数据结构与算法：期末复习

黄宇翔
清华大学计算机系

2023 年 2 月 25 日

1 绪论

1.1 渐进分析

- $T(n) = O(f(n))$：当 n 足够大时，$\exists c > 0, T(n) < cf(n)$.
- $T(n) = \Theta(f(n))$：当 n 足够大时，$\exists c_1, c_2 > 0, c_1 f(n) < T(n) < c_2 f(n)$.
- $T(n) = \Omega(f(n))$：当 n 足够大时，$\exists c > 0, cf(n) < T(n)$.

1.2 时间复杂度排序

$$O(1) < O(\log^* n) < O(\log n) < O(n) < O(n \log^* n) < O(n \log n) < O(n^2) < O(n^*) < O(2^n)$$

1.3 主定理

定理 1.1. *若有* $T(n) = aT(n/b) + O(f(n))$*，则:*

- *若* $f(n) = O(n^{\log_b a - \epsilon})$*，则* $T(n) = \Theta(n^{\log_b a})$
- *若* $f(n) = \Theta(n^{\log_b a} \log^k n)$*，则* $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- *若* $f(n) = \Omega(n^{\log_b a + \epsilon})$*，则* $T(n) = \Theta(f(n))$

2 向量

向量：元素的物理地址相邻，可以按秩访问。

2.1 向量的扩容算法

算法 2.1. *扩容后容量 = 2* 扩容前容量*

2.1.1 容量加倍策略的时间复杂度分析

最坏情况：插入长度为 $n = 2^m$ 的序列
每次复制向量元素的时间成本：$1 + 2 + 4 + 8 + \cdots + 2^{m-1} + 2^m = 2^{m+1} - 1 = O(n)$
分摊到每次插入是 $O(1)$。

2.1.2 容量递增策略的时间复杂度分析

每次递增长度为 I，最坏情况是连续插入 $n = mI$ 个元素
每次复制向量元素的时间成本：$0 + I + 2I + \cdots + (m-1)I = \frac{m(m-1)I}{2} = O(n^2)$
分摊到每次插入是 $O(n)$。

2.2 有序向量的唯一化

算法 2.2. *从左向右扫一遍，当当前 entry 不同于前一个的时候，依次归到向量的前面。*

时间复杂度：$O(n)$，无序向量需要 $O(n^2)$。

2.3 二分查找

算法 2.3. *取中点，当查找点小于中点在左半边查找，当查找点大于中点在右半边查找，不然则命中。*

时间复杂度：$O(\log n) = 1.5 \log n$
不妨设二分查找的常数仅与分割点的位置 λ 有关，即 $O(\log n) = \alpha(\lambda) \log n$, 那么

$$\alpha \log_2 n = \lambda[1 + \alpha \log_2(\lambda n)] + (1 - \lambda)[2 + \lambda \log_2((1 - \lambda)n)]$$

通过求 $\frac{\partial \alpha}{\partial \lambda} = 0$，计算得到 $\lambda = \frac{-1+\sqrt{5}}{2}$, 也就是 $\lambda = 0.618$，$\alpha = 1.440$，长度为 $n = fib(k) - 1$ 的序列应当分为长度为 $fib(k-1) - 1$,$fib(k-2) - 1$ 的两段。

2.4 差值查找

算法 2.4. *设序列为 $[A[lo], A[hi]]$。查找 e 时，令轴点为 $mi = lo + (hi - lo)\frac{e - A[lo]}{A[hi] - A[lo]}$，根据轴点将序列范围缩小，递归查找。*

可以证明，每经一次比较，区间宽度由 n 缩减到 $\sqrt{n}$。
令 $S(n) = T(2^n)$，则 $S(n) = T(2^n) = T(2^{n/2}) + O(1) = S(n/2) + O(1)$。由主定理，$S(n) = O(\log n)$ $T(n) = S(\log n) = O(\log \log n)$.

