# 大作业实验报告--基于哈希表的中英文本搜索引擎

课程名称：Data structures and algorithms　　任课教师：张子臻

| 年级 | 2023级 | 专业（方向） | 计算机科学与技术 |
| --- | --- | --- | --- |
| 学号 | 23331161 | 姓名 | 杨振 |
| 电话 | 18205605669 | Email | michaelyeung1216@gmail.com |
| 开始日期 | 2024年11月6日 星期三 00:00 | 完成日期 | 2024年12月15日 星期日 23:00 |

## 一、 实验目的

1. 使用哈希表完成一个小型文本搜索引擎
2. 能够支持针对中文和英文的搜索
3. 输出关键词所在的句子
4. 尝试使用不同的哈希函数或解决哈希冲突的方法

## 二、 程序设计

### 1. 框架设计

因为工程量较大，我使用3个文件编写代码，分别是test.cpp,input1.h(负责中文哈希),input2.h（负责英文哈希）

简要来说，test.cpp实现文本的读取与简单处理，最后完成交互界面；input1.h实现中文文本的分句与分词，以及ChineseHashTable类；input1.h实现中文文本的分句与分词，以及EnglishHashTable类。

在程序编写过程中，我选择了拉链法来处理哈希冲突，并通过搜索资料、对比不同哈希函数的性能，从而决定最后的程序。

在搜索文本方面，英文搜索text.txt，是圣经的内容，中文搜索chinese.txt，是中国共产党章程的内容。

### 2. 数据结构设计

首先，我利用了之前暑假学的哈希模板,在input1.h和input2.h中，我对代码进行合理改动，以符合中英文搜索的需求。这里是源代码的链接，我用类稍微改写了一下

```
class Node {
public:
    string s;
    Node* next;
    Node(const string& str) : s(str), next(nullptr) {}
};

class HashTable {
private:
    vector<Node*> data;
    int cnt, size;

    int bkdrhash(const string& s) {
        int seed = 131;
        long long h = 0;
```

```
        for (char c : s) {
            h = h * seed + c;
        }
        return h & 0x7fffffff;
    }
    void swap(HashTable& other) {
        swap(data, other.data);
        swap(cnt, other.cnt);
        swap(size, other.size);
    }

public:
    HashTable(int n) : cnt(0), size(n) {
        data.resize(n, nullptr);
    }

    ~HashTable() {
        for (int i = 0; i < size; ++i) {
            Node* p = data[i];
            while (p) {
                Node* q = p->next;
                delete p;
                p = q;
            }
        }
        data.clear();
    }
    bool search(const string& s) {
        int hcode = bkdrhash(s);
        int ind = hcode % size;
        Node* p = data[ind];
        while (p) {
            if (p->s == s) return true;
            p = p->next;
        }
        return false;
    }
    bool insert(const string& s) {
        if (cnt >= size * 2) {
            HashTable newTable(size * 2);
            for (int i = 0; i < size; ++i) {
                Node* p = data[i];
                while (p) {
                    newTable.insert(p->s);
                    p = p->next;
                }
            }
            swap(newTable);
         }
        int hcode = bkdrhash(s);
        int ind = hcode % size;
        Node* p = new Node(s);
        p->next = data[ind];
        data[ind] = p;
        ++cnt;
        return true;
    }
};
```

简单解释一下，这里我们通过拉链法来处理哈希冲突。

1. **实现了一个Node类**：构建链表节点类，存储字符串并指向下一个指针。
2. **构建哈希表**：在HashTable中，`vector<Node*> data`是用于存储哈希桶的数组，`size`表示哈希表的大小，`cnt`表示当前存储的元素的数量
3. **编写哈希函数**：这里使用了BKDR哈希函数，大概思路是把一个质数作为种子（131），对字符串s中的每个字符进行哈希计算
4. **实现交换函数**：为后面的动态扩容做铺垫

