搜索

前置

顺序搜索小优化——设置监视哨

将要搜索的值赋值给数组最后一个元素的下一个位置，这样就不用每次比较都判断是否越界

```
int SeqSearchWatch(T x) {
        int i = 0;
        //将数组最后一个元素的下一个元素赋值为要搜索的值，这样就不用每次循环都判断是否越界（肯定会找到，
        element[currentSize].key = x;
        while (element[i].key != x)i++;
        if (i == currentSize)return -1;
        else return i + 1;
}
```

基于有序数据的二分查找

结点结构体定义：data+key，按key查找

```
template <class T,class E>
struct DataNode {
        T key;
        E data;
};
```

写代码时注意low，high，mid的边界情况和查找结束/失败时的情况就行
下面的代码返回的是物理位置：数组下标+1
迭代实现

```
template <class T>
int BinarySearch(T x, DataNoed<T,E>* element,int size) {
        int low= 0;
        int mid = 0;
        int high = size;
        while (low<=high) {
                mid = (low + high) / 2;
                if (element[mid].key == x)return mid+1;
                if (element[mid].key > x)high = mid - 1;
                if (element[mid].key < x)low = mid + 1;
        }
        return -1;
}
```

递归实现

```
template <class T,class E>
int BinarySearchR(T x,DataNode<T,E> *element,int low,int high) {
        if (low > high)return -1;
        int mid = (low + high) / 2;
        if (x > element[mid].key)mid=BinarySearchR(x,element, mid + 1, high);
        if (x < element[mid].key)mid=BinarySearchR(x,element, low, mid - 1);
        else return mid;//实际位置=mid+1
}
```

BST二叉搜索树

- 思想：用动态搜索结构实现二分查找
- 左子树的值>根节点值>右子树的值
- 中序遍历下输出递增序列
- 搜索：类似于二分查找，很方便
- 插入：
  - 操作：从根节点开始不断比较直至找到可以插入的空位置（插入后成为叶子结点）
  - 同样的值，不同的插入顺序，会构建出不同的BST（这就需要AVL进行优化结构）
- 删除：分为三种情况（设删除结点为 `del` ）
  - `del` 没有孩子:直接删
  - `del` 只有一个孩子：孩子接替 `del` 的位置(孩子的子树一并带过来)，然后删 `del`
  - `del` 有两个孩子：用中序遍历下 `del` 的前驱或 `del` 的后继结点接替 `del` ，并将删除任务转变为删值=接替结点的值（最后还是会变成前两种情况）
- 自己写删除代码时的问题：
  - 删除操作在遍历前/后时，代码里的else写在哪
  - 删除任务转变后，往哪去删除?

1. 用前驱接替： `Remove(temp->data,root->left)`
2. 用后继接替： `Remove(temp->data,root->right)`

代码实现

```cpp
#include<iostream>
using namespace std;
template <class T>
struct BSTNode {
	T data;
	BSTNode* left;
	BSTNode* right;
};
template <class T>
class BST {
public:
	BST() {root = nullptr;};
	~BST() {};
public:
	T getRootData() { return root->data; }
	void insert(T d) {insert(d, root);}
	void MidPrint() {MidPrint(root);}
	BSTNode<T>* search(T d) { return search(d, root); }
	bool Remove(T d) { return Remove(d, root); }
private:
	BSTNode<T>* root;
	void insert(T d,BSTNode<T> *&root) {
		if (root == nullptr) {
			BSTNode<T>* insert = new BSTNode<T>;
			insert->data = d;
			insert->left = insert->right = nullptr;
			root = insert;
			return;
		}
		if (d < root->data)insert(d, root->left);
		else insert(d, root->right);
	}
	void MidPrint(BSTNode<T>* root) {
		if (root == nullptr)return;
		MidPrint(root->left);
		cout << root->data << " ";
		MidPrint(root->right);
	}
	BSTNode<T>* search(T d, BSTNode<T>* root) {
		if (root == nullptr)return nullptr;
		if(d==root->data)return root;
		if (d < root->data)search(d, root->left);
		else search(d, root->right);
	}
	bool Remove(T d, BSTNode<T>*& root) {
		BSTNode<T>* temp;
		if (root == nullptr)return false;
		//如果将删除结点的操作放到遍历前，第一个判断递归Remove就要用else if
		//因为删除结点操作中有一种情况没有return，若没有else，会继续执行下面语句导致错误
		if (d < root->data)Remove(d, root->left);
		if (d > root->data)Remove(d, root->right);
```

```
                //有else才能说明d==root->data,找到了被删结点
                //若没有else，则无论有没有找到，只要前面Remove调用完，就会走到下面的删除，导致乱删
                else if (root->left != nullptr && root->right != nullptr) {//被删结点有两个孩子
                        //temp指针去找在中序遍历下被删结点root的后继结点
                        //中序遍历下结点的后继是该结点右子树中左支走到底的那个结点
                        temp = root->right;
                        //将被删结点的值替换为后继结点的值
                        while (temp->left != nullptr)temp = temp->left;
                        root->data = temp->data;
                        //删除任务转变成了删结点值=temp->data的结点
                        //注意第二个参数，不能直接赋值为temp
                        Remove(temp->data, root->right);
                }
                //被删结点有1个或0个孩子（看似分两种情况，实际上代码可以一次解决）
                else {
                        temp = root;
                        if (root->left == nullptr) {
                                root = root->right;
                        }
                        else root = root->left;
                        delete temp;
                        return true;
                }
        }
};
```

