哈夫曼编码 + 二叉字典树 存储中文语料

1.完成内容

1.1

报告对普通字典树，二叉字典树和哈夫曼二叉字典树存储中文语料进行了对比。（存储效率，查找效率）

1.2

相比第一次报告：
1.计算了存储效率和查找效率，并列出了相关公式，进行了多种情况的对比和数据分析。
2.对于构件哈夫曼树的构建，字典树的构建和查找文本都选用了大量的文本（10M）。
3.报告逻辑更加清晰，易于理解。
4.增加了改进方案和结论分析。

2.公式

存储效率 = 字符串长度 / 字典树空间大小
查找效率 = 字符串长度 / 查找次数。

3.程序设计思路

插入文本（10M）
查找文本 不定
使用unsigned char读取文本

3.1普通字典树

读取文本的每个字节，利用每个字节对应的数字构建字典树。
读取查找文本的字节，按没个字节对应的数字去字典树中查找，找到返回1，否则返回0。

3.2二叉字典树

读取文本的每个字节，利用每个字节对应的数字的二进制构建字典树。
读取查找文本的字节，按没个字节对应的数字的二进制去字典树中查找，找到返回1，否则返回0。

3.3哈夫曼二叉字典树

读取文本的每个字节，利用每个字节对应的数字和频率构建哈夫曼树，然后用哈夫曼树的路径构建字典树
读取文本的每个字节，利用每个字节对应的数字得到哈夫曼树中的路径去字典树中查找，找到返回1，否则返回0。

