数据结构实验报告

姓名	李雨轩
学号	2204112913
班级	计算机2205

表1

实验1：背包问题的求解

一、题目描述

假设有一个能装入总体积为T的背包和n件体积分别为w1,w2,…wn的物品，能否从n件物品中挑选若干件恰好装满背包，即使w1+w2+…+wm=T，要求找出所有满足上述条件的解。例如：当T=10，各件物品的体积{1,8,4,3,5,2}时，可找到下列4组解：

(1,4,3,2)， (1,4,5)， (8,2)， (3,5,2)

二、解题思想

背包问题是一种经典的组合优化问题，在许多实际场景中都有应用，如资源分配、排班等。解决背包问题的方法有很多种，其中递归法是一种直观且易于理解的方式。

1. **构建二叉树表示选择过程：** 我们通过构建一个二叉树来表示在解空间中的选择过程。树的每个节点代表在决策过程中的一个阶段，其左子节点表示选择当前物品，右子节点表示不选择当前物品。根节点表示背包容量为0，叶子节点表示最终解。
2. **递归遍历解空间：** 通过递归方式遍历这个二叉树，从根节点开始，根据每个节点的选择，向左或向右移动。递归过程中，我们记录每个节点的选择总和和层级信息。
3. **输出符合条件的解：** 当递归到叶子节点时，检查选择总和是否等于目标背包容量T。如果等于，表示找到了一组满足条件的解，输出对应的物品组合。

通过这种递归的方式，我们穷举了所有可能的物品组合，找到了所有满足条件的解。这种解题思想虽然直观，但在实际应用中，随着物品数量的增加，其时间复杂度会呈指数级增长，因此需要进一步考虑性能优化的方法，如动态规划等。

下面是进一步的解题思想：

1. **选择树节点的创建：**

   在 addNode 函数中，对于每个父节点，创建了两个子节点，分别表示选择该物品和不选择该物品。这样的设计保证了树的每个节点都有两个分支，分别对应两种选择。

2. **递归遍历过程中的剪枝：**

   在makeChooseTree函数中，对每个节点进行了以下判断：

   - 如果左子节点的 _choose_sum 小于目标容量 target，则可以选择继续遍历左子树；
   - 如果左子节点的 _choose_sum 等于目标容量 target，则找到一组解，输出解，并终止对左子树的遍历。
   - 同样的判断逻辑也应用于右子节点。

   这种判断逻辑实现了剪枝的思想，当发现某个节点的 _choose_sum 已经大于目标容量时，就可以剪掉该节点及其子树，因为在该节点及其子树中不可能找到满足条件的解。

3. **剪枝过程的优化：**

   剪枝的优化体现在 makeChooseTree 函数中的判断逻辑，避免了对整个子树的遍历，提前终止了对某一方向的搜索。这样可以有效减少搜索空间，提高算法效率。

4. **路径记录：**

   在 makeChooseTree 函数中，通过 path 数组记录了当前路径，用于输出满足条件的解。这也是一种剪枝思路，通过路径的记录，可以在满足条件时输出解，而不必等到叶子节点。

三、大致的算法

（一）数据结构定义

```
typedef struct node {
    struct node *_lchild;
    struct node *_rchild;
    int _choose_sum;
    int _level;
} choose_tree_node, *ptr_node;
```

这部分定义了表示二叉树节点的结构体，包括左右子节点、选择总和和层级信息。通过这个结构体，构建了一个二叉树，用于解决背包问题。

（二）二叉树的构建

```
ptr_node createNode(int level);
void addNode(ptr_node parent_node, int level);
```

这两个函数用于创建节点和添加子节点。通过递归的方式，不断地创建和添加节点，构建了表示所有可能解的二叉树。

（三）背包问题的求解

```
void makeChooseTree(ptr_node root, int *list, int target, bool
*path, int start, FILE *output, int *output_list);
```

这个函数通过递归地遍历二叉树，从根节点开始，根据每个节点的选择，向左或向右移动。当遍历到叶子节点时，检查选择总和是否等于目标背包容量T，如果等于，则输出对应的物品组合。

（四）输入输出处理

```
int getData(int **data, FILE *input_file);
char *getAbsolutePath(const char *filename);
void print_help();
```

这些函数处理输入输出，包括从文件或标准输入中读取数据，获取文件的绝对路径，以及输出帮助信息。

（五）主函数

```
int main(int argc, char const *argv[]);
```

主函数处理命令行参数，根据参数配置输入和输出流，获取背包容量T，并调用makeChooseTree函数解决背包问题。同时，提供了帮助信息，以及对输入参数的错误处理。

四、输入输出

（一）对输入输出的处理

1. 输入信息

1. 命令行参数：

    - 如果命令行参数为1个，程序将从标准输入读取数据，并将结果输出到标准输出。
    - 如果命令行参数为4个，前两个参数分别为输入文件和输出文件，第三个参数为背包容量，第四个参数为背包容量值。
    - 如果命令行参数为2个且第一个参数为"--help"，则打印帮助信息。
2. 输入文件格式：
    - 如果指定了输入文件，该文件应包含一行整数，表示物品的体积。各个数字之间用空格或换行符分隔。
    - 如果没有指定输入文件，则程序会从标准输入中读取数据。

2. 输出信息

1. 输出文件格式：
    - 输出文件中包含了所有满足背包容量条件的解。
    - 每个解以 "Solution X:" 开头，其中 X 表示解的编号。
    - 每个解的具体选择以 "take" 或 "do not take" 表示，后面跟着对应物品的体积。
2. 标准输出：
    - 如果没有指定输出文件，结果将被输出到标准输出。

3. 代码中的关键输入输出处理函数

- `getData(int **data, FILE *input_file)`：从文件或标准输入中读取数据。
- `getAbsolutePath(const char *filename)`：获取文件的绝对路径。
- `print_help()`：输出帮助信息。
- `main(int argc, char const *argv[])`：处理命令行参数，配置输入和输出流，获取背包容量，调用解决背包问题的函数，并输出结果。

（二）输入输出样例

1. 输入样例1

```
./main input.txt output.txt 10
```

其中input.txt文件中的内容是：

```
1 8 4 3 5 2
```

2. 输出样例1

```
PACKAGE PEOBLEM solution (Package volume: 10):
Solution 1: take 1 ,do not take 8 ,take 4 ,take 3 ,do not take 5 ,take 2 ,
Solution 2: take 1 ,do not take 8 ,take 4 ,do not take 3 ,take 5 ,do not take 2 ,
Solution 3: do not take 1 ,take 8 ,do not take 4 ,do not take 3 ,do not take 5 ,take 2 ,
Solution 4: do not take 1 ,do not take 8 ,do not take 4 ,take 3 ,take 5 ,take 2 ,
```

3. 输入样例 2

```
./main input.txt output.txt 15
```

其中，input.txt 文件内容如下：

```
1 2 3 4 5 6 7 8
```

4. 输出样例 2

```
PACKAGE PEOBLEM solution (Package volume: 15):
Solution 1: take 1 ,take 2 ,take 3 ,take 4 ,take 5 ,do not take 6 ,
Solution 2: take 1 ,take 2 ,do not take 3 ,take 4 ,do not take 5 ,do not take 6 ,
Solution 3: take 1 ,take 2 ,do not take 3 ,do not take 4 ,take 5 ,do not take 6 ,
Solution 4: take 1 ,do not take 2 ,take 3 ,take 4 ,do not take 5 ,do not take 6 ,
Solution 5: take 1 ,do not take 2 ,take 3 ,do not take 4 ,take 5 ,take 6 ,
Solution 6: take 1 ,do not take 2 ,do not take 3 ,do not take 4 ,do not take 5 ,take 6 ,
Solution 7: do not take 1 ,take 2 ,take 3 ,take 4 ,do not take 5 ,take 6 ,
Solution 8: do not take 1 ,take 2 ,do not take 3 ,do not take 4 ,take 5 ,do not take 6 ,

Solution 9: do not take 1 ,take 2 ,do not take 3 ,do not take 4 ,do
```

```
not take 5 ,take 6 ,
Solution 10: do not take 1 ,do not take 2 ,take 3 ,take 4 ,do not take 5 ,do not take 6 ,
Solution 11: do not take 1 ,do not take 2 ,take 3 ,do not take 4 ,take 5 ,do not take 6 ,
Solution 12: do not take 1 ,do not take 2 ,do not take 3 ,take 4 ,take 5 ,take 6 ,
Solution 13: do not take 1 ,do not take 2 ,do not take 3 ,do not take 4 ,do not take 5 ,do not take 6 ,
```

5. 输入样例 3

```
./PackageProblem --help
```

6. 输出样例 3

```
Usage: main [input_filename] [output_filename]
Reads data from input_filename and writes the result to output_filename.
If all arguments are omitted, the program reads from standard input and writes to standard output.
```

五、总结：分析时间空间复杂度，看看有没有改进的地方

（一）时间复杂度分析：

1. makeChooseTree 函数：递归深度，在树的最坏情况下，每个节点都可能有两个子节点，递归深度为物品数量 n，因此最坏情况下时间复杂度为 $O(2^n)$。在每一步递归中，需要进行一定的比较和输出操作，这些操作的时间复杂度是常量级别，对总体复杂度的影响相对较小。
2. getData 函数：读取输入数据的时间复杂度为 $O(n)$，其中 n 为输入数据的数量。
3. main 函数：主要操作是调用 makeChooseTree 函数，因此主要的时间复杂度取决于 makeChooseTree 的时间复杂度，为 $O(2^n)$。

（二）空间复杂度分析：

1. makeChooseTree 函数：递归调用会占用栈空间，最坏情况下栈的深度是物品数量 n，因此空间复杂度为 $O(n)$。除递归栈外，额外使用的空间主要是 path 数组，其长度为物品数量 n，因此额外的空间复杂度为 $O(n)$。

2. `getData` 函数：为存储输入数据动态分配了一个数组，其大小为物品数量 n，因此空间复杂度为 $O(n)$。
3. `main` 函数：除了调用 `makeChooseTree` 函数外，还分配了额外的空间用于存储路径信息和输入数据，因此额外的空间复杂度为$O(n)$。

（三）总结及改进的地方：

1. 该算法的时间复杂度受到物品数量的指数级增长的影响，因此在处理大规模数据时，性能可能不理想。
2. 空间复杂度主要受到递归栈和额外数组的影响，但总体上是较为合理的。
3. 改进的地方：可以考虑使用动态规划等更高效的算法来解决背包问题，以提高算法的效率。此外，递归深度的指数增长表明存在许多重复的子问题，可以通过记忆化搜索等方法进行优化。

实验2：农夫过河问题的求解

一、题目描述

一个农夫带着一只狼、一只羊和一棵白菜，身处河的南岸。他要把这些东西全部运到北岸。他面前只有一条小船，船只能容下他和一件物品，另外只有农夫才能撑船。如果农夫在场，则狼不能吃羊，羊不能吃白菜，否则狼会吃羊，羊会吃白菜，所以农夫不能留下羊和白菜自己离开，也不能留下狼和羊自己离开，而狼不吃白菜。请求出农夫将所有的东西运过河的方案。

二、解题思想

本题主要通过深度优先搜索（DFS）算法来穷举所有可能的过河策略，并使用剪枝优化算法来减少搜索空间。

1. 状态表示：

   河边问题中，每个状态可以表示为一个4位的二进制数，其中每一位对应一个物体（农夫、狼、羊、白菜），0表示在南岸，1表示在北岸。例如，状态 `0b1010` 表示农夫和羊在北岸，狼和白菜在南岸。

