Systems Programming: Practical 6
Pointers and Dynamic Memory Allocation - Part II

A Singly-linked list

Implement a program that stores numbers in a linked list and then prints them out. The nodes of your linked list should be an appropriate `struct` type and should be dynamically allocated using `malloc()`. The program should let the user input numbers at runtime and should contain a function that adds nodes to the end of the list. (Note that pressing `Ctrl+D` makes `scanf()` return `EOF`.)

B More advanced linked list

Add functions to:

- delete the last number in the list
- add a number to the start of the list
- search for a number in the list and return either a pointer to it or `NULL` if the number is not in the list

C Doubly-linked list

Change your code to use a doubly-linked list. Add a function that takes a pointer to a node in the list and deletes the corresponding node from the list. Remember to `free()` the memory used by the node!

D Optional: Implement `calloc()` and `realloc()`.

Write functions `calloc2()` and `realloc2()`, that use `malloc()` and `free()` to implement the functionality of `calloc()` and `realloc()`, respectively. These functions will have the following declarations:

```
void *calloc2(size_t nmemb, size_t size);
void *realloc2(void *ptr, size_t old_size, size_t new_size);
```

Remember that `calloc()` sets the allocated memory to zero (for this exercise, you may ignore testing for integer overflows when multiplying the arguments of `calloc()` together). When implementing the copying part of `realloc()`, recall that `char` is 1 byte; the C standard states that you may use `char *` pointers to access individual bytes of memory.

For `realloc2()`, since you don't know how large an area of memory `*ptr` points to, you will need to provide the function with both the old size as well as the new size you are requesting. The real `realloc()` normally only needs the latter, as it has access to internal data structures used by `malloc()`, etc. to keep track of the former.