作业 HW5 实验报告

姓名：钱子贤 学号：2054170 日期：2021 年 12 月 20 日

实验报告格式按照模板来即可，对字体大小、缩进、颜色等不做要求

实验报告要求在文字简洁的同时将内容表示清楚

1. 涉及数据结构和相关背景

排序：将一组杂乱无章的数据排列成一个按关键字有序的序列。

数据表(datalist):它是待排序数据对象的有限集合。

关键字(key):通常数据对象有多个属性域，即多个数据成员组成，其中有一个属性域可用来区分对象，作为排序依据。该域即为关键字。

每个数据表用哪个属性域作为关键字，要视具体的应用需要而定。即使是同一个表，在解决不同问题的场合也可能取不同的域做关键字。

主关键字：如果在数据表中各个对象的关键字互不相同，这种关键字即主关键字。按照主关键字进行排序，排序的结果是唯一的。

次关键字：数据表中有些对象的关键字可能相同，这种关键字称为次关键字。按照次关键字进行排序，排序的结果可能不唯一。

排序算法的稳定性：如果在对象序列中有两个对象 r[i]和 r[j]，它们的关键字 k[i] == k[j]，且在排序之前，对象 r[i]排在 r[j]前面。如果在排序之后，对象 r[i]仍在对象 r[j]的前面，则称这个排序方法是稳定的，否则称这个排序方法是不稳定的。

2. 实验内容

2.1

HW5：搜索和排序 > 5-1 折半查找

折半查找

描述

二分法将所有元素所在区间分成两个子区间，根据计算要求决定下一步计算是在左区间还是右区间进行；重复该过程，直到找到解为止。二分法的计算效率是 O(logn),在很多算法中都采用了二分法，例如：折半查找，快速排序，归并排序等。

折半查找要求查找表是有序排列的，本题给定已排序的一组整数，包含重复元素，请改写折半查找算法，找出关键字 key 在有序表中出现的第一个位置或最后一个位置，保证时间代价是 O(logn)。若查找不到，返回-1。

输入

第 1 行输入一个正整数 n,表示查找表的长度;

第 2 行输入 n 个有序排列的整数，以空格分割

后面若干行输入为查找的方式 ope 待查找的整数 x

若 ope 为 lower，则表示查找元素在数列中首次出现的位置

若 ope 为 upper，则表示查找元素在数列中最后一次出现的位置

若 ope 为 done，则表示查找结束
输出
查找元素在有序表中的位置，有序表从 0 开始存储。若查找不到，返回-1。

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <algorithm>
#include<string.h>
#include <string>
#include <vector>
#include <cmath>
using namespace std;
class List {
public:
    /**
     * @brief   初始化时生成一个随机数列
     * @param   size int 数列长度
     * @param   max_num int 数列中最大元素
     */
    List(int size, int max_num) {
        for (int i = 0; i < size; i++)
        {
            int n;
            cin >> n;
            list_.push_back(n);
        }
        std::sort(list_.begin(), list_.end());
    }
    ~List() = default;
    /**
     * @brief   清空数组
     */
    void Clear() {
        list_.clear();
    }
    /**
     * @brief   在数组尾部插入一个元素
     */
    void PushBack(int elem) {
        list_.push_back(elem);
    }
    /**
     * @brief   寻找元素最后一个出现的下标
     * @return  找到：下标；找不到：-1；
     */
```

```
    int UpperBound(int elem) {
        int l = 0, r = list_.size() - 1;
        while (l != r) {
            int m = (l + r) / 2;
            if (list_[m + 1] <= elem)
                l = m + 1;
            else
                r = m;
        }
        return (list_[r] == elem ? r : -1);
    }
    /**
     * @brief   寻找元素第一个出现的下标
     * @return  找到：下标；找不到：-1；
     */
    int LowerBound(int elem) {
        int l = 0, r = list_.size() - 1;
        while (l != r) {
            int m = (l + r) / 2;
            if (list_[m] >= elem)
                r = m;
            else
                l = m + 1;
        }
        return (list_[l] == elem ? l : -1);
    }
    /**
     * @brief 在数列中随机取元素
     */
    int RandElem() {
        return list_[rand() % list_.size()];
    }
    /**
     * @brief 打印整个数列
     */
    void PrintList() {
        for (auto i : list_) {
            cin >> i;
        }
    }
private:
    std::vector<int> list_;
};
```

```
int main()
{
    srand((unsigned)time(NULL));
    int test_num = 10; // 测试点个数

        int n, m;
        cin >> n;