2. DFS遍历：

   通过DFS遍历所有可能的状态，从初始状态 `0b0000` 出发，目标是达到状态 `0b1111`，即所有物体都在北岸。

3. 剪枝优化：

   剪枝的思想在于排除掉一些明显无解的情况，以减小搜索空间，提高效率。在这个问题中，使用了两种剪枝策略：

   a. 剪枝策略1 - `judgeExcess` 函数：- 该函数用于判断在某个状态下是否存在明显的冲突，使得该状态不可能是解。- 对于每一对物体，如果它们在河边相邻，且它们的目标状态存在冲突，就认为这个状态不可能是解，进行剪枝。

   b. 剪枝策略2 - DFS 函数中的 `visited` 数组：- 使用一个 `visited` 数组记录已经访问过的状态，如果某个状态已经被访问过，就不再对其进行DFS，减少重复搜索。

4. 路径输出：

   通过 `printPath` 函数，回溯输出找到的路径，其中 `binnumToStatus` 函数用于将二进制状态转换为人类可读的状态。

5. 输出所有解：

每找到一条可行的过河策略，就输出一次，最终输出所有可能的过河策略。

三、大致的算法

在这个问题中，每个状态可以用一个4位二进制数表示，分别代表农夫、狼、羊、白菜的位置，0表示在河的一岸，1表示在河的对岸。例如，初始状态为`0000`，表示所有东西都在河的一岸。

以下是算法的实现步骤：

（一）数据结构定义

1. `GraphAdjMat` 结构体和 `Graph` 指针：表示图的邻接矩阵，其中 `vertices` 数组存储顶点，`adjMat` 存储边的连接关系，`size` 记录图的大小。
2. `Queue` 结构体：用于实现队列数据结构，支持入队、出队等操作。

（二）算法实现

1. 创建图（`Graph`）：使用邻接矩阵表示图，其中每个节点表示一个状态，每个状态之间的边表示可以通过一步操作从一个状态转移到另一个状态。
2. 深度优先搜索：使用DFS算法，从初始状态开始，递归地尝试所有可能的操作序列，直到找到一条到达目标状态的路径。
3. 递归过程中，使用剪枝策略防止搜索无效路径。
    1. 使用 `judgeExcess` 函数判断是否有潜在的吃食问题，如果有，剪掉该分支。
    2. 在 DFS 递归中，记录已经访问的状态，避免重复访问。
4. 过 `judgeExcess` 函数对搜索空间进行剪枝，避免搜索到不符合条件的状态。

四、输入输出

（一）对输入输出的处理

对输入输出的处理与实验1中类似：

1. 输入处理

1. 命令行参数（argc、argv）：
    - 如果命令行参数个数为1，则输出到标准输出(stdout)。
    - 如果命令行参数个数为2，且第一个参数不是"--help"，则将输出重定向到指定文件。
2. 创建图（Graph）：
    - 使用 `GraphAdjMat` 结构体表示图，通过邻接矩阵 `adjMat` 存储节点之间的连接关系。
    - 通过 `addVertex` 和 `addEdge` 函数添加顶点和边。

2. 输出处理

1. DFS输出策略：
    - 在找到到达目标状态的路径时，输出策略。每条路径的输出包含一系列河边状态的描述。
    - 每个状态的描述使用 `binnumToStatus` 函数将二进制数转化为描述字符串，表示农夫、狼、羊、白菜在南岸或北岸的状态。
2. 文件输出：
    - 通过 `FILE *output` 控制输出，如果输出文件没有成功打开，则在标准错误流中输出错误信息。

（二）输入输出样例

1. 输入样例1

样例 1：标准输出

```
./FarmerCrossingRiver
```

- 输出：将河边问题的解答输出到标准输出。

2. 输出样例1

```
# stdout
The farmer's strategy for crossing the river 1:
*/Each line represents the status but not the Handling process/*
--------Strategy 1 begin--------
    Farmer in the South bank, Wolf in the South bank, Goat in the South bank, Cabbage in the South bank,  (0000)
--> Farmer in the North bank, Wolf in the South bank, Goat in the
```

```
North bank, Cabbage in the South bank,  (1010)
--> Farmer in the South bank, Wolf in the South bank, Goat in the North bank, Cabbage in the South bank,  (0010)
--> Farmer in the North bank, Wolf in the South bank, Goat in the North bank, Cabbage in the North bank,  (1011)
--> Farmer in the South bank, Wolf in the South bank, Goat in the South bank, Cabbage in the North bank,  (0001)
--> Farmer in the North bank, Wolf in the North bank, Goat in the South bank, Cabbage in the North bank,  (1101)
--> Farmer in the South bank, Wolf in the North bank, Goat in the South bank, Cabbage in the North bank,  (0101)
--> Farmer in the North bank, Wolf in the North bank, Goat in the North bank, Cabbage in the North bank,  (1111)
--------Strategy 1 end--------

The farmer's strategy for crossing the river 2:
*/Each line represents the status but not the Handling process/*
--------Strategy 2 begin--------
    Farmer in the South bank, Wolf in the South bank, Goat in the South bank, Cabbage in the South bank,  (0000)
--> Farmer in the North bank, Wolf in the South bank, Goat in the North bank, Cabbage in the South bank,  (1010)
--> Farmer in the South bank, Wolf in the South bank, Goat in the North bank, Cabbage in the South bank,  (0010)
--> Farmer in the North bank, Wolf in the North bank, Goat in the North bank, Cabbage in the South bank,  (1110)
--> Farmer in the South bank, Wolf in the North bank, Goat in the South bank, Cabbage in the South bank,  (0100)
--> Farmer in the North bank, Wolf in the North bank, Goat in the South bank, Cabbage in the North bank,  (1101)
--> Farmer in the South bank, Wolf in the North bank, Goat in the South bank, Cabbage in the North bank,  (0101)
--> Farmer in the North bank, Wolf in the North bank, Goat in the North bank, Cabbage in the North bank,  (1111)
--------Strategy 2 end--------
```

3. 输入样例2

样例 2：输出至文件

```
./FarmerCrossingRiver ouput.txt
```

- 输出：将河边问题的解答输出到output.txt

4. 输出样例2

略，与输出样例1一致

5. 输入样例3

```
./FarmerCrossingRiver --help
```

6. 输出样例3

```
Usage: ./FarmerCrossingRiver [output_filename]
```

五、总结

（一）时间复杂度

1. **DFS 函数**：对于每个节点，进行深度优先搜索。每个节点最多访问一次，因此时间复杂度为 $O(V+E)$，其中 V 是顶点数，E 是边数。
2. **addVertex和 addEdge函数**：对于每个节点和每条边，都进行了常数时间的操作。时间复杂度为 $O(V+E)$，其中 V 是顶点数，E 是边数。
3. **BFS函数**：对于每个节点，进行广度优先搜索。每个节点最多访问一次，因此时间复杂度为$O(V+E)$，其中 V 是顶点数，E 是边数。
4. **整体时间复杂度**：主函数中调用 DFS 函数，因此时间复杂度为 $O(V+E)$。

（二）空间复杂度

1. **GraphAdjMat结构体**：存储了顶点和邻接矩阵，占用空间为 $O(V^2)$，其中 V 是顶点数。
2. **Queue结构体**：存储了队列，占用空间为$O(MAX_QUEUE_SIZE)$。
3. **visited数组**：占用了$O(V)$的空间，其中 V 是顶点数。
4. **path数组**：占用了 $O(V)$ 的空间，其中 V 是顶点数。
5. **其他局部变量**：占用了常数空间。
6. **整体空间复杂度**：主要占用空间的是 GraphAdjMat结构体，因此整体空间复杂度为$O(V^2)$。

（三）可改进的地方

1. **图的表示方式**：当图稀疏时，邻接矩阵的存储方式可能浪费空间。可以考虑使用邻接表等更节省空间的表示方式。

2. **DFS 访问顺序**：在 DFS 函数中，可以通过改变节点的访问顺序，优化搜索速度。例如，将相邻节点按某种规则排序，以提高效率。
3. **Queue 大小**：如果图较大，可能需要动态调整队列的大小而不是固定使用 `MAX_QUEUE_SIZE`。

实验4：八皇后问题

一、题目描述

设在初始状态下在国际象棋的棋盘上没有任何棋子（这里的棋子指皇后棋子）。然后顺序在第1行，第2行……第8行上布放棋子。在每一行中共有8个可选择的位置，但在任一时刻棋盘的合法布局都必须满足3个限制条件：1）任意两个棋子不得放在同一行；2）任意两个棋子不得放在同一列上；3）任意棋子不得放在同一正斜线和反斜线上。

二、解题思想

这段代码是典型的N皇后问题的求解代码，采用了回溯法。具体来说，代码采用了枚举-剪枝的思路。

1. **枚举**：通过两个嵌套循环，在每一行中枚举皇后的放置位置（列），这是一个典型的穷举过程。
2. **剪枝**：在每个位置尝试放置皇后之前，通过检查当前位置所在的列、主对角线和副对角线是否已经有皇后，来判断是否可以放置。如果某一列、主对角线或副对角线已经有皇后，就不再尝试在该位置放置皇后，直接进行下一次枚举。剪枝的过程通过 `if (!cols[col] && !diags1[diag1] && !diags2[diag2])` 实现，其中 `cols` 数组记录每一列是否有皇后，`diags1` 和 `diags2` 数组记录两个方向的对角线上是否有皇后。
3. **回溯**：如果在某个位置放置皇后后，继续递归调用函数，处理下一行。如果发现在下一行无法找到合适位置，就会回溯到当前行，将当前位置的皇后移除，继续尝试下一个位置。
4. **记录解**：当找到一种合法的放置方式时，将当前的棋盘状态保存下来，以备最后输出所有的解。
5. **输出解**：在 `main` 函数中，通过循环遍历保存的所有解，输出每一种解的棋盘状态。

这种枚举-剪枝的思路通过递归的方式在搜索过程中排除了不符合条件的情况，从而减少了搜索的空间，提高了求解效率。由于 N 皇后问题的搜索空间较大，采用回溯法进行枚举和剪枝是一个常见的解法。

三、大致的算法

当解决 N 皇后问题时，"枚举"和"剪枝"是解决问题的两个核心步骤。

（一）枚举：

枚举是指通过循环逐一尝试所有可能的情况，即列举出问题的所有可能解。在 N 皇后问题中，我们需要枚举每一行皇后的放置位置，以找到合适的布局方案。

```
for (int col = 0; col < n; col++) {
    // 尝试在当前行的每一列放置皇后
}
```

这个循环用于枚举每一行皇后可以放置的列，相当于逐一尝试在每一列放置皇后，以找到合适的位置。

（二）剪枝：

剪枝是为了在搜索的过程中减少无效的搜索空间，即通过判断当前情况是否符合要求，如果不符合就提前终止或跳过某些分支的搜索。

在 N 皇后问题中，为了剪枝，我们需要检查当前位置是否符合皇后放置的规则。主要包括三个方面：

列冲突： 不能在同一列放置多个皇后。

```
if (!cols[col]) {
    // 当前列没有皇后，可以尝试放置
}
```

主对角线冲突： 不能在同一条主对角线上放置多个皇后。

```
int diag1 = col - row + n - 1;
if (!diags1[diag1]) {
    // 当前主对角线没有皇后，可以尝试放置
}
```

副对角线冲突： 不能在同一条副对角线上放置多个皇后。

```
int diag2 = col + row;
if (!diags2[diag2]) {
    // 当前副对角线没有皇后，可以尝试放置
}
```

如果当前位置满足以上三个条件，说明可以在该位置放置皇后。然后递归调用下一行的放置，否则直接跳过当前位置，继续尝试下一个位置。

```
if (!cols[col] && !diags1[diag1] && !diags2[diag2]) {
    // 当前位置可以放置皇后
    // ...
    backTrack(n, row + 1, cols, diags1, diags2, this_state, res_states, res_index);
    // ...
}
```

