实验报告电子版

实训五

班级：2

选课序号：04

**数据结构实训**

**—— 多维数字信号的查找**

姓 名： 江一诺

学 号： 2220220821

指导老师： 韩凤

实验日期： 2024年4月24日

**大连海事大学电子信息工程专业**

**Academic Honesty Violations in Practice of Data Structures (PDS)**

In **PDS**, the MINIMUM penalty recommended for a violation of the Academic Honesty Policy will be a ZERO ON THE ASSIGNMENT, PROJECT or EXAM and a LOWERING OF YOUR FINAL GRADE for below what is otherwise earned. You may NOT withdraw from the course if found guilty. Some examples of academic misconduct in **PDS** include but are not limited to the following actions:

1. Picking up and using or discarding another student's written or computer output;

2. Using the computer account of another student;

3. Representing as one's own the work of another on assignments, quizzes, and projects;

4. Giving another student a copy of one's work on an assignment before the due date.

5. Copying work from online resources (Baidu,Chegg, google forums, etc.)

6. Posting work to online resources where other students can view your work.

All submissions will be checked for similarity. This code will check each submission for similarity between other student submissions, past student submissions, the solution manual, and online resources and postings. If your submission is flagged for a high level of similarity, it will be turned in for an academic honesty violation if deemed appropriate.

NOTE: Changing variable names, adding comments, or spacing will l result in a violation.

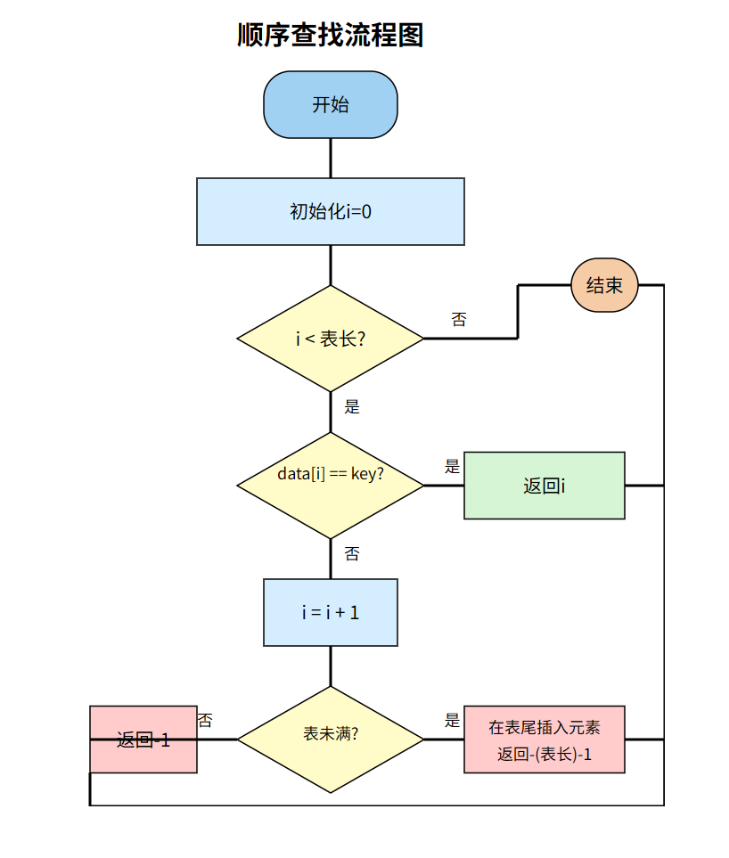
1. 实验目的
2. 深入理解各种查找的算法思想、方法、稳定性及时间和空间复杂度；
3. 掌握静态表、动态表和哈希查找的算法实现；
4. 能够实现多维信号的查找操作；
5. 提高实际动手进行程序设计的能力。
6. 实验内容与要求

实验内容：

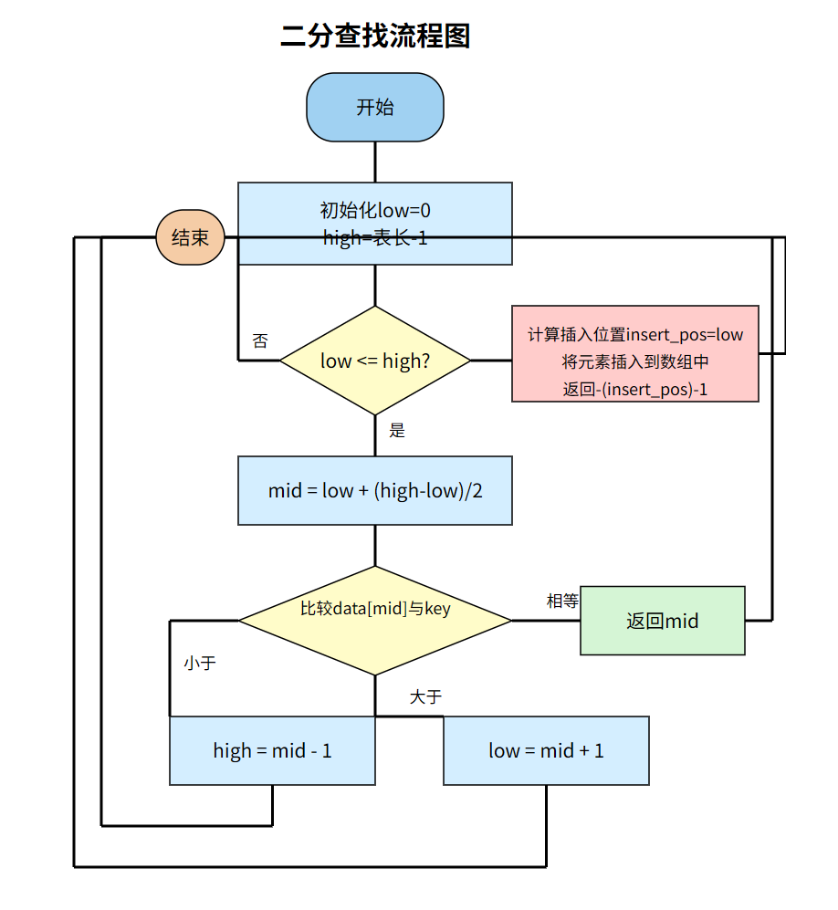
1. 从静态表、动态表和哈希表三类查找算法中各选择一种算法实现；
2. 查找的依据可以是其中的任意一维或者所有维数；
3. 距离的计算方式可以为欧几里得距离、街区距离或者余弦距离；
4. 比较分析三种算法的优缺点和适用范围。

实验要求：

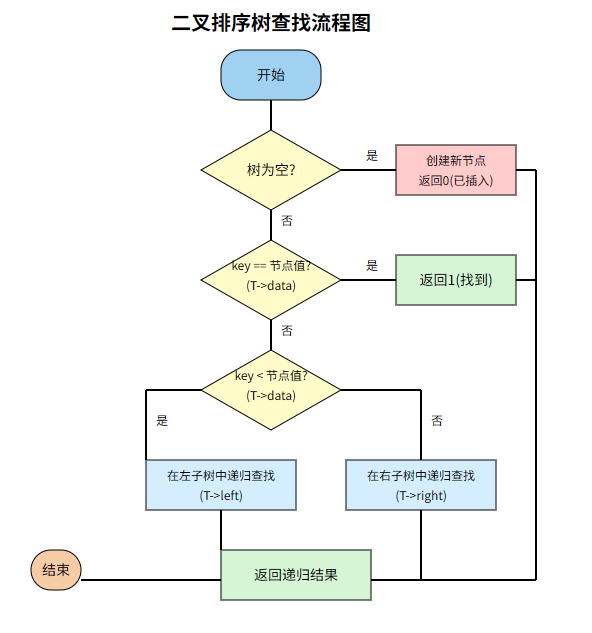
1. 遵守工程伦理和学术规范。
2. 解决的问题要来源于实际工程：要对日常生活中的一个具体应用进行抽象建模与求解，鼓励对国家重大工程中的一个具体点进行抽象建模与求解并融入工程管理理念。
3. 深入理解各种查找算法思想、方法及时间和空间复杂度，要对各种方案进行分析比较。
4. 要对实验结果进行和解释，并通过信息综合得到合理有效地结论。
5. 算法描述与流程
6. 顺序查找法流程图



