

北京邮电大学

数据结构实验报告



题目： 先序建立二叉树 + 哈夫曼编码压缩与解压

姓 名： 魏生辉

学 院： 计算机学院（国家示范性软件学院）

专 业： 计算机类

班 级： 2023211307

学 号： 2023211075

指导教师： 杨震

2024年 11月

目录

先序建立二叉树	3
1 需求分析	3
1.1 题目描述	3
1.2 输入描述	3
1.3 输出描述	3
1.4 样例输入输出	3
1.5 程序功能	4
2 概要设计	4
2.1 问题解决思路概述	4
2.2 数据结构类型定义	5
2.3 主程序的流程	5
3 详细设计	6
4 调试分析报告	8
4.1 调试过程中遇到的问题和解决方法	8
4.2 设计实现的回顾讨论	8
4.3 算法复杂度分析	9
5 用户使用说明	9
6 测试结果	9
6.1 测试 1--基础样例测试	9
6.2 测试 2--非法输入和边界条件	9
6.2 测试 3--利用输出语句与人力手动计算结果校对	9
哈夫曼编码压缩与解压	
1 需求分析	10
1.1 题目描述	10
1.2 输入描述	10
1.3 输出描述	11
1.4 样例输入输出	11
1.5 程序功能	12
2 概要设计	13
2.1 问题解决思路	13
2.2 数据结构类型定义	13
2.3 主程序的流程	14
3 详细设计	15
3.1 伪代码的设计	15
4 调试分析报告	18
4.1 调试过程中遇到的问题和解决方法	18
4.2 设计实现的回顾讨论	18
4.3 算法复杂度分析	19
5 用户使用说明	19
6 测试结果	19
6.1 测试 1--基础样例测试	19
6.2 测试 2--非法输入和边界条件	19
6.2 测试 3--利用输出语句与人力手动计算结果校对	19
7 个人总结	20
8 致谢	20

先序建立二叉树

1 需求分析

1.1 题目描述

用先序递归过程建立二叉树（存储结构：二叉链表），输出二叉树的四种表示法之一。

1.2 输入描述

输入第一行为一串字符串:数据按先序遍历所得序列输入，当某结点左子树或右子树为空时，输入 ‘*’ 号。
输入第二行为一个数字用于选择功能：输入1则可查看可视化二叉树，输入2可以查看先序序列

1.3 输出描述

输出结果划分为三种情况：

- 1. 输入字符串合法且选择功能1：输出可视化二叉树。
- 2. 输入字符串合法且选择功能2：此时输出先序序列。
- 3. 输入字符串不合法："错误输入，请您重新输入

1.4 样例输入输出

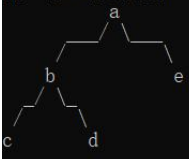
1.4.1 样例输入输出 1

请输入合法二叉树的字符串: abc**d**e**

请选择功能

(1) 查看我的二叉树
(2) 查看先序遍历序列

输入你选择的功能: 1



1.4.2 样例输入输出 2

请输入合法二叉树的字符串: abc**d**e**

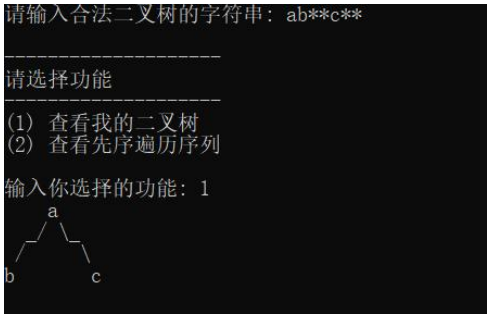
请选择功能

(1) 查看我的二叉树
(2) 查看先序遍历序列

输入你选择的功能: 2

先序遍历序列: a b c d e

1.4.3 样例输入输出 3



1.5 程序功能

程序通过先序遍历序列生成二叉树，根据选项打印二叉树的图形或者先序遍历结果。

2 概要设计

2.1 问题解决思路

概述

①构建二叉树的递归算法

递归构建：采用递归方法根据先序遍历序列构建二叉树。每次读取一个字符：如果字符为*，则返回NULL，表示当前节点为空。

否则，创建一个新节点，并递归构建其左子树和右子树。

位置计算：在构建过程中，为了实现图形化显示，需要为每个节点计算其在图形中的位置。通过递归方式，将子节点的位置累加并平均，以确定父节点的位置。如果子节点为空，则根据特定规则（例如specialNodeCount计数器）分配一个新的位置。

