数据结构实验报告

学号-姓名	桑龙龙-20030540015	实验时间	2020 年 12 月 12 日
诚信声明	本实验及实验报告所写内容为本人所作 桑龙龙		
实验题目	实验五 查找方法 题目一 顺序查找、折半查找 题目二 二叉排序树的建立、查找、插入和删除运算 题目三 哈希表的设计和应用		
实验过程中遇到的主要问题	无		
实验小结	本次试验进行了二分查找、二叉排序树、哈希表的设计，在进行二叉排序树的设计时，实践了书中讲的二叉平衡树，了解为什么要进行 LL, RR, LR, RL 旋转。在哈希表的设计时，考虑到一般的哈希表都是要有遍历操作的，实现了哈希表的迭代器。		
数据结构（自定义数据类型）	二叉树类似的节点		
主要算法（或算法说明）			

```
1.  //1、二分查找
2.  #include <stdio.h>
3.  #include <iostream>
4.  #include <stdlib.h>
5.  using namespace std;
6.  namespace recursion{
7.      template <typename T>
8.      T* binary_search(T* begin,T* end,T tar,bool(*cmp)(const T&
    ,const T&)){
9.          //递归二分查找
10.         //功能:对以 cmp 为排序方式的数组数组下标范围[begin,end)内二
    分查找 tar
11.         //如果查找到，返回一个 T*类型地址，如果未查找到，返回 NULL
12.         if(begin==end) return NULL;
13.         T* mid=begin+(end-begin)/2;
14.         if(*mid==tar) return mid;
15.         else if(cmp(tar,*mid)) return recursion::binary_search
    (begin,mid,tar,cmp);
16.         else return recursion::binary_search(mid+1,end,tar,cmp
    );
17.     }
18. };
```

```
namespace non_recursion{
    template <typename T>
    T* binary_search(T* begin,T* end,T tar,bool(*cmp)(const T&a,const T&b)){
        //非递归二分查找
        T* mid;
        while(begin<end){
            mid=(end-begin)/2+begin;
            if(*mid==tar) return mid;
            else if(cmp(tar,*mid)) end=mid;
            else begin=mid+1;
        }
        return NULL;
    }
};
bool cmp1(const int&a,const int&b){
    return a<b;
}
bool cmp2(const int& a,const int&b){
    return a>b;
}
#define N 100
int main(){
    int arr[N];
    for(int i=0;i<N;i++) arr[i]=i;
    int target=20;
    int* pos1=recursion::binary_search(arr,arr+N,target,cmp1);

    int* pos2=non_recursion::binary_search(arr,arr+N,target,cmp1);
    printf("%d %d\n",pos1?pos1-arr:-1,pos2?pos2-arr:-1);
}
```

```
1.  //2、二叉平衡树
2.  #include <stdio.h>
3.  #include <iostream>
4.  #include <stdlib.h>
5.  #include <time.h>
6.  #include <unordered_set>
7.  #include <vector>
8.  using namespace std;
9.
