《数据结构》上机报告

2020 年 12 月 23 日

姓名: 林日中 学号: 1951112 班级: 10072602 得分:

实验题目	搜索实验报告 – 折半查找, 二叉排序树, 哈希表
问题描述	**HW8_1.cpp** 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$O(\log n)$, 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 (下标最小的位置), 保证时间代价是$O(\log n)$。若查找不到, 返回-1。 **HW8_2.cpp** 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 **HW8_3.cpp** 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$h(key) = ((\ldots(key[0] * 37 + key[1]) * 37+\ldots) * 37 + key[n-2]) * 37 + key[n-1]$, 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)。 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。

<table>
<tr><td rowspan="2">基本要求</td><td colspan="2">(1) 请总结 HW8 的三个实验, 参照前面几次作业的实验报告格式, 完成查找这一章的实验报告。</td></tr>
<tr><td>已完成基本内容 (序号) :</td><td>(1)</td></tr>
<tr><td rowspan="2">选做要求</td><td colspan="2"></td></tr>
<tr><td>已完成选做内容 (序号) :</td><td>无</td></tr>
<tr><td>数据结构设计</td><td colspan="2">
在本次上机实验涵盖的 HW8 三次作业中, 我设计的数据结构和数据结构的成员变量如下。

HW8_1.cpp
• class SSTable : 构建顺序查找表的类
  ○ int length : 表示本 SSTable 对象的元素长度
  ○ SElemType *elem : 表示指向本对象元素数组的指针

HW8_2.cpp
• enum Command : 用于配合主函数的 switch-case 分支结构进行操作类型的筛选
  ○ INSERT : insert a node
  ○ DELETE : delete a node
  ○ COUNT : count the frequency of a node
  ○ MINIMUM : find the smallest node
  ○ PRECURSOR : find the precursor node of a node
• struct Element : 构建基础元素的结构体
  ○ KeyType key : 本 Element 对象的关键字
  ○ int freq : 本 Element 对象关键字的出现次数
• struct Node : 构建元素结点的结构体
  ○ TElemType data : 本 Node 对象的数据
  ○ Node *left_child : 指向本 Node 对象的左子树的指针
  ○ Node *right_child : 指向本 Node 对象的右子树的指针
  ○ Node *father : 指向本 Node 对象的父亲结点的指针
• class BSTree : 构建二叉搜索树的结构体
  ○ Node *root : 指向本 BSTree 对象的根结点的指针

HW8_3.cpp
• struct Element : 表示基础元素的结构体
  ○ KeyType key : 本 Element 对象的关键字
• class HashTable : 表示哈希表的类
  ○ int size : 本 HashTable 对象的最大容量
  ○ int count : 本 HashTable 对象的元素个数
  ○ ElemType *elem : 指向本 HashTable 对象的元素数组的指针
</td></tr>
</table>

功能(函数)说明

HW8_1.cpp

```
class SSTable
{
protected:
    int length = -1;
    SElemType *elem = nullptr;

public:
    SSTable(const int l) : length(l), elem(new SElemType[l]) …
    ~SSTable() { delete[] elem; }
    int binarySearch(const KeyType &key) …
}; // class SSTable

int main() …
```

- `SSTable::SSTable()` : 构造函数，包装了构建本 SSTable 对象的相关操作，负责从 stdin 读取数据
- `SSTable::~SSTable` : 析构函数，负责对本 SSTable 对象中动态申请的空间做 delete 操作
- `int SSTable::binarySearch()` : 包装了折半查找的相关操作，返回一个表示元素在数组中的下标的 int，若不存在则返回 -1
- `int main()` : 包装了关于 SSTable 功能的操作，方便 OJ 的检查

HW8_2.cpp

```
struct Element
{
    KeyType key;
    int freq;
    Element(const KeyType &k, const int f = 1) …
    struct Element &operator=(const struct Element &e) …
};

struct Node
{
    TElemType data;
    Node *left_child = nullptr;
    Node *right_child = nullptr;
    Node *father = nullptr;
    Node(const TElemType &e) : data(e) {}
}; // struct Node

class BSTree
{
private:
    Node *root = nullptr;

    Status delete_all(Node *&p) …

    Status search(Node *T, const KeyType &key, Node *parent, Node *&p) …

    Status insert(Node *&T, const TElemType &e) …

    Status _delete(Node *&T, const KeyType &key) …

    Status delete_elem(Node *&p) …

    KeyType mininum() …

    Status find_pre(Node *cur) …

public:
    BSTree() {}
    ~BSTree() { delete_all(root); }

    Status Search(const KeyType &key, Node *&pre) …

    Status Delete(const KeyType &key) …

    Status Count(const KeyType &key) …

    Status Minimum() …

    Status Insert(const KeyType &key) …

    Status Precursor(const KeyType &key) …

}; // class BSTree

int main() …
```

