根据居民身份证号码查找居民

苏航 202330551331
孙明盛 202330551361
陈宇 202330550201

2024 年 12 月 14 日

1 问题提出与分析

1.1 问题提出

随着信息化社会的发展，居民身份证号码（以下简称“身份证号”）作为每个公民唯一的身份标识，广泛应用于各种场景，如个人信息管理、身份认证、金融交易等。因此，如何高效地存储和查找大量的身份证号码，成为了一个亟待解决的问题。

本实验旨在设计一个系统，能够支持高效查找和存储居民身份证号码。具体要求是：

- 设计并实现数据结构和相应的算法，支持快速查找身份证号码。
- 由于身份证号码是固定长度的（18 位），其中包括出生日期、性别、地区等信息，并且最后一位是校验码（通过特定规则生成），因此身份证号具备一定的结构特性。
- 系统需要根据不同规模的身份证号码集合，设计合理的存储和查找方案。
- 比较多种存储方案，并在实验报告中分析不同方案的优缺点。

1.2 问题分析

1.2.1 身份证号码的特点

- 固定长度：身份证号码是 18 位固定长度，前 17 位为数字，第 18 位为校验码（可为数字或字母 X）。
- 唯一性：每个身份证号码唯一，不会重复。

- 结构化：身份证号码包含地区码、出生日期、性别等信息，具备一定的结构特点。
- 校验规则：最后一位是校验码，根据前 17 位数字通过特定算法计算出来，确保身份证号码的合法性。

这些特点为身份证号的存储和查找提供了不同的优化空间。例如，可以利用号码的唯一性和固定长度进行优化存储，同时还可以借助其结构化特性设计更高效的查找算法。

1.2.2 设计目标

实验设计的目标是在给定的身份证号码集合中，能够高效地进行查找操作。具体来说，需要考虑以下几个方面：

- 查找效率：对于大规模的身份证号码集合，需要支持快速查找操作。查找操作通常是最常见的需求，查询时间复杂度应尽可能低。
- 存储效率：存储效率指的是如何在保证快速查找的前提下，合理使用存储空间。特别是对于大规模数据集，存储空间的使用尤为关键。
- 扩展性：随着身份证号码数量的增加，数据结构应能够扩展，以适应更大规模的数据集。
- 校验规则实现：由于身份证号的最后一位是校验码，需要在插入和查找时进行校验，确保数据的有效性。

1.2.3 不同存储方案分析

为了解决该问题，我们考虑了三种不同的存储方案：

1、外部哈希表:

- 优势：
  - 哈希表提供了常数时间的查找复杂度，适合大规模数据集的快速查找。
  - 操作简单，易于实现。
- 劣势：
  - 哈希冲突会影响查找效率，因此需要设计合理的哈希函数以减少冲突。
  - 哈希表不具备排序功能，对于某些需要按特定顺序查找的应用场景不适用。

2、AVL 树（平衡二叉查找树）:

- 优势：
  - AVL 树是一种自平衡二叉查找树，能够提供对数时间复杂度的查找、插入和删除操作。

  - 支持有序查找，可以方便地进行区间查询。
- 劣势：
  - 平衡操作会带来额外的时间开销，虽然查找效率较高，但插入和删除操作的效率可能稍逊。
  - 对于大量数据时，树的深度增加，可能影响性能。

3、数据库（B 树）

- 优势：
  - B 树特别适用于存储和查询大量数据，尤其是在外部存储（如硬盘）中，B 树可以有效减少磁盘 I/O 操作
  - B 树是一种自平衡的多路查找树，支持高效的范围查询和排序。
- 劣势：
  - B 树的实现和维护较为复杂，适用于需要处理大量数据的场景，对于小规模数据可能显得过于复杂。
  - 对于内存中的小规模数据，B 树可能不如其他数据结构（如哈希表、AVL 树）高效。

1.2.4 方案选择

根据实验要求和目标，我们需要对比不同的方案，在多个规模的号码集合上进行测试，分析不同方案的时间复杂度、空间复杂度以及在不同数据规模下的表现。具体来说，我们将重点考虑：

- 数据规模：从小规模到大规模的性能变化。
- 查询效率：对于每种方案，评估其在查询操作中的表现，尤其是在查找操作较为频繁的情况下。
- 维护成本：分析数据插入、查找等操作的时间复杂度，考虑数据结构的可维护性。

1.3 实验方案

- 身份证号编码规则以及数据生成（使用 $Python$ 中的 $Faker$ 库实现数据生成）
- 方案一：使用 $ExternalHashTable$ 存储数据
  - 使用链表处理哈希冲突
  - 使用二次探测法和再哈希处理哈希冲突
- 方案二：使用 $AVL-Tree$ 存储数据
- 方案三：使用数据库 $SQlite$（$B-Tree$）存储数据

2 身份证号编码规则以及数据生成（使用 $Python$ 中的 $Faker$ 库实现数据生成）

2.1 身份证号编码规则

2.1.1 身份证号号码的含义

18 位身份证号大概可以分为四大部分，分别为 1-6 位地区码、7-14 位出生日期码、15-17 位顺序码以及 18 位校验码。其中 1-6 位地区码代表了户口所在县（市、旗、区）的行政区划代码；7-14 位出生日期码代表出生年月日，前 4 位是年份，中间 2 位是月份，最后 2 位是日期；15-17 位顺序码代表在同一地址码所标示的区域范围内，对同年同月同日生的人编定的顺序号，男性是奇数，女性是偶数；最后一位是校验码，用于检验身份证号的合法性。

2.1.2 校验码的定义

校验位是根据前面十七位数字码，按照 ISO 7064:1983.MOD 11-2 校验码计算出来的检验码。

2.1.3 校验位计算方法

$$result = \left(\sum_{i=1}^{17} a_i w_i \right) MOD11$$

其中：

a_i 表示第 i 位置上的身份证号码字符值；

w_i 表示第 i 位置上的加权因子，数值依据公式 $w_i = 2^{i-1} MOD11$ 计算得出。结果为：$[7, 9, 10, 5, 8, 4, 2, 1, 6, 3, 7, 9, 10, 5, 8, 4, 2]$

2.2 数据生成

2.2.1 使用 $Faker$ 库生成个人信息

$Faker$ 是一个用于生成假数据的 $Python$ 库，它可以用来生成各种模拟数据，包括但不限于姓名、地址、电话号码等，使用`Faker('zh_CN')`创建一个支持中文环境的 $Faker$ 实例，生成的假数据将符合中国的命名和地址习惯。

- 使用`fake.name()`生成姓名
- 使用`fake.address()`生成地址

```
fake = Faker('zh_CN')  # 设置为中文环境
```

生成符合中国的手机号码：

```
def generate_phone_number():
  # 中国手机号一般以 13, 15, 17, 18, 19 开头
  prefix = random.choice(['13', '14', '15', '17', '18', '19'])
  return prefix + ''.join([str(random.randint(0, 9)) for _ in range(9)
     ])
```

生成随机的个人信息：

```
def generate_and_insert_data(num_records):
  data_to_insert = []
  data_for_csv = []  # 用于存储 CSV 数据
  for _ in range(num_records):
      # 生成随机的身份证号码
      id_card_number = str(random.randint(10000000000000000,
         99999999999999999))
      check_code = calculate_check_code(id_card_number) # 计算校验码
      id_card = id_card_number + check_code
      name = fake.name()  # 生成中文姓名
      last_digit = int(id_card_number[-1]) # 判断性别
      if last_digit % 2 == 0:
          gender = "Female"
      else:
          gender = "Male"
      sub_string = id_card[6:14]
      birth_date = f"{sub_string[:4]}-{sub_string[4:6]}-{sub_string
         [6:]}"
      # 生成地址，并去除换行符
      address = fake.address().replace('\n', ' ')
      phone = generate_phone_number()  # 生成符合中国标准的手机号

      # 将数据添加到待插入列表
      data_to_insert.append((id_card, name, gender,
        birth_date, address, phone))

      # 同时准备 CSV 数据
      data_for_csv.append([id_card, name, gender,
        birth_date, address, phone])
```

