# 1、系统进程统计（必做）（链表）

## 简介

本程序读取 /proc 目录下的进程信息，统计进程的内存使用情况，以及进程的运行时间。程序使用链表数据结构来存储和管理进程信息。

## 环境要求

* 本程序需要在 Linux 环境下运行。
* 测试环境：6.12.4-arch1-1 ArchLinux x86\_64

## 功能

* 读取 /proc 目录下的进程信息。
* 统计每个进程的内存使用情况。
* 统计每个进程的运行时间。
* 将进程信息存储在链表中，并按内存使用情况排序。
* 将结束的进程信息存储在另一个链表中，并按运行时间排序。

## 使用方法

1. 克隆或下载本项目到本地。
2. 使用 gcc 编译程序：

gcc -o process\_stat main.c

1. 运行编译后的程序：

./process\_stat

1. 程序将每秒读取一次进程信息，并打印当前活动进程和已结束进程的信息。

## 注意事项

* 本程序需要在具有 /proc 文件的 Linux 环境下运行。

## 数据结构

本程序使用双向链表来存储和管理进程信息。每个节点表示一个进程，包含进程名称、PID、内存使用量、运行时间等信息。

## 算法设计思想

1. 读取 /proc 目录下的进程信息。
2. 将进程信息插入到链表中，按内存使用量排序。
3. 每秒更新一次进程信息，更新运行时间。
4. 将结束的进程从活动链表中移除，插入到已结束进程链表中，按运行时间排序。

## 实现的源程序

源程序见 main.c 文件，代码中包含详细注释。

## 测试数据和结果

测试数据：程序运行时自动读取系统的进程信息。 测试结果：每秒打印当前活动进程和已结束进程的信息。

## 算法的时间复杂度

读取和插入操作的时间复杂度为 ，其中 n 为当前链表的长度。总体时间复杂度为 。

## 算法的改进方法

可以使用更高效的数据结构（如平衡二叉树或堆）来存储和管理进程信息，以降低时间复杂度。

## 源代码

### ../procs/main.c

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <dirent.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <time.h>  
  
typedef struct Proc // Process struct, Linked List  
{  
 char name[256];  
 int mem;  
 int pid;  
 int dur;  
 time\_t end;  
 struct Proc \*next;   
 struct Proc \*prev;  
} Proc;  
  
Proc \*head = NULL; // Head of the linked list  
Proc \*finHead = NULL; // Head of the fin processes linked list  
  
time\_t getCurrentTime()  
{  
 return time(NULL);  
}  
  
void insertProc(char \*name, int pid, int mem)  
{  
 Proc \*newProc = (Proc \*)malloc(sizeof(Proc));  
 strcpy(newProc->name, name);  
 newProc->pid = pid;  
 newProc->mem = mem;  
 newProc->dur = 0;  
 newProc->end = 0;  
 newProc->next = NULL;  
 newProc->prev = NULL;  
  
 if (!head || head->mem < mem)  
 {  
 newProc->next = head;  
 head = newProc;  
 }  
 else  
 {  
 Proc \*cur = head;  
 while (cur->next && cur->next->mem >= mem)  
 {  
 cur = cur->next;  
 }  
 newProc->next = cur->next;  
 cur->next = newProc;  
 }  
}  
  
void insertFinProc(Proc \*proc)  
{  
 proc->next = NULL;  
 proc->prev = NULL;  
  
 if (!finHead || finHead->dur > proc->dur)  
 {  
 proc->next = finHead;  
 if (finHead)  
 {  
 finHead->prev = proc;  
 }  
 finHead = proc;  
 }  
 else  
 {  
 Proc \*cur = finHead;  
 while (cur->next && cur->next->dur <= proc->dur)  
 {  
 cur = cur->next;  
 }  
 proc->next = cur->next;  
 if (cur->next)  
 {  
 cur->next->prev = proc;  
 }  
 cur->next = proc;  
 proc->prev = cur;  
 }  
}  
  
void clearProcs()  
{  
 Proc \*cur = head;  
 while (cur)  
 {  
 Proc \*temp = cur;  
 cur = cur->next;  
 free(temp);  
 }  
 head = NULL;  
}  
  
Proc \*findProc(int pid)  
{  
 Proc \*cur = head;  
 while (cur)  
 {  
 if (cur->pid == pid)  
 {  
 return cur;  
 }  
 cur = cur->next;  
 }  
 return NULL;  
}  
  
Proc \*findFinProc(int pid)  
{  
 Proc \*cur = finHead;  
 while (cur)  
 {  
 if (cur->pid == pid)  
 {  
 return cur;  
 }  
 cur = cur->next;  
 }  
 return NULL;  
}  
  
void moveToActiveProc(Proc \*proc)  
{  
 if (proc->prev)  
 {  
 proc->prev->next = proc->next;  
 }  
 if (proc->next)  
 {  
 proc->next->prev = proc->prev;  
 }  
 if (proc == finHead)  
 {  
 finHead = proc->next;  
 }  
  
 proc->next = NULL;  
 proc->prev = NULL;  
 insertProc(proc->name, proc->pid, proc->mem);  
}  
  
void updateOrInsertProc(char \*name, int pid, int mem)  
{  
 Proc \*existingProc = findProc(pid);  
 if (existingProc)  
 {  
 existingProc->mem = mem;  
 existingProc->dur = -existingProc->dur;  
 }  
 else  
 {  
 Proc \*finishedProc = findFinProc(pid);  
 if (finishedProc)  
 {  
 moveToActiveProc(finishedProc);  
 }  
 else  
 {  
 insertProc(name, pid, mem);  
 }  
 }  
}  
  
void readProcs()  
{  
 Proc \*cur = head;  
  
 while (cur)  
 {  
 cur->dur = -cur->dur;  
 cur = cur->next;  
 }  
  
 DIR \*procDir = opendir("/proc");  
 if (!procDir)  
 {  
 perror("opendir");  
 return;  
 }  
  
 struct dirent \*entry;  
 while ((entry = readdir(procDir)) != NULL)  
 {  
 if (entry->d\_type == DT\_DIR && atoi(entry->d\_name) > 0) // Check if the entry is a directory and the name is a number  
 {  
 char statPath[512];  
 snprintf(statPath, sizeof(statPath), "/proc/%s/stat", entry->d\_name); // Get the path of the stat file  
  
 FILE \*statFile = fopen(statPath, "r");  
 if (!statFile) // Check if the file exists  
 continue;  
  
 char name[256];  
 int pid, mem = 0;  
  
 fscanf(statFile, "%d %s", &pid, name); // Read the pid and name from the stat file  
 fclose(statFile);  
  
 name[strlen(name) - 1] = '\0';  
 memmove(name, name + 1, strlen(name));  
  
 char statmPath[512];  
 snprintf(statmPath, sizeof(statmPath), "/proc/%s/statm", entry->d\_name);   
  
 FILE \*statmFile = fopen(statmPath, "r"); //  
 if (statmFile)  
 {  
 fscanf(statmFile, "%d", &mem); // Read the memory usage from the statm file  
 fclose(statmFile);  
 }  
  
 mem \*= 4;  
  
 updateOrInsertProc(name, pid, mem);  
 }  
 }  
 closedir(procDir);  
  
 Proc \*prev = NULL;  
 cur = head;  
 while (cur)  
 {  
 if (cur->dur < 0)  
 {  
 cur->end = getCurrentTime();  
 if (prev)  
 {  
 prev->next = cur->next;  
 }  
 else  
 {  
 head = cur->next;  
 }  
 Proc \*temp = cur;  
 cur = prev ? prev->next : head;  
 temp->dur = -temp->dur;  
 insertFinProc(temp);  
 }  
 else  
 {  
 prev = cur;  
 cur = cur->next;  
 }  
 }  
}  
  
void printProcs()  
{  
 printf("=== Active Processes ===\n");  
 Proc \*cur = head;  
 while (cur)  
 {  
 printf("PID: %d, Name: %s, Memory: %d kB, Duration: %d seconds\n",  
 cur->pid, cur->name, cur->mem, cur->dur);  
 cur = cur->next;  
 }  
}  
  
void printFinProcs()  
{  
 printf("=== Finnish Processes ===\n");  
 Proc \*cur = finHead;  
 while (cur)  
 {  
 printf("PID: %d, Name: %s, Duration: %d seconds, End Time: %s",  
 cur->pid, cur->name, cur->dur, ctime(&cur->end));  
 cur = cur->next;  
 }  
}  
  
void updateDurations()  
{  
 Proc \*cur = head;  
 while (cur)  
 {  
 cur->dur++;  
 cur = cur->next;  
 }  
}  
  
int main()  
{  
 while (1)  
 {  
 readProcs();  
 printProcs();  
 printFinProcs();  
 updateDurations();  
 sleep(1);  
 }  
  
 return 0;  
}

# 2、迷宫问题（必做）（栈与队列）

## 项目简介

本项目包含两个主要功能：生成迷宫和求解迷宫。迷宫生成器会随机生成一个包含路径和墙壁的迷宫，并将其保存到文件中。迷宫求解器会读取该文件中的迷宫，并尝试找到从入口到出口的路径。

## 文件结构

* generate\_maze.c：生成迷宫并保存到文件中。
* main.c：读取迷宫文件并尝试求解迷宫。
* maze.txt：保存生成的迷宫数据。

## 使用方法

### 生成迷宫

1. 编译 generate\_maze.c 文件：

gcc generate\_maze.c -o generate\_maze

1. 运行生成迷宫程序：

./generate\_maze

该程序会生成一个随机迷宫并保存到 maze.txt 文件中。

### 求解迷宫

1. 编译 main.c 文件：

gcc main.c -o solve\_maze

1. 运行求解迷宫程序：

./solve\_maze

该程序会读取 maze.txt 文件中的迷宫数据，并尝试找到从入口到出口的路径。如果找到路径，会输出 “Path found!”，否则输出 “No path found.”。同时会打印出标记过的迷宫。

## 代码说明

### generate\_maze.c

* generateMaze 函数：随机生成迷宫，1/3 的概率生成墙壁，确保入口和出口是路径。
* saveMazeToFile 函数：将生成的迷宫保存到文件中。

### main.c

* initStack 函数：初始化栈。
* isStackEmpty 函数：判断栈是否为空。
* push 函数：将位置压入栈中。
* pop 函数：从栈中弹出位置。
* isValid 函数：判断当前位置是否有效。
* printMaze 函数：打印迷宫。
* solveMaze 函数：使用深度优先搜索算法求解迷宫。

## 注意事项

* 迷宫的大小由 ROWS 和 COLS 宏定义，默认为 20x20。
* 生成的迷宫文件名为 maze.txt，可以根据需要修改。
* 迷宫的入口和出口位置是固定的，分别为左上角和右下角。

## 源代码

### ../maze/generate\_maze.c

#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
  
#define ROWS 20  
#define COLS 20  
#define PATH 0  
#define WALL 1  
  
void generateMaze(int maze[ROWS][COLS]) {  
 srand(time(NULL));  
 for (int i = 0; i < ROWS; i++) {  
 for (int j = 0; j < COLS; j++) {  
 maze[i][j] = (rand() % 3 == 0) ? WALL : PATH; // 1/3概率生成墙  
 }  
 }  
 maze[0][0] = PATH; // 确保入口是路径  
 maze[ROWS-1][COLS-1] = PATH; // 确保出口是路径  
}  
  
void saveMazeToFile(int maze[ROWS][COLS], const char \*filename) {  
 FILE \*file = fopen(filename, "w");  
 if (file == NULL) {  
 printf("Failed to open file.\n");  
 return;  
 }  
 for (int i = 0; i < ROWS; i++) {  
 for (int j = 0; j < COLS; j++) {  
 fprintf(file, "%d ", maze[i][j]);  
 }  
 fprintf(file, "\n");  
 }  
 fclose(file);  
}  
  
int main() {  
 int maze[ROWS][COLS];  
 generateMaze(maze);  
 saveMazeToFile(maze, "maze.txt");  
 printf("Maze generated and saved to maze.txt\n");  
 return 0;  
}

### ../maze/main.c

#include <stdio.h>  
#include <stdlib.h>  
  
#define ROWS 20  
#define COLS 20  
#define PATH 0  
#define WALL 1  
#define VISITED 2  
  
typedef struct {  
 int x, y;  
} Position;  
  
typedef struct {  
 Position \*data;  
 int top;  
 int capacity;  
} Stack;  
  
void initStack(Stack \*s, int capacity) {  
 s->data = (Position \*)malloc(sizeof(Position) \* capacity);  
 s->top = -1;  
 s->capacity = capacity;  
}  
  
int isStackEmpty(Stack \*s) {  
 return s->top == -1;  
}  
  
void push(Stack \*s, Position pos) {  
 if (s->top < s->capacity - 1) {  
 s->data[++(s->top)] = pos;  
 }  
}  
  
Position pop(Stack \*s) {  
 if (!isStackEmpty(s)) {  
 return s->data[(s->top)--];  
 }  
 Position invalid = {-1, -1};  
 return invalid;  
}  
  
int isValid(int maze[ROWS][COLS], int x, int y) {  
 return x >= 0 && x < ROWS && y >= 0 && y < COLS && maze[x][y] == PATH;  
}  
  
void printMaze(int maze[ROWS][COLS]) {  
 for (int i = 0; i < ROWS; i++) {  
 for (int j = 0; j < COLS; j++) {  
 printf("%d ", maze[i][j]);  
 }  
 printf("\n");  
 }  
}  
  
void solveMaze(int maze[ROWS][COLS], Position start, Position end) {  
 Stack stack;  
 initStack(&stack, ROWS \* COLS);  
 push(&stack, start);  
  
 int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};  
  
 while (!isStackEmpty(&stack)) {  
 Position current = pop(&stack);  
 if (current.x == end.x && current.y == end.y) {  
 printf("Path found!\n");  
 return;  
 }  
  
 maze[current.x][current.y] = VISITED;  
  
 for (int i = 0; i < 4; i++) {  
 int newX = current.x + directions[i][0];  
 int newY = current.y + directions[i][1];  
 if (isValid(maze, newX, newY)) {  
 push(&stack, (Position){newX, newY});  
 }  
 }  
 }  
  
 printf("No path found.\n");  
}  
  
int main() {  
 int maze[ROWS][COLS];  
 FILE \*file = fopen("maze.txt", "r");  
 if (file == NULL) {  
 printf("Failed to open file.\n");  
 return 1;  
 }  
  
 for (int i = 0; i < ROWS; i++) {  
 for (int j = 0; j < COLS; j++) {  
 fscanf(file, "%d", &maze[i][j]);  
 }  
 }  
 fclose(file);  
  