- `Element::Element()` : 构造函数，负责本 Element 对象的初始化，给成员变量赋初值
- `Element &Element::operator=()` : 对运算符 = 的重载函数，包装了对成员变量的赋值操作，令对象的赋值操作更方便
- `Node::Node()` : 构造函数，负责本 Node 对象的初始化，给成员变量赋初值
- `Status BSTree::delete_all()` : 配合析构函数对本 BSTree 对象中的所有 Node * 做 delete 操作
- `Status BSTree::insert()` : 插入一个元素
- `Status BSTree::_delete()` : 用于删除元素操作的递归函数
- `Status BSTree::delete_elem()` : 删除一个元素
- `Status BSTree::minimum()` : 求最小值
- `Status BSTree::find_pre()` : 找某关键字对应元素的前驱
- `BSTree::BSTree()` : 构造函数，默认无操作
- `BSTree::~BSTree()` : 析构函数，调 delete_all()
- `Status BSTree::Search()` : 给用户调用的搜索函数
- `Status BSTree::Delete()` : 给用户调用的删除函数
- `Status BSTree::Count()` : 给用户调用的计数函数
- `Status BSTree::Minimum()` : 给用户调用的求最小值函数
- `Status BSTree::Insert()` : 给用户调用的插入函数
- `Status BSTree::Precursor()` : 给用户调用的找前驱函数
- `int main()` : 包装了关于 BSTree 功能的相关操作，方便 OJ 的检查

HW8_3.cpp

```
int isprime(const int n)…

int nextPrime(const int cur)…

typedef struct Element ElemType;
typedef int Status;
typedef char KT, *KeyType;

struct Element
{
    KeyType key = NULLKEY;
    int length() const { return strlen(key); }
    Element() {}
    Element(const KeyType &k) : key(new KT[strlen(k) + 1]) { EVL(key, k); }
    ~Element() { delete[] key; }
    Element &operator=(const Element &e)…
}; // struct Element

class HashTable
{
private:
    int size = 0;
    int count = 0; // count of elements
    ElemType *elem = nullptr;

    int quadradic_probing_index(const int t)…

    Status collision(int &p, int &c)…

public:
    HashTable(const int N, const int P) : size(nextPrime(P)), elem(new ElemType[size])…

    ~HashTable() { delete[] elem; }

    int hash(const KeyType &key)…

    Status search(const KeyType &key, int &p, int &c)…

    Status insert(const ElemType &e, int &p)…

}; // class HashTable

int main()…
```

	• `int isprime()` : 判断一个整数是否为质数 • `int nextPrime()` : 返回一个大于等于该整数的质数 • `Element::Element()` : 构造函数，申请 `key` 所用空间，并进行赋值操作 • `Element::~Element()` : 析构函数，对 `key` 进行 `delete` 操作 • `Element &Element::operator=()` : 对运算符 `=` 的重载函数，包装了对成员变量的赋值操作，令对象的赋值操作更方便 • `int HashTable::quadradic_probing_index()` : 根据传入参数计算平方探测法当前所需偏移量 • `Status HashTable::collision()` : 处理冲突的函数 • `HashTable::HashTable()` : 构造函数，根据传入参数进行空间的动态申请，从 `stdin` 中读入关键字进行哈希表的构建 • `HashTable::~HashTable()` : 析构函数，对 `elem` 进行 `delete` 操作 • `int HashTable::hash()` : 计算某一元素关键字对应哈希值 • `Status HashTable::search()` : 查找某一元素是否存在，插入位置是否合法 • `Status HashTable::insert()` : 插入某一元素 • `int main()` : 包装了关于 `HashTable` 功能的相关操作，方便 `OJ` 的检查
开发环境	Windows 10, C++ language, Visual Studio Code with MinGW g++

