Project 2: Memory Management

DUE: Sunday, Feb 28th at 11:59pm ***(Extra Credit for Early Submission!)***

Setup

- Download the `project2.zip` and unzip it. This will create a folder `section-yourGMUUserName-p2`.
- Rename the folder replacing `section` with the `DL1`, `DL2`, `003`, `etc` based on the lecture section you are in.
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address.
- After renaming, your folder should be named something like: `DL1-krusselc-p2`.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip `section-username-p2.zip` (no other type of archive) following the same rules for `section` and `username` as described above.
    - The submitted file should look something like this:

```
DL1-krusselc-p2.zip --> DL1-krusselc-p2 --> JavaFile1.java
                                            JavaFile2.java
                                            JavaFile3.java
                                            ...
```

- Submit to blackboard. DOWNLOAD AND VERIFY WHAT YOU HAVE UPLOADED IS THE RIGHT THING. ***Submitting the wrong files will result in a 0 on the assignment!***

Basic Procedures

You must:

- Have code that compiles with the command: **`javac *.java`** in your user directory WITHOUT errors or warnings.
- Have code that runs with the command:
  `java Simulation [numRows] [numCols] [optional:file]`

You may:

- Add additional methods, variables, and anything else you need to **`MemMan`**, **`BareNodes`**, **`MemBlocks`**, or your own classes, however **the provided code still needs to work** with this code when you are done.

You may NOT:

- Use any built in Java Collections Framework classes in your program (e.g. no **`LinkedList`**, **`ArrayList`**, etc.).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- **Create or use any arrays anywhere in your part of the program. Using arrays, in any way, will result in a 0 on this assignment (the goal is to learn linked lists!). The only exception to this is the main test code which can have a String[] in the parameter list.**
- Make your program part of a package.
- Alter provided classes or methods that are complete (**`Simulation`** or **`UserInterface`**).
- Alter any method signatures defined in this document or the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Change any code in any file in a "DO NOT EDIT" section. You still must add comments to any methods and instance variables that do not have comments already.
- Add @SuppressWarnings to avoid fixing warnings. This should not appear anywhere in your program.
- Add any additional libraries/packages which require downloading from the internet.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Where Should I Start?

The primary challenge of this project is most likely going to be one of the following:

1. Understanding the algorithms you are going to implement.

2. Designing your solution.

You are going to have a fair amount of freedom in how you implement this, so you are going to want to do all of the following ***before*** you start coding solutions to ***anything***:

- ☐ Read this document, in full.
- ☐ Trace the provided scenarios by hand.
- ☐ Design your class(es) and any helper methods.
- ☐ Get yourself a code skeleton.
- ☐ Comment ALL the code with JavaDocs (you can add more later if you need!).

Overview: Simulation for Understanding CS211 and CS262 Memory Topics

Languages like Java don't require you to allocate memory manually in the same way as languages like C, but that doesn't mean that memory allocation isn't happening. As you may remember from CS211, every time you use the "**`new`**" operator in Java you allocate "new" memory to store the object you are making, and when you no longer have a reference to some block of memory, the memory is cleaned up for you by the "garbage collector". If you have taken CS262, you may be familiar with the functions **`malloc()`**, **`free()`**, and **`realloc()`** which are ways to "manually" give blocks of memory to your program (**`malloc()`**), return memory to the computer so that it can be used again later (**`free()`**), and resize an already created block of memory (**`realloc()`**).

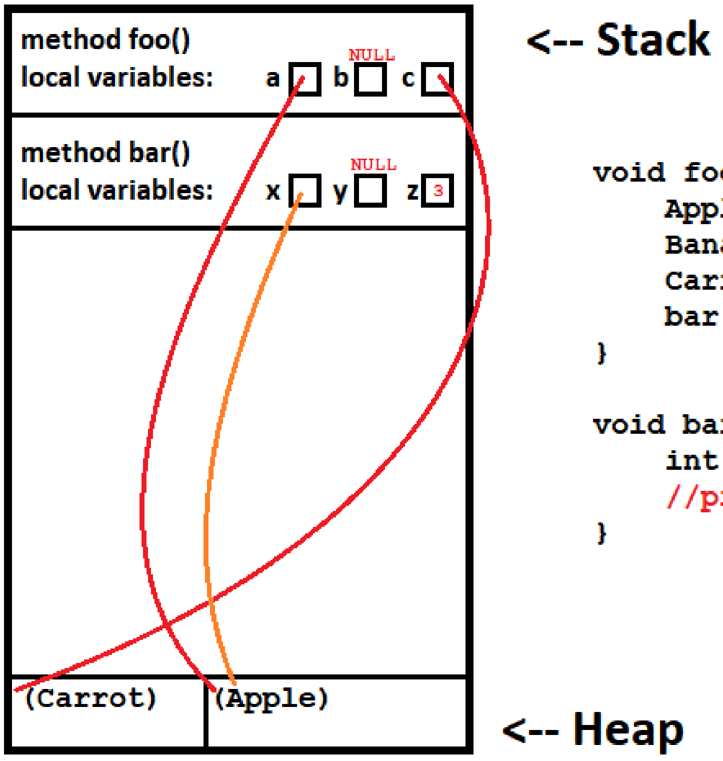
You and your professor have been asked to create a program for students who want to visualize how this memory management works. Students in CS211 can use it to better understand how Java can allocate memory and how the garbage collector works. Students in CS262 can use it for understanding how **`malloc()`**, **`free()`**, and **`realloc()`** work.

Below is some information about memory management and garbage collection (only the minimum you need to know), followed by what you can do to contribute to the project.

TL;DR: You're going to simulate malloc(), realloc(), and free(). Java's "new" operator gives new memory like malloc() and Java's garbage collector returns memory to the system like free().

Memory Basics for Computer Science

First, there are two "areas" of memory used in a computer program, the ***heap*** and the ***stack***. The stack keeps track of methods you've called and is used for storing local references and variables. All other memory is on the "heap" and when you call the **`new`** operator in Java, you end up with new memory on the heap which is "referenced" (pointed to). Here is a *really abstract* image of what you might see in a running Java program. Note: the image below is really, really overly simplified to the extent that it basically wrong in a lot of ways, but this should give you the right *impression*.



```
void foo() {
    Apple a = new Apple();
    Banana b = null;
    Carrot c = new Carrot();
    bar(a,b);
}

void bar(Apple x, Banana y) {
    int z = 3;
    //program is here right now
}
```

For this project, we will be visualizing the ***heap*** part of memory, which you can visualize as just a giant array of bytes:

This memory can be given away to a program that needs to store information in a variable. For example, a 64 bit integer would need 8 consecutive bytes of memory from the computer (1 byte = 8 bits).

(One 64-bit Integer Goes Here)

There are actually many algorithms for allocating memory when it is requested. For this project, you and your professor are going to simulate an algorithm called "***implicit lists***" with four different ***placement policies*** (more on this later), and a garbage collection algorithm called "***mark and sweep***". Your knowledge of linked lists will be extremely helpful here!

