

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Высшая школа программной инженерии



ПОЛИТЕХ
Санкт-Петербургский
политехнический университет
Петра Великого

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
Требования	3
Основная часть работы	4
1. Описание алгоритма решения и используемых структур данных	4
1.1 Классы	4
1.2 Интерфейсы	7
1.3 Методы.....	8
2. Анализ алгоритма.....	14
2.1 Методы двоичного дерева поиска.....	14
2.2 Двоичное дерево поиска	25
3. Описание спецификации программы (детальные требования).....	28
4. Описание программы.....	30
4.1 Структура программы	30
4.2 Форматы входных.....	31
4.3 Выходных данных.....	32
Заключение.....	33
Список использованных источников.....	34
Приложение 1. Текст программы.....	35
Приложение 2. Протоколы отладки.....	65

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

1. Описание алгоритма решения и используемых структур данных

1.1 Классы

- BinarySearchTree

В классе BinarySearchTree сохранил :

root_(корень дерева)

file_name_(имя файла).

```
1. class BinarySearchTree
2. {
3. private:
4.     Node* root_;
5.     string file_name_;
```

- Node

Каждый узел из класса Node, который сохранил:

string key_(слово)

WordInfo* info_(информационная часть)

Node* left_

Node* right_

Node* p_

```
1. struct Node
2. {
3.     string key_; // word
4.     WordInfo* info_; // information of word
5.     Node* left_;
6.     Node* right_;
7.     Node* p_;
8. };
```

● WordInfo

Сохранил информационную часть слова:

LinkedList<pair<int, int>> header_

unsigned int count_same_word_ (Количество одинаковых слов)

unsigned int length_word_ (Длина слова)

```
1. class WordInfo
2. {
3. public:
4.     LinkedList<pair<int, int>> header_; // save the row and col of the word
5.     unsigned int count_same_word_;
6.     unsigned int length_word_;
7. };
```

● LinkStack

```
1. template<typename T>
2. struct LinkStack
3. {
4. private:
5.     StackNode<T>* header_;
6.     unsigned int size_;
```

● LinkedList

```
1. template<typename T>
2. class LinkedList
3. {
4. private:
5.     LinkNode<T>* header_;
```

● LinkNode

```
1. class LinkNode
2. {
3. private:
4.     T data_;
5.     LinkNode* next_;
```

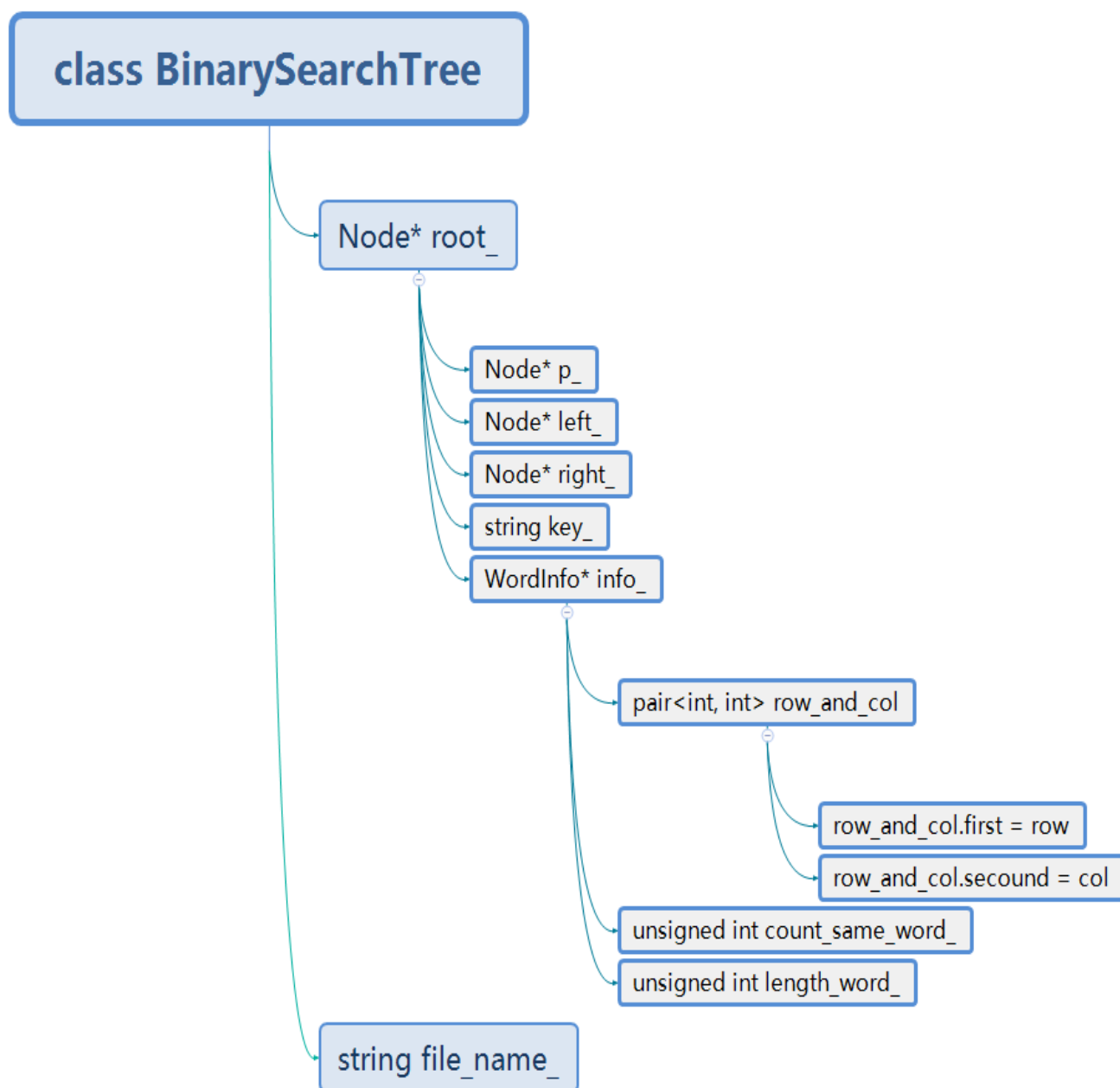


Рис.2

1.2 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 файл

```
void outputInFile();
```


➤ **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.

