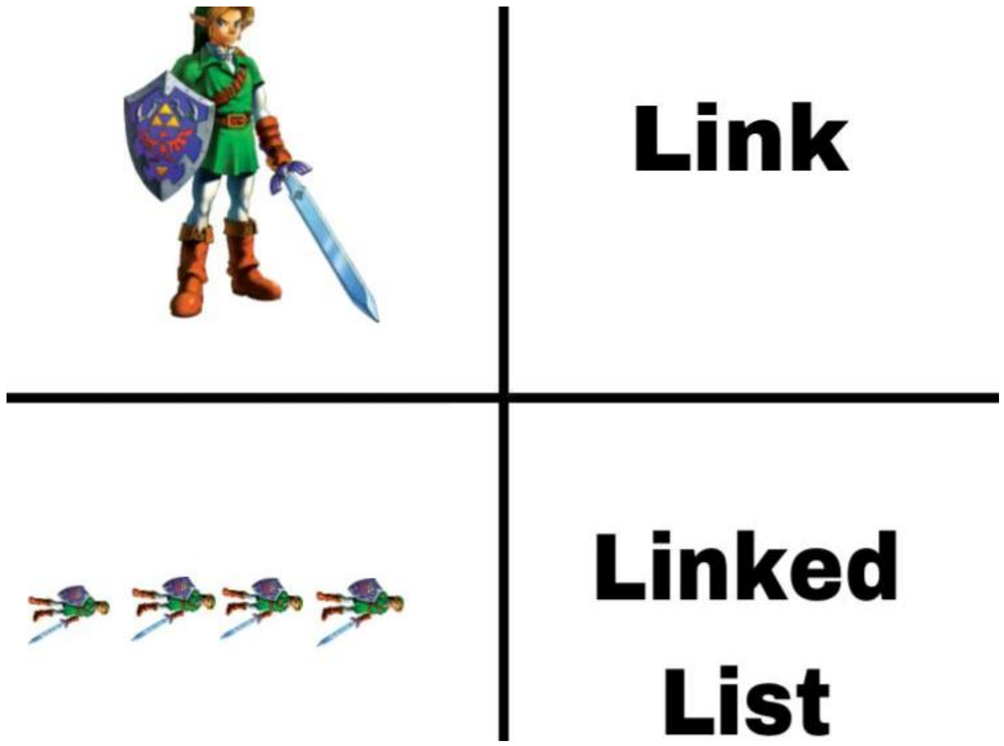
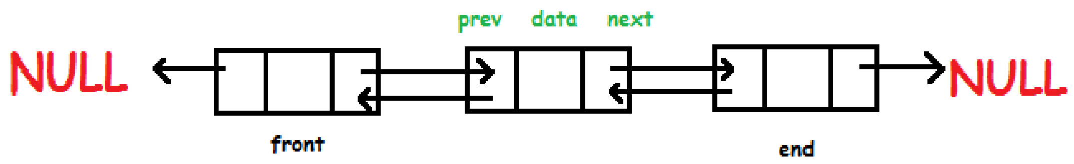


YEAH Hours A6 – Linked Lists

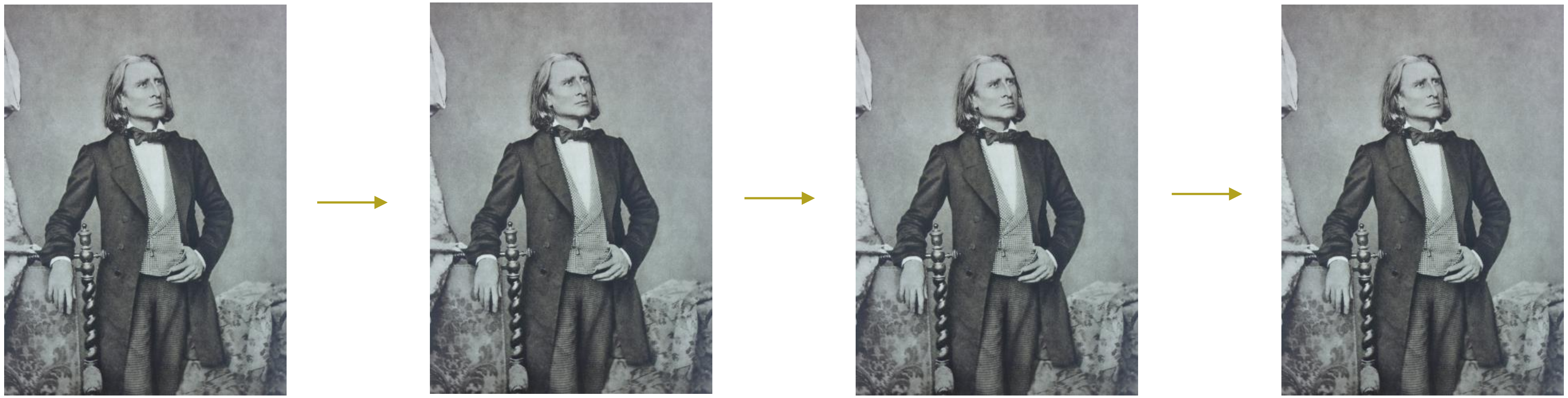
The pointers you know and love just got better!

Let's talk linked lists!

- A **Linked List** is simply a series of **structs** that are chained together using pointers.
- The specific **list node** that you interact with varies from project to project – sometimes you'll be working with quite sophisticated linked lists!
  - One example of this is a **doubly linked list**, a list where nodes store pointers to both the **next** and the **previous** nodes!



Before we start: questions about Linked Lists?



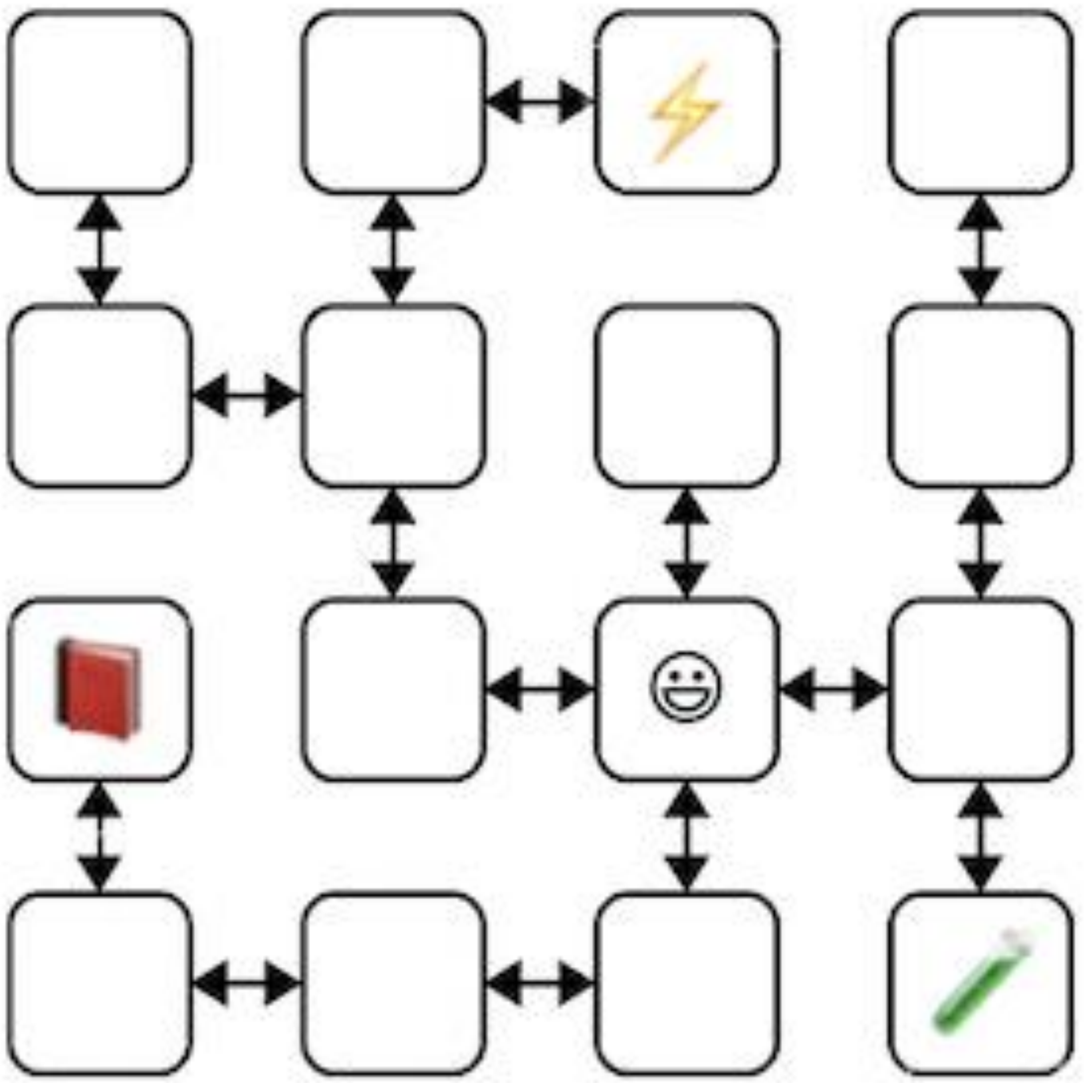
Look! A Linked Liszt!

What you'll need to do:

1. The Labyrinth
   - Using your pointer prowess, can you escape a twisty memory labyrinth?
2. Linked List warmups
   - Linked lists are hard. Here's why!
3. Sorting with Linked Lists!
   - Can you implement a classic sorting algorithm with a linked list?

The Labyrinth

- Imagine that you're placed in a labyrinth like the one on the right. In order to escape, you need to collect (up to) **three** magical items: **a book, a wand and a potion.**
- The labyrinth is constructed as a **linked list** with **four connections, one in each of the cardinal directions.**



The Labyrinth

- More specifically, the labyrinth is a **linked list** of **MazeCell** structs. Each cell has **four MazeCell neighbors** and a **string** that may or may not contain one of the enchanted items!

```
struct MazeCell {
    string contains;   // Either "", "Spellbook", "Potion", or "Wand"
    MazeCell *north;   // The cell to the north, or nullptr if can't go north.
    MazeCell *south;   // The cell to the south, or nullptr if can't go south.
    MazeCell *east;    // The cell to the east, or nullptr if can't go east.
    MazeCell *west;    // The cell to the west, or nullptr if can't go west.
};
```

The Labyrinth

You will need to write the following function:

```
bool isPathToFreedom(MazeCell *start, string path, Set<string> needed)
```

where **start** represents the initial MazeCell, **path** is a string consisting of characters **'N', 'S', 'E', 'W'**, and **needed** is a set of magic items that you need to escape the maze.

- For example, **start** could be any MazeCell *, **path** could look like "NSWWENEWSNEWSENNSNES", and **needed** could just contain "Wand"

- You will read a character at a time off the string and advance to the MazeCell dictated by the character ('N' --> curr = cur->north)
- Along the way, if any cells contain magic items, pick them up!

The Labyrinth

Some notes about isPathToFreedom()

- Not all MazeCells have 4 valid pointers. **Walls** in this world are determined by null pointers. If the following is true:

  ```
  if (curr->north == nullptr) { … }
  ```

  then there exists a wall above your current location. **If a path tells you to move into a wall, you should return false to signify that no escape was possible.**

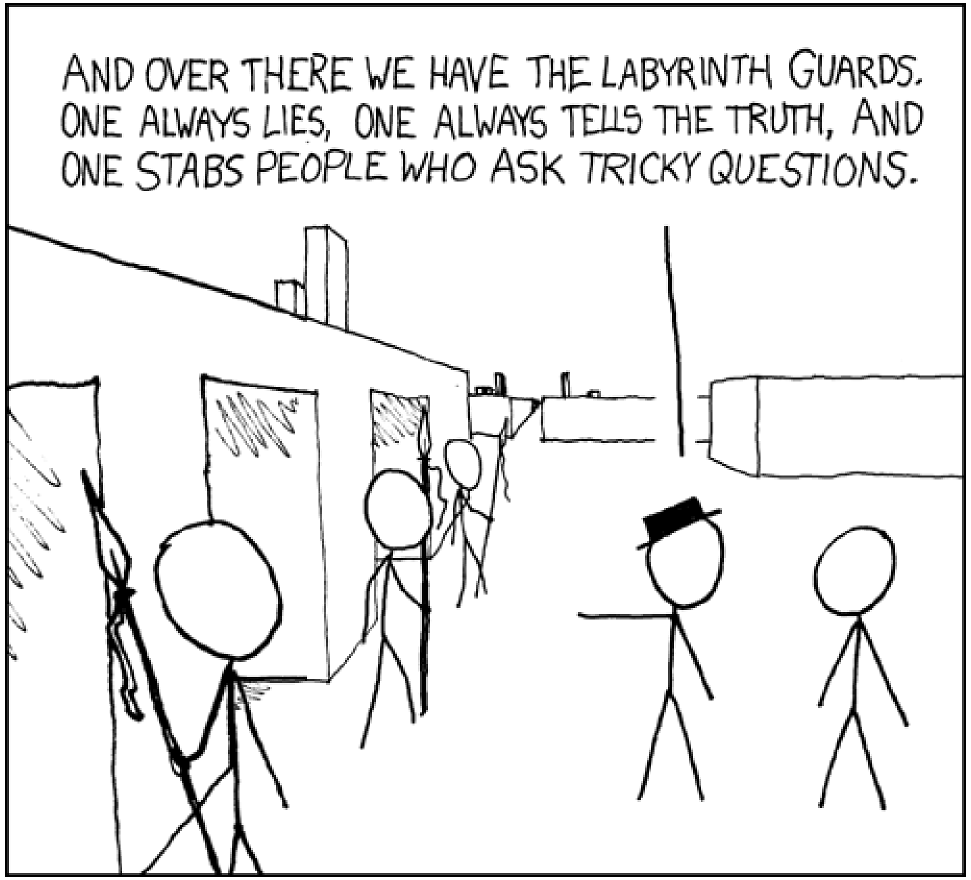
- You don't necessarily need all 3 magical items to escape – just however many are in 'needed' at the very beginning. You might find that you only need 1 or 0 items!
  - In a similar vein, you might find that you have all the items you need well before you've exhausted the path – that's okay
- It is possible that you encounter invalid characters in your path string. If you do, throw an error to signify an invalid path.

The Labyrinth

A few more notes:

- Please use **iteration** and **not recursion**. Although your recursive gears might be grinding, we don’t want to create tons of stack frames here.
- The path you are given may have you visiting the same cell twice. This is okay, and you don’t need to detect it.
- **Do not** allocate any new MazeCell structs with the **new** keyword. You shouldn't need to, but thought I should get that out there…
- The order of the items claimed doesn’t matter.

Questions about isPathToFreedom?



I don't really get this one but it's topical, and we won't ask any questions that are too tricky ☺

Labyrinth part II: Escape!

- Now it's time for you to escape from your own labyrinth! You'll use the function you've just written to escape from a labyrinth personalized to **you!** At the top of your **labyrinth.cpp** file, enter your name as the value of the constant **kYourName**.

```
const string kYourName = "Trip";
```

- Now scroll down to the final test case in the file. Set a **breakpoint** somewhere in this test and fire up the debugger!

```
PROVIDED_TEST("Escape from your personal labyrinth!") {
    Set<string> allThree = {"Spellbook", "Potion", "Wand"};
    /* A maze for you to escape from. This maze will be personalized
     * based on the constant kYourName.
     */
    EXPECT(kYourName != "TODO");
    MazeCell* startLocation = mazeFor(kYourName);

    /* Set a breakpoint here. As a reminder, the labyrinth you
     * get will be personalized to you, so don't start exploring the labyrinth
     * unless you've edited the constant kYourName to include your name(s)!
     * Otherwise, you'll be escaping the wrong labyrinth.
     */
    EXPECT(kPathOutOfNormalMaze != "TODO");
    EXPECT(isPathToFreedom(startLocation, kPathOutOfNormalMaze, allThree));
}
```

Labyrinth part II: Escape!

