Homework 9:

Linked list Application: a Memory Management System

Introduction to Computer Science

Efi Arazi School of Computer Science

IDC Herzliya, 2016

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
- Get a taste of working with exceptions
- Understand a hairy requirements document (this document).

Background: Memory Management

A typical computer (like your PC, or Vic) is equipped with a physical memory unit (RAM). Let us assume that this memory resource is an addressed sequence of 32-bit values. Following convention, let us 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 objects and arrays 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 {

    private int x;
    private 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 **new** command is executed. This command invokes the **Point(int; int)** class constructor, which starts running. Among other things, the constructor tells the MMS: "I need a memory space of 2 words for this new object that I am asked to construct". (from now on, when we say “the constructor does this and that”, we mean the *compiled code of the constructor*. This low-level code consists of the machine instructions that the compiler generated from the programmer’s high-level constructor code, along with additional low-level code, injected by the compiler).

How does the constructor 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, we can assert that each **Point** object needs to occupy two words in the computer's memory.

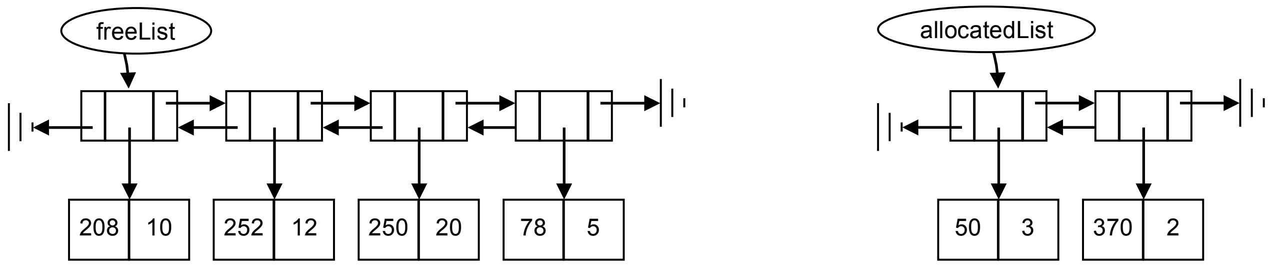
To be more specific, the constructor turns to the MMS and says "I need a memory block of length 2". The MMS does some magic, and tells the constructor: "here is an available 2-word memory block; its base address is 5066252" (or some other number between 0 and the memory size -1).

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. This makes perfect sense: by definition, the **p** variable is designed to refer to objects of type **Point**.

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:

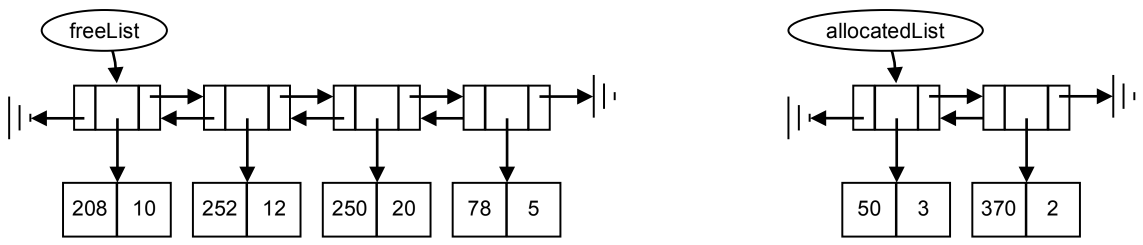


Each memory block is represented by two fields: the base address of the block (a memory address), and its length, in words. For example, the first block in the above **allocatedList** starts at address 50 in the memory and is 3 words long.

