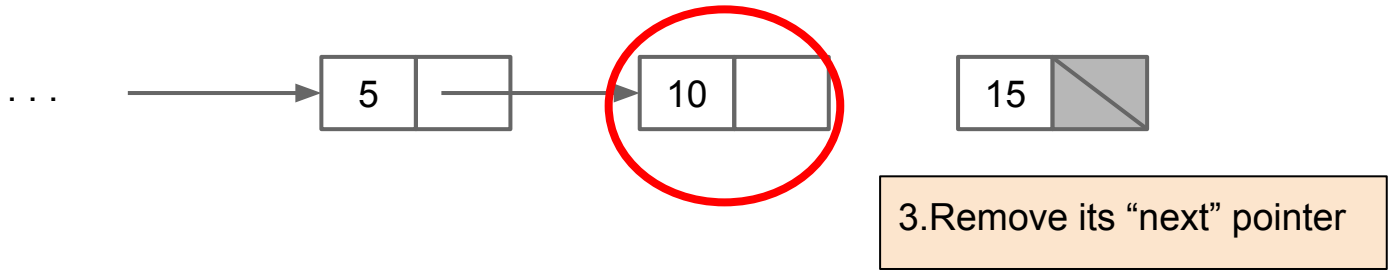
Review: DLLists

- One thing that a list should support is **remove(int i)** which removes the node whose item is i.
- With our brand new SLList with sentinel, what would we have to do to remove a link?



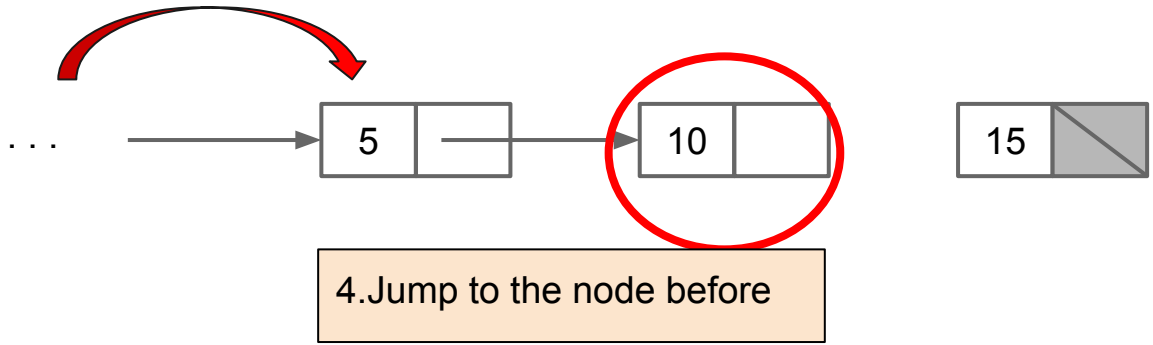
Review: DLLists

- One thing that a list should support is **remove(int i)** which removes the node whose item is i.
- With our brand new SLList with sentinel, what would we have to do to remove a link?



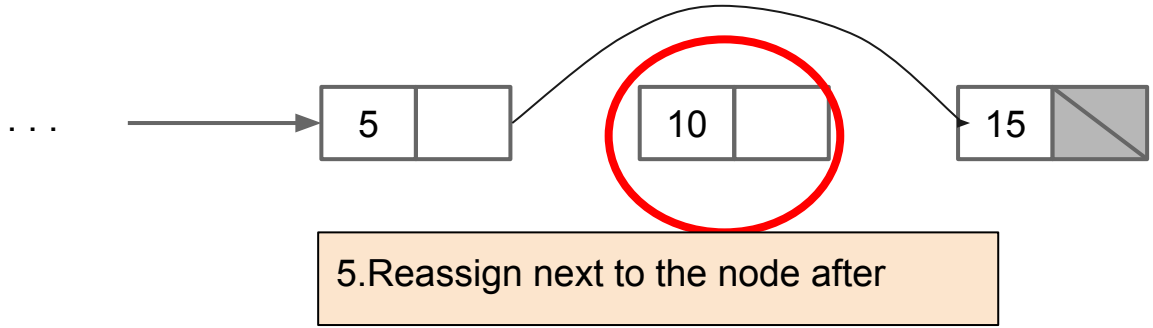
Review: DLLists

- One thing that a list should support is **remove(int i)** which removes the node whose item is i.
- With our brand new SLList with sentinel, what would we have to do to remove a link?



