北京郵電大學

实验报告

实验三： 树-哈夫曼编/译码器

班级: 2022211320

学号: 2022212408

姓名: 胡宇杭

学院: 计算机学院 (国家示范性软件学院)

时间: 2023 年 11 月 29 日

目录

1 实验目的

1. 熟悉哈夫曼树的构造实现及应用；
2. 培养根据实际问题合理选择数据结构的能力；
3. 学习自己查找相关资料以解决实际问题的能力。

2 实验环境

1. **操作系统：** *MacOS Ventura*
   - 版本：version 13.4.1
   - 架构：ARM
2. **C 环境：**
   - 编译器：Clang
   - IDE：Xcode
3. **编译器：** *Clang*
   - 版本：version 14.0.3
   - 编译命令 (开启 $O2$ 优化)：***clang -O2 -o output_file source_file.c***
   - 编译命令 (关闭 $O2$ 优化)：***clang -O0 -o output_file source_file.c***

3 实验内容

3.1 问题描述:

哈夫曼编码是一种基于最优二叉树的无损编码方案。需要根据字符集和频度的实际统计构建哈夫曼树，然后进行编码和译码。

3.2 要求:

- **初始化**：从终端读入字符集大小 **n**，以及 **n** 个字符和 **n** 个权值，建立哈夫曼树，并将它存于文件 **hfmTree** 中；
- **编码**：利用已经建好的哈夫曼树，对文件 **ToBeTran** 中的报文进行编码，然后将结果存入文件 **CodeFile** 中（为简化处理，可以在 **CodeFile** 中用一个字节来存储码字中的一个 **0/1** 比特位）；
- **译码**：利用已建好的哈夫曼树，对 **CodeFile** 中的代码进行译码，结果存入 **TextFile** 中。

3.3 附加要求:

对一个 **512*512** 的 **lena.bmp** 灰度图片进行哈夫曼编码。**BMP** 文件由: **BMP** 文件头 + 像素数据组成，灰度图 **1** 个像素占用 **1** 个字节。**lena.bmp** 文件大小是 **263222** 字节，包括 **1078** 字节的头部 + **512*512** 个像素值。**lena.bmp** 文件见实验作业附件。

4 实验步骤

操作步骤 + 运行截图

4.1 代码解释

由于这次实验涉及到利用哈夫曼树对不同种文件的处理以及利用其他数据结构对哈夫曼编码进行优化, 因此仍采用分文件编写的方式.

4.1.1 Huffman

该文件实现了利用哈夫曼树对文本文件进行编码、解码的功能。

```h
#ifndef huffman_hpp
#define huffman_hpp

#define MAX_CHAR_VAL 128

typedef struct
{
    int parent, lChild, rChild;
    int weight;
    char ch;
} HNode;

typedef struct
{
    HNode* data;
    int num;
} HuffmanTree;

typedef struct
{
    char* cd;
    char ch;
} CNode;

typedef struct
{
    CNode* data;
    int size;
} HCode;

int statistic(int** w, char** ch, const char* file_name);

HuffmanTree* buildHT(int w[], char ch[], int n);

void destroyHT(HuffmanTree* ht);

void saveHT(HuffmanTree* ht, const char* file_name);

HuffmanTree* loadHT(const char* file_name);

HCode* huffmanCoding(HuffmanTree* ht);

void enCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name);

void deCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name);

#endif /* huffman_hpp */
```

接下来我们对每个函数进行解释 (虽然这么说但功能太简单的就直接跳过了)

- **int statistic(int** w, char** ch, const char* file_name)**

  该函数的作用是统计指定文本文件中各个字符出现的数量，并存放在 **w** 和 **ch** 数组中，用于在接下来的步骤中建立哈夫曼树。

  由于每一步操作都是线性的，故总时间复杂度为 **O(n)**。

```
7  int statistic(int** w, char** ch, const char* file_name)
8  {
9      FILE* file           = fopen(file_name, "r");
10     int tw[MAX_CHAR_VAL] = {0};
11
12     if (file == NULL)
13     {
14         puts("Fail to open the file.");
15         return -1;
16     }
17
18     char buffer[1024];
19     while (fgets(buffer, sizeof(buffer), file))
20         for (int i = 0; buffer[i] != '\0'; i ++ )
21             tw[buffer[i]] ++ ;
22
23
24     int n = 0, idx = 0;
25     for (int i = 0; i < MAX_CHAR_VAL; i ++ )
26         if (tw[i]) n ++ ;
27
28     *w  = (int*)malloc((n + 1) * sizeof(int));
29     *ch = (char*)malloc((n + 1) * sizeof(char));
30
31     for (int i = 0; i < MAX_CHAR_VAL; i ++ )
32         if (tw[i])
33         {
34             (*w)[ ++ idx] = tw[i];
35             (*ch)[idx]    = char(i);
36         }
37
38     fclose(file);
39
40     return idx;
41 }
```

- **HuffmanTree* buildHT(int w[], char ch[], int n)**

  该函数的作用是使用 **statistic** 函数中得到的数据为文本文件建立哈夫曼树。

  哈夫曼树的建立过程为每次选取权重最小的两个根结点合并，在不优化的条件下，其每次迭代都需要遍历一遍所有结点，一共遍历 **n-1** 次，故总的时间复杂度为 $O(n^2)$。该步骤在处理结点数较多的情况时会使性能大幅下降。因此这里采用维护**优先队列**的方式对其进行优化。

  我们首先将 **n** 个叶子结点入队，优先队列每次入队、出队的时间复杂度均为

$O(logn)$，故该步操作的时间复杂度为 **O(nlogn)**；在合并的过程中，我们每次迭代需要让队头出队两次，并将新的结点入队，一共迭代 **n-1** 次，时间复杂度为 $O(3(n-1)logn)$；

故该函数整体的时间复杂度为 $O(nlogn)$。

```
HuffmanTree* buildHT(int w[], char ch[], int n)
{
    HuffmanTree* temp = (HuffmanTree*)malloc(sizeof(HuffmanTree));
    int size          = (n << 1) - 1;
    temp->num         = n;
    temp->data        = (HNode*)malloc((size + 1) * sizeof(HNode));

    Priority_Queue* heap = initQueue(n);

    for (int i = 1; i <= n; i ++ )
    {
        temp->data[i] = {0, 0, 0, w[i], ch[i]};
        enQueue(heap, i, temp->data[i].weight);
    }

    for (int i = n + 1; i <= size; i ++ ) temp->data[i] = {0, 0, 0, 0, '\0'};

    for (int i = n + 1; i <= size; i ++ )
    {
        int s1 = deQueue(heap), s2 = deQueue(heap);

        temp->data[s1].parent = temp->data[s2].parent = i;
        temp->data[i].lChild  = s1;
        temp->data[i].rChild  = s2;
        temp->data[i].weight  = temp->data[s1].weight + temp->data[s2].weight;

        enQueue(heap, i, temp->data[i].weight);
    }

    destroyQueue(heap);

    return temp;
}
```

- **HCode* huffmanCoding(HuffmanTree* ht)**

  该函数用于通过已建立的哈夫曼树对每个字符进行哈夫曼编码。

  不难发现，外层循环一共进行了 **n** 次，内层循环在最坏的情况下需要进行 **n** 次（因为哈夫曼树的最大高度为 **n**），故总的时间复杂度为 $O(n^2)$。

```
HCode* huffmanCoding(HuffmanTree* ht)
{
    HCode* temp = (HCode*)malloc(sizeof(HCode));
    int n       = temp->size = ht->num;
    temp->data  = (CNode*)malloc((n + 1) * sizeof(CNode));
    char* cd    = (char*)malloc(n * sizeof(char));
    cd[n - 1]   = '\0';

    for (int i = 1; i <= n; i ++ )
    {
        int start        = n - 1;
        temp->data[i].ch = ht->data[i].ch;

        for (int u = i, f = ht->data[i].parent; f; u = f, f = ht->data[f].parent)
        {
            if (ht->data[f].lChild == u) cd[--start] = '0';
            else cd[--start] = '1';
        }

        temp->data[i].cd = (char*)malloc((n - start) * sizeof(char));
        strcpy(temp->data[i].cd, &cd[start]);
    }

    free(cd);

    return temp;
}
```