- When you fire up the debugger, you'll find yourself with a debugger pane on the right that looks something like this:

▸ allThree	@0x4e5f5fc	stanfordcpplib::collections::GenericSet<stanfordcpplib::collections::SetTraits<std::string>>
▾ startLocation	@0x52f9250	MazeCell
contains	""	std::string
east	0x0	MazeCell *
▸ north	@0x5412e30	MazeCell
south	0x0	MazeCell *
west	0x0	MazeCell *

Disclaimer: These were taken from my ~~crappy~~ windows machine. Not sure if they'll be 100% identical on mac (or linux if you're into that sort of thing)

Doesn't look like there are magical items at my starting point, rats! Looks like I'll need to examine my neighbors! In this case, there are **walls** all around, so I can only look **north.** Let's click on it and see what we can find.

Labyrinth part II: Escape!

- When you fire up the debugger, you'll find yourself with a debugger pane on the right that looks something like this:

allThree	@0x4e5f5fc	stanfordcpplib::collections::GenericSet<stanfordcpplib::collections::SetTraits<std::string>>
startLocation	@0x52f9250	MazeCell
contains	""	std::string
east	0x0	MazeCell *
north	@0x5412e30	MazeCell
south	0x0	MazeCell *
west	0x0	MazeCell *

Disclaimer: These were taken from my ~~crappy~~ windows machine. Not sure if they'll be 100% identical on mac (or linux if you're into that sort of thing)

Doesn't look like there are magical items at my starting point, rats! Looks like I'll need to examine my neighbors! In this case, there are **walls** all around, so I can only look **north.** Let's click on it and see what we can find.

Labyrinth part II: Escape!

▸ allThree	@0x4e5f5fc	stanfordcpplib::collections::GenericSet<stanfordcpplib::collections::SetTraits<std::string>>
▾ startLocation	@0x52f9250	MazeCell
contains	""	std::string
east	0x0	MazeCell *
▾ north	@0x5412e30	MazeCell
contains	""	std::string
east	0x0	MazeCell *
north	0x0	MazeCell *
▸ south	@0x52f9250	MazeCell
▸ west	@0x57062c0	MazeCell
south	0x0	MazeCell *
west	0x0	MazeCell *

This is the contents of our northern neighbor! **Watch out!** It's easy for this window to get cluttered quickly!

- Nothing here either? Double rats! From here, you can keep poking around the debugger. **We highly recommend drawing out a picture of your labyrinth. For every location you examine, mark it in your picture, including any items that might be there! If you don't do this, remembering the correct path to find all 3 items will be very difficult.**

Labyrinth part II: Escape!

- Eventually, you'll find an item, huzzah! Once you've found all 3, **refer to your drawing,** and construct a path, from the start location, of the series of steps needed to pick up all 3 items. **Denote each step as a character,** ('N' -> North), and when you're convinced you have a correct path string, set the constant **kPathOutOfNormalMaze** to your result string. Then run in non-debug mode and *voila*, you're out of the maze!

```
▾ west                @0x55a5f48    MazeCell
    contains          ""            std::string
  ▸ east              @0x54c44d0    MazeCell
    north             0x0           MazeCell *
    south             0x0           MazeCell *
  ▾ west              @0x54cc7f8    MazeCell
      contains        ""            std::string
    ▸ east            @0x55a5f48    MazeCell
      north           0x0           MazeCell *
    ▾ south           @0x55a2a08    MazeCell
        contains      ""            std::string
        east          0x0           MazeCell *
      ▸ north         @0x54cc7f8    MazeCell
      ▾ south         @0x53d0cf0    MazeCell
          contains    ""            std::string
        ▾ east        @0x54cc6f8    MazeCell
          ▸ contains  "Spellbook"   std::string
            east      0x0           MazeCell *
            north     0x0           MazeCell *
            south     0x0           MazeCell *
          ▸ west      @0x53d0cf0    MazeCell
```

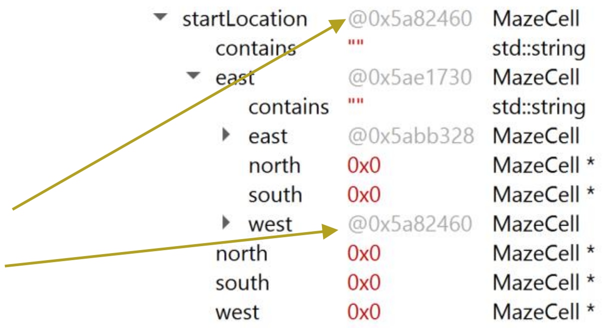
Labyrinth part II: Escape!

Some notes about the question:

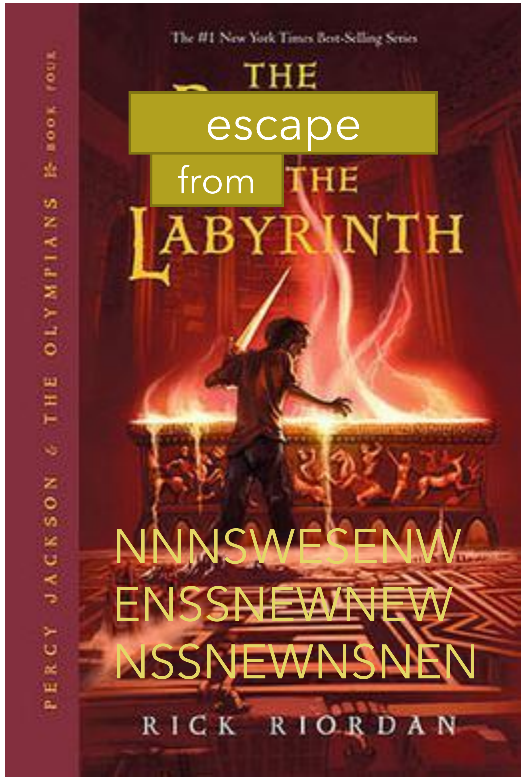
- If you change the **kYourName** constant, you'll get a brand new maze, so keep that in mind if you have to change the name!
- Beware that the labyrinths you are given may have **cycles** in them, and paths may one be uni-directional! Check the addresses of the neighbor pointers to see if they match an above neighbor! If they do, you might be going in a circle!

Not sure I'd call this a cycle, but you can see that the address

Is repeated in 2 places!



Questions about Labyrinth Escape?



Part II: Debugging Warmups

- In this part, you will use the **simple test** framework to detect **memory leaks**!
- The **TRACK_ALLOCATIONS_OF** addendum in the ListNode struct definition will automatically record the number of ListNodes that have been **allocated** and **deleted**. If the numbers don't match up at the end of the program, it'll give you an error!

```
struct ListNode {
    int data;
    ListNode *next

    TRACK_ALLOCATIONS_OF(ListNode);
};
```

Part II: Debugging Warmups

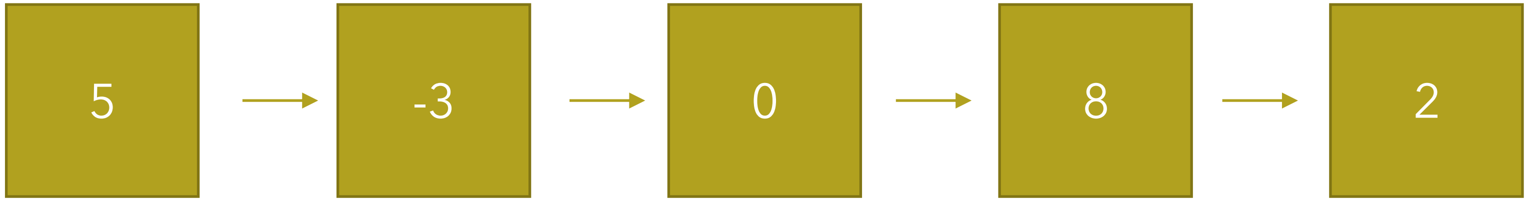
- You will be running some programs in **warmup.cpp** that contain various **memory errors** relating to **linked lists.** In the process of observing them, you'll learn that some errors are quite noticeable, but others are virtually imperceptible without some help. Spooky!
- In this part, you'll see **memory leaks, use-after-free errors, and segmentation faults!** Don't worry, you're ready to face them all!

Part III: Sorting with Linked Lists

- It's time for your big challenge! For this final part, you are tasked with implementing **either mergesort** or **quicksort** using a **linked list** instead of an array!
  - You only need to do one! If you do both, you'll get ~extra credit~
- Let's talk about **mergesort** first!

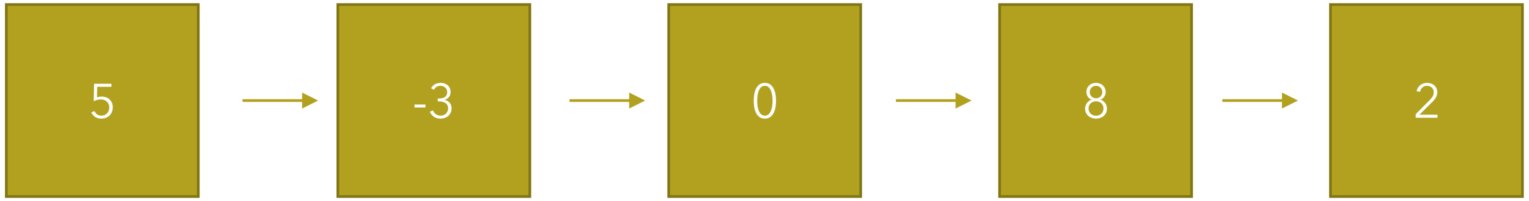
MergeSort Case Study

- Let's say that you want to perform **MergeSort** on this here list.



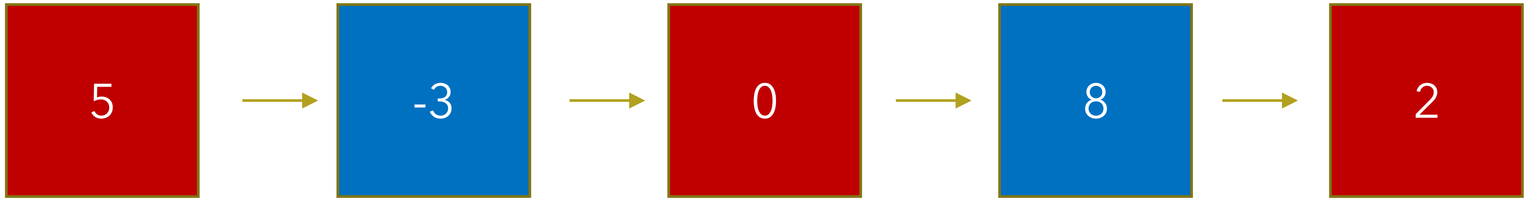
MergeSort Case Study

- Step 1: Split this list into **two linked lists.** To do so, you must **distribute elements in an alternating fashion:**



MergeSort Case Study

- Step 1: Split this list into **two linked lists.** To do so, you must **distribute elements in an alternating fashion:**

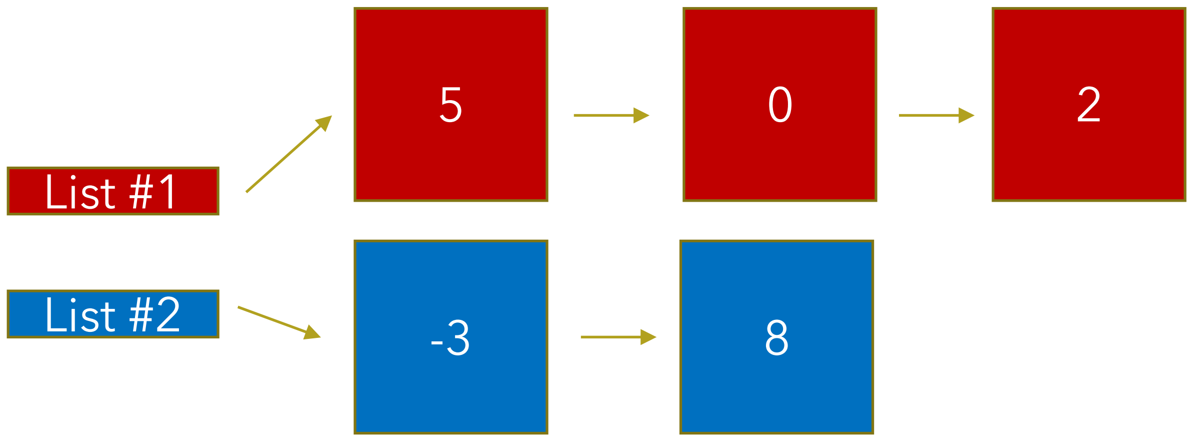


List #1

List #2

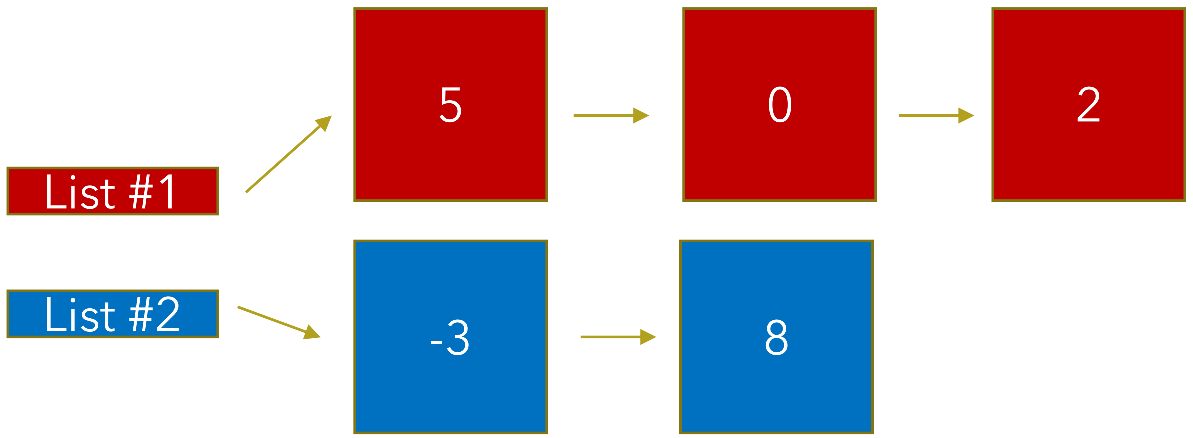
MergeSort Case Study

- Step 1: Split this list into **two linked lists.** To do so, you must **distribute elements in an alternating fashion:**



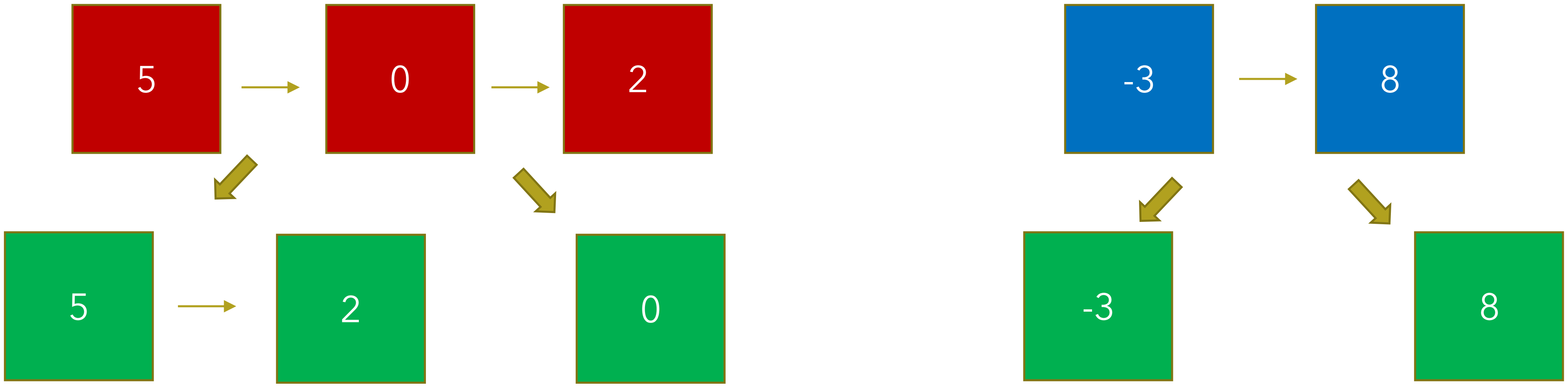
MergeSort Case Study

- Step 1: **Split** this list into **two linked lists.** To do so, you must **distribute elements in an alternating fashion:**
- Step 2: Once you have these sublists, you'll want to **recursively split these lists.** Think back to **Multiway Merge**! Feeling nostalgic?



