Experiment topic: Word frequency statistics and retrieval system for English words based on different strategies

Experiment purpose :

1. the search algorithms based on different storage structures of linear tables, binary sorting trees and hash tables.
2. Master the calculation method of the average search length (ASL) corresponding to different retrieval strategies, and clarify the difference in time performance of different retrieval strategies.
3. Master related sorting algorithms.

Experiment content:

An English article is stored in a text file, based on different storage structures of linear table, binary sorting tree and hash table respectively, to realize word frequency statistics and word retrieval functions. At the same time, calculate the ASL under different retrieval strategies, and make a corresponding comparative analysis of the time performance of different retrieval strategies by comparing the size of the ASL (given in the course design report). The details are as follows.

1. An English article including punctuation marks is stored in the text file InFile.txt, assuming that the number of words in the file does not exceed 5000 at most. Read English words from this file, filtering out all punctuation.
2. Based on the different storage structures of linear table, binary sorting tree and hash table respectively, the statistics of word frequency and the function of word retrieval are realized. Among them, the linear table adopts two different storage structures of sequential table and linked list to realize sequential search respectively, and at the same time realize the half search based on the sequential table; the hash table realizes the hash search based on the open address method and the hash based on the chain address method respectively. find. Thus, a total of 6 different retrieval strategies are implemented.
3. No matter which retrieval strategy is adopted, the functions realized are the same.

Selected test article (InFile.txt):

Excerpt from "I have a dream":

CODE:

Hash.cpp

```
#include "stdafx.h"
#include "Hash.h"

Hash::Hash(const int type, const int size)
{
    Type = type;
    if (Type == Open_Address)
    {
        elem = new Word[size];
        for (int i = 0; i < size; i++)
            elem[i] = { "\0",-1 };
    }
```

```
	else if (Type == Link_Address)
	{
		link_elem = new Link[size];
		for (int i = 0; i < size; i++)
		{
			link_elem[i].data = { "\0",-1 };
			link_elem[i].next = NULL;
		}
	}
	MaxLength = size;
	Length = 0;
}

Hash::~Hash()
{
	if (Type == Open_Address)
		delete[] elem;
	else if (Type == Link_Address)
	{
		for (int i = 0; i < MaxLength; i++)
		{
			Link *ptr = link_elem[i].next;
			while (ptr)
			{
				Link *temp = ptr;
				ptr = ptr->next;
				delete temp;
			}
		}
		delete[] link_elem;
	}
}

int & Hash::operator[](const char *word)
{
	int p = 0;
	switch (Type)
	{
	case Open_Address:
		p = wordadd(word);
		while (strcmp(elem[p % mod]. word, word) && elem[p % mod]. word[0])
		{
			p += 26;
		}
		if (!elem[p % mod].word[0])
		{
			strcpy(elem[p % mod]. word, word);
			elem[p % mod].statistic = 0;
			Length++;
		}
		return elem[p % mod].statistic;
		break;
	case Link_Address:
		p = wordadd(word) % mod;
		Link *ptr = &link_elem[p];
		while (ptr->data.word[0] && strcmp(ptr->data.word, word))
		{
			if (ptr->next)
```

```
                ptr = ptr->next;
            else
            {
                Link *new_link = new Link;
                new_link->data = { "\0",-1 };
                new_link->next = NULL;
                ptr->next = new_link;
                ptr = new_link;
                break;
            }
        }
        if (!ptr->data. word[0])
        {
            strcpy(ptr->data. word, word);
            ptr->data.statistic = 0;
            Length++;
        }
        return ptr->data.statistic;
        break;
    }
}

Word* Hash::show()
{
    Word *word = new Word[5000];
    int n = 0;
    switch (Type)
    {
    case Open_Address:
        for (int i = 0; i < MaxLength; i++)
        {
            if (elem[i].statistic > 0)
            {
                strcpy(word[n].word, elem[i].word);
                word[n++].statistic = elem[i].statistic;
            }
        }
        break;
    case Link_Address:
        for (int i = 0; i < MaxLength; i++)
        {
            if (link_elem[i].data.statistic > 0)
            {
                Link *ptr = &link_elem[i];
                while (ptr)
                {
                    strcpy(word[n].word, ptr->data.word);
                    word[n++].statistic = ptr->data.statistic;
                    ptr = ptr->next;
                }
            }
        }
        break;
    default:
        break;
    }
    return word;
}
```

Hash.h
```
#pragma once
#ifndef HASH_H_
#define HASH_H_
#include "word.h"
class Hash
{
private:
	Word *elem;
	struct Link
	{
		Word data;
		Link *next;
	};
	Link* link_elem;
	int Type;
	int Length, MaxLength;
	int wordadd(const char* word)
	{
		int a = 0;
		while (*word++)
			a += *word;
		return a;
	}
	const int mod = 6656;

public:
	Hash(const int type = Open_Address, const int MaxLength = 6656);
	int & operator[](const char* word);
	int size()
	{
		return Length;
	}
	Word* show();
	~Hash();
};

#endif // !HASH_H_
```

Bst.cpp
```
#include "stdafx.h"
#include "Bst.h"


Bst::Bst()
{
	root->node = { "\0",0 };
	root->LTree = NULL;
	root->RTree = NULL;
	Length = 0;
}

Bst::~Bst()
{
}
```

```
int Bst::insert(const char *word)
{
    Tree *ptr = root, *pre;
    while (ptr)
    {
        int cmp = strcmp(ptr->node. word, word);
        if (cmp == 0)
        {
            ptr->node.statistic++;
            return 0;
        }
        else if(cmp < 0)
        {
            pre = ptr;
            ptr = ptr->RTree;
        }
        else
        {
            pre = ptr;
            ptr = ptr->LTree;
        }
    }
    Tree *new_node = new Tree;
    strcpy(new_node->node. word, word);
    new_node->node.statistic = 1;
    new_node->LTree = NULL;
    new_node->RTree = NULL;
    Length++;
    if (strcmp(pre->node. word, word) > 0)
        pre->LTree = new_node;
    else
        pre->RTree = new_node;
    return 1;
}

Word Bst::find(const char *word)
{
    Tree *ptr = root;
    int cmp;
    while (ptr && (cmp = strcmp(ptr->node. word, word)))
    {
        if (cmp > 0)
            ptr = ptr->LTree;
        else
            ptr = ptr->RTree;
    }
    if (ptr)
        return ptr->node;
    else
        return { "\0",0 };
}

Word *Bst::show()
{
    Word *word = new Word[Length];
    travel(word, root);
    return word;
```