```c
#define _CRT_SECURE_NO_WARNINGS

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

#define MAX_TREE_HT 100
#define MAX_CHAR 256
#define _CRT_SECURE_NO_WARNINGS  // Disable warnings for scanf

// Huffman Tree node structure
typedef struct HuffmanNode {
    char data;
    unsigned freq;
    struct HuffmanNode* left, * right;
} HuffmanNode;

// Linked list node for displaying the tree
typedef struct ListNode {
    HuffmanNode* huffmanNode;
    struct ListNode* next;
} ListNode;

// A Min Heap structure
typedef struct MinHeap {
    unsigned size;
    unsigned capacity;
    HuffmanNode** array;
} MinHeap;

// Function declarations
HuffmanNode* newNode(char data, unsigned freq);
MinHeap* createMinHeap(unsigned capacity);
void swapMinHeapNode(HuffmanNode** a, HuffmanNode** b);
void minHeapify(MinHeap* minHeap, int idx);
HuffmanNode* extractMin(MinHeap* minHeap);
void insertMinHeap(MinHeap* minHeap, HuffmanNode* minHeapNode);
void buildMinHeap(MinHeap* minHeap);
void printCodes(HuffmanNode* root, int arr[], int top);
void storeCodes(HuffmanNode* root, int arr[], int top, char codes[]
[MAX_TREE_HT]);
MinHeap* createAndBuildMinHeap(char data[], int freq[], int size);
HuffmanNode* buildHuffmanTree(char data[], int freq[], int size);
void HuffmanCodes(char data[], int freq[], int size, char codes[][MAX_TREE_HT]);
void encodeFile(const char* sourceFile, const char* encodedFile, char codes[]
[MAX_TREE_HT]);
void decodeFile(const char* encodedFile, const char* decodedFile, HuffmanNode*
root);
void printHuffmanTree(HuffmanNode* root);
void freeTree(HuffmanNode* root);
void addToList(ListNode** head, HuffmanNode* huffmanNode);
void printList(ListNode* head);

// Main function
```

```c
int main() {
    int choice;
    char sourceFile[256], encodedFile[256], decodedFile[256];
    char codes[MAX_CHAR][MAX_TREE_HT];
    HuffmanNode* huffmanTree = NULL;
    int freq[MAX_CHAR] = { 0 }; // Frequency array
    char data[MAX_CHAR];
    int size = 0;

    printf("\n========== 哈夫曼编/译码器 =========="
        "\n1. 读取文件内容并显示"
        "\n2. 统计字符频率并显示"
        "\n3. 建立哈夫曼树"
        "\n4. 编码并输出结果到文件 (*.code)"
        "\n5. 解码并输出结果到文件 (*.decode)"
        "\n6. 退出程序\n");

    while (1) {
        printf("\n请选择操作 (1-6): ");
        scanf_s("%d", &choice);

        switch (choice) {
        case 1: {
            printf("请输入源文件路径 (*.source): ");
            scanf_s("%s", sourceFile, (unsigned)_countof(sourceFile));

            FILE* file;
            if (fopen_s(&file, sourceFile, "r") != 0) {
                printf("文件打开失败: %s\n", sourceFile);
                break;
            }

            printf("\n文件内容:\n");
            char ch;
            while ((ch = fgetc(file)) != EOF) {
                putchar(ch);  // Display the content
            }
            printf("\n");
            fclose(file);
            break;
        }

        case 2: {
            printf("请输入源文件路径 (*.source): ");
            scanf_s("%s", sourceFile, (unsigned)_countof(sourceFile));

            FILE* file;
            if (fopen_s(&file, sourceFile, "r") != 0) {
                printf("文件打开失败: %s\n", sourceFile);
                break;
            }

            char ch;
            while ((ch = fgetc(file)) != EOF) {
                freq[(unsigned char)ch]++;
            }
            fclose(file);

            printf("字符频率:\n");
            for (int i = 0; i < MAX_CHAR; i++) {
                if (freq[i] > 0) {
                    printf("字符 '%c': %d\n", i, freq[i]);
                    data[size++] = (char)i; // Store the character for the
Huffman tree
                }
            }
            break;
        }

        case 3: {
            if (size == 0) {
                printf("请先统计字符频率!\n");
                break;
            }
            huffmanTree = buildHuffmanTree(data, freq, size);
            printf("哈夫曼树构建成功\n");
            //printHuffmanTree(huffmanTree);
            break;
        }

        case 4: {
            if (!huffmanTree) {
                printf("请先建立哈夫曼树!\n");
                break;
            }
            printf("请输入编码文件路径 (*.code): ");
            scanf_s("%s", encodedFile, (unsigned)_countof(encodedFile));
            HuffmanCodes(data, freq, size, codes);
            encodeFile(sourceFile, encodedFile, codes);
            printf("文件编码完成.\n");
            break;
        }

        case 5: {
            if (!huffmanTree) {
                printf("请先建立哈夫曼树!\n");
                break;
            }
            printf("请输入编码文件路径 (*.code): ");
            scanf_s("%s", encodedFile, (unsigned)_countof(encodedFile));
            printf("请输入译码文件路径 (*.decode): ");
            scanf_s("%s", decodedFile, (unsigned)_countof(decodedFile));
            decodeFile(encodedFile, decodedFile, huffmanTree);
            printf("文件解码完成.\n");
            break;
        }

        case 6:
            printf("退出程序.\n");
            if (huffmanTree) {
                freeTree(huffmanTree);
            }
            exit(0);

        default:
            printf("无效选项，请重新选择.\n");
        }
    }

    return 0;
}

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

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

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

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