 Position start = {0, 0}; // 任意入口  
 Position end = {ROWS - 1, COLS - 1}; // 任意出口  
  
 solveMaze(maze, start, end);  
 printMaze(maze);  
  
 return 0;  
}

# 3、家谱管理系统（必做）（树）

## 项目简介

本项目是一个简单的家谱管理系统，允许用户管理家族成员的信息，包括添加、删除、修改成员信息，显示家谱结构，查询成员信息等功能。 家谱数据以 JSON 格式存储在文件中，程序启动时会加载该文件中的数据，用户对家谱数据的修改保存到文件中。 文件 familyData.json 中包含了一个示例家谱数据，可以用于测试和演示。

## 文件结构

* Member.h 和 Member.cpp：定义了家族成员的类及其相关操作。
* FamilyTree.h 和 FamilyTree.cpp：定义了家谱树的类及其相关操作。
* main.cpp：主程序入口，提供用户交互界面。
* familyData.json：存储家谱数据的示例文件。
* Makefile：用于构建项目的 Makefile。

## 使用方法

1. 克隆或下载本项目到本地。
2. 在项目根目录下执行 make 命令构建项目。
3. 运行生成的可执行文件 ./bin/family\_tree。

## 构建步骤

1. 至少下载3/目录和json\_parser/目录下的所有文件，因为json\_parser是依赖项，具体参考Makefile文件。
2. 在项目根目录下执行以下命令：

make

1. 构建成功后，运行以下命令启动程序：

./bin/family\_tree

## 功能说明

* 显示家谱：显示整个家谱结构。
* 显示第 n 代所有人信息：按代数显示成员信息。
* 按姓名查询成员信息：根据姓名查询成员详细信息。
* 按出生日期查询成员名单：根据出生日期查询成员列表。
* 确定两人关系：确定两人之间的关系。
* 添加成员孩子：为指定成员添加孩子。
* 删除成员：删除指定成员及其后代。
* 修改成员信息：修改指定成员的详细信息。
* 保存并退出：保存当前家谱数据并退出程序。

## 源代码

### ../family/FamilyTree.cpp

#include "FamilyTree.h"  
#include "json\_parser.h"  
#include <iostream>  
#include <fstream>  
#include <sstream>  
#include <functional>  
#include <unordered\_map>  
  
void FamilyTree::loadFromFile(const std::string& filename) {  
 std::ifstream file(filename);  
 if (!file.is\_open()) {  
 throw std::runtime\_error("无法打开文件");  
 }  
  
 std::stringstream buffer;  
 buffer << file.rdbuf();  
 std::string jsonString = buffer.str();  
 file.close();  
  
 JsonParser parser(jsonString);  
 std::vector<std::string> memberKeys = parser.queryList("members");  
 for (const auto& key : memberKeys) {  
 std::string memberJson = parser.query(key);  
 JsonParser memberParser(memberJson);  
 Member member(  
 memberParser.query("name"),  
 memberParser.query("birthDate"),  
 memberParser.query("isMarried") == "true",  
 memberParser.query("address"),  
 memberParser.query("isAlive") == "true",  
 memberParser.query("deathDate"),  
 memberParser.query("fatherName")  
 );  
 if (member.fatherName.empty()) {  
 rootName = key;  
 }  
 members[key] = member;  
 }  
  
 for(const auto& member : members) {  
 children[member.second.fatherName].push\_back(member.first);  
 }  
}  
  
void FamilyTree::saveToFile(const std::string& filename) {  
 std::ofstream file(filename);  
 if (!file.is\_open()) {  
 throw std::runtime\_error("无法打开文件");  
 }  
  
 file << "{";  
 file << "\"members\":[";  
 for (auto it = members.begin(); it != members.end(); ++it) {  
 if (it != members.begin()) {  
 file << ",";  
 }  
 file << "\"" << it->first << "\"";  
 }  
 file << "],";  
  
 for (auto it = members.begin(); it != members.end(); ++it) {  
 if (it != members.begin()) {  
 file << ",";  
 }  
 file << "\"" << it->first << "\":{";  
 file << "\"name\":\"" << it->second.name << "\",";  
 file << "\"birthDate\":\"" << it->second.birthDate << "\",";  
 file << "\"isMarried\":\"" << (it->second.isMarried ? "true" : "false") << "\",";  
 file << "\"address\":\"" << it->second.address << "\",";  
 file << "\"isAlive\":\"" << (it->second.isAlive ? "true" : "false") << "\",";  
 file << "\"deathDate\":\"" << it->second.deathDate << "\",";  
 file << "\"fatherName\":\"" << it->second.fatherName << "\"";  
 file << "}";  
 }  
 file << "}";  
 file.close();  
}  
  
void FamilyTree::displayFamilyTree() {  
 std::function<void(const std::string&, int)> displayMember = [&](const std::string& name, int level) {  
 for (int i = 0; i < level; ++i) {  
 std::cout << " ";  
 }  
 std::cout << "|-- " << name << std::endl;  
 if (children.find(name) != children.end()) {  
 for (const auto& child : children[name]) {  
 displayMember(child, level + 1);  
 }  
 }  
 };  
  
 if (!rootName.empty()) {  
 displayMember(rootName, 0);  
 } else {  
 std::cerr << "家谱根节点未找到。" << std::endl;  
 }  
}  
  
void FamilyTree::displayGeneration(int n) {  
 std::function<void(const std::string&, int)> displayGen = [&](const std::string& name, int level) {  
 if (level == n) {  
 std::cout << "=== 第 " << n << " 代 ===" << std::endl;  
 members[name].Print();  
 }  
 if (children.find(name) != children.end()) {  
 for (const auto& child : children[name]) {  
 displayGen(child, level + 1);  
 }  
 }  
 };  
  
 if (!rootName.empty()) {  
 displayGen(rootName, 0);  
 } else {  
 std::cerr << "家谱根节点未找到。" << std::endl;  
 }  
}  
  
Member& FamilyTree::findMemberByName(const std::string& name) {  
 if (members.find(name) == members.end()) {  
 throw std::runtime\_error("未找到成员");  
 }  
 return members[name];  
}  
  
void FamilyTree::searchByBirthDate(const std::string& date) {  
 for (const auto& member : members) {  
 if (member.second.birthDate == date) {  
 member.second.Print();  
 }  
 }  
}  
  
void FamilyTree::determineRelationship(const std::string& name1, const std::string& name2) {  
 std::unordered\_map<std::string, int> depth;  
 std::function<void(const std::string&, int)> calculateDepth = [&](const std::string& name, int level) {  
 depth[name] = level;  
 if (children.find(name) != children.end()) {  
 for (const auto& child : children[name]) {  
 calculateDepth(child, level + 1);  
 }  
 }  
 };  
  
 if (!rootName.empty()) {  
 calculateDepth(rootName, 0);  
 } else {  
 std::cerr << "家谱根节点未找到。" << std::endl;  
 return;  
 }  
  
 if (depth.find(name1) == depth.end() || depth.find(name2) == depth.end()) {  
 std::cerr << "未找到成员。" << std::endl;  
 return;  
 }  
  
 int depth1 = depth[name1];  
 int depth2 = depth[name2];  
  
 if (depth1 == depth2) {  
 std::cout << name1 << " 和 " << name2 << " 是同一代人。" << std::endl;  
 } else if (depth1 < depth2) {  
 std::cout << name1 << " 是 " << name2 << " 的长辈。" << std::endl;  
 } else {  
 std::cout << name2 << " 是 " << name1 << " 的长辈。" << std::endl;  
 }  
}  
  
void FamilyTree::addChild(const std::string& parentName, Member child) {  
 if (members.find(parentName) == members.end()) {  
 throw std::runtime\_error("未找到父母成员");  
 }  
 child.fatherName = parentName;  
 members[child.name] = child;  
 children[parentName].push\_back(child.name);  
}  
  
void FamilyTree::deleteMember(const std::string& name) {  
 if (members.find(name) == members.end()) {  
 throw std::runtime\_error("未找到成员");  
 }  
  
 // 递归删除成员的孩子  
 if (children.find(name) != children.end()) {  
 for (const auto& child : children[name]) {  
 deleteMember(child);  
 }  
 children.erase(name);  
 }  
  
 // 从父母的孩子列表中删除该成员  
 if (!members[name].fatherName.empty()) {  
 auto& siblings = children[members[name].fatherName];  
 for(auto it = siblings.begin(); it != siblings.end(); ++it) {  
 if (\*it == name) {  
 siblings.erase(it);  
 break;  
 }  
 }  
 }  
  
 // 删除成员  
 members.erase(name);  
}  
  
void FamilyTree::modifyMember(const std::string& name) {  
 if (members.find(name) == members.end()) {  
 throw std::runtime\_error("未找到成员");  
 }  
  
 Member& member = members[name];  
 std::cout << "修改成员信息：" << std::endl;  
 std::cout << "当前姓名：" << member.name << std::endl;  
 std::cout << "请输入新姓名（按 Enter 保持不变）：";  
 std::string newName;  
 std::getline(std::cin >> std::ws, newName);  
 if (!newName.empty()) {  
 member.name = newName;  
 }  
  
 std::cout << "当前出生日期：" << member.birthDate << std::endl;  
 std::cout << "请输入新出生日期（按 Enter 保持不变）：";  
 std::string newBirthDate;  
 std::getline(std::cin, newBirthDate);  
 if (!newBirthDate.empty()) {  
 member.birthDate = newBirthDate;  
 }  
  
 std::cout << "当前婚否：" << (member.isMarried ? "是" : "否") << std::endl;  
 std::cout << "请输入新婚否（true/false，按 Enter 保持不变）：";  
 std::string newIsMarried;  
 std::getline(std::cin, newIsMarried);  
 if (!newIsMarried.empty()) {  
 member.isMarried = (newIsMarried == "true");  
 }  
  
 std::cout << "当前地址：" << member.address << std::endl;  
 std::cout << "请输入新地址（按 Enter 保持不变）：";  
 std::string newAddress;  
 std::getline(std::cin, newAddress);  
 if (!newAddress.empty()) {  
 member.address = newAddress;  
 }  
  
 std::cout << "当前健在否：" << (member.isAlive ? "是" : "否") << std::endl;  
 std::cout << "请输入新健在否（true/false，按 Enter 保持不变）：";  
 std::string newIsAlive;  
 std::getline(std::cin, newIsAlive);  
 if (!newIsAlive.empty()) {  
 member.isAlive = (newIsAlive == "true");  
 }  
  
 std::cout << "当前死亡日期：" << member.deathDate << std::endl;  
 std::cout << "请输入新死亡日期（按 Enter 保持不变）：";  
 std::string newDeathDate;  
 std::getline(std::cin, newDeathDate);  
 if (!newDeathDate.empty()) {  
 member.deathDate = newDeathDate;  
 }  
  
 std::cout << "当前父亲姓名：" << member.fatherName << std::endl;  
 std::cout << "请输入新父亲姓名（按 Enter 保持不变）：";  
 std::string newFatherName;  
 std::getline(std::cin, newFatherName);  
 if (!newFatherName.empty()) {  
 member.fatherName = newFatherName;  
 }  
}

### ../family/FamilyTree.h

#ifndef FAMILYTREE\_H  
#define FAMILYTREE\_H  
  
#include "Member.h"  
#include <string>  
#include <map>  
#include <memory>  
  
class FamilyTree {  
public:  
 std::map<std::string, Member> members; // 成员列表, key 为姓名, value 为成员对象  
 std::map<std::string,std::vector<std::string>> children; // 孩子列表, key 为父亲姓名, value 为孩子姓名列表  
 std::string rootName; // 家谱根节点姓名  
  
 void loadFromFile(const std::string& filename); // 从文件加载数据  
 void saveToFile(const std::string& filename); // 保存数据到文件  
 void displayFamilyTree(); // 显示家谱  
 void displayGeneration(int n); // 显示第 n 代所有人信息  
 Member& findMemberByName(const std::string& name); // 按姓名查找成员  
 void searchByBirthDate(const std::string& date); // 按出生日期查询成员  
 void determineRelationship(const std::string& name1, const std::string& name2); // 确定两人关系  
 void addChild(const std::string& parentName, Member child); // 添加孩子  
 void deleteMember(const std::string& name); // 删除成员  
 void modifyMember(const std::string& name); // 修改成员信息  
};  
  
#endif // FAMILYTREE\_H

### ../family/Member.cpp

#include "Member.h"  
#include <iostream>  
  
Member::Member()  
 : name(""), isMarried(false), isAlive(true)  
{  
}  
  
Member::Member(const std::string &name)  
 : name(name), isMarried(false), isAlive(true)  
{  
}  
  
Member::Member(const std::string &name, const std::string &birthDate, bool isMarried, const std::string &address, bool isAlive, const std::string &deathDate, const std::string &fatherName)  
 : name(name), birthDate(birthDate), isMarried(isMarried), address(address), isAlive(isAlive), deathDate(deathDate), fatherName(fatherName)  
{  
}  
  
Member::Member(const Member &member)  
 : name(member.name), birthDate(member.birthDate), isMarried(member.isMarried), address(member.address), isAlive(member.isAlive), deathDate(member.deathDate), fatherName(member.fatherName)  
{  
}  
  
