

实用算法设计笔记

笔记本： 科软

创建时间： 2020/12/5 20:00

更新时间： 2021/5/14 20:00

作者： 栗子学姐

- 余艳玮
 - ywyu@ustc.edu.cn
 - 末位淘汰，教学课60/实验课30
 - 成绩
 - 作业 (10%)
 - 实验 (30%)
 - 期中 (10%)
 - 期末 (50%)
 - 教学目标
 - 数据结构
 - 算法
 - 算法解决问题
 - 实现算法并优化
 - 期末考
 - 填空题 / 选择题 (30)
 - 量化
 - 综合分析题 (45)
 - 性能分析：红黑树 / 堆排，进行性能分析
 - 算法设计思路：步骤 / 流程图，具体代码不要求
 - 算法设计题 (25)
 - 推荐算法流程图，有约束（数据结构，算法），**具体代码要求到准确写出数据结构**
 - 题干：参照排序 (3 - 6.3) 的案例
 - 注意：面试可能问到的数据结构
 - Trie树
 - BitMap
 - Bloomfilter
 - 后缀数组（最长子串）
-

0 期末复习

数据结构

- 线性结构：
 - 基础版：线性结构（顺序表，链表，栈和队列）
 - 复合型：散列表
 - 进阶版：位图, BloomFilter, 后缀数组
- 树形结构：
 - 基础版：二叉树，红黑树
 - 进阶版：B树, B+树, Trie树
- 两类算法：
 - 查找：
 - 基于散列表的查找：按内容的查找
 - 可结合位图、BloomFilter这些数据结构来灵活实现
 - 蛮力查找（基于顺序表）
 - 基于有序表的二分查找
 - 字符串的查找：KMP和BM算法，后缀数组

- 树的查找：基于BST、平衡BST（AVL树, 红黑树, 伸展树）、B树、B+树、Trie树的查找
 - 排序：
 - 简单排序：冒泡，选择，插入，希尔
 - 复杂排序：快速排序，堆排序，归并排序，分配排序
 - 算法的性能：
 - 空间方面：内存开销
 - 时间方面：计算速度
 - 综合案例分析
 - 综合案例：
 - 生成指定范围内若干个不重复的随机整数序列（针对不同的数据结构，进行算法的设计并实现）
 - 最长重复的字符串查找问题：基于后缀数组
 - 海量数据的案例分析：
 - 查询：去重，统计
 - 排序：排序，最大（小）的前K
 - 解题策略中的关键问题：
 - 1、存储什么？如何存储一组数据；（选择数据结构）
 - 2、如何高效地解决问题？（设计算法）
 - 代码优化的技巧：组分配空间，消除递归
-

1 导论

- 算法特征
 - 有穷性，确定性，可行性
 - 输入：零个或多个
 - 输入：一个或多个（检验算法准确性）
 - 算法+数据结构=程序
 - 内存分配消耗
 - 申请分配内存空间一次，额外开销为24~31B
 - 申请1~8B，实际分配32B
 - 申请9~16B，实际分配40B
 - 申请17~24B，实际分配48B
 - 尽可能减少申请分配内存空间的次数
 - 一次性手动申请大片内存空间作为内存池
 - 需要内存空间，从内存池取，用完后，返回给内存池
 - 内存池不用时，需手动释放
 - 代码调优原则
 - 减少输入输出，函数调用次数，限制密集型操作（浮点，除法运算）
 - 确定最耗时操作，提高性能（可用测量和跟踪工具，Profiler，AQTime）
-

2 线性表

- 逻辑结构/物理结构
 - 什么时候用数据结构
 - 操作对象为同类型的很多数据
 - 数据间存在某种关系或某些共性操作
 - 什么时候用线性结构
 - 数据之间没有天然的一对多 和 多对多关系
 - 处理数据的顺序有明显的唯一的先后次序关系
 - 若采用线性结构，具体哪种存储结构？
 - 取决于数据处理中 最频繁的操作，静态操作，动态操作？
 - 线性表
 - 逻辑特征：线性特征
 - 存储结构：顺序表和链表
 - 非线性表
 - 树，图，集合

- 顺序表
 - 静态定义
 - 系统自动分配和回收存储空间
 - 比较机械，分配的内存大小固定
 - 动态定义
 - 需手动分配存储空间: malloc(), 可重新分配空间: realloc()
 - 需手动释放所占的空间: free()
 - 避免“机械”，但是会增加时间开销

顺序表——动态定义

1. 数组≠顺序表.
2. 并不是只有链表中才能有指针.