这种剪枝策略有助于提前排除不符合规则的情况，减小了搜索空间，提高了算法的效率。

四、输入输出

（一）对输入输出的处理

对输入输出的处理与实验1中类似：命令行参数（argc、argv）：

- 如果命令行参数个数为1，则输出到标准输出(stdout)。
- 如果命令行参数个数为2，且第一个参数不是"--help"，则将输出重定向到指定文件。

（二）输入输出样例

1. 输入样例1

样例 1：标准输出

```
./EightQueens
```

- 输出：将问题的解答输出到标准输出。

2. 输出样例1

```
# 省略上面的部分
--------Plan 90 end.--------

Plan 91:
   0 1 2 3 4 5 6 7
   -----------------
0 |             # |
1 |    #          |

2 |#              |
```

```
3 |          #     |
4 |  #             |
5 |        #       |
6 |            #   |
7 |      #         |
   -----------------
--------Plan 91 end.--------

Plan 92:
   0 1 2 3 4 5 6 7
   -----------------
0 |              # |
1 |      #         |
2 |#               |
3 |    #           |
4 |          #     |
5 |  #             |
6 |            #   |
7 |        #       |
   -----------------
--------Plan 92 end.--------


*******TOTAL 92 PLANS*******
```

3. 输入样例2

样例 2：输出至文件

```
./EightQueens ouput.txt
```

- 输出：将问题的解答输出到output.txt

4. 输出样例2

略，与输出样例1一致

5. 输入样例3

```
./EightQueens --help
```

6. 输出样例3

```
Usage: ./EightQueens [output_file]
If output_file is omitted, the program writes to standard output.
```

五、总结

（一）时间复杂度分析：

该算法的时间复杂度主要由回溯算法决定。在最坏情况下，需要考虑所有可能的八皇后布局，因此时间复杂度为 $O(N!)$，其中 N 为皇后的数量。

（二）空间复杂度分析：

1. **backTrack 函数**：额外使用了存储当前状态的二维布尔数组 this_state，其空间复杂度为 $O(N^2)$。使用了存储所有解的三维布尔数组 res_states，其空间复杂度为 $O(N^3)$。递归调用的深度最多为 N，因此递归调用栈的空间复杂度为$O(N)$。

   总的空间复杂度为 $O(N^3 + N^2 + N) = O(N^3)$
2. **createBoard函数**：使用了一个二维布尔数组 board，其空间复杂度为 $O(N^2)$。
3. **printBoard函数**：没有使用额外的空间，只是打印输出。
4. **eightQueens 函数**：
   - 使用了存储棋盘状态的二维布尔数组 checkerboard，其空间复杂度为 $O(N^2)$。
   - 使用了三个一维布尔数组 cols、diag1、diag2，其空间复杂度为 $O(N)$。

   总的空间复杂度为 $O(N^2 + N) = O(N^2)$

（三）可以改进的空间：

1. **使用一维数组代替二维数组**：在 backTrack 函数中，使用了二维数组 this_state 表示当前状态，可以考虑使用一维数组代替。由于每次只处理一行，可以用一个一维数组表示当前行的皇后位置。
2. **避免使用动态分配的数组**：在 backTrack 函数中，使用了动态分配的数组 res_states 用于存储所有解，但这样会导致额外的内存分配和释放。可以考虑使用栈空间或者其他方式避免动态分配。
3. **递归调用的优化**：考虑对递归调用进行优化，避免过深的递归栈。可能通过迭代或其他手段进行优化。

这些改进可以在一定程度上减小程序的空间复杂度，提高效率。

实验5：约瑟夫环问题仿真

一、题目描述

设编号为1，2，…，n(n>0)个人按顺时针方向围坐一圈，每人持有一个正整数密码。开始时任意给出一个报数上限m，从第一个人开始顺时针方向自1起顺序报数，报到m时停止报数，报m的人出列，将他的密码作为新的m值，从他在顺时针方向上的下一个人起重新自1报数；如此下去直到所有人全部出列为止。要求设计一个程序模拟此过程，给出出列人的编号序列

二、解题思路

模拟了一个环形报数的过程，通过删除节点模拟淘汰的效果，最终输出报数路径

三、大致的算法

（一）结构体定义：

```
typedef struct node {
    int _key;
    int _id;
    struct node *_next;
} node, *ptr_node;
```

- 定义了一个结构体 node，包含 _key（键值）、_id（标识符）、_next（指向下一个节点的指针）。
- 定义了结构体指针 ptr_node，用于表示指向 node 结构体的指针。

（二）初始化环形链表函数 initRing：

```
node *initRing(int *key_list, int length)
{
    if (length > 1) {
        // 创建链表，并使最后一个节点指向第一个节点，形成环
        // ...
        return begin_node;
    } else if (length == 1) {
        // 创建一个只包含一个节点的环
        // ...

        return ret_node;
```

```
    } else {
        fprintf(stderr, "no key list no ring\n");
        return NULL;
    }
}
```

- 如果键值列表长度大于1，创建包含多个节点的环形链表。
- 如果长度为1，创建一个只包含一个节点的环。
- 如果长度为0，输出错误信息并返回空指针。

（三）模拟报数函数 simulate：

```
void simulate(node *first_node, int upper, ptr_node *path, int ret_index)
{
    if (first_node->_next == first_node) {
        // 如果链表只有一个节点，直接记录该节点
        path[ret_index] = first_node;
        return;
    }
    // 从第一个节点开始，每次找到第 upper - 1 个节点，删除下一个节点，继续模拟报数
    // ...
}
```

- 如果链表只有一个节点，直接记录该节点。
- 从第一个节点开始，每次找到第 upper - 1 个节点，将下一个节点记录在路径数组中，然后删除该节点，继续模拟报数过程。

（四）主函数 main：

- 通过命令行参数或用户输入获取输入文件、输出文件、以及第一个报数的限制值。
- 调用 getData 函数读取输入文件中的键值列表。
- 调用 initRing 函数初始化环形链表。
- 调用 simulate 函数模拟报数过程，将报数路径记录在 path 数组中。
- 调用 printPath 函数将报数路径输出到文件或标准输出。

四、输入输出

（一）对输入输出的处理

1. 文件输入输出处理：

- `getData` 函数：
  - 从输入文件或标准输入中读取数据，返回数据的数量。
  - 如果文件为标准输入（`stdin`），则通过 `scanf` 读取数字，直到遇到换行符为止。
  - 如果文件非标准输入，通过 `fscanf` 从文件中读取数字，同样遇到换行符或文件结束则停止。
- `getAbsolutePath` 函数：
  - 获取文件的绝对路径，使用 `getcwd` 获取当前工作目录，然后通过字符串拼接生成文件的绝对路径。
- `printPath` 函数：
  - 将报数路径输出到文件或标准输出，使用 `fprintf` 函数。
- `main` 函数：
  - 通过命令行参数判断输入和输出的来源，并打开相应的文件流。
  - 如果命令行参数为 1，表示没有提供输入文件，将输入流设置为标准输入；如果命令行参数为 4，表示提供了输入文件和输出文件。
  - 使用 `getData` 读取输入数据，初始化环形链表，进行报数模拟，然后将结果输出到文件或标准输出。

2. **标准输入输出处理：**
   - 对于标准输入（`stdin`），程序会通过 `printf` 和 `scanf` 进行输入输出。
   - 对于文件输入输出，程序通过文件流和文件 I/O 函数（`fopen`, `fclose`, `fprintf`, `fscanf`）进行读写操作。
3. **错误处理：**
   - 在打开文件时，程序会检查文件是否成功打开，如果打开失败，会输出错误信息并退出程序。
4. **用户交互：**
   - 如果未提供命令行参数或提供了 `--help` 参数，程序会打印使用帮助信息。

（二）输入输出样例

1. 输入样例1

```
./JosephRing data.input output.txt 20
```

其中data.input的内容是：

```
1 8 4 3 5 2
```

2. 输出样例1

```
The 1 gays OUT is id=2 and key=8
The 2 gays OUT is id=5 and key=5
The 3 gays OUT is id=6 and key=2
The 4 gays OUT is id=3 and key=4
The 5 gays OUT is id=1 and key=1
The 6 gays OUT is id=4 and key=3
```

3. 输入样例2

```
./JosephRing --help
```

4. 输出样例2

```
Usage: main [input_filename] [output_filename] [key]
Reads data from input_filename and writes the result to
output_filename.
[key] is the integer as *The first reporting limit*.
If all arguments are omitted, the program reads from standard input
and writes to standard output.
```

五、总结

（一）时间复杂度分析：

1. `initRing` 函数：遍历输入数组创建环形链表，时间复杂度为 O(n)，其中 n 是数组的长度。
2. `simulate` 函数：

- 在环形链表上进行报数模拟，每次删除一个节点，直到链表为空。
- 在最坏情况下，需要删除 n 个节点，其中 n 是数组的长度。
- 每个节点的删除操作的时间复杂度是 O(1)。
- 因此，simulate 函数的总体时间复杂度为 O(n)。

3. getData 函数：

- 在标准输入或文件中读取数据，时间复杂度取决于输入数据的数量。
- 最坏情况下，需要读取 n 个数据，其中 n 是数组的长度。
- 每次读取一个数据的时间复杂度是 O(1)。
- 因此，getData 函数的总体时间复杂度为 O(n)。

（二）空间复杂度分析：

- initRing 函数创建了一个环形链表，需要的额外空间为 O(n)。
- simulate 函数使用了递归，递归深度最大为数组的长度，因此递归栈的空间复杂度为 O(n)。
- getData 函数需要存储输入数据的数组，占用 O(n) 的额外空间。
- 其他变量和数据结构占用的空间是常量级别的。
- 因此，总体空间复杂度为 O(n)。

实验7：二叉排序树与平衡二叉树的实现

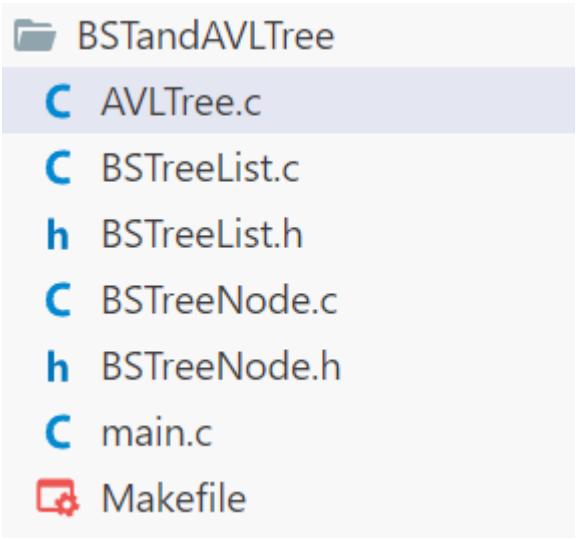
一、题目描述

分别采用二叉链表和顺序表作存储结构，实现对二叉排序树与平衡二叉树的操作

二、解题思路

模板题，参考所学知识分别涉及二叉搜索树和AVL平衡二叉树

三、大致的算法




```
main_list: main.c BSTreeList.c BSTreeList.h
    gcc -DUSE_LIST -o main_list main.c BSTreeList.c

main_node: main.c BSTreeNode.c BSTreeNode.h
    gcc -o main_node main.c BSTreeNode.c

main_avl: AVLTree.c
    gcc AVLTree.c -o main_avl
```

（一）二叉搜索树

1. 插入节点（InsertBST函数）：
    - 如果树为空，创建一个新节点。
    - 如果节点已存在，增加其计数。
    - 否则，根据键值大小递归地插入到左子树或右子树中。

