CS350 Final Project

- Dustin Van Tate Testa
- Fall 2019
- BS Tree In MIPS Assembly

Questions:: How is recursion different between C and MIPS?

In C, pushing values onto the stack and handling parameters and such is handled automatically where as with mips we have to do this manually before calling the function. Realy mips doesn’t support functions or recursion, all it supports are goto’s, and we have to use the other hardware tools to provide the funcitonality needed. Although the C compiler will often assemble the C code to operate the same in assembly sometime it uses optimizations to make it so that we don’t waste resources on recursion (ie- TCO).

Questions:: How does the stack change with recursive calls?

As before we call the function, we push relevant register values onto the stack and they are written in in order. Calling another/the same function again pushes more values onto the stack after the previously mentioned values. When a function returns we then pop the values off the stack and put them back into relevant registers.

Code overview

- I’ll walk through the different functions, but I think they’re essentially direct translations from C to MIPS with some minor optimizations
- Opinion: I don’t think that the bTree struct needs to exist and instead functions should operate on the root node. If I didn’t use the provided skeleton code the performance would have been improved by a constant amount due to overhead of getting the root node from the bTree struct

```
# hint: you need to write a recursive call here
addNode:
    # procedural is just better

    # li $t1, 0           # prev=NULL
    move $t2, $a0       # n = root node

    # this is a do-while loop
    # it's acceptable here because we're gauranteed the root node has a value
    addNode_while:                              # do {
        move $t1, $t2                           # prev = n
        lw $t0, 0($t2)                          # get value from node

        # 3 case if statement
        blt $t0, $a1, addNode_while_if_lt
        bgt $t0, $a1, addNode_while_if_gt

        # else: n->value == value
        jr $ra  # already in tree, return

        addNode_while_if_gt:                    # if (n->value > value)
            lw $t2, 4($t2)                      #   n = n->leftChild
            j addNode_while_cond

        addNode_while_if_lt:                    # if (n->value < value)
            lw $t2, 8($t2)                      #   n = n->rightChild
            j addNode_while_cond

        addNode_while_cond:                     # } while (n != NULL)
            bnez $t2, addNode_while

    # allocate memory for new node
    subu $sp, $sp, 4    # adjust the stack pointer
    sw $ra, 0($sp)      # save the return address on stack

    li $a0, 3       # (struct Node*)malloc(sizeof(struct Node));
    jal malloc      # create a 3 words length space for root node
    # address is in $v0

    sw $a1, 0($v0)          #newNode->value = value;
    sw $zero, 8($v0)        #newNode->rightChild = NULL;
    sw $zero, 4($v0)        #newNode->leftChild = NULL;

    lw $ra, 0($sp)      # get the return address
    addu $sp, $sp, 4    # adjust the stack pointer

    # link it with tree
    blt $t0, $a1, addNode_ins_if_lt
    addNode_ins_if_gt:
        sw $v0, 4($t1)
        jr $ra
    addNode_ins_if_lt:
        sw $v0, 8($t1)
        jr $ra
```

```c
int add (struct bTree* root, int value) {

    struct Node* prev = NULL;
    struct Node* n = root->root;

    // find where to insert
    do {
        prev = n;
        if (n->value > value)
            n = n->leftChild;
        else if (n->value < value)
            n = n->rightChild;
        else // already in the tree
            return 1;
    }while (n != NULL);

    // insert
    struct Node* ret = malloc(sizeof(struct Node));
    ret->leftChild = NULL;
    ret->rightChild = NULL;
    ret->value = value;
    // link w tree
    if (prev->value > value)
        prev->leftChild = ret;
    else
        prev->rightChild = ret;

    return 0;
}
```

Add Node

- As you can see my Assembly for the add function is a near direct translation of my C code
- I chose against doing a recursive function because of the performance, readability, analysis, etc. costs
- Because we are guaranteed that the root node is not NULL, I used a do-while loop in the assembly so that I wouldn't have to waste a jump instruction

Contains

- Another near direct translation
- Again I chose to avoid using recursion for the same reasons as before
- Algorithm similar to what was used in addNode function
- Because the root value is guaranteed to not be null I can use a do while loop like before

```
contain:
    # recursion still bad

    lw $t2, 0($a0)      # n = bTree->root;
    # do while is acceptable here as we're gauranteed root node is not NULL
    contain_while:                      # do {
        lw $t3, 0($t2) # get value      #   v = n->value

        blt $t3, $a1, contain_while_lt  #   if (v < value) ...
        bgt $t3, $a1, contain_while_gt  #   if (v > value) ...

        # else (v == value)
        # return 1
        li $v0, 1
        jr $ra

        contain_while_gt:               #... if (v < value)
            lw $t2, 4($t2)              #       n = n->left
            j contain_while_cond
        contain_while_lt:               #... if (v > value)
            lw $t2, 8($t2)              #       n = n->right
            # j contain_while_cond
        contain_while_cond:
            bnez $t2, contain_while     # } while (n != NULL)

    li $v0, 0
    jr $ra
```

```
int contain (struct bTree* root, int value) {
    struct Node* n = root->root;

    // find where to insert
    while (n)
        if (n->value > value)
            n = n->leftChild;
        else if (n->value < value)
            n = n->rightChild;
        else // already in the tree
            return 1;

    return 0;
}
```

