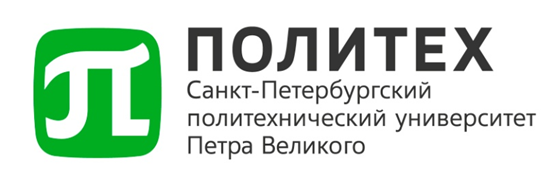
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

***«*САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»**

**Высшая школа программной инженерии**

****

**КУРСОВАЯ РАБОТА**

по дисциплине «**Алгоритмы и структуры данных**»

Студент гр.  Мэн Цзянин

Руководитель Толстолес Алексей Андреевич

Санкт-Петербург

2020 г

**СОДЕРЖАНИЕ**

[Общая постановка задачи 2](#_Toc40727561)

[Требования 3](#_Toc40727562)

[Основная часть работы 4](#_Toc40727563)

[1. Описание алгоритма решения и используемых структур данных 4](#_Toc40727564)

[1.1 Классы 4](#_Toc40727565)

[1.2 Интерфейсы 7](#_Toc40727566)

[1.3 Методы 8](#_Toc40727567)

[2. Анализ алгоритма 14](#_Toc40727568)

[2.1 Методы двоичного дерева поиска 14](#_Toc40727569)

[2.2 Двоичное дерево поиска 25](#_Toc40727570)

[3. Описание спецификации программы (детальные требования) 28](#_Toc40727572)

[4. Описание программы 30](#_Toc40727573)

[4.1 Структура программы 30](#_Toc40727574)

[4.2 Форматы входных 31](#_Toc40727575)

[4.3 Выходных данных 32](#_Toc40727576)

[Заключение 33](#_Toc40727577)

[Список использованных источников 34](#_Toc40727578)

[Приложение 1. Текст программы 35](#_Toc40727579)

[Приложение 2. Протоколы отладки 65](#_Toc40727580)

# Общая постановка задачи

**Вариант 1.2.1**

**Перекрестные ссылки. Бинарное дерево поиска**

* 1. Для разрабатываемого словаря реализовать основные операции:
     + INSERT (ключ, значение) – добавить запись с указанным ключом и значением
     + SEARCH (ключ)- найти запись с указанным ключом
     + DELETE (ключ)- удалить запись с указанным ключом
  2. Предусмотреть обработку и инициализацию исключительных ситуаций, связанных, например, с проверкой значения полей перед инициализацией и присваиванием.
  3. Программа должна быть написана в соответствии со стандартом программирования: C++ Programming Style Guidelines (<http://geosoft.no/development/cppstyle.html>).
  4. Тесты должны учитывать как допустимые, так и не допустимые последовательности входных данных.

# Требования

Разработать и реализовать алгоритм формирования перекрестных ссылок:

* прочитать текст и вывести его с добавлением последовательных номеров строк;
* собрать все слова, встречающиеся в тексте;
* сформировать таблицу, в которой все слова будут расположены в алфавитном порядке и для каждого слова будет указан список строк его нахождения (по возрастанию номеров строк)

Для реализации задания использовать бинарное дерево поиска, узел которого может содержать:

* Ключ – слово
* Информационная часть – ссылка на список, содержащий номера строк

# Основная часть работы

## Описание алгоритма решения и используемых структур данных

### 1.1 Классы

* + BinarySearchTree

В классе BinarySearchTree сохранил :

root\_(корень дерева)

file\_name\_(имя фале).



* + Node

Каждый узел из класса Node, который сохранил:

string key\_(слово)

WordInfo\* info\_(информационная часть)

Node\* left\_

Node\* right\_

Node\* p\_



* + WordInfo

Сохранил информационную часть слова:

LinkList<pair<int, int>> header\_

unsigned int count\_same\_word\_ (Количество одинаковых слов)

unsigned int length\_word\_ (Длина слова)



* + LinkStack



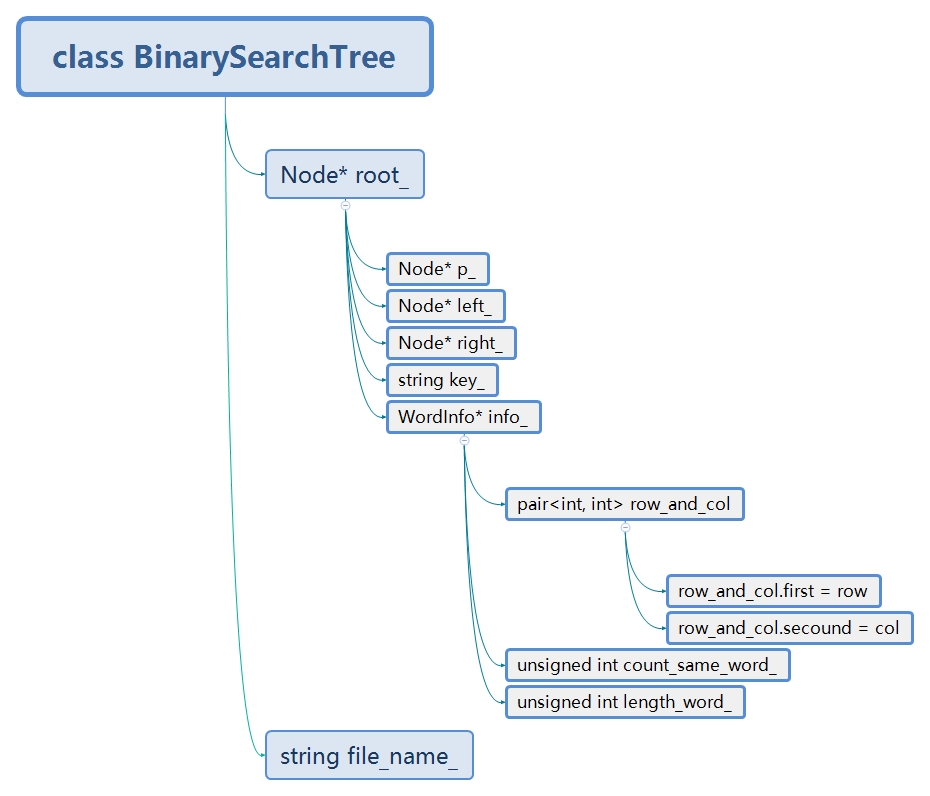
* + LinkList



* + LinkNode



Рис.2



### Интерфейсы

* Сохранить имя файла, которое пользователь вводит

void openFile(string file\_name);

* Выводит текст и вывести его с добавлением последовательных номеров строк в файл

void printTextWithRowNum();

* Найти запись с указанным ключом

void searchWord(string word\_search);

* Удалить запись с указанным ключом

void deleteWord(string dele\_word);

* Собрать все слова, встречающиеся в тексте

void makeTree();

* Печать на экране

void outputDictionaryOrderInFile(bool repetitive\_word);

* true == repetitive\_word

вывод повторяющихся слов

* false == repetitive\_word

не вывод повторяющихся слов

* Выводить в фале

void outputInFile();

### 1.3 Методы

* **void insertNode(Node\* nodeInsert, unsigned int count\_word\_text);**

Каждое слово и его информации (row, col, count, length) – это один узел в бинарном дереве поиска. Функция этой функции заключается в вставке узлов в двоичное дерево поиска.

* **void makeWordTree();**

С помощью предыдущей функции (void insertNode(Node\* nodeInsert, unsigned int count\_word\_text);), эта функция вставляет все слова из статьи в двоичное дерево поиска по лексикографическому порядку, После преобразования слова в нижний регистр и удалил “.”, “.”, “’s”.

**Пример:**

*“This Prestwick House Literary Touchstone Edition includes a glossary*

*and reader's notes to help the modern reader fully appreciate London's masterful weaving of science, philosophy, and the storyteller's”*

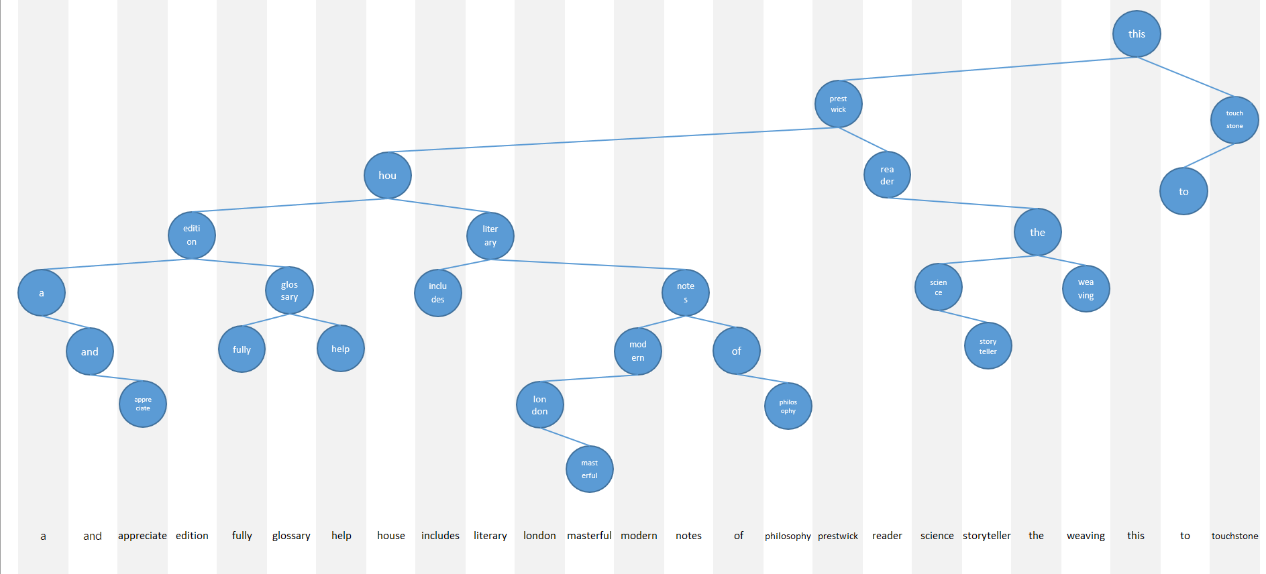
Бинарное дерево поиска должно:

Рис. 1 бинарное дерево поиска

* **void addRowNum();**

Эта функция считывает статью из файла и выводит статью с добавленным номером строки в output.txt. Кроме того, она также подсчитывает общее количество слов в статье, количество строк.

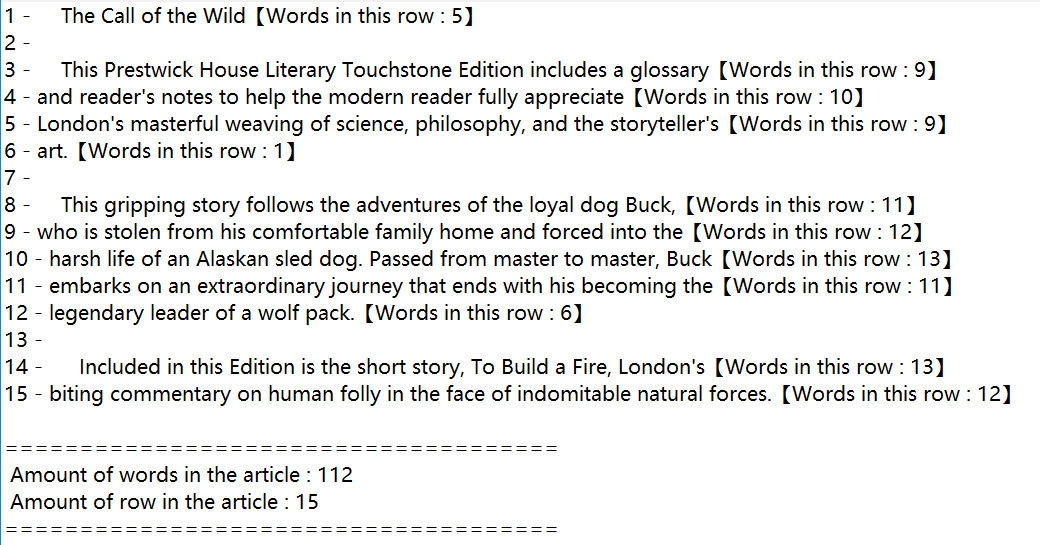


Рис.2 метод void addRowNum()

* **void \_inOrderOnScreen\_ (Node\* root, bool repetitive\_word);**

Сформировать таблицу и печать на экране, в которой все слова будут расположены в алфавитном порядке и для каждого слова будет указан список строк, столбец его нахождения (по возрастанию номеров строк), количество одинаковых слов и количество букв в слове. В аргументе функции **bool repetitive\_word** – это выводит ли повторные слова.

