30/41

```
//Will Foley & Ryan Fu
//CS 241 Fall 2014
//driver program

#include <iostream>
#include <string>
#include "ConsoleInterface.h"

using namespace std;


int main()
{
        ConsoleInterface a;
        string play_again;
        do
        {
                a.initialize();         //will begin a game of Hangman
                do
                {
                        cout << "Would you like to play again?  y - yes or n - no:";
                        cin >> play_again;
                }
                while(play_again != "n" && play_again != "y");
        }while (play_again != "n");     //will play again if the user wants to otherwise it will just close.


        return 0;
}

```

Passes tests

```
//Will Foley & Ryan Fu
//LCMap

#ifndef LCMAP_H
#define LCMAP_H

#include <iostream>
#include <list>

using namespace std;

template <typename K,typename V, typename Comparator=less<K> >
class LCMap
{
  private:
        struct Node
        {
          K key_;
          V value_;
          Node* left_;
          Node* right_;
        };
        Node* root_;
        Comparator comp_;

        //find smallest node within the tree recursive helper
        Node** findMin(Node** n)
        {

                if ((*n)->left_ != NULL)        //if the next left node is not Null then you need to go next again
                        n = findMin(&((*n)->left_));
                return n;
        }

        //find if a node exists with a specific key recursive helper
        bool find_inner(Node* root, K key)
         {
                if (root != NULL )
                {
                        if  (comp_(key , root->key_))           //if the key is less than go left
                                return find_inner(root->left_,key);
                        else if (comp_(root->key_, key))        //otherwise go right
                                return find_inner(root->right_,key);
                }

                return root != NULL;        //if it is no less than or greater then you should have found it
                                                                        //otherwise it does not exist within the tree
        }

        //recursive helper for insert
        bool insert(Node*& node, const K& key, const V& value)
        {
                if (node!=NULL)                                 //goes through until it has found the proper spot for the node
                {
                        if  (comp_(key, node ->key_))
                                insert(node->left_,key , value);

                        else if (comp_(node->key_, key))
```

```
                                insert(node->right_,key , value);
                }
                else                            //and then puts it there
                {
                        node = new Node;
                        node->key_ = key;
                        node->value_ = value;
                        node-> right_=NULL;
                        node-> left_=NULL;
                }

                return node == NULL;
        }

        //remove node with 0 or 1 child
        void removeNodeSimple(Node*& n)
        {
                Node* tmp = n;          //have it point to the tmp

                //then point it to the left or the right node depending which side the child is on.
                if (n->left_== NULL)
                        n = n->right_;
                else if(n->right_ == NULL)
                        n = n->left_;

                delete tmp;     //then delete it
                                                //this will also just delete a leaf
        }

        //delete node recursive helper
        bool deleteNode (Node*& n, const K& key)
        {
                if (n==NULL)                    //if the node does not exist
                        return false;
                else    //otherwise go left or right
                {
                        if(comp_(key, n->key_) )
                                 deleteNode(n->left_, key);
                        else if(comp_(n->key_, key) )
                                 deleteNode(n->right_, key);
                        else if (n->left_!=NULL && n->right_!=NULL)             //if you found the node and if it has two childeren
                        {

                                Node** tmp = findMin(&n->right_);               //find the smallest node in the right subtree
                                n->key_ = (*tmp)->key_;                 //and move the data from that node to this current node
                                n->value_ = (*tmp)->value_;
                                removeNodeSimple(*tmp);         //then delete the node where you got the info from.
                        }
                        else
                                removeNodeSimple(n);    //otherwise just delete the node

                        return true;
                }
        }

        //deletes all nodes in the tree
        void clearHelper(Node *root)
        {
```

