1 Background

In this Project, we’ll be working with structs. In C, a struct is simply a plan for how variables will be laid out inside an object. For instance, consider this struct, which is one of the ones we’ll be using in this project:

```c
struct ListNode {
    ListNode *next;
    int key;
    char *val;
};
```

This struct has three fields: a pointer, an integer, and another pointer. Each is 32 bits in size, and the fields are laid out in the same order as they are declared in the struct. So an instance of a `ListNode` struct looks like this:

```
+------+
| byte 0 | | <----- a pointer to a ListNode points here
| 1 | next |
| 2 | |
| 3 | |
+------+
```

```
+------+
| 4 | | <----- read pointer+4 bytes to get the key
| 5 | key |
| 6 | |
| 7 | |
+------+
```

```
+------+
| 8 | | <----- read pointer+8 bytes to get the val
| 9 | val |
| 10 | |
| 11 | |
+------+
```

(Remember, a `pointer` in C means the address of something. So the `word` at the start of the struct contains the address of another `ListNode` object (or null). The third word (`val`) contains the address of a string.)
2 Overview

In this project, you will write several functions which use structs: three that read or modify (sorted) linked lists, and one that performs an in-order traversal of a Binary Search Tree.

2.1 Required Filenames to Turn in

Name your assembly language file `asm5.s`.

2.2 Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- `add, addi, sub, addu, addiu`
- `and, andi, or, ori, xor, xori, nor`
- `beq, bne, j`
- `jal, jr`
- `slt, slti`
- `sll, sra, srl`
- `lw, lh, lb, sw, sh, sb`
- `la`
- `syscall`

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), do not use them! We want you to learn the fundamentals of how assembly language works - you can use fancy tricks after this class is over.

3 The Data Structures

3.1 Key/Value Pairs

Both of the data structures you use in this project will have key/value pairs. A data structure with a key/value pair is one where there are two (or more) pieces of each element in the structure; the first (the key) tells us how to sort the elements, and the second (the value) is simply “extra” data that we carry around.

For instance, if we had a database of people, the key might be the Social Security Number, and the value might be the Name. No two people can have
the same SSN - although it’s possible that two people have the same name. So when we insert a new item into the data structure, we specify both the key and the value (the SSN and the Name); but will only consider the SSN when sorting the data (or when looking for duplicates).

Famous examples of a key/value data structure are Java’s HashMap and TreeMap classes, or the python dict.

To simplify your code, I won’t require you to allocate new nodes for your list when you insert. Instead, my code will provide you with a Node object to add to the list; the Node already has the correct values for the key, val fields. So, you will only need to insert it into the list.

3.2 Lists

In this project, we’ll use two struct types to represent a list: a “wrapper” class, which represents the list itself, and a separate type for the nodes. An empty list is represented by a (non-null) pointer to the wrapper class; however, the wrapper class contains a head pointer, which is set to null. Similarly, a list with one node is represented with a wrapper object, which has a head pointer that points to a node; inside the node, the next field is null. And so on...

The types used by this project, for the list, are:

```c
struct List { // wrapper object, represents an entire list
    ListNode *head;
    int        count; // number of nodes in the list
};

struct ListNode { // represents a single node
    ListNode *next;
    int        key;
    char       *val;
};
```

3.3 Binary Search Tree

Your BST implementation won’t need a wrapper class, since you’ll only be printing out the in-order traversal - and you’ll be implementing it as a recursive function. In this system, we’ll represent an empty tree (or, an empty subtree) with a null pointer.

```c
struct BSTNode {
    BSTNode  *left;
    BSTNode  *right;
    int       key;
    char      *val;
};
```
4 Tasks

Your file must define the functions listed below.

You will notice that the functions don’t include much error checking; for instance, they don’t check to see if pointers are null. That’s OK - I’m simplifying the functions for you. In some other class, when you’re writing real C code, you can be more careful about checking for errors!

4.1 Task 1: list_getCount()

```c
int list_getCount(List *list)
{
    return list->count;
}
```

4.2 Task 2: list_insert()

Note that I’m passing you a pre-created ListNode object, which already has the key and value fields filled in. So you should just read that struct to find out where it belongs in the list.

```c
void list_insert(List *list, ListNode *node)
{
    if (list->head == NULL || node->key <= list->head->key)
    {
        node->next = list->head;
        list->head = node;
    }
    else
    {
        ListNode *cur = list->head;
        while (cur->next != NULL && cur->next->key < node->key)
        {
            cur = cur->next;
        }
        node->next = cur->next;
        cur->next = node;
    }

    list->count++;
    return;
}
```
4.3 Task 3: list_printAll()

```c
void list_printAll(List *list)
{
    printf("--- list_printAll BEGIN ---\n");

    ListNode *cur = list->head;
    while (cur != NULL)
    {
        printf("key=%d val='%s'\n", cur->key, cur->val);
        cur = cur->next;
    }

    printf("--- list_printAll END ---\n");
    return;
}
```

4.4 Task 4: bst_printInOrder()

You will notice that, in this Task, I have a (non-recursive) wrapper function - which prints the beginning and ending text - and then a recursive helper function which actually performs the traversal.

