CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #3

Spring 2019

Assignment Due: Friday, April 12, 2019 by 11:59 pm

Updates to the Document:

- 4/6/2019: The return value of the example given in Part 8 is 11, not 9.
- 4/9/2019: Clarified that the only change to memory that should be made by `load_hash_table` when the file is not found is to clear the hash table.

Learning Outcomes

After completion of this programming project you should be able to:

- Read and write strings of arbitrary length.
- Implement non-trivial algorithms that require conditional execution and iteration.
- Design and code functions that implement the MIPS assembly register conventions.
- Implement algorithms that process structs.
- Read files from disk and build data structures based on their contents.

Getting Started

Visit Piazza and download the file `proj3.zip`. Decompress the file and then open `proj3.zip`. Fill in the following information at the top of `proj3.asm`:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)

Having this information at the top of the file helps us locate your work. If you forget to include this information but don't remember until after the deadline has passed, don't worry about it – we will track down your submission.

Inside `proj3.asm` you will find several function stubs that consist simply of `jr $ra` instructions. Your job in this assignment is implement all the functions as specified below. Do not change the function names, as the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, but they must be saved in `proj3.asm`. Helper functions will not be graded.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS assembly code.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

Finally, do not define a `.data` section in your `proj3.asm` file. A submission that contains a `.data` section will probably receive a score of zero.

Important Information about CSE 220 Homework Assignments

- Read the entire homework documents twice before starting. Questions posted on Piazza whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- **You must use the Stony Brook version of MARS posted on Piazza.** Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you will need to complete the homework assignments.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. It is much easier for your TAs and the professor to help you if we can quickly figure out what your code does.
- You personally must implement homework assignments in MIPS Assembly language by yourself. You may not write or use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS Assembly code you submit as part of the assignments.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- Do not submit a file with the function/label `main` defined. You are also not permitted to start your label names with two underscores (__). You will obtain a zero for an assignment if you do this.
- Submit your final `.asm` file to Blackboard by the due date and time. Late work will not be accepted or graded. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

With minor exceptions, all aspects of your homework submissions will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this homework assignment you will be writing *functions* in assembly language. The functions will be tested independently of each other. This is very important to note, as you must take care that no function you write ever has side-effects or requires that other functions be called before the function in question is called. Both of these are generally considered bad practice in programming.

Some other items you should be aware of:

- All test cases must execute in 250,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed. To find the instruction count of your code in Mars, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

- Any excess output from your program (debugging notes, etc.) might impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

You must follow the register conventions taught in lecture and reviewed in recitation. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any `$s` registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save `$ra` before calling the callee. In addition, if the caller wants a particular `$a`, `$t` or `$v` register's value to be preserved across the secondary function call, the best practice would be to place a copy of that register in an `$s` register before making the function call.
- A function which allocates stack space by adjusting `$sp` must restore `$sp` to its original value before returning.
- Registers `$fp` and `$gp` are treated as preserved registers for the purposes of this course. If a function modifies one or both, the function must restore them before returning to the caller. There really is no reason for your code to touch the `$gp` register, so leave it alone.

The following practices will result in loss of credit:

- "Brute-force" saving of all `$s` registers in a function or otherwise saving `$s` registers that are not overwritten by a function.
- Callee-saving of `$a`, `$t` or `$v` registers as a means of "helping" the caller.
- "Hiding" values in the `$k`, `$f` and `$at` registers or storing values in main memory by way of offsets to `$gp`. This is basically cheating or at best a form of laziness, so don't do it. We might comment out any such code we find.

How to Test Your Functions

To test your implementated functions, open the provided `main` files in MARS. Next, assemble the `main` file and run it. MARS will include the contents of any `.asm` files referenced with the `.include` directive(s) at the end of the file and then add the contents of your `proj3.asm` file before assembling the program.

Each main file calls a single function with one of the sample test cases and prints any return value(s). You will need to change the arguments passed to the functions to test your functions with the other cases. To test each of your functions thoroughly, create your own test cases in those main files. Your submission will not be graded using the examples provided in this document or using the provided main file(s).

Again, any modifications to the `main` files will not be graded. You will submit only your `proj3.asm` for grading. Make sure that all code required for implementing your functions is included in the `proj3.asm` file. To make sure that your code is self-contained, try assembling your `proj3.asm` file by itself in MARS. If you get any errors (such as a missing label), this means that you need to refactor (reorganize) your code, possibly by moving labels you inadvertently defined in a `main` file (e.g., a helper function) to `proj3.asm`.

A Reminder on How Your Work Will be Graded

It is **imperative** (crucial, essential, necessary, critically important) that you implement the functions below exactly as specified. Do not deviate from the specifications, even if you think you are implementing the program in a better way. Modify the contents of memory only as described in the function specifications!

Autocorrection

The overall theme of this project is to create a program (a function, really) that will perform autocorrection of input strings. The autocorrecting function will rely on a hash table that maps words to their replacements, i.e., which maps strings to strings. To these ends you will implement several string-processing functions and hash table functions. In particular, you will develop functions to implement a hash table data structure that uses open address hashing with linear probing.