Implicit Lists

This technique creates an abstraction of memory where consecutive free bytes of memory are kept together, while allocated bytes are separated from each other and from the free memory. For example, if we allocated memory for our 64 bit int above, it will be treated as separate from the free memory and we'll have two ***blocks*** (consecutive memory areas):

(One 64-bit Integer Goes Here)	(15 Bytes Free Here)

Allocating memory for another 64 bit int will make three blocks (one for each int, and the "free" area). This is shown below visually:

(One 64-bit Integer Goes Here)	(One 64-bit Integer Goes Here)	(7 Bytes Free Here)

It is possible that there will be multiple blocks of free memory, for example, if we put three 32 bit ints (each 4 bytes) on the heap:

(32-bit Int)	(32-bit Int)	(32-bit Int)	(Free)

then "free" the memory for the middle one, we will have this appearance:

(32-bit Int)	(4 Bytes Free)	(32-bit Int)	(11 Bytes Free)

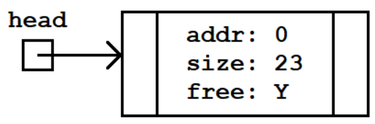
Consecutive blocks of free memory are merged together. Merging together blocks is called "***coalescing***". So if we were to free the third integer, the computer would consider there to be only two blocks again:

(32-bit Int)	(19 Bytes Free)

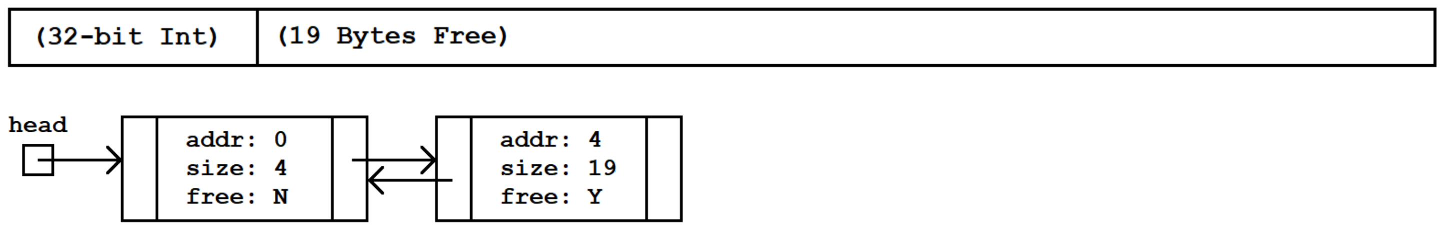
So, how do you track this memory? With a linked list of course! Here's how it works: A doubly linked list tracks all the blocks of memory, one block per node in the linked list. Each block has information about the starting address of the block (the memory address of the first byte), the size of the block (in bytes), and whether or not the block is free. The nodes provide pointers to the previous and next blocks of memory in the heap, so this is a doubly linked list.

Initially, all the memory in the heap is available in one big block, so the heap tracks only a single node in the linked list:

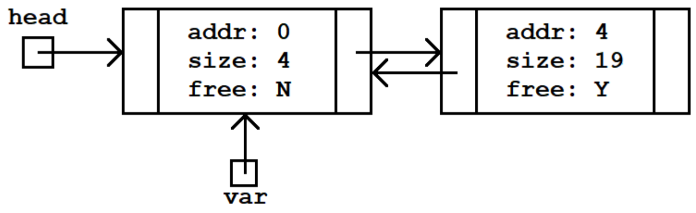
(23 Bytes Free)



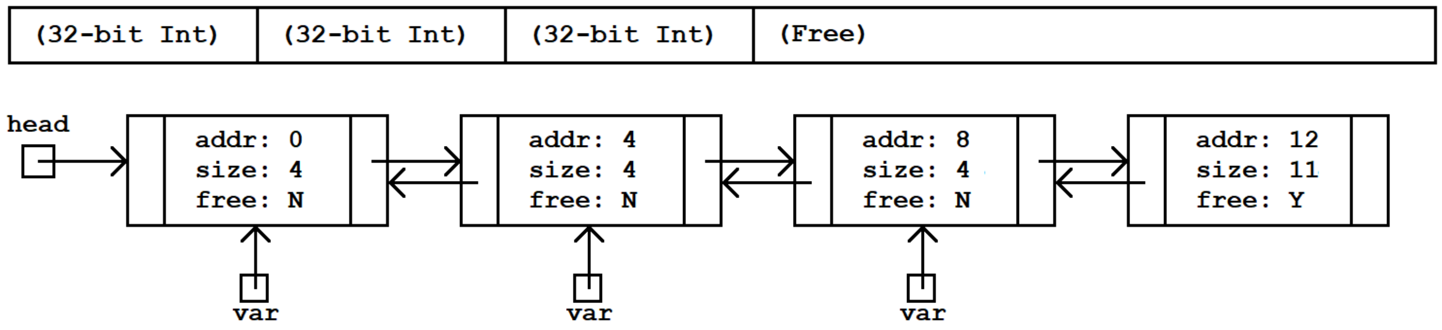
When you allocate memory to the program (call `malloc()` or use "**`new`**" in Java), it needs to know the size of the memory to be allocated. Then, based on the "placement policy" (again, more information about this later on), it will choose a block that is big enough to put the memory in and give back a pointer to that area of memory. The procedure is to "***split***" the node to remove the allocated space. This turns one node into two:



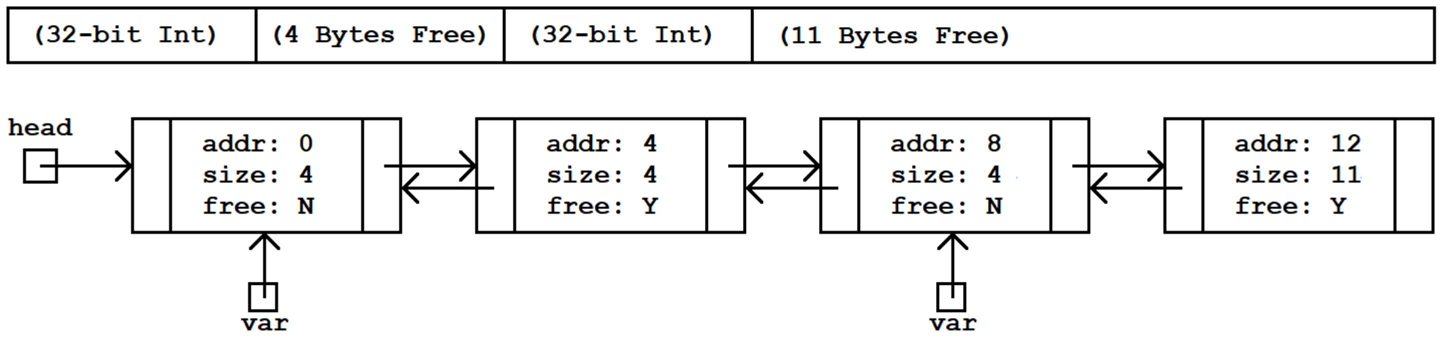
The node that now contains the allocated memory (the first node in this case) is returned to the caller. The caller stores this as a type of "pointer" for a program to use telling it where the memory is:



