Homework 10: Memory Management System

This exercise has three objectives that come together in a unified context. First, you will practice the art of creating and manipulating linked lists. Second, you will learn the general skill of writing exceptions. Third, you will learn how operating systems manage the computer's memory, efficiently and elegantly. They do it using linked lists, and exceptions.

1. Background: Memory Management

A typical computer (like your PC, or cellphone) is equipped with a main memory unit, also called "RAM". A section of the RAM, known as "heap", is dedicated for storing the objects and arrays that running programs create and process. Throughout this document, when we say *memory*, we mean the *heap section* of the RAM. With that in mind, let us assume that the memory is a sequence of 32-bit values, each having an address. Following convention, we call these addressable 32-bit values *words*, and the number of words in the memory *size*. The address of the first word is assumed to be 0, and the address of the last word is *size* – 1.

When programs run on the computer, they create new arrays and objects which must be allocated memory space. When these objects and arrays are no longer needed, the memory space that they occupy can be recycled. The agent that performs the allocation and recycling tasks is a *Memory Management System*, or MMS, which is part of the host operating system. For example, let us illustrate how the MMS comes to play in the context of the following code segment:

```
public class Point {
    int x;
    int y;

    // Constructs a new point
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // More Point methods follow.

}
```

```
public class SomeClass {
    ...

    Point p = new Point(5, 12);

    ...
}
```

Let us track what happens behind the scenes when the client-side statement **`new`** is executed. This statement invokes the `Point` class constructor, which starts running. More accurately, the *low-level code generated by the compiler* starts running. Among other things, this low-level code tells the MMS: "I need a memory space of 2 words for this new object that I am asked to construct".

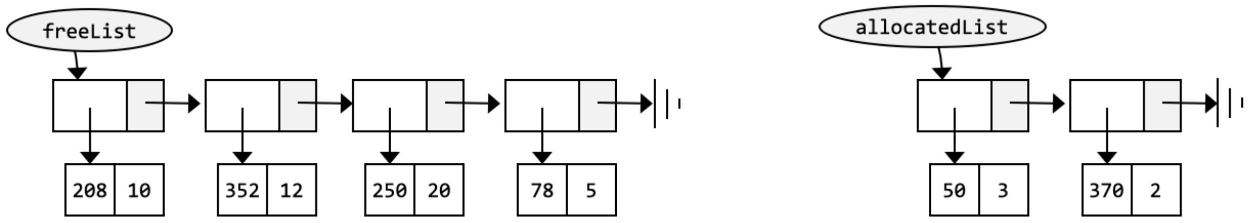
How does the compiler know how much memory space is needed? This can be readily calculated from the number and size of the *fields* of the class to which the constructor belongs. For example, since a `Point` object is represented by two `int` values, the compiler figures out that each `Point` object needs to occupy two words in the computer's memory. With that in mind, the code generated by the compiler tells the MMS: "I need a memory block of length 2". The MMS does some magic, and tells the calling code: "here is an available 2-word memory block; its base address is 5066252" (or some other number between 0 and the memory *size* – 1).

This number is precisely the value that the constructor returns to the caller. Thus, when all the dust clears, the client-side `p` variable, which is actually a pointer (like all object variables), is set to 5066252, the base address of the newly constructed `Point` object.

How does the MMS perform the memory allocation magic? That's what this project is all about.