4.性能对比（理论）

4.1普通字典树

因为每一位都开了256位数组，空间浪费大，所以存储效率低。
每一位只对应一步，所以查找效率高。

4.2二叉字典树

因为每一位是只有2位我数组，空间浪费小，所以存储效率高。
每一位对应8步，所以查找效率最低。

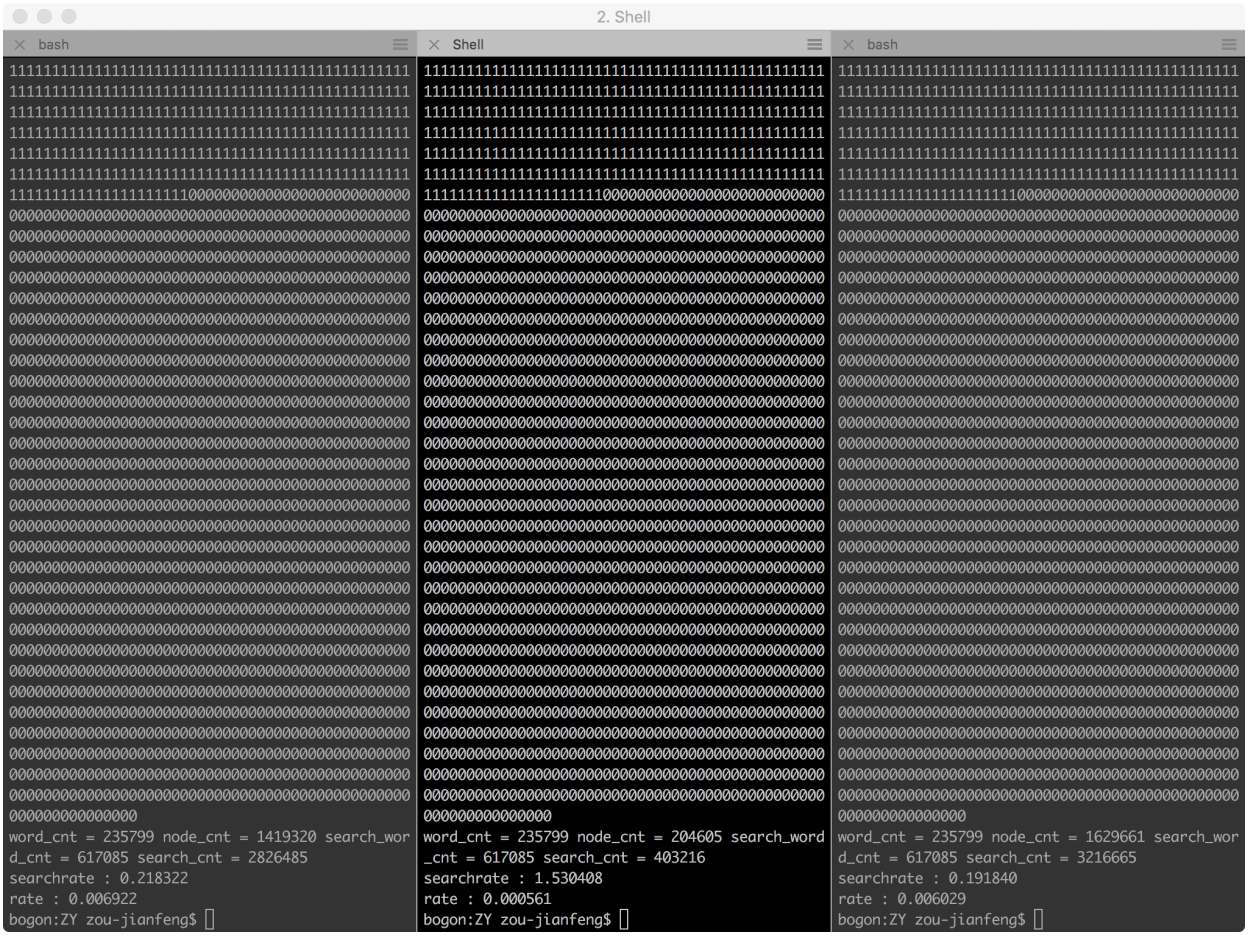
4.3哈夫曼二叉字典树

因为每一位是只有2位我数组，空间浪费小，所以存储效率高。
因为哈夫曼树路径长度和字节对应的数字出现频率有关（频率高，路径短；频率低，路径长），每一位对应哈夫曼路径，所以查找效率低（但会比二叉字典树高）。