Member &Member::operator=(const Member &member)  
{  
 if (this != &member)  
 {  
 name = member.name;  
 birthDate = member.birthDate;  
 isMarried = member.isMarried;  
 address = member.address;  
 isAlive = member.isAlive;  
 deathDate = member.deathDate;  
 fatherName = member.fatherName;  
 }  
 return \*this;  
}  
  
void Member::Print() const  
{  
 std::cout << "=== 成员信息 ===" << std::endl;  
 std::cout << "姓名：" << name << std::endl;  
 std::cout << "出生日期：" << birthDate << std::endl;  
 std::cout << "婚否：" << (isMarried ? "是" : "否") << std::endl;  
 std::cout << "地址：" << address << std::endl;  
 std::cout << "健在否：" << (isAlive ? "是" : "否") << std::endl;  
 if (!isAlive)  
 {  
 std::cout << "死亡日期：" << deathDate << std::endl;  
 }  
 if (!fatherName.empty())  
 {  
 std::cout << "父亲姓名：" << fatherName << std::endl;  
 }  
 else  
 {  
 std::cout << "是家谱根节点" << std::endl;  
 }  
}

### ../family/Member.h

#ifndef MEMBER\_H  
#define MEMBER\_H  
  
#include <string>  
#include <vector>  
  
class Member {  
public:  
 std::string name; // 姓名  
 std::string birthDate; // 出生日期  
 bool isMarried; // 婚否  
 std::string address; // 地址  
 bool isAlive; // 健在否  
 std::string deathDate; // 死亡日期  
 std::string fatherName; // 父亲姓名，如果没有父亲则为空  
  
 Member();  
 Member(const Member& member);  
 Member(const std::string& name);  
 Member(const std::string& name, const std::string& birthDate, bool isMarried, const std::string& address, bool isAlive, const std::string& deathDate, const std::string& fatherName);  
  
 Member& operator=(const Member& member);  
  
 void Print() const;  
};  
  
#endif // MEMBER\_H

### ../family/main.cpp

#include "FamilyTree.h"  
#include <iostream>  
  
const std::string FAMILY\_DATA\_FILE = "familyData.json";  
  
int main() {  
 FamilyTree familyTree;  
 try {  
 familyTree.loadFromFile(FAMILY\_DATA\_FILE);  
 } catch (const std::exception &e) {  
 std::cerr << "加载家谱数据时发生错误: " << e.what() << std::endl;  
 return 1;  
 }  
  
 int choice;  
 do {  
 std::cout << "======= 家谱管理系统 =======" << std::endl;  
 std::cout << "1. 显示家谱" << std::endl;  
 std::cout << "2. 显示第 n 代所有人信息" << std::endl;  
 std::cout << "3. 按姓名查询成员信息" << std::endl;  
 std::cout << "4. 按出生日期查询成员名单" << std::endl;  
 std::cout << "5. 确定两人关系" << std::endl;  
 std::cout << "6. 添加成员孩子" << std::endl;  
 std::cout << "7. 删除成员" << std::endl;  
 std::cout << "8. 修改成员信息" << std::endl;  
 std::cout << "9. 保存并退出" << std::endl;  
 std::cout << "请选择功能编号：";  
 std::cin >> choice;  
  
 try {  
 switch (choice) {  
 case 1:  
 familyTree.displayFamilyTree();  
 break;  
 case 2: {  
 int generation;  
 std::cout << "请输入代数：";  
 std::cin >> generation;  
 familyTree.displayGeneration(generation);  
 break;  
 }  
 case 3: {  
 std::string name;  
 std::cout << "请输入姓名：";  
 std::cin >> name;  
 auto member = familyTree.findMemberByName(name);  
 member.Print();  
 break;  
 }  
 case 4: {  
 std::string birthDate;  
 std::cout << "请输入出生日期：";  
 std::cin >> birthDate;  
 familyTree.searchByBirthDate(birthDate);  
 break;  
 }  
 case 5: {  
 std::string name1, name2;  
 std::cout << "请输入第一个人的姓名：";  
 std::cin >> name1;  
 std::cout << "请输入第二个人的姓名：";  
 std::cin >> name2;  
 familyTree.determineRelationship(name1, name2);  
 break;  
 }  
 case 6: {  
 std::string parentName, childName;  
 std::cout << "请输入父母的姓名：";  
 std::cin >> parentName;  
 std::cout << "请输入孩子的姓名：";  
 std::cin >> childName;  
 Member child(childName);  
 familyTree.addChild(parentName, child);  
 break;  
 }  
 case 7: {  
 std::string name;  
 std::cout << "请输入要删除的成员姓名：";  
 std::cin >> name;  
 familyTree.deleteMember(name);  
 break;  
 }  
 case 8: {  
 std::string name;  
 std::cout << "请输入要修改的成员姓名：";  
 std::cin >> name;  
 familyTree.modifyMember(name);  
 break;  
 }  
 case 9:  
 familyTree.saveToFile(FAMILY\_DATA\_FILE);  
 std::cout << "数据已保存，程序退出。" << std::endl;  
 break;  
 default:  
 std::cout << "无效的选择，请重新输入。" << std::endl;  
 break;  
 }  
 } catch (const std::exception &e) {  
 std::cerr << "操作时发生错误: " << e.what() << std::endl;  
 }  
 } while (choice != 9);  
  
 return 0;  
}

# 4、 平衡二叉树编程

## 项目简介

该项目实现了一个平衡二叉树（AVL树），并使用该树完成以下任务：

1. 对于1-10000的质数序列<2，3，5，7，…，9973>，建立平衡二叉排序树。
2. 依次查询200-300的每个数是否在树中，将结果写入tree1.txt。
3. 依次删除500-2000中的每个质数，再查询600-700之间的每个质数是否在树中，将结果写入tree2.txt。
4. 在任务（2）的基础上，依次插入1-1000的所有偶数，依次查询100-200的每个偶数是否在树中，将结果写入tree3.txt。

## 文件结构

* main.c：主程序，包含质数生成、插入、删除和查询操作。
* avl\_tree.h：AVL树的头文件，定义了树的结构和函数。
* avl\_tree.c：AVL树的实现文件，包含树的插入、删除、查询等操作。

## 编译和运行

make  
./main

## 源代码

### ../avl\_tree/avl\_tree.c

#include "avl\_tree.h"  
#include <stdio.h>  
#include <stdlib.h>  
  
  
  
static int height(AVLNode \*node) {  
 return node ? node->height : 0;  
}  
  
static int max(int a, int b) {  
 return (a > b) ? a : b;  
}  
  
static AVLNode\* create\_node(int key) {  
 AVLNode \*node = (AVLNode\*)malloc(sizeof(AVLNode));  
 node->key = key;  
 node->left = node->right = NULL;  
 node->height = 1;  
 return node;  
}  
  
  
  
static AVLNode\* right\_rotate(AVLNode \*y) {  
 AVLNode \*x = y->left;  
 AVLNode \*T2 = x->right;  
  
 x->right = y;  
 y->left = T2;  
  
 y->height = max(height(y->left), height(y->right)) + 1;  
 x->height = max(height(x->left), height(x->right)) + 1;  
  
 return x;  
}  
  
static AVLNode\* left\_rotate(AVLNode \*x) {  
 AVLNode \*y = x->right;  
 AVLNode \*T2 = y->left;  
  
 y->left = x;  
 x->right = T2;  
  
 x->height = max(height(x->left), height(x->right)) + 1;  
 y->height = max(height(y->left), height(y->right)) + 1;  
  
 return y;  
}  
  
static int get\_balance(AVLNode \*node) {  
 return node ? height(node->left) - height(node->right) : 0;  
}  
  
static AVLNode\* insert\_node(AVLNode\* node, int key) {  
 if (!node) return create\_node(key);  
  
 if (key < node->key)  
 node->left = insert\_node(node->left, key);  
 else if (key > node->key)  
 node->right = insert\_node(node->right, key);  
 else  
 return node;  
  
 node->height = 1 + max(height(node->left), height(node->right));  
  
 int balance = get\_balance(node);  
  
 if (balance > 1 && key < node->left->key)  
 return right\_rotate(node);  
  
 if (balance < -1 && key > node->right->key)  
 return left\_rotate(node);  
  
 if (balance > 1 && key > node->left->key) {  
 node->left = left\_rotate(node->left);  
 return right\_rotate(node);  
 }  
  
 if (balance < -1 && key < node->right->key) {  
 node->right = right\_rotate(node->right);  
 return left\_rotate(node);  
 }  
  
 return node;  
}  
  
static AVLNode\* min\_value\_node(AVLNode\* node) {  
 AVLNode\* current = node;  
 while (current->left != NULL)  
 current = current->left;  
 return current;  
}  
  
static AVLNode\* delete\_node(AVLNode\* root, int key) {  
 if (!root) return root;  
  
 if (key < root->key)  
 root->left = delete\_node(root->left, key);  
 else if (key > root->key)  
 root->right = delete\_node(root->right, key);  
 else {  
 if ((root->left == NULL) || (root->right == NULL)) {  
 AVLNode \*temp = root->left ? root->left : root->right;  
  
 if (!temp) {  
 temp = root;  
 root = NULL;  
 } else  
 \*root = \*temp;  
  
 free(temp);  
 } else {  
 AVLNode\* temp = min\_value\_node(root->right);  
 root->key = temp->key;  
 root->right = delete\_node(root->right, temp->key);  
 }  
 }  
  
 if (!root) return root;  
  
 root->height = 1 + max(height(root->left), height(root->right));  
  
 int balance = get\_balance(root);  
  
 if (balance > 1 && get\_balance(root->left) >= 0)  
 return right\_rotate(root);  
  
 if (balance > 1 && get\_balance(root->left) < 0) {  
 root->left = left\_rotate(root->left);  
 return right\_rotate(root);  
 }  
  
 if (balance < -1 && get\_balance(root->right) <= 0)  
 return left\_rotate(root);  
  
 if (balance < -1 && get\_balance(root->right) > 0) {  
 root->right = right\_rotate(root->right);  
 return left\_rotate(root);  
 }  
  
 return root;  
}  
  
static int search\_node(AVLNode\* root, int key) {  
 if (!root) return 0;  
 if (key == root->key) return 1;  
 if (key < root->key) return search\_node(root->left, key);  
 return search\_node(root->right, key);  
}  
  
AVLTree\* create\_avl\_tree() {  
 AVLTree \*tree = (AVLTree\*)malloc(sizeof(AVLTree));  
 tree->root = NULL;  
 return tree;  
}  
  
void insert\_avl\_tree(AVLTree \*tree, int key) {  
 tree->root = insert\_node(tree->root, key);  
}  
  
void delete\_avl\_tree(AVLTree \*tree, int key) {  
 tree->root = delete\_node(tree->root, key);  
}  
  
int search\_avl\_tree(AVLTree \*tree, int key) {  
 return search\_node(tree->root, key);  
}  
  
void free\_avl\_tree(AVLTree \*tree) {  
 // Free all nodes (not implemented for brevity)  
 free(tree);  
}

### ../avl\_tree/avl\_tree.h

#ifndef AVL\_TREE\_H  
#define AVL\_TREE\_H  
  
typedef struct AVLNode {  
 int key;  
 struct AVLNode \*left;  
 struct AVLNode \*right;  
 int height;  
} AVLNode;  
  
typedef struct AVLTree {  
 AVLNode \*root;  
} AVLTree;  
  
AVLTree\* create\_avl\_tree();  
void insert\_avl\_tree(AVLTree \*tree, int key);  
void delete\_avl\_tree(AVLTree \*tree, int key);  
int search\_avl\_tree(AVLTree \*tree, int key);  
void free\_avl\_tree(AVLTree \*tree);  
  
#endif // AVL\_TREE\_H

### ../avl\_tree/main.c

#include "avl\_tree.h"  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
  
int is\_prime(int num) {  
 if (num <= 1) return 0;  
 if (num <= 3) return 1;  
 if (num % 2 == 0 || num % 3 == 0) return 0;  
 for (int i = 5; i \* i <= num; i += 6) {  
 if (num % i == 0 || num % (i + 2) == 0) return 0;  
 }  
 return 1;  
}  
  
void generate\_primes(int \*primes, int max) {  
 int count = 0;  
 for (int i = 2; i <= max; i++) {  
 if (is\_prime(i)) {  
 primes[count++] = i;  
 }  
 }  
}  
  
void write\_results(const char \*filename, int \*nums, int size, AVLTree \*tree) {  
 FILE \*file = fopen(filename, "w");  
 for (int i = 0; i < size; i++) {  
 fprintf(file, "%d %s\n", nums[i], search\_avl\_tree(tree, nums[i]) ? "yes" : "no");  
 }  
 fclose(file);  
}  
  
int main() {  
 AVLTree \*tree = create\_avl\_tree();  
 int primes[10000];  
 generate\_primes(primes, 10000);  
  
 for (int i = 0; i < 10000 && primes[i] != 0; i++) {  
 insert\_avl\_tree(tree, primes[i]);  
 }  
  
 int query1[101];  
 for (int i = 200; i <= 300; i++) {  
 query1[i - 200] = i;  
 }  
 write\_results("tree1.txt", query1, 101, tree);  
  
 for (int i = 500; i <= 2000; i++) {  
 if (is\_prime(i)) {  
 delete\_avl\_tree(tree, i);  
 }  
 }  
  
 int query2[101];  
 for (int i = 600; i <= 700; i++) {  
 if (is\_prime(i)) {  
 query2[i - 600] = i;  
 }  
 }  
 write\_results("tree2.txt", query2, 101, tree);  
  
 for (int i = 2; i <= 1000; i += 2) {  
 insert\_avl\_tree(tree, i);  
 }  
  
 int query3[51];  
 for (int i = 100; i <= 200; i += 2) {  
 query3[(i - 100) / 2] = i;  
 }  
 write\_results("tree3.txt", query3, 51, tree);  
  
 free\_avl\_tree(tree);  
 return 0;  
}

# 5、Huffman编码与解码 (必做)（Huffman编码、二叉树）

## 项目简介

本项目实现了Huffman编码和解码算法。Huffman编码是一种无损数据压缩算法，通过使用变长编码来减少数据的存储空间。该项目包括以下功能：

* 计算字符频率
* 构建Huffman树
* 生成Huffman编码
* 编码文件
* 解码文件

## 文件结构

* huffman.h：包含Huffman编码和解码的函数声明和数据结构定义。
* huffman.c：包含Huffman编码和解码的函数实现。
* main.c：包含主函数，用于调用编码和解码函数。
* Makefile：用于构建和测试项目的Makefile。
* source.txt：示例输入文件。
* code.dat：编码后的二进制文件。
* Huffman.txt：保存Huffman编码的文件。
* recode.txt：解码后的输出文件。