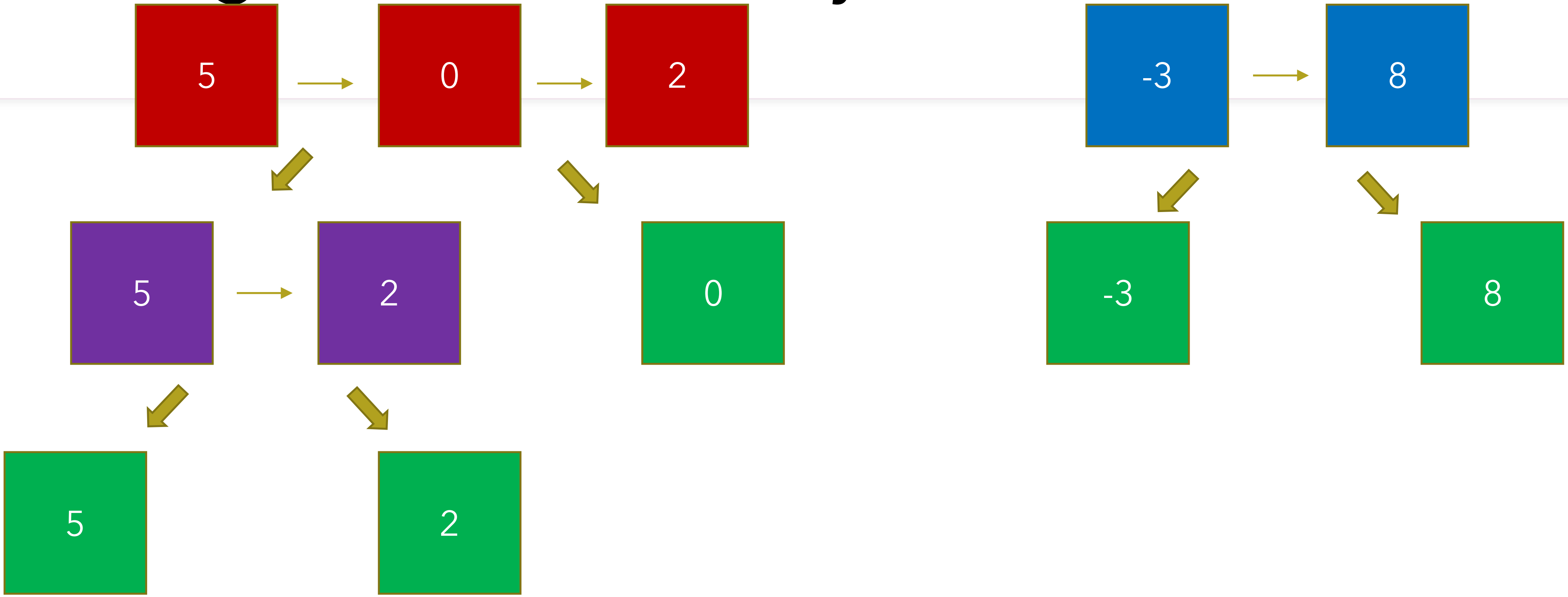
MergeSort Case Study

- Step 1: **Split** this list into **two linked lists.** To do so, you must **distribute elements in an alternating fashion:**
- Step 2: Once you have these sublists, you'll want to **recursively split these lists.** Think back to **Multiway Merge**! Feeling nostalgic?

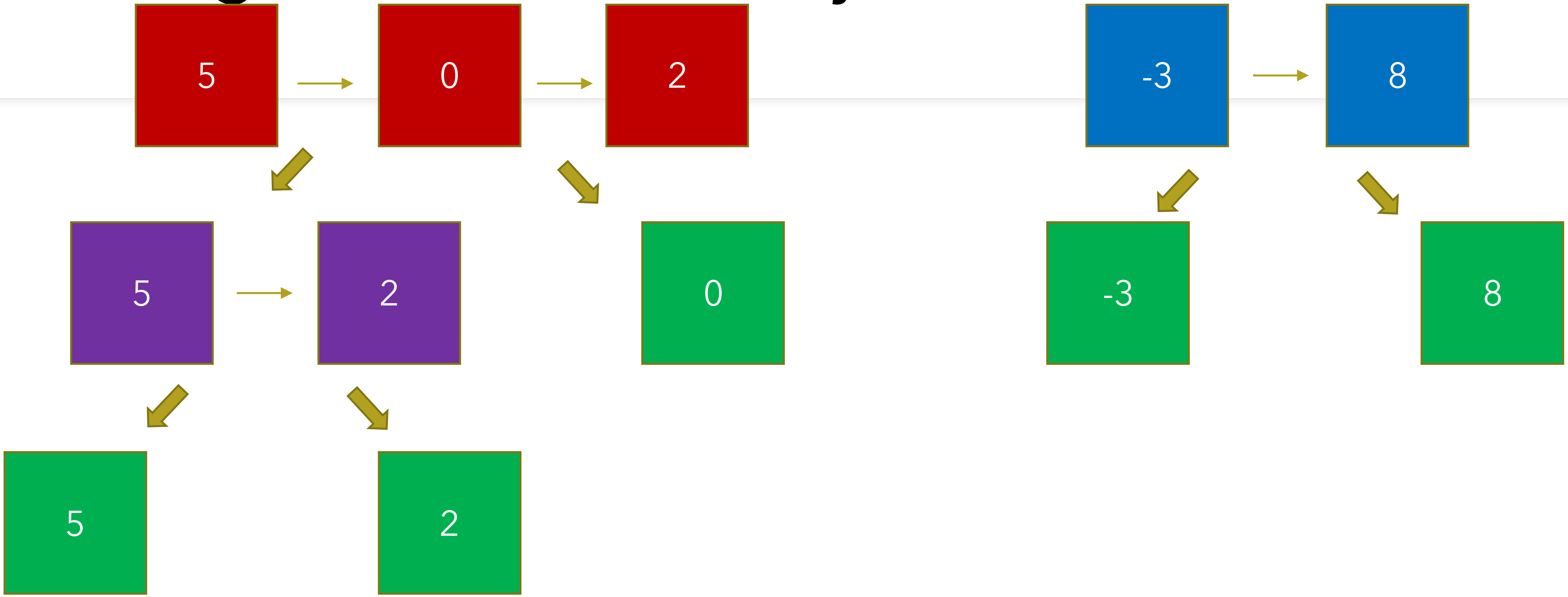


Notice how the order changed here – when we split our list we need to choose alternating elements!

MergeSort Case Study



MergeSort Case Study



The green nodes here have all been split down to base-case level!

MergeSort Case Study

Step 3: Merge time!
(in order!)

0

-3

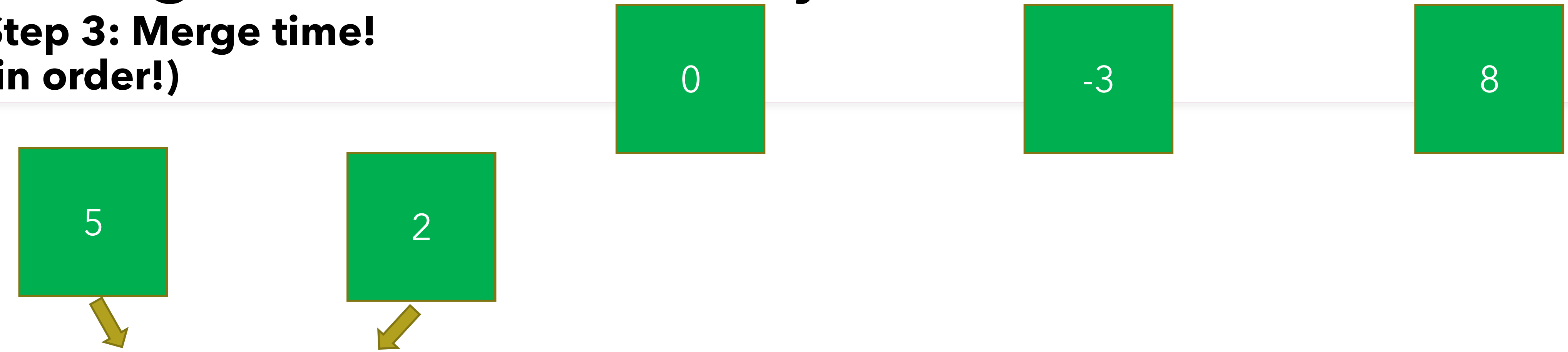
8

5

2

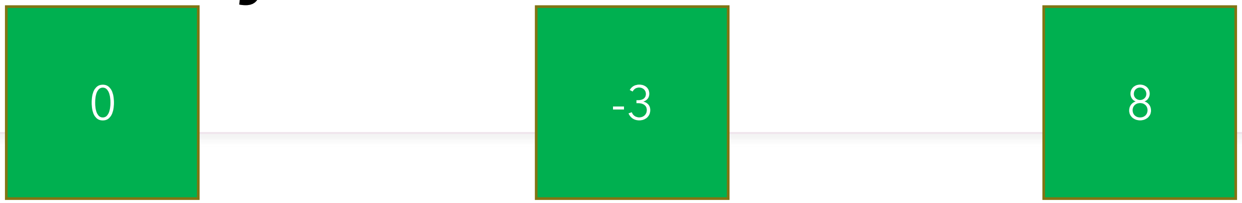
MergeSort Case Study

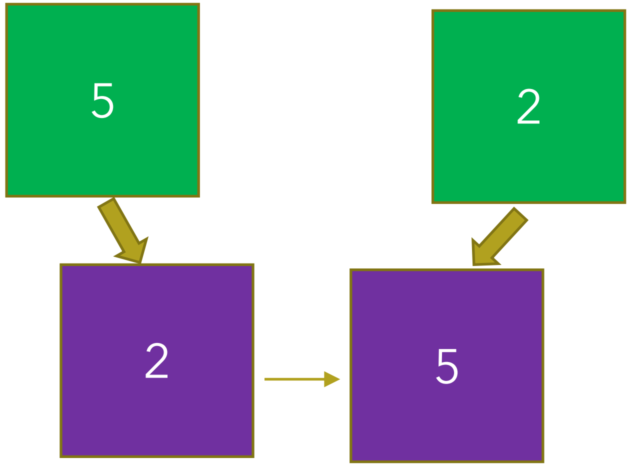
Step 3: Merge time!
(in order!)



MergeSort Case Study

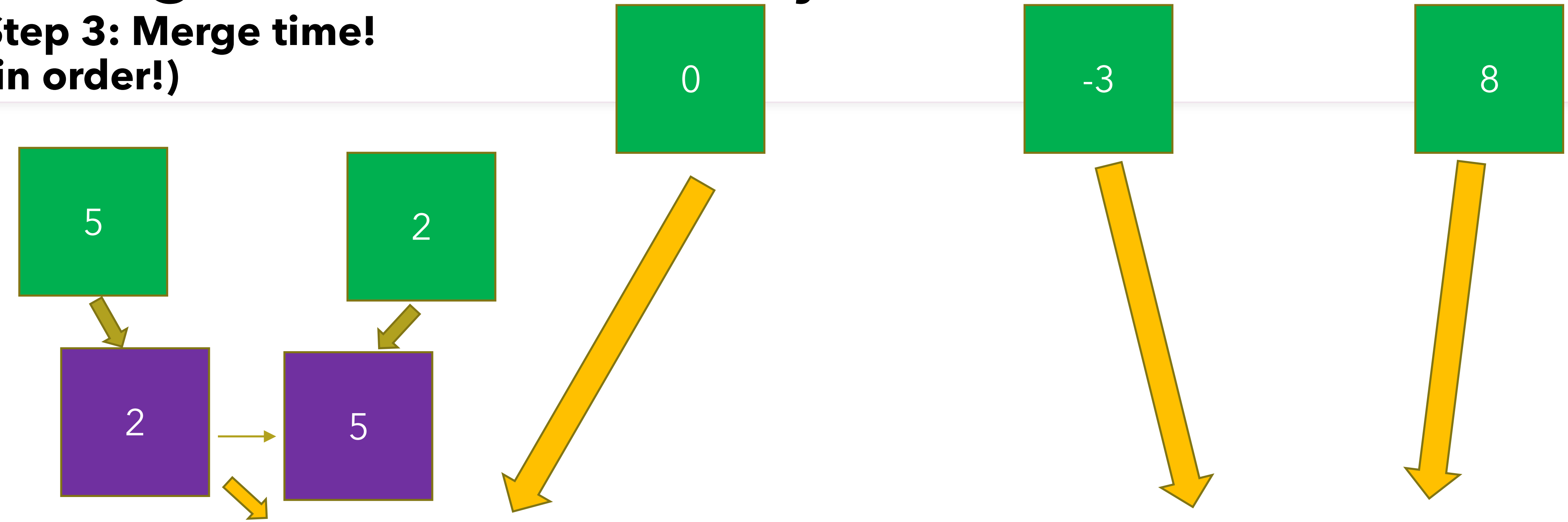
Step 3: Merge time!
(in order!)





