```
/*============ CSCI203/803 ASSIGNMENT-1 MARKING (10 Marks) ==============

========================== YOUR OUTPUT ====================================

Number of words in dictionary: 370103
First 10 emordnilap words:
aa : aa
aaa : aaa
aas : saa
ab : ba
aba : aba
abac : caba
abas : saba
abay : yaba
abba : abba
abo : oba
Longest: kinnikinnik (11 characters).

Number of valid words read: 612
Number of unique words read: 283
Number of unique words read that were found in the dictionary: 280

admire: armied damier dimera merida
after: afret frate trefa
again: angia
along: anglo gonal langO logan longa nogal
and: dan nad
another: athenor rheotan
answer: resawn
any: nay yan
are: aer ear era rea
as: sa

Word with the most anagrams: tears
Longest word with anagram(s): streaming
Total number of words with anagram(s): 185
Total number of anagrams found: 394

Total run time (secs): 10

======== MARKING & FEEDBACK ON YOUR OUTPUT ===================================

Your output is correct.

========= MARKING & FEEDBACK ON YOUR CODE (6 marks) ==========================

Step-2 (2 marks)

Step 2 was done correctly.

Step-3 (3 marks)

The pre-processing was done correctly.

The unique words were done correctly.

Step-4 (3 marks)

Some good optimisations here.

========= MARKING & FEEDBACK ON YOUR REPORT (2 marks) ========================

Your report is ok and consistent with the code.

========== OTHER DEDUCTIONS ==================================================

No other deductions, great work! :D
```

```
--------------------------------------------------------------------------
TOTAL MARKS FOR ASSIGNMENT 1 STEPS 2 to 4:  10 MARKS (OUT OF 10)
--------------------------------------------------------------------------

============================ YOUR CODE =======================================*/
/*
Assignment 1
Ben Malen
bm365

This program finds emordnilap words from a dictionary file, and anagrams of words in a sample
file.
*/
#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
using namespace std;

const unsigned int MAX_PRINT = 10; // How many emordnilap and anagrams should be printed?
const unsigned int DICT_SIZE = 400000; // Max dictionary word array size
const unsigned int SAMPLE_SIZE = 40000; // Max sample word array size
const unsigned int WORD_SIZE = 35; // Max word length + 1

// Prototypes
void swapChar(char& x, char& y);
void swapStr(string& x, string& y);
void swapInt(unsigned int& x, unsigned int& y);
void reverseStr(string& str, unsigned int len);
string sortString(string& in, unsigned int len);
int linearSearch(const string& str, unsigned int len, string* arr, unsigned int n);
int binarySearch(const string& str, string* arr, int left, int right);

class Dictionary {
    public:
        Dictionary();
        ~Dictionary();
        bool load();
        unsigned int getNumWords();
        string* getWords();
        void findEmordnilaps();

    private:
        string* dictWords_;
        unsigned int nDictWords_;
};

class Sample {
    public:
        Sample(Dictionary& d);
        ~Sample();
        string process(const string& in);
        bool spellcheck();
        void findAnagrams();

    private:
        struct Node {
            string item;
            Node* left;
            Node* right;
        };
        Dictionary& dict_;
        string* sampleWords_;
        unsigned int nSampleWords_;
        bool* lenExists_;
        void printAnagrams(unsigned int* indices, const string& word, unsigned int first,
```

```
        unsigned int last);
        // BST functions
        void visit(Node* node, unsigned int& nUniqueWords);
        void insert(const string& str, Node* node);
        Node* insertFirst(const string& str);
        // Heap sort functions for parallel array
        void siftDown(string* arr1, unsigned int* arr2, int n, int i);
        void heapSort(string* arr1, unsigned int* arr2, int n);
};

/*-----------------------------------------------------------------------------*/

// Driver
int main() {
    time_t timeBegin = time(0);
    Dictionary dict;
    if (!dict.load())
        return -1;
    dict.findEmordnilaps();
    Sample sample(dict);
    if (!sample.spellcheck())
        return -1;
    sample.findAnagrams();
    cout << "\nTotal run time (secs): " << difftime(time(0), timeBegin) << endl;
    return 0;
}

/*-----------------------------------------------------------------------------*/
// Dictionary class implementations

// Constructor
Dictionary::Dictionary() : nDictWords_(0) {
    dictWords_ = new string[DICT_SIZE];
}

// Destructor
Dictionary::~Dictionary() {
    delete[] dictWords_;
}

unsigned int Dictionary::getNumWords() {
    return nDictWords_;
}

string* Dictionary::getWords() {
    return dictWords_;
}

// Reads words from the file and adds them to the array.
