华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析	年级：22级	上机实践成绩：
指导教师：金澈清	姓名：朱天祥	
上机实践名称：散列表	学号：10225501461	上机实践日期：23/4/13
上机实践编号：No.7	组号：1-461	

一、目的

1.熟悉散列表的基本思想。

2.掌握各种散列表的实现方法。

二、实验内容

1. 设计一个数据生成器，输入参数为 N；可以生成N个不重复的随机键或键值对。设计一个操作生成器，输入参数为N’，method；可以生成N’组操作method。操作包括插入和查询。
2. 基于开放寻址法实现哈希表及其插入和查询操作，选择合适的数据规模，计算在不同表的大小和不同已占用数量下的所需时间。
3. 以顺序访问的方式实现插入和查询。选择合适的数据规模，计算在不同表的大小和不同已占用数量下的所需的时间。
4. 对比散列表（哈希表）和顺序访问。
5. （思考题）探究不同散列函数对散列表性能的影响。

三、使用环境

推荐使用C/C++集成编译环境。

四、实验过程

1.写出数据生成器和两种算法的源代码。

1.数据生成器：

```
vector<int> createData(int N){
    srand((unsigned)time(NULL));
    unordered_set<int> set;
    vector<int> arr(N);
    int low = 0;
    int high = 1000000;
    int i = 0;
    while(i < N){
```

```
        int num = rand() % (high - low + 1) + low;
        if(set.find(num) != set.end())
            continue;
        set.insert(num);
        arr[i++] = num;
    }
    return arr;
}
```

用了c++的hashSet结构保证数组没有重复的数据

2.操作生成器

```
vector<int> createMethod(int N){
    srand((unsigned)time(NULL));
    vector<int> method(N);
    for(int i=0;i<N;i++)
        method[i] = rand() % 2;
    return method;
}
```

用函数指针可能有点麻烦，这里直接用0，1随机数生成器生成取值范围为0或1的数组来模拟创建N个函数，值为1时表示调用插入方法，值为0时表示调用查找方法

3.基于开放定址法的哈希表实现

```
class HashTable{
public:
    HashTable(int N){
        size = N;
        arr = vector<int>(size,-0x7fffffff-1);
    }

    void insert(int num){
        int index = hashCode(num);
        if(arr[index] == -0x7fffffff-1){
            arr[index] = num;
            return;
        }
        int i = (index + 1) % size;
        while(arr[i] != -0x7fffffff-1 && i != index)
            i = (i + 1) % size;
        if(i == index){
            return;
        }
        arr[i] = num;
    }

    bool search(int num){
        int index = hashCode(num);
        if(arr[index] == num)
            return true;
        else if(arr[index] == -0x7fffffff-1)
            return false;
        int i = (index + 1) % size;
```

```
        while(arr[i] != num && arr[i] != (-0x7fffffff-1) && i != index)
            i = (i + 1) % size;
        if(arr[i] != num)
            return false;
        return true;
    }

private:
    int size;
    vector<int> arr;
    int hashCode(int key){
        return abs(key % size);
    }
};
```

底层维护一个数组以及哈希函数，数组用于存储用户插入的数据，哈希函数参数为值，返回值为其对应的哈希值，我在这里没有用很复杂的哈希函数，只是用了最基本的取模，并且为了让数组得到充分利用，取模值设为size。需要注意的是负数取模依然是负数，如果不加abs，调完哈希函数后返回值是负数，数组索引值为负数的话会报RuntimeError。还有需要注意查找和插入时都应该是i = (i + 1) % size，而不是直接i++。

4.基于顺序表进行插入和查找

```
class Table{
public:
    vector<int> arr;
    int size;

    Table(int N){
        arr = vector<int>(N);
        size = 0;
    }

    void insert(int num){
        if(size == arr.size())
            return;
        arr[size++] = num;
    }

    bool search(int num){
        for(int i=0;i<size;i++){
            if(arr[i] == num)
                return true;
        }
        return false;
    }
};
```