Preliminaries

In this assignment, you will familiarize yourself with a feature of the C programming language known as the **struct**. A struct is a composite data type that is used to group variables under one name in a contiguous block of memory. You will be reading and writing structs that implement hash tables which map strings to strings. More specifically, each entry in a hash table will map the starting address of a string to the starting address of another string.

```
struct HashTable {
    int capacity;    // 4 bytes: the number of elements in the keys[] and
                     // values arrays (see below)
    int size;        // 4 bytes: the number of occupied elements in the keys[]
                     // and values arrays (see below)
    string[] keys;   // 4 bytes per element; "capacity" total elements
    string[] values; // 4 bytes per element; "capacity" total elements
}
```

We can see that the total memory consumed by the hash table is given by the unsimplified formula:

```
    4 + 4 + 4*capacity + 4*capacity
```

Below is a visualization of an example hash table struct:

```
Capacity: 7
Size:     5
Slots:
   0: 101 -> CSE101
   1: ams -> Applied Mathematics
```

```
2: 0x00000000 -> 0x00000000
3: 0x00000001 -> 0x00000000
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

In memory, the strings shown here would not be stored in the hash table itself, but rather in a separate part of the static (`.data`) section. For example, consider slot #1:

```
1: ams -> Applied Mathematics
```

Imagine for a moment that the string `"ams"` were stored at memory address `0x010019ABC` and `"Applied Mathematics"` were stored at memory address `0x0100100FF`. A more realistic visualization of slot #1 in the hash table would be:

```
1: 0x010019ABC -> 0x0100100FF
```

Looking closely at the hash table we find that there is something special about slots #2 and 3:

```
2: 0x00000000 -> 0x00000000
3: 0x00000001 -> 0x00000000
```

A value of zero for a key (e.g., slot #2) indicates that the slot has never been used. We will refer to this as an “empty” slot. A value of 1 for a key (e.g., slot #3) indicates that the slot was used in the past, but that the (key,value) pair was later deleted. We will refer to this as an “available” slot. If you are not clear on the important distinction in hash tables between “empty” and “available” slots, then you should review how hash tables are implemented using linear probing.

String Functions

Before implementing the autocorrection function, we will need to write a few functions for working with strings and then several functions for manipulating hash tables.

Part I: String Comparison

`int strcmp(string str1, string str2)`

This function takes two null-terminated strings (either or both of which could be empty) and returns an integer that indicates the lexicographic order of the strings. The function begins by comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached. Note that the strings can be of different lengths.

The `strcmp` function will be used for searching through hash tables.

The function takes the following arguments, in this order:

- `str1`: the starting address of the first string
- `str2`: the starting address of the second string

Returns in `$v0`:

- the difference between the ASCII values of the first mismatch, if any: `str1[n] - str2[n]`, where `n` is the index of the first mismatch
- `0`: the contents of both strings are identical (including the case where they are both empty strings)
- length of `str1`: `str2` is an empty string but `str1` is non-empty
- negated length of `s2`: `str1` is an empty string but `str2` is non-empty

Additional requirements:

- The function must not write any changes to main memory.

Examples:

Function Arguments	**Return Value**	**Explanation**
`"ABCD", "ABCGG"`	`-3`	First string is smaller; mismatch in middle
`"WHOOP!", "WHOA"`	`14`	First string is larger; mismatch in middle
`"Intel", "pentium"`	`-39`	First string is smaller; mismatch at start
`"STONY", "BROOK"`	`17`	First string is larger; mismatch at start
`"", "mouse"`	`-5`	First string is empty
`"lonely guy", ""`	`10`	Second string is empty
`"Wolfie", "Wolfie"`	`0`	Identical non-empty strings
`"", ""`	`0`	Two empty strings
`"happy", "Z"`	`14`	One argument is very short
`"WOLF", "WOLFIE"`	`-73`	First string is substring of second string
`"StonyBrook", "Stony"`	`66`	Second string is substring of first string

Part II: Find a String Among Several Concatenated Strings

```
int find_string(string target, string strings, int strings_length)
```

This function takes a null-terminated string called `target` and searches inside another string called `strings` for `target`. `strings` is more like an array of strings in the sense that it really consists of multiple null-terminated strings that have been concatenated together. As an example, suppose that `strings` were the following string, which contains 53 characters (including all null-terminators):

```
thanks\0thx\0CSE 220\0CSE 215\0can\0\0dm\0220\0help\0cna\0hepl\0
```

The function call `find_string("220", strings, 53)` would return `35` because the leftmost character of `"220"` can be found at index 35 of the `strings` array. The function would not return `15` (the starting index of the `"220"` inside `"CSE 220"`) because we are looking for the string `"220"` to appear as a single "unit" and not as part of a larger substring. Let's consider another example to drive this idea home.

The function call `find_string("215", strings, 53)` would return `-1` to indicate that `"215"` *as a single string* could not be found in `strings`. Although `"215"` appears as a substring of `"CSE 215"` inside of `strings`, the string `"215"` does not appear as a distinct entity inside `strings`.

The `find_string` function will be used towards the end of the assignment for building hash tables based on data loaded from disk.

The function takes the following arguments, in this order:

- `target`: the starting address of the target string we want to search for
- `strings`: the starting address of a string that consists of several null-terminated strings that have been concatenated together
- `strings_length`: the number of bytes in the `strings` string; this number includes all null-terminators in the count

Returns in `$v0`:

- the index of the first letter of `target` inside `strings`, or `−1` for error as described below

Returns `−1` in `$v0` for error under the following circumstances:

- `target` (including its null-terminator) is not found in `strings` using the searching algorithm described above
- `strings_length` is less than 2

Additional requirements:

- `find_string` must call `strcmp`. This call should be made to repeatedly check for matches between `target` and null-terminated substrings of `strings`.
- The function must not write any changes to main memory.

Examples:

In the examples given below, `strings` has the value