            List& L = *(new List(n, n));
            //L.PrintList();
            char* s = new char[10];
            cin >> s;
            while (1) {
                int elem;
                if (strcmp(s, "upper") == 0)
                {
                    cin >> elem;
                    cout << L.UpperBound(elem) << std::endl;
                }
                else if (strcmp(s, "lower") == 0)
                {
                    cin >> elem;
                    cout << L.LowerBound(elem) << std::endl;
                }
                else if (strcmp(s, "done") == 0)
                    break;
                cin >> s;
            }
            delete& L;


    return 0;
}
```

2.1.5 调试分析（遇到的问题和解决方法）

简要描述调试过程

可以使用图片辅助说明

2.1.6 总结和体会

可以说说做此题的收获、分析这道题目的难点和易错点（数据边界、对算法效

率的要求等）

折半查找

如果从文件中读取的数据记录的关键字是有序排列的（递增的或是递减的），则可以用一种更有效率的查找方法来查找文件中的记录，这就是折半查找法，又称为二分搜索。

折半查找的基本思想：减少查找序列的长度，分而治之地进行关键字的查找。他的查找过程是：先确定待查找记录的所在的范围，然后逐渐缩小查找的范围，直至找到该记录为止（也可能查找失败）。例如文件记录的关键字序列为：

（1，3，5，7，9，11，13，17，21，28，32）

该序列包含 11 个元素，而且关键字单调递增。现在想查找关键字 key 为 28 的记录。如果应用顺序查找法进行查找，需要将 28 之前的所有关键字与 key 进行比较，共需 10 次，如果用折半查找可以这样做：

设指针 low 和 high 分别指向关键字序列的上界和下界，即 low=0,high=10。指针 mid 指向序列的中间位置，即 mid=[(low+high)/2]=5。在这里 low 指向关键字 1，high 指向关键字 32，mid 指向关键字 11。

（1，3，5，7，9，11，13，17，21，28，32）

↑ ↑ ↑

low mid high

(1)首先将 mid 所指向的元素与 key 进行比较，因为 key=28，大于 11，这就是说明待查找的关键字一定位于 mid 和 high 之间。这是因为原关键字序列是有序递增的。因此下面的查找工作只需在[mid+1,high]中进行。于是指针 low 指向 mid+1 的位置，即 low=6,也就是指向关键字 13,并将 mid 调整到指向关键字 21，即 mid=8。

（1，3，5，7，9，11，13，17，21，28，32）

↑ ↑ ↑

low mid high

(2)再将 mid 所指向的元素与 key 进行比较，因为 key=28，大于 21，说明待查找的关键字一定位于 mid 和 high 之间。所以下面的查找工作仍然只需在[mid+1,high]中进行。于是指针 low 指向 mid+1 的位置，即 low=9，也就是指向关键字 28，并将 mid 调整到指向关键字 28，即 mid=9。high 保持不变。

（1，3，5，7，9，11，13，17，21，28，32）

↑ ↑

mid high

low

(3)接下来仍然将 mid 所指元素与 key 进行比较，比较相等，查找成功，返回 mid 的值 9。

假设要查找的关键字 key 为 29，那么上述的查找还要继继续下去。由于当前 mid 所指的元素是 28，小于 29，因此下面的查找工作仍然只需在[mid+1,high]中进行。将指针 iow 指向 mid+1 的位置，并调整指针 mid 的位置。这时指针 mid,low 与 high 三者重合，都指向关键字 32，它们的值都为 10。

（1，3，5，7，9，11，13，17，21，28，32）

↑

high

mid

low

再将 mid 所指的元素与 key 进行比较，因为 key=29,小于 32，说明待查找的关键字一定位于 low 和 mid 之间。所以下面的查找工作仍然只需在[low,mid-1]中进行。于是令指针 high 指向 mid-1 的位置，即 high=9，也就是指向关键字 28.这是指针 high 小于指针 low，这表明本次查找失败。

2.2 题目二

HW5：搜索和排序 > 5-2 二叉排序树

二叉排序树
描述
二叉排序树 BST（二叉查找树）是一种动态查找表，或者是一棵空树，或者是具有下列性质的二叉树：
（1）每个结点都有一个作为查找依据的关键字(key)，所有关键字的键值互不相等。
（2）左子树(若非空)上所有结点的键值都小于它的根结点的键值。
（3）右子树(若非空)上所有结点的键值都大于它的根结点的键值。
（4）左子树和右子树也是二叉排序树。

二叉排序树的基本操作集包括：创建、查找，插入，删除，查找最大值，查找最小值等。本题实现一个维护整数集合（允许有重复关键字）的 BST，并具有以下功能：1. 插入一个整数 2.删除一个整数 3.查询某个整数有多少个 4.查询最小值 5. 查询某个数字的前驱

输入
第 1 行一个整数 n，表示操作的个数；
接下来 n 行，每行一个操作，第一个数字 op 表示操作种类：

若 op=1，后面跟着一个整数 x，表示插入数字 x

若 op=2，后面跟着一个整数 x，表示删除数字 x（若存在则删除，否则输出 None，若有多个则只删除一个），

若 op=3，后面跟着一个整数 x，输出数字 x 在集合中有多少个（若 x 不在集合中则输出 0）

若 op=4，输出集合中的最小值（保证集合非空）

若 op=5，后面跟着一个整数 x，输出 x 的前驱（若不存在前驱则输出 None，x 不一定在集合中）

输出
一个操作输出 1 行（除了插入操作没有输出）

```
/**
 * @name    p90.cpp
 * @brief   p90 鏍蜂緰绋嬪簭
 * @author  寮犳劀
```

```
 * @date    2021-12-11
 */
#include <iostream>
using namespace std;