- 顺序表的动态定义：利用指针。

```
#define List_Size 100 /*分配空间的大小*/
typedef struct{
    int *elem;    /*顺序表的存储空间*/
    int len;      /*实际长度*/
    int ListSize; /*当前分配的空间大小*/
} Sqlist;
```

- 顺序表适用于输入数据大小一致，且无太多动态操作的应用问题
- 位图/位向量
 - 用位向量存储数字，一位代表1

```
class IntSetBitVec {
private:
    enum { BITSPERWORD = 32, SHIFT = 5, MASK = 0x1F };
    int n, hi, *x;
    void set(int i) { x[i>>SHIFT] |= (1<<(i & MASK)); }
    void clr(int i) { x[i>>SHIFT] &= ~(1<<(i & MASK)); }
    int test(int i) { return x[i>>SHIFT] & (1<<(i & MASK)); }
public:
    IntSetBitVec(int maxelements, int maxval)
    {
        hi = maxval;
        x = new int[1 + hi/BITSPERWORD];
        for (int i = 0; i < hi; i++)
            clr(i);
        n = 0;
    }
    int size() { return n; }
    void insert(int t)
    {
        if (test(t))
            return;
        set(t);
        n++;
    }
    void report(int *v)
    {
        int j=0;
        for (int i = 0; i < hi; i++)
            if (test(i))
                v[j++] = i;
    }
};
```

作用分别是？

set(i):将整数x的第i比特置为1;
clr(i):将整数x的第i比特置为0;

位向量

顺序表的长度为何这样计算？

1个整数用BITSPERWORD个bit描述。假设位向量中的最大整数为maxval，意味着描述该位向量需要maxval比特来，即需要(1+maxval/BITSPERWORD)个整数;

Q1: New语句执行了几次？

Q2: 位向量的三个基本操作（函数set(),clr(),test()）的功能分别是？

3 算法设计技术

- 程序编译和运行
 - 源代码 -> PE文件
 - 源代码，如.c, .h-> 预编译(#开始的预编译指令) .i-> 编译 .s-> 汇编 .o-> 链接(组装模块)-> PE文件，比如exe/dll
 - 程序的编译和运行
 - 运行PE文件以后？ (OS角度)

- Step1: 创建一个进程。
- 进程的最关键特征: 拥有独立的虚拟地址空间, 即拥有一个虚拟空间 VM 到物理内存的映射关系
- Step2: 装载相应的 PE 文件并执行
- 读取可执行文件头, 并且建立虚拟空间 VM 与可执行文件的映射关系
- 将 CPU 的指令寄存器设置成可执行文件的入口地址, 启动运行
- C 中进程的虚拟内存

1、**栈区 (stack)**: 由编译器自动分配和释放, 存放函数的参数值、局部变量的值等, 甚至函数的调用过程都是用栈来完成。其操作方式类似于数据结构中的栈。

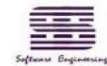
2、**堆区 (heap)**: 一般由程序员手动申请以及释放, 若程序员不释放, 程序结束时可能由 OS 回收。分配方式类似于链表。



3、**全局区 (静态区) (static)**: 全局变量和静态变量的存储是放在一块的, 初始化的全局变量和静态变量在一块区域, 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放空间。

4、**文字常量区**: 常量字符串就是放在这里的。程序结束后由系统释放空间。

5、**程序代码区**: 存放函数体的二进制代码。



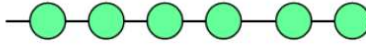
- 递归程序
 - 组成
 - 递归结束条件
 - 递归规则
- 几个重要算法设计技术
 - 保存状态, 避免重复计算
 - 将信息预处理至数据结构中
 - 分治算法
 - 扫描算法
 - 下界: 证明问题的性能下界, 才能确定设计的算法是最佳的