2.以合适的图表来表示你的实验数据。

main方法

```
int main(){
    for(int i=999;i<=999999;i=i*10+9){
        cout << "数据规模为：" << i << endl;
```

```
        HashTable table1(i);
        Table table2(i);
        vector<int> data = createData(i);
        vector<int> method = createMethod(i);
        for(int k=i/3;k<=i;k+=i/3){
            long time1 = clock();
            for(int j=k-i/3;j<k;j++){
                if(method[j])
                    table1.insert(data[j]);
                else
                    table1.search(data[j]);
            }
            long time2 = clock();
            cout << "占用率从" << (k - (double)i / 3) / i << "到" << (double)k / i
<< "时，哈希表插入以及查询耗时：" << ((double)time2 - time1) / CLOCKS_PER_SEC <<
endl;
        }
        for(int k=i/3;k<=i;k+=i/3){
            long time1 = clock();
            for(int j=k-i/3;j<k;j++){
                if(method[j])
                    table2.insert(data[j]);
                else
                    table2.search(data[j]);
            }
            long time2 = clock();
            cout << "占用率在" << (k - (double)i / 3) / i << "到" << (double)k / i
<< "时，顺序表插入以及查询耗时：" << ((double)time2 - time1) / CLOCKS_PER_SEC <<
endl;
        }
    }
}
```

得到输出数据：

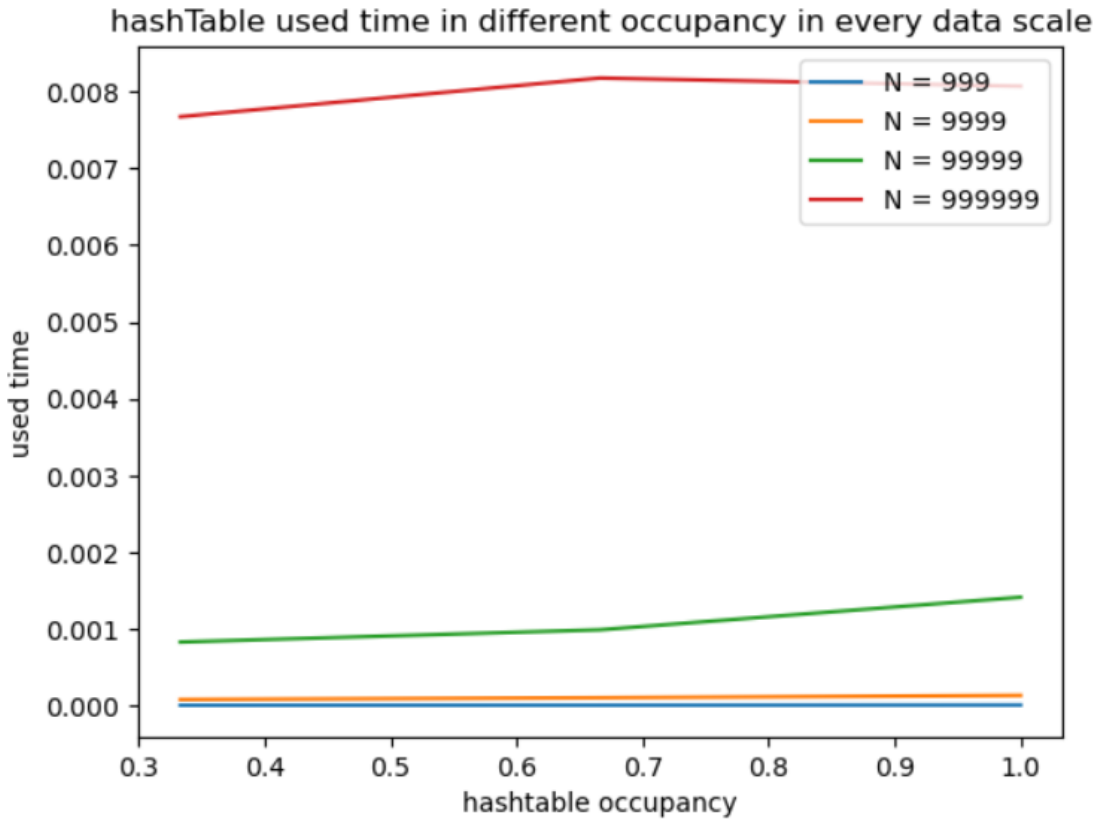
```
数据规模为：999
占用率从0到0.333333时，哈希表插入以及查询耗时：1.2e-05
占用率从0.333333到0.666667时，哈希表插入以及查询耗时：1.3e-05
占用率从0.666667到1时，哈希表插入以及查询耗时：1.5e-05
占用率在0到0.333333时，顺序表插入以及查询耗时：4.7e-05
占用率在0.333333到0.666667时，顺序表插入以及查询耗时：0.000105
占用率在0.666667到1时，顺序表插入以及查询耗时：0.000208
```

```
数据规模为：9999
占用率从0到0.333333时，哈希表插入以及查询耗时：8.6e-05
占用率从0.333333到0.666667时，哈希表插入以及查询耗时：0.000109
占用率从0.666667到1时，哈希表插入以及查询耗时：0.000139
占用率在0到0.333333时，顺序表插入以及查询耗时：0.003508
占用率在0.333333到0.666667时，顺序表插入以及查询耗时：0.010214
占用率在0.666667到1时，顺序表插入以及查询耗时：0.017545
```

```
数据规模为：99999
占用率从0到0.333333时，哈希表插入以及查询耗时：0.000834
占用率从0.333333到0.666667时，哈希表插入以及查询耗时：0.000993
占用率从0.666667到1时，哈希表插入以及查询耗时：0.001417
占用率在0到0.333333时，顺序表插入以及查询耗时：0.350943
占用率在0.333333到0.666667时，顺序表插入以及查询耗时：1.05362
占用率在0.666667到1时，顺序表插入以及查询耗时：1.76222
```

```
数据规模为：999999
占用率从0到0.333333时，哈希表插入以及查询耗时：0.011672
占用率从0.333333到0.666667时，哈希表插入以及查询耗时：0.008175
占用率从0.666667到1时，哈希表插入以及查询耗时：0.008069
占用率在0到0.333333时，顺序表插入以及查询耗时：35.1734
占用率在0.333333到0.666667时，顺序表插入以及查询耗时：105.44
占用率在0.666667到1时，顺序表插入以及查询耗时：176.381
```