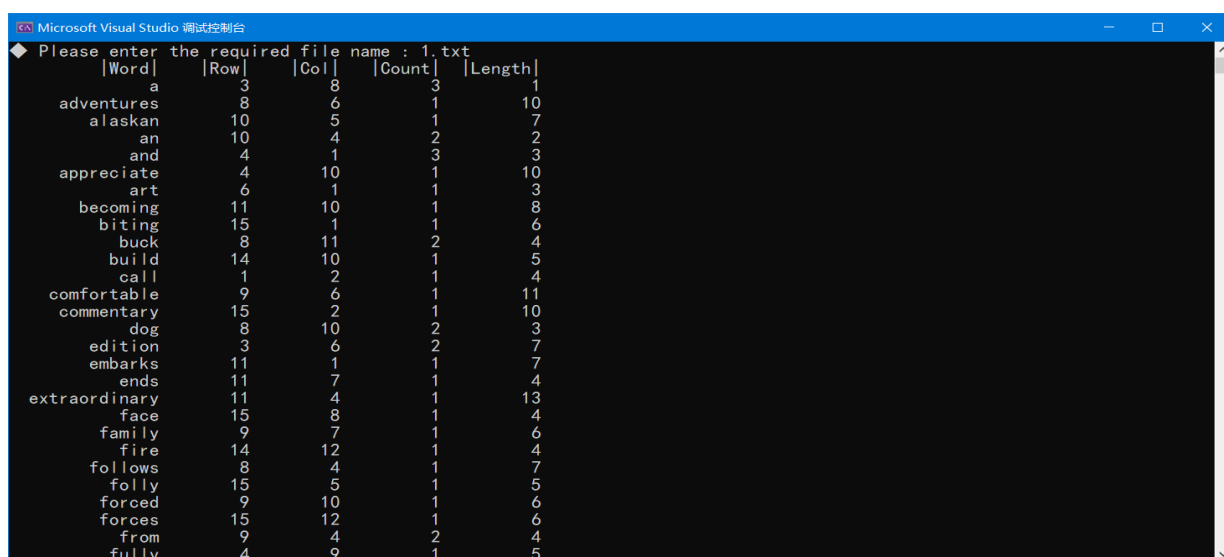
```
1 - The Call of the Wild 【Words in this row : 5】
2 -
3 - This Prestwick House Literary Touchstone Edition includes a glossary 【Words in this row : 9】
4 - and reader's notes to help the modern reader fully appreciate 【Words in this row : 10】
5 - London's masterful weaving of science, philosophy, and the storyteller's 【Words in this row : 9】
6 - art. 【Words in this row : 1】
7 -
8 - This gripping story follows the adventures of the loyal dog Buck, 【Words in this row : 11】
9 - who is stolen from his comfortable family home and forced into the 【Words in this row : 12】
10 - harsh life of an Alaskan sled dog. Passed from master to master, Buck 【Words in this row : 13】
11 - embarks on an extraordinary journey that ends with his becoming the 【Words in this row : 11】
12 - legendary leader of a wolf pack. 【Words in this row : 6】
13 -
14 - Included in this Edition is the short story, To Build a Fire, London's 【Words in this row : 13】
15 - biting commentary on human folly in the face of indomitable natural forces. 【Words in this row : 12】

=====
Amount of words in the article : 112
Amount of row in the article : 15
=====
```

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.



Word	Row	Col	Count	Length
a	3	8	3	1
adventures	8	6	1	10
alaskan	10	5	1	7
an	10	4	2	2
and	4	1	3	3
appreciate	4	10	1	10
art	6	1	1	3
becoming	11	10	1	8
biting	15	1	1	6
buck	8	11	2	4
build	14	10	1	5
call	1	2	1	4
comfortable	9	6	1	11
commentary	15	2	1	10
dog	8	10	2	3
edition	3	6	2	7
embarks	11	1	1	7
ends	11	7	1	4
extraordinary	11	4	1	13
face	15	8	1	4
family	9	7	1	6
fire	14	12	1	4
follows	8	4	1	7
folly	15	5	1	5
forced	9	10	1	6
forces	15	12	1	6
from	9	4	2	4
fully	4	9	1	5

Рис.3 Сформировать таблицу и печать на экране

- **void _inOrderOutputInFile_ (Node* root, ofstream& ofs, bool repetitive_word);**

Сформировать таблицу и выводить в файле, 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

Word	Row	Col	Count	Length
a	3	8	3	1
adventures	8	6	1	10
alaskan	10	5	1	7
an	10	4	2	2
and	4	1	3	3
appreciate	4	10	1	10

....

Пример

....

touchstone	3	5	1	10
weaving	5	3	1	7
who	9	1	1	3
wild	1	5	1	4
with	11	8	1	4
wolf	12	5	1	4

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 “ ,”, “ .”.

This	→	this
reader's	→	reader
London's	→	london
science,	→	science
art.	→	art

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

```
=====Search Word=====
Word : fire
Row : 14
Col : 12
```

Рис.6 искать fire

- Искать student

```
=====Search Word=====
Don't have word 【student】 !
```

Рис.7 искать student

- Искать science

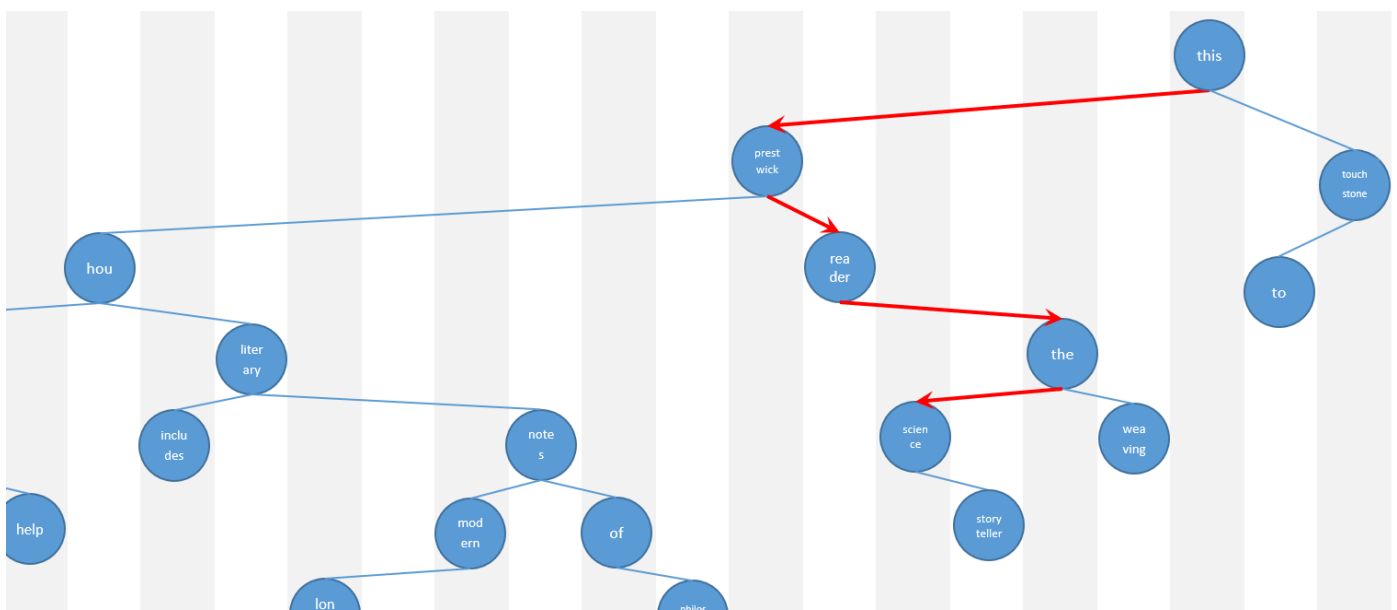


Рис. 8

➤ **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

```
=====Delete Word=====
Succeed delete word 【fire】!
```

Рис.9 удалить file

- удалить student

```
4 =====Delete Word=====
4
4 Delete fail, don't have word 【student】!
4
```

Рис. 10 удалить students

Интерфейсы

void openFile(string file_name)

this->file_name_ = file_name

void printTextWithRowNum

addRowNum()

private

void makeTree()

makeWordTree()

private

void printDictionaryOrder(bool repetitive_word)

inOrderOnScreen(this->root_, repetitive_word);

private

void outputDictionaryOrderInFile(bool repetitive_word)

void outputDictionaryOrderInFile(bool repetitive_word)

private

void searchWord(string word_search)

searchWordInTree(word_search)

private

void deleteWord(string dele_word)

deleteNodeBinarySearchTree(dele_word)

private

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*.

