

定义都很自然且高中知识居多, 因此整理不会像树那么细, 简介算法为主

## 7 图

### 7.1 定义细节

- DAG (directed acyclic graph) : 有向无环图
- 无向图中两节点间有平行边不算环, (路径长度  $\geq 3$ , 环=路径, 不等式秒了)
  - 有向图两条边可以构成环
- 有根图 : 路径能到其他所有点
- 强连通 : 双方互有有向路径联通
  - 强连通分量 : 非强连通图的极大强连通子图
- 网络 : 带权的连通图
- adjacency matrix : 邻接矩阵
- 稀疏因子,  $\delta < 5$  认为是稀疏矩阵

$$\delta = \frac{\text{非0元素个数}(t)}{\text{矩阵大小}(m \times n)}$$

### 7.2 储存

- 邻接表节省时间空间
  - 无向图的邻接表表示, 同一边出现两次即可
  - 有向图也可以有逆邻接表

#### 十字链表

- 每一弧 : 头, 尾, 下一条共尾弧, 下一条共头弧, 弧权值(info域)
- 每一顶点 : data域, 第一条以该顶点为终点的弧, 第一条以该顶点为始点的弧

稀疏矩阵的十字链表 : 行列指针序列 + 每个非0结点(值, 行后继, 列后继)

稀疏矩阵相乘 : 遍历A的行B的列即可.

### 7.3 遍历

实质上是节点的遍历

- 解决回路和非连通图问题 : 标志位

#### dfs, bfs

- 邻接表表示 : 有向图  $\Theta(n + e)$ , 无向图  $\Theta(n + 2e)$
- 相邻矩阵表示 :  $\Theta(n^2)$

#### 拓扑排序

- 对 DAG 不断删除 0 度边入队
- 邻接表表示 :  $\Theta(n + e)$
- 相邻矩阵表示 :  $\Theta(n^2)$

图算法需要考虑的问题: 有向无向, 回路, 连通性, 权值正负

### 7.4 算法

#### Dijkstra单源最短路径

分成已知最短路径和未知最短路径两组, 按长度递增将第二组节点逐个加入第一组

- 最坏  $\Theta((|V| + |E|) \log |E|)$ , 但要具体分析, 维护优先队列只有找最小值和松弛两个操作.
- 不支持负权值, 支持负权值需要 Bellman-Ford 算法或者 SPFA 算法

# Floyd算法求所有最短路径

实际上就是动态规划, 三层 for 循环即可

$$d[i][j] = \min\{d[i][j], d[i][k] + d[k][j]\}$$

- 复杂度  $\Theta(n^3)$

# Prim算法求最小生成树

- MST (minimum-cost spanning tree)

框架与 Dijkstra 算法相同, 但距离值直接用最小边, 总时间  $O(n^2)$

- 适合稠密图, 对稀疏图可以像 Dijkstra 算法那样用堆来保存距离值.

# Kruskal算法求最小生成树

边排序然后逐个入队, 遇到破坏连通性的就跳过  
最坏情况是  $\Theta(e \log e)$ , 通常代价是  $\Theta(n \log e)$

# 8 内排序

- 内排序: 整个排序过程在内存中完成
- 稳定性: 存在多个具有相同排序码的记录, 排序后这些记录的相对次序保持不变

# 算法一览

排序种类	名称	思路	时间代价			空间代价	稳定性	适用于	备注
			最优	最坏	平均				
插入排序	直接插入	每次连续地把a[i]向前插, 顺序遍历完即可	n	n^2	n^2	1	稳定	短序列, 已经有序的序列	
	shell排序	选取间隔序列, 每次对间隔为n的子序列直接插入排序, 直到间隔为1	Hibbard增量序列 n^(3/2), 可进一步减少	n^2	增量除2递减: n^2	1	不稳定		
选择排序	直接选择	每次直接选最小的	n^2	n^2	n^2	1	不稳定		
	堆排序	利用最小堆实现输出, 一次建堆(n), n次删除堆顶(log n)	nlogn	nlogn	nlogn	1	不稳定		
交换排序	冒泡排序	起点从0到n-1遍历, 每次遍历地交换相邻元素, 假如某次冒泡没有发生交换就表明已完成排序	n	n^2	n^2	1	稳定		
	快速排序	分治算法, 选择轴值pivot并划分比之大和比之小的子序列	nlogn	n^2	nlogn	时间代价的1/n	不稳定		划分方法采用双指针实现的所谓分割函数
归并排序	归并排序	分治算法, 直接分成两个子序列, 排序完后用双指针归并	nlogn	nlogn	nlogn	n	稳定		
	R.Sedgewick 优化	小序列改成插入排序, 归并时对右序列颠倒	同上	同上	同上	同上			
分配排序和基数排序	桶排序	将相同值的元素放一起, 然后按编号依次从桶中取出	n+m	n+m	n+m	n+m	稳定	m<<r	m是值的数量
	基数排序	将值拆分为多个部分进行桶排序	d(n+r)	d(n+r)	d(n+r) (实质还是nlogn)		稳定		r个计数器, d次桶排序, 用静态链实现无需移动元素