/**
 * @brief 骞宠　浜屽弶鎺掑簭鏍?
 */
class SORTTREE {
public:
    struct NODE {
        NODE* left_, * right_, * parent_;
        int elem_, count_, height_;
        NODE(int elem, NODE* parent) : elem_(elem), parent_(parent) {
            left_ = NULL;
            right_ = NULL;
            count_ = 1;
            height_ = 1;
        }
        ~NODE() {
            if (left_)
                delete left_;
            if (right_)
                delete right_;
        }
        /**
         * @brief 璁＄畻骞宠　搴?
         */
        int Balance() {
            return -(left_ ? left_->height_ : 0) + (right_ ? right_->height_ : 0);
        }
        /**
         * @brief 鏇存柊鑺傜偣楂樺害
         */
        void UpdateHeight() {
            height_ = std::max((left_ ? left_->height_ : 0), (right_ ?
right_->height_ : 0)) + 1;
        }
        /**
         * @name    bool Find(int elem, NODE *&last, NODE *&thiss, NODE *&next, bool
&found)
         * @brief   鏌ユ壘涓€涓   厓绱犵殑瀵瑰簲鑺傜偣 ┍銆佸墠椹辫妭鐐广€佸悗缁ц妭
鐐?
         * @param   elem 寰呮煡鎵剧殑鍏冪礌
```

```
         * @param   last 鍓嶉┍鑺傜偣锛屽鏋滀负NULL锛岃〃绀哄綋鍓嶈妭鐐逛负绗竴涓妭鐐?
         * @param   thiss 瀵瑰簲鑺傜偣锛屽鏋滀负NULL
         * @param   next 鍚庣户锛屽鏋滀负NULL锛岃〃绀哄綋鍓嶈妭鐐逛负鏈€鍚庝竴涓妭鐐?
         * @param   found 鏄惁鎵惧埌鎸囧畾鍏冪礌锛寀rue琛ㄧず鎵惧埌鎸囧畾鍏冪礌锛屽惁鍒欒〃绀烘湭鎵惧埌
鎸囧畾鍏冪礌鐨勫悗缁婧€€鍑?
         * @return   鏄惁鎸囧畾鍏冪礌鐨勫墠椹辫妭鐐逛负绌?
         */
        bool Find(int elem, NODE*& last, NODE*& thiss, NODE*& next, bool& found) {
            if (left_)
                if (left_->Find(elem, last, thiss, next, found))
                    return true;
            if (found) { // 宸茬粡鎵惧埌鍏冪礌锛屽垯鍏冪礌鐨勫悗缁负鍚庣户锛岃€屼笖鍚庣户鏄綋鍓嶈
妭鐐癸紝杩斿洖鐪熴€€鍑?
                next = this;
                return true;
            }
            if (elem_ > elem) { // 鍏冪礌瀵瑰簲鐨勮妭鐐逛笉瀛樺湪锛屽綋鍓嶈妭鐐逛负鍚庣户锛岃繑鍥炵湡
€€鍑?
                thiss = NULL;
                next = this;
                return true;
            }
            if (elem_ == elem) { // 鎸囧畾瀵瑰簲鐨勮妭鐐癸紝杩斿洖鏍囪瘑鎵惧埌锛屽苟缁х画鎼滅储鍚?
                thiss = this;
                found = true;
            }
            else
                last = this; // 鍏冪礌鎸囧畾瀵瑰簲鐨勮妭鐐癸紝鍒欏綋鍓嶈妭鐐瑰厛€煎彲浣滀负鍓嶉
┍鑺傜偣缁х画鎼滅储鍚庣户鍓嶉┍
            if (right_)
                return right_->Find(elem, last, thiss, next, found);
            return false; // 鎵ц鍒版澶勮繑鍥炲亣
        }
    };

    SORTTREE() { root_ = NULL; }
    ~SORTTREE() { delete root_; }
    /**
     * @brief 鎻掑叆涓€涓   厓绱?
     */
    void Insert(int elem) {
        if (!root_) {
            root_ = new NODE(elem, NULL);
            return;
```

```
        }
        NODE* l = NULL, * p = NULL, * n = NULL;
        bool found = false;
        root_->Find(elem, l, p, n, found);

        if (p) {
            p->count_++;
            return;
        }
        // 鍓嶉┍鑺傜偣瀛樺湪涓旀病鏈夊彸瀛╁瓙锛屽皢鏂拌妭鐐规彃鍏ュ埌鍓嶉┍鑺傜偣鐨勫彸瀛╁瓙
鑺傜偣鍗冲彲
        if (l && !l->right_) {
            p = (l->right_ = new NODE(elem, l));
        }
        else {
            p = (n->left_ = new NODE(elem, n));
        }
        KeepBalance(p);
    }
    /**
     * @brief 鍒犻櫎鑺傜偣鍏冪礌
     */
    bool Delete(int elem) {
        if (!root_)
            return false;
        NODE* l = NULL, * p = NULL, * n = NULL;
        bool found = false;
        root_->Find(elem, l, p, n, found);
        if (!p) // 娌℃壘鍒?
            return false;
        if (p->count_ > 1) { // 瀵瑰簲鍏冪礌鏁伴噺澶т簬1鏃讹紝鍙渶鍑忓皯璁℃暟鍗冲彲
            p->count_--;
            return true;
        }
        // 瀵瑰簲鍏冪礌鐨勬暟閲忎负1锛岄渶瑕佸垹闄よ妭鐐?
        if (!p->left_ && !p->right_) { // 瀵瑰簲鑺傜偣涓哄彾瀛愯妭鐐
            if (!p->parent_) {
                root_ = NULL;
            }
            else {
                ChangeParent(p, NULL);
                KeepBalance(p->parent_);
            }
        }
```

