数据结构复习

> `实验/`为个人提交
> `PPT/`为2025任国珍老师上课所用，有一点我的勘误
> `题目/`为前人搜集的往年题
> `一些资料/`如有侵权，联系我删除

这门课不难
实验课一定要拿满分（46开）
多练多练多练！先理解，再记代码。

考试题目为少量填空+大题。真正写代码的题目（30分）有两或三道，其它题目（70分）可能考察算法思想，或者让你画出某一时刻的结果、运行过程之类的。
对于每个数据结构，明白它的用处，增删改查写法；对于算法，明白算法思想，运行过程，能手动进行这个算法。
写代码的题（算法题）主要考察贪心、分治、动态规划，如果结合上前面学过的算法会比较难，看平时的积累。实在没思路就写个暴力枚举，图/树写个回溯，应该能给点分吧。

第一章 C++回顾

递归

写一个函数，相信它能完成任务，并在其中调用自己完成局部任务。一般包含递归出口和任务处理。

STL

我认为常见的STL考试时是可以写的，当然前提是本题的主要考点不是这个STL本身。比如本题就是考双向链表，肯定不能直接用list写。

第二、三、四章

这几章搞懂大O渐进记法就可以了。

空间复杂度

一般来说，不使用数组，$O(1)$，一维数组 $O(n)$，二维数组$O(n^2)$...
题目里说不使用额外空间时，不能开大小与规模n相关的数组，但是可以使用任意个变量。

时间复杂度

多数情况，可以看循环，t层循环就为$O(n^t)$。但考的难的话，是会两层循环但其实是$O(n)$的，这需要仔细分析。

第五章 线性表-数组描述

线性表数据结构

有序集合，形式为$(e_0, e_1, e_2, \dots e_{n-1})$，n是线性表的长度/大小。需要支持的操作有：

- 创建
- 销毁
- 判空
- 获取长度
- 按索引查找元素
- 按元素获取索引
- 按索引删除
- 按索引插入
- 从左至右输出

映射公式

最一般地，$location(i) = i$。也可以倒过来: $location(i) = arrayLength - 1 - i$。或者从特定位置开始存储：
$location(i) = (location(0) + i)\%arrayLength$等等，可以搞得比较花哨。

变长一维数组

arrayList

C++迭代器*（推荐掌握，但是考试应该不会考）

vector*

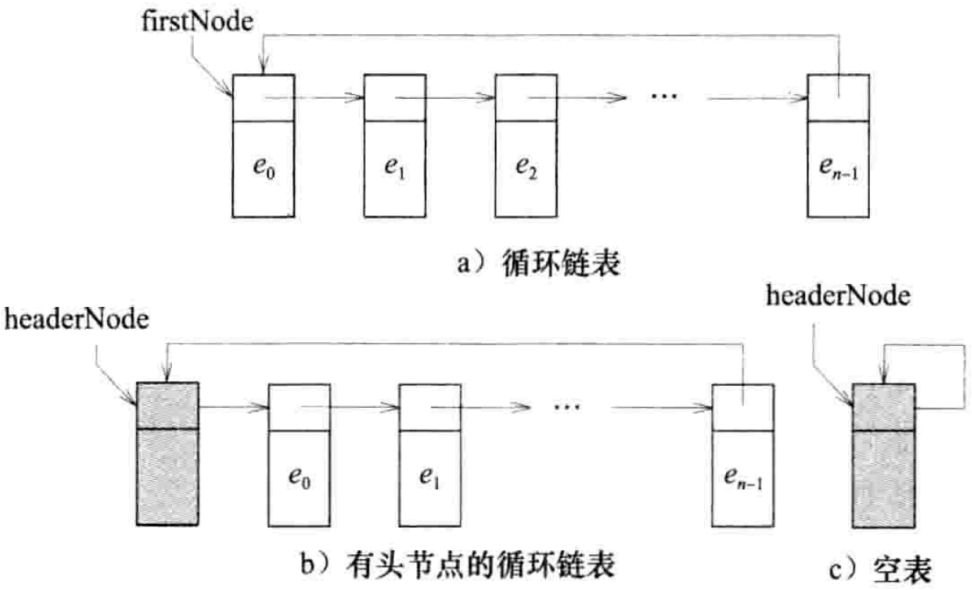
第六章 线性表-链式描述

单向链表

循环链表和头节点

为了方便某些操作，可以为链表单独设置一个头节点，它不存数据，只作为链表的入口使用。

把链表的最后一个节点的next指向链表的头节点，那么这个链表就是循环链表。



双向链表

每个节点，同时维护下一个元素与前一个元素指针

箱子排序（桶排序）

如果我们知道待排序数组的数值范围，那我们可以直接**把值相同的元素放在一起，然后按顺序收集**一下就可以啦。因为我们不知道每个值可能有几个元素，所以我们可以使用链表来存储某个值包含的元素，这样可以节省空间。