2.2.2 创建表格

使用SQL CREATE TABLE语句创建一个名为person_info的表。表中包含以下字段：

- id_card_number: 身份证号，唯一 (UNIQUE)
- name: 姓名
- gender: 性别
- birth_date: 出生日期
- address: 地址
- phone: 手机号

```
# 创建表格
cursor.execute('''
CREATE TABLE IF NOT EXISTS person_info (
    id_card_number TEXT UNIQUE,
    name TEXT,
    gender TEXT,
    birth_date TEXT,
    address TEXT,
    phone TEXT
)
''')
```

2.2.3 连接到 *SQlite* 库并创建表

使用sqlite3.connect()连接到person_info.db数据库。如果数据库文件不存在，它将自动创建。

```
conn = sqlite3.connect('ID_CARD/DataBase/person_info.db')
cursor = conn.cursor()
```

2.2.4 批量产生数据，将数据写入 CSV 以及数据库文件

将数据分别写入数据库文件以及 CSV 文件中便完成了数据生成的部分，但由于 *SQlite* 默认只支持utf-8编码，而终端以及 *VisualStudio* 默认的编码为 *GBK* 编码，因此，最终生成的数据会有一点小小的问题，但是经过简单的处理之后即可正常使用。

```
# 批量插入数据到数据库
    cursor.executemany('''
    INSERT OR IGNORE INTO person_info (id_card_number, name, gender,
        birth_date, address, phone)
    VALUES (?, ?, ?, ?, ?, ?)
    ''', data_to_insert)

    # 写入 CSV 文件（编码为 GBK）
    with open('ID_CARD/DataBase/person_info.csv', mode='w', newline='',
        encoding='gbk') as file:
        writer = csv.writer(file)
        writer.writerow(['id_card_number', 'name', 'gender', 'birth_date
            ', 'address', 'phone'])  # 写入表头
        writer.writerows(data_for_csv)  # 写入数据
```

我们选择生成数据量分别为100,000、1,000,000、10,000,000的数据，以便之后测试不同的数据结构处理不同数据量的效率。

3 方案一：使用 *ExternalHashTable* 存储数据

3.1 使用链表处理哈希冲突

在此方案中，我们使用下面的数据结构对身份证号以及个人信息进行存储：

```
struct PersonInfo {
    string name;
    string gender;
    string birthdate;
    string address;
    string phone;

    PersonInfo(const string& n, const string& g, const string& b, const
        string& a, const string& p)
        : name(n), gender(g), birthdate(b), address(a), phone(p) {}
};
```

3.1.1 外部哈希表实现

我们创建了一个ExternalHashTable类,实现了一个外部哈希表,存储了PersonInfo数据，使用链表来处理哈希冲突。每个哈希桶使用std::list来存储值。

主要功能：

- `loadFromFile()`：从 CSV 文件中加载数据并插入哈希表
- `insert()`：根据身份证号插入数据
- `find()`：根据身份证号查找对应的个人信息
- `saveToFile()`：将哈希表中的所有数据保存到文件
- 使用 `std::vector` 存储 1024 个桶，每个桶是一个链表 (`std::list`)
- 每个链表存储多个`pair<string，PersonInfo>`，键是身份证号，值是对应的个人信息

```
class ExternalHashTable {
  private:
      vector<list<pair<string, PersonInfo>>> table;

  public:
      ExternalHashTable();

      // 从CSV文件加载数据
      void loadFromFile(const string& filename);

      // 插入数据到哈希表
      void insert(const string& id, const string& name,
        const string& gender, const string& birthdate,
        const string& address, const string& phone);

      // 查找身份证号对应的个人信息
      PersonInfo find(const string& id);

      // 将哈希表保存到文件
      void saveToFile(const string& filename);

  private:
      // 简单的哈希函数
      unsigned long hashFunction(const string& key) {
      unsigned long hashValue = 0;
      for (char c : key) {
          hashValue = hashValue * 31 + c;
      }
      return hashValue % 1024;  // 返回桶的索引
  }
};
```

这部分代码遍历字符串key（即身份证号id）中的每个字符c，并对hashValue进行逐步更新。具体操作是：

hashValue * 31：将当前的hashValue扩展。选择31是因为它是一个质数，使用质数进行乘法有助于减少哈希冲突。字符c会被转换为它对应的 ASCII 值，并加到hashValue中。使用数学公式表示如下：

$$hashValue = \sum_{i=0}^{17} ord\left(key_i\right) \times 31^{17-i}$$

$$Index = \left[\sum_{i=0}^{17} ord\left(key_i\right) \times 31^{17-i}\right] mod 1024$$

其中：$ord(key_i)$ 为字符 key_i 的 ASCII 值。

3.1.2 时间复杂度分析

哈希函数的计算时间是 $O(k)$，其中 k 是身份证号的长度，身份证号的长度固定为 18，因此计算哈希函数只需要常数时间。

insert():

我们通过emplace_back()直接将数据添加到链表的尾部，因此插入到链表的时间为 $O(1)$。

因此总体的时间复杂度为 $O(1)$，只需要常数时间即可完成插入操作。

find():

查找链表中的元素需要遍历该桶的链表，最坏情况下，当所有元素都被哈希到同一个桶时时间复杂度为 $O(n)$，其中 n 是哈希表中存储的元素个数，因此查找的时间复杂度为 $O(n)$。

saveToFile():

遍历整个哈希表并写入文件的时间复杂度为 $0(n)$，其中 n 是哈希表中存储的元素个数。

3.1.3 空间复杂度分析

哈希表使用了std::vector<list<pair<string，PersonInfo>>>来存储数据，每个元素都是一个pair<string，PersonInfo>，假设我们有 n 个数据，哈希表的空间复杂度是 $O(n)$，其中 n 是哈希表中存储的元素个数。

另外，每个pair<strin2，PersonInfo>中的string也占用一定的空间。因此，存储每个元素的空间复杂度为 $O(m)$，其中 m 是每个string的平均长度 (例如身份证号、姓名等的长度)。

因此，总体空间复杂度为 $O(n*m)$，其中 n 是存储的元素个数，m 是每个元素的平均数据大小。

3.1.4 测试结果分析

表 3.1: 使用外部哈希表存储数据（使用链表处理哈希冲突）

数据集大小	100,000	1,000,000	10,000,000
`loadData()`	0.2125470	1.9399900	20.348400
`insert()`/s	0.0000018	0.0000023	0.0000018
`find()`/s	0.0007023	0.0006479	0.0012411

通过数据可知，无论数据集大小如何变化，`insert()`函数的平均执行时间几乎保持不变。这表明该方案在处理大量数据时的插入性能仍然很稳定。查找的效率虽然比插入的效率低很多，但依旧能够轻松在千万级别的数据集中查询。不过加载数据至内存中的操作花费的时间很长，因此，加载数据这一操作成为了该方案的瓶颈。