AVL平衡树（优化BST）

- 提出：有时构建出的BST极度不平衡，甚至退化为线性结构，搜索效率降低
- 前置规定：
  - 平衡因子bf=右子树高度-左子树高度
  - 什么时候平衡：$|bf|$<=1
  - 最小失衡子树：从新插入的结点开始向上查找，以第一个bf失衡的结点为根的子树
- 四种失衡类型及对应旋转方法（设最小失衡子树根节点root）
  详细图解点这里
  思路：判断类型后先明确旋转完后以谁为新的根节点。右子树高，则向左旋转，左子树高，则向右旋转，旋转完还要更新bf值

> *书面描述旋转（结合上面的图解）*
> 左旋转：root的右孩子变成其右孩子的左孩子，然后root变成其右孩子的左孩子
> 右旋转：root的左孩子变成其左孩子的右孩子，然后root变成其左孩子的右孩子

1. 单旋转（根与较高子树根结点bf同号）
   旋转后，根节点变为root的l或r（看类型）
   - LL型(右旋)：root左子树高，root的左孩子也是左边高

    - RR型（左旋）：root右子树高，root的右孩子也是右边高
2. 双旋转（根与较高子树根节点bf异号）
   从下往上调整，旋转后根节点变为root的l的r或root的r的l（看类型）
    - LR型（先左后右旋）：root的左子树高，root的左孩子是右边高
    - RL型（先右后左选）：root的右子树高，root的右孩子是左边高

- 插入
  找到插入位置并插入$\Longrightarrow$回溯更新沿途节点bf，找最小失衡子树 $\Longrightarrow$ 找到后判断失衡类型并旋转(只需调整一次，不用再回溯了)$\Longrightarrow$将调整的那部分接回AVL树
    - 实现细节
      1. 沿途结点用栈维护；找插入位置，更新父节点bf，判断失衡类型都要用到双指针
      2. 更新后的父节点bf
        1. bf=0,已经平衡，结束回溯，插入成功
        2. $|bf|$=1,父节点平衡，仍要回溯，更新父节点的父节点bf并判断
        3. $|bf|$>1，父节点失衡，判断失衡类型并旋转