- 索引排序
- 对时间复杂度的理解
  - 序列平均有  $\frac{1}{2} \binom{n}{2}$  对逆序
    - 任何一种只对相邻记录进行比较的排序算法的平均时间代价都是  $\Theta(n^2)$
  - n 很小或基本有序时插入排序比较有效
  - Shell 排序选择增量以3的倍数递减
    - 需要保证最后一趟增量为1
  - 综合性能快速排序最佳
  - 排序问题的时间复杂度下限, 也即判定树的最小深度:  $\Omega(\log n!) = \Omega(n \log n)$

# 9 文件与外排序

官方复习大纲中只有置换选择排序和多路归并两个板块, 因此笔记只记这些. 内外存和磁盘相关知识请移步 ics.

## 9.1 置换选择排序

### 思路

- 首先传入  $m$  个元素进内存建堆, 设置堆尾标志  $last = m - 1$ , 然后只要  $last \geq 0$ :
  - 堆的根节点传送到输出缓冲区, 记为  $mval$
  - 从输入缓冲区读入一个数  $r$ :
    - 若  $r \geq mval$ , 则把  $r$  放到根节点
    - 否则  $last$  位置的元素放到根节点,  $last-- = 1$ ,
  - 刷新堆以获得新的  $mval$
- 算法结束后, 内存中也填满了未能处理的元素, 直接建堆等待下一顺串处理

### 分析

- 输出的一个顺串最小长度是  $m$ , 最优长度为整个文件(正序输入), 平均情况为  $2m$

## 9.2 多路归并

- 对于每趟需要归并的  $k$  个顺串, 每步就是  $k$  个指针扫描, 直到全部  $n$  个顺串合并完成.
- 合并趟数:  $\lceil \log_k m \rceil$
- 优化:
  - 创建尽可能大的初始顺串
  - 把初始顺串长度作为权, 转化为Huffman树最优化问题, 得到一个  $k$  叉Huffman 树, 称为**最佳归并树**

进一步优化? 每次比较  $k - 1$  次开销还是太大了, 实际上一些比较可以保留

### 赢者树

- 完全二叉树结构, 每个叶子节点表示待归并顺串上的当前元素, 每个内部节点储存胜者(最小值)所在顺串的序号.
- 更新时只需输出对应叶子节点元素, 写入新元素后调整从叶子节点到根节点的路径, 时间复杂度为  $O(\log k)$ .

### 败者树

- 胜者树的优化, 没有本质区别, 但在每个节点保留败者所在顺串的序号, 以及增加一个根节点储存最终胜者
- 也即优化了重构过程, 直接找父节点即可, 无需再与兄弟节点比较(降低复杂度的常数)

### 时间复杂度

- 原始方法:  $O(nk)$
- 败者树方法:  $O(k + n \log k)$

为什么不用堆?

堆的每层调整都要对三个值(父节点、两个子节点)进行至少两次比较, 使用胜者败者树少一次比较, 败者树相对胜者树更少一次寻址

# 10 检索

## 10.1 定义

- 平均检索长度 (ASL)

$$ASL = \sum_{i=1}^n P_i C_i$$

注意概率分布的隐含条件  $\sum p = 1$

## 线性表检索

### "监视哨"顺序检索算法

- 即为在待检索的数组末尾加上  $a[n] = key$ , 以节省遍历时的数组越界检查, 复杂度还是  $O(n)$
- 性能分析: (假设检索成功率是  $p$ )

$$\frac{n+1}{2} < ASL = \frac{n+1}{2}p + (n+1)(1-p) < n+1$$

- 优点: 插入元素可以直接加在末尾
- 缺点: 检索时间太长  $\Theta(n)$

### 二分检索法

- 就是二分法, 需要先对数据排序
- 成功的平均检索长度

$$ASL_{succ} = \frac{1}{n} \sum_{i=1}^j i \cdot 2^{i-1} = \frac{n+1}{n} \log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

