CS 2110 Homework 10: Using Dynamic Memory Allocation in C

Patrick Tam, Sanjay Sood, and Preston Olds

Spring 2018

Contents

1 Overview

You will be writing a Circular Linked List and Array List in C. The goal of this assignment is to familiarize you with dynamically allocating memory.

1.1 Circular Linked List

You will be writing a generic linked list library whose underlying structure is a circular doubly linked list. The list structure will contain the size (number of nodes in the linked list), and a head pointer which points to the first node in the list. Each list node contains a next pointer, previous pointer, and a data pointer. Do not change the definition of the node struct. Do not use sentinel or dummy nodes in your list.

Note that the list is circular. That means that the first node's previous points to the last node, and the last node's next points to the first node. The pointers essentially form a circle.

The diagram below illustrates the general structure of the circular linked list with data.

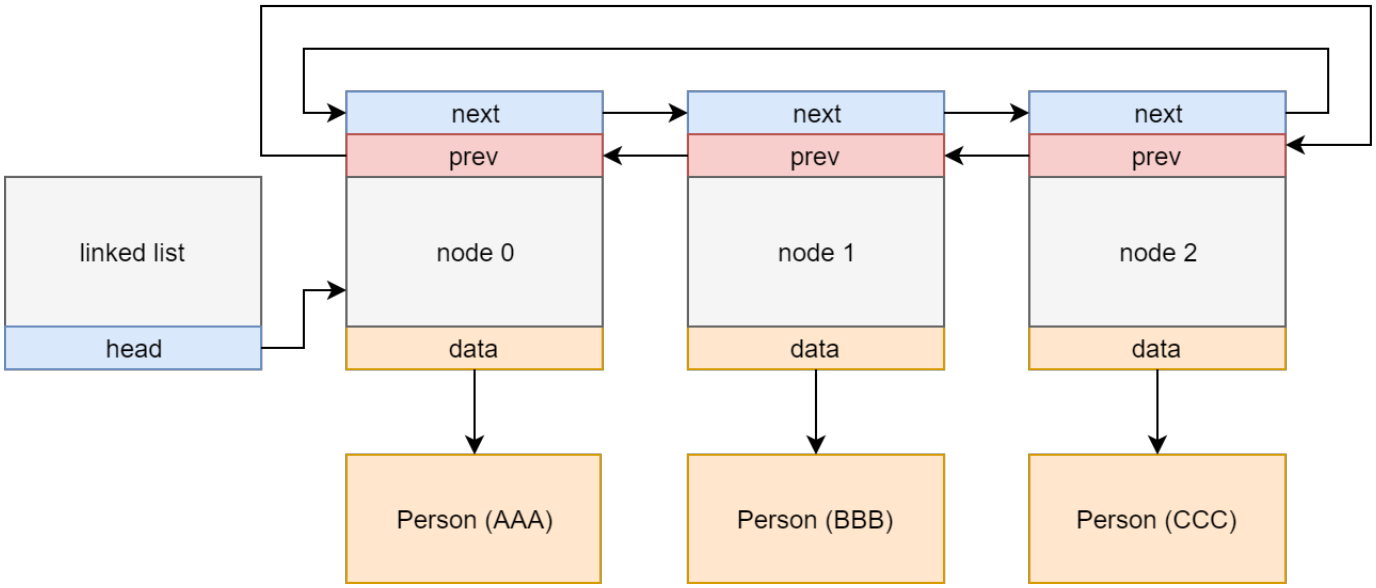

Figure 1: linked list with 3 nodes

Throughout this assignment. I'll use a couple of terms related to circular linked lists.

- **head** - the 1st (index 0) node in the linked list
- **size** - number of nodes in the linked list

1.2 Array List

Essentially, an Array List works like a resizeable array. It will have a capacity N array as its backing structure (also referred to as backing array capacity), and as elements are added, it will place them into this capacity N array. Note that the size of the Array List (i.e the number of elements actually stored within it) will always be less than or equal to N. This just means that it is not using all of the space in the backing array. Elements will all be stored at the front of the backing array (i.e starting at index 0) without gaps in between them (e.g elements 2 and 3 will be right next to each other in the backing array, and will have indices 2 and 3 respectively in the backing array).

It can continue adding without a problem until S, the size of the list (i.e the number of elements it is storing) is equal to N. This signifies that the array list is using up all of the space in its backing array. When this is happening, it means that on the next add attempt, there will not be enough space to store the new element. To rectify this, the Array List will create a new, bigger array, copy its contents into that bigger array, and start using the bigger array instead of the old one it was using (making N bigger). The new element can then be added to this bigger array without a problem, as there is now enough space.

As elements are removed from the list, they are removed from the backing array as well. However, the backing array should not have any empty space between elements. See the section on removing from an array list below.

