The main test I did was testing if shortest word ladder function worked in a new test.c file. I altered the make file so that it ran and compiled test.c instead of main.c

```
build_test:
	rm -f test.exe
	gcc tests/test.c -o test.exe

run_test:
	./test.exe
```

Then i had boolean return test cases to check each functio if based on hardcoded inputs if they were right. The functions that use and implement freeladder was tested using valgrind to check if any memory errors happened. If there were none then i assumed it was working properly.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>

typedef struct WordNode_struct {
    char* myWord;
    struct WordNode_struct* next;
} WordNode;

typedef struct LadderNode_struct {
    WordNode* topWord;
    struct LadderNode_struct* next;
} LadderNode;

int countWordsOfLength(char* filename, int wordSize) {
    //---------------------------------------------------------
    // TODO - write countWordsOfLength()
    //      open a file with name <filename> and count the
    //      number of words in the file that are exactly
    //      <wordSize> letters long, where a "word" is ANY set
    //      of characters that falls between two whitespaces
    //      (or tabs, or newlines, etc.).
    //          return the count, if filename is valid
    //          return -1 if the file cannot be opened
    //---------------------------------------------------------
    FILE* infile = fopen(filename, "r");
    if(infile == NULL){
        return -1; //Invaild filename
    }

    int count = 0;
    char word[100];

    while(fscanf(infile, "%s", word) == 1){
        if(strlen(word) == wordSize){ //count the size of the word
            count++;
        }
    }

    fclose(infile);
    return count; //modify this
}

bool test_countWordsOfLength(){
    int count = countWordsOfLength("tests/test.txt", 3);
    // printf("%d", count);
    if(count != 4){
        return false;
    }
    else{
        return true;
    }
}

bool buildWordArray(char* filename, char** words, int numWords, int
wordSize) {
    //---------------------------------------------------------
    // TODO - write buildWordArray()
    //      open a file with name <filename> and fill the
    //      pre-allocated word array <words> with only words
    //      that are exactly <wordSize> letters long;
    //      the file should contain exactly <numWords> words
    //      that are the correct number of letters; thus,
    //      <words> is pre-allocated as <numWords> char* ptrs,
    //      each pointing to a C-string of legnth <wordSize>+1;
    //          return true iff the file is opened successfully
    //                      AND the file contains exactly
    //                      <numWords> words of exactly
    //                      <wordSize> letters, and those words
    //                      are stored in the <words> array
    //          return false otherwise
    //---------------------------------------------------------

    FILE* infile = fopen(filename, "r");
    if(infile == NULL || numWords != countWordsOfLength(filename,
wordSize)){ //check file and numwords
        return false;
    }

    char word[100];
    int index = 0;
    while(fscanf(infile, "%s", word) == 1){
        if(strlen(word) == wordSize){ //insert the word
            strcpy(words[index], word);
            index++;
        }
    }

    fclose(infile);

    return true;
}

// bool test_buildWordArray(){
//     int numWords = 3;
//     int wordSize = 3;
//     char *words[numWords];
//     for (int i = 0; i < numWords; i++) {
//         words[i] = malloc((wordSize + 1) * sizeof(char));
//     }

//     bool success = buildWordArray("tests/test.txt", words, numWords,
wordSize);
```

```
//     if (!success || strcmp(words[0], "cat") != 0 || strcmp(words[1],
"dog") != 0 || strcmp(words[2], "bat") != 0) {
//         return false;
//     }

//     freeWords(*words, numWords);
//     remove("testfile.txt");
//     return true;
// }


int findWord(char** words, char* aWord, int loInd, int hiInd) {
    //---------------------------------------------------------
    // TODO - write findWord()
    //          binary search for string <aWord> in an
    //          alphabetically sorted array of strings <words>,
    //          only between <loInd> & <hiInd>
    //              return index of <aWord> if found
    //              return -1 if not found b/w loInd & hiInd
    //---------------------------------------------------------

    if(loInd > hiInd){
        return -1; //word not found
    }

    int middleIndex = (loInd + hiInd) / 2;
    if(strcmp(words[middleIndex], aWord) == 0){
        return middleIndex; //word found
    }
    else if(strcmp(words[middleIndex], aWord) > 0){
        return findWord(words, aWord, loInd, middleIndex - 1);
//recursively search left half
    }
    else{
        return findWord(words, aWord, middleIndex + 1, hiInd);
//recursively search right half
    }

