查找

顺序查找 折半查找

- 顺序查找，折半查找判定树（王道）
- 平均查找长度ASL计算
- B树

真题2019

- 已知一组关键字{21，33，12，40，68，59，25，51}

（1）试依次插入关键字生成一棵 3 阶的 B-树；

（2）在生成的 3 阶 B-树中依次删除 40 和 12，画出每一步执行后 B-树的状态。

二叉排序树

二叉排序树（Binary Sort Tree）又称**二叉查找树**，它是一种对排序和查找都很有用的特殊二叉树。

1. 二叉排序树的定义

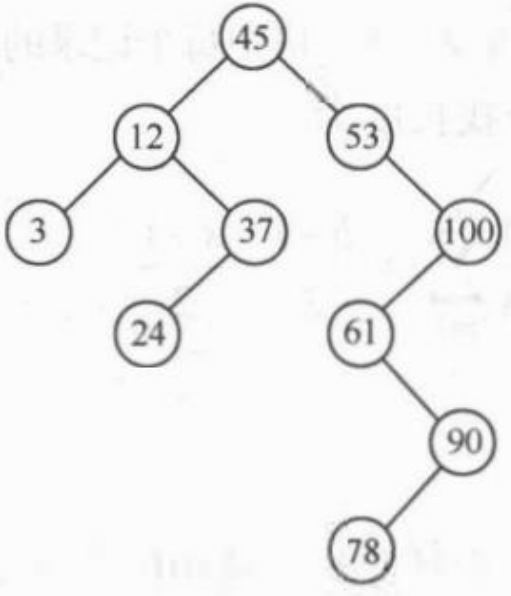
二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

（1）若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

（2）若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

（3）它的左、右子树也分别为二叉排序树。

二叉排序树是递归定义的。由定义可以得出二叉排序树的一个重要性质：中序遍历一棵二叉树时可以得到一个结点值递增的有序序列。



算法 7.4　二叉排序树的递归查找

【算法步骤】

① 若二叉排序树为空，则查找失败，返回空指针。

② 若二叉排序树非空，将给定值 key 与根结点的关键字 T->data. key 进行比较：

- 若 key 等于 T->data. key，则查找成功，返回根结点地址；
- 若 key 小于 T->data. key，则递归查找左子树；
- 若 key 大于 T->data. key，则递归查找右子树。

```
BSTNode * Search(BSTNode * T, int value){
    if (!T || value == T->data) return T;
    else if (value<T->data) return Search(T->lchild, value);
    else return Search(T->rchild, value);
}
```

算法 7.5　二叉排序树的插入

【算法步骤】

① 若二叉排序树为空，则待插入结点*S 作为根结点插入到空树中。

② 若二叉排序树非空，则将 key 与根结点的关键字 T->data.key 进行比较：

- 若 key 小于 T->data.key，则将*S 插入左子树；
- 若 key 大于 T->data.key，则将*S 插入右子树。

```
int BSTInsert(BTNode *&bt,int key)//再次强调，因为指针 bt 要改变，所以要用引用型指针
{
    if(bt==NULL)                    //当前为空指针时说明找到插入位置，创建新结点进行插入
    {
        bt=(BTNode*)malloc(sizeof(BTNode));//创建新结点
        bt->lchild=bt->rchild=NULL;
        bt->key=key;                //将待插关键字存入新结点内，插入成功，返回 1
        return 1;
    }
    else                            //如果结点不空，则查找插入位置，这部分和查找算法类似
    {
        if(key==bt->key)            //关键字已存在于树中，插入失败，返回 0
            return 0;
        else if(key<bt->key)
            return BSTInsert(bt->lchild,key);
        else
            return BSTInsert(bt->rchild,key);
    }
}
```

真题

1. 已知一棵二叉排序树(结点值大小按字母排序)的前序遍历序列为 EBACDFHG，请回答下列问题:

(1) 画出此二叉排序树(3 分)

(2) 若将此二叉排序树看做森林的二叉链表存储，请画出对应的森林(3 分)

