班级：2022211301　　　**姓名：**卢安来　　　**学号：**2022212720

wordst_bad.cpp 用于统计一文本文件里的每个单词的数量以及出现在文本文件中的行号，最后按照出现次数从大到小排列显示出来，如果出现的行号比较多，那么只显示前 20 次出现的行号。

根据所学知识，对其进行分析和改进。

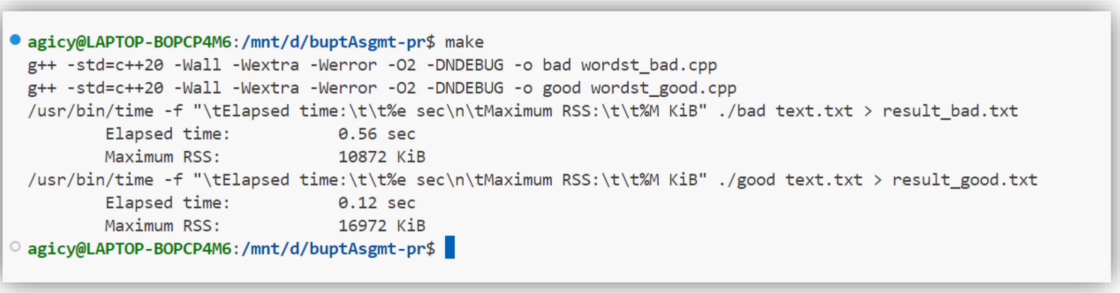
首先说明程序的执行结果：

测试文件中共有 1229240 个单词，共有 6334 个不同的单词。

出现次数前三多的单词对应的统计信息如下。

```
the                    43340 10,12,12,13,33,35,38,38,62,65,65,75,88,92,94,109,114,119,120,122
to                     41620 15,25,28,28,35,36,37,46,68,69,72,81,83,86,87,87,96,110,111,114
of                     36170 8,8,10,12,13,13,33,34,37,38,46,53,54,59,64,65,68,70,72,80
```

即出现次数前三多的单词为 the, to 和 of，出现次数分别为 43340，41620 和 36170，程序的执行时间为 0.56 s。



```
● agicy@LAPTOP-BOPCP4M6:/mnt/d/buptAsgmt-pr$ make
  g++ -std=c++20 -Wall -Wextra -Werror -O2 -DNDEBUG -o bad wordst_bad.cpp
  g++ -std=c++20 -Wall -Wextra -Werror -O2 -DNDEBUG -o good wordst_good.cpp
  /usr/bin/time -f "\tElapsed time:\t\t%e sec\n\tMaximum RSS:\t\t%M KiB" ./bad text.txt > result_bad.txt
          Elapsed time:           0.56 sec
          Maximum RSS:            10872 KiB
  /usr/bin/time -f "\tElapsed time:\t\t%e sec\n\tMaximum RSS:\t\t%M KiB" ./good text.txt > result_good.txt
          Elapsed time:           0.12 sec
          Maximum RSS:            16972 KiB
○ agicy@LAPTOP-BOPCP4M6:/mnt/d/buptAsgmt-pr$ 
```

下面对程序的性能进行分析。

分析一：

在编译选项中加入 -pg，运行程序，然后调用 gprof 查看性能评测结果，可以看出，程序花费了 11.78%的时间在 std::_Rb_tree（红黑树，对应程序中的 std::map）的 lower_bound()（查找）操作上，花费了 9.86%的时间在 std::vector<int>的 push_back() 方法上（插入行号），花费了 7.69%的时间在字符串比较上（红黑树维护），花费了 7.45%的时间在 std::_Rb_tree 的_S_key()方法上，等等。

改进一：数据结构受限

由上述分析可以看出，程序的开销主要在数据结构方面，尤其

是 `std::map<std::string, ?>`，故必须要将其替换掉。经过测量替换为 `std::unordered_map<std::string, ?>`后，性能有 1.6 倍的提升，但这还是不够。

因此，我选择了使用 Trie 树。Trie 树，即字典树，正如其名，天然地适合用于以 `std::string` 为键的增删查改操作。在本题中，其插入复杂度总和为 $\Theta(n)$，其中 n 为文本长度，最终输出查询结果复杂度为 $O(n)$。而原先采用的 `std::map` 的插入复杂度综合则是 $O(n \log n)$ 的。

此外，由于题目只要求显示前 20 次出现的行号，故可以在信息统计到 20 个行号之后不再插入，经过测量，在替换为 Trie 后，这样对运行时间反而没有太大影响，但是节省了 94.58%的空间。

分析二：

改进过后性能分析显示`_init` 占用了 11.22%的时间，`main` 占用了 10.73%的时间，这可能是 `fgetc()`和相应字符串处理的问题，说明程序的 I/O 可能受到了限制。