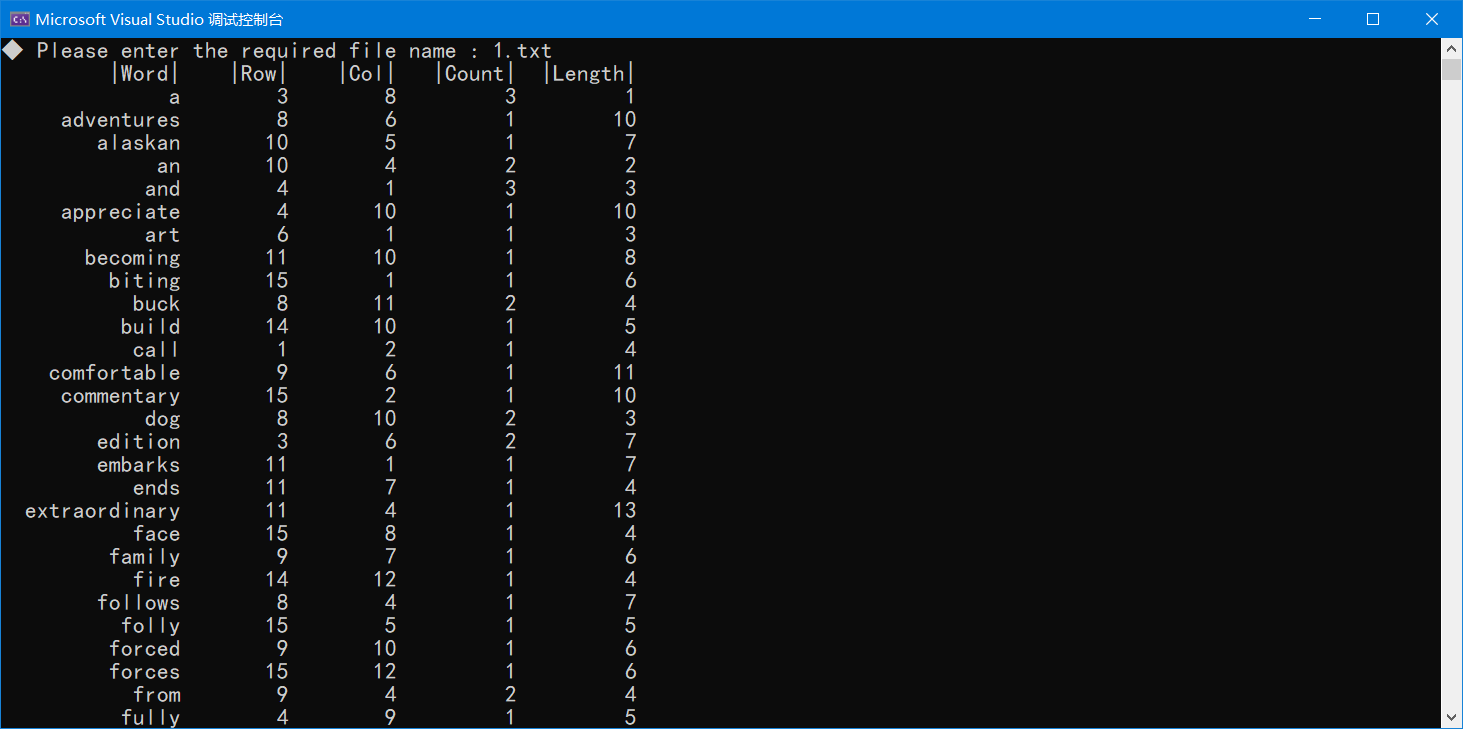


Рис.3 Сформировать таблицу и печать на экране

* **void \_inOrderOutputInFile\_ (Node\* root, ofstream& ofs, bool repetitive\_word);**

Сформировать таблицу и выводить в фале, в которой все слова будут расположены в алфавитном порядке и для каждого слова будет указан список строк, столбец его нахождения (по возрастанию номеров строк), количество одинаковых слов и количество букв в слове. В аргументе функции **bool repetitive\_word** – это выводит ли повторные слова

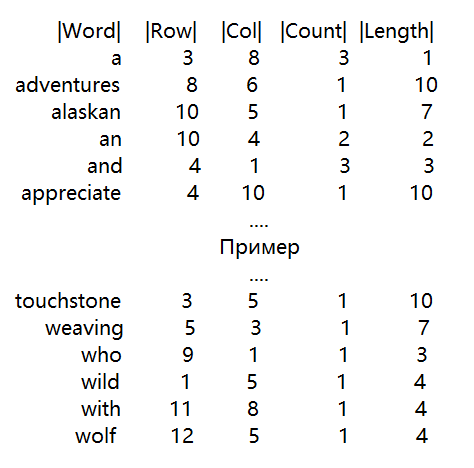


Рис.4

* **void \_changeCaseAndClearSign\_(string& word);**

Для одного и того же слова, но с большой буквы или с “ ’s ” или на конце слова имеет “ **,**”, “ **.**”. Эта функция может преобразовать все буквы в маленькую букву и удалить “ **,**”, “ **.**”.

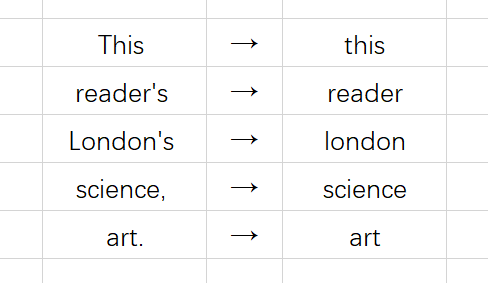


Рис. 5

* **bool \_isHaveSameWord\_(string word, pair<unsigned int, unsigned int> row\_col\_word)**

Эта функция предназначена для определения того, существует ли одно и то же слово в двоичном дереве поиска при вставке нового слова. Если есть однородное слово **return true**, если нет **return false.**

* **Node\* \_searchWordInTree\_(string word\_search);**

Поиск, существует ли введенное пользователем слово.

Пример:

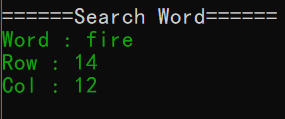
* Искать fire

Рис.6 искать fire

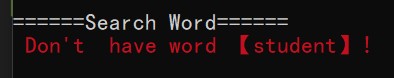
* Искать student

Рис.7 искать student

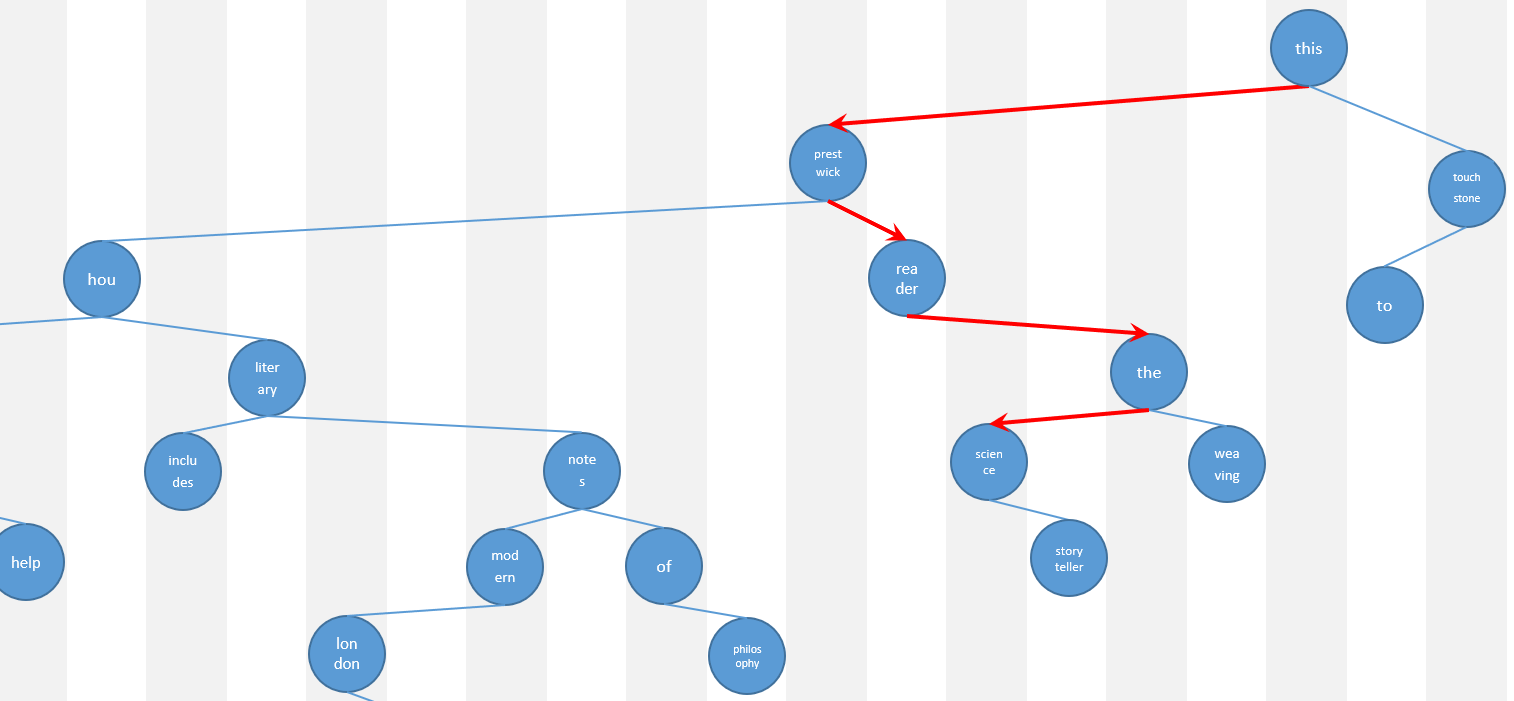
* Искать science

Рис. 8

* **string& \_clearAllSpace\_(string& str);**

Чтобы удаление пробелов в строке, при вычислении длины строки.

* **bool \_deleteNodeBinarySearchTree\_(string dele\_key);**

Удаляет узлы двоичного дерева поиска на основе введенных пользователем слов.

