Data Structures and Algorithms

Assignment #2

Due: Wednesday April 12

In this assignment, you will have to simulate a "private memory manager".

Assume that some application wants to manage its own "private memory". At initialization, it asks for a large block of memory of size M (M units; you don't need to know what a unit is; it could be a word, a byte etc.) which is the size of the application's "private memory".

The application will ask for allocation and deallocation of blocks of various sizes, between 10 and 100, with requests for size 50 being treated differently from others. The "private memory manager" that you have to implement keeps track of the memory that is free and the memory that is busy (i.e., blocks that have been requested by the application and have not been freed). In the following, a "free" block is a consecutive amount of memory that has not been allocated. A "busy" block is a consecutive amount of memory that has been allocated.

Allocation can take one of two forms:

- If the requested block size is 50, it is allocated via a stack that grows from the highest part of private memory downward.
- If the size is not 50, say s, find the first block of size greater than s, say t, starting from the lower address of the private memory and create a busy block of size s and a free block of size $t - s$ (if there is an exact match, i.e. $s = t$, then a free block won't be created).

Deallocation is construed to make your task easier. If there is a request to deallocate a block of size 50, the last block of size 50 that was allocated on the stack is made free. If there is a request to deallocate a block of any other size, the first block found from the beginning of memory that has that exact size will be deallocated (you can assume that you will always find one). If the deallocated block has one (or two) neighbor(s) that is (are) free then they should be coalesced into a single free block of adequate size.

To make this concrete consider the following example.

Initialization. Assume that the whole private memory has M units (say, M = 1000). This will be a parameter of the program. There is a single free block of size 1000 that we denote (+1000) (+ is for free).
Allocate 20. The first (and only at this point) free block of size greater than 20 is of size 1000. It is split into a busy block of size 20 and a free block of size 980. We denote this as ((-20),(+980)). (- is for busy)
Allocate 70. The first block that is big enough is the first (and only) free one. It is therefore split and the memory looks like ((-20) (-70) (+910))
Allocate 50 aka "push". The memory is allocated from the "high end" and the allocation results into: ((-20) (-70) (+860) (-50)).
Allocate 50. The memory is is allocated from the "high end" and the allocation results into: ((-20) (-70) (+810) (-50) (-50)).
Allocate 40. Similar to the "Allocate 70" step above. The memory becomes ((-20) (-70) (-40) (+770)

(-50) (-50)).
Deallocate 50 aka "pull". The last block pushed on the stack, i.e., the second from the right becomes free and is coalesced with its neighboring free block yielding ((-20) (-40) (-70) (+820) (-50)).
Deallocate 40 yields ((-20) (+40) (-70) (+820) (-50)).
Allocate 80 The first free block of size 40 is not big enough. We therefore have to go to the second one yielding ((-20) (+40) (-70) (-80) (+740) (-50))
Deallocate 70 The busy block of size 70 is freed and coalesced with its free neighbor yielding ((-20) (+110) (-80) (+740) (-50))
etc

The simulation of the memory manager stops when there is an allocation request that cannot be fulfilled.

Programming part

Your assignment is **to implement a private memory manager**.

The input is a file of integers. The first integer gives the size of the primary memory (the parameter denoted M above). Positive integers between 10 and 100 denote allocations of units of that size. Negative integers denote deallocations of blocks whose size is the absolute value of the integer. In the example above, the file would be:

```
1000
20 70 50 50 40 -50 -40 -70
```

(Of course the file is incomplete since the simulation does not terminate.)

It is **strongly recommended** that you employ a doubly linked list to link the free and busy blocks together. You should have head and tail nodes. You can also use "auxiliary pointers" if you want to keep track of the first free block, or the first busy block, or the last element of size 50 that was allocated (for the stack you may want use to use a single block for the whole stack allocation if you so desire but be careful!) etc.

You should print the list of free and busy blocks using a notation similar to the one used here every 10 allocations and when the simulation terminates.

A template for your program is attached and can be found on the web also.

Instructions for turnin are forthcoming.

Paper Assignment

Questions on the program.

1. The text above states: "The simulation of the memory manager stops when there is an allocation request that cannot be fulfilled." Although, we are sure that you implemented the correct ending conditions in your program, state them here. Be sure to distinguish the cases where the requests are for the stack (request of size 50) and for any other size. In particular, can the simulation end although there is a block of the requested size that is free?

2. Give an invariant for the sum of the sizes of the free and busy blocks.

3. Give an invariant for the number of consecutive free blocks.

Questions to be answered using pseudo-code.

4. Book problem C-3.8
5. Book problem C-3.10
6. Book problem C-4.19
7. A linked list contains a *cycle* if, starting from some node p, it is true that following a certain numbers of *next* links brings us back to node p. Note that p does not have to be the first node in the list. Assume that you are given a linked list, i.e., a pointer to the first node in the list, that contains N nodes; however, the value of N is unknown.
(a) Design an O(N) algorithm to determine if the list has a cycle. You may use O(N) extra space.
(b) *Extra credit* Repeat part (a) by use only O(1) extra space (Hint: this is a little tricky. Use two pointers that start both at the beginning of the list but advance at different speeds)

Questions on Big-Oh and mathematical fundamentals

8. Book R-4.22 and C-4.10.
9. Book C-4.9

```
import java.io.*;
import java.net.*;
import java.util.*;

public class Manager {
    private static final String FILENAME = "test1.text";

    // See comments from Assignment1.
    private static Scanner scannerFromFilename(String filename) {
        URL url = MemManager.class.getClassLoader().getResource(filename);
        if (url == null)
            throw new RuntimeException("File not found: " + filename);
        try {
            Scanner s = new Scanner(url.openStream());
            return s;
        } catch (Exception e) {
            throw new RuntimeException(e + filename +
                  " couldn't be openStreamed");
        }
    }

    private static class ListNode {
        int data;
        ListNode prev;
        ListNode next;
        // add any ctors you may want
    }

    private int size;
    private Scanner scanner;
    // add variables here for list head, tail, etc.

    public MemManager(Scanner s) {
        scanner = s;
        size = s.nextInt();
    }

    // Returns a String representing  the memory blocks as specified.
    // Do System.out.println(toString()) to print.
    public String toString() {
        throw new UnsupportedOperationException();
    }

    // Reads a request from the Scanner and modifies the memory blocks
    // accordingly.  Returns true if the request succeeded.
    public boolean processOne() {
        throw new UnsupportedOperationException();
    }

    // Reads requests from the Scanner and processes them until a
```

```
    // request cannot be fulfilled.
    public void processAll() {
        throw new UnsupportedOperationException();
    }

    public static void main(String[] args) {
        Scanner s;
        s = scannerFromFilename(FILENAME);
        MemManager m = new MemManager(s);
        m.processAll();
    }

}
```