ASS1

	Section	Page
Introduction	1	1
Main	4	3
Reading Words	9	4
AVL Functions	13	5
Rotation	21	8
Sorting the Words	26	10
Output	33	12
All Done	36	13
Index	37	14

1. Introduction. This is a literate program which solves the problem set in Assignment one—text processing.

This program is required to:

1. Read the name of the text file from the console.
2. Read in a text file, not all at once. (This can be line by line, word by word or character by character.)
3. The file content must be converted to a sequence of words, discarding punctuation and folding all letters into lower case.
4. Store the unique words and maintain a count of each different word.
5. The words should be ordered by decreasing count and, if there are multiple words with the same count, alphabetically. (This ordering may be achieved as the words are read in, partially as the words are read or at the end of all input processing.)
6. Output the first ten words in the sorted list, along with their counts.
7. Output the last ten words in the list, along with their counts.

We will start with the outline of the program.

```
using namespace std;
⟨ Headers 6 ⟩
⟨ Prototypes for functions 14 ⟩
⟨ Global data 2 ⟩
⟨ The main program 4 ⟩
⟨ Implementation of functions 15 ⟩
```

2. For this program we will start by declaring the main global data structures that will be needed. These will include storage for unique words, for which we will use a string pool, *pool*—an array of **char** in which the words will be stored packed tightly together.

In addition to this we will need storage for the data related to each word. This could be maintained in an array of structs or in a set of arrays—we will use the latter approach.

These arays are:

1. *wordStart*, the starting position of each word in *pool*.
2. *wordEnd*, the ending position of each word in *pool*.
3. *wordCount*, the number of times that this word has been encountered in the text file.

We will also need to declare two constants, `POOL_SIZE`, the size of *pool* and `MAX_WORD_COUNT`, the maximum number of unique words in the input text.

```
⟨ Global data 2 ⟩ ≡
  const int POOL_SIZE = 500000;
  const int MAX_WORD_COUNT = 50000;
  char pool[POOL_SIZE];
  int wordStart[MAX_WORD_COUNT];
  int wordEnd[MAX_WORD_COUNT];
  int wordCount[MAX_WORD_COUNT];
```

See also section 3.

This code is used in section 1.

3. To make the process of searching the words efficient we will organise the words in the form of an AVL tree, ordered alphabetically. To do this I will need further arrays:

1. *treeLeft*, the index in these arrays of the left child of the current word.
2. *treeRight*, the index of the right child of the current word.
3. *treeHeight*, the depth of the current word.

The initialisation of *treeLeft*, *treeRight* and *treeHeight* will allow the use of index 0 to designate empty links. This means that index 1 will be used to manage the first word in the pool.

We will also need to keep track of some global counters. These are:

1. *numChars*, the number of valid characters in *pool*.
2. *numWords*, the number of unique words.
3. *root*, the index of the word which is the root of the AVL tree.

Each of these variables is initialised to a suitable value. The *treeLeft* and *treeRighht* arrays are set to 0 for all elements. We set the *treeHeight*[0] to the value −1, the remaining elements will be set to 0 as well. As a result of this, empty nodes, for which their location, *node*, has the value 0 will return a *treeHeight* value of −1, which is exactly what we want.

This hack removes the need for a chunk of special-case coding for empty leaf-nodes.

⟨ Global data 2 ⟩ +≡

```
int treeLeft[MAX_WORD_COUNT] = {0};
int treeRight[MAX_WORD_COUNT] = {0};
int treeHeight[MAX_WORD_COUNT] = {−1};
int numChars = 0;
int numWords = 0;
int root = 0;
```

4. Main. Ok, let's start writing *main*. The skeleton of the *main* program is as follows.

```
⟨ The main program 4 ⟩ ≡
  int main( )
  {
    ⟨ Variables of main 5 ⟩
    ⟨ Open and validate the input file 7 ⟩
    ⟨ Read the words 9 ⟩
    ⟨ Sort the words 26 ⟩
    ⟨ Write the results 33 ⟩
    ⟨ Finish up 36 ⟩
  }
```

This code is used in section 1.

5. The first thing we need to do is declare the variables we need to input words. Let's start with the character array *filename* and the input stream *fin*.

```
⟨ Variables of main 5 ⟩ ≡
  char filename[20];
  ifstream fin;
```

See also section 8.

This code is used in section 4.