Пример:

* удалить file

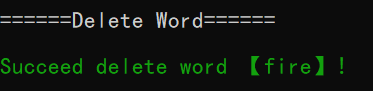


Рис.9 удалить file

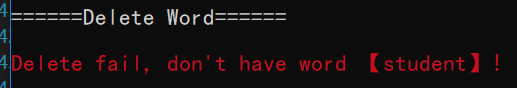
* удалить student

Рис. 10 удалить students

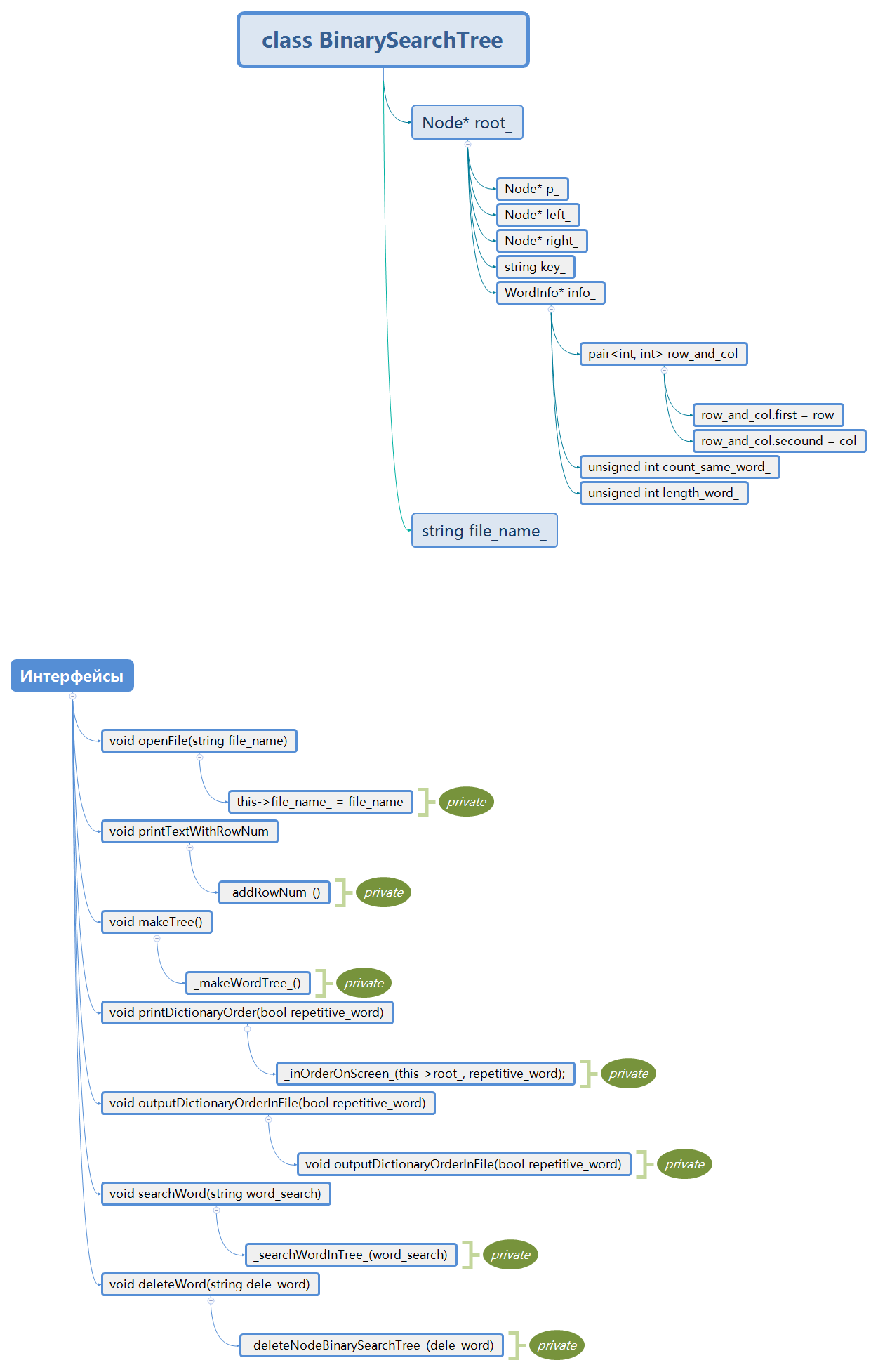


Рис. 12

## 2. Анализ алгоритма

### 2.1 Методы двоичного дерева поиска

В информатике двоичное дерево поиска представляет собой комбинацию абстрактных структур данных дерево поиска и двоичное дерево.

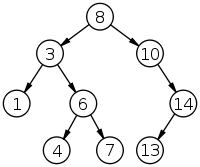
[](https://commons.wikimedia.org/wiki/File:Binary_search_tree.svg?uselang=ru)

Рис. 11 Пример двоичного дерева поиска

**Двоичное дерево поиска** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *binary search tree*, BST) — это [двоичное дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

* Оба поддерева — левое и правое — являются двоичными деревьями поиска.
* У всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше либо равны*, нежели значение ключа данных самого узла X.
* У всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше либо равны*, нежели значение ключа данных самого узла X.

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения *меньше*.

Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных. Однако это касается реализации, а не природы двоичного дерева поиска.

Для целей реализации двоичное дерево поиска можно определить так:

* Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) — ссылки на родительский элемент.
* Данные (data) обладают ключом (key), на котором определена операция сравнения «меньше». В конкретных реализациях это может быть пара (key, value) — (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
* Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], то есть ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

#### 2.1.1 Добавление элемента (INSERT)

**Дано**: дерево Тree и Key.

**Задача**: вставить Key в дерево Тree (при совпадении Key).

**Сложность:** O(log n) в среднем; O(n) в худшем случае.

**Алгоритм**:

1. Если дерево пусто, заменить его на дерево с одним корневым узлом (Key, null, null, null) и остановиться.
2. Иначе сравнить Key с ключом корневого узла X.
   * Если Key > X, рекурсивно добавить (Key) в правое поддерево Тree.
   * Если Key < X, рекурсивно добавить (Key) в левое поддерево Тree.
   * Если Key = X, запишите информацию о повторении второго слова в info\_.
3. /\*
4. \*\* Insert one word into the binary search tree
5. \*/
6. **void** \_insertNode\_(Node\* nodeInsert, unsigned **int** count\_word\_text)
7. {
8. **if** (nullptr == root\_ || nullptr == nodeInsert) **throw** MyErrorInfo("nullptr\_root\_or\_nodeInsert\_in\_insertNode()");
10. **if** (1 == count\_word\_text)
11. {
12. root\_->p\_ = nullptr;
13. root\_->left\_ = nullptr;
14. root\_->right\_ = nullptr;
15. root\_->key\_ = nodeInsert->key\_;
16. root\_->info\_ = nodeInsert->info\_;
17. **return**;
18. }
20. Node\* pCurrent = root\_;
22. **while** (**true**)
23. {
24. **while** (pCurrent->key\_ > nodeInsert->key\_ && pCurrent->left\_ != nullptr)
25. {
26. pCurrent = pCurrent->left\_;
27. }
28. **while** (pCurrent->key\_ < nodeInsert->key\_ && pCurrent->right\_ != nullptr)
29. {
30. pCurrent = pCurrent->right\_;
31. }
33. **if** (pCurrent->key\_ > nodeInsert->key\_ && pCurrent->left\_ == nullptr) **break**;
34. **else** **if** (pCurrent->key\_ < nodeInsert->key\_ && pCurrent->right\_ == nullptr) **break**;
35. **else** **if** (pCurrent->key\_ == nodeInsert->key\_) **break**;
36. }
38. **if** (pCurrent->key\_ > nodeInsert->key\_)
39. {
40. pCurrent->left\_ = nodeInsert;
41. nodeInsert->p\_ = pCurrent;
42. }
43. **else** **if** (pCurrent->key\_ < nodeInsert->key\_)
44. {
45. pCurrent->right\_ = nodeInsert;
46. nodeInsert->p\_ = pCurrent;
47. }
48. **else**
49. {
50. **throw** MyErrorInfo("error\_insert\_in\_insertNode()");
51. }
52. }

#### 2.1.2 Поиск элемента (SEARCH)

**Дано**: дерево Тree и ключ Key.

**Задача**: проверить, есть ли узел с ключом Key в дереве Тree, и если да, то вернуть ссылку на этот узел.

**Сложность:** O(log2n) в среднем; O(n) в худшем случае.

**Алгоритм**:

1. Если дерево пусто, сообщить, что узел не найден, и остановиться.
2. Иначе сравнить Key со значением ключа корневого узла X.
   * Если Key = X, выдать ссылку на этот узел и остановиться.
   * Если Key > X, рекурсивно искать ключ Key в правом поддереве Тree.
   * Если Key < X, рекурсивно искать ключ Key в левом поддереве Тree.
3. Node\* \_searchWordInTree\_(string word\_search)
4. {
5. **if** (nullptr == root\_) **return** nullptr;
7. Node\* pCurrent = **this**->root\_;
9. **while** (nullptr != pCurrent)
10. {
11. **if** (word\_search == pCurrent->key\_) **return** pCurrent;
13. **while** (nullptr != pCurrent && word\_search < pCurrent->key\_)
14. {
15. pCurrent = pCurrent->left\_;
16. }
18. **while** (nullptr != pCurrent && word\_search > pCurrent->key\_)
19. {
20. pCurrent = pCurrent->right\_;
21. }
22. }
23. **return** nullptr;
24. }

#### 2.1.3 Удаление узла (DELETE)

**Дано**: дерево Тree с корнем n и ключом Key.

**Задача**: удалить из дерева Тree узел с ключом Key (если такой есть).

**Сложность:**O(log N) в среднем; O(n) в худшем случае.

**Алгоритм**:

1. Если дерево Tree пусто, остановиться;
2. Иначе сравнить Key с ключом X корневого узла n.
   * Если Key > X, рекурсивно удалить Key из правого поддерева Тree;
   * Если Key < X, рекурсивно удалить Key из левого поддерева Тree;
   * Если Key = X, то необходимо рассмотреть три случая.
     + Если обоих детей нет, то удаляем текущий узел и обнуляем ссылку на него у родительского узла;
     + Если одного из детей нет, то значения полей ребёнка m ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом m;
     + Если оба ребёнка присутствуют, то
       - Если левый узел m правого поддерева отсутствует (n->right->left)
         * Копируем из правого узла в удаляемый поля Key
       - Иначе
         * Возьмём самый левый узел m, правого поддерева n->right;
         * Скопируем данные (кроме ссылок на дочерние элементы) из m в n;
         * Рекурсивно удалим узел m.
3. **bool** \_deleteNodeBinarySearchTree\_(string dele\_key)
4. {
5. **if** (nullptr == root\_) **throw** MyErrorInfo("nullptr\_root\_in\_DeleteNodeBinarySearchTree(string key\_del)");
7. // First of all, need to find the node, which need to delete
8. Node\* pCurrent = root\_;
9. Node\* pParent = pCurrent;
10. **while** (**true**)
11. {
12. **if** (pCurrent->key\_ > dele\_key)
13. {
14. pParent = pCurrent;
15. pCurrent = pCurrent->left\_;
16. }
17. **else** **if** (pCurrent->key\_ < dele\_key)
18. {
19. pParent = pCurrent;
20. pCurrent = pCurrent->right\_;
21. }
22. **else** **if** (pCurrent->key\_ == dele\_key)
23. {
24. **break**;
25. }
27. **if** (pCurrent == nullptr)
28. {
29. **return** **false**;
30. }
31. }
33. // if delete node is the leaf
34. **if** (pCurrent->left\_ == nullptr && pCurrent->right\_ == nullptr)
35. {
36. **if** (pParent->left\_ == pCurrent)
37. {
38. pParent->left\_ = nullptr;
39. }
40. **else** pParent->right\_ = nullptr;
42. **delete** pCurrent;
43. pCurrent = nullptr;
44. **return** **true**;
45. }
47. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
48. S  <-- pParent
49. /  \
50. DELETE-> E      X
51. /   \
52. A    null
53. /  \
54. \*    \*
55. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
56. **if** (pCurrent->left\_ != nullptr && pCurrent->right\_ == nullptr && pCurrent == pParent->left\_)
57. {
58. Node\* deleNode = pCurrent;
59. pParent->left\_ = deleNode->right\_;
60. **delete** deleNode;
61. deleNode = nullptr;
62. **return** **true**;
63. }

66. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
67. S  <-- pParent
68. /  \
69. DELETE-> E      X
70. /   \
71. null    R
72. /  \
73. \*    \*
74. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
75. **else** **if** (pCurrent->left\_ == nullptr && pCurrent->right\_ != nullptr && pCurrent == pParent->left\_)
76. {
77. Node\* deleNode = pCurrent;
78. pParent->left\_ = pCurrent->right\_;
79. **delete** deleNode;
80. deleNode = nullptr;
81. **return** **true**;
82. }

85. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
86. S  <-- pParent
87. /  \
88. E      X <-DELETE
89. /  \
90. Q   null
91. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
92. **else** **if** (pCurrent->left\_ != nullptr && pCurrent->right\_ == nullptr && pCurrent == pParent->right\_)
93. {
94. Node\* deleNode = pCurrent;
95. pParent->right\_ = deleNode->left\_;
96. **delete** deleNode;
97. deleNode = nullptr;
98. **return** **true**;
99. }

102. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
103. S  <-- pParent
104. /  \
105. E      X <-DELETE
106. /  \
107. null  Z
108. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
109. **else** **if** (pCurrent->left\_ == nullptr && pCurrent->right\_ != nullptr && pCurrent == pParent->right\_)
110. {
111. Node\* deleNode = pCurrent;
112. pParent->right\_ = deleNode->right\_;
113. **delete** deleNode;
114. deleNode = nullptr;
115. **return** **true**;
116. }

119. // 最复杂的情况 - The WORST situation
120. **else** **if** (pCurrent->left\_ != nullptr && pCurrent->right\_ != nullptr)
121. {
122. Node\* deleNode = pCurrent;
123. pCurrent = pCurrent->right\_;
125. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
126. S  <-- pParent
127. /  \
128. DELETE-> E      X
129. /   \
130. A     R  <- pCurrent
131. /  \
132. \*    \*
133. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
134. **if** (pCurrent->left\_ == nullptr && pCurrent->right\_ == nullptr)
135. {
136. deleNode->key\_ = pCurrent->key\_;
137. deleNode->right\_ = nullptr;
139. **delete** pCurrent;
140. pCurrent = nullptr;
141. **return** **true**;
142. }

145. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
146. S  <-- pParent
147. /  \
148. DELETE-> E      X
149. /     \
150. A        R <- pCurrent
151. /  \     /   \
152. \*    \*  H      \*
153. /   \
154. pParent --->  \*      \*  ( pParent after "while")
155. /
156. G <---- pCurrent  (之后的pCurrent - pCurrent after "while")
157. /   \
158. null   \*
159. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
160. **if** (pCurrent->left\_ != nullptr)
161. {
162. **while** (pCurrent->left\_ != nullptr)
163. {
164. pParent = pCurrent;
165. pCurrent = pCurrent->left\_;
166. }
168. deleNode->key\_ = pCurrent->key\_;
169. pParent->left\_ = pCurrent->right\_;
170. **delete** pCurrent;
171. pCurrent = nullptr;
172. }
173. **return** **true**;
174. }
175. }

### 2.2 Двоичное дерево поиска

### Время выполнения алгоритма, использующего двоичное дерево поиска, зависит от формы дерева, которая, в свою очередь, зависит от порядка вставки ключей. В лучшем случае полностью сбалансировано дерево с узлами N. Расстояние между каждым пустым звеном и корневым узлом составляет ~logN. В худшем случае в пути поиска может быть N узлов. Как показано на Рис.12. Но в целом форма дерева ближе к лучшему.

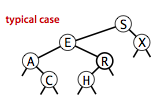


Рис.13

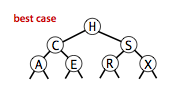


Рис. 12

Для многих случай, показанная на Рис.12 - Рис.14: мы предполагаем, что распределение ключей (равномерно) случайное, или что их порядок вставки является случайным. Для анализа этой модели бинарное дерево поиска и быстрая сортировка почти «близнецы». Корневой узел дерева является первым разделенным элементом в быстрой сортировке (ключи слева меньше его, а ключи справа больше его), и это также относится ко всем поддеревьям. Рекурсивная сортировка подмассивов точно соответствует. Это позволяет нам анализировать некоторые свойства бинарного дерева поиска.

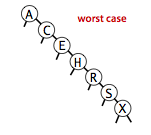


Рис.14

В бинарном дереве поиска, построенном из N случайных ключей, среднее число сравнений, необходимых для поиска совпадения, составляет ~ 2InN (около 1,3919N).

**Доказать :**

Количество сравнений, необходимых для поиска совпадений, который заканчивается в данном узле, равно глубине пути поиска плюс один. Если мы сложим глубину всех узлов в дереве, мы можем получить длину внутреннего пути дерева. Следовательно, среднее число сравнений во втором дереве поиска равно средней длине внутреннего пути плюс один. Пусть *CN* - длина внутреннего пути двоичного дерева поиска, построенного из N случайно упорядоченных разных ключей, тогда средняя стоимость поискового попадания равна (*1 + CN / N*). У нас есть *C0=C1=0*, и для N > 1 мы можем напрямую получить индуктивные отношения в соответствии с рекурсивной структурой второго и второго дерева поиска:

*CN = N-1 + (C0 + CN-1) / N + (C1 + CN-2) / N + … + (CN-1 + C0) / N*

Где N-1 означает корневой узел, так что все N-1 некорневые узлы в дереве имеют 1 добавленный к пути. Другие элементы выражения представляют все поддеревья, и их метод вычисления такой же, как и у дерева двоичного поиска размера N. После сортировки выражений мы обнаружим, что эта индуктивная формула почти совпадает с формулой, которую получили для быстрой сортировки, поэтому мы также можем получить *CN~2NInN.*

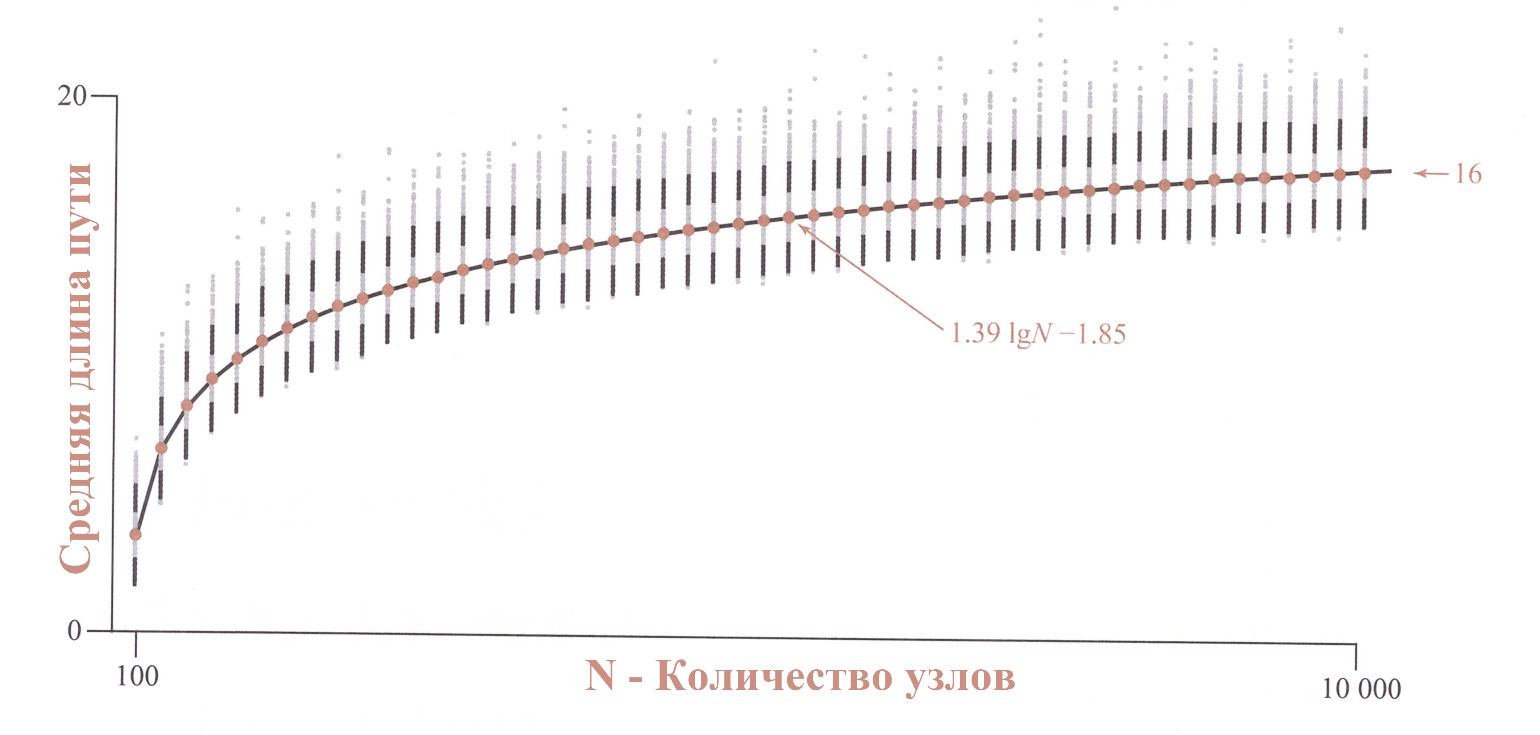


Таблица 1 Средняя длина пути от корня до любого узла в случайно построенном бинарном дереве поиска

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Алгоритм | Средняя ситуация | | Худшая ситуация | |
| найти | вставлять | найти успех | вставлять |
| Последовательный запрос (неупорядоченный связанный список) | *N* | *N* | *N/2* | *N* |
| Бинарный поиск (упорядоченный массив) | *lgN* | *N* | *lgN* | *N/2* |
| Бинарное дерево поиска | *N* | *N* | *1.39lgN* | *1.39lgN* |

Таблица 2 Краткое описание

## Описание спецификации программы (детальные требования)

1). Файл должен быть открыт

* + 1. В противном случае выводим ошибку, содержащую текст: «cant\_open\_the\_file»

2). root, который в бинарном дереве поиска нельзя было nullptr

3). Все данные должны иметь правильный тип

1. Поля Номер строки(row), столбца(col), счетчик(count) и длина слова(length) должны иметь тип данных unsigned int
2. Номер строки, столбца и длина слова должны больше чем 1

4). Если пустая строка пропускается, счетчик не накапливается

5). Для повторяющих слов

1. В одной статье может быть несколько повторяющихся слов,и номера строк и столбцов этих слов должны храниться в определенном контейнере
2. При печати функции(printWordAndInfo), которая выводит слова, должен быть предоставлен внешний интерфейс, чтобы пользователь мог выбрать, следует ли печатать повторяющиеся слова
   * + 1. Если «bool repetitive\_word = true»: выводит все повторяющиеся слова вместе с их информацией
       2. Если «bool repetitive\_word = false»: выводит это слово, номер строк и столбцов, которые появляются в первый раз

6). Для одного и того же слова, но с большой буквы или с “ ’s ”. Все буквы должны быть преобразованы в маленькую букву.

