Linked Lists vs Arrays

Memory Management

(Estimated duration three weeks)

In this assignment we are going to implement a Memory Manager that is used by an Operating System to keep track of dynamic memory allocation (allocation from heap). This Memory Manager processes requests like “malloc” and “free” in C or “new” and “delete” in C++. In our case the equivalent to “malloc/free” will be called *claimMemory* and *freeMemory.*

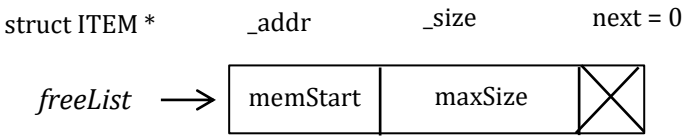
To control memory allocations, the Memory Manager maintains two linked lists: *allocList* and *freeList*. The *allocList* keeps track of allocated memory chunks, the *freeList* of all free memory chunks.

Whenever *claimMemory* is called, Memory Manager searches the *freeList* from the head of the list to find out whether necessary amount of memory is available. If free memory is found, the *freeList* will be adjusted to reflect that this memory is not available anymore and the newly allocated memory will be added to the *allocList*.

To implement both *allocList* and *freeList* you are going to use a list structure like this:

```
typedef struct ITEM
{
    struct ITEM *next=0;
    int _addr;
    int _size;
    ITEM(int addr,int size) {_addr=addr; _size=size;}
}ITEM;
```

Initially, the *freeList* contains only one block, where the beginning of the allocation memory (*memStart*) and total size of memory (*maxSize*) is stored.

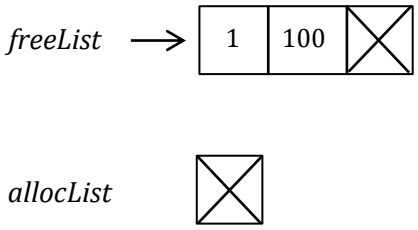


Any new call to *claimMemory* and *freeMemory* will effect both *allocList* and *freeList*, so that these two lists will reflect the used and free memory chunks.

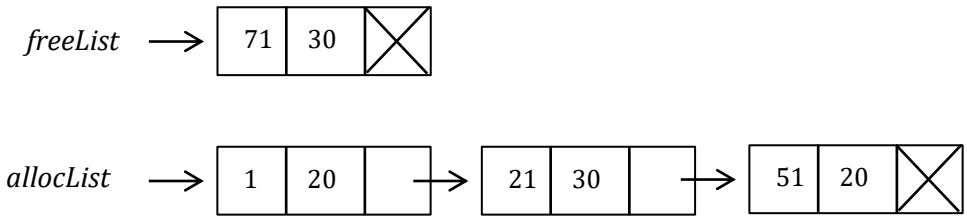
Example

Initial memory consists of 100 addresses numbered from 1 to 100. (memStart is 1 and maxSize is 100).

Then the *allocList* and *freeList* look like this:

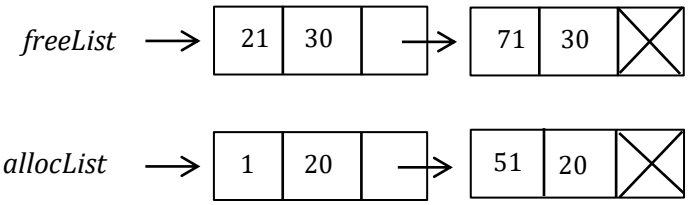


After memory of 20, 30 and 20 was claimed, the situation will be:

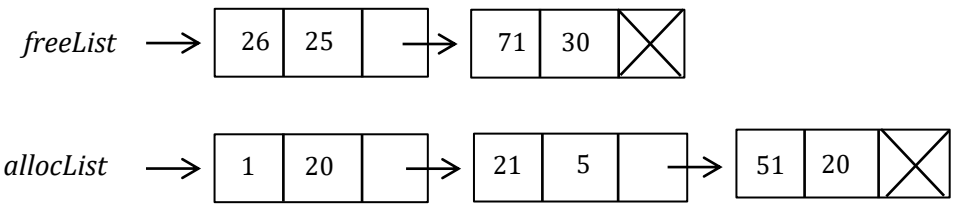


Look carefully whether this is correct.

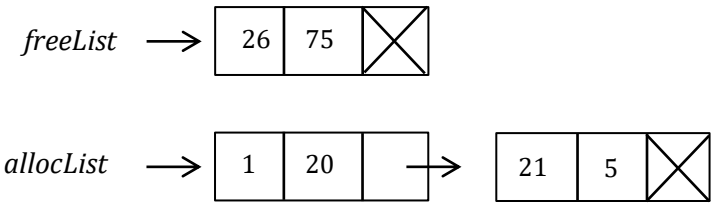
After freeing of address 21 the lists will be:



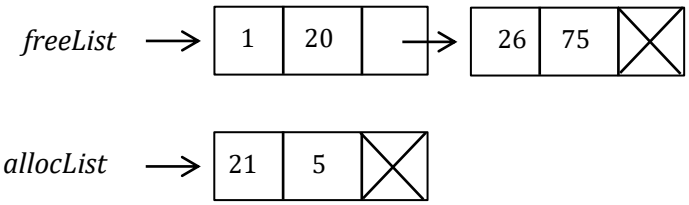
After claiming of memory of size 5 the lists will be:



After freeing of memory 51 the lists will be:



After freeing of memory 1 the lists will be:



Study the above pictures carefully by executing the same steps as the application should do. Start creating your design and implementing the application only when you understand all these pictures.

Note that it is not allowed for the *freeList* to have 2 consecutive blocks, these have to be merged.

Input

As an input you should use the project files that are made available for this course (see the *mm_C++* directory).

Delivery

1. Create a design where you explain how are you going to tackle this problem. A good part of design could be UML diagram, flow chart or method description. Think well what is appropriate to the problem described.

2. Implement the following functions:
   a. *claimMemory*
   b. *freeMemory*

3. Test the application with help of provided test program (*MemManager* executable). During testing don't forget to test ALL possible situations, so also exceptional scenario's.
   MemManager can be run in the following way:

   MemManager < *input.txt* > *out*.

   The out file from *input.txt* should be the same as *correct.txt* (you can use either Linux *diff* or *cmp* command to compare them).
   The *out* file from *input1.txt* should be the same as correct1.txt.

4. Test your final application for memory leaks by using valgrind
5. Test your final application by using your own unit tests (the basic framework is provided in the *test* directory):

   make test

6. Next to the design document, deliver zip file of the mm_C++ folder, mm_C++.zip. Please be correct on the format of the delivery and don't forget to run make clean before delivery.

TIPS:

1. Reuse your *ll* or *stack* library made in the previous assignments. It will probably have to be adjusted to a different data structure for the element of the linked list and maybe you must extend it with some APIs/methods that are necessary for your design.

2. During your design think well about object-oriented principles when structuring your code.

3. Use pen and paper to understand and analyse the problem and implementation!

4. Use of unit tests is highly recommended.

5. In case unit tests still lead to errors, debugger is a good tool to find your problem/especially crash quickly

6. Last but not least: take your time to study template code