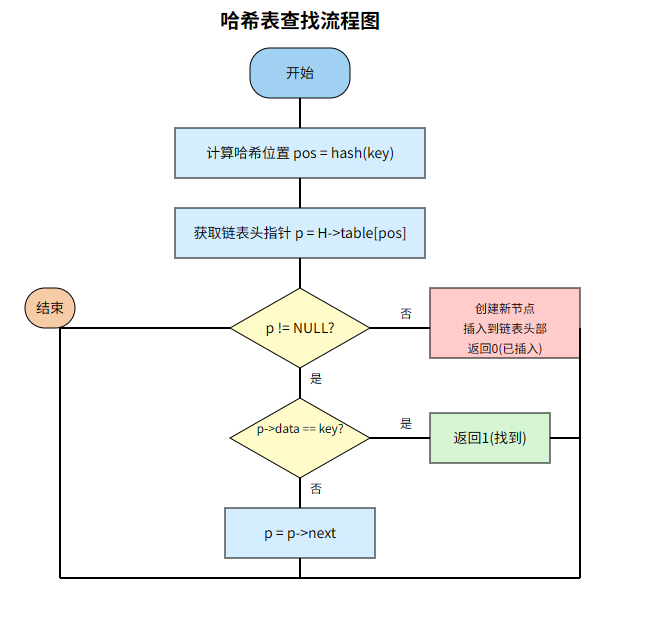
1. 二分查找流程图



1. 二叉排序树查找流程图



1. 哈希查找流程图



本次实验使用了折半查找法、二叉排序树法、哈希表查找法对随机生成的数据集进行排序。随机生成的数据集的关键字值使用余弦距离来计算。

1. 算法实现

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define MAX\_SIZE 10000

#define HASH\_SIZE 10007  *// 选择一个较大的质数作为哈希表大小*

typedef int ElementType;

*// 顺序表结构*

typedef struct {

    ElementType data[MAX\_SIZE];

    int length;

} SeqList;

*// 二叉排序树节点结构*

typedef struct BSTNode {

    ElementType data;

    struct BSTNode \*left;

    struct BSTNode \*right;

} BSTNode, \*BSTree;

*// 哈希表节点结构*

typedef struct HashNode {

    ElementType data;

    struct HashNode \*next;  *// 拉链法解决冲突*

} HashNode;

*// 哈希表结构*

typedef struct {

    HashNode \*table[HASH\_SIZE];

    int count;  *// 记录哈希表中的元素数量*

} HashTable;

*// 初始化顺序表*

void InitSeqList(SeqList \*L) {

    L->length = 0;

}

*// 从文件加载数据到顺序表*

int LoadSeqListFromFile(SeqList \*L, const char \*filename) {

    FILE \*fp = fopen(filename, "r");

    if (fp == NULL) {

        printf("无法打开文件 %s\n", filename);

        return 0;

    }

    int value;

    while (fscanf(fp, "%d", &value) == 1 && L->length < MAX\_SIZE) {

        L->data[L->length++] = value;

    }

    fclose(fp);

    return 1;

}

*//静态表：顺序查找、二分查找*

*// 顺序查找并在未找到时插入*

*// 在顺序表中逐个比较元素，查找关键字key，若未找到则将key插入到表尾*

*// 参数：L - 顺序表结构指针，key - 要查找的关键字*

*// 返回值：若找到则返回元素下标，未找到但插入成功返回-插入位置-1，插入失败返回-1*

*// 时间复杂度：O(n)，n为表长*

int SequentialSearchAndInsert(SeqList \*L, ElementType key) {

    int i;

*// 先进行顺序查找*

    for (i = 0; i < L->length; i++) {

        if (L->data[i] == key) {

            return i;  *// 找到匹配元素，返回下标 (0或正数)*

        }

    }

*// 未找到元素，尝试插入到表尾*

    if (L->length < MAX\_SIZE) {

        L->data[L->length] = key;

        int pos = L->length;

        L->length++;

        return -pos-1;  *// 返回插入位置的负值减1，以区分查找成功的情况*

    }

    return -1;  *// 表已满，插入失败*

}

*// 对顺序表进行排序（用于二分查找前的准备）*

*//使用快速排序提高效率*

void Swap(ElementType \*a, ElementType \*b) {

    ElementType temp = \*a;

    \*a = \*b;

    \*b = temp;

}

int Partition(ElementType arr[], int low, int high) {

    ElementType pivot = arr[high];

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {

        if (arr[j] < pivot) {

            i++;

            Swap(&arr[i], &arr[j]);

        }

    }

    Swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

void QuickSort(ElementType arr[], int low, int high) {

    if (low < high) {

        int pi = Partition(arr, low, high);

        QuickSort(arr, low, pi - 1);

        QuickSort(arr, pi + 1, high);

    }

}

void SortSeqList(SeqList \*L) {

    QuickSort(L->data, 0, L->length - 1);

}

*// 将排序后的表内容保存到文件*

int SaveSortedListToFile(SeqList L, const char \*filename) {

    FILE \*fp = fopen(filename, "w");

    if (fp == NULL) {

        printf("无法创建文件 %s\n", filename);

        return 0;

    }

    for (int i = 0; i < L.length; i++) {

        fprintf(fp, "%d\n", L.data[i]);

    }

    fclose(fp);

    return 1;

}

*// 二分查找并在未找到时插入*

*// 在已按升序排列的顺序表中使用二分策略查找关键字key，若未找到则将key插入适当位置*

*// 参数：L - 已排序的顺序表结构指针，key - 要查找的关键字*

*// 返回值：若找到则返回元素下标，未找到但插入成功返回-插入位置-1，插入失败返回-1*

*// 时间复杂度：O(log n) + O(n)，查找O(log n)，插入最坏需要O(n)移动元素*

int BinarySearchAndInsert(SeqList \*L, ElementType key) {

    int low = 0;

    int high = L->length - 1;

    int mid;

    int insert\_pos = 0;

*// 二分查找过程*

    while (low <= high) {

        mid = low + (high - low) / 2;

        if (L->data[mid] == key) {

            return mid;  *// 找到匹配元素，返回正数下标*

        } else if (L->data[mid] > key) {

            high = mid - 1;

        } else {

            low = mid + 1;

        }

    }

*// 未找到元素，计算应该插入的位置*

    insert\_pos = low;  *// low为应插入的位置*

*// 检查表是否已满*

    if (L->length >= MAX\_SIZE) {

        return -1;  *// 表已满，插入失败*

    }

*// 从后向前移动元素，为新元素腾出位置*

    for (int i = L->length; i > insert\_pos; i--) {

        L->data[i] = L->data[i-1];

    }

*// 插入新元素*

    L->data[insert\_pos] = key;

    L->length++;

    return -insert\_pos-1;  *// 返回插入位置的负值减1，以区分查找成功*

}

*// 创建二叉排序树节点*

BSTNode\* CreateBSTNode(ElementType data) {

    BSTNode \*node = (BSTNode\*)malloc(sizeof(BSTNode));

    if (node == NULL) {

        printf("内存分配失败\n");

        exit(1);

    }

    node->data = data;

    node->left = NULL;

    node->right = NULL;

    return node;

}

*// 向二叉排序树中插入节点*

BSTree InsertBST(BSTree T, ElementType key) {

    if (T == NULL) {

*// 创建新节点*

        return CreateBSTNode(key);

    }

    if (key < T->data) {

*// 插入左子树*

        T->left = InsertBST(T->left, key);

    } else if (key > T->data) {

*// 插入右子树*

        T->right = InsertBST(T->right, key);

