**Preface**

Almost done! The only thing left to make the game playable is a boss and a fight, and then from there we can add features as we see fit. This document lays out the bare bones for finishing the game, as well as a single nice feature that we will include. Once completed, you are free to add anything you would like to add!

**Function Prototypes**

You will implement the boss.c and parse.c files, as well as create and use the makefile. If you are going to add your own helper functions, make sure to include the prototypes for those functions in game.h.

**boss.c Functions**

**Boss\* loadBoss();**

**Input: None**

**Return: a fully imported boss**

This will be the controller for all the boss loading. The boss will have all the basic stats as if it’s a hero, but it will have a decision tree for its AI. This tree will have nodes that have a health floor and health ceiling. Having the boss’s current health lie in between those two values will determine if that node will be selected. Once the appropriate node is found, a pointer inside the node will lead to a linked list of decisions that the boss will take in order. See the sample output for a picture of this. Use the functions below to help complete the loading process for this function.

**ActionNode\* addActionToList(ActionNode\* , ActionNode\*);**

**Input: the head of the linked list and the node to be added to the linked list**

**Return: the head of the linked list**

This function is for the linked list that is connected to its corresponding tree node. Starting from the head of the list, it will tack on the new node at the end of the list each time it is called.

**DecisionTreeNode\* addNodeToTree(DecisionTreeNode\*, DecisionTreeNode\*);**

**Input: the root of the BST described above, and the node to be added to this tree**

**Return: the root of the BST**

This function is for dealing with each of the nodes once the linked list is built for it and attached. Add the node to the tree as the same way you have built BSTs before. Keep in mind that each node covers a range of values: if the node’s range is lower than the range of the root, put it on the left side of the tree. If the range is above the range of the root, put it on the right.

**ActionNode\* fetchNewList(Boss\*, DecisionTreeNode\*);**

**Input: the boss and the root of its BST**

**Return: the LINKED LIST that is found inside the node that has the health range that contains the current health**

As described, search for the node in the BST that has the health range that contains the boss’s current health. Once found, return the head of the linked list found inside the node.

**void freeBossTree(DecisionTreeNode\*);**

**Input: the root of the boss’s BST**

**Return: none**

Free the entire tree. Remember: freeing the nodes isn’t good enough. Each node leads to a linked list, so you have to free that first!

**void freeActionListInNode(ActionNode\*);**

**Input: the head of the linked list connected to the tree node**

**Return: none**

Call this function in freeBossTree(). This will be used in conjunction to free the linked list connected to the node before freeing the node itself.

**parse.c Functions**

Recall: the names of the shields and swords had periods where spaces would be. This made file reading easy. However, that isn’t too realistic, so we will use this opportunity to parse a file with desired formatting. This also gives us the opportunity to add a description to each item. Use the functions below to handle this parsing.

**void ParseShieldInfo(Shield\*, FILE\*);**

**void ParseSwordInfo(Sword\*, FILE\*);**

**Input: a (ALREADY MALLOCED) shield/sword, and the OPEN file pointer**

**Return: none**

These functions are identical except for one line: the defense value vs the attack value. The file pointer passed into the function is pointing at the next line in the file to be read. You will have to get that line and, from that, fill the shield/sword with its information. **DO NOT USE FSCANF!** A suggestion would be to start with fgets(), since you can specify the maximum number of characters to read from the line. This way, you can guarantee to read the entire line that you need to parse.

**Makefile**

You will also be adding to the makefile. You have two options on how to treat the makefile:

1. Using the preexisting makefile from HW2, add the necessary ingredients to make HW3 work. Next, you will need to describe the makefile using comments that you will add in the makefile itself. Describe in as much detail as possible what the makefile is doing, what each component of the makefile is and does, and why each component is important. This includes the targets, special/automatic variables, defined variables, dependencies, and commands.
2. Rewrite the makefile using no special/automatic variables. This will require laying out all the targets needed for HW2, as well as adding the ones needed for HW3. Consult your resources for details on what the special/automatic variables are.

The choice is yours.

**File Format**

**Items.txt:**

<Name of shield 1> | <price of shield> | <defense value> | <item description>

<Name of shield 2> | <price of shield> | <defense value> | <item description>

<Name of shield 3> | <price of shield> | <defense value> | <item description>

…

<Name of sword 1> | <price of sword> | <attack value> | <item description>

<Name of sword 2> | <price of sword> | <attack value> | <item description>

<Name of sword 3> | <price of sword> | <attack value> | <item description>

…

**Boss.txt:**

<Name> | <boss health> | <boss defense> | <boss attack> | <boss speed>

<node 1 health floor> | <node 1 health ceiling> | <number of actions the node’s linked list will have>

<action 1 in the linked list>

<action 2 in the linked list>

…

<node 2 health floor> | <node 2 health ceiling> | <number of actions the node’s linked list will have>

<action 1 in the linked list>

<action 2 in the linked list>

…

…

**Sample Output**

See the PowerPoint slides for pictures and notes.

**Grading Breakdown**

Your code must compile with no warnings and no errors. Failure to do so will result in a **ZERO**, no exceptions. Any global variables will also result in a **ZERO**, no exceptions.

**ANY** segmentation fault, infinite loop or other runtime error will result in a minimum deduction of 50%. This will be increased at your TA’s discretion. **We will be attempting to break your code, so make sure your code handles any corner cases that come with working with linked lists. TEST ALL THE POSSIBLE SCENARIOS.**

**(Potentially) Useful statistics from Daniel’s solution (# of lines of code):**

* **loadBoss(): 30**
* **addActionToList(): 6**
* **addNodeToTree(): 13**
* **fetchNewList(): 9**
* **freeBossTree() and freeActionListInNode(): 15**
* **Parsing: 30-35/function**

If you are heavily over these numbers, you might want to rethink your design. There’s not much to it!

**5 points: loadBoss()**

* Function is laid out appropriately to utilize each of the helper functions
* Function adheres to the file format described above

**5 points: addActionToList()**

* Function successfully adds nodes to the end of the linked list
* Function properly handles any special cases that may be present
* Use of an array for this will result in no credit for this function or the loadBoss function.

**10 points: addNodeToTree()**

* Function successfully adds nodes to the BST
* The BST adheres to the layout described
* Function properly handles any special cases that may be present

**10 points: fetchNewList()**

* Function properly traverses the BST as described
* Function returns the correct pointer requested

**10 points: freeBossTree() and freeActionListInNode()**

* Function frees all pieces of the BST, including the linked lists connected to the nodes
* Failure to clean up any piece of memory will result in -8 points minimum

**10 points: ParseShieldInfo() and ParseSwordInfo()**

* Function successfully fills the structures passed to it
* Function adheres to the guildelines described above

**10 points: Makefile**

* Appropriately pick one of the options given
* (Option 1) describe each of the pieces talked about in the instructions
* (Option 2) create the equivalent of the makefile in HW2, leaving nothing out, and adding the required pieces

Failure to follow these guidelines will result in deductions (At your TA’s discretion):

* All prewritten code, prototypes and files given to you are unmodified (any tampering with prewritten code without Daniel’s consent = major point deductions). If you do not follow the prototypes you are told to use, you will receive a zero.
* Code is formatted in an acceptable and readable manner
* Comments are present explaining logical reasoning (HINT: more meaningful/thoughtful comments = more likely to get partial credit)