-3

-3

```
                if(root!=NULL)
                {
                        clearHelper(root->left_);                       //L
                        clearHelper(root->right_);                      //R
                        delete root;                             //V
                }
        }

        //get the size of the tree
        int Getsize(Node* n)  const
        {
          if (n==NULL)
                return 0;
          else
                return(Getsize(n->left_) + 1 + Getsize(n->right_));
        }

        //preform an inorder traversal of the tree and make a list of all the keys
        void inOrder(Node* n,list<K>& keyList)
        {
                if (n!=NULL)
                {
                        inOrder(n->left_,keyList);
                        keyList.push_back(n->key_);
                        inOrder(n->right_,keyList);

                }
        }

        //copy helper
        void copy(Node * n)
        {
                 if(n != NULL)
                 {
                         insert(n->key_,n->value_);    inefficient
                         copy(n->left_);
                         copy(n->right_);
                 }
        }
  public:
        /* constructor */
        LCMap(Comparator c=Comparator())
        {
                comp_=c;
                root_=NULL;
        }

        /* copy constructor */
        LCMap(const LCMap<K,V,Comparator>& orig)
        {
                root_=NULL;
                copy(orig.root_);       //calling the copy helper with orig
        }
        /* cleans up all memory for stroage and calls the destructor for the keys and values stored */
        virtual ~LCMap()
        {
                clear();        //recursive helper

        }

        /* assignment operator*/
        LCMap<K,V,Comparator>& operator =(const LCMap<K,V,Comparator>& rhs)
        {

                this->clear();  //clear out current tree
                root_=NULL;
                copy(rhs.root_);        //then copy everything into tree

                return *this;
        }

        /* inserts the key vaule pair referenced by key
                returns true if successful */
        bool insert(const K& key, const V& value)
        {
                return  insert(root_, key, value);      //call insert helper function
        }

        /* erase key vaule pair referenced by key
                return true if successful */
        bool erase(const K& key)
        {
                return deleteNode(root_,key);
        }

        /* lookup the value associated with a key. if the key is not in the
        map, insert it with default value. Should provide l-value access to
        value.*/
        V& operator[](const K& k)
        {
                V tmp;
                if (!in(k))             //if the key does not exist
                {
                        insert(k,tmp);  //insert it with default value
                }
                Node* n = root_;
                while(comp_(k ,n->key_) || comp_(n->key_,k ))   //find the node with the key
                {
                        if (comp_(k ,n->key_))
                                n = n->left_;
                        else if (comp_(n->key_,k ))
                                n = n->right_;
                }

                return n->value_;       //and return its value
        }

        /* return true if this key maps to a value */
        bool in(const K& key)
        {
                Node* root = root_;
                return find_inner(root,key);    //calls recursive helper
        }



        /* return a list of keys in this map */
        list<K> keys()
        {
                list<K> keyList_;
                inOrder(root_,keyList_);        //calls recursive helper
                return keyList_;
        }

```

always 2 lookups, figure out how to do it in one

-2

-2

```
        /* return true if the map is empty */
        bool empty() const
        {
                return (this->size()) ==0;
        }

        /* number of key value pairs stored */
        int size() const
        {
                return Getsize(root_);  //calls recursive helper
        }

        /* empties the map */
        void clear()
        {
                clearHelper(root_);                     //calls recursive helper
                root_=NULL;
        }



};
#endif
```

