中国科学院大学
University of Chinese Academy of Sciences

数据结构与算法分析实验报告

哈夫曼树及应用

姓名	徐昕妍
学号	2023K8009970008
班级	2311 班

2025 年 5 月 11 日

目录

1 实验内容

1.1 主要实验内容

1. 完全二叉树的顺序存储或链式存储
2. 构造哈夫曼树并生成哈夫曼编码实验

1.2 实验目的

1. 学习完全二叉树的顺序存储和链式存储，实现对完全二叉树的创建和插入
2. 通过实际数据构造哈夫曼树，深入理解贪心算法在最优二叉树构建中的应用。
3. 生成哈夫曼编码，掌握前缀编码的原理及其在数据压缩中的关键作用。

2 实验设计

2.1 完全二叉树的顺序存储和链式存储

2.1.1 完全二叉树设计

完全二叉树是每一层的节点都尽可能地填满，只有最底层的节点可以不满，且这些节点都集中在左侧。它有两个常见的存储方式：顺序存储和链式存储。

顺序存储

顺序存储完全二叉树的核心思想是使用一个一维数组来存储树的节点。

在顺序存储中，假设树的根节点位于数组的第一个位置 (索引为 0)。对于数组中的任意一个节点：

- 左子节点的索引为 2i + 1
- 右子节点的索引为 2i + 2
- 父节点的索引为 (i - 1) / 2

链式存储

链式存储的设计基于指针结构，每个节点不仅存储数据，还包含指向左右子节点的指针。这种方法适用于树形结构中节点较多且结构比较复杂的情况，尤其适合动态结构 (例如树的节点插入和删除)。链式存储的主要特点是它使用队列来实现树的层次遍历。

在链式存储中，树的每个节点都有左右指针指向其子节点。根节点通常指向整个树的起点，然后依次连接其他节点。

2.1.2 函数功能描述

1. 顺序存储

(1) 顺序存储结构

```c
    typedef struct {
    int data[MAX_SIZE];  // 用数组存储节点数据
    int size;            // 当前树中节点数量
} CompleteBinaryTree;
```

(2) 函数功能

表 1: 函数功能描述表

函数名称	功能描述	输入参数	返回值
initTree	初始化完全二叉树，设置树的大小为 0	CompleteBinaryTree* tree	无返回值
insertNode	向完全二叉树中插入一个节点，若树满则返回错误	CompleteBinaryTree* tree, int value	无返回值
levelOrderTraversal	层次遍历顺序存储的完全二叉树并打印每个节点的数据	CompleteBinaryTree* tree	无返回值

2. 链式存储

(1) 链式存储结构及辅助队列

```c
    typedef struct TreeNode {
    int data;                 // 节点数据
    struct TreeNode* left;   // 左孩子指针
    struct TreeNode* right;  // 右孩子指针
} TreeNode;

typedef struct {
    TreeNode** nodes;    // 存储树节点指针的数组
    int front, rear;     // 队首和队尾指针
    int capacity;        // 队列容量
} Queue;
```

(2) 函数功能

表 2: 链式存储函数功能描述表

函数名称	功能描述	输入参数	返回值
isEmpty	检查队列是否为空	Queue* queue	1 (空) 或 0 (非空)
initQueue	初始化队列	Queue* queue, int capacity	无返回值
enqueue	向队列中添加一个节点	Queue* queue, TreeNode* node	无返回值
dequeue	从队列中取出一个节点，并返回	Queue* queue	TreeNode* (出队的节点)
creatNode	创建一个新的树节点并初始化节点数据及左右子节点指针	int value	TreeNode* (新节点)
Insert	按层次顺序插入新节点到二叉树中	TreeNode** root, int value, Queue* queue	无返回值
levelOrderTraversalWithQueue	使用队列进行层次遍历，并打印二叉树的节点数据	TreeNode* root	无返回值

2.1.3 代码及流程图

1. 顺序存储

(1) 节点插入

```
    void insertNode(CompleteBinaryTree* tree, int value) {
    if (tree->size >= MAX_SIZE) {
        printf("Error: Tree is full!\n");
        return;
    }
    tree->data[tree->size] = value;
    tree->size++;
 }
```

