《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软件工程 | 姓名 | 梅子羽 | 学号 | 2327406107 |
| 实验布置日期 | | 2024.12.17 | | 提交  日期 | 2024.12.23 | | 成绩 |  |

课程实践实验7：查找

## 一、问题描述及要求

查找算法

各查找算法的实测时间性能包括两个指标：

算法执行的绝对时间和关键字的平均比较次数。

各查找算法要求评测查找成功与不成功的两种情形。

题目：线性表查找算法的实现及性能测试与比较

[问题描述]

在顺序线性表中存放n个整数，n的值由用户输入确定，线性表可以是有序表或无序表。

为了能比较出各种查找算法执行的绝对时间，需要对表中的数据进行较大量的查找，设为m次，m的值也由用户输入确定。

当输入m为1000000时，则对线性表作1000000次查找。

（1）比较在有序表和无序表中进行顺序查找时，查找成功和查找失败时的算法执行的绝对时间和关键字的平均比较次数。

（2）比较在同一有序表中进行顺序查找和二分查找时的时间性能。

（3）比较在同一有序表中进行非递归二分查找和递归二分查找的时间性能。

比较各查找算法在不同情况下的时间性能。

## 二、概要设计

#### 1. ****对实验内容的理解****

本实验通过实现顺序查找和二分查找两类查找算法，分析它们在不同情况下的时间性能和关键字比较次数。重点在于：

* 比较顺序查找在有序表和无序表中的性能差异。
* 比较顺序查找和二分查找在有序表中的性能差异。
* 比较非递归二分查找和递归二分查找的性能差异。

实验旨在探索不同查找算法在数据规模和查找次数较大情况下的时间效率。

#### 2. ****功能列表****

1. **线性表的生成**
   * 支持生成指定大小的无序表。
   * 支持生成指定大小的有序表（从无序表排序得到）。
2. **查找算法的实现**
   * 顺序查找：适用于有序表和无序表。
   * 非递归二分查找：适用于有序表。
   * 递归二分查找：适用于有序表。
3. **性能测试**
   * 测试查找成功与失败的平均比较次数。
   * 测试查找成功与失败的总执行时间。
4. **结果展示**
   * 输出每种查找算法的平均比较次数和时间性能。

#### 3. ****程序运行的界面设计****

1. **启动界面**  
   程序启动后，提示用户输入：
   * 线性表的大小 n。
   * 查找操作的次数 m。

Enter the size of the list (n): 1000

Enter the number of searches (m): 100000

1. **测试结果展示**  
   程序运行后，输出每种算法的平均比较次数和时间性能：

Testing unordered list:

Sequential search average comparisons: 668

Sequential search time: 1821 ms

Testing ordered list:

Sequential search average comparisons: 100

Sequential search time: 123 ms

Non-recursive binary search average comparisons: 8

Non-recursive binary search time: 45 ms

Recursive binary search average comparisons: 8

Recursive binary search time: 50 ms

#### 4. ****总体设计思路****

1. **数据结构**
   * 使用 std::vector<int> 存储线性表，支持动态调整大小和随机访问。
2. **算法设计**
   * 顺序查找：逐个遍历线性表，支持无序表和有序表。
   * 非递归二分查找：基于迭代的方式，在有序表中定位目标。
   * 递归二分查找：通过递归函数定位目标。
3. **性能测量**  
   使用 chrono 记录各算法执行的时间性能，统计查找的比较次数。

#### 5. ****程序结构设计****

* 本实验代码为单文件程序，包含主函数及所有算法实现。
* 结构清晰，模块化组织，各部分功能明确，便于理解和测试。

## 三、详细设计

#### 1. ****主函数设计****

主函数的主要任务是：

* 接受用户输入：线性表的大小 nnn 和查找次数 mmm。
* 根据用户的选择，生成无序或有序的线性表。
* 调用查找算法（顺序查找、非递归二分查找、递归二分查找），并记录相关的时间性能和比较次数。
* 输出测试结果：包括每种查找算法的执行时间和平均比较次数。

**主函数流程：**

1. **输入**  
   程序首先提示用户输入线性表的大小 n 和查找次数 m。

Enter the size of the list (n): 1000

Enter the number of searches (m): 1000000

1. **生成数据**  
   根据用户选择，生成无序表或有序表。如果选择有序表，程序会先生成一个无序表，然后进行排序。
2. **选择查找算法**  
   提示用户选择测试的查找算法，并执行性能测试。
3. **执行测试**  
   对每种查找算法执行 m 次查找，并分别计算平均比较次数和总执行时间。
4. **输出结果**  
   显示每个算法的性能，包括：
   * 查找的平均比较次数；
   * 查找的总执行时间；