改进二：I/O 受限。

`fgetc()`可能因其缓冲区较小，存在频繁系统调用的可能，同理，输出时反复 `printf()`，也可能造成其频繁进行系统调用，开销过大。

因此，我选择了封装类 `FileReader` 和 `FileWriter`，负责维护输入输出缓冲区，将缓冲区大小开到 1 MiB，输入输出时调用 `fread()`或 `fwrite()`，这样将有效地减少输入输出时的系统调用次数。

分析三：

排序算法是输出前的重要组成部分，其内部可能涉及频繁交换元素，而元素 `WordInfo` 的规模比较大，这其中存在极大的内存读写开销。可以通过对信息的索引进行排序，将较大读写开销转化为访问开销和较小的读写开销。

同时注意到排序关键字为整数，且绝大多数元素的排序关键字

count 较小，故可部分采用桶排序，部分采用快速排序（或者直接换用基数排序）。

改进三：排序算法受限。

按照上述分析对排序部分进行修改。

下面展示改进后的代码如下。

```
#include <algorithm>
#include <array>
#include <cassert>
#include <cctype>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <format>
#include <iostream>
#include <numeric>
#include <string>
#include <unordered_map>
#include <vector>

static constexpr std::size_t buffer_size = 1 << 20;

struct FileReader {
    std::FILE *file;
    char *buffer;
    char *p1, *p2;
    std::size_t line;

    /**
     * Constructor for FileReader.
     *
     * @param fp std::FILE to read from.
     *
     * Initializes the FileReader object by setting the file pointer, allocating a
     * buffer and setting the line number to 1. The file pointer is not checked for
     * validity.
     */
    FileReader(std::FILE *fp) {
        file = fp;
        buffer = new char[buffer_size];
        p1 = p2 = buffer;
        line = 1;
    }

    /**
     * Destructor for FileReader.
     *
     * Deallocates the buffer allocated in the constructor. The file pointer is not
     * closed.
     */
    ~FileReader(void) {
        delete[] buffer;
    }

    /**
     * Gets a character from the file.
     *
     * If the current buffer is exhausted, reads a new one from the file.
     * If the end of the file is reached, returns EOF.
     *
     * @return The character read, or EOF if the end of the file is reached.
     */
    auto get_char(void) -> char {
        return (p1 == p2 && (p2 = (p1 = buffer) + fread(buffer, 1, buffer_size, file), p1 ==
p2) ? EOF : *p1++);
```