3.1.5 最大数据集支持分析

哈希表大小：哈希表的大小固定为 1024(即 1024 个桶)。理论上，哈希表可以支持无限量的元素，但随着元素的增加，哈希冲突会增加，性能会下降。

最大数据集的大小：尽管采用了外部哈希表并将所有数据存储在磁盘上，但在执行查找和插入操作之前，仍需将数据加载到内存中。由于内存容量远远小于磁盘容量，这种方法在处理大规模数据集时存在局限性。特别是在处理千万级数据时，数据加载到内存的过程变得相当耗时，这一步骤成为了整个操作的瓶颈。因此，这种方案在应对亿级别数据集时，会受到内存容量和数据读取效率的双重限制。然而，对于较小规模的数据集，该方案的实施较为简便，且易于管理，$O(n)$ 的查找效率以及 $O(1)$ 的插入效率也十分高效。

3.2 结合二次探测法和再散列处理哈希冲突

我们使用与之前相同的数据结构存储身份证号以及个人信息。

3.2.1 再散列的实现

`rehash`是在哈希表负载因子 (存储元素数量/哈希表大小) 超过设定阈值时触发的操作。其主要目的是动态扩容并重新分布已有数据，以减少冲突和提高查询效率。

在insert()方法中，检查负载因子:

```
  if ((double)num_elements / table_size > load_factor_threshold) {
    rehash(); // 动态扩容
}
```

当负载因子超过 0.8 时，触发rehash。

接下来扩容，将哈希表容量扩展为当前容量的两倍:

```
  size_t new_size = table_size * 2;
  vector<pair<string, PersonInfo>> new_table(new_size, { "", PersonInfo
     () });
  vector<bool> new_occupied(new_size, false);
```

之后重新分布数据，遍历旧表，找到所有已占用的槽位，再对每个键重新计算哈希值，并插入新表中:

```
  table = move(new_table);
  occupied = move(new_occupied);
  table_size = new_size;
```

通过扩容和重新分布，减小了每个槽位的冲突链长度，从而提高了查询和插入效率。

3.2.2 二次探测法

二次探测法是解决哈希冲突的一种开放寻址方法，通过平方探测间距，尽量分散冲突元素，减少连续冲突。

探测公式:

$$index_{new} = \left(index_{original} + probe^2\right) \%table\_size$$

其中，$probe$ 是当前探测次数。

在insert()方法中，当目标槽位被占用时，二次探测法用于计算下一个可能的槽位:

```
  while (occupied[index]) {
    if (table[index].first == id) {
        table[index].second = PersonInfo(name, gender, birthdate,
            address, phone);
        return;
    }
    index = (original_index + probe * probe) % table_size; // 二次探测
    ++probe;
}
```

在find()方法中，二次探测法同样用于解决冲突时的查询:

```
  while (occupied[index]) {
    if (table[index].first == id) {
        return table[index].second;
    }
    index = (original_index + probe * probe) % table_size; // 二次探测
    ++probe;
}
```

该方法避免了线性探测法中连续槽位的冲突问题，使得探测链长度更短，查询和插入效率更高。

3.2.3 时间复杂度分析

find():

- 平均复杂度：
  - 在负载因子较低时 ($\alpha = 0.8$)，冲突链较短，查询复杂度接近 $O(1)$
- 最坏复杂度：
  - 由于二次探测法需要遍历冲突链，最坏情况复杂度为 $O(k)$，其中 k 是冲突链长度
  - 当哈希函数不均匀或表接近满载时，冲突链长度增加，查询可能退化到 $O(n)$

insert():

- 平均复杂度：
  - 插入同样使用二次探测法解决冲突，平均复杂度接近 $O(1)$
- 最坏复杂度：
  - 若插入时需要探测多个槽位 (冲突链较长)，复杂度为 $O(k)$
  - 如果触发rehash()，需要重新分布所有已有数据，插入复杂度上升到 $O(n)$。

`rehash()`:

单次扩容复杂度：

需要遍历当前表中的所有元素，并重新插入到新表中，复杂度为 $O(n)$。

累积扩容复杂度:

假设最终插入的数据量为 N，扩容次数为 $\log N$，累积复杂度为:

$$O\left(N + N/2 + N/4 + ...\right) = O\left(2N\right) = O\left(N\right)$$

3.2.4 空间复杂度分析

哈希表存储键值对和状态数组，空间复杂度为 $O(m)$，其中 m 是哈希表的当前容量。每个键值对存储`std::pair<std::string, PersonInfo>`，当触发 $rehash$ 时，需要同时存储旧表和新表，临时占用约 $O(2m)$ 空间。

3.2.5 测试结果分析

表 3.2: 使用外部哈希表存储数据（使用二次探测法和再哈希处理哈希冲突）

数据集大小	100,000	1,000,000	10,000,000
`loadData()`	0.2581430	2.751360	26.016200
`insert()`/s	0.0000039	0.0000038	0.0000023
`find()`/s	0.0008173	0.0010228	0.0009381

根据数据，插入操作的效率依然非常高，查找效率与使用链表处理哈希冲突相比差距并不太大。但是这种方案再哈希的方式，能够让我们更容易找到适合这个数据库的哈希表的大小。因此，在处理大小变化的数据集时，效率会更高一些。

尽管如此，加载数据的操作仍然占用了绝大多数的时间，该方案仍然没能突破加载数据的瓶颈。

3.2.6 最大数据集支持分析

哈希表大小：建立的哈希表可以通过`rehash`方法不断扩大。理论上可以支持无限量的元素，但随着元素的增加，需要很多次二次探测才能找到合适的位置，性能会下降。

最大数据集的大小：尽管该方案使用了再哈希和二次探测法进行优化，但仍然未能突破内存的限制以及加载数据花费时间太长的问题。不过在面对较小的数据集

时，该方案实现较为简单，便于维护，查找的效率也不错，并且可拓展性很强，在数据集大小未知时，仍然能够满足需求。

4 方案二：使用 $AVL-Tree$ 存储数据

4.1 AVL 树的结构

AVL 树是二叉查找树的一种，在二叉查找树的基础上添加了平衡条件，保证了树的深度是 $O(\log N)$，其中 N 为节点的总数。在此方案中，我们对存储个人信息使用的数据结构和哈希部分的相同，此处不再赘述。

通过一个AVLTree类，实现 AVL 树来存储上面的个人信息，其中节点结构AVLNode包括key，value，headheight，left，right属性。

构造函数：

```
AVLNode(string k, PersonInfo val)  // 键是身份证号码，值是个人信息
```

类主要的功能有：

```
void insert(const string& key, const PersonInfo& value)  // 插入数据
PersonInfo* search(const string& key)  // 查找数据
```

find():

查找的操作和二叉查找树类似，从根节点开始向下递归查找，直到找到叶子节点为止。通过比较键，继续向下递归查找。

```
    PersonInfo* search(const string& key) {
        AVLNode* node = search(root, key);
        if (node) return &node->value;
        return nullptr;
    }
    // 对返回的节点判断，如果和找到的节点相同则为查询成功。
    // 查找操作
    AVLNode* search(AVLNode* node, const string& key) {
        if (!node || node->key == key)
            return node;
    // 如果查到了叶节点或者符合需求的节点，就返回。
        if (key < node->key)
            return search(node->left, key);

        return search(node->right, key);
    }// 否则继续递归查找
```

insert():

插入操作分为 3 步，首先要找到合适的位置。这一步和二叉查找树一样，比较和递归即可。如果该节点不是叶节点，比较key和node->key的大小，然后继续比较其子节点，直到找到合适的位置。