#### 2. ****关键算法设计****

1. **顺序查找 (Sequential Search)**  
   顺序查找是最简单的查找算法，遍历整个列表，逐个比较每个元素直到找到目标元素。如果没有找到，则返回失败。
   * 对于无序表，查找顺序是从头到尾逐个检查每个元素；
   * 对于有序表，如果在查找过程中发现当前元素已经大于目标值，可以直接停止查找，因为有序表的剩余部分已经不可能包含目标值。

**难点分析**：

* + **有序表优化**：顺序查找在有序表中不需要遍历整个列表，一旦当前元素大于目标值，可以提前结束查找，减少不必要的比较。
  + **查找成功和失败的处理**：成功时返回比较次数，失败时返回负的比较次数，便于在后续分析中区分成功和失败的情况。

1. **非递归二分查找 (Non-recursive Binary Search)**  
   二分查找通过不断将搜索区间缩小为一半来查找目标元素。该算法要求数据是有序的。
   * 通过比较中间元素与目标值的大小，决定继续在左半区还是右半区查找，直到找到目标元素或者搜索区间为空。

**难点分析**：

* + **非递归实现**：与递归版不同，非递归版需要使用 while 循环控制查找过程，维护搜索区间的左右边界。
  + **查找失败**：若目标值不在数组中，则需要返回一个失败的标识，表示未找到目标元素。

1. **递归二分查找 (Recursive Binary Search)**  
   递归二分查找与非递归版本相同，只是通过递归函数调用来实现。每次递归都会将搜索区间缩小为一半，直到找到目标或区间为空。

**难点分析**：

* + **递归终止条件**：递归版需要明确的终止条件，即当区间为空时，返回失败。
  + **递归调用**：每次递归调用会消耗栈空间，过多的递归可能导致栈溢出（在非常大的数据集上可能需要考虑栈深度）。

1. **性能测试与结果分析**  
   性能测试模块负责测量每个算法的执行时间，并统计查找成功与失败时的比较次数。
   * 使用 chrono 库来测量时间性能，通过循环进行 mmm 次查找操作，记录总时间并计算平均时间。
   * 通过 comparisons 变量记录每次查找的比较次数，并在测试结束后计算出平均比较次数。

**难点分析**：

* + **多次测试与结果统计**：测试中需要对每个算法执行多次查找（m 次），同时分别统计查找成功和查找失败的情况，确保结果的准确性。
  + **成功与失败的区分**：成功时返回实际的比较次数，失败时返回负的比较次数，需要在计算时特别注意对成功和失败的统计。

#### 3. ****难点部分分析****

1. **大数据量的性能测试**  
   在数据量较大时（例如 n=1000,m=1000000n = 1000, m = 1000000n=1000,m=1000000），性能测试的结果可能会受到系统环境和硬件性能的影响。因此，需要确保性能测试过程中对时间的测量足够精确，并且通过多次测试来消除偶然误差。
2. **查找算法优化**  
   对于顺序查找算法，在有序表中提前停止查找是一个显著的优化点。这种优化可以显著减少查找失败时的执行时间。而对于二分查找，虽然它的时间复杂度较低，但仍需要考虑递归版与非递归版的栈深度问题。
3. **递归与非递归实现的对比**  
   递归二分查找虽然实现简单，但在大规模数据时容易导致栈溢出，因此在实际应用中往往选择非递归版本。性能测试中需要特别注意递归调用的深度，尤其是在数据量非常大的情况下。

#### 4. ****关键部分的流程描述****

* **顺序查找流程**：
  1. 初始化比较次数 comparisons = 0。
  2. 遍历线性表中的每个元素：
     + 如果找到目标元素，返回当前的比较次数；
     + 如果当前元素大于目标元素（仅在有序表中），则结束查找；
  3. 如果未找到目标，返回负的比较次数表示查找失败。
* **非递归二分查找流程**：
  1. 初始化左右边界 left = 0, right = n - 1，并初始化比较次数 comparisons = 0。
  2. 在区间 [left, right] 内循环：
     + 计算中间位置 mid；
     + 如果 list[mid] == key，返回当前比较次数；
     + 如果 list[mid] < key，移动左边界 left = mid + 1；
     + 如果 list[mid] > key，移动右边界 right = mid - 1；
  3. 如果未找到目标，返回负的比较次数表示查找失败。
* **递归二分查找流程**：
  1. 初始化比较次数 comparisons = 0。
  2. 递归调用查找函数，每次将搜索区间一分为二，直到找到目标或区间为空。