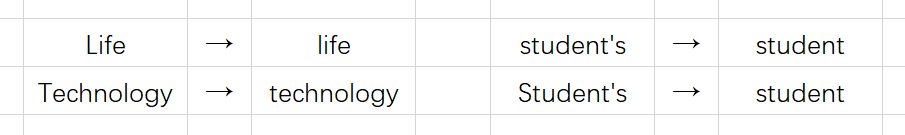
**Пример:**

Рис.13

## Описание программы

### 4.1 Структура программы

Рис. 14

### 4.2 Форматы входных

Программа считывает данные из текстового документа. В текстовом документе должно содержать разветвленную статью. Для того, чтобы программа могла правильно читать текстовый документ:

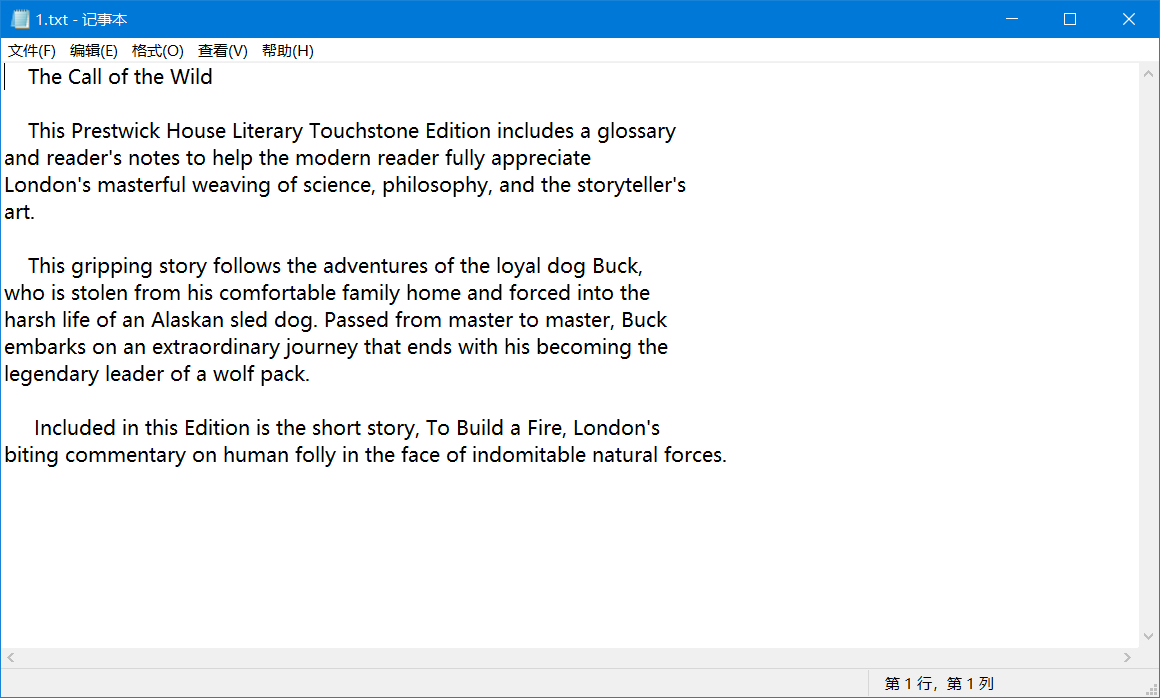
* в имени файла не должно быть недопустимых символов
* расширение должно быть".txt".
* слова в статье должны быть разделены хотя бы одним пробелом (рекомендуется использовать один пробел）

Рис. 15

### 4.3 Выходных данных

Для удобства чтения и отладки программа будет разделена на две части для вывода, а результаты будут распечатаны на экране и выведены в файл соответственно.

Первая часть выводит статьи с номерами строк и статистикой.

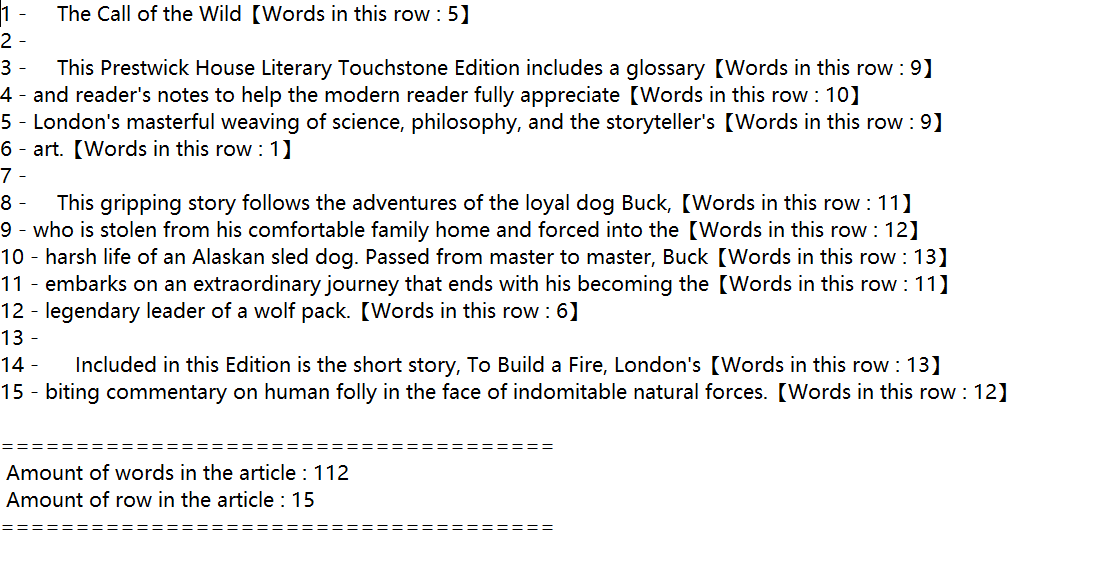


Рис. 16 первая часть

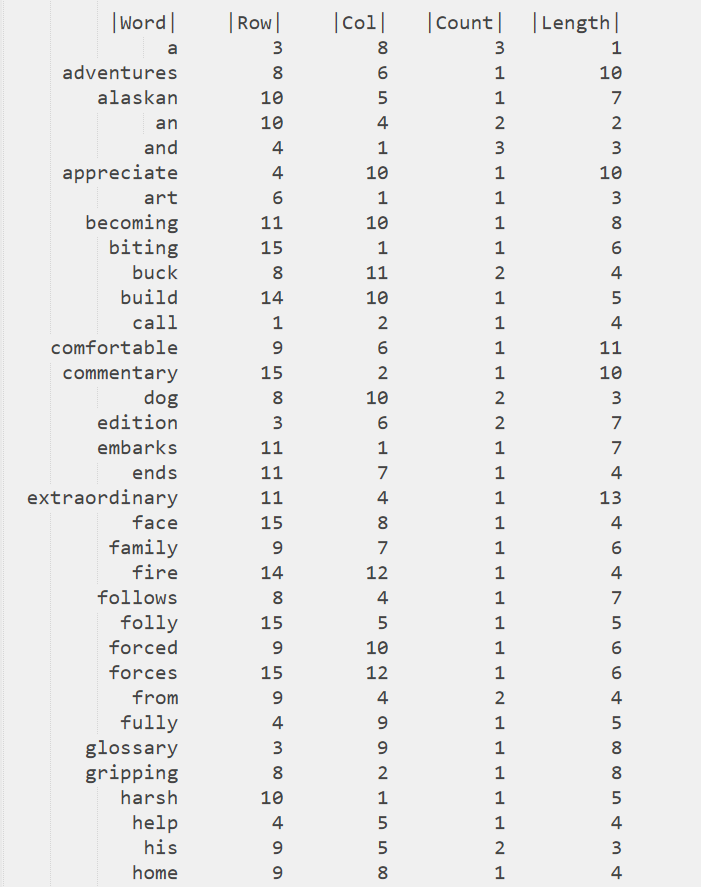
Вторая часть выводит слова из статьи в лексикографическом порядке, включая номер строки, номер столбца, количество вхождений, длину и т. д.

Рис. 17 вторая часть

# Заключение

Чтобы выполнить требования по данной теме, я разработал класс BinarySearchTree, который использует структуру данных двоичного дерева поиска. Чтобы соответствовать требованиям этой темы, были также разработаны другие методы классификации. Слова из текстового документа вставляются в двоичное дерево поиска путем вызова функции \_insertNode\_(Node\* nodeInsert, unsigned int count\_word\_text), которая содержит номер строки, номер столбца, количество вхождений, длину в каждом узле двоичного дерева поиска. Эта программа проверяет эффективность структуры данных бинарного дерева поиска. В соответствии с определением двоичного дерева поиска реализуются такие методы, как рекурсивный обход, обход стека, поиск, вставка, удаление узлов и т. д.

# Список использованных источников

Двоичное дерево поиска. (б.д.).