代码实现

```cpp
#include<iostream>
#include<vector>
using namespace std;
template <class T>
struct AVLNode {
	T data;
	int bf;//平衡因子=右子树高度-左子树高度
	AVLNode* left;
	AVLNode* right;
	AVLNode(T d) :data(d), bf(0), left(nullptr), right(nullptr) {};
};
template <class T>
class AVL {
public:
	AVL() { root = nullptr; };
	~AVL() {};
public:
	T getRootData() { return root->data; }
	void MidPrint() { MidPrint(root); }
	bool insert(T d) {
		return insert(root, d);
	}
private:
	AVLNode<T>* root;
	void MidPrint(AVLNode<T>* root) {
		if (root == nullptr)return;
		MidPrint(root->left);
		cout << root->data << " ";
		MidPrint(root->right);
	}
	void RotateL(AVLNode<T>*& parent) {
		AVLNode<T>* newchild = parent;
		parent = parent->right;
		newchild->right = parent->left;
		parent->left = newchild;
		parent->bf = newchild->bf = 0;
	};
	void RotateR(AVLNode<T>*& parent) {
		AVLNode<T>* newchild = parent;
		parent = parent->left;
		newchild->left = parent->right;
		parent->right = newchild;
		parent->bf = newchild->bf = 0;
	};
	void RotateLR(AVLNode<T>*& parent) {
		AVLNode<T>* ancestor = parent;
		parent = parent->left;
		AVLNode<T>* father = parent;
		parent = parent->right;
		//先左旋
		father->right = parent->left;
```

```
        parent->left = father;
        if (parent->bf <= 0)father->bf = 0;
        else father->bf = -1;
        //后右旋
        ancestor->left = parent->right;
        parent->right = ancestor;
        if (parent->bf == -1)ancestor->bf = 1;
        else ancestor->bf = 0;
        parent->bf = 0;
};
void RotateRL(AVLNode<T>*& parent) {
        AVLNode<T>* ancestor = parent;
        parent = parent->right;
        AVLNode<T>* father = parent;
        parent = parent->left;
        //先右旋
        father->left = parent->right;
        parent->right = father;
        if (parent->bf >= 0)father->bf = 0;
        else father->bf = 1;
        //后左旋
        ancestor->right = parent->left;
        parent->left = ancestor;
        if (parent->bf == 1)ancestor->bf = -1;
        else ancestor->bf = 0;
        parent->bf = 0;
};
bool insert(AVLNode<T>*& root, T d) {
        //空树直接插入并返回true
        if (root == nullptr) {
                root = new AVLNode<T>(d);
                return true;
        }
        vector<AVLNode<T>*> stack;//存放经过的结点,为了插入后的回溯
        AVLNode<T>* temp = root;//用做遍历指针
        AVLNode<T>* parent = nullptr;//双指针--父指针
        //找插入位置
        while (temp != nullptr) {
                if (d == temp->data)return false;//有重复值则不插入
                else {
                        parent = temp;
                        stack.push_back(parent);
                        if (d < temp->data)temp = temp->left;
                        else temp = temp->right;
                }
        }
        //找到插入位置后，创建新节点并插入
        AVLNode<T>* newNode = new AVLNode<T>(d);
        if (d < parent->data)parent->left = newNode;
        else parent->right = newNode;
        //回溯：更新父节点bf，并判断是否要调整（只需调整一次）
```

```
                AVLNode<T>* child = newNode;
                while (!stack.empty()) {
                        //获得当前结点的父节点
                        parent = stack.back();
                        stack.pop_back();
                        //更新父节点bf
                        if (parent->left == child)parent->bf--;
                        else parent->bf++;
                        //判断是否要调整
                        if (parent->bf == 0)break;
                        if (parent->bf == 1||parent->bf==-1)child = parent;
                        else {
                                int childBf = (parent->bf < 0) ? -1 : 1;
                                //判断子节点bf值的同时也判断了父与子的bf是否同号
                                if (child->bf == childBf) {//父与子的bf同号，单旋转
                                        if (child->bf == -1)RotateR(parent);
                                        else RotateL(parent);
                                }
                                else {//父与子的bf异号，双旋转
                                        if (child->bf == -1)RotateRL(parent);
                                        else RotateLR(parent);
                                }
                                break;//调整一次就行
                        }
                }//end while
                //将调整的那一部分接回AVL里
                if (stack.empty())root = parent;
                else {
                        AVLNode<T>* n = stack.back();
                        if (parent->data < n->data)n->left = parent;
                        else n->right = parent;
                }
                return true;
        }
};
```

红黑树

DisJoint Set(union-find set并查集)

前置

干啥用的

（动态）等价问题；解决一些图论问题（[例子：kruskal算法](Graph.md#kruskal算法)

主要就是两个操作： `Find` 和 `Union`

> An efficient data structure to solve the equivalence problem.These operations（Find/Union） are important in many graph theory problems and also in compilers which process equivalence (or type) declarations.

啥是不相交集合（Disjoint Set）

> Each set has a different
> element, so that Si ∩ Sj = ∅; this makes the sets disjoint.

等价关系和等价类：离散数学的知识

存储结构
同一个等价类放在一个树里（不是二叉树），这样的话，树根就可以代表这一个等价类（或者说set）。这么多树的集合就构成了一个森林

> One idea might be to use a tree to represent each set, since each element in a tree has the same root. Thus, the root can be used to name the set. We will represent each set by a tree. (Recall that a collection of trees is known as a forest.) Initially, each set contains one element.

用数组s存储上述的森林，

$$s[i]=\begin{cases}元素i的parent & i不是root\\ 按union方法而异（<0，只有根节点时是-1） & i是root\end{cases}$$

> The name of a set is given by the node at the root. Since only the name of the parent is required, we can assume that this tree is stored implicitly in an array: Each entry s[i] in the array represents the parent
> of element i. If i is a root, t

两个基本操作：Find和Union

不需要比较

> We do not perform any operations comparing the relative values of elements
> but merely require knowledge of their location.