②先序遍历的实现

递归遍历：实现先序遍历函数，通过递归访问根节点、左子树和右子树，依次输出节点的值。

③ 二叉树图形化显示的实现

层序遍历与队列：采用层序遍历的方法，通过一个队列来逐层访问二叉树的节点。每一层的节点根据其计算出的position值，在控制台上按相应的位置输出。

位置对齐与符号绘制：

节点输出：根据节点的position值，在适当的位置输出节点的字符值。

连接线绘制：在节点的下方绘制斜杠 (/) 和反斜杠 (\) 以及下划线 (_)，以表示父子节点之间的连接关系

队列管理：使用固定大小的队列（如数组）来存储待处理的节点，并动态调整队列的头尾指针以实现层序遍历

2.2 数据结构类型定义

```
typedef struct TreeNode
{
    char value;
    struct TreeNode *leftChild, *rightChild;
    int position;
} TreeNode;
```

2.3 主程序的流程

本程序的主程序（main 函数）负责整体流程的控制，从接收用户输入的先序遍历序列开始，构建二叉树，提供用户选择功能的界面，并根据用户的选择执行相应的操作。以下是主程序的详细流程说明：

1. 程序启动与用户提示

输出提示信息：程序启动后，首先通过 `printf` 函数向用户提示输入合法的二叉树字符串。

接收输入并构建二叉树：调用 `buildBinaryTree` 函数，根据用户输入的先序遍历序列构建二叉树。该函数通过递归读取每个字符，并根据字符是否为 * 决定是否创建新节点或返回 `NULL`。

2. 显示功能菜单

输出功能选项：构建完二叉树后，程序通过一系列 `printf` 和 `puts` 函数输出功能选择菜单，供用户选择执行的操作。

3. 用户选择功能

接收用户输入：定义一个字符数组 `userChoice` 用于存储用户的选择。通过 `scanf` 函数读取用户输入的功能选项，并进行输入验证，确保输入有效。

4. 执行用户选择的功能

功能选项判断与执行：根据用户输入的选项，通过条件判断（`if-else` 结构）执行相应的功能：

选项 '1'：查看二叉树图形

调用 `printGraph` 函数，传入根节点 `root`，在控制台上以图形化的方式显示二叉树的结构

选项 '2'：查看先序遍历序列

调用 `preOrderTraverse` 函数，传入根节点 `root`，并在控制台上输出先序遍历的节点序列。遍历完成后输出一个换行符以美化显示。

3 详细设计

3.1 伪代码的设计及树的实现

```
// 定义构建二叉树的函数
函数 buildBinaryTree() 返回 TreeNode 指针:
    读取一个字符 ch
    如果 ch 等于 '*' 则
        返回 NULL
    创建一个新的 TreeNode 节点 currentNode
    如果 创建失败, 则终止程序
    将 currentNode->value 设为 ch

// 递归构建左子树
currentNode->leftChild = 调用 buildBinaryTree()
如果 currentNode->leftChild 不为 NULL 则
    currentNode->position += currentNode->leftChild->position
否则
    currentNode->position += (specialNodeCount++) * 8 + 1

// 递归构建右子树
currentNode->rightChild = 调用 buildBinaryTree()
如果 currentNode->rightChild 不为 NULL 则
    currentNode->position += currentNode->rightChild->position
否则 如果 currentNode->leftChild 存在 则
    currentNode->position += (specialNodeCount++) * 8 + 1

// 计算当前节点的位置
如果 currentNode 有左子节点 或 右子节点 则
    currentNode->position = currentNode->position / 2

返回 currentNode

// 定义先序遍历的函数
函数 preOrderTraverse(node: TreeNode 指针):
    如果 node 为 NULL 则
        返回
    输出 node->value 和一个空格
    调用 preOrderTraverse(node->leftChild)
    调用 preOrderTraverse(node->rightChild)

// 定义输出二叉树图形的函数
函数 printGraph(root: TreeNode 指针):
    如果 root 为 NULL 则
        输出 "二叉树为空。"
        返回
    定义一个大小为 100 的队列 queue
    设置队列的头指针 head = 1
    设置队列的尾指针 tail = 1
    将 root 放入 queue[1]

    当 head <= tail 时循环:
        设置 currentPos = 1

        // 输出当前层的节点值
        对于 i 从 head 到 tail:
            当 currentPos < queue[i]->position 时:
```

```
    输出空格
    currentPos += 1
    输出 queue[i]->value
    currentPos += 1
    输出换行符
    设置 currentPos = 1

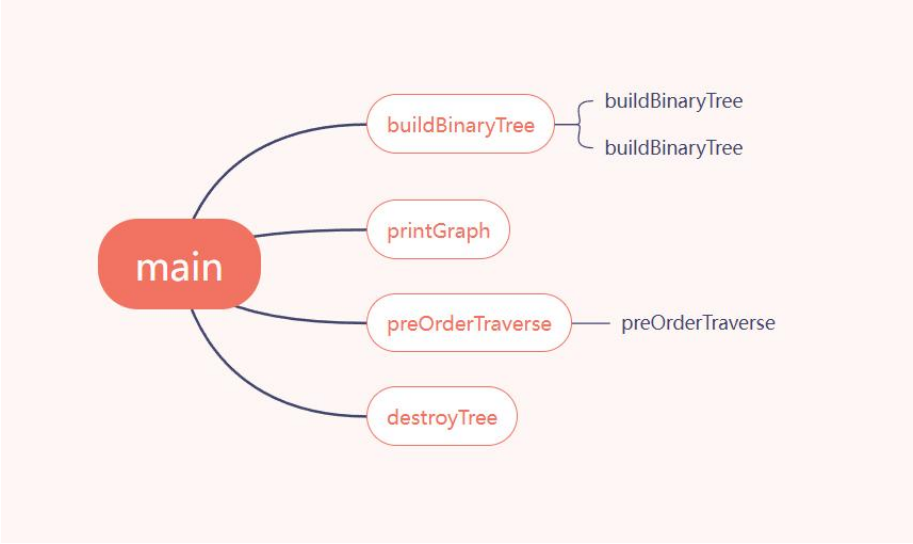
// 输出连接线（斜杠和反斜杠）
对于 i 从 head 到 tail:
    如果 queue[i]->leftChild 存在 则:
        当 currentPos < queue[i]->leftChild->position + 2 时:
            输出空格
            currentPos += 1
        当 currentPos < queue[i]->position - 1 时:
            输出 '_'
            currentPos += 1
        输出 '/'
        currentPos += 1
    如果 queue[i]->rightChild 存在 则:
        当 currentPos < queue[i]->position + 1 时:
            输出空格
            currentPos += 1
        输出 '\\'
        currentPos += 1
        当 currentPos < queue[i]->rightChild->position - 1 时:
            输出 '_'
            currentPos += 1
    输出换行符

    设置 end = tail
    设置 currentPos = 1

// 输出下一层的连接斜杠和反斜杠，并将子节点入队
对于 i 从 head 到 end:
    如果 queue[i]->leftChild 存在 则:
        当 currentPos < queue[i]->leftChild->position + 1 时:
            输出空格
            currentPos += 1
        输出 '/'
        currentPos += 1
    如果 queue[i]->rightChild 存在 则:
        当 currentPos < queue[i]->rightChild->position - 1 时:
            输出空格
            currentPos += 1
        输出 '\\'
        currentPos += 1
    如果 queue[i]->leftChild 存在 则:
        tail += 1
        queue[tail] = queue[i]->leftChild
    如果 queue[i]->rightChild 存在 则:
        tail += 1
        queue[tail] = queue[i]->rightChild
    head += 1
    输出换行符

// 定义释放二叉树内存的函数
函数 destroyTree(node: TreeNode 指针):
```