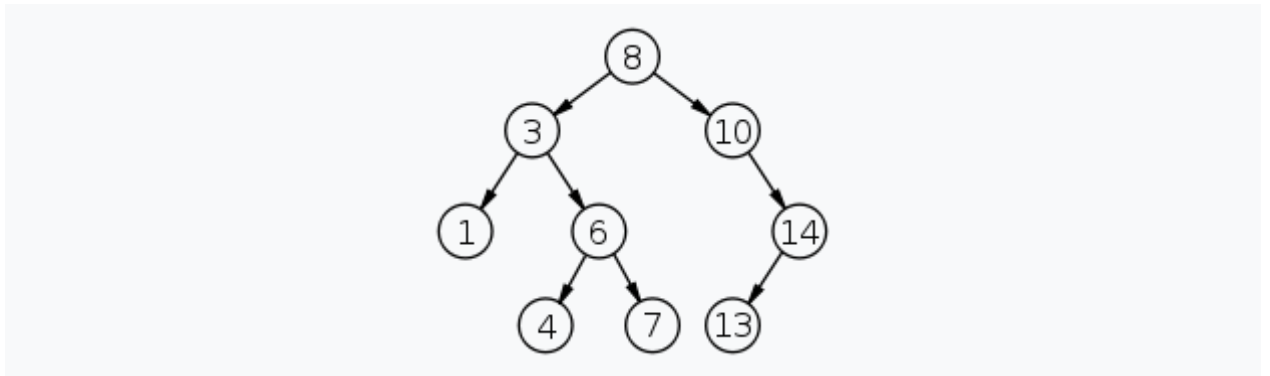


Рис. 11 Пример двоичного дерева поиска

Двоичное дерево поиска (англ. *binary search tree*, BST) — это двоичное дерево, 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: $\text{key}[\text{left}[X]] < \text{key}[X] \leq \text{key}[\text{right}[X]]$, то есть ключи данных родительского узла больше ключей данных левого сына и нестрогое меньше ключей данных правого.

2.1.1 Добавление элемента (INSERT)

Дано: дерево Tree и Key.

Задача: вставить Key в дерево Tree (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 $\text{Key} > X$, рекурсивно добавить (Key) в правое поддереву Tree.
 - Если $\text{Key} < X$, рекурсивно добавить (Key) в левое поддереву Tree.
 - Если $\text{Key} = X$, запишите информацию о повторении второго слова в info_.

```

1. /*
2. ** Insert one word into the binary search tree
3. */
4. void _insertNode_(Node* nodeInsert, unsigned int count_word_text)
5. {
6.     if (nullptr == root_ || nullptr == nodeInsert) throw MyErrorInfo("nullptr_root_or_nodeInsert_in_insertNode()");
7.
8.     if (1 == count_word_text)
9.     {
10.         root_>p_ = nullptr;
11.         root_>left_ = nullptr;
12.         root_>right_ = nullptr;
13.         root_>key_ = nodeInsert->key_;
14.         root_>info_ = nodeInsert->info_;
15.         return;
16.     }

```

```

17.
18.
19.     Node* pCurrent = root_;
20.
21.     while (true)
22.     {
23.         while (pCurrent->key_ > nodeInsert->key_ && pCurrent->left_ != nullptr)
24.         {
25.             pCurrent = pCurrent->left_;
26.         }
27.         while (pCurrent->key_ < nodeInsert->key_ && pCurrent->right_ != nullptr)
28.         {
29.             pCurrent = pCurrent->right_;
30.         }
31.
32.         if (pCurrent->key_ > nodeInsert->key_ && pCurrent->left_ == nullptr) break;
33.         else if (pCurrent->key_ < nodeInsert->key_ && pCurrent->right_ == nullptr) break;
34.         else if (pCurrent->key_ == nodeInsert->key_) break;
35.     }
36.
37.     if (pCurrent->key_ > nodeInsert->key_)
38.     {
39.         pCurrent->left_ = nodeInsert;
40.         nodeInsert->p_ = pCurrent;
41.     }
42.     else if (pCurrent->key_ < nodeInsert->key_)
43.     {
44.         pCurrent->right_ = nodeInsert;
45.         nodeInsert->p_ = pCurrent;
46.     }
47.     else
48.     {
49.         throw MyErrorInfo("error_insert_in_insertNode()");
50.     }
51. }

```

2.1.2 Поиск элемента (SEARCH)

Дано: дерево Tree и ключ Key.

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

Сложность: $O(\log_2 n)$ 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 $\text{Key} = X$, выдать ссылку на этот узел и остановиться.
 - Если $\text{Key} > X$, рекурсивно искать ключ Key в правом поддереве Tree.
 - Если $\text{Key} < X$, рекурсивно искать ключ Key в левом поддереве Tree.

```

1. Node* _searchWordInTree_(string word_search)
2. {
3.     if (nullptr == root_) return nullptr;
4.
5.     Node* pCurrent = this->root_;
6.
7.     while (nullptr != pCurrent)
8.     {
9.         if (word_search == pCurrent->key_) return pCurrent;
10.
11.         while (nullptr != pCurrent && word_search < pCurrent->key_)
12.         {
13.             pCurrent = pCurrent->left_;
14.         }
15.
16.         while (nullptr != pCurrent && word_search > pCurrent->key_)
17.         {
18.             pCurrent = pCurrent->right_;
19.         }

```

```
20.     }  
21.     return nullptr;  
22. }
```

2.1.3 Удаление узла (DELETE)

Дано: дерево Tree с корнем n и ключом Key.

Задача: удалить из дерева Tree 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 $\text{Key} > X$, рекурсивно удалить Key из правого поддерева Tree;
 - Если $\text{Key} < X$, рекурсивно удалить Key из левого поддерева Tree;
 - Если $\text{Key} = X$, 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 ($n \rightarrow \text{right} \rightarrow \text{left}$)
 - Копируем из правого узла в удаляемый поля Key
 - Иначе
 - Возьмём самый левый узел m, правого поддерева $n \rightarrow \text{right}$;
 - Скопируем данные (кроме ссылок на дочерние элементы) из m в n;

- Рекурсивно удалим узел m.

```

1. bool _deleteNodeBinarySearchTree_(string dele_key)
2. {
3.     if (nullptr == root_) throw MyErrorInfo("nullptr_root_in_DeleteNodeBinarySearchTree
         (string key_del)");
4.
5.     // First of all, need to find the node, which need to delete
6.     Node* pCurrent = root_;
7.     Node* pParent = pCurrent;
8.     while (true)
9.     {
10.        if (pCurrent->key_ > dele_key)
11.        {
12.            pParent = pCurrent;
13.            pCurrent = pCurrent->left_;
14.        }
15.        else if (pCurrent->key_ < dele_key)
16.        {
17.            pParent = pCurrent;
18.            pCurrent = pCurrent->right_;
19.        }
20.        else if (pCurrent->key_ == dele_key)
21.        {
22.            break;
23.        }
24.
25.        if (pCurrent == nullptr)
26.        {
27.            return false;
28.        }
29.    }
30.
31.    // if delete node is the leaf
32.    if (pCurrent->left_ == nullptr && pCurrent->right_ == nullptr)
33.    {
34.        if (pParent->left_ == pCurrent)
35.        {
36.            pParent->left_ = nullptr;
37.        }
38.        else pParent->right_ = nullptr;
39.
40.        delete pCurrent;
41.        pCurrent = nullptr;
42.        return true;

