Homework 9: Memory Management System

In this exercise you will learn how to construct and manipulate linked lists. As a side benefit, you will also learn how operating systems manage the computer's memory (RAM) resources.

Goals

- Work with linked lists
- Understand and implement a memory management system
- Work with exceptions.

Background: Memory Management

A typical computer (like your PC) is equipped with a physical memory unit (RAM). Let us assume that the memory is a sequence of 32-bit values, each having an address (index) starting at 0. Following convention, we call these 32-bit values "words", and the number of words in the memory "size". Thus the address of the first memory word is 0 and the address of the last memory 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 must be recycled. The agent that performs these tasks is a Memory Management System, or MMS for short, 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 scene when the object-construction statement **`new`** is executed. This statement invokes the `Point(int,int)` 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 in question. For example, since a `Point` object is represented by two `int` values, the compiler asserts that each `Point` object needs to occupy two words in the computer's memory. With that in mind, the code generated by the compiler turns to the MMS and says "I need a memory block of length 2". The MMS does some magic, and tells

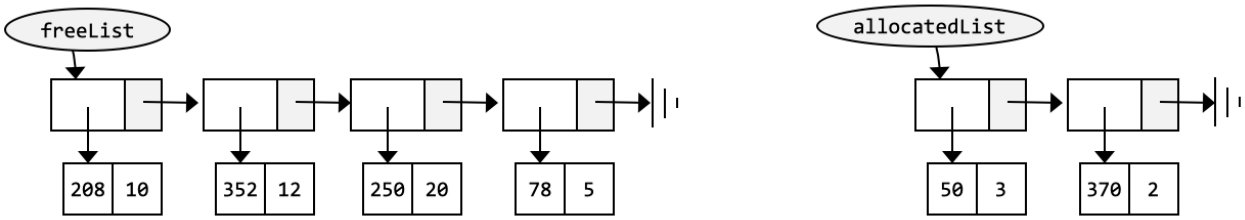
the calling code: "here is an available 2-word memory block; its base address is 5066252" (or some other number between 0 and the memory *size* -1).

This number is precisely the value that the constructor returns to the caller. Thus, when all the dust clears, the `p` variable of the calling code is set to 5066252, which is 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.

The 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:



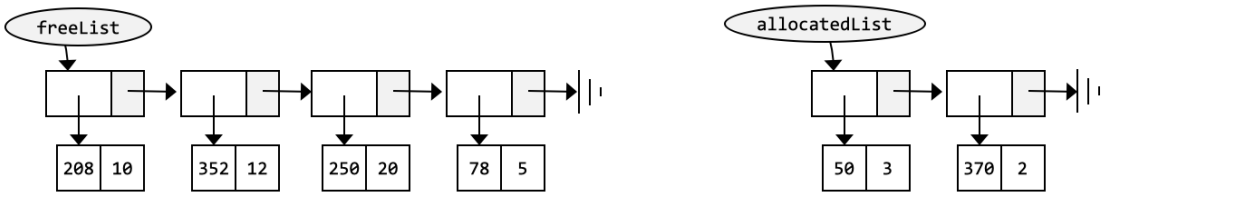
Each memory block is represented by two fields: the base address of the block (a memory address), and its length, in words. For example, the first block in the `allocatedList` shown above starts at address 50, and is 3 words long.

The MMS supports two key operations:

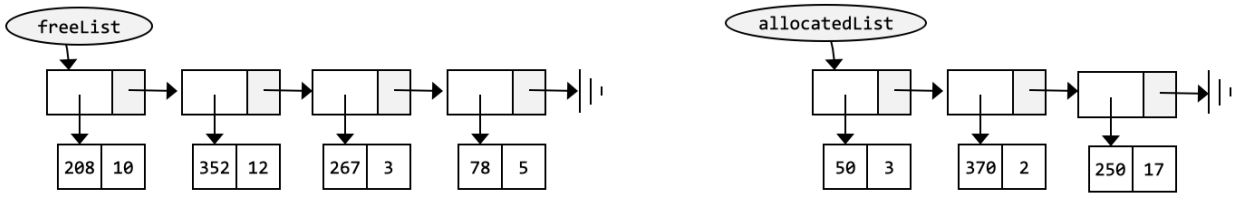
Memory allocation is carried out by the `int` **`malloc`**`(int length)` method. This method, whose name derives from "memory allocation", searches the `freeList` for a block which is *at least* `length` words long. If such a block is found, a block of `length` words is carved from it, and handed to the caller. In practice, the method also performs some additional housekeeping tasks, to be discussed later.

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 block is removed from the `allocatedList`, and added to the end of the `freeList`.