1.哈希表在各数据规模下不同占用率时的插入与查询用时曲线：

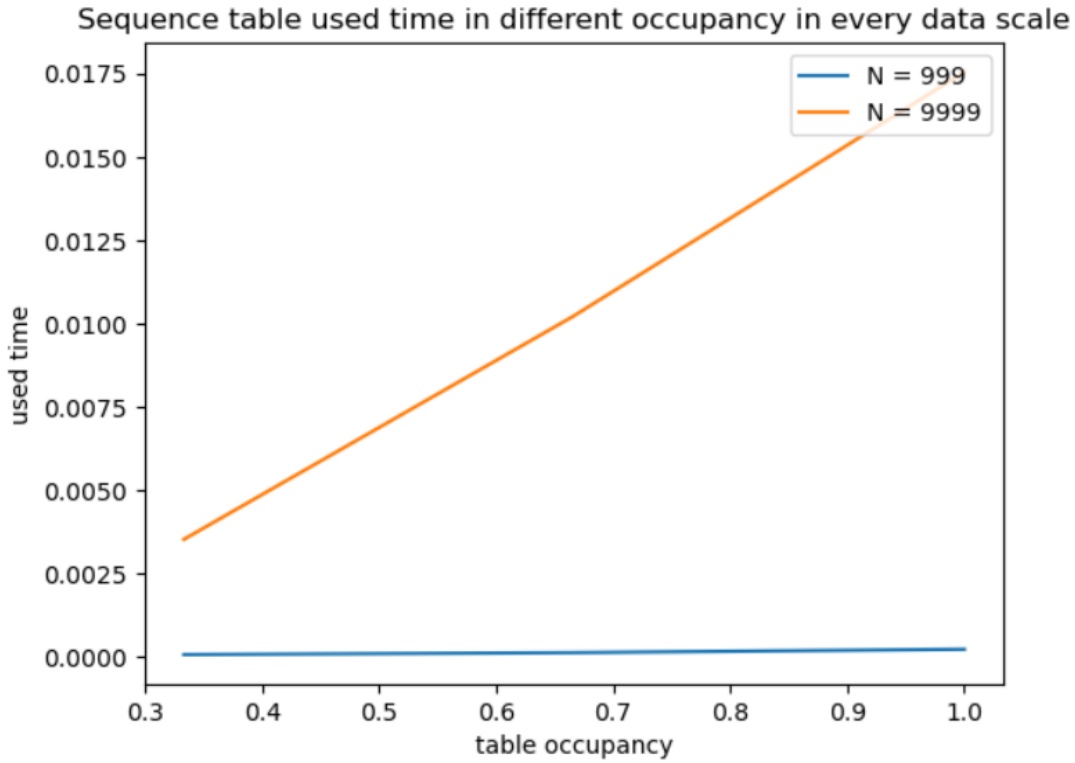


不难看出哈希表的用时并不会很大程度上受到占用率的影响，即使是当前的占用率已经达到约0.5及以上了，其插入和查询的速度依然非常快。而且非常显然地，数据规模越大，哈希表的用时相对会变大，虽然也是呈指数级别地上涨，底数也是比较小的，并不会有耗时变得突然非常大的情况。