- **void enCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name)**

  该函数的作用是通过之前得到的字符编码为整个文本文件进行编码并存储在指定文件中。

  由于我们需要遍历整个文本文件，因此外层循环的时间复杂度为 $O(n)$，而在处理每一个字符时，我们需要通过遍历来找到指定的字符编码，因此总的时间复杂度就会来到 $O(nm)$，这显然不是我们希望看到的。

  回顾之前我们 **statistic** 函数统计的过程，可以发现，最终得到的 **ch** 数组的字符是升序排列的，这意味着我们可以使用二分搜索来优化。此时总的时间复杂度为 $O(nlogm)$。

```
152 void enCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name)
153 {
154     HCode* hc          = huffmanCoding(ht);
155     FILE* input_file  = fopen(input_file_name, "r");
156     FILE* output_file = fopen(output_file_name, "w");
157 
158     if (input_file == NULL || output_file == NULL)
159     {
160         puts("Fail to open the file.");
161         return;
162     }
163 
164     char buffer[1024];
165     while (fgets(buffer, sizeof(buffer), input_file))
166     {
167         for (int i = 0; buffer[i] != '\0'; i ++ )
168         {
169             int l = 1, r = hc->size;
170             while (l < r)
171             {
172                 int mid = (l + r) >> 1;
173                 if (hc->data[mid].ch >= buffer[i]) r = mid;
174                 else l = mid + 1;
175             }
176 
177             fputs(hc->data[l].cd, output_file);
178         }
179     }
180 
181     destroyHC(hc);
182 
183     fclose(input_file);
184     fclose(output_file);
185 }
```

- **void deCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name)**

  该函数的作用是通过之前得到的哈夫曼树实现对编码后文件的解码工作。

  和之前类似，外层循环一共进行了 **n** 次，内层循环在最坏的情况下需要进行 **m** 次（因为哈夫曼树的最大高度为 **m**），因此总的时间复杂度为 $O(nm)$。

```
187 void deCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name)
188 {
189     FILE* input_file  = fopen(input_file_name, "r");
190     FILE* output_file = fopen(output_file_name, "w");
191     int root          = (ht->num << 1) - 1;
192     int curr          = root;
193 
194     if (input_file == NULL || output_file == NULL)
195     {
196         puts("Fail to open the file.");
197         return;
198     }
199 
200     char ch;
201     while ((ch = fgetc(input_file)) != EOF)
202     {
203         curr = (ch == '0') ? ht->data[curr].lChild : ht->data[curr].rChild;
204 
205         if (!ht->data[curr].lChild && !ht->data[curr].rChild)
206         {
207             fputc(ht->data[curr].ch, output_file);
208             curr = root;
209         }
210     }
211 
212     fclose(input_file);
213     fclose(output_file);
214 }
```

4.1.2 Queue

该文件实现了优先队列，并提供了基础的出入队功能。

此处仿照 **STL**，使用 **小根堆** 实现优先队列。

```
#ifndef queue_hpp
#define queue_hpp

typedef struct
{
    int index;
    int weight;
} QNode;

typedef struct
{
    QNode* base;
    int size;
    int capacity;
} Priority_Queue;

Priority_Queue* initQueue(int capacity);

void destroyQueue(Priority_Queue* q);

void heapSwap(Priority_Queue* q, int a, int b);

void adjustDown(int u);

void adjustUp(int u);

void enQueue(Priority_Queue* q, int index, int weight);

int deQueue(Priority_Queue* q);

#endif /* queue_hpp */
```

- **Priority_Queue* initQueue(int capacity)**

  该函数用于初始化优先队列。由于我们是使用该优先队列优化取最小结点的操作，因此我们只需要开辟 $\mathbf{n+1}$ 结点大小的内存空间（因为优先队列下标从 **1** 开始）。

```
Priority_Queue* initQueue(int capacity)
{
    Priority_Queue* temp = (Priority_Queue*)malloc(sizeof(Priority_Queue));

    if (temp == NULL) return NULL;

    temp->base     = (QNode*)malloc((capacity + 1) * sizeof(QNode));
    temp->size     = 0;
    temp->capacity = capacity + 1;

    return temp;
}
```

- **adjustDown(Priority_Queue* q, int u)**

  该函数实现的功能是将某一结点 **下沉**，直到满足小根堆的性质。

  由小根堆的性质，某一结点的权重必须大于它的两个孩子结点，故我们需要在某一结点的权重不完全大于两个孩子时将其向小于它的权重的孩子结点 **下沉**；当两个孩子结点的权重均小于当前结点时，需要向较小的孩子结点 **下沉**，保证在执行完 **heapSwap** 操作后二叉堆的性质不会被破坏。

  由于每次是从两个结点中选择一个，数据空间每次减少一半，故时间复杂度为 $O(logn)$。

```
32 void adjustDown(Priority_Queue* q, int u)
33 {
34     int temp = u, lc = u << 1, rc = lc | 1;
35     if (lc <= q->size && q->base[lc].weight < q->base[temp].weight) temp = lc;
36     if (rc <= q->size && q->base[rc].weight < q->base[temp].weight) temp = rc;
37     if (u != temp)
38     {
39         heapSwap(q, u, temp);
40         adjustDown(q, temp);
41     }
42 }
```

- **adjustUp(Priority_Queue* q, int u)**

  该函数实现的功能是将某一结点 **上浮**，直到满足小根堆的性质。

  和之前类似，我们需要维护小根堆的性质，因此需要在当前结点的权重小于其父节点的权重时将其 **上浮**。由于小根堆的性质保证 $child_1 < parent < child_2$（假设我们需要移动的结点为 $child_1$），故在交换之后仍能保证 $parent < child_2$，故不需要像 **下沉**函数那样进行判断。

  由于二叉堆是一棵完全二叉树，因此最大上浮次数为 **层数 - 1**，故时间复杂度也为 $O(logn)$

```
44 void adjustUp(Priority_Queue* q, int u)
45 {
46     while (u >> 1 && q->base[u].weight < q->base[u >> 1].weight)
47     {
48         heapSwap(q, u, u >> 1);
49         u >>= 1;
50     }
51 }
```

- **void enQueue(Priority_Queue* q, int idx, int weight)**

  该函数用于实现将某一元素入队的操作。

  adjustUp 函数的时间复杂度为 $O(logn)$，其他操作均为 $O(1)$，故总的时间复杂度为 **O(logn)**。

```
53 void enQueue(Priority_Queue* q, int idx, int weight)
54 {
55     if (q == NULL || idx <= 0 || weight < 0) return;
56 
57     q->base[ ++ q->size] = {idx, weight};
58     adjustUp(q, q->size);
59 }
```

- **int deQueue(Priority_Queue* q)**

  该函数用于实现将队头元素出队的操作，同时由于我们只需要获得权重最小的结点的索引，故返回值为 **int**，而不是 **QNode**。

  adjustDown 函数的时间复杂度为 $O(logn)$，其他操作均为 $O(1)$，故总的时间复杂度为 **O(logn)**。

```
61 int deQueue(Priority_Queue* q)
62 {
63     if (q == NULL) return -1;
64 
65     int idx = q->base[1].index;
66     heapSwap(q, 1, q->size);
67     q->size -- ;
68     adjustDown(q, 1);
69 
70     return idx;
71 }
```