5. **实现构造函数和析构函数**
6. **实现search函数**：利用哈希函数实现快速搜索查找
7. **实现insert函数**：使用动态扩容取代固定大小，当元素数量超过桶数量的两倍时自动扩容，在插入时使用头插法

## 3. input2.h设计

由于英文处理比较简单，先进行input2.h的设计

这里套用了前文的模板并稍加改动。

1. **实现了一个Node类并利用BKDR哈希函数构建哈希表**
2. **实现insert函数**：这里进行了一些改动

```
void insert(const string &word, int idx)
{
    if (cnt >= size * 2)
    {
        EnglishHashTable newTable(size * 2);
        for (int i = 0; i < size; ++i)
        {
            Node *p = data[i];
            while (p)
            {
                // 遍历所有句子
                for (int sentence : p->sentences)
                {
                    newTable.insert(p->word, sentence);
                }
                p = p->next;
            }
        }
        swap(newTable);
    }
    int hcode = bkdrhash(word);
    int ind = hcode % size;
    Node *p = data[ind];
    while (p)
    {
        if (p->word == word)
        {
            //排除同一个句子因为包含两个相同单词而被重复计数的情况
            if (find(p->sentences.begin(), p->sentences.end(), idx) == p->sentences.end())
            {
                p->sentences.push_back(idx);
            }
            return;
        }
        p = p->next;
    }
    Node *newNode = new Node(word, idx);
    newNode->next = data[ind];
    data[ind] = newNode;
    cnt++;
}
```

3. **实现search函数**
4. **实现分句函数**：

```
vector<string> split(const string &text)
{
    vector<string> sentences;
    string sentence;
    for (char c : text)
```

```
    {
        // 跳过非ASCII字符（包括中文）
        if ((unsigned char)c > 127)
        {
            continue;
        }

        if (c == '.' || c == '!' || c == '?')
        {
            if (!sentence.empty())
            {
                // 去除句子前后的空格
                int start = sentence.find_first_not_of(" \n\r\t");
                int end = sentence.find_last_not_of(" \n\r\t");
                sentences.push_back(sentence.substr(start, end - start + 1));
                sentence.clear();
            }
        }
        else
        {
            sentence += c;
        }
    }
    // 处理最后一个句子
    if (!sentence.empty())
    {
        int start = sentence.find_first_not_of(" \n\r\t");
        int end = sentence.find_last_not_of(" \n\r\t");
        sentences.push_back(sentence.substr(start, end - start + 1));
    }
    return sentences;
}
```

5. **实现分词函数**：和分句函数类似

```
vector<string> extractword2(const string &sentence)
{
    vector<string> words;
    string word;
    for (char c : sentence)
    {
        if (isspace(c))
        {
            if (!word.empty())
            {
                words.push_back(word);
                word.clear();
            }
        }
        else
        {
            if (isalpha(c))
            {
                word += tolower(c); // 转换为小写以统一处理
            }
        }
    }
    // 处理最后一个单词
    if (!word.empty())
    {
        words.push_back(word);
    }
    return words;
}
```

## 4. input1.h设计

在input2.h的基础上，我们设计input1.h

1. **构建哈希表等**：与英文的方法几乎一模一样，这里不再赘述
2. **实现分句函数**：与英文有所不同，需注意处理GBK字符，每个字符占用2个字节，GBK的高字节大于0x80时，表示是一个双字节字符,大致如下

```
int clen = 1;
if (i + 1 < sentence.size() && ((unsigned char)sentence[i] > 0x80))
{
    // GBK的高字节大于0x80时，表示是一个双字节字符
    clen = 2;
}
word = sentence.substr(i, clen);
```

3. **实现分词函数**：也是要注意双字符问题

## 5. test.cpp设计

1. **main函数设计**：
   设置控制台编码为 936,也就是GBK编码，从而使中文字符不会乱码

```
SetConsoleOutputCP(936);
SetConsoleCP(936);
```