```

```

43.     }
44.
45.     /*****
46.             S  <-- pParent
47.             /  \
48. DELETE-> E      X
49.             /  \
50.             A   null
51.             /  \
52.             *   *
53.     *****/
54.     if (pCurrent->left_ != nullptr && pCurrent->right_ == nullptr && pCurrent == pParent->left_)
55.     {
56.         Node* deleNode = pCurrent;
57.         pParent->left_ = deleNode->right_;
58.         delete deleNode;
59.         deleNode = nullptr;
60.         return true;
61.     }
62.
63.
64.     /*****
65.             S  <-- pParent
66.             /  \
67. DELETE-> E      X
68.             /  \
69.             null R
70.             /  \
71.             *   *
72.     *****/
73.     else if (pCurrent->left_ == nullptr && pCurrent->right_ != nullptr && pCurrent == pParent->left_)
74.     {
75.         Node* deleNode = pCurrent;
76.         pParent->left_ = pCurrent->right_;
77.         delete deleNode;
78.         deleNode = nullptr;
79.         return true;
80.     }
81.
82.
83.     /*****
84.             S  <-- pParent
85.             /  \

```

```

86.             E      X <-DELETE
87.             /  \
88.             Q   null
89.  *****/
90.  else if (pCurrent->left_ != nullptr && pCurrent->right_ == nullptr && pCurrent == p
      Parent->right_)
91.  {
92.      Node* deleNode = pCurrent;
93.      pParent->right_ = deleNode->left_;
94.      delete deleNode;
95.      deleNode = nullptr;
96.      return true;
97.  }
98.
99.
100.  /***/
101.      S <-- pParent
102.      /  \
103.      E      X <-DELETE
104.      /  \
105.      null  Z
106.  *****/
107.  else if (pCurrent->left_ == nullptr && pCurrent->right_ != nullptr && pCurrent ==
      pParent->right_)
108.  {
109.      Node* deleNode = pCurrent;
110.      pParent->right_ = deleNode->right_;
111.      delete deleNode;
112.      deleNode = nullptr;
113.      return true;
114.  }
115.
116.
117.  // 最复杂的情况 - The WORST situation
118.  else if (pCurrent->left_ != nullptr && pCurrent->right_ != nullptr)
119.  {
120.      Node* deleNode = pCurrent;
121.      pCurrent = pCurrent->right_;
122.
123.      /***/
124.      S <-- pParent
125.      /  \
126.  DELETE-> E      X
127.      /  \
128.      A      R <- pCurrent

```



```

129.          /  \
130.         *    *
131.  *****/
132.  if (pCurrent->left_ == nullptr && pCurrent->right_ == nullptr)
133.  {
134.      deleNode->key_ = pCurrent->key_;
135.      deleNode->right_ = nullptr;
136.
137.      delete pCurrent;
138.      pCurrent = nullptr;
139.      return true;
140.  }
141.
142.
143.  /*****
144.          S <-- pParent
145.         /  \
146.  DELETE-> E    X
147.         /    \
148.        A      R <- pCurrent
149.       /  \    /  \
150.      *    *  H    *
151.         /    \
152.  pParent ---> *    * ( pParent after "while")
153.         /
154.        G <---- pCurrent (之后的 pCurrent - pCurrent after "while")
155.       /  \
156.      null  *
157.  *****/
158.  if (pCurrent->left_ != nullptr)
159.  {
160.      while (pCurrent->left_ != nullptr)
161.      {
162.          pParent = pCurrent;
163.          pCurrent = pCurrent->left_;
164.      }
165.
166.      deleNode->key_ = pCurrent->key_;
167.      pParent->left_ = pCurrent->right_;
168.      delete pCurrent;
169.      pCurrent = nullptr;
170.  }
171.  return true;
172.  }
173. }

```

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 $\sim \log N$. В худшем случае в пути поиска может быть N узлов. Как показано на Рис.12. Но в целом форма дерева ближе к лучшему.

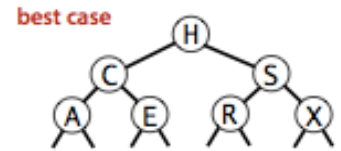


Рис. 12

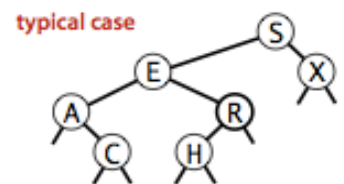


Рис.13

Для многих случаев, 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

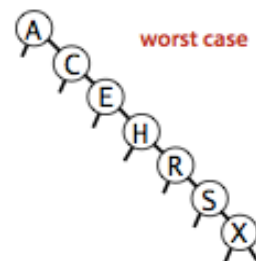


Рис.14

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.

В бинарном дереве поиска, построенном из N случайных ключей, среднее число сравнений, необходимых для поиска совпадения, составляет $\sim 2\ln N$ (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 C_N - длина внутреннего пути двоичного дерева поиска, построенного из N случайно упорядоченных разных ключей, тогда средняя стоимость поискового попадания равна $(1 + C_N / N)$. У нас есть $C_0 = C_1 = 0$, и для $N > 1$ мы можем напрямую получить индуктивные отношения в соответствии с рекурсивной структурой второго и второго дерева поиска:

$$C_N = N-1 + (C_0 + C_{N-1}) / N + (C_1 + C_{N-2}) / N + \dots + (C_{N-1} + C_0) / N$$

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 $C_N \sim 2N\ln N$.

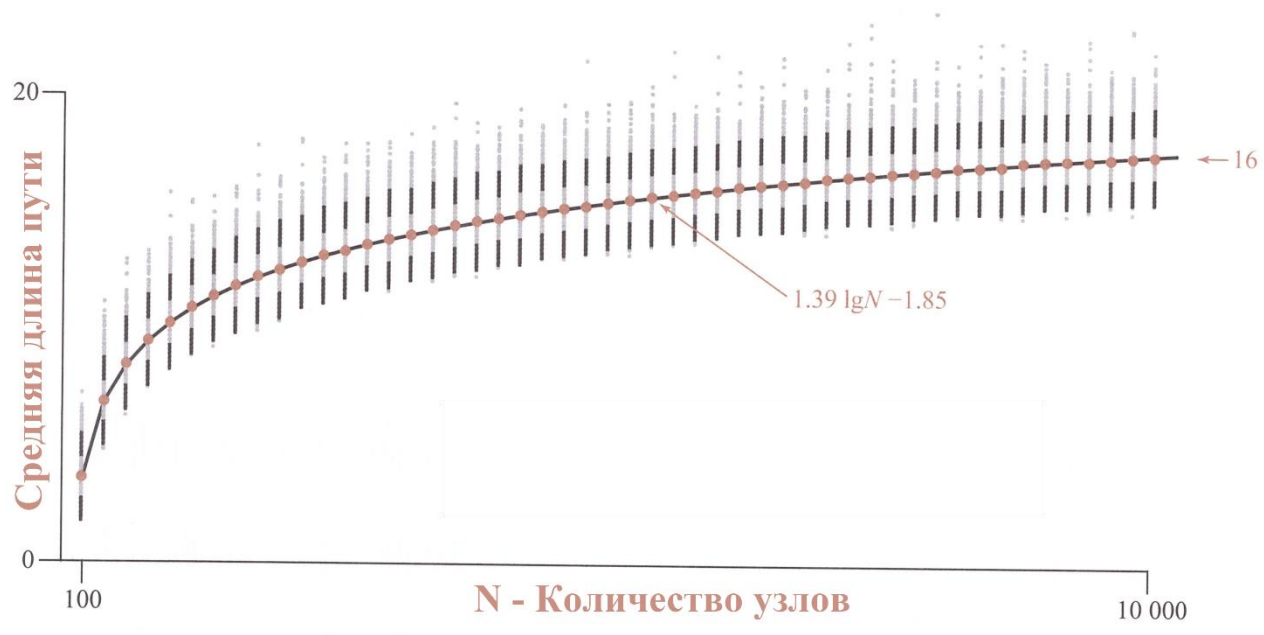


Таблица 1 Средняя длина пути от корня до любого узла в случайно построенном бинарном дереве поиска