2. 创建二叉排序树（CreateBST函数）：
    - 遍历数据数组，通过调用 `InsertBST` 函数插入节点，从而构建二叉排序树。
3. 中序遍历（InOrderTraverse函数）：
    - 通过递归，按中序遍历方式输出树中的节点值，即左子树、根节点、右子树。
4. 计算平均查找长度（AverageSearchPath函数）：
    - 通过递归遍历树的节点，累加每个节点的深度，最后除以节点总数得到平均查找长度。
5. 删除节点（DeleteNode函数）：
    - 根据键值递归地查找要删除的节点。
    - 如果找到，根据左右子树的情况删除节点。
        - 如果有两个子树，用左子树的最大值覆盖要删除的节点，然后递归删除左子树的最大节点。
        - 如果只有一个子树或没有子树，直接删除节点。
6. 查找键值（FindKey函数）：
    - 通过递归查找树中是否存在给定键值。

（二）AVL平衡树

AVL树（Adelson-Velsky and Landis树）的实现，它是一种自平衡二叉搜索树。AVL树确保了树的左右子树的高度差不超过1，通过旋转操作来维护平衡。

以下是代码的主要部分和关键函数：

1. AVL树节点定义：

```
typedef struct node {
    int _val;
    struct node *_lchild;
    struct node *_rchild;
    int _height;
} avlnode, *avl;
```

每个节点包含一个值、左右子节点指针和节点高度。

2. LL_rotation和RR_rotation：

```
avl LL_rotation(avl root);
avl RR_rotation(avl root);
```

分别执行左子树的左旋和右子树的右旋，用于调整失衡的节点。

3. insertNode：

```
avl insertNode(avl root, int val);
```

插入节点并通过旋转操作保持AVL树的平衡。

4. AverageSearchPath：

```
double AverageSearchPath(avl T, int n);
```

计算AVL树的平均查找路径长度。

5. _AverageSearchPath：

```
double _AverageSearchPath(avl T, int level);
```

递归计算每个节点的查找路径长度。

6. getData：

```
int getData(int **data, FILE *input_file);
```

从输入文件或标准输入中获取数据，存储在数组中。

7. layerOrderPut：

```
void layerOrderPut(avl root);
```

层序遍历输出AVL树的节点值。

8. main函数：处理命令行参数，调用相应的函数，并输出结果。

解决思路和算法核心：

- 插入节点时，通过递归保持AVL树的平衡，检查并执行旋转操作。
- 计算平均查找路径长度时，使用递归遍历树，计算每个节点的深度，并最终计算平均深度。

实现：

- 使用C语言实现，通过结构体和指针来表示AVL树节点。
- 通过LL和RR旋转操作来维护树的平衡。

四、输入输出

（一）对输入输出的处理

基本运行逻辑与之前的代码类似，运行可执行程序来测试这个AVL树的实现，例如：

```
./main_avl input.txt output.txt
```

其中，`input.txt` 包含要插入的数据，`output.txt` 用于输出结果。

（二）输入输出样例

1. 输入输出样例1

```
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/BSTandAVLTree$ make main_avl
gcc AVLTree.c -o main_avl
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/BSTandAVLTree$ ./main_avl
-------------QUESTION 1-------------
Please input array with enter as END:
12 23 43 1 4 225 123 988 73

-------------QUESTION 2-------------
Average Search Path: 3.000000

----------------END.----------------
```

2. 输入输出样例2

```
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/BSTandAVLTree$ make main_node
gcc -o main_node main.c BSTreeNode.c
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/BSTandAVLTree$ ./main_node
-------------QUESTION 1-------------
Please input array with enter as END:

12 34 23 4 9 8 7 6 44 10 93 2
```

```
7
8
9  ------------QUESTION 2------------
10 InOrder Traverse:
11 2 4 6 7 8 9 10 12 23 34 44 93
12
13 ------------QUESTION 3------------
14 Average Search Path3.333333
15
16 *
17 Enter the key that you what to delete:
18 2
19 *
20
21 ------------QUESTION 4------------
22 The key you choose to delete is 2.
23 After delete 2, Inorder Traverse:
24 4 6 7 8 9 10 12 23 34 44 93
25
26 ----------------END.----------------
```

3. 输入输出样例3

```
1  rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/BSTandAVLTree$
   make main_list
2  gcc -DUSE_LIST -o main_list main.c BSTreeList.c
3  rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/BSTandAVLTree$
   ./main_list
4  ------------QUESTION 1------------
5  Please input array with enter as END:
6  9 8 71367 5613 83 74 7198 2387 789 7 89 67 56 45 78 687 578 756
7
8
9  ------------QUESTION 2------------
10 InOrder Traverse:
11 7 8 9 45 56 67 74 78 83 89 578 687 756 789 2387 5613 7198 71367
12
13 ------------QUESTION 3------------
14 Average Search Path5.277778
15
16 *
17 Enter the key that you what to delete:
18 9
19 *
20

21 ------------QUESTION 4------------
```

```
The key you choose to delete is 9.
After delete 9, Inorder Traverse:
7 8 45 56 67 74 78 83 89 578 687 756 789 2387 5613 7198 71367

----------------END.----------------
```

五、总结

（一）1. BST（Binary Search Tree）

时间复杂度：

- **插入操作：** $O(\log n)$ - 在平均情况下，插入节点的时间复杂度是对数级别，取决于树的高度。
- **查找操作：** $O(\log n)$ - 平均情况下，查找节点的时间复杂度是对数级别。
- **删除操作：** $O(\log n)$ - 平均情况下，删除节点的时间复杂度是对数级别。

空间复杂度：

- $O(n)$ - 每个节点需要存储值和两个指针（左子节点和右子节点）。

（二）AVL树：

时间复杂度：

- **插入操作：** $O(\log n)$ - 由于AVL树的自平衡性质，插入时需要进行旋转操作，但旋转操作是常数时间的。
- **查找操作：** $O(\log n)$ - 平均情况下，查找节点的时间复杂度是对数级别。
- **删除操作：** $O(\log n)$ - 删除操作也可能触发旋转，但是旋转操作是常数时间的。

空间复杂度：

- $O(n)$ - 每个节点需要存储值和额外的平衡信息（高度等）。

实验10：一元稀疏多项式计算器

一、题目描述

设计一个一元稀疏多项式简单计算器。要求这个一元稀疏多项式简单计算器的基本功能是：

(1)输入并建立多项式；

(2)输出多项式，输出形式为整数序列

n,c1,e1,c2,e2,…,cn,en,

其中n是多项式的项数，ci,ei分别是第i项的系数和指数，序列按指数降序排列；

(3)多项式a和b相加，建立多项式a+b；

(4)多项式a和b相减，建立多项式a-b；

(5)计算多项式在x处的值；即给定x值，计算多项式值。

二、解题思路

1. **定义多项式数据结构：** 创建一个数据结构，用于表示一元稀疏多项式。这个数据结构应该包括项数信息以及每一项的系数和指数。这可以是一个抽象的容器，例如列表、集合或链表。
2. **输入多项式：** 设计一个输入功能，允许用户输入多项式的信息。这可能包括项数，以及每一项的系数和指数。确保用户输入的信息能够被正确地映射到多项式的数据结构中。
3. **输出多项式：** 提供一个输出功能，将多项式的信息以规定的格式呈现给用户。通常，这包括将每一项按照指数降序排列，并以易于理解的形式展示。
4. **多项式运算：** 实现多项式的基本运算，如相加和相减。这涉及遍历多项式的项并执行相应的运算操作。确保处理好边界条件和运算规则。
5. **计算多项式值：** 提供功能，允许用户计算多项式在给定值（例如 x）处的结果。这涉及遍历多项式的项，并将每一项的值乘以对应的 x 次幂，最后求和得到结果。

三、大致的算法

代码实现了一个多项式类 `Polynomial`，使用链表来表示多项式的每一项。以下是对代码主要部分的分析：

1. 链表表示多项式： Node 类包含了一个项的系数 _coef、指数 _exp 和指向下一个节点的指针 _next。Polynomial 类通过 _head 成员表示多项式的头节点，初始时有一个虚拟节点。这种设计方便了插入、删除和遍历操作。

```
class Node {
public:
    int _coef;
    int _exp;
    Node *_next;

    Node(int coef, int exp, Node *next)
        : _coef(coef), _exp(exp), _next(next) {}
};

class Polynomial {
private:
    Node *_head;

public:
    Polynomial() : _head(new Node(0, 0, nullptr)) {}
    ~Polynomial();
    // ...
};
```

2. 插入项操作： insertTerm 方法用于按照指数降序的方式插入项。如果插入的项与已有项具有相同的指数，则合并这两项，如果合并后的系数为零，则删除该项。

```
void Polynomial::insertTerm(int coef, int exp) {
    if (coef == 0)
        return;

    Node *p = _head;
    while (p->_next != nullptr && p->_next->_exp > exp) {
        p = p->_next;
    }

    if (p->_next != nullptr && p->_next->_exp == exp) {
        // 合并相同指数的项
        p->_next->_coef += coef;
        if (p->_next->_coef == 0) {
            // 如果系数为零，删除该项
            Node *q = p->_next;
            p->_next = q->_next;

            delete q;
```

```
            _head->_coef--;
        }
    } else {
        // 插入新项
        p->_next = new Node(coef, exp, p->_next);
        _head->_coef++;
    }
}
```

3. 读取文件中的多项式： `readFromFile` 方法通过解析文件中的字符串，将系数和指数插入到多项式中。

```
void Polynomial::readFromFile(const char *filename) {
    // ...
}
```

4. 输出多项式： `outputPoly` 方法将多项式按照指定格式输出到文件。

```
void Polynomial::outputPoly(FILE *outputFile) {
    // ...
}
```

5. 多项式相加和相减： `addPoly` 和 `subPoly` 方法实现了多项式相加和相减，合并同指数项并计算结果。

```
Polynomial *Polynomial::addPoly(Polynomial &poly1, Polynomial &poly2) {
    // ...
}

Polynomial *Polynomial::subPoly(Polynomial &poly1, Polynomial &poly2) {
    // ...
}
```

6. 计算多项式值： `calcPoly` 方法通过遍历多项式的节点，计算多项式在给定 `x` 处的值。

```
double Polynomial::calcPoly(double x) {
    // ...
}
```

整体来说，这个实现为多项式提供了基本的插入、删除、计算以及文件读写的功能。链表的使用使得对多项式的操作更加高效，同时对于大多数基本操作都进行了清晰的处理。

四、输入输出

（一）对输入输出的处理

对文件中多项式信息的规定和处理：

```
3x^2 - 5x + 2
```

在文件中，这个多项式会被表示为：

```
3*x^2 - 5*x + 2
```

具体的规定如下：

- 空格被用作分隔符，用于分隔每一项。
- 系数和变量（x）之间不能有空格。
- 指数用 ^ 符号表示，紧跟在变量 x 的后面。
- 多项式中的每一项用加法或减法符号连接。

在代码中，readFromFile 方法解析了文件中的每一行，通过遍历字符来识别并提取系数和指数。它处理了正负号、乘号、变量 x，以及可能存在的指数。这样，通过符合规定格式的文件，代码就可以正确地读取多项式的信息。

```
void Polynomial::readFromFile(const char *filename)
{
    FILE *file = fopen(filename, "r");

    if (!file) {
        fprintf(stderr, "Unable to open file.\n");
        exit(1);
    }

    char line[1024];
    while (fgets(line, sizeof(line), file)) {
        char *p = line;
        int sign = 1;
        while (*p) {
            // ...
        }
    }