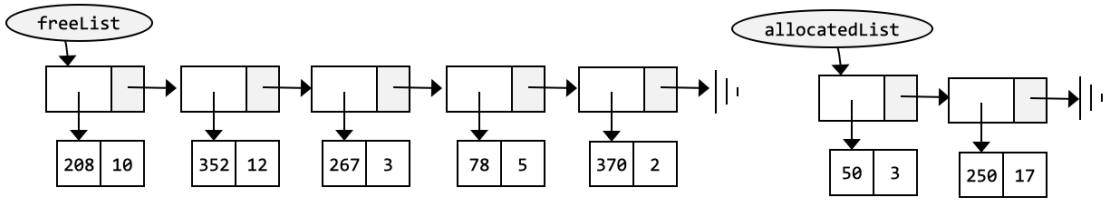
A picture is worth a thousand words, so here is one:



Following `malloc(17)`:

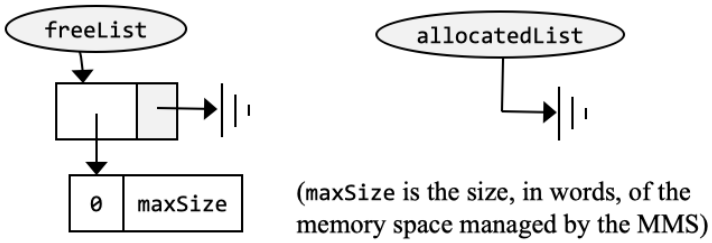


Following `free(370)`:



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 scene, so application developers don't have to worry about them. The people who should worry about memory allocation algorithms and implementations are the developers who write operating systems and infrastructure software. These developers make a nice living, so read on.

Initialization: When you boot up your computer, the OS carries out all sort of initialization routines. Among other things, the MMS initializes the two lists, as follows:



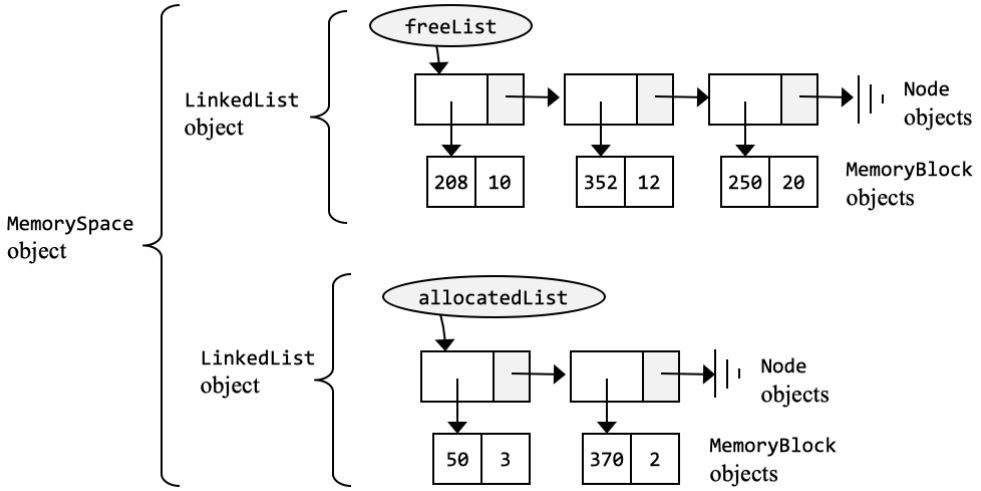
In other words, before programs start running and using memory resources, the `freeList` contains a single block which represents the entire memory space, and the `allocatedList` contains no blocks.

Note that so long as `free()` 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.

Implementation

The MMS (Memory Management System) can be implemented in several different ways. We present an object-oriented Java 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 list iterator, to be discussed later.



There are essentially two ways for implementing such a system. One is to create a *generic* `LinkedList` class implementation, designed to represent `Node` objects of any possible type. Another is to implement a customized `LinkedList` class, designed specifically for the MMS needs. We have chosen the first approach. We now turn to describe the MMS classes, in the order in which we recommend to implement and test them.

MemoryBlock

This very simple class represents a memory block. See the class API for details.

Node<T>

This generic class represents a node in a linked list. Each `Node` object points to an object of type `T`, and to the next node in the list. See the class API for details.

Implementation: a `Node` object consists of two pointer fields. One field points to the data, and the other field points to `Node` object. (In the MMS, the data is a `MemoryBlock`).

LinkedList<T>

This class is similar to the `LinkedList` class discussed in lecture 9-2, but there are many small differences.