9. 对二叉排序树的查找都是从根开始的，查找失败时，一定落在_____结点上。

A. 根结点　　B.右孩子　　C.内结点　　D.叶子结点

真题

算法题 将二叉排序树中的各节点降序排列输出
迭代
递归

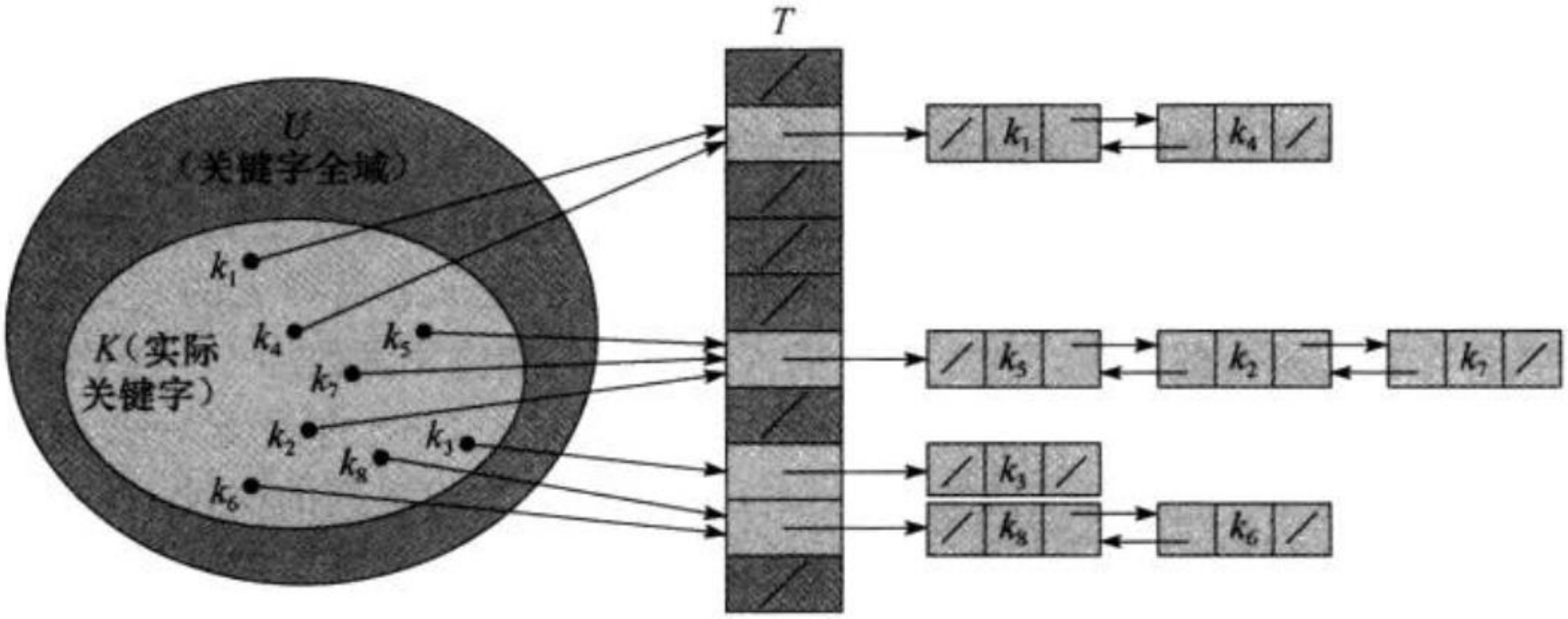
散列表

散列表是普通数组概念的推广。由于对普通数组可以直接寻址，使得能在 $O(1)$ 时间内访问数组中的任意位置。11.1 节将更详细地讨论直接寻址。如果存储空间允许，我们可以提供一个数组，为每个可能的关键字保留一个位置，以利用直接寻址技术的优势。

当实际存储的关键字数目比全部的可能关键字总数要小时，采用散列表就成为直接数组寻址的一种有效替代，因为散列表使用一个长度与实际存储的关键字数目成比例的数组来存储。在散列表中，不是直接把关键字作为数组的下标，而是根据关键字计算出相应的下标。11.2 节