下一步是插入节点，最后检查树的高度是否平衡，如果出现高度不平衡，那么进行适当的旋转，更新高度，进而完成插入。

```
int getBalance(AVLNode* node) // 计算平衡因子
int height(AVLNode* node) // 计算节点高度
一次旋转:
AVLNode* leftRotate(AVLNode* x) // 左旋
AVLNode* rightRotate(AVLNode* y) // 右旋
/*双旋转时，我们分情况讨论，反复应用一次旋转的函数解决问题:
例如
左左情况: */
   if (balance > 1 && key < node->left->key)
            return rightRotate(node);
/*右旋一次即可。
左右情况: */
if (balance > 1 && key > node->left->key) {//判断旋转条件
            node->left = leftRotate(node->left);
            return rightRotate(node);//使用两次旋转，左旋，右旋完成平
                衡
        }
// 右右、右左的情况相反即可。
```

Delete():

删除操作分为 3 步，首先查找到要删除的节点，这一部分和查找一样，递归。然后看该节点有几个子节点，0 个直接删去，1 个则让其复制到该节点再删除原来的子节点，2 个则选择右子树的最小节点值复制到当前节点，再删除原来的最小节点。

```
AVLNode* findMin(AVLNode* node) // 找到当前子树最小的节点
// 最后检查平衡，进行左右旋转或者右左旋转。操作和插入时相同。
```

4.2 时间复杂的分析

find():

在平均情况和最坏情况下，由于 AVL 树是平衡的，其高度始终保持在 $O(\log n)$ 的范围内（其中 n 是树中节点的数量）。因此，查找操作需要遍历的节点数量也为 $O(\log n)$。

`insert()`:

在平均情况和最坏情况下，插入操作都包括找到插入位置、插入新节点和重新平衡树三个步骤。由于 AVL 树的平衡性，找到插入位置需要 $O(\log n)$ 时间。插入新节点后，可能需要通过旋转操作来重新平衡树，但这一步骤的时间复杂度也是 $O(\log n)$（因为最多只涉及到从插入点到根节点的一条路径上的节点）。因此，插入操作的总时间复杂度为 $O(\log n)$。

`Delete()`:

在平均情况和最坏情况下，删除操作都包括找到要删除的节点、删除节点和重新平衡树三个步骤。找到要删除的节点需要 $O(\log n)$ 时间。删除节点后，同样可能需要通过旋转操作来重新平衡树，时间复杂度也是 $O(\log n)$。因此，删除操作的总时间复杂度为 $O(\log n)$。

4.3 空间复杂的分析

AVL 树的空间复杂度主要与其节点数量 n 有关。每个节点都需要存储数据、左右孩子指针和高度信息（有些实现中可能还需要存储平衡因子）。因此，在存储 n 个节点的情况下，不论平均情况还是最坏情况，AVL 树的空间复杂度为 $O(n)$。

4.4 测试结果分析

表 4.1: 使用 $AVL-Tree$ 存储数据

数据集大小	100,000	1,000,000	10,000,000
`loadData()`	0.3639700	4.8887500	83.493100
`insert()`/s	0.0000035	0.0000083	0.0000075
`find()`/s	0.0008741	0.0011096	0.0021834

加载大量数据到内存中的时间成本非常高，尤其是当数据量达到一千万时，加载时间接近一分钟多。

AVL 树的插入操作非常快速且稳定，即使在一千万条数据的情况下，单次插入操作的时间很少。这说明 AVL 树在维持平衡的同时，保持了高效的插入性能。

虽然查找操作比插入操作稍慢一些，但整体来看，AVL 树依然提供了良好的查找性能。即使在最大的数据集中，查找一个元素需要的时间也很少。

总体来说，AVL 树在不同的数据集大小下插入和查找的效率都很高，但是加载数据的时间会随着数据量的增加显著增加。

4.5 最大数据集支持分析

虽然该方案的查找效率和插入效率十分高，但是仍然需要将数据读入内存中才能进行操作，并且由于每次读入数据都可能导致树不平衡，从而进行旋转等操作，花费的时间很长。因此该方案无法胜任更大的数据集，不过均为 $O(\log n)$ 的插入和查找效率十分亮眼。

4.6 前两个方案完成后的总结与反思

至此，我们已经实现了哈希表和 AVL 树的方案，两个方案的插入和查找都很高效，但瓶颈都在于内存的限制以及读取数据的限制。

我们想，能否有一种数据结构，将数据保存在磁盘中，读写时直接从磁盘中读取，而不写入内存，并且查找和插入效率还能保持在 $O(\log n)$ 呢？于是我们将目光放在使用 $B-Tree$ 实现的 $SQlite$ 上，进行方案三。

5 方案三：用数据库 $SQlite$（$B-Tree$）存储数据

5.1 $SQlite$ 的使用方式

5.1.1 $SQlite$ 中的数据存储结构

$SQlite$ 的基本数据结构是表格，每个表格中由多个记录 (`rows`) 和字段 (`colomns`) 组成。对于身份证以及个人信息，常见的做法是设计一个表，表中包含一个记录每个人的身份证号、姓名、性别、出生日期、地址、电话等个人信息。

设计表格结构：

我们要储存的个人信息为：

- 身份证号 (`ID Card Number`)
- 姓名 (`Name`)
- 性别 (`Gender`)
- 出生日期 (`Birth Date`)
- 地址 (`Address`)
- 电话 (`Phone`)

我们可以设计一个表格，表格的每一行存储一位个人的完整信息。

```
1  CREATE TABLE person_info(
2    id_card_number TEXT PRIMARY KEY, -- 身份证号
3    name TEXT NOT NULL,              -- 姓名
4    gender TEXT,                     -- 性别
```

```
    birth_date TEXT,                    -- 出生日期
    address TEXT,                       -- 地址
    phone TEXT                          -- 电话
);
```

解析字段：

- `id_card_number`：由于部分身份证号码中有`'X'`字符，因此使用`TEXT`类型存储
- `name`：姓名字段，`TEXT`类型
- `gender`：性别字段，`TEXT`类型，储存为“Male”或“Female”
- `birth_date`：出生日期字段，`TEXT`类型，储存为“YYYY-MM-DD”格式
- `address`：地址字段，`TEXT`类型
- `phone`：电话字段，`TEXT`类型

关于主键和约束：

- `PRIMARY_KEY`：因为每个人的身份证号在表中是唯一的，不会重复，所以我们将`id_card_number`设为主键
- `NOT NULL`：我们设置了`name`字段为`NOT NULL`，即每个人的姓名必须填写

5.1.2 如何在 *SQlite* 中存储数据

在 *SQlite* 中存储数据时，数据会以行 (`row`) 的形式存情到表格里，每一行代表一条完整的个人信息记录。下面是一个示例记录的存储方式：

表 5.1: 示例表格

id_card_number	name	gender	birth_date	address	phone
330281200506021530	张三	Male	2005-06-02	浙江省宁波市余姚市	13954865435
220503198501068411	李四	Male	1985-01-06	吉林省通化市二道江区	19846515796
440823199305208875	王五	Male	1993-05-20	广东省湛江市遂溪县	15556874692

5.2 *SQlite* 中 $B-Tree$ 的基本结构代码分析

5.2.1 *SQlite* 中 $B-Tree$ 的基本结构

SQlite 的 $B-Tree$ 是一个自平衡的多路搜素树，用于存储数据库中的表数据和索引，每个 $B-Tree$ 节点可以有多个子节点和多个键值对，这使得 $B-Tree$ 在

大规模数据存储和检索时效率较高。

在 $SQlite$ 中，$B-Tree$ 是由以下几个部分组成：

- 页 (Page): 数据车的数据和索引是分布在多个页 (Page) 上的，每个页的大小通常为 1024 字节、2048 字节、4096 字节等。每个 $B-Tree$ 节点 (无论是叶子节点还是非叶子节点) 都存储在页中
- $B-Tree$ 节点：$B-Tree$ 节点由键值对、指向子节点的指针组成。每个非叶子节点存储指向子节点的指针，而叶子节点则存储实际的数据记录或索引条目
- 页头 (Header) 页头：每个页的开始部分存储该页的元数据，如页类型、页的大小、父节点的指针、页的引用计数等