Here's a diagram illustrating the Array List structure:

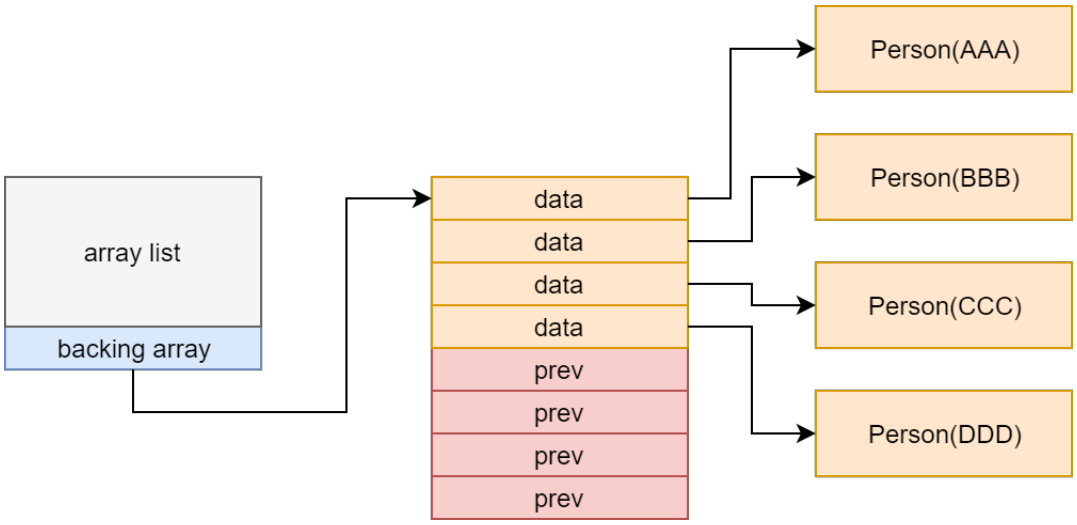

Figure 2: array list with 4 elements

Some terms that we will use:

- **size** - the number of elements in the array list
- **capacity** - the maximum possible capacity of the backing array
- **entries** - also known as the backing array

1.3 Array List examples for adding

Let's start with an example array of 4 elements called "testArr".

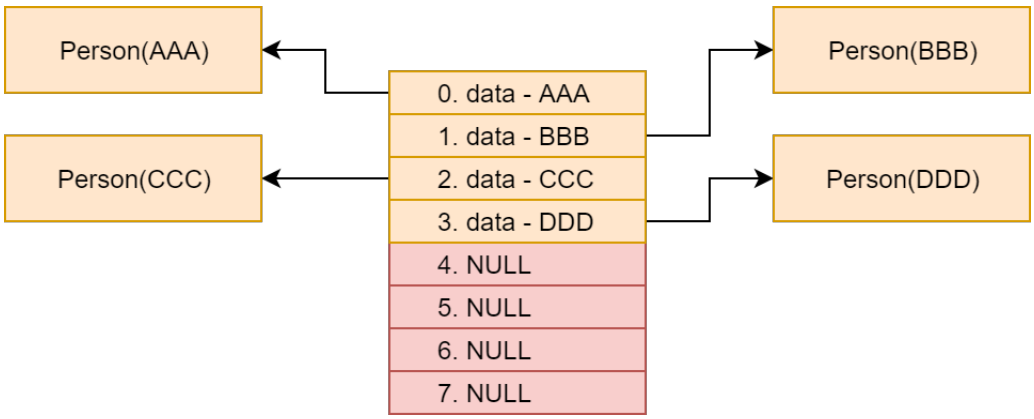

Figure 3: testArr - array with four elements, size=4, capacity=8

We will now add a 5th element to this array at index 4. Note that index 3 is the last index in the array list, so no shifting operations are required. Since our new size is 5 and our capacity is 8, no memory allocation operations are needed.

```
add_to_array_list(testArr, 4, EEE);
```

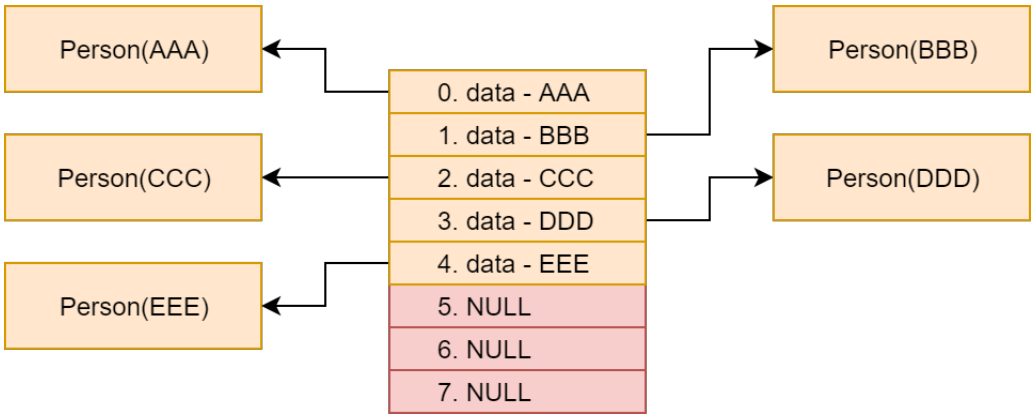

Figure 4: testArr2 - array with five elements demonstrating add at end of array with no realloc, size=5, capacity=8