    }

    return T;

}

*// 从文件构建二叉排序树*

BSTree BuildBSTFromFile(const char \*filename) {

    FILE \*fp = fopen(filename, "r");

    if (fp == NULL) {

        printf("无法打开文件 %s\n", filename);

        return NULL;

    }

    BSTree root = NULL;

    int value;

    while (fscanf(fp, "%d", &value) == 1) {

        root = InsertBST(root, value);

    }

    fclose(fp);

    return root;

}

*//动态表：二叉排序树查找*

*// 在二叉排序树中查找并在未找到时插入*

*// 参数：T - 二叉排序树根节点指针的指针，key - 要查找的关键字*

*// 返回值：1表示找到元素，0表示未找到但已插入新元素*

int SearchBSTAndInsert(BSTree \*T, ElementType key) {

*// 树为空或找到目标节点*

    if (\*T == NULL) {

*// 未找到，创建新节点并插入*

        \*T = CreateBSTNode(key);

        return 0;  *// 表示插入了新节点*

    }

    if (key == (\*T)->data) {

        return 1;  *// 找到匹配元素*

    } else if (key < (\*T)->data) {

*// 在左子树中继续查找*

        return SearchBSTAndInsert(&((\*T)->left), key);

    } else {

*// 在右子树中继续查找*

        return SearchBSTAndInsert(&((\*T)->right), key);

    }

}

*// 释放二叉排序树*

void FreeBST(BSTree T) {

    if (T != NULL) {

        FreeBST(T->left);

        FreeBST(T->right);

        free(T);

    }

}

*// 统计二叉排序树高度*

int BSTHeight(BSTree T) {

    if (T == NULL) {

        return 0;

    }

    int leftHeight = BSTHeight(T->left);

    int rightHeight = BSTHeight(T->right);

*//取较大值*

    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

}

*// 统计二叉排序树节点数*

int BSTNodeCount(BSTree T) {

    if (T == NULL) {

        return 0;

    }

    return BSTNodeCount(T->left) + BSTNodeCount(T->right) + 1;

}

*// 打印顺序表前几个元素和总数*

void PrintSeqListSummary(SeqList L) {

    int i, display\_count = 20;  *// 只显示前20个元素*

    printf("顺序表数据总数: %d\n", L.length);

    printf("前%d个数据: ", display\_count < L.length ? display\_count : L.length);

    for (i = 0; i < L.length && i < display\_count; i++) {

        printf("%d ", L.data[i]);

    }

    printf("...\n");

}

*// 哈希函数*

int HashFunction(ElementType key) {

    return key % HASH\_SIZE;

}

*// 初始化哈希表*

void InitHashTable(HashTable \*H) {

    int i;

    for (i = 0; i < HASH\_SIZE; i++) {

        H->table[i] = NULL;

    }

    H->count = 0;

}

*// 在哈希表中插入元素*

void InsertHash(HashTable \*H, ElementType key) {

    int pos = HashFunction(key);

*// 创建新节点*

    HashNode \*newNode = (HashNode\*)malloc(sizeof(HashNode));

    if (newNode == NULL) {

        printf("内存分配失败\n");

        return;

    }

    newNode->data = key;

*// 头插法插入链表*

    newNode->next = H->table[pos];

    H->table[pos] = newNode;

    H->count++;

}

*// 从文件构建哈希表*

HashTable\* BuildHashFromFile(const char \*filename) {

    FILE \*fp = fopen(filename, "r");

    if (fp == NULL) {

        printf("无法打开文件 %s\n", filename);

        return NULL;

    }

    HashTable \*H = (HashTable\*)malloc(sizeof(HashTable));

    if (H == NULL) {

        printf("内存分配失败\n");

        fclose(fp);

        return NULL;

    }

    InitHashTable(H);

    int value;

    while (fscanf(fp, "%d", &value) == 1) {

        InsertHash(H, value);

    }

    fclose(fp);

    return H;

}

*// 在哈希表中查找并在未找到时插入*

*// 参数：H - 哈希表，key - 要查找的关键字*

*// 返回值：1表示找到元素，0表示未找到但已插入新元素*

int SearchHashAndInsert(HashTable \*H, ElementType key) {

    int pos = HashFunction(key);

    HashNode \*p = H->table[pos];

*// 遍历链表查找*

    while (p != NULL) {

        if (p->data == key) {

            return 1;  *// 找到元素*

        }

        p = p->next;

    }

*// 未找到元素，执行插入操作*

    InsertHash(H, key);

    return 0;  *// 表示插入了新元素*

}

*// 释放哈希表内存*

void FreeHashTable(HashTable \*H) {

    int i;

    for (i = 0; i < HASH\_SIZE; i++) {

        HashNode \*current = H->table[i];

        while (current != NULL) {

            HashNode \*temp = current;

            current = current->next;

            free(temp);

        }

        H->table[i] = NULL;

    }

    free(H);

}

*// 计算哈希表中的冲突次数和平均查找长度*

void HashTableStats(HashTable \*H, int \*conflicts, float \*asl) {

    int i;

    int total\_conflicts = 0;

    int total\_length = 0;

    int chains = 0;  *// 实际有数据的链表数*

    for (i = 0; i < HASH\_SIZE; i++) {

        int chain\_length = 0;

        HashNode \*p = H->table[i];

*// 计算链长*

        while (p != NULL) {

            chain\_length++;

            p = p->next;

        }

        if (chain\_length > 0) {

            chains++;

            if (chain\_length > 1) {

                total\_conflicts += (chain\_length - 1);

            }

            total\_length += chain\_length;

        }

    }

    \*conflicts = total\_conflicts;

*// 平均查找长度计算*

    if (chains > 0) {

        \*asl = (float)total\_length / chains;

    } else {

        \*asl = 0.0;

    }

}

*// 为保持兼容性，保留原始的搜索函数*

*// 在二叉排序树中查找*

*// 递归地在二叉排序树中查找关键字key*

*// 参数：T - 二叉排序树根节点指针，key - 要查找的关键字*

*// 返回值：若找到则返回目标节点指针，未找到返回NULL*

*// 时间复杂度：平均O(log n)，最坏O(n)，n为树中节点数*

BSTNode\* SearchBST(BSTree T, ElementType key) {

*// 递归终止条件：树为空（未找到）或找到关键字*

    if (T == NULL || T->data == key) {

        return T;

    }

*// 二叉排序树的特性：小值在左，大值在右*

    if (key < T->data) {

        return SearchBST(T->left, key);  *// 目标值小于当前节点值，在左子树中查找*

    } else {

        return SearchBST(T->right, key); *// 目标值大于当前节点值，在右子树中查找*

    }

}

int main() {

    SeqList L;

    InitSeqList(&L);

    printf("正在从data.txt加载数据...\n");

*// 加载数据*

    if (!LoadSeqListFromFile(&L, "data.txt")) {

        printf("加载数据失败，创建测试数据...\n");

*// 如果文件不存在，创建一些测试数据*

        int i;

        for (i = 0; i < 100; i++) {

            L.data[L.length++] = rand() % 10000;

        }

*// 保存测试数据到文件*

        FILE \*fp = fopen("data.txt", "w");

        if (fp != NULL) {

            for (i = 0; i < L.length; i++) {

                fprintf(fp, "%d\n", L.data[i]);

            }

            fclose(fp);

            printf("已创建测试数据并保存到data.txt\n");

        }

    }

*// 打印顺序表内容摘要*

    PrintSeqListSummary(L);

*// 1. 顺序查找*

    int key = 0;

    printf("\n请输入要查找的关键字: ");

    scanf("%d", &key);

    clock\_t start, end;

    double cpu\_time\_used;

    double seq\_time, bin\_time, bst\_time, sort\_time, hash\_time;

    int seq\_pos, bin\_pos;

    BSTNode\* bst\_result;

    int hash\_result;