2.顺序表在各数据规模下不同占用率时的插入与查询用时曲线：

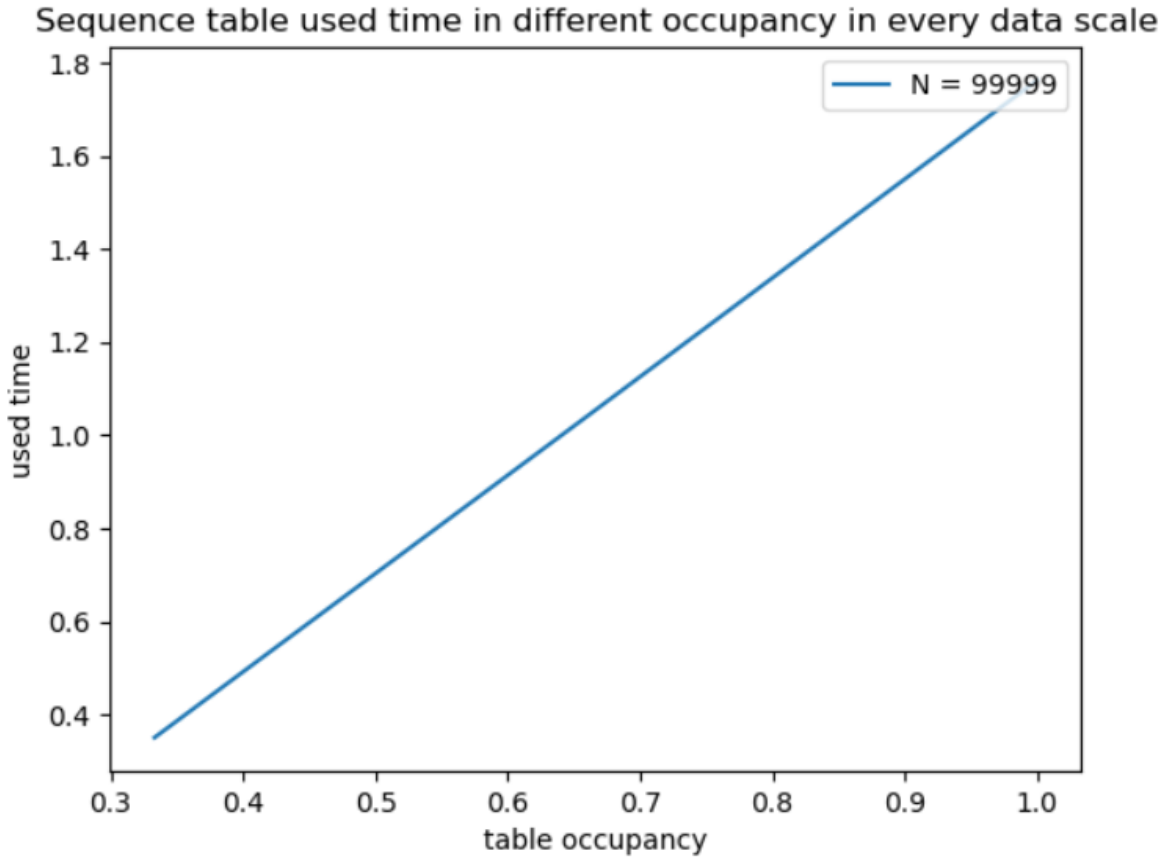
因为数据规模乘10后耗时飙升，如果把N=999 9999 99999 999999的曲线都放在一块的话会使得规模较小的曲线不明显，看起来基本上就是平的，因此把曲线放在不同的图中