2. Memory Management System

At any given point of time, the MMS maintains two lists. The first list, called `freeList`, keeps track of all the memory blocks which are available for allocation. The second list, called `allocatedList`, keeps track of all the memory blocks that have been allocated so far. Here is an example of the two lists, at some arbitrary point of time:



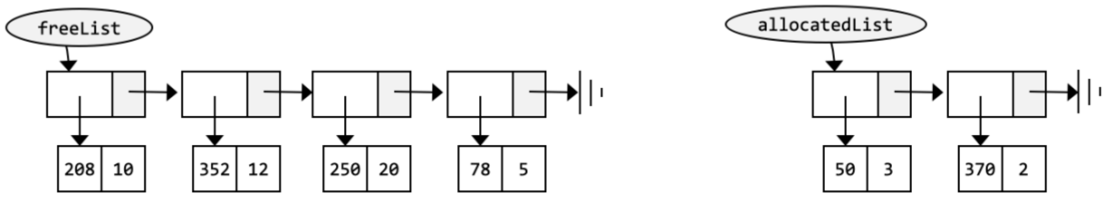
Each memory block is represented by two fields: the base address of the memory block (an address), and its length, as two integers. For example, the first block in the `freeList` shown above starts at address 208, and is 10 words long. Note that when we say *memory block*, we don't mean the memory segment itself. Rather, we mean an object that represents its base address, and length. This is all the information that the MMS needs in order to allocate and recycle memory.

The MMS supports two key operations:

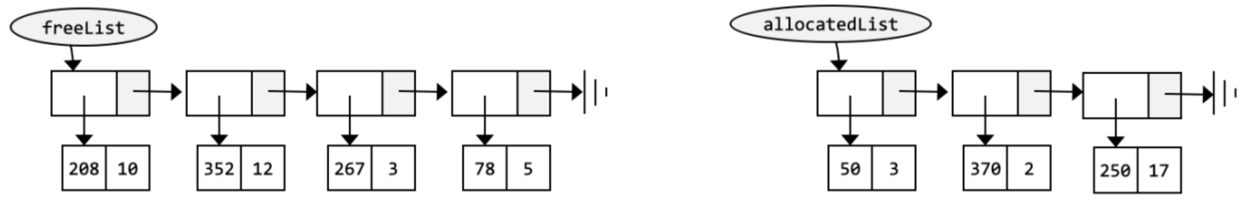
Memory allocation is carried out by the `int **malloc**(int length)` method. This method, whose name derives from "memory allocation", searches the `freeList` for a block which is *at least* `length` words long. If such a block is found, a block of `length` words is carved from it, and handed to the caller. Said more accurately, the method returns the base address of the carved block.

Memory recycling is carried out by the `void **free**(int baseAddress)` method. This method searches the `allocatedList` for a block whose base address equals `baseAddress`. If such a block is found, the method removes it from the `allocatedList`, and adds it to the end of the `freeList`.

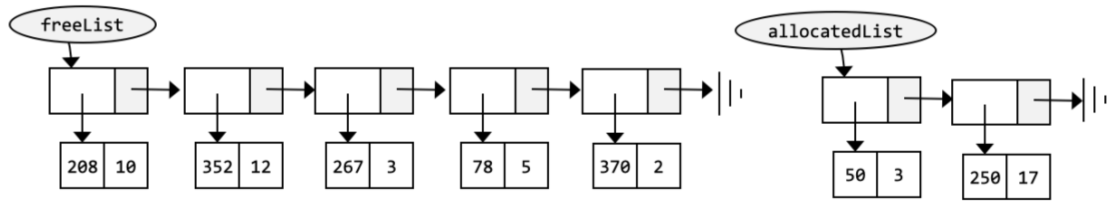
A picture is worth a thousand words, so here is one (next page):



Following `malloc(17)`:

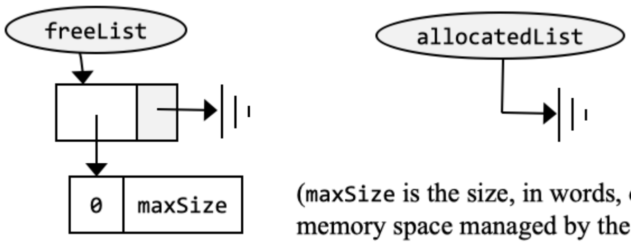


Following `free(370)`:



We see that `malloc` allocates memory blocks, while `free` recycles memory blocks. The former method is called by class constructors of running programs, and the latter method is called by the garbage collector. These calls occur behind the scenes, so application programmers need not worry about them. That's one of the greatest benefits of high-level programming – no need to manage memory. The people who *do* worry about memory allocation algorithms and implementations are the developers of operating systems and infrastructure software. These developers make a nice living, so read on.