5.2.2 $B-Tree$ 的节点结构

每个 $B-Tree$ 节点通常由以下部分组成：

- 页头 (Header)：包含元信息，如页类型、父节点指针等。
- 键值对 (Key-Value pairs)：在非叶子节点中，存储分割键和指向子节点的指针。在叶子节点中，存储实际的数据或索引条目。

在 $SQlite$ 中，节点的结构和页面的内存管理方式紧密相关。每个 $B-Tree$ 节点都通过页来管理存储，节点的键和子节点指针按照特定的格式存情在页中。

5.2.3 查询操作的实现

查询操作从根节点开始，递归地查找直到找到叶子节点。如果节点是非叶子节点，它会根据查询键决定跳转到哪个子节点继续查找。叶子节点存体着实际的数据或索引条目，查询在这里停止。

以下是一个简化的查找操作的代码分析：

- `isLeaf(pPage)`：检查当前页是否为叶子节点
- `compareKeyWithNode(pIdxKey, pPage, i)`:将查询键与当前节点的键进行比较,决定是否向左或向右子树查找
- `searchLeafNode(pPage, pIdxKey, pRes)`：在叶子节点中执行实际的查找操作

通过这样的方式，sQLte 使用递归的方式来逐级查找每个非叶子节点和叶子节点，直到找到目标数据。

```
1  static int sqlite3BtreeMovetoUnpacked(
2    BtCursor* pCur,  // 游标，指向当前的 B-tree 节点 */
3    UnpackedRecord* pIdxKey,  // 查询的键 */
4    int* pRes  // 返回的结果: 0 表示找到，非 0 表示未找到 */
5  ){
6    BtPage* pPage = pCur->pPage;  // 当前页 */
```

```
    int i = 0;  // 当前节点的索引 */
    int rc;  // 返回值 */

    /* 遍历非叶子节点 */
    while (!isLeaf(pPage)) {
        rc = compareKeyWithNode(pIdxKey, pPage, i);
        if (rc < 0) {
            pPage = getLeftChild(pPage, i);
        }
        else {
            pPage = getRightChild(pPage, i);
        }
    }

    /* 当到达叶子节点时，进行实际的数据查找 */
    rc = searchLeafNode(pPage, pIdxKey, pRes);
    return rc;
}
```

5.2.4 插入操作的实现

插入操作涉及找到合适的插入位置，并根据情况决定是否需要分裂节点。以下是插入操作的步骤：

- 查找插入位置：首先执行查找操作，找到插入位置
- 插入数据：如果插入节点不满，可以直接插入
- 节点分裂：如果插入导致节点满了，节点会分裂，并将中间键上升到父节点，可能会导致父节点的分裂

以下是插入操作的代码分析：

```
  static int sqlite3BtreeInsert(
    BtCursor *pCur,                /* 当前游标*/
    UnpackedRecord *pData,         /* 插入的数据*/
    int flags                      /* 插入标志*/
  ){
    BtPage* pPage = pCur->pPage;
    int rc;
    int i;

    /* 查找插入位置*/
    rc = sqlite3BtreeMovetoUnpacked(pCur, pData, &i);
```

```
    if (rc != SQLITE_OK) {
      return rc;
    }

    /* 检查页是否已满*/
    if (isFull(pPage)) {
      /* 如果满了，执行节点分裂操作*/
      rc = sqlitPage(pPage, i);
      if (rc != SQLITE_OK) {
        return rc;
      }
    }

    /* 插入数据到当前页*/
    rc = insertIntoPage(pPage, pData, i);
    return rc;
  }
```

- `sqlite3BtreeMovetoUnpacked`：执行查找操作，找到插入位置
- `isFull(pPage)`：检查当前页是否已满
- `sqlitPage(pPage, i)`：如果页已满，执行节点分裂操作，将节点分裂成两个，并将中间键上升到父节点
- `insertIntoPage(pPage, pData, i)`：将数据插入到当前节点

5.2.5 节点分裂的实现

当一个节点满时，需要分裂成两个节点，并将中间的键传递给父节点。具体的分裂操作会将父节点的某些键移动到新创建的节点中，这样就会保持树的平衡。

```
static int splitPage(BtPage* pPage, int i)
{
    BtPage* pNewPage;
    int midIndex;
    int rc;

    /* 分裂当前页为两个页 */
    rc = createNewPage(pPage, &pNewPage);
    if (rc != SQLITE_OK) return rc;

    /* 计算中间键 */
    midIndex = pPage->numKeys / 2;
```

```
13
14        /* 将中间的键上升到父节点 */
15        rc = moveMiddleKeyToParent(pPage, midIndex);
16        return rc;
17    }
```

- `createNewPage(pPage, &pNewPage)`：创建一个新的页面
- `moveMiddleKeyToParent(pPage, midIndex)`：将分裂后的中间键上升到父节点

5.2.6 删除操作的实现

删除操作与插入操作类似，需要从树中删除键并保持树的平衡。删除可能导致节点的下溢 (即节点元素不足)，此时需要合并节点或借用兄弟节点的元素来平衡树。

删除操作的流程：

1. 查找并删除键：首先查找待删除的键
2. 处理节点下溢：如果删除后某个节点的元素不足，执行节点合并或借用操作

以下是删除操作的代码分析：

```
1   static int sqlite3BtreeDelete(
2     BtCursor* pCur,   // 当前游标 *
3     UnpackedRecord* pKey  // 要删除的键 *
4   ) {
5     BtPage* pPage = pCur->pPage;
6     int rc;
7
8     /* 查找要删除的键 */
9     rc = sqlite3BtreeMovetoUnpacked(pCur, pKey, NULL);
10    if (rc != SQLITE_OK) {
11        return rc;
12    }
13
14    /* 删除该键 */
15    rc = deleteFromPage(pPage, pKey);
16    if (rc != SQLITE_OK) {
17        return rc;
18    }
19
20    /* 删除后检查节点是否下溢 */
21    if (isUnderfilled(pPage)) {
22        rc = rebalancePage(pPage);
23    }
```

```
24
25      return rc;
26  }
```

- `deleteFromPage(pPage, pKey)`：从页面中删除指定的键
- `isUnderfilled(pPage)`：检查节点是否下溢
- `rebalancePage(pPage)`：执行节点合并或借用操作以恢复树的平衡

5.3 时间复杂度分析

SQlite 通过 $B-Tree$ 实现了高效的查询操作。在 $B-Tree$ 中，查询、插入、删除等操作的平均时间复杂度为 $O(\log N)$，其中 N 是树中节点的数量。由于 *SqLite* 将数据存储在磁盘中，每次操作都需要从磁盘读取数据，因此磁盘 I/O 成为性能的瓶颈。

- 查找操作：使用索引时，可以通过 $B-Tree$ 在 $O(\log N)$ 的时间内完成查询操作，避免全表扫描
- 插入/删除操作：通过 $B-Tree$ 的自平衡特性，插入和删除操作的时间复杂度也是 $O(\log N)$，但这些操作可能会引起磁盘的写操作，进而影响性能

5.4 空间复杂度分析

SQlite 的空间复杂度主要取决于以下几个因素：

- 数据库文件大小：数据库文件的大小取决于数据量的大小、页的大小、索引等。在每个数据页中存储的行数取决于页的大小和每行数据的大小
- $B-Tree$ 的分枝因子：$B-Tree$ 节点的分枝因子（即每个节点包含的子节点数量）影响了树的高度和存储的效率。较高的分支因子能够减少树的高度，从而提高查询效率