## 使用方法

### 构建项目

在项目根目录下运行以下命令以构建项目：

make

### 编码文件

在项目根目录下运行以下命令以编码source.txt文件：

./huffman

### 测试项目

在项目根目录下运行以下命令以测试项目，测试将编码和解码source.txt文件：

make test

该命令将运行编码和解码过程，并使用diff命令检查recode.txt和source.txt是否相同。如果相同，将输出”Test passed!“，否则输出”Test failed!“。

### 清理项目

在项目根目录下运行以下命令以清理项目：

make clean

该命令将删除生成的目标文件和中间文件。

## 数据结构

### HuffmanNode

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

### MinHeap

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

## 算法设计思想

1. 计算每个字符的频率。
2. 使用这些频率构建一个最小堆。（插入一个EOF节点，频率为1）
3. 从最小堆中提取两个最小频率的节点，创建一个新节点，其频率为这两个节点频率之和，并将这两个节点作为新节点的子节点。
4. 将新节点插入最小堆。
5. 重复步骤3和4，直到堆中只剩下一个节点，这个节点就是Huffman树的根节点。
6. 通过遍历Huffman树生成每个字符的Huffman编码。
7. 使用生成的Huffman编码对文件进行编码和解码。

## 代码示例

### 编码文件

const char\* inputFile = "source.txt";  
const char\* encodedFile = "code.dat";  
const char\* decodedFile = "recode.txt";  
const char\* codeFile = "Huffman.txt";  
  
// 编码文件  
encodeFile(inputFile, encodedFile, codeFile);  
  
// 解码文件  
decodeFile(encodedFile, decodedFile, codeFile);

## 算法时间复杂度

Huffman编码算法的时间复杂度主要由以下几个部分组成：

1. 计算字符频率：，其中是输入字符的数量。
2. 构建最小堆：，其中是不同字符的数量。
3. 构建Huffman树：。
4. 生成Huffman编码：。
5. 编码文件：。
6. 解码文件：。

因此，Huffman编码算法的总体时间复杂度为。

## 改进方法

1. 使用更高效的数据结构，如斐波那契堆，可以进一步优化最小堆的构建和操作。
2. 在实际应用中，可以结合其他压缩算法（如LZW）以提高压缩效率。
3. 对于特定类型的数据，可以根据其特点设计专门的编码方案，以获得更好的压缩效果。

## 源代码

### ../huffman/huffman.c

#include "huffman.h"  
#include <string.h>  
#include <assert.h>  
  
#define EOF\_MARKER 0x00  
  
HuffmanNode \*createNode(char data, unsigned freq)  
{  
 HuffmanNode \*node = (HuffmanNode \*)malloc(sizeof(HuffmanNode));  
 node->data = data;  
 node->freq = freq;  
 node->left = node->right = NULL;  
 return node;  
}  
  
void swapHuffmanNode(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) // 需要旋转  
 {  
 swapHuffmanNode(&minHeap->array[smallest], &minHeap->array[idx]);  
 minHeapify(minHeap, smallest);  
 }  
}  
  
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 buildMinHeap(MinHeap \*minHeap)  
{  
 int n = minHeap->size - 1;  
 for (int i = (n - 1) / 2; i >= 0; --i)  
 {  
 minHeapify(minHeap, i);  
 }  
}  
  
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;  
}  
  
MinHeap \*createAndBuildMinHeap(char data[], int freq[], int size)  
{  
 MinHeap \*minHeap = createMinHeap(size);  
  
 for (int i = 0; i < size; ++i)  
 minHeap->array[i] = createNode(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 = createNode('$', left->freq + right->freq);  
 top->left = left;  
 top->right = right;  
  
 insertMinHeap(minHeap, top);  
 }  
  
 return extractMin(minHeap);  
}  
  
void calculateFrequencies(const char \*inputFile, int freq[])  
{  
 FILE \*file = fopen(inputFile, "r");  
 if (!file)  
 {  
 perror("Failed to open file");  
 exit(EXIT\_FAILURE);  
 }  
  
 char ch;  
 while ((ch = fgetc(file)) != EOF)  
 {  
 freq[(unsigned char)ch]++;  
 }  
  
 freq[EOF\_MARKER] = 1; // Add frequency for EOF marker  
  
 fclose(file);  
}  
  
void generateCodes(HuffmanNode \*root, int arr[], int top, char codes[][256])  
{  
 if (root->left)  
 {  
 arr[top] = 0;  
 generateCodes(root->left, arr, top + 1, codes);  
 }  
  
 if (root->right)  
 {  
 arr[top] = 1;  
 generateCodes(root->right, arr, top + 1, codes);  
 }  
  
 if (!root->left && !root->right)  
 {  
 char code[256];  
 for (int i = 0; i < top; ++i)  
 {  
 code[i] = arr[i] + '0';  
 }  
 code[top] = '\0';  
 strcpy(codes[(unsigned char)root->data], code);  
 }  
}  
  
void saveCodesToFile(char data[], char codes[][256], int size, const char \*codeFile)  
{  
 FILE \*file = fopen(codeFile, "w");  
 if (!file)  
 {  
 perror("Failed to open file");  
 exit(EXIT\_FAILURE);  
 }  
  
 for (int i = 0; i < size; i++)  
 {  
 fprintf(file, "%d %s\n", (unsigned char)data[i], codes[(unsigned char)data[i]]);  
 }  
  
 fprintf(file, "%d %s\n", EOF\_MARKER, codes[EOF\_MARKER]); // Save EOF marker  
  
 fclose(file);  
}  
  
static int bitBuffer = 0; //公共的writeBit 缓冲区，当剩余不足8位时，先不写入文件  
static int bitCount = 0;  
  
void writeBit(FILE \*file, int bit)  
{  
 bitBuffer = (bitBuffer << 1) | bit;  
 bitCount++;  
  
 if (bitCount == 8)  
 {  
 fputc(bitBuffer, file);  
 bitBuffer = 0;  
 bitCount = 0;  
 }  
}  
  
void flushBits(FILE \*file)  
{  
 if (bitCount > 0)  
 {  
 bitBuffer <<= (8 - bitCount);  
 fputc(bitBuffer, file);  
 bitBuffer = 0;  
 bitCount = 0;  
 }  
}  
  
void encodeFile(const char \*inputFile, const char \*outputFile, const char \*codeFile)  
{  
 int freq[257] = {0}; // 多一个位置用于存储EOF标记  
 calculateFrequencies(inputFile, freq);  
  
 char data[257];  
 int size = 0;  
 for (int i = 0; i < 257; i++)  
 {  
 if (freq[i] > 0)  
 {  
 data[size++] = (char)i;  
 }  
 }  
  
 HuffmanNode \*root = buildHuffmanTree(data, freq, size);  
  
 char codes[257][256] = {{0}}; // 编码表  
 int arr[256], top = 0;  
 generateCodes(root, arr, top, codes);  
  
 saveCodesToFile(data, codes, size, codeFile);  
  
 FILE \*input = fopen(inputFile, "r");  
 FILE \*output = fopen(outputFile, "wb");  
 if (!input || !output)  
 {  
 perror("Failed to open file");  
 exit(EXIT\_FAILURE);  
 }  
  
 char ch;  
 while ((ch = fgetc(input)) != EOF)  
 {  
 char \*code = codes[(unsigned char)ch];  
 for (int i = 0; code[i] != '\0'; i++)  
 {  
 writeBit(output, code[i] - '0');  
 }  
 }  
  
 // 写入EOF标记  
 char \*code = codes[EOF\_MARKER];  
 for (int i = 0; code[i] != '\0'; i++)  
 {  
 writeBit(output, code[i] - '0');  
 }  
  
 flushBits(output);  
  
 fclose(input);  
 fclose(output);  
}  
  
HuffmanNode \*buildHuffmanTreeFromCodes(char codes[257][256], char data[], int size)  
{  
 HuffmanNode \*root = createNode('$', 0);  
 for (int i = 0; i < size; i++)  
 {  
 HuffmanNode \*current = root;  
 char \*code = codes[(unsigned char)data[i]];  
 for (int j = 0; code[j] != '\0'; j++)  
 {  
 if (code[j] == '0')  
 {  
 if (!current->left)  
 {  
 current->left = createNode('$', 0);  
 }  
 current = current->left;  
 }  
 else  
 {  
 if (!current->right)  
 {  
 current->right = createNode('$', 0);  
 }  
 current = current->right;  
 }  
 }  
 current->data = data[i];  
 }  
 return root;  
}  
  
void decodeFile(const char \*inputFile, const char \*outputFile, const char \*codeFile)  
{  
 FILE \*codeFilePtr = fopen(codeFile, "r");  
 if (!codeFilePtr)  
 {  
 perror("Failed to open code file");  
 exit(EXIT\_FAILURE);  
 }  
  
 char codes[257][256] = {{0}};  
 char data[257];  
 int size = 0;  
 int ascii;  
 char code[256];  
  
 char line[256];  
 while (fgets(line, sizeof(line), codeFilePtr))  
 {  
 if (sscanf(line, "%d %s", &ascii, code) == 2)  
 {  
 char ch = (char)ascii;  
 strcpy(codes[(unsigned char)ch], code);  
 data[size++] = ch;  
 }  
 }  
 fclose(codeFilePtr);  
  
 FILE \*input = fopen(inputFile, "rb");  
 FILE \*output = fopen(outputFile, "w");  
 if (!input || !output)  
 {  
 perror("Failed to open file");  
 exit(EXIT\_FAILURE);  
 }  
  
 HuffmanNode \*root = buildHuffmanTreeFromCodes(codes, data, size);  
 HuffmanNode \*current = root;  
 int bit;  
 while ((bit = fgetc(input)) != EOF)  
 {  
 for (int i = 7; i >= 0; i--)  
 {  
 int currentBit = (bit >> i) & 1;  
 if (currentBit == 0)  
 {  
 current = current->left;  
 }  
 else  
 {  
 current = current->right;  
 }  
 assert(current != NULL); // 保证编码表正确  
 if (!current->left && !current->right)  
 {  
 if ((unsigned char)current->data == EOF\_MARKER) // 到达EOF标记  
 {  
 fclose(input);  
 fclose(output);  
 return;  
 }  
 fputc(current->data, output);  
 current = root;  
 }  
 }  
 }  
  
 fclose(input);  
 fclose(output);  
}

### ../huffman/huffman.h

#ifndef HUFFMAN\_H  
#define HUFFMAN\_H  
  
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct HuffmanNode {  
 char data;  
 unsigned freq;  
 struct HuffmanNode \*left, \*right;  
} HuffmanNode;  
  
typedef struct MinHeap {  
 unsigned size;  
 unsigned capacity;  
 HuffmanNode \*\*array;  
} MinHeap;  
  
HuffmanNode\* createNode(char data, unsigned freq);  
MinHeap\* createMinHeap(unsigned capacity);  
void insertMinHeap(MinHeap\* minHeap, HuffmanNode\* node);  
HuffmanNode\* extractMin(MinHeap\* minHeap);  
void buildMinHeap(MinHeap\* minHeap);  
void printCodes(HuffmanNode\* root, int arr[], int top, FILE\* output);  
void HuffmanCodes(char data[], int freq[], int size, FILE\* output);  
void encodeFile(const char\* inputFile, const char\* outputFile, const char\* codeFile);  
void decodeFile(const char\* inputFile, const char\* outputFile, const char\* codeFile);  
  
#endif // HUFFMAN\_H

### ../huffman/main.c

#include "huffman.h"  
  
int main() {  
 const char\* inputFile = "source.txt";  
 const char\* encodedFile = "code.dat";  
 const char\* decodedFile = "recode.txt"; //为什么不是decode呢?  
 const char\* codeFile = "Huffman.txt";  
  
 // 编码文件  
 encodeFile(inputFile, encodedFile, codeFile);  
  
 // 解码文件  
 decodeFile(encodedFile, decodedFile, codeFile);  
  
 return 0;  
}

# 6、地铁修建 (必做) （图）

## 编译运行

make test

## 采用的数据结构

本项目采用了以下数据结构： - **边结构体（Edge）**：用于存储每条隧道的信息，包括两个交通枢纽和施工时间。 - **并查集（Union-Find）**：用于管理交通枢纽的连通性，帮助判断是否形成环。

## 算法设计思想

本项目使用了Kruskal算法来解决最小生成树问题。具体步骤如下： 1. 读取输入数据，存储为边结构体的数组。 2. 对边数组按施工时间从小到大排序。 3. 初始化并查集。 4. 遍历排序后的边数组，逐条加入最小生成树，直到1号枢纽和n号枢纽连通。 5. 输出最小生成树中最大边的权重，即修建整条地铁线路最少需要的天数。

## 测试数据和结果

### 测试数据

6 6  
1 2 4  
2 3 4  
3 6 7  
1 4 2  
4 5 5  
5 6 6

### 测试结果

6

## 算法的时间复杂度

Kruskal算法的时间复杂度主要由排序和并查集操作决定： - 排序的时间复杂度为O(m log m)，其中m是边的数量。 - 并查集操作的时间复杂度接近于O(1)，因此整体时间复杂度为O(m log m)。