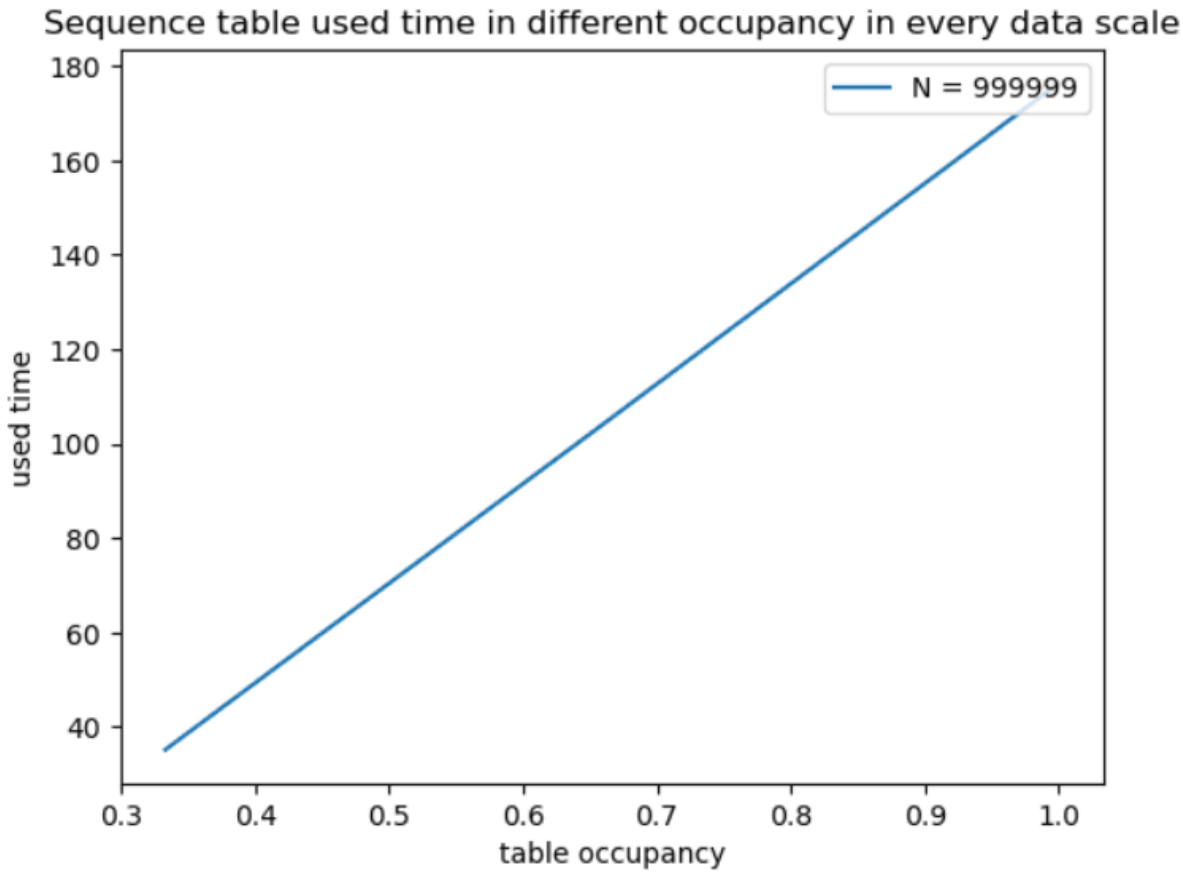
下图是规模分别为999和9999时的顺序表耗时曲线（可以看到顺序表会因为规模增大而耗时急剧上升，让规模比较小的曲线看起来几乎是平的，所以可以猜测顺序表在数据规模较大时效率会变得极差）