- 最大检索长度  $\lceil \log_1(n+1) \rceil$ , 失败检索长度  $\lceil \log_2(n+1) \rceil$  或  $\lfloor \log_2(n+1) \rfloor$
- 优点: 平均与最大检索长度相近, 检索速度快
- 缺点: 要排序, 顺序存储, 不易更新(插/删)

### 分块检索

- 按块有序, 块内无序, 是顺序与二分法的折衷
- $n$  个元素,  $b$  个块, 每个块至多  $s$  个元素

$$ASL = ASL_b + ASL_w \\ \approx \log_2(b+1) - 1 + \frac{s+1}{2}$$

- 当大量插入/删除时, 或结点分布不均匀时, 速度下降

## 集合检索

- 对于密集型集合, 用位向量表示,  $\{0, 1\}$  表示是否有这个元素
  - 用 `ulong` 存, 每个存32个位
  - 位运算模拟集合交并

## 散列表

- **负载因子**  $\alpha =$  填入表中的结点  $n$  / 散列表空间大小  $m$
- **冲突**: 不同关键码计算出了相同的散列地址
  - 现实中不冲突的散列函数极少存在
  - **同义词**: 产生冲突的两个关键码

## 10.2 散列函数选取

$$Address = Hash(Key)$$

- 运算简单, 值域在表长范围内, 尽可能单射

### 常用散列函数

- **除余法**:  $hash(key) = key/M$ , 其中  $M$  是一个接近散列长度的质数
  - 缺点: 连续的关键码映射成连续的散列值, 占据连续数组单元, 可能导致性能降低
- **乘余取整法**:  $hash(key) = \lfloor n\{Akey\} \rfloor$ , 其中  $A \in (0, 1)$ 
  - 若地址空间为  $p$ , 就取  $n = 2^p$ , 则地址恰是小数点后  $p$  位
  - $A$  可以取任何值, 与待排序的数据特征有关, 一般取黄金分割最理想
- **平方取中法**: 先求平方, 再取其中几位或其组合作为地址
- **数字分析法**: 对  $n$  个  $d$  位数, 每一位可能有  $r$  种不同的符号, 在各位上出现频率不一定相同. 可根据散列表大小, 选取期中分布均匀的若干位作为地址.
  - 这一方法完全依赖于关键码集合的性质

各位数字中符号分布的均匀度  $\lambda_k$

$$\lambda_k = \sum_{i=1}^r (\alpha_i^k - n/r)^2$$

- $\alpha_i^k$ : 第  $i$  个符号在第  $k$  位上出现的次数
- $n/r$ : 各种符号在  $n$  个数中均匀出现的期望值
- $\lambda_k$  越小, 说明越均匀

- **基数转换法**: 把关键码看成是另一进制上的数, 再把它转换成原来进制上的数, 取其中若干位作为散列地址
  - 一般取大于原来基数的数, 且互素
- **折叠法**: 将关键码分割成几部分, 将其叠加 (舍去进位) 作为散列地址
  - 移位叠加: 把各部分的最后一位对齐相加
  - 分界叠加: 各部分不断折, 来回折叠对齐相加
  - 适用于长关键码
- **ELFhash字符串散列函数**:
  - 长字符串和短字符串都很有效
  - 字符串中每个字符都有同样的作用
  - 对于散列表中的位置不可能产生不均匀的分布

```
int ELFhash(char* key) {
    unsigned long h = 0;
    while(*key) {
        h = (h << 4) + *key++;
        unsigned long g = h & 0xF0000000L;
        if (g) h ^= g >> 24;
        h &= ~g;
    }
    return h % M;
}
```

## 10.3 解决冲突

### 开散列方法

**拉链法**: 适用于内存

- 把散列表中的每个槽定义为一个链表的表头, 散列到特定槽的所有记录都放到这个槽的链表中
  - 同义词表可以根据输入, 访问频率, 值的顺序组织
- 优点: 简单, 适用于大量数据, 可取  $\alpha \geq 1$ , 结点较大时效果好
- 缺点: 在磁盘上会产生大量访问开销

**桶式散列**: 适用于磁盘

没啥意思, 感觉不会考

### 闭散列方法

- $d_0 = h(K)$  称为**基地址**, 当冲突发生时, 使用某种方法为关键码  $K$  产生一个新的散列地址序列, 称为**探查序列**
  - $d_i = d_0 + p(K, i), i \in [M - 1]$ , 其中  $p$  是探查函数
- 检索要遵循插入时同样的探查序列
  - 重复冲突解决过程
  - 找出在基位置没有找到的记录
- 插入和检索函数都假定每个关键码的探查序列中欧至少有一个存储位置是空的
  - 也可以限制探查序列长度