散列表

- 假设要存储一系列关键字，U是关键字的全集



（1）**散列函数和散列地址**：在记录的存储位置 p 和其关键字 key 之间建立一个确定的对应关系 H，使 $p = H(key)$，称这个对应关系 H 为散列函数，p 为散列地址。

（2）**散列表**：一个有限连续的地址空间，用以存储按散列函数计算得到相应散列地址的数据记录。通常散列表的存储空间是一个一维数组，散列地址是数组的下标。

（3）**冲突和同义词**：对不同的关键字可能得到同一散列地址，即 $key_1 \neq key_2$，而 $H(key_1) = H(key_2)$，这种现象称为冲突。具有相同函数值的关键字对该散列函数来说称作同义词，key_1 与 key_2 互称为同义词。

散列表建立了关键字和存储地址之间的一种直接映射关系

例子

例如，对 C 语言某些关键字集合建立一个散列表，关键字集合为

$$S_1 = \{\text{main, int, float, while, return, break, switch, case, do}\}$$

设定一个长度为 26 的散列表应该足够，散列表可定义为

```
char HT[26][8];
```

假设散列函数的值取为关键字 *key* 中第一个字母在字母表{a, b,…, z}的序号（序号范围为 0～25），即

$$H(key) = key[0] - \text{'a'}$$

其中，设 key 的类型是长度为 8 的字符数组，根据此散列函数构造的散列表如表 7.1 所示。

表 7.1　　关键字集合 S_1 对应的散列表

0	1	2	3	4	5	…	8	…	12	…	17	18	…	22	…	25
	break	case	do		float		int		main		return	switch		while		

假设关键字集合扩充为：

$$S_2=S_1+\{\text{short, default, double, static, for, struct}\}$$

如果散列函数不变，新加入的七个关键字经过计算得到：$H(\text{short})=H(\text{static})=H(\text{struct})=18$，$H(\text{default})=H(\text{double})=3$，$H(\text{for})=5$，而 18、3 和 5 这几个位置均已存放相应的关键字，这就发生了冲突现象，其中，switch、short、static 和 struct 称为同义词；float 和 for 称为同义词，do、default 和 double 称为同义词。

实际情况下，不产生冲突的散列函数极少,这是因为通常散列表中的关键字的取值集合远远大于表空间的地址集。所以我们尽可能构造一个好的散列函数，并对冲突的情况进行处理。

常用的散列函数

1．直接定址法

直接取关键字的某个线性函数值为散列地址，散列函数为

$$H(\text{key}) = \text{key} \text{ 或 } H(\text{key}) = a \times \text{key} + b$$

式中，a 和 b 是常数。这种方法计算最简单，且不会产生冲突。它适合关键字的分布基本连续的情况，若关键字分布不连续，空位较多，则会造成存储空间的浪费。

2．除留余数法

这是一种最简单、最常用的方法，假定散列表表长为 m，取一个不大于 m 但最接近或等于 m 的质数 p，利用以下公式把关键字转换成散列地址。散列函数为

$$H(\text{key}) = \text{key} \% p$$

除留余数法的关键是选好 p，使得每个关键字通过该函数转换后等概率地映射到散列空间上的任一地址，从而尽可能减少冲突的可能性。

处理冲突的方法

选择一个“好”的散列函数可以在一定程度上减少冲突，但在实际应用中，很难完全避免发生冲突，所以选择一个有效的处理冲突的方法是散列法的另一个关键问题。创建散列表和查找散列表都会遇到冲突，两种情况下处理冲突的方法应该一致。下面以创建散列表为例，来说明处理冲突的方法。

处理冲突的方法与散列表本身的组织形式有关。按组织形式的不同，通常分两大类：开放地址法和链地址法。

1. 开放地址法

开放地址法的基本思想是：把记录都存储在散列表数组中，当某一记录关键字 *key* 的初始散列地址 $H_0 = H(key)$发生冲突时，以 H_0 为基础，采取合适方法计算得到另一个地址 H_1，如果 H_1 仍然发生冲突，以 H_1 为基础再求下一个地址 H_2，若 H_2 仍然冲突，再求得 H_3。依次类推，直至 H_k 不发生冲突为止，则 H_k 为该记录在表中的散列地址。