## 结束语

本项目实现了一个基于Kruskal算法的最小生成树求解程序，成功解决了地铁修建问题。程序代码行数为约50行，测试数据和结果均符合预期。通过本次项目，我深入理解了Kruskal算法和并查集的应用，提升了算法设计和实现能力。

## 源代码

### ../subway/subway.cpp

#include <iostream>  
#include <vector>  
#include <algorithm>  
  
using namespace std;  
  
struct Edge {  
 int u, v, weight;  
 bool operator<(const Edge& other) const {  
 return weight < other.weight;  
 }  
};  
  
int find(vector<int>& parent, int u) {  
 if (parent[u] != u) {  
 parent[u] = find(parent, parent[u]);  
 }  
 return parent[u];  
}  
  
void unite(vector<int>& parent, vector<int>& rank, int u, int v) {  
 int rootU = find(parent, u);  
 int rootV = find(parent, v);  
 if (rootU != rootV) {  
 if (rank[rootU] > rank[rootV]) {  
 parent[rootV] = rootU;  
 } else if (rank[rootU] < rank[rootV]) {  
 parent[rootU] = rootV;  
 } else {  
 parent[rootV] = rootU;  
 rank[rootU]++;  
 }  
 }  
}  
  
int main() {  
 int n, m;  
 cin >> n >> m;  
 vector<Edge> edges(m);  
 for (int i = 0; i < m; ++i) {  
 cin >> edges[i].u >> edges[i].v >> edges[i].weight;  
 }  
  
 sort(edges.begin(), edges.end());  
  
 vector<int> parent(n + 1), rank(n + 1, 0);  
 for (int i = 1; i <= n; ++i) {  
 parent[i] = i;  
 }  
  
 int maxWeight = 0;  
 for (const auto& edge : edges) {  
 if (find(parent, edge.u) != find(parent, edge.v)) {  
 unite(parent, rank, edge.u, edge.v);  
 maxWeight = max(maxWeight, edge.weight);  
 if (find(parent, 1) == find(parent, n)) {  
 break;  
 }  
 }  
 }  
  
 cout << maxWeight << endl;  
 return 0;  
}

# 7、公交线路提示 (必做) （图）

## 数据结构

1. **Station**: 表示一个公交站点，包含站点名称、站点ID和该站点所属的线路及在线路中的编号。

struct Station {  
 string name;  
 int stationId;  
 vector<pair<int, int>> lines; // <lineId, InlineId>  
};

1. **Line**: 表示一条公交线路，包含线路名称、线路ID和该线路经过的站点ID。

struct Line {  
 string name;  
 int lineId;  
 vector<int> stationIds;  
};

## 算法设计思想

1. **读取CSV文件**: 读取CSV文件并解析每一行数据，构建站点和线路的映射关系。

void readCSV(const string &filename);

1. **最少转车次数的乘车路线**: 使用Dijkstra算法，计算从起始站点到终点站点的最少转车次数的路线。

vector<pair<string, string>> findTransferLeastPath(const int startStationId, const int endStationId);

1. **经过站点数量最少的乘车路线**: 使用Dijkstra算法，计算从起始站点到终点站点的经过站点数量最少的路线。

vector<pair<string, string>> findLeastStationPath(const int startStationId, const int endStationId);

## 测试数据和结果

### 测试数据

CSV文件内容如下：

线路名,站点,公交线路ID,线内ID,站点ID  
1路(建康路·夫子庙--南堡公园),建康路·夫子庙,0,0,1942  
1路(建康路·夫子庙--南堡公园),长白街,0,1,6196  
1路(建康路·夫子庙--南堡公园),太平巷,0,2,4319  
...

完整的CSV文件见 nanjing.csv。

### 测试结果

输入起始站点和终点站点，程序输出最少转车次数和经过站点数量最少的乘车路线。

请输入起始站点: 南航北门  
请输入终点站点: 理工大  
---------------------------------  
最少转车次数: 1  
路线:   
南航北门  
 |  
 |  
 | 乘坐 719路(武夷绿洲--南京南站)  
 |  
 |  
茶花路  
 |  
 |  
 | 乘坐 84路(南京南站--理工大)  
 |  
 |  
理工大  
---------------------------------  
经过站点数量最少: 8  
路线:   
南航北门  
 |  
 |  
 | 乘坐 719路(武夷绿洲--南京南站)  
 |  
 |  
翠屏山  
 |  
 |  
 | 乘坐 711路(综保区场站--安德门)  
 |  
 |  
托乐嘉  
 |  
 |  
 | 乘坐 711路(综保区场站--安德门)  
 |  
 |  
麻田路  
 |  
 |  
 | 乘坐 768路(宁杭高铁江宁站--安德门)  
 |  
 |  
景明佳园  
 |  
 |  
 | 乘坐 61路(景明佳园--江东中路·江东门北街)  
 |  
 |  
应天大街  
 |  
 |  
 | 乘坐 306路(楠溪江东街--银城东苑)  
 |  
 |  
莫愁湖公园南门  
 |  
 |  
 | 乘坐 306路(楠溪江东街--银城东苑)  
 |  
 |  
瑞金路  
 |  
 |  
 | 乘坐 Y14路夜间(中花岗总站--公交四公司)  
 |  
 |  
孝陵卫  
 |  
 |  
 | 乘坐 5路(南湾营--长江路·估衣廊)  
 |  
 |  
理工大

## 算法的时间复杂度

1. **最少转车次数的乘车路线**: 使用Dijkstra算法，时间复杂度为 ，其中 是边的数量， 是顶点的数量。具体来说：
   * 初始化距离数组和优先队列的时间复杂度为 。
   * 每个顶点最多会被加入优先队列一次，加入和取出优先队列的时间复杂度为 ，因此总的时间复杂度为 。
   * 每条边最多会被处理一次，处理每条边的时间复杂度为 ，因此总的时间复杂度为 。
   * 综合起来，Dijkstra算法的时间复杂度为 。
2. **经过站点数量最少的乘车路线**: 使用Dijkstra算法，时间复杂度为 ，其中 是边的数量， 是顶点的数量。具体来说：
   * 初始化距离数组和优先队列的时间复杂度为 。
   * 每个顶点最多会被加入优先队列一次，加入和取出优先队列的时间复杂度为 ，因此总的时间复杂度为 。
   * 每条边最多会被处理一次，处理每条边的时间复杂度为 ，因此总的时间复杂度为 。
   * 综合起来，Dijkstra算法的时间复杂度为 。

## 算法的改进方法

1. **启发式搜索**: 使用A\*算法等启发式搜索算法来提高搜索效率。
2. **并行计算**: 使用多线程或并行计算技术来加速路径搜索过程。

## 源代码

### ../bus/main.cpp

#include <iostream>  
#include <fstream>  
#include <sstream>  
#include <vector>  
#include <unordered\_map>  
#include <queue>  
#include <limits>  
#include <algorithm>  
#include <cassert>  
  
using namespace std;  
  
const int MAX\_STATION = 10000;  
const int MAX\_LINE = 1000;  
  
struct Station  
{  
 string name;  
 int stationId;  
 vector<pair<int, int>> lines; // <lineId, InlineId> 该站点所属的线路, 以及在线路中的编号  
 bool operator<(const Station &other) const  
 {  
 return stationId < other.stationId;  
 }  
 Station()  
 {  
 stationId = -1;  
 }  
};  
  
struct Line  
{ // 公交线路  
 string name;  
 int lineId;  
 vector<int> stationIds;  
 bool operator<(const Line &other) const  
 {  
 return lineId < other.lineId;  
 }  
 Line()  
 {  
 lineId = -1;  
 }  
};  
  
vector<Station> stations(MAX\_STATION);  
vector<Line> lines(MAX\_LINE);  
  
void readCSV(const string &filename)  
{  
 ifstream file(filename);  
 string line, word;  
  
 // Skip the header  
 getline(file, line);  
  
 while (getline(file, line))  
 {  
 stringstream ss(line);  
 vector<string> row;  
 while (getline(ss, word, ','))  
 {  
 row.push\_back(word);  
 }  
 string lineName = row[0];  
 string stationName = row[1];  
 int lineId = stoi(row[2]);  
 int InlineId = stoi(row[3]);  
 int stationId = stoi(row[4]);  
  
 if (stations[stationId].stationId == -1)  
 {  
 stations[stationId].name = stationName;  
 stations[stationId].stationId = stationId;  
 }  
 stations[stationId].lines.push\_back({lineId, InlineId});  
 if (lines[lineId].lineId == -1)  
 {  
 lines[lineId].name = lineName;  
 lines[lineId].lineId = lineId;  
 }  
 lines[lineId].stationIds.push\_back(stationId);  
 // cerr << "lineId: " << lineId << " stationId: " << stationId << " InlineId: " << InlineId << endl;  
 assert(lines[lineId].stationIds.size() == InlineId + 1); // 检查站点编号是否连续  
 }  
}  
  
// 转车次数最少的乘车路线  
vector<pair<string, string>> findTransferLeastPath(const int startStationId, const int endStationId)  
{  
 vector<pair<string, string>> path; // <stationName, lineName>  
 vector<int> dist(MAX\_STATION, numeric\_limits<int>::max());  
 vector<pair<int, int>> prev(MAX\_STATION, {-1, -1}); // <stationId, lineId>  
 priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;  
  
 dist[startStationId] = 0;  
 pq.push({0, startStationId});  
  
 while (!pq.empty())  
 {  
 int u = pq.top().second;  
 pq.pop();  
  
 if (u == endStationId)  
 {  
 break;  
 }  
  
 for (const auto &line : stations[u].lines)  
 {  
 int lineId = line.first;  
 for (const auto &stationId : lines[lineId].stationIds)  
 {  
 if (dist[stationId] > dist[u] + 1)  
 {  
 dist[stationId] = dist[u] + 1;  
 prev[stationId] = make\_pair(u, lineId);  
 pq.push({dist[stationId], stationId});  
 }  
 }  
 }  
 }  
  
 if (prev[endStationId] == make\_pair(-1, -1))  
 {  
 return path;  
 }  
  
 for (int v = endStationId; v != -1; v = prev[v].first)  
 {  
 path.push\_back(make\_pair(stations[v].name, lines[prev[v].second].name));  
 }  
 reverse(path.begin(), path.end());  
 return path;  
}  
  
// 经过站点数量最少的乘车路线  
vector<pair<string, string>> findLeastStationPath(const int startStationId, const int endStationId)  
{  
 vector<pair<string, string>> path; // <stationName, lineName>  
 vector<int> dist(MAX\_STATION, numeric\_limits<int>::max());  
 vector<pair<int, int>> prev(MAX\_STATION, {-1, -1}); // <stationId, lineId>  
 priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;  
  
 dist[startStationId] = 0;  
 pq.push({0, startStationId});  
  
 while (!pq.empty())  
 {  
 int u = pq.top().second;  
 pq.pop();  
  
 if (u == endStationId)  
 {  
 break;  
 }  
  
 for (const auto &line : stations[u].lines)  
 {  
 int lineId = line.first;  
 int inlineId = line.second;  
 if (inlineId + 1 < lines[lineId].stationIds.size())  
 {  
 int v = lines[lineId].stationIds[inlineId + 1];  
 if (dist[v] > dist[u] + 1)  
 {  
 dist[v] = dist[u] + 1;  
 prev[v] = make\_pair(u, lineId);  
 pq.push({dist[v], v});  
 }  
 }  
 if (inlineId - 1 >= 0)  
 {  
 int v = lines[lineId].stationIds[inlineId - 1];  
 if (dist[v] > dist[u] + 1)  
 {  
 dist[v] = dist[u] + 1;  
 prev[v] = make\_pair(u, lineId);  
 pq.push({dist[v], v});  
 }  
 }  
 }  
 }  
  
 if (prev[endStationId] == make\_pair(-1, -1))  
 {  
 return path;  
 }  
  
 for (int v = endStationId; v != -1; v = prev[v].first)  
 {  
 path.push\_back(make\_pair(stations[v].name, lines[prev[v].second].name));  
 }  
 reverse(path.begin(), path.end());  
 return path;  
}  
  
int main()  
{  
 readCSV("nanjing.csv");  
  
 string start, end;  
 cout << "请输入起始站点: ";  
 cin >> start;  
 cout << "请输入终点站点: ";  
 cin >> end;  
  
 int startStationId = -1, endStationId = -1;  
 for (const auto &station : stations)  
 {  
 if (station.name == start)  
 {  
 startStationId = station.stationId;  
 }  
 if (station.name == end)  
 {  
 endStationId = station.stationId;  
 }  
 }  
 assert(startStationId != -1 && endStationId != -1);  
 cout << "---------------------------------" << endl;  
 auto path = findTransferLeastPath(startStationId, endStationId);  
 if (path.empty())  
 {  
 cout << "无法到达终点站点" << endl;  
 }  
 else  
 {  
 cout << "最少转车次数: " << path.size() - 2 << endl;  
 cout << "路线: " << endl;  
 for (size\_t i = 0; i < path.size(); ++i)  
 {  
 if (i > 0)  
 {  
 cout << " |" << endl;  
 cout << " |" << endl;  
 cout << " | 乘坐 " << path[i].second << endl;  
 cout << " |" << endl;  
 cout << " |" << endl;  
 }  
 cout << path[i].first << endl;  
 }  
 }  
 cout << "---------------------------------" << endl;  
 path = findLeastStationPath(startStationId, endStationId);  
 if (path.empty())  
 {  
 cout << "无法到达终点站点" << endl;  
 }  
 else  
 {  
 cout << "经过站点数量最少: " << path.size() - 2 << endl;  
 cout << "路线: " << endl;  
 for (size\_t i = 0; i < path.size(); ++i)  
 {  
 if (i > 0)  
 {  
 cout << " |" << endl;  
 cout << " |" << endl;  
 cout << " | 乘坐 " << path[i].second << endl;  
 cout << " |" << endl;  
 cout << " |" << endl;  
 }  
 cout << path[i].first << endl;  
 }  
 }  
 return 0;  
}