**线性探查**

- $p(K, i) = i$
- 产生的问题: "聚集", 小聚集汇合成大聚集, 导致很长的探查序列
- 改进:  $p(K, i) = i * c$ , 但还是会有探查序列纠缠

**二次探查**

- $p(K, 2i - 1) = i^2, p(K, 2i) = -i^2$

#### 伪随机数序列探查

- $p(K, i) = R(i - 1)$ , 其中  $R$  是一个值从  $[M - 1]$  的伪随机数序列
- 二次探查和伪随机数探查都能避免基本聚集
- **二级聚集**：两个关键码散列到同一个基地址, 还是得到同样的探查序列, 所产生的聚集
  - 避免方法：探查序列是原来关键码值的函数, 而不仅仅是基地址的函数

#### 双散列探查法

- $p(K, i) = i \cdot h_2(K)$ , 其中  $h_2(K)$  是第二个散列函数
- $h_2(K)$  必须与  $M$  互素
  - 使发生冲突的同义词地址均匀地分布在表中
  - 否则可能造成同义词地址的循环计算
- 优点：无聚集, 无循环, 无二级聚集
- 缺点：散列函数计算量大

## 10.4 散列字典实现

用散列表实现的字典

#### 插入

- 若表中基地址空间未被占用, 则插入记录
- 若基地址的值与关键码相等, 则报告重复 (不允许重复记录)
- 否则按设定的处理冲突方法去找下一个地址, 重复上述过程

#### 检索

- 若基地址中的值为空, 则找不到
- 地址中的值与关键码相等, 则找到
- 否则按设定的处理冲突方法去找下一个地址, 重复上述过程

#### 删除

两点需要考虑:

- 删除后, 该地址的记录是否会影响检索
- 释放存储位置能为将来所用
  - 只有开散列方法可以真正删除, 空间重用
  - 闭散列方法都只能作标记, 不能真正删除, 空间未再次分配之前不可用
- **墓碑**：用一个特殊的值表示删除
  - 不能将空单元与删除单元混淆：检索时遇到已删除标记还需要继续检索下去
- 删除操作与检索类似, 但是删除后地址标记为删除
- 带墓碑的插入：可以插, 但是检索要继续, 防止插入两个相同的关键码

# 11 索引

## 11.1 定义

- 主码：数据库中每条记录的唯一标识
- 辅码：数据库中可出现重复值的码 (属性)
  - 辅码索引把一个辅码值与具有这个辅码值的多条记录的主码值关联起来
  - 多数检索都是利用辅码索引完成的
- 索引：(key, pointer)
- 索引文件：用于记录这种联系的文件
  - 稠密索引：对每个记录建立一个索引项
    - 主文件不需按关键码次序排列
  - 稀疏索引：对一组记录建立一个索引
    - 主文件必须按照关键码次序存放

## 线性索引

按照索引码值的顺序进行排序的文件

- 优缺点
  - 可对变长的数据库记录访问
  - 支持对数据的高效检索
    - 二分检索
  - 线性索引太大, 存储在磁盘中
    - 一次检索可能多次访问磁盘, 影响检索的效率
    - 使用二级线性索引

## 静态索引

- 索引结构在文件创建时生成
- 一旦生成就固定下来, 在系统运行(例如插入和删除记录)过程中, 索引结构不做改变
- 只有当文件再组织时才允许改变索引结构

多分树 : 大大减少访问外存的次数

## 倒排索引

- 按属性值建立索引, 索引表中的每一项包括 (attr, ptrList):
- 一个属性值
- 具有该属性值的各键码或者记录地址
- 优缺点
  - 支持基于属性的高效检索
  - 花费储存倒排表的代价, 降低更新的效率

## 11.2 动态索引 (B树)

动态索引 : 文件创建时生成, 结构随着插入, 删除等操作而改变, 以保持最佳的检索性能, 例如B树和B+树

### B树

#### B树的定义

$m$  阶  $B$  树是一棵  $m$  路查找树, 或者为空, 或者:

- 下界 : 根节点至少有两个子树, 其他非叶节点至少有  $\lceil m/2 \rceil$  个子树
- 上界 : 每个节点至多有  $m$  个子树
- 有  $k$  个子树的节点有  $k - 1$  个键码
- 所有叶节点都在同一层, 有  $\lceil m/2 \rceil - 1$  到  $m - 1$  个键码