3 列表

3.1 选择排序

算法 3.1. *第 k 次循环选择第 k 个最值，然后待排序区间长度减一。*

时间复杂度：$O(n^2)$，在列表上只换两个 entry，不需要对掉结点。

3.2 插入排序

算法 3.2. *第 k 次循环，将第 k 个元素按照顺序插入前 $k-1$ 个元素。前 $k-1$ 个元素是有序的。*

3.3 归并排序

归并排序求逆顺对个数：在归并排序递归返回时，将两个已经排好顺序的序列 L, R 合并。需要同时记录相同元素 e 在 L, R 中的位置。e 将 L 划分为 LL, LR，将 R 划分为 RL, RR。则 $|LR|$ 是 e 对应的逆序对数量。（逆序对算在后一个元素上）

4 栈与队列

4.1 栈混洗

使用栈来判断是否为合法的混洗序列：逆向进栈出栈，保证栈中元素一定单调递增（不然就将 entry 直接进入原先的栈中），查看是否能恢复 $1, 2, \ldots, n$ 的序列。

4.2 中缀表达式求值

- ’<’（栈顶优先级小于当前）：将操作数和符号入队；
- ’>’（栈顶优先级大于当前）：弹出相应的操作数，计算，并将计算结果压入数栈中；
- ’=’：右括号对应左括号，将括号去掉即可。

平级操作符是’>’ 的原因：希望从左向右计算，减少栈中存储的元素数量。

栈顶为任意符号，当前是’(’，操作符是’<’ 的原因：左括号都需要压入栈中

除’(’ 外，其他任意符号遇到’)’ 都是’>’ 的原因：有小括号结束，需要计算小括号内的内容，直到左右括号相遇。

左右括号优先级是’=’：只需要将括号成对移除，没有栈操作或运算，因此需要单独设置优先级

开始和结束的一对’\0’ 可以视作一种特殊的括号，优先级小于普通括号。

[栈顶][当前]	+	-	*	/	^	!	(	)	\0
+	$>$	$>$	$<$	$<$	$<$	$<$	$<$	$>$	$>$
-	$>$	$>$	$<$	$<$	$<$	$<$	$<$	$>$	$>$
*	$>$	$>$	$>$	$>$	$<$	$<$	$<$	$>$	$>$
/	$>$	$>$	$>$	$>$	$<$	$<$	$<$	$>$	$>$
^	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$	$>$
!	$>$	$>$	$>$	$>$	$>$	$>$		$>$	$>$
(	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$=$	
)									
\0	$<$	$<$	$<$	$<$	$<$	$<$	$<$		$=$

表 1: 运算优先级

逆波兰表达式的求值：只需要一个栈，按顺序压入，遇到符号，弹出对应的操作数计算后将结果压入栈中即可。

由中缀表达式转化为逆波兰表达式：在上面中缀表达式求值的算法中，读入数字时自动入 RPN 栈，优先级为'>'（做运算时）将符号压入 RPN 栈，最后将其弹出后倒序即可。

4.3 双栈当队

算法 4.1. *定义上栈 F，下栈 R. 入队时候向 R 压入元素，出队时先将 R 中元素依次弹出并压入 F 中，再出队。*

4.3.1 证明其分摊时间复杂度仍然是 $O(1)$

定义势能 $\Phi_k = |R_k| - |F_k|$，则 $A_k := T_k + \Phi_k - \Phi_{k-1} \equiv 2$.

由于 $|T(n)| \leq n, T(0) = 0$，则有

$$2n = \sum_{k=1}^{n} A_k = T(n) + \Phi_n - \Phi_0 \geq T(n) - n$$

也就是 $T(n) \leq 3n = O(n)$.