3.2 函数的调用关系图



4 调试分析报告

4.1 调试过程中遇到的问题和解决方法

初步完成本实验的基础代码后，通过了几组简单的样例，随后进行严格的非法输入，发现了些许问题。

问题 1：输入缓冲区未正确读取所有字符

原因：使用 `getchar()` 函数逐个读取字符时，输入缓冲区中可能存在多余的换行符或空格，导致读取过程提前终止或错误解析。

解决方法：清空输入缓冲区-在读取完所有必要的字符后，添加代码以清空缓冲区，确保不会影响后续的输入操作

问题 2：二叉树图形化显示位置计算错误

输出的二叉树图形显示不对称，节点位置错乱，导致树形结构无法正确反映实际的二叉树结构。

原因：节点 `position` 的计算逻辑存在缺陷，未能正确分配和调整每个节点在图形中的位置，尤其是在处理空节点和子节点位置累加时。

解决方法：重新设计位置计算逻辑：确保每个节点的 `position` 能够反映其在图形中的准确水平位置。

修改 `buildBinaryTree` 函数中的位置计算方式，使其基于子节点的相对位置进行合理分配。

引入水平偏移量：为每个节点分配一个基于其深度和兄弟节点的水平偏移量，确保树形结构的对称性

4.2 设计实现的回顾讨论

给每个叶结点和度数为 1 的结点的空子树留下长度为 8 的空格。通过' /'（斜杠），' |'（反斜杠），' _'（下划线）符号来表示二叉树的边。但由于需要兼顾时间复杂度和编程复杂度，在树中存在较多的链时会略微缺少可读性。

4.3 算法复杂度分析

整体时间复杂度为 $O(n^2)$ 。
整体空间复杂度为 $O(n)$ 。

5 用户使用说明

- 1.使用 gcc 编译生成可执行文件。
`gcc -o main -std=c11 main.c`
- 2. 执行可执行文件：
在在 Linux或macOS 环境下： `./main`
• 在 Windows cmd 环境下： `main`
- 3. 输入数据和输出参照1.2/1.3部分