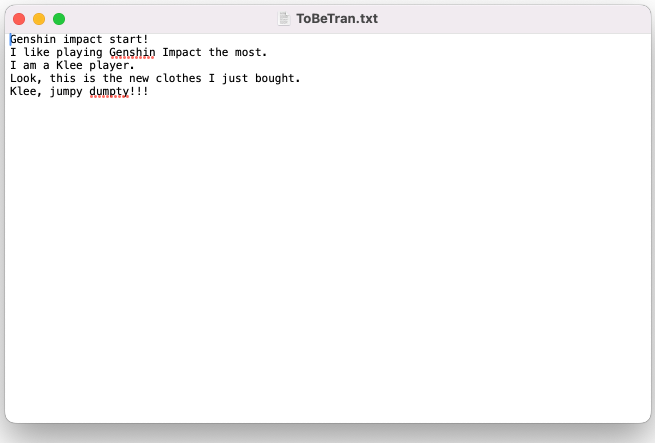
4.1.3 Bmp

该文件实现了对 **bmp** 文件编码的支持，由于实际上只是把原来的 **char** 类型改成了 **unsigned char**，其他功能的实现基本一致，故此处不再赘述。

4.2 运行截图

1. **操作 0**：建立哈夫曼树。

   为了测试编写的哈夫曼树是否能正确工作，我们首先先在 **“ToBeTran.txt”** 中输入想要编码的文本：



   然后我们编写 **testHT** 函数。

```
52  void testHT()
53  {
54      int* w;
55      char* ch;
56      int n = statistic(&w, &ch, "ToBeTran.txt");
57
58      HuffmanTree* ht = buildHT(w, ch, n);
59      HCode* hc = huffmanCoding(ht);
60
61      if (w) free(w);
62      if (ch) free(ch);
63
64      for (int i = 1; i <= n; i ++ ) printf("%c: %s\n", hc->data[i].ch, hc->data[i].cd);
65
66      destroyHT(ht);
67  }
```

   调用 **testHT** 函数，查看输出，经过缜密的分析（手算），发现构建出的哈夫曼树是正确的。

```
: 10011
 : 101
!: 01001
,: 011110
.: 100011
G: 011111
I: 10000
K: 010000
L: 1000100
a: 0001
b: 1000101
c: 00000
d: 0100010
e: 1110
g: 010111
h: 0010
i: 0011
j: 011100
k: 011101
l: 11011
m: 11000
n: 11001
o: 10010
p: 11010
r: 010110
s: 0110
t: 1111
u: 01010
w: 0100011
y: 00001
Program ended with exit code: 0
```

2. **操作 1**：哈夫曼树的存储与读取功能。

   编写 **testSaveLoadFuction** 函数，并输出存储之前的哈夫曼树的结点信息与存储后读取的哈夫曼树的结点信息。

```
void testSaveLoadFuction()
{
    // readText("ToBeTran.txt");
    int* w;
    char* ch;
    int n = statistic(&w, &ch, "ToBeTran.txt");

    HuffmanTree* source = buildHT(w, ch, n);

    if (w) free(w);
    if (ch) free(ch);

    saveHT(source, "hfmTree.dat");

    HuffmanTree* copy = loadHT("hfmTree.dat");

    puts("The source data:");
    for (int i = 1; i <= 2 * n - 1; i ++ )
        if (i <= n) printf("char: %c parent: %d lchild: %d rchild: %d weight: %d\n", source->data[i].ch, source->data[i].parent, source->data[i].lChild, source->data[i].rChild, source->data[i].weight);
        else printf("EMPTY   parent: %d lchild: %d rchild: %d weight: %d\n", source->data[i].parent, source->data[i].lChild, source->data[i].rChild, source->data[i].weight);

    puts("");

    puts("The copy data:");
    for (int i = 1; i <= 2 * n - 1; i ++ )
        if (i <= n) printf("char: %c parent: %d lchild: %d rchild: %d weight: %d\n", copy->data[i].ch, copy->data[i].parent, copy->data[i].lChild, copy->data[i].rChild, copy->data[i].weight);
        else printf("EMPTY   parent: %d lchild: %d rchild: %d weight: %d\n", copy->data[i].parent, copy->data[i].lChild, copy->data[i].rChild, copy->data[i].weight);

    destroyHT(source);
    destroyHT(copy);
}
```

   两次输出完全一致，正确完成 **操作 1**。

```
The source data:
char:
 parent: 43 lchild: 0 rchild: 0 weight: 5
char:   parent: 55 lchild: 0 rchild: 0 weight: 22
char: ! parent: 39 lchild: 0 rchild: 0 weight: 4
char: , parent: 35 lchild: 0 rchild: 0 weight: 2
char: . parent: 37 lchild: 0 rchild: 0 weight: 3
char: G parent: 35 lchild: 0 rchild: 0 weight: 2
char: I parent: 42 lchild: 0 rchild: 0 weight: 4
char: K parent: 33 lchild: 0 rchild: 0 weight: 2
char: L parent: 31 lchild: 0 rchild: 0 weight: 1
char: a parent: 47 lchild: 0 rchild: 0 weight: 7
char: b parent: 31 lchild: 0 rchild: 0 weight: 1
char: c parent: 38 lchild: 0 rchild: 0 weight: 3
char: d parent: 32 lchild: 0 rchild: 0 weight: 1
char: e parent: 52 lchild: 0 rchild: 0 weight: 12
char: g parent: 36 lchild: 0 rchild: 0 weight: 2
char: h parent: 46 lchild: 0 rchild: 0 weight: 7
char: i parent: 46 lchild: 0 rchild: 0 weight: 7
char: j parent: 34 lchild: 0 rchild: 0 weight: 2
char: k parent: 34 lchild: 0 rchild: 0 weight: 2
char: l parent: 44 lchild: 0 rchild: 0 weight: 6
char: m parent: 45 lchild: 0 rchild: 0 weight: 6
char: n parent: 45 lchild: 0 rchild: 0 weight: 6
char: o parent: 43 lchild: 0 rchild: 0 weight: 5
char: p parent: 44 lchild: 0 rchild: 0 weight: 6
char: r parent: 36 lchild: 0 rchild: 0 weight: 2
char: s parent: 48 lchild: 0 rchild: 0 weight: 8
char: t parent: 52 lchild: 0 rchild: 0 weight: 12
char: u parent: 40 lchild: 0 rchild: 0 weight: 4
char: w parent: 32 lchild: 0 rchild: 0 weight: 1
char: y parent: 38 lchild: 0 rchild: 0 weight: 4
EMPTY   parent: 37 lchild: 9 rchild: 11 weight: 2
EMPTY   parent: 33 lchild: 13 rchild: 29 weight: 2
EMPTY   parent: 39 lchild: 8 rchild: 32 weight: 4
EMPTY   parent: 41 lchild: 18 rchild: 19 weight: 4
EMPTY   parent: 41 lchild: 4 rchild: 6 weight: 4
EMPTY   parent: 40 lchild: 25 rchild: 15 weight: 4
EMPTY   parent: 42 lchild: 31 rchild: 5 weight: 5
EMPTY   parent: 47 lchild: 12 rchild: 30 weight: 7
EMPTY   parent: 49 lchild: 33 rchild: 3 weight: 8
EMPTY   parent: 49 lchild: 28 rchild: 36 weight: 8
EMPTY   parent: 48 lchild: 34 rchild: 35 weight: 8
EMPTY   parent: 50 lchild: 7 rchild: 37 weight: 9
EMPTY   parent: 50 lchild: 23 rchild: 1 weight: 10
EMPTY   parent: 51 lchild: 24 rchild: 20 weight: 12
EMPTY   parent: 51 lchild: 21 rchild: 22 weight: 12
EMPTY   parent: 53 lchild: 16 rchild: 17 weight: 14
EMPTY   parent: 53 lchild: 38 rchild: 10 weight: 14
EMPTY   parent: 54 lchild: 26 rchild: 41 weight: 16
EMPTY   parent: 54 lchild: 39 rchild: 40 weight: 16
EMPTY   parent: 55 lchild: 42 rchild: 43 weight: 19
EMPTY   parent: 56 lchild: 45 rchild: 44 weight: 24
EMPTY   parent: 56 lchild: 14 rchild: 27 weight: 24
EMPTY   parent: 57 lchild: 47 rchild: 46 weight: 28
EMPTY   parent: 57 lchild: 49 rchild: 48 weight: 32
EMPTY   parent: 58 lchild: 50 rchild: 2 weight: 41
EMPTY   parent: 58 lchild: 51 rchild: 52 weight: 48
EMPTY   parent: 59 lchild: 53 rchild: 54 weight: 60
EMPTY   parent: 59 lchild: 55 rchild: 56 weight: 89
EMPTY   parent: 0 lchild: 57 rchild: 58 weight: 149
```

