Linked Lists vs Arrays

Linked lists: Memory Management

In this challenge 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++.

To keep track of memory allocations, the Memory Manager maintains two linked lists: *allocList* and *freeList*. Whenever a "malloc" is called, Memory Manager searches the *freeList* 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 this newly allocated memory will be added to the *allocList*.

Initially, the *freeList* contains only one element, where the start of the allocation memory (MEMORY_ADDRESS) and total size of memory (MEMORY_SIZE) are stored.

The first allocation request will decrease the size of the *freeList* element. Consequent allocation/free requests will create more elements in *freeList*, which will indicate which parts of memory are still free. *allocList* is a similar list that contains all allocated memory blocks.

You are going to implement this Memory Manager.

Part 0: your first linked list

As this is your first linked list implementation, we'll start making an initial linked list version. A proof of concept if you will.

Please view the code in the `Linked_List_POC` folder. You will find a `Makefile` and a `product` folder with some source files. Please implement all linked_list functions in `linked_list.c` and please implement `PrintAllElements` in `main.c`. If you have implemented everything correctly, you will get the following output:

```
At start of program, the list is:
  <empty>

After adding 3 elements:
  0:  addr: 100  size:   5
  1:  addr: 200  size:  10
  2:  addr: 300  size:   8

After adding 4th element just after the one on address 200:
  0:  addr: 100  size:   5
  1:  addr: 200  size:  10
  2:  addr: 250  size:  50
  3:  addr: 300  size:   8

After adding 5th element before 100:
  0:  addr:  99  size:   1
  1:  addr: 100  size:   5
  2:  addr: 200  size:  10
  3:  addr: 250  size:  50
  4:  addr: 300  size:   8

After removing tail (address 300):
  0:  addr:  99  size:   1
  1:  addr: 100  size:   5
  2:  addr: 200  size:  10
  3:  addr: 250  size:  50

After removing 2nd element (address 100):
  0:  addr:  99  size:   1
  1:  addr: 200  size:  10
  2:  addr: 250  size:  50
Temp is now: (nil). Temp should be NULL, printed as (nil)

After removing all:
  <empty>
```

Try to match this layout as closely as possible, that'll help you in part 1. It goes without saying that the Valgrind report should indicate that you have no memory leaks.

Part 1: a proper linked list module

If everything went according to plan, the linked list you wrote in part 0 works properly. As a proof of concept, it's really great. As a well designed and reusable module… not so much.

Problems.

The code we wrote has some problems. If you look at the Hardware Abstraction Layer presentation, you might find some clues. Well written code is often divided into layers: low level code at the bottom, middle level code in the middle and higher level code at the top. A golden rule in such a model is that the lower level code doesn't know what it is used for and that the higher level code does not know how the lower layer works.

Your `linked_list` module is low level code, `main.c` is the high level code.

1. Does the high level code (`main.c`) know how `linked_list` works? Yes it does: `PrintAllElements` needs to call `->next` on the head pointer.
2. Does the low level code (`linked_list`) know what it is used for? Yes it does: its `Element` struct contains the details about what `main.c` wants to store (`address` and `size`). If you want to store something different you need to change the `linked_list` module.

Another problem in this proof of concept is that you can only use one list in your application. The memory manager we want to make needs two.

We can tackle all these problems at once to create a generic linked list module that:

- Can store anything you want.
- Can be initialized multiple times (so you can use any number of lists in your application)
- Is well designed, so linked list doesn't know what it is storing or why and that the higher level code (in this case: memory manager) doesn't know how the linked list works.

Part 1.1

First things first. How do you store *any* data in your struct? There is only one solution for that, which immediately makes sure that linked list doesn't know what it's storing.

Part 1.2

How do you initialise your list multiple times? The solution lies in how OO language tackle this problem: you need to create multiple list 'objects'. If you don't know how you can do that in C, please watch this video: https://www.youtube.com/watch?v=A7Z2ZQuwNfk&list=PLULj7scb8cHgDLDHcRcJ1osiOORic86wR&index=8

Part 1.3

If you've solved the problem of storing *any* data, you get a new problem: you don't know how big the data is that your client wants to store. You need to allocate enough data to store your client's data and you need to know how many bytes you need to copy, so you must know the size. Your initialise module (read: constructor) therefor needs a size parameter and need to store it somewhere.

Part 1.4

In part 1.1 you have already solved the problem that linked list knew about the data it needed to store. Now you also need to make sure that your memory manager doesn't need to know any implementation details of linked list. The question to ask yourself is: if you redesign the linked list module (other list structure, store the data in an array, ...): does memory manager then need to change as well? If your answer is: no, well done, you are good to go! If yes: what can you change to fix this?