时间复杂度$O(size + range)$，空间复杂度$O(range)$，**仅适用于数值范围已知且较小的情况。**可以是**稳定排序**

下面给出的是范围从0到range的处理，还可以针对范围写得更健壮。

```
void binSort(int* arr, int size, int range){
    //初始化
    list<int> *bin = new list<int> [range+1];

    //把元素放到对应的桶里
    for(int i = 0; i < size; ++i){
        //要达到稳定排序，存和取必须从不同端
        bin[ arr[i] ].push_front( arr[i] );
    }

    //收集结果
    size = 0;
    for(int i = 0; i <= range; ++i){
        while(!bin[i].empty()){
            //取
            arr[size++] = bin[i].back();
            bin[i].pop_back();
        }
    }

    delete[] bin;
}
```

基数排序

当待排序数组数值范围很大时，我们不能直接使用桶排序，会消耗很大的空间。

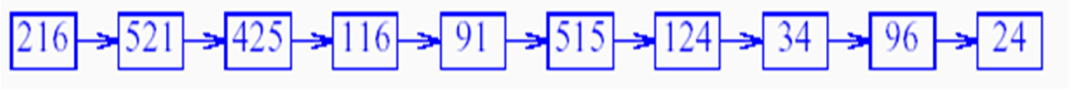
考虑对每个元素，**按照基数r进行分解**，每个元素得到c个r进制数，**依次对元素以它的一个r进制数为标准进行桶排序**，最终可以达到原数组的排序。因为桶排序是稳定排序，对第k位排序后，前k-1位也是有序的，排完c位之后，整个数组就是有序的了。

分解某个数：从低到高，各位依次是

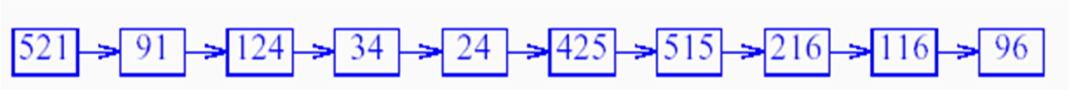
$$x\%r, \frac{x\%r^2}{r}, \frac{x\%r^3}{r^2}, \dots \frac{x\%r^c}{r^{c-1}}$$

十进制数928可以按照基数10分解为数字9，2和8
3725用基数60来进行分解则可以得到1，2和5: $(3725)_{10} = (125)_{60}$

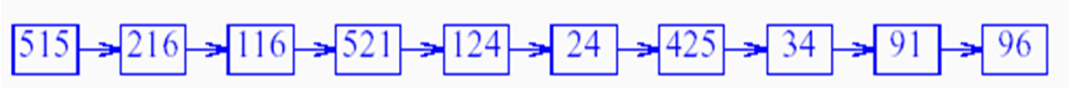
例如对下面的n=10数组，按基数r=10，进行排序：



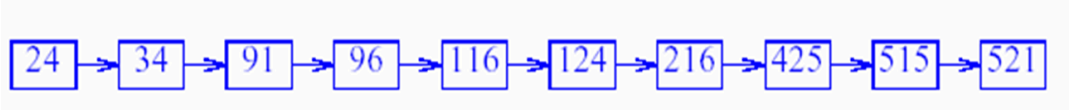
输入链表



按最后一位数字排序后的链表



按倒数第2位数字排序后的链表



按最高位数字排序后的链表

当使用基数r=n对n个介于0到$n^c - 1$范围内的整数进行分解时，每个数将可以分解出c个数字。
因此，可以采用c次箱子排序，每次排序时取range=n。

时间复杂度为$O(cn) = O(n)$（因为c是一个常量）

并查集

代码
解决在线等价类问题

第七章 数组和矩阵

数组的行主映射和列主映射

内存实际上是一维连续的，在存储多维数组时，需要把它映射到一维空间上。有行主映射和列主映射两种方式。

我们先来看二维数组的情况。现在有一个数组 `int a[m][n];`，那么它将占内存中的a ~ a + m*n-1的地址空间，为了简化，我们不考虑数据类型的字节数。需要把每个元素a[i][j]放到一个位置。

