Data Structures and Algorithms

Programming Assignment #1

Due: Friday January 31

In this programming assignment, you will have to simulate a "private store allocator" (recall Lecture 2 slide 20).

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 "private memory".

The application will ask for nodes of various sizes, between 10 and 100, with requests for size 50 being much more frequent. If there is a node of size s available in private memory it will be allocated to the application. 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., nodes that have been requested by the application and have not been freed).

Allocation can take one of two forms:

- If the size is 50, it is allocated via a stack that grows from the highest past of private store downward (i.e., the first node on the stack will be between "addresses" (M - 49, M)).
- If the size is not 50, it will be allocated from the free list (see below) that essentially links the nodes that have not been yet allocated, or allocated and subsequently freed.

The allocated nodes that are not of size 50, are inserted at the tail of a busy list. Nodes are removed from the busy list in a first-in-first-out manner, i.e., the busy list is a queue.

To make this concrete consider the following example that shows the states of the data structures (free list, busy list, stack) that the "private memory manager" uses for managing allocation and deallocation in the private memory. (You are strongly advised to draw pictures.)

Initialization. Assume that the whole private memory has M units (say, M = 1000). This will be a parameter of the program. The free list is a single node of size 1000. The stack is empty. The busy list is empty.
Allocate 20. The free list is a single node of size 980. The stack is empty. The busy list is a single node of size 20.
Allocate 70. The free list is a single node of size 910. The stack is empty. The busy list consists of two nodes with a "front" pointing to a node of size 20 and a "rear" pointing to a node of size 70.
Allocate 50 aka "push". The free list is a single node of size 860 (note the size change!). The stack has one entry. The busy list is unchanged.
Free. The first element of the busy list (the one pointed to by front) is deleted from the busy list (which has one node left) and inserted at the front of the free list, which now has two nodes, the first one of size 20, and the second of size 860. The stack is unchanged.
Allocate 50 aka "push". The last node of the free list must have its size decremented by 50 (i.e., it becomes 810). The stack has now 2 entries. The busy list is unchanged. If the last node of the free list were of size less than 50, the stack would have grown too big and one should compact the free list. YOU ARE NOT ASKED TO DO THAT. Instead you should print out a message of the form

"Compaction needed due to stack overflow" and stop the simulation.
Free from the stack, i.e., "pop". The last node of the free list must have its size incremented by 50 (i.e., it becomes 860). The stack has now 1 entry. The busy list is unchanged.

etc...

Now, there is still one operation that is not precisely defined. When the allocator requests a node of size s (*Allocate s*), one might have many choices from the nodes in the free list, i.e., all nodes that have size at least s.

In the *first fit* scheme, the allocated node is the first one (starting from the front of the free list) that has size greater than or equal to s. Of course if the size is greater than s, say t, you give a node of size s to the busy list but you keep a node of size $t - s$ on the free list where the node of size s used to be. If the node is of size exactly s you give it to the busy list and delete it from the free list.

In the *best fit* scheme, the allocated node is either the first node of exactly size s or the node of size t such that $t - s$ is minimal for all nodes on the free list.

In both schemes, if there is no free node that has size at least s, you should print out a message of the form "Compaction needed due to free list overflow" and stop the simulation.

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

1. The *first fit* scheme for the free list
2. The *best fit* scheme for the free list

At the end of the simulation (as defined below), you should traverse the free list and count the number of nodes in the free list as well as their cumulative sizes.

A simulation ends either because there is a need to compact as mentioned in the two cases above or because you have reached the end of the input file that gives the commands for allocation/deallocation. Naturally, you should also check for underflow as a logical condition when deallocating and in case underflow occurs, print a message to that effect, indicating the reason for underflow (stack empty or busy list empty) and stop the simulation.

You will be given a number of input test files on which to test your programs. An input file is a sequence of integers, one per line, with the following meanings:

- The first integer, the only one that will be larger than 100, gives M, the size of the private memory
- All positive integers between 10 and 100 are requests for allocation
- The input with value 50 corresponds to a stack "push"
- The negative integer of value -1 corresponds to a stack "pop"
- The negative integer of value -2 corresponds to a deallocation (free) from the busy list
- The negative integer of value -999 indicates the end of the simulation

The example above corresponds to the input file (although they would be only one integer per line):
1000 20 70 50 -2 50 -1 -999

You should run your programs for *first fit* and *best fit* on each input file. Your output should be printouts (on the screen) of the form:

- Scheme: First Fit. Underflow stack empty.
- Scheme: Best fit. Compaction needed due to stack overflow.
- Scheme: First Fit. 16 nodes left of cumulative size 260.
- etc.

Of course, we will test your programs on files different from the very short ones you can find on the Assignment page on the Web (files testipr1.txt, with i = 1,...,5). You are encouraged to test your programs on your own input tests. Tian has written two programs, readText.cpp and readText.java (also on the Assignment page on the Web) that should help you for reading the files passed as command line arguments. In order for these programs to work, there must be **exactly** one integer per line. (Incidentally, these programs sum the values of the integers in the file.)

In addition to YOUR SUPERBLY DOCUMENTED PROGRAMS, you should include a README file indicating how to compile and run your program and whether we should be aware of something out of the ordinary.

You can use either Java or C++. You can use any source code given on the Web sites of the book (see CSE 373 "For more info" page). For additional information on java, the following link will be useful: http://java.sun.com/j2se/1.4.1/docs/api/

Instructions for Turn-in will be given later.