MergeSort Case Study

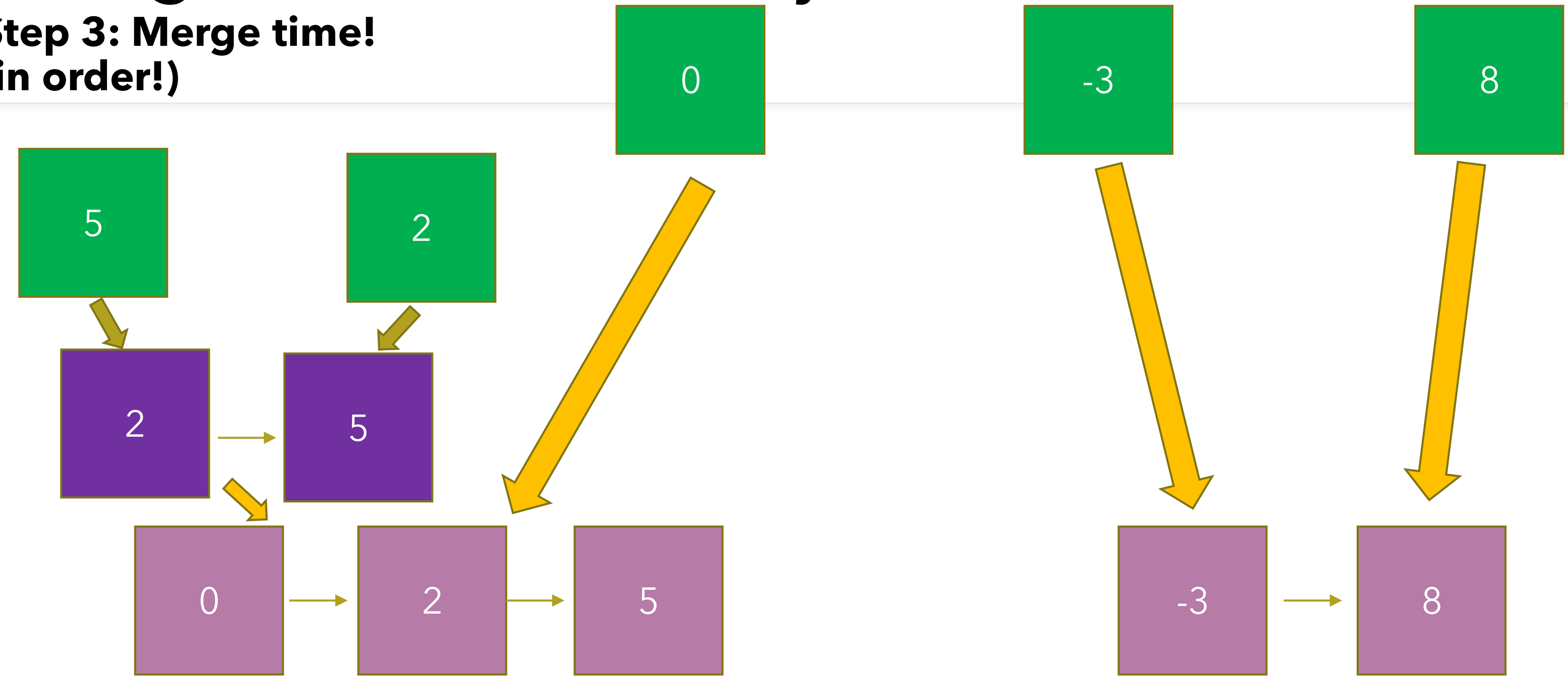
Step 3: Merge time!
(in order!)



Can you tell I'm running out of space ☺

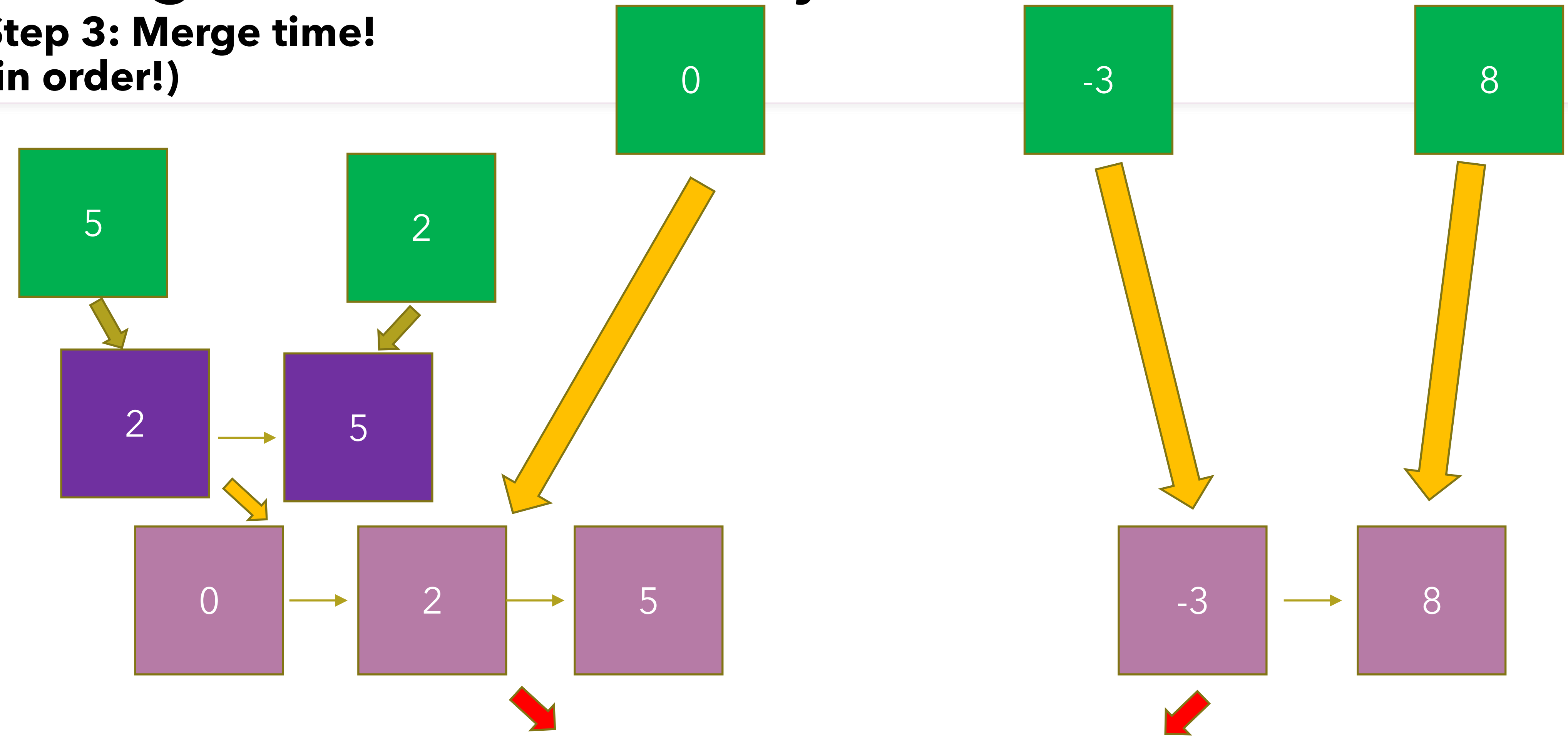
MergeSort Case Study

Step 3: Merge time!
(in order!)



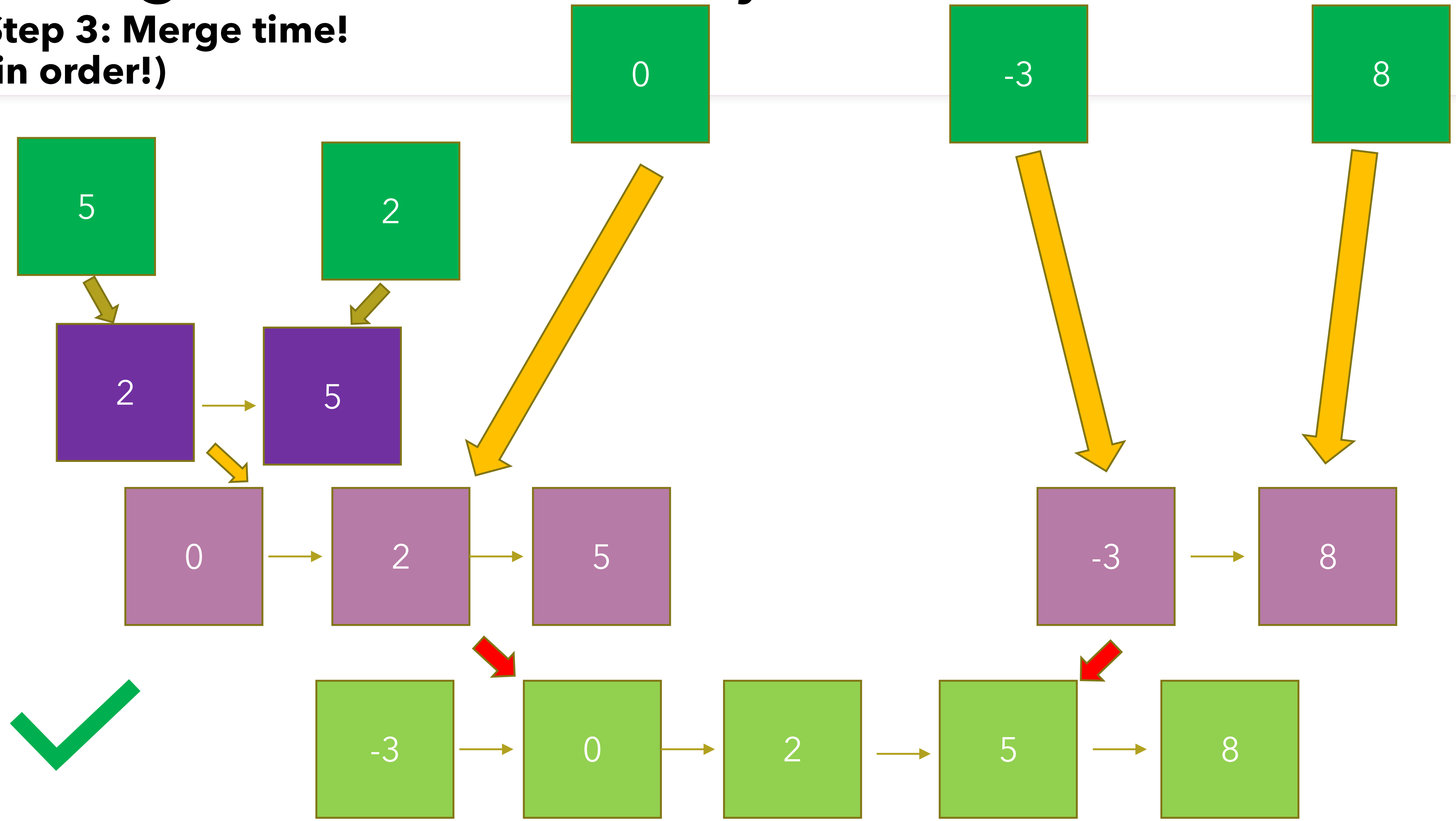
MergeSort Case Study

Step 3: Merge time!
(in order!)



MergeSort Case Study

Step 3: Merge time!
(in order!)



Done!

MergeSort Case Study

- Step 1: **Split** this list into **two linked lists.** To do so, you must **distribute elements in an alternating fashion:**
- Step 2: Once you have these sublists, you'll want to **recursively split these lists.** Think back to **Multiway Merge**! Feeling nostalgic?
- Step 3: Finally, you'll want to merge the result of your **mergesort calls** on **list 1** and **list 2.**
  - This is very similar to the merge in **Multiway** – you can assume that the lists you are merging are sorted, because, by starting by merging single elements, you will **always** be creating sorted sublists!

Part III: Sorting with Linked Lists

Some tips / tricks for **MergeSort**

- **You're not given much to go off for this part.** If you're confused about how to start this one, looking at **Multiway Merge** would be a good idea. The structure of the algorithm is virtually the same, just swap Vectors and Queues for Linked Lists.
- **I found that** writing helper functions like **printList()** and **listToVector()** were really helpful for debugging and testing purposes. Because MergeSort has many steps, being able to verify that an individual part works can be crucial!
  - On that note, **we don't provide ANY functionality tests for this part. It's up to you to write a barrage of tests to verify the robustness of your sort.** Once again, **listToVector()** will be helpful here in order to compare your sorting algorithm to the built-in vector.sort() algorithm.
- This goes without saying, but **decomposition is crucial here**. You need to be able to test your **merge** and **divide** routines separately in order for this assignment to be manageable. If you don't test incrementally, it will be very hard to tell where your bugs are coming from!

Part III: Sorting with Linked Lists

Things to watch out for:

- Although you may end up calling your **MergeSort** function **recursively**, the routine for dividing a list into two sublists **must be done iteratively**, and **so must the merge function**.
  - Because of **recursion's** stack-frame-intensive nature, we don't want you to blow out your stack on a simple sort!
- You are not allowed to **add or remove any ListNodes**. The sorting must be done by **rewiring nodes** only! **You may not modify the "data" field in the ListNode.**
- This might go without saying, but you are not allowed to use data structures like **Vectors** or **Stacks** in your implementation.
  - **Vectors** may be **very very very helpful** for debugging, however!
- **Segmentation faults**. I'll just leave this here…

Questions about MergeSort?

```
struct thank_you {
    int ex;
    thank_you *next;
};
```

In case anyone wanted to thank_you->next

QuickSort Case Study

- Let's talk about **Quicksort**! Although you may be less familiar with **QuickSort** as an algorithm, I actually found it **slightly easier** to implement. This may not be true for you all, given that you just implemented **Multiway Merge**, but I do believe the code for **QuickSort** is a little less hairy.

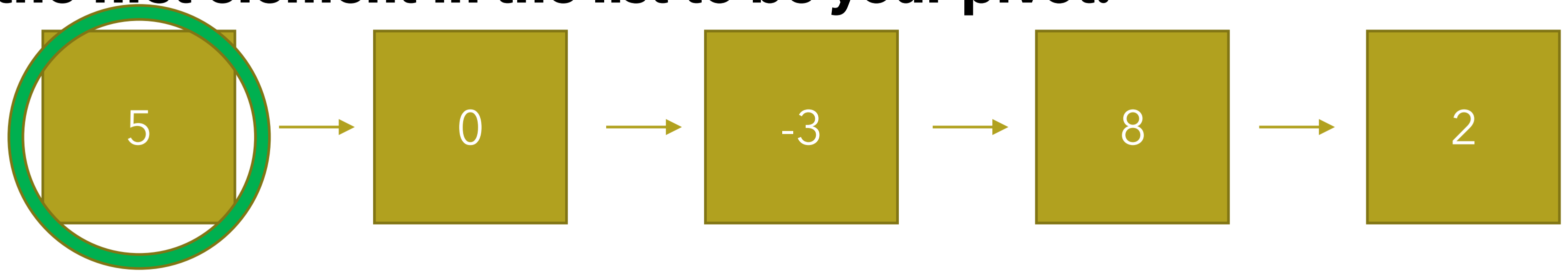
- Let's jump in!