# 8、B-树应用 （必做） （查找）

## 问题描述

设计并实现B-树的动态查找。

## 基本要求

1. 从文件读取数据
2. 实现B-树的三种基本功能：查找、插入和删除。
3. 以可验证的方式输出结果

## 数据结构

我们使用B-树（B-Tree）来存储和管理数据。B-树是一种自平衡的树数据结构，能够保持数据有序，并允许高效的插入、删除和查找操作。

### B-树节点结构

typedef struct BTreeNode {  
 int keys[M - 1];  
 struct BTreeNode \*children[M];  
 int numKeys;  
 int isLeaf;  
} BTreeNode;

### B-树结构

typedef struct BTree {  
 BTreeNode \*root;  
} BTree;

## 算法设计思想

1. **查找**：从根节点开始，逐层向下查找，直到找到目标关键字或到达叶子节点。
2. **插入**：在非满节点中插入关键字，如果节点已满，则需要分裂节点并递归插入。
3. **删除**：从节点中删除关键字，如果节点关键字数不足，则需要合并或借用兄弟节点的关键字。

## 实现的源程序

源程序包括以下文件：

* prime\_gen.c：生成1到10000的所有质数，并将其写入primes.txt文件。
* main.c：从文件读取质数，构建B-树，执行查询和删除操作，并将结果写入文件。
* b\_tree.h：定义B-树的结构和基本操作函数。
* b\_tree.c：实现B-树的创建、插入、删除、查找和释放内存的功能。
* Makefile：编译和链接程序的规则。

## 测试数据和结果

### 生成质数

运行prime\_gen程序生成primes.txt文件，包含1到10000的所有质数。

### 查询和删除操作

1. 查询200-300的每个数是否在B-树中，将结果写入b-tree1.txt。
2. 删除500-2000中的每个质数，再查询600-700之间的每个质数是否在，将结果写入b-tree2.txt。
3. 插入1-1000的所有偶数，查询100-200的每个偶数是否在B-树中，将结果写入b-tree3.txt。

### 结果文件格式

XXX no  
XXX yes

## 算法的时间复杂度

* **查找**：O(log n)
* **插入**：O(log n)
* **删除**：O(log n)

## 改进方法

1. **优化节点分裂和合并**：通过调整节点分裂和合并的策略，可以减少树的高度，提高操作效率。（如：2-3树、2-3-4树）
2. **并行化操作**：在多核处理器上，可以并行化B-树的操作，提高性能。

## 源代码

### ../b\_tree/b\_tree.c

#include "b\_tree.h"  
#include <stdio.h>  
#include <stdlib.h>  
  
BTreeNode\* createNode(int isLeaf) {  
 BTreeNode\* node = (BTreeNode\*)malloc(sizeof(BTreeNode));  
 node->isLeaf = isLeaf;  
 node->numKeys = 0;  
 for (int i = 0; i < M; i++) {  
 node->children[i] = NULL;  
 }  
 return node;  
}  
  
BTree\* createBTree() {  
 BTree\* tree = (BTree\*)malloc(sizeof(BTree));  
 tree->root = NULL;  
 return tree;  
}  
  
void splitChild(BTreeNode\* parent, int i, BTreeNode\* child) {  
 BTreeNode\* newChild = createNode(child->isLeaf);  
 newChild->numKeys = M / 2 - 1;  
  
 for (int j = 0; j < M / 2 - 1; j++) {  
 newChild->keys[j] = child->keys[j + M / 2];  
 }  
  
 if (!child->isLeaf) {  
 for (int j = 0; j < M / 2; j++) {  
 newChild->children[j] = child->children[j + M / 2];  
 }  
 }  
  
 child->numKeys = M / 2 - 1;  
  
 for (int j = parent->numKeys; j >= i + 1; j--) {  
 parent->children[j + 1] = parent->children[j];  
 }  
  
 parent->children[i + 1] = newChild;  
  
 for (int j = parent->numKeys - 1; j >= i; j--) {  
 parent->keys[j + 1] = parent->keys[j];  
 }  
  
 parent->keys[i] = child->keys[M / 2 - 1];  
 parent->numKeys++;  
}  
  
void insertNonFull(BTreeNode\* node, int key) {  
 int i = node->numKeys - 1;  
  
 if (node->isLeaf) {  
 while (i >= 0 && key < node->keys[i]) {  
 node->keys[i + 1] = node->keys[i];  
 i--;  
 }  
 node->keys[i + 1] = key;  
 node->numKeys++;  
 } else {  
 while (i >= 0 && key < node->keys[i]) {  
 i--;  
 }  
 i++;  
 if (node->children[i]->numKeys == M - 1) {  
 splitChild(node, i, node->children[i]);  
 if (key > node->keys[i]) {  
 i++;  
 }  
 }  
 insertNonFull(node->children[i], key);  
 }  
}  
  
void insert(BTree\* tree, int key) {  
 if (tree->root == NULL) {  
 tree->root = createNode(1);  
 tree->root->keys[0] = key;  
 tree->root->numKeys = 1;  
 } else {  
 if (tree->root->numKeys == M - 1) {  
 BTreeNode\* newRoot = createNode(0);  
 newRoot->children[0] = tree->root;  
 splitChild(newRoot, 0, tree->root);  
 int i = 0;  
 if (newRoot->keys[0] < key) {  
 i++;  
 }  
 insertNonFull(newRoot->children[i], key);  
 tree->root = newRoot;  
 } else {  
 insertNonFull(tree->root, key);  
 }  
 }  
}  
  
void merge(BTreeNode\* node, int idx) {  
 BTreeNode\* child = node->children[idx];  
 BTreeNode\* sibling = node->children[idx + 1];  
  
 child->keys[M / 2 - 1] = node->keys[idx];  
  
 for (int i = 0; i < sibling->numKeys; ++i) {  
 child->keys[i + M / 2] = sibling->keys[i];  
 }  
  
 if (!child->isLeaf) {  
 for (int i = 0; i <= sibling->numKeys; ++i) {  
 child->children[i + M / 2] = sibling->children[i];  
 }  
 }  
  
 for (int i = idx + 1; i < node->numKeys; ++i) {  
 node->keys[i - 1] = node->keys[i];  
 }  
  
 for (int i = idx + 2; i <= node->numKeys; ++i) {  
 node->children[i - 1] = node->children[i];  
 }  
  
 child->numKeys += sibling->numKeys + 1;  
 node->numKeys--;  
  
 free(sibling);  
}  
  
void fill(BTreeNode\* node, int idx) {  
 if (idx != 0 && node->children[idx - 1]->numKeys >= M / 2) {  
 BTreeNode\* child = node->children[idx];  
 BTreeNode\* sibling = node->children[idx - 1];  
  
 for (int i = child->numKeys - 1; i >= 0; --i) {  
 child->keys[i + 1] = child->keys[i];  
 }  
  
 if (!child->isLeaf) {  
 for (int i = child->numKeys; i >= 0; --i) {  
 child->children[i + 1] = child->children[i];  
 }  
 }  
  
 child->keys[0] = node->keys[idx - 1];  
  
 if (!child->isLeaf) {  
 child->children[0] = sibling->children[sibling->numKeys];  
 }  
  
 node->keys[idx - 1] = sibling->keys[sibling->numKeys - 1];  
  
 child->numKeys += 1;  
 sibling->numKeys -= 1;  
 } else if (idx != node->numKeys && node->children[idx + 1]->numKeys >= M / 2) {  
 BTreeNode\* child = node->children[idx];  
 BTreeNode\* sibling = node->children[idx + 1];  
  
 child->keys[child->numKeys] = node->keys[idx];  
  
 if (!child->isLeaf) {  
 child->children[child->numKeys + 1] = sibling->children[0];  
 }  
  
 node->keys[idx] = sibling->keys[0];  
  
 for (int i = 1; i < sibling->numKeys; ++i) {  
 sibling->keys[i - 1] = sibling->keys[i];  
 }  
  
 if (!sibling->isLeaf) {  
 for (int i = 1; i <= sibling->numKeys; ++i) {  
 sibling->children[i - 1] = sibling->children[i];  
 }  
 }  
  
 child->numKeys += 1;  
 sibling->numKeys -= 1;  
 } else {  
 if (idx != node->numKeys) {  
 merge(node, idx);  
 } else {  
 merge(node, idx - 1);  
 }  
 }  
}  
  
void removeFromLeaf(BTreeNode\* node, int idx) {  
 for (int i = idx + 1; i < node->numKeys; ++i) {  
 node->keys[i - 1] = node->keys[i];  
 }  
 node->numKeys--;  
}  
  
void deleteNode(BTreeNode\* node, int key);  
  
void removeFromNonLeaf(BTreeNode\* node, int idx) {  
 int key = node->keys[idx];  
  
 if (node->children[idx]->numKeys >= M / 2) {  
 BTreeNode\* cur = node->children[idx];  
 while (!cur->isLeaf) {  
 cur = cur->children[cur->numKeys];  
 }  
 int pred = cur->keys[cur->numKeys - 1];  
 node->keys[idx] = pred;  
 deleteNode(node->children[idx], pred);  
 } else if (node->children[idx + 1]->numKeys >= M / 2) {  
 BTreeNode\* cur = node->children[idx + 1];  
 while (!cur->isLeaf) {  
 cur = cur->children[0];  
 }  
 int succ = cur->keys[0];  
 node->keys[idx] = succ;  
 deleteNode(node->children[idx + 1], succ);  
 } else {  
 merge(node, idx);  
 deleteNode(node->children[idx], key);  
 }  
}  
  
void deleteNode(BTreeNode\* node, int key) {  
 int idx = 0;  
 while (idx < node->numKeys && node->keys[idx] < key) {  
 ++idx;  
 }  
  
 if (idx < node->numKeys && node->keys[idx] == key) {  
 if (node->isLeaf) {  
 removeFromLeaf(node, idx);  
 } else {  
 removeFromNonLeaf(node, idx);  
 }  
 } else {  
 if (node->isLeaf) {  
 return;  
 }  
  
 int flag = (idx == node->numKeys);  
  
 if (node->children[idx]->numKeys < M / 2) {  
 fill(node, idx);  
 }  
  
 if (flag && idx > node->numKeys) {  
 deleteNode(node->children[idx - 1], key);  
 } else {  
 deleteNode(node->children[idx], key);  
 }  
 }  
}  
  
void delete(BTree\* tree, int key) {  
 if (!tree->root) {  
 return;  
 }  
  
 deleteNode(tree->root, key);  
  
 if (tree->root->numKeys == 0) {  
 BTreeNode\* tmp = tree->root;  
 if (tree->root->isLeaf) {  
 tree->root = NULL;  
 } else {  
 tree->root = tree->root->children[0];  
 }  
 free(tmp);  
 }  
}  
  
int searchNode(BTreeNode\* node, int key) {  
 int i = 0;  
 while (i < node->numKeys && key > node->keys[i]) {  
 i++;  
 }  
 if (i < node->numKeys && key == node->keys[i]) {  
 return 1;  
 }  
 if (node->isLeaf) {  
 return 0;  
 }  
 return searchNode(node->children[i], key);  
}  
  
int search(BTree\* tree, int key) {  
 if (tree->root == NULL) {  
 return 0;  
 }  
 return searchNode(tree->root, key);  
}  
  
void freeNode(BTreeNode\* node) {  
 if (!node->isLeaf) {  
 for (int i = 0; i <= node->numKeys; i++) {  
 freeNode(node->children[i]);  
 }  
 }  
 free(node);  
}  
  
void freeBTree(BTree\* tree) {  
 if (tree->root != NULL) {  
 freeNode(tree->root);  
 }  
 free(tree);  
}

### ../b\_tree/b\_tree.h

#ifndef B\_TREE\_H  
#define B\_TREE\_H  
  
#define M 4 // B-tree order  
  
typedef struct BTreeNode {  
 int keys[M - 1];  
 struct BTreeNode \*children[M];  
 int numKeys;  
 int isLeaf;  
} BTreeNode;  
  
typedef struct BTree {  
 BTreeNode \*root;  
} BTree;  
  
BTree\* createBTree();  
void insert(BTree \*tree, int key);  
void delete(BTree \*tree, int key);  
int search(BTree \*tree, int key);  
void freeBTree(BTree \*tree);  
  
#endif // B\_TREE\_H

### ../b\_tree/main.c

#include "b\_tree.h"  
#include <stdio.h>  
#include <stdlib.h>  
  
void readPrimes(int \*primes, int size);  
void writeResults(const char \*filename, int \*results, int size, int start);  
  
int main() {  
 BTree \*tree = createBTree();  
 int primes[10000];  
 readPrimes(primes, 10000);  
  
 // Build B-tree with primes  
 for (int i = 0; i < 10000; i++) {  
 insert(tree, primes[i]);  
 }  
  
 // Query 200-300  
 int results[101];  
 for (int i = 200; i <= 300; i++) {  
 results[i - 200] = search(tree, i);  
 }  
 writeResults("b-tree1.txt", results, 101, 200);  
  
 // Delete primes 500-2000  
 for (int i = 500; i <= 2000; i++) {  
 if (search(tree, i)) {  
 delete(tree, i);  
 }  
 }  
  
 // Query 600-700  
 for (int i = 600; i <= 700; i++) {  
 results[i - 600] = search(tree, i);  
 }  
 writeResults("b-tree2.txt", results, 101, 600);  
  
 // Insert even numbers 1-1000  
 for (int i = 2; i <= 1000; i += 2) {  
 insert(tree, i);  
 }  
  