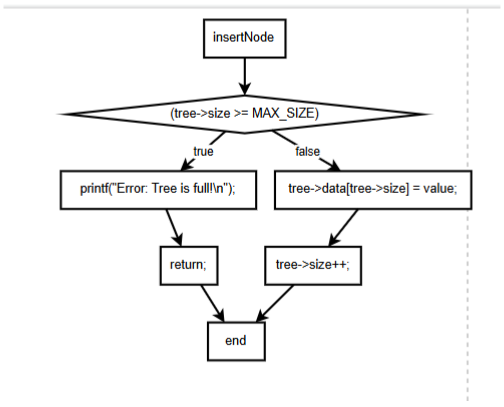

图 1: InsertNode 流程图

(2) 遍历二叉树

```
    void levelOrderTraversal(CompleteBinaryTree*tree){
    if ((tree->size==0))
    {
        printf("Empty\n");
        return;
    }
    printf("Orde:\n");
    for (int i = 0; i < tree->size; i++)
    {
        printf("%d ",tree->data[i]);
    }
    printf("\n");
}
```

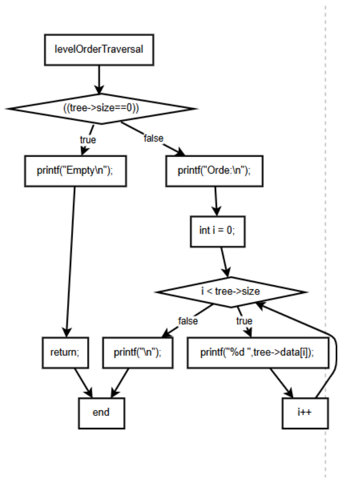

图 2: levelOrderTraversal 流程图

2. 链式存储

(1) 节点插入

```c
void Insert(TreeNode** root, int value, Queue* queue) {
// 创建一个新的树节点，并初始化它的值
TreeNode* newnode = creatNode(value);

// 如果树为空（根节点为 NULL），直接将新节点设置为根节点
if (*root == NULL) {
    *root = newnode;
    return; // 插入完成，返回
}

// 将根节点加入队列，以便开始层次遍历插入
enqueue(queue, *root);

// 层次遍历，直到找到第一个空位（子节点为空）
while (!isEmpty(queue)) {
    // 从队列中取出当前节点
    TreeNode* current = dequeue(queue);

    // 如果当前节点的左子节点为空，将新节点插入左子节点
    if (current->left == NULL) {
        current->left = newnode;
        break; // 插入完成，退出循环
    }

    // 如果当前节点的右子节点为空，将新节点插入右子节点
```

```
        if (current->right == NULL) {
            current->right = newnode;
            break;  // 插入完成，退出循环
        }

        // 如果当前节点的左子节点和右子节点都已满，继续将右子节点加入队列
        enqueue(queue, current->left);
        enqueue(queue, current->right);
    }
}
```

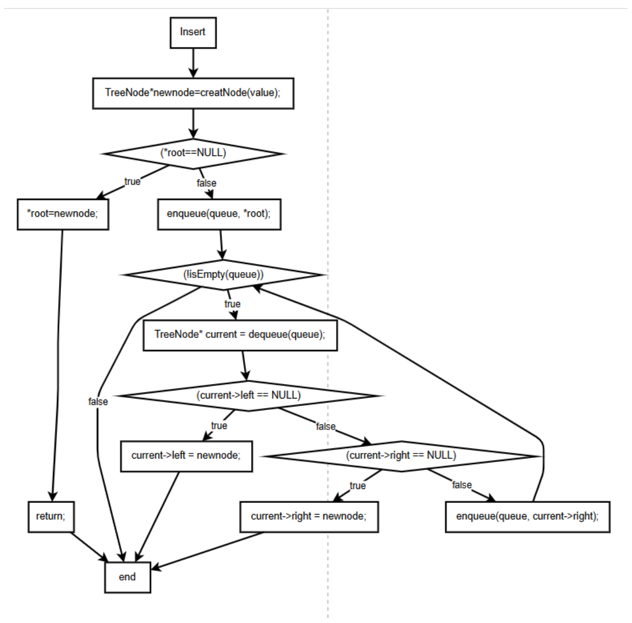

图 3: Insert 流程图

(2) 遍历二叉树

```
    void levelOrderTraversalWithQueue(TreeNode* root) {
    if (root == NULL) {
        printf("Empty\n");
        return;
    }
    Queue queue;
    initQueue(&queue, MAX_SIZE);
    enqueue(&queue, root);
    printf("Order:\n");

    while (!isEmpty(&queue)) {
        TreeNode* current = dequeue(&queue);
        printf("%d ", current->data);
        if (current->left != NULL) {
            enqueue(&queue, current->left);
        }
        if (current->right != NULL) {
            enqueue(&queue, current->right);
        }
    }

    printf("\n");
}
```