```c
    minHeap->size--;
    minHeapify(minHeap, 0);
    return temp;
}

void insertMinHeap(MinHeap* minHeap, HuffmanNode* 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];
        i = (i - 1) / 2;
    }
    minHeap->array[i] = minHeapNode;
}

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

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

    while (minHeap->size != 1) {
        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 addToList(ListNode** head, HuffmanNode* huffmanNode) {
    ListNode* newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->huffmanNode = huffmanNode;
    newNode->next = *head;
    *head = newNode;
}
```

```c
void printList(ListNode* head) {
    while (head != NULL) {
        if (head->huffmanNode->data != '$') {
            printf("字符: '%c', 频率: %d\n", head->huffmanNode->data, head-
>huffmanNode->freq);
        }
        else {
            printf("合并节点, 频率: %d\n", head->huffmanNode->freq);
        }
        head = head->next;
    }
}

//void printHuffmanTree(HuffmanNode* root, int space) {
//    // Base case
//    if (root == NULL) return;
//
//    // Increase distance between levels
//    space += 10;
//
//    // Process right child first
//    printHuffmanTree(root->right, space);
//
//    // Print current node after space
//    printf("\n");
//    for (int i = 10; i < space; i++) {
//        printf(" ");
//    }
//    // If it's an internal node, print '*'
//    printf("%c (%d)\n", root->data ? root->data : '*', root->freq);
//
//    // Print connecting line for left child
//    if (root->left != NULL || root->right != NULL) {
//        for (int i = 10; i < space - 10; i++) {
//            printf(" ");
//        }
//        printf("|\n");
//    }
//
//    // Process left child
//    printHuffmanTree(root->left, space);
//}

void freeTree(HuffmanNode* root) {
    if (root) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}

void HuffmanCodes(char data[], int freq[], int size, char codes[][MAX_TREE_HT]) {
    HuffmanNode* root = buildHuffmanTree(data, freq, size);
    int arr[MAX_TREE_HT], top = 0;
    storeCodes(root, arr, top, codes);
}
```

```c
void storeCodes(HuffmanNode* root, int arr[], int top, char codes[][MAX_TREE_HT])
{
    if (root->left) {
        arr[top] = 0;
        storeCodes(root->left, arr, top + 1, codes);
    }
    if (root->right) {
        arr[top] = 1;
        storeCodes(root->right, arr, top + 1, codes);
    }
    if (!(root->left) && !(root->right)) {
        codes[(unsigned char)root->data][top] = '\0';
        for (int i = 0; i < top; i++) {
            codes[(unsigned char)root->data][i] = arr[i] + '0';
        }
    }
}


void encodeFile(const char* sourceFile, const char* encodedFile, char codes[]
[MAX_TREE_HT]) {
    FILE* inFile = fopen(sourceFile, "r");
    if (inFile == NULL) {
        printf("无法打开源文件: %s\n", sourceFile);
        return;
    }

    FILE* outFile = fopen(encodedFile, "wb");
    if (outFile == NULL) {
        printf("无法打开编码文件: %s\n", encodedFile);
        fclose(inFile);
        return;
    }

    printf("编码内容:\n");
    char ch;
    while ((ch = fgetc(inFile)) != EOF) {
        // Write the Huffman code for the character to the output file
        for (int i = 0; codes[(unsigned char)ch][i] != '\0'; i++) {
            fputc(codes[(unsigned char)ch][i], outFile);
            // Also print to the console
            putchar(codes[(unsigned char)ch][i]);
        }
    }

    fclose(inFile);
    fclose(outFile);
    printf("\n编码完成，输出到文件: %s\n", encodedFile);
}

void decodeFile(const char* encodedFile, const char* decodedFile, HuffmanNode*
root) {
    FILE* inFile = fopen(encodedFile, "rb");
    if (inFile == NULL) {
        printf("无法打开编码文件: %s\n", encodedFile);
        return;
    }

    FILE* outFile = fopen(decodedFile, "w");
    if (outFile == NULL) {
        printf("无法打开解码文件: %s\n", decodedFile);
        fclose(inFile);
        return;
    }

    printf("解码内容:\n");
    HuffmanNode* current = root;
    char bit;
    while ((bit = fgetc(inFile)) != EOF) {
        if (bit == '0') {
            current = current->left;
        }
        else if (bit == '1') {
            current = current->right;
        }

        // If we reach a leaf node, write the character to the output file
        if (current->left == NULL && current->right == NULL) {
            fputc(current->data, outFile);
            putchar(current->data);  // Print the decoded character to the
console
            current = root; // Go back to the root for the next character
        }
    }

    fclose(inFile);
    fclose(outFile);
    printf("\n解码完成, 输出到文件: %s\n", decodedFile);
}
```