Find
理论
find返回的是元素所在集合（等价类）的名字

> A find(x) on element x is performed by returning the root of the tree containing x.finds on two elements return the same answer if and only if they are in the same set.

Union
理论

> If we want to add the relation a ∼ b, then we first see if a and b are already related. This is done by performing finds on both a and b and checking whether they are in the same equivalence class. If they are not, then we apply union.This operation merges the two equivalence classes containing a and b into a new equivalence class.

一个不太好的Union方法：容易搞出退化的树
将一个树的root直接置为另一个树的子女

> Making the second tree a subtree of the first

优化并查集性能

并查集主要的操作就是 `find` 和 `union` ，从上述讨论中发现，简单的 `find` 和 `union` 性能并不理想，需要去优化

优化Union：为了避免产生退化的树

- **union-by-size**
  $$s[i]=\begin{cases}\text{元素}i\text{的}parent & i\text{不是}root\\ \text{树的结点总数（}<0\text{，只有根节点时是}-1\text{）} & i\text{是}root\end{cases}$$
  理论
  并操作时，结点少的树成为结点多的树的孩子
  > A simple improvement is always to make the smaller tree a subtree of the larger,breaking ties by any method

  性能分析
  > If unions are done by size, the depth of any node is never more than logN.This implies that the running time for a find operation is O(logN),and a sequence of M operations takes O(M logN).

- **union-by-height**
  $$s[i]=\begin{cases}\text{元素}i\text{的}parent & i\text{不是}root\\ \text{树的高度（按结点定义高度）（}<0\text{，初值为}-1\text{）} & i\text{是}root\end{cases}$$
  理论
  高度小的树成为高度大的树的孩子，实际上和union-by-size思路是一样的
  > We keep track of the height, instead of the size,of each tree and perform unions by making the shallow tree a subtree of the deeper tree.

性能
和union-by-size一样

优化Find：Path Compression
当 `union` 优化到极致后，我们发现worst-case trees的产生无法避免，那就只能在 `find` 操作上优化

> This is based on the observation that any method to perform the unions will yield the same worst-case trees, since it must break ties arbitrarily. Therefore, the only way to speed the algorithm up, without reworking the data structure entirely, is to do something clever on the find operation

路径压缩什么情况下使用
路径压缩是在 `Find` 操作里进行的，所以理论上适配任何一种 `union` （独立）。但实际 union-by-size最适配路径压缩。路径压缩会改变树的高度，所以搭配union-by-height就有点复杂

> Path compression is performed during a find operation and is independent of the strategy used to perform unions. Path compression is perfectly compatible with union-by-size, and thus both routines can be implemented at the same time. Path compression is not entirely compatible with union-by-height, because path compression can change the heights of the trees.

做法
查x，则将x到root的路径上所有结点都变成root的子女（包括x）

> Suppose the operation is find(x). Then the effect of path compression is that every node on the path from x to the root has its parent changed to the root.

代码示例

这里优化 `union` 采用union-by-size的方法，因此数组s[i]的含义为：

$$s[i]=\begin{cases}\text{元素}i\text{的}parent & i\text{不是}root\\ \text{树的结点总数（}<0\text{，只有根节点时是}-1\text{）} & i\text{是}root\end{cases}$$

```cpp
#include<iostream>
#include<vector>
using namespace std;
class DisjSets {
public:
	DisjSets() :size(10) {
		s = new int[size];
		for (int i = 0; i < size; i++)s[i] = -1;
	}
	DisjSets(int m) :size(m) {
		s = new int[size];
		for (int i = 0; i < size; i++)s[i] = -1;
	}
	~DisjSets() { delete[]s; }
public:
	int find(int x) {//未优化的find
		int root = x;
		//找x的根
		while (s[root] >= 0)root = s[root];
		return root;//返回x的根
	}
	void unionSets(int root1, int root2) {//未优化的union
		if (root1 == root2 || s[root1] >= 0 || s[root2] >= 0)return;
		//将root2变成root1的子女
		s[root1] += s[root2];//更新结点数
		s[root2] = root1;
	}
	void unionBySize(int x1, int x2) {//优化union
		//合并x1所在的集合和x2所在的集合
		int root1 = find(x1);
		int root2 = find(x2);
		if (root1 == root2)return;//本来就在一个集合里，不用合并
		if (s[root1] > s[root2]) {//root2的结点多，root1变成root2的子女
			s[root2] += s[root1];
			s[root1] = root2;
		}
		else {//root1的结点多，root2变成root1的子女
			s[root1] += s[root2];
			s[root2] = root1;
		}
	}
	int CollapsingFind1(int x) {//压缩路径优化find，非递归
		int root = x;
		while (s[root] >= 0)root = s[root];//找到root
		while (x != root) {//从x开始将沿途结点变成root的子女
			int temp = s[x];//保存x的父结点
			s[x] = root;//x变成root的子女
			x = temp;//往上继续改造
		}
		return root;
	}
```

```
        int CollapsingFind2(int x) {//压缩路径优化find，递归
                if (s[x] < 0)return x;
                else {
                        s[x] = CollapsingFind2(s[x]);
                }
        }
private:
        int* s;
        int size;//一般不会添加元素，不用设置curSize，new多大就有多少元素
};
```