QuickSort Case Study

- Step 1: Choose a **pivot**. The **pivot** will be one element in the list that will act as your **dividing element**, splitting the list into two (three if you count the pivot separately) lists. Choosing a good pivot can be tricky, but for this assignment, **you simply have to pick the first element in the list to be your pivot.**

QuickSort Case Study

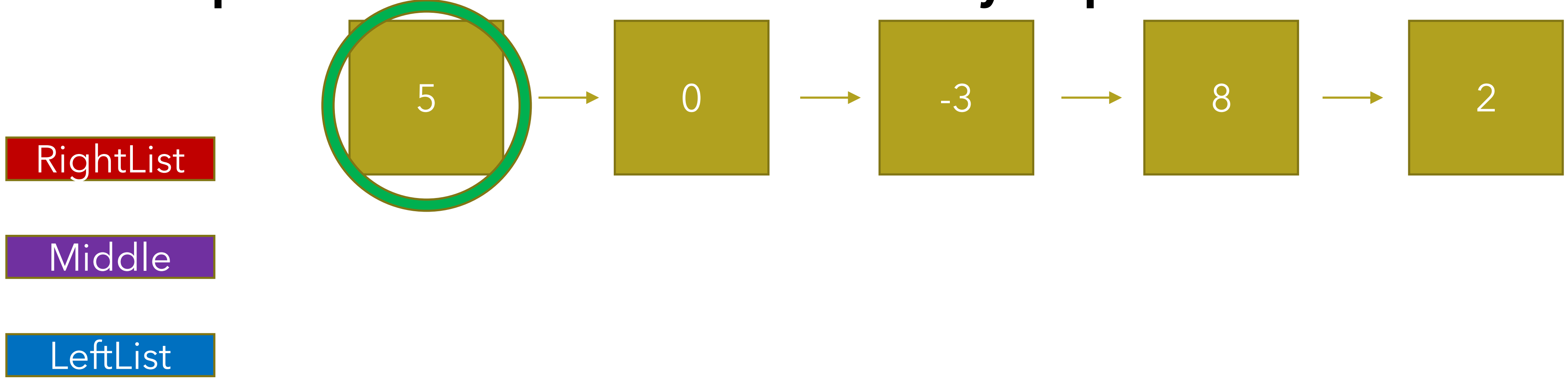
- Step 1: Choose a **pivot**. The **pivot** will be one element in the list that will act as your **dividing element**, splitting the list into two (three if you count the pivot separately) lists. Choosing a good pivot can be tricky, but for this assignment, **you simply have to pick the first element in the list to be your pivot.**



How pivotal!

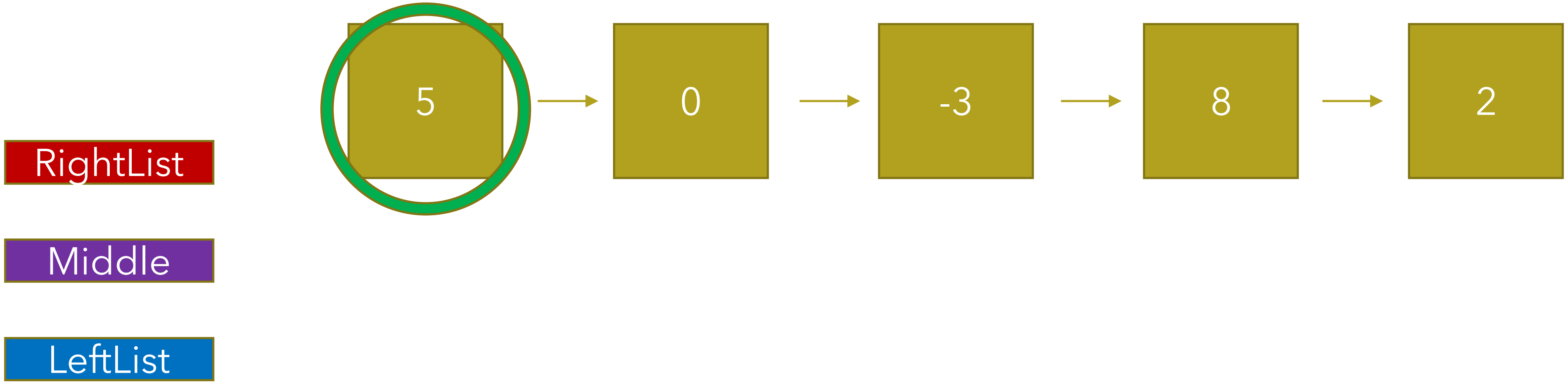
QuickSort Case Study

- Step 1: Choose a **pivot**. The **pivot** will be one element in the list that will act as your **dividing element**, splitting the list into two (three if you count the pivot separately) lists. Choosing a good pivot can be tricky, but for this assignment, **you simply have to pick the first element in the list to be your pivot.**



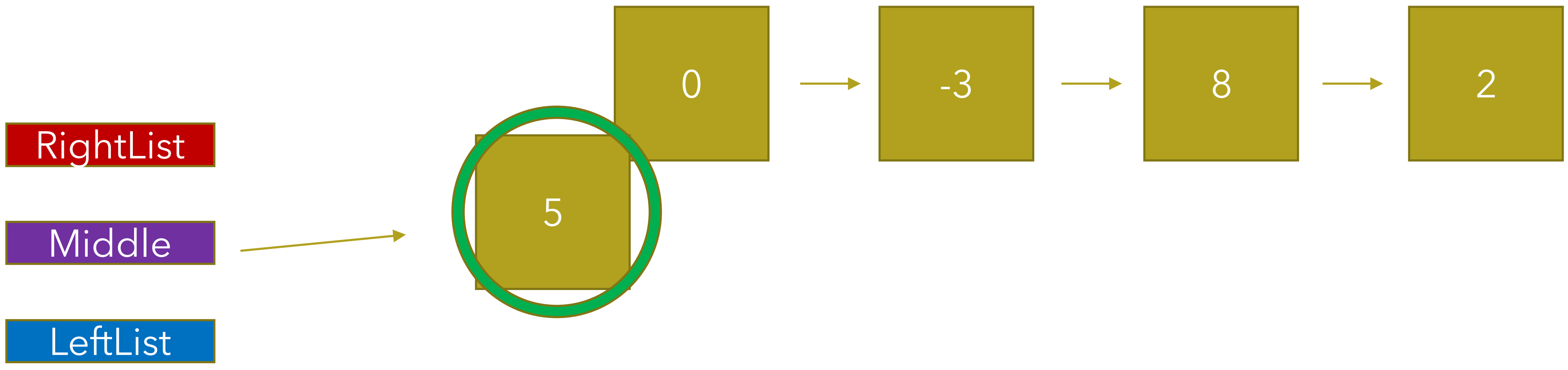
QuickSort Case Study

- Step 1.5: With your **pivot** in hand, do a linear scan of the list, assigning elements to the correct sublist depending on their relation to the **pivot** (less, greater, equal)



QuickSort Case Study

- Step 1.5: With your **pivot** in hand, do a linear scan of the list, assigning elements to the correct sublist depending on their relation to the **pivot** (less, greater, equal)

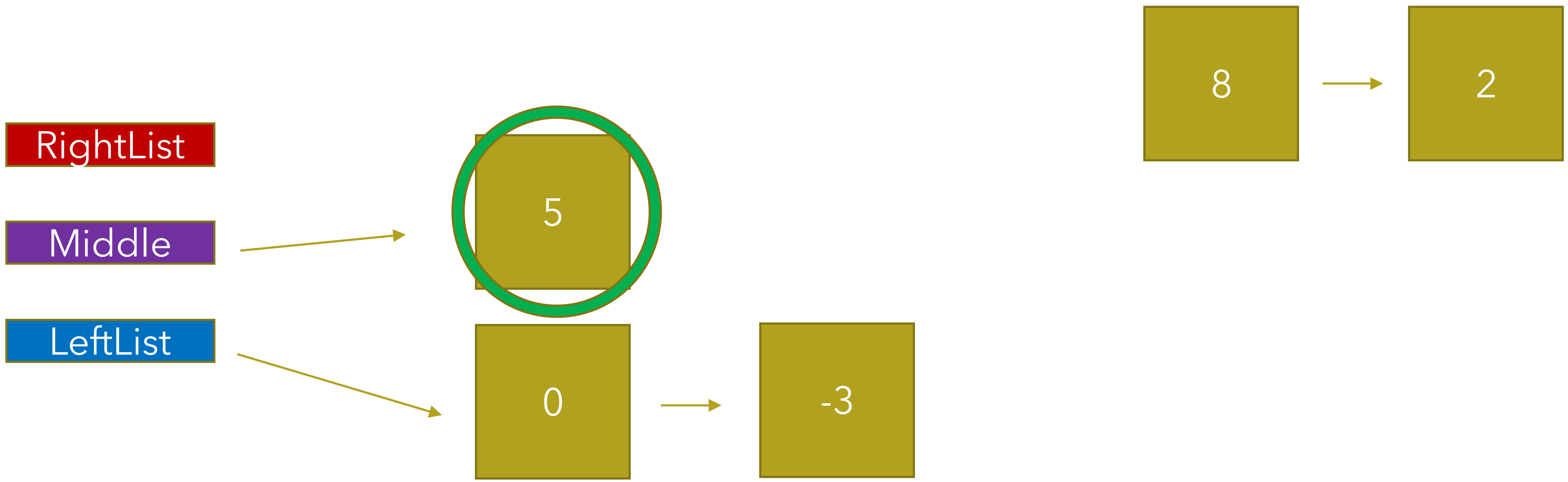


QuickSort Case Study

- Step 1.5: With your **pivot** in hand, do a linear scan of the list, assigning elements to the correct sublist depending on their relation to the **pivot** (less, greater, equal)

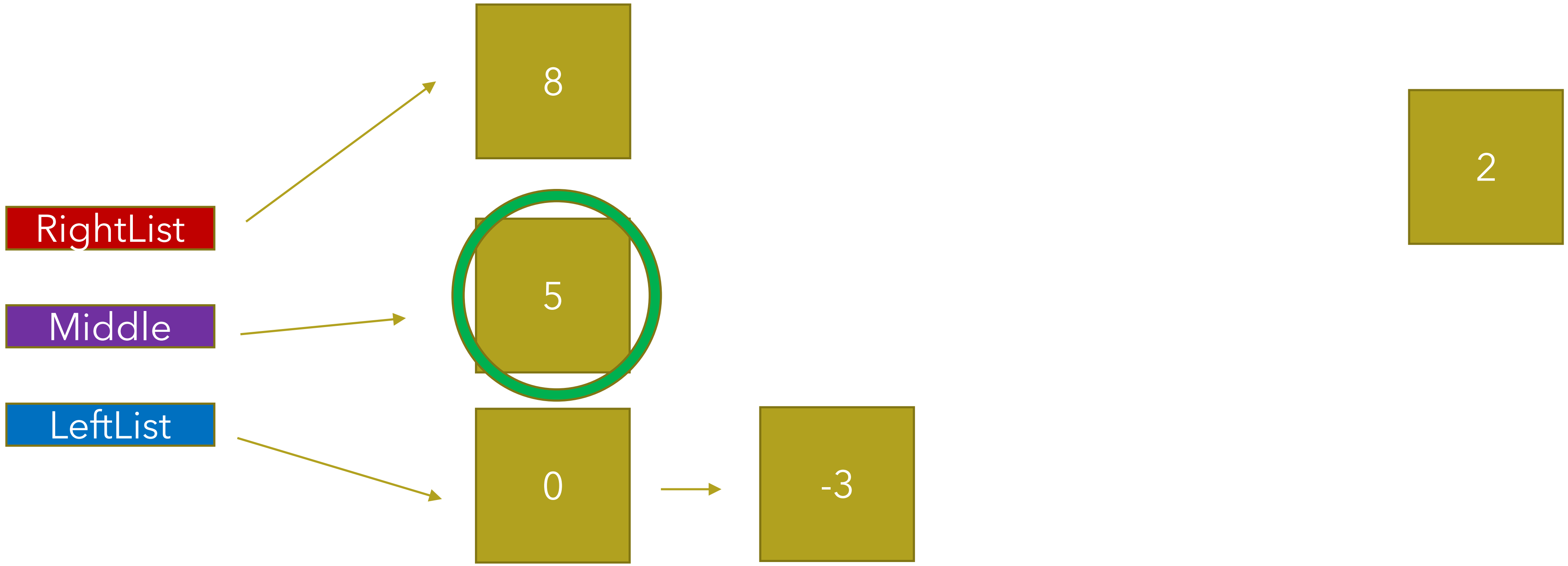
QuickSort Case Study

- Step 1.5: With your **pivot** in hand, do a linear scan of the list, assigning elements to the correct sublist depending on their relation to the **pivot** (less, greater, equal)



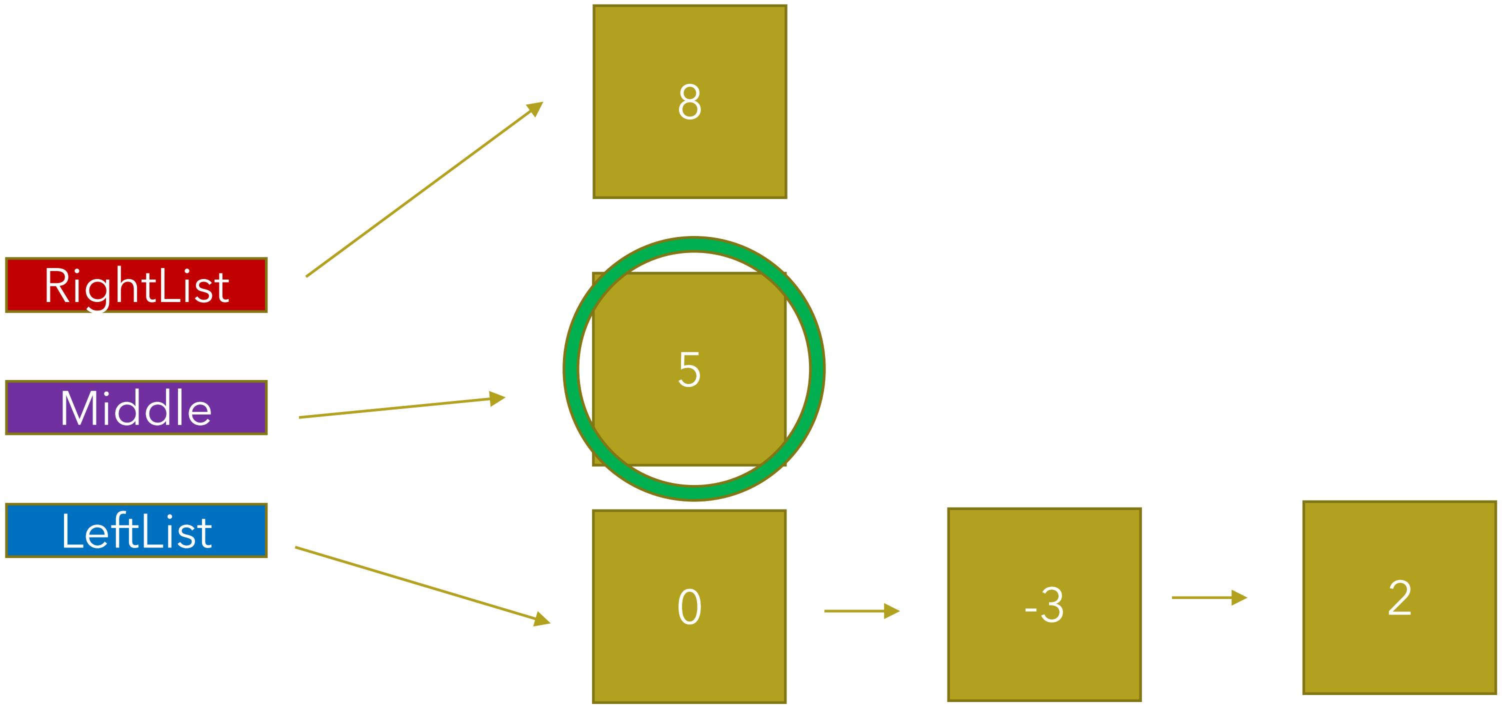
QuickSort Case Study

- Step 1.5: With your **pivot** in hand, do a linear scan of the list, assigning elements to the correct sublist depending on their relation to the **pivot** (less, greater, equal)



QuickSort Case Study

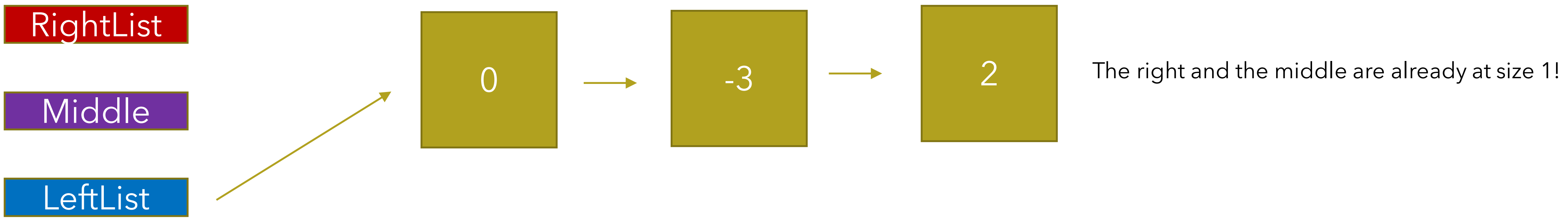
- Step 1.5: With your **pivot** in hand, do a linear scan of the list, assigning elements to the correct sublist depending on their relation to the **pivot** (less, greater, equal)



ho hum, we've chosen a shoddy pivot. Want to learn how to choose a great pivot while not burning the efficiency books? Take CS161!