$B$  树的一个包含  $j$  个键码,  $j + 1$  个指针的节点的一般形式为:

- $P_0, K_1, P_1, K_2, P_2, \dots, K_j, P_j$ 
  - $K_i$  是键码值,  $K_1 < K_2 < \dots < K_j$
  - $P_i$  是指向包括  $K_i$  到  $K_{i+1}$  的子树的指针

每个键码对应一个记录指针域

#### B树的特点

- 访问局部性
- 高度平衡
- 父节点键码值是子节点的分界
- 节点键码至少一定比例是满的
  - 改进空间利用率, 减少检索, 更新操作的I/O次数

**B树的查找** : 从根节点开始, 逐层查找, 如果指向空节点, 则查找失败

**B树的检索长度** : 假如高度是  $h$ , 则自顶向下检索到叶节点的过程可能需要  $h$  次读盘, 最多  $h + 1$  访外

#### B树的插入

- 找到最底层插入
- 若溢出, 分裂节点, 中间关键码连同新指针插入父节点
- 若父节点溢出, 继续分裂, 可能一直到根节点(则树升高一层)

节点分裂方法, 假如插入后节点状态为  $p = (m, A_0, K_1, A_1, \dots, K_m, A_m)$ , 则拆分成两个节点  $p = (\lceil m/2 \rceil - 1, A_0, K_1, A_1, \dots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$  和  $q = (m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \dots, K_m, A_m)$ , 中间关键码  $K_{\lceil m/2 \rceil}$  与指向新节点  $q$  的指针插入父节点

## B树的删除

- 删除的关键码在叶节点
  - 若删除后节点关键码数小于  $\lceil m/2 \rceil - 1$ 
    - 若兄弟节点关键码数大于  $\lceil m/2 \rceil - 1$ , 则从兄弟节点借关键码 (父节点分界关键码也要调整)
    - 若兄弟节点关键码数等于  $\lceil m/2 \rceil - 1$ , 则合并节点
  - 若删除后节点关键码数不小于  $\lceil m/2 \rceil - 1$ , 则直接删除
- 删除的关键码不在叶节点层
  - 先把此关键码替换为其后继关键码, 再删除该关键码

## B+树

B树的变种, 所有关键码都在叶节点, 各层节点中的关键码是下一层相应节点中最大关键码(或最小关键码)的拷贝

### B+树的特点

- 每个节点子节点数量在  $[\lceil m/2 \rceil, m]$ , 根节点可以只有两个子节点 (空, 独根除外) 这一点与B树相同
- B+树和B树的差异
  - B+树中  $n$  棵子树的结点中含有  $n$  个关键码, 而B树中  $n$  棵子树的结点中含有  $n - 1$  个关键码
  - B+树叶子结点包含了完整的索引的信息, 而B树所有结点共同构成全部索引信息
  - B+树所有的非叶结点可以看成是高层索引, 结点中仅含有其子树中最大(或最小)关键码

### B+树的查找: 逐层找

- b+树叶节点一般链接起来, 形成双链表, 便于范围查询, 需要的话每一层节点也都可以这样链接

**B+树的插入:** 与B树类似, 左  $\lceil m/2 \rceil$ , 右  $\lfloor m/2 \rfloor$ , 注意上层关键码的维护

两棵树分裂时都是左节点  $\lceil m/2 \rceil$  个子节点

**B+树的删除:** 与B树类似, 但被删除关键码的上层副本可以选择保留, 作为"分界关键码"存在

## 性能分析

m 阶 B树, 关键码为  $N$  个, 内部节点数为  $p$ , 则存取次数  $k$  满足

$$k \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left( \frac{N+1}{2} \right)$$

插入一个关键码的平均分裂节点次数为

$$s = \frac{p-1}{N-1} \leq \frac{N-1}{(\lceil m/2 \rceil - 1)(N-1)} = \frac{1}{\lceil m/2 \rceil - 1}$$

## 11.3 红黑树

### 定义

BST的染色版:

- 每个节点要么是红色, 要么是黑色
- 根节点, 叶节点是黑色
- 父子节点不能同时为红色
- 任意结点到叶节点的路径上黑色结点数相同

### 节点的阶

- 即从该节点到叶节点的黑色节点数, 不包括该节点本身, 包括叶节点



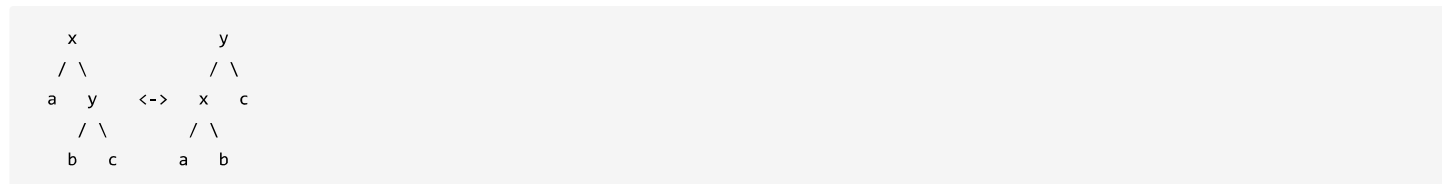
- 叶节点的阶为0, 根节点的阶为该树的阶

### 红黑树的性质

- 满二叉树 : 空树叶也看作节点
- $k$  阶红黑树树高  $\in [k + 1, 2k + 1]$
- $k$  阶红黑树内部节点数最少是  $2^k - 1$
- $n$  个内部节点的红黑树树高最大是  $2 \log_2(n + 1) + 1$
- 红黑树检索, 插入, 删除的最差时间都是  $O(\log n)$

## 操作

**左旋与右旋** -> 左旋(x), <-右旋(y)



保留BST的性质, 旋转后的树的中序遍历结果不变 ( $a < A < b < B < c$ )

### 红黑树的插入

- 插入节点为红色
- 若父节点为黑色, 不违反性质, **直接插入**
- 若父节点为红色, 则违反性质, 需要调整, 不妨设父节点在左, 叔节点在右
  - 叔节点为红色, 则将父节点, 叔节点染黑, 祖父节点染红, **递归**调整祖父节点
  - 叔节点为黑色, 且当前节点为右子节点, 则左旋父节点, **转化**为左子节点的情况
  - 叔节点为黑色, 且当前节点为左子节点, 则右旋祖父节点, 旋转前的祖父节点染红, 父节点染黑, **结束**
- 时间复杂度  $O(\log n)$ , 因为递归调整的次数不超过树高的一半

### BST的删除

- 没有儿子, 直接删除
- 有一个儿子, 用儿子替代
- 有两个儿子, 找到右子树的最小节点(即从右子树根出发递归找左儿子), 用该节点替代, 再删除该节点 (该节点最多只有一个右儿子) (或者找左子树的最大节点)

**红黑树的删除** : 不考, 懒得写了

# 12 高级数据结构

## 12.1 多维数组和稀疏矩阵

### 多维数组

行优先顺序: 先排最右下标, 从右到左, 最后排最左下标

列优先顺序: 先排最左下标, 从左向右, 最后排最右下标

对于行优先顺序:

$$loc(A[j_0, \dots, j_{n-1}]) = loc(A[0, \dots, 0]) + d[\sum_{i=0}^{n-2} j_i \prod_{k=i+1}^{n-1} d_k + j_{n-1}]$$

### 特殊矩阵

#### 对称矩阵/三角矩阵

- 只存上三角或者下三角

#### 稀疏矩阵

- 三元组表示法：行数  $M$ , 列数  $N$ , 非零元素个数  $T$ ,  $\{(i, j, a_{ij}) | a_{ij} \neq 0\}$
- 十字链表：行链表头指针数组, 列链表头指针数组,  $\{(i, j, a_{ij}, *nextcol, *nextrow)\}$
- **CSR** (压缩稀疏矩阵表示)：行指针数组, 列索引数组, 非零元素数组
  - 每一行对应列索引数组的一个区间, 该区间内的元素为矩阵该行非零元素的列索引, 行指针指向该区间的起始位置, 列索引与非零元素数组的下标对齐

## 12.2 广义表

对于**广义表**  $L = (x_0, \dots, x_{n-1})$

- $x_i$  可以是单个元素 (即原子 atom), 也可以是广义表 (即子表 sublist)
- 广义表深度：递归定义, 原子的深度为0, 非原子的深度为其元素的最大深度加1
- 表头  $x_0$ , 表尾  $x_{1:n-1}$

### 广义表的类型

- **纯表** (pure list)
  - 从根节点到任何叶节点只有一条路径
  - 任何元素 (原子, 子表) 只能出现一次
- **可重入表(再入表)** (reentrant list)
  - 任何元素可以出现多次
  - 对应一个DAG
- **循环表(递归表)** (circular list)
  - 包含回路, 深度无穷大

图  $\supseteq$  再入表  $\supseteq$  纯表(树)  $\supseteq$  线性表

应用: Lisp语言中的S表达式, 函数调用关系, 内存引用关系

### 广义表的ADT

```
struct TreeNode {
    bool isAtom;        // true: 原子, false: 子表
    union {
        char atom;      // 原子值
        TreeNode* sublist; // 子表指针
    };
    TreeNode* sibling;    // 指向同层兄弟节点
};
```

为什么广义表要有头结点? 否则删除结点必须进行链调整, 增加头指针可以简化删除, 插入操作.