5.性能对比（实际）

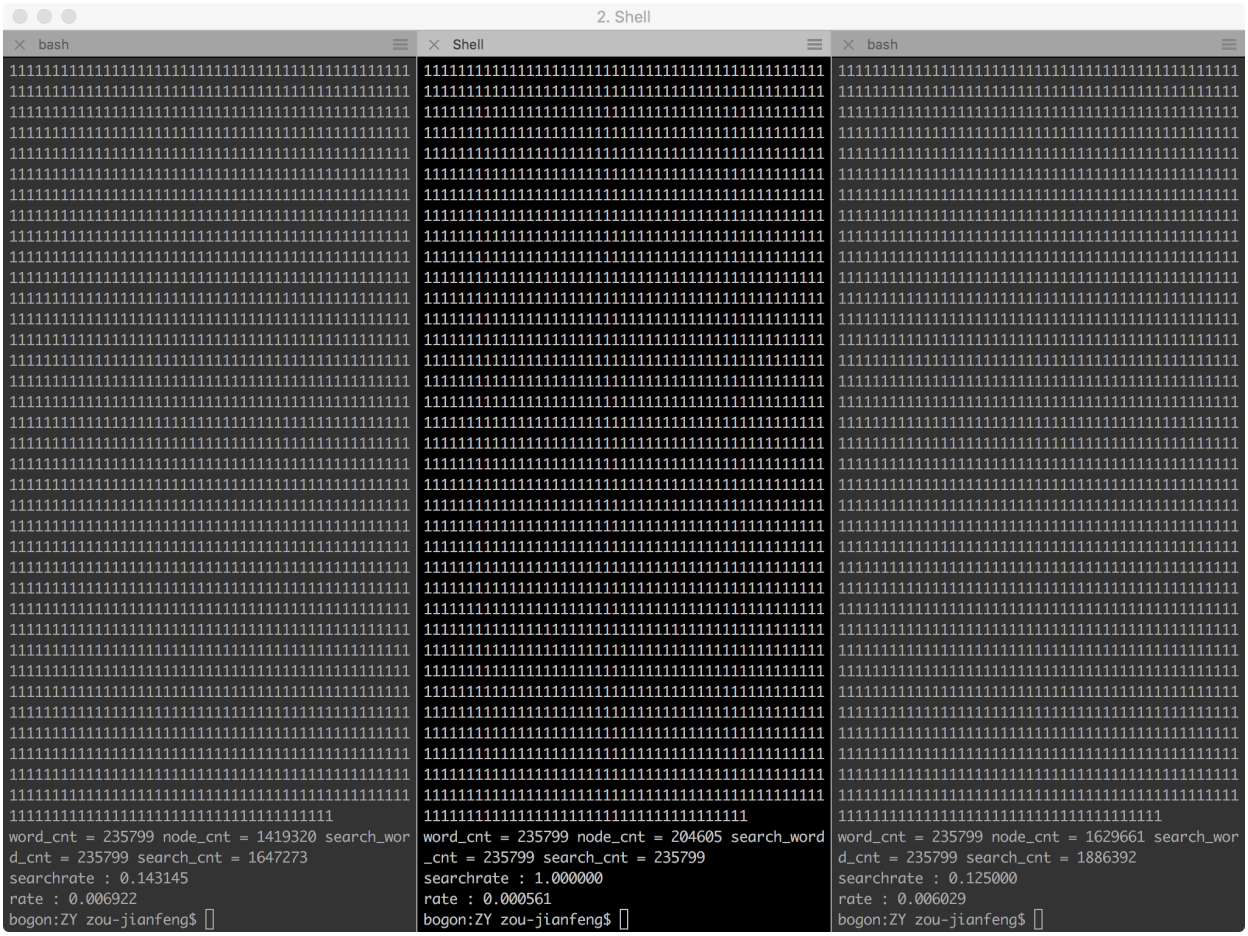
从左至右依次是哈夫曼二叉字典树，普通字典树，二叉字典树
采用相同的插入文本和查找文本。

