CSC 252 (Fall 24): Computer Organization

Assembly Project #6

Structs in Assembly

due at 5pm, Wed 4 Dec 2024

1 Purpose

In this Project, we'll be writing functions that manipulate structs - in particular, we'll be implementing some classic, simple functions to manipulate BSTs.

For each function, I've provided the C implementation - so you don't have to write the BST code, you just have to figure out how to make it work in assembly.

1.1 Required Filenames to Turn in

Name your assembly language file `asm6.s`.

1.2 Allowable Instructions

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

- `add, addi, sub, addu, addiu, subu`
- `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`
- `mult, div, mfhi, mflo`

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.

2 How do Structs Work in C?

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 the one we'll be using in this project:

```
struct BSTNode {
    int       key;
    BSTNode *left;
    BSTNode *right;
};
```

This struct has three fields: an integer, and two pointers. 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 `BSTNode` struct looks like this:

```
          +-------+
byte 0 |       |    <----- a pointer to a BSTNode points here
     1 | key   |
     2 |       |
     3 |       |
       +-------+
     4 |       |    <----- read pointer+4 bytes to get the 'left' pointer
     5 | left  |
     6 |       |
     7 |       |
       +-------+
     8 |       |    <----- read pointer+8 bytes to get the 'right' pointer
     9 | right |
    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 `BSTNode` object (or null).

2.1 What is `NULL` in C Code?

Remember, in C, the constant `NULL` represents a "null pointer" - that is, a pointer which is set to 0. Like `null` in Java or `None` in Python, it represents "no object."

Therefore, if you see a line of code like this:

```
if (ptr == NULL)
```

then it is asking "does the pointer point at nothing?" And the way you would check it is to compare the pointer to `$zero`.

3 Task Overview

You must implement the following functions:

3.1 `void bst_init_node(BSTNode *node, int key)`

You will never allocate memory in this project. Instead, the testcase will have the space already allocated - your job will be simply to **fill in** the memory, to initialize it.

This function accepts a pointer to a `BSTNode` object (you may assume that it is not `NULL`), and a key. The function initializes the fields of the struct.

```
void bst_init_node(BSTNode *node, int key)
{
    node->key   = key
    node->left  = NULL;
    node->right = NULL;
};
```

3.2 `BSTNode *bst_search(BSTNode *node, int key)`

This function takes a pointer to the root of a tree (which might be `NULL`, meaning that the tree is empty), and searches the tree for the key specified. If it finds the key, it returns the node which contains that key; if it doesn't, then it returns `NULL`.

```
BSTNode *bst_search(BSTNode *node, int key)
{
    BSTNode *cur = node;

    while (cur != NULL)
    {
        if (cur->key == key)
            return cur;

        if (key < cur->key)
            cur = cur->left;
        else
            cur = cur->right;
    }

    return NULL;
}
```

(spec continues on the next page)

3.3 `int bst_count(BSTNode *node)`

This function counts the number of nodes in a tree. If the tree given is empty, then this must return 0.

```
int bst_count(BSTNode *node)
{
    if (node == NULL)
        return 0;
    return bst_count(node->left) + 1 + bst_count(node->right);
}
```

3.4 Traversals

These two functions print out all of the keys stored in the tree. You must implement both an in-order and pre-order traversal.

```
void bst_in_order_traversal(BSTNode *node)
{
    if (node == NULL)
        return;

    bst_in_order_traversal(node->left);
    printf("%d\n", node->key);
    bst_in_order_traversal(node->right);
}

void bst_pre_order_traversal(BSTNode *node)
{
    if (node == NULL)
        return;

    printf("%d\n", node->key);
    bst_pre_order_traversal(node->left);
    bst_pre_order_traversal(node->right);
}
```

(spec continues on the next page)

3.5 `BSTNode *bst_insert(BSTNode *root, BSTNode *newNode)`

This function inserts a new node into a BST. Since we haven't implemented dynamic memory allocation in assembly, you will not allocate a new node; instead, I have provided a pre-allocated node for you.

The function is passed two pointers. The first represents the "old" tree; if it is `NULL`, then the tree is empty. The second pointer is a new node; you may assume that it is never `NULL`, that it has no children, and that it's not part of any other tree.

The function must return the root of the tree, after the insertion has been performed. Often, this will be the same as the old root; however, if the old tree was empty, then this will be the new node (since it's now the root of a tree).

```
BSTNode *bst_insert(BSTNode *root, BSTNode *newNode)
{
    if (root == NULL)
        return newNode;

    if (newNode->key < root->key)
        root->left  = bst_insert(root->left,  newNode);
    else
        root->right = bst_insert(root->right, newNode);

    return root;
}
```

(spec continues on the next page)

3.6 BSTNode *bst_delete(BSTNode *root, int key)

This function removes a single value from the BST. Again, you haven't implemented dynamic memory allocation, so we won't bother about freeing the node that you remove.

Your code will search the tree for the given key. If it exists, then the node will be removed; if not, then the tree will be unchanged.

Like `bst_insert()` above, this function returns the modified tree.

```
BSTNode *bst_delete(BSTNode *root, int key)
{
    if (root == NULL)
        return NULL;

    if (key < root->key) {
        root->left  = bst_delete(root->left,  key);
        return root;
    }
    if (root->key < key) {
        root->right = bst_delete(root->right, key);
        return root;
    }

    // delete, case 1: node is a leaf.
    if (root->left == NULL && root->right == NULL)
        return NULL;

    // delete, case 2: node has a single child
    if (root->left == NULL)
        return root->right;
    if (root->right == NULL)
        return root->left;

    // delete, case 3: the node has two children
    BSTNode *replace = root->right;
    while (replace->left != NULL)
        replace = replace->left;

    root->right = bst_delete(root->right, replace->key);   // could be changed

    replace->left  = root->left;
    replace->right = root->right;

    return replace;
}
```

4 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.

4.1 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**[1] 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)!

4.1.1 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.

4.2 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).[2] 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:

```
java -jar <marsJarFileName> sm <testcaseName>.s <yourSolution>.s
```

[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!

5 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!

5.1 Testcases

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.**

5.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_asm6`, 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 inside the standard Docker container that I've provided. It **might** also work on your Mac or Linux box, but no promises!)

5.3 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!

6 Turning in Your Solution

You must turn in your code to GradeScope. Turn in only your program; do not turn in any testcases.