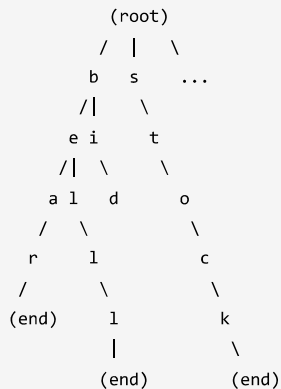
头指针tag可以是-1, value为空. 只有指向第一个结点的指针

## 12.3 字符树

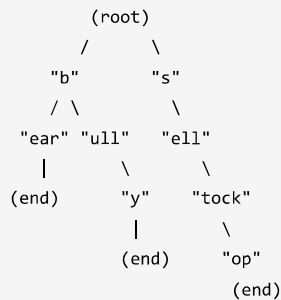
本节只了解

以存储字符串集合 ["bear", "bell", "bid", "bull", "buy", "sell", "stock", "stop"] 为例

前缀trie



patricia树



## 12.4 最佳BST树

一个  $n$  个关键码的集合, 关键码可以有  $n!$  种不同的排列法, 但是对应的二叉搜索树只有  $C_{2n}^n / (n + 1)$  种不同的形态.

效率度量

- 先扩充二叉搜索树, 添加外部空节点, 为检索失败的情形提供一个关键码区间
- 成功检索: 关键码层数加1 (根是第0层)
- 失败检索: 被检索关键码所属的外部空节点层数

$$ASL = \frac{\sum_{i=1}^n p_i(l_i + 1) + \sum_{i=0}^n q_i l'_i}{\sum_{i=1}^n p_i + \sum_{i=0}^n q_i}$$

- $p_i$ : 成功检索第  $i$  个内部节点的概率 (总共  $n$  个)
- $q_i$ : 失败检索第  $i$  个外部节点的概率 (总共  $n + 1$  个)
- $l_i$ : 第  $i$  个内部节点的层数
- $l'_i$ : 第  $i$  个外部节点的层数

最佳BST: 使得平均检索长度最小的二叉搜索树, 这取决于关键码的概率分布

- 关键码概率分布均匀时 ( $p_i = q_i = \frac{1}{2n+1}$ ), 等价于最小化内部路径长度和, 最佳BST是一棵完全二叉树
- 关键码概率分布不均匀时, 其任何子树都是最佳BST
  - **动态规划**: 从底向上, 逐层计算最佳二叉搜索树的平均检索长度

## 12.5 AVL树

AVL树的性质

- 空树是AVL树
- 左右子树都是AVL树, 且左右子树的高度差的绝对值不超过1
- $n$  个结点的AVL树的高度不超过  $O(\log n)$

$bf(x)$ : 节点  $x$  的平衡因子, 即右子树高度减去左子树高度, 取值范围  $\{-1, 0, 1\}$

AVL 树的  $ASL$  为  $O(\log n)$

AVL树的插入

- 与BST类似, 执行失败的查找, 确定插入位置插入新结点

- 若插入后不破坏平衡性, 则逐层调整平衡因子
- 若插入后破坏平衡性, 则进行旋转操作, 从新加入的结点  $u$  向根方向往上搜索, 直到找到距离  $u$  最近的不平衡祖先结点  $A$  (平衡因子为2或者-2) 定位路径  $A \rightarrow B \rightarrow C \rightarrow \dots \rightarrow u$ .
  - **LL型**:  $A \xrightarrow{left} B \xrightarrow{left} C$ , 右旋一次, 让  $B$  取代  $A$  成为新的子树根即可
  - **RR型**:  $A \xrightarrow{right} B \xrightarrow{right} C$ , 左旋一次, 让  $B$  取代  $A$  成为新的子树根即可
  - **LR型**:  $A \xrightarrow{left} B \xrightarrow{right} C$ , 先对  $B$  左旋,  $C$  取代  $B$ , 再对  $A$  右旋,  $C$  取代  $A$  成为新的子树根
  - **RL型**:  $A \xrightarrow{right} B \xrightarrow{left} C$ , 先对  $B$  右旋,  $C$  取代  $B$ , 再对  $A$  左旋,  $C$  取代  $A$  成为新的子树根