*// 顺序查找测试*

    start = clock();

    seq\_pos = SequentialSearchAndInsert(&L, key);

    end = clock();

    seq\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    if (seq\_pos >= 0) {

        printf("\n1. 顺序查找：找到关键字 %d，位置为 %d\n", key, seq\_pos);

    } else if (seq\_pos != -1) {

*// 插入成功，计算真实位置*

        int insert\_pos = -seq\_pos - 1;

        printf("\n1. 顺序查找：未找到关键字 %d，已插入到位置 %d\n", key, insert\_pos);

    } else {

        printf("\n1. 顺序查找：未找到关键字 %d，表已满无法插入\n", key);

    }

    printf("   顺序查找耗时: %.3f 毫秒\n", seq\_time);

*// 2. 二分查找测试*

*// 先排序*

    printf("\n对数据进行排序中...\n");

    start = clock();

    SortSeqList(&L);

    end = clock();

    sort\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    printf("排序耗时: %.3f 毫秒\n", sort\_time);

*// 将排序后的内容保存到文件*

    if (SaveSortedListToFile(L, "sorted\_data.txt")) {

        printf("排序后的数据已保存到 sorted\_data.txt\n");

    } else {

        printf("保存排序后的数据失败\n");

    }

    start = clock();

    bin\_pos = BinarySearchAndInsert(&L, key);

    end = clock();

    bin\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    if (bin\_pos >= 0) {

        printf("\n2. 二分查找：找到关键字 %d，位置为 %d\n", key, bin\_pos);

    } else if (bin\_pos != -1) {

*// 插入成功，计算真实位置*

        int insert\_pos = -bin\_pos - 1;

        printf("\n2. 二分查找：未找到关键字 %d，已插入到位置 %d\n", key, insert\_pos);

    } else {

        printf("\n2. 二分查找：未找到关键字 %d，表已满无法插入\n", key);

    }

    printf("   二分查找耗时: %.3f 毫秒\n", bin\_time);

*// 3. 二叉排序树查找*

*// 构建二叉排序树*

    printf("\n正在构建二叉排序树...\n");

    start = clock();

    BSTree bst = BuildBSTFromFile("data.txt");

    end = clock();

    double bst\_build\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    printf("构建二叉排序树耗时: %.3f 毫秒\n", bst\_build\_time);

    int bst\_height = BSTHeight(bst);

    int bst\_node\_count = BSTNodeCount(bst);

    printf("二叉排序树高度: %d, 节点数: %d\n", bst\_height, bst\_node\_count);

    start = clock();

    int bst\_insert\_result = SearchBSTAndInsert(&bst, key);

    end = clock();

    bst\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    if (bst\_insert\_result == 1) {

        printf("\n3. 二叉排序树查找：找到关键字 %d\n", key);

    } else {

        printf("\n3. 二叉排序树查找：未找到关键字 %d，已成功插入\n", key);

*// 更新树的高度和节点数*

        bst\_height = BSTHeight(bst);

        bst\_node\_count = BSTNodeCount(bst);

        printf("   插入后二叉排序树高度: %d, 节点数: %d\n", bst\_height, bst\_node\_count);

    }

    printf("   二叉排序树查找耗时: %.3f 毫秒\n", bst\_time);

*// 释放二叉排序树*

    FreeBST(bst);

*// 4. 哈希表查找*

    printf("\n正在构建哈希表...\n");

    start = clock();

    HashTable \*hashTable = BuildHashFromFile("data.txt");

    end = clock();

    double hash\_build\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    printf("构建哈希表耗时: %.3f 毫秒\n", hash\_build\_time);

*// 计算冲突次数和平均查找长度*

    int conflicts;

    float asl;

    HashTableStats(hashTable, &conflicts, &asl);

    printf("哈希表冲突次数: %d, 平均查找长度: %.2f\n", conflicts, asl);

    start = clock();

    int hash\_insert\_result = SearchHashAndInsert(hashTable, key);

    end = clock();

    hash\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    if (hash\_insert\_result == 1) {

        printf("\n4. 哈希表查找：找到关键字 %d\n", key);

    } else {

        printf("\n4. 哈希表查找：未找到关键字 %d，已成功插入\n", key);

*// 重新计算哈希表统计信息*

        HashTableStats(hashTable, &conflicts, &asl);

        printf("   插入后哈希表冲突次数: %d, 平均查找长度: %.2f\n", conflicts, asl);

    }

    printf("   哈希表查找耗时: %.3f 毫秒\n", hash\_time);

*// 释放哈希表*

    FreeHashTable(hashTable);

    printf("\n性能分析：\n");

    printf("1. 顺序查找 - 时间复杂度O(n)，适合于数据量小的情况\n");

    printf("2. 二分查找 - 时间复杂度O(log n)，但需要先对数据排序\n");

    printf("3. 二叉排序树 - 平均查找时间复杂度O(log n)，最坏情况O(n)\n");

    printf("4. 哈希表查找 - 平均时间复杂度O(1)，最坏情况O(n)\n");

*// 将查找时间和性能分析保存到文件*

    FILE \*perf\_file = fopen("性能分析.txt", "w");

    if (perf\_file != NULL) {

        fprintf(perf\_file, "查找算法性能测试报告\n");

        fprintf(perf\_file, "==========================\n\n");

        fprintf(perf\_file, "数据量: %d 个元素\n", L.length);

        fprintf(perf\_file, "查找关键字: %d\n\n", key);

        fprintf(perf\_file, "| 算法名称         | 运行时间 (毫秒) |\n");

        fprintf(perf\_file, "|-----------------|---------------|\n");

        fprintf(perf\_file, "| 顺序查找         | %13.3f |\n", seq\_time);

        fprintf(perf\_file, "| 二分查找         | %13.3f |\n", bin\_time);

        fprintf(perf\_file, "| 二叉排序树查找    | %13.3f |\n", bst\_time);

        fprintf(perf\_file, "| 哈希表查找       | %13.3f |\n", hash\_time);

        fprintf(perf\_file, "\n");

        fprintf(perf\_file, "预处理时间:\n");

        fprintf(perf\_file, "| 预处理步骤       | 运行时间 (毫秒) |\n");

        fprintf(perf\_file, "|-----------------|---------------|\n");

        fprintf(perf\_file, "| 排序操作         | %13.3f |\n", sort\_time);

        fprintf(perf\_file, "| 构建二叉排序树    | %13.3f |\n", bst\_build\_time);

        fprintf(perf\_file, "| 构建哈希表       | %13.3f |\n\n", hash\_build\_time);

        fprintf(perf\_file, "查找结果:\n");

        if (seq\_pos >= 0) {

            fprintf(perf\_file, "- 顺序查找：找到关键字 %d，位置为 %d\n", key, seq\_pos);

        } else if (seq\_pos != -1) {

            int insert\_pos = -seq\_pos - 1;

            fprintf(perf\_file, "- 顺序查找：未找到关键字 %d，已插入到位置 %d\n", key, insert\_pos);

        } else {

            fprintf(perf\_file, "- 顺序查找：未找到关键字 %d，表已满无法插入\n", key);

        }

        if (bin\_pos >= 0) {

            fprintf(perf\_file, "- 二分查找：找到关键字 %d，位置为 %d\n", key, bin\_pos);

        } else if (bin\_pos != -1) {

            int insert\_pos = -bin\_pos - 1;

            fprintf(perf\_file, "- 二分查找：未找到关键字 %d，已插入到位置 %d\n", key, insert\_pos);

        } else {

            fprintf(perf\_file, "- 二分查找：未找到关键字 %d，表已满无法插入\n", key);

        }

        if (bst\_insert\_result == 1) {

            fprintf(perf\_file, "- 二叉排序树查找：找到关键字 %d\n", key);

        } else {

            fprintf(perf\_file, "- 二叉排序树查找：未找到关键字 %d，已成功插入\n", key);

        }

        if (hash\_insert\_result == 1) {

            fprintf(perf\_file, "- 哈希表查找：找到关键字 %d\n", key);