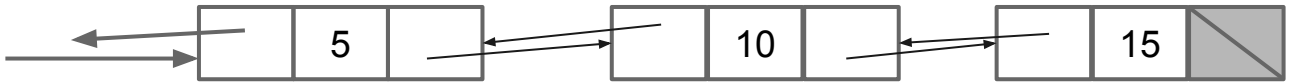
Review: DLLists

- One thing that a list should support is **remove(int i)** which removes the node whose item is i.
- With our brand new SLList with sentinel, what would we have to do to remove a link?



Review: DLLists

- We had to traverse the list **TWICE** in order to do the job. That's pretty inefficient.
- To remedy this, we simply add some more pointers!
- Instead of just having pointers that point to the "next" link, we add pointers that point to the "previous" link as well.
    - This allows you to traverse the list both forwards and backwards!



Review: ALists

- Even with the super sophisticated Doubly linked list, it'll still take a long time to traverse through the list.
- This makes us wish we could go back to arrays.. which have that sweeeet constant access time.
- But the bad thing about arrays is that you can't add or remove items.
- Let's write a list class that takes advantage of **both** an array's constant access time and a linked lists adding and removing feature.
- AList is a list whose underlying structure is an array. To support add and remove, we choose to **resize** the array when the array is too full which consists of:
    - Making a new array that has twice as much space (this is called geometric resizing)
    - Copying all the items from the old array into the new array

Exercise 1: insert

Implement `SLList.insert` which takes in an integer `x` and an integer `position`. It inserts `x` at the given `position`. If `position` is after the end of the list, insert the new node at the end.

For example, if the `SLList` is $5 \to 6 \to 2$, `insert(10, 1)` results in $5 \to 10 \to 6 \to 2$ and if the `SLList` is $5 \to 6 \to 2$, `insert(10, 7)` results in $5 \to 6 \to 2 \to 10$. Additionally, for this problem assume that `position` is a non-negative integer.

```
public void insert(int item, int position) {
```

- First, think about what types of things we need to check for (special cases).
- How do we get to the specified position?
- After we get there, how can we re-assign pointers such that we add the new node?

Exercise 1: insert

Implement `SLList.insert` which takes in an integer `x` and an integer `position`. It inserts `x` at the given `position`. If `position` is after the end of the list, insert the new node at the end.

For example, if the `SLList` is $5 \to 6 \to 2$, `insert(10, 1)` results in $5 \to 10 \to 6 \to 2$ and if the `SLList` is $5 \to 6 \to 2$, `insert(10, 7)` results in $5 \to 6 \to 2 \to 10$. Additionally, for this problem assume that `position` is a non-negative integer.

```
public void insert(int item, int position) {
    if (first == null || position == 0) {
        addFirst(item);
        return;
    }
    IntNode currentNode = first;
    while (position > 1 && currentNode.next != null) {
        position--;
        currentNode = currentNode.next;
    }
    IntNode newNode = new IntNode(item, currentNode.next);
    currentNode.next = newNode;
```

Exercise 2: reverse

1.2 Add another method to the `SLList` class that reverses the elements. Do this using the existing `IntNode` objects (you should not use **`new`**).

```
1  public void reverse() {
```

Exercise 2: reverse

1.2 Add another method to the `SLList` class that reverses the elements. Do this using the existing `IntNode` objects (you should not use **new**).

```
public void reverse() {
    if (first == null || first.next == null) {
        return;
    }

    IntNode ptr = first.next;
    first.next = null;

    while (ptr != null) {
        IntNode temp = ptr.next;
        ptr.next = first;
        first = ptr;
        ptr = temp;
    }
}
```

Exercise 2.5: reverse Extra

Extra: If you wrote `reverse` iteratively, write a second version that uses recursion (you may need a helper method). If you wrote it recursively, write it iteratively.