6 测试结果

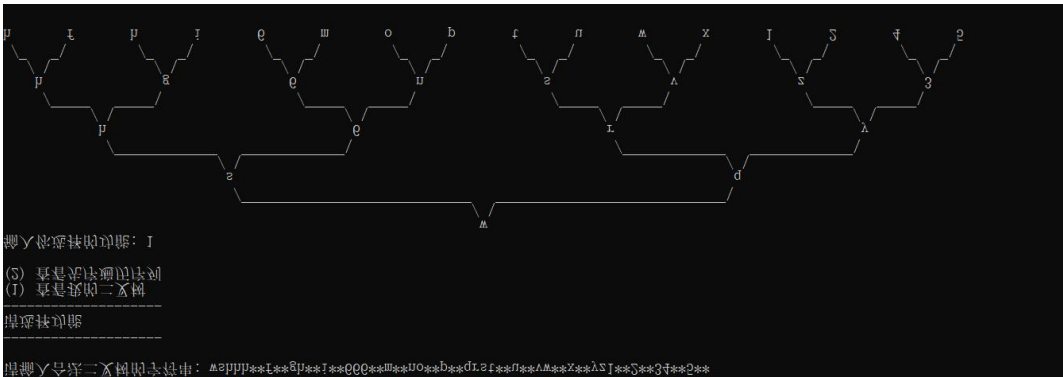
测试部分划分为如下环节。

6.1 测试1—基础样例测试

进行1.4 节的样例测试。

6.2测试2—非法输入和边界条件

测试非法输入和边界条件。



6.2 测试3—利用输出语句与人力手动计算结果校对

与人力手动计算结果校对，完全一致

哈夫曼编码压缩、解压文件

1 需求分析

1.1 题目描述

对于一个指定的文件，应用哈夫曼编码压缩文件，或者对本程序生成的压缩文件进行解压。

1.2 输入描述

①程序启动后的用户选择输入

当用户运行程序时，程序将显示一个交互式菜单，用户需要通过输入数字来选择相应的功能。菜单选项如下：

请选择一个选项：

1. 压缩文件
2. 解压文件
3. 显示帮助
4. 退出

请输入您的选择：

用户需要输入 1、2、3 或 4 来选择相应的功能。

②压缩文件功能的输入

当用户选择 1 进行压缩操作时，程序将提示用户输入以下信息：

请输入要压缩的文件路径，不要输入引号：

请输入输出文件路径（按回车使用默认路径）：

③解压文件功能的输入

当用户选择 2 进行解压操作时，程序将提示用户输入以下信息：

请输入要解压的文件路径：

请输入输出文件路径（按回车使用默认路径）：

④显示帮助功能的输入

当用户选择 3 显示帮助信息时，无需进一步输入，程序将直接输出使用说明。

⑤退出程序的输入

当用户选择 4 退出程序时，无需进一步输入，程序将终止运行。

1.3 输出描述

①压缩文件功能的输出

当用户完成压缩操作后，程序将生成一个 .hzip 格式的压缩文件。

压缩文件包含以下内容：文件头（Header）：4 个字节，内容为 "HZip"，用于标识该文件是由该程序生成的压缩文件。

频率表（Frequency Table）：256 个字节，每个字节对应一个可能的字符（0-255）的出现频率。

压缩数据（Compressed Data）：使用哈夫曼编码后的比特流，以字节为单位存储。

剩余位数（Remainer）：1 个字节，表示最后一个字节中未使用的位数（用于正确解压）。

②解压文件功能的输出

当用户完成解压操作后，程序将生成一个解压后的文件，根据用户提供的压缩文件路径和输出文件路径，生成一个解压后的文件。