Delete Node::Overview

- Because I wanted to ensure the BSTree property stays true after deleting a node, I needed to make a more complex function
- I needed 3 functions in C to accomplish this
  - removeNodeN(): recursive function that returns a pointer to update subtree reference
  - findMin(): needed for rotating subtrees
  - removeNode(): function provided in assignment, because I didn’t want to change function signature I just made this function call removeNodeN()
- These were all directly translated from C to assembly

Delete Node::Find Min

- I modified my C findMin() that I used in lab8 to not use recursion
- I had to use a while loop here because the left branch may not be defined for the root node, this added an additional jump instruction to the condition
- The assembly version returns the left-most node->value as this is all that gets used
- This function diverged the most from the C code I wrote
- Node checked for null value before calling findMin so don’t need to check

```c
struct Node* findMin(struct Node* root) {
    if (root == NULL)
        return NULL;
    while (root->leftChild != NULL)
        root = root->leftChild;
    return root;
}
```

```asm
findMin:
    # not a do-while as that would change logic
    move $t0, $a0                   # n = root
    lw $a0, 0($t0)                  # v = n.value
    j findMin_while_cond            # while
    findMin_while:                  # {
        lw $a0, 0($t0)              #   v = n.value
    findMin_while_cond:             # } while (
        lw $t0, 4($t0)              #   (n = n.left
        bnez $t2, findMin_while     # ) != NULL);
    jr $ra                          # return n
```

Delete Node::removeNode

- Another near direct translation
- Instead of having the `return root` at the end of the function, I added it to each condition. This results in one less jump instruction at the cost of increased binary size. I think this change is fairly insignificant but results in one less jump instruction getting run and falls more in line with procedural programming ideology
- I chose to keep this function recursive as making the logic procedural wasn’t straightforward
- The function operates by deleting the node and updating the pointer in previous node using the return value
- It branches down the left and right subtrees until it finds the appropriate node, at which point it runs the appropriate code to delete it (different code depending on number of child nodes we have to move)

removeNode C code

- Assembly equivalent is too long for one slide

```c
//you need to use free() to release the memory when a node is removed
struct Node* removeNodeN(struct Node* root, int value) {
    if (root == NULL)
        return NULL;

    // b left
    if (value < root->value) {
        root->leftChild = removeNodeN(root->leftChild, value);
        return root;
    // b right
    } else if (value > root->value) {
        root->rightChild = removeNodeN(root->rightChild, value);
        return root;
    } else {

        // no children
        if (root->leftChild == NULL && root->rightChild == NULL) {
            free(root);
            root = NULL;

        // one child (right)
        } else if (root->leftChild == NULL) {
            struct Node* temp = root; // save current node as a backup
            root = root->rightChild;
            free(temp);

        // one child (left)
        } else if (root->rightChild == NULL) {
            struct Node* temp = root; // save current node as a backup
            root = root->leftChild;
            free(temp);

        // two children
        } else {
            struct Node* temp = findMin(root->rightChild); // find minimal value of right sub tree
            root->value = temp->value; // duplicate the node
            root->rightChild = removeNodeN(root->rightChild, temp->value); // delete the duplicate node
        }
    }


    return root; // parent node can update reference

}
```

Free

- I probably should have found correct syscall for free but this is what assignment says to do

```
# please implement this method
#
# free the memory block, here we can simply set
# all bits of the memory blocks to be zero
#
# input: $a0 the address of memory block
#    $a1 the size of memory block in words
free:
    # while ($a1--) *$a0++ = 0;
    j free_loop_cond
    free_loop:
        sw $zero, 0($a0)        # *$a0 = 0
        addi $a0, $a0, 4        # $a0 += sizeof(word)
        subi $a1, $a1, 1        # $a1--
    free_loop_cond:
        bnez $a1, free_loop
    jr $ra
```

Test cases

- I did at least two test cases for each subroutine to line up with what was required for lab8.c
- Additionally I added a test case for the findMin subroutine despite it not needing
- I observed that the memory in the heap which free was called on were overwritten with zeros. However I chose not to determine a way to print this out because if free used the correct syscalls, doing so would cause a segmentation fault
- As with my lab8.c I didn't add special test cases to my create tree subroutine as the only edge case is if the system is out of memory which isn't in the scope of my program