Now let's add a 6th element to this array at index 2. Since index 2 is in the middle of our array list, we will need to shift all elements with index 2 and greater up by one. Since our new size of 6 is less than our capacity, again, no memory allocation operations are required.

```
add_to_array_list(testArr2, 2, FFF);
```

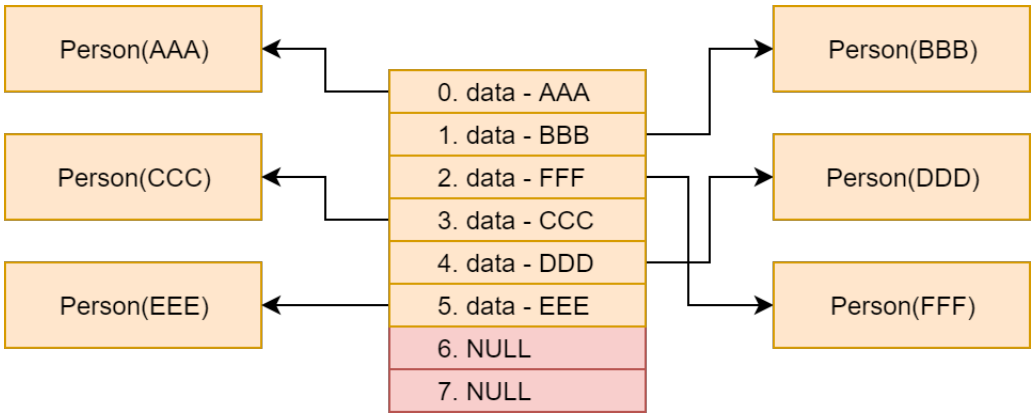

Figure 5: testArr3 - array with six elements demonstrating add in middle of array with no realloc, size=6, capacity=8

Finally, let's add a 7th element at index 7. Since index 7 is after the end of our array, no shifts are required. The new size of 8 is less than or equal to our capacity, so again, no memory allocation operations are required.

Note that since we're adding after the end of our array, we will backfill all array indices from the end of the old array (index 6), to the index before the index we are adding to (index 6) with NULL elements.

When you're coding up add, you don't actually need to do anything to make this happen. All unused array elements should be NULL by default.

```
add_to_array_list(testArr3, 7, GGG);
```

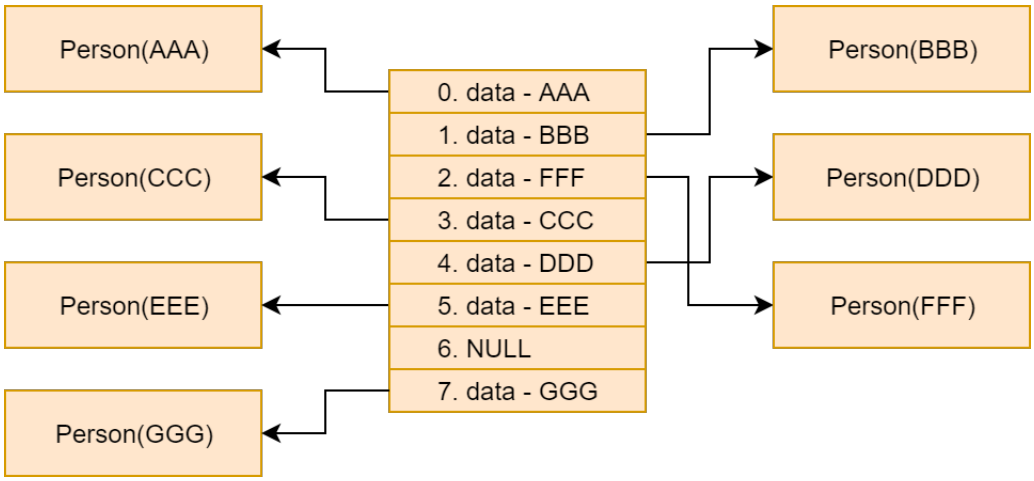

Figure 6: testArr4 - array with eight elements demonstrating add after end of array with no realloc, size=8, capacity=8

Let's revisit *testArr3*

We're going to add a 7th element at index 12. Since index 12 is after the end of our array, no shifts are required. However, our new size will be 13, so we must expand our backing array's capacity.