## AVL树的删除

- 与BST类似, 执行失败的查找, 确定删除位置, 与后继交换再删除
- 删除会导致树高及平衡因子变化, 需要沿着被删除结点到根结点的路径来调整这种变化. 但删除比插入复杂, 可能需要在相应的路径上的不止一次实施单旋或双旋

AVL树的插入和删除操作的时间复杂度为  $O(\log n)$

## AVL树的扩展

允许树的高度差  $\Delta$  超过1, 但是仍然保持平衡性, 最差情况下高度随  $\Delta$  增长, 与单纯的AVL树 ( $\Delta = 1$ ) 相比, 平均访问节点数目增加了, 但重组的数目降低了

# 12.6 伸展树

**展开(splaying)**: 访问一次节点  $x$ , 完成一次展开过程

- 插入检索  $x$  时, 把  $x$  旋转到根节点
- 删除检索  $x$  时, 把  $x$  的父节点旋转到根节点
- 当  $x$  是根节点的子节点时, 用单旋转 (和AVL树的旋转相同)
- 否则递归地进行双旋转
  - **一字型旋转**, 也称为同构调整, 适用于  $x$  是祖父节点的左左孙子节点或右右孙子节点, 相当于两次单旋转
  - **之字型旋转**, 也称为异构调整, 适用于  $x$  是祖父节点的左右孙子节点或右左孙子节点, 和AVL树的双旋转相同

## 基于伸展树的区间操作

- 区间提取: 给定数列  $\{n_i\}_m$ , 提取区间  $[n_a, n_b]$  的子序列
  - 将  $n_a$  旋转到根节点
  - 将  $n_b$  旋转到根节点的右子节点
  - 提取树根右子树的左子树即为所求子序列
- 区间删除: 删除区间  $[n_a, n_b]$  的子序列
  - 同区间提取
  - 删除树根右子树的左子树
- 区间插入: 插入区间  $[n_a, n_b]$  的子序列
  - 将  $n_a$  旋转到根节点
  - 将  $n_{a+1}$  旋转到根节点的右子节点
  - 将待插入子序列构建成伸展树, 插入成为树根右子树的左子树
- 区间翻转: 翻转区间  $[n_a, n_b]$  的子序列
  - 同区间提取
  - 对树根右子树的左子树进行翻转 (交换每个节点的左右儿子)

**半伸展树(Semi-Splay)**: 每次伸展只对祖父节点进行一次旋转, 父节点取代祖父节点成为新的子树根, 然后当前节点由  $x$  变为  $x$  的父节点 (此时的子树根)

- 半伸展不需要把  $x$  旋转到根节点, 如  $x$  的原层数是  $d$ , 则伸展后的层数是  $d/2$
- 如果记录信息存储在外结点, 内部结点只作为索引结构, 则检索到某个外部结点时, 其内部父结点设为当前结点

**后缀树(Suffix Trie)**: 类似前缀树, 但每个从根到叶子的路径都对字符串的一个后缀, 用于字符串匹配, 模式匹配, 模式搜索等.

- 为区分后缀子串, 在每个叶子节点后添加一个特殊字符\$
- 很容易查找  $P$  是否是  $T$  的子串 (依次检索)
- 查找  $P$  出现次数: 沿着  $P$  的路径走到叶子节点, 计算出出现次数  $k$ , 复杂度  $O(|P| + k)$
- 找到T中最长重复子串: 找到最深的分叉节点, 从根到该节点的路径即为最长重复子串 (子串可能相交)

后缀树似乎不考

# 几种平衡机制的比较

- **AVL树** : 完全平衡, 与访问频率无关, 只与插入, 删除顺序有关
- **伸展树** : 与操作频率相关, 根据插入、删除、检索等动态调整
- **红黑树** : 局部平衡, 统计性能好于 AVL树, 增删记录算法性能好

**单旋** : AVL树插入-LL,RR型旋转, 伸展树splay-第一层节点, 红黑树插入-叔节点为黑-LL,RR型

**两次同向单旋** : 伸展树splay-一字型旋转

**双旋(两次异向单旋)** : AVL树插入-LR,RL型旋转, 伸展树splay-之字型旋转, 红黑树插入-叔节点为黑-LR,RL型

无论是哪种二叉树结构, LR型和RL型旋转后都是**孙子节点**成为新子树根

而LL 型和 RR型就未必了, 除了伸展树的splay是**孙子取代祖父**, 其他都是**父亲取代祖父**