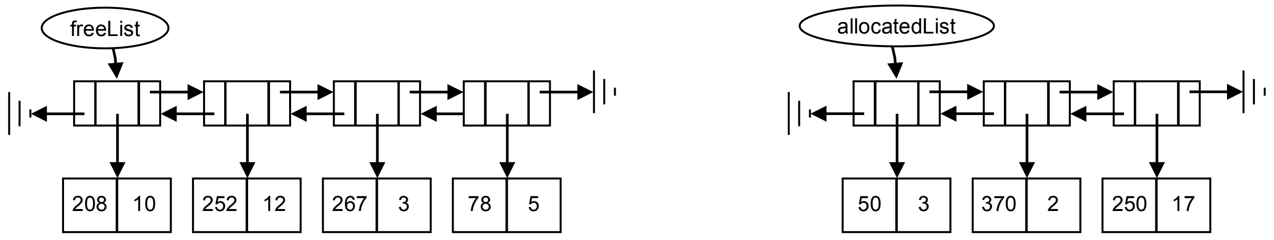
The MMS supports two key operations:

- 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 **length** is carved from it and handed to the caller. The method 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 appended to the **freeList**.

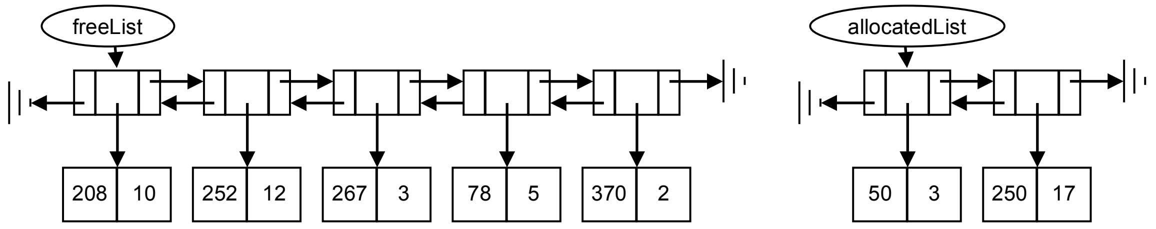
A picture is worth a thousand words, so here is one:



Following malloc(17): (allocate a block of length 17)

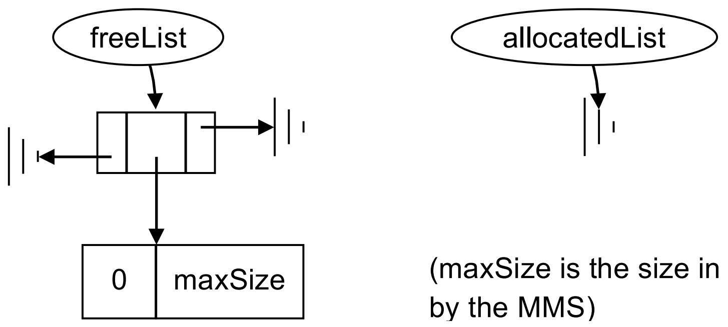


Following free(370): (recycle the memory space of the object whose base address is 370)



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 high-level application programmers don't have to worry about them. However, people who know how to write behind-the-scene system code 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:



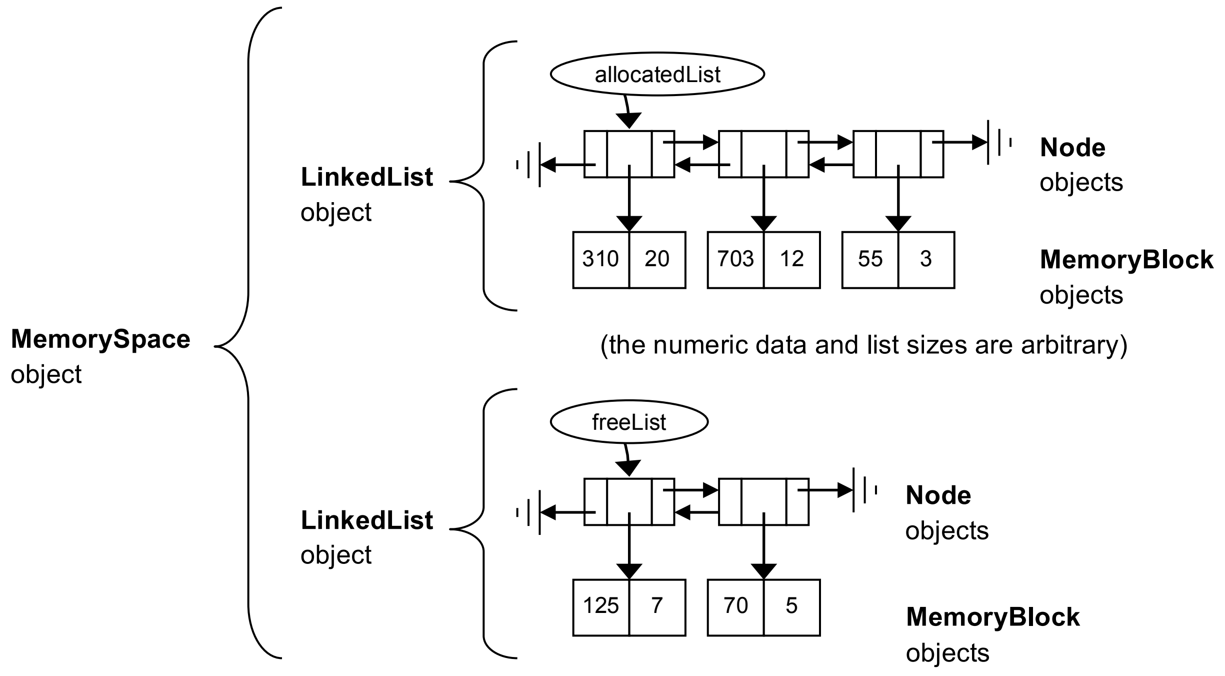
(maxSize is the size in words of the memory space managed by the MMS)