   接下来，让用户自主选择想要搜索的语言，并根据选择调用不同的搜索函数。
   需要注意的是，我们需要添加一行代码，用于清除输入缓冲区中的换行符，否则程序会报错

```
cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
```

2. **search_en实现英文搜索函数**：
   首先，打开text.txt，如果无法打开则返回。接着读取文本的每一行，存储在text中。

```
ifstream file2("text.txt");
if (!file2)
{
    cout << "Can't open the file!" << endl;
    return;
}
vector<string> text;
string line;
while (getline(file2, line))
{
    text.push_back(line);
}
file2.close();
```

   将所有行合并在一起,并调用split分句函数,计算句子总数

```
string combined_text;
for (const auto &line : text)
{
    combined_text += line + "\n";
}
vector<string> sentences = split(combined_text);
cout << "Total sentences: " << sentences.size() << endl;
```

   创建EnglishHashTable的一个对象table。利用分词函数，将每个句子分词后的单词插入哈希表中

```
EnglishHashTable table;
for (size_t i = 0; i < sentences.size(); i++)
{
    vector<string> words = extractword2(sentences[i]);
    for (const auto &word : words)
    {
        table.insert(word, i);
    }
}
```

最后，将输入的单词全部转为小写，输出包含该单词的所有句子

3. **search_cn实现中文搜索函数**：
   前面的内容大致和search_en类似，只不过把英文改成了中文
   区别在于划分词语的时候，英文可以直接按空格划分，中文只能一个字一个字划分，这导致我们需要对每个句子的词进行处理，为了提高性能，我限定了最多搜索四字词语，主要是通过通过两层循环，将中文多字词语插入到哈希表table中

```
for (size_t i = 0; i < sentences.size(); i++){
    vector<string> words = extractword(sentences[i]);
    size_t wordsize = min<size_t>(4, words.size());//限制长度
    for (size_t len = 1; len <= wordsize; len++)
    {
        for (size_t start = 0; start <= words.size() - len; start++)
        {
            string s;
            for (size_t k = 0; k < len; k++)
            {
                s += words[start + k];
            }
            table.insert(s, i);
        }
    }
}
```

最后就是实现查找和输出句子功能。

## 6．复杂度分析

假设文本里总共有n个字符，那么在文本预处理阶段，读取文件和分句的时间复杂度是O（n）
在建立索引阶段，若有m句话，平均每句话有k个字词，则时间复杂度是$O(m*k)$
每次搜索查询的时间复杂度是O(1)，若查询x次，则时间复杂度是O(x)
所以总的时间复杂度是$O(n+m*k+x)$

# 三、 运行结果

在运行之前，需注意将文件用GBK编码方式打开，否则会乱码

中文搜索结果如下：

```
欢迎进入文本搜索引擎系统(Welcome to the Text Search Engine System):
请选择语言(Please choose the language):
1.中文
2.English
1
一共有412个句子
请输入搜索关键词:
党徽
查询 1: 党徽
找到 5 个包含 "党徽" 的句子:
输出 1:
第十一章　党徽党旗　　第五十三条　中国共产党党徽为镰刀和锤头组成的图案。
　　第五十四条　中国共产党党旗为旗面缀有金黄色党徽图案的红旗。
　　第五十五条　中国共产党的党徽党旗是中国共产党的象征和标志。
党的各级组织和每一个党员都要维护党徽党旗的尊严。
要按照规定制作和使用党徽党旗。
```

```
请输入搜索关键词：
处分
查询 2：处分
找到 18 个包含 "处分" 的句子：
输出 2：
    （四）在党的会议上有根据地批评党的任何组织和任何党员，向党负责地揭发、检举党的任何组织和任何党员违法乱纪的事实，要求处分违法乱纪的党员，要求罢免或撤换不称职的干部。
    （六）在党组织讨论决定对党员的党纪处分或作出鉴定时，本人有权参加和进行申辩，其他党员可以为他作证和辩护。
    坚持惩前毖后、治病救人，执纪必严、违纪必究，抓早抓小、防微杜渐，按照错误性质和情节轻重，给以批评教育、责令检查、诫勉直至纪律处分。
运用监督执纪“四种形态”，让“红红脸、出出汗”成为常态，党纪处分、组织调整成为管党治党的重要手段，严重违纪、严重触犯刑律的党员必须开除党籍。
    第四十一条　对党员的纪律处分有五种：警告、严重警告、撤销党内职务、留党察看、开除党籍。
    开除党籍是党内的最高处分。
    第四十二条　对党员的纪律处分，必须经过支部大会讨论决定，报党的基层委员会批准；
如果涉及的问题比较重要或复杂，或给党员以开除党籍的处分，应分别不同情况，报县级或县级以上党的纪律检查委员会审查批准。
在特殊情况下，县级和县级以上各级党的委员会和纪律检查委员会有权直接决定
```