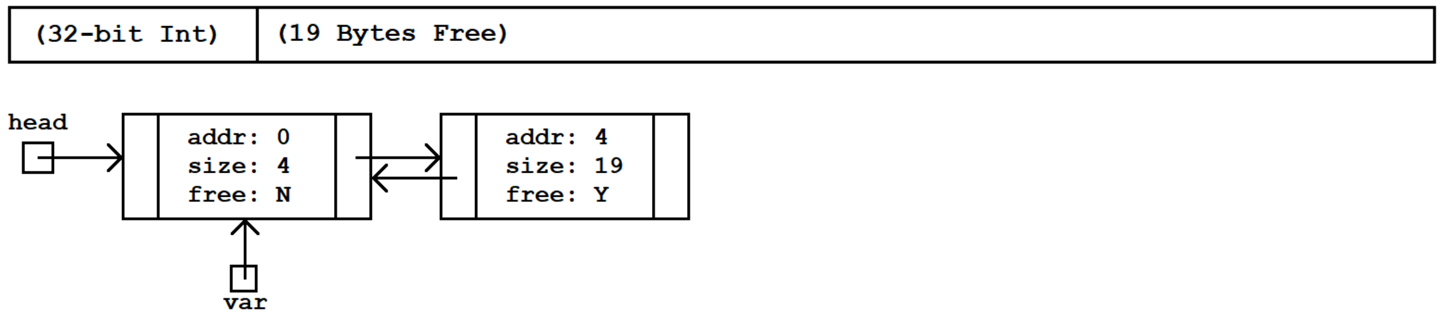
When you free memory used by the program (by calling `free()` or having Java's garbage collector call it for you), the pointer (node for the allocated memory) is given back to the heap. Using this node information, the heap will either simply mark it as free or, if it creates two adjacent free blocks, it will "merge" the free nodes together. So if we allocate two more 32-bit ints we get this:



Now freeing the memory for the middle int we get:



And freeing the third block and coalescing we're back to:



Some Answers to Misc. Questions You Might Have at This Point

Why is this called an "implicit list"? It looks pretty "explicit"...
The way this is *actually* done in the computer is that you have a special area at the start of each block of memory that stores the block size and a lot of other information, this allows you to "jump to the next node" by just moving forward in the memory the correct number of bytes. If you want the doubly linked version, this requires storing both header and footer information so that you can go both ways. These headers use up some of the free memory you're trying to allocate, which further complicates matters. This is more of a CS367 topic and not one for this CS310. So we're going to use *actual* linked lists because this is a *simulation* of memory and there is no reason to overcomplicate the implementation to optimize it for hardware.

You mentioned `malloc()`, `free()`, and `realloc()`, but you only allocated memory (used `malloc()`) and freed memory so far. What is `realloc()`?
Sometimes you want to get more memory for some variable without losing the current information stored there, or you might discover you need less memory than you thought. This is most common with arrays, where you might want to "extend" or "shrink" an array without losing the information currently in the array (think how useful this would be for dynamic arrays!). **`realloc()`** allows you to resize some memory by doing one of the following:

1. **Shrinking** – the allocated area shrinks and the second half gets freed. Example of shrinking 8 bytes to 4:

`(8 Bytes Used For Something)`	`(15 Bytes Free Here)`

`(4 Bytes Now)`	`(19 Bytes Free)`

2. **Expanding** – if the *next* block in memory is free, you may be able to "expand out" into that block. The memory gets reallocated so that you are allowed to use that space, we don't ever extend backwards (since our data starts at a certain location this isn't too useful).

`(Only 4 Bytes)`	`(19 Bytes Free)`

`(8 Bytes Now!)`	`(15 Bytes Free Here)`

3. **`malloc() + free()`** – if you need more memory and can't expand, you can try to **`malloc()`** enough space, copy the memory from the old block, and then **`free()`** the old block. It's very important that you **`malloc()`** *then* **`free()`** or you could lose the data! Also, if you can't **`malloc()`**, you want to leave the variable alone. Here's an example of expanding the first int in memory from 4 bytes to 8 bytes:

`(Only 4 Bytes)`	`(32-bit Int)`	`(32-bit Int)`	`(11 Bytes Free)`

`(4 Bytes Free)`	`(32-bit Int)`	`(32-bit Int)`	`(8 Bytes Now!)`	`(3 Free)`

You mentioned "placement policies", what are those?
The computer may have multiple blocks of free memory and a placement policy helps the computer decide which of those blocks should be used. For example, given the memory below, if you want to allocate 1 byte, where should it go?

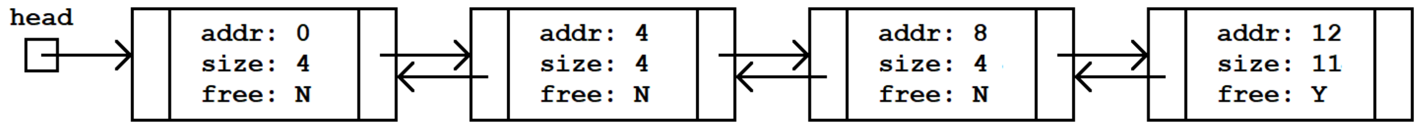
`(32-bit Int)`	`(4 Bytes Free)`	`(32-bit Int)`	`(11 Bytes Free)`

For this project, the simulation is going to provide four different placement policies:

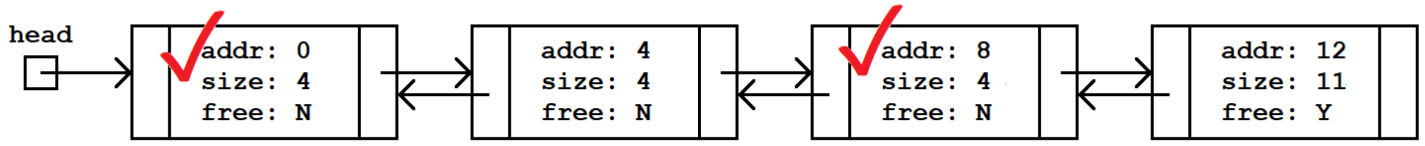
1. **First Fit** – Starting with the first block of memory, use the first free block that's big enough. Advantage: the end of the list tends to have big blocks! Disadvantage: the beginning of the list tends to have a ton of tiny blocks. This would place the memory in the 4-bytes space in the above example.

2. **Best Fit** – Find the block of memory that is as close as possible to the size you want (while still being big enough). Advantage: Tends to "fit" more memory allocations than other methods. Disadvantage: Requires searching the entire heap before deciding where to put anything. This would place the 1 byte in the 4-bytes space in the above example since 4 bytes is much closer in size you want compared to the 11-byte block.
3. **Worst Fit** – Find the biggest block of memory (if there is one big enough) and use that. Advantage: tends to create larger "free" fragments compared to best fit. Disadvantage: large allocations are more likely to fail. This would place the memory in the 11-byte space since it's bigger than the 4-byte space.
4. **Next Fit** – Start with the last free block you used, and try to allocate there. If not enough space, continue forward (coming back around to the front if necessary). Advantage: Proposed by Donald Knuth and faster than first fit if there are a lot of small blocks at the "front". Disadvantage: doesn't utilize memory as well as first fit. This algorithm would place it in the 4-byte space if that had been the last block split by a call to `malloc()`, or the 11-byte space if that one had been the last block split. (Note: splitting a block for `realloc()` to shrink/expand does not affect this, but `realloc()` that calls `malloc()` does.)

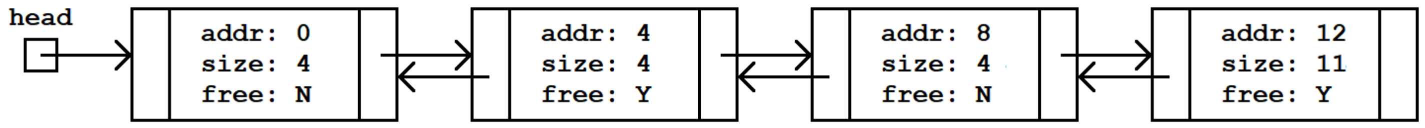
I heard something about garbage collection?
Yes. In Java the garbage collector is automatic, so we're going to make this as well! We're going to use a simple algorithm called "***mark and sweep***". Here's how it works: given a set of pointers (memory addresses) you know are used in the program, walk through each block of memory. If the memory block is not free, and is used by the program, mark it. Once all the memory is marked, go through all the memory blocks again, freeing unmarked blocks that aren't free. You also un-mark everything that is marked. Example:



Given addresses as a set: { 8, 0 } we mark the blocks that at those addresses. Note that a ***set*** in this case is an ***unordered collection of data with no duplicates***, just like in math.