调试分析	**HW8_1.cpp**
	本程序采用重定向的方式输入所需数据。本程序所需输入数据为：1 个 int 表示顺序表长度, 1个 int 的递增序列表示顺序表元素 (关键字), 1个 int 表示搜索次数, 多个 int 表示搜索关键字。我设计了 1 组含有多个重复元素的测试数据, 放入 HW8_1_input.txt 中, 其内容如下。
	`30`
	`1 1 2 2 4 4 5 8 8 8 9 9 9 10 11 11 11 11 12 12 12 12 13 13 14 14 16 17 17 18`
	`20`
	`0`
	`1`
	`2`
	`3`
	`4`
	`5`
	`6`
	`7`
	`8`
	`9`
	`10`
	`11`
	`12`
	`13`
	`14`
	`15`
	`16`
	`17`
	`18`
	`19`
	经过测试, 本程序功能完好, 能够根据要搜索的关键字, 找到关键字第一次出现的位置, 若未出现则输出-1。输出的所有文本如下:
	`-1`

```
0
2
-1
4
6
-1
-1
7
10
13
14
18
22
24
-1
26
27
29
-1
```

综上, 本程序设计的数据结构很好地完成了题目对折半查找顺序表的要求, 功能完好, 使用方便, 并且能够较好地处理非法数据的输入并反馈相关错误提示；程序界面友好, 具有比较恰当的人性化提醒, 具有一定的鲁棒性。

HW8_2.cpp

本程序采用重定向的方式输入所需数据。本程序所需输入数据为：1 个 int 表示操作个数, n 组操作。我设计了 1 组多次查找最小值, 多次查看关键字出现次数, 多次删除同一元素的测试数据, 放入 HW8_2_input.txt 中, 其内容如下。

```
30
1 4
1 6
2 7
1 4
1 4
1 3
1 8
1 9
1 4
2 4
3 4
4
5 0
5 9
2 3
1 2
1 0
4
5 1
2 3
1 3
2 9
2 8
2 4
3 4
2 4
3 4
2 4
3 4
```

经过测试, 本程序功能完好, 能够根据不同命令做出相应的反应, 在多次进行插入删除操作时不出错, 准确反馈相关信息。输出的所有文本如下:

```
None
3
3
None
8
0
0
None
2
```

```
1
0
```

综上, 本程序设计的数据结构很好地完成了题目对二叉搜索树的要求, 功能完好, 使用方便, 并且能够较好地处理非法数据的输入并反馈相关错误提示；程序界面友好, 具有比较恰当的人性化提醒, 具有一定的鲁棒性。

HW8_3.cpp

本程序采用重定向的方式输入所需数据。本程序所需输入数据为：2 个 int 分别表示待插入关键字总数和散列表长度, n 个只含有英文字母的字符串表示 n 个人名。我设计了 1 组测试数据, 放入 HW8_3_input.txt 中, 其内容如下。

```
100 100
UbpTFotlBH EDMORqcfHP ASVcwGhxDa CIgkVzQyuH USFHeyhWJI MrwYcTjNve xzVtKmycSo NrPFqnVfah tjGHdXzpOE
nitHWodlhb pFOIcdisPf dUjwrcnQiE DuxHAJwbYi ISsGvFkgfj HcoyktKreu hcJHIvuwsU DiCWHFeVZu FemBScrKQR
mIWZTrkeiL NXDmbxUZeM syMYLghScl wTfvOapYce tSVzTPXLHA vTaMfiRHXx zlOPnoSNWj bSdaYupqKy TeJxtpLIYF
RgJXkSBVeI SiGhezxWPy BYixnRhcVM yKRJVcNhIp ourmIaZKOp LPfObDhuRt gwOFPXktZm kFcuMdOfSi eWmkYDBILG
FnhcxDPlZw SXqElCJNBb oYAVrQSsGq avKZJCVBSQ gdFKVZykSz cqnXPuLbNC AmoflCEwGP PolJbFXUfS cMYTbJxsBm
zpiTkMAoIL pOqZKQHaYh xsSKVaLPgQ cNibqhLXQn xPgObLlSTR omcTGfxthp AumYjeVcxf wDarGysMch axWtrmQpqj
KxbgthjNTR IdfhSKqksX AfVncshTKL aXAExOuhnG lgtbdnURmY rqMLFYwZUu KRUBfuNGav efkWiXxISD vSBXhWsEcV
kjWQUaeuIm qzlUbHKLIT XNbwpnVvAs ahrEpclBCq GJsiLwOjPS njFgSMrQHP NlFcGoCRiS mwpsFaAxez EqkAuyFcQZ
LNFrimjIfc cNqmgZCVQT DZvNmgrSMh UmRSbfsCze fuxMGpbjaP qajpmAzTon bTiqtVhfnz ulmVbkTfnL xznHSICFZP
oQKjOJPsSp vqXbonTfrL QHouYtMXar exIdcRfZzV IhztOVjvfp PpNaqIwXtD wuNzHonOSX NEsckyQjze VfCHLxXeow
SfUAoFMweX NCyrwEGoOB KfAvbYaexn fDTENRoMJg PNHawbOZdF MInixtVoTX AlZzgEpRjP LVUxdmNrpS sldBNeFuaj
NzWFUObfAI
```