5.4.1 影响分支因子的因素

分支因子大致由以下公式计算：

$$\text{分支因子} = \frac{\text{页面大小} - \text{节点头部大小}}{\text{每个条目的大小}}$$

节点头部大小包括了用于存储指针、键、子节点指针等信息的空间，而每个条目的大小则取决于实际的数据结构。

影响分支因子的因素总结：

- 页面大小：页面越大，每个节点可以存储更多的子节点，从而提高分支因子

- 条目大小：条目越小，每页可以存储更多条目，从而提高分支因子

较高的分支因子通常意味着较低的树高度，从而可以减少查询时需要遍历的节点数，提高查询效率。在本方案中，每个条目储存的内容跟仅是简单的字符串，因此每个条目的大小十分小，分支因子较大，树的高度较小，查询效率很高。

5.5 测试结果分析

表 5.2: 用数据库 $SQlite$（$B-Tree$）存储数据

数据集大小	100,000	1,000,000	10,000,000
`openDataBase()`	0.0025092	0.0087273	0.0850549
`insert()`/s	0.0120147	0.0133698	0.0139313
`find()`/s	0.0008904	0.0010099	0.0012580

从插入所用的时间来看 10 万，100 万，1000 万的数据集进行插入，插入时间都基本没有差别。但根据之前的分析，插入的时间复杂度为 $O(\log N)$，插入所用的时间应该随数据集大小增大而变长，但实际测量并非如此。我们猜想这样的结果可能是由于磁盘 I/O 的限制导致的，随着数据量的增加，磁盘 I/O 可能会成为瓶颈。当测试数据集的大小还没有达到使磁盘 I/O 成为瓶颈的程度时，插入时间可能不会有显著变化。

从总体所用的时间来看，该方案表现十分良好，哪怕是千万的数据集，插入和查询的操作所需的时间仍然没有超过 0.1s。

5.6 最大数据集支持分析

$SQlite$ 数据库的最大文件大小主要受到文件系统的限制，$SQlite$ 本身的理论上限是 140TB（即 $SQlite$ 文件最大可以达到字节）。

虽然 $SQlite$ 理论上能处理非常大的数据库文件，但实际上，性能瓶颈通常出现在内存和磁盘 I/O。$SQlite$ 是单线程的，所有操作 (包括查询、插入、删除) 都会在主线程上顺序执行。如果数据库太大，性能可能会迅速下降，特别是磁盘 I/O 和内存访问的速度会影响查询和更新操作的效率。

经过实测，千万级别的数据集保存在`.db`文件中仅需 800MB，再根据测量的插入查询所需的时间，我们一致认为，该方案仍然可以很好地满足 10 亿级别的需求。不过由于 $Python$ 的性能受限，再加上电脑储存空间并不充足，我们没能够生成亿级别的数量级进行测试，不过该方案的优越性依然显而易见。