        } else {

            fprintf(perf\_file, "- 哈希表查找：未找到关键字 %d，已成功插入\n", key);

        }

        fprintf(perf\_file, "\n二叉排序树信息: 高度: %d, 节点数: %d\n", bst\_height, bst\_node\_count);

        fprintf(perf\_file, "哈希表信息: 冲突次数: %d, 平均查找长度: %.2f\n\n", conflicts, asl);

        fprintf(perf\_file, "算法复杂度分析:\n");

        fprintf(perf\_file, "- 顺序查找: O(n)\n");

        fprintf(perf\_file, "- 二分查找: O(log n), 但需要先排序 O(n log n)\n");

        fprintf(perf\_file, "- 二叉排序树查找: 平均O(log n), 最坏O(n)\n");

        fprintf(perf\_file, "- 哈希表查找: 平均O(1), 最坏O(n)\n\n");

        fprintf(perf\_file, "注：时间单位为毫秒。测试环境：%s。\n", "MacOS");

        fprintf(perf\_file, "数据来源：data.txt（%d个数据点）\n", L.length);

        fclose(perf\_file);

        printf("\n性能分析结果已保存到 性能分析.txt 文件\n");

    } else {

        printf("\n无法创建性能分析结果文件\n");

    }

    return 0;

} #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define MAX\_SIZE 10000

#define HASH\_SIZE 10007  *// 选择一个较大的质数作为哈希表大小*

typedef int ElementType;

*// 顺序表结构*

typedef struct {

    ElementType data[MAX\_SIZE];

    int length;

} SeqList;

*// 二叉排序树节点结构*

typedef struct BSTNode {

    ElementType data;

    struct BSTNode \*left;

    struct BSTNode \*right;

} BSTNode, \*BSTree;

*// 哈希表节点结构*

typedef struct HashNode {

    ElementType data;

    struct HashNode \*next;  *// 拉链法解决冲突*

} HashNode;

*// 哈希表结构*

typedef struct {

    HashNode \*table[HASH\_SIZE];

    int count;  *// 记录哈希表中的元素数量*

} HashTable;

*// 初始化顺序表*

void InitSeqList(SeqList \*L) {

    L->length = 0;

}

*// 从文件加载数据到顺序表*

int LoadSeqListFromFile(SeqList \*L, const char \*filename) {

    FILE \*fp = fopen(filename, "r");

    if (fp == NULL) {

        printf("无法打开文件 %s\n", filename);

        return 0;

    }

    int value;

    while (fscanf(fp, "%d", &value) == 1 && L->length < MAX\_SIZE) {

        L->data[L->length++] = value;

    }

    fclose(fp);

    return 1;

}

*//静态表：顺序查找、二分查找*

*// 顺序查找并在未找到时插入*

*// 在顺序表中逐个比较元素，查找关键字key，若未找到则将key插入到表尾*

*// 参数：L - 顺序表结构指针，key - 要查找的关键字*

*// 返回值：若找到则返回元素下标，未找到但插入成功返回-插入位置-1，插入失败返回-1*

*// 时间复杂度：O(n)，n为表长*

int SequentialSearchAndInsert(SeqList \*L, ElementType key) {

    int i;

*// 先进行顺序查找*

    for (i = 0; i < L->length; i++) {

        if (L->data[i] == key) {

            return i;  *// 找到匹配元素，返回下标 (0或正数)*

        }

    }

*// 未找到元素，尝试插入到表尾*

    if (L->length < MAX\_SIZE) {

        L->data[L->length] = key;

        int pos = L->length;

        L->length++;

        return -pos-1;  *// 返回插入位置的负值减1，以区分查找成功的情况*

    }

    return -1;  *// 表已满，插入失败*

}

*// 对顺序表进行排序（用于二分查找前的准备）*

*//使用快速排序提高效率*

void Swap(ElementType \*a, ElementType \*b) {

    ElementType temp = \*a;

    \*a = \*b;

    \*b = temp;

}

int Partition(ElementType arr[], int low, int high) {

    ElementType pivot = arr[high];

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {

        if (arr[j] < pivot) {

            i++;

            Swap(&arr[i], &arr[j]);

        }

    }

    Swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

void QuickSort(ElementType arr[], int low, int high) {

    if (low < high) {

        int pi = Partition(arr, low, high);

        QuickSort(arr, low, pi - 1);

        QuickSort(arr, pi + 1, high);

    }

}

void SortSeqList(SeqList \*L) {

    QuickSort(L->data, 0, L->length - 1);

}

*// 将排序后的表内容保存到文件*

int SaveSortedListToFile(SeqList L, const char \*filename) {

    FILE \*fp = fopen(filename, "w");

    if (fp == NULL) {

        printf("无法创建文件 %s\n", filename);

        return 0;

    }

    for (int i = 0; i < L.length; i++) {

        fprintf(fp, "%d\n", L.data[i]);

    }

    fclose(fp);

    return 1;

}

*// 二分查找并在未找到时插入*

*// 在已按升序排列的顺序表中使用二分策略查找关键字key，若未找到则将key插入适当位置*

*// 参数：L - 已排序的顺序表结构指针，key - 要查找的关键字*

*// 返回值：若找到则返回元素下标，未找到但插入成功返回-插入位置-1，插入失败返回-1*

*// 时间复杂度：O(log n) + O(n)，查找O(log n)，插入最坏需要O(n)移动元素*

int BinarySearchAndInsert(SeqList \*L, ElementType key) {

    int low = 0;

    int high = L->length - 1;

    int mid;

    int insert\_pos = 0;

*// 二分查找过程*

    while (low <= high) {

        mid = low + (high - low) / 2;

        if (L->data[mid] == key) {

            return mid;  *// 找到匹配元素，返回正数下标*

        } else if (L->data[mid] > key) {

            high = mid - 1;

        } else {

            low = mid + 1;

        }

    }

*// 未找到元素，计算应该插入的位置*

    insert\_pos = low;  *// low为应插入的位置*

*// 检查表是否已满*

    if (L->length >= MAX\_SIZE) {

        return -1;  *// 表已满，插入失败*

    }

*// 从后向前移动元素，为新元素腾出位置*

    for (int i = L->length; i > insert\_pos; i--) {

        L->data[i] = L->data[i-1];

    }

*// 插入新元素*

    L->data[insert\_pos] = key;

    L->length++;

    return -insert\_pos-1;  *// 返回插入位置的负值减1，以区分查找成功*

}

*// 创建二叉排序树节点*

BSTNode\* CreateBSTNode(ElementType data) {

    BSTNode \*node = (BSTNode\*)malloc(sizeof(BSTNode));

    if (node == NULL) {

        printf("内存分配失败\n");

        exit(1);

    }

    node->data = data;

    node->left = NULL;

    node->right = NULL;

    return node;

}

*// 向二叉排序树中插入节点*

BSTree InsertBST(BSTree T, ElementType key) {

    if (T == NULL) {

*// 创建新节点*

        return CreateBSTNode(key);

    }

    if (key < T->data) {

*// 插入左子树*

        T->left = InsertBST(T->left, key);

    } else if (key > T->data) {

*// 插入右子树*

        T->right = InsertBST(T->right, key);

    }

    return T;

}

*// 从文件构建二叉排序树*

BSTree BuildBSTFromFile(const char \*filename) {

    FILE \*fp = fopen(filename, "r");

    if (fp == NULL) {

        printf("无法打开文件 %s\n", filename);

        return NULL;

    }

    BSTree root = NULL;

    int value;

    while (fscanf(fp, "%d", &value) == 1) {

        root = InsertBST(root, value);

    }

    fclose(fp);

    return root;