Again we will backfill unused elements with NULL.

```
add_to_array_list(testArr3, 12, GGG);
```

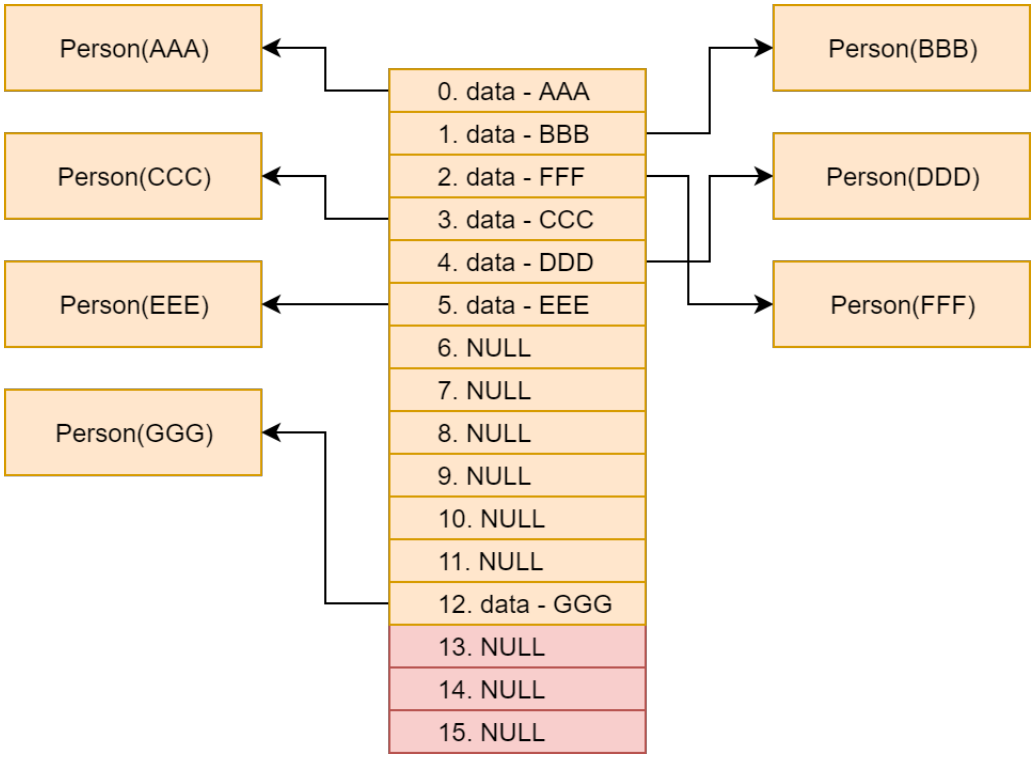

Figure 7: testArr5 - array with thirteen elements demonstrating add after end of array with realloc, size=13, capacity=16

1.4 Array List examples for removing

Let's start with the same example array of 4 elements called "testArr" from above.

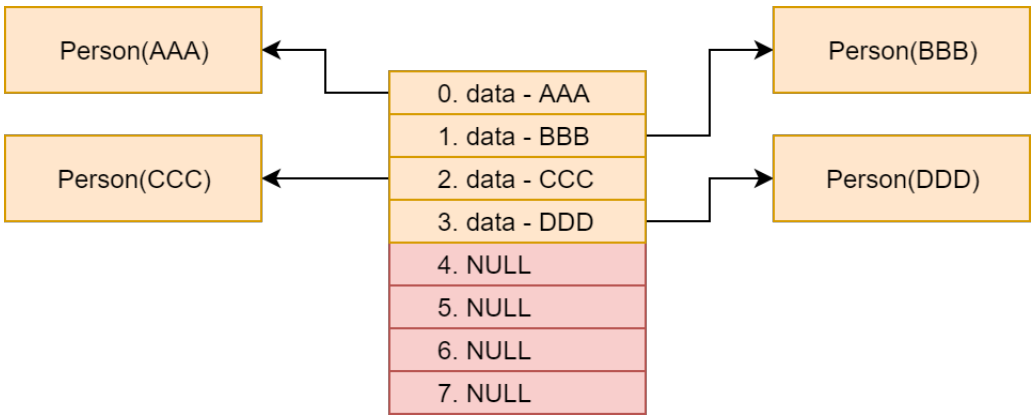

Figure 8: testArr - array with four elements, size=4, capacity=8

Let's remove the element at index 3. Since index 3 is at the end of the array, no shift operations are required. The new size of the array will be 3, which is greater than or equal to a quarter of our capacity. That means no realloc operations are required.

```
remove_from_array_list(testArr, 3)
```