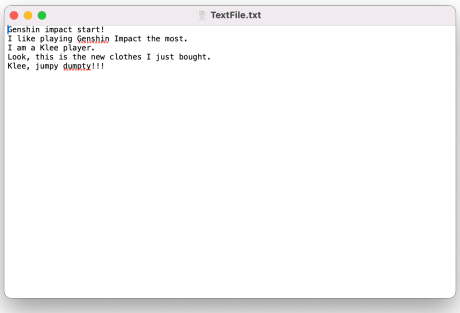
```
The copy data:
char:
 parent: 43 lchild: 0 rchild: 0 weight: 5
char:   parent: 55 lchild: 0 rchild: 0 weight: 22
char: ! parent: 39 lchild: 0 rchild: 0 weight: 4
char: , parent: 35 lchild: 0 rchild: 0 weight: 2
char: . parent: 37 lchild: 0 rchild: 0 weight: 3
char: G parent: 35 lchild: 0 rchild: 0 weight: 2
char: I parent: 42 lchild: 0 rchild: 0 weight: 4
char: K parent: 33 lchild: 0 rchild: 0 weight: 2
char: L parent: 31 lchild: 0 rchild: 0 weight: 1
char: a parent: 47 lchild: 0 rchild: 0 weight: 7
char: b parent: 31 lchild: 0 rchild: 0 weight: 1
char: c parent: 38 lchild: 0 rchild: 0 weight: 3
char: d parent: 32 lchild: 0 rchild: 0 weight: 1
char: e parent: 52 lchild: 0 rchild: 0 weight: 12
char: g parent: 36 lchild: 0 rchild: 0 weight: 2
char: h parent: 46 lchild: 0 rchild: 0 weight: 7
char: i parent: 46 lchild: 0 rchild: 0 weight: 7
char: j parent: 34 lchild: 0 rchild: 0 weight: 2
char: k parent: 34 lchild: 0 rchild: 0 weight: 2
char: l parent: 44 lchild: 0 rchild: 0 weight: 6
char: m parent: 45 lchild: 0 rchild: 0 weight: 6
char: n parent: 45 lchild: 0 rchild: 0 weight: 6
char: o parent: 43 lchild: 0 rchild: 0 weight: 5
char: p parent: 44 lchild: 0 rchild: 0 weight: 6
char: r parent: 36 lchild: 0 rchild: 0 weight: 2
char: s parent: 48 lchild: 0 rchild: 0 weight: 8
char: t parent: 52 lchild: 0 rchild: 0 weight: 12
char: u parent: 40 lchild: 0 rchild: 0 weight: 4
char: w parent: 32 lchild: 0 rchild: 0 weight: 1
char: y parent: 38 lchild: 0 rchild: 0 weight: 4
EMPTY   parent: 37 lchild: 9 rchild: 11 weight: 2
EMPTY   parent: 33 lchild: 13 rchild: 29 weight: 2
EMPTY   parent: 39 lchild: 8 rchild: 32 weight: 4
EMPTY   parent: 41 lchild: 18 rchild: 19 weight: 4
EMPTY   parent: 41 lchild: 4 rchild: 6 weight: 4
EMPTY   parent: 40 lchild: 25 rchild: 15 weight: 4
EMPTY   parent: 42 lchild: 31 rchild: 5 weight: 5
EMPTY   parent: 47 lchild: 12 rchild: 30 weight: 7
EMPTY   parent: 49 lchild: 33 rchild: 3 weight: 8
EMPTY   parent: 49 lchild: 28 rchild: 36 weight: 8
EMPTY   parent: 48 lchild: 34 rchild: 35 weight: 8
EMPTY   parent: 50 lchild: 7 rchild: 37 weight: 9
EMPTY   parent: 50 lchild: 23 rchild: 1 weight: 10
EMPTY   parent: 51 lchild: 24 rchild: 20 weight: 12
EMPTY   parent: 51 lchild: 21 rchild: 22 weight: 12
EMPTY   parent: 53 lchild: 16 rchild: 17 weight: 14
EMPTY   parent: 53 lchild: 38 rchild: 10 weight: 14
EMPTY   parent: 54 lchild: 26 rchild: 41 weight: 16
EMPTY   parent: 54 lchild: 39 rchild: 40 weight: 16
EMPTY   parent: 55 lchild: 42 rchild: 43 weight: 19
EMPTY   parent: 56 lchild: 45 rchild: 44 weight: 24
EMPTY   parent: 56 lchild: 14 rchild: 27 weight: 24
EMPTY   parent: 57 lchild: 47 rchild: 46 weight: 28
EMPTY   parent: 57 lchild: 49 rchild: 48 weight: 32
EMPTY   parent: 58 lchild: 50 rchild: 2 weight: 41
EMPTY   parent: 58 lchild: 51 rchild: 52 weight: 48
EMPTY   parent: 59 lchild: 53 rchild: 54 weight: 60
EMPTY   parent: 59 lchild: 55 rchild: 56 weight: 89
EMPTY   parent: 0 lchild: 57 rchild: 58 weight: 149
```

3. **操作 2**：利用哈夫曼树进行编码和解码。（两个功能放一块了）

   编写 **testCodeFunction** 函数，调用 **enCodeText** 和 **deCodeText** 函数实现编解码功能。

```
69  void testCodeFunction()
70  {
71      HuffmanTree* ht = loadHT("hfmTree.dat");
72
73      enCodeText(ht, "ToBeTran.txt", "CodeFile.txt");
74      deCodeText(ht, "CodeFile.txt", "TextFile.txt");
75
76      destroyHT(ht);
77  }
```

   查看 **“CodeFile.txt”** 和 **“TextFile.txt”** 文件，可以看出程序正确地完成了编解码的功能。



4. **操作 3**：利用哈夫曼树对 **bmp** 文件进行编码。

   编写 **testBmpCodeFunction** 函数，并调用 **bmpStatistic**、**buildBmpHT** 和 **bmpCoding** 函数实现建立哈夫曼树并求其哈夫曼编码。

```
void testBmpCodeFunction()
{
    int* w;
    unsigned char* pixel;
    int n = bmpStatistic(&w, &pixel, "lena.bmp");

    Bmptree* bmp = buildBmpHT(w, pixel, n);
    BCode* bc = bmpCoding(bmp);

    for (int i = 1; i <= n; i ++ ) printf("pixel %d: weight: %d code:%s\n", bc->data[i].pixel, w[i], bc->data[i].cd);

    printf("The sum of the pixel is %d\n", bmp->data[2 * n - 1].weight);

    if (w) free(w);
    if (pixel) free(pixel);

    destroyBmpHT(bmp);
    destroyBmpCode(bc);
}
```