```
        else if (!p->left_) { // 娌℃湁宸﹀瓙鏍戠殑鎯呭喌锛堝湪杩欑鎯呭喌锛屾垜浠笉涓嶅彲鑳芥槸鍙跺瓙锛
瀛愮為浠ｆ浛 繁
                                 // 濡傛灉娌℃湁鍙冲瓙鏍戜涔熻兘浠ｈ鍐嗭紝鎵ｅ垹鎺変簡锛屾垜浠
氬綋鍋跺嚭?
            ChangeParent(p, p->right_);
            KeepBalance(p->parent_);
        }
        else { // 鏈夊乏瀛愮為锛岃繘鍓嶉┍鍔涓€瀵逛氦涓嶅瓙鏍戦兘鏈夊嚭鍓嶉┍锛涘 f 浛鎹
繁
            if (p->left_ == l) { // 鍓嶉┍灏辨槸宸﹀瀛?
                l->right_ = p->right_;
                if (p->right_)
                    p->right_->parent_ = l;
                ChangeParent(p, l);
                KeepBalance(l);
            }
            else { // 鍓嶉┍涓嶆槸宸﹀瀛愶紝涔熶笉鏄彾瀛愶紝闇€瑕佸鐞嗗鍚?
                NODE* t = l->parent_;
                t->right_ = l->left_;
                if (l->left_)
                    l->left_->parent_ = t;

                l->left_ = p->left_;
                p->left_->parent_ = l;
                l->right_ = p->right_;
                if (p->right_)
                    p->right_->parent_ = l;
                ChangeParent(p, l);
                KeepBalance(t);
            }
        }
        p->left_ = p->right_ = NULL;
        delete p;
        return true;
    }
    /**
     * @brief 鍏冪礌璁℃暟
     */
    int Count(int elem) {
        if (!root_)
            return 0;
        NODE* l = NULL, * p = NULL, * n = NULL;
        bool found = false;
        root_->Find(elem, l, p, n, found);
```

```
        if (p)
            return p->count_;
        return 0;
    }
    /**
     * @brief 鏌ユ壘鏈€灏忓厓绱?
     */
    int FindMinElem() {
        if (!root_)
            return -1;
        NODE* p = root_;
        while (p->left_)
            p = p->left_;
        return p->elem_;
    }
    /**
     * @brief 鏌ユ壘鍓嶉┍鍏冪礌
     */
    int FindFrontElem(int elem) {
        if (!root_)
            return -1;
        NODE* l = NULL, * p = NULL, * n = NULL;
        bool found = false;

        root_->Find(elem, l, p, n, found);
        if (l)
            return l->elem_;
        return -1;
    }
private:
    /**
     * @brief 淇濇寔鏍戠殑骞宠
     * @param p 鍙  浠澶辫   鐨勮妭鐐瑰紑濮?
     */
    void KeepBalance(NODE* p) {
        while (p) {
            p->UpdateHeight();
            if (-2 >= p->Balance() || p->Balance() >= 2) {
                // 鏃嬭浆
                if (p->Balance() < 0) {
                    if (p->left_->Balance() <= 0) {
                        RotateRight(p);
                    }
                    else {
```

```
                    RotateLeft(p->left_);
                    RotateRight(p);
                }
            }
            else {
                if (p->right_->Balance() >= 0) {
                    RotateLeft(p);
                }
                else {
                    RotateRight(p->right_);
                    RotateLeft(p);

                }
            }
        }
        p = p->parent_;
    }
}
/**
 * @brief 鍙虫棆杞?
 */
void RotateRight(NODE* p) {
    NODE* t = p->left_;
    p->left_ = t->right_;
    if (t->right_)
        t->right_->parent_ = p;
    t->right_ = p;
    UpdateRelation(p, t);
}
/**
 * @brief 宸︽棆杞?
 */
void RotateLeft(NODE* p) {
    NODE* t = p->right_;
    p->right_ = t->left_;
    if (t->left_)
        t->left_->parent_ = p;
    t->left_ = p;
    UpdateRelation(p, t);
}
/**
 * @brief 鏇存柊鏃嬭浆鍚庤妭鐐瑰叧绯?
 */
void UpdateRelation(NODE* p, NODE* t) {
```

```
        ChangeParent(p, t);
        p->parent_ = t;
        p->UpdateHeight();
        t->UpdateHeight();
    }
    /**
     * @brief 鏖ㄥ曉p鐨勫鲶鏃忓另绯昏禍浜块
     */
    void ChangeParent(NODE* p, NODE* t) {
        if (t)
            t->parent_ = p->parent_;
        if (p->parent_) {
            if (p == p->parent_->left_)
                p->parent_->left_ = t;
            else
                p->parent_->right_ = t;
        }
        else
            root_ = t;
    }
    NODE* root_;
};

int main()
{
    int n;
    SORTTREE& T = *(new SORTTREE);
    std::cin >> n;
    while (n--) {
        int ope, elem = 0, last = 0;
        std::cin >> ope;
        if (ope != 4)
            std::cin >> elem;
        switch (ope) {
            case 1:
                T.Insert(elem);
                break;
            case 2:
                if (!T.Delete(elem))
                    std::cout << "None" << std::endl;
                break;
            case 3:
                std::cout << T.Count(elem) << std::endl;
                break;
```