Now we "sweep" through, removing the marks and freeing anything that is allocated but not marked. So the second block gets freed in this case, and the marks are removed:



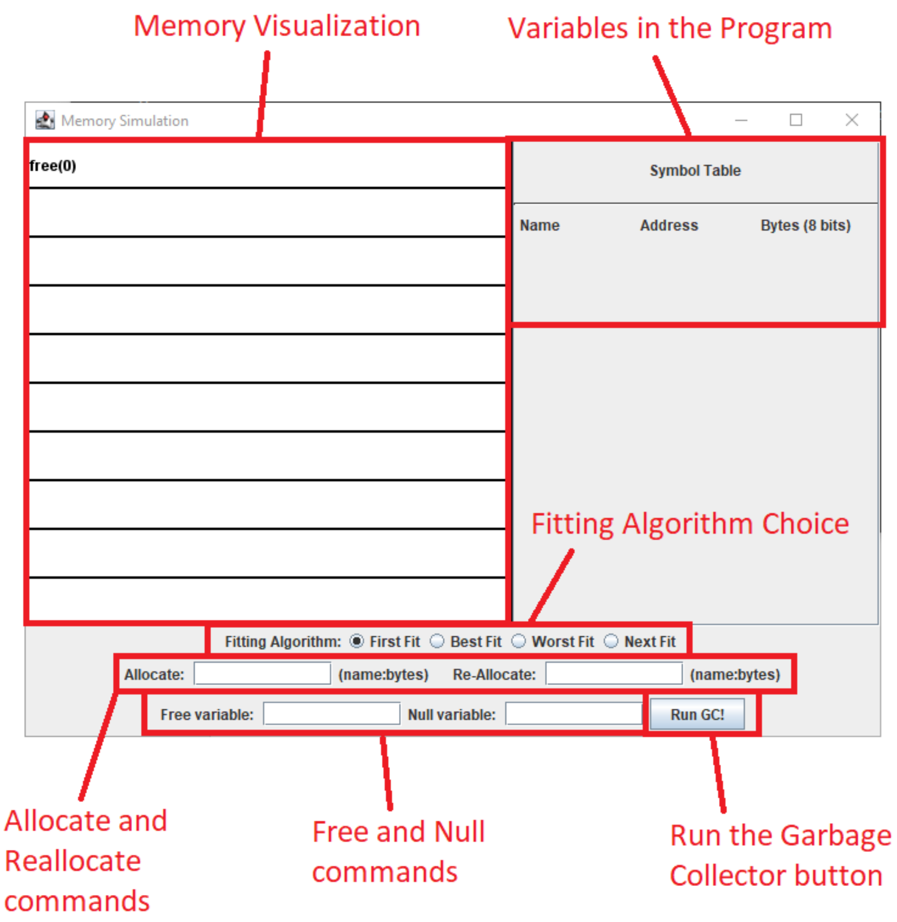
Why are the marks removed? So that you can do mark-and-sweep in the future without working about the old marks!

TL;DR: You probably want to read that... it explains 99% of the whole project.

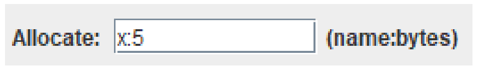
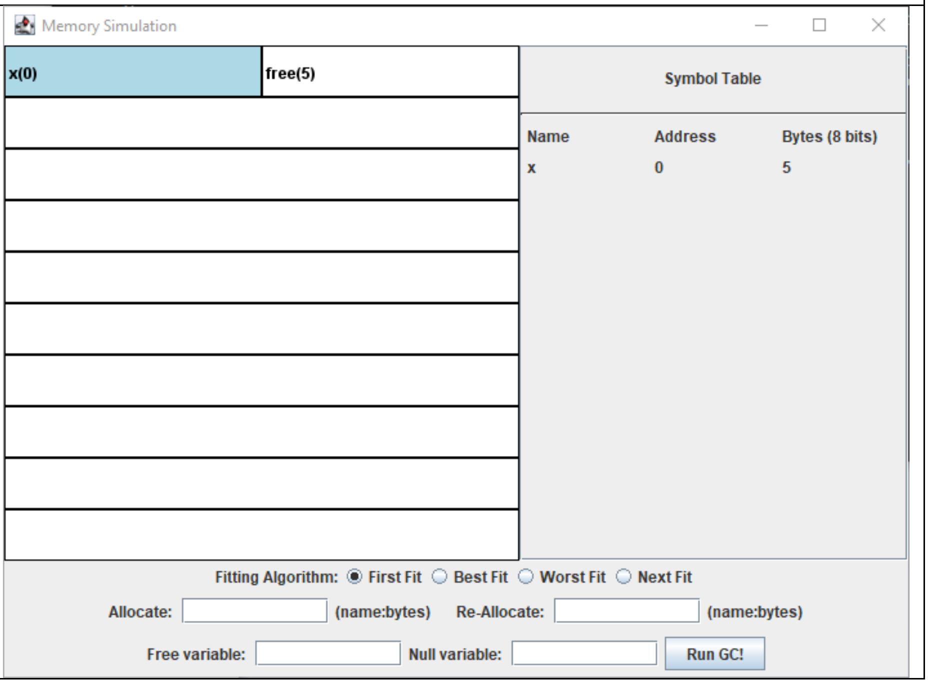
Simulation Introduction

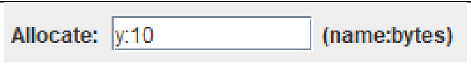
Now that you know about memory, let's run though a quick introduction to the memory simulator you're going to be helping to build. The following command will create a visualization with a 10x10 area of memory (100 bytes):

```
>java Simulator 10 10
```


Memory Visualization
Variables in the Program
Memory Simulation
free(0)
Symbol Table
Name Address Bytes (8 bits)
Fitting Algorithm Choice
Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit
Allocate: (name:bytes) Re-Allocate: (name:bytes)
Free variable: Null variable: Run GC!
Allocate and Reallocate commands
Free and Null commands
Run the Garbage Collector button

User Action + Comments	Simulation Result

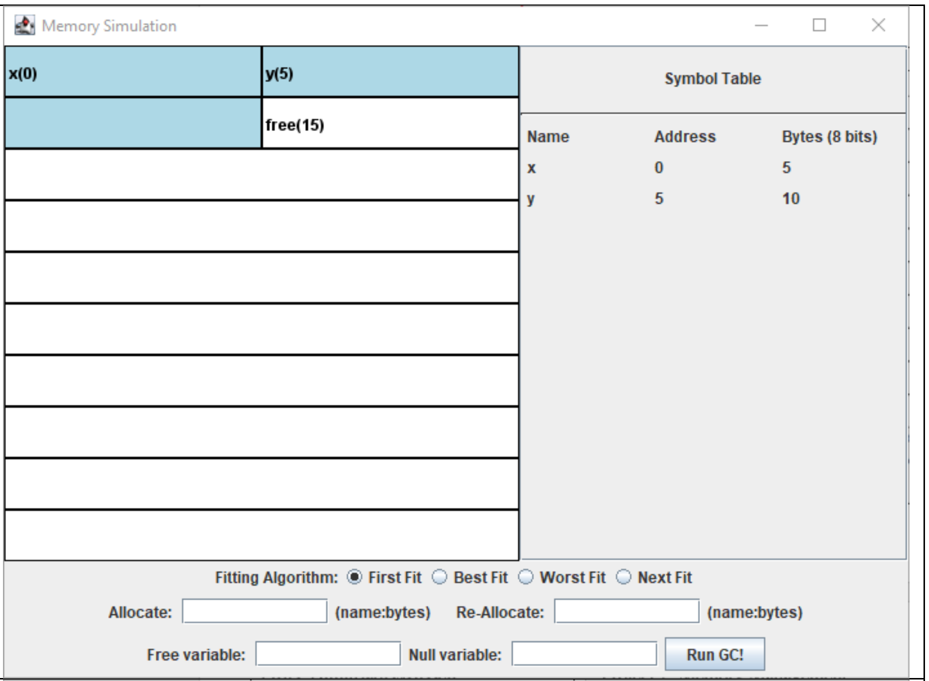
 Allocate: x:5 (name:bytes)
 	