解压文件的内容应与原始文件完全一致。

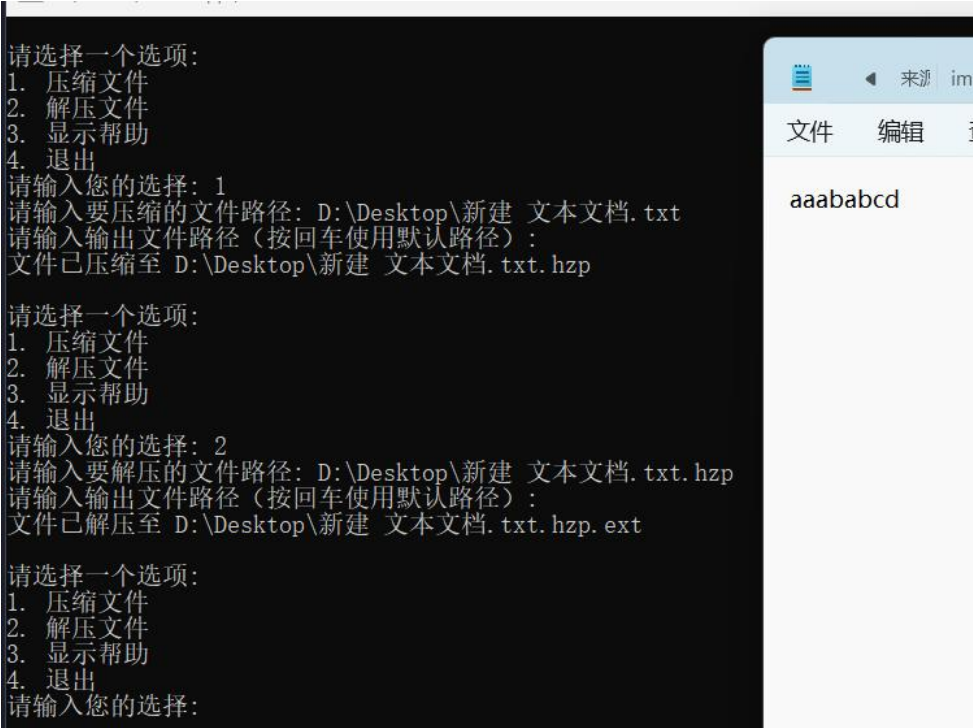
1.4 样例输入输出

1.4.1 压缩样例输入输出 1



(使用010editor / HxD 打开 hzip 文件)

1.4.2 解压样例输入输出 2



1.5 程序功能

- ①文件压缩 (Compress)
 - 输入文件路径: 用户指定需要压缩的源文件路径。
 - 输出文件路径: 用户指定压缩后的文件保存路径。如果用户未提供, 程序会自动在源文件名后添加 .hzip 扩展名作为默认输出路径。
 - 压缩过程: 频率统计: 程序首先读取源文件的前4KB数据, 统计每个字符出现的频率, 以构建频率表。
 - 构建霍夫曼树: 根据字符频率表, 构建霍夫曼树, 用于生成每个字符的二进制编码。
 - 编码生成: 遍历霍夫曼树, 生成每个字符对应的霍夫曼编码表。
 - 写入压缩文件: 文件头写入标识符 "HZip", 用于验证文件格式。
 - 写入字符频率表, 以便在解压时重建霍夫曼树。
 - 将源文件的内容按照霍夫曼编码进行压缩, 并写入输出文件。
- ②文件解压 (Extract)
 - 输入文件路径: 用户指定需要解压的压缩文件路径 (应为 .hzip 文件)。
 - 输出文件路径: 用户指定解压后的文件保存路径。如果用户未提供, 程序会自动在压缩文件名后添加 .ext 扩展名作为默认输出路径。
 - 解压过程: 文件验证: 读取并验证压缩文件的标识符是否为 "HZip", 以确保文件格式正确。
 - 读取频率表: 从压缩文件中读取字符频率表, 用于重建霍夫曼树。
 - 重建霍夫曼树: 根据读取的频率表, 重建霍夫曼树, 以便解码压缩数据。
 - 解码数据: 读取压缩文件中的二进制数据, 按照霍夫曼编码进行解码, 恢复原始文件内容, 并写入输出文件。
- ③显示帮助信息 (Help)
 - 当用户选择显示帮助时, 程序会输出使用说明, 详细介绍各个选项的功能和使用方法, 包括如何压缩文件、解压文件、查看帮助信息以及退出程序。
- 退出程序 (Exit)
 - 用户选择退出选项后, 程序会终止运行。
- ④用户界面: 提供交互式菜单。