4 栈、队列和哈希表

- 栈
 - 顺序表
 - 表尾插入: 定义表尾为 top
 - 链表
 - 表头插入: 定义表头为 top
 - 若表尾为 top, pop() 后找不到下一个 top
 - esp (栈指针寄存器), ebp (基址指针寄存器) 的移动
 - 应用: 函数调用, undo/redo
 - 什么时候用?
 - 需要跟踪程序当前所在的位置
 - 需要知道什么动作或数据位于当前的动作之前
 - 递归的实现
 - 调用前:
 - 现场保护 (参数、返回地址、旧基址入栈), 被调函数的局部变量的空间分配, 控制转移至被调用的函数入口。
 - 调用后:
 - 保存计算结果, 释放被调函数的数据区, 控制转移回调用处。
- 队列
 - 如何区分队空和队满?

- 1 设标志位
 - 2 引入长度成员
 - 3 少用一个元素空间
 - 循环队列
 - 额外判断队满，队空
 - 链队列
 - 直接利用带头结点的双向链表实现（推荐使用）
 - 引入freelist，减少频繁申请/释放空间的开销，结束后统一回收空间
- 哈希表
 - 适用场景
 - 要求在内存中存储具有线性结构的数据集合
 - 集合中的数据项的数量预先无法确定
 - 要求能快速、近似随机的访问数据项（按值查找）
 - 再散列法
 - 通常认为，**负载因子 $\alpha > 0.5$** 时，再散列将不是一种切实可行的解决方案。

5.1 什么是哈希表

- **Hash表（哈希表）**是一种线性结构。 
 - 有限个数据项组成的序列，记作(a1,a2, ... , an)
- **Hash表（哈希表）**可以建立数据项的关键字和其逻辑存储位置之间的对应关系。即：**HashKey = H (key)**
 - **Key（关键字）**：是数据项（或记录）中某个分量的值，它可以用来标识一个数据元素（或记录）。
 - **主关键字**：唯一标识一个记录的关键字
 - **HashKey（hash键）**：也称为槽。它是关键字的“像”，是关键字在该表中的逻辑存储位置。（必须合法， $\in [0, \text{hash表长}-1]$ ）
 - **h（Hash函数）**：是一个映像，它将一组关键字映像到一个有限的、地址连续的地址区间上。（ $h : \text{Key} \rightarrow \text{HashKey}$ ）。
- **Hash表的构造过程**，是将关键字映像到其逻辑存储位置（或hash键）的过程。
- **冲突**：两个不同的数据项映像到同一个HashKey上。即：**Key1 \neq Key2，但H(Key1)= H(Key2)**。

5 查找

- 应用
 - 编译器查询变量得到类型和地址
 - 拼写检查器查字典
 - 电话号码簿程序查用户名得到号码
 - 因特尔域名服务器查找域名发现IP地址
- 查找特征
 - 准备时间
 - 比如，二分查找须在有序表上才能生效，必须先排序
 - 准备时间可能会超过查找本身节约的时间：顺序查找vs二分查找
 - 运行时间
 - 回溯的需要
 - N皇后问题

基于散列表的查找

- 查找槽号
- 在每个槽的链表上查找

- ELF和PJW哈希函数比较
 - <https://blog.csdn.net/zhangxuri198/article/details/78535577>

bloom filter

- 是bitmap的扩展
- 将一个数据映射为k个bit位
- 可采用counting Bloomfilter实现删除字符串过程
- 缺点：无法恢复所表示的数据集合

蛮力查找

- 在数组末尾放置一个哨兵值SearTarg
- 在循环过程中无需检测是否已到数组末尾。
- 大约加速了5%

二分查找

代码调优

- 利用等价的代数表达式：比如，模运算优化。
- 利用宏替换函数。
- 使用哨兵来合并测试条件
- 展开循环
- 高速缓存需经常处理的数据。

模式匹配算法

- 朴素的模式匹配
 - 字符串：s = "good" strlen(s) = 4, sizeof(s) = 5
 - 需进行回溯
- KMP算法
 - 从左往右字符匹配
 - next[]：下标从1开始