经过测试, 本程序功能完好, 能够调用 hash() 函数计算出关键字对应哈希值, 并在找不到插入位置时准确反馈相关信息。输出的所有文本如下:

```
12 48 13 40 42 3 65 41 5 93 24 6 22 20 82 83 59 35 62 25 33 14 81 26 9 21 39 43 80 71 7 68 91 55 15 16
89 75 85 10 45 86 23 92 70 53 57 37 79 46 69 90 67 8 44 47 27 38 56 4 78 95 49 94 17 72 77 36 2 54 100
34 87 88 76 28 66 29 18 98 74 31 97 63 30 19 1 96 32 61 73 51 50 58 0 52 99 60 - 64
```

综上, 本程序设计的数据结构很好地完成了题目对哈希表的要求, 功能完好, 使用方便, 并且能够较好地处理非法数据的输入并反馈相关错误提示；程序界面友好, 具有比较恰当的人性化提醒, 具有一定的鲁棒性。

心得体会

一、 实验总结

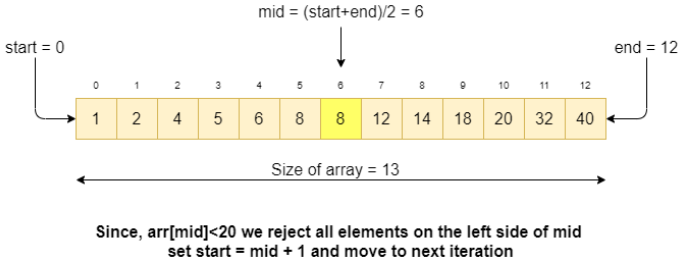
在本次上机实验中, 我复习了理论课上学到的顺序查找表, 二叉搜索树, 哈希表的结构和相关的查找操作, 并封装成了 class SSTree, class BSTree 和 class HashTable, 完成了本章作业, 并在 OJ 上全部 AC。

二、 关键算法性能分析

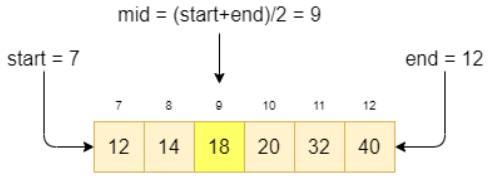
（1） 折半查找

时间复杂度: $O(\log n)$。

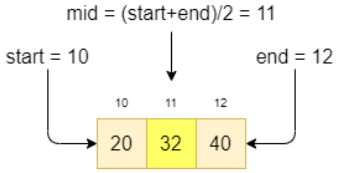
假设需要搜索的关键字为 20, 搜索过程如下。



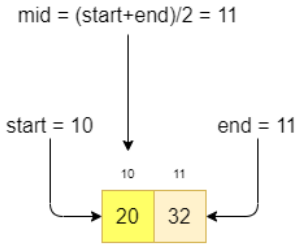
Since, arr[mid]<20 we reject all elements on the left side of mid
set start = mid + 1 and move to next iteration



Since, arr[mid]<20 we reject all elements on left side of mid
set start = mid + 1 and move to next iteration

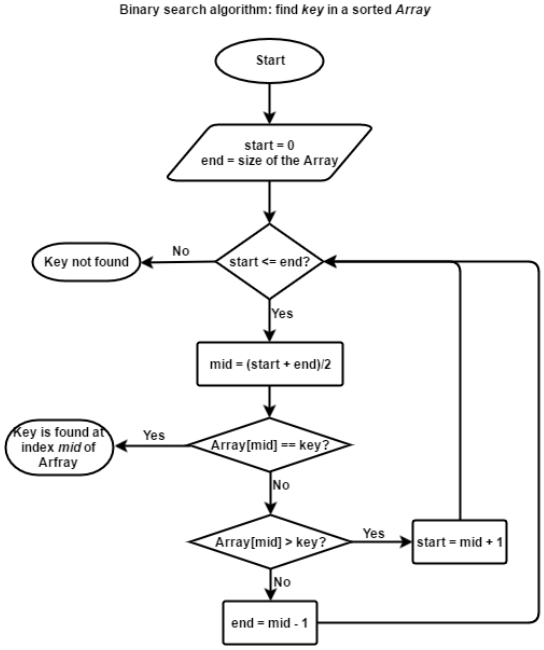


Since, arr[mid]>20 we reject all elements on the right side of mid
set end = mid - 1 and move to next iteration