根据N=9999这条曲线不难看出顺序表受占用率影响极大，占用率较低时插入和查询的速度比较快，一旦占用率到0.5及以后会使得顺序表的效率急剧下降

下面两个图分别是N=99999和N=999999时顺序表的表现：

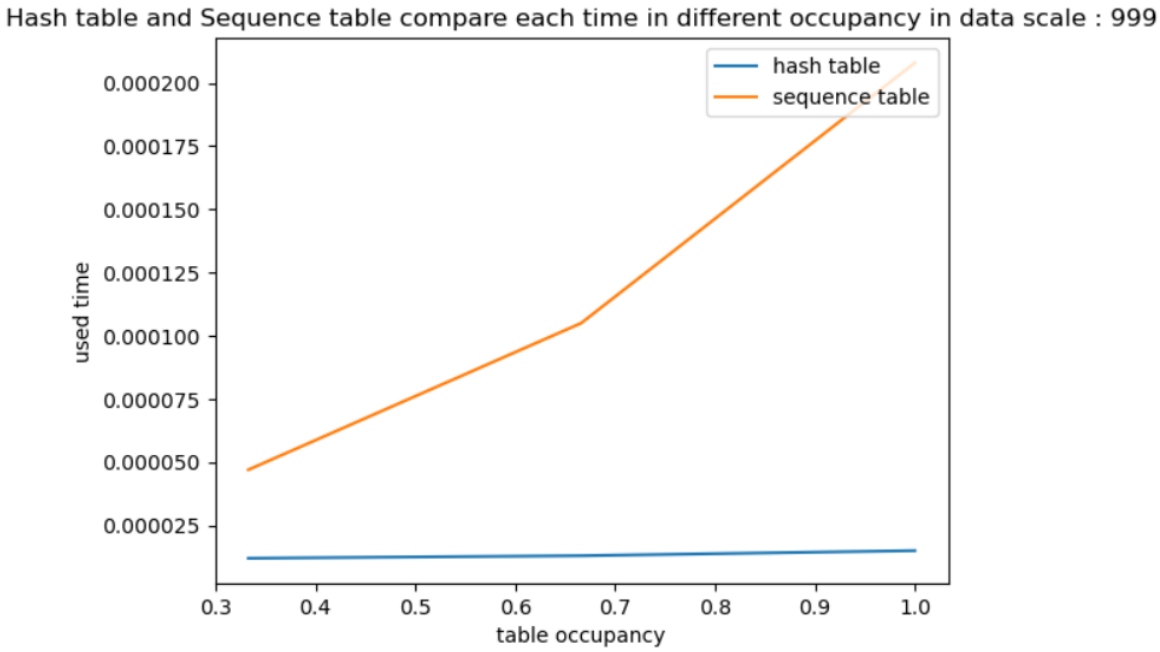




可以看到当数据规模到一百万之后，顺序表在占用率较高时的查询或插入30多万数据的耗时已经达到了3分钟。

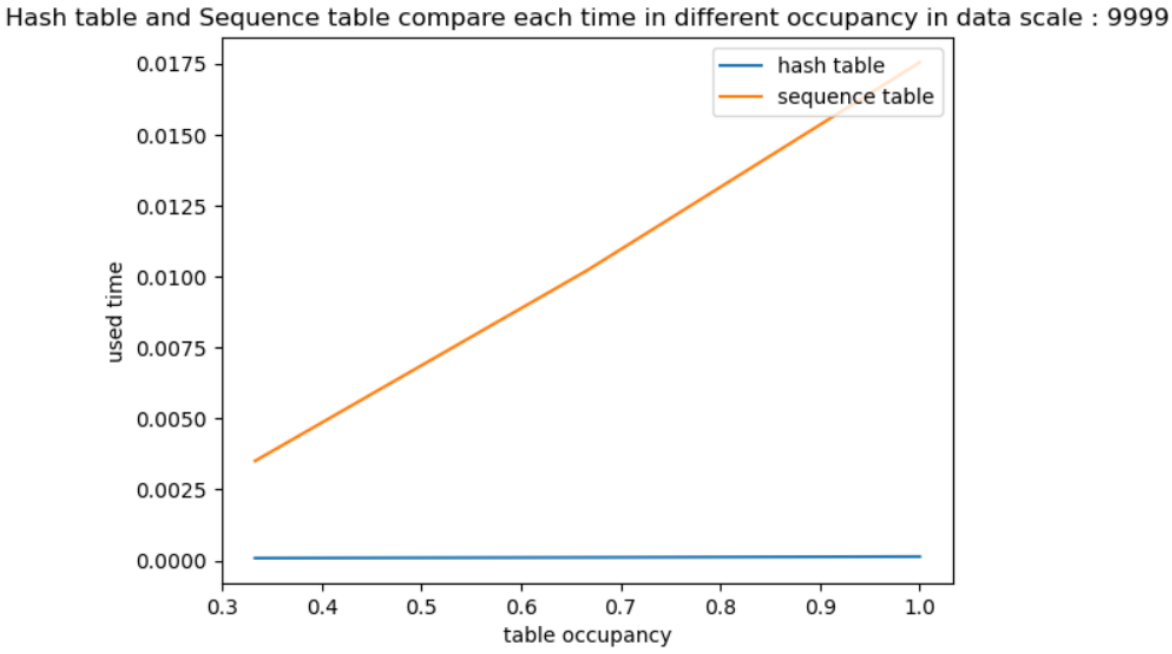
3.对比哈希表和顺序表

数据规模为999时：

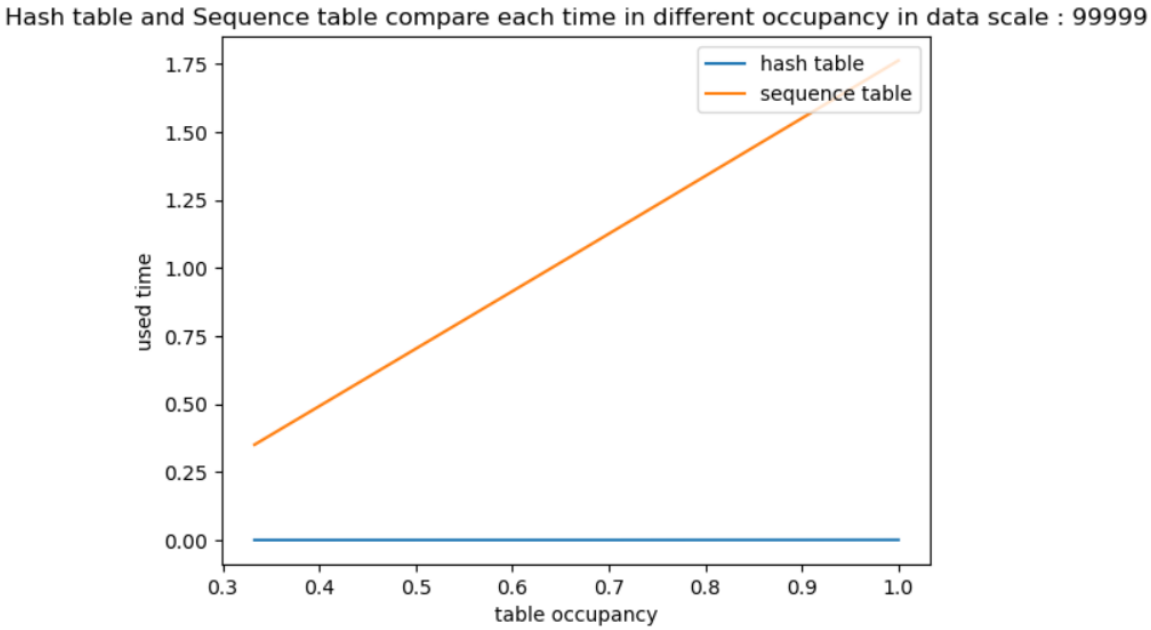


虽然数据规模还不大但是已经能看到哈希表的优势了，哈希表不但不受到占用率的大幅影响，比顺序表的耗时还少了好几个数量级。

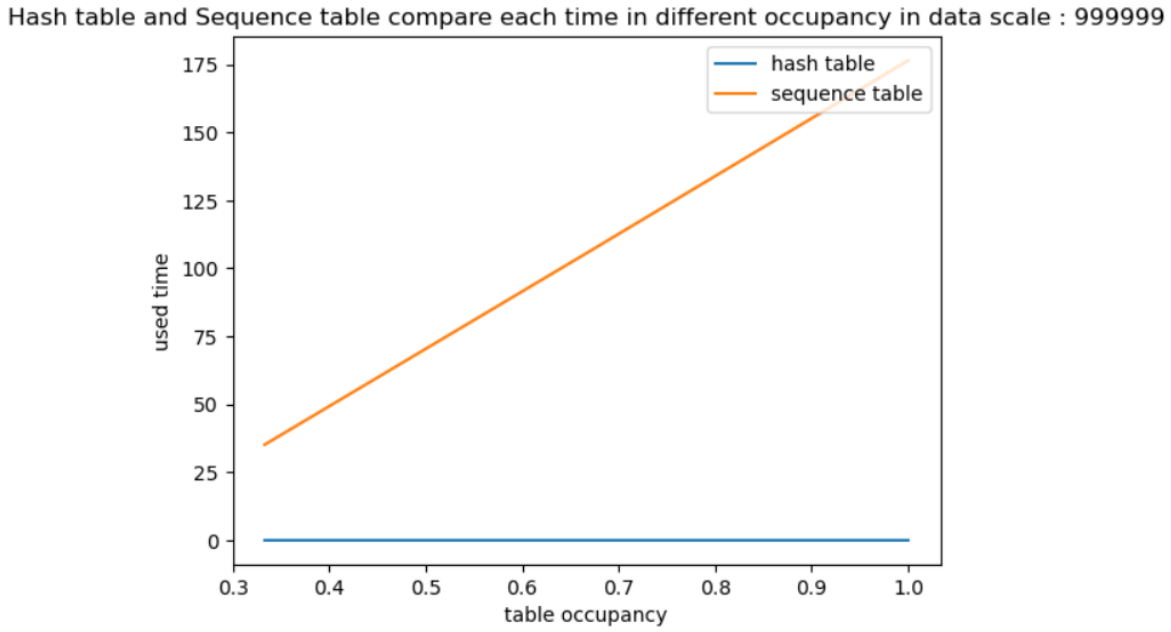
N=9999时二者对比



N=十万时二者对比