}

*//动态表：二叉排序树查找*

*// 在二叉排序树中查找并在未找到时插入*

*// 参数：T - 二叉排序树根节点指针的指针，key - 要查找的关键字*

*// 返回值：1表示找到元素，0表示未找到但已插入新元素*

int SearchBSTAndInsert(BSTree \*T, ElementType key) {

*// 树为空或找到目标节点*

    if (\*T == NULL) {

*// 未找到，创建新节点并插入*

        \*T = CreateBSTNode(key);

        return 0;  *// 表示插入了新节点*

    }

    if (key == (\*T)->data) {

        return 1;  *// 找到匹配元素*

    } else if (key < (\*T)->data) {

*// 在左子树中继续查找*

        return SearchBSTAndInsert(&((\*T)->left), key);

    } else {

*// 在右子树中继续查找*

        return SearchBSTAndInsert(&((\*T)->right), key);

    }

}

*// 释放二叉排序树*

void FreeBST(BSTree T) {

    if (T != NULL) {

        FreeBST(T->left);

        FreeBST(T->right);

        free(T);

    }

}

*// 统计二叉排序树高度*

int BSTHeight(BSTree T) {

    if (T == NULL) {

        return 0;

    }

    int leftHeight = BSTHeight(T->left);

    int rightHeight = BSTHeight(T->right);

*//取较大值*

    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

}

*// 统计二叉排序树节点数*

int BSTNodeCount(BSTree T) {

    if (T == NULL) {

        return 0;

    }

    return BSTNodeCount(T->left) + BSTNodeCount(T->right) + 1;

}

*// 打印顺序表前几个元素和总数*

void PrintSeqListSummary(SeqList L) {

    int i, display\_count = 20;  *// 只显示前20个元素*

    printf("顺序表数据总数: %d\n", L.length);

    printf("前%d个数据: ", display\_count < L.length ? display\_count : L.length);

    for (i = 0; i < L.length && i < display\_count; i++) {

        printf("%d ", L.data[i]);

    }

    printf("...\n");

}

*// 哈希函数*

int HashFunction(ElementType key) {

    return key % HASH\_SIZE;

}

*// 初始化哈希表*

void InitHashTable(HashTable \*H) {

    int i;

    for (i = 0; i < HASH\_SIZE; i++) {

        H->table[i] = NULL;

    }

    H->count = 0;

}

*// 在哈希表中插入元素*

void InsertHash(HashTable \*H, ElementType key) {

    int pos = HashFunction(key);

*// 创建新节点*

    HashNode \*newNode = (HashNode\*)malloc(sizeof(HashNode));

    if (newNode == NULL) {

        printf("内存分配失败\n");

        return;

    }

    newNode->data = key;

*// 头插法插入链表*

    newNode->next = H->table[pos];

    H->table[pos] = newNode;

    H->count++;

}

*// 从文件构建哈希表*

HashTable\* BuildHashFromFile(const char \*filename) {

    FILE \*fp = fopen(filename, "r");

    if (fp == NULL) {

        printf("无法打开文件 %s\n", filename);

        return NULL;

    }

    HashTable \*H = (HashTable\*)malloc(sizeof(HashTable));

    if (H == NULL) {

        printf("内存分配失败\n");

        fclose(fp);

        return NULL;

    }

    InitHashTable(H);

    int value;

    while (fscanf(fp, "%d", &value) == 1) {

        InsertHash(H, value);

    }

    fclose(fp);

    return H;

}

*// 在哈希表中查找并在未找到时插入*

*// 参数：H - 哈希表，key - 要查找的关键字*

*// 返回值：1表示找到元素，0表示未找到但已插入新元素*

int SearchHashAndInsert(HashTable \*H, ElementType key) {

    int pos = HashFunction(key);

    HashNode \*p = H->table[pos];

*// 遍历链表查找*

    while (p != NULL) {

        if (p->data == key) {

            return 1;  *// 找到元素*

        }

        p = p->next;

    }

*// 未找到元素，执行插入操作*

    InsertHash(H, key);

    return 0;  *// 表示插入了新元素*

}

*// 释放哈希表内存*

void FreeHashTable(HashTable \*H) {

    int i;

    for (i = 0; i < HASH\_SIZE; i++) {

        HashNode \*current = H->table[i];

        while (current != NULL) {

            HashNode \*temp = current;

            current = current->next;

            free(temp);

        }

        H->table[i] = NULL;

    }

    free(H);

}

*// 计算哈希表中的冲突次数和平均查找长度*

void HashTableStats(HashTable \*H, int \*conflicts, float \*asl) {

    int i;

    int total\_conflicts = 0;

    int total\_length = 0;

    int chains = 0;  *// 实际有数据的链表数*

    for (i = 0; i < HASH\_SIZE; i++) {

        int chain\_length = 0;

        HashNode \*p = H->table[i];

*// 计算链长*

        while (p != NULL) {

            chain\_length++;

            p = p->next;

        }

        if (chain\_length > 0) {

            chains++;

            if (chain\_length > 1) {

                total\_conflicts += (chain\_length - 1);

            }

            total\_length += chain\_length;

        }

    }

    \*conflicts = total\_conflicts;

*// 平均查找长度计算*

    if (chains > 0) {

        \*asl = (float)total\_length / chains;

    } else {

        \*asl = 0.0;

    }

}

*// 为保持兼容性，保留原始的搜索函数*

*// 在二叉排序树中查找*

*// 递归地在二叉排序树中查找关键字key*

*// 参数：T - 二叉排序树根节点指针，key - 要查找的关键字*

*// 返回值：若找到则返回目标节点指针，未找到返回NULL*

*// 时间复杂度：平均O(log n)，最坏O(n)，n为树中节点数*

BSTNode\* SearchBST(BSTree T, ElementType key) {

*// 递归终止条件：树为空（未找到）或找到关键字*

    if (T == NULL || T->data == key) {

        return T;

    }

*// 二叉排序树的特性：小值在左，大值在右*

    if (key < T->data) {

        return SearchBST(T->left, key);  *// 目标值小于当前节点值，在左子树中查找*

    } else {

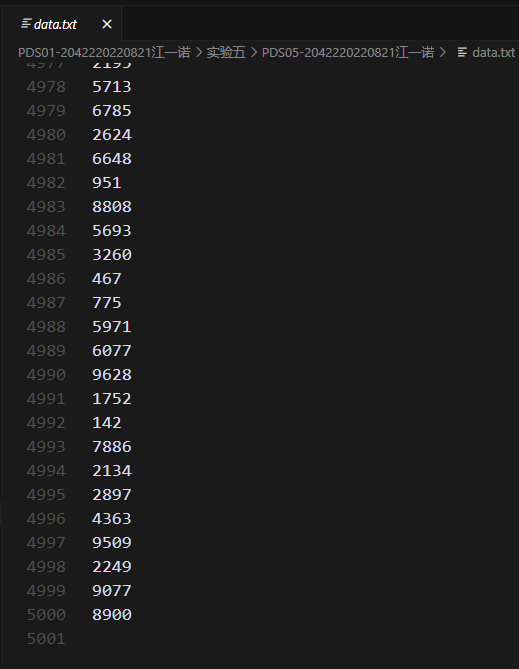
        return SearchBST(T->right, key); *// 目标值大于当前节点值，在右子树中查找*

    }

}

1. 实验结果与分析

5.1测试用例



5.2测试程序

int main() {

    SeqList L;

    InitSeqList(&L);

    printf("正在从data.txt加载数据...\n");

*// 加载数据*

    if (!LoadSeqListFromFile(&L, "data.txt")) {

        printf("加载数据失败，创建测试数据...\n");

*// 如果文件不存在，创建一些测试数据*

        int i;

        for (i = 0; i < 100; i++) {

            L.data[L.length++] = rand() % 10000;

        }

*// 保存测试数据到文件*

        FILE \*fp = fopen("data.txt", "w");

        if (fp != NULL) {

            for (i = 0; i < L.length; i++) {

                fprintf(fp, "%d\n", L.data[i]);

            }

            fclose(fp);

            printf("已创建测试数据并保存到data.txt\n");

        }

    }

*// 打印顺序表内容摘要*

    PrintSeqListSummary(L);