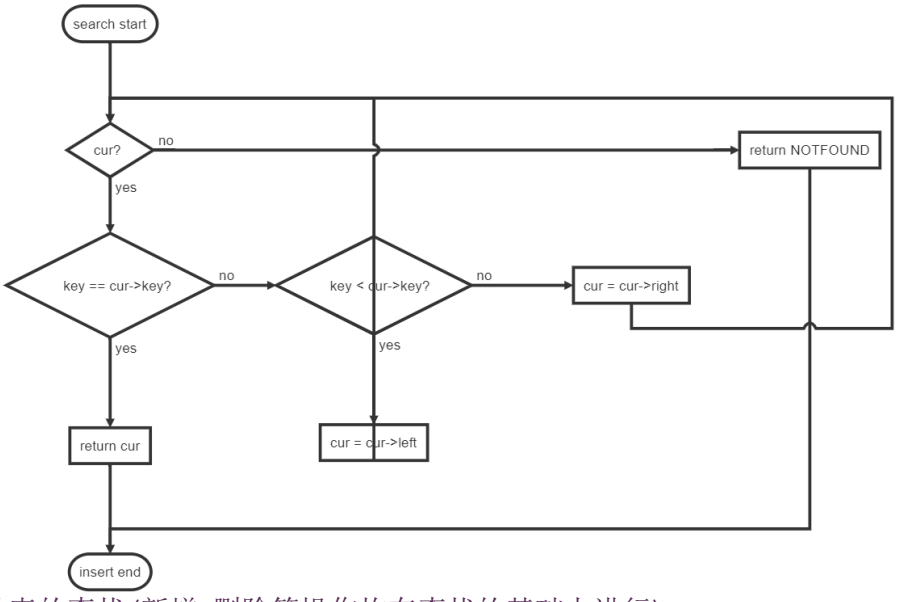


arr[mid]==20, that means we have found the element. After that, we return mid
and terminate binary search function

程序框图如下。



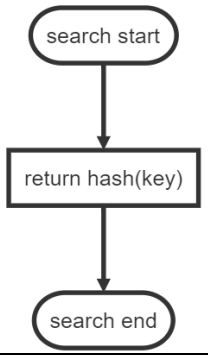
（2） 二叉搜索树的查找 (插入, 删除, 计数, 找前驱等操作均在查找的基础上进行)
时间复杂度: 完全平衡: $O(\log_2 n)$, 完全不平衡 (单支): $O(n)$。
程序框图如下。



（3） 哈希表的查找 (新增, 删除等操作均在查找的基础上进行)

时间复杂度: $O(1)$。

程序框图如下。



源代码

HW8_1.cpp

```
#if defined(__GNUC__)
#include <bits/stdc++.h>
#elif defined(_MSC_VER)
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <climits>
#include <ctime>
#include <iomanip>
#include <queue>
#endif

#define ERROR 0
#define OK 1
#define LOVERFLOW -1
```

```
18.
19. typedef int Status;
20. typedef int KeyType;
21. typedef int SElemType;
22.
23. using namespace std;
24.
25. class SSTable
26. {
27. protected:
28.     int length = -1;
29.     SElemType *elem = nullptr;
30.
31. public:
32.     SSTable(const int l) : length(l), elem(new SElemType[l])
33.     {
34.         for (int i = 0; i < length; i++)
35.         {
36.             scanf("%d", &elem[i]);
37.         }
38.     }
39.     ~SSTable() { delete[] elem; }
40.     int binarySearch(const KeyType &key)
41.     {
42.         int low = 0, high = length - 1, mid;
43.         while (low < high)
44.         {
45.             mid = ((high + low) >> 1);
46.             if (key == elem[mid])
47.             {
48.                 high = mid;
49.             }
50.             else if (key > elem[mid])
51.             {
52.                 low = mid + 1;
53.             }
54.             else // if (key < elem[mid])
55.             {
56.                 high = mid - 1;
57.             }
58.         }
59.         return key == elem[high] ? high : LOVERFLOW;
60.     }
61. }; // class SSTable
```

```
62.
63. int main()
64. {
65.     int size;
66.     KeyType key;
67.     scanf("%d", &size);
68.
69.     SSTable t(size);
70.     scanf("%d", &size);
71.     while (size--)
72.     {
73.         scanf("%d", &key);
74.         printf("%d\n", t.binarySearch(key));
75.     }
76.     return 0;
77. }
```