Алгоритм	Средняя ситуация		Худшая ситуация	
	найти	вставлять	найти успех	вставлять
Последовательный запрос (неупорядоченный связанный список)	N	N	$N/2$	N
Бинарный поиск (упорядоченный массив)	$\lg N$	N	$\lg N$	$N/2$
Бинарное дерево поиска	N	N	$1.39 \lg N$	$1.39 \lg N$

Таблица 2 Краткое описание

3. Описание спецификации программы (детальные требования)

1). Файл должен быть открыт

- i. 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

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

4). Если пустая строка пропускается, счетчик не накапливается

5). Для повторяющихся слов

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

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

Пример:

Life	→	life		student's	→	student
Technology	→	technology		Student's	→	student

Рис.13

4. Описание программы

4.1 Структура программы

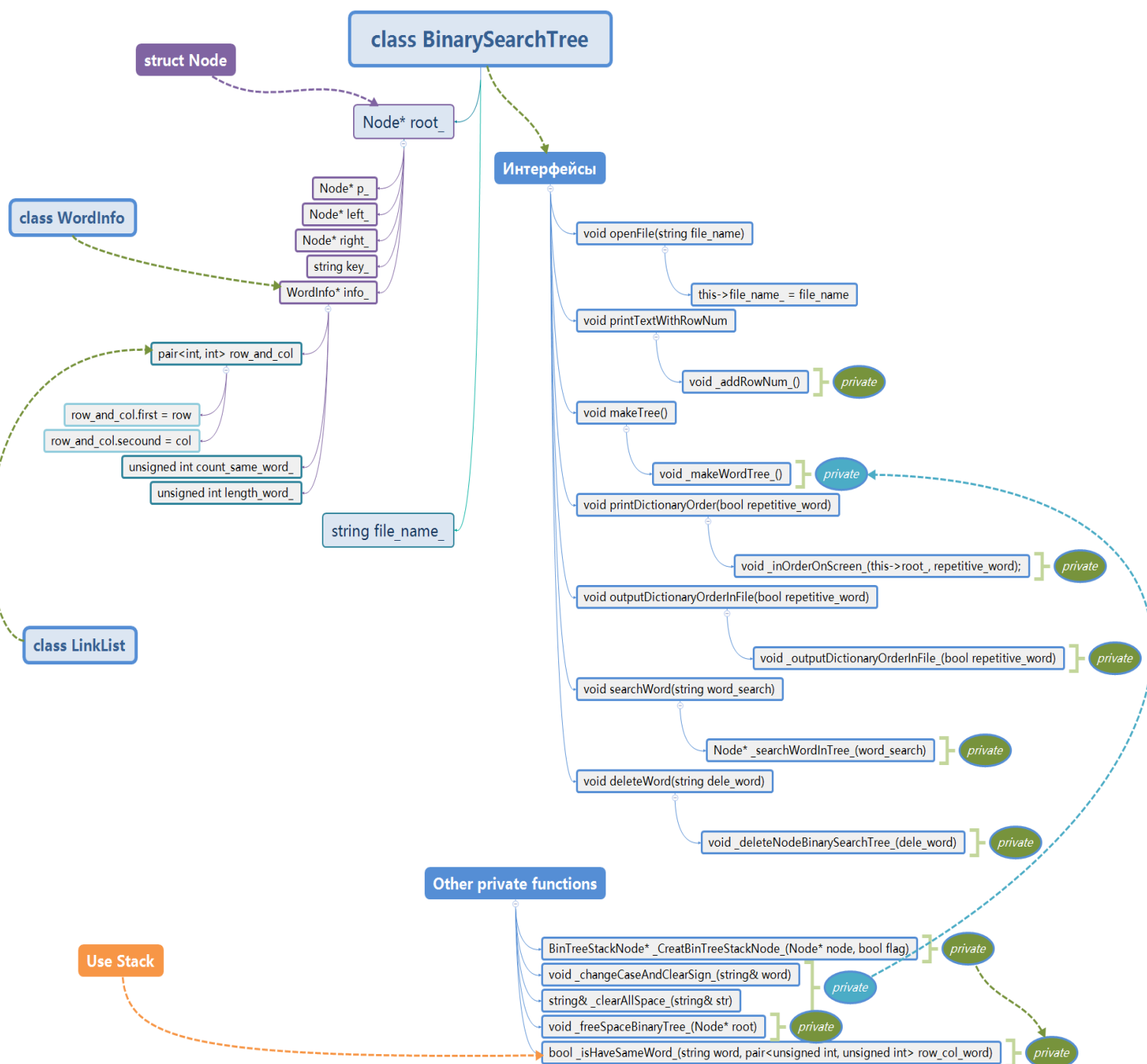


Рис. 14

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)