// Returns true on success, false otherwise.
bool Dictionary::load() {
    ifstream fin;
    fin.open("dictionary.txt");
    if (!fin) {
        cerr << "Could not open dictionary.txt." << endl;
        return false;
    }
    string word;
    while (fin >> word) {
        // Check if array is full.
        if (nDictWords_ >= DICT_SIZE) {
            cerr << "Dictionary array is full." << endl;
            return false;
        }
        dictWords_[nDictWords_++] = word;
    }
    fin.close();
```

```
    cout << "Number of words in dictionary: " << nDictWords_ << endl;
    return true;
}

// Finds and prints the first 10 emordnilap words, and the longest from the entire dictionary.
void Dictionary::findEmordnilaps() {
    cout << "First " << MAX_PRINT << " emordnilap words: " << endl;
    int unsigned i = 0, counter = 0, longestLen = 0;
    string longestWord;
    // Find and print the first 10 only
    for (; i < nDictWords_; ++i) {
        unsigned int len = dictWords_[i].length();
        if (counter == MAX_PRINT)
            break; // Exit the loop once we have found 10.
        if (len == 1)
            continue; // Skip words that are only 1 character long.
        // reverse the word
        string word = dictWords_[i];
        reverseStr(word, len);
        // see if the reversed word exists in the dictionary
        if (binarySearch(word, dictWords_, 0, nDictWords_ - 1) != -1) {
            ++counter;
            cout << dictWords_[i] << " : " << word << endl;
            // Store the longest word and length
            if (len > longestLen) {
                longestLen = len;
                longestWord = word;
            }
        }
    }
    // Here, we only check if a word is an emordnilap if it has more characters than the
    current longest.
    // We resume the search from the previous index position.
    for (; i < nDictWords_; ++i) {
        unsigned int len = dictWords_[i].length();
        if (len <= longestLen)
            continue; // Don't check words shorter than the current longest
        string word = dictWords_[i];
        reverseStr(word, len); // reverse the word
        // see if it exists in the dictionary
        if (binarySearch(word, dictWords_, 0, nDictWords_ - 1) != -1) {
            // We have a new winner
            longestLen = len;
            longestWord = word;
        }
    }
    cout << "Longest: " << longestWord << " (" << longestLen << " characters)." << endl;
}

/*-----------------------------------------------------------------------------*/
// Sample class implementations

// Constructor
Sample::Sample(Dictionary& dict) : dict_(dict), nSampleWords_(0) {
    sampleWords_ = new string[SAMPLE_SIZE];
    lenExists_ = new bool[WORD_SIZE];
}

// Destructor
Sample::~Sample() {
    delete[] sampleWords_;
    delete[] lenExists_;
}

// Replaces A-Z with a-z and removes all other characters in the given string.