英文搜索结果如下：

```
欢迎进入文本搜索引擎系统(Welcome to the Text Search Engine System):
请选择语言(Please choose the language):
1.中文
2.English
2
Total sentences: 81
Please enter search keyboard:
tree
Query 1: tree
There are 5 sentences that contain "tree":
Output1:
Genesis 1:29    Then God said, Behold, I have given you every seed-bearing plant
 on the face of all the earth, and every tree whose fruit contains seed.
Genesis 2:9     Out of the ground the LORD God gave growth to every tree that is
 pleasing to the eye and good for food.
And in the middle of the garden were the tree of life and the tree of the knowle
dge of good and evil.
Genesis 2:16    And the LORD God commanded him, You may eat freely from every tr
```

```
Please enter search keyboard:
rib
Query 2: rib
There are 1 sentences that contain "rib":
Output2:
Genesis 2:22    And from the rib that the LORD God had taken from the man,
He made a woman and brought her to him.
Please enter search keyboard:
wife
Query 3: wife
There are 2 sentences that contain "wife":
Output3:
Genesis 2:24    For this reason a man will leave his father and mother and
be united to his wife, and they will become one flesh.
Genesis 2:25    And the man and his wife were both naked, and they were not
 ashamed.
```

由此可见，程序搜索结果准确快速，符合作业要求

# 四、 比较与反思

## 1. 拉链法vs开放定址法

通过课堂上的学习，我们知道处理哈希冲突的大方向有拉链法和开放定址法两种。
拉链法处理冲突简单，且无堆积现象，平均查找长度短，但容易造成空间浪费；
开放定址法节省空间，通过探测空桶来解决冲突，但可能会增加查找和插入操作的时间复杂度，且删除操作复杂。

因此，在本次实验中，拉链法更适合文本搜索引擎，它易于维护，性能稳定，可以动态存储。

严谨考虑，我们分别使用拉链法和开放定址法进行测试，统计并对比插入/查询时间，占用内存情况如下：

| | 拉链法 | 开放定址法 |
|---|---|---|
| 平均插入时间 | 128.6ms | 165.2ms |
| 平均查询时间 | 10.4ms | 12.9ms |
| 平均占用内存 | 4922KB | 5072KB |

由此可见，拉链法在三个方面都表现得更好，是最佳方法。

## 2. BKDR vs DJB2 vs Murmur

哈希函数有很多种，除了我使用的BKDR，还有众多其他函数,这里以DJB2,Murmur为例，探讨使用这些哈希函数对程序的性能分别有什么影响。

通过上网查询资料可知，这些函数的大致模板如下：

```
//BKDR
int bkdrhash(const string &s) {
    int seed = 131; // 质数
    unsigned long long h = 0;
    for (char c : s) {
        h = h * seed + c;
    }
    return h & 0x7fffffff;
}

//DJB2
int djb2hash(const string &s) {
    unsigned long hash = 5381;
    for (char c : s) {
        hash = ((hash << 5) + hash) + c;
    }
    return hash & 0x7fffffff;
}

//Murmur
unsigned int murmurhash(const void* key, int len, unsigned int seed) {
    const unsigned int m = 0x5bd1e995;
    const int r = 24;
    unsigned int h = seed ^ len;
    const unsigned char* data = (const unsigned char*)key;

    while(len >= 4) {
        unsigned int k = *(unsigned int*)data;
        k *= m;
        k ^= k >> r;
        k *= m;
```