5.1查找文本同时存在插入文本中存在和不存在的语句。



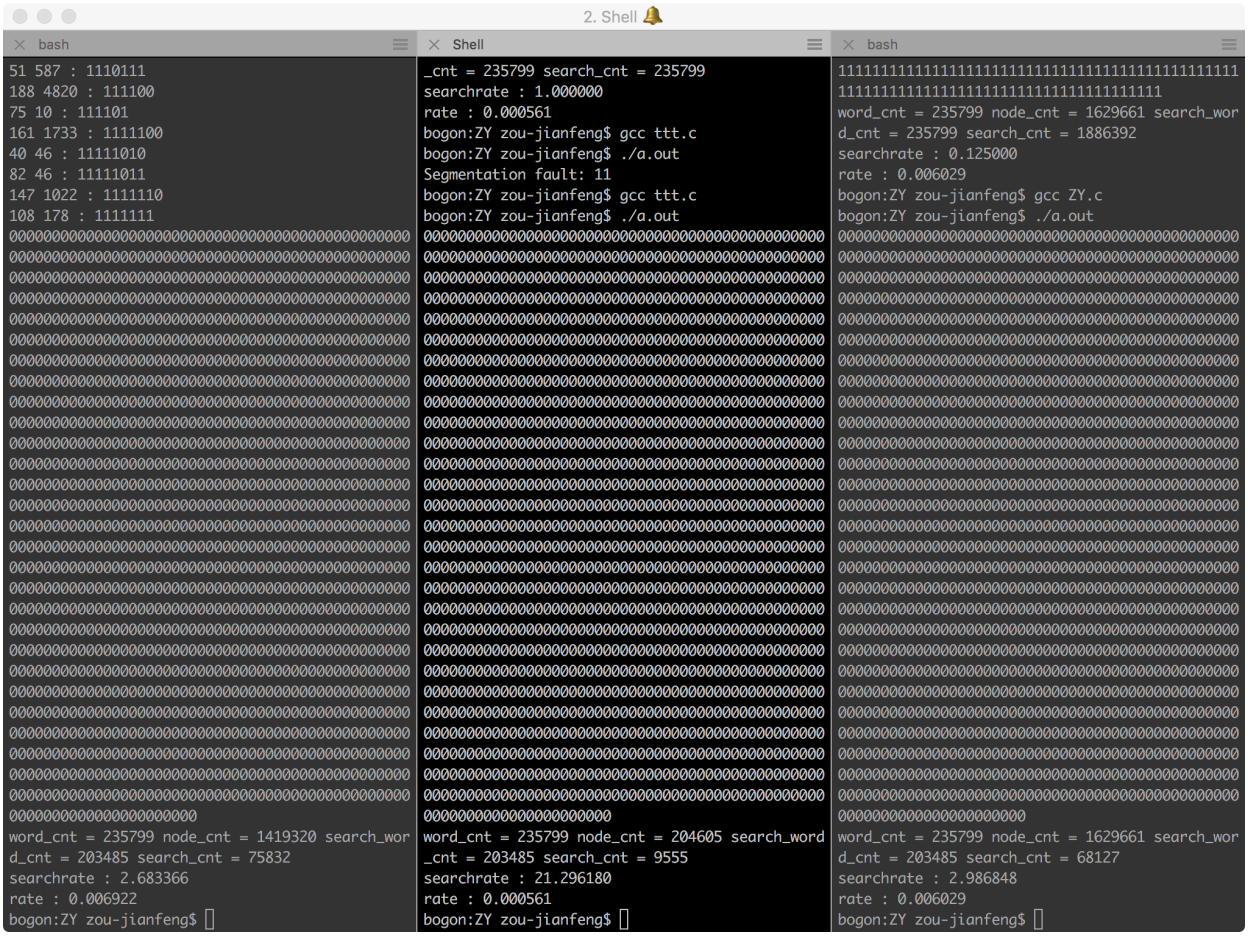
1.存储效率：哈夫曼二叉字典树（接近千分之七）> 二叉字典树（接近千分之六）> 普通字典树（接近万分之六）
2.查找效率：普通字典树（接近1.5）> 哈夫曼二叉字典树（接近0.21）> 二叉字典树（接近0.19）
3.插入文本相同，查找次数：普通字典树（接近4万）< 哈夫曼二叉字典树（接近282万）< 二叉字典树（接近321万）

5.2查找文本只存在插入文本中存在的语句。



1.存储效率不变。
2.查找效率：普通字典树（1） > 哈夫曼二叉字典树（接近0.14）> 二叉字典树（接近0.12）
3.插入文本相同，查找次数：普通字典树（接近2万）< 哈夫曼二叉字典树（接近164万）< 二叉字典树（接近186万）