```
            case 4:
                std::cout << T.FindMinElem() << std::endl;
                break;
            case 5:
                last = T.FindFrontElem(elem);
                if (last == -1)
                    std::cout << "None" << std::endl;
                else
                    std::cout << last << std::endl;
                break;
            default:
                break;
        }
    }
    delete& T;
    return 0;
}
```

```
12
1 32
1 12
1 85
1 12
2 0
None
1 1
1 10
1 12
3 12
3
4
1
5 12
10
5 30
12
```

1.二叉排序树

二叉排序树（Binary sort tree，BST），又称为二叉查找树，或者是一棵空树；或者是具有下列性质的二叉树：

(1)若它的左子树不为空，则左子树上所有节点的值均小于它的根节点的值；

(2)若它的右子树不为空，则右子树上所有节点的值均大于它的根节点的值；

(3)它的左、右子树也分别为二叉排序树。

二、二叉排序树操作

二叉排序树是一种动态树表。它的特点是树的结构不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的节点时再进行插入。新插入的节点一定是一个新添加的叶子结点，并且是查找不成功时查找路径上访问的最后一个节点的左孩子或右孩子节点，

二叉排序树的操作有插入、删除、查询和遍历等。

注意：二叉排序树中没有值相同的节点

(1)插入

a.插入过程比较简单，首先判断当前要插入的值是否已经存在二叉排序树中，如果已经存在，则直接返回；如果不存在，则转 b;

b.当前要插入的值不存在，则应找到适当的位置，将其插入。注意插入的新节点一定是叶子节点；

(2)删除

a.和插入一样，要删除一个给定值的节点，首先要判断这样节点是否存在，如果已经不存在，则直接返回；如果已经存在，则获取给定值节点的位置，根据不同情况进行删除、调整，转 b;

b.如果待删节点只有左子树（只有右子树），则直接将待删节点的左子树（右子树）放在待删节点的位置，并释放待删节点的内存,否则转 c

c.如果待删节点既有左子树又有右子树，此时的删除可能有点复杂，但是也比较好理解。就是在待删节点的左子树中找到值最大的那个节点，将其放到待删节点的位置。

(3)查询

查询过程比较简单，首先将关键字和根节点的关键字比较，如果相等则返回节点的位置(指针)；否则，如果小于根节点的关键字，则去左子树中继续查找；如果大于根节点的关键字，则去右子树中查找；如果找到叶子节点也没找到，则返回 NULL。

查询过程的最好情况就是上面的图中那样，节点在左右子树中分布比较均匀，此时查找的时间复杂度为 O(logn)；最坏的情况就是在建立二叉排序树时，输入的关键字序列正好是有序的，此时形成的二叉排序树是一棵单支二叉树，此时查找退化成了单链表的查找，时间的复杂度为 O(n).如下图：

（4)遍历

由上面二排序树的定义可知，左子树的所有值均小于根节点，右子树的所有值均大于根节点，而这个特点正好和二叉树的中序遍历中--左子树->根节点->右子树不谋而合，所以对二叉排序树进行中序遍历得到的正好是一个有序的

2.3 题目三

HW5：搜索和排序 > 5-3 哈希表

哈希表

描述

哈希表（hash table，散列表）是一种用于以常数平均时间执行插入、删除和查找的查找表，

其基本思想是：找到一个从关键字到查找表的地址的映射 h（称为散列函数），将关键字 key 的元素存到 h(key)所指示的存储单元中。当两个不相等的关键字被散列到同一个值时称为冲突，产生冲突的两个（或多个）关键字称为同义词，冲突处理的方法主要有：开放定址法，再哈希法，链地址法。

本题针对字符串设计哈希函数。假定有一个班级的人名名单，用汉语拼音（英文字母）表示。

要求：

首先把人名转换成整数，采用函数 h(key)=((...(key[0] * 37+key[1]) * 37+...)*37+key[n-2])*37+key[n-1]，其中 key[i]表示人名从左往右的第 i 个字母的 ascii 码值(i 从 0 计数,字符串长度为 n，1<=n<=100)。
采取除留余数法将整数映射到长度为 P 的散列表中，h(key)=h(key)%M，若 P 不是素数，则 M 是大于 P 的最小素数，并将表长 P 设置成 M。
采用平方探测法（二次探测再散列）解决冲突。(有可能找不到插入位置，当探测次数>表长时停止探测）
注意：计算 h(key)时会发生溢出，需要先取模再计算。

输入
第 1 行输入 2 个整数 N、P，分别为待插入关键字总数、散列表的长度。若 P 不是素数，则取大于 P 的最小素数作为表长。
第 2 行给出 N 个字符串，每一个字符串表示一个人名
输出
在 1 行内输出每个字符串插入到散列表中的位置，以空格分割，若探测后始终找不到插入位置，输出一个'-'。

```
/**
 * @name    p91_data.cpp
 * @brief   p91 闄忔満濡娴嬭瘯鏁版嵁鐢熸垚绋嬪簭
 * @author  寮犳晱
 * @date    2021-12-7
 */