   输出每个灰度值对应的权重和编码，可以发现，权重越低的灰度值对应的编码越长。（为了美观把后面两张放到了前面）

```
pixel 160: weight: 2062 code:1001000
pixel 161: weight: 1846 code:0101100
pixel 162: weight: 1769 code:0100000
pixel 163: weight: 1628 code:0010000
pixel 164: weight: 1454 code:0000000
pixel 165: weight: 1403 code:11111011
pixel 166: weight: 1370 code:11110111
pixel 167: weight: 1247 code:11100010
pixel 168: weight: 1201 code:11010001
pixel 169: weight: 1241 code:11011110
pixel 170: weight: 1139 code:10111000
pixel 171: weight: 1187 code:11000101
pixel 172: weight: 1350 code:11110001
pixel 173: weight: 1242 code:11011111
pixel 174: weight: 1225 code:11010111
pixel 175: weight: 1184 code:11000100
pixel 176: weight: 1190 code:11001000
pixel 177: weight: 1236 code:11011011
pixel 178: weight: 1108 code:10101000
pixel 179: weight: 950 code:01101101
pixel 180: weight: 919 code:01010001
pixel 181: weight: 820 code:00100110
pixel 182: weight: 790 code:00011010
pixel 183: weight: 681 code:111100101
pixel 184: weight: 626 code:111001000
pixel 185: weight: 619 code:110110101
pixel 186: weight: 641 code:111001001
pixel 187: weight: 691 code:111110001
pixel 188: weight: 647 code:111001110
pixel 189: weight: 664 code:111001111
pixel 190: weight: 726 code:00001000
pixel 191: weight: 815 code:00100101
pixel 192: weight: 825 code:00101011
pixel 193: weight: 912 code:01001111
pixel 194: weight: 926 code:01100000
pixel 195: weight: 1016 code:10000011
pixel 196: weight: 876 code:00111010
pixel 197: weight: 880 code:01000010
pixel 198: weight: 892 code:01001010
pixel 199: weight: 824 code:00101010
pixel 200: weight: 843 code:00101110
```

```
pixel 201: weight: 910 code:01001110
pixel 202: weight: 848 code:00110001
pixel 203: weight: 868 code:00110101
pixel 204: weight: 905 code:01001100
pixel 205: weight: 987 code:01111001
pixel 206: weight: 947 code:01101011
pixel 207: weight: 1067 code:10011011
pixel 208: weight: 1104 code:10100100
pixel 209: weight: 1032 code:10001101
pixel 210: weight: 936 code:01100011
pixel 211: weight: 1048 code:10011010
pixel 212: weight: 939 code:01100101
pixel 213: weight: 814 code:00100100
pixel 214: weight: 689 code:111110000
pixel 215: weight: 562 code:101110010
pixel 216: weight: 502 code:011111110
pixel 217: weight: 379 code:000010011
pixel 218: weight: 379 code:000010010
pixel 219: weight: 309 code:1101101001
pixel 220: weight: 218 code:0010111111
pixel 221: weight: 238 code:0110101010
pixel 222: weight: 206 code:0010111110
pixel 223: weight: 175 code:11110010010
pixel 224: weight: 119 code:01101010110
pixel 225: weight: 78 code:111100100001
pixel 226: weight: 69 code:111100100000
pixel 227: weight: 55 code:011010100101
pixel 228: weight: 55 code:011010100100
pixel 229: weight: 22 code:11110010001011
pixel 230: weight: 20 code:11110010001001
pixel 231: weight: 8 code:111100100010001
pixel 232: weight: 7 code:011010100110010
pixel 233: weight: 6 code:011010100110000
pixel 234: weight: 8 code:111100100010000
pixel 235: weight: 1 code:0110101001100111101
pixel 238: weight: 1 code:0110101001100111110
pixel 242: weight: 2 code:011010100110011101
pixel 244: weight: 1 code:0110101001100111111
pixel 245: weight: 1 code:011010100110011100
```

```
pixel 25: weight: 1 code:0110101001100111000
pixel 26: weight: 7 code:011010100110001
pixel 27: weight: 22 code:11110010001010
pixel 28: weight: 28 code:0110101001101
pixel 29: weight: 63 code:011010100111
pixel 30: weight: 93 code:111100100011
pixel 31: weight: 135 code:01101010111
pixel 32: weight: 177 code:11110010011
pixel 33: weight: 224 code:0110101000
pixel 34: weight: 306 code:1101101000
pixel 35: weight: 420 code:001011110
pixel 36: weight: 508 code:011111111
pixel 37: weight: 607 code:101110011
pixel 38: weight: 778 code:00010110
pixel 39: weight: 922 code:01010110
pixel 40: weight: 1082 code:10100001
pixel 41: weight: 1279 code:11100101
pixel 42: weight: 1429 code:11111110
pixel 43: weight: 1642 code:0010100
pixel 44: weight: 1756 code:0011110
pixel 45: weight: 1846 code:0101101
pixel 46: weight: 2038 code:1000010
pixel 47: weight: 2066 code:1001001
pixel 48: weight: 2007 code:1000000
pixel 49: weight: 2081 code:1001010
pixel 50: weight: 2053 code:1000101
pixel 51: weight: 2039 code:1000011
pixel 52: weight: 1969 code:0111101
pixel 53: weight: 1846 code:0101010
pixel 54: weight: 1741 code:0011100
pixel 55: weight: 1673 code:0010110
pixel 56: weight: 1549 code:0001001
pixel 57: weight: 1478 code:0000011
pixel 58: weight: 1289 code:11100110
pixel 59: weight: 1249 code:11100011
pixel 60: weight: 1193 code:11001001
pixel 61: weight: 1026 code:10001100
pixel 62: weight: 959 code:01110001
pixel 63: weight: 948 code:01101100
pixel 64: weight: 910 code:01001101
pixel 65: weight: 868 code:00110100
pixel 66: weight: 786 code:00010111
pixel 67: weight: 801 code:00011011
pixel 68: weight: 771 code:00010000
pixel 69: weight: 776 code:00010001
pixel 70: weight: 848 code:00110010
pixel 71: weight: 812 code:00011101
pixel 72: weight: 820 code:00100111
pixel 73: weight: 801 code:00011100
pixel 74: weight: 880 code:00111011
pixel 75: weight: 903 code:01001011
pixel 76: weight: 850 code:00110011
pixel 77: weight: 890 code:01000011
pixel 78: weight: 938 code:01100100
pixel 79: weight: 935 code:01100001
pixel 80: weight: 916 code:01010000
pixel 81: weight: 924 code:01010111
pixel 82: weight: 847 code:00110000
pixel 83: weight: 951 code:01110000
pixel 84: weight: 935 code:01100010
pixel 85: weight: 965 code:01111000
pixel 86: weight: 994 code:01111110
pixel 87: weight: 1015 code:10000010
pixel 88: weight: 1072 code:10100000
pixel 89: weight: 1108 code:10100101
pixel 90: weight: 1124 code:10101001
pixel 91: weight: 1224 code:11010110
pixel 92: weight: 1194 code:11010000
pixel 93: weight: 1346 code:11110000
pixel 94: weight: 1413 code:11111101
pixel 95: weight: 1557 code:0001010
pixel 96: weight: 1628 code:0010001
pixel 97: weight: 1791 code:0100100
pixel 98: weight: 1853 code:0101110
pixel 99: weight: 2105 code:1001100
pixel 100: weight: 1930 code:0111010
pixel 101: weight: 1899 code:0110111
pixel 102: weight: 1941 code:0111011
pixel 103: weight: 1840 code:0101001
pixel 104: weight: 1763 code:0011111
pixel 105: weight: 1737 code:0011011
pixel 106: weight: 1588 code:0001100
pixel 107: weight: 1498 code:0000101
pixel 108: weight: 1466 code:0000010
pixel 109: weight: 1336 code:11101010
pixel 110: weight: 1369 code:11110110
pixel 111: weight: 1389 code:11111010
pixel 112: weight: 1363 code:11110011
pixel 113: weight: 1442 code:11111111
pixel 114: weight: 1381 code:11111001
pixel 115: weight: 1336 code:11101011
pixel 116: weight: 1409 code:11111100
pixel 117: weight: 1464 code:0000001
pixel 118: weight: 1544 code:0000111
pixel 119: weight: 1517 code:0000110
pixel 120: weight: 1626 code:0001111
pixel 121: weight: 1789 code:0100010
pixel 122: weight: 1790 code:0100011
pixel 123: weight: 1878 code:0110011
pixel 124: weight: 2061 code:1000111
pixel 125: weight: 2097 code:1001011
pixel 126: weight: 2311 code:1011101
pixel 127: weight: 2300 code:1011011
pixel 128: weight: 2385 code:1100101
pixel 129: weight: 2483 code:1101110
pixel 130: weight: 2390 code:1100110
pixel 131: weight: 2262 code:1010110
pixel 132: weight: 2235 code:1010101
pixel 133: weight: 2116 code:1001110
pixel 134: weight: 1993 code:0111110
pixel 135: weight: 1858 code:0101111
pixel 136: weight: 1929 code:0111001
pixel 137: weight: 1889 code:0110100
pixel 138: weight: 2040 code:1000100
pixel 139: weight: 2128 code:1001111
pixel 140: weight: 2270 code:1011000
pixel 141: weight: 2270 code:1010111
pixel 142: weight: 2341 code:1100001
pixel 143: weight: 2433 code:1101010
pixel 144: weight: 2466 code:1101100
pixel 145: weight: 2409 code:1101001
pixel 146: weight: 2214 code:1010011
pixel 147: weight: 2340 code:1100000
pixel 148: weight: 2285 code:1011001
pixel 149: weight: 2202 code:1010001
pixel 150: weight: 2314 code:1011110
pixel 151: weight: 2327 code:1011111
pixel 152: weight: 2374 code:1100011
pixel 153: weight: 2611 code:1110100
pixel 154: weight: 2723 code:1111010
pixel 155: weight: 2690 code:1110111
pixel 156: weight: 2673 code:1110110
pixel 157: weight: 2493 code:1110000
pixel 158: weight: 2391 code:1100111
pixel 159: weight: 2286 code:1011010
pixel 160: weight: 2062 code:1001000
```