4.4 Queap

Queap 用来维护滑动窗口内的最值，入队的时候向前更新队列内所有 entry 的滑动窗口内最值。

优化：在队列中不存多个实例，而是存一个实例 + 实例本应重复的次数。

4.5 直方图内最大矩形

使用一个栈来维护:

- 对于每个 entry，先弹出栈中所有比之大的元素，记录最后一个弹出的元素 r；
- 将其对应矩形左端点记录为栈顶元素位置 $+1$；
- 当前位置 t，矩阵左起点是 s, 则最大矩形与 $H[r]*(t-s)$ 做比较，取大者。

5 二叉树

5.1 一些指标

- 路径长度：路径所含边的数目
- 根结点的深度是 0，任意结点 v 的深度是从根结点到 v 的路径长度（边数）
- 叶结点的高度为 0，任意结点 v 的高度是 $h(v)=\max(h(v.lc),h(v.rc))+1$
- 满二叉树：$n=2^{h+1}-1$
- 真二叉树：每个非叶结点都有两个孩子
- 完全二叉树：除最后一层外，其余所有层的结点都应当是满的，最后一层左侧是满的，右侧是空的

5.2 三种遍历方式的迭代算法

5.2.1 先序遍历

沿藤访问：沿着左侧藤，先访问结点，再将右孩子压入栈中，然后进入左孩子，直至没有左孩子。

先序遍历：对根结点做沿藤访问；结束后若栈非空，弹出一个结点，对它沿藤访问，直至栈空。

5.2.2 中序遍历

沿藤访问：沿着左侧藤，将结点压入栈中，然后进入左孩子，直至没有左孩子。

中序遍历：从根结点开始，先沿藤访问，若栈非空，弹出结点并访问，然后进入其右孩子，再沿藤访问，弹出结点访问，进入右孩子直至栈空。

5.2.3 后序遍历

进入最左结点：1）将根结点先入栈，2）cur= 栈顶，按顺序入栈 (右子)（左子）3）进入左子

后序遍历：栈顶与下面一个元素是父子：向上（此时父结点可以访问）即可；栈顶与下面一个元素是兄弟：需访问右兄弟的子树。

5.3 二叉树重构

先序（后序）+ 中序 => 重构二叉树：在中序遍历中查询先序（后序）的首结点（尾结点）的位置，左边是左子树，右边是右子树。（可以完全地恢复二叉树）

先序 + 后序：先序遍历的首结点和后序遍历的末尾结点是同一个结点。左孩子和右孩子在序列中的位置可以区分左右子树（pp.457）（只能恢复二叉树的拓扑连接，单个孩子的左右方向无法确定）

5.4 huffman tree

统计词频，每次合并两个权重最小的结点，直到合并到只有一棵树。

叶节点带权路径之和最小的树是 huffman tree。

建立 huffman tree 的时间复杂度 $O(n\log n)$: 维护一个优先队列，在 $O(\log n)$ 时间内拿到两个权重最小的结点。

也可以令所有字符按频率排序，构成一个栈，每次取栈顶两个元素放入一个有序的队列当中。（单调队列维护：$O(n)$）

6 二叉搜索树

6.1 朴素的二叉搜索树

查找：类似二分查找。（_hot 指向目标节点的父亲，或者是叶子结点（查找失败时））

插入：永远插入在叶子结点上。先通过查找获得 _hot 位置，然后在 _hot 下面对应位置插入新结点

删除：

1）只有一个孩子结点：直接用孩子结点接替当前结点。

2）有两个孩子：将当前结点与直接后继对调，然后在直接后继的位置上做删除。

6.2 AVL 树

平衡因子：$balFac(v) = height(lc(v)) - height(rc(v))$. 希望

$$\forall v \in V, |balFac(v)| \leq 1(balFac(v) = -1, 0, 1)$$

6.2.1 插入

插入结点后，可能会破坏 AVL 树的平衡性质。需要 rebalance。从插入结点向上回溯，设第一个失衡结点是 g，g 的更高孩子结点是 p，p 的更高孩子结点是 v。