- 行主映射：依次安排每一行的元素，映射公式为$map_h(i,j) = n * i + j$。
  也就是说，在内存中按顺序是$a[0][0], a[0][1], \dots a[0][n-1], a[1][0], \dots$
- 列主映射：依次安排每一列的元素，映射公式为$map_l(i,j) = m * j + i$。
  在内存中按顺序是$a[0][0], a[1][0], a[2][0], \dots a[m-1][0], a[0][1], \dots$

再来看多维的情况。我们知道，n维数组可以理解为n-1维数组的**数组**，我们需要先安排n-1维数组，然后再把它们组织起来。现在我们有一个n维数组$a[u_1][u_2]\dots[u_n]$

- 行主映射：
  $map_n(i_1, i_2, \dots i_n) = map_h(i_1, map_{n-1}(i_2, i_3, \dots i_n)) = i_1 * u_2 * u_3 \dots * u_n + map_{n-1}(i_2, i_3, \dots i_n) = \dots = i_1 * u_2 * u_3 \dots * u_n + i_2 * u_3 * u_4 \dots u_n$
- 列主映射：
  $map_n(i_1, i_2, \dots i_n) = map_l(map_{n-1}(i_1, i_2, \dots i_{n-1}), i_n) = i_n * u_1 * u_2 \dots * u_{n-1} + map_{n-1}(i_1, i_2, \dots i_{n-1}) = \dots = i_n * u_1 * u_2 \dots * u_{n-1} + i_{n-1} * u_1$

记忆法：行主映射后面的下标变化引起的变化小，列主映射前面的下标变化引起的变化小。

矩阵

定义和操作

使用二维数组存储数据，支持的操作：

- 转置
- 矩阵加
- 矩阵乘

matrix

特殊矩阵

- 方阵：行数和列数相同的矩阵。
  常用方阵：
    - 对角矩阵:当且仅当i!=j时，有M(i,j) = 0
    - 三对角矩阵:当且仅当|i-j| > 1时，有M(i,j) = 0
    - 下三角矩阵:当且仅当i < j时，有M(i,j) = 0
    - 上三角矩阵:当且仅当i > j时，有M(i,j) = 0
    - 对称矩阵(symmetric):当且仅当对于所有的i和j，有M(i,j) = M(j,i)

因为这些方阵的特殊性质，我们可以采用一些手段节省存储空间，不开m*n的二维数组。

稀疏矩阵

只有少量非零元素的矩阵

我们可以只存储非零元素

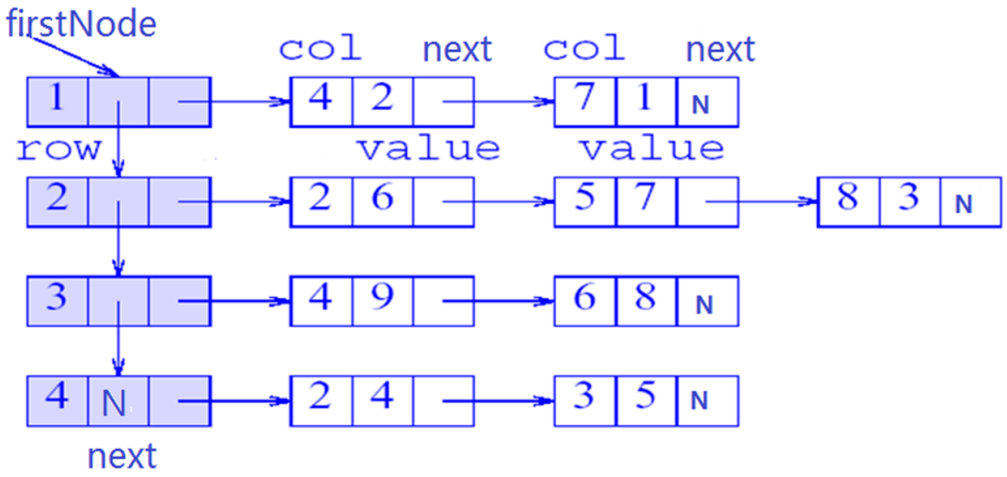
用单个线性表描述(数组存储)