同时，总的权重之和为 $512^2 = 262144$，侧面说明了我们的编码过程是正确的。

```
The sum of the pixel is 262144
Program ended with exit code: 0
```

5 实验分析和总结

5.1 数据结构分析

本次实验主要是通过哈夫曼树实现对文件的编解码，其中，我使用了优先队列、二分查找等数据结构和算法对其性能进行了优化，整体时间复杂度相较于未优化版本下降了一个量级，在处理大规模问题时的效率会更高。

缺点：这份代码针对大规模数据的性能虽然差强人意，但对于小规模的数据或数据分布较为平均的情况，与其使用二分查找等算法，不如牺牲一些空间来换取较快的访问速度（类似于桶排，通过下标直接对应相应的数据）。

5.2 算法复杂度分析

在代码解释时说明了各函数的时间复杂度，故此处只对其进行汇总

- **操作一**：根据之前的分析，时间复杂度为 $O(nlogn)$
- **操作二**：根据之前的分析，时间复杂度为 $O(nlogm)$
- **操作三**：根据之前的分析，时间复杂度为 $O(n^2)$
- **操作四**：根据之前的分析，时间复杂度为 $O(nlogn)$

5.3 总结

通过这次实验，我掌握了哈夫曼树、优先队列的原理和具体实现方法，积累了对树型数据结构 **Debug** 的经验；更重要的是，了解了课上学的数据结构在现实中的应用，理论与实践相结合，让我受益匪浅。

6 程序源代码

主函数

```
#include <stdio.h>
#include <stdlib.h>
#include "queue.hpp"
#include "huffman.hpp"
#include "bmp.hpp"

void testSaveLoadFuction()
{
    int* w;
    char* ch;
    int n = statistic(&w, &ch, "ToBeTran.txt");

    HuffmanTree* source = buildHT(w, ch, n);

    if (w) free(w);
    if (ch) free(ch);

    saveHT(source, "hfmTree.dat");

    HuffmanTree* copy = loadHT("hfmTree.dat");

    puts("The source data:");
    for (int i = 1; i <= 2 * n - 1; i ++ )
        if (i <= n) printf("char: %c parent: %d lchild: %d rchild: %d weight: %d\n", source->data[i
            ].ch, source->data[i].parent, source->data[i].lChild, source->data[i].rChild, source->
            data[i].weight);
        else printf("EMPTY   parent: %d lchild: %d rchild: %d weight: %d\n", source->data[i].parent,
             source->data[i].lChild, source->data[i].rChild, source->data[i].weight);

    puts("");

    puts("The copy data:");
    for (int i = 1; i <= 2 * n - 1; i ++ )
        if (i <= n) printf("char: %c parent: %d lchild: %d rchild: %d weight: %d\n", copy->data[i].
            ch, copy->data[i].parent, copy->data[i].lChild, copy->data[i].rChild, copy->data[i].
            weight);
```

```
32          else printf("EMPTY   parent: %d lchild: %d rchild: %d weight: %d\n", copy->data[i].parent,
                copy->data[i].lChild, copy->data[i].rChild, copy->data[i].weight);
33
34      destroyHT(source);
35      destroyHT(copy);
36  }
37
38  void testHT()
39  {
40      int* w;
41      char* ch;
42      int n = statistic(&w, &ch, "ToBeTran.txt");
43
44      HuffmanTree* ht = buildHT(w, ch, n);
45      HCode* hc = huffmanCoding(ht);
46
47      if (w) free(w);
48      if (ch) free(ch);
49
50      for (int i = 1; i <= n; i ++ ) printf("%c: %s\n", hc->data[i].ch, hc->data[i].cd);
51
52      destroyHT(ht);
53  }
54
55  void testCodeFunction()
56  {
57      HuffmanTree* ht = loadHT("hfmTree.dat");
58
59      enCodeText(ht, "ToBeTran.txt", "CodeFile.txt");
60      deCodeText(ht, "CodeFile.txt", "TextFile.txt");
61
62      destroyHT(ht);
63  }
64
65  void testBmpCodeFunction()
66  {
67      int* w;
68      unsigned char* pixel;
69      int n = bmpStatistic(&w, &pixel, "lena.bmp");
70
71      Bmptree* bmp = buildBmpHT(w, pixel, n);
```

```
72      BCode* bc = bmpCoding(bmp);
73
74      for (int i = 1; i <= n; i ++ ) printf("pixel %d: weight: %d code:%s\n", bc->data[i].pixel, w[i],
            bc->data[i].cd);
75
76      printf("The sum of the pixel is %d\n", bmp->data[2 * n - 1].weight);
77
78      if (w) free(w);
79      if (pixel) free(pixel);
80
81      destroyBmpHT(bmp);
82      destroyBmpCode(bc);
83  }
84
85  int main()
86  {
87      // testHT();
88
89      // testSaveLoadFuction();
90
91      // testCodeFunction();
92
93       testBmpCodeFunction();
94
95      return 0;
96  }
```