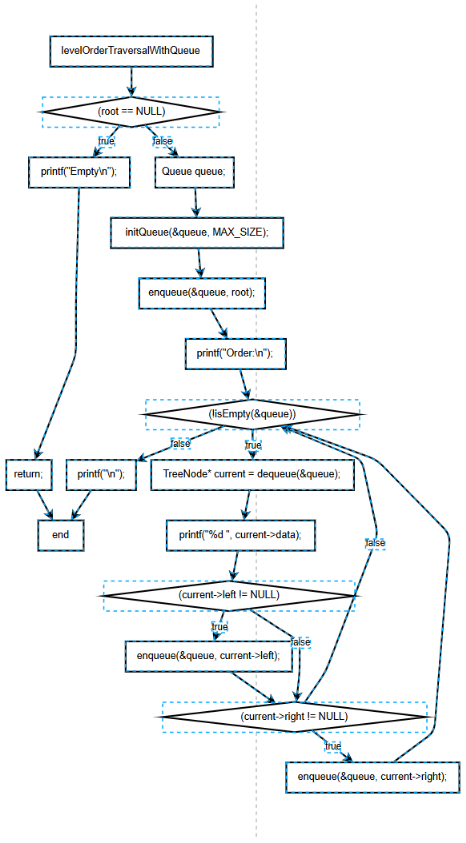

图 4: levelOrderTraversalWithQueue

2.1.4 性能对比

1. 顺序存储

优点

- 空间紧凑：只需存储数据，无需额外指针，空间利用率高。

- 随机访问高效
- 缓存友好

缺点

- 静态大小，需预先分配固定大小数组，可能浪费空间（非完全二叉树）或溢出。
- 插入/删除成本高: 时间复杂度 O（n）

2. 链式存储

优点

- 动态扩展：节点动态分配内存，无需预定义大小
- 插入/删除灵活：修改指针即可，时间复杂度：O(1)
- 结构通用：可轻松扩展为其他树结构

缺点

- 空间开销大
- 缓存不友好

2.2 构造哈夫曼树并生成哈夫曼编码实验

2.2.1 哈夫曼树的设计

给定 N 个权值作为 N 个叶子结点，构造一棵二叉树，若该树的带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树 (Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。

哈夫曼树的实现利用了贪心算法的理念。给定若干权值，选择其中两个权值最小的结点生成新的结点，而后删除被选择的结点，让生成的结点继续参与选拔，直至无结点进行选拔。

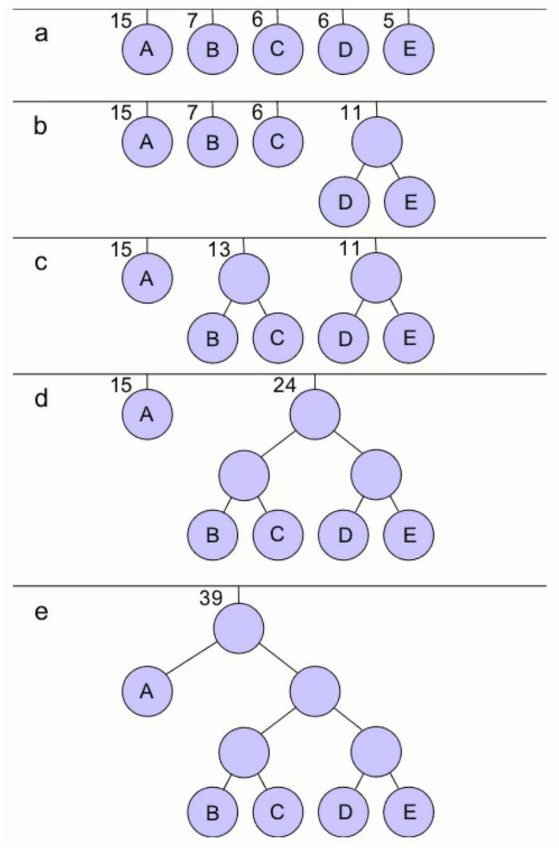

图 5: 哈夫曼树构造示意图

2.2.2 哈弗曼编码的设计

哈夫曼编码是一种基于字符出现频率的最优解编码方法。它通过构建一颗哈夫曼树，将频率较高的字符分配较短的编码，频率较低的字符分配较长的编码，从而实现数据压缩。[1]

哈夫曼编码的构建过程大致如下：

首先统计编码数据中各个字符出现的频率，将每个字符及其频率作为一个节点，构建一个优先队列，节点频率从小到大排序；根据频率构建哈夫曼树，从根节点开始，为左子节点赋值 0，右子节点赋值 1，遍历整棵树，直到达到叶子节点；叶子节点的路径就是对应字符的哈夫曼树。