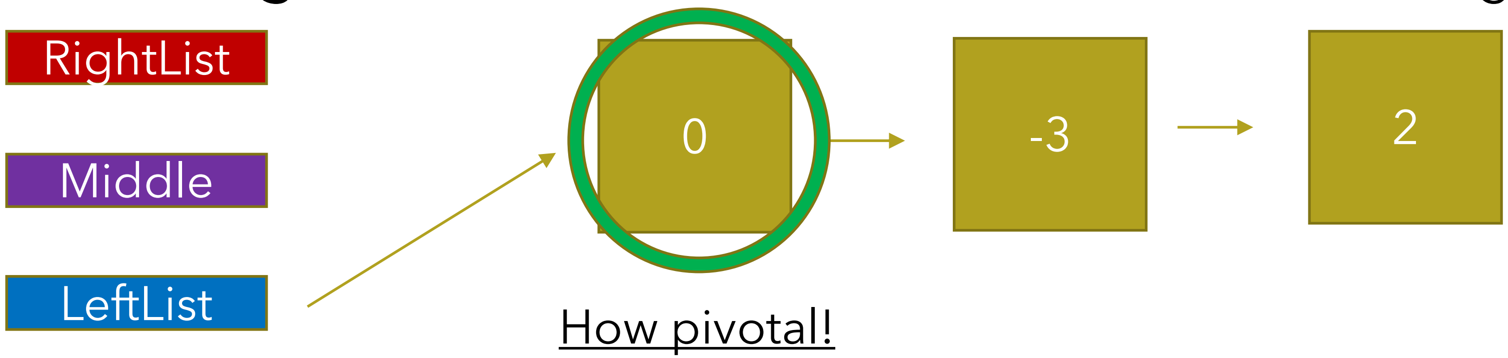
QuickSort Case Study

- Step 2: Similar to in **MergeSort**, you're going to recursively **QuickSort** the **left** and **right** sublists. No need to sort the **middle**, because guess what, it's already sorted!



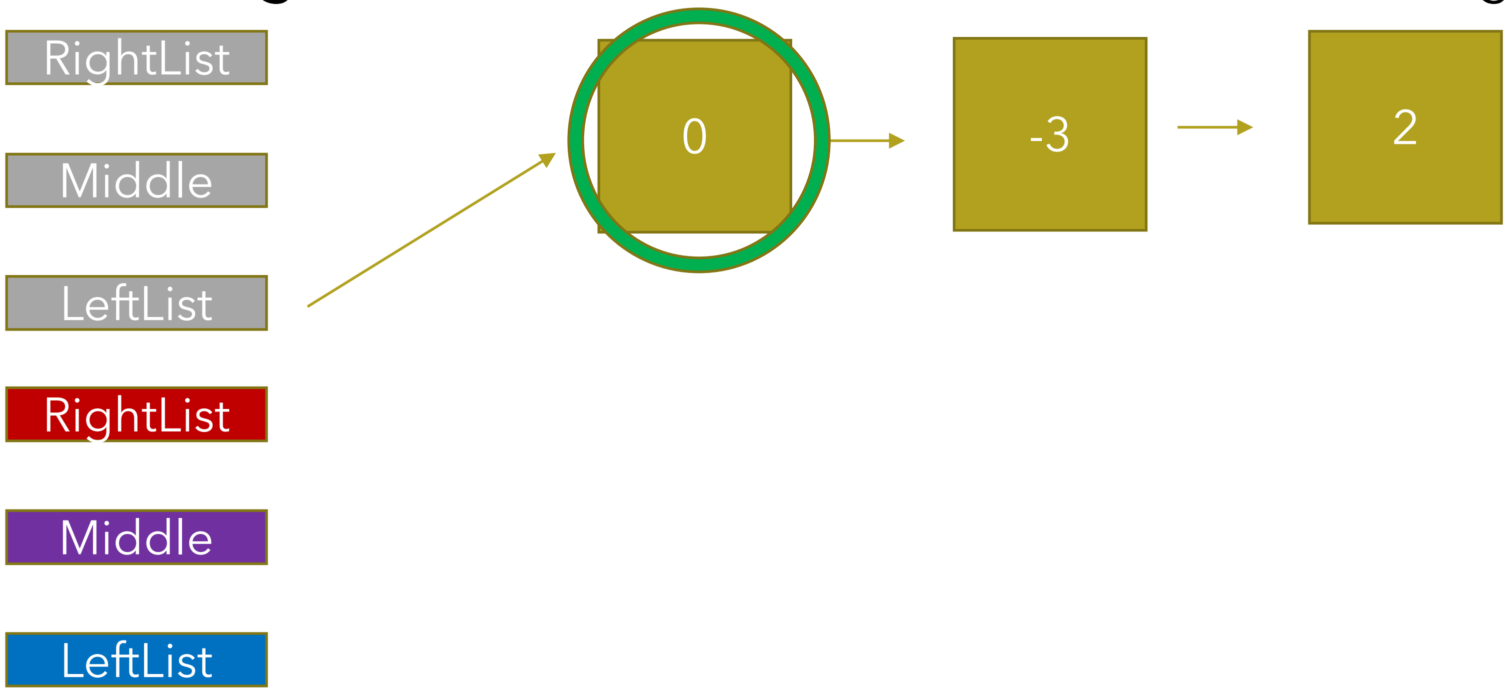
QuickSort Case Study

- Step 2: Similar to in **MergeSort**, you're going to recursively **QuickSort** the **left** and **right** sublists. No need to sort the **middle**, because guess what, it's already sorted!



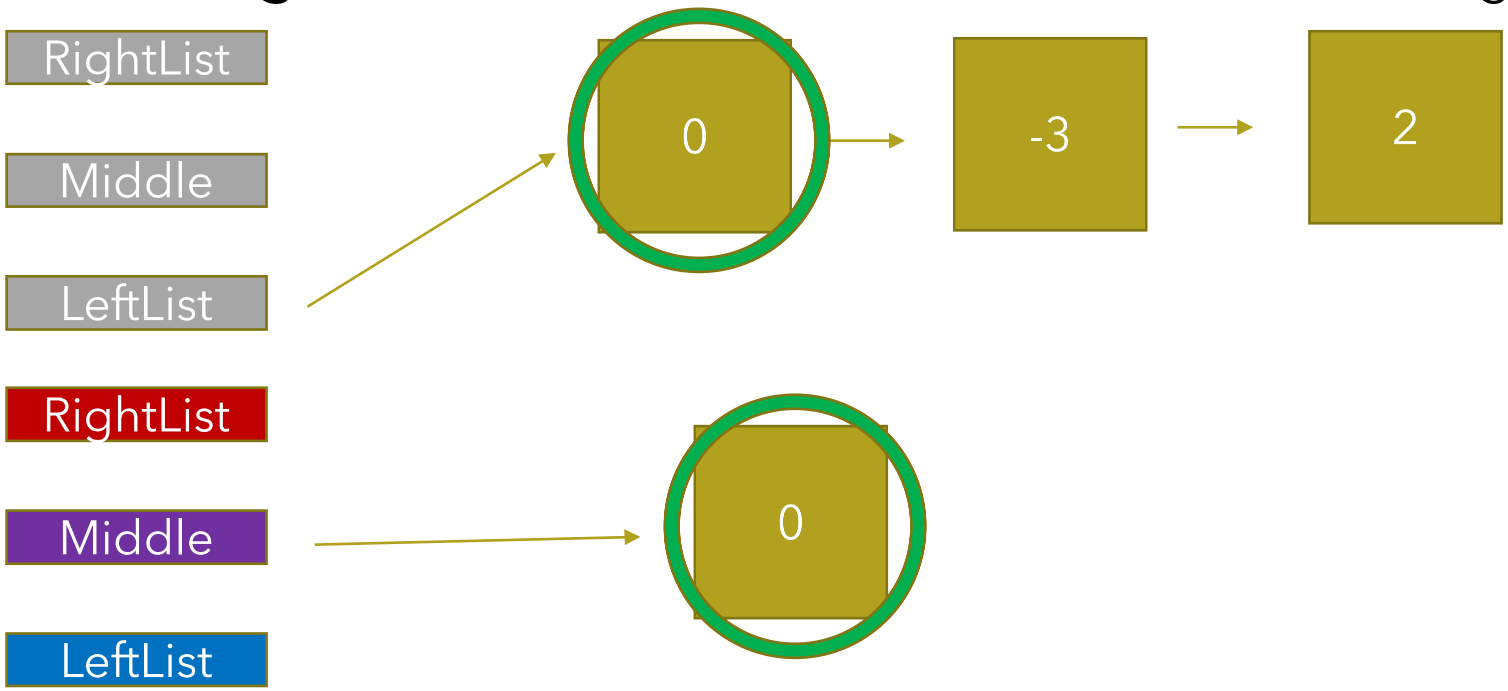
QuickSort Case Study

- Step 2: Similar to in **MergeSort**, you're going to recursively **QuickSort** the **left** and **right** sublists. No need to sort the **middle**, because guess what, it's already sorted!



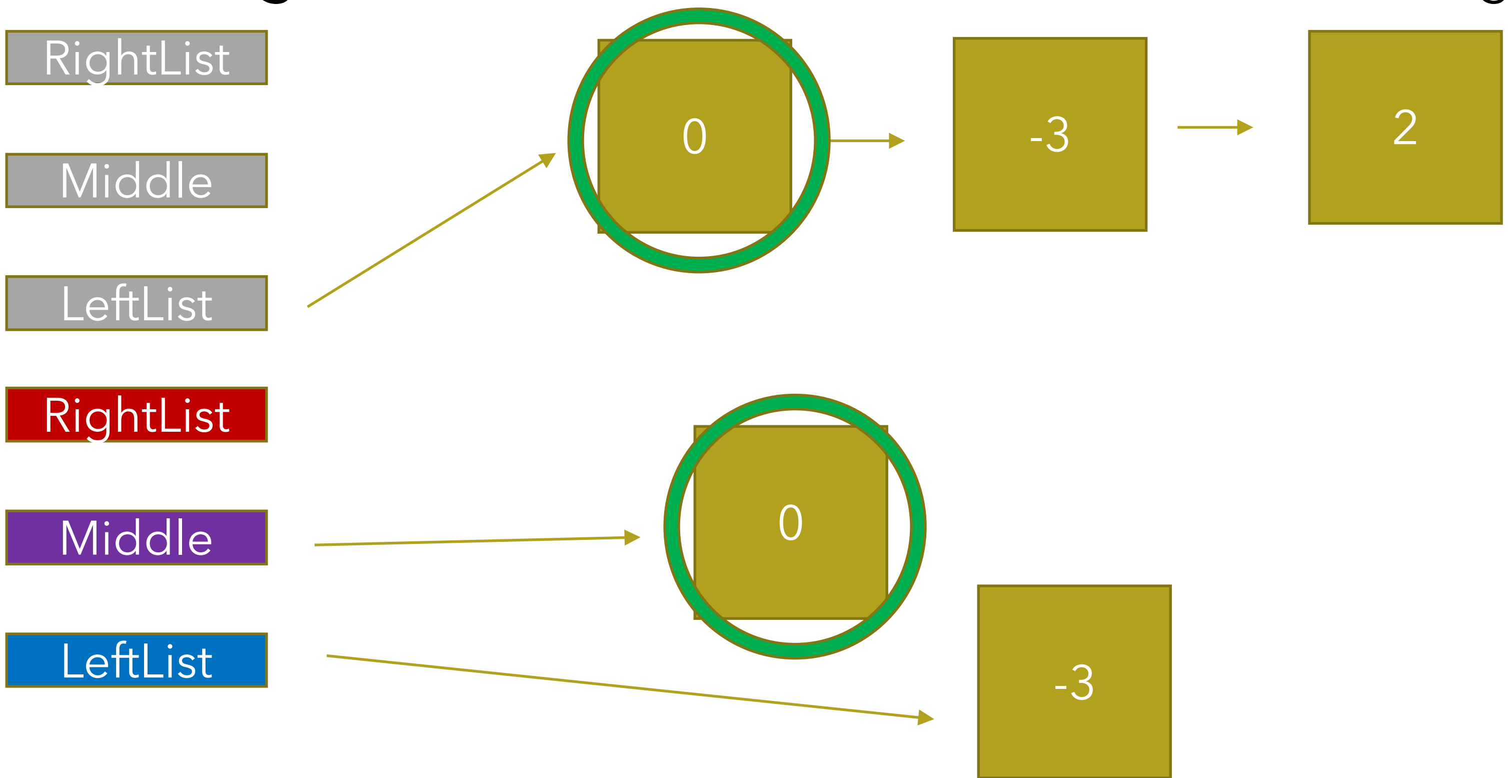
QuickSort Case Study

- Step 2: Similar to in **MergeSort**, you're going to recursively **QuickSort** the **left** and **right** sublists. No need to sort the **middle**, because guess what, it's already sorted!