按照行优先，列其次的顺序，把包含元素的对象{col, row, data}存到数组中。

增加元素或查找元素时，二分查找位置。
转置时，需要维护行列顺序，不是简单的把每一项行列交换。
矩阵加时，考虑使用双指针，结果矩阵的元素个数并不能在$O(1)$内得出。

用多个线性表描述（链式存储）

链表数组/链表如图：



第八章 栈

定义和应用

定义：栈（stack）是一个线性表，其插入（也称为添加）和删除操作都在表的**同一端**进行。
其中允许插入和删除的一端被称为栈顶（top），另一端被称为栈底（bottom）

bottom $e_0, e_1, e_2, \ldots, e_i, \ldots, e_{n-2}, e_{n-1}$ top

栈是一个后进先出(LIFO (Last-In, First-Out))表

抽象数据类型

```
template <class T>
class stack{
    public:
        virtual ~stack() {}
        //栈为空时返回true，否则返回false
        virtual bool empty() const = 0;
        //返回栈中元素个数
        virtual int size() const = 0;
        //返回栈顶元素
        virtual T& top() = 0;
        //删除栈顶元素
        virtual void pop() = 0;
```

```
        //将元素theElement压入栈
        virtual void push(const T& theElement) = 0;
}
```

数组描述

一个数组存数据`T *data;`，一个指针指栈顶`int top;`

push: `data[++top] = val;`必要时扩容
pop: `top--;`

链表描述

用单链表，入栈头节点插入，出栈头节点删除。

应用

括号匹配

括号匹配-洛谷
个人题解

汉诺塔*

递归解法
汉诺塔-力扣：传统的
汉诺塔-洛谷：提供了另一个解法思路
新汉诺塔-洛谷：变体，比较难

列车车厢重排

贪心+模拟

开关盒布线*

离线等价类问题*

迷宫老鼠

深度优先搜索

第九章 队列

定义和应用

定义:
队列（queue）是一个线性表，其插入和删除操作分别在表的不同端进行。
添加新元素的那一端被称为队尾(queueBack)。
删除元素的那一端被成为队首(queueFront)。

```
   e0, e1, e2, …, ei, …, en-1
   ↑                      ↑
queueFront             queueBack
```

队列是一个先进先出（ first-in-first-out, FIFO）的线性表。

抽象数据类型

```
template <class T>
class queue{
    public:
        virtual ~queue() {}
        //队列为空时返回true，否则返回false
        virtual bool empty() const = 0;
        //返回队列中元素个数
        virtual int size() const = 0;
        //返回队列头元素;
        virtual T& front() = 0;
        //返回队列尾元素
        virtual T& back() = 0;
        //队列头元素
        virtual void pop() = 0;
        //将元素theElement加入队尾
        virtual void push(const T& theElement) = 0;
}

```

数组描述

注意循环位置映射的处理

链表描述

应用

列车车厢重排

电路布线*

图元识别*

工厂仿真*

第十章 跳表和哈希

我觉得这章叫字典是不是更合适?

字典

也称为Map，C++里常用unordered_map，Java里的HashMap
有序对< key, value >的集合。普通字典要求每个对的key不能相同，一个对的唯一标志是key，通过key查找value。

抽象数据类型

```
template <class K, class E>
class dictionary {
  public:
    virtual ~ dictionary() {}
    virtual bool empty() const = 0;//字典为空时返回true，否则返回false
    virtual int size() const = 0;//返回字典中数对的个数
    virtual pair<const K,E>* find (const K&) const = 0;//返回匹配数对中的指针；
    virtual void erase(const K&) = 0;//删除匹配的数对
    virtual void insert(const pair<const K,E>&) = 0;//在字典中插入一个数对
}
```

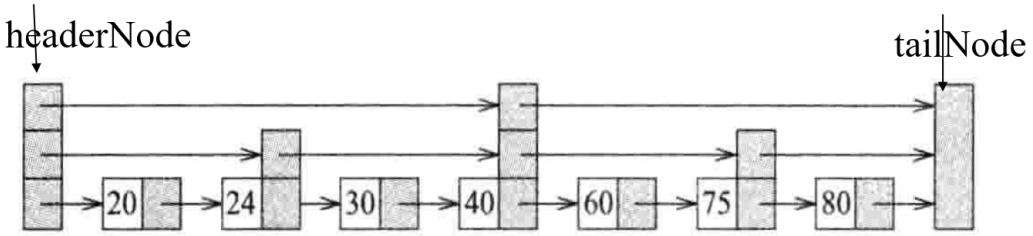
线性表描述

把键值对存储到线性表中。

- 数组：按key单调组织键值对
  - 查找：二分查找$O(logn)$
  - 插入删除：先查找，再移动元素$O(n)$
- 链表：
  - 插入删除查找：$O(n)$

