Evil Hangman

This lab assumes that you have completed all steps involved in lab1 through 10 and provides hints and targets for finishing more of the lab given in lab9.

We will be working on the last part of the game in this week.

In this lab, you need to build a binary search tree that would be used a dictionary to do two things:

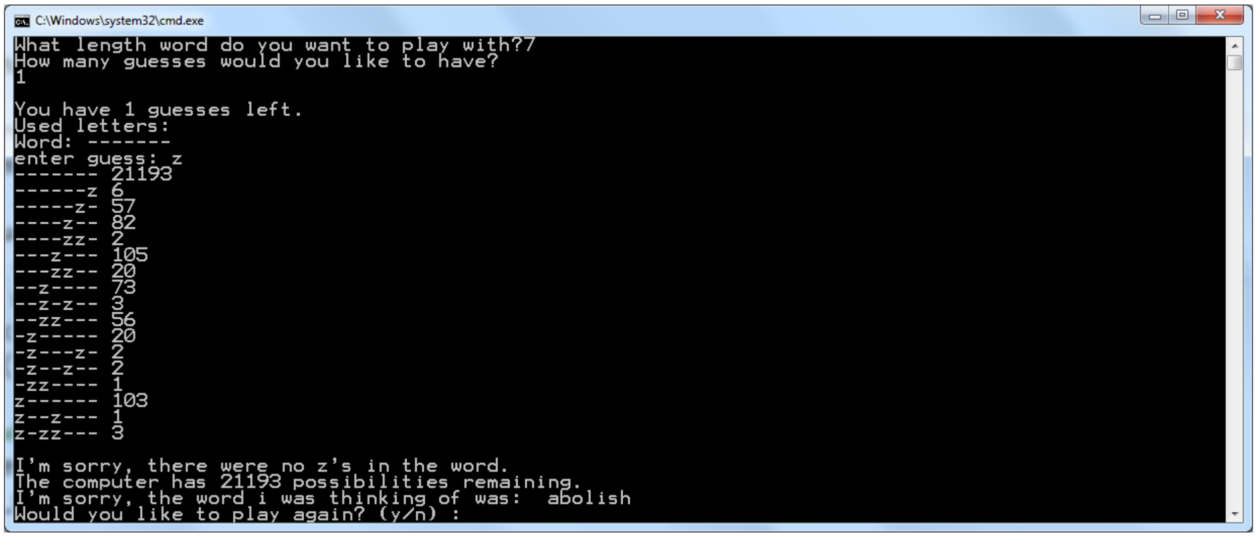
- First, holds all the possible key-families given a certain guess
- Second, search through the tree to get the key-family that contains the largest number of words.

Here is how a single node inside the BinarySearchTree looks like:

```
struct binarynode {

  String_Ptr key;
  Vector_String_Ptr words;

};

typedef       struct binarynode BinaryNode, *BinaryNode_Ptr;
```

Each **BinaryNode** holds a **key,** which is the family key the words belong too given a certain letter, e.g., **-----,** and **words** is a vector of strings holding all the words belonging to this key.

Let's examine a sample play of the game.



```
C:\Windows\system32\cmd.exe
What length word do you want to play with?7
How many guesses would you like to have?
1

You have 1 guesses left.
Used letters:
Word: -------
enter guess: z
------- 21193
------z 6
-----z- 57
----z-- 82
----zz- 2
---z--- 105
---zz-- 20
--z---- 73
--z-z-- 3
--zz--- 56
-z----- 20
-z---z- 2
-z--z-- 2
-zz---- 1
z------ 103
z--z--- 1
z-zz--- 3

I'm sorry, there were no z's in the word.
The computer has 21193 possibilities remaining.
I'm sorry, the word i was thinking of was:  abolish
Would you like to play again? (y/n) :
```

In this example, the user has selected to play the game with a word that is 7 letters long and they want to have only 1 guess. The user selects the letter 'z' as their guess. Behind the scenes, the program reads through every 7 letters word in the dictionary and examines them individually. Each word can be sorted into a family of words that all have a similar pattern. Using **get_word_key_value** function that was created last week, you can decide the family for each word based on the given guess. Afterwards, based on the **new_key,** you need to traverse the binary search tree and add the **word** to the node that has the same **family key as the word.** For example, there are 221193 words in the dictionary that are 7 letters long and do not contain any occurrence of the letter ‘z’. There are 6 words that end in a single ‘z’, 57 words that have a single ‘z’ as the next to last character and so on.