huffman.hpp

```
1   #ifndef huffman_hpp
2   #define huffman_hpp
3
4   #define MAX_CHAR_VAL 128
5
6   typedef struct
7   {
8       int parent, lChild, rChild;
9       int weight;
10      char ch;
11  } HNode;
```

```
12
13  typedef struct
14  {
15      HNode* data;
16      int num;
17  } HuffmanTree;
18
19  typedef struct
20  {
21      char* cd;
22      char ch;
23  } CNode;
24
25  typedef struct
26  {
27      CNode* data;
28      int size;
29  } HCode;
30
31  int statistic(int** w, char** ch, const char* file_name);
32
33  HuffmanTree* buildHT(int w[], char ch[], int n);
34
35  void destroyHT(HuffmanTree* ht);
36
37  void saveHT(HuffmanTree* ht, const char* file_name);
38
39  HuffmanTree* loadHT(const char* file_name);
40
41  HCode* huffmanCoding(HuffmanTree* ht);
42
43  void enCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name);
44
45  void deCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name);
46
47  void destroyHC(HCode* hc);
48
49  #endif /* huffman_hpp */
```

huffman.cpp

```cpp
#include "huffman.hpp"
#include "queue.hpp"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int statistic(int** w, char** ch, const char* file_name)
{
    FILE* file           = fopen(file_name, "r");
    int tw[MAX_CHAR_VAL] = {0};

    if (file == NULL)
    {
        puts("Fail to open the file.");
        return -1;
    }

    char buffer[1024];
    while (fgets(buffer, sizeof(buffer), file))
        for (int i = 0; buffer[i] != '\0'; i ++ )
            tw[buffer[i]] ++ ;


    int n = 0, idx = 0;
    for (int i = 0; i < MAX_CHAR_VAL; i ++ )
        if (tw[i]) n ++ ;

    *w  = (int*)malloc((n + 1) * sizeof(int));
    *ch = (char*)malloc((n + 1) * sizeof(char));

    for (int i = 0; i < MAX_CHAR_VAL; i ++ )
        if (tw[i])
        {
            (*w)[ ++ idx] = tw[i];
            (*ch)[idx]    = char(i);
        }

    fclose(file);

    return idx;
```

```
41  }
42
43  HuffmanTree* buildHT(int w[], char ch[], int n)
44  {
45      HuffmanTree* temp = (HuffmanTree*)malloc(sizeof(HuffmanTree));
46      int size          = (n << 1) - 1;
47      temp->num         = n;
48      temp->data        = (HNode*)malloc((size + 1) * sizeof(HNode));
49
50      Priority_Queue* heap = initQueue(n);
51
52      for (int i = 1; i <= n; i ++ )
53      {
54          temp->data[i] = {0, 0, 0, w[i], ch[i]};
55          enQueue(heap, i, temp->data[i].weight);
56      }
57
58      for (int i = n + 1; i <= size; i ++ ) temp->data[i] = {0, 0, 0, 0, '\0'};
59
60      for (int i = n + 1; i <= size; i ++ )
61      {
62          int s1 = deQueue(heap), s2 = deQueue(heap);
63
64          temp->data[s1].parent = temp->data[s2].parent = i;
65          temp->data[i].lChild  = s1;
66          temp->data[i].rChild  = s2;
67          temp->data[i].weight  = temp->data[s1].weight + temp->data[s2].weight;
68
69          enQueue(heap, i, temp->data[i].weight);
70      }
71
72      destroyQueue(heap);
73
74      return temp;
75  }
76
77  void destroyHT(HuffmanTree* ht)
78  {
79      if (ht == NULL) return;
80
81      if (ht->data) free(ht->data);
```

```
82      free(ht);
83  }
84
85  void saveHT(HuffmanTree* ht, const char* file_name)
86  {
87      if (ht == NULL) return;
88
89      FILE* file = fopen(file_name, "wb");
90
91      if (file == NULL)
92      {
93          puts("Fail to open the file.");
94          return;
95      }
96
97      fwrite(&ht->num, sizeof(int), 1, file);
98      fwrite(ht->data, sizeof(HNode), (ht->num << 1), file);
99
100     fclose(file);
101 }
102
103 HuffmanTree* loadHT(const char* file_name)
104 {
105     HuffmanTree* temp = (HuffmanTree*)malloc(sizeof(HuffmanTree));
106     FILE* file        = fopen(file_name, "rb");
107
108     if (file == NULL || temp == NULL)
109     {
110         puts("Fail to load the file.");
111         return NULL;
112     }
113
114     fread(&temp->num, sizeof(int), 1, file);
115
116     temp->data = (HNode*)malloc(temp->num << 1 * sizeof(HNode));
117     fread(temp->data, sizeof(HNode), temp->num << 1, file);
118
119     fclose(file);
120
121     return temp;
122 }
```

```
123
124 HCode* huffmanCoding(HuffmanTree* ht)
125 {
126     HCode* temp = (HCode*)malloc(sizeof(HCode));
127     int n       = temp->size = ht->num;
128     temp->data  = (CNode*)malloc((n + 1) * sizeof(CNode));
129     char* cd    = (char*)malloc(n * sizeof(char));
130     cd[n - 1]   = '\0';
131
132     for (int i = 1; i <= n; i ++ )
133     {
134         int start        = n - 1;
135         temp->data[i].ch = ht->data[i].ch;
136
137         for (int u = i, f = ht->data[i].parent; f; u = f, f = ht->data[f].parent)
138         {
139             if (ht->data[f].lChild == u) cd[--start] = '0';
140             else cd[--start] = '1';
141         }
142
143         temp->data[i].cd = (char*)malloc((n - start) * sizeof(char));
144         strcpy(temp->data[i].cd, &cd[start]);
145     }
146
147     free(cd);
148
149     return temp;
150 }
151
152 void enCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name)
153 {
154     HCode* hc          = huffmanCoding(ht);
155     FILE* input_file   = fopen(input_file_name, "r");
156     FILE* output_file = fopen(output_file_name, "w");
157
158     if (input_file == NULL || output_file == NULL)
159     {
160         puts("Fail to open the file.");
161         return;
162     }
163
```

```
164     char buffer[1024];
165     while (fgets(buffer, sizeof(buffer), input_file))
166     {
167         for (int i = 0; buffer[i] != '\0'; i ++ )
168         {
169             int l = 1, r = hc->size;
170             while (l < r)
171             {
172                 int mid = (l + r) >> 1;
173                 if (hc->data[mid].ch >= buffer[i]) r = mid;
174                 else l = mid + 1;
175             }
176 
177             fputs(hc->data[l].cd, output_file);
178         }
179     }
180 
181     destroyHC(hc);
182 
183     fclose(input_file);
184     fclose(output_file);
185 }
186 
187 void deCodeText(HuffmanTree* ht, const char* input_file_name, const char* output_file_name)
188 {
189     FILE* input_file  = fopen(input_file_name, "r");
190     FILE* output_file = fopen(output_file_name, "w");
191     int root          = (ht->num << 1) - 1;
192     int curr          = root;
193 
194     if (input_file == NULL || output_file == NULL)
195     {
196         puts("Fail to open the file.");
197         return;
198     }
199 
200     char ch;
201     while ((ch = fgetc(input_file)) != EOF)
202     {
203         curr = (ch == '0') ? ht->data[curr].lChild : ht->data[curr].rChild;
204 
```

```
205         if (!ht->data[curr].lChild && !ht->data[curr].rChild)
206         {
207             fputc(ht->data[curr].ch, output_file);
208             curr = root;
209         }
210     }
211 
212     fclose(input_file);
213     fclose(output_file);
214 }
215 
216 void destroyHC(HCode* hc)
217 {
218     if (hc == NULL) return;
219 
220     if (hc->data) free(hc->data);
221     free(hc);
222 }
```