这种方法在寻找“下一个”空的散列地址时，原来的数组空间对所有的元素都是开放的，所以称为开放地址法。通常把寻找“下一个”空位的过程称为**探测**，上述方法可用如下公式表示：

$$H_i = (H(key) + d_i)\%m \qquad i = 1, 2, \cdots, k\ (k \leqslant m - 1)$$

其中，$H(key)$为散列函数，m 为散列表表长，d_i 为增量序列。根据 d_i 取值的不同，可以分为以下 3 种探测方法。

（1）线性探测法

$$d_i = 1, 2, 3, \cdots, m-1$$

这种探测方法可以将散列表假想成一个循环表，发生冲突时，从冲突地址的下一单元顺序寻找空单元，如果到最后一个位置也没找到空单元，则回到表头开始继续查找，直到找到一个空位，就把此元素放入此空位中。如果找不到空位，则说明散列表已满，需要进行溢出处理。

（2）二次探测法

$$d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \cdots, +k^2, -k^2 \ (k \leqslant m/2)$$

（3）伪随机探测法

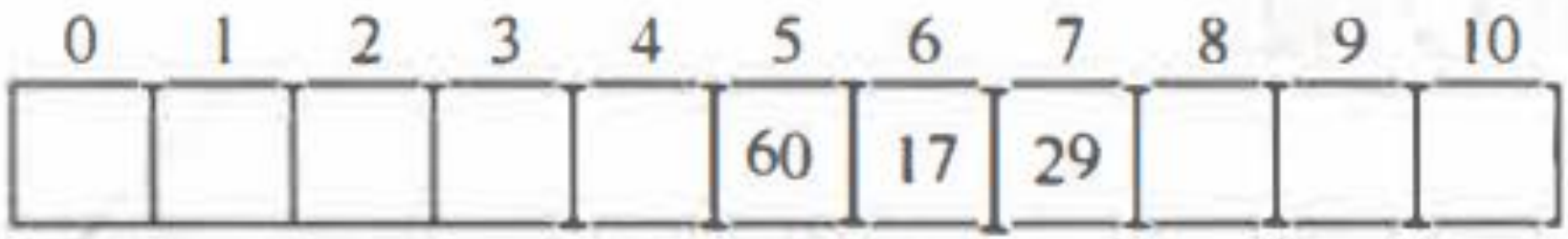
d_i = 伪随机数序列

例如，散列表的长度为 11，散列函数 $H(key) = key\%11$，假设表中已填有关键字分别为 17、60、29 的记录，如图 7.29（a）所示。现有第四个记录，其关键字为 38，由散列函数得到散列地址为 5，产生冲突。

若用线性探测法处理时，得到下一个地址 6，仍冲突；再求下一个地址 7，仍冲突；直到散列地址为 8 的位置为“空”时为止，处理冲突的过程结束，38 填入散列表中序号为 8 的位置，如图 7.29（b）所示。

