Computer Systems Organization
CSCI-UA.0201 Spring 2024
Programming Assignment 1

Due Tuesday, February 20 at 11:55pm

This programming assignment is to be done entirely in C. It will give you practice using pointers and structs in C, in this case to build a hash table and a binary search tree.

As described below, you will be submitting three files, hashing.c, tree.c, and tree.h. To do so, just upload the three files to the "Programming Assignment 1" page under the Assignments tab in Brightspace.

Important: Do not submit the files until the program is fully working. Although there is a modest late penalty, **you will lose far more points if you submit a program that doesn't work**.

This assignment looks long (5 pages), but that is only because I have given very detailed instructions. That being said, please start working on the assignment immediately! If you have general questions about this assignment, please post them on the discussions board on Brightspace, in the "Programming Assignment 1 Questions" topic, and I will answer quickly. If you have questions about your code, please email the course assistant. **Do not post your code to the discusions board**.

You can discuss the assignment with your fellow students but you must write your own code. I will be asking you to write similar kinds of C code on the midterm exam and, if you don't do your own work on this assignment, you will do very poorly on the midterm exam.

You should complete the assignment by performing the following steps.

Step 1

In a file hashing.c, do the following:

- Declare a struct type **`HASHCELL`**, which is used to contain data in a hash table. A **`HASHCELL`** should contain two fields: a **`word`** field that is a string – it should be of type **`char *`**, so it's a pointer, not an allocated array – and a **`next`** field that is a pointer to a **`HASHCELL`**, so that **`HASHCELL`**s can be linked together in a list.
- Declare a global variable **`hashtable`**, representing a hash table, which is an array of **`HASHCELL`** pointers. You should use **`#define`** to define a constant **`SIZE`** to be 100 and declare **`hashtable`** to have **`SIZE`** elements.
- Define a function **`hash_string`** which takes a string (a **`char *`**) as a parameter and returns an unsigned integer between 0 and **`SIZE -1`**, by hashing the string. You are free to choose your own hash algorithm, but here is a simple one you can use:
    - Define an unsigned integer variable **`hash`** and initialize it to 1.
    - In a loop that iterates over each character in the string, set **`hash`** equal to (**`hash`** * 7) + **`c`** in each iteration of the loop, where **`c`** is the current character. Since a **`char`** is just an 8-bit number, so it's fine to do arithmetic on it.

- Return the value of **hash** mod **SIZE** (where **%** is the mod operator).

Remember that you can recognize the end of a string by the terminating 0.

The quality of the hash (*i.e.,* how evenly it distributes hash values between 0 and **SIZE** - 1) isn't so important here.

- Define a function **insert_hash_cell** that takes a string (again, a **char ***) as a parameter and inserts the string into the hash table as follows:
  - It calls **hash_string** on the string, assigning the result of the hash to an unsigned integer variable **index**.
  - It creates a **HASHCELL** (using malloc), such that the **word** field of the cell points to the string. IMPORTANT: You will need to make a copy of the string by performing the following steps:
    - Using **malloc** again, allocate a block of memory large enough to hold the string (including the 0 at the end). I suggest using the built-in **strlen** function, described at the bottom of this assignment, which gives you the length of a string without the 0 at the end (so you'll need to add 1).
    - Set the **word** field of the cell to point to the new block of memory.
    - Copy the characters of the string into the new block. I suggest using the **strcpy** function, described at the bottom of this assignment.
  - It inserts the new cell into the linked list of cells pointed to by **hashtable[index]**. However, if the word in the new cell already exists in that linked list, do not insert the new cell. This prevents duplicate words from being inserted into the hash table. I suggest using the built-in **strcmp** function, described below, to compare two strings to see if they are the same.
- Define a function **print_hash_table()** that prints out the elements of the hash table. Specifically, in a loop, for each **i** from 0 to **SIZE**-1, the function should print out **i**, then ":", then (on the same line) all the words in the linked list at **hashtable[i]**, and then a carriage return. This way, the list of words at each element of the hash table is printed on their own line.

Step 2

Still within hashing.c, define a **main** function that does the following:

- It declares a variable **str** that is an array of 100 chars (which should be large enough to hold any string that is read in from the terminal).
- In a loop, it uses **scanf** to repeatedly read a string from the terminal into **str**, and then calls **insert_hash_cell** on **str** to insert the string into the hash table (which is why **insert_hash_cell** has to make a copy of the string). The loop should stop when there are no more strings to read, which **scanf** will indicate by returning the value **EOF**. That is, your loop could look like,

```
while (scanf(…) != EOF) {…}".
```

- It calls **`print_hash_table()`** to print the contents of the hashtable.

Step 3

Compile and debug hashing.c. I have provided on Brightspace a sample input file containing a large number of words. The file is called dickens.txt, because it contains the first chapter of Charles Dickens's *A Tale of Two Cities*, from which I removed all punctuation and made all letters lower case. If you compile your program in a shell by doing

```
gcc -o hashing hashing.c
```

then to run your program and have it read the input from dickens.txt, simply type

```
./hashing < dickens.txt
```

The "<" tells the operating system to take the contents of dickens.txt and send it to the hashing program as if you were typing all those words from the terminal. Also, if you want to send the output of the program to a file, you can run the program by

```
./hashing < dickens.txt > results.txt
```

The ">" tells the operating system to send the output of the hashing program to results.txt instead of the terminal. You should not use any file I/O functions in your C program.