```

```
    fclose(file);
}
```

（二）输入输出样例

```
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/PolynomialCalculator$ make PolynomialCalculator
g++ -o PolynomialCalculator main.cpp Implementation.cpp
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/PolynomialCalculator$ cat poly1.input
19*x^3-9*x^2+0*x-9
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/PolynomialCalculator$ cat poly2.input
23*x^3-12*x^1+9-x^12
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/PolynomialCalculator$ ./PolynomialCalculator poly1.input poly2.input poly.output
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/PolynomialCalculator$ cat poly.output
p1 = 3,19,3,-9,2,-9,0
p2 = 4,-1,12,23,3,-12,1,9,0
p3 = p1 + p2 = 4,-1,12,42,3,-9,2,-12,1
p4 = p1 - p2 = 5,1,12,-4,3,-9,2,12,1,-18,0
p1(2) = 107.000000
```

五、总结

（一）时间复杂度分析：

1. insertTerm 方法： 最坏情况下，需要遍历链表找到合适的插入位置，因此时间复杂度为$O(n)$，其中 n 是多项式的项数。
2. readFromFile 方法： 外层循环遍历文件的每一行，内层循环遍历每一行的字符，因此时间复杂度是 $O(m*k)$，其中 m 是文件的行数，k 是文件中最长的一行的字符数。
3. outputPoly 方法： 遍历链表输出每一项的信息，时间复杂度为 $O(n)$，其中 n 是多项式的项数。
4. addPoly 和 subPoly 方法： 遍历两个多项式的链表，时间复杂度为 $O(max(n,m))$，其中 n 和 m 分别是两个多项式的项数。
5. calcPoly 方法： 遍历多项式的链表，时间复杂度为 $O(n)$，其中 n 是多项式的项数。

（二）空间复杂度分析：

1. `insertTerm` 方法： 除了常数个变量外，主要的空间消耗在新节点的分配，因此空间复杂度是$O(1)$。
2. `readFromFile` 方法： 除了常数个变量外，主要的空间消耗在新节点的分配，因此空间复杂度是$O(1)$。
3. `outputPoly` 方法： 除了常数个变量外，主要的空间消耗在输出结果所需的空间，因此空间复杂度是 $O(1)$。
4. `addPoly` 和 `subPoly` 方法： 除了常数个变量外，主要的空间消耗在新节点的分配，因此空间复杂度是 $O(1)$。
5. `calcPoly` 方法： 除了常数个变量外，主要的空间消耗在 `calcExp` 方法中，因此空间复杂度是$O(1)$。

（三）可能的改进空间：

1. **文件读取方式：** 当文件内容较大时，逐行读取可能会导致效率较低。可以考虑使用更高效的文件读取方式，比如一次性读取整个文件或者缓存一定量的字符。
2. **插入操作：** 可以考虑使用双向链表，以便在链表中间插入元素时，能够更快地找到插入位置。
3. **多项式相加和相减：** 在 `addPoly` 和 `subPoly` 方法中，每次插入项都要遍历链表。如果链表是按照指数降序排列的，可以优化插入逻辑，减少遍历次数。
4. **文件输入输出错误处理：** 在文件读取错误时，应该有更友好的错误处理方式，而不是直接退出程序。
5. **内存管理：** 当频繁插入和删除节点时，可能会导致内存碎片。可以考虑使用对象池或其他内存管理策略以优化内存使用。

实验11：哈夫曼压缩/解压缩算法（编译码器）

一、题目描述

利用哈夫曼编码进行信息通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，要求在发送端通过一个编码系统对传输数据预先编码（压缩）；在接收端将传来的数据进行译码（解压缩复原）。试为这样的通信站编写一个哈夫曼编译码系统---哈夫曼压缩/解压缩算法。

基本要求是：1）通信内容可以是任意的多媒体文件；2）自己设定字符大小，统计该文件中不同字符的种类（字符集、个数）、出现频率（在该文件中）；3）构建相应的哈夫曼树，并给出个字符的哈夫曼编码；4）对源文件进行哈夫曼压缩编码形成新的压缩后文件（包括哈夫曼树）；5）编写解压缩文件对压缩后文件进行解码还原成源文件。

二、解题思路

为实现哈夫曼的压缩和解压缩，首先需要统计原始文本的字符信息，比如对于字符X他的出现频率。然后根据出现的频率构建一颗哈夫曼树。接下来，根据构建的哈夫曼树绘制编码表。

在完成编码表的构建后开始写入文件。首先是文件头部，文件头部主要写入原始文件的字符个数以及编码表。然后根据编码表中的编码写入正文。**其中值得一提的是，编码写入的过程中将对编码表构建了一个Hash表，键值为编码对应的原始字符，这样能节约编码写入的时间。**

同时，由于C只能对字节操作，因此在写入时先建立一个缓冲区，对每八个bit值（实现时是选择了一个长度为8的0-1数组）转化为一个长度为1字节的字符写入文件，**也就是说，这一步将bit流转化为字节流写入文件**。

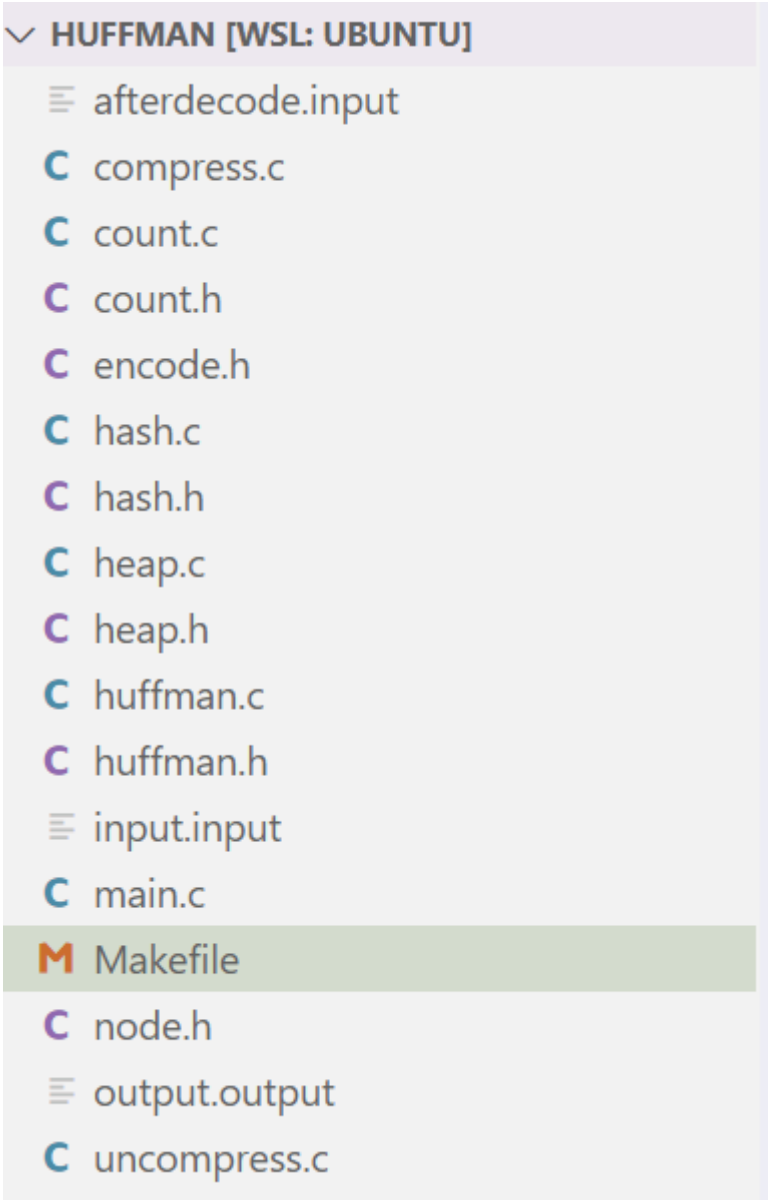
在解码时，首先应读取编码表，进而根据编码表还原一个哈夫曼树以方便解码。

解码过程中，首先要再次建立一个缓冲区，与写入时一样，**将读入的字节流转化为bit流**，进而与还原的哈夫曼树进行匹配，每匹配到了一个字符即清空缓冲区并写入输出文件。

三、大致的算法

（一）总体结构

代码主题由main以及compress压缩部分和uncompress解压缩部分构成，此外还有huffman.c及huffman.h用以构建huffman树，hash.c及hash.h用以构建hash表，heap.c及heap.h用以构建一个优先队列，用以辅助构建huffman树。同时，还有count.h和count.c以处理原始的读入文件，进行初始的文本分析。如图所示



（二）hash以构建hash表

实现了一个基于开放寻址法的哈希表。以下是对代码的算法分析：

1. cmpArray 函数

```
1  bool cmpArray(bool *array1, bool *array2, int length)
```

该函数用于比较两个布尔数组是否相等。它通过逐一比较数组中的元素，如果有任何不同的元素，则返回false；否则，返回true。

2. hash 函数

```
unsigned long hash(HashKey key)
```

该函数实现了一个简单的哈希算法，用于将给定的哈希键（HashKey）映射为一个无符号长整型数。具体来说，它对哈希键中的布尔数组进行处理，将每个布尔值转换为整数（0或1），然后使用一系列运算（包括位运算和取模运算）得到最终的哈希值。

3. insertHashTable 函数

```
void insertHashTable(HashKey key, ORIGINAL_DATA_TYPE value,
HashTable my_hash_table[])
```

该函数用于向哈希表中插入键值对。它首先通过调用 hash 函数计算键的哈希值，然后使用开放寻址法解决冲突，找到一个可用的位置插入。在插入之前，它会先复制键的布尔数组，以确保哈希表中存储的是键的副本。最后，更新哈希表中的相应条目，并将 isOccupied 标志设置为true。

4. searchHashTable 函数

```
ORIGINAL_DATA_TYPE searchHashTable(HashKey key, HashTable
my_hash_table[])
```

该函数用于在哈希表中查找给定键的值。它首先计算键的哈希值，然后使用开放寻址法遍历哈希表，直到找到匹配的键或遍历一遍整个哈希表。如果找到匹配的键，则返回对应的值；否则，返回0。

5. cleanHashTable 函数

```
void cleanHashTable(HashTable my_hash_table[])
```

该函数用于清理哈希表。它遍历哈希表的每个条目，释放占用的内存，并将键的 _code 指针设置为 DELETED。这是一种在开放寻址法中表示删除的方法，因为在查找时会检查键的 _code 是否为 DELETED。

这部分的代码实现了一个简单的哈希表，使用布尔数组作为键，并采用开放寻址法解决冲突。在插入、查找和清理操作中，都考虑了键的副本和删除标志。

（三）huffman以构建huffman树

实现了Huffman编码树的构建和相关操作。以下是对代码的算法分析：

1. createHuffmanNode 函数

```
pHuffmanNode createHuffmanNode(CharInfo element, int frequency)
``````
该函数用于创建一个Huffman节点，包含一个字符信息和对应的频率。该节点的左右子节点和父节点初始化为NULL。返回创建的节点指针。

2. `destroyHuffmanNode` 函数
```c
void destroyHuffmanNode(pHuffmanNode node)
```

该函数用于销毁一个Huffman节点，释放分配给节点的内存。

3. createHuffmanTree 函数

```
Huffman createHuffmanTree(CharInfo *char_list, int size)
```

该函数用于构建Huffman编码树。它首先创建一个最小堆（minHeap），将字符信息和频率作为元素，然后将所有节点依次插入最小堆。接着，通过不断取出堆顶的两个最小节点，创建一个新节点，该新节点的频率为两个最小节点的频率之和，并将这个新节点插入最小堆。直到最小堆中只剩一个节点为根节点，构建完成，返回根节点。