// Returns the processed string.
string Sample::process(const string& in) {
```

```
    unsigned int len = in.length();
    string out;
    out.reserve(len);
    for (unsigned int i = 0; i < len; ++i) {
        if (in[i] >= 'A' && in[i] <= 'Z')
            out += (char)(in[i] + 32);
        else if (in[i] >= 'a' && in[i] <= 'z')
            out += in[i];
    }
    return out;
}

// Insert into BST
void Sample::insert(const string& str, Node* node) {
    Node* next;
    bool isLeft;
    if (str == node->item) // already in the tree
        return;
    if (str < node->item) { // we need to go left
        next = node->left;
        isLeft = true;
    }
    else { // we need to go right
        next = node->right;
        isLeft = false;
    }
    if (next != NULL)
        insert(str, next); // keep trying
    else {
        Node* next = new Node; // make a new node
        // Initialise the contents
        next->item = str;
        next->left = NULL;
        next->right = NULL;
        if (isLeft) // update the parent
            node->left = next;
        else
            node->right = next;
    }
}

// Insert the first node into the BST
Sample::Node* Sample::insertFirst(const string& str) {
    Node* first = new Node;
    // Initialise the contents
    first->item = str;
    first->left = NULL;
    first->right = NULL;
    return first;
}

// Visit each node in BST and add words to the array if they also exist in the dictionary
void Sample::visit(Node* node, unsigned int& nUniqueWords) {
    if (node->left != NULL)
        visit(node->left, nUniqueWords);
    ++nUniqueWords;
    // Check if the unique word is also in the dictionary
    if (binarySearch(node->item, dict_.getWords(), 0, dict_.getNumWords() - 1) != -1) {
        // Check if array is full.
        if (nSampleWords_ >= SAMPLE_SIZE) {
            cerr << "Sample words array is full." << endl;
            return;
        }
        // A boolean array [0..35] is used to represent which word lengths exist to help cull
        possible
        // candidates when checking for anagrams.
        lenExists_[node->item.length()] = true;
```

```
        // Add it to the final array.
        sampleWords_[nSampleWords_++] = node->item;
    }
    if (node->right != NULL)
        visit(node->right, nUniqueWords);
}

// Reads words from the file and adds them to the array.
// Returns true on success, false otherwise.
bool Sample::spellcheck() {
    ifstream fin;
    fin.open("sample.txt");
    if (!fin) {
        cerr << "Could not open sample.txt." << endl;
        return false;
    }
    unsigned int nValidWords = 0, nUniqueWords = 0;
    Node* root = NULL; // BST
    string in;
    while (fin >> in) {
        string word = process(in);
        if (word.empty())
            continue; // skip
        ++nValidWords;
        if (root == NULL)
            root = insertFirst(word);
        insert(word, root); // insert word into BST
    }
    fin.close();
    visit(root, nUniqueWords); // visit each node to count unique, and to check if in
    dictionary/add to final array
    cout << "\nNumber of valid words read: " << nValidWords << "\n" // 612
         << "Number of unique words read: " << nUniqueWords << "\n" // 283
         << "Number of unique words read that were found in the dictionary: " << nSampleWords_
         << "\n"; // 280
    return true;
}

// Restores the heap property.
void Sample::siftDown(string* arr1, unsigned int* arr2, int n, int i) {
    int parent = i; // parent element we are sifting down
    int leftChild = i * 2 + 1; // index of the left child (but does it exist?)
    if (leftChild < n) { // then the left child exists
        int biggestChild = leftChild;
        int rightChild = leftChild + 1; // index of the right child (but does it exist?)
        if (rightChild < n && arr1[biggestChild] < arr1[rightChild]) // then the right child
        exists and it is the biggest child
            biggestChild = rightChild;
        if (arr1[parent] < arr1[biggestChild]) { // if the biggest child is bigger than the
        parent, then swap them and siftDown
            swapStr(arr1[parent], arr1[biggestChild]);
            swapInt(arr2[parent], arr2[biggestChild]);
            siftDown(arr1, arr2, n, biggestChild);
        }
    }
}

// Performs a heap sort on two arrays in parallel based on the values in the first array.