*// 1. 顺序查找*

    int key = 0;

    printf("\n请输入要查找的关键字: ");

    scanf("%d", &key);

    clock\_t start, end;

    double cpu\_time\_used;

    double seq\_time, bin\_time, bst\_time, sort\_time, hash\_time;

    int seq\_pos, bin\_pos;

    BSTNode\* bst\_result;

    int hash\_result;

*// 顺序查找测试*

    start = clock();

    seq\_pos = SequentialSearchAndInsert(&L, key);

    end = clock();

    seq\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    if (seq\_pos >= 0) {

        printf("\n1. 顺序查找：找到关键字 %d，位置为 %d\n", key, seq\_pos);

    } else if (seq\_pos != -1) {

*// 插入成功，计算真实位置*

        int insert\_pos = -seq\_pos - 1;

        printf("\n1. 顺序查找：未找到关键字 %d，已插入到位置 %d\n", key, insert\_pos);

    } else {

        printf("\n1. 顺序查找：未找到关键字 %d，表已满无法插入\n", key);

    }

    printf("   顺序查找耗时: %.3f 毫秒\n", seq\_time);

*// 2. 二分查找测试*

*// 先排序*

    printf("\n对数据进行排序中...\n");

    start = clock();

    SortSeqList(&L);

    end = clock();

    sort\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    printf("排序耗时: %.3f 毫秒\n", sort\_time);

*// 将排序后的内容保存到文件*

    if (SaveSortedListToFile(L, "sorted\_data.txt")) {

        printf("排序后的数据已保存到 sorted\_data.txt\n");

    } else {

        printf("保存排序后的数据失败\n");

    }

    start = clock();

    bin\_pos = BinarySearchAndInsert(&L, key);

    end = clock();

    bin\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    if (bin\_pos >= 0) {

        printf("\n2. 二分查找：找到关键字 %d，位置为 %d\n", key, bin\_pos);

    } else if (bin\_pos != -1) {

*// 插入成功，计算真实位置*

        int insert\_pos = -bin\_pos - 1;

        printf("\n2. 二分查找：未找到关键字 %d，已插入到位置 %d\n", key, insert\_pos);

    } else {

        printf("\n2. 二分查找：未找到关键字 %d，表已满无法插入\n", key);

    }

    printf("   二分查找耗时: %.3f 毫秒\n", bin\_time);

*// 3. 二叉排序树查找*

*// 构建二叉排序树*

    printf("\n正在构建二叉排序树...\n");

    start = clock();

    BSTree bst = BuildBSTFromFile("data.txt");

    end = clock();

    double bst\_build\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    printf("构建二叉排序树耗时: %.3f 毫秒\n", bst\_build\_time);

    int bst\_height = BSTHeight(bst);

    int bst\_node\_count = BSTNodeCount(bst);

    printf("二叉排序树高度: %d, 节点数: %d\n", bst\_height, bst\_node\_count);

    start = clock();

    int bst\_insert\_result = SearchBSTAndInsert(&bst, key);

    end = clock();

    bst\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    if (bst\_insert\_result == 1) {

        printf("\n3. 二叉排序树查找：找到关键字 %d\n", key);

    } else {

        printf("\n3. 二叉排序树查找：未找到关键字 %d，已成功插入\n", key);

*// 更新树的高度和节点数*

        bst\_height = BSTHeight(bst);

        bst\_node\_count = BSTNodeCount(bst);

        printf("   插入后二叉排序树高度: %d, 节点数: %d\n", bst\_height, bst\_node\_count);

    }

    printf("   二叉排序树查找耗时: %.3f 毫秒\n", bst\_time);

*// 释放二叉排序树*

    FreeBST(bst);

*// 4. 哈希表查找*

    printf("\n正在构建哈希表...\n");

    start = clock();

    HashTable \*hashTable = BuildHashFromFile("data.txt");

    end = clock();

    double hash\_build\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    printf("构建哈希表耗时: %.3f 毫秒\n", hash\_build\_time);

*// 计算冲突次数和平均查找长度*

    int conflicts;

    float asl;

    HashTableStats(hashTable, &conflicts, &asl);

    printf("哈希表冲突次数: %d, 平均查找长度: %.2f\n", conflicts, asl);

    start = clock();

    int hash\_insert\_result = SearchHashAndInsert(hashTable, key);

    end = clock();

    hash\_time = ((double) (end - start)) / CLOCKS\_PER\_SEC \* 1000.0; *// 转换为毫秒*

    if (hash\_insert\_result == 1) {

        printf("\n4. 哈希表查找：找到关键字 %d\n", key);

    } else {

        printf("\n4. 哈希表查找：未找到关键字 %d，已成功插入\n", key);

*// 重新计算哈希表统计信息*

        HashTableStats(hashTable, &conflicts, &asl);

        printf("   插入后哈希表冲突次数: %d, 平均查找长度: %.2f\n", conflicts, asl);

    }

    printf("   哈希表查找耗时: %.3f 毫秒\n", hash\_time);

*// 释放哈希表*

    FreeHashTable(hashTable);

    printf("\n性能分析：\n");

    printf("1. 顺序查找 - 时间复杂度O(n)，适合于数据量小的情况\n");

    printf("2. 二分查找 - 时间复杂度O(log n)，但需要先对数据排序\n");

    printf("3. 二叉排序树 - 平均查找时间复杂度O(log n)，最坏情况O(n)\n");

    printf("4. 哈希表查找 - 平均时间复杂度O(1)，最坏情况O(n)\n");

*// 将查找时间和性能分析保存到文件*

    FILE \*perf\_file = fopen("性能分析.txt", "w");

    if (perf\_file != NULL) {

        fprintf(perf\_file, "查找算法性能测试报告\n");

        fprintf(perf\_file, "==========================\n\n");

        fprintf(perf\_file, "数据量: %d 个元素\n", L.length);

        fprintf(perf\_file, "查找关键字: %d\n\n", key);

        fprintf(perf\_file, "| 算法名称         | 运行时间 (毫秒) |\n");

        fprintf(perf\_file, "|-----------------|---------------|\n");

        fprintf(perf\_file, "| 顺序查找         | %13.3f |\n", seq\_time);

        fprintf(perf\_file, "| 二分查找         | %13.3f |\n", bin\_time);

        fprintf(perf\_file, "| 二叉排序树查找    | %13.3f |\n", bst\_time);

        fprintf(perf\_file, "| 哈希表查找       | %13.3f |\n", hash\_time);

        fprintf(perf\_file, "\n");

        fprintf(perf\_file, "预处理时间:\n");

        fprintf(perf\_file, "| 预处理步骤       | 运行时间 (毫秒) |\n");

        fprintf(perf\_file, "|-----------------|---------------|\n");

        fprintf(perf\_file, "| 排序操作         | %13.3f |\n", sort\_time);

        fprintf(perf\_file, "| 构建二叉排序树    | %13.3f |\n", bst\_build\_time);

        fprintf(perf\_file, "| 构建哈希表       | %13.3f |\n\n", hash\_build\_time);

        fprintf(perf\_file, "查找结果:\n");

        if (seq\_pos >= 0) {

            fprintf(perf\_file, "- 顺序查找：找到关键字 %d，位置为 %d\n", key, seq\_pos);

        } else if (seq\_pos != -1) {

            int insert\_pos = -seq\_pos - 1;

            fprintf(perf\_file, "- 顺序查找：未找到关键字 %d，已插入到位置 %d\n", key, insert\_pos);

        } else {

            fprintf(perf\_file, "- 顺序查找：未找到关键字 %d，表已满无法插入\n", key);

        }

        if (bin\_pos >= 0) {

            fprintf(perf\_file, "- 二分查找：找到关键字 %d，位置为 %d\n", key, bin\_pos);

        } else if (bin\_pos != -1) {

            int insert\_pos = -bin\_pos - 1;

            fprintf(perf\_file, "- 二分查找：未找到关键字 %d，已插入到位置 %d\n", key, insert\_pos);