散列表

哈希函数

- 要求
  1. 函数定义域包含所有的key
  2. 设哈希表有m个位置，则函数值域为0~m-1
  3. 计算出来的地址均匀分布在地址空间

处理冲突

- Collision

> Two keys may hash to the same slot. We call this situation a collision.

- Load factor

> Given a hash table T with m slots that stores n elements, we define the load factor α for T as n=m

Chaining

> In chaining, we place all the elements that hash to the same slot into the same linked list

注：当链表很长时，搜索效率降低，可以设计算法，当长度到一定值时，将链表转变成AVL或红黑树
代码

- 哈希函数：除留余数法
- 结点以键值对的形式
- 注意二级指针的用法

```cpp
#include<iostream>
using namespace std;
//K:关键码的数据类型      V:value的数据类型
template <class K,class V>
struct ChainNode {//链地址结点
        K key;//关键码
        V value;//值
        ChainNode<K,V>* next;
        ChainNode<K, V>():next(nullptr) {};
        ChainNode<K, V>(K k,V v):key(k),value(v),next(nullptr) {};
};
template <class K, class V>
class HashTable {
public:
        HashTable() :divisor(97), tableSize(100) {
                //全部初始化为nullprt (new出来的默认不是nullptr，要手动置空)
                table = new ChainNode<K, V>*[100] {nullptr};
        }
        HashTable(int sz) :tableSize(sz) {
                //全部初始化为nullprt (new出来的默认不是nullptr，要手动置空)
                table = new ChainNode<K, V>*[sz] {nullptr};
                //除数取不大于最大容量的素数
                divisor = sz;
                while (!isPrime(divisor))divisor--;
        };
        ~HashTable() {
                for (int i = 0; i < tableSize; i++)delete[]table[i];
                delete []table;
        }
public:
        bool insert(K key,V value) {//插入
                //通过哈希函数找到slot位置
                int pos = key % divisor;
                ChainNode<K, V> *put = new ChainNode<K, V>(key, value);
                //插入操作
                if (table[pos] == nullptr)table[pos] = put;
                else {
                        ChainNode<K, V>* temp = table[pos];
                        while (temp->next != nullptr) {
                                //如果有重复value则插入失败
                                if (value == temp->value)return false;
                                temp = temp->next;
                        }
                        temp->next = put;
                }
                        return true;
        };
        bool search(const K key, V& value) {//搜索
                //按关键码寻找
                ChainNode<K, V> *temp = Find(key);
                if (temp == nullptr)return false;//找不到false
```

```
                else {//找得到就获得对应的值，返回true
                        value = temp->value;
                        return true;
                }
        };
        bool remove(const K key, V& value) {//删除
                if (Find(key) == nullptr)return false;//没有这个数据，删除失败
                int pos = key % divisor;//找到在表中哪个槽位
                ChainNode<K, V>* pre = nullptr;//被删结点的前驱
                ChainNode<K, V>* temp = table[pos];//被删结点
                while (temp->key != key) {//找被删结点
                        pre = temp;
                        temp = temp->next;
                }
                if (pre == nullptr) table[pos] = table[pos]->next;//删除首结点时
                else pre->next = temp->next;//否则，常规的链表删除
                value = temp->value;//保存被删结点的vlaue
                delete temp;
                return true;
        };
private:
        int divisor;//除数（必须质数）
        ChainNode<K, V>** table;
        int tableSize;//哈希表的slots总量
        //判断是否素数
        bool isPrime(int d) {
                if (d <= 1)return false;
                for (int i = 2; i < sqrt(d); i++) {
                        if ((d % i) == 0) return false;
                }
                return true;
        }
        //根据关键码寻找，找到则返回结点，没找到则返回nullprt
        ChainNode<K, V>* Find(const K key) {
                //先找到是在哪个链表中
                int pos = key % divisor;
                //创指针指向key在的那个链表
                ChainNode<K, V>* p = table[pos];
                //遍历找到关键码为key的元素并返回
                while (p != nullptr) {
                        if (p->key == key)break;
                        else p = p->next;
                }
                return p;
        }
};
```

Open Addressing

> In open addressing, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table. No lists and no elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made; one consequence is that the load factor α can never exceed 1.