queue.hpp

```
1  #ifndef queue_hpp
2  #define queue_hpp
3  
4  typedef struct
5  {
6      int index;
7      int weight;
8  } QNode;
9  
10 typedef struct
11 {
12     QNode* base;
13     int size;
14     int capacity;
15 } Priority_Queue;
16 
17 Priority_Queue* initQueue(int capacity);
18 
19 void destroyQueue(Priority_Queue* q);
```

```
20
21 void heapSwap(Priority_Queue* q, int a, int b);
22
23 void adjustDown(int u);
24
25 void adjustUp(int u);
26
27 void enQueue(Priority_Queue* q, int index, int weight);
28
29 int deQueue(Priority_Queue* q);
30
31 #endif /* queue_hpp */
```

queue.cpp

```
1  #include "queue.hpp"
2  #include <stdlib.h>
3
4  Priority_Queue* initQueue(int capacity)
5  {
6      Priority_Queue* temp = (Priority_Queue*)malloc(sizeof(Priority_Queue));
7
8      if (temp == NULL) return NULL;
9
10     temp->base     = (QNode*)malloc((capacity + 1) * sizeof(QNode));
11     temp->size     = 0;
12     temp->capacity = capacity + 1;
13
14     return temp;
15 }
16
17 void destroyQueue(Priority_Queue* q)
18 {
19     if (q == NULL) return;
20
21     if (q->base) free(q->base);
22     free(q);
23 }
24
25 void heapSwap(Priority_Queue* q, int a, int b)
```

```
{
    QNode temp = q->base[a];
    q->base[a] = q->base[b];
    q->base[b] = temp;
}

void adjustDown(Priority_Queue* q, int u)
{
    int temp = u, lc = u << 1, rc = lc | 1;
    if (lc <= q->size && q->base[lc].weight < q->base[temp].weight) temp = lc;
    if (rc <= q->size && q->base[rc].weight < q->base[temp].weight) temp = rc;
    if (u != temp)
    {
        heapSwap(q, u, temp);
        adjustDown(q, temp);
    }
}

void adjustUp(Priority_Queue* q, int u)
{
    while (u >> 1 && q->base[u].weight < q->base[u >> 1].weight)
    {
        heapSwap(q, u, u >> 1);
        u >>= 1;
    }
}

void enQueue(Priority_Queue* q, int idx, int weight)
{
    if (q == NULL || idx <= 0 || weight < 0) return;

    q->base[ ++ q->size] = {idx, weight};
    adjustUp(q, q->size);
}

int deQueue(Priority_Queue* q)
{
    if (q == NULL) return -1;

    int idx = q->base[1].index;
    heapSwap(q, 1, q->size);
```

```
67      q->size -- ;
68      adjustDown(q, 1);
69
70      return idx;
71  }
```

bmp.hpp

```
1   #ifndef bmp_hpp
2   #define bmp_hpp
3
4   #define MAX_PIXEL_VAL 256
5   #define BMP_HEADER_SIZE 1078
6
7   typedef struct
8   {
9       int parent, lChild, rChild;
10      int weight;
11      unsigned char pixel;
12  } BmpNode;
13
14  typedef struct
15  {
16      BmpNode* data;
17      int num;
18  } Bmptree;
19
20  typedef struct
21  {
22      char* cd;
23      unsigned char pixel;
24  } BNode;
25
26  typedef struct
27  {
28      BNode* data;
29      int size;
30  } BCode;
31
32  int bmpStatistic(int** w, unsigned char** ch, const char* file_name);
```

```
33
34 Bmptree* buildBmpHT(int w[], unsigned char ch[], int n);
35
36 void destroyBmpHT(Bmptree* bmp);
37
38 BCode* bmpCoding(Bmptree* bmp);
39
40 void destroyBmpCode(BCode* bc);
41
42 #endif /* bmp_hpp */
```

bmp.cpp

```
1  #include "bmp.hpp"
2  #include "queue.hpp"
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  int bmpStatistic(int** w, unsigned char** pixel, const char* file_name)
8  {
9      FILE* file            = fopen(file_name, "r");
10     int tw[MAX_PIXEL_VAL] = {0};
11
12     if (file == NULL)
13     {
14         puts("Fail to open the file.");
15         return -1;
16     }
17
18     for (int i = 0; i < BMP_HEADER_SIZE; i ++ )
19         if (getc(file) == EOF) return -1;
20
21     unsigned char pi;
22     while (fread(&pi, sizeof(unsigned char), 1, file)) tw[pi] ++ ;
23
24     int n = 0, idx = 0;
25     for (int i = 0; i < MAX_PIXEL_VAL; i ++ )
26         if (tw[i]) n ++ ;
27
```

```
28      *w  = (int*)malloc((n + 1) * sizeof(int));
29      *pixel = (unsigned char*)malloc((n + 1) * sizeof(char));
30
31      for (int i = 0; i < MAX_PIXEL_VAL; i ++ )
32          if (tw[i])
33          {
34              (*w)[ ++ idx] = tw[i];
35              (*pixel)[idx] = i;
36          }
37
38      fclose(file);
39
40      return idx;
41  }
42
43  Bmptree* buildBmpHT(int w[], unsigned char pixel[], int n)
44  {
45      Bmptree* temp = (Bmptree*)malloc(sizeof(Bmptree));
46      int size      = (n << 1) - 1;
47      temp->num     = n;
48      temp->data    = (BmpNode*)malloc((size + 1) * sizeof(BmpNode));
49
50      Priority_Queue* heap = initQueue(n);
51
52      for (int i = 1; i <= n; i ++ )
53      {
54          temp->data[i] = {0, 0, 0, w[i], pixel[i]};
55          enQueue(heap, i, temp->data[i].weight);
56      }
57
58      for (int i = n + 1; i <= size; i ++ ) temp->data[i] = {0, 0, 0, 0, '\0'};
59
60      for (int i = n + 1; i <= size; i ++ )
61      {
62          int s1 = deQueue(heap), s2 = deQueue(heap);
63
64          temp->data[s1].parent = temp->data[s2].parent = i;
65          temp->data[i].lChild  = s1;
66          temp->data[i].rChild  = s2;
67          temp->data[i].weight  = temp->data[s1].weight + temp->data[s2].weight;
68
```

```
69          enQueue(heap, i, temp->data[i].weight);
70      }
71
72      destroyQueue(heap);
73
74      return temp;
75  }
76
77  void destroyBmpHT(Bmptree* bmp)
78  {
79      if (bmp == NULL) return;
80
81      if (bmp->data) free(bmp->data);
82      free(bmp);
83  }
84
85  BCode* bmpCoding(Bmptree* bmp)
86  {
87      BCode* temp = (BCode*)malloc(sizeof(BCode));
88      int n       = temp->size = bmp->num;
89      temp->data  = (BNode*)malloc((n + 1) * sizeof(BNode));
90      char* cd    = (char*)malloc(n * sizeof(char));
91      cd[n - 1]   = '\0';
92
93      for (int i = 1; i <= n; i ++ )
94      {
95          int start           = n - 1;
96          temp->data[i].pixel = bmp->data[i].pixel;
97
98          for (int u = i, f = bmp->data[i].parent; f; u = f, f = bmp->data[f].parent)
99          {
100             if (bmp->data[f].lChild == u) cd[ -- start] = '0';
101             else cd[ -- start] = '1';
102         }
103
104         temp->data[i].cd = (char*)malloc((n - start) * sizeof(char));
105         strcpy(temp->data[i].cd, &cd[start]);
106     }
107
108     free(cd);
109
```

```
110     return temp;
111 }
112
113 void destroyBmpCode(BCode* bc)
114 {
115     if (bc == NULL) return;
116
117     if (bc->data) free(bc->data);
118     free(bc);
119 }
```