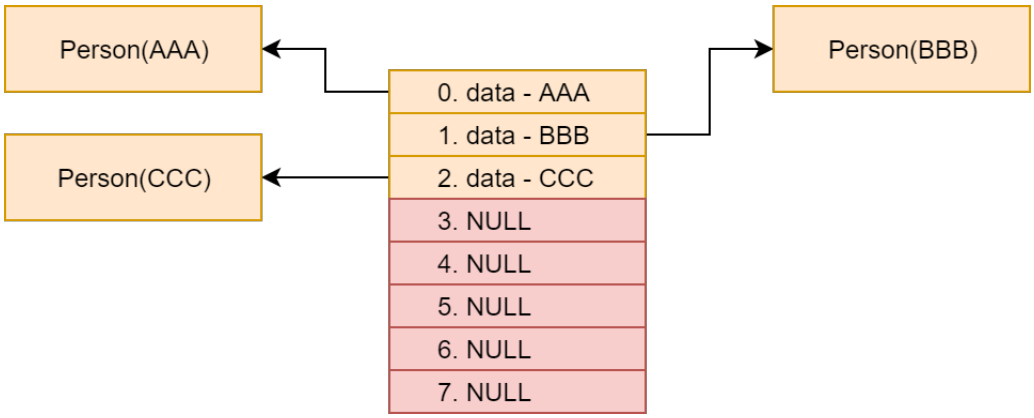

Figure 9: testArr6 - array with three elements showing removal at end of array, size=3, capacity=8

Let's remove an element in the middle of the array (index 1). Since we're deleting from the middle of the array, we will need to shift the indices of all elements above the one we're removing down by one.The new size of the array will be 2, which is greater than or equal to a quarter of our capacity. That means no realloc operations are required.

```
remove_from_array_list(testArr6, 1)
```

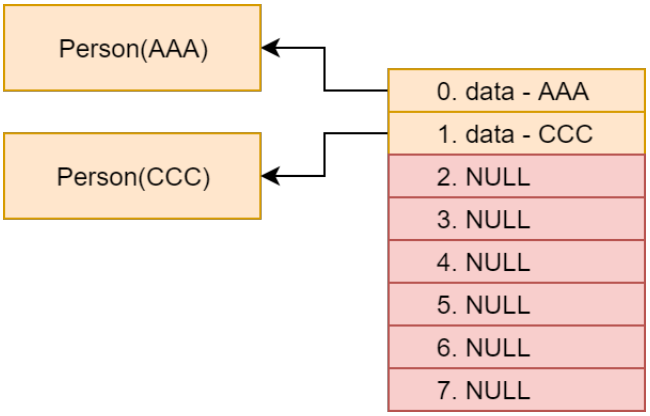

Figure 10: testArr7 - array with two elements showing removal in middle of array, size=2, capacity=8

Let's remove the 1st element (index 0). Since there are elements above index 0, we will need to shift all of them down by one. Furthermore, our new size will be 1, which is less than a quarter of our capacity. This means we will reduce the backing array's capacity by a factor of *GROWTH_FACTOR*

```
remove_from_array_list(testArr7, 0)
```

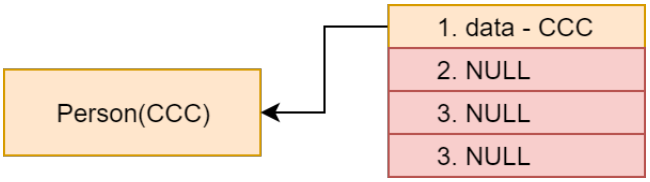

Figure 11: testArr8 - array with one elements showing removal in middle of array with realloc, size=2, capacity=8

1.5 Linked List examples

Let's start with an exmaple linked list called "testList".

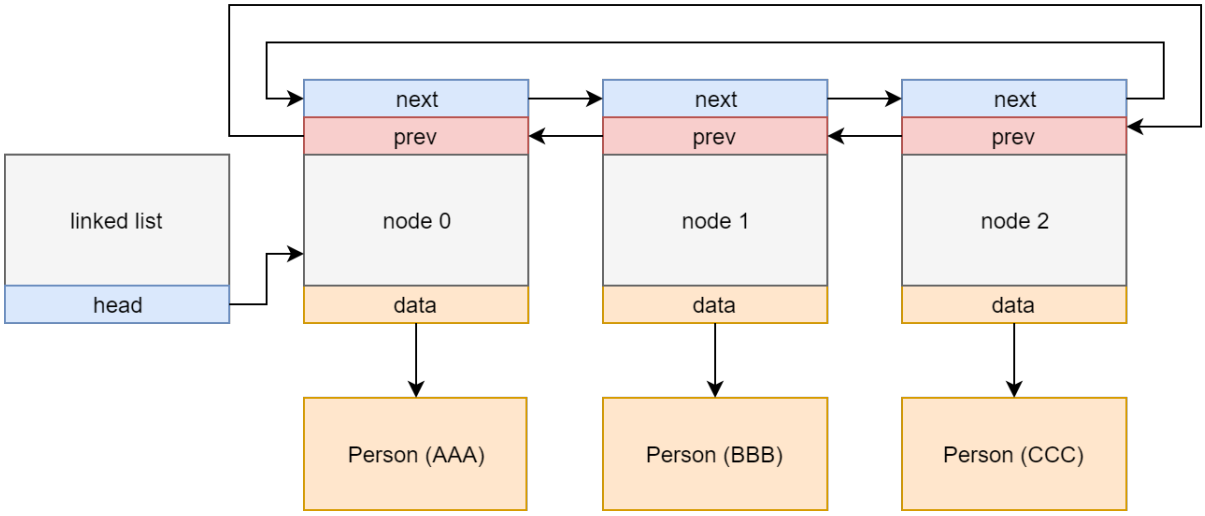

Figure 12: testList - linked list with 3 elements, size=3

Let's remove the linked list node at index 1. The node 0's next should point to old node 2, and the opposite should be true for the previous pointers.

```
remove_from_linked_list(testList, 1)
```