**正确输出：对应该输入，算法正确时的输出内容**

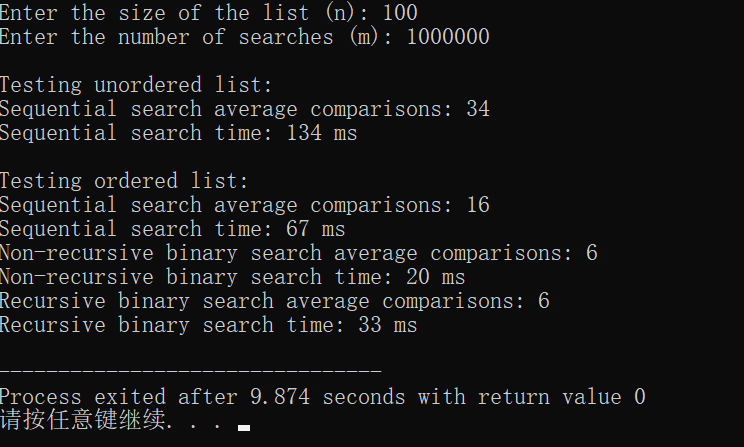
**实际输出：实际测试得到的输出（粘贴输入及运行情况截图）**

**错误原因：如果实际输出与正确输出不符合，分析可能的错误原因**

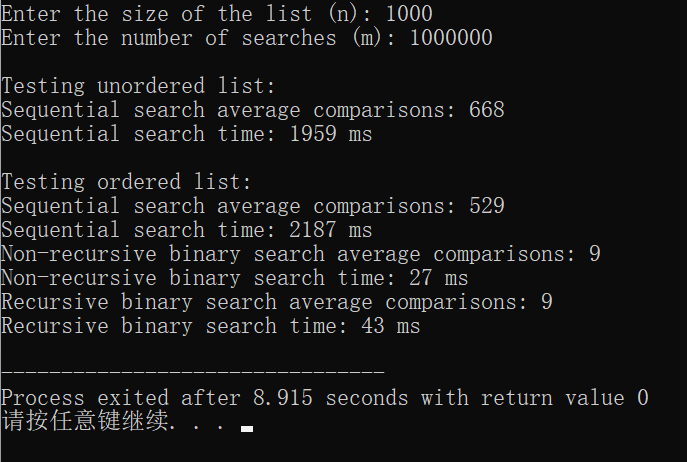
**测试结论：通过、已改正和待修改三种。**

## 四、实验分析与探讨

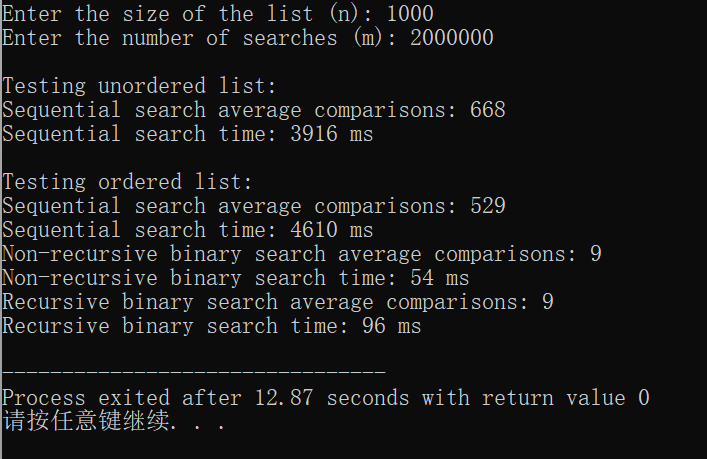
测试样例1：输入表元素个数n = 100， 查找次数m = 1000000；输出展示了随机生成的含有100个整数的有序表和无序表的各种查找情况，属于普通测试，内容如下：