size = 0

write a single lookup function for insert, erase, in, and [] to share

-4

-4

```
//Will Foley & Ryan Fu
//Hangman class

#ifndef HANGMAN_H
#define HANGMAN_H

#include <iostream>
#include <vector>
#include "LCMap.h"


using namespace std;

class Hangman
{
public:
        Hangman(int guesses, int length, vector<string> words, char var);
        void adjustWordList();
        void getLetter();
        void partitionWords();
        void chooseLargest();
        void results();
        string generateKey(int cursor);
        bool newKey(string key);
        bool lost();
        bool won();
        bool gameComplete();
        void print();

private:
        const static int LWR_ASCII = 97;
        const static int UPR_ASCII = 122;
        int guesses_;
        int length_;
        bool word_number;
        ~~LCMap<string,~~ vector<string> ~~>~~ possible_words_;   //map of family of words
        ~~vector<string> keys_;~~
   set  ~~string~~ letters_used_;           //string of letters used
        ~~string current_word_;~~           //word being guessed
        ~~string current_key_;~~            //key to the largest group of words
        char current_letter_;           //letter that was guessed
        int win_count_;
};

#endif
```

```
//Will Foley & Ryan Fu
//Hangman functions

#include "Hangman.h"

//constructor
Hangman::Hangman(int guesses, int length, vector<string> words, char var)
{
        guesses_ = guesses;
        length_ = length;
        string key;
        int i = 0;
        while (i < length_)             //creates the word being guessed and the first key consisting of - ’s
        {
                key.push_back(’-’);
                current_word_.push_back(’-’);
                ++i;
        }
        current_key_ = key;
        possible_words_[key] = words;   //assigning the key to have the list of words as its value
        if (var != ’y’)
                word_number = false;
        win_count_ = 0;         //used to determine if the user has won
}

//organizes the words into families
void Hangman::partitionWords()
{
        string key;
        LCMap<string, vector<string> > newMap;

        for (unsigned int x =0; x < (possible_words_[current_key_]).size(); ++x)
        {
                key = generateKey(x);   //generate key for word
                (newMap[key]).push_back((possible_words_[current_key_])[x]); //add that the word from the current list of words to the new one
                if (newKey(key))
                        keys_.push_back(key);
        }
        possible_words_ = newMap;       //replace new word list with old one
        ~~newMap.clear();~~
}

//checks to see if the key is a new key
bool Hangman::newKey(string key)
{
        size_t i = 0;
        string match;
        while (match != key && i < keys_.size())
        {
                match = keys_[i];
                ++i;
        }
        return match != key;
}

//chooses largest family of words
void Hangman::chooseLargest()
{
        partitionWords();       //first group them up
        int size;
```

```cpp
        int max = 0;
        for (size_t i =0; i < keys_.size(); ++i)        //go through with key list and find the largest
        {
                size = (possible_words_[(keys_[i])]).size();
                if (size > max)
                {
                        current_key_ = keys_[i];
                        max = size;
                }
        }
        keys_.erase(keys_.begin(), keys_.end());        //erase the list of keys
}

//calculate results and make changes to word being guessed
void Hangman::results()
{
        int changes = 0;
        for(unsigned int i = 0; i < current_word_.size(); ++i)
        {
                if ((((possible_words_[current_key_])[0])[i]) == current_letter_)       //if the letter being guessed is in the family
                {                                                                       //of words
                        current_word_[i] = current_letter_;             //modify the word
                        ++changes;
                        ++win_count_;
                }
        }
        if (changes == 0)       //if there are no changes then subtract from guesses
                --guesses_;
}

//generates a key based on the location of the letter
string Hangman::generateKey(int cursor)
{
        string key;
        for(int i = 0; i < length_; ++i)
        {
                if (((possible_words_[current_key_])[cursor])[i] == current_letter_)
                        key.push_back(current_letter_);                         //if the letter is in the word then put add it to the key
                else
                        key.push_back('-');             //otherwise just add -
        }
        return key;
}

//gets a letter from the user
void Hangman::getLetter()
{
        int letter;
        size_t pos = string::npos;      //npos used to determine if the letter has been used
        string input;
        do
        {
                cout << "Please enter in a lower case letter: ";
                cin >> input;
                letter = input[0];
                pos = letters_used_.find_first_of(input[0]);    //looking for the position of input character
                if (pos != string::npos)
                        cout << "You have already used that letter." << endl;
        }while((letter < LWR_ASCII || letter > UPR_ASCII) || pos != string::npos || input.size() != 1);
                                                                //while its a lowercase letter or hasnt been used or the input was greater than 1
                                                                                       //get another
        current_letter_ = input[0];
        letters_used_.push_back(current_letter_);

}

//outputting to the screen
void Hangman::print()
{
        cout << "You have " << guesses_ << " remaining." << endl;               //guesses
        cout << "You have used these letters: ";