5.3查找文本只存在插入文本中不存在的语句。

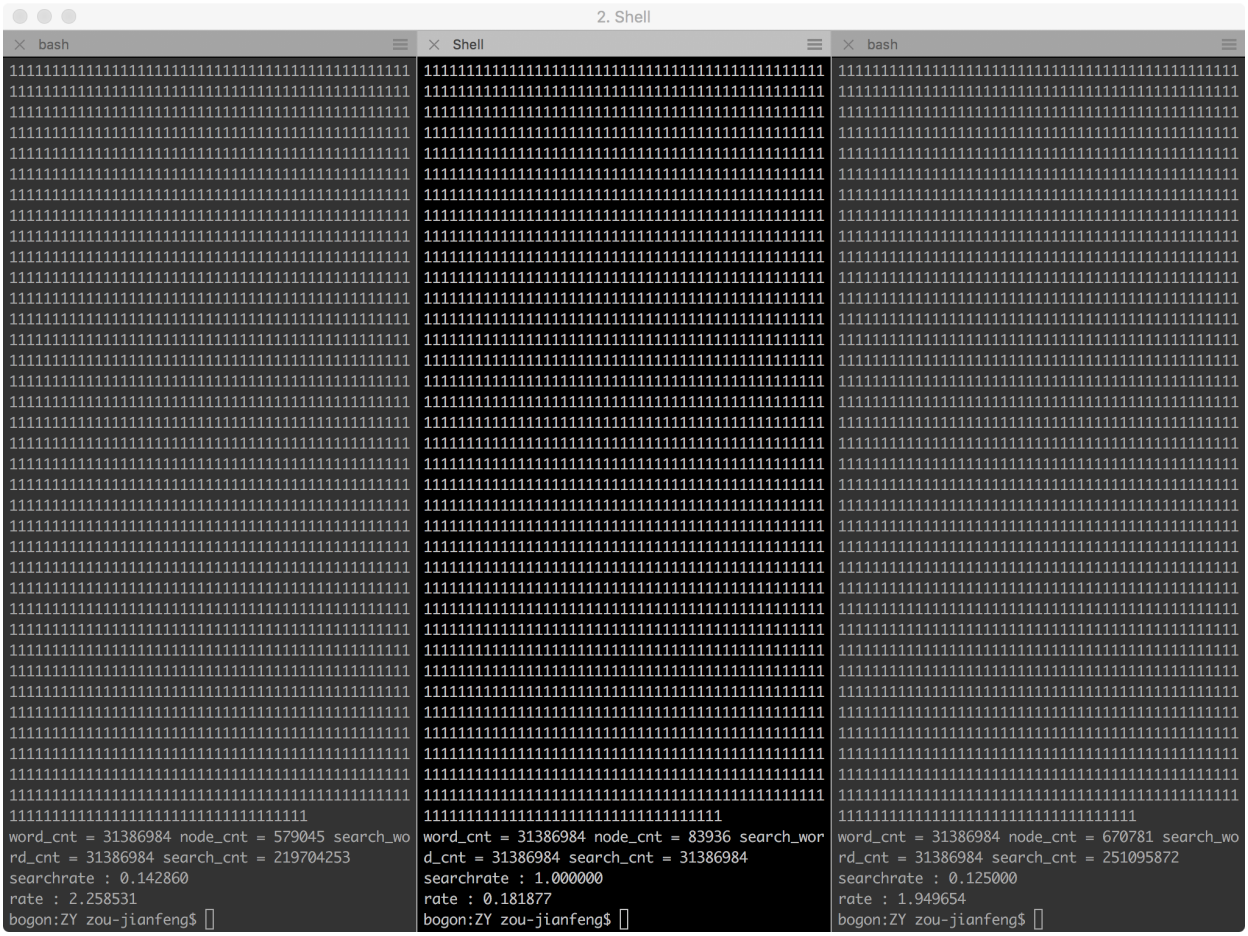


1.存储效率不变

2.查找效率：普通字典树（接近21）> 二叉字典树（接近2.98）> 哈夫曼二叉字典树（接近2.68）

3.插入文本相同，查找次数：普通字典树（9555）< 二叉字典树（68127）< 哈夫曼二叉字典树（75832）

5.4插入文本句式相同 （每行开头都是四个空格）



1.存储效率： 哈夫曼二叉字典树（接近2.25）> 二叉字典树（接近1.94）> 字典树（接近0.18）
2.查找效率： 普通字典树（1） > 哈夫曼二叉字典树（接近0.14）> 二叉字典树（接近0.12）

6结论分析

6.1存储效率

符合预期，哈夫曼二叉字典树和二叉字典树接近，并远远大于普通字典树。
但是当插入文本句式相同时，超找效率都有大幅度提高，但他们之间的关系不会变。

6.2查找效率

1.普通字典树符合预期，查找效率非常高，远远大于另外两个字典树。
2.哈夫曼二叉字典树和二叉字典树在查找文本中插入文本中存在的语句占比大的条件下时符合预期。
当查找文本只存在插入文本语句或同时存在非插入文本语句和插入文本语句时，哈夫曼二叉字典树查找效率快于二叉字典树查找效率。（快大约百分之二）
但是当非插入文本语句占比增加时它们的查找效率越来越接近。当全为非插入文本语句时，二叉字典树查找效率甚至超过了哈夫曼二叉字典树。