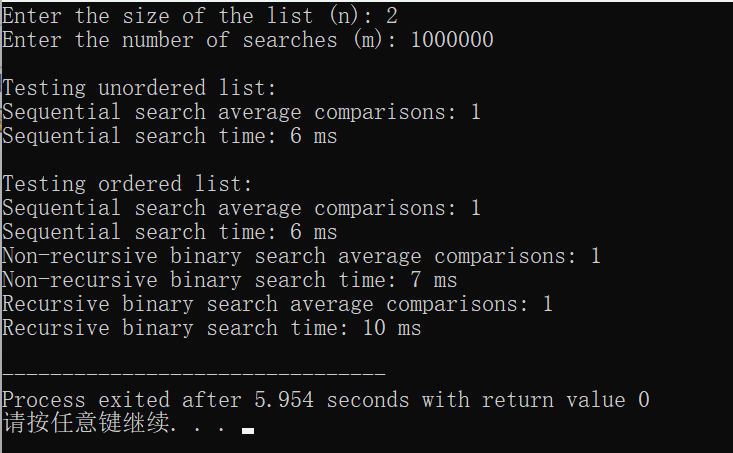
测试样例2：输入表元素个数n = 1000， 查找次数m = 1000000；输出展示了随机生成的含有10000个整数的有序表和无序表的各种查找情况，属于普通测试，内容如下：



测试样例3：输入表元素个数n = 1000， 查找次数m = 2000000；输出展示了随机生成的含有1000000个整数的有序表和无序表的各种查找情况，属于普通测试，内容如下：



测试样例4：输入表元素个数n = 2， 查找次数m = 1000000；输出展示了随机生成的含有1个整数的有序表和无序表的各种查找情况，属于特殊测试，内容如下：



实验结果分析：顺序查找、非递归二分查找和递归二分查找在不同情况下的时间性能差异显著。顺序查找的时间复杂度为 O(n)，无论数据是否有序，查找成功时最坏情况也需要遍历整个表，查找失败时必须扫描整个表，因此在数据量较大时性能较差。而对于有序表，二分查找通过每次将查找范围缩小一半，将时间复杂度降低到 O(logn)，在查找成功时，无论使用递归还是非递归，性能差异较小，主要体现在实现方式上：非递归二分查找通过循环而递归二分查找使用递归调用栈，后者可能导致栈溢出。总的来说，二分查找在有序表中的性能远优于顺序查找，但在无序表中只能退回到顺序查找，而顺序查找对无序表来说则更加高效。

## 五、小结

在本实验中，我实现了多种查找算法（顺序查找、非递归二分查找和递归二分查找），并对它们在不同情况下的性能进行了测试和比较。通过对查找成功和失败时的比较次数和执行时间的统计，深入分析了各个算法在有序表和无序表中的表现。

#### ****完成的内容：****

1. **查找算法实现：**
   * 实现了顺序查找、非递归二分查找和递归二分查找。
   * 对每种查找算法在有序表和无序表中的性能进行了测试。
2. **性能测试：**
   * 对每个查找算法进行了多次查找（m 次），并记录了每次查找的比较次数与执行时间。
   * 成功和失败情况下的查找性能得到了全面的评估，展示了不同算法的优劣。
3. **界面设计：**
   * 设计了简单的用户交互界面，用户可以选择输入线性表的大小、查找次数以及查找方式。

#### ****未完成的内容：****

1. **优化：**
   * 由于时间和资源限制，对于大数据量的查找测试尚未进行深入优化，例如在更大规模的数据集上进行压力测试。
2. **高级数据结构：**
   * 实验中没有引入更复杂的查找算法，例如哈希查找、跳表等。如果能够引入这些更高级的数据结构，可以进一步提高查找效率，尤其是在处理大量数据时。
3. **图形界面：**
   * 当前的界面仅为命令行交互，若采用图形用户界面（GUI），可以增强用户体验。

#### ****选做部分的完成情况：****

本实验没有涉及到其他选做部分，如引入更多种类的查找算法或对算法进行更深入的优化分析。未来可以考虑将这些内容作为后续扩展的方向。

#### ****程序的局限性：****

1. **数据规模**：
   * 程序目前的性能测试适用于中等规模的数据（n 最大为1000）。当数据量更大时，程序的执行效率可能会受到影响，尤其是在顺序查找算法中，随着数据量的增加，执行时间会呈线性增长。
2. **内存管理：**
   * 程序中使用了 vector<int> 存储数据，虽然这种数据结构适合本实验的需求，但在非常大的数据集下，内存占用和管理可能成为瓶颈。
3. **查找次数的影响：**
   * 程序中未考虑到查找次数 m 对结果的影响，特别是在极大数据量下，查找次数过多时可能会导致测试结果的不准确。