`Allocating 5 bytes in memory for a new variable x.`
 	
`Variable x appears in the symbol table (list of variables) and also appears in the memory area.`
	 Memory Simulation
	x(0) free(5)
	Symbol Table
	Name Address Bytes (8 bits)
	x 0 5
	Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit
	Allocate: (name:bytes) Re-Allocate: (name:bytes)
	Free variable: Null variable: Run GC!



Allocate: y:10 (name:bytes)

```
Allocating 10 bytes in memory for a
new variable y.

Variable y appears in the symbol
table and appears in the memory area.
```



x(0) | y(5)

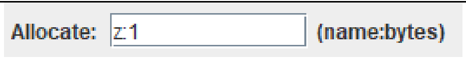
free(15)

Symbol Table

Name	Address	Bytes (8 bits)
x	0	5
y	5	10

Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit

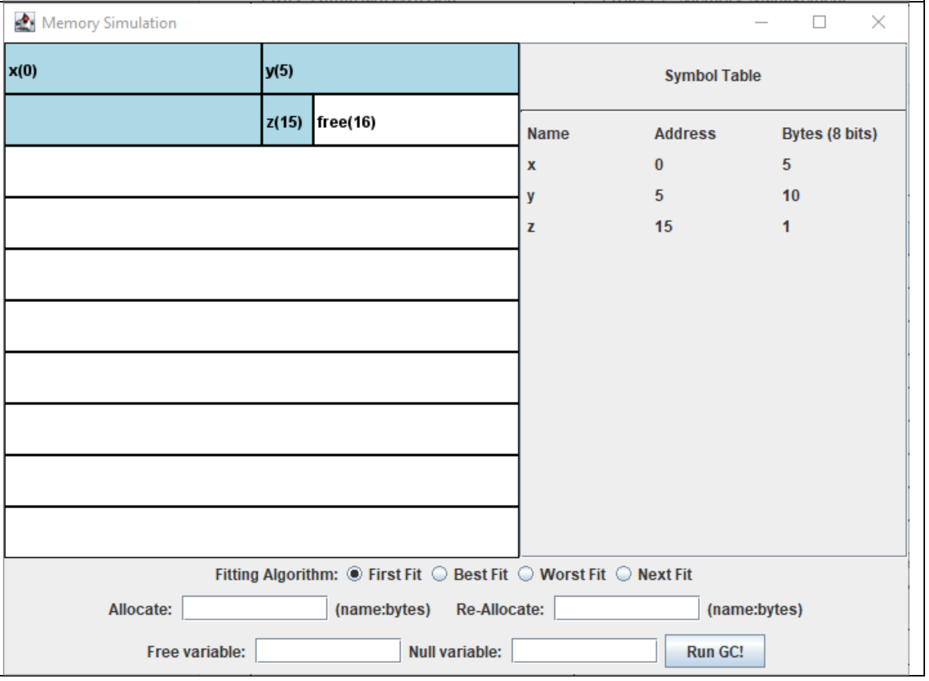
Allocate: (name:bytes) Re-Allocate: (name:bytes)
Free variable: Null variable: Run GC!



Allocate: z:1 (name:bytes)

```
Allocating 1 byte in memory for a new
variable z.

Variable z appears in the symbol
table and appears in the memory area.
```



x(0) | y(5)

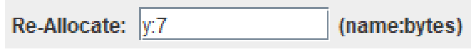
z(15) | free(16)

Symbol Table

Name	Address	Bytes (8 bits)
x	0	5
y	5	10
z	15	1

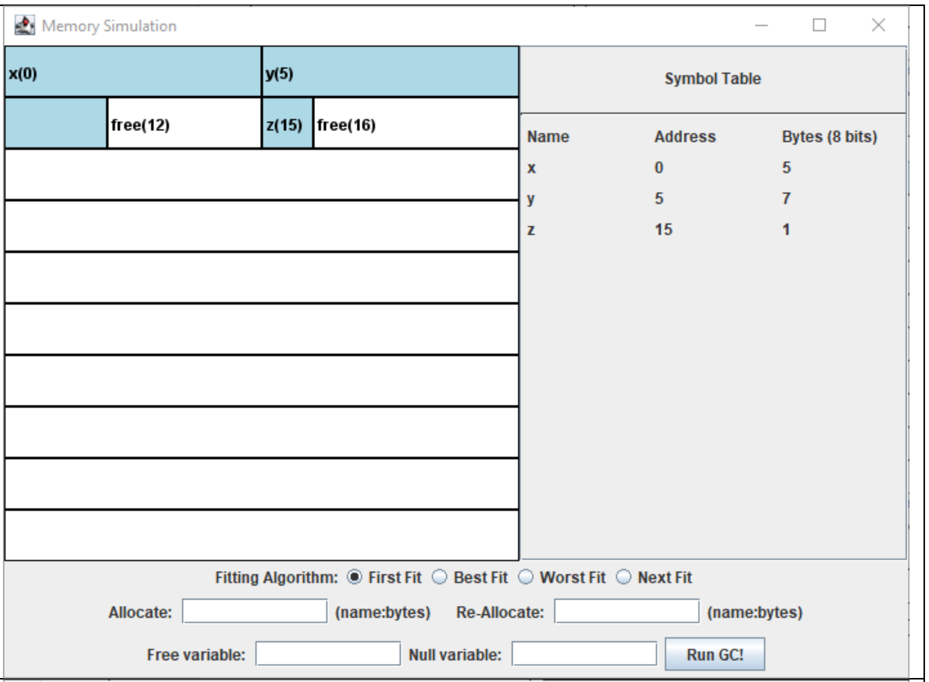
Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit

Allocate: (name:bytes) Re-Allocate: (name:bytes)
Free variable: Null variable: Run GC!


