Exercise 4 — Quicksort and Hash Tables

Andreas Nüchter, Fabian Arzberger, Jasper Zevering

Winter 2025/2026

Goals: In this assignment you will write a sorting program similar to the program from lab 3, except that it uses linked lists instead of arrays to hold the list of numbers read in off the command line. In the process, you will learn about one of the fundamental constructs in the C language (structs) and more about pointers.

Prerequisites:

- structs

  Structs provide a way to bundle together several related pieces of data into a single piece of data. Structs are commonly used to define new data types.

- typedefs

  Typedefs are a way to give an alias (alternative name) for a type name. Used correctly, they can make code both more concise and more readable.

- linked lists

  Linked lists are a fundamental data type that is constructed using structs and pointers. Linked lists are very common in real code, so it is important that you become comfortable with using them.

- Hash tables

Programs to write:

1. **Quciksorter**

   The program to write for this assignment is a variation on assignment 2. Recall that in that program, you input a list of numbers and the program printed them out in sorted order. This time you will be doing the same thing, with these differences:

   (a) You will use a linked list to store the numbers entered on the command line. This will have the advantage that you can enter an arbitrary number of numbers (i.e. you're not limited to just 32 numbers).

   (b) You will use the "quicksort" algorithm to sort the numbers instead of the minimal insertion sort or bubble sort. Quicksort is much more efficient than those algorithms, and is described below.

   Your program will be called `quicksorter`. Here is the specification:

   (a) The program will accept an arbitrary number of numbers on the command line.

   (b) If there are no command-line arguments, or if there are arguments but none that represent numbers to be sorted, the program should print out instructions on its use (i.e. a usage message). By the way, don't assume that the test script checks this; currently it doesn't, so check it yourself (we will).

(c) If any of the arguments to the program are not integers, your program's response is undefined – it can do anything. (i.e. you shouldn't worry about handling non-integer arguments). The only exception to this is the optional "-q" argument to suppress output (see below).

(d) The program will sort the numbers using the quicksort algorithm. The numbers on the command line will be stored in a linked list. You should have a separate `quicksort` function to do the sorting. This function should receive exactly one argument (the list to be sorted) and return a pointer to a newly-allocated list which contains the same integers as the input list, but in sorted (ascending) order. Do not alter the original list in any way. Also, do not pass the length of the list to the `quicksort` function (it isn't necessary anyway).

(e) As in assignment 2, use assert to check that the list is sorted after doing the sort. Use the `is_sorted()` function in `linked_list.c` to check the list. Put the assert check at the end of the `quicksort` function, right before it returns the sorted list.

(f) Print out the numbers from smallest to largest, one per line.

(g) Use the memory leak checker as in the previous assignment to check that there are no memory leaks.

The Sample Output

```
% quicksorter 5 9 -2 150 -95 23 2 5 80
-95
-2
2
5
5
9
23
80
150
% quicksorter
usage: quicksorter number1 [number2 ... ]
```

Command-line options

Add a "-q" command-line argument that suppresses output like you did in assignment 2. Make sure that even when this is in effect, you still sort the numbers. Also, make sure in all cases that you use the `is_sorted()` function in `linked_list.c` (see below) to test whether the sorted list is in fact sorted. Your program should accept single or multiple "-q" arguments on the command line (just like assignment 2), where multiple "-q" arguments are the same as a single one, and if there are no numbers on the command line (e.g. just "-q" arguments) the program should exit with a usage message.

The quicksort algorithm

Quicksort is a sorting algorithm invented by the computer scientist C. A. R. Hoare. It is much more efficient than bubble sort or insertion sort (technically, it has an $O(n \log n)$ average time complexity as compared to $O(n^2)$ for the other two algorithms). Quicksort is usually used to sort an array in place (there is even a C library function called `qsort` which does this), but in this case we are sorting a linked list and we are not modifying the original list, which changes the details of the algorithm somewhat. Quicksort is a recursive algorithm, like the algorithm you used to solve the peg game in the second part of assignment 2.

Here is the algorithm:

(a) If the list has zero nodes (i.e. it's empty) or one nodes, copy the list as-is and return it (it's already sorted). Note that this algorithm works fine on empty lists.

(b) Otherwise, create a copy of the first node of the list and "put it aside" for now.

(c) Divide the rest of the list into two new lists:

   i. The nodes with values larger than or equal to the value of the first node;

   ii. the nodes with value smaller than the value of the first node.

   Remember, you have to make these lists yourself and they have to copy the values in the original nodes.

(d) Sort these two new lists using a recursive call to the quicksort routine.

(e) Append everything together in order:

   i. The list of nodes with values smaller than the first node,

   ii. the copy of the first node,

   iii. and the list of nodes with values larger than or equal to the first node.