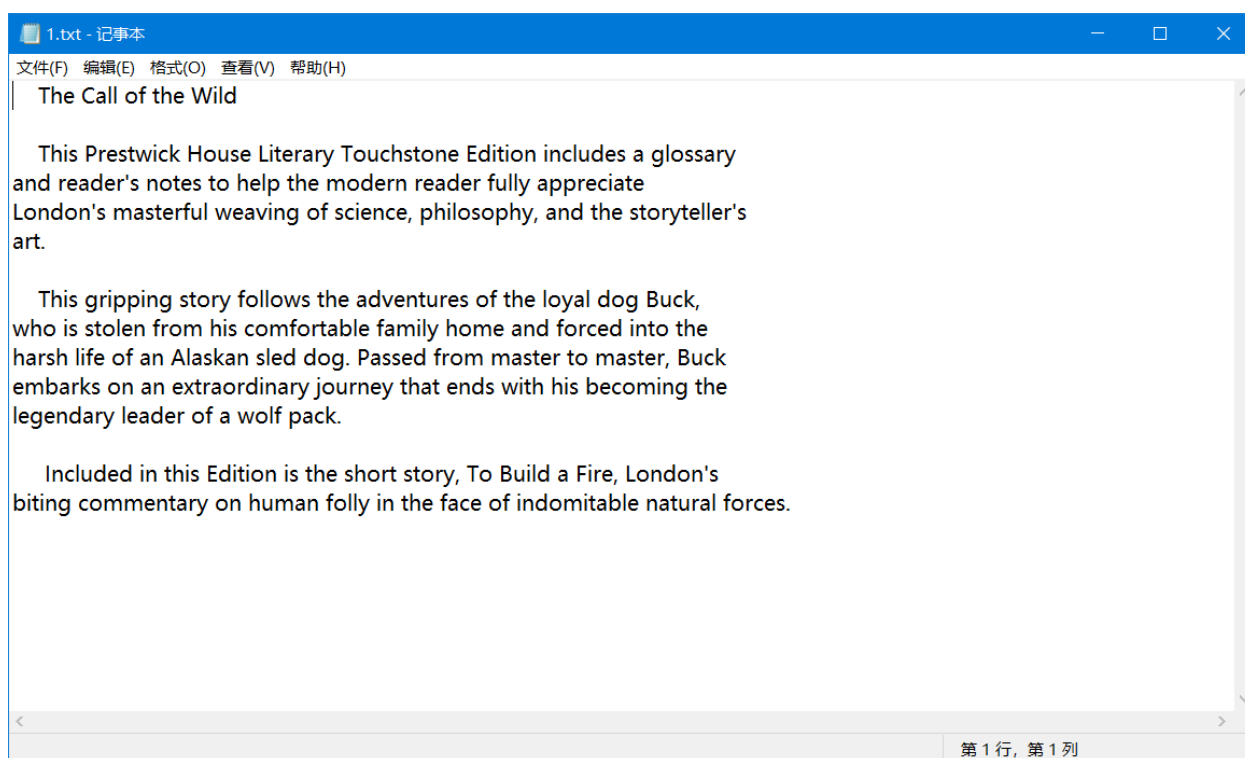


Рис. 15

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.

```

1 - The Call of the Wild 【Words in this row : 5】
2 -
3 - This Prestwick House Literary Touchstone Edition includes a glossary 【Words in this row : 9】
4 - and reader's notes to help the modern reader fully appreciate 【Words in this row : 10】
5 - London's masterful weaving of science, philosophy, and the storyteller's 【Words in this row : 9】
6 - art. 【Words in this row : 1】
7 -
8 - This gripping story follows the adventures of the loyal dog Buck, 【Words in this row : 11】
9 - who is stolen from his comfortable family home and forced into the 【Words in this row : 12】
10 - harsh life of an Alaskan sled dog. Passed from master to master, Buck 【Words in this row : 13】
11 - embarks on an extraordinary journey that ends with his becoming the 【Words in this row : 11】
12 - legendary leader of a wolf pack. 【Words in this row : 6】
13 -
14 - Included in this Edition is the short story, To Build a Fire, London's 【Words in this row : 13】
15 - biting commentary on human folly in the face of indomitable natural forces. 【Words in this row : 12】

=====
Amount of words in the article : 112
Amount of row in the article : 15
=====

```

Рис. 16 первая часть

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

Word	Row	Col	Count	Length
a	3	8	3	1
adventures	8	6	1	10
alaskan	10	5	1	7
an	10	4	2	2
and	4	1	3	3
appreciate	4	10	1	10
art	6	1	1	3
becoming	11	10	1	8
biting	15	1	1	6
buck	8	11	2	4
build	14	10	1	5
call	1	2	1	4
comfortable	9	6	1	11
commentary	15	2	1	10
dog	8	10	2	3
edition	3	6	2	7
embarks	11	1	1	7
ends	11	7	1	4
extraordinary	11	4	1	13

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>
9.
10. class BinarySearchTree;
11. struct Node;
12. class WordInfo;
13.
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_(
22.             nullptr), p_(nullptr) {}
23.         Node(string key, WordInfo* info, Node* left, Node* right, Node* p) : key_(key), in
24.             fo_(info), left_(left), right_(right), p_(p) {}
25.
26.         string key_; // word
27.         WordInfo* info_;
28.         Node* left_;
29.         Node* right_;
30.         Node* p_;
31.     };
32.
33.     struct BinTreeStackNode
34.     {
35.         Node* root;
36.         bool flag;
37.     };

```

```

36.
37.     Node* root_;
38.     string file_name_;
39.
40. public:
41.
42.     BinarySearchTree()
43.     {
44.         this->root_ = new Node;
45.     }
46.
47.     ~BinarySearchTree()
48.     {
49.         _freeSpaceBinaryTree_(this->root_);
50.     }
51.
52.     void openFile(string file_name)
53.     {
54.         this->file_name_ = file_name;
55.     }
56.
57.     void printTextWithRowNum()
58.     {
59.         _addRowNum_();
60.     }
61.
62.     void makeTree()
63.     {
64.         _makeWordTree_();
65.     }
66.
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_foreachDictionaryOr
           der()");
75.
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.     }
84.
85.     _inOrderOnScreen_(this->root_, repetitive_word);
86. }
87.
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_outputDictionaryOrd
erInFile(bool repetitive_word)");
94.
95.     ofstream ofs;
96.     ofs.open("./output.txt", ios::app); // ::app
97.     if (!ofs.is_open()) throw MyErrorInfo("cant_open_the_file_output_in_outputDictiona
ryOrderInFile(bool repetitive_word)");
98.
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.     }
107.
108.     _inOrderOutputInFile_(this->root_, ofs, repetitive_word);
109.     cout << endl << "Successful output in 【output.txt】!" << endl;
110. }
111.
112. /*
113.  ** Search 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[0
    m" << 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 << e
        ndl
130.         << "Col : " << searchWord->info_->header_.header_->next_->data_.second <<
        endl
131.         << "\033[0m" << endl;
132.     }
133. }
134.
135. /*
136. ** Delete word by cin
137. */
138. void deleteWord(string dele_word)
139. {
140.     bool is_dele = _deleteNodeBinarySearchTree_(dele_word);
141.
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. }
160.
161. // =====
162.

```

```

163. private:
164.
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()");
171.
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.         }
181.
182.         Node* pCurrent = root_;
183.
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.             }
194.
195.             if (pCurrent->key_ > nodeInsert->key_ && pCurrent->left_ == nullptr) break;
196.             else if (pCurrent->key_ < nodeInsert->key_ && pCurrent->right_ == nullptr) bre
ak;
197.             else if (pCurrent->key_ == nodeInsert->key_) break;
198.         }
199.
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. }
215.
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;
223.
224.     ifs_row.open(file_name_, ios::in);
225.     if (!ifs_row.is_open()) throw MyErrorInfo("cant_open_the_file_ifs_row_in_makeWordT
ree()");
226.
227.     ifs_word.open(file_name_, ios::in);
228.     if (!ifs_word.is_open()) throw MyErrorInfo("cant_open_the_file_ifs_word_in_makeWor
dTree()");
229.
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;
237.
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);
246.
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_wor
rd);
255.
256.         /*
257.         ** Insert to binary search tree
258.         */
259.         if (!_isHaveSameword_( temp_word, row_col_word)) // First determine if the
re 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();
265.
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. }
280.
281. void _addRowNum_()
282. {
283.     /*
284.     ** read data(text)
285.     */
286.     string temp_word, temp_line;
287.     ifstream ifs_word, ifs_row;
288.
289.     ifs_row.open(file_name_, ios::in);
290.     if (!ifs_row.is_open()) throw MyErrorInfo("cant_open_the_file_ifs_row_in_addRowNum
()");

```