4. destroyHuffmanTree 函数

```
void destroyHuffmanTree(Huffman root)
```

该函数用于销毁Huffman编码树。通过递归地释放树中每个节点的内存，最终释放整个树的内存。

5. findMaxDepth 函数

```
int findMaxDepth(Huffman root)
```

该函数用于找到Huffman编码树的最大深度。通过递归地计算左右子树的深度，返回左右子树深度的较大值加1。如果树为空，则返回0。

可见，这部分段代码实现了Huffman编码树的构建、销毁和深度计算等基本操作。Huffman编码树是一种用于压缩数据的树形结构，其中频率较高的字符位于树的较低层，频率较低的字符位于树的较高层，以达到压缩数据的目的。

（四）heap以构建优先队列

这段代码实现了最小堆的基本操作，包括初始化、销毁、插入和删除堆顶元素等功能。下面是对代码的详细分析：

1. 堆结构体定义

```
typedef struct {
    HEAP_DATA_TYPE *_heap;  // 堆元素的指针数组
    int _size;              // 当前堆的大小
    int _capacity;          // 堆的最大容量
} MinHeap, *pMinHeap;
```

2. 初始化最小堆

```
pMinHeap initMinHeap(int capacity)
```

- 通过malloc为堆结构体和堆元素数组分配内存。
- 初始化堆的大小为0。
- 设置堆的最大容量。
- 返回指向最小堆结构体的指针。

3. 销毁最小堆

```
void destroyMinHeap(pMinHeap minHeap)
```

- 释放堆元素指针数组的内存。
- 释放堆结构体的内存。

4. 交换数组中两个元素的位置

```
void swap(HEAP_DATA_TYPE *a, HEAP_DATA_TYPE *b)
```

用于交换两个元素的值。

5. 向上调整堆

```
void heapifyUp(pMinHeap minHeap, int index)
```

- 从指定的索引向上调整堆，维持最小堆性质。
- 通过不断与父节点比较，并交换值，将元素移动到正确的位置。

6. 向下调整堆

```
1  void heapifyDown(pMinHeap minHeap, int index)
```

- 从指定的索引向下调整堆，维持最小堆性质。
- 通过比较当前节点和其左右子节点，找到最小值，并将当前节点与最小值的节点交换，然后递归调整。

7. 添加元素到堆

```
1  void pushHeap(pMinHeap minHeap, HEAP_DATA_TYPE value)
```

- 将元素添加到堆的末尾。
- 调用heapifyUp向上调整堆，确保维持最小堆性质。

8. 弹出堆顶元素

```
1  HEAP_DATA_TYPE popHeap(pMinHeap minHeap)
```

- 弹出堆顶元素，并返回其值。
- 将堆的最后一个元素放到堆顶。
- 调用heapifyDown向下调整堆，确保维持最小堆性质。

（五）compress压缩

1. 哈夫曼编码表生成

在 createCodeTable 函数中，实现了哈夫曼编码表的生成。该函数通过递归地遍历哈夫曼树，根据左右子树的情况，不断更新当前编码，并在叶子节点处将编码表中对应的字符的编码、长度和数据保存起来。这样，函数最终完成了哈夫曼编码表的构建。值得注意的是，为了实现递归过程中对编码表的正确填充，使用了一个静态变量 table_index 来追踪当前插入的位置。

2. 哈夫曼编码表的释放

destroyCodeTable 函数负责释放哈夫曼编码表占用的内存。对每一个编码表项，释放对应的编码数组和结构体，然后释放整个编码表数组的内存。这样做是为了防止内存泄漏。

3. 哈夫曼编码表扩展

expandCodeTable 函数将原始的哈夫曼编码表扩展为一个更大的表，以方便查表。该函数基于哈希表的思想，将原始表中的元素复制到一个新表中，新表的大小为 MAX_CHAR，表示 ASCII 字符集的大小。如果原始表中没有某个字符的编码，对应位置的编码数组为空。

4. 写入哈夫曼编码表到文件

writeCodeTable 函数负责将生成的哈夫曼编码表写入文件。它按照一定的格式，先写入编码表的大小、总字符数，然后逐项写入字符、编码长度和编码内容。这样，压缩后的文件就能够通过这个编码表进行解码。

5. 布尔数组到字节的转换

boolList2Byte 函数将长度为 8 的布尔数组转换为一个 8 位的字节。通过左移和按位或操作，将布尔数组中的每一位合并成一个字节

（六）uncompress解压缩

1. 读取哈夫曼编码表

readCodeTable 函数用于从压缩文件中读取哈夫曼编码表。首先，它读取编码表的大小和总字符数。然后，通过循环读取每个编码表项的字符、编码长度和编码内容。在读取编码内容时，为每个编码分配内存，并逐位读取编码内容。最终，函数返回构建好的哈夫曼编码表。

2. 从缓冲区中读取下一个比特

readBit 函数用于从缓冲区中读取下一个比特。它接收一个布尔型数组和一个缓冲区索引作为输入参数，并返回索引指向的比特值。在解压缩的过程中，这个函数用于逐位读取压缩后的比特流。

3. 构建哈夫曼树

codeTableBuildHuffmanTree 函数根据哈夫曼编码表构建哈夫曼树。它遍历每个编码表项，根据编码内容构建树的路径，最终将字符存储在叶子节点。该函数返回构建好的哈夫曼树的根节点。

4. 解压缩主函数

Huffman_Uncompress 函数是整个解压缩算法的主要执行部分。它首先读取哈夫曼编码表，然后根据编码表构建哈夫曼树。接着，它通过读取压缩后的比特流，根据哈夫曼树进行解码，并将解码结果写入输出文件。最后，释放相关的内存。

四、输入输出

（一）对输入输出的处理

此程序对于输入输出的处理基本于前面的方法类似，即：

1. 输入处理

该程序通过命令行参数处理输入。在 main 函数中，首先检查参数的数量是否为 4，即程序名称和两个文件名（输入和输出）。如果参数数量不为 4，会打印相应的错误信息，并调用 printHelp 函数显示用法信息，然后退出程序。在 printHelp 函数中，使用 fprintf 将帮助信息输出到指定的文件流（这里是标准输出）。

然后，使用 getAbsolutePath 函数获取输入和输出文件的绝对路径，以确保打开文件时使用的路径是正确的。该函数通过调用 getcwd 获取当前工作目录，并将文件名连接到路径上。获取绝对路径后，使用 fopen 分别打开输入和输出文件。如果文件打开失败，会输出错误信息并退出程序。在打开文件之前，还会释放使用 malloc 分配的绝对路径字符串的内存。

2. 输出处理

在打开输入和输出文件后，根据命令行参数选择执行压缩或解压缩操作。通过检查 argv[1] 的值来确定执行的操作，如果是压缩，则调用 Huffman_Compress 函数，如果是解压缩，则调用 Huffman_Uncompress 函数。如果命令不是合法的 -c 或 -d，则输出错误信息并调用 printHelp 函数显示用法信息，然后退出程序。

最后，在完成操作后，使用 fclose 关闭打开的文件，释放资源并正常退出程序。这样，整个程序通过命令行参数接收输入，处理文件的打开和关闭，并执行相应的压缩或解压缩操作。

（二）输入输出样例

1. 例子 1：压缩文件

```
./huffman_encode -c input.txt compressed.bin
```

这个命令将会压缩 input.txt 文件，生成一个压缩后的二进制文件 compressed.bin。

2. 例子 2：解压文件

```
./huffman_encode -d compressed.bin output.txt
```

这个命令将会解压 compressed.bin 文件，生成一个解压后的文本文件 output.txt。

上面的两个例子能够成功的处理大体积文件并实现真正的压缩\解压缩，由于篇幅问题在此不做展示

3. 例子 3：显示帮助信息

```
./huffman_encode -h
# 或者 ./huffman_encode --help
```

这两个命令都将显示程序的帮助信息，解释如何正确使用程序以及支持的选项。帮助信息如下

```
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/Huffman$
./huffman_encode -h
Usage: huff -[c|d] <infile> <outfile>
Compress or decompress file using Huffman coding.
<infile>  Input file, it's required to be in the same directory as
the executable file.
<outfile> Output file, it's required to be in the same directory as
the executable file.
Example: huff -c input.txt output.txt
Using --help or -h to get help.
Options:
  -c  Compress infile to outfile
  -d  Decompress infile to outfile
```

4. 例子 4：错误处理

```
./huffman_encode input.txt
```

这个命令将会输出错误信息，因为缺少了输出文件名。如下：

```
rouqi@ROUGE-LAPTOP-LEGION:~/Cpp-repo/XJTUDS-Labs/Huffman$
./huffman_encode input.input
Too few arguments

Usage: huff -[c|d] <infile> <outfile>
Compress or decompress file using Huffman coding.
<infile>  Input file, it's required to be in the same directory as
the executable file.
<outfile> Output file, it's required to be in the same directory as
the executable file.
Example: huff -c input.txt output.txt
Using --help or -h to get help.
Options:
  -c  Compress infile to outfile
  -d  Decompress infile to outfile
```

又或者下面的错误命令

```
./huffman_encode -x input.txt output.txt
```

这个命令将会输出错误信息，因为 -x 不是一个合法的选项。

五、总结

（一）时间复杂度分析

1. compress部分

1. **createCodeTable 函数**：时间复杂度：O(n) ，其中 n 是哈夫曼树的节点数。
2. **destroyCodeTable 函数**：时间复杂度：O(size)，其中 size 是编码表的大小。
3. **expandCodeTable 函数**：时间复杂度：O(MAX_CHAR)，其中 MAX_CHAR 是字符集的大小。
4. **writeCodeTable 函数**：时间复杂度：O(table_size * avg_code_length)，其中 table_size 是编码表的大小，avg_code_length 是平均编码长度。
5. **boolList2Byte 函数**：时间复杂度：O(1)
6. **writeByte 函数**：时间复杂度：O(1)

7. **Huffman_Compress 函数**：时间复杂度：O(n + h + MAX_CHAR)，其中 n 是文件字符数，h 是哈夫曼树的高度。

2. uncompress部分

1. **readCodeTable函数**：时间复杂度：O(table_size * avg_code_length)，其中 table_size 是哈夫曼编码表的大小，avg_code_length 是平均编码长度。
2. **codeTableBuildHuffmanTree函数**：时间复杂度：O(table_size * max_code_length)，其中 table_size 是哈夫曼编码表的大小，max_code_length 是最大编码长度。
3. **inorderTraversal函数**：时间复杂度：O(n)，其中 n 是哈夫曼树的节点数。
4. **Huffman_Uncompress函数**：时间复杂度：O(text_ch_num)，其中 text_ch_num 是文件字符数。

3. huffman部分

1. **createHuffmanNode函数**：时间复杂度：O(1)，分配节点内存的操作是常数时间。
2. **destroyHuffmanNode函数：时间复杂度**：O(1)，释放节点内存的操作是常数时间。
3. **createHuffmanTree函数：时间复杂度**：O(n log n)，其中 n 是字符列表的大小。在建堆阶段，插入 n 个节点的时间复杂度是 O(n log n)，然后每次弹出最小元素和插入新元素的时间复杂度也是 O(log n)。总体来说，是 O(n log n)。
4. **destroyHuffmanTree函数：时间复杂度**：O(n)，其中 n 是哈夫曼树的节点数。每个节点都会被访问一次。
5. **findMaxDepth函数：时间复杂度**：O(n)，其中 n 是哈夫曼树的节点数。每个节点都会被访问一次。

4. heap部分

1. **initMinHeap函数：时间复杂度**：O(1)，分配内存和初始化操作是常数时间。
2. **destroyMinHeap函数：时间复杂度**：O(1)，释放内存的操作是常数时间。
3. **swap函数：时间复杂度**：O(1)，交换两个元素的值是常数时间。
4. **heapifyUp函数：时间复杂度**：O(log n)，其中 n 是堆的大小。在最坏情况下，需要向上调整的次数等于堆的高度，即 log n。