6. We need a couple of header files *iostream* for stream-based input and *fstream* for managing files.

```
⟨ Headers 6 ⟩ ≡
#include <iostream>
#include <fstream>
```

See also sections 10 and 25.

This code is used in section 1.

7. Now we can get the file opened and ready for input. We will prompt for the input file name using *cerr* so that we can redirect the output without getting the prompt in the output file and so that we can see the prompt even when we redirect standard output. We will then read in the file name and open an input stream. We should test for errors too, I guess.

```
⟨ Open and validate the input file 7 ⟩ ≡
  cerr ≪ "Please␣enter␣the␣name␣of␣the␣input␣file:␣";
  cin ≫ filename;
  fin.open(filename);
  if (¬fin) {
    cerr ≪ "Error␣opening␣file␣" ≪ filename ≪ ".␣Program␣will␣exit." ≪ endl;
    return 0;
  }
```

This code is used in section 4.

8. We are now ready to start the program proper.

We need a character array *word* to assemble words from the input file into. We will assume that individual words are never more than `WORD_SIZE` in length.

```
⟨ Variables of main 5 ⟩ +≡
  const int WORD_SIZE = 100;
  char word[WORD_SIZE];
```

9. Reading Words. The process of getting the words wil be performed in a loop containing a finite-state machine. The file will be read one character at a time and the input character will be classified as:

alpha an alphabetic character;
space a character we can treat as a space;
other 66any other character which we will ignore.

The variable *wordLen* will be used to keep track of the length of the current word.

```
⟨ Read the words 9 ⟩ ≡
  int wordLen = 0;
  char inChar;
  while (fin) {
    fin.get(inChar);
    if (isalpha(inChar)) {
      ⟨ Process an alphabetic character 11 ⟩
    }
    else if (isspace(inChar)) {
      ⟨ Process a space character 12 ⟩
    }
  }
```

This code is used in section 4.

10. We need another header for *isalpha*( ), etc.

```
⟨ Headers 6 ⟩ +≡
#include <cctype>
```

11. When we encounter a letter we need to force it to lower case and add it in to the current word.

```
⟨ Process an alphabetic character 11 ⟩ ≡
  word[++wordLen] = tolower(inChar);
```

This code is used in section 9.

12. When we encounter a space we need to do a bit more. First we record the word, if there is one, in the AVL tree and then we reset *wordLen* ready to continue. Multiple spaces will not cause a problem as we will only do something when we have a word we have not yet dealt with.

```
⟨ Process a space character 12 ⟩ ≡
  if (wordLen > 0) {
    root = AVLinsert(root, word, wordLen);
    wordLen = 0;
  }
```

This code is used in section 9.

13. AVL Functions. We will deal with the functions that are used to maintain the AVL-balanced tree before proceding with the rest of *main*. Apart from *AVLinsert* we will also define the functions that it calls. For each will we define both the prototype and the implementation.

14. Let us start with *AVLinsert*

```
⟨ Prototypes for functions 14 ⟩ ≡
  int AVLinsert(int, char[], int);
```

See also sections 19, 21, 23, 27, 29, 31, and 34.

This code is used in section 1.

15. This function takes a *word* of specified *length* and inserts it into the AVL tree. If the curent *node* is empty we create a new one and store the word in it. Otherwise we check to see whether *word* is is alphabetically before, after or equal to the contents of *node* and proceed accordingly.

The implementation of the *AVLinsert* function is as follows:

```
⟨ Implementation of functions 15 ⟩ ≡
  int AVLinsert(int node, char word[], int length)
  {
    if (node ≡ 0) {
      ⟨ Add a word to the tree 16 ⟩
    }
    int test = compareWord(node, word, length);
    if (test < 0) {
      treeLeft[node] = AVLinsert(treeLeft[node], word, length);
      ⟨ Left insertion balance check 17 ⟩;
    }
    else if (test > 0) {
      treeRight[node] = AVLinsert(treeRight[node], word, length);
      ⟨ Right insertion balance check 18 ⟩;
    }
    else {
      wordCount[node]++;
    }
    treeHeight[node] = max(treeHeight[treeLeft[node]], treeHeight[treeRight[node]]) + 1;
    return node;
  }
```

See also sections 20, 22, 24, 28, 30, 32, and 35.

This code is used in section 1.