Your implementation must print out exactly what I show here in the spec. Your implementation also must use a recursive function to perform the traversal (the TAs will check by hand). However, if you want to change the details of how you do the recursion, that’s OK - we won’t test the bst_printInOrder_r() function. Instead, we’ll only test bst_printInOrder().

```c
void bst_printInOrder(BSTNode *root)
{
    printf("--- bst_printInOrder BEGIN ---\n");
    bst_printInOrder_r(root);
    printf("--- bst_printInOrder END ---\n");
}

void bst_printInOrder_r(BSTNode *root)
{
    if (root == NULL)
        return; // empty tree or subtree, nothing to print

    bst_printInOrder_r(root->left);
    printf("key=%d val='%s'\n", root->key, root->val);
    bst_printInOrder_r(root->right);
}
```
5 Running Your Code

You should always run your code using the grading script before you turn it in. However, while you are writing (or debugging) your code, it often handy to run the code yourself.

5.1 Running With Mars (GUI)

To launch the Mars application (as a GUI), open the JAR file that you downloaded from the Mars website. You may be able to just double-click it in your operating system; if not, then you can run it by typing the following command:

```
java -jar <marsJarFileName>
```

This will open a GUI, where you can edit and then run your code. Put your code, plus one testcase, in some directory. Open your code in the Mars editor; you can edit it there. When it’s ready to run, assemble it (F3), run it (F5), or step through it one instruction at a time (F7). You can even step backwards in time (F8)!

5.1.1 Running the Mars GUI the First Time

The first time that you run the Mars GUI, you will need to go into the Settings menu, and set two options:

- **Assemble all files in directory** - so your code will find, and link with, the testcase
- **Initialize Program Counter to 'main' if defined** - so that the program will begin with `main()` (in the testcase) instead of the first line of code in your file.

5.2 Running Mars at the Command Line

You can also run Mars without a GUI. This will only print out the things that you explicitly print inside your program (and errors, of course). However, it’s an easy way to test simple fixes. (And of course, it’s how the grading script works.) Perhaps the nicest part of it is that (unlike the GUI, as far as I can tell), you can tell Mars exactly what files you want to run - so multiple testcases in the directory is OK.

To run Mars at the command line, type the following command:

```
```

---

1 Why can’t you put multiple testcases in the directory at the same time? As far as I can tell (though I’m just learning Mars myself), the Mars GUI only runs in two modes: either (a) it runs only one file, or (b) it runs all of the files in the same directory. If you put multiple testcases in the directory, it will get duplicate-symbol errors.

2 Mars has lots of additional options that allow you to dump more information, but I haven’t investigated them. If you find something useful, be sure to share it with the class!
java -jar <marsJarFileName> sm <testcaseName>.s <yourSolution>.s

6 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).

- **Not** use any other files (unless allowed by the project spec) - since our grading script won’t know to use them.

- Follow the spec precisely (don’t change any names, or edit the files I give you, unless the spec says to do so).

- (In projects that require output) match the required output **exactly**! Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I strongly **recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

6.1 mips_checker.pl

In addition to downloading grade_asm5, you should also download mips_checker.pl, and put it in the same directory. The grading script will call the checker script.

6.2 Testcases

You can find a set of testcases for this project at

http://lecturer-russ.appspot.com/classes/cs252/spring18/asm/asm5/

You can also find them on Lectura at

/home/russelll/cs252_website/asm/asm5/

For assembly language programs, the testcases will be named test.*.s . For C programs, the testcases will be named test.*.c . For Java programs, the testcases will be named Test.*.java . (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in .out; our grading script needs to have both files available in order to test your code.

For many projects, we will have “secret testcases,” which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. You are **encouraged to write testcases of your own, in order to better test your code.**
6.3 Automatic Testing

We have provided a testing script (in the same directory), named `grade_asm5`, along with a helper script, `mips_checker.pl`. Place both scripts, all of the testcase files (including their `.out` files), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It might also work on your Mac or Linux box, but no promises!)

6.4 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, testcases are the exception. We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

7 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.

7.1 Late Work

No late work for this assignment!