数据规模到100000之后顺序表在占用率较高时的耗时已经达到1秒以上



数据规模达到1000000后顺序表的效率非常低，平均插入或查询十万条数据的时间长达30秒，而哈希表依然是毫秒级别的插入查询

4.不同哈希函数对于哈希表性能的影响

1. 简单余数法散列函数

优点：

- 算法简单，计算速度较快
- 适用于处理小型数据集时，其表现稳定
- 性能与关键字有较强的相关性，容易在某些特定场景下得到很好的表现

缺点：

- 容易产生冲突
- 散列值的分布不均匀，如若糟糕导致出现大量的哈希冲突，会极大影响查询效率
- 对应某些特定关键字，可能会产生更多的哈希冲突，影响性能

应用场景：适用于名字、出生日期等非常基础、关注量并不是很高，但是要追求极速匹配的业务场景。

2. 平方取中法散列函数

优点：

- 安全性较高，冲突概率相对较低
- 均匀分布合理，能够有效减少哈希冲突
- 简单可靠，在查询少量数据时表现良好

缺点：

- 在数据集过大时效率偏低
- 固定范围内映射后可能导致卡一个位置

应用场景：适用于海量的查询请求，并且共享相同值域的场景，比如敏感词过滤、关键字搜索。

3. 随机数散列函数

优点：

- 支持很好的均匀分布，可避免哈希冲突
- 安全性较高，难以预测散列结果

- 适用于更大量数据集时，在不知道具体取值范围情况下还是可靠的

缺点：

- 散列函数在创建时产生的开销相对较大
- 单个请求映射关键字时由于额外的计算步骤会导致速度偏低

应用场景：适用于大型数据集，并且快速良好的处理大量查询请求的场景，如路由寻址与明文哈希表，或者加密相关的其中一环。

4. 通用散列函数

优点：

- 处理能力强，无论怎样调整参数，它都可以实现一个高度均匀的散列分布
- 适用于所有数据类型的散列表，支持非数字类型的哈希操作
- 具有较强的安全性，也难以通过暴力破解得到答案

缺点：

- 计算量较大，导致处理时间较大
- 空间占用较大，需要保存多组算法生成的参数

应用场景：适用于对数据安全性要求较高，同时需要处理大型数据集的场景，如加密算法里的随机种子数量，以及哈希表中涉及到复杂类型的元素查询。