P="ababaaaba"（如表 5-7-5 所示）

表 5-7-5

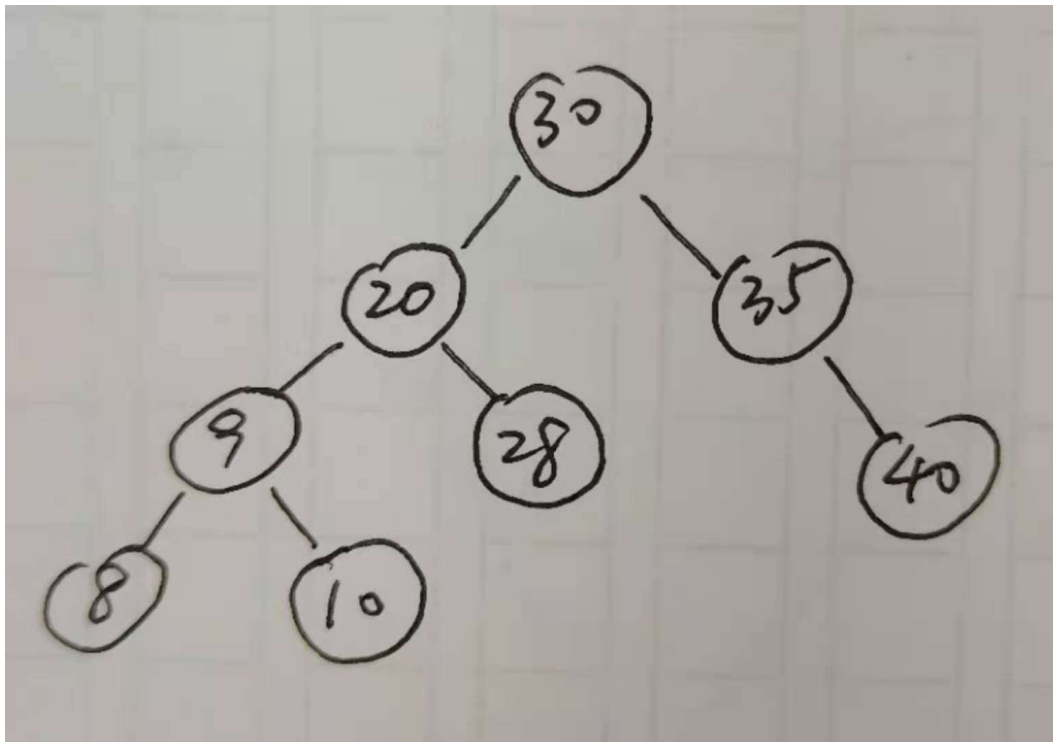
j	123456789
模式串	ababaaaba
next[j]	011234223
nextval[j]	010104210

- nextval[]：减少字符比对次数

- `nextval[1]=0;`
- `for(j>1;j<=n;j++)`
 - 若 $P[j]=P[next[j]]$, 则 $nextval[j]=nextval[next[j]]$;
 - 若 $P[j]\neq P[next[j]]$, 则 $nextval[j]=next[j]$;
- BM算法 Boyer-Moore
 - 从右往左字符匹配
 - 坏字符
 - 好后缀 https://blog.csdn.net/qq_21201267/article/details/92799488

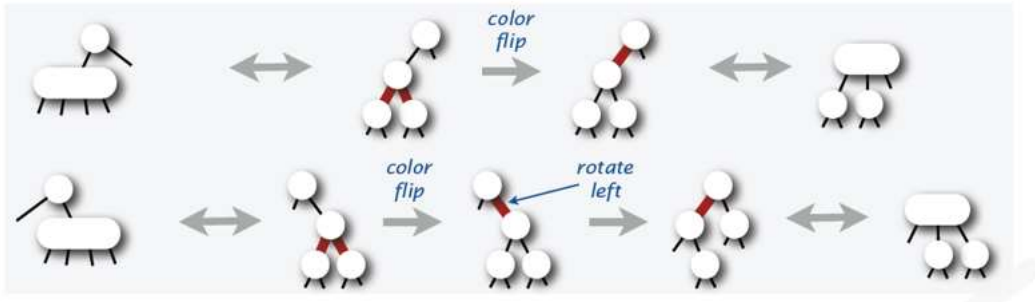
基于树的查找

- 二叉搜索树
 - 删除节点：用中序遍历的直接前驱替换
- 二叉平衡树AVL
 - 插入节点：若不平衡，**寻找最近的不平衡节点**（更低层双亲节点）
 - 练习：从空树出发构造AVL树，待插的关键字序列为10,20,30,40,35,28,8,9