        } else {

            fprintf(perf\_file, "- 二分查找：未找到关键字 %d，表已满无法插入\n", key);

        }

        if (bst\_insert\_result == 1) {

            fprintf(perf\_file, "- 二叉排序树查找：找到关键字 %d\n", key);

        } else {

            fprintf(perf\_file, "- 二叉排序树查找：未找到关键字 %d，已成功插入\n", key);

        }

        if (hash\_insert\_result == 1) {

            fprintf(perf\_file, "- 哈希表查找：找到关键字 %d\n", key);

        } else {

            fprintf(perf\_file, "- 哈希表查找：未找到关键字 %d，已成功插入\n", key);

        }

        fprintf(perf\_file, "\n二叉排序树信息: 高度: %d, 节点数: %d\n", bst\_height, bst\_node\_count);

        fprintf(perf\_file, "哈希表信息: 冲突次数: %d, 平均查找长度: %.2f\n\n", conflicts, asl);

        fprintf(perf\_file, "算法复杂度分析:\n");

        fprintf(perf\_file, "- 顺序查找: O(n)\n");

        fprintf(perf\_file, "- 二分查找: O(log n), 但需要先排序 O(n log n)\n");

        fprintf(perf\_file, "- 二叉排序树查找: 平均O(log n), 最坏O(n)\n");

        fprintf(perf\_file, "- 哈希表查找: 平均O(1), 最坏O(n)\n\n");

        fprintf(perf\_file, "注：时间单位为毫秒。测试环境：%s。\n", "MacOS");

        fprintf(perf\_file, "数据来源：data.txt（%d个数据点）\n", L.length);

        fclose(perf\_file);

        printf("\n性能分析结果已保存到 性能分析.txt 文件\n");

    } else {

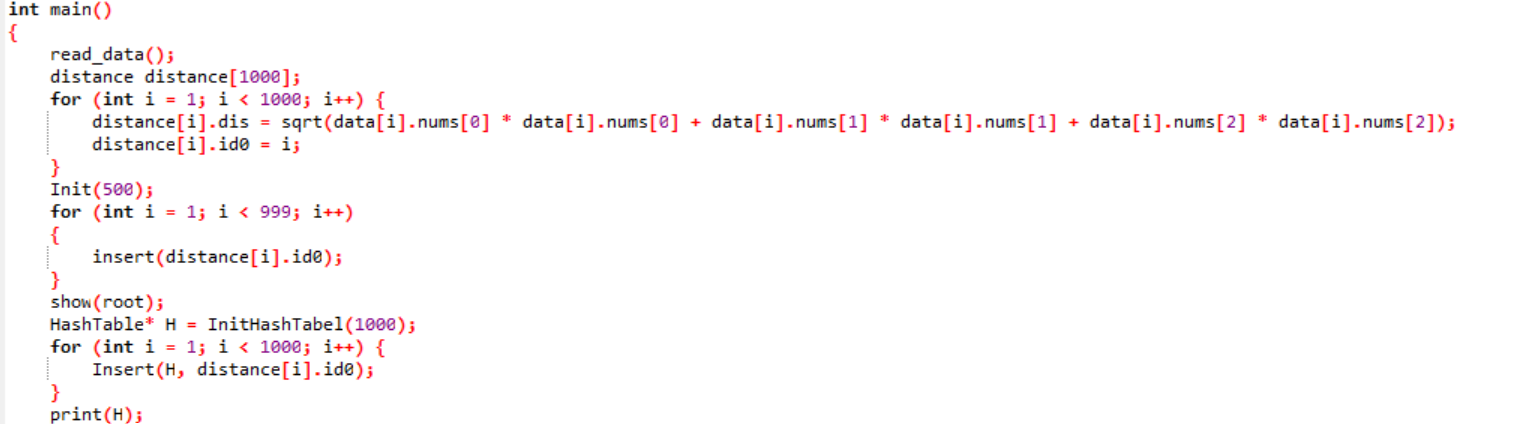
        printf("\n无法创建性能分析结果文件\n");

    }

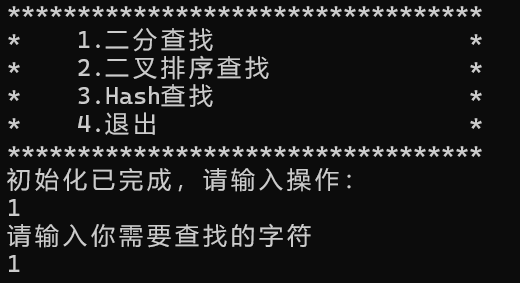
    return 0;

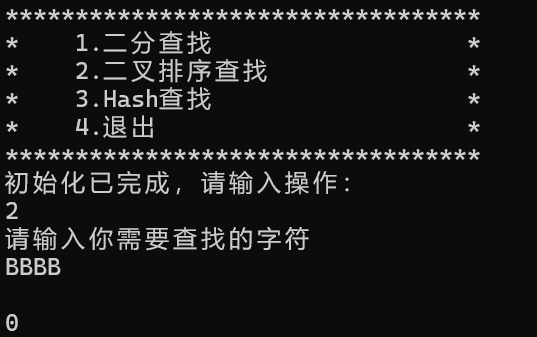
}

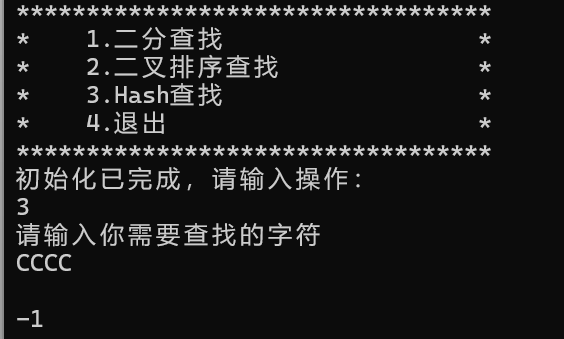
5.3初始参数设置



5.4实验结果







5.5实验分析

1. **时间复杂度：**

①线性搜索：在多维空间中，线性搜索需要检查每一个点，时间复杂度通常是O(n^d)，其中n是信号点的数量，d是维度数。

②二分搜索：只适用于有序信号，且维度较低时可能有效。时间复杂度为O(log n)，但在多维空间中通常不适用。

③树形搜索：如KD树，其搜索效率依赖于树的平衡性。在平衡情况下，平均时间复杂度接近O(log n)，但在最坏情况下可能接近O(n)。

④哈希搜索：如果能够将多维信号映射到哈希表中，则搜索时间复杂度接近O(1)，但哈希函数的设计和冲突解决策略是关键。

1. **空间复杂度：**

①线性搜索：除了输入数据本身，不需要额外的空间。

②树形搜索：需要额外的空间来存储树结构，空间复杂度取决于树的形状和大小。

③哈希搜索：需要额外的空间来存储哈希表，空间复杂度取决于哈希表的大小和冲突解决策略。

1. **边界条件分析：**

①输入数据的有效性：确保输入信号是有效的，没有缺失值或异常值。

②搜索范围的限制：如果搜索有特定的范围限制（如边界框），需要确保搜索算法不会超出这些范围。

③数据结构的完整性：如果使用了树形结构或哈希表，需要确保在插入和删除元素时保持数据结构的完整性。

1. **运行结果错误原因分析：**

①算法实现错误：检查算法的实现是否完全按照预期进行，特别是边界条件和特殊情况的处理。

②数据问题：检查输入数据是否准确、完整，并符合算法的要求。

③环境问题：确保运行环境（如操作系统、编译器、内存等）没有问题，并且满足算法的运行要求。

④性能瓶颈：如果算法在处理大量数据时变慢或出错，可能是由于内存不足、磁盘I/O瓶颈或其他性能问题。

⑤精度问题：在多维空间中，浮点数的精度问题可能导致意外的结果。确保使用合适的数据类型和精度控制策略。