Sorted linked list

Submission deadline:	**2022-12-26 11:59:59**	1522605.303 sec
Late submission with malus:	**2022-12-31 23:59:59** (Late submission malus: 100.0000 %)	
Evaluation:	**6.3360**	
Max. assessment:	**4.0000** (Without bonus points)	
Submissions:	1 / 20 Free retries + 10 Penalized retries (-10 % penalty each retry)	
Advices:	0 / 2 Advices for free + 2 Advices with a penalty (-10 % penalty each advice)	

The task is to implement three functions that ease handling and sorting of linked lists.

The functions need to fit into the following declarations:

`TITEM`
: This structure represents an item of a singly-linked list. The declaration is fixed in the testing environment, your implementation must use the existing declaration (it cannot modify it in any way). The structure contains the following fields:

  - `m_Next` a pointer to the next element in the linked list or `NULL` for the last element in the list,
  - `m_Name` a string with the name of the element,
  - `m_Secret` some secret data associated with the item. With the exception of function `newItem`, your implementation does not need to access/modify the data.

```
TITEM * newItem ( const char * name, TITEM * next )
```
: This function is a helper function that eases the creation of a linked list. The function dynamically allocates the new item, it initializes the fields, and finally, it returns the pointer to that newly created item. Fields `m_Next` and `m_Name` must be initialized from the corresponding parameters, field `m_Secret` is to be filled with zero bytes ('\0'). The implementation is your task.

```
void freeList ( TITEM * l )
```
: The function deallocates the memory used by the linked list. The parameter is a pointer to the first element in the list. The implementation is your task.

```
TITEM * sortList ( TITEM * l, int ascending )
```

: The function is used to sort the elements in the given linked list. The parameter is a pointer to the first element of the list to sort (`l`) and the requested sort order (`ascending`). The function re-arranges the elements in the list such that the elements follow the requested sort order. The function **must not** free the elements in the original list (and return their copies). Instead, the function **must** use the existing elements, it just needs to modify the links. Return value is the pointer to the first element of the newly re-arranged list.

: The sorting uses element name (`m_Name`) as the sort key. The strings are compared in the case sensitive way. The sort order is either ascending (parameter `ascending` is not zero), or the sort order is descending (parameter `ascending` is zero). The function must perform **stable** sorting.

Submit a source file with the implementation of the required functions. Further, the source file must include your auxiliary functions which are called from `sortList` / `newItem` / `freeList`. The function will be called from the testing environment, thus, it is important to adhere to the required interface. Use the attached source code as a basis for your development. There is an example `main` with some tests in the attached source. These values will be used in the basic test. Please note the header files as well as `main` is nested in a conditional compile block (`#ifdef/#endif`). Please keep these conditional compile block in place. They are present to simplify the development. When compiling on your computer, the headers and `main` will be present as usual. On the other hand, the headers and `main` will "disappear" when compiled by Progtest. Thus, your testing `main` will not interfere with the testing environment's `main`.

Advice:

- Copy the sample code from the archive and use it as a base for your development.

- The main in your program may be modified (e.g. a new test may be included). The conditional compile block must remain, however.
- The testing environment disables some functions. In particular, library functions `qsort` and `qsort_r` are not available (they are not designated for linked lists anyway).
- The input linked lists are quite short in the mandatory tests. Sorting efficiency is not very important.
- The first bonus test checks the time efficiency of the sorting algorithm. It inputs very long linked lists, thus quadratic algorithms exceed the time limit.
- The first bonus test checks the time and memory efficiency of the sorting algorithm. Similarly to the first bonus, it inputs very long linked lists. Moreover, the available memory is limited, only a few hundred KiB is available.
- The attached source lists the header files available in the testing environment. Other header files are not present, moreover, you cannot include them by means of `#include` directives in your submitted code.

Sample data:		**Download**
Submit:	Choose File	**Submit**

☐ **Reference**

1	**2022-12-08 20:59:05**	**Download**
Submission status:	Evaluated	
Evaluation:	6.3360	

- **Evaluator: computer**
  - Program compiled
  - Test 'Basic test with sample input data': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.000 s (limit: 2.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Borderline test': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 0.003 s (limit: 2.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Random test': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 0.003 s (limit: 1.997 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Random test + mem debugegr': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 0.008 s (limit: 2.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Bonus - speed': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 1.148 s (limit: 5.000 s)
    - Bonus test - success, evaluation: 120.00 %
  - Test 'Bonus - speed + memory': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 2.509 s (limit: 5.000 s)
    - Bonus test - success, evaluation: 120.00 %
  - Overall ratio: 144.00 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.20 * 1.20)
- Total percent: 144.00 %
- Early submission bonus: 0.40
- Total points: 1.44 * (4.00 + 0.40) = 6.34

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**7**	--	--	--
Lines of code:	**192**	**27.43 ± 31.34**	**99**	`main`

Cyclomatic complexity: **72** **10.29 ± 16.18** **49** `main`