7.改进

查找效率和预期存在偏差，查找效率公式和查找算法有待改进。
对于一些特殊情况，实际和预期存在偏差，应该对特殊情况做特殊处理。

8.程序代码

8.1普通字典树

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BASE 256

typedef struct Node {
    int flag;
    struct Node *next[BASE];
} Node, *Trie;

int node_cnt = 0;
Node *get_new_node() {
    Node *p = (Node *)calloc(sizeof(Node), 1);
    p->flag = 0;
    node_cnt += 1;
    return p;
}

void clear(Trie root) {
    if (!root) return;
    for (int i = 0; i < BASE; ++i) clear(root->next[i]);
    free(root);
    return ;
}

Node *insert(Trie root, unsigned char *str) {
    if (!root) root = get_new_node();
    Node *p = root;
    for (int i = 0; str[i]; ++i) {
        int ind = str[i];
        if (!p->next[ind]) p->next[ind] = get_new_node();
        p = p->next[ind];
    }
    p->flag = 1;
    return root;
}

int search_cnt = 0;
int search(Trie root, unsigned char *str) {
    Node *p = root;
    int n = strlen((char *)str);
    int i = 0;
```

```
    while (p && i < n) {
        search_cnt += 1;
        int ind = str[i++];
        p = p->next[ind];
    }
    return (p && p->flag);
}

int main() {
    Trie root = NULL;

    int word_cnt = 0;
    int search_word_cnt = 0;
    unsigned char buffer[10000];

    FILE *fpp = NULL;
    fpp = fopen("/Users/zou-jianfeng/HZ/ZY/1.txt", "r"); //读取60M中文文本
    while (~fscanf(fpp, "%s", buffer)) {
        //printf("%s\n", buffer);
        int n = strlen((char *)buffer);
        word_cnt += n;
        root = insert(root, buffer);
    }
    fclose(fpp);

    FILE *fp = NULL;
    fp = fopen("/Users/zou-jianfeng/HZ/ZY/1.txt", "r"); //读取60M中文文本
    while (~fscanf(fp, "%s", buffer)) {
        int n = strlen((char *)buffer);
        search_word_cnt += n;
        if (search(root, buffer)) {
            printf("1");
        } else {
            printf("0");
            search_cnt += 1;
        }
    }
    fclose(fpp);
    printf("\n");