Memory management

Remember, when you use `malloc` or `calloc` to allocate memory, that memory must be explicitly freed (using the `free` function) before the end of the program or you have a *memory leak*. It can be tricky avoiding memory leaks, but here is a specific suggestion: don't ever do this:

```
node *list;
/* ... some code which assigns values to 'list' ... */
/* Sort the list and set the list pointer to point to the sorted list. */
list = sort_list(list);
```

In this case, `sort_list()` returns a new (sorted) `list`, and the list pointer list is set to the head of that list. This is a memory leak, because the old list that `list` pointed to was never freed. The right way to do this is as follows:

```
node *list;
node *sorted_list;
/* ... some code which assigns values to 'list' ... */
/* Sort the list and set the list pointer to point to the sorted list. */
sorted_list = sort_list(list);
free(list);
list = sorted_list;
```

The result is the same, but there's no memory leak.

Again, use the memory leak checker as you did last lab to make sure that you haven't inadvertently leaked any memory. Recall that you do this by putting these lines at the top of the file `quicksorter.c`:

```
#include <stdlib.h>
#include "memcheck.h"
```

and then putting this line in the main function before exiting:

```
print_memory_leaks();
```

In addition, if you can't track down a memory leak you might try changing the following line in memcheck.c from

```
#define DEBUG 0
```

to

```
#define DEBUG 1
```

and recompiling. This will give more verbose output relating to memory allocation.

Code supplied

There are a number of useful utility functions supplied in `linked_list.c` and declared in `linked_list.h`. Please use these instead of writing your own versions; it will save you a lot of time. Furthermore, we provide a Makefile and an updated test script.

2. **Hash table**

Your program will use a hash table to count the number of occurrences of words in a text file. The basic algorithm goes as follows for each word in the file:

(a) Look up the word in the hash table.

(b) If it isn't there, add it to the hash table with a value of 1. In this case you'll have to create a new node and link it to the hash table as described above.

(c) If it is there, add 1 to its value. In this case you don't create a new node.

At the end of the program, you should output all the words in the file, one per line, with the count next to the word e.g.

```
cat 2
hat 4
green 12
eggs 3
ham 5
algorithmic 14
deoxyribonucleic 3
dodecahedron 400
```

Also, make sure that you explicitly free all memory that you've allocated during the run of the program before your program exits. This includes:

(a) all the nodes in the linked lists that the hash table array points to,

(b) the hash table array,

(c) the hash table struct itself,

(d) the string keys used in the linked list nodes (allocated in the main() function (see below)).

The memory leak checker will help ensure that you get this right ;-)

The hash function

The hash function you will use is extremely simple: it will go through the string a character at a time, convert each character to an integer, add up the integer values, and return the sum modulo 128 (but don't use a magic number for this). This will give an integer in the range of [0, 127]. This is a poor hash function; for one thing, anagrams will always hash to the same value. However, it's very simple to implement. Note that a value of type `char` in a C program can also be treated as if it were an `int`, because internally it's a one-byte integer represented by the ASCII character encoding. If you like, you can convert the `char` to an `int` explicitly using a type cast. However, a string of characters is not an `int`, and you can't use `atoi` to convert it into one (mainly because most of the keys will be words, not string representations of numbers). You also shouldn't try to type cast the string to an `int` (it'll just cast the address into an integer, which might work but it's not what you want). A string is an array of `char`s, which is not the same as a single `char`; keep that in mind. So you'll need a loop to go through the string character-by-character.

Other details

Since a value associated with a key will always be positive, if you search for a key and don't find it in a hash table, just return 0. That's the special "not found" value for this hash table.

For simplicity, the program assumes that the input file has one word per line (we've written this code for you). The order of the words you output isn't important, since the test script will sort them.

Things to watch out for

You should design your hash table functions to be generic i.e. independent of the purpose those functions are used for. Specifically, don't assume that particular values to be assigned mean anything special (for instance, don't assume that assigning a value of 1 means that a key doesn't exist in the table, just because that's how it will be used in the rest of the program). You should think of the hash table functions as being part of a hash table library that you are writing. Functions in a library could be used in many different programs, so the fewer the assumptions you make about how the functions will be used, the better. The only exception to this is the rule that says if you are searching for a key and it isn't found, you should return zero. That's not very generic (you can imagine a hash table in which it would be OK to store a value of zero), but it makes the program simpler.

There is one memory leak which may be hard to get rid of which involves the `set_value` function. You should assume that the key value that is passed to `set_value` is newly-allocated, and so it either has to go into the hash table or it has to be freed.

Supporting files

To simplify your task, we're supplying you with: a header file, a template file for your code (this file will be called `hash_table.c`), a main file, a Makefile, a test input file, a test output file, a test script, and of course, the memory leak checker: `memcheck.c` and `memcheck.h`.

Submission: The source code of the `quicksorter` and the hash table program.

Please send the programs to `fabian.arzberger@uni-wuerzburg.de` and `jasper.zevering@uni-wuerzburg.de` until November 12, 2025 23.59. Include `[CS4SE]` in your subject line and state the names of all team members in the message body. Please work in groups of two students. One submission per team is sufficient. Late homework won't be accepted!