Re-Allocate: y:7 (name:bytes)

```
Re-allocating variable y to only 7
bytes of memory.

Variable y is trying to shrink, so it
maintains the same starting address,
and frees the remaining space.
```



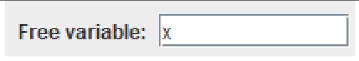
Symbol Table

Name	Address	Bytes (8 bits)
x	0	5
y	5	7
z	15	1

Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit

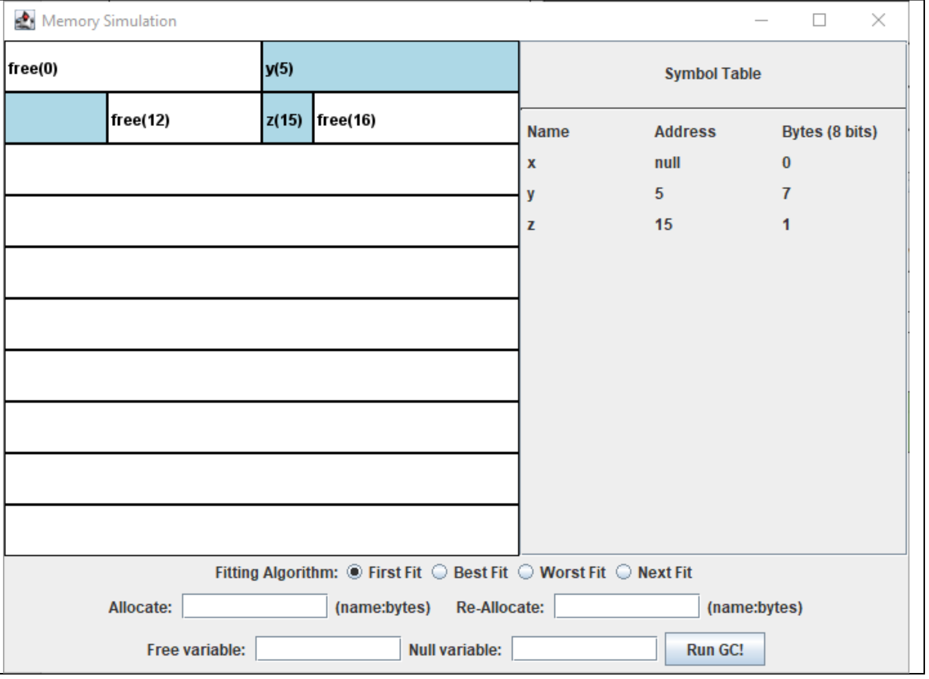
Allocate: (name:bytes) Re-Allocate: (name:bytes)

Free variable: Null variable: Run GC!


Free variable: x

```
Frees variable x, returning those 5
bytes.

Variable x remains in the symbol
table but takes no room in memory.
```



Symbol Table

Name	Address	Bytes (8 bits)
x	null	0
y	5	7
z	15	1

Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit

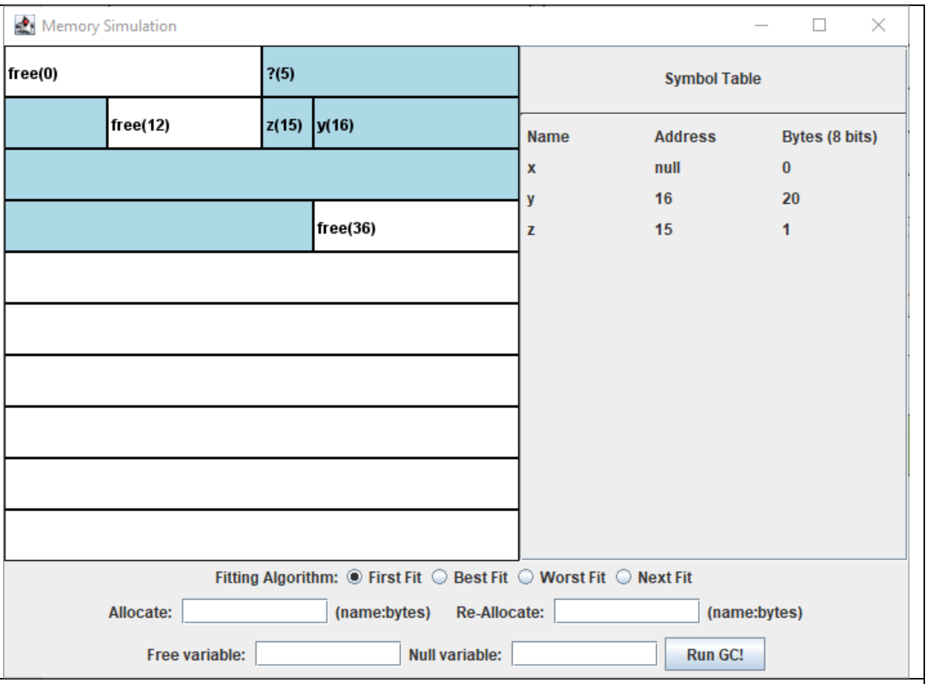
Allocate: (name:bytes) Re-Allocate: (name:bytes)

Free variable: Null variable: Run GC!

Allocate: y:20 (name:bytes)

```
Variable y is given new memory (20
byres), but the old memory was never
freed. So there is now garbage
sitting in memory.

This is what happens when Java
references are reassigned, or when
you forget to manually free memory in
languages like C.
```



Symbol Table

Name	Address	Bytes (8 bits)
x	null	0
y	16	20
z	15	1

Fitting Algorithm: ● First Fit ○ Best Fit ○ Worst Fit ○ Next Fit

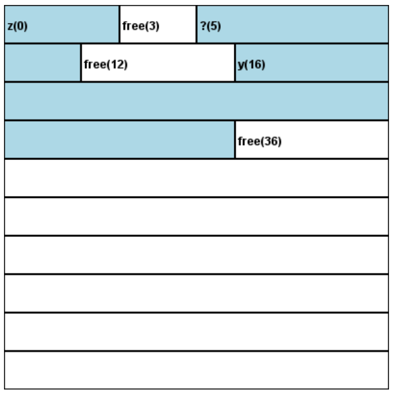
Allocate: (name:bytes) Re-Allocate: (name:bytes)

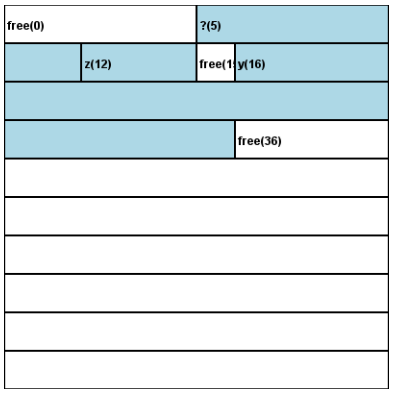
Free variable: Null variable:

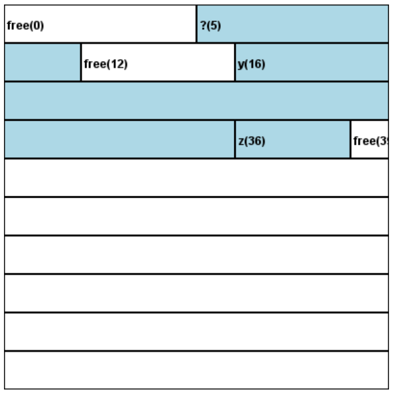
```
IMPORTANT: The next four images show
what would happen when re-allocating
z given different placement policies
```

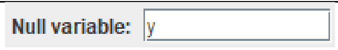
Re-Allocate: z:3 (name:bytes)

```
Top Left – First Fit
Top Right – Best Fit
Bottom – Worst Fit or Next Fit
```