2 概要设计

2.1 问题解决思路

压缩过程实现

压缩过程涉及多个步骤，确保数据的有效压缩和正确写入输出文件：

读取输入文件并统计频率：

读取输入文件的前4KB数据，统计每个字符出现的频率，生成频率表 `freq[256]`。

频率表用于构建霍夫曼树，反映字符在文件中的分布情况。

构建霍夫曼树：

使用优先队列，根据字符频率构建霍夫曼树。

频率最低的两个节点合并为一个新的父节点，重复此过程直到只剩下一个根节点。

生成编码表：

遍历霍夫曼树，为每个叶子节点（即实际字符）生成对应的二进制编码。

通过递归遍历树，记录从根节点到叶子节点的路径（'0' 表示左子节点，'1' 表示右子节点），生成编码表。

写入压缩文件：

写入文件头标识符 "HZip"，用于后续解压时验证文件格式。

写入字符频率表，以便在解压时重建霍夫曼树。

读取输入文件的内容，使用编码表将每个字符替换为对应的二进制编码，写入输出文件。

使用缓冲区管理位数据，确保高效的写入操作。

处理剩余位：

在压缩完成后，清空缓冲区，将剩余的位数信息写入文件头，确保解压时能够正确还原最后的字节。

解压过程实现

解压过程需要准确地还原压缩前的文件内容，具体步骤如下：

验证文件格式：

读取并验证文件头标识符是否为 "HZip"，确保文件为有效的 Hzip 压缩文件。

读取字符频率表：

从压缩文件中读取字符频率表 `freq[256]`，用于重建霍夫曼树。

重建霍夫曼树：

根据读取的频率表，使用与压缩时相同的优先队列和霍夫曼树构建方法，重建霍夫曼树。

确保霍夫曼树的结构与压缩时一致，以便正确解码。

解码压缩数据：

读取压缩文件中的二进制数据，通过霍夫曼树进行逐位解码。

处理剩余位：

根据文件头中的剩余位数信息，正确处理最后一个字节中的有效位，确保数据的完整性。

2.2 数据结构类型定义

霍夫曼树节点 (node)：

包含左右子节点指针、字符频率、字符值等信息。

用于构建和遍历霍夫曼树。

优先队列 (priority_queue)：

用于构建霍夫曼树，按照字符频率进行排序。

自定义比较器 `comp`，确保频率低的节点优先出队。

文件结构体 (fileW 和 fileR)：

`fileW` 用于处理输出文件，包括文件指针、编码表、缓冲区等。

`fileR` 用于处理输入文件，包括文件指针、编码表、剩余位数等。

编码表 (string encoding[256])：

存储每个字符对应的霍夫曼编码，便于压缩和解压过程中的快速查找

2.3 主程序的流程

程序启动与初始化：

程序启动后，初始化所需的变量，包括用户选择和文件路径存储变量。

显示主菜单并等待用户输入：

程序进入主循环，显示压缩工具的主菜单，提示用户选择操作。

读取并验证用户输入：

使用 `scanf` 读取用户的选择，并验证输入是否为有效的数字。

如果输入无效，清除输入缓冲区，并提示用户重新输入。

根据用户选择执行相应操作：

选择1（压缩文件）：提示用户输入源文件路径和输出文件路径。

若用户未提供输出路径，自动生成默认输出路径。

调用 `compress` 函数执行压缩操作，并显示结果。

选择2（解压文件）：提示用户输入压缩文件路径和输出文件路径。

若用户未提供输出路径，自动生成默认输出路径。

调用 `extract` 函数执行解压操作，并显示结果。

选择3（显示帮助）：显示程序的使用说明，指导用户如何使用各项功能。

选择4（退出程序）：显示退出提示信息，并跳出主循环，结束程序。

无效选择：显示无效选择的提示信息，返回主菜单等待下一次输入。

重复显示主菜单：

操作完成后，程序返回主菜单，等待用户的下一次选择，直到用户选择退出。

程序结束：

用户选择退出后，程序执行结束，返回操作系统。

3 详细设计

3.1 伪代码的设计

压缩函数

函数 `Compress`(输入: `inputFileName`, 输出: `outputFileName`):

打开 `inputFileName` 以二进制读取, 存储在 `fin`

如果 `fin` 打开失败:

打印 "无法打开输入文件 " + `inputFileName`

返回

打开 `outputFileName` 以二进制写入, 存储在 `fout`

如果 `fout` 打开失败:

打印 "无法创建输出文件 " + `outputFileName`

关闭 `fin`

返回

初始化 `freq[256]` 为 0

分配 `buf` 为 4KB 大小

从 `fin` 读取 4KB 数据到 `buf`, 存储读取的字节数到 `count`

对每个字节 `i` 从 0 到 `count - 1`:

如果 `freq[buf[i]] < 255`:

`freq[buf[i]] += 1`

释放 `buf`

将 `fin` 的文件指针回到文件开始

调用 函数 `getHuffmanTree(freq)` 得到 `hTree`

初始化 `enc[256]` 为字符串数组

初始化 `state` 为空的字符向量

调用 函数 `getEncoding(enc, hTree, state)`

写入 "HZip" 到 `fout`

跳过 `fout` 的下一个字节 (用于存储剩余位数)

写入 `freq[256]` 到 `fout`

写入 '\0' 到 `fout`

分配 `buf` 为 1KB 大小

循环直到 `fin` 到达文件末尾:

从 `fin` 读取 1KB 数据到 `buf`, 存储读取的字节数到 `count`

对每个字节 `i` 从 0 到 `count - 1`:

调用 函数 `fileWrite(fout, enc[buf[i]])`

调用 函数 `flushW(fout)` 得到 `remainder`

将 `remainder` 写入 `fout` 的第5个字节

调用 函数 `deleteHuffmanTree(hTree)`

释放 `buf`

关闭 `fin` 和 `fout`

解压函数

函数 Extract(输入: inputFileName, 输出: outputFileName):

打开 inputFileName 以二进制读取, 存储在 fin

如果 fin 打开失败:

打印 "无法打开输入文件 " + inputFileName

返回

打开 outputFileName 以二进制写入, 存储在 fout

如果 fout 打开失败:

打印 "无法创建输出文件 " + outputFileName

关闭 fin

返回

读取前5个字节到 verify[5]

如果 verify 前4个字节不等于 "HZip":

打印 "错误的文件格式! 不是 Hzip 文件。"

关闭 fin 和 fout

返回

设置 remainder = verify[4]

如果 remainder == 0:

设置 remainder = 8

读取 freq[256] 从 fin

如果 读取的字节数小于256:

打印 "Hzip 文件已损坏!"