```
thanks\0thx\0CSE 220\0CSE 215\0can\0\0dm\0220\0help\0cna\0hepl\0
```

Function Arguments	**Return Value**
`"thanks", strings, 53`	`0`
`"hepl", strings, 53`	`48`
`"220", strings, 53`	`35`
`"CSE 220", strings, 53`	`11`
`"MIPS", strings, 53`	`-1`

Up Next: Basic Hash Table Operations

The next five parts of the assignment will require you to implement functions important for implementing hash tables. Remember that these functions support the implementation of hash table that uses open address hashing with linear probing.

One challenge you will need to overcome is that visualizing a hash table is not always that easy. You might wish to write some code to print the content of the `keys[]` and `values[]` arrays

Part III: Hashing Function

```
int hash(HashTable hash_table, string key)
```

An essential aspect of implementing hash tables is to have a hash function that maps a key to an index ("slot") in the hash table. For this assignment we will use a simple hashing formula, namely, the remainder that is generated when the sum the ASCII codes of the key is divided by the capacity of the hash table. This remainder is the hash code of the key.

As an example, suppose the key is `"SbU"` and the hash table's capacity is 11. The hash code will be $(83 + 98 + 85) \bmod 11 = 2$.

The function takes the following arguments, in this order:

- `hash_table`: a pointer to a hash table of string substitutions.
- `key`: a null-terminated string

Returns in `$v0`:

- the hash code of the string using the hashing formula described above

Given the starting address of a hash table, we can easily write code to extract the capacity, size and contents of the hash table. For example, suppose `$a0` contained the starting address of the hash table. Then `lw $t0, 0($a0)` and `lw $t1, 4($a0)` will load the capacity and size of the hash table into `$t0` and `$t1` respectively. We can used similar arithmetic to load the contents of the `keys` and `values` arrays, which will be discussed in a later part of the assignment.

Examples:

In the examples below, assume that the hash table has a capacity of 7.

Function Arguments	Return Value
`hash_table, "ams"`	6
`hash_table, "subtraction"`	1
`hash_table, "class"`	2

Part IV: Clear the Contents of a Hash Table

```
void clear(HashTable hash_table)
```

This function takes a pointer to a hash table struct (i.e., the starting address of a hash table struct) and performs the following actions by writing changes to main memory as needed:

1. sets the `size` field of the hash table to zero
2. sets all values in the hash table's `keys[]` array to zero

3. sets all values in the hash table's `values[]` array to zero

The function does not write any changes to the hash table's `capacity` field. You may assume that the value of the `capacity` field is valid.

The function takes the following argument:

- `hash_table`: a pointer to a hash table of string substitutions.

The function has no return value.

Additional requirements:

- The function must not write any changes to main memory except as specified.

Part V: Get the Value Associated with a Key in a Hash Table

`int, int get(HashTable hash_table, string key)`

This function takes a pointer to a hash table and the starting address of a key to search for in the hash table. Starting at index `hash(hash_table, key)` in the hash table's `keys[]` array, the function performs a linear search for the key, wrapping around back to index 0 when the maximum index at the right-hand end of the array is reached. For each slot beyond slot #`hash(hash_table, key)` that it inspects, the function adds 1 to a counter (call it `probes`, initially 0). Slots marked as available and slots containing non-matching keys are skipped over. Each time the function skips over such a slot it adds 1 to `probes`. The search terminates when one of the following conditions is met:

- The key is found. The function returns `index, probes`, where `index` is the index of the slot where the key was found. For example, if the slot at index `hash(hash_table, key)+2` happened to contain the key, the function would return `hash(hash_table, key)+2, 2`.
- An empty slot is encountered. The function returns `-1, probes`. For example, if the slot at index `hash(hash_table, key)` is empty, the function returns `-1, 0`. On the other hand, if the empty slot is encountered during probing, we have to add 1 to `probes` before returning `-1, probes`. (See Example #4, below.)
- Every slot in the hash table is checked and the key is not found. The function returns `-1, capacity-1`.

The function takes the following arguments, in this order:

- `hash_table`: a pointer to a hash table of string substitutions.
- `key`: a null-terminated string

Returns in `$v0`:

- the index in the hash table's `keys[]` array where they key was found, or `-1` if the key was not found

Returns in `$v1`:

- the number of probes required to find the key in the hash table's `keys[]` array or to determine that the key is not in the `keys[]` array

Additional requirements:

- The function must call `strcmp` and `hash`.
- The function must not write any changes to main memory.

Example #1:

Hash table contents:

```
0: 0x00000000 -> 0x00000000
1: 0x00000000 -> 0x00000000
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: 0x00000000 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 0x00000000 -> 0x00000000
```

Function call: `get(hash_table, "usb")`

Hash code for key `"usb"` = 1

Return values: `-1, 0`

Example #2:

Hash table contents:

```
0: 0x00000000 -> 0x00000000
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: 0x00000000 -> 0x00000000
6: 101 -> CSE101
```

Function call: `get(hash_table, "kk")`

Hash code for key `"kk"` = 4

Return values: `4, 0`

Example #3: Hash table contents:

```
0: 0x00000000 -> 0x00000000
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
```

```
4: kk -> OK thanks
5: 0x00000000 -> 0x00000000
6: 101 -> CSE101
```

Function call: `get(hash_table, oh")`

Hash code for key `"oh"` = 5

Return values: `-1, 0`

Example #4:

Hash table contents:

```
   0: i -> I
   1: usb -> Universal Serial Bus
   2: 0x00000001 -> 0x00000000
   3: 0x00000001 -> 0x00000000
   4: thx -> thanks
   5: 0x00000000 -> 0x00000000
   6: 101 -> CSE101
```

Function call: `get(hash_table, "calss")`

Hash code for key `"calss"` = 2

Return values: `-1, 3`

Example #5:

Hash table contents:

```
0: 101 -> CSE101
1: ams -> Applied Mathematics
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Function call: `get(hash_table, "ams")`

Hash code for key `ams""` = 6

Return values: `1, 2`

Example #6:

Hash table contents:

```
0: kk -> OK thanks
1: 0x00000001 -> 0x00000000
2: 0x00000001 -> 0x00000000
3: ams -> Applied Mathematics
4: 0x00000000 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 0x00000001 -> 0x00000000
```

Function call: `get(hash_table, "ams")`

Hash code for key `"ams"` = 6

Return values: `3, 4`. Note how the probing wrapped around from index 6 back to index 0.

Example #7:

Hash table contents:

```
0: 101 -> CSE101
1: ams -> Applied Mathematics
2: cs -> Computer Science
3: oh -> OH
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Function call: `get(hash_table, "gg")`

Hash code for key `"gg"` = 3

Return values: `-1, 6`

Part VI: Insert/Update a (Key, Value) Pair in a Hash Table

```
int, int put(HashTable hash_table, string key, string value)
```

This function takes a pointer to a hash table and the starting addresses of a key and its corresponding value (both of which are strings). The purpose of the function is to insert the (key, value) pair into the hash table if the key is not present in the hash table. If the key is present in the hash table, then its associated value is updated to `value`. The function begins by calling `get(hash_table, key)` to check if the key is already in the table.

Depending on `get`'s return values and the state of the hash table, the function performs one of several actions:

- If the key is in the hash table. `put` updates the value associated with the key to the `value` passed as the function argument. The function returns `get_v0, get_v1`, the return values of the `get` function.
- If the key is not in the hash table, `put` might or might not be able to add the key and value to the hash table:
    - If the hash table's size equals its capacity, the key and value cannot be inserted. The function returns `-1, -1`

  - Assuming there is at least one free slot in the hash table, `put` will be able to insert the key and value. Starting at `hash_table.keys[hash(key)]`, `put` uses linear probing to search for an empty or available slot, wrapping around back to index 0 if it reaches the maximum index at the right-hand end of the array. For each non-empty, non-available slot the function encounters, it increments a counter called `probes`. Eventually, the function is guaranteed to find an empty or available slot at some index. Call this location `index`. The function writes the (key, value) pair into the hash table, increments the hash table's size by 1, and returns `index, probes`.

The function takes the following arguments, in this order:

- `hash_table`: a pointer to a hash table of string substitutions.
- `key`: a null-terminated string that represents a key
- `value`: a null-terminated string that represents the value associated with the given key

Returns in `$v0`:

- the first return value of `get`, `-1`, or `index`, depending on which case described above is relevant

Returns in `$v1`:

- the second return value of `get`, `-1`, or `probes`, depending on which case described above is relevant

Additional requirements:

- The function must call `get` and `hash`.
- The function must not write any changes to main memory except as specified.

Example #1:

Hash table contents before function call:

```
0: 0x00000000 -> 0x00000000
1: 0x00000000 -> 0x00000000
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: 0x00000000 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 0x00000000 -> 0x00000000
```

Function call: `put(hash_table, "usb", "Universal Serial Bus")`

Hash code for key `"usb"` = 1

Return values: `1, 0`

Hash table contents after function call:

```
0: 0x00000000 -> 0x00000000
1: usb -> Universal Serial Bus
```

```
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: 0x00000000 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 0x00000000 -> 0x00000000
```

Example #2:

Hash table contents before function call:

```
0: 0x00000000 -> 0x00000000
1: 0x00000000 -> 0x00000000
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: oh -> OH
6: 0x00000000 -> 0x00000000
```

Function call: `put(hash_table, "sbu", "Stony Brook University")`

Hash code for key `"sbu"` = 1

Return values: `1, 0`

Hash table contents after function call:

```
0: 0x00000000 -> 0x00000000
1: sbu -> Stony Brook University
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: oh -> OH
6: 0x00000000 -> 0x00000000
```

Example #3:

Hash table contents before function call:

```
0: 0x00000000 -> 0x00000000
1: bsu -> Boise State University
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: oh -> OH
6: 0x00000000 -> 0x00000000
```

Function call: `put(hash_table, "sbu", "Stony Brook University")`

Hash code for key `"sbu"` = 1

Return values: `2, 1`

Hash table contents after function call:

```
0: 0x00000000 -> 0x00000000
1: bsu -> Boise State University
2: sbu -> Stony Brook University
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: oh -> OH
6: 0x00000000 -> 0x00000000
```

Example #4:

Hash table contents before function call:

```
0: 0x00000000 -> 0x00000000
1: bsu -> Boise State University
2: usb -> Universal Serial Bus
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: oh -> OH
6: 0x00000000 -> 0x00000000
```

Function call: `put(hash_table, "sbu", "Stony Brook University")`

Hash code for key `"sbu"` = 1

Return values: `3, 2`

Hash table contents after function call:

```
0: 0x00000000 -> 0x00000000
1: bsu -> Boise State University
2: usb -> Universal Serial Bus
3: sbu -> Stony Brook University
4: kk -> OK thanks
5: oh -> OH
6: 0x00000000 -> 0x00000000
```

Example #5:

Hash table contents before function call:

```
0: 101 -> CSE101
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
```

```
5: thx -> thanks
6: yuo -> you
```

Function call: `put(hash_table, "ams", "Applied Mathematics")`

Hash code for key `"ams"` = 6

Return values: `2, 3`

Hash table contents after function call:

```
0: 101 -> CSE101
1: usb -> Universal Serial Bus
2: ams -> Applied Mathematics
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Example #6:

Hash table contents before function call:

```
0: 0x00000001 -> 0x00000000
1: 0x00000001 -> 0x00000000
2: thx -> thanks
3: 0x00000001 -> 0x00000000
4: 0x00000001 -> 0x00000000
5: 0x00000001 -> 0x00000000
6: 0x00000001 -> 0x00000000
```

Function call: `put(hash_table, "thx", "OK thanks")`

Hash code for key `"thx"` = 4

Return values: `2, 5`. Note how the probing wrapped around from index 6 back to index 0.

Hash table contents after function call:

```
0: 0x00000001 -> 0x00000000
1: 0x00000001 -> 0x00000000
2: thx -> OK thanks
3: 0x00000001 -> 0x00000000
4: 0x00000001 -> 0x00000000
5: 0x00000001 -> 0x00000000
6: 0x00000001 -> 0x00000000
```

Example #7:

Hash table contents before function call:

```
0: 101 -> CSE101
1: 0x00000001 -> 0x00000000
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Function call: `put(hash_table, "oh", "OH")`

Hash code for key `"oh"` = 5

Return values: `1, 3`. Note how the probing wrapped around from index 6 back to index 0.

Hash table contents after function call:

```
0: 101 -> CSE101
1: oh -> OH
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Additional requirements:

- The function must call `get` and `hash`.
- The function must not write any changes to main memory except as specified.

Part VII: Delete a (Key, Value) Pair from a Hash Table

`int, int delete(HashTable hash_table, string key)`

This function attempts to delete the (key, value) pair associated with the key named `key` in the hash table. If the hash table is empty, the function simply returns `-1, 0`. Assuming the hash table is not empty, the function calls `get(hash_table, key)` to check if the key is present in the hash table. If not, then `delete` returns whatever return values were returned by `get`. If so, then `delete` sets the value of the key in the hash table to 1, its associated value in the hash table to 0, and then decrements the hash table's size by 1.

The function takes the following arguments, in this order:

- `hash_table`: a pointer to a hash table of string substitutions.
- `key`: a null-terminated string that represents a key

Returns in `$v0`:

- `-1` or the first return value of `get`, depending on which case described above is relevant

Returns in `$v1`:

- `0` or the second return value of `get`, depending on which case described above is relevant

Additional requirements:

- The function must call `get`.
- The function must not write any changes to main memory except as specified.

Example #1:

Hash table contents before function call:

```
0: 0x00000000 -> 0x00000000
1: 0x00000000 -> 0x00000000
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: 0x00000000 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 0x00000000 -> 0x00000000
```

Function call: `delete(hash_table, "usb")`

Hash code for key `"usb"` = 1

Return values: `-1, 0`

Hash table contents after function call:

```
0: 0x00000000 -> 0x00000000
1: 0x00000000 -> 0x00000000
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: 0x00000000 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 0x00000000 -> 0x00000000
```

Example #2:

Hash table contents before function call:

```
0: 0x00000000 -> 0x00000000
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: 0x00000000 -> 0x00000000
6: 101 -> CSE101
```

Function call: `delete(hash_table, "kk")`

Hash code for key `"kk"` = 4

Return values: `4, 0`

Hash table contents after function call:

```
0: 0x00000000 -> 0x00000000
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: 0x00000001 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 101 -> CSE101
```

Example #3:

Hash table contents before function call:

```
0: 0x00000000 -> 0x00000000
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: 0x00000000 -> 0x00000000
6: 101 -> CSE101
```

Function call: `delete(hash_table, "oh")`

Hash code for key `"oh"` = 5

Return values: `-1, 0`

Hash table contents after function call:

```
0: 0x00000000 -> 0x00000000
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: 0x00000000 -> 0x00000000
6: 101 -> CSE101
```

Example #4:

Hash table contents before function call:

```
0: i -> I
1: usb -> Universal Serial Bus
```

```
2: 0x00000001 -> 0x00000000
3: 0x00000001 -> 0x00000000
4: thx -> thanks
5: 0x00000000 -> 0x00000000
6: 101 -> CSE101
```