    printf("word_cnt = %d node_cnt = %d search_word_cnt = %d search_cnt =
 %d\n", word_cnt, node_cnt, search_word_cnt, search_cnt);
    printf("searchrate : %lf\n", 1.0 * search_word_cnt / (1.0 * search_cn
t));
    printf("rate : %lf\n", 1.0 * word_cnt / (1.0 * node_cnt * sizeof(Nod
e)));
    clear(root);
    return 0;
}
```

8.1二叉字典树

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BASE 2

typedef struct Node {
    int flag;
    struct Node *next[BASE];
} Node, *Trie;

int node_cnt = 0;
Node *get_new_node() {
    Node *p = (Node *)calloc(sizeof(Node), 1);
    p->flag = 0;
    node_cnt += 1;
    return p;
}

void clear(Trie root) {
    if (!root) return;
    for (int i = 0; i < BASE; ++i) clear(root->next[i]);
    free(root);
    return ;
}

Node *insert(Trie root, const int *CN, int n) {
    if (!root) root = get_new_node();
    Node *p = root;
    for (int i = 0; i < n; ++i) {
        int ind = CN[i];
        if (!p->next[ind]) p->next[ind] = get_new_node();
        p = p->next[ind];
    }
    p->flag = 1;
    return root;
}

int search_cnt = 0;
int search(Trie root, const int *CN, int n) {
    Node *p = root;
    int i = 0;
    while (p && i < n) {
        search_cnt += 1;
        int ind = CN[i++];
        p = p->next[ind];
    }
    return (p && p->flag);
}
```

```
int *str_to_num(const unsigned char *str) {
    int n = strlen((char *)str);
    int *num = (int *)calloc(sizeof(int), n * 8);
    int j = 0;
    for (int i = 0; str[i]; ++i) {
        int n = str[i] , ind = 8;
        while (ind--) {
            if (n) {
                num[j++] = n % 2;
                n /= 2;
            }
            else num[j++] = 0;
        }
    }
    return num;
}

int main() {
    Trie root = NULL;
    unsigned char buffer[10000];
    int word_cnt = 0;
    int search_word_cnt = 0;
    FILE *fpp = NULL;

    fpp = fopen("/Users/zou-jianfeng/HZ/ZY/1.txt", "r"); //读取10M中文文本
    while (~fscanf(fpp, "%s", buffer)) {
        int n = strlen((char *)buffer);
        word_cnt += n;
        int *CN = str_to_num(buffer);
        root = insert(root, CN, n * 8);
    }
    fclose(fpp);

    FILE *fp = NULL;
    fp = fopen("/Users/zou-jianfeng/HZ/ZY/1.txt", "r");
    while (~fscanf(fp, "%s", buffer)) {
        int n = strlen((char *)buffer);
        search_word_cnt += n;
        int *CN = str_to_num(buffer);
        if (search(root, CN, n * 8)) {
            printf("1");
        } else {
            printf("0");
            search_cnt += 1;
        }
    }
    fclose(fp);
    printf("\n");

    printf("word_cnt = %d node_cnt = %d search_word_cnt = %d search_cnt =
 %d\n", word_cnt, node_cnt, search_word_cnt, search_cnt);
```

```
    printf("searchrate : %lf\n", 1.0 * search_word_cnt / (1.0 * search_cn
t));
    printf("rate : %lf\n", 1.0 * word_cnt / (1.0 * node_cnt * sizeof(Nod
e)));
    clear(root);
    return 0;
}
```

8.1哈夫曼二叉字典树

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#define HBASE 256
#define TBASE 2
#define MAX_LEN 100000
#define swap(a, b) { \
    __typeof(a) temp; \
    temp = a; \
    a = b; \
    b = temp; \
}

typedef struct Node {
    int flag;
    struct Node *next[TBASE];
} Node, *Trie;

typedef struct HFNode {
    int data, freq;
    struct HFNode *lchild, *rchild;
} HFNode;

HFNode *get_Node() {
    HFNode *p = (HFNode *)malloc(sizeof(HFNode));
    p->freq = p->data = 0;
    p->lchild = p->rchild = NULL;
    return p;
}

void build(int n, HFNode *arr[n]) {
    for (int times = 0; times < n - 1; times++) {
        HFNode *minNode = arr[0];
        int ind = 0;
        for (int i = 1; i < n - times; i++) {
            if (arr[i]->freq >= minNode->freq) continue;
            minNode = arr[i];
            ind = i;
        }
```

```
            swap(arr[ind], arr[n - times - 1]);
            minNode = arr[0];
            ind = 0;

            for (int i = 1; i < n - times; i++) {
                if (arr[i]->freq >= minNode->freq) continue;
                minNode = arr[i];
                ind = i;
            }
            swap(arr[ind], arr[n - times - 2]);
            HFNode *new_node = get_Node();
            new_node->lchild = arr[n - times - 1];
            new_node->rchild = arr[n - times - 2];
            new_node->freq = arr[n - times - 1]->freq + arr[n - times - 2]->fr
eq;
            arr[n - times - 2] = new_node;
        }
        return ;
    }

    void get_cn(unsigned char *str, int *CN) {
        for(int i = 0; str[i]; ++i) {
            CN[str[i]]++;
        }
    }

    void extract(HFNode *root, char *buff, char (*huffman_code)[100], int n) {
        buff[n] = '\0';
        if (root->lchild == NULL && root->rchild == NULL) {
            strcpy(huffman_code[root->data], buff);
            //printf("%d %d : %s\n",root->data, root->freq, buff);
            return ;
        }
        buff[n] = '0';
        extract(root->lchild, buff, huffman_code, n + 1);
        buff[n] = '1';
        extract(root->rchild, buff, huffman_code, n + 1);
        return ;
    }

    int node_cnt = 0;
    Node *get_new_node() {
        Node *p = (Node *)calloc(sizeof(Node), 1);
        p->flag = 0;
        node_cnt += 1;
        return p;
    }

    void clear(Trie root) {
        if (!root) return;
        for (int i = 0; i < TBASE; ++i) clear(root->next[i]);
        free(root);
```