```

291.
292.     ifs_word.open(file_name_, ios::in);
293.     if (!ifs_word.is_open()) throw MyErrorInfo("cant_open_the_file_ifs_word_in_addRowNum()");
294.
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()");
298.
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();
307.
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.         }
316.
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.     }
330.
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. }
340.
341. /*
342. ** print on screen
343. */
344. void _inOrderOnScreen_(Node* root, bool repetitive_word)
345. {
346.     if (nullptr == root) return;
347.
348.     _inOrderOnScreen_(root->left_, repetitive_word);
349.
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; /
357.             // first - row
358.             cout.width(9); cout << root->info_->header_.header_->next_->data_.second;
359.             // secound - col
360.             cout.width(10); cout << root->info_->count_same_word_;
361.             cout.width(10); cout << root->info_->length_word_;
362.             cout << endl;
363.         }
364.         else
365.         {
366.             LinkNode<pair<int, int>>* pCurrent = root->info_->header_.header_->next_;
367.
368.             for (int i = 0; i < root->info_->count_same_word_; i++)
369.             {
370.                 cout.width(15); cout << root->key_;
371.                 cout.width(9); cout << pCurrent->data_.first;
372.                 cout.width(9); cout << pCurrent->data_.second;
373.                 cout.width(10); cout << root->info_->count_same_word_;
374.                 cout.width(10); cout << root->info_->length_word_;
375.                 cout << endl;
376.                 pCurrent = pCurrent->next_;
377.             }
378.         }
379.     }
380. }

```

```

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. }
389.
390. /*
391. ** output in file [output.txt]
392. */
393. void _inOrderOutputInFile_(Node* root, ofstream& ofs, bool repetitive_word)
394. {
395.     if (nullptr == root) return;
396.
397.     _inOrderOutputInFile_(root->left_, ofs, repetitive_word);
398.
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 // second - 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.
415.             for (int i = 0; i < root->info_->count_same_word_; i++)
416.             {
417.                 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. }
438.
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.     }
451.
452.     int index = 0;
453.     while (word.find(',', index) != string::npos)
454.     {
455.         index = word.find(',', index);
456.         word.erase(index, 1);
457.     }
458.
459.     while (word.find('.', index) != string::npos)
460.     {

```

```

461.         index = word.find('.', index);
462.         word.erase(index, 1);
463.     }
464.
465.     while (word.find("'s", index) != string::npos)
466.     {
467.         index = word.find("'s", index);
468.         word.erase(index, 2);
469.     }
470. }
471.
472. /*
473.  ** Creat element in stack
474.  */
475. BinTreeStackNode* _CreatBinTreeStackNode_(Node* node, bool flag)
476. {
477.     BinTreeStackNode* newNode = new BinTreeStackNode;
478.
479.     newNode->root = node;
480.     newNode->flag = false;
481.     return newNode;
482. }
483.
484. /*
485.  ** 【Use Stack】
486.  Does the same word exist in the tree, if have change "header_.insert(pair)" in WordInfo
487.  */
488. bool _isHaveSameWord_(string word, pair<unsigned int, unsigned int> row_col_word)
489. {
490.     if (nullptr == root_) return false;
491.
492.     // Make a Stack
493.     LinkStack<BinTreeStackNode*> stack;
494.
495.     // first, make the root of the tree in the stack
496.     stack.push(_CreatBinTreeStackNode_(root_, false));
497.
498.     while (stack.size() > 0)
499.     {
500.         // Pop the element, which is on the top of the stack
501.         BinTreeStackNode* node = (BinTreeStackNode*)stack.top();
502.         stack.pop();
503.
504.         // if nullptr in stack ->continue
505.         if (node->root == nullptr) continue;

```

```

505.
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. }
525.
526. Node* _searchWordInTree_(string word_search)
527. {
528.     if (nullptr == root_) return nullptr;
529.
530.     Node* pCurrent = this->root_;
531.
532.     while (nullptr != pCurrent)
533.     {
534.         if (word_search == pCurrent->key_)
535.         {
536.             return pCurrent;
537.         }
538.
539.         while (nullptr != pCurrent && word_search < pCurrent->key_)
540.         {
541.             pCurrent = pCurrent->left_;
542.         }
543.
544.         while (nullptr != pCurrent && word_search > pCurrent->key_)
545.         {
546.             pCurrent = pCurrent->right_;
547.         }
548.     }
549.

```



```

550.     return nullptr;
551. }
552.
553.  /*
554.  ** Delete all the " "(space)
555.  */
556.  string& _clearAllSpace_(string& str)
557.  {
558.      int index = 0;
559.
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.  }
569.
570.  bool _deleteNodeBinarySearchTree_(string dele_key)
571.  {
572.      if (nullptr == root_) throw MyErrorInfo("nullptr_root_in_DeleteNodeBinarySearchTre
e(string key_del)");
573.
574.      // First of all, need to find the node, which need to delete
575.      Node* pCurrent = root_;
576.      Node* pParent = pCurrent;
577.
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.          }

```

```

594.
595.     if (pCurrent == nullptr)
596.     {
597.         return false;
598.     }
599. }
600.
601. #if 0
602.     cout << "pParent : " << pParent->key_ << endl;
603.     cout << "pCurrent : " << pCurrent->key_ << endl;
604. #endif
605.
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;
614.
615.     delete pCurrent;
616.     pCurrent = nullptr;
617.     return true;
618. }
619.
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;
632.
633.     pParent->left_ = deleNode->right_;
634.     delete deleNode;
635.     deleNode = nullptr;
636.     return true;
637. }

```

```

638.
639.
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;
652.
653.          pParent->left_ = pCurrent->right_;
654.          delete deleNode;
655.          deleNode = nullptr;
656.          return true;
657.      }
658.
659.
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;
670.
671.          pParent->right_ = deleNode->left_;
672.          delete deleNode;
673.          deleNode = nullptr;
674.          return true;
675.      }
676.
677.
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;
688.
689.         pParent->right_ = deleNode->right_;
690.         delete deleNode;
691.         deleNode = nullptr;
692.         return true;
693.     }
694.
695.
696.     // 最复杂的情况 - The WORST situation
697.     else if (pCurrent->left_ != nullptr && pCurrent->right_ != nullptr)
698.     {
699.         Node* deleNode = pCurrent;
700.
701.         pCurrent = pCurrent->right_;
702.
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;
716.
717.             delete pCurrent;
718.             pCurrent = nullptr;
719.             return true;
720.         }
721.
722.
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.        }
745.
746.        deleteNode->key_ = pCurrent->key_;
747.        pParent->left_ = pCurrent->right_;
748.        delete pCurrent;
749.        pCurrent = nullptr;
750.    }
751.    return true;
752. }
753. }
754.
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. }
765.
766. };

```

● WordInfo.hpp

```
1. #pragma once
2. #include "LinkList.hpp"
3. #include "MyErrorInfo.hpp"
4. #include "BinarySearchTree.hpp"
5.
6. class BinarySearchTree;
7. class WordInfo;
8. struct Node;
9.
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;
5.
6.  template<typename T>
7.  struct LinkStack
8.  {
9.      // ----- StackNode -----
10. private:
11.     struct StackNode
12.     {
13.         T data_;
14.         StackNode* next_;
15.     };
16.
17.     StackNode* header_;
18.     unsigned int size_;
19.
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");
28.
29.     }
30.
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.     }
37.
38.     void push(const T& data)
39.     {
40.         _push_(data);
41.     }
42.
43.     T top() const

```