Initialization: When you boot up your computer, the OS executes a set of initialization routines. Among other things, the MMS initializes the two lists, as follows:



(`maxSize` is the size, in words, of the memory space managed by the MMS)

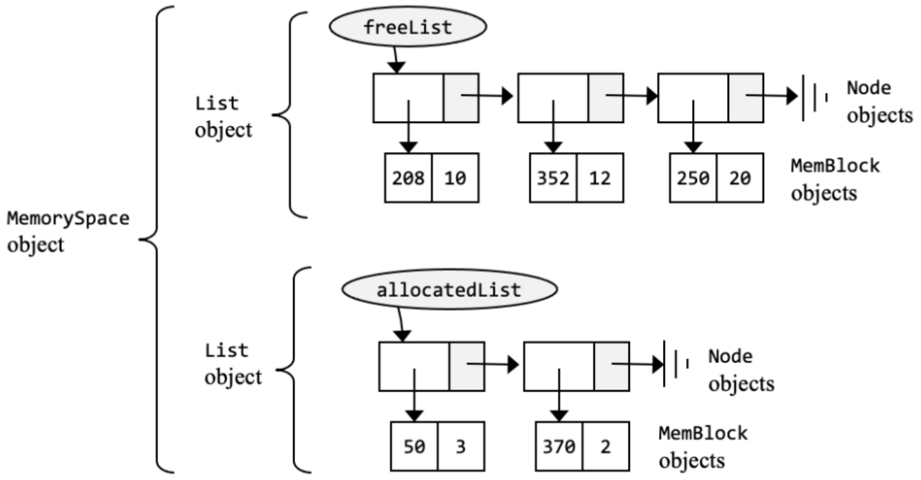
In other words, before programs start running and using memory resources, the `freeList` contains a single block which represents the entire memory space, and the `allocatedList` contains no blocks.

Note that so long as the `free` method has not been called, `freeList` will consist of one block only, and new blocks will be carved away and allocated from this single block. As a result, the block will become shorter and shorter. At some point though the garbage collector will kick into action, and start calling `free`. Each such call will cause the `freeList` to grow by one recycled block.

This is the fundamental architecture that enables dynamic memory allocation and recycling. Before delving into the details, allow yourself a minute or two to appreciate the beauty of this memory management solution.

3. Architecture

The MMS (Memory Management System) can be implemented in several different ways. We present an object-oriented implementation, consisting of five classes. The following image shows how four of these classes work together to represent the system's building blocks. The fifth class is a general purpose list iterator, to be discussed later.



There are essentially two ways for implementing such an architecture in Java. One is to use a *generic* `List<T>` class, designed to represent generic `Node<T>` objects, and to instantiate these classes using the `<MemBlock>` “parameter” (as we learned in lecture 10-2). Another approach is to implement a customized and optimized `List` class, designed specifically for the MMS needs. We have chosen the second approach. We now turn to describe the MMS classes, in the order in which we recommend to inspect and implement them.

<u>`MemBlock`</u>

This simple class represents a memory block. Note that the actual memory block is not represented. Only the *base address* and the *size* of the block are represented, as two fields of every `MemBlock` object. The code of this class is given. For more details, see the `MemBlock` class API.

<u>`Node`</u>

This class represents a node in a linked list. A `Node` object consists of two fields: a pointer to a `MemBlock` object, and a pointer to a `Node` object. The code of this class is given. For more details, see the `Node` class API.