10. template <class T>
11. class AVL{
12.     private:
13.         struct node{
14.             T val;
15.             node* left;
16.             node* right;
17.             int height;
18.             node(){};
19.             node(T v):val(v),height(1),left(NULL),right(NULL){
    }
20.         };
21.         node* root;
22.         bool (*cmp)(T&,T&);//排序比较函数
23.         bool (*eq)(T&,T&);//相等比较函数
24.         int capacity;
25.         bool isdel;
26.         /*
27.         假如我们遍历到一个节点 root，出现了第一次不平衡
28.         本质：本质 height(root->left)-height(root->right)>=2
29.         1、LL 类型不平衡：
30.             height(root->left->left)>=height(root->left->right
    );
31.         2、LR 类型不平衡
32.             height(root->left->right)>=height(root->left->left
    );
33.         3、RR 类型不平衡
34.             height(root->right->right)>=height(root->right->le
    ft);
35.         4、RL 类型不平衡
36.             height(root->right->left)>=height(root->right->rig
    ht);
37.         */
38.         void LL(node* &root){
39.             node* temp=root->left;
```

```
40.             root->left=temp->right;
41.             temp->right=root;
42.             root->height=max(height(root->left),height(root->r
    ight))+1;
43.             root=temp;
44.             root->height=max(height(root->left),height(root->r
    ight))+1;
45.         }
46.         void RR(node*& root){
47.             node* temp;
48.             temp=root->right;
49.             root->right=temp->left;
50.             temp->left=root;
51.             root->height=max(height(root->left),height(root->r
    ight))+1;
52.             root=temp;
53.             root->height=max(height(root->left),height(root->r
    ight))+1;
54.         }
55.         void LR(node*& root){
56.             RR(root->left);
57.             LL(root);
58.         }
59.         void RL(node*& root){
60.             LL(root->right);
61.             RR(root);
62.         }
63.         int height(node*& root){
64.             if(!root) return 0;
65.             else return root->height;
66.         }
67.         void adjust(node*& root){
68.             //调整 root 节点
69.             int dev=height(root->left)-height(root->right);
70.             if(abs(dev)!=2) return;
71.             if(dev==2){
72.                 if(height(root->left->left)>=height(root->left
    ->right)) LL(root);
73.                 else LR(root);
74.             }else{
75.                 if(height(root->right->right)>=height(root->ri
    ght->left)) RR(root);
76.                 else RL(root);
77.             }
78.         }
```

```
79.         void inner_insert(node* &root,T& val){
80.             //向 root 中插入 val 这个值
81.             if(!root){
82.                 //插入查询到空节点，表明原来的 avl 树中没有插入值
    val
83.                 root=new node(val);
84.                 return;
85.             }
86.             if(val==root->val){
87.                 //存在该节点取消插入
88.                 capacity--;
89.                 return;
90.             }else if(cmp(val,root->val)){
91.                 //插到左子树
92.                 inner_insert(root->left,val);
93.                 if(abs(height(root->left)-height(root->right))
    ==2){
94.                     //不平衡
95.                     /*
96.                     方法一进行 LL，LR 调整
97.                     if(cmp(val,root->left->val)){
98.                         //插入到 root 的左子树的左子树导致不平衡
99.                         //进行 LL 调整
100.                          LL(root);
101.                     }else{
102.                         //插入到 root 的左子树的右子树导致不平衡
103.                         //进行 LR 调整
104.                         //先对 root->left 进行 RR 调整
105.                         //再对 root 进行 LL 调整
106.                         LR(root);
107.                     }
108.                     */
109.                     //方法二
110.                     adjust(root);
111.                 }
112.             }else{
113.                 //插到右子树
114.                 inner_insert(root->right,val);
115.                 if(abs(height(root->right)-height(root->left)
    )==2){
116.                     //不平衡
117.                     /*
118.                     方法一进行 LL，LR 调整
119.                     if(cmp(root->right->val,val)){
120.                         //插入到 root 的右子树的右子树导致不平衡
```

```
                        //进行 RR 调整
                        RR(root);
                    }else{
                        //插入到 root 的右子树的左子树导致不平衡
                        //进行 RL 调整
                        //先对 root->right 进行 LL 调整