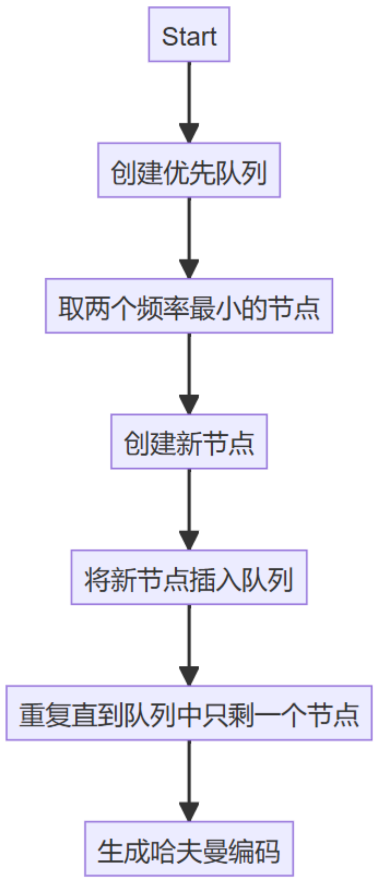

图 6: 哈夫曼编码构建过程

2.2.3 函数功能描述

(1) 哈夫曼树结构

```
    // 定义最小堆节点结构体
struct MinHeapNode {
    char data;              // 节点存储的字符数据（比如哈夫曼编码中
        的字符）
    unsigned freq;          // 节点的频率，表示该字符在文本中出现的
        次数
    struct MinHeapNode *left, *right;  // 左右子节点指针，用于构建二
        叉树结构
};

// 定义最小堆结构体
struct MinHeap {
    unsigned size;          // 当前堆中的节点数
    unsigned capacity;      // 堆的最大容量（即最多可以存储多少节
```

```c
    点)
    struct MinHeapNode** array;  // 指向最小堆节点数组的指针，存储堆
        中的节点
};
```

(2) 函数功能

表 3: 哈夫曼树实现函数功能描述表

函数名称	功能描述	输入参数	返回值
newNode	创建并初始化一个新的最小堆节点	‘char data‘：节点数据, ‘unsigned freq‘：节点频率	‘struct MinHeapNode*‘ (新节点)
createMinHeap	创建一个新的最小堆并初始化其容量和大小	‘unsigned capacity‘：最小堆的容量	‘struct MinHeap*‘ (新堆)
swapMinHeapNode	交换最小堆中的两个节点	‘struct MinHeapNode** a‘, ‘struct MinHeapNode** b‘	无返回值
minHeapify	维护最小堆的性质：递归地将节点放到合适的位置	‘struct MinHeap* minHeap‘, ‘int idx‘	无返回值
isSizeOne	检查最小堆的大小是否为 1	‘struct MinHeap* minHeap‘	‘1‘ (大小为 1) 或 ‘0‘ (非 1)
extractMin	从最小堆中提取最小的节点，并调整堆结构	‘struct MinHeap* minHeap‘	‘struct MinHeapNode*‘ (最小节点)
insertMinHeap	向最小堆中插入一个节点，并维护堆的性质	‘struct MinHeap* minHeap‘, ‘struct MinHeapNode* minHeapNode‘	无返回值
buildMinHeap	构建最小堆，从一个无序数组构建堆	‘struct MinHeap* minHeap‘	无返回值
printArr	打印数组中的元素	‘int arr[]‘：数组, ‘int n‘：数组大小	无返回值
isLeaf	判断给定的节点是否为叶子节点（即没有子节点）	‘struct MinHeapNode* root‘	‘1‘ (是叶子节点) 或 ‘0‘ (不是叶子节点)
createAndBuildMinHeap	创建并构建一个最小堆，通过数据和频率数组构建最小堆	‘char data[]‘：字符数组, ‘int freq[]‘：频率数组, ‘int size‘：数组大小	‘struct MinHeap*‘ (新堆)
buildHuffmanTree	使用最小堆构建哈夫曼树	‘char data[]‘：字符数组, ‘int freq[]‘：频率数组, ‘int size‘：数组大小	‘struct MinHeapNode*‘ (哈夫曼树的根节点)
printCodes	递归打印哈夫曼编码，并显示每个字符及其对应的编码	‘struct MinHeapNode* root‘：哈夫曼树根节点, ‘int arr[]‘：存储编码的数组, ‘int top‘：数组索引	无返回值
HuffmanCodes	构建哈夫曼树并打印哈夫曼编码	‘char data[]‘：字符数组, ‘int freq[]‘：频率数组, ‘int size‘：数组大小	无返回值