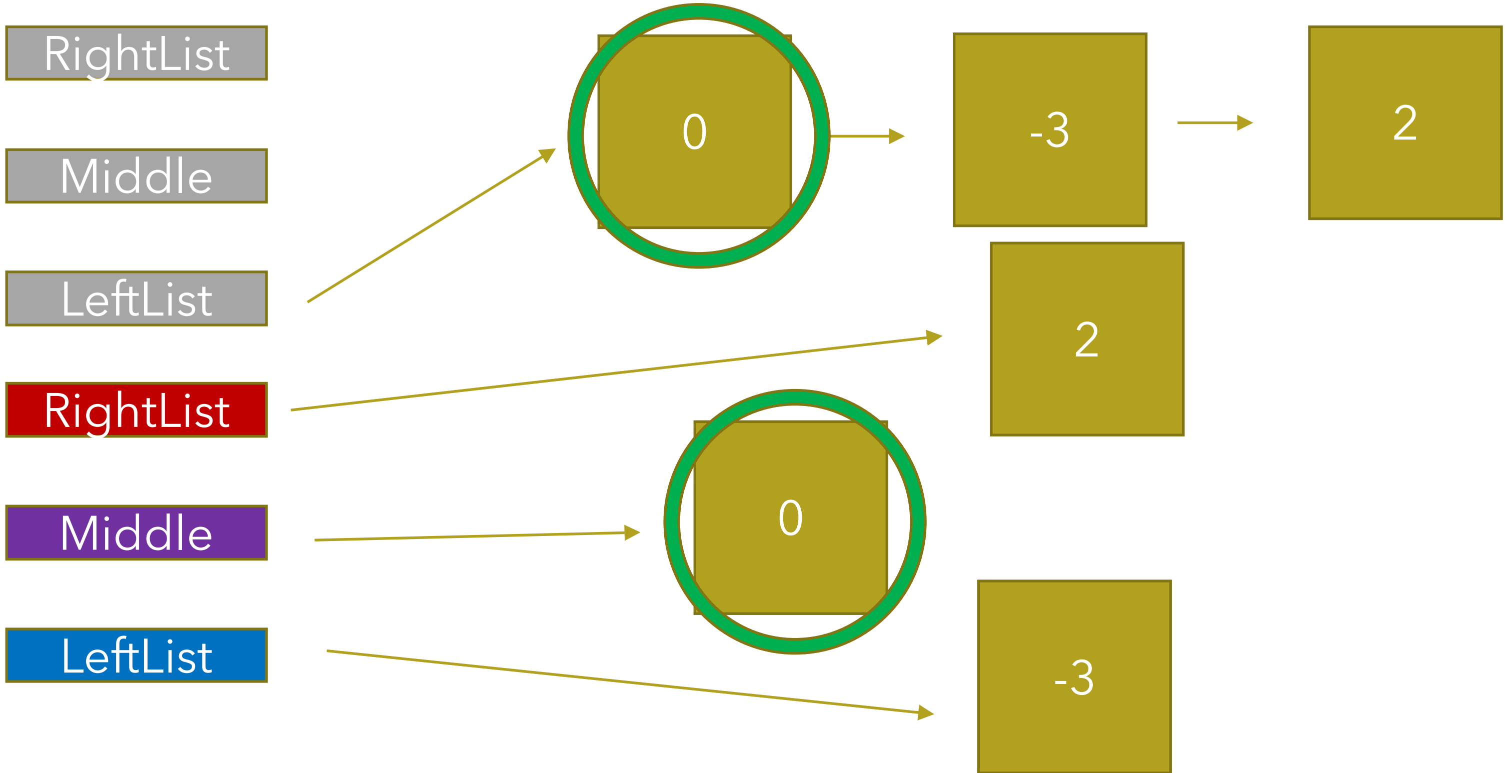
QuickSort Case Study

- Step 2: Similar to in **MergeSort**, you're going to recursively **QuickSort** the **left** and **right** sublists. No need to sort the **middle**, because guess what, it's already sorted!



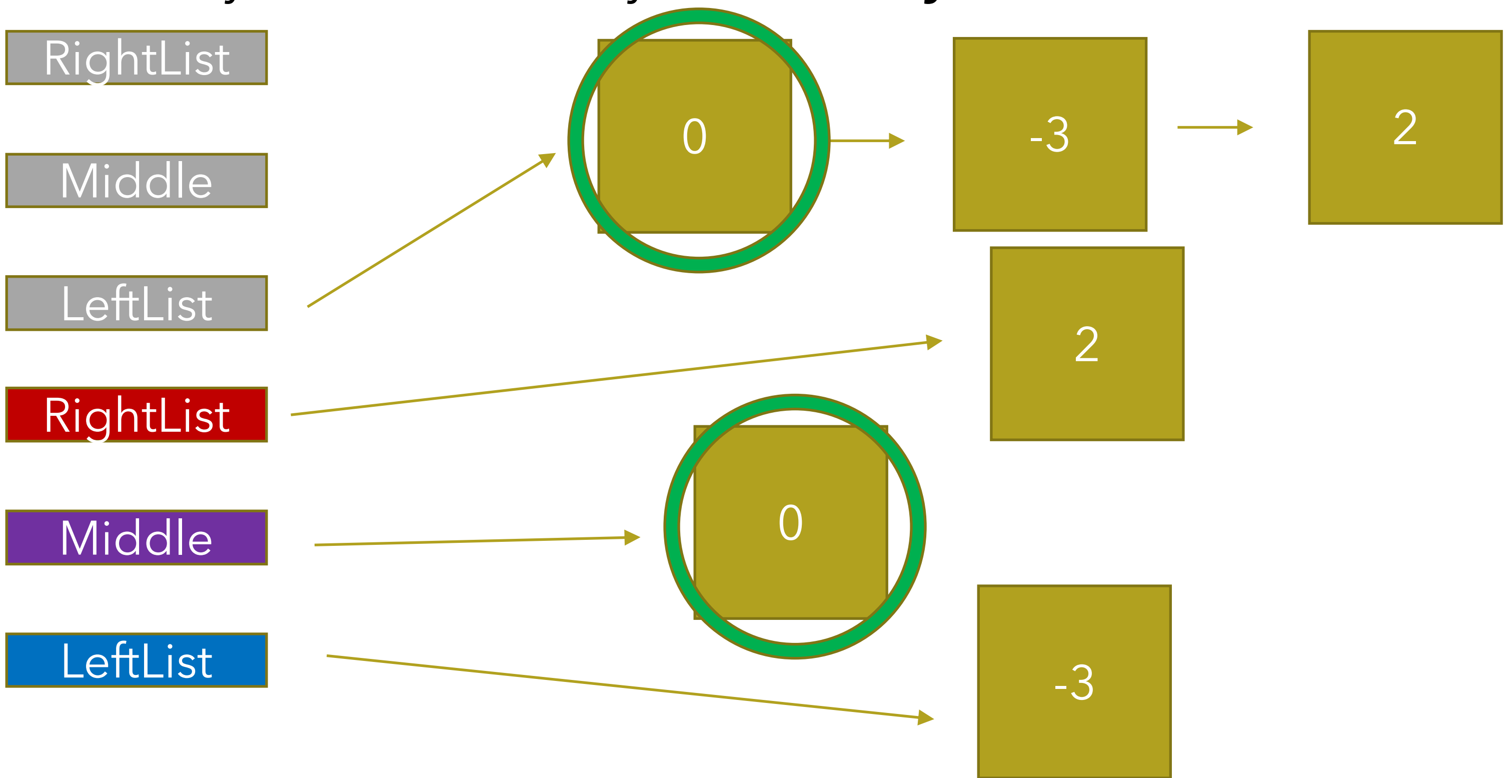
QuickSort Case Study

- Step 2: Similar to in **MergeSort**, you're going to recursively **QuickSort** the **left** and **right** sublists. No need to sort the **middle**, because guess what, it's already sorted!



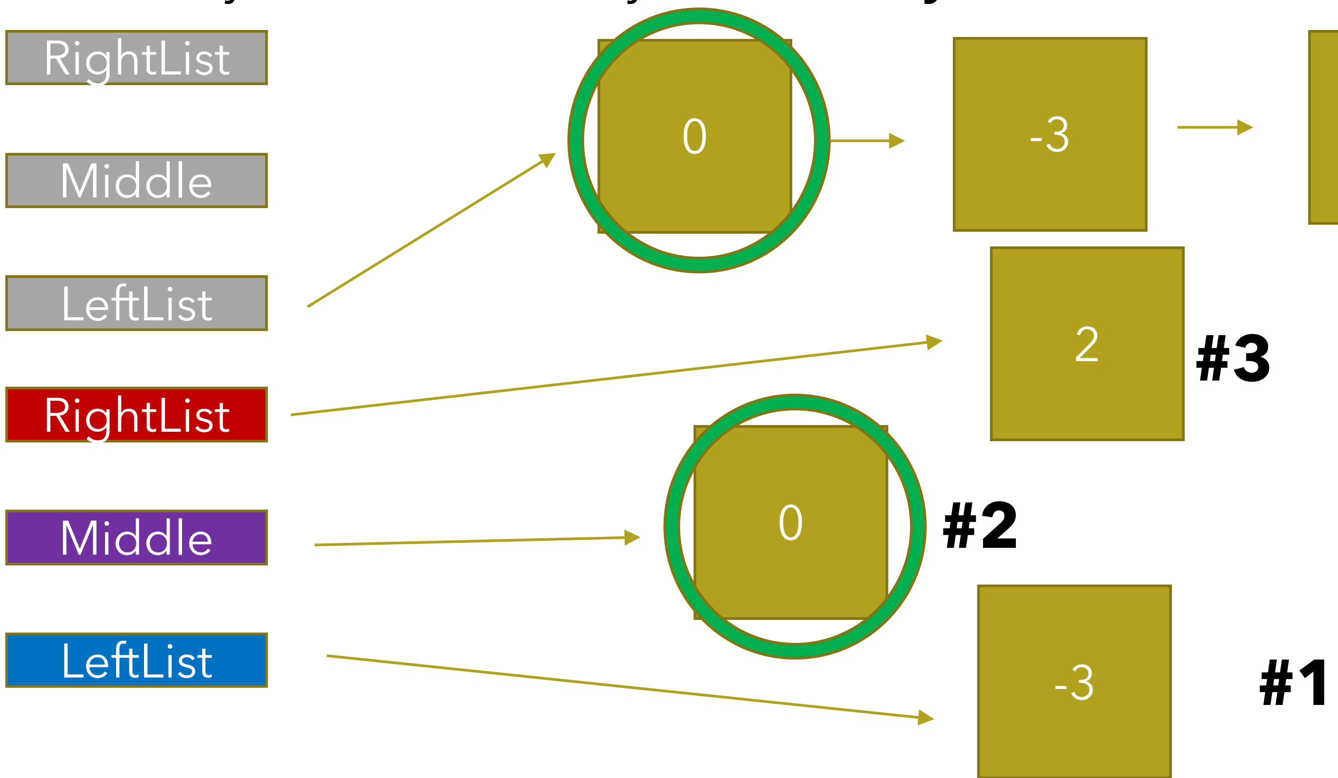
QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



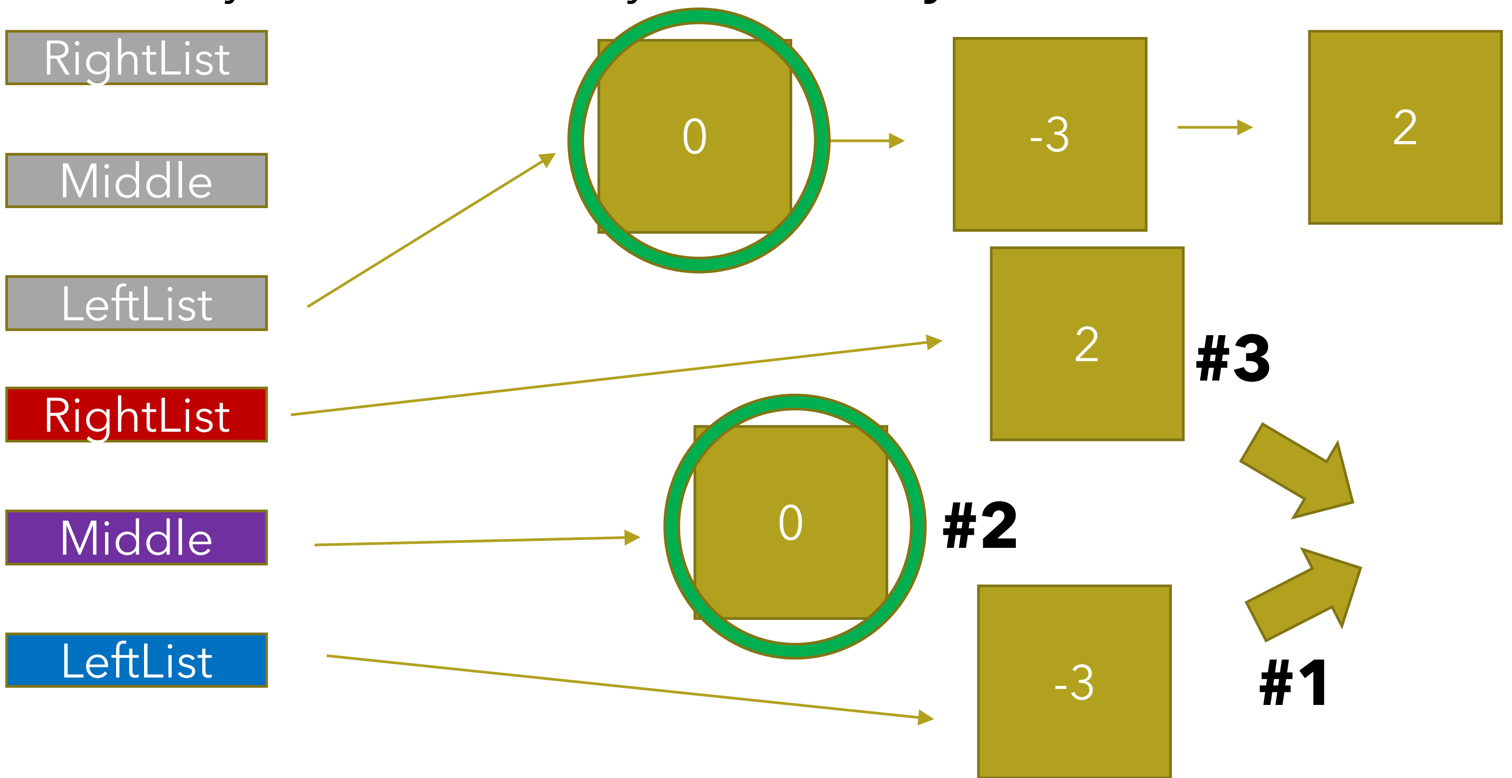
QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



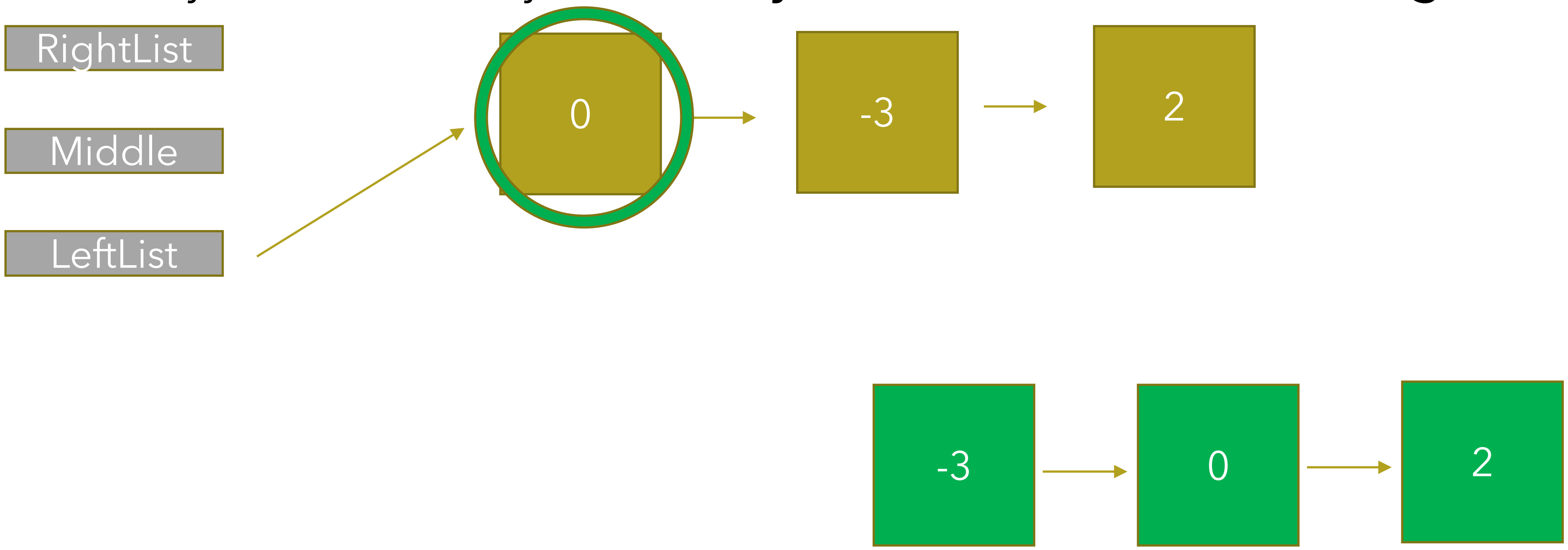
QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