#### ****进一步学习与完善的方向：****

1. **引入更复杂的数据结构：**
   * 可以考虑研究哈希查找、AVL树等数据结构，比较它们与线性查找算法的效率差异。
2. **大数据处理：**
   * 随着数据规模的增加，查找算法的执行效率成为关键问题。可以尝试对大数据集进行优化，使用并行计算或分布式计算框架（如 Hadoop）来提高查找性能。
3. **优化算法的实现：**
   * 对递归版二分查找进行优化，避免过深的递归导致栈溢出问题；同时，非递归版二分查找可以进一步优化，以提升执行效率。
4. **图形界面与用户交互：**
   * 目前程序仅使用命令行交互，未来可以使用Qt或其他GUI库，开发一个更加直观的用户界面，使得用户体验更加友好。

#### ****必要的补充说明：****

* **输入文件格式与位置要求：**
  + 当前实验程序通过命令行接收输入参数，用户需要在启动时手动输入表的大小 n 和查找次数 m。
  + 如果想进一步扩展功能，可以考虑通过配置文件输入数据，如CSV格式或JSON格式，来实现批量处理不同的测试用例。
* **输出结果格式：**
  + 输出结果以文本格式显示，展示每种查找算法的平均比较次数和执行时间。未来可以考虑导出为CSV格式，便于后续分析和可视化。

## 附录：源代码

**1、实验环境：Dev-C++ 5.11 C++11标准**

2、

（1）//main.cpp

#include <iostream>

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

#include <queue>

#include <chrono>

using namespace std;

using namespace chrono;

int sequentialSearch(vector<int> &list, int key, bool ordered)

{

int count = 0;

for(int i = 0; i < list.size(); i ++)

{

count ++;

if(list[i] == key)

return count;

else if(ordered && list[i] > key)

break;

}

return -count;

}

int binarySearchNonRecursive(vector<int> &list, int key)

{

int count = 0, left = 0, right = list.size() - 1;

while(left <= right)

{

count ++;

int mid = (left + right) / 2;

if(list[mid] > key)

right = mid - 1;

else if(list[mid] < key)

left = mid + 1;

else

return count;

}

return -count;

}

int binarySearchRecursiveHelper(vector<int> &list, int key, int left, int right, int &count)

{

if(left > right)

return -count;

count ++;

int mid = (left + right) / 2;

if(list[mid] > key)

return binarySearchRecursiveHelper(list, key, left, mid - 1, count);

else if(list[mid] < key)

return binarySearchRecursiveHelper(list, key, mid + 1, right, count);

else

return count;

}

int binarySearchRecursive(vector<int> &list, int key)

{

int count = 0;

return binarySearchRecursiveHelper(list, key, 0, list.size() - 1, count);

}

void performanceTest(vector<int>& list, int m, bool ordered)

{

int key = list[rand() % list.size()]; // Randomly generate search target

int comparisons = 0;

// Sequential search test

auto start = high\_resolution\_clock::now();

for(int i = 0; i < m; i ++)

{

comparisons += abs(sequentialSearch(list, key, ordered));

}

auto end = high\_resolution\_clock::now();

cout << "Sequential search average comparisons: " << comparisons / (double)m << endl;

cout << "Sequential search time: " << duration\_cast<milliseconds>(end - start).count() << " ms" << endl;

if(ordered)

{

// Non-recursive binary search test

comparisons = 0;

start = high\_resolution\_clock::now();

for(int i = 0; i < m; i ++)

{

comparisons += abs(binarySearchNonRecursive(list, key));

}

end = high\_resolution\_clock::now();

cout << "Non-recursive binary search average comparisons: " << comparisons / (double)m << endl;

cout << "Non-recursive binary search time: " << duration\_cast<milliseconds>(end - start).count() << " ms" << endl;

// Recursive binary search test

comparisons = 0;

start = high\_resolution\_clock::now();

for(int i = 0; i < m; i ++)

{

comparisons += abs(binarySearchRecursive(list, key));

}

end = high\_resolution\_clock::now();

cout << "Recursive binary search average comparisons: " << comparisons / (double)m << endl;

cout << "Recursive binary search time: " << duration\_cast<milliseconds>(end - start).count() << " ms" << endl;

}

}

int main()

{

int n, m;

cout << "Enter the size of the list (n): ";

cin >> n;

cout << "Enter the number of searches (m): ";

cin >> m;

vector<int> list(n);

for(int i = 0; i < n; ++i)

{

list[i] = rand() % 100000; // Generate random integers between 0 and 99999

}

cout << "\nTesting unordered list:" << endl;

performanceTest(list, m, false);

sort(list.begin(), list.end());

cout << "\nTesting ordered list:" << endl;

performanceTest(list, m, true);

return 0;

}