跳表描述*

试图在链表的基础上，做到“二分查找”。



每个节点可以同时属于第0~k级链表，所以需要一个指针数组存放不同级的后继节点`skipNode<K, V> **nexts;`(一个*表示指针，一个*表示数组)

插入删除时，需要维护跳表的性质。

散列表描述

均匀散列函数：映射到一个桶里的关键字大致相同
b=11，[0,98],每个桶里大概9个
b=11，[0,999],每个桶里大概91个
除余散列：f(k) = k % b 能做到均匀散列
良好的散列函数：性能较好的均匀散列函数
除余散列：f(k) = k % D，D的选择对于散列的性能有着重大的影响(D等于桶的个数b)。
如：应用中全是偶数关键字，D也是偶数，则可能会扎堆
当D为素数或D没有小于20的素数因子时，可以使性能达到最佳。

TODO好的Hash

应用*

第十一章 二叉树和其它树

树

二叉树

二叉树的特性

二叉树的描述

二叉树常用操作

二叉树遍历

前序遍历 中序遍历 后序遍历 层次遍历
递归 非递归实现

由中序+任意序可唯一确定一颗二叉树，其它组合不行。

抽象数据类型BinaryTree

类linkedBinaryTree

应用

树的二叉树描述

用二叉树表示一个树

对于树t的每个节点x

- x 节点的 leftChild 指向x的第一个孩子。
- x 节点的 rightChild 指向x的下一个兄弟。

树的遍历：

- 深度优先：
  - 先根遍历
  - 后根遍历
- 广度优先

树的遍历	对应	森林的遍历	对应	二叉树的遍历
先根遍历	->	先序遍历	->	先序遍历
后根遍历	->	中序遍历	->	中序遍历

第十二章 优先队列

定义

优先级队列
优先级队列(priority queue)是0个或多个元素的集合， 每个元素都有一个优先级或值，允许相同。
与FIFO结构的队列不同，优先级队列中元素出队列的顺序由元素的优先级决定。
从优先级队列中删除元素是根据优先级高或低的次序，而不是元素进入队列的次序。
对优先级队列执行的操作有:

- 查找一个元素(top)
- 插入一个新元素(push)
- 删除一个元素(pop)

堆

定义

- 大根树(小根树)：**每个节点**的值都大于(小于)或等于其子节点(如果有的话)值的树。
- 大根堆(小根堆)：即是大根树(小根树)，又是**完全**二叉树。

描述

使用一维数组

- 数据成员：
  - T *heap; // 元素数组
  - int arrayLength; //数组的容量
  - int heapSize; //堆中的元素个数
- 方法：
  - empty()
  - Size()
  - top()
  - pop()
  - push(x)

插入

首先追加到末尾，如果优于父节点，则向上调整。
不必考虑其兄弟节点，因为它>父节点>兄弟节点，所以直接与父节点交换。

时间复杂度 $O(logn)$

删除

删除堆顶，把末尾元素置于堆顶；如果孩子中的较大者大于该节点，则交换，并重复此步。

时间复杂度 $O(logn)$

初始化

给定n个数，建立大根堆。
通过n次插入，复杂度$O(logn)$ 。

$O(n)$ 的初始化：

把原序列作为层次序建立完全二叉树。
从最后一个有子节点的节点开始往前，把以此节点为根的子树调整为大根堆。
调整方式同删除的调整，因为此时其左子树和右子树分别为大根堆。

左高树*

堆是用数组实现的，需要连续空间。如果没有足够的连续空间就需要使用左高树，它是**链式**存储的。

定义

- 外部节点(External node)：代替树中的空子树的节点。
- 内部节点(Internal node)：具有非空子树的节点。
- 扩充二叉树(Extended binary tree)：增加了外部节点的二叉树。
- 对扩充二叉树中的任意节点x，s(x)：从节点x 到它的子树的外部节点的所有路径中最短的一条路径长度。
  - 若x是外部节点，则 s(x) = 0
  - 否则，s(x) = min{ s(L), s(R) } + 1
- **高度优先左高树(HBLT)**：当且仅当一棵二叉树的任何一个内部节点，其左孩子的 s 值大于等于右孩子的 s 值时，该二叉树为高度优先左高树
- 最大HBLT：即同时又是最大树的HBLT