https://ru.wikipedia.org/wiki/Двоичное\_дерево\_поиска

Algorithms Fourth Edition

Robert Sedgewich, Kevin Wayne

# Приложение 1. Текст программы

* BinarySearchTree.hpp

1. #pragma once
2. #include "MyErrorInfo.hpp"
3. #include "WordInfo.hpp"
4. #include "LinkStack.hpp"
5. #include <iostream>
6. #include <string>
7. #include <iomanip>
8. #include <fstream>
10. **class** BinarySearchTree;
11. **struct** Node;
12. **class** WordInfo;
14. **class** BinarySearchTree
15. {
16. **private**:
17. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Struct - Node\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
18. **struct** Node
19. {
20. Node() : key\_("null"), p\_(nullptr), left\_(nullptr), right\_(nullptr), info\_(nullptr) {}
21. Node(string key, WordInfo\* info) : key\_(key), info\_(info), left\_(nullptr), right\_(nullptr), p\_(nullptr) {}
22. Node(string key, WordInfo\* info, Node\* left, Node\* right, Node\* p) : key\_(key), info\_(info), left\_(left), right\_(right), p\_(p) {}
24. string key\_; // word
25. WordInfo\* info\_;
26. Node\* left\_;
27. Node\* right\_;
28. Node\* p\_;
29. };
31. **struct** BinTreeStackNode
32. {
33. Node\* root;
34. **bool** flag;
35. };
37. Node\* root\_;
38. string file\_name\_;
40. **public**:
42. BinarySearchTree()
43. {
44. **this**->root\_ = **new** Node;
45. }
47. ~BinarySearchTree()
48. {
49. \_freeSpaceBinaryTree\_(**this**->root\_);
50. }
52. **void** openFile(string file\_name)
53. {
54. **this**->file\_name\_ = file\_name;
55. }
57. **void** printTextWithRowNum()
58. {
59. \_addRowNum\_();
60. }
62. **void** makeTree()
63. {
64. \_makeWordTree\_();
65. }
67. /\*
68. \*\* print the words on screen
69. \*\* true - print same word;
70. \*\* false - do not print same word
71. \*/
72. **void** printDictionaryOrder(**bool** repetitive\_word)
73. {
74. **if** (nullptr == **this**->root\_) **throw** MyErrorInfo("nullptr\_root\_in\_foreachDictionaryOrder()");
76. {
77. cout.width(15); cout << "|Word|";
78. cout.width(9); cout << "|Row|";
79. cout.width(9); cout << "|Col|";
80. cout.width(10); cout << "|Count|";
81. cout.width(10); cout << "|Length|";
82. cout << endl;
83. }
85. \_inOrderOnScreen\_(**this**->root\_, repetitive\_word);
86. }
88. /\*
89. \*\* out put the word by Dictionary order in to the file "output.txt"
90. \*/
91. **void** outputDictionaryOrderInFile(**bool** repetitive\_word)
92. {
93. **if** (nullptr == **this**->root\_) **throw** MyErrorInfo("nullptr\_root\_in\_outputDictionaryOrderInFile(bool repetitive\_word)");
95. ofstream ofs;
96. ofs.open("./output.txt", ios::app);  // ::app
97. **if** (!ofs.is\_open()) **throw** MyErrorInfo("cant\_open\_the\_file\_output\_in\_outputDictionaryOrderInFile(bool repetitive\_word)");
99. {
100. ofs.width(15); ofs << "|Word|";
101. ofs.width(9); ofs << "|Row|";
102. ofs.width(9); ofs << "|Col|";
103. ofs.width(10); ofs << "|Count|";
104. ofs.width(10); ofs << "|Length|";
105. ofs << endl;
106. }
108. \_inOrderOutputInFile\_(**this**->root\_, ofs, repetitive\_word);
109. cout << endl << "Successful output in 【output.txt】!" << endl;
110. }
112. /\*
113. \*\* Seach word by cin
114. \*/
115. **void** searchWord(string word\_search)
116. {
117. Node\* searchWord = \_searchWordInTree\_(word\_search);
118. **if** (nullptr == searchWord)
119. {
120. cout << endl
121. << "======Search Word======" << endl
122. << "\033[31m" << " Don't  have word 【" << word\_search << "】!" << "\033[0m" << endl <<endl;
123. }
124. **else**
125. {
126. cout << endl
127. << "======Search Word======" << endl
128. << "\033[32m" << "Word : " << searchWord->key\_ << endl
129. << "Row : " << searchWord->info\_->header\_.header\_->next\_->data\_.first << endl
130. << "Col : " << searchWord->info\_->header\_.header\_->next\_->data\_.second << endl
131. << "\033[0m" << endl;
132. }
133. }
135. /\*
136. \*\* Delete word by cin
137. \*/
138. **void** deleteWord(string dele\_word)
139. {
140. **bool** is\_dele = \_deleteNodeBinarySearchTree\_(dele\_word);
142. cout << "======Delete Word======" << endl;
143. **if** (**true** == is\_dele)
144. {
145. cout << endl
146. <<  "\033[32m"
147. << "Succeed delete word 【" << dele\_word << "】!"
148. << "\033[0m"
149. << endl;
150. }
151. **else**
152. {
153. cout << endl
154. << "\033[31m"
155. << "Delete fail, don't have word 【" << dele\_word << "】!"
156. << "\033[0m"
157. << endl;
158. }
159. }
161. // ===================================================================================
163. **private**:
165. /\*
166. \*\* Insert one word into the binary search tree
167. \*/
168. **void** \_insertNode\_(Node\* nodeInsert, unsigned **int** count\_word\_text)
169. {
170. **if** (nullptr == root\_ || nullptr == nodeInsert) **throw** MyErrorInfo("nullptr\_root\_or\_nodeInsert\_in\_insertNode()");
172. **if** (1 == count\_word\_text)
173. {
174. root\_->p\_ = nullptr;
175. root\_->left\_ = nullptr;
176. root\_->right\_ = nullptr;
177. root\_->key\_ = nodeInsert->key\_;
178. root\_->info\_ = nodeInsert->info\_;
179. **return**;
180. }
182. Node\* pCurrent = root\_;
184. **while** (**true**)
185. {
186. **while** (pCurrent->key\_ > nodeInsert->key\_ && pCurrent->left\_ != nullptr)
187. {
188. pCurrent = pCurrent->left\_;
189. }
190. **while** (pCurrent->key\_ < nodeInsert->key\_ && pCurrent->right\_ != nullptr)
191. {
192. pCurrent = pCurrent->right\_;
193. }
195. **if** (pCurrent->key\_ > nodeInsert->key\_ && pCurrent->left\_ == nullptr) **break**;
196. **else** **if** (pCurrent->key\_ < nodeInsert->key\_ && pCurrent->right\_ == nullptr) **break**;
197. **else** **if** (pCurrent->key\_ == nodeInsert->key\_) **break**;
198. }
200. **if** (pCurrent->key\_ > nodeInsert->key\_)
201. {
202. pCurrent->left\_ = nodeInsert;
203. nodeInsert->p\_ = pCurrent;
204. }
205. **else** **if** (pCurrent->key\_ < nodeInsert->key\_)
206. {
207. pCurrent->right\_ = nodeInsert;
208. nodeInsert->p\_ = pCurrent;
209. }
210. **else**
211. {
212. **throw** MyErrorInfo("error\_insert\_in\_insertNode()");
213. }
214. }
216. /\*
217. \*\* Insert all word into the binary search tree
218. \*/
219. **void** \_makeWordTree\_()
220. {
221. // read data(text)
222. ifstream ifs\_word, ifs\_row;
224. ifs\_row.open(file\_name\_, ios::in);
225. **if** (!ifs\_row.is\_open()) **throw** MyErrorInfo("cant\_open\_the\_file\_ifs\_row\_in\_makeWordTree()");
227. ifs\_word.open(file\_name\_, ios::in);
228. **if** (!ifs\_word.is\_open()) **throw** MyErrorInfo("cant\_open\_the\_file\_ifs\_word\_in\_makeWordTree()");
230. string temp\_word, temp\_line;
231. unsigned **int** row\_word = 0, col\_word = 0, row\_length = 0, count\_word\_text = 0;
232. **while** (getline(ifs\_row, temp\_line))
233. {
234. row\_word++;
235. col\_word = 0;
236. row\_length = 0;
238. /\*
239. \*\* If a blank line, skip, the counter does not accumulate
240. \*/
241. **if** (0 == temp\_line.length())
242. {
243. **continue**;
244. }
245. \_clearAllSpace\_(temp\_line);
247. **while** (!ifs\_word.eof())
248. {
249. ifs\_word >> temp\_word;
250. col\_word++;
251. count\_word\_text++;
252. row\_length += temp\_word.length();
253. \_changeCaseAndClearSign\_(temp\_word);
254. pair<unsigned **int**, unsigned **int**> row\_col\_word = make\_pair(row\_word, col\_word);
256. /\*
257. \*\* Insert to binary search tree
258. \*/
259. **if** (!\_isHaveSameWord\_( temp\_word, row\_col\_word)) // First determine if there are identical words in tree
260. {
261. WordInfo\* newWordInfo = **new** WordInfo;
262. newWordInfo->header\_.insert(row\_col\_word);
263. newWordInfo->count\_same\_word\_ = 1;
264. newWordInfo->length\_word\_ = temp\_word.length();
266. Node\* newNode = **new** Node;
267. newNode->key\_ = temp\_word;
268. newNode->info\_ = newWordInfo;
269. \_insertNode\_(newNode, count\_word\_text);
270. }
271. **if** (temp\_line.length() == row\_length) // Broken line
272. {
273. **break**;
274. }
275. }
276. }
277. ifs\_row.close();
278. ifs\_word.close();
279. }
281. **void** \_addRowNum\_()
282. {
283. /\*
284. \*\* read data(text)
285. \*/
286. string temp\_word, temp\_line;
287. ifstream ifs\_word, ifs\_row;
289. ifs\_row.open(file\_name\_, ios::in);
290. **if** (!ifs\_row.is\_open()) **throw** MyErrorInfo("cant\_open\_the\_file\_ifs\_row\_in\_addRowNum()");
292. ifs\_word.open(file\_name\_, ios::in);
293. **if** (!ifs\_word.is\_open()) **throw** MyErrorInfo("cant\_open\_the\_file\_ifs\_word\_in\_addRowNum()");
295. ofstream ofs;
296. ofs.open("./output.txt", ios::out);  // ::out
297. **if** (!ofs.is\_open()) **throw** MyErrorInfo("cant\_open\_the\_file\_output\_in\_addRowNum()");
299. unsigned **int** row\_length = 0, row\_num = 1, count\_word\_text = 0, count\_word\_row = 0;
300. **while** (getline(ifs\_row, temp\_line))
301. {
302. row\_length = 0;
303. count\_word\_row = 0;
304. ofs << row\_num++ << " - " << temp\_line;
305. \_clearAllSpace\_(temp\_line);
306. // row\_length = temp\_line.length();
308. /\*
309. \*\* If a blank line, skip, the counter does not accumulate
310. \*/
311. **if** (0 == temp\_line.length())
312. {
313. ofs << endl << flush;
314. **continue**;
315. }
317. **while** (!ifs\_word.eof())
318. {
319. ifs\_word >> temp\_word;
320. count\_word\_row++;
321. row\_length += temp\_word.length();
322. **if** (temp\_line.length() == row\_length)  // Broken line
323. {
324. ofs << "【Words in this row : " << count\_word\_row << "】" << endl;
325. **break**;
326. }
327. }
328. count\_word\_text += count\_word\_row;
329. }
331. ofs << endl
332. << "=====================================" << endl
333. << " Amount of words in the article : " << count\_word\_text << endl
334. << " Amount of row in the article : " << row\_num - 1 << endl
335. << "=====================================" << endl << endl;
336. ifs\_row.close();
337. ifs\_word.close();
338. ofs.close();
339. }
341. /\*
342. \*\*  print on screen
343. \*/
344. **void** \_inOrderOnScreen\_(Node\* root, **bool** repetitive\_word)
345. {
346. **if** (nullptr == root) **return**;
348. \_inOrderOnScreen\_(root->left\_, repetitive\_word);
350. **if** (**true** == repetitive\_word)
351. {
352. // ------------------------print same word
353. **if** (1 == root->info\_->count\_same\_word\_)
354. {
355. cout.width(15); cout << root->key\_;
356. cout.width(9); cout << root->info\_->header\_.header\_->next\_->data\_.first; // first - row
357. cout.width(9); cout << root->info\_->header\_.header\_->next\_->data\_.second; // secound - col
358. cout.width(10); cout << root->info\_->count\_same\_word\_;
359. cout.width(10); cout << root->info\_->length\_word\_;
360. cout << endl;
361. }
362. **else**
363. {
364. LinkNode<pair<**int**, **int**>>\* pCurrent = root->info\_->header\_.header\_->next\_;
365. **for** (**int** i = 0; i < root->info\_->count\_same\_word\_; i++)
366. {
367. cout.width(15); cout << root->key\_;
368. cout.width(9); cout << pCurrent->data\_.first;
369. cout.width(9); cout << pCurrent->data\_.second;
370. cout.width(10); cout << root->info\_->count\_same\_word\_;
371. cout.width(10); cout << root->info\_->length\_word\_;
372. cout << endl;
373. pCurrent = pCurrent->next\_;
374. }
375. }
376. // ------------------------
377. }
378. **else**
379. {
380. cout.width(15); cout << root->key\_;
381. cout.width(9); cout << root->info\_->header\_.header\_->next\_->data\_.first;
382. cout.width(9); cout << root->info\_->header\_.header\_->next\_->data\_.second;
383. cout.width(10); cout << root->info\_->count\_same\_word\_;
384. cout.width(10); cout << root->info\_->length\_word\_;
385. cout << endl;
386. }
387. \_inOrderOnScreen\_(root->right\_, repetitive\_word);
388. }
390. /\*
391. \*\*  output in file [output.txt]
392. \*/
393. **void** \_inOrderOutputInFile\_(Node\* root, ofstream& ofs, **bool** repetitive\_word)
394. {
395. **if** (nullptr == root) **return**;
397. \_inOrderOutputInFile\_(root->left\_, ofs, repetitive\_word);
399. **if** (**true** == repetitive\_word)
400. {
401. // ------------------------
402. **if** (1 == root->info\_->count\_same\_word\_)
403. {
404. ofs << setw(15) << root->key\_
405. << setw(9) << root->info\_->header\_.header\_->next\_->data\_.first // first - row
406. << setw(9) << root->info\_->header\_.header\_->next\_->data\_.second // secound - col
407. << setw(10) << root->info\_->count\_same\_word\_
408. << setw(10) << root->info\_->length\_word\_
409. << endl;
410. }
411. **else**
412. {
413. LinkNode<pair<**int**, **int**>>\* pCurrent = root->info\_->header\_.header\_->next\_;
414. **for** (**int** i = 0; i < root->info\_->count\_same\_word\_; i++)
415. {
416. ofs << setw(15) << root->key\_
417. << setw(9) << pCurrent->data\_.first
418. << setw(9) << pCurrent->data\_.second
419. << setw(10) << root->info\_->count\_same\_word\_
420. << setw(10) << root->info\_->length\_word\_
421. << endl;
422. pCurrent = pCurrent->next\_;
423. }
424. }
425. // ------------------------
426. }
427. **else**
428. {
429. ofs << setw(15) << root->key\_
430. << setw(9) << root->info\_->header\_.header\_->next\_->data\_.first
431. << setw(9) << root->info\_->header\_.header\_->next\_->data\_.second
432. << setw(10) << root->info\_->count\_same\_word\_
433. << setw(10) << root->info\_->length\_word\_
434. << endl;
435. }
436. \_inOrderOutputInFile\_(root->right\_, ofs, repetitive\_word);
437. }
439. /\*
440. \*\* Delete all the " "(space), (,), (.), ('s)
441. \*\* Change case " Student -> student"
442. \*\* If have word : "Student's" change it to "student"
443. \*/
444. **void** \_changeCaseAndClearSign\_(string& word)
445. {
446. **if** (word.empty()) **throw** MyErrorInfo("null\_string\_in\_changeCaseAndClearSign(string& word)");
447. **if** (word.at(0) > 64 && word.at(0) < 91)
448. {
449. word.at(0) += 32; // use ASCII to change case
450. }
452. **int** index = 0;
453. **while** (word.find(',', index) != string::npos)
454. {
455. index = word.find(',', index);
456. word.erase(index, 1);
457. }
459. **while** (word.find('.', index) != string::npos)
460. {
461. index = word.find('.', index);
462. word.erase(index, 1);
463. }
465. **while** (word.find("'s", index) != string::npos)
466. {
467. index = word.find("'s", index);
468. word.erase(index, 2);
469. }
470. }
472. /\*
473. \*\*  Creat element in stack
474. \*/
475. BinTreeStackNode\* \_CreatBinTreeStackNode\_(Node\* node, **bool** flag)
476. {
477. BinTreeStackNode\* newNode = **new** BinTreeStackNode;
479. newNode->root = node;
480. newNode->flag = **false**;
481. **return** newNode;
482. }
484. /\*
485. \*\*  【Use Stack】Does the same word exist in the tree, if have change "header\_.insert(pair)" in WordInfo
486. \*/
487. **bool** \_isHaveSameWord\_(string word, pair<unsigned **int**, unsigned **int**> row\_col\_word)
488. {
489. **if** (nullptr == root\_) **return** **false**;
491. // Make a Stack
492. LinkStack<BinTreeStackNode\*> stack;
494. // first, make the root of the tree in the stack
495. stack.push(\_CreatBinTreeStackNode\_(root\_, **false**));
497. **while** (stack.size() > 0)
498. {
499. // Pop the element, which is on the top of the stack
500. BinTreeStackNode\* node = (BinTreeStackNode\*)stack.top();
501. stack.pop();
503. // if nullptr in stack ->continue
504. **if** (node->root == nullptr) **continue**;
506. **if** (node->flag == **true**)
507. {
508. **if** (word == node->root->key\_)
509. {
510. node->root->info\_->header\_.insert(row\_col\_word);
511. node->root->info\_->count\_same\_word\_++;
512. **return** **true**;
513. }
514. }
515. **else** {
516. // put the child of the node in the stack
517. stack.push(\_CreatBinTreeStackNode\_(node->root->right\_, **false**));
518. stack.push(\_CreatBinTreeStackNode\_(node->root->left\_, **false**));
519. node->flag = **true**;
520. stack.push(node);
521. }
522. }
523. **return** **false**;
524. }
526. Node\* \_searchWordInTree\_(string word\_search)
527. {
528. **if** (nullptr == root\_) **return** nullptr;
530. Node\* pCurrent = **this**->root\_;
532. **while** (nullptr != pCurrent)
533. {
534. **if** (word\_search == pCurrent->key\_)
535. {
536. **return** pCurrent;
537. }
539. **while** (nullptr != pCurrent && word\_search < pCurrent->key\_)
540. {
541. pCurrent = pCurrent->left\_;
542. }
544. **while** (nullptr != pCurrent && word\_search > pCurrent->key\_)
545. {
546. pCurrent = pCurrent->right\_;
547. }
548. }
550. **return** nullptr;
551. }
553. /\*
554. \*\* Delete all the " "(space)
555. \*/
556. string& \_clearAllSpace\_(string& str)
557. {
558. **int** index = 0;
560. **if** (!str.empty())
561. {
562. **while** ((index = str.find(' ', index)) != string::npos)
563. {
564. str.erase(index, 1);
565. }
566. }
567. **return** str;
568. }
570. **bool** \_deleteNodeBinarySearchTree\_(string dele\_key)
571. {
572. **if** (nullptr == root\_) **throw** MyErrorInfo("nullptr\_root\_in\_DeleteNodeBinarySearchTree(string key\_del)");
574. // First of all, need to find the node, which need to delete
575. Node\* pCurrent = root\_;
576. Node\* pParent = pCurrent;
578. **while** (**true**)
579. {
580. **if** (pCurrent->key\_ > dele\_key)
581. {
582. pParent = pCurrent;
583. pCurrent = pCurrent->left\_;
584. }
585. **else** **if** (pCurrent->key\_ < dele\_key)
586. {
587. pParent = pCurrent;
588. pCurrent = pCurrent->right\_;
589. }
590. **else** **if** (pCurrent->key\_ == dele\_key)
591. {
592. **break**;
593. }
595. **if** (pCurrent == nullptr)
596. {
597. **return** **false**;
598. }
599. }
601. #if 0
602. cout << "pParent : " << pParent->key\_ << endl;
603. cout << "pCurrent : " << pCurrent->key\_ << endl;
604. #endif
606. // if delete node is the leaf
607. **if** (pCurrent->left\_ == nullptr && pCurrent->right\_ == nullptr)
608. {
609. **if** (pParent->left\_ == pCurrent)
610. {
611. pParent->left\_ = nullptr;
612. }
613. **else** pParent->right\_ = nullptr;
615. **delete** pCurrent;
616. pCurrent = nullptr;
617. **return** **true**;
618. }
620. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
621. S  <-- pParent
622. /  \
623. DELETE-> E      X
624. /   \
625. A    null
626. /  \
627. \*    \*
628. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
629. **if** (pCurrent->left\_ != nullptr && pCurrent->right\_ == nullptr && pCurrent == pParent->left\_)
630. {
631. Node\* deleNode = pCurrent;
633. pParent->left\_ = deleNode->right\_;
634. **delete** deleNode;
635. deleNode = nullptr;
636. **return** **true**;
637. }

640. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
641. S  <-- pParent
642. /  \
643. DELETE-> E      X
644. /   \
645. null    R
646. /  \
647. \*    \*
648. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
649. **else** **if** (pCurrent->left\_ == nullptr && pCurrent->right\_ != nullptr && pCurrent == pParent->left\_)
650. {
651. Node\* deleNode = pCurrent;
653. pParent->left\_ = pCurrent->right\_;
654. **delete** deleNode;
655. deleNode = nullptr;
656. **return** **true**;
657. }

660. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
661. S  <-- pParent
662. /  \
663. E      X <-DELETE
664. /  \
665. Q   null
666. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
667. **else** **if** (pCurrent->left\_ != nullptr && pCurrent->right\_ == nullptr && pCurrent == pParent->right\_)
668. {
669. Node\* deleNode = pCurrent;
671. pParent->right\_ = deleNode->left\_;
672. **delete** deleNode;
673. deleNode = nullptr;
674. **return** **true**;
675. }

678. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
679. S  <-- pParent
680. /  \
681. E      X <-DELETE
682. /  \
683. null  Z
684. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
685. **else** **if** (pCurrent->left\_ == nullptr && pCurrent->right\_ != nullptr && pCurrent == pParent->right\_)
686. {
687. Node\* deleNode = pCurrent;
689. pParent->right\_ = deleNode->right\_;
690. **delete** deleNode;
691. deleNode = nullptr;
692. **return** **true**;
693. }

696. // 最复杂的情况 - The WORST situation
697. **else** **if** (pCurrent->left\_ != nullptr && pCurrent->right\_ != nullptr)
698. {
699. Node\* deleNode = pCurrent;
701. pCurrent = pCurrent->right\_;
703. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
704. S  <-- pParent
705. /  \
706. DELETE-> E      X
707. /   \
708. A     R  <- pCurrent
709. /  \
710. \*    \*
711. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
712. **if** (pCurrent->left\_ == nullptr && pCurrent->right\_ == nullptr)
713. {
714. deleNode->key\_ = pCurrent->key\_;
715. deleNode->right\_ = nullptr;
717. **delete** pCurrent;
718. pCurrent = nullptr;
719. **return** **true**;
720. }

723. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
724. S  <-- pParent
725. /  \
726. DELETE-> E      X
727. /     \
728. A        R <- pCurrent
729. /  \     /   \
730. \*    \*  H      \*
731. /   \
732. pParent --->  \*      \*  ( pParent after "while")
733. /
734. G <---- pCurrent  (之后的pCurrent - pCurrent after "while")
735. /   \
736. null   \*
737. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
738. **if** (pCurrent->left\_ != nullptr)
739. {
740. **while** (pCurrent->left\_ != nullptr)
741. {
742. pParent = pCurrent;
743. pCurrent = pCurrent->left\_;
744. }
746. deleNode->key\_ = pCurrent->key\_;
747. pParent->left\_ = pCurrent->right\_;
748. **delete** pCurrent;
749. pCurrent = nullptr;
750. }
751. **return** **true**;
752. }
753. }
755. /\*
756. \*\* Delete all the nodes in tree
757. \*/
758. **void** \_freeSpaceBinaryTree\_(Node\* root)
759. {
760. **if** (nullptr == root) **return**;
761. \_freeSpaceBinaryTree\_(root->left\_);
762. \_freeSpaceBinaryTree\_(root->right\_);
763. **delete** root;
764. }
766. };

* WordInfo.hpp

1. #pragma once
2. #include "LinkList.hpp"
3. #include "MyErrorInfo.hpp"
4. #include "BinarySearchTree.hpp"
6. **class** BinarySearchTree;
7. **class** WordInfo;
8. **struct** Node;
10. **class** WordInfo
11. {
12. **public**:
13. WordInfo() : count\_same\_word\_(0), length\_word\_(0) {}
14. **public**:
15. LinkList<pair<**int**, **int**>> header\_; // save the row and col of the world
16. unsigned **int** count\_same\_word\_;
17. unsigned **int** length\_word\_;
18. };