16. When we add a new word to the tree we must set up an additional word in our **int** arrays and add the word to the end of the sring pool. The new index is used as the return value of the *AVLinsert* function.

```
⟨ Add a word to the tree 16 ⟩ ≡
  node = ++numWords;
  wordStart[node] = numChars + 1;
  wordEnd[node] = numChars + length;
  wordCount[node] = 1;
  treeLeft[node] = 0;
  treeRight[node] = 0;
  treeHeight[node] = 0;
  for (int i = 1; i ≤ length; i++) pool[numChars + i] = word[i];
  numChars += length;
  return node;
```

This code is used in section 15.

17. After inserting a word we need to check and, if needed, restore the AVL balance. We start with the left subtree insertion.

```
⟨ Left insertion balance check 17 ⟩ ≡
  if ((treeHeight[treeLeft[node]] − treeHeight[treeRight[node]]) ≡ 2) {
    int test = compareWord(treeLeft[node], word, length);
    if (test < 0) node = rotate_right(node);
    else node = double_right(node);
  }
```

This code is used in section 15.

18. And then the right subtree insertion.

```
⟨ Right insertion balance check 18 ⟩ ≡
  if ((treeHeight[treeRight[node]] − treeHeight[treeLeft[node]]) ≡ 2) {
    int test = compareWord(treeRight[node], word, length);
    if (test < 0) node = double_left(node);
    else node = rotate_left(node);
  }
```

This code is used in section 15.

19. The *compareWord* function has not yet been defined. Its prototype is as follows.

```
⟨ Prototypes for functions 14 ⟩ +≡
  int compareWord(int, char[], int);
```

20. This function compares the word stored at the *current* node with the *word* and returns −1, 0 or 1 depending in this comparison.

```
⟨ Implementation of functions 15 ⟩ +≡
  int compareWord(int current, char word[], int length)
  {
    int clen = wordEnd[current] − wordStart[current] + 1;
    int shorter = min(clen, length);
    int offset = wordStart[current] − 1;
    for (int i = 1; i ≤ shorter; i++)
      if (word[i] < pool[offset + i]) return −1;
      else if (word[i] > pool[offset + i]) return 1;
    if (clen > length) return −1;
    else if (clen < length) return 1;
    return 0;
  }
```

21. Rotation. We now need the rotation functions, starting with the single rotations. First the prototypes.

```
⟨ Prototypes for functions 14 ⟩ +≡
  int rotate_right(int);
  int rotate_left(int);
```

22. In each of the single rotations the child of the current *node* replaces it as the local root and the current *node* moves down to become one of its children. One child of the new local root is detached and reattached as a child of the old root. Once the rotation has been completed we need to recompute the value of *treeHeight* for the two nodes that we have changed.

```
⟨ Implementation of functions 15 ⟩ +≡
  int rotate_right(int node)
  {
    int k1 = treeLeft[node];      /* set k1 to the index of the currrent node's left child */
    treeLeft[node] = treeRight[k1];      /* move the right child of k1 to be the left child of node */
    treeRight[k1] = node;      /* now attach the old root as the right child of k1 */
    treeHeight[node] = max(treeHeight[treeLeft[node]], treeHeight[treeRight[node]]) + 1;
    treeHeight[k1] = max(treeHeight[treeLeft[k1]], treeHeight[treeRight[k1]]) + 1;
    return k1;      /* link the left child back up the tree. */
  }
  int rotate_left(int node)
  {
    int k1 = treeRight[node];
    treeRight[node] = treeLeft[k1];
    treeLeft[k1] = node;
    treeHeight[node] = max(treeHeight[treeLeft[node]], treeHeight[treeRight[node]]) + 1;
    treeHeight[k1] = max(treeHeight[treeLeft[k1]], treeHeight[treeRight[k1]]) + 1;
    return k1;
  }
```

23. We also need the double rotations. Prototypes, first.

```
⟨ Prototypes for functions 14 ⟩ +≡
  int double_right(int);
  int double_left(int);
```

24. Double rotation is used when a single rotation is not enough to reinstate the AVL balance property. The mechanics of double rotation should be clear from the implementation.

```
⟨ Implementation of functions 15 ⟩ +≡
  int double_right(int node)
  {
    treeLeft[node] = rotate_left(treeLeft[node]);
    node = rotate_right(node);
    return node;
  }
  int double_left(int node)
  {
    treeRight[node] = rotate_right(treeRight[node]);
    node = rotate_left(node);
    return node;
  }
```

25. we need another header file for *strncmp*.