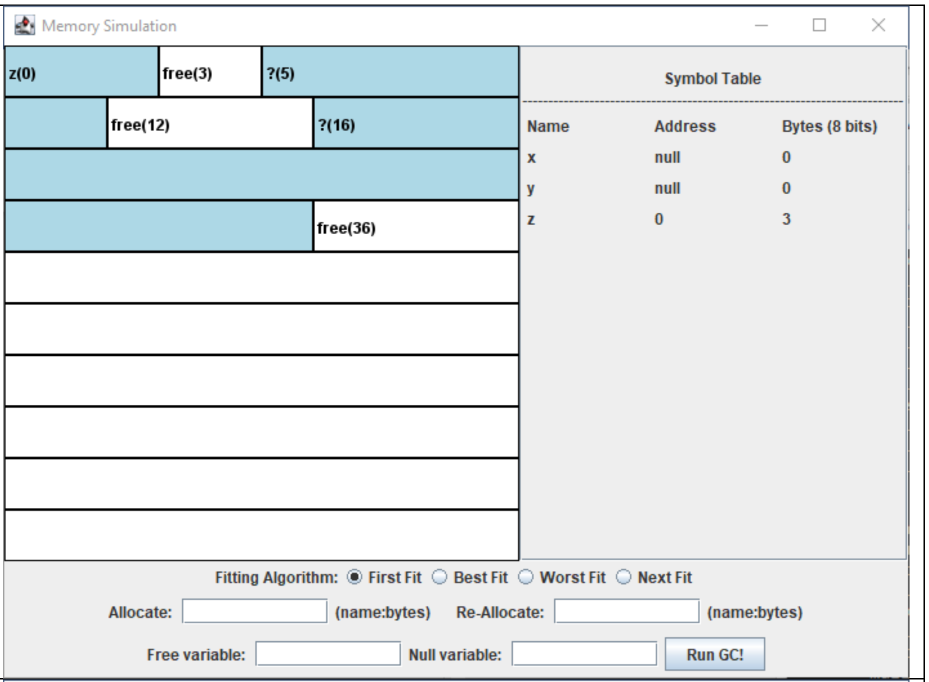







Null variable: y

Assigning a variable to null without freeing the memory will also create garbage :(


Memory Simulation

z(0) | free(3) | ?(5) | free(12) | ?(16) | free(36)

Symbol Table

Name	Address	Bytes (8 bits)
x	null	0
y	null	0
z	0	3

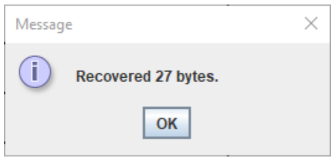
Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit

Allocate: (name:bytes) Re-Allocate: (name:bytes)

Free variable: Null variable: Run GC!

Run GC!

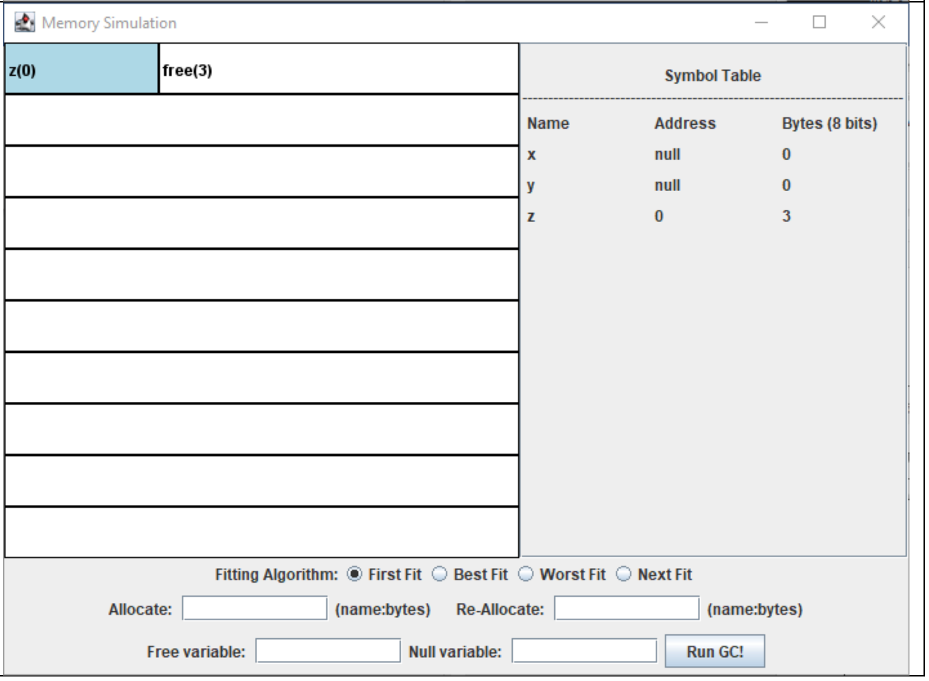
Running the garbage collector will recover the abandoned memory and give you a nice little message about how useful it is!


Message

Recovered 27 bytes.

OK

Isn't a garbage collector great? Java made a good choice!


Memory Simulation

z(0) | free(3)

Symbol Table

Name	Address	Bytes (8 bits)
x	null	0
y	null	0
z	0	3

Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit

Allocate: (name:bytes) Re-Allocate: (name:bytes)

Free variable: Null variable: Run GC!

Warning: There is little to no error checking on variable names and the GUI doesn't resize well. Prof. Russell is not a professional GUI programmer and is simply amused that you can create variables " " and "10".

TL;DR: The output of the simulator is above, this is what you can use to check that your algorithms are working correctly once you've got a code skeleton.

How Can You Help?

Before you start coding, make sure you have a solid understanding of what the end program will produce and a good understanding of what code has and hasn't been written. Again, you are "working with your professor" for this project, so there is a lot more code already written. You are just writing the *support code* for the simulator, *not* the simulator itself.

Code Overview

You're going to write just one class for this project, although you may add additional classes if you want. That class is called **`MemMan`** which represents the memory management part of the computer. To do this you are going to use two additional classes which are already written:

1. **`MemBlock`** – represents a block of memory in the computer. It is basically a tuple: (address, size, isFree).

2. **`BareNode`** – represents the node of a linked list of **`MemBlocks`**.

You may add any public, private, or protected methods you want to these classes, but the provided/required classes, methods, and variables must still work and remain unmodified. You cannot change provided code in any way.

Here are some things which are going to annoy you if you're new to Java (which we expect you are 😉):

1. **`MemBlock`** only has final instance variables. This means you need a new instance just to change any one item in the tuple. This is done on purpose to force you to deep copy these blocks when you want to change something. It's a pretty common technique, but not one you may have seen before.

2. **`MemBlock`** and **`BareNode`** both have publicly accessible instance variables. Your Java teacher (correctly) told you that this was a bad style choice. However, there are two things about this which are important. First, **`MemBlock`**'s instance variables are final, so it doesn't matter that they're public, they are properly encapsulated such that no one can edit them incorrectly (which is the purpose of encapsulation). Second, **`BareNode`** is a linked list node class that's going to be used throughout an entire program. It is very common to allow linked list nodes to have accessible **`data`** and **`next`** components because they are generally understood by programmers (as is the consequences to editing them). You can find this type of thing in Java professional code if you look at **`System.out`** and **`System.err`**; the writers of Java knew people would be annoyed at having to type **`System.getOut().println()`**, so they allowed you to just access the variables directly.

3. We're passing around "bare nodes" in this program, but it is usually recommended in Java that you "hide" the nodes from other parts of a program, e.g. create a linked list library class and encapsulate the nodes. While this is *definitely* what you want to do if you are writing a *library* class called "LinkedList", here we're using linked lists as a tool for something completely different. We want the "outside program" to get pointer/reference to our nodes so that they can hand them back to us later. This allows us to have fast access to the memory they want to edit.