* LinkStack.hpp

1. #pragma once
2. #include "MyErrorInfo.hpp"
3. #include <string>
4. **using** **namespace** std;
6. **template**<**typename** T>
7. **struct** LinkStack
8. {
9. // ------------------------------ StackNode ------------------------------
10. **private**:
11. **struct** StackNode
12. {
13. T data\_;
14. StackNode\* next\_;
15. };
17. StackNode\* header\_;
18. unsigned **int** size\_;
20. **public**:
21. LinkStack()
22. {
23. **this**->header\_ = **new** StackNode;
24. **this**->header\_->data\_ = NULL;
25. **this**->header\_->next\_ = nullptr;
26. **this**->size\_ = 0;
27. **if** (nullptr == **this**->header\_) **throw** MyErrorInfo("bad\_exception");
29. }
31. LinkStack(**const** LinkStack& stack)
32. {
33. **if** (nullptr == stack) **throw** MyErrorInfo("null\_data");
34. stack->header\_ = **this**->header\_;
35. stack->size\_ = **this**->size\_;
36. }
38. **void** push(**const** T& data)
39. {
40. \_push\_(data);
41. }
43. T top() **const**
44. {
45. **return** \_top\_();
46. }
48. **int** size() **const**
49. {
50. **return** \_size\_();
51. }
53. **bool** empty() **const**
54. {
55. **return** \_empty\_();
56. }
58. **void** pop()
59. {
60. \_pop\_();
61. }
63. **void** clear() **const**
64. {
65. \_clear\_();
66. }
68. **void** desgroy() **const**
69. {
70. \_desgroy\_();
71. }
73. **private**:
75. **void** \_push\_(**const** T& data)
76. {
77. **if** (nullptr == **this**->header\_) **throw** MyErrorInfo("data\_nullptr");
78. StackNode\* newNode = **new** StackNode;
79. newNode->next\_ = **this**->header\_->next\_;
80. newNode->data\_ = data;
81. **this**->header\_->next\_ = newNode;
82. **this**->size\_++;
83. }
85. T \_top\_() **const**
86. {
87. **if** (nullptr == **this**->header\_ || 0 == **this**->size\_) **throw** MyErrorInfo("null\_stack");
88. **return** **this**->header\_->next\_->data\_;
89. }
91. **int** \_size\_() **const**
92. {
93. **if** (nullptr == **this**->header\_ || 0 == **this**->size\_) **return** 0;
94. **return** **this**->size\_;
95. }
97. **bool** \_empty\_() **const**
98. {
99. **return** **this**->size\_;
100. }
102. **void** \_pop\_()
103. {
104. **if** (nullptr == **this**->header\_ || 0 == **this**->size\_) **return**;
105. StackNode\* deleNode = **this**->header\_->next\_;
106. **this**->header\_->next\_ = **this**->header\_->next\_->next\_;
107. **delete** deleNode;
108. deleNode = nullptr;
109. **this**->size\_--;
110. }
112. **void** \_clear\_()
113. {
114. **if** (nullptr == **this**->header\_ || 0 == **this**->size\_) **return**;
115. StackNode\* pCurrent = **this**->header\_->next\_->next\_;
116. StackNode\* pRear = **this**->header\_->next\_;
118. **while** (pCurrent != nullptr)
119. {
120. **delete** pRear;
121. pRear = nullptr;
122. pRear = pCurrent;
123. pCurrent = pCurrent->next\_;
124. }
125. **delete** pRear;
126. pRear = nullptr;
127. }
129. **void** \_desgroy\_()
130. {
131. **if** (nullptr == **this**->header\_) **return**;
132. StackNode\* pCurrent = **this**->header\_->next\_;
133. StackNode\* pRear = **this**->header\_;
135. **while** (pCurrent != nullptr)
136. {
137. **delete** pRear;
138. pRear = nullptr;
139. pRear = pCurrent;
140. pCurrent = pCurrent->next\_;
141. }
142. **delete** pRear;
143. pRear = nullptr;
144. }
146. };

* LinkList.hpp

1. #pragma once
2. #include "MyErrorInfo.hpp"
3. #include <string>
4. #include <iostream>
6. **template**<**typename** T> **class** LinkNode;
7. **template**<**typename** T> **class** LinkList;
8. **template**<**typename** T> ostream& operator<<(ostream& cout,  LinkNode<T>& pCurrent);
9. **template**<**typename** T> **bool** operator>(**const** LinkNode<T>& node1, **const** LinkNode<T>& node2);
11. **template**<**typename** T>
12. ostream& operator<<(ostream& cout, LinkNode<T>& pCurrent)
13. {
14. cout << pCurrent.data\_;
15. **return** cout;
16. }
18. **template**<**typename** T>
19. **bool** operator>(**const** LinkNode<T>& node1, **const** LinkNode<T>& node2)
20. {
21. **return** node1.data\_ > node2.data\_;
22. }
24. **template**<**typename** T>
25. **class** LinkNode
26. {
27. **friend** ostream& operator<<<T>(ostream& cout, LinkNode<T>& pCurrent); // !! Pay att-en <T>
28. **friend** **bool** operator><T>(**const** LinkNode<T>& node1, **const** LinkNode<T>& node2);
29. **friend** **class** LinkList<T>;
30. **friend** **class** BinarySearchTree;
31. **private**:
32. T data\_;
33. LinkNode\* next\_;
34. };
36. **template**<**typename** T>
37. **class** LinkList
38. {
39. **friend** **class** BinarySearchTree;
41. **private**:
42. LinkNode<T>\* header\_;
44. **public**:
45. LinkList()
46. {
47. header\_ = **new** LinkNode<T>;
48. header\_->next\_ = nullptr;
49. }
51. **void** insert(T data)
52. {
53. \_insert\_(data);
54. }
56. **void** foreach() **const**
57. {
58. \_foreach\_();
59. }
61. **void** sort()
62. {
63. \_sort\_();
64. }
66. ~LinkList()
67. {
68. \_desgroy\_();
69. }
71. **private**:
72. **void** \_insert\_(T data)
73. {
74. **if** (nullptr == header\_) **throw** MyErrorInfo("null\_header\_in\_function\_insert()");
76. // if is the first node
77. **if** (nullptr == header\_->next\_)
78. {
79. LinkNode<T>\* newNode = **new** LinkNode<T>;
80. newNode->data\_ = data;
81. newNode->next\_ = nullptr;
82. header\_->next\_ = newNode;
83. **return**;
84. }
86. LinkNode<T>\* pCurrent = header\_->next\_;
87. **while** (nullptr != pCurrent->next\_)
88. {
89. pCurrent = pCurrent->next\_;
90. }
92. LinkNode<T>\* newNode = **new** LinkNode<T>;
93. newNode->data\_ = data;
94. newNode->next\_ = nullptr;
95. pCurrent->next\_ = newNode;
96. }
98. **void** \_foreach\_() **const**
99. {
100. **if** (nullptr == header\_) **throw** MyErrorInfo("null\_header\_in\_function\_foreach()");
102. LinkNode<T>\* pCurrent = header\_->next\_;
103. **while** (nullptr != pCurrent)
104. {
105. cout << \*pCurrent;
106. pCurrent = pCurrent->next\_;
107. }
108. cout << endl;
109. }
111. **void** \_sort\_()
112. {
113. **if** (nullptr == header\_ || nullptr == header\_->next\_) **throw** MyErrorInfo("null\_header\_in\_function\_sort()");
115. // get the amount of node
116. unsigned **int** count\_node = 0;
117. LinkNode<T>\* pCurrent = header\_->next\_;
118. **while** (nullptr != pCurrent)
119. {
120. count\_node++;
121. pCurrent = pCurrent->next\_;
122. }
124. //-------------------------------------------------
125. pCurrent = header\_->next\_;
126. LinkNode<T>\* pNext = pCurrent->next\_;
128. T temp;
129. **for** (**int** i = 0; i < count\_node; ++i)
130. {
131. **for** (**int** j = 0; j < count\_node - i; ++j)
132. {
133. **if** (\*pCurrent > \* pNext)
134. {
135. temp = pCurrent->data\_;
136. pCurrent->data\_ = pNext->data\_;
137. pNext->data\_ = temp;
138. }
140. pCurrent = pNext;
141. pNext = pNext->next\_;
143. **if** (nullptr == pNext)
144. {
145. pCurrent = header\_->next\_;
146. pNext = pCurrent->next\_;
147. }
149. }
150. }
151. }
153. **void** \_desgroy\_()
154. {
155. **if** (nullptr == header\_ || nullptr == header\_->next\_)
156. {
157. **delete** header\_;
158. header\_ = nullptr;
159. **return**;
160. }
161. LinkNode<T>\* pCurrent = header\_;
162. LinkNode<T>\* pNext = header\_->next\_;
164. **while** (nullptr != pNext && nullptr != pCurrent)
165. {
166. **delete**  pCurrent;
167. pCurrent = nullptr;
168. pCurrent = pNext;
169. pNext = pNext->next\_;
170. }
171. header\_ = nullptr;
172. }
173. };

* MyErrorInfo.hpp

1. #pragma once
2. #include<string>
3. #include<stdexcept>
4. **using** **namespace** std;
6. // ------------------------------ ERROR INFO ------------------------------
7. **class** MyErrorInfo : **public** exception
8. {
9. **public**:
10. MyErrorInfo(string errorInfo)
11. {
12. **this**->m\_error = errorInfo;
13. }
14. ~MyErrorInfo() {}
16. **virtual** **const** **char**\* what()
17. {
18. **return** **this**->m\_error.c\_str();
19. }
21. **private**:
22. string m\_error;
23. };

* main.cpp

1. #include "LinkList.hpp"
2. #include "BinarySearchTree.hpp"
3. **using** **namespace** std;
5. **void** testFunc()
6. {
7. string file\_name;
8. cout << "＃ Please enter the required file name : "; cin >> file\_name; // 1.txt
10. BinarySearchTree tree;
11. tree.openFile(file\_name);
13. /\*
14. \*\*  first add rol number, and output in file (output.txt)
15. \*/
16. tree.printTextWithRowNum();
18. /\*
19. \*\* Insert all word into the binary search tree
20. \*/
21. tree.makeTree();
23. /\*
24. \*\* print the words on screen
25. \*\* true - print same word;
26. \*\* false - do not print same word
27. \*/
28. tree.printDictionaryOrder(**false**);
30. /\*
31. \*\* Seach word
32. \*\* if have this word print this word with its info (row, col, count, length...)
33. \*/
34. tree.searchWord("fire");
36. /\*
37. \*\*  Delete word
38. \*/
39. tree.deleteWord("student");
41. /\*
42. \*\*  out put the word by Dictionary order in to the file "output.txt"
43. \*/
44. tree.outputDictionaryOrderInFile(**false**);
45. }
47. **int** main()
48. {
49. **try**
50. {
51. testFunc();
52. }
53. **catch** ( MyErrorInfo& err)
54. {
55. cerr << err.what() << endl;
56. }
57. **catch** (...)
58. {
59. cerr << "Unknow error!!!" << endl;
60. }
61. }

# Приложение 2. Протоколы отладки

1. Сначала программа запрашивает имя файла, имя файла должно быть правильно введено (xxx.txt)
2. tree.openFile(file\_name);
3. Вывод статьи с номером строки в файл
4. /\*
5. \*\*  first add rol number, and output in file (output.txt)
6. \*/
7. tree.printTextWithRowNum();
8. Вставить все слова в двоичное дерево поиска
9. /\*
10. \*\* Insert all word into the binary search tree
11. \*/
12. tree.makeTree();
13. Выводит слова (с информационными полями) на экран в лексикографическом порядке
14. /\*
15. \*\* print the words on screen
16. \*\* true - print same word;
17. \*\* false - do not print same word
18. \*/
19. tree.printDictionaryOrder(**false**);
20. Тест поиск слов
21. /\*
22. \*\* Seach word
23. \*\* if have this word print this word with its info (row, col, count, length...)
24. \*/
25. tree.searchWord("fire");
26. Тест удаление узла по слову
27. /\*
28. \*\*  Delete word
29. \*/
30. tree.deleteWord("student");
31. Выводит слова (с информационными полями) в файл в лексикографическом порядке
32. /\*
33. \*\*  out put the word by Dictionary order in to the file "output.txt"
34. \*/
35. tree.outputDictionaryOrderInFile(**false**);

* Для некорректного ввода данных:

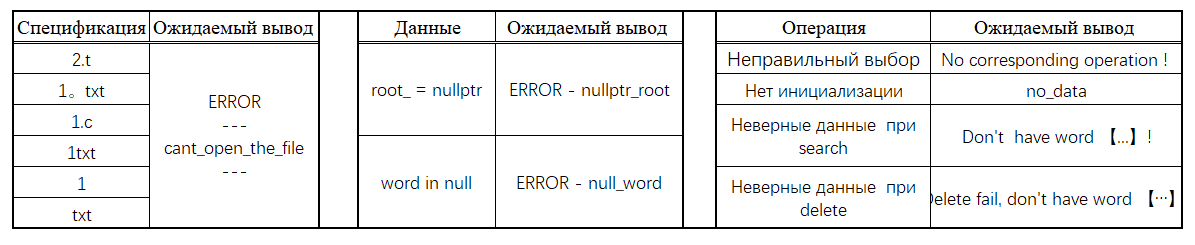


Рис. 18