 // Query 100-200 even numbers  
 for (int i = 100; i <= 200; i += 2) {  
 results[(i - 100) / 2] = search(tree, i);  
 }  
 writeResults("b-tree3.txt", results, 51, 100);  
  
 freeBTree(tree);  
 return 0;  
}  
  
void readPrimes(int \*primes, int size) {  
 FILE \*file = fopen("primes.txt", "r");  
 if (file == NULL) {  
 perror("Failed to open primes.txt");  
 exit(EXIT\_FAILURE);  
 }  
 for (int i = 0; i < size; i++) {  
 if (fscanf(file, "%d", &primes[i]) != 1) {  
 perror("Failed to read prime number");  
 fclose(file);  
 exit(EXIT\_FAILURE);  
 }  
 }  
 fclose(file);  
}  
  
void writeResults(const char \*filename, int \*results, int size, int start) {  
 FILE \*file = fopen(filename, "w");  
 for (int i = 0; i < size; i++) {  
 fprintf(file, "%d %s\n", start + i, results[i] ? "yes" : "no");  
 }  
 fclose(file);  
}

### ../b\_tree/prime\_gen.c

#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
  
int isPrime(int num) {  
 if (num <= 1) return 0;  
 if (num <= 3) return 1;  
 if (num % 2 == 0 || num % 3 == 0) return 0;  
 for (int i = 5; i \* i <= num; i += 6) {  
 if (num % i == 0 || num % (i + 2) == 0) return 0;  
 }  
 return 1;  
}  
  
int main() {  
 FILE \*file = fopen("primes.txt", "w");  
 if (file == NULL) {  
 perror("Failed to create primes.txt");  
 return EXIT\_FAILURE;  
 }  
  
 int count = 0;  
 int num = 2;  
 while (count < 10000) {  
 if (isPrime(num)) {  
 fprintf(file, "%d\n", num);  
 count++;  
 }  
 num++;  
 }  
  
 fclose(file);  
 return 0;  
}

# 9、排序算法比较 （必做）（排序）

## 问题描述

利用随机函数产生10个样本，每个样本有100000个随机整数（并使第一个样本是正序，第二个样本是逆序），利用直接插入排序、希尔排序，冒泡排序、快速排序、选择排序、堆排序，归并排序、基数排序8种排序方法进行排序（结果为由小到大的顺序），并统计每一种排序算法对不同样本所耗费的时间。

## 要求

1. 原始数据存在文件中，用相同样本对不同算法进行测试；
2. 屏幕显示每种排序算法对不同样本所花的时间；
3. 任意选择1个样本完成数字排序，即统计出每个整数出现的次数，按出现次数从多到少的顺序输出（输出的信息包含数字和出现的频率）。提示：可用一个数组，以下标作为数，数组内容存储该数出现次数来实现（这就相当于直接映射，是机试题目里面常用的一种解题法，很多看似非线性的题型最后其实都可以采取哈希或者映射的方法来巧解，体会哈希思想在机试题目中的巧用）。

## 数据结构

使用 std::vector<int> 来存储样本数据。

## 算法设计思想

1. 生成10个样本，每个样本包含100000个随机整数，并将它们存储在文件中。
2. 实现8种排序算法：直接插入排序、希尔排序、冒泡排序、快速排序、选择排序、堆排序、归并排序、基数排序。
3. 对每个样本使用不同的排序算法进行排序，并统计每种算法的耗时。
4. 显示每种排序算法对不同样本所花的时间。
5. 对任意选择的一个样本进行数字排序，并统计每个整数出现的次数，按出现次数从多到少的顺序输出。

## 测试数据和结果

程序生成10个样本数据文件，每个文件包含100000个随机整数。对每个样本使用8种排序算法进行排序，并统计每种算法的耗时。以下是部分测试结果示例：

====样本1======  
直接插入排序排序耗时: 0.00053369秒  
希尔排序排序耗时: 0.00685221秒  
冒泡排序排序耗时: 17.5284秒  
快速排序排序耗时: 37.1794秒  
选择排序排序耗时: 17.3865秒  
堆排序排序耗时: 0.0254844秒  
归并排序排序耗时: 0.0227283秒  
基数排序排序耗时: 0.0163487秒  
====样本2======  
直接插入排序排序耗时: 17.9416秒  
希尔排序排序耗时: 0.00990094秒  
冒泡排序排序耗时: 44.5773秒  
快速排序排序耗时: 21.6641秒  
选择排序排序耗时: 16.859秒  
堆排序排序耗时: 0.0239351秒  
归并排序排序耗时: 0.0226194秒  
基数排序排序耗时: 0.0159799秒  
====样本3======  
直接插入排序排序耗时: 8.96412秒  
希尔排序排序耗时: 0.0278527秒  
冒泡排序排序耗时: 36.0912秒  
快速排序排序耗时: 0.016276秒  
选择排序排序耗时: 17.4401秒  
堆排序排序耗时: 0.029981秒  
归并排序排序耗时: 0.0316348秒  
基数排序排序耗时: 0.0162168秒  
====样本4======  
直接插入排序排序耗时: 9.04911秒  
希尔排序排序耗时: 0.0281864秒  
冒泡排序排序耗时: 36.6653秒  
快速排序排序耗时: 0.0152831秒  
选择排序排序耗时: 17.5027秒  
堆排序排序耗时: 0.0301401秒  
归并排序排序耗时: 0.0317878秒  
基数排序排序耗时: 0.0166479秒

## 算法的时间复杂度

* 直接插入排序:
* 希尔排序:
* 冒泡排序:
* 快速排序: 最坏情况
* 选择排序:
* 堆排序:
* 归并排序:
* 基数排序:

## 改进方法

对于快速排序，可以使用三向切分快速排序来优化处理重复元素较多的情况。对于基数排序，可以使用更高效的计数排序来进一步优化。

## 源代码

### ../sort/main.cpp

#include <iostream>  
#include <fstream>  
#include <vector>  
#include <algorithm>  
#include <random>  
#include <chrono>  
#include <unordered\_map>  
#include <filesystem>  
#include "sorting\_algorithms.h"  
  
void generateSamples() {  
 std::filesystem::create\_directory("samples");  
 std::vector<int> sample(100000);  
 std::random\_device rd;  
 std::mt19937 gen(rd());  
 std::uniform\_int\_distribution<> dis(1, 1000000);  
  
 // 正序样本  
 for (int i = 0; i < 100000; ++i) {  
 sample[i] = i + 1;  
 }  
 for (int i = 0; i < 10; ++i) {  
 std::ofstream outFile("samples/sample" + std::to\_string(i) + ".txt");  
 if (i == 1) {  
 // 逆序样本  
 std::reverse(sample.begin(), sample.end());  
 } else if (i > 1) {  
 // 随机样本  
 std::generate(sample.begin(), sample.end(), [&]() { return dis(gen); });  
 }  
 for (const auto& num : sample) {  
 outFile << num << " ";  
 }  
 outFile.close();  
 }  
}  
  
void loadSamples(std::vector<std::vector<int>>& samples) {  
 for (int i = 0; i < 10; ++i) {  
 std::ifstream inFile("samples/sample" + std::to\_string(i) + ".txt");  
 std::vector<int> sample(100000);  
 for (int j = 0; j < 100000; ++j) {  
 inFile >> sample[j];  
 }  
 samples.push\_back(sample);  
 inFile.close();  
 }  
}  
  
void measureSortTime(void (\*sortFunc)(std::vector<int>&), std::vector<int> sample, const std::string& sortName) {  
 auto start = std::chrono::high\_resolution\_clock::now();  
 sortFunc(sample);  
 auto end = std::chrono::high\_resolution\_clock::now();  
 std::chrono::duration<double> duration = end - start;  
 std::cout << sortName << "排序耗时: " << duration.count() << "秒" << std::endl;  
}  
  
void countFrequency(const std::vector<int>& sample) {  
 std::unordered\_map<int, int> frequency;  
 for (const auto& num : sample) {  
 frequency[num]++;  
 }  
  
 std::vector<std::pair<int, int>> freqVec(frequency.begin(), frequency.end());  
 std::sort(freqVec.begin(), freqVec.end(), [](const auto& a, const auto& b) {  
 return a.second > b.second;  
 });  
  
 std::cout << "数字出现频率：" << std::endl;  
 for (const auto& [num, freq] : freqVec) {  
 std::cout << "数字: " << num << " 出现次数: " << freq << std::endl;  
 }  
}  
  
int main() {  
 generateSamples();  
 std::vector<std::vector<int>> samples;  
 loadSamples(samples);  
 int sampleCount = 0;  
 for (const auto& sample : samples) {  
 std::cout << "====样本" << ++sampleCount << "======" << std::endl;  
 measureSortTime(insertionSort, sample, "直接插入排序");  
 measureSortTime(shellSort, sample, "希尔排序");  
 measureSortTime(bubbleSort, sample, "冒泡排序");  
 measureSortTime(quickSort, sample, "快速排序");  
 measureSortTime(selectionSort, sample, "选择排序");  
 measureSortTime(heapSort, sample, "堆排序");  
 measureSortTime(mergeSort, sample, "归并排序");  
 measureSortTime(radixSort, sample, "基数排序");  
 }  
  
 countFrequency(samples[0]);  
  
 return 0;  
}

### ../sort/sorting\_algorithms.h

#ifndef SORTING\_ALGORITHMS\_H  
#define SORTING\_ALGORITHMS\_H  
  
#include <vector>  
  
// 直接插入排序  
void insertionSort(std::vector<int>& arr) {  
 for (size\_t i = 1; i < arr.size(); ++i) {  
 int key = arr[i];  
 int j = i - 1;  
 while (j >= 0 && arr[j] > key) {  
 arr[j + 1] = arr[j];  
 --j;  
 }  
 arr[j + 1] = key;  
 }  
}  
  
// 希尔排序  
void shellSort(std::vector<int>& arr) {  
 for (int gap = arr.size() / 2; gap > 0; gap /= 2) {  
 for (size\_t i = gap; i < arr.size(); ++i) {  
 int temp = arr[i];  
 size\_t j;  
 for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {  
 arr[j] = arr[j - gap];  
 }  
 arr[j] = temp;  
 }  
 }  
}  
  
// 冒泡排序  
void bubbleSort(std::vector<int>& arr) {  
 for (size\_t i = 0; i < arr.size() - 1; ++i) {  
 for (size\_t j = 0; j < arr.size() - i - 1; ++j) {  
 if (arr[j] > arr[j + 1]) {  
 std::swap(arr[j], arr[j + 1]);  
 }  
 }  
 }  
}  
  
// 快速排序  
void quickSortHelper(std::vector<int>& arr, int low, int high) {  
 if (low < high) {  
 int pivot = arr[high];  
 int i = low - 1;  
 for (int j = low; j < high; ++j) {  
 if (arr[j] < pivot) {  
 ++i;  
 std::swap(arr[i], arr[j]);  
 }  
 }  
 std::swap(arr[i + 1], arr[high]);  
 int pi = i + 1;  
 quickSortHelper(arr, low, pi - 1);  
 quickSortHelper(arr, pi + 1, high);  
 }  
}  
  
void quickSort(std::vector<int>& arr) {  
 quickSortHelper(arr, 0, arr.size() - 1);  
}  
  
// 选择排序  
void selectionSort(std::vector<int>& arr) {  
 for (size\_t i = 0; i < arr.size() - 1; ++i) {  
 size\_t minIndex = i;  
 for (size\_t j = i + 1; j < arr.size(); ++j) {  
 if (arr[j] < arr[minIndex]) {  
 minIndex = j;  
 }  
 }  
 std::swap(arr[i], arr[minIndex]);  
 }  
}  
  
// 堆排序  
void heapify(std::vector<int>& arr, int n, int i) {  
 int largest = i;  
 int left = 2 \* i + 1;  
 int right = 2 \* i + 2;  
 if (left < n && arr[left] > arr[largest]) {  
 largest = left;  
 }  
 if (right < n && arr[right] > arr[largest]) {  
 largest = right;  
 }  
 if (largest != i) {  
 std::swap(arr[i], arr[largest]);  
 heapify(arr, n, largest);  
 }  
}  
  
void heapSort(std::vector<int>& arr) {  
 for (int i = arr.size() / 2 - 1; i >= 0; --i) {  
 heapify(arr, arr.size(), i);  
 }  
 for (int i = arr.size() - 1; i > 0; --i) {  
 std::swap(arr[0], arr[i]);  
 heapify(arr, i, 0);  
 }  
}  
  
// 归并排序  
void merge(std::vector<int>& arr, int left, int mid, int right) {  
 int n1 = mid - left + 1;  
 int n2 = right - mid;  
 std::vector<int> L(n1), R(n2);  
 for (int i = 0; i < n1; ++i) {  
 L[i] = arr[left + i];  
 }  
 for (int i = 0; i < n2; ++i) {  
 R[i] = arr[mid + 1 + i];  
 }  
 int i = 0, j = 0, k = left;  
 while (i < n1 && j < n2) {  
 if (L[i] <= R[j]) {  
 arr[k] = L[i];  
 ++i;  
 } else {  
 arr[k] = R[j];  
 ++j;  
 }  
 ++k;  
 }  
 while (i < n1) {  
 arr[k] = L[i];  
 ++i;  
 ++k;  
 }  
 while (j < n2) {  
 arr[k] = R[j];  
 ++j;  
 ++k;  
 }  
}  
  
void mergeSortHelper(std::vector<int>& arr, int left, int right) {  
 if (left < right) {  
 int mid = left + (right - left) / 2;  
 mergeSortHelper(arr, left, mid);  
 mergeSortHelper(arr, mid + 1, right);  
 merge(arr, left, mid, right);  
 }  
}  
  
void mergeSort(std::vector<int>& arr) {  
 mergeSortHelper(arr, 0, arr.size() - 1);  
}  
  