<u>`List`</u>

This class is similar, but not identical, to the `List` class discussed in lecture 9-2. The class represents a list of linked `Node` objects. Since each `Node` object points to a memory block, it is also reasonable to say that the class represents a list of `MemBlock` objects. With that in mind, when we say "add / remove a given memory block to / from the list", we actually mean "add / remove the node that contains the given memory block to / from the list".

The only `List` class methods that are needed for memory management are `addLast`, `remove`, and `indexOf`. You will notice though that the `List` class includes more methods that are not used by the MMS. You are to implement these methods also, in order to practice your list management skills. For more details, see the `List` class API.

4. Implementation

The two core classes of the MMS are `List` and `MemorySpace`. This section discusses the implementation of these two classes (the code of the other classes in this project is given).

A list – an instance of the `List` class – is an ordered sequence of `Node` objects, which serve as modular connectors. There are two ways to think about these `Node` objects. First, a `Node` is simply an address in memory. Second, a `Node` is an object that “carries”, or “embeds”, a `MemBlock` object. Both views are correct, and useful. This dual view of objects is critically important for understanding and writing object-oriented code.

A `List` object consists of three fields: `first`, which is a `Node` that points to the first element in the list (that’s an example of the “address view” of a `Node`), `last`, a `Node` that points to the last element in the list, and `size`, the number of elements in the list. Each element in the list is a `Node` that holds a `MemBlock` object (that’s an example of the “object view” of each `Node`).

The `List` class features one constructor and three key methods. The `List` constructor builds an empty list and sets its size to 0. The `addLast(MemBlock)` method creates a `Node` object that points to the given memory block, and adds it to the end of the list. The `indexOf(MemBlock)` method searches the list and returns the index of the given memory block. The `remove(MemBlock)` method finds, and removes, the given memory block from the list. These are the only `List` methods needed by the MMS. In terms of required efficiency, `addLast` should be $O(1)$, while `indexOf` and `remove` (which uses `indexOf`) are $O(length)$, where *length* is the number of elements in the list.

As mentioned above, the `List` class includes additional methods, and all should be implemented. For your convenience, the “easier” methods (in terms of implementation effort) are declared first. With that in mind, we recommend implementing the `List` class methods in the order in which they appear in the given `List.java` class.

This `List` class provides many opportunities for code reuse, i.e. having methods call other methods for their effect, rather than coding everything from scratch. As usual, we recommend re-using code whenever possible.

ListIterator: This class represents an iterator over a linked list. The `ListIterator` class comes to play via the following `List` class method:

```
 /** Returns an iterator over the elements of this list. */
public ListIterator iterator() {
   return new ListIterator(first.next);
}
```

This code creates a new `ListIterator` object, and initializes it to start iterating at the list’s first element (following the dummy node). By featuring this public method, the `List` class provides clients with the ability to do list processing, using client-controlled iterations. The code of the `ListIterator.java` class is given. Read it and make sure that you understand it.

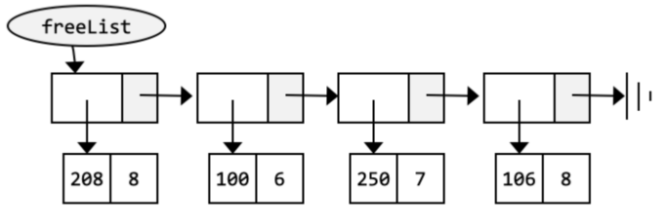
MemorySpace

This class represents a managed memory space, also known as “heap”. Given the size (in words) of the memory segment that we wish to manage, the class constructor creates a new managed memory space. When clients (low level OS programs) need a memory block of some length, they make the method call `malloc(length)`. The method arranges a memory block of size `length`, and returns its base address to the caller. When clients wish to recycle a given object whose base address is `obj`, they make the method call `free(obj)`. The method locates the allocated memory block whose base address is `obj`, removes it from the allocated list, and adds it to the free list.