HW8_2.cpp

```
1.  #if defined(__GNUC__)
2.  #include <bits/stdc++.h>
3.  #elif defined(_MSC_VER)
4.  #define _CRT_SECURE_NO_WARNINGS
5.  #include <iostream>
6.  #include <cstdio>
7.  #include <cstdlib>
8.  #include <cstring>
9.  #include <climits>
10. #include <ctime>
11. #include <iomanip>
12. #endif
13.
14. #define ERROR 0
15. #define OK 1
16. #define FALSE 0
17. #define TRUE 1
18. #define LOVERFLOW -1
19.
20. typedef int Status;
21. typedef int KeyType;
22. typedef int ValueType;
23. typedef struct Element TElemType;
24.
25. using namespace std;
```

```
26.
27. enum Command
28. {
29.     INSERT = 1, // insert a node
30.     DELETE,     // delete a node
31.     COUNT,      // count the frequency of a node
32.     MINIMUM,    // find the smallest node
33.     PRECURSOR   // find the precursor node of a node
34. };
35.
36. struct Element
37. {
38.     KeyType key;
39.     int freq;
40.     Element(const KeyType &k, const int f = 1)
41.         : key(k), freq(f) {}
42.     struct Element &operator=(const struct Element &e)
43.     {
44.         key = e.key;
45.         freq = e.freq;
46.         return *this;
47.     }
48. };
49.
50. struct Node
51. {
52.     TElemType data;
53.     Node *left_child = nullptr;
54.     Node *right_child = nullptr;
55.     Node *father = nullptr;
56.     Node(const TElemType &e) : data(e) {}
57. }; // struct Node
58.
59. class BSTree
60. {
61. private:
62.     Node *root = nullptr;
63.
64.     Status delete_all(Node *&p)
65.     {
66.         if (p)
67.         {
68.             delete_all(p->left_child);
69.             delete_all(p->right_child);
```

```
70.             delete p;
71.             p = nullptr;
72.         }
73.         return OK;
74.     }
75.
76.     Status search(Node *T, const KeyType &key, Node *parent, Node *&p)
77.     {
78.         if (!T)
79.         {
80.             p = parent;
81.             return FALSE;
82.         }
83.         if (key == T->data.key)
84.         {
85.             p = T;
86.             return TRUE;
87.         }
88.         else if (key < T->data.key)
89.         {
90.             return search(T->left_child, key, T, p);
91.         }
92.         else // if (key > T->data.key)
93.         {
94.             return search(T->right_child, key, T, p);
95.         }
96.     }
97.
98.     Status insert(Node *&T, const TElemType &e)
99.     {
100.         Node *p = nullptr;
101.         if (!search(T, e.key, nullptr, p))
102.         {
103.             Node *s = new Node(e);
104.             s->father = p;
105.             if (!p)
106.             {
107.                 T = s; // new Node becomes root node
108.             }
109.             else if (e.key < p->data.key)
110.             {
111.                 p->left_child = s; // new Node becomes left child of p (p has no
    left child)
112.             }
```

```
113.             else // if (e.key > p->data.key)
114.             {
115.                 p->right_child = s; // new Node becomes right child of p (p has
    no right child)
116.             }
117.             return TRUE;
118.         }
119.         else
120.         {
121.             p->data.freq++;
122.             return TRUE;
123.         }
124.     }
125.
126.     Status _delete(Node *&T, const KeyType &key)
127.     {
128.         if (!T)
129.         {
130.             cout << "None" << endl;
131.             return FALSE;
132.         }
133.         else
134.         {
135.             if (key == T->data.key)
136.             {
137.                 return delete_elem(T);
138.             }
139.             else if (key < T->data.key)
140.             {
141.                 return _delete(T->left_child, key);
142.             }
143.             else // if (key > T->data.key)
144.             {
145.                 return _delete(T->right_child, key);
146.             }
147.         }
148.     }
149.
150.     Status delete_elem(Node *&p)
151.     {
152.         Node *q = nullptr, *s = nullptr;
153.         if (p->data.freq > 1)
154.         {
155.             p->data.freq--;
```

```
156.         }
157.         else
158.         {
159.             if (!p->right_child) // right is null (whereas left may be null)
160.             {
161.                 q = p;
162.                 p = p->left_child;
163.                 if (p)
164.                 {
165.                     p->father = q->father;
166.                 }
167.                 delete q;
168.                 q = nullptr;
169.             }
170.             else if (!p->left_child) // left is null (right cannot be null)
171.             {
172.                 q = p;
173.                 p = p->right_child;
174.                 p->father = q->father;
175.                 delete q;
176.                 q = nullptr;
177.             }
178.             else // neither right nor left is null
179.             {
