专业：计算机科学与技术班级：23052322 姓名：张逸轩学号：23051214

1. 实验目的

实现一个哈夫曼编码、译码器，具体理解哈夫曼树和哈夫曼编码的作用和使用。

2. 实验过程(实验方案、流程、程序等)(参考书上的格式需要写详细)

建立最小堆：

```c
#include "minHeap.h"

static void swapNodes(HfmNode** a, HfmNode** b) {
    HfmNode* t = *a;
    *a = *b;
    *b = t;
}

static void minHeapify(MinHeap* minHeap, size_t idx) {
    size_t smallest = idx;
    size_t left = 2 * idx + 1;
    size_t right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq) {
        smallest = left;
    }

    if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq) {
        smallest = right;
    }

    if (smallest != idx) {
        swapNodes(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

MinHeap* createMinHeap(size_t capacity) {
    MinHeap* minHeap = (MinHeap*)malloc(sizeof(MinHeap));
    minHeap->array = (HfmNode**)malloc(sizeof(HfmNode*) * capacity);
    minHeap->size = 0;
    minHeap->capacity = capacity;
    return minHeap;
}

void freeMinHeap(MinHeap* minHeap) {
    if (!minHeap) return;
    free(minHeap->array);
    free(minHeap);
}

void insertMinHeap(MinHeap* minHeap, HfmNode* node) {
    if (minHeap->size == minHeap->capacity) {
        fprintf(stderr, "MinHeap is full!\n");
        return;
    }

    size_t i = minHeap->size++;
    minHeap->array[i] = node;

    // 向上调整，保证父节点频率小于子节点
    while (i != 0 && minHeap->array[(i - 1) / 2]->freq > minHeap->array[i]->freq) {
        swapNodes(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

HfmNode* extractMin(MinHeap* minHeap) {
    if (minHeap->size == 0) return NULL;

    HfmNode* root = minHeap->array[0];

    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    minHeap->size--;
    minHeapify(minHeap, 0);

    return root;
}


```

有了最小堆用于哈夫曼算法，建立哈夫曼树：

```c
static HfmNode* createNode(unsigned char symbol, size_t freq) {
    HfmNode* node = (HfmNode*)malloc(sizeof(HfmNode));
    node->symbol = symbol;
    node->freq = freq;
    node->left = NULL;
    node->right = NULL;
    return node;
}

HfmNode* buildHuffmanTreeImpl(const unsigned char *symbols, const size_t *freq, size_t size) {

    MinHeap* minHeap = createMinHeap(size);

    for (size_t i = 0; i < size; i++) {
        if (freq[i] > 0) {
            insertMinHeap(minHeap, createNode(symbols[i], freq[i]));
        }
    }

    while (minHeap->size > 1) {

        HfmNode* left = extractMin(minHeap);
        HfmNode* right = extractMin(minHeap);

        HfmNode* parent = createNode('\0', left->freq + right->freq);
        parent->left = left;
        parent->right = right;

        insertMinHeap(minHeap, parent);
    }

    HfmNode* root = NULL;
    if (minHeap->size == 1) {
        root = extractMin(minHeap);
    }

    freeMinHeap(minHeap);
    return root;
}

void freeHuffmanTree(HfmNode* root) {
    if (!root) return;
    freeHuffmanTree(root->left);
    freeHuffmanTree(root->right);
    free(root);
}

void printHuffmanTreeImpl(HfmNode *root, int depth) {
    if (!root) return;
    for (int i = 0; i < depth; i++) {
        printf("    ");
    }
    if (root->symbol)
        printf("%c \n", root->symbol);
    else
        printf("* \n");

    printHuffmanTreeImpl(root->left, depth + 1);
    printHuffmanTreeImpl(root->right, depth + 1);
}
```

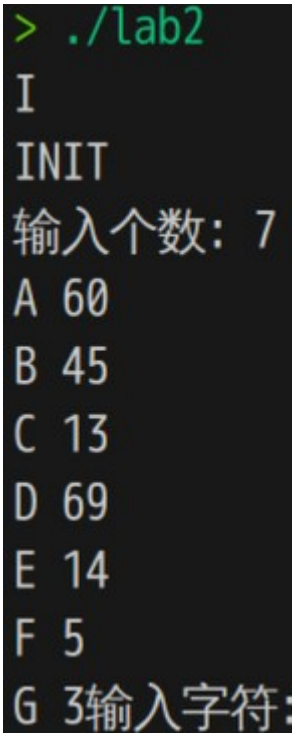
实现 huffmantree 的保存和读取（这里使用二进制编码）

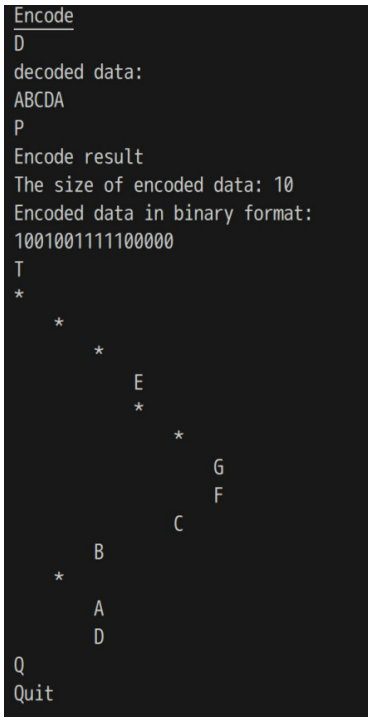
```c
int saveHuffmanTreeToFileImpl(HfmNode *root, FILE *fp) {
    if (!fp) return -1;
    if (!root) return 0;

    if (root->left == NULL && root->right == NULL) {

        fputc(1, fp);
        fputc((unsigned char)root->symbol, fp);
    } else {

        fputc(0, fp);
        saveHuffmanTreeToFileImpl(root->left, fp);
        saveHuffmanTreeToFileImpl(root->right, fp);
    }
    return 0;
}

HfmNode* loadHuffmanTreeFromFileImpl(FILE *fp) {
    if (!fp) return NULL;
    int flag = fgetc(fp);
    if (flag == EOF) return NULL;

    if (flag == 1) {

        int symbol = fgetc(fp);
        if (symbol == EOF) return NULL;
        return createNode((unsigned char)symbol, 0);
    } else if (flag == 0) {

        HfmNode* left = loadHuffmanTreeFromFileImpl(fp);
        HfmNode* right = loadHuffmanTreeFromFileImpl(fp);
        HfmNode* parent = createNode('\0', 0);
        parent->left = left;
        parent->right = right;
        return parent;
    } else {
        return NULL;
    }
}