关闭 fin 和 fout

返回

调用 函数 getHuffmanTree(freq) 得到 hTree

分配 buf 为 1KB 大小

设置 cur = hTree

循环直到 fin 到达文件末尾:

从 fin 读取 1KB 数据到 buf, 存储读取的字节数到 count

设置 maxSize = 8

对每个字节 i 从 0 到 count - 2:

temp = buf[i]

循环 j 从 0 到 7:

将 (temp & 1) 转换为字符 '0' 或 '1' 并加入队列 q

temp >>= 1

设置 maxSize = remainder 如果 fin 到达文件末尾, 否则 8

temp = buf[count - 1]

循环 j 从 0 到 maxSize - 1:

将 (temp & 1) 转换为字符 '0' 或 '1' 并加入队列 q

temp >>= 1

循环直到队列 q 为空:

temp = q.front()

如果 temp == '0':

设置 cur = cur->left

否则:

设置 cur = cur->right

如果 cur == NULL:

打印 "Hzip 文件已损坏!"

返回


```
    如果 cur 是叶子节点:
        写入 cur->val 到 fout
        设置 cur = hTree

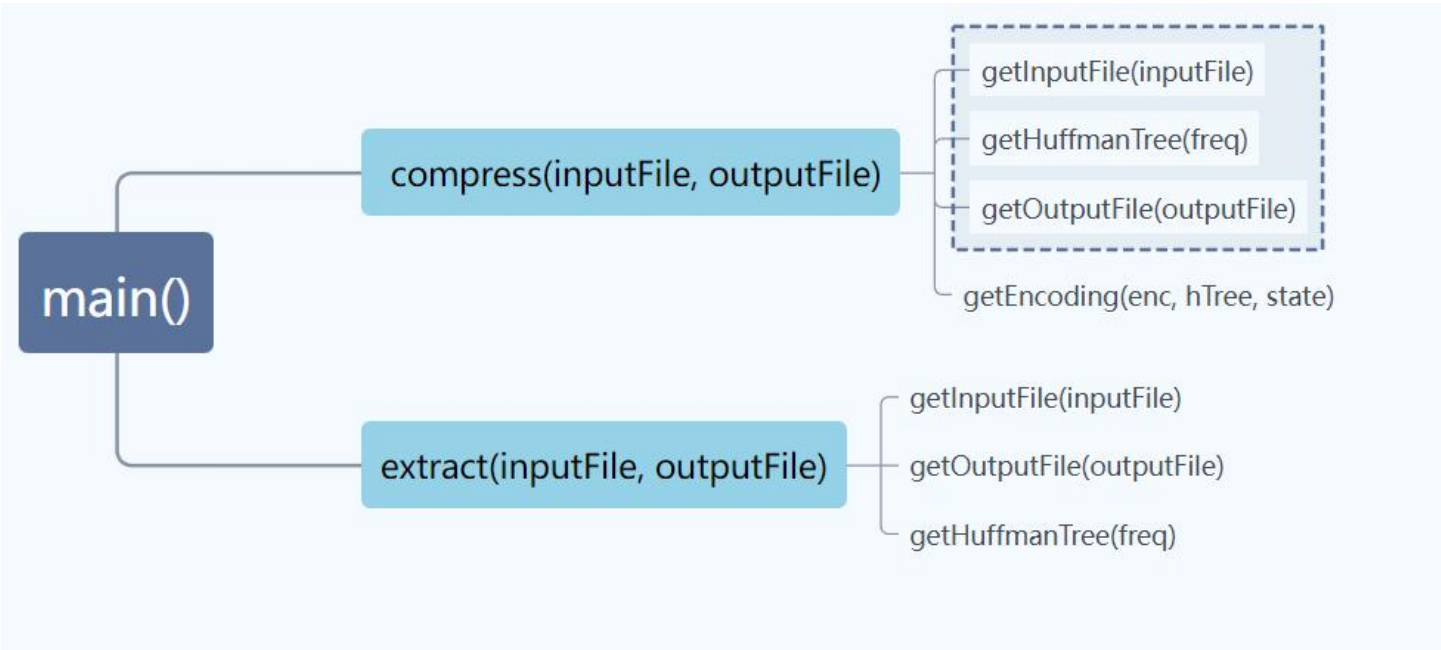
    从队列 q 中移除第一个元素

    如果 cur != hTree:
        打印 "Hzip 文件已损坏! 但仍可提取。"

    调用 函数 deleteHuffmanTree(hTree)

    释放 buf
    关闭 fin 和 fout
    // 输出连接线 (斜杠和反斜杠)
    对于 i 从 head 到 tail:
        如果 queue[i]->leftChild 存在 则:
            当 currentPos < queue[i]->leftChild->position + 2 时:
                输出空格
                currentPos += 1
            当 currentPos < queue[i]->position - 1 时:
                输出 '_'
                currentPos += 1
            输出 '/'
            currentPos += 1
        如果 queue[i]->rightChild 存在 则:
            当 currentPos < queue[i]->position + 1 时:
                输出空格
                currentPos += 1
            输出 '\\'
            currentPos += 1
            当 currentPos < queue[i]->rightChild->position - 1 时:
                输出 '_'
                currentPos += 1
    设置 end = tail
    设置 currentPos = 1
    // 输出下一层的连接斜杠和反斜杠, 并将子节点入队
    对于 i 从 head 到 end:
        如果 queue[i]->leftChild 存在 则:
            当 currentPos < queue[i]->leftChild->position + 1 时:
                输出空格
                currentPos += 1
            输出 '/'
            currentPos += 1
        如果 queue[i]->rightChild 存在 则:
            当 currentPos < queue[i]->rightChild->position - 1 时:
                输出空格
                currentPos += 1
            输出 '\\'
            currentPos += 1
        如果 queue[i]->leftChild 存在 则:
            tail += 1
            queue[tail] = queue[i]->leftChild
        如果 queue[i]->rightChild 存在 则:
            tail += 1
            queue[tail] = queue[i]->rightChild
    head += 1
    输出换行符
```