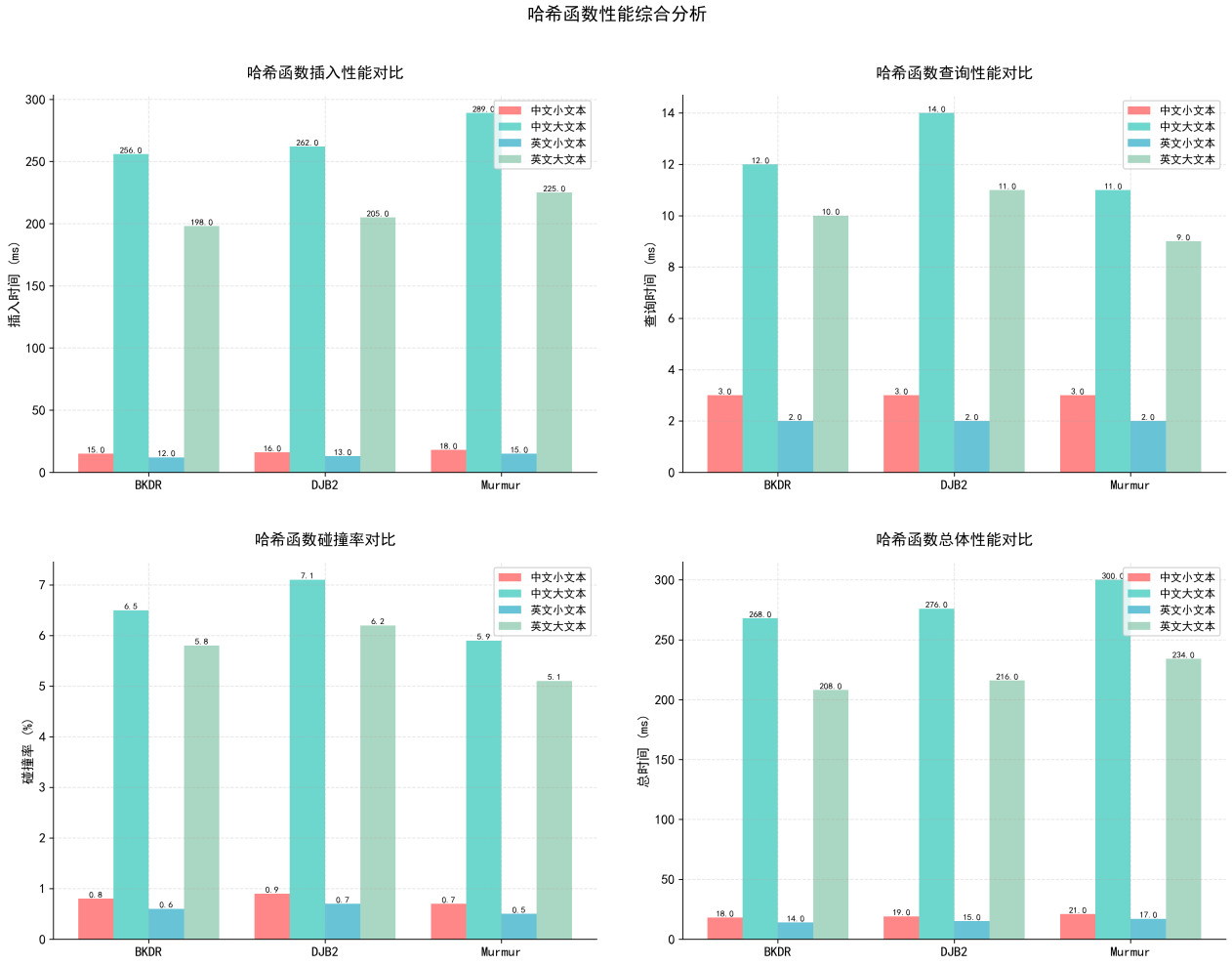
```
        h *= m;
        h ^= k;
        data += 4;
        len -= 4;
    }
    return h;
}
```