5. **heapifyDown函数：时间复杂度**：O(log n)，其中 n 是堆的大小。在最坏情况下，需要向下调整的次数等于堆的高度，即 log n。
6. **pushHeap函数：时间复杂度**：O(log n)，其中 n 是堆的大小。由于调用了 heapifyUp 函数，插入元素的最坏情况时间复杂度为 log n。
7. **popHeap函数：时间复杂度**：O(log n)，其中 n 是堆的大小。由于调用了 heapifyDown 函数，弹出元素的最坏情况时间复杂度为 log n。

5. hash部分

1. **cmpArray函数：时间复杂度**：O(length)，其中 length 是数组的长度。需要比较数组中的所有元素。
2. **hash函数：时间复杂度**：O(key.*length)，其中 key.*length 是哈希键的长度。需要遍历哈希键的所有元素。
3. **insertHashTable函数：时间复杂度**：平均情况下为 O(1)，最坏情况下为 O(HASH_TABLE_SIZE)，其中 HASH_TABLE_SIZE 是哈希表的大小。在哈希冲突较少的情况下，插入操作的时间复杂度是常数级别的；但在哈希冲突较多时，可能需要线性探测直到找到一个空槽。
4. **searchHashTable函数：时间复杂度**：平均情况下为 O(1)，最坏情况下为 O(HASH_TABLE_SIZE)，其中 HASH_TABLE_SIZE 是哈希表的大小。在哈希冲突较少的情况下，查找操作的时间复杂度是常数级别的；但在哈希冲突较多时，可能需要线性探测直到找到目标元素或确定不存在。
5. **cleanHashTable函数：时间复杂度**：O(HASH_TABLE_SIZE)，其中 HASH_TABLE_SIZE 是哈希表的大小。需要遍历整个哈希表清理每个槽。

（二）空间复杂度分析

1. compress部分

1. **createCodeTable 函数**：空间复杂度：O(h)，其中 h 是哈夫曼树的高度。
2. **destroyCodeTable 函数**：空间复杂度：O(1)
3. **expandCodeTable 函数**：空间复杂度：O(MAX_CHAR)，其中 MAX_CHAR 是字符集的大小。
4. **writeCodeTable 函数**：空间复杂度：O(1)
5. **boolList2Byte 函数**：空间复杂度：O(1)
6. **writeByte 函数**：空间复杂度：O(1)
7. **Huffman_Compress 函数**：空间复杂度：O(h + MAX_CHAR + n)，其中 h 是哈夫曼树的高度，MAX_CHAR 是字符集的大小，n 是文件字符数。

2. uncompress部分

1. **readCodeTable函数**：空间复杂度：O(table_size * avg_code_length)，其中 table_size 是哈夫曼编码表的大小，avg_code_length 是平均编码长度。
2. **codeTableBuildHuffmanTree函数**：空间复杂度：O(table_size * max_code_length)，其中 table_size 是哈夫曼编码表的大小，max_code_length 是最大编码长度。
3. **inorderTraversal函数**：空间复杂度：O(h)，其中 h 是哈夫曼树的高度。
4. **Huffman_Uncompress函数**：空间复杂度：O(BUFFER_MAX_FILE_SIZE * sizeof(__uint8_t) * 8)，其中 BUFFER_MAX_FILE_SIZE 是缓冲区大小。

3. huffman部分

1. **createHuffmanNode 函数**：**空间复杂度**：O(1)，每个节点分配的内存是常数。
2. **destroyHuffmanNode函数**：**空间复杂度**：O(1)，每个节点释放的内存是常数。
3. **createHuffmanTree函数**：**空间复杂度**：O(n)，建堆所需的额外空间是O(n)。
4. **destroyHuffmanTree函数**：**空间复杂度**：O(1)，递归调用并不会累积额外的空间。
5. **findMaxDepth函数**：**空间复杂度**：O(h)，其中 h 是哈夫曼树的高度。递归调用的最大深度由树的高度决定。

4. heap部分

1. **initMinHeap函数**：**空间复杂度**：O(1)，仅分配了堆结构体和元素指针数组的常数空间。
2. **destroyMinHeap函数**：**空间复杂度**：O(1)，释放内存的操作是常数空间。
3. **swap函数**：**空间复杂度**：O(1)，仅使用了常数额外空间。
4. **heapifyUp函数**：**空间复杂度**：O(1)，递归调用时并未累积额外空间。
5. **heapifyDown函数**：**空间复杂度**：O(1)，递归调用时并未累积额外空间。
6. **pushHeap函数**：**空间复杂度**：O(1)，调用 heapifyUp 时没有额外空间使用。
7. **popHeap函数**：**空间复杂度**：O(1)，调用 heapifyDown 时没有额外空间使用。

5. hash部分

1. **cmpArray函数：空间复杂度**：O(1)，没有使用额外的动态内存。
2. **hash函数：空间复杂度**：O(1)，没有使用额外的动态内存。
3. **insertHashTable函数：空间复杂度**：O(1)，每次插入时分配了一个新的 bool 数组，但空间复杂度是常数级别的。
4. **searchHashTable函数：空间复杂度**：O(1)，没有使用额外的动态内存。
5. **cleanHashTable函数：空间复杂度**：O(1)，没有使用额外的动态内存。

（三）可以优化的地方

1. compress部分

1. **createCodeTable函数**：避免使用静态变量 table_index，将其作为函数的返回值，减少静态变量的使用。
2. 考虑使用非递归方式实现，避免深度较大的树导致栈溢出。

2. uncompress部分

1. **readCodeTable函数** ：考虑采用非递归的方式读取编码表，减少递归调用带来的额外开销。此外在为每个编码表项分配内存时，可以考虑批量分配以减少频繁的内存分配操作。

2. **codeTableBuildHuffmanTree函数**：优化构建哈夫曼树的过程，可能考虑使用堆等数据结构来提高效率。

3. **Huffman_Uncompress函数**：考虑以块的方式读取和解压缩文件，而不是一次性读取整个文件，以减少内存的使用。这在处理大文件时尤为重要。

3. huffman部分

1. **createHuffmanTree函数**：对于频繁插入和弹出的场景，使用斐波那契堆等更高效的数据结构可能提高性能。
2. **destroyHuffmanTree函数**：递归销毁节点时，考虑使用迭代方式以避免深度递归调用带来的堆栈开销。
3. **findMaxDepth函数**：考虑使用迭代方式计算树的深度，以避免深度递归调用。

4. heap部分

1. **堆结构体的大小动态调整**：如果堆的大小变化不大，可以考虑使用动态调整堆结构体大小，以减少内存占用。
2. **自适应容量调整**：当堆的元素数量达到容量上限时，考虑动态扩展堆的容量，以避免堆满后的无法插入情况。
3. **异常处理**：在 pushHeap 和 popHeap 函数中，对于堆已满或空的情况，可以使用异常处理或返回错误码，而不是直接在标准错误流上输出信息。

5. hash部分

1. **处理哈希冲突的方式**：考虑采用更高效的处理哈希冲突的方式，如开放寻址法中的二次探测、链地址法等。这可以改善在哈希冲突较多时的性能。
2. **动态调整哈希表大小**：当哈希表的负载因子过高时，可以考虑动态调整哈希表的大小，以减少冲突的可能性。这涉及到重新哈希和数据迁移的问题。
3. **使用更复杂的哈希函数**：考虑使用更复杂、更均匀分布的哈希函数，以降低冲突的概率。
4. **使用现成的哈希表实现**：考虑使用标准库或其他现成的哈希表实现，它们通常经过充分优化，并考虑了各种处理冲突的方式。

实验13：迷宫问题

一、题目描述

迷宫实验是取自心理学的一个古典实验。在该实验中，把一只老鼠从一个无顶大盒子的门放入，在盒中设置了许多墙，对行进方向形成了多处阻挡。盒子仅有一个出口，在出口处放置一块奶酪，吸引老鼠在迷宫中寻找道路以到达出口。对同一只老鼠重复进行上述实验，一直到老鼠从入口到出口，而不走错一步。老鼠经多次试验终于得到它学习走迷宫的路线。

迷宫由m行n列的二维数组设置，0表示无障碍，1表示有障碍。设入口为（1，1），出口为（m，n），每次只能从一个无障碍单元移到周围四个方向上任一无障碍单元。编程实现对任意设定的迷宫，求出一条从入口到出口的通路，或得出没有通路的结论。

二、解题思路

1. **生成迷宫：**
    - 使用`generate_maze`函数生成一个迷宫，迷宫是一个二维数组，其中每个元素表示一个单元格，包含四个方向的墙和一个标志位表示是否已经访问过。
    - 通过随机选择单元格，使用深度优先搜索（DFS）算法生成迷宫路径，并打开路径两侧的墙。迷宫路径生成的具体步骤在注释中有详细解释。
    - 起点和终点分别在迷宫的左上角和右下角，将它们的墙打开。
2. **绘制迷宫：**
    - 使用`display_maze`函数将迷宫以图像形式显示。
    - 每个单元格用一个方块表示，单元格内部填充，墙部分用像素表示。
    - 外墙加粗，入口和出口打开，起点和终点标记。
3. **BFS寻找路径：**
    - 使用`find_path`函数利用广度优先搜索（BFS）算法在生成的迷宫中寻找从起点到终点的路径。
    - 使用一个队列存储待访问的单元格，并记录每个单元格的前一个单元格，从而构建路径。
4. **显示迷宫和路径：**
    - 使用`display_maze_solve`函数在显示的图像上标记出找到的路径。
    - 如果找不到路径，则打印提示信息。

5. **命令行参数处理：**
    - 使用sys.argv处理命令行参数，接受两个正整数作为行和列的尺寸。
    - 如果没有提供参数，默认生成一个10x10的迷宫。
6. **主函数：**
    - 在main函数中调用上述函数，生成迷宫、显示迷宫、寻找路径，并将结果以图像形式展示。
    - 如果提供了命令行参数，则使用指定的行和列尺寸生成迷宫。

三、大致的算法

（一）迷宫的生成

迷宫生成算法的部分主要使用了深度优先搜索（DFS）的思想。下面对代码中的关键部分进行详尽的说明：

1. `generate_maze` 函数的目标是生成一个迷宫，其中 `maze_total_rows` 和 `maze_total_cols` 分别表示迷宫的总行数和总列数。
2. `message_of_point` 是一个三维数组，用于保存每个迷宫单元格的信息。其形状是 `(maze_total_rows, maze_total_cols, 5)`，其中：
    - 第一个维度表示行数。
    - 第二个维度表示列数。
    - 第三个维度有五个元素，分别表示当前单元格的左、上、右、下是否有墙以及该单元格是否已被访问。
3. 迷宫生成的主要过程：
    - 初始位置为 `(0, 0)`，将其添加到 `history_found` 中。
    - 在 `while` 循环中，不断从 `history_found` 中随机选择一个单元格 `(this_row, this_col)`。
    - 将选择的单元格标记为已访问（`message_of_point[this_row, this_col, 4] = 1`）。
    - 检查当前单元格四个方向，如果相邻的单元格未被访问，将其添加到 `history_found` 中，并在当前单元格和相邻单元格之间打通路径（打通墙）。
    - 迭代直到 `history_found` 为空，即无法再继续探索时，生成的迷宫就是一个完整的迷宫。
4. 关键变量和操作：
    - `history_found` 是一个存储已访问位置的堆栈，用于深度优先搜索。
    - `check` 存储当前单元格中有相邻未访问单元格的方向。