#include <iostream>
#include <cstring>
#include <vector>
#include <string>
#include <iostream>
using namespace std;

class HASH {
public:
    static const int MULT_FACTOR = 37;
    static const int MAX_NAME_LENGTH = 100;
```

```
    HASH(int size) : size_(CalculateNextPrimeNumber(size)), list_(size_) {}
    ~HASH() = default;

    /**
     * @brief   鎻掑叆涓€涓   厓绱?
     * @return  鑻ユ彃鍏ユ垚鍔燂紝杩斿洖涓嬫爣锛涘惁鍒欒繑鍥?1
     */
    int Insert(const std::string& str) {
        int count = Hash(str);
        for (int i = 0; i <= size_; i++) {
            int index = ((count + Square((i + 1) / 2 % size_) * (i % 2 ? 1 : -1)) %
size_ + size_) % size_;
            if (list_[index] == "") {
                list_[index] = str;
                return index;
            }
        }
        return -1;
    };
    /**
     * @brief 璁＄畻绗竴涓   笉灏忎簬P鐨勭礌鏁?
     */
    static int CalculateNextPrimeNumber(const int P) {
        if (P <= 2)
            return 2;
        bool* chart = new bool[P];
        memset(chart, 0, sizeof(bool) * P);
        int count = 0;
        while (count * 2 + 3 < P || chart[count]) {
            if (!chart[count])
                for (int i = count; i < P; i += count * 2 + 3)
                    chart[i] = true;
            count++;
        }
        delete[] chart;
        if (count == P)
            return -1;
        return count * 2 + 3;
    }
    /**
     * @brief 鐢熸垚闅忔満鍚嶅瓧
     */
    static std::string RandomName(int length) {
        std::string name;
```

```
        for (int i = 0; i < length; i++) {
            if (rand() % 2)
                name += rand() % 26 + 'A';
            else
                name += rand() % 26 + 'a';
        }
        return name;
    }
private:
    /**
     * @brief 璁＄畻骞虫柟
     */
    static int Square(int n) {
        return (n * n);
    }
    /**
     * @brief 璁＄畻hash鍊?
     */
    int Hash(const std::string& str) const {
        int count = 0;
        for (int i = 0; i < str.length(); i++)
            count = (count * MULT_FACTOR + str[i]) % size_;
        return count;
    }

    int size_;
    std::vector<std::string> list_;
};

int main()
{
        int N, P;
        cin >> N >> P;

        HASH& H = *(new HASH(P));
        while (N--) {
            std::string name = HASH::RandomName(1 + rand() % HASH::MAX_NAME_LENGTH);
            cin >> name;
            int index = H.Insert(name);
            if (index == -1)
                cout << "- ";
            else
                cout << index << ' ';
        }
```

```
        cout << std::endl;
        delete& H;

    return 0;
}
```

```
4 4
e c d i
1 4 0 -
```

哈希表：
数字分析法
假设关键字集合中的每个关键字 key 都是由 s 位数字组成（k 1 , k 2 , …… , k n k_1,k_2,……,k_nk
1

,k
2

,……,k
n

），分析 key 中的全体数据，并从中提取分布均匀的若干位或他们的组合构成全体

使用举例
我们知道身份证号是有规律的，现在我们要存储一个班级学生的身份证号码，假设这个班级的学生都出生在同一个地区，同一年，那么他们的身份证的前面数位都是相同的，那么我们可以截取后面不同的几位存储，假设有 5 位不同，那么就用这五位代表地址。
H（key）=key%100000
此种方法通常用于数字位数较长的情况，必须数字存在一定规律，其必须知道数字的分布情况，比如上面的例子，我们事先知道这个班级的学生出生在同一年，同一个地区。
平方取中法
如果关键字的每一位都有某些数字重复出现频率很高的现象，可以先求关键字的平方值，通过平方扩大差异，而后取中间数位作为最终存储地址。
使用举例
比如 key=1234 1234^2=1522756 取 227 作 hash 地址
比如 key=4321 4321^2=18671041 取 671 作 hash 地址
这种方法适合事先不知道数据并且数据长度较小的情况
折叠法
如果数字的位数很多，可以将数字分割为几个部分，取他们的叠加和作为 hash 地址
使用举例
比如 key=123 456 789
我们可以存储在 61524，取末三位，存在 524 的位置
该方法适用于数字位数较多且事先不知道数据分布的情况

除留余数法用的较多
H（key）=key MOD p （p<=m m 为表长）
很明显，如何选取 p 是个关键问题。

使用举例
比如我们存储 3 6 9，那么 p 就不能取 3
因为 3 MOD 3 == 6 MOD 3 == 9 MOD 3
p 应为不大于 m 的质数或是不含 20 以下的质因子的合数,这样可以减少地址的重复(冲突)

比如 key = 7，39，18，24，33，21 时取表长 m 为 9 p 为 7 那么存储如下

2.4 题目四

HW5：搜索和排序 > 5-4 排序

Problem 1 (排序)
描述
排序算法分为简单排序（时间复杂度为 O(n^2))和高效排序（时间复杂度为 O(nlog n)）。

本题给定 N 个整数，要求输出从小到大排序后的结果。 请用不同的排序算法（快速排序，归并排序，堆排序，选择排序，冒泡排序，直接插入排序，希尔排序）分别进行测试，查看运行时间。

输入
第 1 行一个正整数 n，表示元素个数
第 2 行 n 个整数，用空格分割
输出
从小到大排序后的结果，以空格分割。
样例输入 1
13
81 68 33 40 89 72 75 21 75 3 63 15 54
样例输出 1
3 15 21 33 40 54 63 68 72 75 75 81 89