                        //再对 root 进行 RR 调整
                        RL(root);
                    }
                    */

                    //方法二
                    adjust(root);
                }

            }
            root->height=max(height(root->left),height(root->right))+1;
            //更新 root 的高度
        }
        bool inner_find(node* root,T& val){
            //查询 root 为根的树中是否包含值为 val 的节点
            if(!root) return false;
            if(root->val==val) return true;
            else if(cmp(val,root->val)) return inner_find(root->left,val);
            else return inner_find(root->right,val);
        }
        void inner_mid_traverse(node* root,vector<T>& ans,int &cnt){
            if(!root) return;
            inner_mid_traverse(root->left,ans,cnt);
            ans[cnt++]=root->val;
            inner_mid_traverse(root->right,ans,cnt);
        }
        node* find_max(node*& root){
            //找到以 root 为跟子树中的最大值的节点
            if(!root || !root->right){
                return root;
            }else{
                return find_max(root->right);
            }
        }
        node* find_min(node*& root){
            //找到以 root 为跟子树中的最小值的节点
```

```
163.             if(!root || !root->left){
164.                 return root;
165.             }else{
166.                 return find_min(root->left);
167.             }
168.         }
169.         void inner_erase(node*& root,T& val){
170.             //在 root 中查找删除 val
171.             if(!root) return;
172.             if(root->val==val){
173.                 //查找到了要删除的值
174.                 isdel=true;
175.                 if(root->left && root->right){
176.                     //左右子树均为非空
177.                     //将 root->val 的值与 root->left 为根的子树中
    的最大值进行替换
178.                     //然后再递归进行删除
179.                     //这样就能保证我们最后删除的值一定在叶子结点
    上
180.                     node*temp=find_max(root->left);
181.                     swap(temp->val,root->val);
182.                     inner_erase(root->left,val);
183.                 }else{
184.                     //左右子树至少有一个为空树
185.                     node* temp=root;
186.                     root=root->left?root->left:root->right;
187.                     delete temp;
188.                 }
189.             }else if(cmp(val,root->val)){
190.                 inner_erase(root->left,val);
191.             }else{
192.                 inner_erase(root->right,val);
193.             }
194.             if(root){
195.                 adjust(root);
196.                 root->height=max(height(root->left),height(ro
    ot->right))+1;
197.             }
198.         }
199.         int is_avl_tree(node*root,bool &good){
200.             //判断是否是一颗高度平衡的树
201.             if(!root) return 0;
202.             int left=0,right=0;
203.             if(good) left=is_avl_tree(root->left);
204.             if(good) right=is_avl_tree(root->right);
```

```
205.             if(abs(left-right)>1) good=false;
206.             return max(left,right)+1;
207.         }
208.     public:
209.         AVL(bool (*cmp)(T&,T&), bool (*eq)(T&,T&)){
210.             //构造函数 2
211.             root=NULL;
212.             this->eq=eq;
213.             this->cmp=cmp;
214.             capacity=0;
215.         }
216.         AVL(bool (*cmp)(T&,T&)){
217.             //构造函数 2
218.             root=NULL;
219.             this->cmp=cmp;
220.             capacity=0;
221.         }
222.         void insert(T val){
223.             //插入 val 这个值
224.             capacity++;
225.             inner_insert(root,val);
226.         }
227.         void erase(T val){
228.             //删除 val 这个节点
229.             isdel=false;
230.             inner_erase(root,val);
231.             if(isdel) capacity--;
232.         }
233.         bool find(T val){
234.             //查询是否有 val 这个节点
235.             return inner_find(root,val);
236.         }
237.         int size(){
238.             //返回 avl 树中的节点数
239.             return capacity;
240.         }
241.         int height(){
242.             //返回 avl 树的高度
243.             return height(root);
244.         }
245.         void mid_traverse(vector<int>& ans){
246.             //返回中序遍历
247.             ans.resize(capacity);
248.             int cnt=0;
249.             inner_mid_traverse(root,ans,cnt);
```

```
250.         }
251.         bool is_avl_tree(){
252.             //是否是一颗高度平衡的树
253.             bool good=true;
254.             is_avl_tree(root,good);
255.             return good;
256.         }
257. };
258.
259.
260.