}
```

```
bool test_findWord() {
    char *words[] = {"apple", "banana", "cherry", "date", "fig"};
    int foundIndex = findWord(words, "cherry", 0, 4);
    if (foundIndex != 2) {
        return false;
    }
    foundIndex = findWord(words, "kiwi", 0, 4);
    if (foundIndex != -1) {
        return false;
    }

    return true;
}


void freeWords(char** words, int numWords) {
    //---------------------------------------------------------
    // TODO - write freeWords()
    //          free up all heap-allocated space for <words>,
    //          which is an array of <numWords> C-strings
    //           - free the space allocated for each C-string
    //           - then, free the space allocated for the array
    //                  of pointers, <words>, itself
    //---------------------------------------------------------

    for(int i = 0; i < numWords; i++){
        free(words[i]);
    }
    free(words);
}

void insertWordAtFront(WordNode** ladder, char* newWord) {
    //---------------------------------------------------------
    // TODO - write insertWordAtFront()
    //          allocate space for a new [WordNode], set its
    //          [myWord] subitem using <newWord> and insert
    //          it to the front of <ladder>, which is a
    //          pointer-passed-by-pointer as the head node of
    //          ladder changes inside this function;
    //          <newWord> is a pointer to a C-string from the
    //          full word array, already heap-allocated
    //---------------------------------------------------------
    WordNode* newNode = (WordNode*)malloc(sizeof(WordNode));
    newNode->myWord = newWord;
    newNode->next = *ladder; //next points to head node

    *ladder = newNode; //head node updated

}

bool test_insertWordAtFront() {
    WordNode *ladder = NULL;
    insertWordAtFront(&ladder, "hello");
    insertWordAtFront(&ladder, "world");

    if (strcmp(ladder->myWord, "world") != 0 ||
strcmp(ladder->next->myWord, "hello") != 0) {
        return false;
    }

    freeLadder(ladder);
    return true;
}

int getLadderHeight(WordNode* ladder) {
    //---------------------------------------------------------
    // TODO - write getLadderHeight()
    //          find & return number of words in <ladder> list
    //---------------------------------------------------------
    int numWords = 0;
    WordNode* current = ladder;
    while(current != NULL){
        numWords++;
        current = current->next; //increemnt height counter until next
pointer is null
    }
    return numWords; // modify this line
}

bool test_getLadderHeight() {
    WordNode *ladder = NULL;
    insertWordAtFront(&ladder, "one");
    insertWordAtFront(&ladder, "two");
    insertWordAtFront(&ladder, "three");

    int height = getLadderHeight(ladder);
    if (height != 3) {
        return false;
    }

    freeLadder(ladder);
    return true;
}


WordNode* copyLadder(WordNode* ladder) {
    //---------------------------------------------------------
    // TODO - write copyLadder()
    //          make a complete copy of <ladder> and return it;
    //          the copy ladder must have new space allocated
    //          for each [WordNode] in <ladder>, BUT the
    //          C-string pointers to elements of the full word
    //          array can be reused; i.e. the actual words do
    //          NOT need another allocation here
    //---------------------------------------------------------

    if(ladder == NULL){
        return NULL;
    }

    WordNode* copyLadder = (WordNode*)malloc(sizeof(WordNode));
    copyLadder->myWord = ladder->myWord;
    copyLadder->next = NULL; //copy head node

    WordNode* temp = ladder->next;
    WordNode* temp2 = copyLadder;
    while(temp != NULL){
        WordNode* newNode = (WordNode*)malloc(sizeof(WordNode)); //deep
copy of wordnode and shallow copy of myword
        newNode->myWord = temp->myWord;
        temp2->next = newNode;
        newNode->next = NULL;

        temp2 = newNode;
        temp = temp->next;
    }
    return copyLadder; //modify this
}

bool test_copyLadder() {
    WordNode *ladder = NULL;
    insertWordAtFront(&ladder, "one");
    insertWordAtFront(&ladder, "two");

    WordNode *copiedLadder = copyLadder(ladder);

    if (strcmp(copiedLadder->myWord, "two") != 0 ||
strcmp(copiedLadder->next->myWord, "one") != 0) {
        return false;
    }

    freeLadder(ladder);
    freeLadder(copiedLadder);
    return true;
}


void freeLadder(WordNode* ladder) {
    //---------------------------------------------------------
    // TODO - write freeLadder()
    //          free up all heap-allocated space for <ladder>;
    //          this does NOT include the actual words,
    //          instead just free up the space that was
    //          allocated for each [WordNode]
    //---------------------------------------------------------
    WordNode* freeNode = ladder;
    while(freeNode != NULL){
        WordNode* next = freeNode->next;
        free(freeNode);
        freeNode = next;
    }
}

void insertLadderAtBack(LadderNode** list, WordNode* newLadder) {
    //---------------------------------------------------------
    // TODO - write insertLadderAtBack()
    //          allocate space for a new [LadderNode], set its
    //          [topWord] subitem using <newLadder>; then, find
    //          the back of <list> and append the newly created
    //          [LadderNode] to the back; Note that <list> is a
    //          pointer-passed-by-pointer, since this function
    //          must handle the edge case where <list> is empty
    //          and the new [LadderNode] becomes the head node
    //---------------------------------------------------------