⟨ Headers 6 ⟩ +≡
```
#include <cstring>
```

26. Sorting the Words. This is a two-step process. First we perform an in-order traversal of our BST tree to obtain a list of indices in alphabetical order. Then we perform a merge sort of this resulting array in ascending order by *wordCount*. I use merge sort as it will preserve the existing alphabetical order we obtained from the tree traversal.

Now the tree is finished with, we can use *treeHeight*[ ] to store the in-order traversal and, once we have performed this step, we can use *treeLeft*[ ] as the scratch array for the merge sort.

This hack of reusing these, now redundant, arrays will save us a bit of memory.

⟨ Sort the words 26 ⟩ ≡
```
inOrder(root);
mergeSort(1, numWords);
```
This code is used in section 4.

27. The in-order traversal of the tree is accomplished by the *inOrder* function. This has a prototype of:

⟨ Prototypes for functions 14 ⟩ +≡
```
void inOrder(int);
```

28. This recursive function traverses the tree and notes the *node* of the traversal order sequentially into the *treeHeight* array. We will use the **static int** variable *index* to keep track of where we are storing the index.

⟨ Implementation of functions 15 ⟩ +≡
```
void inOrder(int node)
{
  static int index = 0;
  if (node ≡ 0) return;
  inOrder(treeLeft[node]);
  treeHeight[++index] = node;
  inOrder(treeRight[node]);
}
```

29. We also need to code the *mergeSort* function. We start with the prototype.

⟨ Prototypes for functions 14 ⟩ +≡
```
void mergeSort(int, int);
```

30. The implementation of *mergeSort* splits the array in two, recursively calling *mergeSort* on the halves. It then calls *merge* to combine the two sorted halves into a sorted whole. The implementation proceeds as follows:

⟨ Implementation of functions 15 ⟩ +≡
```
void mergeSort(int left, int right)
{
  if (left < right) {
    int mid = (left + right)/2;
    mergeSort(left, mid);
    mergeSort(mid + 1, right);
    merge(left, mid, mid + 1, right);
  }
  return;
}
```

31. We also need the *merge* function. Its prototype is:

```
⟨ Prototypes for functions 14 ⟩ +≡
  void merge(int, int, int, int);
```

32. This function takes two contiguous sorted sub arrays and merges them into the same piece of a scratch array. It then copies the merged result back into the starting array.

As noted earlier we are using *treeHeight* as our data array and *treeLeft* as the scratch array.

The sections of the array to be merged are nominated by the two pairs of integers in the calling sequence. We assume that *treeHeight*[*l1* .. *l2*] and *treeHeight*[*r1* .. *r2*] are sorted on entry and that *treeHeight*[*l1* .. *r2*] will be sorted on return.

The variables *apos*, *bpos* and *cpos* are used to track the current character postions in the left, right and scratch arrays.

```
⟨ Implementation of functions 15 ⟩ +≡
  void merge(int l1, int l2, int r1, int r2)
  {
    int apos = l1;
    int bpos = r1;
    int cpos = l1;
    while (apos ≤ l2 ∧ bpos ≤ r2)
      if (wordCount[treeHeight[apos]] ≥ wordCount[treeHeight[bpos]])
        treeLeft[cpos++] = treeHeight[apos++];
      else treeLeft[cpos++] = treeHeight[bpos++];
    while (apos ≤ l2) treeLeft[cpos++] = treeHeight[apos++];
    while (bpos ≤ r2) treeLeft[cpos++] = treeHeight[bpos++];
    for (cpos = l1; cpos ≤ r2; cpos++) treeHeight[cpos] = treeLeft[cpos];
    return;
  }
```

33. Output. It is now time to write the results. We need the words and counts pointed to by the first and last 10 entries in our sorted *treeHeight* array. I will use a function *printWord* to accomplish this for each individual word.

```
⟨ Write the results 33 ⟩ ≡
  cout ≪ endl;
  cout ≪ "The␣first␣10␣words␣sorted␣alphabetically␣within␣frequency:" ≪ endl;
  for (int i = 1; i ≤ 10; i++) printWord(i);
  cout ≪ endl;
  cout ≪ "The␣last␣10␣words␣sorted␣alphabetically␣within␣frequency:" ≪ endl;
  for (int i = numWords − 9; i ≤ numWords; i++) printWord(i);
```

This code is used in section 4.