If your hash table is working correctly, you should not see any duplicate words in the output.

Step 5

In a file tree.h, define a **`NODE`** structure type to be used for nodes in a binary search tree, such that a **`NODE`** has the following fields: a **`word`** field that is a string (again, **`char *)`**, a **`left`** field that is a **`NODE`** pointer, and a **`right`** field that is also a **`NODE`** pointer. These represent pointers to a node's left and right children, of course.

Step 6

In a file tree.c, write the following code to create and print a binary search tree:

- Declare a global variable **`root`** that is a **`NODE`** pointer.
- Define a recursive function **`rec_insert_node`** that takes two parameters: a **`NODE`** pointer **`n`**, representing a node to be inserted into a tree, and a **`NODE`** pointer **`r`**, pointing to a tree into which **`n`** should be inserted. You can assume that **`r`** is not NULL, so inserting **`n`** into the tree that **`r`** points to is easy, as follows:
    - if the value of **`n`**'s **`word`** field is less than the value of **`r`**'s **`word`** field, then:
        - if the **`left`** field of **`r`** is NULL, set the **`left`** field of **`r`** to point to **`n`**,
        - otherwise, call **`rec_insert_node`** recursively to insert **`n`** into the tree pointed to by **`r`**'s **`left`** field.

- Otherwise, *i.e.*, if the value of **n**'s **word** field is not less than the value of **r**'s **word** field, then:
  - if the **right** field of **r** is NULL, set the **right** field of **r** to point to **n**,
  - otherwise, call **rec_insert_node** recursively to insert **n** into the tree pointed to the **right** field of **r**.

Important: Since the **word** field of a **NODE** is a string, the above string comparisons should again use the **strcmp** function, described below. You cannot use the "<", "==", or ">" operators.

- Define a function **insert_node** which takes a string (again, a **char ***) as a parameter. It should create a new **NODE** whose **word** field points to the string (no copying needed) and – if **root** is not NULL – call **rec_insert_node** to insert the node into the tree pointed to by **root**. If **root** is NULL, though, just set **root** to point to the new node.
- Define a recursive function **print_tree** that takes a **NODE** pointer **r** as a parameter and prints the **word** field in each node of the tree pointed to by **r**. The nodes should be visited so that the strings are printed in sorted order. To have **print_tree** print the tree pointed to **r**, it should simply:
  - check if **r** is NULL. If so, return. Otherwise:
    - call **print_tree** recursively to print **r**'s **left** subtree,
    - print the **word** field of **r**, using **printf**,
    - call **print_tree** recursively to print **r**'s **right** subtree.

Step 7

Do the following:

- Modify tree.h so that the **root** variable, the **insert_node** function, and the **print_tree** function can be used outside of tree.c.
- Modify the hashing.c file so that it can access the items declared in tree.h.
- Modify the **main** function in hashing.c in the following ways:
  - Comment out the call to **print_hash_table()**, since it is not needed anymore (it was only for debugging). Don't delete the call, since you might need to go back and do more debugging.
  - Write a loop that iterates over the elements of **hashtable** (where each element is a linked list), with a nested loop that iterates over each cell in the linked list and calls **insert_node** on the string in the cell. In this way, **insert_node** is called on every string that was originally inserted into the hash table, so that now all of those strings are in the binary search tree that **root** points to.
  - Call **print_tree**, passing **root**, so that the strings in the entire binary search tree are printed in sorted order.

Step 8

Compile your program in a shell by typing

```
gcc -o hashing hashing.c tree.c
```

and run your program in exactly the way specified by step 3 (using the dickens.txt file). The output should show the words from dickens.txt, but with duplicates removed and in alphabetical order.

I have provided on Brightspace compiled versions of my solution to the assignment. The executable files are called ben_macOS, ben_linux, and ben_cygwin.exe, to run on MacOS, Linux, or Windows/Cygwin, respectively (Note: On Windows, download ben_cygwin.zip and unzip it – Brightspace won't let me post a .exe file). This way, you can run to see if your program is generating the same output as mine. Run my code by typing "**./ben_***version* **< dickens.txt**", where *version* is either **macOS**, **linux**, or **cygwin.exe**.

Step 9

When everything is working, upload hashing.c, tree.c, and tree.h to the course website.

Built-in String Functions in C

C provides numerous built-in functions for operating on strings. To use these functions, put

```
#include <string.h>
```

at the top of any file in which the string functions are used. The three functions that you might find useful for this assignment are:

- **strcpy(char *s1, char *s2)**: This copies the contents of **s2** into **s1**, including the terminating 0.
- **strlen(char *s)**: This returns the length of the string **s**, not including the terminating 0. For example, **strlen("hello")** returns 5.
- **strcmp(char *s1, char *s2)**: This compares **s1** and **s2** using alphabetical order (like in a dictionary). The return value indicates one of three possibilities:
  - If **s1** is less than **s2** (*i.e.,* if **s1** comes before **s2** in a dictionary), then **strcmp** returns a negative number,
  - if **s1** is equal to **s2** (i.e. the strings are identical), **strcmp** returns 0, and
  - if **s2** is greater than **s1**, **strcmp** returns a positive number.

  For example, **strcmp("hello", "goodbye")** would return a positive number, since "hello" comes after "goodbye" in the dictionary. You cannot use "<", "==", or ">" to compare strings, since those would just compare the addresses in memory where the strings reside.

If you need more information about these functions, just google them.