The `LinkedList` class represents a list of generic `Node` objects. Therefore, `LinkedList` is also a generic class. With that in mind, when we say "add a given memory block to the list" or "remove a given memory block from the list", we actually mean "add to the list a generic node in which `T` represents the given `MemoryBlock` object " or "remove from the list the generic node that points to the given memory block". See the class API for details.

Why do we need all this "`T`" trouble? Because once we'll be done implementing the generic `LinkedList` class, we'll be able to use it for *any system* that requires linked lists of any type, not only for the MMS.

You will note that the `LinkedList` API includes some methods that are not needed for the MMS implementation. We ask you to implement these methods also, in order to practice your list management skills.

Implementation

A `LinkedList` object consists of three fields: a `first` pointer, which points to the first element in the list, a `last` pointer, which points to the last element in the list, and `size`, the number of elements in the list. The `LinkedList` constructor constructs a new list and sets its size to 0. The `add(int,T)` method creates a `Node` object that points to the given memory block, and inserts this node just before the given location in the list. Once you've developed this method, developing the `addFirst()` and `addLast()` methods should be easy. The former inserts a given object at the list's beginning, and the latter inserts a given object at the list's end. The `get()`, and `indexOf()` methods are self-explanatory, as is the `remove` method. See the API.

The `iterator()` method implementation is as follows:

```
/** Returns an iterator over the elements in this list.
 *  The iterator starts at the first element in this list. */
public LinkedListIterator<T> iterator() {
   return new LinkedListIterator<T>(first.next);
}
```

This code creates a new `LinkedListIterator` object and initializes it to iterate over this list.

LinkedListIterator<T>

This class represents an iterator over a linked list. See the class API for details.

The complete source code of this class is given. You have to explore the code and make sure that you understand it.

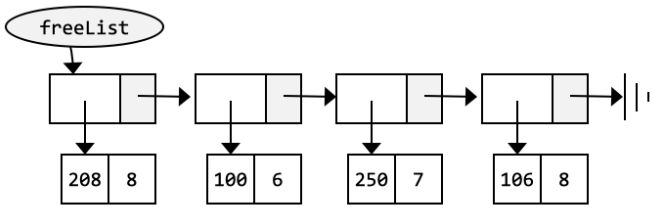
MemorySpace

This class represents a managed memory space. 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 need a memory block of some length, they call the `malloc(length)` method. The method returns the base address of a memory block of size `length`. When clients wish to recycle a given object, say `obj`, they call the `free(obj)` method. This method finds the allocated memory block whose base address is the same as that of `obj` and recycles the block.

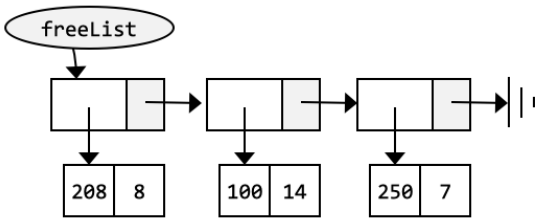
Implementation: The managed memory space is characterized by two lists: `freeList` represents all the memory blocks that are available for allocation, and `allocatedList` represents all the memory blocks that have been allocated so far. The class constructor creates these two lists of `MemoryBlocks`,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

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 of this document, and in the `MemorySpace` API. Both methods make use of `ListIterator` objects to scan the free list and the allocated list for objects of interest.

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



Note however that the second and fourth blocks can be joined, yielding the following `freeList`:



Once we make this change, we'll be able to allocate a block of size 10 from the second block in the `freeList`.

The `deFrag` method in the `MemorySpace` class is designed to carry out such operations, in a general way. The method scans the entire `freeList` and tries to join all the blocks that create continuous and larger blocks in memory. The result is a `freeList` that has fewer and larger blocks.

*Defragmentation is an important part of any MMS. For the purposes of the assignment, though, you are **not** asked to implement or use this method.*

Packages

It is important to note that the project is composed of *two packages*. The first is the `linkedlist` package, which holds the 3 classes needed for managing general-purpose linked lists, of any type. The second package, named `mms`, contains the specific `MemorySpace` and `MemoryBlock` classes. Before implementing each class, make sure you understand the visibility of the fields, and how it might affect your code.

Do not change any package declaration or function signature. Also, do not change the visibility of any field.

Testing

Stage 1: Develop and test the classes `MemoryBlock`, `Node`, `LinkedList`, and `MemorySpace`, in this order. For each method, or set or related methods, write a separate tester, and use it for checking your work.

Stage 2: We provide some stress testing in a class named `Test`. Use it only when you are done with the stage 1 testing. The supplied tests are not comprehensive by any means and you should add tests of your own.

Submission

Submit your solution no later than February 2nd.

Submit the following classes only:

```
Node.java
LinkedList.java
MemoryBlock.java
MemorySpace.java
```