Function call: `delete(hash_table, "calss")`

Hash code for key `"calss"` = 2

Return values: `-1, 3`

Hash table contents after function call:

```
0: i -> I
1: usb -> Universal Serial Bus
2: 0x00000001 -> 0x00000000
3: 0x00000001 -> 0x00000000
4: thx -> thanks
5: 0x00000000 -> 0x00000000
6: 101 -> CSE101
```

Example #5:

Hash table contents before function call:

```
0: 101 -> CSE101
1: ams -> Applied Mathematics
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Function call: `delete(hash_table, "ams")`

Hash code for key `"ams"` = 6

Return values: `1, 2`

Hash table contents after function call:

```
0: 101 -> CSE101
1: 0x00000001 -> 0x00000000
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Example #6:

Hash table contents before function call:

```
0: kk -> OK thanks
1: 0x00000001 -> 0x00000000
2: 0x00000001 -> 0x00000000
3: ams -> Applied Mathematics
4: 0x00000000 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 0x00000001 -> 0x00000000
```

Function call: `delete(hash_table, "ams")`

Hash code for key `"ams"` = 6

Return values: `3, 4`

Hash table contents after function call:

```
0: kk -> OK thanks
1: 0x00000001 -> 0x00000000
2: 0x00000001 -> 0x00000000
3: 0x00000001 -> 0x00000000
4: 0x00000000 -> 0x00000000
5: 0x00000000 -> 0x00000000
6: 0x00000001 -> 0x00000000
```

Example #7:

Hash table contents before function call:

```
0: 101 -> CSE101
1: ams -> Applied Mathematics
2: cs -> Computer Science
3: oh -> OH
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Function call: `delete(hash_table, "gg")`

Hash code for key `"gg"` = 3

Return values: `-1, 6`

Hash table contents after function call:

```
0: 101 -> CSE101
1: ams -> Applied Mathematics
```

```
2: cs -> Computer Science
3: oh -> OH
4: kk -> OK thanks
5: thx -> thanks
6: yuo -> you
```

Part VIII: Build a Hash Table from a File

```
int build_hash_table(HashTable hash_table, string strings, int strings_length,
                     string filename)
```

This function loads a file containing (key, value) pairs and builds a hash table struct based on those file contents. Each line of the file is formatted as follows:

```
KEY VALUE\n
```

where `KEY` consists of printable, non-delimiter characters and `VALUE` consists of any printable characters. A single space separates the `KEY` and `VALUE` strings. A single newline character terminates each line.

A sample hash table file is shown below:

```
yuo you
bsu Boise State University
sub subtraction
cna can
sbu Stony Brook University
220 CSE 220
thx thanks
kk OK thanks
wat what
kk thanks
220 CSE 220 is da best
```

Notice that there are duplicate keys in the file. This issue will be addressed below.

To assist with reading and writing files, MARS has several system calls:

Service	**Code in `$v0`**	**Arguments**	**Results**
open file	13	`$a0` = address of null-terminated filename string `$a1` = flags `$a2` = mode	`$v0` contains file descriptor (negative if error)
read from file	14	`$a0` = file descriptor `$a1` = address of input buffer `$a2` = maximum # of characters to read	`$v0` contains # of characters read (0 if end-of-file, negative if error)
close file	16	`$a0` = file descriptor	

Service 13: MARS implements three *flag* values: 0 for read-only, 1 for write-only with create, and 9 for write-only with create and append. It ignores *mode*. The returned file descriptor will be negative if the operation failed. MARS maintains file descriptors internally and allocates them starting with 3. File descriptors 0, 1 and 2 are always open for reading from standard input, writing to standard output, and writing to standard error, respectively. An example of how to use these syscalls can be found on the MARS syscall web page.

The function must assume that every line of the file ends only with a `'\n'` character, *not* the two-character combination `"\r\n"` employed in Microsoft Windows. If you create your own hash table files for testing purposes, use MARS to edit the files. If you are developing on a Windows computer, do not use a regular text editor like Notepad. Such an editor will insert both characters. In contrast, MARS will insert only a `'\n'` at the end of each line, *so only use MARS to create custom hash tables*.

The function must not attempt to allocate new memory to store the strings found in the hash table file. Rather, the function must call the `find_string` function from Part II on the `strings` array to find the starting addresses of the keys and values it discovers in the file. The function may assume that every string in the file will be present in the `strings` array. The overall algorithm that this function must implement is as follows:

1. Clear the hash table.
2. Attempt to open the file using service #13. If the file does not exist, the function returns `-1` and makes no other changes to main memory besides clearing the hash table.
3. Assuming the file was opened successfully, repeat the following steps until service #14 returns 0, indicating that the last line of the file was read:
   a. Use service #14 to read the key into a temporary buffer. This temporary buffer must be allocated on the system stack. You may assume that the key is never more than 79 bytes in length. Thus, before this loop starts iteration your function should allocate 80 bytes on the stack: 79 bytes to store the characters of the key, plus 1 more byte to accommodate a null-terminator.
   b. Call `find_string` to locate the starting address of the key in the `strings` array.
   c. Use service #14 to read the value into a temporary buffer. As for the key, you may assume that the value is never more than 79 bytes in length.
   d. Call `find_string` to locate the starting address of the value in the `strings` array.
   e. With the starting addresses of the key and value now available, call `put` on the hash table to store the (key, value) pair in the hash table. Note that if the key that was just ready is already in the hash table, the `put` function will simply update the contents of the hash table with the new value.
4. Deallocate any memory that was allocated on the stack.
5. Close the file using service #16.
6. Return the number of (key, value) pairs that were read from disk. This number is not necessarily the same as the size of the hash table.

The function takes the following arguments, in this order:

- `hash_table`: a pointer to a hash table of string substitutions.
- `strings`: a string consisting of several null-terminated strings that have been concatenated together
- `strings_length`: the length of `strings`, including all null-terminators (including the final null-

terminator)

- `filename`: the name of a text file whose contents represent a hash table as described above

Returns in `$v0`:

- the number of (key, value) pairs that were read from the file

Additional requirements:

- The function must call `put` and `find_string`.
- The function must not write any changes to main memory except as specified.

Example:

Contents of the file `hash_table1.txt`:

```
yuo you
bsu Boise State University
sub subtraction
cna can
sbu Stony Brook University
220 CSE 220
thx thanks
kk OK thanks
wat what
kk thanks
220 CSE 220 is da best
```

The `strings` array, which has length 661:

```
"wat\0class range\0usb\0I\0gg\0price hose\0hmmmm\0disimprison\0can\0hmm\0
Applied Mathematics\0defuze\0Universal Serial Bus\0jazzy\0inapprehensiven
ess\0OK thanks\0what\0good game\0giftless\0rotate\0sillllllly\0hartselle\0
thx\0succinct street\0brockport\0Boise State University\0silly\0class\0bil
lhead\0first floor\0sub\0languid lip\0Stony Brook\0CSE101\0OH\0MIPS\0colo
rado\0cs\0yuo\0calss\0able\0oh\0help\0sbu\0Stony Brook University\0imbark
\0kuwait\0i\0receipt marble\0kk\0MIPSR10000\0fido\0teddy\0second-hand
skate\0bsu\0sto\0conclave\0you\0melrose\0subtraction\0altarage\0hepl\0220
\0you\0orange clam\0class brush\0Computer Science\0101\0CSE 220 is da best
\0argh\0arrgghh\0zebra fruit\0onions\0u\0thanks\0Uni\0cyclamycin\0CSE 220
\0ams\0cna\0"
```

The resulting hash table with capacity 13 is depicted below. The return value of the function is `11`:

```
0: 0x00000000 -> 0x00000000
1: 0x00000000 -> 0x00000000
2: thx -> thanks
3: 0x00000000 -> 0x00000000
```

```
4: 0x00000000 -> 0x00000000
5: bsu -> Boise State University
6: sub -> subtraction
7: cna -> can
8: sbu -> Stony Brook University
9: 220 -> CSE 220 is da best
10: kk -> thanks
11: yuo -> you
12: wat -> what
```

Additional requirements:

- The function must not write any changes to main memory except as specified.

Part IX: Autocorrect a String

```
int autocorrect(HashTable hash_table, string src, string dest, string strings,
                int string_length, string filename)
```

The function takes a pointer to a hash table and the starting addresses of a null-terminated string (`src`) and an output buffer (`dest`). The function iterates over the input buffer, searching for substrings (candidate keys) that match keys in the hash table. Candidate keys could be delimited by spaces, commas, periods, question marks or exclamation points. It is possible that multiple delimiters could appear next to each other in `src` with no non-delimiters in between. As the function identifies substrings, it calls `get` on the hash table to check if the substring is a key in the hash table. If a substring is a key, then its corresponding value is written into `dest`. All other substrings and all delimiters are written without modification into `dest`. The function returns the number of substitutions that were made.

The hash table is initially filled with garbage data, so `autocorrect` must call `build_hash_table(hash_table, strings, strings_length, filename)` to initialize the hash table with the string substitutions proscribed by the contents of `filename`.

The function takes the following arguments, in this order:

- `hash_table`: a pointer to an uninitialized (not cleared) hash table of string substitutions that will be built (indirectly) by the function. The function is guaranteed that the capacity of the hash table is at least great as the number of lines in `filename`.
- `src`: a null-terminated string whose contents will be autocorrected according to the contents of the hash table.
- `dest`: the destination buffer to store the autocorrected result. You may assume that this buffer is long enough to accommodate the corrected string and its null-terminator.
- `strings`: the starting address of a string that consists of several null-terminated strings that have been concatenated together. The function is guaranteed that every string in the `filename` appears in `strings`.
- `strings_length`: the number of bytes in the `strings` string; this number includes all null-terminators in the count

- `filename`: the name of a text file whose contents represent a hash table as described in Part VIII

Returns in `$v0`:

- the number of substitutions that were made by the function

Additional requirements:

- The function must call `build_hash_table` and `get`.
- The function must not write any changes to main memory except as specified.

Example #1:

For this first example we will provide comprehensive details of the test case, including all arguments. For the subsequent examples we will show only the contents of the hash table, `src` argument and `dest` argument.