1）g, p, v 处于同一方向：在 g 上做旋转，令 p, v 成为 g 的孩子

2）$g - p, p - v$ 处于相反方向：先在 p 上做旋转，转化为 1)，然后再在 g 上旋转

插入时。做一次调整即可消除不平衡，不会将不平衡依次向上传播

6.2.2 删除

删除结点之后，也可能会破坏 AVL 树的平衡性质，需要 rebalance。与插入相同，层层回溯调整。

删除可能会将不平衡一直传播到根节点，需要不断旋转。

7 Application of binary trees

7.1 kd-tree(2d-tree)

第奇数次分割是纵向的，偶数次分割是横向的。

建树：确定横/纵向后，取中位数，将剩下的点分给左右两棵子树，递归建树

每个结点对应的范围：根结点对应全部空间，根的左孩子对应根节点所在纵线以左，根的有孩子对应根节点所在纵线以右。递归地将空间分下去。

查询：

算法 7.1 (kdSearch(v, R))**.** *If v represents a leaf node: if $v \in R$, report v.*

If v is not a leaf node:

1) if $v -> lc \in R$, report $v -> lc$

2) if $v -> rc \in R$, report $v -> rc$

3) if $v -> lc \cap R \neq \emptyset$, kdSearch(v->lc, R)

4) if $v -> rc \cap R \neq \emptyset$, kdSearch(v->rc, R)

搜索时间：$Q(n) = 2Q(n/4) + O(1)$，通过主定理 $Q(n) = O(\sqrt{n})$，加上输出时间 $O(r)$，总时间复杂度是 $O(\sqrt{n} + r)$

kd-tree: 查询复杂度 $O(r + n^{1-1/d})$, 空间复杂度 $O(n)$, 建树时间复杂度 $O(n \log n)$.

7.2 Range tree

在 x 维度建二叉树，每个结点索引一棵 y 方向的二叉树。

建树：$O(n\log n)$，可以先排序 $O(n\log n)$，然后在有序的序列上每次 $O(n)$ 选出子节点的集合。时间递推式是

$T(n)=2T(n/2)+O(n)$，解得 $T(n)=O(n\log n)$。

多维 range tree：建树 $O(n\log^{d-1}n)$，空间复杂度 $O(n\log^{d-1}n)$，查询复杂度 $O(r+\log^{d}n)$.

7.3 Interval tree

建树：储存所有区间的端点。寻找中位数，结点对应的区间是中位数穿过的所有区间，按照顺序储存。将当前节点对应区间去掉之后，将在中位线左的区间给左孩子建树，在中位线右的区间给右孩子建树。

查询：按照大小关系，在结点的左/右翼上筛选区间是否满足条件。然后类似二分查找递归下去。

7.4 Segment tree

建树：将定义域按照区间端点离散化。根节点对应所有元区间，左孩子对应左半边元区间，右孩子对应右半边元区间。递归进行下去。

查询：类似 kd-tree。时间复杂度 $O(\log n+r)$

8 高级搜索树

8.1 Splay

算法 8.1 (双层伸展)**.** *设当前节点为 v，它的父亲结点和祖父结点是 f,g。*

1）$g->lc=f,f->lc=v$：先右旋 g，再右旋 f

2）$g->rc=f,f->rc=v$：先左旋 g，再左旋 f

3）$g->rc=f,f->lc=v$：先右旋 f，再左旋 g

4）$g->lc=f,f->rc=v$：先左旋 f，再右旋 g

搜索：如果查找到结点之后，通过双层伸展将访问结点上推到根；查找失败时，将 _hot 通过双层伸展上推到根。

插入：先在树中做失败查找 (_hot 被上推到根)，记根和左右孩子为 t,l,r，新节点为 s. 如果 $s\leq t$, 新树以 s 为根，$s->lc=l,s->rc=t;t->lc=NULL,t->rc=r$. 如果 $s>t$，新树以 s 为根，$s->rc=r,s->lc=t;t->rc=NULL,t->lc=l$.