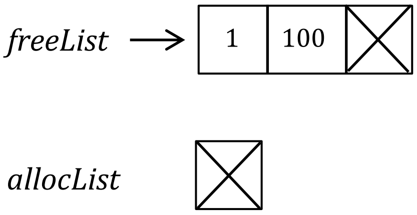
Implement your generic linked list in the `Linked_Lists_vs_Arrays` folder. You can probably reuse quite a lot of your POC code.

Part 2: memory manager

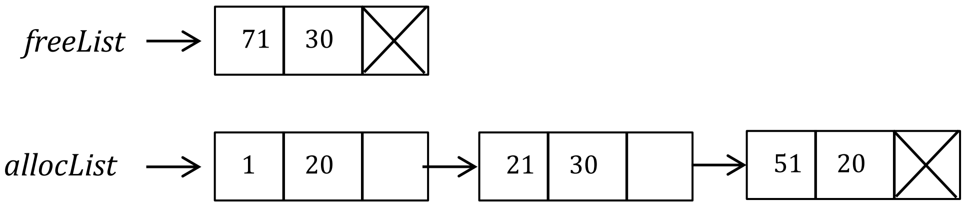
We will only implement the administration of a memory manager. The memory manager will work as follows:

Initial memory consists of 100 addresses numbered from 1 to 100. (StartAddress is 1 and the size parameter in ConstructList is 100).
Please note that your memory manager will use 1000 as StartAddress, not 1 as shown here.

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

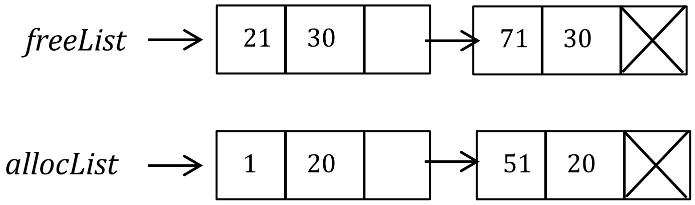


After memory of 20, 30 and 20 was claimed, the situation will be:

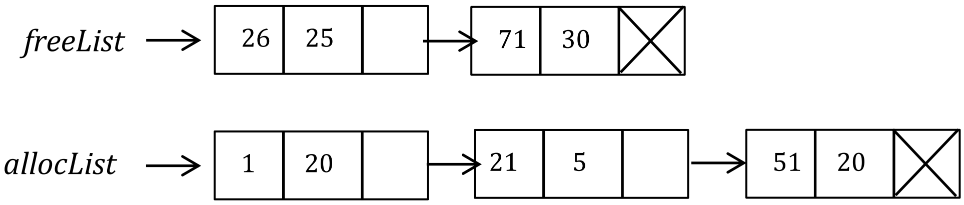


Look carefully whether this is correct.

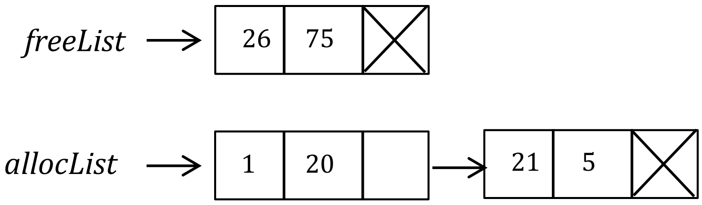
After freeing of address 21 the lists will be:



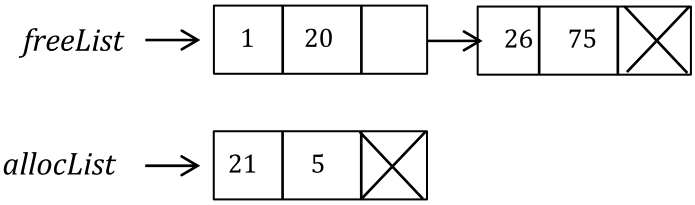
After claiming of memory of size 5 the lists will be:



After freeing of memory 51 the lists will be:



After freeing of memory 1 the lists will be:



Challenge

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

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

Testing

You can run 2 simple tests:

```
./linkedlist < testdata/input_simple.txt > out_simple.txt
```

(and then compare out_simple.txt with testdata/correct_simple.txt)

And:

```
./linkedlist < testdata/input_full.txt > out_full.txt
```

(and then compare out_full.txt with testdata/correct_full.txt)

Demonstrate

1. Implement all memory manager functions

2. Properly test your application, during testing don't forget to test ALL possible situations, so also exceptional scenario’s.

3. Demonstrate your final application for (memory) bugs by using klocwork and valgrind.