Exercise 2.5: reverse Extra

Extra: If you wrote `reverse` iteratively, write a second version that uses recursion (you may need a helper method). If you wrote it recursively, write it iteratively.

```
1  public void reverseRecur() {
2      first = reverseHelper(first);
3  }
4
5  private IntNode reverseHelper(IntNode lst) {
6      if (lst == null || lst.next == null) {
7          return lst;
8      } else {
9          IntNode endOfReversed = lst.next;
10         IntNode reversed = reverseHelper(lst.next);
11         endOfReversed.next = lst;
12         lst.next = null;
13         return reversed;
14     }
15 }
```

Exercise 3: Array insert

Consider a method that inserts an `int item` into an `int[] arr` at the given position. The method should return the resulting array. For example, if `x = [5, 9, 14, 15]`, `item = 6`, and `position = 2`, then the method should return `[5, 9, 6, 14, 15]`. If `position` is past the end of the array, insert `item` at the end of the array.

Is it possible to write a version of this method that returns void and changes `arr` in place (i.e., destructively)? *Hint:* These arrays are filled meaning an array containing `n` elements will have length `n`.

Exercise 3: Array insert

Consider a method that inserts an **`int`** `item` into an **`int`**`[]` `arr` at the given position. The method should return the resulting array. For example, if `x = [5, 9, 14, 15]`, `item = 6`, and `position = 2`, then the method should return `[5, 9, 6, 14, 15]`. If `position` is past the end of the array, insert `item` at the end of the array.

Is it possible to write a version of this method that returns void and changes `arr` in place (i.e., destructively)? *Hint:* These arrays are filled meaning an array containing `n` elements will have length `n`.

No, because arrays have a fixed size, so to add an element, you need to create a new array.

Exercise 3: Array insert Extra

Extra: Fill in the below according to the method signature:

```
1  public static int[] insert(int[] arr, int item, int position) {
```

Exercise 3: Array insert Extra

Extra: Fill in the below according to the method signature:

```
1  public static int[] insert(int[] arr, int item, int position) {
```

```
1       int[] result = new int[arr.length + 1];
2       position = Math.min(arr.length, position);
3       for (int i = 0; i < position; i++) {
4           result[i] = arr[i];
5       }
6       result[position] = item;
7       for (int i = position; i < arr.length; i++) {
8           result[i + 1] = arr[i];
9       }
10      return result;
11  }
```

Exercise 4: Array reverse

Consider a method that destructively reverses the items in `arr`. For example calling `reverse` on an array `[1, 2, 3]` should change the array to be `[3, 2, 1]`. Write the `reverse` method:

```
public static void reverse(int[] arr) {
```

Exercise 4: Array reverse

Consider a method that destructively reverses the items in `arr`. For example calling `reverse` on an array `[1, 2, 3]` should change the array to be `[3, 2, 1]`. Write the `reverse` method:

```
public static void reverse(int[] arr) {
    for (int i = 0; i < arr.length / 2; i++) {
        int j = arr.length - i - 1;
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

Exercise 5: Replicate Extra

Extra: Write a non-destructive method `replicate(int[] arr)` that replaces the number at index `i` with `arr[i]` copies of itself. For example, `replicate([3, 2, 1])` would return `[3, 3, 3, 2, 2, 1]`. For this question assume that all elements of the array are positive.

```
public static int[] replicate(int[] arr) {
```

Exercise 5: Replicate Extra

Extra: Write a non-destructive method `replicate(int[] arr)` that replaces the number at index `i` with `arr[i]` copies of itself. For example, `replicate([3, 2, 1])` would return `[3, 3, 3, 2, 2, 1]`. For this question assume that all elements of the array are positive.

```
public static int[] replicate(int[] arr) {
    int total = 0;
    for (int item : arr) {
        total += item;
    }
    int[] result = new int[total];
    int i = 0;
    for (int item : arr) {
        for (int counter = 0; counter < item; counter++) {
            result[i] = item;
            i++;
        }
    }
    return result;
}
```