3.2 函数的调用关系图



4 调试分析报告

4.1 调试过程中遇到的问题和解决方法

- 文件操作错误处理不足
问题描述：在 `compress` 和 `extract` 函数中，虽然对文件打开失败进行了检查，但在一些情况下，文件关闭和内存释放操作可能未被执行，导致资源泄漏。
- 霍夫曼编码生成错误
问题描述：在 `getEncoding` 函数中，编码生成过程中对叶子节点的判断可能存在逻辑错误，导致某些字符的编码未正确生成或覆盖。
- 缓冲区管理不当
问题描述：在 `fileWrite` 和 `flushW` 函数中，缓冲区的位操作和剩余位数处理可能存在错误，尤其是在处理最后一个字节时。

4.2 设计实现的回顾讨论

首先，在需求分析阶段，我们明确了程序的核心功能，包括文件压缩、解压、用户交互界面以及错误处理机制。这确保了开发过程中各项功能的有序推进，并为后续的设计提供了明确的方向。

其次，算法的选择至关重要。霍夫曼编码因其无损压缩特性和较高的压缩效率成为首选。通过统计字符频率，构建霍夫曼树，并生成对应的二进制编码，实现了对高频字符的高效编码，从而显著减少了文件大小。

在数据结构设计方面，我们设计了霍夫曼树节点结构、优先队列及文件操作的辅助结构体。这些数据结构不仅支持高效的霍夫曼树构建与遍历，还优化了文件的读写操作，提升了整体性能。

功能模块的划分，如压缩、解压、编码生成及用户交互等，采用模块化设计，提高了代码的可维护性和可扩展性。通过将不同功能封装在独立的函数中，减少了模块间的耦合，增强了程序的灵活性。

在实现过程中，遇到了诸如缓冲区管理不当、内存泄漏及编码逻辑错误等问题。通过系统化的调试与测试，逐步识别并修复了这些问题，确保程序的稳定性和可靠性。

4.3 算法复杂度分析

时间复杂度: $O(m)$, 线性依赖于输入文件的大小。

空间复杂度: $O(n) + O(1) = O(n)$, 其中 $n = 256$ (频率表和霍夫曼树的空间需求), 即常数空间。

5 用户使用说明

1. 使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main.c
```

- ## 2. 执行可执行文件:

在 Linux 或 macOS 环境下: `./main`

- 在 Windows cmd 环境下: *main*

- ### 3. 输入数据和输出参照1.2/1.3部分

6 测试结果

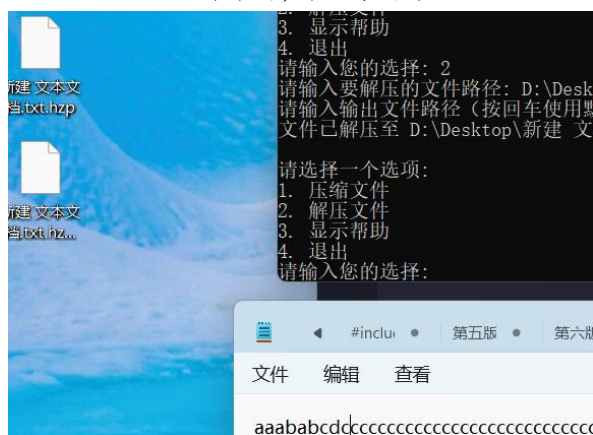
测试部分划分为如下环节。

6.1 测试1--基础样例测试

进行1.4 节的样例测试。

6.2测试2--非法输入和边界条件

测试非法输入和边界条件,即长字符串。



6.2 测试3—利用输出语句与人力手动计算结果校对

与人力手动计算结果校对, 完全一致

7 个人总结

首先，不得不说这次实验挺难的，尤其赶在期中考试期间，完成下来感觉收获满满，第二个实验，一度把我击溃，最后还是战胜了困难，还挺感慨！

8致谢

最后，感谢我的老师和助教，你们辛苦啦！