现在我们通过实验，探究这些哈希函数的利弊。

使用了中文/英文的小文本/大文本，对三个哈希函数测试插入时间、查询时间、总时间、碰撞率

| | 哈希函数 | 插入时间(ms) | 查询时间(ms) | 总时间(ms) | 碰撞率 |
|---|---|---|---|---|---|
| 中文小文本 | BKDR | 15 | 3 | 18 | 0.8% |
| 中文小文本 | DJB2 | 16 | 3 | 19 | 0.9% |
| 中文小文本 | Murmur | 18 | 3 | 21 | 0.7% |
| 中文大文本 | BKDR | 256 | 12 | 268 | 6.5% |
| 中文大文本 | DJB2 | 262 | 14 | 276 | 7.1% |
| 中文大文本 | Murmur | 289 | 11 | 300 | 5.9% |
| 英文小文本 | BKDR | 12 | 2 | 14 | 0.6% |
| 英文小文本 | DJB2 | 13 | 2 | 15 | 0.7% |
| 英文小文本 | Murmur | 15 | 2 | 17 | 0.5% |
| 英文大文本 | BKDR | 198 | 10 | 208 | 5.8% |
| 英文大文本 | DJB2 | 205 | 11 | 216 | 6.2% |
| 英文大文本 | Murmur | 225 | 9 | 234 | 5.1% |