Here is the interface for the whole binary search tree (You need to implement all of the given functions):

```
struct binarysearchtree {
  BinaryNode_Ptr root;
};

typedef struct binarysearchtree BinarySearchTree, * BinarySearchTree_Ptr;

// Precondition: none
// Postcondition: returns a handle to a binary search tree
// The function allocates memory for a BinarySearchTree_Ptr
// and sets the root node to null
BinarySearchTree_Ptr binary_search_tree_init_default();

// Precondition: Key is the key to be stored inside the node
// Postcondition: a BinaryNode is created with the given key
// The function copies the given key into the key inside the
// the node. It also calls vector_string_init_default
// to create an empty vector
BinaryNode_Ptr binary_node_init_key(String_Ptr key);

// Precondition: hTree is a handle to a binary search tree
// word: is the word to be added to the tree
// key: is the family key to the word. The node will be picked
// based on the key value. The word is inserted in the corresponding
// array in the same node found by the key.
// Postcondition: the word is added to the correct node or a new
// node is created if the key for the word does not exist
// The function does that in two steps:
// - First: call binary_search_tree_search. If the returned node is
// - not null, then the word can be added to the corresponding node
// - Second: if the node returned from the search is null, then call
// - binary_search_tree_insert_node to add a node to the tree with the
// - given key
Status binary_search_tree_insert_word(BinarySearchTree_Ptr hTree, String_Ptr word,
String_Ptr key);

// Precondition: hTree a handle to a tree that would be searched
// key: is the key used in the searching process
// Postcondition: return a pointer to the BinaryNode if found, otherwise,
// return NULL
BinaryNode_Ptr binary_search_tree_search(BinarySearchTree_Ptr hTree, String_Ptr key);

// Precondition: hTree is a handle to a tree to add a node to
// Key is the key used to decide where to add a node
// Postcondition: create a new node and add it to the tree based on the key
// Returns a pointer to the new added node
BinaryNode_Ptr binary_search__tree_insert_node(BinarySearchTree_Ptr hTree, String_Ptr
key);

// Precondition: hTree is a handle to a binary tree to be searched for
// the given key. hVector is an empty vector.
// Postcondition: returns Success of Failure based on the swapping operation
// The function does the following:
// - It searches for the node that has the given key.
// - Takes an empty string vector (it is allocated in memory but contains no words
inside)
// - Then it swaps the address of the vector_string inside the found node with the
given one
// Allowing you to destroy the tree after wards without any problem
Status binary_search_tree_swap(BinarySearchTree_Ptr hTree, String_Ptr key,
Vector_String_Ptr * phVector);

// Precondition: hTree is a handle to a binary search tree that would be
// traversed
// Postcondition: returns a pointer to the node with the largest number of
// words
BinarNode_Ptr binary_search_tree_find_max(BinarySearchTree_Ptr hTree);

// Precondition: phTree is a pointer to a handle to a tree that needs to be
// destroyed
// Postcondition: destroies the tree including the nodes
void binary_search_tree_destroy(BinarySearchTree_Ptr * phTree);

// Precondition: phNode is a node to destroy
// Postcondition: destroying the node include the key and
// the words vector inside the node
void binary_node_destroy(BinaryNode_Ptr * phNode);
```

After creating the binary search tree that holds all the key families for the words, you need to traverse the tree to find the node with the largest words. After finding this node, you need to copy the vector inside this node into an outside vector (using the **binary_search_tree_swap function**). Afterwards, destroy the whole tree and show the user the new word family.

TA CHECKPOINT 1: Demonstrate to your TA the size of each word family based on the given guess (similar to the example given above)

You need to keep doing the above until the user runs out of guesses.

For each new guess, you need to run through all the words remaining in the vector of strings and create a new binary search tree.

TA CHECKPOINT 2: Demonstrate to your TA the whole game running.