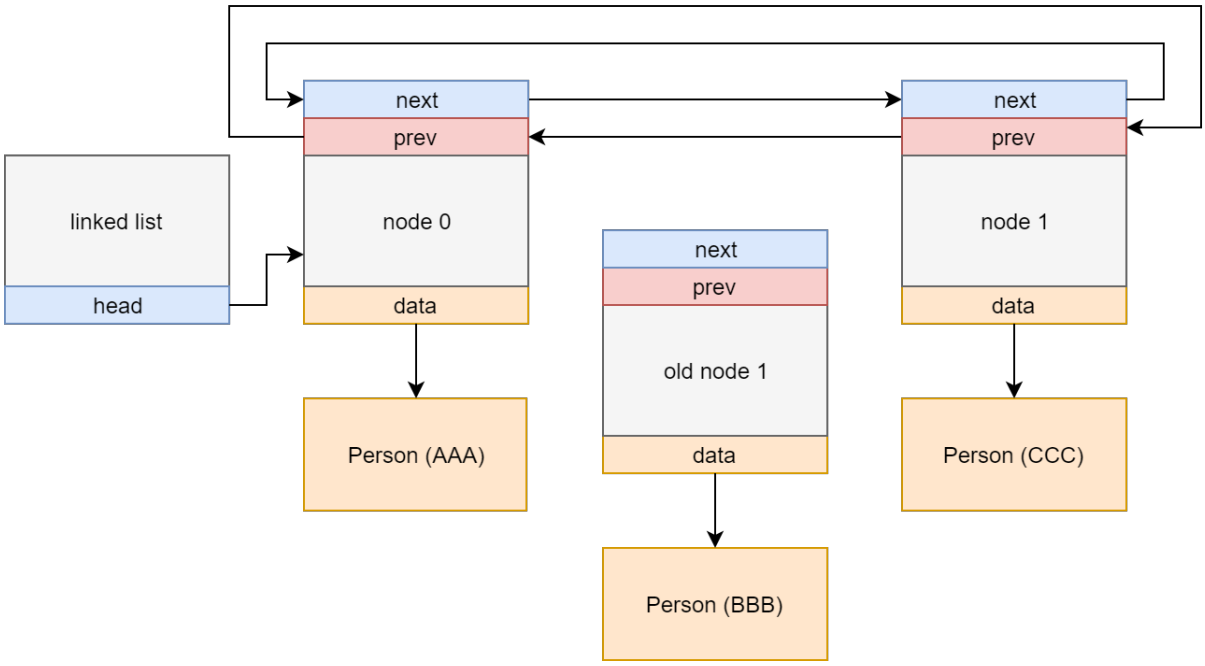

Figure 13: testList2 - linked list with two elements showing removal in middle of list, size=2

After the node has been freed and the data has been returned to the user, our linked list should look like this: only two elements.