// Used to sort candidates and indices for the anagrams algorithm.
void Sample::heapSort(string* arr1, unsigned int* arr2, int n) {
    for (int i = (n - 1) / 2; i >= 0; --i) // Converts an array into a heap.
        siftDown(arr1, arr2, n, i);
    for (int i = n - 1; i >= 0; i--) {
        swapStr(arr1[0], arr1[i]);
        swapInt(arr2[0], arr2[i]);
        siftDown(arr1, arr2, i, 0);
    }
```

```
}

// To print a word along with its anagrams, the indices array is traversed starting and ending
at the
// index positions that were found. The associated word from the dictionary is placed into an
// auxiliary array, which is sorted and printed.
void Sample::printAnagrams(unsigned int* indices, const string& word, unsigned int first,
unsigned int last) {
    unsigned int nWords = 0;
    string words[100]; // auxiliary array
    for (; first <= last; ++first) {
        if (word == dict_.getWords()[indices[first]]) // Make sure we don't print the same word
        that was searched for
            continue;
        int j; // Insert the word into the array into the correct position
        for (j = nWords - 1; (j >= 0 && words[j] > dict_.getWords()[indices[first]]); --j) //
        Loop through the array, starting with the last element
            words[j + 1] = words[j]; // Keep shifting each element, to make room for the new
            element
        words[j + 1] = dict_.getWords()[indices[first]]; // Insert the word
        ++nWords; // Update the total
    }
    cout << "\n" << word << ": ";
    for (unsigned int i = 0; i < nWords; ++i)
        cout << words[i] << " ";
}

// Finds anagrams of the sample words in the dictionary.
void Sample::findAnagrams() {
    unsigned int nWordsWithAnagrams = 0,
                 nAnagrams = 0,
                 nCandidates = 0,
                 longestLen = 0,
                 longestIndex = 0,
                 mostAnagrams = 0,
                 mostAnagramsIndex = 0,
                 printCount = 0;
    string* candidates = new string[DICT_SIZE]; // Array to store sorted strings, e.g. animal =
    aailmn
    unsigned int* indices = new unsigned int[DICT_SIZE]; // Index of the word in the dictionary
    (so we can print it later)
    for (unsigned int i = 0; i < dict_.getNumWords(); ++i) { // For each dictionary word
        unsigned int len = dict_.getWords()[i].length();
        if (len == 1 || !lenExists_[len]) // Skip dictionary words if their length does not
        match that of any sample word
            continue;
        candidates[nCandidates] = sortString(dict_.getWords()[i], len); // Sort the characters
        in the dictionary word, and add it to the candidates array
        indices[nCandidates] = i; // Remember the original dictionary index
        ++nCandidates;
    }
    heapSort(candidates, indices, nCandidates); // Sort the candidates and indices arrays in
    parallel, based on the candidate string
    for (unsigned int i = 0; i < nSampleWords_; ++i) { // For each sample word
        unsigned int len = sampleWords_[i].length(), diff = 0;
        if (len == 1)
            continue;
        string keyword = sortString(sampleWords_[i], len); // Sort the characters in the sample
        word
        int index = binarySearch(keyword, candidates, 0, nCandidates - 1); // Perform a binary
        search
        if (index == -1)
            continue;
        unsigned int first = index, last = index;
        while ((first - 1) < first && keyword == candidates[first - 1]) // Find the first
        occurrence
            --first;
```

```
        while ((last + 1) < nCandidates && keyword == candidates[last + 1]) // Find the last
        occurrence
            ++last;
        diff = last - first;
        if (diff < 1) // A difference greater than 0 means we found anagrams
            continue;
        ++nWordsWithAnagrams;
        nAnagrams += diff;
        if (len > longestLen) { // Check if it is the longest word with anagrams
            longestLen = len;
            longestIndex = i;
        }
        if (diff > mostAnagrams) { // Check if it is the word with the most anagrams
            mostAnagrams = diff;
            mostAnagramsIndex = i;
        }
        if (printCount >= MAX_PRINT) // Print the first 10 only
            continue;
        ++printCount;
        printAnagrams(indices, sampleWords_[i], first, last); // Sort and print the anagrams
        for this word
    }
    delete[] indices;
    delete[] candidates;
    cout << "\n\nWord with the most anagrams: " << (mostAnagrams > 0 ?