(3) 算法分析

1) 哈夫曼树的构建

- 初始化最小堆

  每个字符和它的频率构成一个节点，所有的字符都成为最小堆中的节点，堆中的节点按频率从小到大排列。

```c
// 创建最小堆节点
struct MinHeapNode* newNode(char data, unsigned freq) {
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc(
        sizeof(struct MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

// 创建最小堆
struct MinHeap* createMinHeap(unsigned capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(
        struct MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->
        capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}
```

- 哈夫曼树的创建

  每次从堆中提取两个频率最小的节点，合并成一个新的节点，新的节点的频率是两个节点频率之和。

  将新节点插入回最小堆，重复以上过程，直到堆中只剩一个节点，该节点即为哈夫曼树的根。

  伪代码如下：

```
    function buildHuffmanTree(data, freq, size):
    minHeap = createMinHeap(size)   // 创建最小堆
    for i = 0 to size-1:
        node = newNode(data[i], freq[i])  // 创建节点
        insertMinHeap(minHeap, node)      // 插入节点到最小堆

    while minHeap.size > 1:  // 当堆中还有多个节点时，继续合并
        left = extractMin(minHeap)  // 提取最小节点
        right = extractMin(minHeap)  // 提取第二小节点

        newNode = newNode('$', left.freq + right.freq)  // 创建
            新节点
        newNode.left = left  // 设置左子节点
        newNode.right = right  // 设置右子节点

        insertMinHeap(minHeap, newNode)  // 将新节点插入最小堆

    return extractMin(minHeap)  // 返回堆中的唯一节点，即哈夫曼
        树的根
```

- 生成哈夫曼编码

  从哈夫曼树的根节点开始，左子节点对应编码的 0，右子节点对应编码的 1，直到所有叶子节点（字符）都有对应的编码。叶子节点的路径即为对应字符的哈夫曼编码。

  伪代码如下：

```
    function printCodes(root, arr, top):
    if root.left exists:
        arr[top] = 0  // 左子树的编码为 0
        printCodes(root.left, arr, top + 1)

    if root.right exists:
        arr[top] = 1  // 右子树的编码为 1
        printCodes(root.right, arr, top + 1)

    if isLeaf(root):  // 如果当前节点是叶子节点，打印字符的哈夫曼编码
        print root.data, ": ", arr[0..top-1]  // 打印该节点的字符及其编码
```

代码流程图如下图所示：

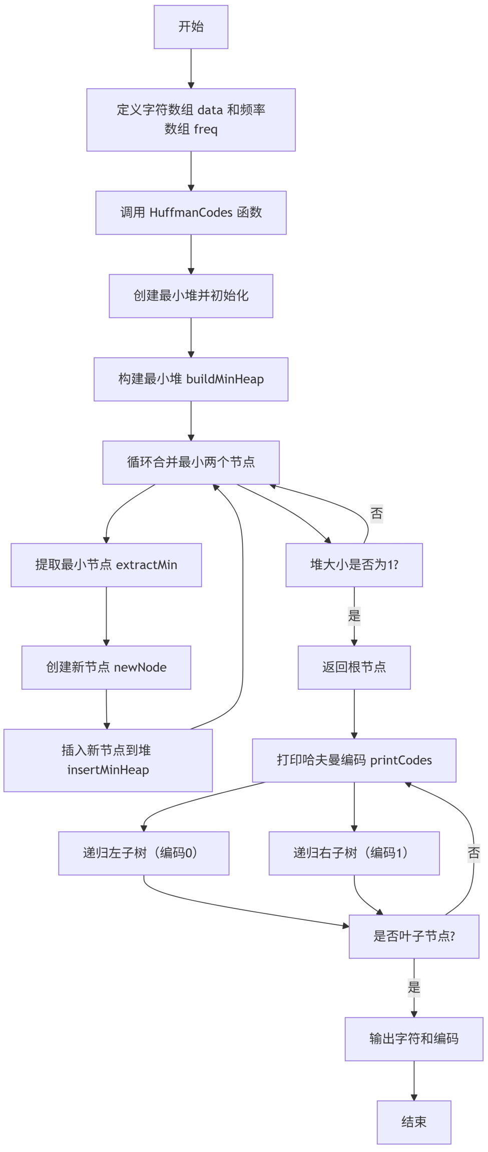

图 7: 生成哈夫曼树及哈夫曼编码流程图

输入：'a', 'b', 'c', 'd', 'e', 'f';

权值：5, 9, 12, 13, 16, 45

程序运行结果如下所示：

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

3 实验心得

通过本次实验，我系统掌握了哈夫曼树的构建原理及其编码生成过程，深入理解了贪心算法在最优前缀编码问题中的应用。在实现过程中，通过对比完全二叉树的顺序存储与链式存储结构，进一步认识到不同存储方式在空间复杂度和操作效率上的特性差异。实验过程中遇到的主要难点包括最小堆的维护、递归生成编码时的路径管理以及内存泄漏问题。通过逐步调试、添加验证性输出语句以及完善内存释放机制，最终解决了这些问题。

此外，本次实验不仅提升了我的 C 语言编程能力，特别是对指针和动态内存管理的运用，也让我更深刻地体会到算法设计与代码实现之间的紧密联系。

参考文献

[1] https://blog.51cto.com/u_1150085/13285238. Accessed: 2025-5-11.