- 2-3-4树
 - 4-节点，分裂，利用颜色翻转和旋转来等价实现
 - 颜色翻转：两个子节点是红色
 - 旋转：连续两条链为红色

2-3-4树中，双亲节点为2-节点时，4-节点的分裂：
在对应的二叉查找树中，可利用颜色翻转和旋转来等价实现

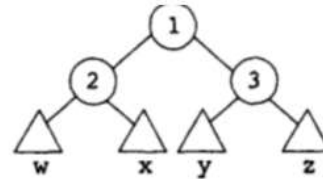
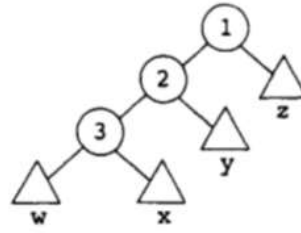
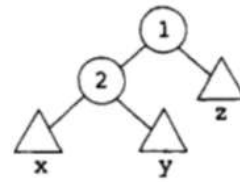


- 基于2-3-4树的红黑树
 - 新插入节点带红色链
 - 根节点两个子节点都为红色，变为黑色
 - 若仅右子节点为红色链，向左翻转
 - 从根开始往下走，未经过两个子节点都为红色链，可暂时不颜色翻转
 - 先颜色翻转，再右子红色链向左翻转
- 红黑树（《程序员实用算法》）
 - 所有数据都在叶子节点
 - 红色链用于绑定”内部节点
- 伸展树
 - 每次访问树，访问过的节点旋转到根
 - 局部性原理
 - 插入操作
 - 先插入，再旋转
 - 删除操作
 - 先旋转
 - 若n的左孩子非空，则用n的中序遍历的直接前驱替换n；
 - 若n的左孩子为空，则直接用n的右孩子替换n
 - 自底向上
 1. 若访问的节点c没有祖父节点，则：直接在c与其父节点p之间进行旋转。
 2. 若访问的节点c，其父节点p，其祖父节点g三者之间的相关位置呈LL或RR型，则：先在p和g间旋转，再在p和c间旋转。（由上至下）
 3. 若访问的节点，其父节点，其祖父节点三者之间的相关位置呈LR或RL型，则：先在c和p间旋转，再在c和g间旋转。（由下至上）

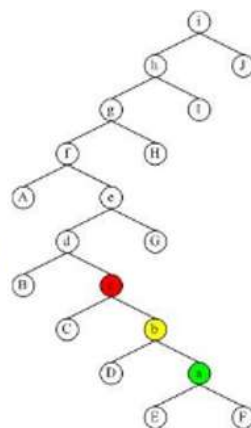

```

graph TD
    2((2)) --- x((x))
    2 --- 1((1))
    1 --- y((y))
    1 --- z((z))

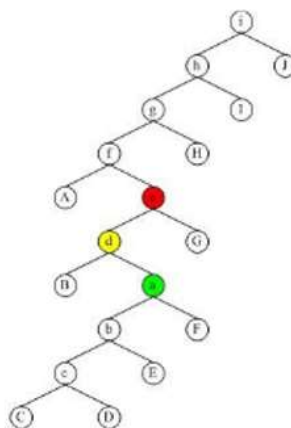
```



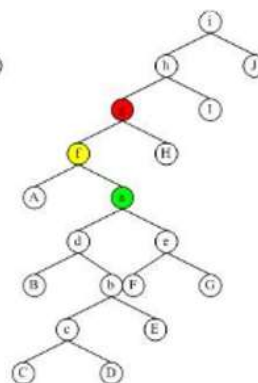
示例



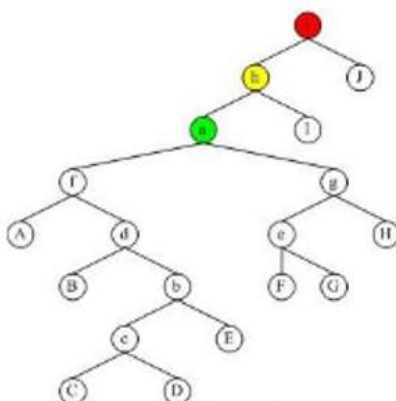
一: X 、 $P(X)$ 同为右孩子情况: 先左旋 $G(X)$ 再左旋 $P(X)$