180.                 s = p->left_child;
181.                 while (s->right_child)
182.                 { // s is the direct precursors of the in-order traversal,
    q is parent of s
183.                     s = s->right_child;
184.                 }
185.                 p->data = s->data; // s replaces p, former children of p become
    children of s
186.
187.                 q = s->father;
188.                 if (q != p)
189.                 {
190.                     q->right_child = s->left_child;
191.                 }
192.                 else
193.                 {
194.                     q->left_child = s->left_child;
195.                 }
196.
197.                 if (s->left_child)
```

```
                {
                    s->left_child->father = q;
                }
                delete s;
            }
        }
        return TRUE;
    }

    KeyType mininum()
    {
        Node *p = root;
        if (!p)
        {
            return LOVERFLOW;
        }
        while (p->left_child)
        {
            p = p->left_child;
        }
        return p->data.key;
    }

    Status find_pre(Node *cur)
    {
        if (!cur)
        {
            return LOVERFLOW;
        }
        Node *ret = nullptr;
        if (cur->left_child) // cur has a left child
        {
            ret = cur->left_child;
            while (ret->right_child)
            {
                ret = ret->right_child;
            }
        }
        else // cur has no left child
        {
            // 2 possibilities:
            //   1) cur is a right child, whose precursor is the father
            //   2) cur is a left child, then we should search for its "lowest"
            // father, while the father has a right child
```

```
242.             ret = cur->father;
243.             while (ret && cur == ret->left_child)
244.             {
245.                 cur = ret;
246.                 ret = ret->father;
247.             }
248.         }
249.         if (ret)
250.         {
251.             cout << ret->data.key << endl;
252.         }
253.         else
254.         {
255.             cout << "None" << endl;
256.         }
257.     }
258.
259. public:
260.     BSTree() {}
261.     ~BSTree() { delete_all(root); }
262.
263.     Status Search(const KeyType &key, Node *&pre)
264.     {
265.         Node *p = root;
266.         return search(p, key, nullptr, pre);
267.     }
268.
269.     Status Delete(const KeyType &key)
270.     {
271.         return _delete(root, key);
272.     }
273.
274.     Status Count(const KeyType &key)
275.     {
276.         Node *p = nullptr;
277.         Search(key, p);
278.         cout << (p && key == p->data.key ? p->data.freq : 0) << endl;
279.         return OK;
280.     }
281.
282.     Status Minimum()
283.     {
284.         cout << mininum() << endl;
285.         return OK;
```

```
286.     }
287.
288.     Status Insert(const KeyType &key)
289.     {
290.         // TElemType t(key);
291.         return insert(root, key);
292.     }
293.
294.     Status Precursor(const KeyType &key)
295.     {
296.         Node *ret = nullptr, *cur = root;
297.         // Search(key, cur);
298.
299.         if (!cur)
300.         {
301.             cout << "None" << endl;
302.             return ERROR;
303.         }
304.         while (cur)
305.         {
306.             if (key == cur->data.key)
307.             {
308.                 find_pre(cur);
309.                 return OK;
310.             }
311.             if (key < cur->data.key)
312.             {
313.                 ret = cur;
314.                 cur = cur->left_child;
315.             }
316.             else
317.             {
318.                 ret = cur;
319.                 cur = cur->right_child;
320.             }
321.         }
322.
323.         if (ret)
324.         {
325.             if (key < ret->data.key)
326.             {
327.                 find_pre(ret);
328.             }
329.             else
```

```
            {
                cout << ret->data.key << endl;
            }
        }
        else
        {
            cout << "None" << endl;
        }

        return OK;
    }

}; // class BSTree

int main()
{
    int n = 0;
    cin >> n;

    BSTree tree;

    while (n--)
    {
        int op, key;
        cin >> op;
        switch (op)
        {
        case INSERT:
            cin >> key;
            tree.Insert(key);
            break;

        case DELETE:
            cin >> key;
            tree.Delete(key);
            break;

        case COUNT:
            cin >> key;
            tree.Count(key);
            break;

        case MINIMUM:
            tree.Minimum();
```

```
374.             break;
375.
376.         case PRECURSOR:
377.             cin >> key;
378.             tree.Precursor(key);
379.             break;
380.         }
381.     }
382.
383.     return 0;
384. }
```