若用二次探测法，散列地址 5 冲突后，得到下一个地址 6，仍冲突；再求得下一个地址 4，无冲突，38 填入序号为 4 的位置，如图 7.29（c）所示。

若用伪随机探测法，假设产生的伪随机数为 9，则计算下一个散列地址为(5+9)%11 = 3，所以 38 填入序号为 3 的位置，如图 7.29（d）所示。

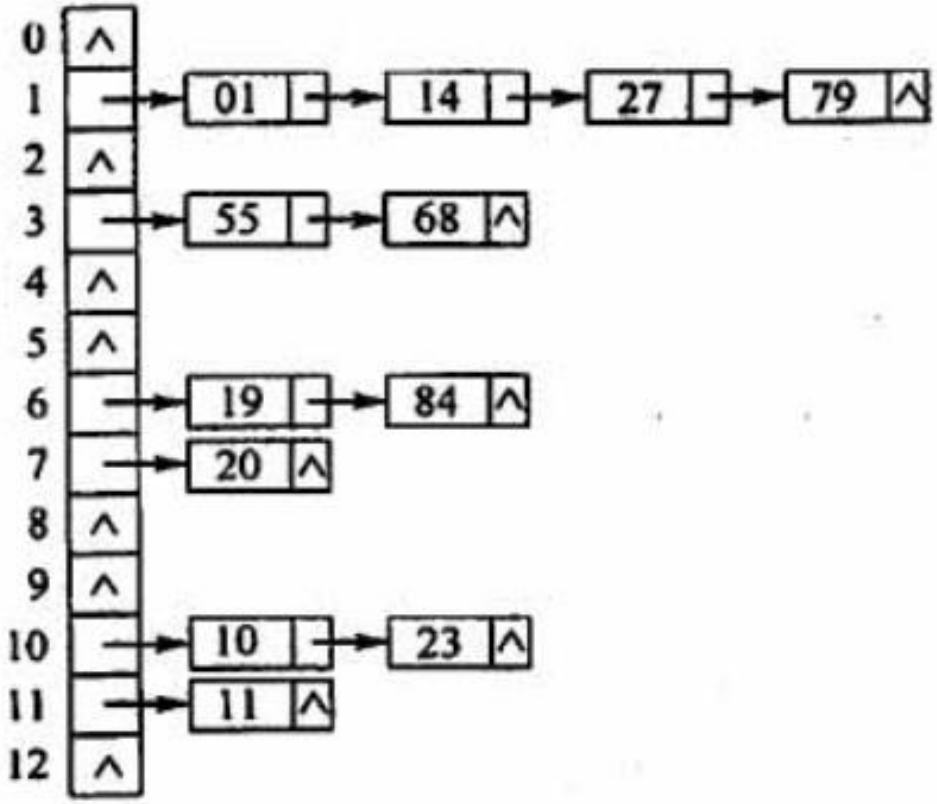


处理冲突的方法

2．拉链法（链接法，chaining）

显然，对于不同的关键字可能会通过散列函数映射到同一地址，为了避免非同义词发生冲突，可以把所有的同义词存储在一个线性链表中，这个线性链表由其散列地址唯一标识。假设散列地址为 i 的同义词链表的头指针存放在散列表的第 i 个单元中，因而查找、插入和删除操作主要在同义词链中进行。拉链法适用于经常进行插入和删除的情况。

例如，关键字序列为{19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79}，散列函数 H(key)=key%13，用拉链法处理冲突，建立的表如图 7.9 所示（学完下节内容后，可以尝试计算本例的平均查找长度 ASL）。



真题2012 2018

6. 在初始为空的 Hash 表中依次插入以下关键字序列 18,75,60，43,54,90,46,31,58,73,15,34，地址值域为[0,12]，Hash 函数为 H(k)=k mod 13，其中 k 为关键字，用线性探测再散列处理冲突，请画出 Hash 表，并求出在等概率下的平均查找长度。(8 分)

3. 设散列函数 f(k)=kmod13，散列表地址空间为 0-12，对给定的关键字序列(19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79），分别以拉链法和线性探查法解决冲突构造散列表，画出所构造的散列表，并指出在这两个散列表中查找每一个关键字时进行比较的次数。

2015

11. 为关键字(17, 33, 31, 40, 48)构造一个长度为 7 的散列表，设散列函数为 h(key)=key%7，用开放定址法解决冲突的探查序列是:(8 分)

hi=(h(key)+i(key%5+1))%7　0≤i≤6

(1) 画出构造所得的散列表

(2) 求出在等概率情况下查找成功时的平均查找长度

散列表的存储——开放地址法

```
//- - - - -开放地址法散列表的存储表示- - - - -
#define m 20                                    //散列表的表长
typedef struct{
    KeyType key;                                //关键字项
    InfoType otherinfo;                         //其他数据项
}HashTable[m];
```