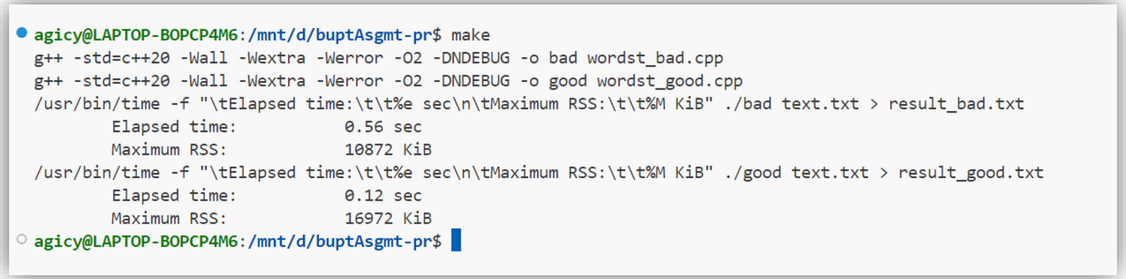
```
    }

    /**
     * Extracts and returns the next word from the file.
     *
     * Skips non-alphabetic characters until a letter is found, then collects
     * characters to form a word until a non-alphabetic character is encountered.
     * Converts the word to lowercase before returning.
     * Increments the line counter when a newline character is encountered.
     *
     * @return The next word in lowercase, or an empty string if EOF is reached.
     */
    auto get_word(void) -> std::string {
        std::string result;
        char ch = get_char();
        while (!std::isalpha(ch)) {
            if (ch == EOF)
                return "";
            if (ch == '\n')
                ++line;
            ch = get_char();
        }
        while (std::isalpha(ch))
            result.push_back(ch), ch = get_char();
        if (ch == '\n')
            ++line;
        for (char &c : result)
            c = std::tolower(c);
        return result;
    }
};

struct FileWriter {
    std::FILE *file;
    char *buffer;
    char *ptr, *end;

    /**
     * Constructs a FileWriter object that writes to the specified file.
     *
     * @param fp std::FILE to write to.
     *
     * Initializes the FileWriter object by setting the file pointer, allocating a
     * buffer and setting the buffer pointers to the beginning of the buffer.
     * The file pointer is not checked for validity.
     */
    FileWriter(std::FILE *fp) {
        file = fp;
        buffer = new char[buffer_size];
        ptr = buffer;
        end = buffer + buffer_size;
    }

    /**
     * Destructor for FileWriter.
     *
     * Ensures any buffered data is written to the file by calling flush() and
     * deallocates the buffer memory. This is called automatically when the
     * FileWriter object goes out of scope.
     */
    ~FileWriter(void) {
        flush();
        delete[] buffer;
    }

    /**
     * Flushes the buffer by writing any buffered data to the file.
     *
     * Ensures any data stored in the buffer is written to the file by calling
     * fwrite() and resets the buffer pointer to the beginning of the buffer.
     */
    auto flush(void) -> void {
        fwrite(buffer, 1, ptr - buffer, file);
        ptr = buffer;
    }

    /**
     * Puts a single character into the output stream.
     *
     * The character is placed in the buffer. If the buffer is full, it is flushed
     * automatically.
     */
    auto put_char(char ch) -> void {
        if (ptr == end)
            flush();
        *ptr++ = ch;
    }

    /**
     * Writes a string to the output stream.
     *
     * The string is copied into the buffer and if the buffer is full, it is
     * flushed automatically.
     */
    auto put_string(const std::string &str) -> void {
        for (size_t i = 0; i < str.length();)
            if (static_cast<ssize_t>(str.length() - i) >= end - ptr) {
                std::memcpy(ptr, str.data() + i, end - ptr);
                flush();
                i += end - ptr;
            } else {
                std::memcpy(ptr, str.data() + i, str.length() - i);
                ptr += str.length() - i;
                i = str.length();
            }
    }
};

constexpr std::size_t limit = 20;

struct WordInfo {
    std::size_t count;
    std::array<size_t, limit> lines;
};

class Trie {
  private:
    struct unit_t {
        std::array<size_t, 26> ch;
        std::optional<WordInfo> info;
    };
    size_t root;
    std::vector<unit_t> units;

  public:
    /**
     * Constructor for the Trie class.
     *
     * Initializes the Trie with a single root node. The root node is
     * represented by index 0 in the units vector, and the vector is
     * resized to accommodate this root node.
     */
    Trie() {
        root = 0;
        units.resize(1);
    }

    /**
     * Inserts a word into the Trie and records its occurrence line.
     *
     * Traverses or creates nodes corresponding to each character in the word.
     * If the word does not exist in the Trie, it creates a path for it.
     * Once the word is inserted, it updates the WordInfo for the terminal node
     * to increase the occurrence count and store the line number, if within limits.
     *
     * @param word The string to be inserted into the Trie.
     * @param line The line number where the word occurs.
     */
    auto insert(const std::string &word, const size_t line) -> void {
        size_t p = root;
        for (char ch : word) {
            size_t c = ch - 'a';
            if (!units[p].ch[c]) {
                units.emplace_back();
                units[p].ch[c] = units.size() - 1;
            }
            p = units[p].ch[c];
        }
        if (!units[p].info)
            units[p].info = WordInfo{0, {}};
        WordInfo &info = units[p].info.value();
        ++info.count;
        if (info.count <= limit)
            info.lines[info.count - 1] = line;
    }

    /**
     * Retrieves all words stored in the Trie along with their associated WordInfo.
     *
     * Performs a depth-first search of the Trie to collect all words and their
     * WordInfo data, including occurrence count and line numbers. Each word is
     * reconstructed from the path in the Trie and paired with its WordInfo.
     *
     * @return A vector of pairs containing each word and its WordInfo.
     */
    auto get_all(void) -> std::vector<std::pair<std::string, WordInfo>> {
        std::vector<std::pair<std::string, WordInfo>> result;

        std::string current;
        auto search = [&](size_t p, auto &&search) -> void {
            if (units[p].info) {
                WordInfo &info = units[p].info.value();
                result.emplace_back(current, info);
            }
            for (size_t i = 0; i < 26; ++i)
                if (units[p].ch[i]) {
                    current.push_back(i + 'a');
                    search(units[p].ch[i], search);
                    current.pop_back();
                }
        };

        search(root, search);
        return result;
    }
};

/**
 * Prints the statistics of words stored in a vector of pairs.
 *
 * This function takes a vector containing pairs of words and their associated
 * WordInfo, and prints the word statistics in a formatted table. The statistics
 * include the word, its occurrence count, and the line numbers where it appears.
 * The words are sorted by their occurrence count in descending order.
 *
 * @param wordsVector A vector of pairs, where each pair consists of a word
 *                    (std::string) and its corresponding WordInfo.
 * @details
 * This function first creates an index vector, then divides the words into buckets
 * based on their occurrence counts. The words in each bucket are sorted, and then
 * the words in the buckets are concatenated in descending order of occurrence
 * counts. Finally, the statistics are printed in a formatted table.
 */
static inline auto printStatis(const std::vector<std::pair<std::string, WordInfo>>
&wordsVector) -> void {
    assert(!wordsVector.empty());

    std::vector<size_t> index(wordsVector.size());
    std::iota(index.begin(), index.end(), 0);

    constexpr size_t buckets_size = 1e3;
    std::vector<std::vector<size_t>> buckets(buckets_size);
    std::vector<size_t> residual;
    for (const auto i : index) {
        const WordInfo &info = wordsVector[i].second;
        if (info.count < buckets_size)
            buckets[info.count].push_back(i);
        else
            residual.push_back(i);
    }

    std::sort(residual.begin(), residual.end(), [&](const size_t a, const size_t b) -> bool {
        return wordsVector[a].second.count > wordsVector[b].second.count;
    });

    std::vector<size_t> result;
    result.insert(result.end(), residual.begin(), residual.end());
    for (size_t i = buckets_size - 1; i < buckets_size; --i)
        result.insert(result.end(), buckets[i].begin(), buckets[i].end());

    FileWriter writer(stdout);

    writer.put_string(std::format("WORD                  COUNT APPEARS-LINES\n"));
    for (const auto i : result) {
        const auto &p = wordsVector[i];
        const std::string &word = p.first;
        const WordInfo &info = p.second;

        writer.put_string(std::format("{:<20} {:<5} ", word.data(), info.count));
        writer.put_string(std::format("{}", info.lines.front()));
        for (size_t j = 1; j < std::min(info.lines.size(), info.count); ++j)
            writer.put_string(std::format(",{}", info.lines[j]));

        writer.put_char('\n');
    }
}

/**
 * The main entry point of the program.
 *
 * This function takes a filename as its argument, reads the file line by line,
 * and inserts each word into a Trie. The line number of each word is stored
 * in the Trie. Finally, the word statistics are printed in a formatted table.
 *
 * @param argc The number of arguments passed to the program.
 * @param argv The array of arguments passed to the program. The first element
 *             is the program name, and the second element is the filename.
 * @return 0 if the program runs successfully, 1 otherwise.
 */
int main(int argc, const char *argv[]) {
    if (argc != 2) {
        std::cerr
            << std::format("Usage: {} filename", argv[0])
            << std::endl;
        return 1;
    }

    std::FILE *fp = fopen(argv[1], "r");
    FileReader file_reader(fp);

    Trie trie;

    std::string word;
    while (word = file_reader.get_word(), word.length())
        trie.insert(word, file_reader.line);
    fclose(fp);

    printStatis(trie.get_all());
    return 0;
}
```