```

实现解码、编码

```c
#define MAX_SYMBOLS 256

typedef struct {
    unsigned char *code;
    size_t length;
} CodeTableEntry;

static void buildCodeTable(HfmNode *root, CodeTableEntry *table, unsigned char *buffer, size_t depth) {
    if (!root) return;

    if (!root->left && !root->right) {

        unsigned char *code = (unsigned char*)malloc(depth+1);
        for (size_t i = 0; i < depth; i++) {
            code[i] = buffer[i];
        }
        code[depth] = '\0';
        table[root->symbol].code = code;
        table[root->symbol].length = depth;
        return;
    }

    buffer[depth] = '0';
    buildCodeTable(root->left, table, buffer, depth+1);

    buffer[depth] = '1';
    buildCodeTable(root->right, table, buffer, depth+1);
}

static void freeCodeTable(CodeTableEntry *table) {
    for (int i = 0; i < MAX_SYMBOLS; i++) {
        if (table[i].code) {
            free(table[i].code);
        }
    }
}


unsigned char* encodeDataImpl(HfmNode *root, const unsigned char *data, size_t dataSize, size_t *outSize) {
    if (!root || !data || dataSize == 0) {
        *outSize = 0;
        return NULL;
    }

    CodeTableEntry table[MAX_SYMBOLS] = {0};
    unsigned char buffer[1024];
    buildCodeTable(root, table, buffer, 0);

    size_t totalBits = 0;
    for (size_t i = 0; i < dataSize; i++) {
        if (table[data[i]].length == 0) {
            fprintf(stderr, "Symbol '%c' not found in the Huffman tree.\n", data[i]);
            freeCodeTable(table);
            return NULL;
        }
        totalBits += table[data[i]].length;
    }

    size_t byteCount = (totalBits + 7) / 8;
    size_t headerSize = sizeof(size_t);
    unsigned char *encoded = (unsigned char*)calloc(byteCount + headerSize, 1);
    if (!encoded) {
        fprintf(stderr, "Failed to allocate memory for encoded data.\n");
        freeCodeTable(table);
        return NULL;
    }

    size_t originalSize = dataSize;
    for (size_t i = 0; i < headerSize; i++) {
        encoded[i] = (originalSize >> (8 * (headerSize - 1 - i))) & 0xFF;
    }

    size_t bitPos = 0;
    for (size_t i = 0; i < dataSize; i++) {
        unsigned char *code = table[data[i]].code;
        size_t length = table[data[i]].length;
        for (size_t j = 0; j < length; j++) {
            size_t totalBitPos = bitPos;
            size_t byteIndex = headerSize + (totalBitPos / 8);
            size_t bitIndex = totalBitPos % 8;
            if (code[j] == '1') {
                encoded[byteIndex] |= (1 << (7 - bitIndex));
            }
            bitPos++;
        }
    }

    *outSize = byteCount + headerSize;
    freeCodeTable(table);
    return encoded;
}


unsigned char* decodeDataImpl(HfmNode *root, const unsigned char *encodedData, size_t encodedSize, size_t *outSize) {
    if (!root || !encodedData || encodedSize <= sizeof(size_t)) {
        *outSize = 0;
        return NULL;
    }

    size_t headerSize = sizeof(size_t);
    size_t originalDataSize = 0;
    for (size_t i = 0; i < headerSize; i++) {
        originalDataSize = (originalDataSize << 8) | encodedData[i];
    }

    if (originalDataSize == 0) {
        *outSize = 0;
        return NULL;
    }

    size_t totalBits = (encodedSize - headerSize) * 8;

    unsigned char *decoded = (unsigned char*)malloc(originalDataSize);
    if (!decoded) {
        fprintf(stderr, "Failed to allocate memory for decoded data.\n");
        return NULL;
    }
    size_t decIndex = 0;

    HfmNode *current = root;
    for (size_t bitPos = 0; bitPos < totalBits && decIndex < originalDataSize; bitPos++) {
        size_t byteIndex = headerSize + (bitPos / 8);
        size_t bitIndex = bitPos % 8;
        int bit = (encodedData[byteIndex] & (1 << (7 - bitIndex))) ? 1 : 0;

        current = bit == 0 ? current->left : current->right;

        if (!current->left && !current->right) {
            decoded[decIndex++] = current->symbol;
            current = root;

            if (decIndex == originalDataSize) {
                break;
            }
        }
    }

    *outSize = decIndex;
    return decoded;
}
```

3. 实验结果及结果分析





4. 实验总结

成功实现了 哈夫曼编/译码器，理解了哈夫曼算法，对编码和文件有了更加深入的理解。