A 完全二叉树的顺序存储和链式存储源码

```c
    #include<stdio.h>
#include<stdlib.h>

#define MAX_SIZE 100

typedef struct{
    int data[MAX_SIZE];
    int size;
}CompleteBinaryTree;
```

```c

typedef struct TreeNode{
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
}TreeNode;

typedef struct{
    TreeNode** nodes;
    int front;
    int rear;
    int capacity;
}Queue;

//顺序存储
void initTree(CompleteBinaryTree* tree) {
    tree->size = 0;
 }

void insertNode(CompleteBinaryTree* tree, int value) {
    if (tree->size >= MAX_SIZE) {
        printf("Error: Tree is full!\n");
        return;
    }
    tree->data[tree->size] = value;
    tree->size++;
 }

void levelOrderTraversal(CompleteBinaryTree*tree){
    if ((tree->size==0))
    {
        printf("Empty\n");
        return;
    }
    printf("Orde:\n");
    for (int i = 0; i < tree->size; i++)
    {
        printf("%d ",tree->data[i]);
    }
    printf("\n");
```

```c
}
//链式存储
int isEmpty(Queue*queue){
    if (queue->front==queue->rear)
    {
        return 1;
    }else
    {
        return 0;
    }


}

void initQueue(Queue* queue, int capacity) {
    queue->nodes = (TreeNode**)malloc(sizeof(TreeNode*) * capacity);
    queue->front = queue->rear = 0;
    queue->capacity = capacity;
}

void enqueue(Queue* queue, TreeNode* node) {
    if (queue->rear == queue->capacity) {
        printf("Queue is full!\n");
        return;
    }
    queue->nodes[queue->rear++] = node;
}

TreeNode* dequeue(Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty!\n");
        return NULL;
    }
    return queue->nodes[queue->front++];
}

TreeNode*creatNode(int value){
    TreeNode*newnode=(TreeNode*)malloc(sizeof(TreeNode));
    newnode->data=value;
    newnode->left=newnode->right=NULL;
    return newnode;
}
void Insert(TreeNode**root,int value,Queue*queue){
    TreeNode*newnode=creatNode(value);
    if (*root==NULL)
    {
        *root=newnode;
        return;
    }
    enqueue(queue, *root);
    while (!isEmpty(queue))
    {
         TreeNode* current = dequeue(queue);
        if (current->left == NULL) {
        current->left = newnode;
        break;
    }

        if (current->right == NULL) {
        current->right = newnode;
        break;
        }
    enqueue(queue, current->right);
    }
}
void levelOrderTraversalWithQueue(TreeNode* root) {
    if (root == NULL) {
        printf("Empty\n");
        return;
    }
    Queue queue;
    initQueue(&queue, MAX_SIZE);
    enqueue(&queue, root);
    printf("Order:\n");

    while (!isEmpty(&queue)) {
        TreeNode* current = dequeue(&queue);
        printf("%d ", current->data);
        if (current->left != NULL) {
            enqueue(&queue, current->left);
        }
        if (current->right != NULL) {
            enqueue(&queue, current->right);
        }
    }

    printf("\n");
}

int main() {
    CompleteBinaryTree tree;
    initTree(&tree);
    for (int i = 0; i < 9; i++)
    {
        insertNode(&tree, i);
    }
    levelOrderTraversal(&tree);

    TreeNode* root = NULL;
    Queue queue;
    initQueue(&queue, MAX_SIZE);
    for (int i = 0; i < 9; i++)
    {
        Insert(&root, i, &queue);
    }
    levelOrderTraversalWithQueue(root);

    return 0;
}
```

B 构造哈夫曼树并生成哈夫曼编码实验源码

```
    #include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_TREE_HT 100

struct MinHeapNode {
    char data;
    unsigned freq;
    struct MinHeapNode *left, *right;
};

struct MinHeap {
    unsigned size;
    unsigned capacity;
    struct MinHeapNode** array;
};


struct MinHeapNode* newNode(char data, unsigned freq) {
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc(sizeof(
       struct MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}


struct MinHeap* createMinHeap(unsigned capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct
       MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity *
        sizeof(struct MinHeapNode*));
    return minHeap;
}


void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
   {
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