What goes in `MemMan`*?*

The following <u>*public*</u> methods are required:

1. A **`constructor`** of your choice, preferably a protected one – See next method.

2. **`static MemMan factory(int type, BareNode head)`** – We don't know if you want to put everything in **`MemMan`**, make subclasses of **`MemMan`**, or do some other crazy make-um-ups to get different placement policies. But if we tell you to give us a **`MemMan`** of with a certain placement policy (a type), you need to give us back one that works. This **`factory()`** method will take in the parameters we care about (a policy type and a memory block), and you will give us back an *instance* that works how we want (or null if the type or head pointer is invalid). This will allow you to build the constructor however you want for **`MemMan`**, subclass **`MemMan`**, and/or do a lot of other cool things. You may assume these parameters are valid (valid type and a valid non-null head).

3. **`BareNode getHead()`** – We're going to allow users to "switch" memory managers so they need to be able to "hand off" the memory from one memory manager to another. This method will return the head of the linked list of memory blocks so that it can be handed to another manager. *Required Big-O: O(1).*

4. **`BareNode malloc(int size)`** – Update the memory list to allocate memory of the correct size and in the correct place (using the placement policy). Returns the block of memory that's been allocated (so that the program can use it as a pointer) or null if it wasn't possible to allocate the memory. You cannot allocate memory for invalid sizes (must be at least one byte) and you cannot allocate memory if there is not enough space anywhere. *Required Big-O: O(n) where n = the number of blocks (not bytes) in memory.*

5. **`boolean free(BareNode node)`** – Given a node, free it. Coalesce if necessary. Return true if free was possible, return false otherwise. You cannot free something that is already free. You cannot free something when the pointer (bare node) is null. *Required Big-O: O(1).*

6. **`BareNode realloc(BareNode node, int size)`** – Given a node and a size, reallocate the memory. Return the new block (or the resized block) if you were able to reallocate, otherwise return null. You cannot reallocate with invalid sizes, invalid pointers, or when you don't have enough memory. *Required Big-O: O(1) when shrinking or expanding, otherwise O(n) when where n = the number of blocks (not bytes) in memory.*

7. **`int garbageCollect(Set<Integer> addrs)`** – Perform mark-and-sweep. Return how many bytes of memory you recovered. Set is an interface in the Java Collections Framework which you can see more information on here: https://docs.oracle.com/javase/8/docs/api/java/util/Set.html. You may assume that this set is non-null but not that it is not-empty. Your code should work with an empty set. *Required Big-O: O(n) where n = the number of blocks (not bytes) in memory.*

8. **`Iterator<MemBlock> iterator()`** – Return an iterator over the blocks of memory so that we can traverse them for drawing. The simulator uses this for the enhanced for loop so only **`next()`** and **`hasNext()`** are required for this iterator. Note that it is not an iterator over **`BareNodes`**, but an iterator over **`MemBlocks`**. This iterator should go from the front of the linked list to the back (not in reverse, random order, etc.). *Required Big-O: O(1) for both methods of the iterator.*

Final Warning: You may NOT create or use any arrays anywhere in your part of the program. Using arrays, in any way, will result in a 0 on this assignment (the goal is to learn linked lists!). The only exception to this is the main test code which can have a String[] in the parameter list.

TL;DR: You're just writing MemMan, but you'll need to read this completely as you write it. There are no in-code comments for this.

Testing?

We highly recommend slow, incremental testing of your program. Get a code skeleton first, then slowly add in features one at a time. Once you think everything is working, test, test, test. Make a **`main()`** in **`MemMan`** for tests you want to run.

To assist with testing, we have provided one additional feature of the simulator: you can write a little program and use that to "pre-load" your memory (for fast testing of advanced scenarios). Here is an example program:

```
0
x = newmemory(5)
y = newmemory(10)
z = newmemory(1)
realloc(y,7)
free(x)
y = newmemory(20)
realloc(z,3)
y = null
```

The first line gives the type id of the memory manager you want (see **`MemMan`**). You cannot change a memory manager in the middle of a program in the same way you can with the simulator, and the memory manager will be reset to First-Fit after the program is run. After that, you can see the syntax for creating new memory, reallocating memory, freeing memory, and assigning a variable to null.

The garbage collector cannot be run with a program and unfortunately many variable names (such as those with spaces, commas, equal signs, etc.) can confuse the extremely simple parser for this, so keep it simple. The program parser is a really simple and you can easily break it by doing odd things. While we won't be improving the parser, we welcome proposals for better code parsers if anyone feels strongly about it :P

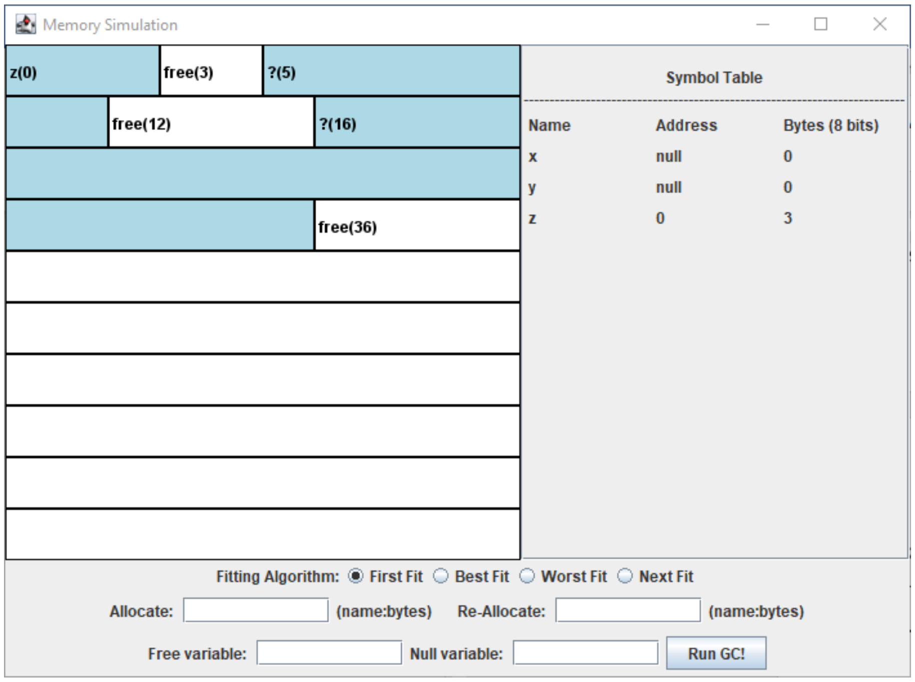
You can run the simulator with your program file by simply adding a path to it after the other command line args. For example, if you are in your user folder and demoprogram.txt is up one directory, you can use this command:

```
>java Simulation 10 10 ../demoprogram.txt
```

And it should produce the following summary of the program it ran (and try to tell you where it got confused if it couldn't run the whole program):

```
Set MemMan to type 0
        malloc "x" to 5 bytes
        malloc "y" to 10 bytes
        malloc "z" to 1 bytes
        realloc "y" to 7 bytes
        free "x"
        malloc "y" to 20 bytes
        realloc "z" to 3 bytes
        null "y"
```

Then you'll get the GUI window with the appropriate state:



Fitting Algorithm: [✓ First Fit] Best Fit Worst Fit Next Fit

Symbol Table

Name	Address	Bytes (8 bits)
x	null	0
y	null	0
z	0	3

Note: Yes, you may share test "programs" (like demoprogram.txt) on Piazza publicly, along with screenshots of the memory/GUI you expect. You may *never* share *real* code publicly on Piazza or anywhere else (individual effort…).

TL;DR: Don't just write code without testing. Write your project WHILE testing. You can write little test programs using a mini-programming language if you want.