合并两颗最大HBLT

A、B：需要合并的两棵最大HBLT

- A、B其中一个为空，将另一个作为合并的结果；
- A、B均不为空
  - 比较A、B的根元素，较大者为合并后的HBLT的根
  - 设A具有较大的根，A的左子树为AL，A的右子树AR
  - A的右子树AR与B合并的结果：C
  - A与B合并的结果：以A的根为根，AL与C为左右子树的最大HBLT

- 如果L的s值小于C的s值，则C为左子树，否则L为左子树

```
void merge(MaxHBLT *A, MaxHBLT *B){
    if( !A ){
        A = B;
        return;
    };

    //A有较大的根
    if( A->root < B->root ){
        swap(A, B)
    }

    //合并AR与B，s值较大者为左子树
    merge(A->right, B);
    if( A->right.s > A->left.s ){
        swap( A->right, A->left);
    }
}
```

最大HBLT的插入

合并原树与新的单元素最大HBLT

最大HBLT的删除

删掉根节点，合并左右子树。

初始化最大HBLT

1. 创建n 个最大HBLT ，每个树中包含一个元素，这n 棵树排成一个FIFO队列
2. 从队列中依次删除两个最大HBLT ，将其合并，然后再加入队列末尾。
3. 重复第2步，直到最后只有一棵最大HBLT 。

时间复杂性: $O(n)$

堆排序

使用 $O(n)$ 初始化创建大根堆，删除 n 次，得到排序序列。

时间复杂度 $O(nlogn)$

```
    //TODO
```

霍夫曼编码

一种文本压缩算法，使用变长编码对使用到的字符编码，故而没有任何一个代码是另一代码的前缀。

定义

- n扩充二叉树外部节点标记为(1...n)的**加权外部路径长度 (Weighted External Path length)** :

$$WEP = \sum_{i=1}^{n} L(i) \times F(i)$$

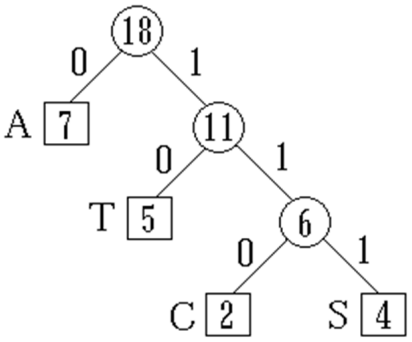
- **L(i)** ： 从根到达外部节点i 的路径长度(即路径的边数);
- **F(i)** ： 外部节点 i 的权值(weight) .
- 如果F(i)是字符串中被编码的字符的频率, WEP 就是**压缩编码串的长度**.
- **霍夫曼树**：对于给定的频率具有**最小加权外部路径长度的二叉树**。

构造霍夫曼树

1. 初始化二叉树集合，每个二叉树含一个外部节点，每个外部节点代表字符串中一个不同的字符。
2. 从集合中选择两棵具有最小权值的二叉树，并把它们合并成一棵新的二叉树。合并方法是把这两棵二叉树分别作为左右子树，然后增加一个新的根节点。新二叉树的权值为两棵子树的权值之和。
3. 重复第2步，直到仅剩下一棵树为止。

获取霍夫曼编码

从根节点开始，左边标0，右边标1（或相反），外部节点（字符）的编码为从根节点到它路径上的数字。



第十四章 搜索树

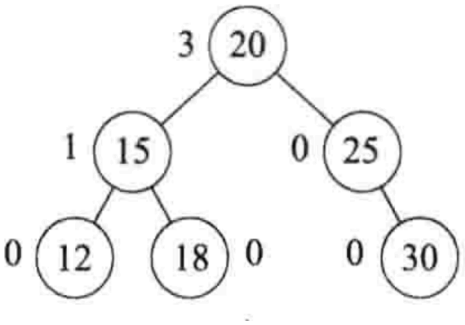
比跳表更优秀的链式存储表示字典。

二叉搜索树

定义

二叉搜索树是一棵可能为空的二叉树，一棵非空的二叉搜索树满足以下特征：

- 每个元素有一个关键字，关键字都是唯一的。
- 根节点左子树的关键字(如果有的话)小于根节点的关键字。
- 根节点右子树的关键字(如果有的话)大于根节点的关键字。
- 根节点的左右子树也都是二叉搜索树。