```

```

void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int 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) {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[
            idx]);
        minHeapify(minHeap, smallest);
    }
}

int isSizeOne(struct MinHeap* minHeap) {
    return (minHeap->size == 1);
}

struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode*
   minHeapNode) {
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq
        ) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
```

```c
        i = (i - 1) / 2;
    }
    minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap* minHeap) {
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

void printArr(int arr[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}

int isLeaf(struct MinHeapNode* root) {
    return !(root->left) && !(root->right);
}

struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
    struct MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
    struct MinHeapNode *left, *right, *top;
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    while (!isSizeOne(minHeap)) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }

    return extractMin(minHeap);
}

void printCodes(struct MinHeapNode* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    if (isLeaf(root)) {
        printf("%c: ", root->data);
        printArr(arr, top);
    }
}


void HuffmanCodes(char data[], int freq[], int size) {
    struct MinHeapNode* root = buildHuffmanTree(data, freq, size);
    int arr[MAX_TREE_HT], top = 0;
    printCodes(root, arr, top);
}


int main() {
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr) / sizeof(arr[0]);
    HuffmanCodes(arr, freq, size);

    return 0;
}
```

C 基于哈夫曼树的文件压缩、解压源码

```
    #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define MAX_TREE_HT 256
#define BUFFER_SIZE 4096

typedef struct HuffmanNode {
    unsigned char data;
    unsigned freq;
    struct HuffmanNode *left, *right;
} HuffmanNode;

typedef struct MinHeap {
    unsigned size;
    unsigned capacity;
    HuffmanNode** array;
} MinHeap;

typedef struct HuffmanCode {
    unsigned char code[MAX_TREE_HT];
    int length;
} HuffmanCode;

HuffmanNode* newNode(unsigned char data, unsigned freq) {
    HuffmanNode* node = (HuffmanNode*)malloc(sizeof(HuffmanNode));
    node->left = node->right = NULL;
    node->data = data;
    node->freq = freq;
    return node;
}

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

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

void minHeapify(MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int 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);
    }
}

int isSizeOne(MinHeap* minHeap) {
    return (minHeap->size == 1);
}

```

```c
HuffmanNode* extractMin(MinHeap* minHeap) {
    HuffmanNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

void insertMinHeap(MinHeap* minHeap, HuffmanNode* node) {
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && node->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    minHeap->array[i] = node;
}

void buildMinHeap(MinHeap* minHeap) {
    int n = minHeap->size - 1;
    for (int i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

MinHeap* createAndBuildMinHeap(unsigned char data[], unsigned freq[],
     int size) {
    MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

HuffmanNode* buildHuffmanTree(unsigned char data[], unsigned freq[],
    int size) {
    HuffmanNode *left, *right, *top;
    MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    while (!isSizeOne(minHeap)) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }

    return extractMin(minHeap);
}

int isLeaf(HuffmanNode* root) {
    return !(root->left) && !(root->right);
}

void generateCodes(HuffmanNode* root, HuffmanCode codes[], unsigned
   char arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        generateCodes(root->left, codes, arr, top + 1);
    }
    if (root->right) {
        arr[top] = 1;
        generateCodes(root->right, codes, arr, top + 1);
    }
    if (isLeaf(root)) {
        for (int i = 0; i < top; ++i) {
            codes[root->data].code[i] = arr[i];
        }
        codes[root->data].length = top;
    }
}

void freeHuffmanTree(HuffmanNode* root) {
    if (root == NULL) return;
    freeHuffmanTree(root->left);
    freeHuffmanTree(root->right);
    free(root);
}
```

```c

void countFrequencies(FILE* file, unsigned freq[]) {
    unsigned char buffer[BUFFER_SIZE];
    size_t bytesRead;

    rewind(file);
    while ((bytesRead = fread(buffer, 1, BUFFER_SIZE, file)) > 0) {
        for (size_t i = 0; i < bytesRead; ++i) {
            freq[buffer[i]]++;
        }
    }
}

void buildHuffmanCodes(FILE* inputFile, HuffmanCode codes[],
   HuffmanNode** treeRoot) {
    unsigned freq[256] = {0};
    countFrequencies(inputFile, freq);

    int size = 0;
    for (int i = 0; i < 256; ++i) {
        if (freq[i] > 0) size++;
    }

    unsigned char data[size];
    unsigned frequencies[size];
    int index = 0;
    for (int i = 0; i < 256; ++i) {
        if (freq[i] > 0) {
            data[index] = (unsigned char)i;
            frequencies[index] = freq[i];
            index++;
        }
    }

    *treeRoot = buildHuffmanTree(data, frequencies, size);

    unsigned char arr[MAX_TREE_HT];
    generateCodes(*treeRoot, codes, arr, 0);
}