```

44.     {
45.         return _top_();
46.     }
47.
48.     int size() const
49.     {
50.         return _size_();
51.     }
52.
53.     bool empty() const
54.     {
55.         return _empty_();
56.     }
57.
58.     void pop()
59.     {
60.         _pop_();
61.     }
62.
63.     void clear() const
64.     {
65.         _clear_();
66.     }
67.
68.     void desgroy() const
69.     {
70.         _desgroy_();
71.     }
72.
73. private:
74.
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.     }
84.
85.     T _top_() const
86.     {
87.         if (nullptr == this->header_ || 0 == this->size_) throw MyErrorInfo("null_stack");

```



```

88.         return this->header_->next_->data_;
89.     }
90.
91.     int _size() const
92.     {
93.         if (nullptr == this->header_ || 0 == this->size_) return 0;
94.         return this->size_;
95.     }
96.
97.     bool _empty() const
98.     {
99.         return this->size_;
100.    }
101.
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.    }
111.
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_;
117.
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.    }
128.
129.    void _desgroy()
130.    {
131.        if (nullptr == this->header_) return;
132.        StackNode* pCurrent = this->header_->next_;

```

```
133.         StackNode* pRear = this->header_;
134.
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.     }
145.
146. };
```

● LinkedList.hpp

```

1. #pragma once
2. #include "MyErrorInfo.hpp"
3. #include <string>
4. #include <iostream>
5.
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)
   ;
10.
11. template<typename T>
12. ostream& operator<<(ostream& cout, LinkNode<T>& pCurrent)
13. {
14.     cout << pCurrent.data_;
15.     return cout;
16. }
17.
18. template<typename T>
19. bool operator>(const LinkNode<T>& node1, const LinkNode<T>& node2)
20. {
21.     return node1.data_ > node2.data_;
22. }
23.
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. };
35.
36. template<typename T>
37. class LinkList
38. {
39.     friend class BinarySearchTree;
40.
41. private:

```

```

42.     LinkNode<T>* header_;
43.
44. public:
45.     LinkList()
46.     {
47.         header_ = new LinkNode<T>;
48.         header_->next_ = nullptr;
49.     }
50.
51.     void insert(T data)
52.     {
53.         _insert_(data);
54.     }
55.
56.     void foreach() const
57.     {
58.         _foreach_();
59.     }
60.
61.     void sort()
62.     {
63.         _sort_();
64.     }
65.
66.     ~LinkList()
67.     {
68.         _desgroy_();
69.     }
70.
71. private:
72.     void _insert_(T data)
73.     {
74.         if (nullptr == header_) throw MyErrorInfo("null_header_in_function_insert()");
75.
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.         }
85.

```

```

86.     LinkNode<T>* pCurrent = header_->next_;
87.     while (nullptr != pCurrent->next_)
88.     {
89.         pCurrent = pCurrent->next_;
90.     }
91.
92.     LinkNode<T>* newNode = new LinkNode<T>;
93.     newNode->data_ = data;
94.     newNode->next_ = nullptr;
95.     pCurrent->next_ = newNode;
96. }
97.
98. void _foreach() const
99. {
100.     if (nullptr == header_) throw MyErrorInfo("null_header_in_function_foreach()")
    ;
101.
102.     LinkNode<T>* pCurrent = header_->next_;
103.     while (nullptr != pCurrent)
104.     {
105.         cout << *pCurrent;
106.         pCurrent = pCurrent->next_;
107.     }
108.     cout << endl;
109. }
110.
111. void _sort()
112. {
113.     if (nullptr == header_ || nullptr == header_->next_) throw MyErrorInfo("null_h
eader_in_function_sort()");
114.
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.     }
123.
124.     //-----
125.     pCurrent = header_->next_;
126.     LinkNode<T>* pNext = pCurrent->next_;
127.
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.             }
139.
140.             pCurrent = pNext;
141.             pNext = pNext->next_;
142.
143.             if (nullptr == pNext)
144.             {
145.                 pCurrent = header_->next_;
146.                 pNext = pCurrent->next_;
147.             }
148.
149.         }
150.     }
151. }
152.
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_;
163.
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;
5.
6. // ----- ERROR INFO -----
7. class MyErrorInfo : public exception
8. {
9. public:
10.     MyErrorInfo(string errorInfo)
11.     {
12.         this->m_error = errorInfo;
13.     }
14.     ~MyErrorInfo() {}
15.
16.     virtual const char* what()
17.     {
18.         return this->m_error.c_str();
19.     }
20.
21. private:
22.     string m_error;
23. };
```

● main.cpp

```

1. #include "LinkList.hpp"
2. #include "BinarySearchTree.hpp"
3. using namespace std;
4.
5. void testFunc()
6. {
7.     string file_name;
8.     cout << "# Please enter the required file name : "; cin >> file_name; // 1.txt
9.
10.    BinarySearchTree tree;
11.    tree.openFile(file_name);
12.
13.    /*
14.    ** first add rol number, and output in file (output.txt)
15.    */
16.    tree.printTextWithRowNum();
17.
18.    /*
19.    ** Insert all word into the binary search tree
20.    */
21.    tree.makeTree();
22.
23.    /*
24.    ** print the words on screen
25.    ** true - print same word;
26.    ** false - do not print same word
27.    */
28.    tree.printDictionaryOrder(false);
29.
30.    /*
31.    ** Seach word
32.    ** if have this word print this word with its info (row, col, count, length...)
33.    */
34.    tree.searchWord("fire");
35.
36.    /*
37.    ** Delete word
38.    */
39.    tree.deleteWord("student");
40.
41.    /*
42.    ** out put the word by Dictionary order in to the file "output.txt"
43.    */

```



```
44.     tree.outputDictionaryOrderInFile(false);
45. }
46.
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)

```
1. tree.openFile(file_name);
```

2. Вывод статьи с номером строки в файл

```
1. /*
2. ** first add rol number, and output in file (output.txt)
3. */
4. tree.printTextWithRowNum();
```

3. Вставить все слова в двоичное дерево поиска

```
1. /*
2. ** Insert all word into the binary search tree
3. */
4. tree.makeTree();
```

4. Выводит слова (с информационными полями) на экран в лексикографическом порядке

```
1. /*
2. ** print the words on screen
3. ** true - print same word;
4. ** false - do not print same word
5. */
6. tree.printDictionaryOrder(false);
```

5. Тест поиск слов

```
1. /*
2. ** Search word
3. ** if have this word print this word with its info (row, col, count, length...)
4. */
5. tree.searchWord("fire");
```

6. Тест удаление узла по слову

```
1. /*
2. ** Delete word
3. */
4. tree.deleteWord("student");
```

7. Выводит слова (с информационными полями) в файл в лексикографическом порядке

```
1. /*
2. ** out put the word by Dictionary order in to the file "output.txt"
3. */
4. tree.outputDictionaryOrderInFile(false);
```

➤ Для некорректного ввода данных:

Спецификация	Ожидаемый вывод	Данные	Ожидаемый вывод	Операция	Ожидаемый вывод
2.t	ERROR --- cant_open_the_file ---	root_ = nullptr	ERROR - nullptr_root	Неправильный выбор	No corresponding operation !
1. txt				Нет инициализации	no_data
1.c		word in null	ERROR - null_word	Неверные данные при search	Don't have word [...] !
1txt				Неверные данные при delete	delete fail, don't have word [...]
1					
txt					

Рис. 18