删除：先在树中做成功查找，将目标节点上推到根。删除根节点，在右子树中再次查询目标节点（失败），此时右子树的根节点没有左孩子，那么可以将左子树接在右子树根的左孩子位置。

定理 8.1. *Splay 的平均查询复杂度是* $O(m\log n)$（m *是查询次数*，$m >> n$）

定义 8.1 (Splay 势能)**.**

$$rank(v) = \log(size(v))$$

$$\Phi(S) = \log(\prod_{v\in S} size(v)) = \sum_{v\in S}\log(size(v)) = \sum_{v\in S} rank(v) = \sum_{v\in S}\log V$$

越平衡的树，势能越小；越倾斜的树，势能越大。

记号 8.1. $A^k = T^k = \Delta\Phi^k$，$k = 0, 1, \ldots, m$，*上标* k *表示第* k *次操作后的树。*

注意到有 $O(n) \le \Phi \le O(n\log n)$，那么有 $O(n) \le \Delta\Phi^k \le O(n\log n)$，所以有

$$A - O(n\log n) \le T = A - \Delta\Phi \le A + O(n\log n).$$

如果可以证明 $A = O(m\log n)$，那么必然有 $T = O(m\log n)$.

注意到 $0 \le rank^k(v) = \log(size^k(v)) \le \log n$，所以有 $|\Delta rank^k(v)| \le O(\log n)$.

引理 8.1.

$$O(rank^k(v) - rank^{k-1}(v)) = O(\log n)$$

下面考察三种情况：单纯 zig(zag)，zig-zag(zag-zig)，zig-zig(zag-zag).

zig(zag):

设 $r- > lc = v$，在 r 上做 zig 操作。下标 i 表示第 i 次调整操作。

$$A_i^k = T_i^k + \Delta\Phi(S_i^k) = 1 + \Delta rank_i(v) + \Delta rank_i(r)$$
$$= 1 + [rank_i(v) - rank_{i-1}(v)] + [rank_i(r) - rank_{i-1}(r)]$$
$$< 1 + rank_i(v) - rank_{i-1}(v)$$

zig-zag(zag-zig):

$$A_i^k = T_i^k + \Delta\Phi(S_i^k) = 2 + \Delta rank_i(g) + \Delta rank_i(p) + \Delta rank_i(v)$$
$$= 2 + [rank_i(v) - rank_{i-1}(v)] + [rank_i(p) - rank_{i-1}(p)] + [rank_i(g) - rank_{i-1}(g)]$$
$$< 2 + rank_i(g) + rank_i(p) - 2rank_{i-1}(v)$$
$$< 2 + 2rank_i(v) - 2 - 2rank_{i-1}(r)$$

$$= 2(rank_i(v) - rank_{i-1}(v))$$

zig-zig(zag-zag):

$$A_i^k = T_i^k + \Delta\Phi(S_i^k) = 2 + \Delta rank_i(g) + \Delta rank_i(p) + \Delta rank_i(v)$$

$$= 2 + [rank_i(v) - rank_{i-1}(v)] + [rank_i(p) - rank_{i-1}(p)] + [rank_i(g) - rank_{i-1}(g)]$$

$$< 2 + rank_i(g) + rank_i(p) - 2rank_{i-1}(v)$$

$$< 2 + rank_i(g) + rank_i(v) - 2rank_{i-1}(v)$$

$$< 3(rank_i(v) - rank_{i-1}(v))$$

最终，$A^k = O(\log n), A = O(m \log n)$.