261. bool cmp(int& a,int &b){
262.     return a<b;
263. }
264. bool is_ok(vector<int>& num){
265.     //检查中序遍历数组是否为 cmp 规定的比较规则
266.     for(int i=1,n=num.size();i<n;i++){
267.         if(!cmp(num[i-1],num[i])) return false;
268.     }
269.     return true;
270. }
271. void test1(){
272.     //该例子只简介 int 类型的用法
273.     AVL<int> avl(cmp);
274.     srand(time(0));
275.     int n=100000,dev=10000;
276.     unordered_set<int> have;
277.     int del_cnt=0;
278.     for(int i=0;i<n;i++){
279.         int v=rand()%dev;
280.         have.insert(v);
281.         avl.insert(v);
282.         v=rand()%dev;
283.         avl.erase(v);
284.         if(have.find(v)!=have.end()) have.erase(v),del_cnt++;

285.     }
286.     vector<int> ans;
287.     avl.mid_traverse(ans);
288.     cout<<"###after manipulate###"<<endl;
289.     cout<<"avl.size() is: "<<avl.size()<<" avl.height()is: "<
    <avl.height()<<" unordered_set<int>'s size() is: "<<have.size(
    )<<endl;
290.     cout<<"In this test totally delete "<<del_cnt<<" times"<<
    endl;
```

```
    if(is_ok(ans)) cout<<"it is a avl(bst) tree"<<endl;
    else cout<<"it is not a avl(bst) tree"<<endl;
}

void introduction(){
    /*
        用法简介
        构造一颗中序遍历序列为 cmp 函数规定的比较规则的 AVL
        1、初始化该类型，必须传入一个比较函数
        2、你的类型 T 为自定义类型，你必须还要重载运算符==

        AVL<T> avl(cmp);
        bool cmp(T& a,T&b){
            //write your compare rule//
        }
        可用函数
        avl.size() 大小
        avl.height() 树高
        vector<int> ans=avl.mid_traverse();
        avl.find(v) 查询是否有 v 这个值
        avl.insert(v) 插入 v
        avl.erase(v) 删除 v
    */
}
int main(){
    test1();
    return 0;
}
```

```cpp
//3、hash 表
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
using namespace std;
template <class T,class V>
class myhash{
    public:
        struct node{
            T first;
            V second;
            node* next;
            node(T& a,V& b):first(a),second(b),next(NULL){}
        };
    private:
        int(*hash_mapping)(T&);
        int n,cap;
        node** head;
        pair<node*,node*> find_inner(T& key){
            int index=hash_mapping(key)%n;
            node* cur=head[index];
            node* last=NULL;
            bool vis=false;
            while(cur && !vis){
                if(cur->first==key){
                    vis=true;
                    break;
                }
                last=cur;
                cur=cur->next;
            }
            if(vis) return {last,cur};
            else return {NULL,NULL};
        }
        void initialize(){
            cap=0;
            head=new node*[n];
            for(int i=0;i<n;i++) head[i]=NULL;
        }
    public:

        myhash(int(*hash_mapping)(T&)){
            n=10000;
            initialize();
```

```
45.             this->hash_mapping=hash_mapping;
46.         }
47.         myhash(int(*hash_mapping)(T&),int cap){
48.             n=cap;
49.             initialize();
50.             this->hash_mapping=hash_mapping;
51.         }
52.         void insert(T key,V val){
53.             int index=hash_mapping(key)%n;
54.             node* cur=head[index];
55.             bool vis=false;
56.             while(cur && !vis){
57.                 if(cur->first==key){
58.                     vis=true;
59.                     break;
60.                 }
61.                 cur=cur->next;
62.             }
63.             if(!vis){
64.                 node* temp=new node(key,val);
65.                 temp->next=head[index];
66.                 head[index]=temp;
67.                 cap++;//大小加一
68.             }else{
69.                 cur->second=val;
70.             }
71.         }
72.         void erase(T key){
73.             int index=hash_mapping(key)%n;
74.             pair<node*,node*> temp=find_inner(key);
75.             if(temp.second && temp.first){
76.                 temp.first=temp.second->next;
77.                 delete temp.second;
78.                 cap--;//大小减 1
79.             }else if(temp.second){
80.                 head[index]=temp.second->next;
81.                 delete temp.second;
82.                 cap--;//大小减 1
83.             }
84.         }
85.         node* find(T key){
86.             return find_inner(key).second;
87.         }
88.         V& operator[](T key){
89.             V val;
```

```
90.             if(find(key)==NULL) insert(key,val);
91.             return find(key)->second;
92.         }
93.         int size(){
94.             return cap;
95.         }
96.         node* end(){
97.             return NULL;
98.         }
99.         pair<node**,int> begin(){
100.             return {head,n};
101.          }
102.