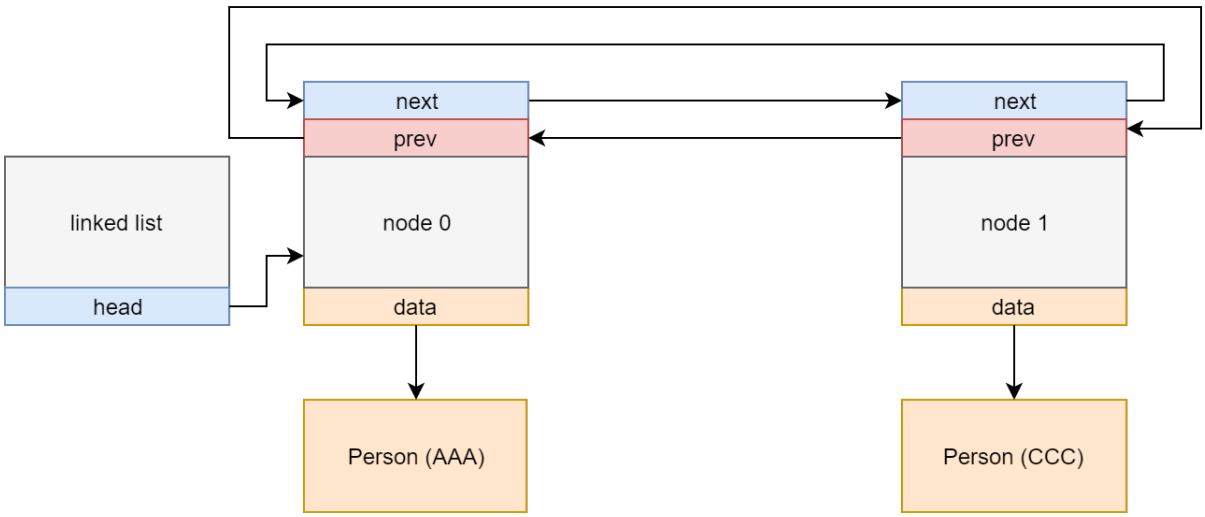

Figure 14: testList2 - linked list with two elements, size=2

Now that we've removed the node at index 1, let's add it back.

```
add_to_linked_list(testList, 1)
```

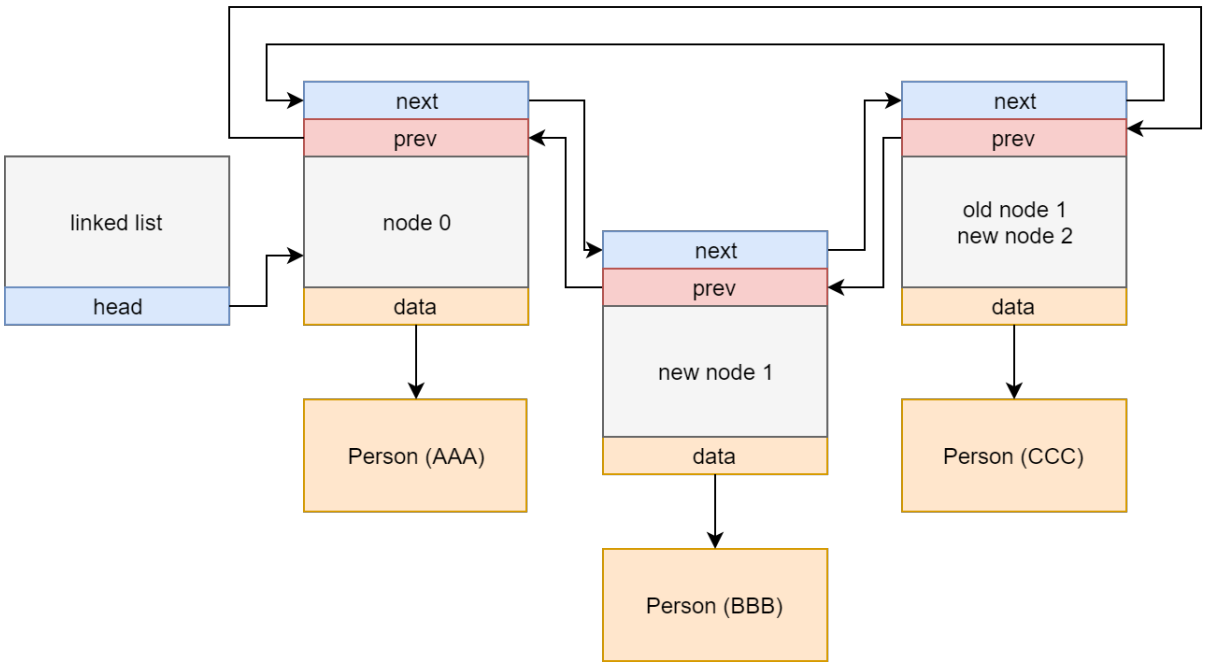

Figure 15: testList3 - linked list with three elements showing adding in middle of list, size=3

2 Instructions

Complete all functions in the following two files

1. circ_list.c
2. array_list.c

Your implementation must be optimal in terms of memory allocation operations. For example, if you call malloc 30 times when you could've called it only once, that's not optimal. If you allocate more memory than is needed, that's not optimal. If you call malloc, memcpy, memset, and free in your resize function, that's not optimal. Etc, etc. Non optimal implementations will receive **NO CREDIT**.

Your implementation **DOES NOT** need to be optimal in terms of running time. As long as our autograder completes in reasonable amount of time, then you'll receive credit.

3 Testing Your Work

Install valgrind and gdb:

```
sudo apt-get install valgrind gdb
```

Data structures in C must be tested with all those pointers flying everywhere, and it's hard to get them right the first time. For this reason, you should thoroughly test your code.

We have provided you with a file called test.c with which to test your code. Note that it contains no tests cases. You must write your own.

Printing out the contents of your structures can't catch all logical and memory errors, so we also require you run your code through valgrind. If you need help with debugging, there is a C debugger called gdb that will help point out problems. We certainly will be checking for memory leaks by using valgrind, so if you learn how to use it, you'll catch any memory errors before we do.

Here are tutorials on valgrind:

```
http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/
http://valgrind.org/docs/manual/quick-start.html
```

Your code must not crash, run infinitely, nor generate memory leaks/errors.
Any test we run for which valgrind reports a memory leak or memory error will receive no credit.