```

```
void writeTreeToFile(HuffmanNode* root, FILE* out) {
    if (root == NULL) return;

    if (isLeaf(root)) {
        fputc('1', out);
        fputc(root->data, out);
    } else {
        fputc('0', out);
        writeTreeToFile(root->left, out);
        writeTreeToFile(root->right, out);
    }
}

HuffmanNode* readTreeFromFile(FILE* in) {
    int bit = fgetc(in);
    if (bit == '1') {
        unsigned char data = fgetc(in);
        return newNode(data, 0);
    } else {
        HuffmanNode* left = readTreeFromFile(in);
        HuffmanNode* right = readTreeFromFile(in);
        HuffmanNode* node = newNode('$', 0);
        node->left = left;
        node->right = right;
        return node;
    }
}

void compressFile(const char* inputFileName, const char*
   outputFileName) {
    FILE* inputFile = fopen(inputFileName, "rb");
    if (!inputFile) {
        perror("Unable to open input file");
        exit(EXIT_FAILURE);
    }

    FILE* outputFile = fopen(outputFileName, "wb");
    if (!outputFile) {
        perror("Unable to create output file");
        fclose(inputFile);
        exit(EXIT_FAILURE);
    }

    HuffmanCode codes[256] = {0};
    HuffmanNode* huffmanTree = NULL;
    buildHuffmanCodes(inputFile, codes, &huffmanTree);

    writeTreeToFile(huffmanTree, outputFile);
    fputc('\0', outputFile);

    unsigned char buffer[BUFFER_SIZE];
    unsigned char byte = 0;
    int bitCount = 0;
    size_t bytesRead;

    rewind(inputFile);
    while ((bytesRead = fread(buffer, 1, BUFFER_SIZE, inputFile)) >
       0) {
        for (size_t i = 0; i < bytesRead; ++i) {
            unsigned char c = buffer[i];
            for (int j = 0; j < codes[c].length; ++j) {
                byte = (byte << 1) | codes[c].code[j];
                bitCount++;
                if (bitCount == 8) {
                    fputc(byte, outputFile);
                    byte = 0;
                    bitCount = 0;
                }
            }
        }
    }

    if (bitCount > 0) {
        byte <<= (8 - bitCount);
        fputc(byte, outputFile);
    }

    fclose(inputFile);
    fclose(outputFile);
    freeHuffmanTree(huffmanTree);
    printf("SUCCESS: %s -> %s\n", inputFileName, outputFileName);
}

void decompressFile(const char* inputFileName, const char*
   outputFileName) {
    FILE* inputFile = fopen(inputFileName, "rb");
    if (!inputFile) {
        perror("Unable to open the file");
        exit(EXIT_FAILURE);
    }

    FILE* outputFile = fopen(outputFileName, "wb");
    if (!outputFile) {
        perror("Unable to create output file");
        fclose(inputFile);
        exit(EXIT_FAILURE);
    }

    HuffmanNode* huffmanTree = readTreeFromFile(inputFile);
    fgetc(inputFile);

    HuffmanNode* currentNode = huffmanTree;
    int bitPos = 7;
    unsigned char byte;
    int inputChar;

    while (inputChar = fgetc(inputFile)) {
        if (inputChar == EOF) break;

        byte = (unsigned char)inputChar;
        bitPos = 7;

        while (bitPos >= 0) {
            int bit = (byte >> bitPos) & 1;
            bitPos--;

            if (bit) {
                currentNode = currentNode->right;
            } else {
                currentNode = currentNode->left;
```

```c
            }

            if (isLeaf(currentNode)) {
                fputc(currentNode->data, outputFile);
                currentNode = huffmanTree;
            }
        }
    }

    fclose(inputFile);
    fclose(outputFile);
    freeHuffmanTree(huffmanTree);
    printf("File decompressed successfully: %s -> %s\n", inputFileName, outputFileName);
}

int main(int argc, char* argv[]) {
    if (argc != 4) {
        printf("Usage:\n");
        printf("  Compress: %s -c inputfile outputfile\n", argv[0]);
        printf("  Decompress: %s -d inputfile outputfile\n", argv[0]);
        return EXIT_FAILURE;
    }

    if (strcmp(argv[1], "-c") == 0) {
        compressFile(argv[2], argv[3]);
    } else if (strcmp(argv[1], "-d") == 0) {
        decompressFile(argv[2], argv[3]);
    } else {
        printf("Invalid option. Use -c to compress or -d to decompress.\n");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```