        for (size_t i = 0; i < letters_used_.size(); ++i)                                      //letters used
                cout << letters_used_[i] << " ";
        cout << endl;
        cout << "word: " << current_word_ << endl;                                                     //current word
        if (word_number)                                                                                                       //number of words used
                cout << "There are " << (possible_words_[current_key_]).size() << " possible words." << endl;
        cout << endl;
}

//determines if game has been lost
bool Hangman::lost()
{
        if (guesses_ == 0)
        {
                cout << "You have lost.  The word was " << (possible_words_[current_key_])[0] << endl;
                keys_.erase(keys_.begin(), keys_.end());
                possible_words_.clear();
        }
        return guesses_ == 0;
}

//determines if game has been won
bool Hangman::won()
{

        if (win_count_ == length_)
        {
                cout << "You have won" << endl;
                cout << "The word was: " << current_word_ << endl;
        }
        return win_count_ == length_;
}

//determines if game is complete
bool Hangman::gameComplete()
{
        return won() || lost();
}
```

```cpp
//Will Foley & Ryan Fu
//Console Interface header file
//controls the game of hangman and sets it up

#ifndef CONSOLEINTERFACE_H
#define CONSOLEINTERFACE_H

#include <iostream>
#include <fstream>
#include <string>
#include <stdio.h>
#include <ctype.h>
#include <cstdio>
#include <stdlib.h>
#include <cstring>
#include <vector>
#include "Hangman.h"
using namespace std;

class ConsoleInterface
{
public:
        ConsoleInterface(){}
        void initialize();
        int get_word_length();
        void play();
private:
        vector<string> words_; //list of words to be passed into the hangman game
        int guesses_;
        int length_;
};

#endif
```

```cpp
//Will Foley & Rayan Fu
//Console Interface functions

#include "ConsoleInterface.h"

//gets the number of guesses you would like to have
void ConsoleInterface::initialize()
{
        guesses_=0;
        string guesses;
        length_ = get_word_length();    //getting word length
        unsigned int i = 0;
        while(guesses_ <=  0)   //loop to make sure that the user enters in a number
        {
                do
                {
                        cout << "Please enter in the number of guesses you would like to have: ";
                        cin >> guesses;
                        i = 0;
                        char temp = guesses[i];
                        while (isdigit(temp) && i < guesses.size()-1)   //check each character in the string for a digit
                        {
                                ++i;
                                temp = guesses[i];
                        }
                }while (!(isdigit(guesses[i])));        //if the current character isn’t a digit get a new one

                // CONVERTING STRING TO INT  //
                char* guess_char = new char[guesses.size()+1];
                strcpy(guess_char, guesses.c_str());
                guesses_ = atoi(guess_char);
                delete [] guess_char;
        }
        i = 0;

        play();         //plays a game
}

//plays a game of hangman
void ConsoleInterface::play()
{
        char var;
        do
        {
                cout << "Would you like to to see the number of possible words.  Enter y or n. ";
                cin >> var;
        }while (var != ’n’ && var != ’y’);
        Hangman game(guesses_, length_, words_, tolower(var));          //constructing a game of hangman
        do
        {
                game.print();           //outputs info to screen
                game.getLetter();       //getting a letter from the user
                game.chooseLargest();   //choose the largest family of words
                game.results();         //calculate the results with the largest family
        }while(!game.gameComplete());

        words_.erase(words_.begin(), words_.end());
}
```

```

//getting the length of the word the user wants to guess and creates a list of words
int ConsoleInterface::get_word_length()
{
        string word;
        int size;
        ifstream file;
        string length;
        unsigned int i = 0;
        int lngt;
        do
        {
                i = 0;
                file.open ("lexicon.txt");
                if (file.is_open())
                {
                        cout << "Please enter in an acceptable word length: ";
                        cin >> length;

                        while(i < length.size() && isdigit(length[i]) ) //checking to make sure that the input is a number
                        {
                                ++i;
                        }
                        if (i == length.size())
                                --i;
                        if (isdigit(length[i]))
                        {
                                //converting string to int
                                char* guesses_char = new char[length.size()+1];
                                strcpy(guesses_char, length.c_str());
                                lngt = atoi(guesses_char);

                                //this will go through the file of words and add any words of that length and push it into a vector
                                while (file.good())
                                {
                                        file >> word;
                                        size = word.size();
                                        if (size == lngt)
                                                words_.push_back(word);
                                }
                        }
                }
                file.close();
        }while(!isdigit(length[i]) || words_.empty());  //while there is a non digit in the input or there are no words in the vector
        return lngt;
}
```

read the file once, storing a map of word lengths to a list of words