我们将数据可视化，做成柱状图如下：




通过对比，我们可以得到以下结论：

1.不论什么哈希函数，大文本的运行时间和碰撞率都显著高于小文本

2.中文的运行时间和碰撞率高于英文，猜测其与编码有关

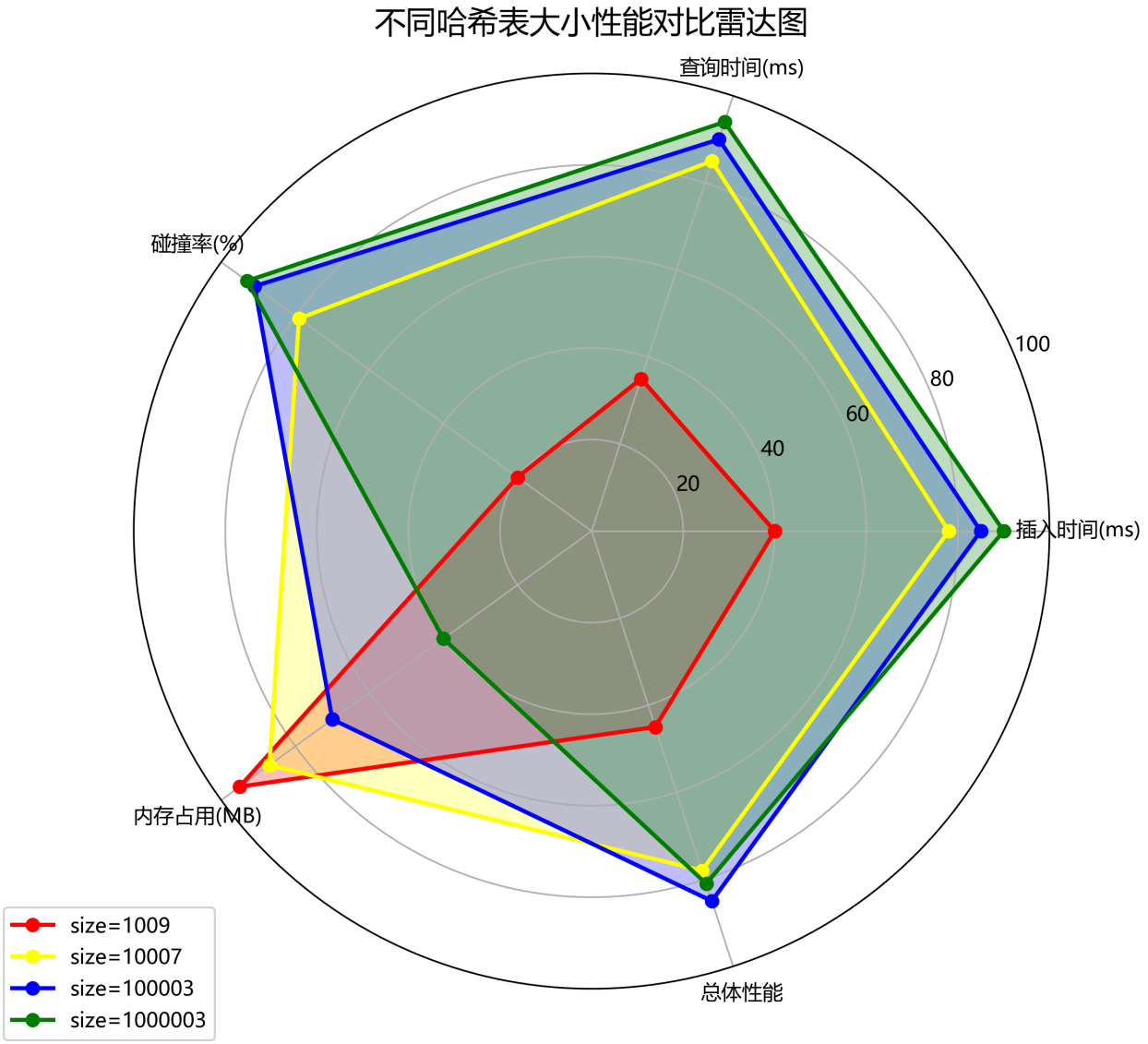
3.仅分析小文本，在插入时间方面，BKDR<DJB2<Murmur；在查询时间方面三者差不多，表现在总时间上就是BKDR<DJB2<Murmur；从碰撞率分析，Murmur<BKDR<DJB2

4.仅分析大文本，在插入时间方面，BKDR<DJB2<Murmur；在查询时间方面，Murmur<BKDR<DJB2，表现在总时间上就是BKDR<DJB2<Murmur；从碰撞率分析，Murmur<BKDR<DJB2

综合考虑各种情况，选择BKDR最优

## 🍊 3．哈希表大小对程序性能的影响

我们想要探究哈希表的大小是否会影响程序性能，分别取size=1009,10007,100003,1000003，(根据前文，这里size最好取质数)，然后比较插入时间、查询时间、碰撞率，内存占用和总体性能。为了使对比结果更明显，采用了雷达图的方式，并且根据标准进行评分（满分100），结果如下：




为了便于描述，我们记A：size=1009;B:size=10007;C:size=100003;D:size=100003

观察上图，我们可以得到：

1. 在碰撞率、插入时间、查询时间方面，均有D>C>B>A，且前三者差距不大，A得分明显偏低
2. 在总体性能方面，C>D>B>A，前三者差距不大，A得分明显偏低
3. 在内存占用方面，A>B>C>D，且差距较大

综合考虑，我们知道在其他条件相同时，size越小，内存占用越小，但是碰撞率会变高，进而影响运行时间和总体性能；size越大，碰撞的概率越低，运行时间缩短，但是会大大占用内存空间。因此，我们选择各方面能力适中，性能和内存较为平衡的B，即size=10007