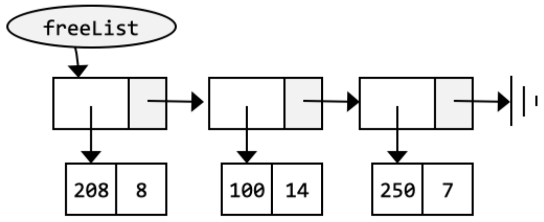
Implementation: The managed memory space is characterized by two lists: `freeList` represents all the memory blocks that are available for allocation, and `allocatedList` represents all the memory blocks that have been allocated so far. The class constructor creates these two lists, using the initialization logic described in page 3 of this document.

The logic of the `malloc()` and the `free()` methods, which are the bread and butter of the MMS, is described in detail in the above pages, and in the `MemorySpace` class API.

Defragmentation: Suppose you want to allocate a block of 10 words, and all the blocks in the `freeList` have sizes that are less than 10. For example:



Note however that the second and fourth blocks in the list can be joined, because they form a continuous block in memory (100 + 6 = 106). This “joining” will yield the following `freeList`:



Once we make this little surgery, we’ll be able to allocate a block of size 10 from the second block in the `freeList`. This technique is called *defragmentation*. Defragmentation takes a fragmented list of memory blocks, and makes it less fragmented. Typically, operating systems do defragmentation periodically, to optimize the memory allocation, or when `malloc` fails to find an available object.

In our MMS, defragmentation is done by the `defrag` method. The method scans the entire `freeList`, and tries to join all the blocks that create continuous and larger blocks in memory. If successful, the result is a `freeList` that has fewer and larger blocks.

5. Development and Testing

Stage 1: `MemBlock`, `Node`

Review the given code of `MemBlock.java`, and `Node.java`,. Make sure that you undersrand both classes. Recommended: Write a test class – you can call it `MemTest.java`, and use it to construct and print several Node objects, each containing a `MemBlock` object.

Stage 2: `List`

We supply a `ListTest.java` class, designed to unit-test the `List` methods. After developing each `List` method, test your code using a test method in `ListTest`.

Important: You are welcome to write tests of your own. The tests that we provide are the minimum. Feel free to make as much testing as you feel necessary, and put the tests that you write in the given `ListTest` class. You will not have to submit this class – it's for your own work and eyes only.

Start by inspecting the `ListTest` class. Make sure that you understand how the test methods (like the given `testAddLast` method) use the `ListTest` class-level variables.

Now turn to inspecting the given `List` class. Write the `addLast` method, and test it using the given `testAddLast` method. Next, implement the `addFirst` method (in `List`), and the `TestAddFirst method` (in `ListTest`), and use the latter to test the former.

Continue in exactly the same manner: For each `List` method, implement the `List` method and a `ListTestMethod` method, and use the latter to test the former.

Don't proceed to the next stage until your `List` class is working properly.

Stage 3: `MemorySpace`

We supply a `MemTest.java` class, designed to test the `MemorySpace` methods. Develop and test first the `malloc` method, and then the `free` method.

When you are done, proceed to implement the `defrag` method. This method is a bonus. Tiy don't have to implement it. But if you will, you will get a 10 points bonus (110 points for a perfect grade instead of 100). If you implement defrag, here are two implementation tips: (1) You will need two list iterators, both operating on the freeList. (2) Don't change a list (add / remove / update elements) while iterating over it. Find another way to do it.

5. Submission

Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines. In addition to the 2 code files, create one PDF document that lists the code of the classes that you wrote, with the indentation. You may want to experiment with several "paste special" options, for transferring the code from your code editor to your word processing software. Whatever you do, the printed code must be well indented, and easy to read a review. Use the font Consolas, or Arial, size 12. The name of this file should be `HW10Code.pdf`.

Compress the following files into a file named `HW10.zip`:

- `List.java`
- `MemorySpace.java`

Upload two files to Moodle: `HW10.zip`, and `HW10Code.pdf`.

Submission deadline: January 2, 2022, 23:55.