    sampleWords_[mostAnagramsIndex] : "Not found") << "\n" // = tears
         << "Longest word with anagram(s): " << (longestLen > 0 ? sampleWords_[longestIndex] :
         "Not found") << "\n" // = streaming
         << "Total number of words with anagram(s): " << nWordsWithAnagrams << "\n" // = 185
         << "Total number of anagrams found: " << nAnagrams << endl; // = 394
}

/*------------------------------------------------------------------------------*/

// Swaps the values of two chars.
void swapChar(char& x, char& y) {
    char temp = x;
    x = y;
    y = temp;
}

// Swaps the values of two strings.
void swapStr(string& x, string& y) {
    string temp = x;
    x = y;
    y = temp;
}

// Swaps the values of two integers.
void swapInt(unsigned int& x, unsigned int& y) {
    unsigned int temp = x;
    x = y;
    y = temp;
}

// Reverses the given string.
void reverseStr(string& str, unsigned int len) {
    // Swap chars starting from both ends
    for (unsigned int i = 0; i < len / 2; ++i)
        swapChar(str[i], str[len - i - 1]);
}

// Returns a sorted string.
string sortString(string& in, unsigned int len) {
    string out;
    out.reserve(len);
    // To represent a-z
```

```
    unsigned int charCount[26] = {};
    // Count chars ('a' - 'a' = 0, 'b' - 'a' = 1, etc.)
    for (unsigned int i = 0; i < len; ++i)
        ++charCount[in[i] - 'a'];
    for (unsigned int i = 0; i < 26; ++i)
        for (unsigned int j = 0; j < charCount[i]; ++j)
            out += (char)('a' + i);
    return out;
}

// Linear search for Step 1
int linearSearch(const string& str, unsigned int len, string* arr, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i) {
        // Checking the length first was tested and proved to be faster than a direct
        comparison using ==.
        if (len != arr[i].length())
            continue;
        if (str == arr[i])
            return i;
    }
    return -1;
}

// Returns the index if the given string is found in the given array, -1 otherwise.
int binarySearch(const string& str, string* arr, int left, int right) {
    while (left <= right) {
        int middle = left + (right - left) / 2;
        if (arr[middle] < str)
            left = middle + 1;
        else if (arr[middle] > str)
            right = middle - 1;
        else
            return middle;
    }
    return -1;
}

/*-----------------------------------------------------------------------------

Specifications:

Final run time: 10 seconds
Machine: Banshee

Binary search found 2860 emordnilap words in 3.65 seconds on Banshee. It is estimated through
linear regression that it would take 10.2 hours to find 2860 emordnilap words using linear
search,
making binary search ~10,000 times faster on Banshee (10.2 hours / 3.65 seconds).

Data structures and algorithms used:

(a) Emordnilap words

We loop through each word in the dictionary. The word is reversed and binary search looks for the
reversed word in the dictionary. Optimisation: After the first 10 have been printed, words are
not reversed/searched for unless they are longer than the current longest found.

(b) Spell-check
If a processed word is valid, it is counted and placed into a BST. Each node in the final BST is
visited, where the number of unique words is incremented. If binary search finds the word in the
dictionary, it is counted and added to the final array.

(c) Anagrams

The dictionary is traversed only once. If a word is a possible candidate (i.e. there exists a
sample
word with the same length), the characters in the word are sorted (e.g. animal becomes aailmn).
```

```
The
sorted word is added to an array named candidates. A second array is used to remember the index
of
the original word in the dictionary (for printing). These arrays are sorted in parallel with
heap-
sort based on the candidate string value.

We loop through each sample word. The characters in the sample word are sorted. If binary search
finds the sorted word in the candidates array, we traverse up and down to find the first and last
occurrence, determining the number of anagrams the word has.

-------------------------------------------------------------------------------*/
```