34. The *printWord* function is next, starting with the prototype.

```
⟨ Prototypes for functions 14 ⟩ +≡
  void printWord(int);
```

35. This function outputs a word from the string pool accompanied by its count.

```
⟨ Implementation of functions 15 ⟩ +≡
  void printWord(int word)
  {
    word = treeHeight[word];
    cout ≪ "The␣word:␣";
    for (int i = wordStart[word]; i ≤ wordEnd[word]; i++) cout ≪ pool[i];
    cout ≪ "␣occurs␣";
    cout ≪ wordCount[word] ≪ "␣times." ≪ endl;
  }
```

36. All Done. All that remains is to close the input file.

⟨ Finish up 36 ⟩ ≡
```
fin.close( );
```
This code is used in section 4.

37. Index. This index is automatically created. It lists all the variables used in the program and the section(s) in which they are used. Underlined entries indicate where a variable is defined. The remaining sections of this document are also created automatically.

apos: 32.
AVLinsert: 12, 13, 14, 15, 16.
bpos: 32.
cerr: 7.
cin: 7.
clen: 20.
close: 36.
coding hacks: 3, 26.
compareWord: 15, 17, 18, 19, 20.
cout: 33, 35.
cpos: 32.
current: 20.
double_left: 18, 23, 24.
double_right: 17, 23, 24.
endl: 7, 33, 35.
filename: 5, 7.
fin: 5, 7, 9, 36.
fstream: 6.
get: 9.
i: 16, 20, 33, 35.
ifstream: 5.
inChar: 9, 11.
index: 28.
inOrder: 26, 27, 28.
iostream: 6.
isalpha: 9, 10.
isspace: 9.
k1: 22.
left: 30.
length: 15, 16, 17, 18, 20.
l1: 32.
l2: 32.
main: 4, 13.
max: 15, 22.
`MAX_WORD_COUNT`: 2, 3.
merge: 30, 31, 32.
mergeSort: 26, 29, 30.
mid: 30.
min: 20.
node: 3, 15, 16, 17, 18, 22, 24, 28.
numChars: 3, 16.
numWords: 3, 16, 26, 33.
offset: 20.
open: 7.
pool: 2, 3, 16, 20, 35.
`POOL_SIZE`: 2.
printWord: 33, 34, 35.
right: 30.
root: 3, 12, 26.
rotate_left: 18, 21, 22, 24.
rotate_right: 17, 21, 22, 24.
r1: 32.
r2: 32.
shorter: 20.
std: 1.
strncmp: 25.
test: 15, 17, 18.
tolower: 11.
treeHeight: 3, 15, 16, 17, 18, 22, 26, 28, 32, 33, 35.
treeLeft: 3, 15, 16, 17, 18, 22, 24, 26, 28, 32.
treeRighht: 3.
treeRight: 3, 15, 16, 17, 18, 22, 24, 28.
word: 8, 11, 12, 15, 16, 17, 18, 20, 35.
`WORD_SIZE`: 8.
wordCount: 2, 15, 16, 26, 32, 35.
wordEnd: 2, 16, 20, 35.
wordLen: 9, 11, 12.
wordStart: 2, 16, 20, 35.

⟨ Add a word to the tree 16 ⟩ Used in section 15.
⟨ Finish up 36 ⟩ Used in section 4.
⟨ Global data 2, 3 ⟩ Used in section 1.
⟨ Headers 6, 10, 25 ⟩ Used in section 1.
⟨ Implementation of functions 15, 20, 22, 24, 28, 30, 32, 35 ⟩ Used in section 1.
⟨ Left insertion balance check 17 ⟩ Used in section 15.
⟨ Open and validate the input file 7 ⟩ Used in section 4.
⟨ Process a space character 12 ⟩ Used in section 9.
⟨ Process an alphabetic character 11 ⟩ Used in section 9.
⟨ Prototypes for functions 14, 19, 21, 23, 27, 29, 31, 34 ⟩ Used in section 1.
⟨ Read the words 9 ⟩ Used in section 4.
⟨ Right insertion balance check 18 ⟩ Used in section 15.
⟨ Sort the words 26 ⟩ Used in section 4.
⟨ The main program 4 ⟩ Used in section 1.
⟨ Variables of main 5, 8 ⟩ Used in section 4.
⟨ Write the results 33 ⟩ Used in section 4.