103.         public:
104.             class iterator{
105.                 public:
106.                     node** arr;
107.                     node* ptr;
108.                     int n,cur;
109.                     iterator();
110.                     bool operator==(iterator& a);
111.                     bool operator!=(iterator& a);
112.                     bool operator!=(node* a);
113.                     void forward();
114.                     void operator++();
115.                     void operator++(int);
116.                     iterator& operator=(pair<node**,int> temp
    );
117.                     iterator& operator=(iterator& temp);
118.                     node* operator->();
119.
120.             };
121. };
122. template <class T,class V>
123. bool myhash<T,V>::iterator::operator==(myhash<T,V>::iterator&
     a){
124.     return ptr==a.ptr;
125. }
126. template <class T,class V>
127. bool myhash<T,V>::iterator::operator!=(myhash<T,V>::iterator&
     a){
128.     return ptr!=a.ptr;
129. }
130. template <class T,class V>
131. bool myhash<T,V>::iterator::operator!=(node* a){
```

```
132.     return ptr!=a;
133. }
134. template <class T,class V>
135. void myhash<T,V>::iterator::forward(){
136.     if(ptr && ptr->next){
137.         ptr=ptr->next;
138.         return;
139.     }
140.     ptr=NULL;
141.     while(cur!=n && ptr==NULL){
142.         ptr=arr[cur++];
143.     }
144. }
145. 
146. template <class T,class V>
147. void myhash<T,V>::iterator::operator++(){
148.     forward();
149. }
150. template <class T,class V>
151. void myhash<T,V>::iterator::operator++(int){
152.     forward();
153. }
154. 
155. template <class T,class V>
156. typename myhash<T,V>::iterator& myhash<T,V>::iterator::operat
    or=(pair< myhash<T,V>::node **,int> temp){
157.     this->arr=temp.first;
158.     this->n=temp.second;
159.     this->cur=0;
160.     this->ptr=NULL;
161.     forward();
162.     return *this;
163. }
164. template <class T,class V>
165. typename myhash<T,V>::iterator& myhash<T,V>::iterator::operat
    or=(myhash<T,V>::iterator& temp){
166.     memcpy(this,&temp,sizeof(temp));
167.     return *this;
168. }
169. template <class T,class V>
170. myhash<T,V>::iterator::iterator(){
171.     arr=NULL,ptr=NULL;
172.     n=0,cur=0;
173. }
174. template <class T,class V>
```

```
typename myhash<T,V>::node* myhash<T,V>::iterator::operator->(){
    return this->ptr;
}


struct infor{
    char name[21];
    char phone[12];
    char add[51];
}arr[10000];
int n;
int mapping_phone(char*& phone){
    //hash 映射函数
    int ans=0;
    int dev=1e9+7;
    for(int i=0;phone[i];i++){
        ans=(ans*10+phone[i]-'0')%dev;
    }
    return ans;
}
void test1(){
    //以电话号码作为键
    myhash<char*,pair<char*,char*>> dp(mapping_phone);
    for(int i=0;i<n;i++){
        dp[(char*)arr[i].phone]={(char*)arr[i].add,(char*)arr[i].name};
    }
    myhash<char*,pair<char*,char*>>::iterator it;
    for(it=dp.begin();it!=dp.end();it++){
        printf("phone num:%s address:%s name:%s\n",it->first,it->second.first,it->second.second);
    }
    dp.erase((char*)arr[0].phone);
    printf("after delete one items\n");
    for(it=dp.begin();it!=dp.end();it++){
        printf("phone num:%s address:%s name:%s\n",it->first,it->second.first,it->second.second);
    }
}
void input(){
    scanf("%d",&n);
    getchar();
    for(int i=0;i<n;i++){
```

```
        scanf("%s %s %s",arr[i].name,arr[i].phone,arr[i].add);
    }
    /*
    3
    sanglonglong 12345 xidian
    wangyifa 110 xidainnan
    lixiaofei 120 xidianbei
    */
}
int main(){
    input();
    test1();
}
```