- `move_direction` 随机选择一个方向。
- `message_of_point` 用于记录每个单元格的信息，包括墙和是否已访问。

5. 最终，迷宫生成算法保证了每个单元格都可达，并且通过深度优先搜索的方式生成了一条路径，形成了一个随机生成的迷宫。

（二）迷宫寻路部分

这段代码包含了一个基于广度优先搜索（BFS）的迷宫寻路算法。下面详细解释关键部分：

1. `find_path` 函数:
   - `maze`: 生成的迷宫，其中包含了每个单元格的信息，包括墙的位置。
   - `start`: 起点的坐标 `(row, col)`。
   - `end`: 终点的坐标 `(row, col)`。
2. `message_of_point` 数组:
   - `message_of_point` 是一个与迷宫数组结构相同的数组，用于存储每个单元格的信息，包括墙的位置和访问状态。
   - 初始化时，将 `message_of_point[:, :, :4]` 的值设置为迷宫的信息，即墙的位置。
   - 第五个维度 `message_of_point[:, :, 4]` 用于标记单元格是否已经在搜索中被访问，初始值为0。
3. BFS 寻路算法:
   - 使用队列 `queue` 存储待访问的单元格坐标，初始时将起点 `start` 加入队列。
   - 使用字典 `previous_point` 存储每个单元格的前驱，以便在找到终点后回溯路径。
   - 在每次循环中，从队列中取出一个单元格`(this_row, this_col)` 进行如下判断：
     - 如果左边的单元格未访问且有通路，则将其加入队列，并标记为已访问。
     - 如果上方的单元格未访问且有通路，则将其加入队列，并标记为已访问。
     - 如果右边的单元格未访问且有通路，则将其加入队列，并标记为已访问。
     - 如果下方的单元格未访问且有通路，则将其加入队列，并标记为已访问。
     - 如果当前单元格是终点，停止搜索。

4. 回溯路径:
    - 从终点开始，通过 previous_point 字典回溯到起点，得到路径上的每个单元格坐标。
    - 将路径逆序插入 path 列表中。
5. display_maze_solve 函数:
    - 该函数根据找到的路径，将路径上的单元格在迷宫图像上标记为通路，以突显路径。
6. main 函数:
    - 读取用户输入的行列数，生成迷宫，显示原始迷宫图像。
    - 寻找路径，如果找到则显示带有路径的迷宫图像。

总体而言，这一部分通过广度优先搜索算法在生成的迷宫中找到起点到终点的路径，并通过图像展示了迷宫和找到的路径。算法保证了找到的路径是最短路径。

四、输入输出

（一）输入输出处理

让我们分析输入输出的处理：

1. 输入处理

命令行参数（sys.argv）：

- 如果命令行参数的数量既不是2个也不是0个，就会输出错误信息并调用help()函数。
- 如果参数数量是0个，则使用默认的迷宫尺寸（10x10）。
- 如果参数数量是2个，会尝试将它们解析为整数，如果无法解析或者解析结果不是正整数，同样输出错误信息并调用help()函数。

2. 输出处理

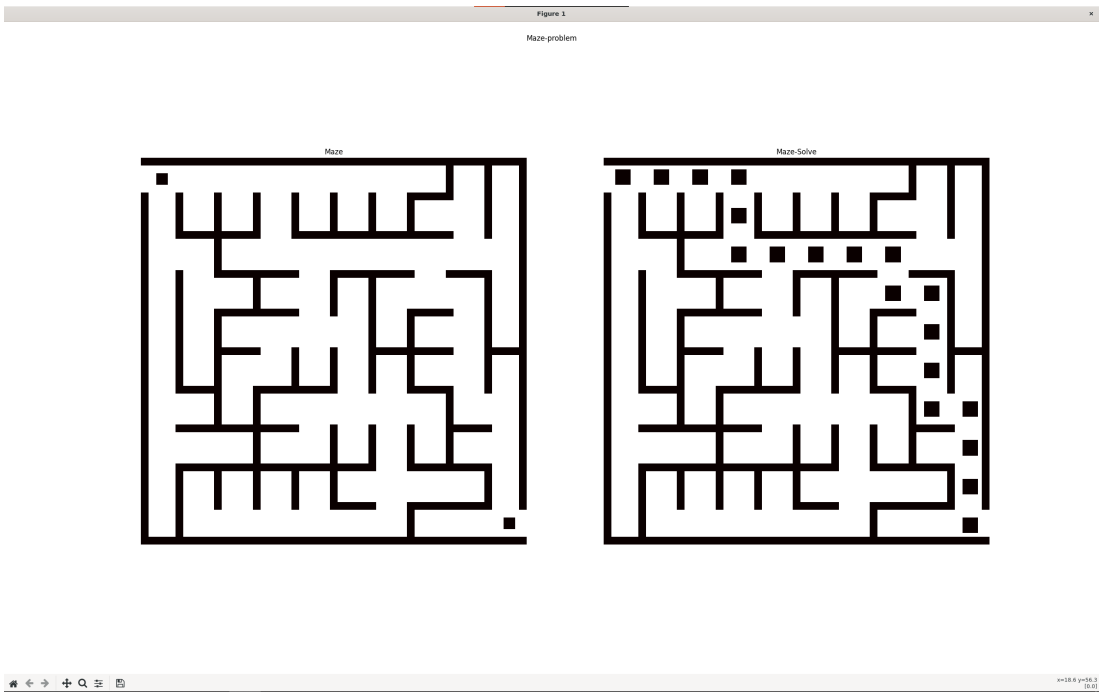
1. 生成迷宫输出：generate_maze 函数会返回一个三维数组 message_of_point 表示迷宫的结构，其中包含每个单元格四面的墙以及是否被访问过。输出时通过 print(message_of_point) 打印生成的迷宫信息。
2. 绘制迷宫输出：display_maze 函数通过Matplotlib库生成迷宫的可视化图像，其中墙用二进制表示。输出时使用 plt.imshow 函数显示图像。

3. **找到路径输出：**`find_path` 函数通过BFS算法寻找迷宫中的路径，返回路径的坐标。如果找到路径，会打印路径的坐标。
4. **迷宫求解显示输出：**`display_maze_solve` 函数通过将找到的路径在图像上标记为不同的颜色来展示求解的迷宫。输出时使用 `plt.imshow` 函数显示带有路径的图像。
5. **主程序输出：**主程序通过调用上述函数，并使用Matplotlib库展示生成的迷宫和找到的路径。

（二）输入输出样例

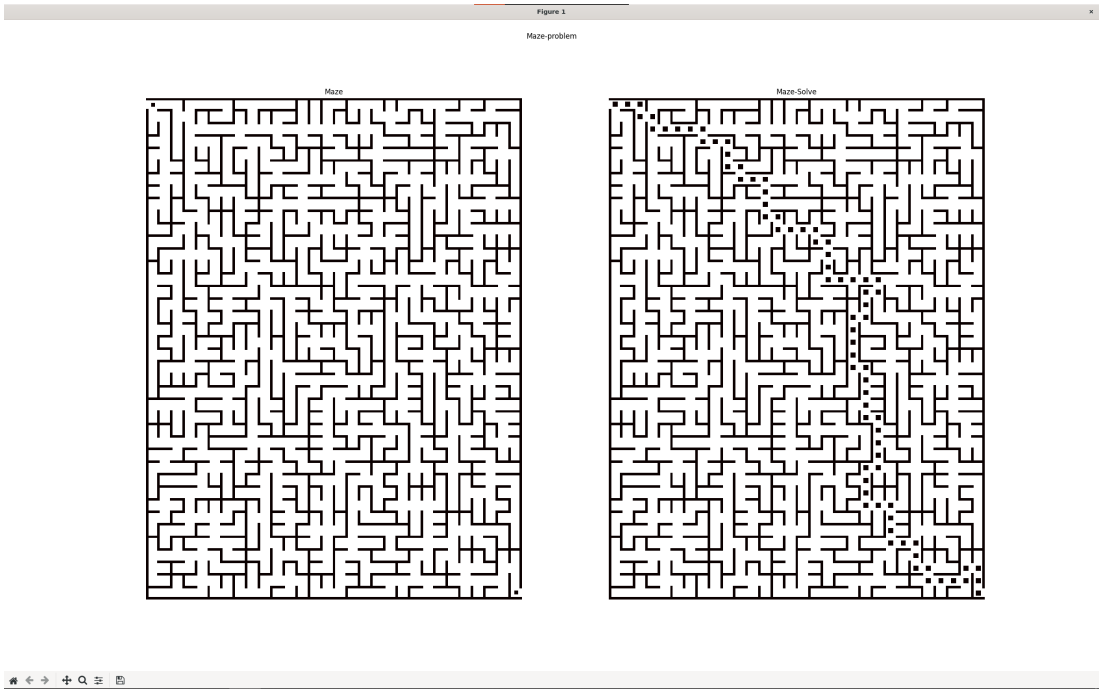
1. 输入输出样例1

```
# 以默认大小运行（10x10）
python3 maze.py
```



2. 输入输出样例2

```
python3 maze.py 40 30
```



3. 输入输出样例3

```
rouqi@ROUGE-LAPTOP-LEGION:~/python-repo$ python3 maze.py --help
Usage: python3 maze.py <num_rows> <num_cols>
       <num_rows> and <num_cols> must be positive integers,
       and <num_rows> * <num_cols> must be greater than 1.
       If no arguments are provided, the default maze size is 10x10.
```

五、总结

（一）算法复杂度分析

1. 生成迷宫算法 generate_maze

- **时间复杂度**：主要由 while 循环控制，每次循环中的操作都是常数时间的。最坏情况下，每个单元格都会被访问一次，所以总的时间复杂度为 O(maze_total_rows * maze_total_cols)，其中 maze_total_rows 为行数，maze_total_cols 为列数。
- **空间复杂度**：使用了大小为O(maze_total_rows * maze_total_cols * 5) 的三维数组 message_of_point 来保存每个单元格的信息。所以总的空间复杂度为 O(maze_total_rows * maze_total_cols)。

2. 绘制迷宫算法 display_maze

- **时间复杂度**：主要由两个嵌套的循环控制，嵌套循环的次数与迷宫的大小成正比。所以总的时间复杂度为 O(maze_total_rows * maze_total_cols)。
- **空间复杂度**：使用了大小为 O(rows_num * cols_num * 10 * 10) 的二维数组 image_output 来保存图像信息。所以总的空间复杂度为 O(rows_num * cols_num)。

3. 使用BFS算法寻找路径 find_path

- **时间复杂度**：BFS算法的时间复杂度是O(V + E)，其中 V 为顶点数，E 为边数。在这里，V为迷宫的单元格数，E为边的数量（每个单元格四个方向的边），时间复杂度O(maze_total_rows * maze_total_cols)。
- **空间复杂度**：
  - 使用了大小为 O(maze_total_rows * maze_total_cols * 5) 的三维数组 message_of_point 来保存每个单元格的信息。
  - 使用了大小为 O(maze_total_rows * maze_total_cols) 的字典 previous_point 用于存储路径。
  - 所以总的空间复杂度O(maze_total_rows * maze_total_cols)。

（二）改进空间

1. 减少历史记录的选择操作：
   - 当随机选择的单元格已经被访问过，可以避免将其添加到历史记录中，而是直接选择下一个单元格。这样可以减少历史记录的大小，提高生成迷宫的效率。
2. 迭代生成算法：
   - 可以考虑使用迭代生成算法，如Prim's算法或Kruskal's算法，来替代当前的生成迷宫算法。这些算法在某些情况下可能比随机生成迷宫更高效。
3. BFS算法优化：
   - 在BFS算法中，可以尝试优化数据结构的选择，例如使用队列来代替列表，以提高查找和弹出元素的效率。
4. 图像显示优化：
   - 在图像显示部分，可以考虑使用更高效的图像生成方法，例如直接使用matplotlib的imshow方法来绘制墙和路径，而不是使用循环逐个设置像素值。