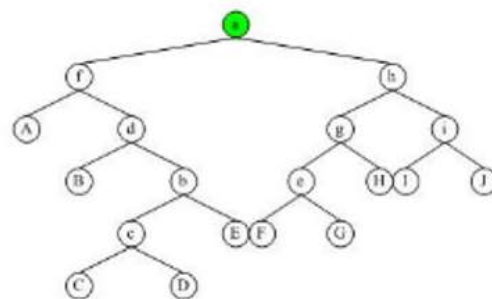
二: X 为右孩子, $P(X)$ 为左孩子情况: 先左旋 $P(X)$ 再右旋 $P(X)$



三: X为右孩子, P(X)为左孩子情况: 先左旋P(X)再右旋P(X)



四: X 、 $P(X)$ 同为左孩子情况: 先右旋 $G(X)$ 再右旋 $P(X)$



五、结果

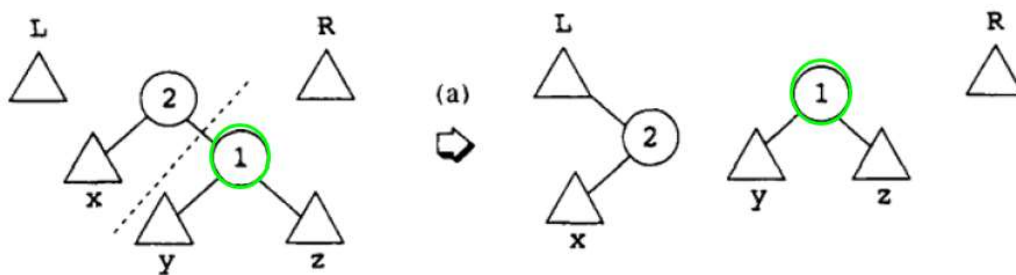
- 自顶向下
 - 每次分裂操作会将当前树分为左，中，右三个片断。
 - 左片断为中间片断的前驱，右片断为中间片断的后继；其中，前驱、后继是指中序遍历顺序。
 - 左，右片断分别插入到左右存储树中。

- **左连接点**：指左存储树中插入新片断的位置，指向左存储树中最右下角的节点。
- **右连接点**：右存储树中插入新片断的位置，指向右存储树中最左下角的节点。

重组操作：（重组前后，中序遍历的顺序是不变的，都为 $L, (y, 1, z), R$ ）

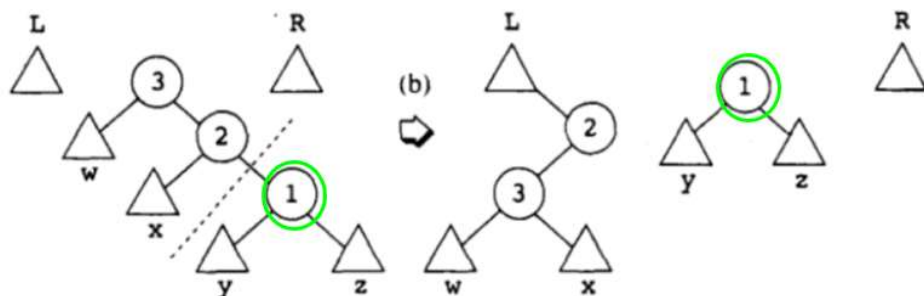


• R型“分裂”操作：



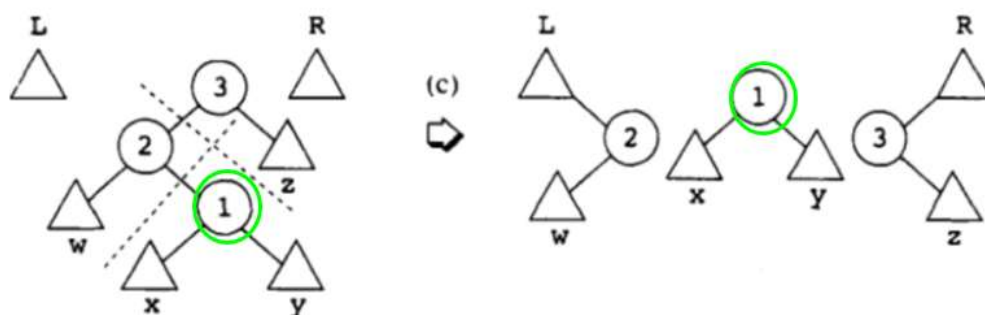
查找“1”