    LadderNode* newLadderNode = (LadderNode*)malloc(sizeof(LadderNode));
    newLadderNode->topWord = newLadder;
    newLadderNode->next = NULL; //create new tail node

    if(*list == NULL){
        *list = newLadderNode;
        return;
    }

    LadderNode* current = *list;
    while(current->next != NULL){
        current = current->next; //traverse the list
    }

    current->next = newLadderNode; //insert node at end
}

bool test_insertLadderAtBack() {
    LadderNode *list = NULL;
    WordNode *ladder1 = NULL;
    insertWordAtFront(&ladder1, "ladder1");

    insertLadderAtBack(&list, ladder1);

    WordNode *ladder2 = NULL;
    insertWordAtFront(&ladder2, "ladder2");
    insertLadderAtBack(&list, ladder2);

    if (strcmp(list->topWord->myWord, "ladder1") != 0 ||
strcmp(list->next->topWord->myWord, "ladder2") != 0) {
        return false;
    }

    freeLadderList(list);
    return true;
}


WordNode* popLadderFromFront(LadderNode** list) {
    //---------------------------------------------------------
    // TODO - write popLadderFromFront()
    //          pop the first ladder from the front of the list
    //          by returning the pointer to the head node of
    //          the ladder that is the subitem of the head node
    //          of <list> AND updating the head node of <list>
    //          to the next [LadderNode]; Note that <list> is a
    //          pointer-passed-by-pointer, since this function
    //          updates the head node to be one down the list;
    //          the [LadderNode] popped off the front must have
    //          its memory freed in this function, since it
    //          will go out of scope, but the ladder itself,
    //          i.e. the head [WordNode], should NOT be freed.
    //---------------------------------------------------------
    LadderNode* freeNode = *list;
    *list = (*list)->next; //remove front

    WordNode* oldFrontWord = freeNode->topWord;
    free(freeNode); //free the old front

    return oldFrontWord; //modify this
}

bool test_popLadderFromFront() {
    LadderNode *list = NULL;
    WordNode *ladder1 = NULL;
    insertWordAtFront(&ladder1, "ladder1");

    insertLadderAtBack(&list, ladder1);

    WordNode *ladder2 = NULL;
    insertWordAtFront(&ladder2, "ladder2");
    insertLadderAtBack(&list, ladder2);

    WordNode *poppedLadder = popLadderFromFront(&list);
    if (strcmp(poppedLadder->myWord, "ladder1") != 0 ||
strcmp(list->topWord->myWord, "ladder2") != 0) {
        return false;
    }

    freeLadder(poppedLadder);
    freeLadderList(list);
    return true;
}


void freeLadderList(LadderNode* myList) {
    //---------------------------------------------------------
    // TODO - write freeLadderList()
    //          free up all heap-allocated space for <myList>;
    //          for each ladder in <myList>:
    //           - free the space allocated for each [WordNode]
    //                  in the ladder using freeLadder()
    //           - then, free the space allocated for the
    //                  [LadderNode] itself
    //---------------------------------------------------------
    LadderNode* freeNode = myList;
    while(freeNode != NULL){
        LadderNode* next = freeNode->next;

        freeLadder(freeNode->topWord);
        free(freeNode);

        freeNode = next;
    }
}
```

```
// checks if 2 words have only 1 difference
// assumes both words are same size
bool checkWord(char* x, char* y){
    int len = strlen(x);
    int count = 0;

    for(int i = 0; i < len; i++){
        if(x[i] != y[i]){
            count++;
        }
    }