HW8_3.cpp

```
1.  #if defined(__GNUC__)
2.  #include <bits/stdc++.h>
3.  #elif defined(_MSC_VER)
4.  #define _CRT_SECURE_NO_WARNINGS
5.  #include <iostream>
6.  #include <cstdio>
7.  #include <cstdlib>
8.  #include <cstring>
9.  #include <climits>
10. #include <ctime>
11. #include <iomanip>
12. #include <queue>
13. #endif
14.
15. #define EQ(a, b) !strcmp(a, b)
16. #define LEN(key) strlen(key)
17. #define EVL(dst, src) strcpy(dst, src)
18. #define NULLKEY nullptr
19. #define OK 1
20. #define ERROR 0
21. #define TRUE 1
22. #define FALSE 0
23. #define LOVERFLOW -1
24.
25. using namespace std;
26.
27. int isprime(const int n)
28. {
29.     if (n <= 1)
30.     {
```

```
31.         return 0;
32.     }
33.     int threshold = (int)sqrt(n);
34.     for (int i = 2; i <= threshold; i++)
35.     {
36.         if (!(n % i))
37.         {
38.             return 0;
39.         }
40.     }
41.     return 1;
42. }
43.
44. int nextPrime(const int cur)
45. {
46.     for (int i = cur; true; i++)
47.     {
48.         if (isprime(i))
49.         {
50.             return i;
51.         }
52.     }
53.     return 0;
54. }
55.
56. typedef struct Element ElemType;
57. typedef int Status;
58. typedef char KT, *KeyType;
59.
60. struct Element
61. {
62.     KeyType key = NULLKEY;
63.     int length() const { return strlen(key); }
64.     Element() {}
65.     Element(const KeyType &k) : key(new KT[strlen(k) + 1]) { EVL(key, k); }
66.     ~Element() { delete[] key; }
67.     Element &operator=(const Element &e)
68.     {
69.         if (key)
70.         {
71.             delete[] key;
72.             key = nullptr;
73.         }
74.         key = new KT[LEN(e.key) + 1];
```

```
75.         EVL(key, e.key);
76.         return *this;
77.     }
78. }; // struct Element
79.
80. class HashTable
81. {
82. private:
83.     int size = 0;
84.     int count = 0; // count of elements
85.     ElemType *elem = nullptr;
86.
87.     int quadradic_probing_index(const int t)
88.     {
89.         return (t % 2) ? (int)pow((t + 1) / 2, 2) : -(int)pow(t / 2, 2);
90.     }
91.
92.     Status collision(int &p, int &c)
93.     {
94.         p += quadradic_probing_index(++c);
95.         p %= size;
96.         if (p < 0)
97.         {
98.             p += size;
99.         }
100.          return OK;
101.     }
102.
103. public:
104.     HashTable(const int N, const int P) : size(nextPrime(P)), elem(new ElemType[
    size])
105.     {
106.         for (int i = 0; i < N; i++)
107.         {
108.             int p;
109.             KT key[1000] = {'\0'};
110.             cin >> key;
111.             if (insert(key, p))
112.             {
113.                 if (EQ(elem[p].key, key))
114.                 {
115.                     cout << p << " ";
116.                 }
117.                 else
```

```
                {
                    cout << "- ";
                }
            }
            else
            {
                cout << "- ";
            }
        }
    }

    int hash(const KeyType &key)
    {
        int len = LEN(key), h = key[0];
        for (int i = 1; i < len; i++)
        {
            h *= 37;
            h %= size;
            h += key[i];
            h %= size;
        }
        h %= size;
        if (h < 0)
        {
            h += size;
        }

        return h;
    }

    Status search(const KeyType &key, int &p, int &c)
    {
        int q = p = hash(key);
        while (elem[p].key != NULLKEY) // && !EQ(key, elem[p].key))
        {
            p = q;
            if (c == size)
            {
                return LOVERFLOW;
            }
            collision(p, c);
        }
        if (p >= 0)
        {
```

```
162.             return elem[p].key == NULLKEY ? FALSE : TRUE;
163.         }
164.         else
165.         {
166.             return LOVERFLOW;
167.         }
168.     }
169.
170.     Status insert(const ElemType &e, int &p)
171.     {
172.         int c = 0, v = 0;
173.         if (TRUE == (v = search(e.key, p, c)))
174.         {
175.             elem[p] = e;
176.             count++;
177.             return LOVERFLOW; // existed
178.         }
179.         else if (LOVERFLOW == v)
180.         {
181.             return ERROR;
182.         }
183.         else // if (c < size)
184.         {
185.             elem[p] = e;
186.             count++;
187.             return OK;
188.         }
189.     }
190.
191. }; // class HashTable
192.
193. int main()
194. {
195.     int a, b;
196.     cin >> a >> b;
197.     HashTable ht(a, b);
198.     return 0;
199. }
```