• RR型“分裂”操作：先进行旋转，转化为R型“分裂”操作



查找“1”

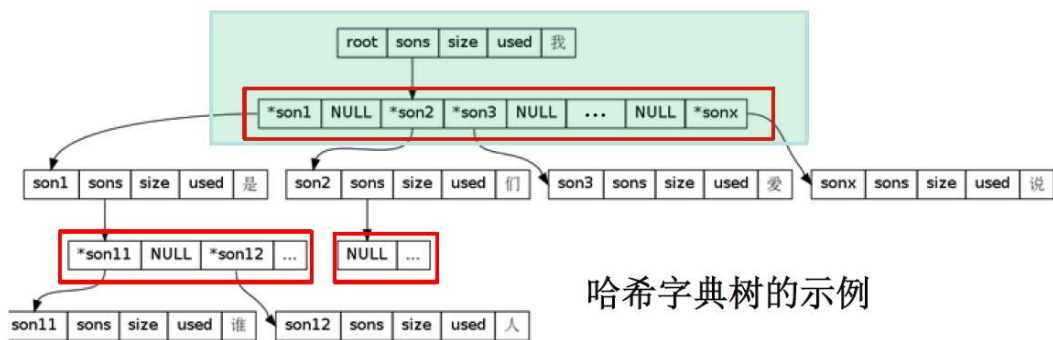
LR型“分裂”操作：



查找“1”

- B树
 - 键只出现1次

- 查询效率与键在树中的位置有关
 - 应用场景
 - 数据库: Sqlserver ,Oracle及Sysbase
- B+树
 - 键可能出现多次, 且一定会出现在叶节点
 - 插入或删除的位置必定在叶子节点
 - 查询效率为常数
 - 应用场景
 - 数据库: mysql, Berkeley DB ,sqlite
 - 支持精确检索和模糊检索: 使用操作<、<=、=、>=的比较
- B*树
 - 内部节点指向下一个节点
- Trie树
 - 字典记录



6 排序

- 基本特征
 - 稳定性: 相同的数顺序不变
 - 哨兵: 确定key不方便, 一般不用
 - 数据结构: 通常数组, 链表用于插入和快速排序
 - 额外空间
- 应用
 - 排序数字, 单词, 热点词汇

简单排序

冒泡排序、选择排序、插入排序、希尔排序

- 冒泡排序
 - 移动最大的记录, 使之更接近尾部
- 选择排序
 - 从待排记录中选择最小的, 放到已排序记录的后面
- 直接插入排序
 - 逐个处理待排序的记录
 - 用移位操作代替交换 (一次移位相当于1/3次交换)
- 希尔插入排序
 - $T(n)=O(n^{1.25})$

排序方式	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(n^{1.3})$			$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	较复杂
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定	较复杂
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定	较复杂
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定	较复杂

复杂排序

快速排序、堆排序、归并排序、分配排序

- 快排
 - 方法1
 - m 记录 \leq 基准点下标结尾， i 遍历数组
 - 方法2
 - 双指针： $i \geq$ 基准点时右移， $j \leq$ 基准点时左移，尽量减少递归高度
 - 用额外空间，变成树状结构，每次基准点为树根
 - 方法3
 - 双向划分& 将随机选择的元素作为基准。
 - 对于小的子数组，采用插入排序方法
 - 用内联代码来替换函数调用：展开循环体中`swap`函数的代码。
- 堆排
 - siftup
 - 当在堆的尾部插入新元素后，需重新获取堆性质。
 -
 - siftdown
 - 用新元素替换堆中的旧根后，需重新获取堆性质。
 -
 - 升序排序
 - 基于数组序列，建立大根堆
 - 依次提取根节点，放入待排序数组的最后，调整大根堆