编写 Makefile 文件用于执行测试命令。

```
CXX := g++
CXXFLAGS := -std=c++20 -Wall -Wextra -Werror -O2 -DNDEBUG

run: build_bad build_good
    /usr/bin/time -f "\tElapsed time:\t\t%e sec\n\tMaximum RSS:\t\t%M KiB" ./bad text.txt >
result_bad.txt
    /usr/bin/time -f "\tElapsed time:\t\t%e sec\n\tMaximum RSS:\t\t%M KiB" ./good text.txt >
result_good.txt

build_bad:
    $(CXX) $(CXXFLAGS) -o bad wordst_bad.cpp

build_good:
    $(CXX) $(CXXFLAGS) -o good wordst_good.cpp
```

其中 `text.txt` 即为下发文件中的 `pride_and_prejudice-10.txt`。

输入命令 `make run` 并执行，结果如下。



```
agicy@LAPTOP-BOPCP4M6:/mnt/d/buptAsgmt-pr$ make
g++ -std=c++20 -Wall -Wextra -Werror -O2 -DNDEBUG -o bad wordst_bad.cpp
g++ -std=c++20 -Wall -Wextra -Werror -O2 -DNDEBUG -o good wordst_good.cpp
/usr/bin/time -f "\tElapsed time:\t\t%e sec\n\tMaximum RSS:\t\t%M KiB" ./bad text.txt > result_bad.txt
        Elapsed time:           0.56 sec
        Maximum RSS:            10872 KiB
/usr/bin/time -f "\tElapsed time:\t\t%e sec\n\tMaximum RSS:\t\t%M KiB" ./good text.txt > result_good.txt
        Elapsed time:           0.12 sec
        Maximum RSS:            16972 KiB
agicy@LAPTOP-BOPCP4M6:/mnt/d/buptAsgmt-pr$
```

可以看出，内存占用略有上升（即使消除了 `std::vector` 的大量插入，但换用 Trie 还是带来了大量的内存消耗），而运行速度则变成了原来的 4.6 倍（78.57%）。