We have provided a Makefile for this assignment that will build your project.
Here are the commands you should be using with this Makefile:

1. To run the tests in test.c: `make run-test`
2. To debug your code using gdb: `make run-gdb`
3. To run your code with valgrind: `make run-valgrind`

If your code generates a segmentation fault then you should first run gdb on the debug version of your executable before asking questions. We will not look at your code to find your segmentation fault. This is why gdb was written to help you find your segmentation fault yourself.

Here are some tutorials on gdb:

```
https://www.cs.cmu.edu/~gilpin/tutorial/
http://www.cs.yale.edu/homes/aspnes/pinewiki/C%282f%29Debugging.html
```

`http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html`
`http://heather.cs.ucdavis.edu/~matloff/debug.html`
`http://www.delorie.com/gnu/docs/gdb/gdb_toc.html`

Getting good at debugging will make your life with C that much easier.

You are allowed to share test cases with other students. We will pin a post on piazza for sharing student generated test cases. Make sure not to share any of the code from *array_list.c* or *circ_list.c*.

4 Rubric

- circ_list.c (50)
  - create_linked_list (2)
  - shallow_copy_circ_list (3)
  - deep_copy_list (6)
  - destroy_linked_list (3)
  - add_to_linked_list (10)
  - get_from_linked_list (3)
  - linked_list_contains (3)
  - remove_from_linked_list (10)
  - zip (10)
- arr_list.c (50)
  - create_array_list (2)
  - shallow_copy_array_list (3)
  - deep_copy_array_list (6)
  - destroy_array_list (4)
  - add_to_array_list (15)
  - array_list_contains (3)
  - remove_from_array_list (15)
  - trim_to_size (2)

5 Deliverables

You can build the submission tarball with the following command

```
make submit
```

This command will generate *list_submission.tar.gz*

Please upload the following files to Autolab:

1. `list_submission.tar.gz`

Note that Autolab will not reveal your grade until after the assignment is due.

Download and test your submission to make sure you submitted the right files.

6 Rules and Regulations

6.1 General Rules

1. Starting with the assembly homeworks, Any code you write (if any) must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing we all know what the line of code does.

2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.

3. Please read the assignment in its entirety before asking questions.

4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.

5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

6.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to Canvas or you may submit an archive (zip or tar.gz only please) of the files (preferred). You can create an archive by right clicking on files and selecting the appropriate compress option on your system.

3. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want (see Deliverables).

4. Do not submit compiled files that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

5. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

6.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas.

3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM. You alone are responsible for submitting your homework before the grace period begins or ends; neither Canvas, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

6.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

6.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, as well as help each other debug code. What you shouldn't be doing, however, is paired programming where you collaborate with each other on a low level. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, and it is often the case that the recipient will simply modify the code and submit it as their own.

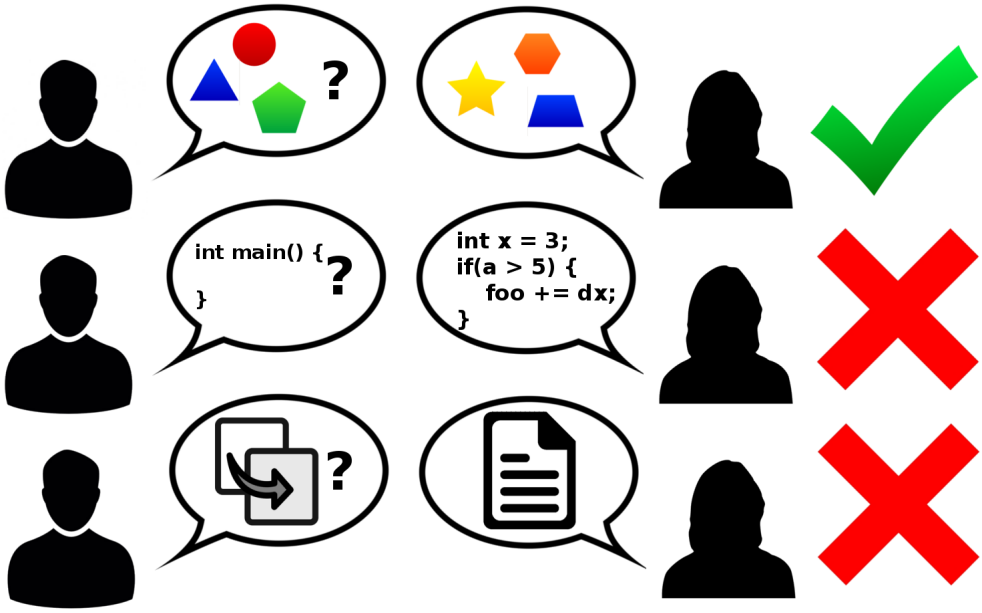

Figure 16: Collaboration rules, explained