    return (count == 1);
}

WordNode* findShortestWordLadder(   char** words,
                                    bool* usedWord,
                                    int numWords,
                                    int wordSize,
                                    char* startWord,
                                    char* finalWord ) {
    //---------------------------------------------------------
    // TODO - write findShortestWordLadder()
    //          run algorithm to find the shortest word ladder
    //          from <startWord> to <finalWord> in the <words>
    //          word array, where each word is <wordSize> long
    //          and there are <numWords> total words;
    //          <usedWord> also has size <numWords>, such that
    //          usedWord[i] is only true if words[i] has
    //          previously be entered into a ladder, and should
    //          therefore not be added to any other ladders;
    //          the algorithm creates partial word ladders,
    //          which are [WordNode] linked lists, and stores
    //          them in a [LadderNode] linked list.
    //              return a pointer to the shortest ladder;
    //              return NULL if no ladder is possible;
    //                  before return, free all heap-allocated
    //                  memory that is created here that is not
    //                  the returned ladder
    //----------------------------------------------------------
    LadderNode* ladderList = NULL;
    WordNode* firstladder = NULL;

    insertWordAtFront(&firstladder, startWord);
    insertLadderAtBack(&ladderList, firstladder);

    while(ladderList != NULL){
        WordNode* current = popLadderFromFront(&ladderList); //check the
word in queue format
        char* currWord = current->myWord;

        if(strcmp(currWord, finalWord) == 0){
            freeLadderList(ladderList);
            return current;
        }

        // breadth first search
        for(int i = 0; i < numWords; i++){
            if(checkWord(currWord, words[i]) && !usedWord[i]){ //mark used
words
                usedWord[i] = true;
                WordNode* newLadder = copyLadder(current);
                insertWordAtFront(&newLadder, words[i]); //make partial
ladders and add to ladderlist
                insertLadderAtBack(&ladderList, newLadder);
            }
        }

        freeLadder(current);
    }

    freeLadderList(ladderList);

    return NULL; //ladder not found
}

bool test_findShortestWordLadder(){
    char* words[] = {"demo", "deme", "dene", "done"};
    bool usedWord[4] = {false, false, false, false};
    int numWords = 4;
    int wordSize = 4;
    char startWord[] = "demo";
    char finalWord[] = "done";

    WordNode* shortestLadder = findShortestWordLadder(words, usedWord,
numWords, wordSize, startWord, finalWord);
    char* expected[] = {"done", "dene", "deme", "demo"};
    WordNode* check = shortestLadder;
    int index = 0;

    while(check != NULL){
        if(strcmp(check->myWord, expected[index]) != 0){
            freeLadder(shortestLadder);
            return false;
        }

        check = check->next;
        index++;
    }

    freeLadder(shortestLadder);
    return true;
}

// interactive user-input to set a word;
//  ensures sure the word is in the dictionary word array
void setWord(char** words, int numWords, int wordSize, char* aWord) {
    bool valid = false;
    printf("  Enter a %d-letter word: ", wordSize);
    int count = 0;
    while (!valid) {
        scanf("%s",aWord);
        count++;
        valid = (strlen(aWord) == wordSize);
        if (valid) {
            int wordInd = findWord(words, aWord, 0, numWords-1);
            if (wordInd < 0) {
                valid = false;
                printf("    Entered word %s is not in the
dictionary.\n",aWord);
                printf("  Enter a %d-letter word: ", wordSize);
            }
        } else {
            printf("    Entered word %s is not a valid %d-letter
word.\n",aWord,wordSize);
            printf("  Enter a %d-letter word: ", wordSize);
        }
        if (!valid && count >= 5) { //too many tries, picking random word
            printf("\n");
            printf("  Picking a random word for you...\n");
            strcpy(aWord,words[rand()%numWords]);
            printf("  Your word is: %s\n",aWord);
            valid = true;
        }
    }
}

// helpful debugging function to print a single Ladder
void printLadder(WordNode* ladder) {
    WordNode* currNode = ladder;
    while (currNode != NULL) {
        printf("\t\t\t%s\n",currNode->myWord);
        currNode = currNode->next;
    }
}

// helpful debugging function to print the entire list of Ladders
void printList(LadderNode* list) {
    printf("\n");
    printf("Printing the full list of ladders:\n");
    LadderNode* currList = list;
    while (currList != NULL) {
        printf("  Printing a ladder:\n");
        printLadder(currList->topWord);
        currList = currList->next;
    }
    printf("\n");
}
```

```
//-----------------------------------------------------
// The primary application is fully provided in main();
//  no changes should be made to main()
//-----------------------------------------------------
int main() {
    if(test_findShortestWordLadder()){
        printf("findShortestWordLadder test passed.\n");
    }
    else{
        printf("findShortestWordLadder test failed.\n");
    }

    if(test_countWordsOfLength()){
        printf("countWordsOfLength test passed.\n");
    }
    else{
        printf("countWordsOfLength test failed.\n");
    }

    if(test_findWord()){
        printf("findWord test passed.\n");
    }
    else{
        printf("findWord test failed.\n");
    }

    return 0;
}
```