查找

从根节点开始，

- 如果相等则返回
- 如果目标值大于节点值，则向右子树寻找
- 如果目标值小于节点值，则向左子树寻找
- 如果到达空节点，未找到

操作IndexSearch(k)

IndexSearch(k,e)返回第k个元素

- 如果k=x.LeftSize, 第k个元素是x.element
- 如果k<x.LeftSize, 第k个元素是x的左子树的第k个元素
- 如果k>x.LeftSize, 第k个元素是x的右子树的第(k - x.LeftSize)个元素

插入

在二叉搜索树中插入一个新元素e

- 首先搜索，验证e 的关键值是否存在
- 如果搜索成功，那么新元素将不被插入
- 如果搜索不成功，那么新元素将被插入到搜索的中断点（最后的null节点）

删除

删除一个节点 v ，分三种情况

1. v 是叶子节点，直接删除
2. v 只有一个子节点
    - v 无父节点，即v为根节点，直接删，v 的唯一子节点称为根节点
    - v 有父节点p，删除 v ，其位置放置 v 的唯一子节点

3. v 有两个子节点：找到 v 的左子树中的最大子节点（或右子树中的最小子节点）c ，把 v 的值赋为 c 的值，删除 c 的位置。删除 c 回到1. 或2.，因为 c 最多有一个子，否则不可能为最值。

第十五章 平衡搜索树

AVL搜索树

定义

AVL树：

- 空树是AVL树
- 非空二叉树T，左右子树分别为T_L、T_R，需满足：
  - T_L、T_R都是AVL树
  - $|h_L - h_R| \leq 1$ （树高）