- **Linear probing**
  - 做法
    位置冲突，则向后逐个找直到有空位置，将元素放入空位置
  - 公式
    假设H_0=Hash(key)冲突，则H_i=(H_0+1)%m, i=1,2,3,...,m-1
- **Quadratic probing**
  - 做法
    位置冲突，则向后找距离为i^2的位置，若有空位则安放；
    若没空位置，则往前找距离为i^2的位置...重复上述操作直到有位置放（$i = 1, 2, 3, ...$）
  - 要求
    1. 表大小需满足4*k+3的质数
    2. α<=0.5时，新元素一定有位置可以插入，且任何位置不会被探查两次
  - 公式
    假设H_0=Hash(key)冲突
    H_i=(H_0+i^2)%m
    H_i=(H_0-i^2)%m, i=1,2,3,...m-1
- **Double hashing**
  - 做法
    第一个哈希函数算基准位置，第二个哈希函数算偏移量
  - 公式
    j=H_0=Hash(key),p=ReHash(key)
    H_i=(H_0+i*p)%m, i=1,2,3,...,m-1

Open Adressing的删除操作比较特殊

代码（以Linear probing为例）

```
#include<iostream>
using namespace std;
//K:关键码的数据类型      V:value的数据类型
template <class K, class V>
struct HashNode {//哈希结点
        K key;//关键码
        V value;//值
        bool isActive=false;//标志该结点是否处于激活状态：默认/被删除是false
        HashNode<K, V>():isActive(false) {};
        HashNode<K, V>(K k, V v) :key(k), value(v),isActive(true) {};
};
template <class K, class V>
class HashTable {
public:
        HashTable() :divisor(97), tableSize(100) {
                table = new HashNode<K,V>[100];
                currentSize = 0;
        }
        HashTable(int sz) :tableSize(sz) {
                table = new HashNode<K, V>[sz] ;
                //除数取不大于最大容量的素数
                divisor = sz;
                while (!isPrime(divisor))divisor--;
        };
        ~HashTable() {delete []table;}
public:
        bool insert(K key,V value) {//插入
                if (FindPos(key) != -1)return false;
                HashNode<K, V> s(key, value);
                int pos = key % divisor;
                //找插入位置：必须找到未被激活（使用）的位置
                while (table[pos].isActive==true)pos++;
                table[pos] = s;
                return true;
        };
        bool search(const K key, V& value) {//搜索
                int pos = FindPos(key);
                if (pos == -1) return false;
                else value = table[pos].value;//保存搜索到的元素值
                return true;
        };
        bool remove(const K key, V& value) {//删除
                //删除：将该位置设置为未激活状态
                int pos = FindPos(key);
                if (pos == -1)return false;//元素不在表里，删除失败
                else {
                        value = table[pos].value;//保存被删元素的值
                        table[pos].isActive=false;//将该位置标记为未激活状态
                }
                return true;
        };
```

```
private:
        int divisor;//除数（必须质数）
        HashNode<K,V>* table;
        int tableSize;//哈希表的slots总量
        int currentSize;//当前元素数量
        //判断是否素数
        bool isPrime(int d) {
                if (d <= 1)return false;
                for (int i = 2; i < sqrt(d); i++) {
                        if ((d % i) == 0) return false;
                }
                return true;
        }
        //根据关键码寻找，找到则返回结点，没找到则返回nullprt
        int FindPos(const K key) {
                //通过哈希函数获得位置坐标，flag是为了保存这个原始位置
                int pos = key % divisor;
                int flag = pos;
                while (table[pos].key != key) {
                        pos=(pos+1)%tableSize;
                        if (pos == flag)return -1;//找了一圈，又回到了最初的起点，说明不在哈希表里
                }
                if (table[pos].isActive == false)return -1;//匹配到了，但处于未激活状态（被删），也
                else return pos;
        }
};
```

性能分析

完美哈希

B树B+树