算法 7.10　散列表的查找

【算法步骤】

① 给定待查找的关键字 key，根据造表时设定的散列函数计算 $H_0 = H(key)$。

② 若单元 H_0 为空，则所查元素不存在。

③ 若单元 H_0 中元素的关键字为 key，则查找成功。

④ 否则重复下述解决冲突的过程：

- 按处理冲突的方法，计算下一个散列地址 H_i；
- 若单元 H_i 为空，则所查元素不存在；
- 若单元 H_i 中元素的关键字为 key，则查找成功。

线性探测法哈希表查找过程

```
#define MaxSize 100
#define NULLKEY 0
int SearchHash(HashTable HT, int key){
    int h0 = HaskFun(key);
    if (HT[h0].key == NULLKEY) return -1;
    else if (HT[h0].key == key) return h0;
    // 线性探测法
    else{
        for(int i=1;i<MaxSize;i++){
            int hi = (h0+i)%MaxSize;
            if (HT[hi].key == NULLKEY) return -1;
            else if (HT[hi].key == key) return hi;
        }
        return -1;
    }
}
```

```c
int InsertHash(HashTable HT, int key, OtherInfo info){
    int h0 = HashFun(key);
    if (HT[h0].key == NULLKEY || HT[h0].key == key){
        HT[h0].key = key;
        HT[h0].otherInfo = info;
        return 1;
    }
    else{
        for(int i=1;i<MaxSize;i++){
            int hi = (h0+i)%MaxSize;
            // 为空
            if (HT[hi].key == NULLKEY) {
                HT[hi].key = key;
                HT[hi].otherInfo = info;
                return 1;
            }
            // 已经有相同的key
            else if (HT[hi].key == key){
                // 更新otherinfo
                HT[hi].otherInfo = info;
                return 1;
            }
        }
        return -1;
    }
}
```

线性探测法
哈希表插入过程

```c
int InsertHash(HashTable HT, int key, OtherInfo info){
    int h0 = HashFun(key);
    if (HT[h0].key == NULLKEY || HT[h0].key == key || HT[h0].key == DELETED){
        HT[h0].key = key;
        HT[h0].otherInfo = info;
        return 1;
    }
    else{
        for(int i=1;i<MaxSize;i++){
            int hi = (h0+i)%MaxSize;
            // 为空
            if (HT[hi].key == NULLKEY || HT[hi].key == DELETED) {
                HT[hi].key = key;
                HT[hi].otherInfo = info;
                return 1;
            }
            // 已经有相同的key
            else if (HT[hi].key == key){
                // 更新otherinfo
                HT[hi].otherInfo = info;
                return 1;
            }
        }
        return -1;
    }
}
```

线性探测法
哈希表删除过程?

```
#define DELETED -1
int DeleteHash(HashTable &HT, int key){
    int h0 = HashFun(key);
    if (HT[h0].key == NULLKEY) return 0;
    else if (HT[h0].key==key){
        HT[h0].key = DELETED;
        return 1;
    }
    else{
        for(int i=1;i<MaxSize;i++){
            int hi = (h0+i)%MaxSize;
            if (HT[hi].key == NULLKEY) return 0;
            else if (HT[hi].key == key){
                HT[hi].key = DELETED;
                return 1;
            }
        }
        return 0;
    }
}
```

散列表的存储——链接法

```
struct ListNode{
    int key;
    int value;
    struct ListNode *next;
}

#define Size 100
#define NULLKEY 0
typedef sturct {
    ListNode* bucket[Size];
}HashTable;
```

```
void InitHaskTable(HashTable ht){
    for(int i = 0;i<Size;i++)
        bucket[i] = nullptr;
}
```

链接法哈希表插入过程

```
void Insert(HashTable &ht, int key, int val){
    int h0 = HashFun(key);
    // 在当前桶中寻找key
    ListNode *p = ht.bucket[h0];
    while(p){
        if (p->key == key){
            // 找到key则更新value
            p->value = val;
            return ;
        }
        p = p->next;
    }
    // 找不到key 则插入key
    p = new ListNode;
    p->key = key;
    p->value = val;
    p->next = nullptr;
    // 头插法
    if (ht.bucket[h0])
        p->next = ht.bucket[h0];
    ht.bucket[h0] = p;
}
```

链接法哈希表查找/删除过程？

真题

五、编程题（每道 10 分，共 20 分）

1、假设散列表长为 m，散列函数为 H（K），用链地址处理冲突。试编写输入一组关键字构造散列表的算法。

作业

- 删除链表倒数第N个节点
- 包含min函数的栈 https://leetcode.cn/problems/bao-han-minhan-shu-de-zhan-lcof/
- 判断是否为满二叉树