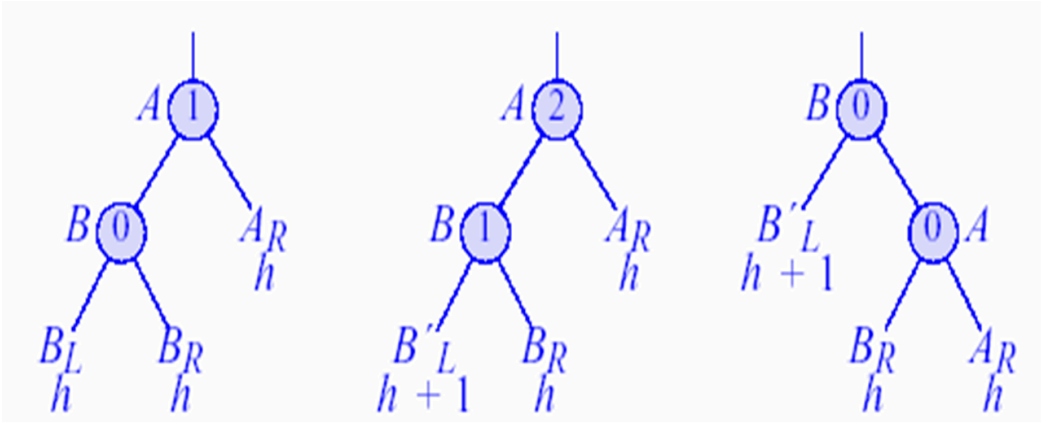
AVL搜索树：满足AVL的二叉搜索树。

平衡因子

节点需要属性平衡因子Balance Factor，简称 BF，为左子树高度减右子树高度。

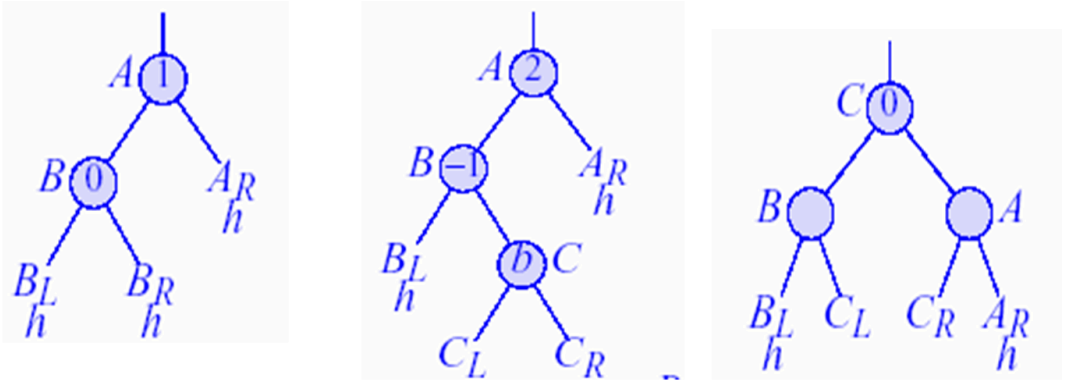
插入

LL



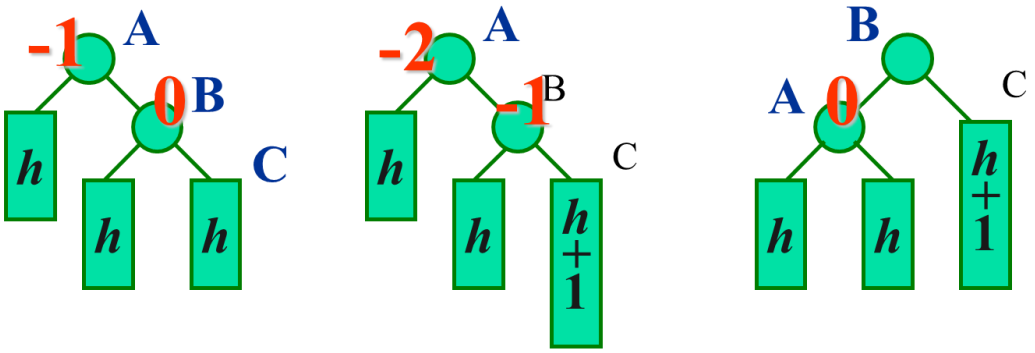
(a) 插入之前　　(b)插入到B_L中之后　　(c) LL旋转后

LR



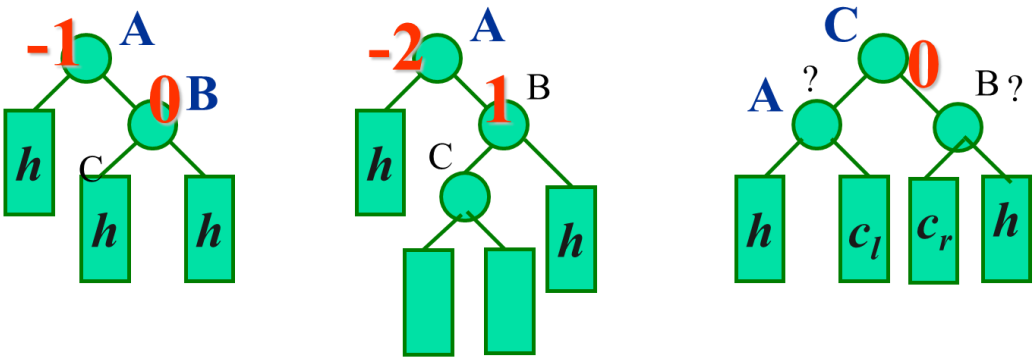
(a)插入之前　　(b)插入到B_R之后　　(c)LR旋转之后

RR



(a)插入前　(b) 插入后　(c)旋转后

RL



(a)插入前　(b) 插入后　(c)旋转后

记忆法：A节点需要向另一侧（平衡侧）退，非平衡侧选一个上到A的位置，从路径上涉及到的离A最近的两个。非平衡侧为左，则选二者中的较大者；非平衡侧为右，则选二者中的较小者；并把其子节点左仍左，右仍右（如果有）。

红黑树*

B-树

第十六章 图

基本概念

应用和更多的概念

特性

抽象数据类型graph

无权图的描述

有权图的描述

类实现

图的遍历

应用

寻找路径

连通图及其构件

生成树

第十七章 贪心

拓扑排序

单源最短路

最小生成树

第十八章 分治

归并排序

快速排序

选择

第十九章 动态规划

01背包

多源最短路

排序

排序算法有两个额外关注的点

- **在线**排序：排序过程中，会依次完成前k个元素的排序，不依赖后续的元素。即排序完成后，再追加一个元素可以继续按原来的算法，而用不重新排序，就把这个元素放到正确的位置。包括：
  - 插入排序
- **稳定**排序：如果一个排序算法能够保持同值元素之间的相对次序，则该算法被称之为稳定排序。如$3_1, 2, 3_2$排序后为$2, 3_1, 3_2$，两个3仍是原来的顺序。包括：
  - 桶排序

> 这些排序**可以**是稳定的，取决于特定步骤

冒泡排序

选择排序

插入排序

名次排序（计数排序）

箱子排序（桶排序）

基数排序

堆排序

归并排序

快速排序