6 比较不同的方案并总结

6.1 不同方案所需时间对比图

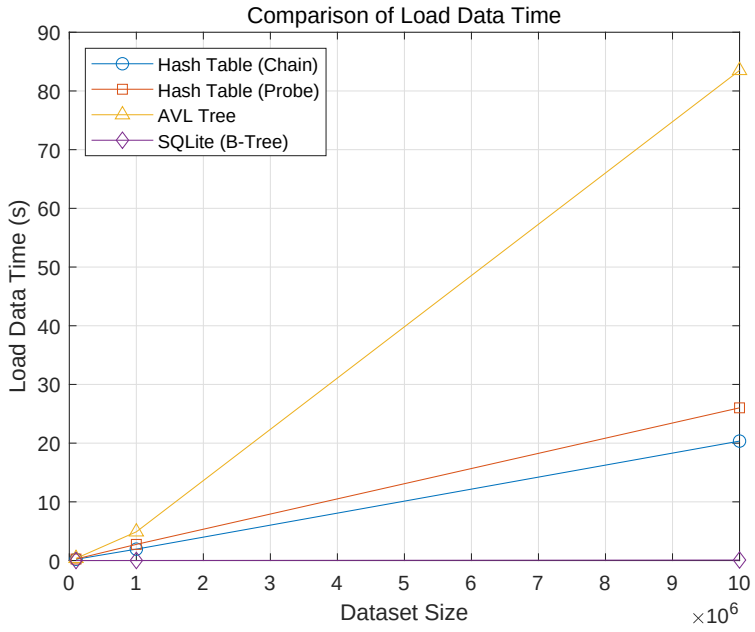

图 6.1: 加载数据时间比较图

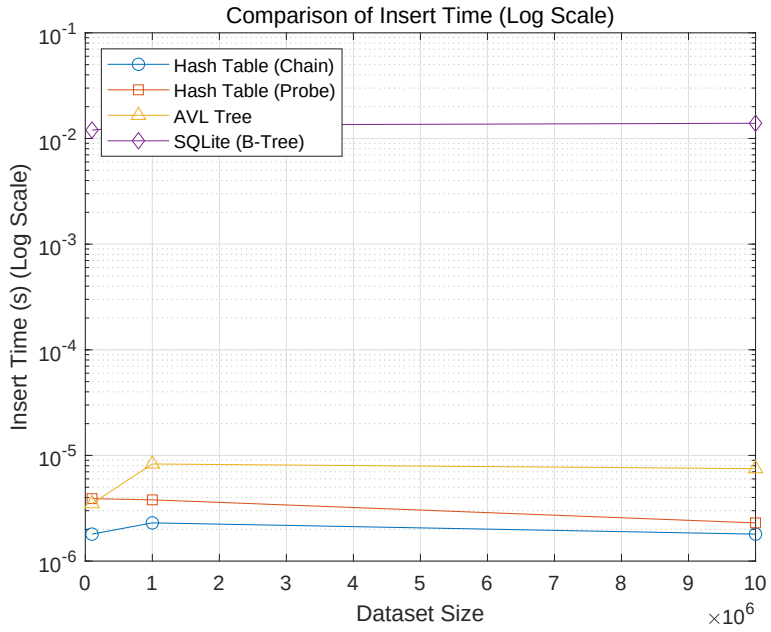

图 6.2: 插入时间比较图

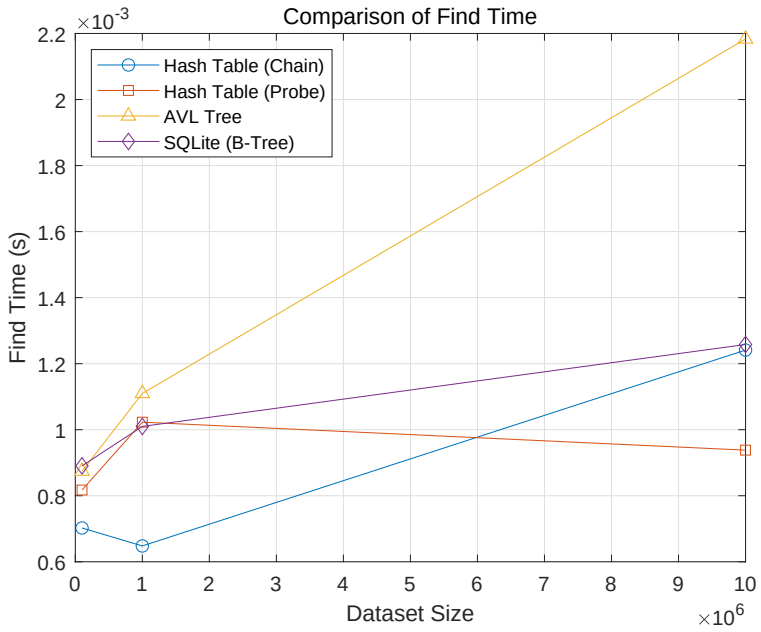

图 6.3: 查询时间比较图

6.2 加载数据时间

哈希表（链地址法）和哈希表（探测法）：

加载时间随着数据集大小的增加而线性增长。总体来说，链地址法比探测法稍快。

AVL 树：

加载时间增长最快，尤其在数据集达到 1,000 万时，加载时间显著增加。这是因为 AVL 树需要在插入每个元素时保持树的平衡，导致额外的时间开销。

$SQlite$（B 树）：

加载时间最短，远低于其他三种方案。由于 $SQlite$ 对数据进行了高效的批量处理和磁盘写入优化。

6.3 插入时间

哈希表（链地址法）和哈希表（探测法）：

插入时间极短，几乎可以忽略不计。插入时间不随数据集大小的增加而显著变化。

AVL 树：

插入时间较短，但比哈希表稍长。随着数据集增大，插入时间略有增加。

$SQlite$（B 树）：

插入时间显著高于其他方案。由于涉及磁盘 I/O 和事务处理，导致插入操作较慢。

6.4 查询时间

哈希表（链地址法）和哈希表（探测法）：

查询时间稳定且较短。哈希表提供了平均 $O(1)$ 的查询时间，不受数据集大小影响。插入时间不随数据集大小的增加而显著变化。

AVL 树：

查询时间随着数据集增大而增加。在大型数据集下，查询性能不如哈希表。

$SQlite$（B 树）：

查询时间稳定，略高于哈希表，但低于 AVL 树。B 树结构在磁盘上有较好的性能表现。

6.5 综合分析

性能：

哈希表在加载、插入和查询操作上表现出色，适用于对速度要求高的场景。AVL 树在维护数据有序性方面有优势，但在大数据集下性能下降明显。$SQlite$（B 树）在加载数据方面优势显著，但插入和查询速度相对较慢。

适用场景：

哈希表适合内存充足、需要快速读写的应用，如缓存、实时数据处理等。AVL 树适用于需要有序数据且数据量较小的场景，如有序列表、范围查询等。$SQLite$（B 树）适合需要持久化存储、事务支持的应用，如嵌入式数据库、本地存储等。

资源消耗：

哈希表需要预先分配足够的内存空间，可能占用较多内存。AVL 树内存占用较为稳定，但由于频繁的旋转操作，CPU 开销较大。$SQlite$（B 树）主要依赖磁盘存储，内存占用较少，但受限于磁盘 I/O 速度。

6.6 总结

如果需要高速的插入和查询，且可以提供足够的内存，哈希表是最佳选择。当需要维护数据的有序性，且数据规模不大时，可以考虑使用 AVL 树。如果需要持久化存储、支持事务和并发，且对性能要求不如内存方案高，选择 $SQlite$（B 树）更为合适。

7 小组分工及成员互评

7.1 苏航对陈宇

陈宇在完成 AVL 树部分的工作中表现出了极高的专业素养和责任感。他能够主动深入理解 AVL 树的平衡机制，并对复杂的旋转、插入和删除操作进行详细实现。面对技术难点，陈宇总是充满耐心，并能通过学习资料和调试解决问题，展现了严谨的学术态度。

陈宇的工作完成得非常出色，AVL 树的实现逻辑清晰，并且能够很好地在大规模数据下验证其效率。他的代码注释详细，便于我们三人交流和理解。此外，他对实验结果进行了深入的分析，特别是 AVL 树的平衡调整机制在查找和插入中的表现。唯一的不足是，实验结果的文档化没有完全细致，这部分需要其他人稍作补充。

7.2 苏航对孙明盛

孙明盛对哈希表部分的工作表现出高度的专注。他不仅熟练实现了链地址法和二次探测法，还在哈希函数的设计上提出了自己的优化方案。他在与团队沟通时很开放，乐于分享自己的设计思路，并积极听取其他人的意见，展示了良好的团队合作精神。

孙明盛的分工任务较为独立，主要集中在哈希表的实现与优化部分。他的工作进度稳定，实验结果清晰明了，尤其是在处理哈希冲突方面展示了高效的技术方法。不过，在文档总结部分稍显不足，未能完全阐明实验数据的具体意义，导致后续撰写报告时需要苏航补充部分内容。

7.3 陈宇对苏航

苏航在完成 $SQlite$ 部分的工作中表现出了卓越的领导能力和责任感。他不仅高效实现了 $SQlite$ 方案，还主动承担了实验报告的撰写任务。苏航在报告中详细阐述了 SQL 存储机制、实验数据和性能对比，同时还利用 $MATLAB$ 进行图表的绘制，使得实验结果更具说服力。工作过程中，他非常注重细节，对实验数据和报告排版的要求非常高，展现了对团队成果负责的态度。

苏航承担了较大的工作量，既完成了 $SQlite$ 部分的技术实现，又独自完成了实验报告的撰写。他分工明确，效率极高，为团队提供了非常清晰的实验报告框架。然而，由于他的任务较多，在与团队协作时，偶尔会出现与其他成员沟通时间不足的情况，但他总是及时调整，保证了分工任务的顺利完成。

7.4 陈宇对孙明盛

孙明盛在完成哈希表部分工作时非常努力，对技术细节非常关注。他能够深入研究哈希函数设计的优化方法，并且在链地址法和二次探测法中提出了解决冲突的有效方案。此外，他对结果的验证十分耐心，特别是在百万级数据集测试中，展示了严谨的态度。

孙明盛完成了哈希表的技术实现，并对结果进行了验证，分工较为明确且执行力强。不过，在团队合作中，他的沟通较为简洁，实验结果的文档化不足，导致后续撰写报告时需要补充部分细节。如果能够进一步细化实验结果的总结，将大幅提升整体团队效率。

7.5 孙明盛对苏航

苏航在团队中不仅承担了技术任务，还主动承担了报告撰写的责任，表现出了极高的责任心。他在完成 $SQlite$ 方案的过程中，能够高效利用工具和框架，并熟练使用 $Latex$ 和 $MATLAB$ 处理实验报告，这为团队节省了大量时间。同时，他对结果分析的深入程度令人钦佩，每个图表和结论都经过反复验证，展现了对工作细节的高度关注。

苏航的分工任务明确且完成度极高，$SQlite$ 的实现和实验报告的撰写均达到了高水准，为团队成果奠定了扎实的基础。尽管在撰写报告时，他倾向于自己完成大部分工作，但他仍会主动与其他人分享报告的进展，并及时采纳大家的建议，这体现了良好的团队合作意识。

7.6 孙明盛对陈宇

陈宇在完成 AVL 树部分的工作中非常专注。他对 AVL 树的复杂机制进行了全面理解，并在实现中展示了扎实的算法功底。他能够耐心处理 AVL 树插入和删除时的平衡调整问题，面对调试中的困难始终保持冷静，展现了良好的问题解决能力。

陈宇的工作完成度高，他的 AVL 树实现准确且高效，为团队实验结果的多样性提供了有力支撑。然而，他在文档化部分的贡献稍显不足，实验结果的总结和分析不够深入，导致后续撰写报告时需要苏航进行补充。如果在实验结果的整理和呈现方面能进一步细化，将会对团队整体效率有更大提升。