```
hash_table = address of unitiaialized hash table (capacity = 29)
src = "I left my usb drive in the ams classroom. thx for returning it!\0"
dest = "gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
        gggggggggggggggggggggggggggggggggg"
strings = "kk\0cs\0Stony Brook University\0what\0yuo\0CSE 220\0argh\0oh\0
           OK thanks\0arrgghh\0u\0101\0hmmmm\0good game\0Universal Serial
           Bus\0you\0help\0sto\0gg\0Stony Brook\0subtraction\0Boise State
           University\0thx\0OH\0silly\0sillllllly\0thanks\0hepl\0hmm\0can
           \0usb\0wat\0calss\0sbu\0220\0MIPS\0class\0CSE101\0Applied
           Mathematics\0ams\0bsu\0i\0cna\0MIPSR10000\0sub\0you\0Computer
           Science\0I\0"
strings_length = 334
filename = "subs1.txt"
```

Hash table contents after the call to `build_hash_table`:

```
 0: 0x00000000 -> 0x00000000
 1: u -> you
 2: 101 -> CSE101
 3: 220 -> CSE 220
 4: gg -> good game
 5: yuo -> you
 6: ams -> Applied Mathematics
 7: 0x00000000 -> 0x00000000
 8: 0x00000000 -> 0x00000000
 9: 0x00000000 -> 0x00000000
10: 0x00000000 -> 0x00000000
11: sbu -> Stony Brook University
12: cs -> Computer Science
13: oh -> OH
14: usb -> Universal Serial Bus
15: bsu -> Boise State University
```

```
16: cna -> can
17: sub -> subtraction
18: i -> I
19: hepl -> help
20: kk -> OK thanks
21: thx -> thanks
22: wat -> what
23: calss -> class
24: sto -> Stony Brook
25: sillllllly -> silly
26: arrgghh -> argh
27: hmmmm -> hmm
28: MIPSR10000 -> MIPS
```

Return value: `3`

Updated contents of `dest`:

```
dest = "I left my Universal Serial Bus drive in the Applied Mathematics
        classroom. thanks for returning it!\0"
```

In this example the destination buffer was exactly the right length to store the autocorrected string. Note how `"usb"` was replaced with `"Universal Serial Bus"`, `"ams"` was replaced with `"Applied Mathematics"` and `"thx"` was replaced with `"thanks`.

Example #2:

Hash table contents:

```
0: ams -> Applied Mathematics
1: sbu -> Stony Brook University
2: hmmmm -> hmm
3: sub -> subtraction
4: cs -> Computer Science
5: 0x00000000 -> 0x00000000
6: MIPSR10000 -> MIPS
```

Other function arguments:

```
src =  "hmmm... I really like the MIPSR10000 processor. And I like cs,
        ams and sbu too!\0"
dest = "gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
        gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg"
```

Return value: `4`

Updated contents of `dest`:

```
dest = "hmmm... I really like the MIPS processor. And I like Computer
```

```
        Science, Applied Mathematics and Stony Brook University too!\0"
```

Example #3:

Hash table contents:

```
0: arrgghh -> argh
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: thx -> thanks
5: sillllllly -> silly
6: ams -> Applied Mathematics
```

Other function arguments:

```
src =  "That guy...arrgghh...he is so sillllllly, you know?\0"
dest = "gggggggggggggggggggggggggggggggggggggggggggg"
```

Return value: `2`

Updated contents of `dest`:

```
dest = "That guy...argh...he is so silly, you know?\0"
```

Example #4:

Hash table contents:

```
0: arrgghh -> argh
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: thx -> thanks
5: sillllllly -> silly
6: ams -> Applied Mathematics
```

Other function arguments:

```
src =  ".  . ,,, ?? !!  ,,,,??!\0"
dest = "gggggggggggggggggggggggg"
```

Return value: `0`

Updated contents of `dest`:

```
dest = ".  . ,,, ?? !!  ,,,,??!\0"
```

Example #5:

Hash table contents:

```
0: arrgghh -> argh
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: thx -> thanks
5: sillllllly -> silly
6: ams -> Applied Mathematics
```

Other function arguments:

```
src =  "thxusbthxsillllllllyamsamsokokarrgghh\0"
dest = "ggggggggggggggggggggggggggggggggggggg"
```

Return value: 0 Updated contents of `dest`:

```
dest = "thxusbthxsillllllllyamsamsokokarrgghh\0"
```

Example #6:

Hash table contents:

```
0: arrgghh -> argh
1: usb -> Universal Serial Bus
2: 0x00000000 -> 0x00000000
3: 0x00000000 -> 0x00000000
4: thx -> thanks
5: sillllllly -> silly
6: ams -> Applied Mathematics
```

Other function arguments:

```
src =  "thxusbthxsillllllllyamsamsokokarrgghh\0"
dest = "gggggggggggggggggggggggggggggggggggggggg"
```

Return value: 0 Updated contents of `dest`:

```
dest = "thxusbthxsillllllllyamsamsokokarrgghh\0ggg"
```

In this example the destination buffer is three bytes longer than it needs to be. Notice that `autocorrect` does not overwrite those three bytes.

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work to Blackboard you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.

2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the College of Engineering and Applied Sciences.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating in a homework may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.

How to Submit Your Work for Grading

To submit your `proj3.asm` file for grading:

1. Login to Blackboard and locate the course account for CSE 220.
2. Click on “Assignments” in the left-hand menu and click the link for this assignment.
3. Click the “Browse My Computer” button and locate the `proj3.asm` file. Submit only that one `.asm` file.
4. Click the “Submit” button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the steps again. We will grade only your last submission.