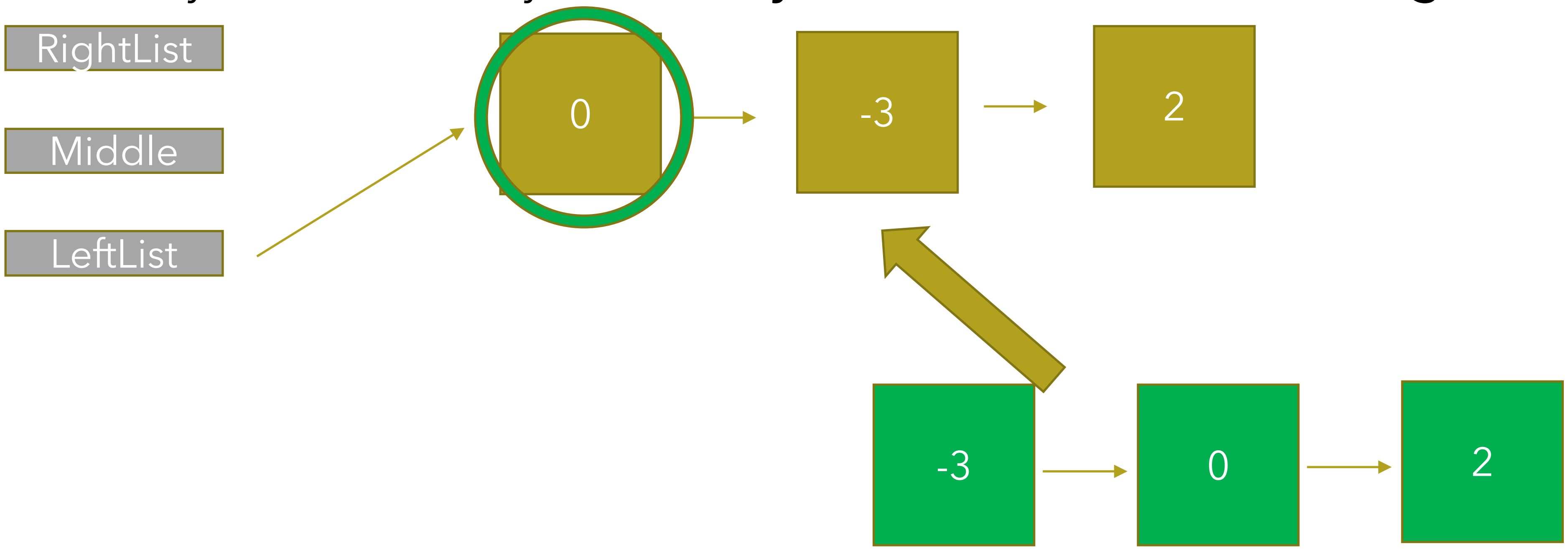
QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



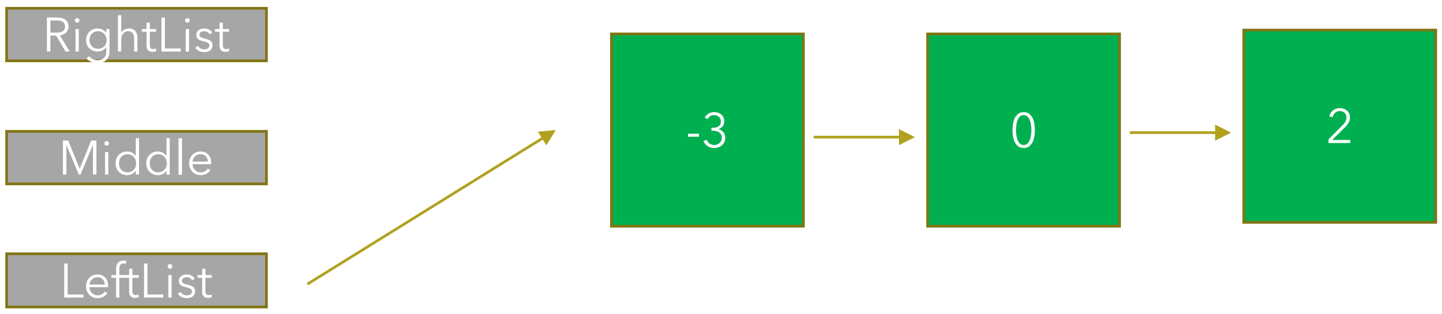
QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



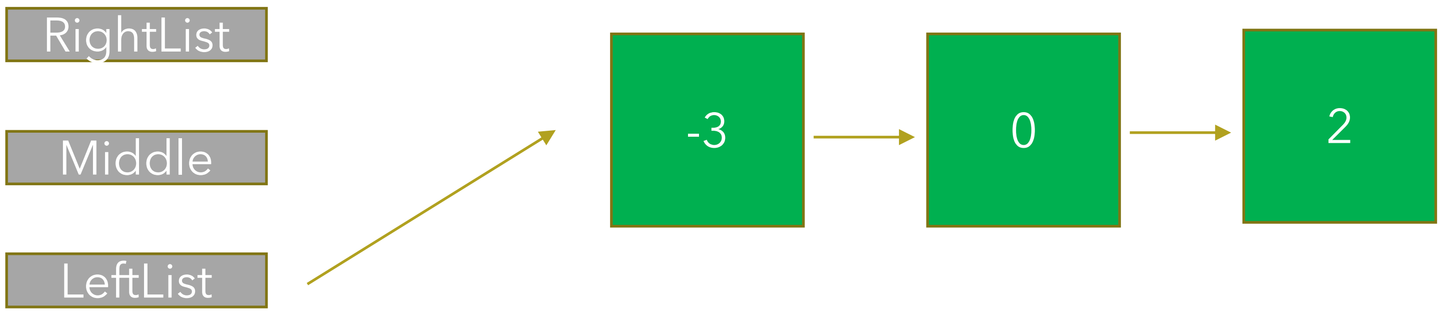
QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



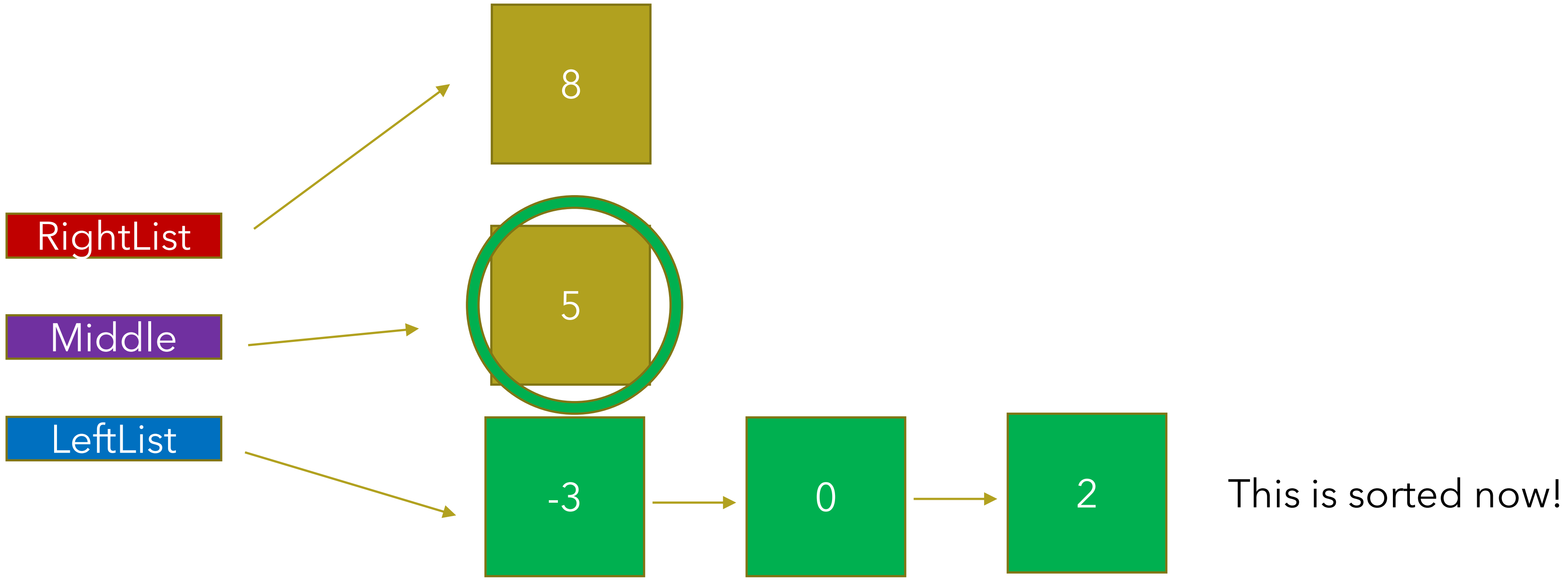
QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



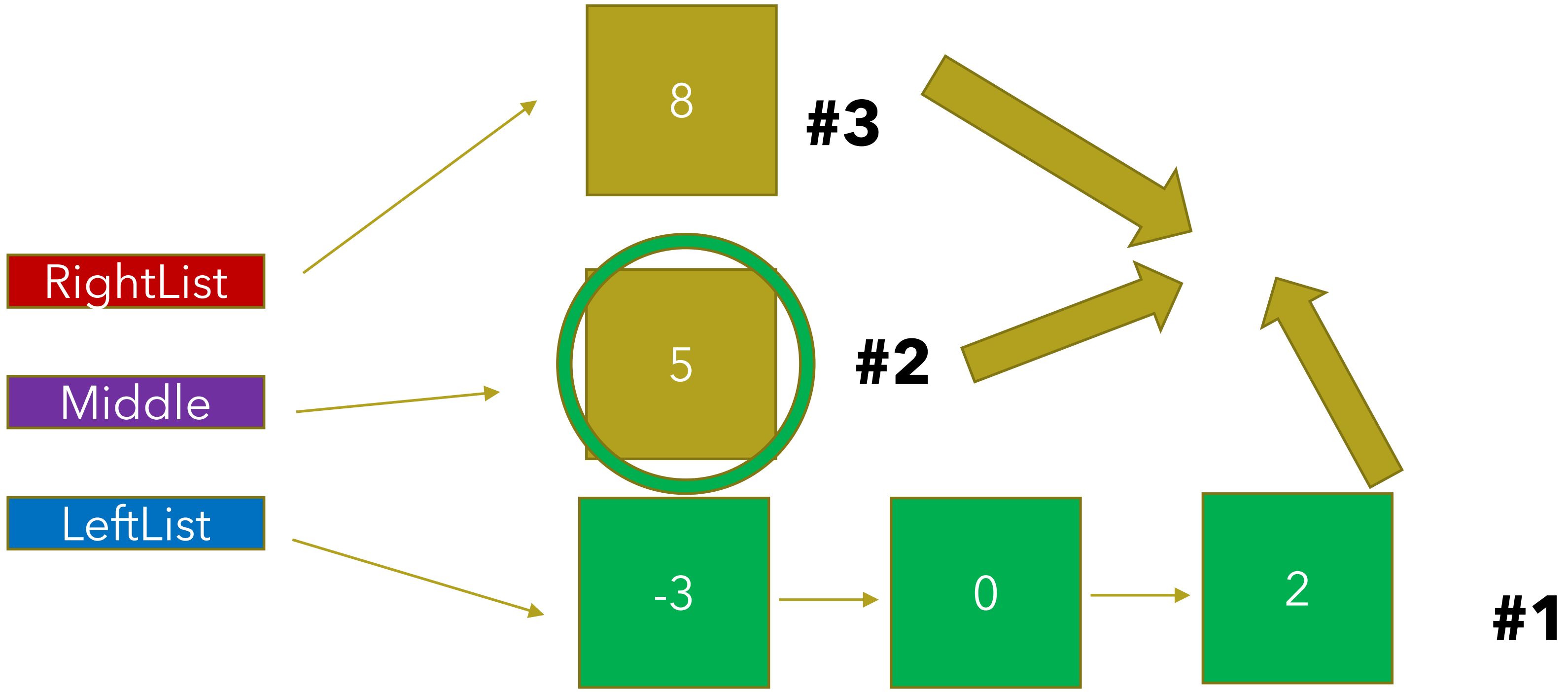
QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



QuickSort Case Study

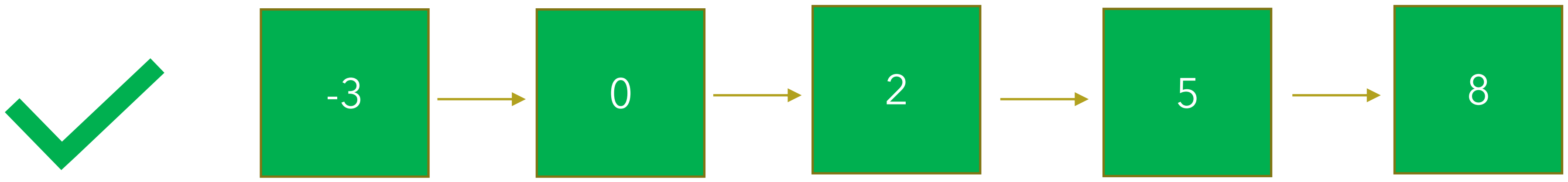
- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**



QuickSort Case Study

- Step 3: Not too dissimilarly to **MergeSort**, you're then going to take the 3 lists you've recursively made and **join them**, **Left -> Middle -> Right**

Done!



Part III: Sorting with Linked Lists

Some tips / tricks about QuickSort

- For those of you who do both, I found that the logic (code) for **splitting** into lists for both **MergeSort** and **QuickSort** were very similar (the actual splicing code was the same really!)
- The merge is very simple – just rewire the end of LeftList to point to the MiddleList, and rewire the end of MiddleList to point to RightList.
  - Beware that these lists can be empty at merge time – this **will** cause problems in your code!!
- The general structure of "calling the function recursively and joining the result" is the same between the two sorts! I think **QuickSort** is a more interesting challenge.
- **Do not** call **QuickSort** on the middle list! It's just a waste of time, and it can do strange things if not handled / avoided.

Part III: Sorting with Linked Lists

Some things to note:

- Everything that applied to the last problem applies here: no new nodes, no changing the data field, and no data structures.
- There are still **no correctness test cases**, so be sure you write your own.
- As with **MergeSort**, your **partition** and **join** routines must be **iterative**. You may call your **QuickSort** function **recursively**, however, and you probably will.

Questions about QuickSort?

Food for thought: can you think of a comparative-based sorting algorithm that runs in time faster than O(nlog(n))? **Extra credit** if you can!

Quantum Bogo Sort

QuantumBogoSort a quantum sorting algorithm which can sort any list in O(1), using the "many worlds" interpretation of quantum mechanics.

It works as follows:

1. Quantumly randomise the list, such that there is no way of knowing what order the list is in until it is observed. This will divide the universe into O(n!) universes; however, the division has no cost, as it happens constantly anyway.

2. If the list is not sorted, destroy the universe. (This operation is left as an exercise to the reader.)

3. All remaining universes contain lists which are sorted.