// 基数排序  
int getMax(const std::vector<int>& arr) {  
 int max = arr[0];  
 for (size\_t i = 1; i < arr.size(); ++i) {  
 if (arr[i] > max) {  
 max = arr[i];  
 }  
 }  
 return max;  
}  
  
void countSort(std::vector<int>& arr, int exp) {  
 std::vector<int> output(arr.size());  
 int count[10] = {0};  
 for (size\_t i = 0; i < arr.size(); ++i) {  
 count[(arr[i] / exp) % 10]++;  
 }  
 for (int i = 1; i < 10; ++i) {  
 count[i] += count[i - 1];  
 }  
 for (int i = arr.size() - 1; i >= 0; --i) {  
 output[count[(arr[i] / exp) % 10] - 1] = arr[i];  
 count[(arr[i] / exp) % 10]--;  
 }  
 for (size\_t i = 0; i < arr.size(); ++i) {  
 arr[i] = output[i];  
 }  
}  
  
void radixSort(std::vector<int>& arr) {  
 int max = getMax(arr);  
 for (int exp = 1; max / exp > 0; exp \*= 10) {  
 countSort(arr, exp);  
 }  
}  
  
#endif // SORTING\_ALGORITHMS\_H

# 【18】树的应用 (选做)（树）

## 项目简介

这是一个简单的JSON解析器项目，使用C++实现。该解析器能够解析JSON字符串并支持查询嵌套的JSON对象。

## 文件结构

* main.cpp: 主程序文件，包含测试用例和程序入口。
* json\_parser.h: 头文件，包含JSON解析器和哈希表的类定义和声明。
* json\_parser.cpp: 源文件，包含JSON解析器和哈希表的类方法实现。
* README.md: 项目说明文件。

## 使用方法

1. 编译项目：

make

1. 运行项目：

./json\_parser

## 示例

以下是一些示例JSON字符串及其查询结果：

std::string jsonString = R"({"name": "John", "age": "30", "city": "New York"})";  
JsonParser parser(jsonString);  
std::cout << "name: " << parser.query("name") << std::endl; // 输出: John  
std::cout << "age: " << parser.query("age") << std::endl; // 输出: 30  
std::cout << "city: " << parser.query("city") << std::endl; // 输出: New York

## 测试

项目中包含了一些测试用例，可以通过运行编译后的程序来验证解析器的功能。

./json\_parser

如果所有测试通过，程序将输出 “All tests passed”。

## 注意事项

* 该解析器目前支持简单的JSON对象、字符串类型的值、布尔值和列表。

## 源代码

### ../json\_parser/json\_parser.cpp

#include "json\_parser.h"  
  
// HashTable 类方法实现  
HashTable::HashTable(size\_t size) : table(size) {}  
  
void HashTable::insert(const std::string &key, const std::string &value)  
{  
 size\_t index = hashFunction(key) % table.size();  
 for (auto &pair : table[index])  
 {  
 if (pair.first == key)  
 {  
 pair.second = value;  
 return;  
 }  
 }  
 table[index].emplace\_back(key, value);  
}  
  
std::string HashTable::find(const std::string &key) const  
{  
 size\_t index = hashFunction(key) % table.size();  
 for (const auto &pair : table[index])  
 {  
 if (pair.first == key)  
 {  
 return pair.second;  
 }  
 }  
 throw std::runtime\_error("Key not found");  
}  
  
size\_t HashTable::hashFunction(const std::string &key) const  
{  
 size\_t hash = 0;  
 for (char ch : key)  
 {  
 hash = 31 \* hash + ch;  
 }  
 return hash;  
}  
  
// JsonParser 类方法实现  
JsonParser::JsonParser(const std::string &jsonString)  
{  
 parse(jsonString);  
}  
  
std::string JsonParser::query(const std::string &key) const  
{  
 size\_t dotPos = key.find('.');  
 if (dotPos == std::string::npos)  
 {  
 return jsonObject.find(key);  
 }  
 else  
 {  
 std::string firstKey = key.substr(0, dotPos);  
 std::string remainingKey = key.substr(dotPos + 1);  
 std::string nestedJson = jsonObject.find(firstKey);  
 if (nestedJson.front() == '{' && nestedJson.back() == '}')  
 {  
 JsonParser nestedParser(nestedJson);  
 return nestedParser.query(remainingKey);  
 }  
 else  
 {  
 throw std::runtime\_error("Key not found");  
 }  
 }  
}  
  
std::vector<std::string> JsonParser::queryList(const std::string &key) const  
{  
 std::string listJson = jsonObject.find(key);  
 std::istringstream stream(listJson);  
 return parseList(stream);  
}  
  
void JsonParser::parse(const std::string &jsonString)  
{  
 std::istringstream stream(jsonString);  
 char ch;  
 while (stream >> ch)  
 {  
 if (ch == '{')  
 {  
 parseObject(stream);  
 }  
 }  
}  
  
void JsonParser::parseObject(std::istringstream &stream)  
{  
 std::string key, value;  
 char ch;  
 while (stream >> ch)  
 {  
 if (ch == '"')  
 {  
 key = parseString(stream);  
 stream >> ch; // skip ':'  
 stream >> ch; // skip whitespace  
 if (ch == '"')  
 {  
 value = parseString(stream);  
 }  
 else if (ch == '{')  
 {  
 value = parseObjectString(stream);  
 }  
 else if (ch == 't' || ch == 'f')  
 {  
 stream.putback(ch);  
 value = parseBoolean(stream);  
 }  
 else if (ch == '[')  
 {  
 std::vector<std::string> list = parseList(stream);  
 value = "[";  
 for (size\_t i = 0; i < list.size(); ++i)  
 {  
 value += "\"" + list[i] + "\"";  
 if (i < list.size() - 1)  
 {  
 value += ",";  
 }  
 }  
 value += "]";  
 }  
 jsonObject.insert(key, value);  
 }  
 else if (ch == '}')  
 {  
 return;  
 }  
 }  
}  
  
std::string JsonParser::parseString(std::istringstream &stream) const  
{  
 std::string result;  
 char ch;  
 while (stream.get(ch))  
 {  
 if (ch == '\\') // escape sequence  
 {  
 stream.get(ch);  
 if (ch == '"' || ch == '\\')  
 {  
 result += ch;  
 }  
 else if (ch == 'n')  
 {  
 result += '\n';  
 }  
 else if (ch == 't')  
 {  
 result += '\t';  
 }  
 else  
 {  
 throw std::runtime\_error("Invalid escape sequence");  
 }  
 }  
 else if (ch == '"')  
 {  
 break;  
 }  
 else  
 {  
 result += ch;  
 }  
 }  
 return result;  
}  
  
std::string JsonParser::parseObjectString(std::istringstream &stream)  
{  
 std::string result = "{";  
 char ch;  
 while (stream.get(ch))  
 {  
 result += ch;  
 if (ch == '}')  
 {  
 break;  
 }  
 }  
 return result;  
}  
  
std::string JsonParser::parseBoolean(std::istringstream &stream)  
{  
 std::string result;  
 char ch;  
 while (stream.get(ch))  
 {  
 if (ch == ',' || ch == '}' || ch == ' ' || ch == '\n' || ch == '\t')  
 {  
 stream.putback(ch);  
 break;  
 }  
 result += ch;  
 }  
 if (result == "true" || result == "false")  
 {  
 return result;  
 }  
 else  
 {  
 throw std::runtime\_error("Invalid boolean value");  
 }  
}  
  
std::vector<std::string> JsonParser::parseList(std::istringstream &stream) const  
{  
 std::vector<std::string> result;  
 std::string value;  
 char ch;  
 while (stream >> ch)  
 {  
 if (ch == '"')  
 {  
 value = parseString(stream);  
 result.push\_back(value);  
 }  
 else if (ch == ']')  
 {  
 break;  
 }  
 }  
 return result;  
}

### ../json\_parser/json\_parser.h

#ifndef JSON\_PARSER\_H  
#define JSON\_PARSER\_H  
  
#include <iostream>  
#include <string>  
#include <vector>  
#include <list>  
#include <stdexcept>  
#include <sstream>  
  
class HashTable  
{  
public:  
 HashTable(size\_t size = 101);  
 void insert(const std::string &key, const std::string &value);  
 std::string find(const std::string &key) const;  
  
private:  
 std::vector<std::list<std::pair<std::string, std::string>>> table;  
 size\_t hashFunction(const std::string &key) const;  
};  
  
class JsonParser  
{  
public:  
 JsonParser(const std::string &jsonString);  
 std::string query(const std::string &key) const;  
 std::vector<std::string> queryList(const std::string &key) const;  
  
private:  
 HashTable jsonObject;  
 void parse(const std::string &jsonString);  
 void parseObject(std::istringstream &stream);  
 std::string parseString(std::istringstream &stream) const;  
 std::string parseObjectString(std::istringstream &stream);  
 std::string parseBoolean(std::istringstream &stream);  
 std::vector<std::string> parseList(std::istringstream &stream) const;  
};  
  
#endif // JSON\_PARSER\_H

### ../json\_parser/main.cpp

#include "json\_parser.h"  
#include <fstream>  
#include <cassert>  
  
void runTest(const std::string &jsonFile, const std::string &queryFile, const std::string &expectedFile)  
{  
 std::ifstream jsonInput(jsonFile);  
 std::ifstream queryInput(queryFile);  
 std::ifstream expectedInput(expectedFile);  
  
 if (!jsonInput.is\_open() || !queryInput.is\_open() || !expectedInput.is\_open())  
 {  
 std::cerr << "Unable to open test files" << std::endl;  
 return;  
 }  
  
 std::string jsonString;  
 std::string line;  
 while (std::getline(jsonInput, line))  
 {  
 jsonString += line;  
 }  
 jsonInput.close();  
  
 JsonParser parser(jsonString);  
  
 while (std::getline(queryInput, line))  
 {  
 std::string expected;  
 std::getline(expectedInput, expected);  
  
 try  
 {  
 std::string result = parser.query(line);  
 if (result.front() == '{' && result.back() == '}')  
 {  
 assert(expected == "OBJECT");  
 }  
 else  
 {  
 assert(expected == "STRING " + result);  
 }  
 }  
 catch (const std::runtime\_error &)  
 {  
 assert(expected == "NOTEXIST");  
 }  
 }  
  
 queryInput.close();  
 expectedInput.close();  
}  
  
int main()  
{  
 try  
 {  
 runTest("tests/test1.json", "tests/queries1.txt", "tests/expected1.txt");  
 runTest("tests/test2.json", "tests/queries2.txt", "tests/expected2.txt");  
 runTest("tests/test3.json", "tests/queries3.txt", "tests/expected3.txt");  
  
 std::cout << "\033[32mAll tests passed\033[0m" << std::endl;  
 }  
 catch (const std::exception &e)  
 {  
 std::cerr << "Exception: " << e.what() << std::endl;  
 return 1;  
 }  
  
 return 0;  
}

# 总结

在本次数据结构课程设计中，我的主要目标是通过实现多个数据结构项目，解决特定的实际问题。通过本次设计，我不仅加深了对数据结构基本概念和算法的理解，还提升了编程能力和解决问题的能力。

## 项目背景

本项目涵盖了多个数据结构的应用，包括图、树、栈、队列等。我选择了多个实际应用场景，设计并实现了相应的数据结构和算法，以满足特定需求。

## 实现过程

1. **需求分析**：明确项目需求，确定需要实现的数据结构和功能。
2. **设计方案**：根据需求设计数据结构和算法，绘制相关的类图和流程图。
3. **编码实现**：按照设计方案进行编码，逐步实现各个模块和功能。
4. **测试与调试**：编写测试用例，对各个模块进行单元测试和集成测试，发现并修复问题。
5. **优化与完善**：根据测试结果和性能分析，对代码进行优化，提高运行效率和稳定性。

## 项目总结

### 代码行数统计

avl\_tree: 211  
bus: 242  
b\_tree: 366  
family: 436  
huffman: 343  
json\_parser: 316  
maze: 125  
procs: 268  
sort: 258  
subway: 55  
总行数: 2620

通过本次课程设计，我实现了多个数据结构项目，解决了不同的实际问题。下面对每个项目进行简要总结：

### 1. 地铁修建项目

在地铁修建项目中，我使用了Kruskal算法来解决最小生成树问题。通过并查集管理交通枢纽的连通性，成功计算出修建整条地铁线路最少需要的天数。

### 2. 迷宫问题

在迷宫问题项目中，我实现了迷宫的生成和求解。通过使用栈和队列数据结构，分别实现了深度优先搜索和广度优先搜索算法，成功找到了迷宫的解。

### 3. JSON解析器

在JSON解析器项目中，我使用C++实现了一个简单的JSON解析器。该解析器能够解析JSON字符串并支持查询嵌套的JSON对象，提升了我对树结构的理解和应用能力。

### 4. Huffman编码与解码

在Huffman编码与解码项目中，我实现了Huffman编码和解码算法。通过构建Huffman树，成功实现了数据的压缩和解压，掌握了二叉树的应用。

### 5. 家谱管理系统

在家谱管理系统项目中，我实现了一个简单的家谱管理系统。通过树结构管理家族成员的信息，实现了添加、删除、修改成员信息，显示家谱结构，查询成员信息等功能。

### 6. 公交线路提示

在公交线路提示项目中，我使用图数据结构实现了公交线路的查询功能。通过Dijkstra算法，成功计算出最少转车次数和经过站点数量最少的乘车路线。

### 7. 平衡二叉树

在平衡二叉树项目中，我实现了一个AVL树。通过对质数序列的插入、删除和查询操作，掌握了平衡二叉树的基本操作和应用。

## 遇到的问题及解决方案

在项目实施过程中，我遇到了一些问题。例如，在实现某些复杂数据结构时，遇到了性能瓶颈和内存管理问题。通过查阅资料、讨论和反复调试，我最终找到了解决方案，成功克服了这些困难。

## 最终成果

通过本次课程设计，我成功实现了预定的数据结构和功能，项目运行稳定，性能达到了预期目标。我还编写了详细的文档和使用说明，便于后续维护和扩展。

## 收获与体会

本次课程设计不仅让我掌握了数据结构的理论知识，还锻炼了我的实际编程能力和团队合作精神。我学会了如何分析和解决问题，如何进行有效的沟通和协作。这些宝贵的经验将对我未来的学习和工作产生积极的影响。