```
    return ;
}

Node *insert(Trie root, const  char *str) {
    if (!root) root = get_new_node();
    Node *p = root;
    for (int i = 0; str[i]; ++i) {
        int ind = str[i] - '0';
        if (!p->next[ind]) p->next[ind] = get_new_node();
        p = p->next[ind];
    }
    p->flag = 1;
    return root;
}

int search_cnt = 0;
int search(Trie root, const char *str) {
    Node *p = root;
    int i = 0;
    while (p && str[i]) {
        search_cnt++;
        int ind = str[i++] - '0';
        p = p->next[ind];
    }
    return (p && p->flag);
}

int main() {
    HFNode *arr[HBASE] = {0};
    FILE *fp = NULL;
    unsigned char buffer[MAX_LEN];
    int CN[HBASE] = {0};
    fp = fopen("/Users/zou-jianfeng/HZ/ZY/3.txt", "r"); //读取60M中文文本
    while (~fscanf(fp, "%s", buffer)) {
        //printf("%s\n", buffer);
        get_cn(buffer, CN); //获取字节相应数出现频率存入CN数组中
    }
    fclose(fp);

    int n = 0;
    for (int i = 0; i < HBASE; ++i) {
        if(!CN[i]) continue;
        HFNode *new_node = get_Node();
        new_node->data = i;
        new_node->freq = CN[i];
        arr[n++] = new_node;
    }
    build(n, arr); //构建哈夫曼树
    char buff[100];
    char huffman_code[256][100] = {0};
    extract(arr[0], buff, huffman_code, 0); //将哈夫曼树节点和其路径存入huffma
n--code中
```

```
    Trie root = NULL;
    unsigned char str[MAX_LEN];
    int word_cnt = 0;
    int search_word_cnt = 0;
    FILE *fpp = NULL;
    fpp = fopen("/Users/zou-jianfeng/HZ/ZY/1.txt", "r"); //读取60M中文文本
    while (~fscanf(fpp, "%s", buffer)) {
        char TCN[MAX_LEN] = {0};
        for (int i = 0; buffer[i]; ++i) {
            word_cnt += 1;
            strcat(TCN, huffman_code[buffer[i]]); //获取输入字符串的哈夫曼路径
        }
        root = insert(root, TCN); //将路径插入二叉字典树中
    }
    fclose(fpp);
    FILE *fppp = NULL;
    fppp = fopen("/Users/zou-jianfeng/HZ/ZY/1.txt", "r"); //读取60M中文文本
    while (~fscanf(fppp, "%s", buffer)) {
        char TCN[MAX_LEN] = {0};
        for (int i = 0; buffer[i]; ++i) {
            search_word_cnt += 1;
            strcat(TCN, huffman_code[buffer[i]]); //获取输入字符串的哈夫曼路径
        }
        if (search(root, TCN)) {
            printf("1");
        } else {
            printf("0");
            search_cnt += 1;
        }
    }
    fclose(fppp);
    clear(root);
    printf("\n");
    printf("word_cnt = %d node_cnt = %d search_word_cnt = %d search_cnt =
 %d\n", word_cnt, node_cnt, search_word_cnt, search_cnt);
    printf("searchrate : %lf\n", 1.0 * search_word_cnt / (1.0 * search_cn
t));
    printf("rate : %lf\n", 1.0 * word_cnt / (1.0 * node_cnt * sizeof(Nod
e)));
    return 0;
}
```