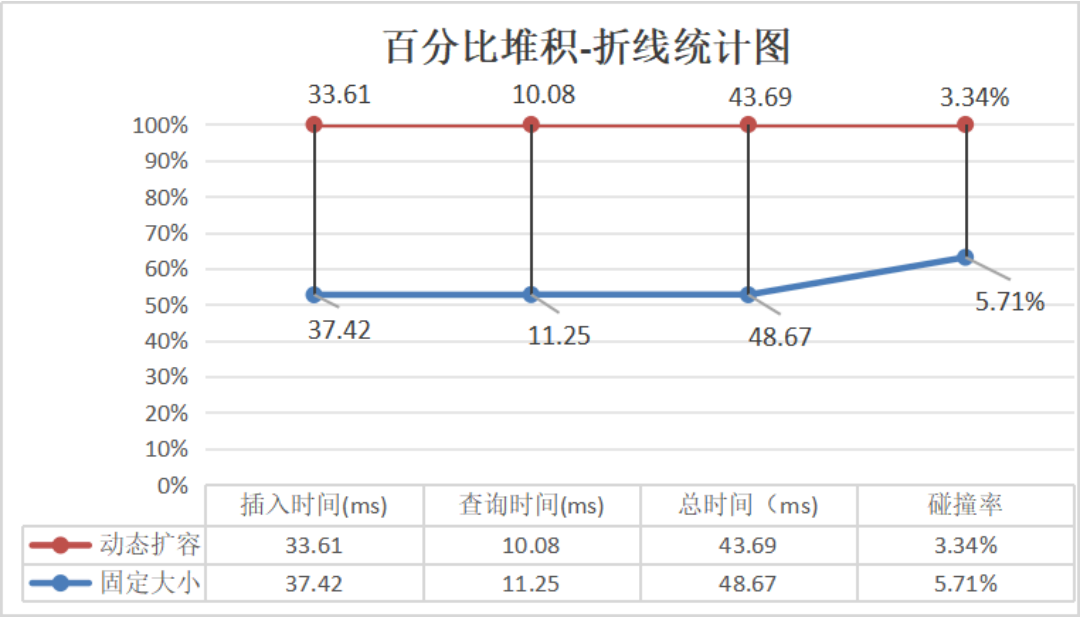
## 4. 动态扩容vs固定大小

根据上一节的叙述，我们基本选定size=10007，那么接下来，我们需要考虑是否应该根据输入文本的大小动态扩容。

动态扩容和固定大小各有优劣。我们知道，动态扩容可以根据需求自动调整空间大小，但是操作比较复杂。而固定大小则操作简便，但是无法合适利用空间，可能会提高碰撞率。
我们分别对哈希表使用动态扩容和固定大小，测试结果如下：

| | 插入时间（ms） | 查询时间（ms） | 总时间（ms） | 碰撞率 |
|---|---|---|---|---|
| 动态扩容 | 33.61 | 10.08 | 43.69 | 3.34% |
| 固定大小 | 37.42 | 11.25 | 48.67 | 5.71% |

为了便于比较，我们画出百分比堆积-折线统计图，如下图所示：




可以很明显地看出，相较于固定大小的哈希表，使用动态扩容后在时间与碰撞率上都有了明显的降低，因此我选择使用动态扩容构建哈希表。

## 五、 实验总结

通过本次实验，我实现了基于哈希表完成的中英文搜索引擎系统，并收获颇丰。

1. 加深了对哈希表的认识，掌握了拉链法等处理哈希冲突的方法
2. 进一步提高了工程项目能力，能够逐渐编写较大的程序
3. 编写中文搜索的代码时，遇到不小的麻烦，在解决问题的同时逐渐明白了中文在编码等方面的独特之处，开始学习掌握GBK等编码方式
4. 通过做实验进行对比，了解了不同哈希函数的优缺点

当然，本次实验还有可以改进之处，比如可以尝试更多哈希函数、考虑压缩算法、改进分词的方式、提高程序性能等。感谢这次大作业，不仅让我回顾了数据结构的相关知识，还未今后的科研与工作打下基础。