```
#include<stdio.h>
#include<stdlib.h>
#include <iostream>
#include <cstring>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

////快速排序   hoare版本(左右指针法)
//void QuickSort(int* arr, int begin, int end)
```

```
//{
//  //只有一个数或区间不存在
//  if (begin >= end)
//      return;
//  int left = begin;
//  int right = end;
//  int mid = (left + right) / 2;
//  swap(arr[left], arr[mid]);
//  //选左边为key
//  int keyi = begin;
//  while (begin < end)
//  {
//      //右边选小   等号防止和key值相等    防止顺序begin和end越界
//      while (arr[end] >= arr[keyi] && begin < end)
//      {
//          --end;
//      }
//      //左边选大
//      while (arr[begin] <= arr[keyi] && begin < end)
//      {
//          ++begin;
//      }
//      //小的换到右边，大的换到左边
//      swap(arr[begin], arr[end]);
//  }
//  swap(arr[keyi], arr[end]);
//  keyi = end;
//  //[left,keyi-1]keyi[keyi+1,right]
//  QuickSort(arr, left, keyi - 1);
//  QuickSort(arr, keyi + 1, right);
//}
int s[10000000];
int main()
{
    int n; cin >> n;
    for (int i = 0; i < n; i++)
        cin >> s[i];
    sort(s, s + n);
    for (int i = 0; i < n; i++)
        cout << s[i] << " ";
    return 0;
}
```

```
13
81 68 33 40 89 72 75 21 75 3 63 15 54
3 15 21 33 40 54 63 68 72 75 75 81 89
```

快速排序的基本思想是：通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，已达到整个序列有序。一趟快速排序的具体过程可描述为：从待排序列中任意选取一个记录(通常选取第一个记录)作为基准值，然后将记录中关键字比它小的记录都安置在它的位置之前，将记录中关键字比它大的记录都安置在它的位置之后。这样，以该基准值为分界线，将待排序列分成的两个子序列。

一趟快速排序的具体做法为：设置两个指针 low 和 high 分别指向待排序列的开始和结尾，记录下基准值 baseval(待排序列的第一个记录)，然后先从 high 所指的位置向前搜索直到找到一个小于 baseval 的记录并互相交换，接着从 low 所指向的位置向后搜索直到找到一个大于 baseval 的记录并互相交换，重复这两个步骤直到 low=high 为止。

2.5 题目五

HW5：搜索和排序 > 5-5 求逆序对数

求逆序对
描述
对于一个长度为 N 的整数序列 A，满足 i < j 且 Ai > Aj 的数对(i,j)称为整数序列 A 的一个逆序。

请求出整数序列 A 的所有逆序对个数

输入
输入包含多组测试数据，每组测试数据有两行
第一行为整数 N(1 <= N <= 20000)，当输入 0 时结束
第二行为 N 个整数，表示长为 N 的整数序列

输出
每组数据对应一行，输出逆序对的个数

```
/**
 * @name     p85_data.cpp
 * @brief    p85 閿忔湪娴嬭瘯鏁版嵁鐢熸垚绋嬪簭
 * @author   寮犳湇
```

```
 * @date    2021-12-7
 */

#include <iostream>
#include <cstring>
using namespace std;

class List {
public:
    List(int size) : size_(size) { list_ = new int[size]; key_ = new int[size]; }
    ~List() {
        if (!list_)
            delete[] list_;
        if (!key_)
            delete[] key_;
    }

    /**
     * @brief 鏊熠垚闐忆满搴忏坍
     */
    void GenerateRandomList() {
        for (int i = 0; i < size_; i++)
            list_[i] = rand() % int(size_ * 1.2) + 1;
    }
    /**
     * @brief 鏊熠垚椤哄箨搴忏坍
     */
    void GenerateOrderList() {
        for (int i = 0; i < size_; i++)
            list_[i] = i + 1;
    }
    /**
     * @brief 鏊熠垚闆喟箨搴忏坍
     */
    void GenerateDescList() {
        for (int i = 0; i < size_; i++)
            list_[i] = size_ - i + 1;
    }
    /**
     * @brief 鎵撴嵚搴忏坍
     */
    void PrintList() {
        for (int i = 0; i < size_; i++)
            cin >> list_[i];
```

```
    }
    /**
     * @brief 璁＄畻闆嗗悎簿鑶?
     */
    int Solve() {
        memcpy(key_, list_, size_ * sizeof(int));
        return MergeSort(list_, key_, 0, size_);
    }
private:
    /**
     * @brief 褰掑苟鎺掑簭
     */
    int MergeSort(int* list, int* key, int l, int r) {
        if (r - l == 1)
        {
            list[l] = key[l];
            return 0;
        }
        int m = (l + r) / 2;
        int count = MergeSort(key, list, l, m) + MergeSort(key, list, m, r);
        for (int i1 = l, i2 = m, c = l; ; ) {
            if (i1 == m) {
                while (i2 < r)
                    list[c++] = key[i2++];
                break;
            }
            if (i2 == r) {
                while (i1 < m)
                    list[c++] = key[i1++];
                break;
            }
            if (key[i1] > key[i2]) {
                count += m - i1;
                list[c++] = key[i2++];
            }
            else {
                list[c++] = key[i1++];
            }
        }
        return count;
    }

    int size_, * list_, * key_;
};
```

```
int main()
{
    int n;
    cin >> n;
    while (1) {

        List& L = *(new List(n));