8 运行结果

8.1 使用 $ExternalHashTable$ 存储数据（使用链表处理哈希冲突）

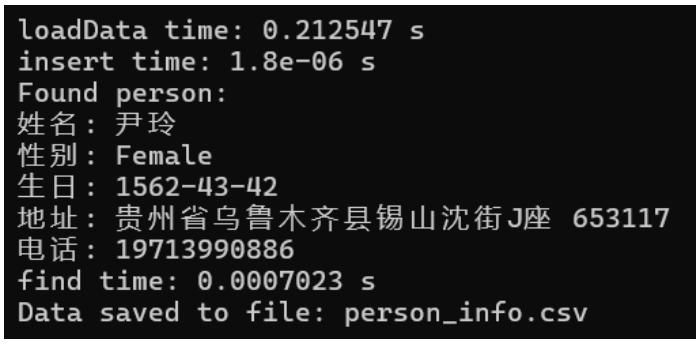

图 8.1: 100,000 数据集

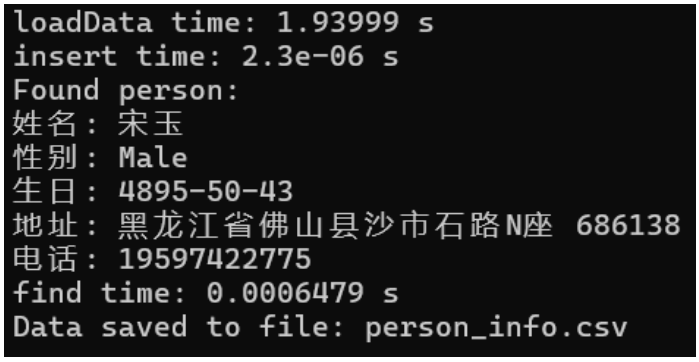

图 8.2: 1,000,000 数据集

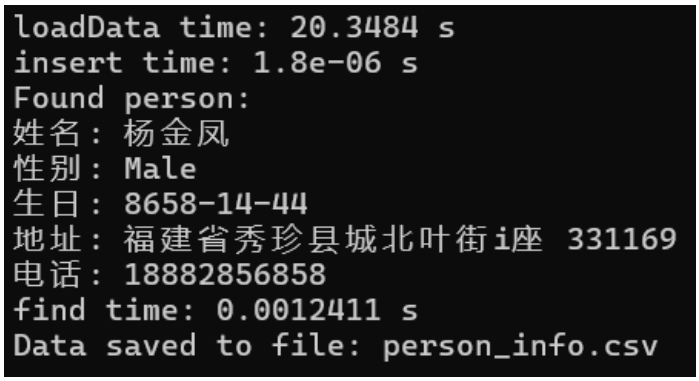

图 8.3: 10,000,000 数据集

8.2 使用 $ExternalHashTable$ 存储数据（使用二次探测法和再哈希处理哈希冲突）

```
loadData time: 0.258143 s
insert time: 3.9e-06 s
Found person:
姓名：严阳
性别：Female
生日：6092-05-32
地址：福建省杭州市友好关岭街y座 132881
电话：13179202042
find time: 0.0008173 s
Data saved to file: person_info.csv
```

图 8.4: 100,000 数据集

```
loadData time: 2.75136 s
insert time: 3.8e-06 s
Found person:
姓名：刘柳
性别：Male
生日：9148-85-35
地址：福建省沈阳市兴山孟街r座 454267
电话：17313652224
find time: 0.0010228 s
Data saved to file: person_info.csv
```

图 8.5: 1,000,000 数据集

```
loadData time: 26.0162 s
insert time: 2.3e-06 s
Found person:
姓名：韩坤
性别：Female
生日：8130-76-25
地址：辽宁省云县黄浦牛路K座 276694
电话：13642757799
find time: 0.0009381 s
Data saved to file: person_info.csv
```

图 8.6: 10,000,000 数据集

8.3 使用 $AVL-Tree$ 存储数据

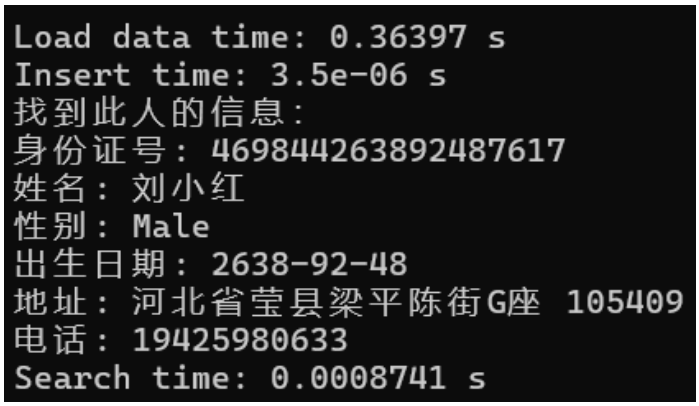

图 8.7: 100,000 数据集

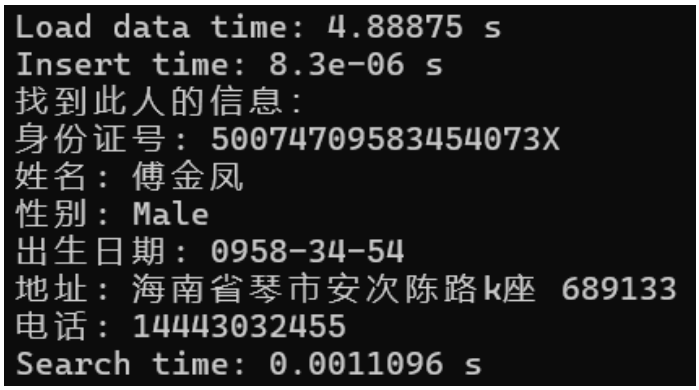

图 8.8: 1,000,000 数据集

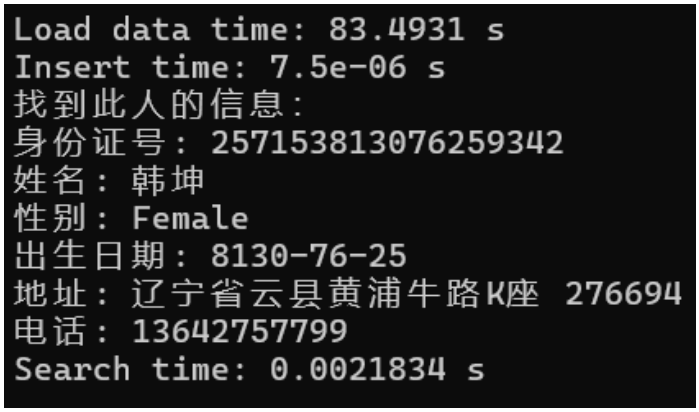

图 8.9: 10,000,000 数据集

8.4 使用数据库 $SQlite$（$B-Tree$）存储数据

```
open db time: 0.0025092 s
insert time: 0.0120147 s
ID Card Number: 33720560920532796X
Name: 严阳
Gender: Female
Birth Date: 6092-05-32
Address: 福建省杭州市友好关岭街y座 132881
Phone: 13179202042
find time: 0.0008904 s
```

图 8.10: 100,000 数据集

```
open db time: 0.0087273 s
insert time: 0.0133698 s
ID Card Number: 9269439148853564 36
Name: 刘柳
Gender: Male
Birth Date: 9148-85-35
Address: 福建省沈阳市兴山孟街r座 454267
Phone: 17313652224
find time: 0.0010099 s
```

图 8.11: 1,000,000 数据集

```
open db time: 0.0850549 s
insert time: 0.0139313 s
ID Card Number: 257153813076259342
Name: 韩坤
Gender: Female
Birth Date: 8130-76-25
Address: 辽宁省云县黄浦牛路K座 276694
Phone: 13642757799
find time: 0.001258 s
```

图 8.12: 10,000,000 数据集