We see that 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 however the garbage collector will kick into action, and start calling **free().** Each such call will cause the **freeList** to grow by one recycled block.

Class Description and Implementation

The Memory Management System consists 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.



We now turn to describe these 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

This class represents a Node in a doubly-connected linked list. Each Node object points to a MemoryBlock object. See the class API for details.

Proposed implementation: a Node object consists of three pointer fields. One field points to a MemoryBlock object, and the two other fields point to Node objects.

LinkedList

This class is similar in spirit to the **LinkedList** class discussed in lecture 11-2, but there are many small differences.

This class represents a list of **Node** objects. Since each Node object points to a memory block, it is also reasonable to say that the class represents a list of **MemoryBlock** objects. 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 node pointing to the given memory block" or "remove from the list the node that points to the given memory block". Note that **MemoryBlock** objects are always embedded within **Node** objects, and write your code accordingly. See the class API for details.

Proposed implementation: A LinkedList object consists of three fields: a head pointer, which points to the first element in the list, a tail 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 **insert()** method inserts a given memory block in a certain index (location in the list). Once you write this method, writing the **append()** and **prepend()** methods should be easy. The former inserts a given memory block at the list's beginning, and the latter inserts a given memory block at the list's end. The **getNode()**, **getBlock()**, and **indexOf()** methods are self-explanatory, as are the three over-loaded **remove** methods. See the API.

This class presents many opportunities for code reuse, i.e. having methods call other methods for their effect, rather than coding everything from scratch. We recommend to re-use code whenever possible.

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

```
/**
 * Returns an iterator over the elements in this list
 */
public LinkedListIterator iterator() {
   return new LinkedListIterator(this);
}
```

This code creates a new **LinkedListIterator** object, and initializes it to iterate over an instance of the **LinkedList** class.

Once again, the **LinkedList** class is quite similar to the singly-linked list that was presented in class. For each required operation, we recommend drawing a linked list and simulating the operation on paper, and then using the visual insights to write the code itself.

LinkedListIterator

This class represents an iterator that works on a given linked list. See the class API for details.

There are several ways to implement iterators in Java, and this implementation represents one of them. 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 certain object, say obj, they call the **free(obj)** method. This method finds the allocated memory block whose base address is the same as obj, and recycles the block.

Proposed 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 already. The class constructor creates these two lists, using the initialization logic described at the top of page 4.

The logic of the **malloc()** and the **free()** methods, which are the bread and butter of the MMS, is described in detail in page 3 and in the class API. Both methods make use of **ListIterator** objects to scan the free list and the allocated list for objects of interest.

Testing

We propose developing and testing the classes in the same order in which they were presented. We provide some basic testing in a class called **Test**, but its your responsibility to write your own test code for all of the methods that you have to develop.

Submission

Submit your solution no later than January 29, 2017, 23:59.