        L.PrintList();
        cout << L.Solve() << std::endl;
        delete& L;
        cin >> n;
        if (n == 0)
            break;
    }

    return 0;
}
```

```
5
1 2 3 4 5
0
5
5 4 3 2 1
10
1
1
0
0
```

当时写的时候有好几个点没注意，所以 WA 了好几次，这里点出来给自己一个提醒。第一是关于归并排序，将左右两个子段合并的时候必须要新开一个 O(n)的空间来存储临时数组，否则必须要通过交换元素来达到排序的目的，但是这样会超时。第二是关于逆序数的定义一定要看清，必须要严格满足 i < j 且 Ai > Aj，也就是说 i < j 但是 Ai = Aj 时是不算逆序数的。

这个问题最简单的方法就是遍历整个序列，判断实数与它后面的每个实数是否构成逆序对，

这种算法的时间复杂度为O (n2)。
如果用分治方法求出逆序对数，这就类似与归并排序算法。先将数组从中间分成两个部分，然后分别递归左半部分和右半部分，再合并排好序的左右两个部分，从而统计逆序对数。比如合并两个排好序的数组{1,4,5,7}和{2,3}，
1. 先比较后取出 1，再比较发现 2 小于 4，那么 2 肯定小于 4 后面的元素，从而构成了(4,2),(5,2),(7,2) 3 个逆序对。
2. 取出 2 后，再比较 4 和 3,3<4,所以 3 和 4 以及后面的元素都构成了逆序对，共 3 对。
利用这个方法将数组的左半边和右半边递归分解，在合并过程中统计逆序对的个数。

3. 实验总结

内排序与外排序: 内排序是指在排序期间数据对象全部存放在内存的排序; 外排序是指在排序期间全部对象个数太多，不能同时存放在内存，必须根据排序过程的要求，不断在内、外存之间移动的排序。

排序的时间开销:
排序的时间开销是衡量算法好坏的最重要的标志。
排序的时间开销可用算法执行中的数据比较次数与数据移动次数来衡量。
通常给出的算法运行时间代价的大略估算一般都按平均情况进行估算。对于那些受对象关键字序列初始排列及对象个数影响较大的，需要按最好情况和最坏情况进行估算。

静态排序: 排序的过程是对数据对象本身进行物理的重排，经过比较和判断，将对象移到合适的位置。这时，数据对象一般都存放在一个顺序的表中。
动态排序: 给每个对象增加一个链接指针，在排序的过程中不移动对象或传送数据，仅通过修改链接指针来改变对象之间的逻辑顺序，从而达到排序的目的。
其实很简单，只需要在每个输入元素加一个 index，表示初始时的数组索引，当不稳定的算法排好序后，对于相同的元素对 index 排序即可。

基于比较的排序都是遵循“决策树模型”，而在决策树模型中，我们能证明给予比较的排序算法最坏情况下的运行时间为Ω(nlgn)，证明的思路是因为将 n 个序列构成的决策树的叶子节点个数至少有 n!，因此高度至少为 nlgn。
线性时间排序虽然能够理想情况下能在线性时间排序,但是每个排序都需要对输入数组做一些假设，比如计数排序需要输入数组数字范围为[0,k]等。
在排序算法的正确性证明中介绍了"循环不变式“，他类似于数学归纳法，"初始"对应"n=1"，"保持"对应"假设 n=k 成立，当 n=k+1 时"。
常见的快速排序、归并排序、堆排序、冒泡排序等属于比较排序。在排序的最终结果里，元素之间的次序依赖于它们之间的比较。每个数都必须和其他数进行比较，才能确定自己的位置。

在冒泡排序之类的排序中，问题规模为 n，又因为需要比较 n 次，所以平均时间复杂度为 $O(n^2)$。在归并排序、快速排序之类的排序中，问题规模通过分治法消减为 logN 次，所以平均时间复杂度为 O(nlogn)。
比较排序的优势是,适用于各种规模的数据,也不在乎数据的分布,都能进行排序。可以说,比较排序适用于一切需要排序的情况。

计数排序、基数排序、桶排序则属于非比较排序。非比较排序是通过确定每个元素之前，应该有多少个元素来排序。针对数组 arr,计算 arr[i]之前有多少个元素，则唯一确定了 arr[i]排序后数组中的位置。
非比较排序只要确定每个元素之前的已有的元素个数即可，所有一次遍历即可解决。算法时间复杂度 O(n)。
非比较排序的时间复杂度低，但由于非比较排序需要占用空间来确定唯一的位置。所以对数据规模和数据分布有一定的要求。

算法执行时所需的附加存储：评价算法好坏的另一标准。
静态排序过程中所用到的数据表的定义，体现了抽象数据类型的思想。
衡量排序方法的标准：
1、 排序时所需要的平均比较次数
2、 排序时所需要的平均移动次数
3、排序时所需要的平均辅助存储空间