8.2 B-Tree

m 阶 B-Tree：又称作 ($\lceil m/2 \rceil$, m)-Tree, 即所有除了叶子结点之外的结点（内部节点），其分支数目 $\lceil m/2 \rceil \le s \le m$，结点数目是分支数目-1.（根的元素个数没有下限）

查找：在当前结点线性查找，无法找到时从两个 entry 之间的分支下到下一层的某个结点，递归地进行这一过程。

B-Tree 的树高：$\log_m(N+1) \le h \le 1 + \lfloor \log_{\lceil m/2 \rceil} \frac{N+1}{2} \rfloor$，所以查找复杂度是 $O(\log n)$ 的。

插入：先通过搜索确认目标结点将要插入的叶子结点，在叶子结点上进行插入操作。如果结点数超过 $m-1$，进行上溢操作。

算法 8.2 (上溢)**.** *设当前结点为 v, v 的父亲是 f. 设 v 上的结点为 $\{k_0, \ldots, k_{m-1}\}$，设 f 结点第 i 个结点的右孩子是 v.*

取中位数 $s = \lfloor m/2 \rfloor$，将 v 分裂成 $l = \{k_0, \ldots, k_{s-1}\}, \{k_s\}, r = \{k_{s+1}, k_{m-1}\}$，然后将 k_s 上升一层，插入到 f 中，并令它的左孩子、右孩子为 l, r.

递归地上升上去，直到根结点。如果根节点元素个数超过要求的话，分裂出一个新根，树高 +1.

删除：先类似二叉树的删除，将目标节点交换到它的直接后继，在叶结点上完成删除操作。如果叶结点删除后，元素个数 +1 少于 $\lceil m/2 \rceil$, 完成调整操作。

算法 8.3 (删除调整)**.** *设当前结点为 v, v 的父亲是 f. 设 f 结点第 i 个结点的右孩子是 v. 设 v 的左右兄弟分别是 l, r.*

1) 若 $|l| + 1 > \lceil m/2 \rceil$：将 f_i 下降到 v 的最左侧结点，用 l 的最后一个结点代替 f_i 的位置（旋转）

2) 若 $|r| + 1 > \lceil m/2 \rceil$：类似 3)

3) 若 $|l| = |r| = \lceil m/2 \rceil - 1$*:* 与左侧结点做合并。将 f_i 下放到 l, v 之间，将 l, f_i, v 合并成一个结点。

当进行到 3) 的时候，需要回溯父亲结点依次向上做检查。

8.3 红黑树

红黑树的定义：

- 先对每个不具有两个孩子的结点加上 NULL 结点。
- 树根：黑色
- 外部结点：黑色
- 红结点：只能有黑父亲，只能有黑孩子
- 外部节点的黑深度（黑色真祖先数目）相等

红黑树的实质:将红色节点提升到它的黑色父亲结点旁边,则红黑树是一棵 $(2,4)-Tree.$

插入：先按照二叉树的插入方法插入一个新的结点，为了不影响外部结点的黑深度，将新的结点设置为红色。有可能它的父亲也是红色的，进行双红修正。

算法 8.4 (双红修正)**. 双红修正看叔父 *(uncle)* 结点**

1) 叔父结点是黑色: 用 B 树思维, 将结点重新染色。实际上: 在祖父结点处先做 zig/zag, 让父亲结点达到祖父结点的高度。然后将父亲节点染红，祖父结点染黑。

2) 叔父结点是红色：将父亲结点与叔父结点染黑，祖父结点染红即可。

如果进行了 *1)*，双红不会继续传递。如果进行了 *2)*，双红还可能继续传递。

删除：先与直接后继交换数值（不交换颜色），设直接后继 v 有孩子 c. 如果 v 是红色，直接删去即可。如果 v 是黑色，c 是红色，删去 v 后染黑 c 即可。如果 v, c 都是黑色，需要进行双黑修正。

算法 8.5 (双黑修正)**. 先看兄弟结点，再看父节点**

1)v 的兄弟结点是 s，如果 s 是黑色且有一个红孩子，那么在 p 做一次 *zig/zag*，令 s 达到 p 的高度。令 s 接替 p 的颜色，p 与红色孩子都染黑。

2)s 黑，且没有红孩子

2.1) p 是红色：令 p 是黑色，s 变成红色即可。

2.2) p 是黑色：此时无法 $O(1)$ 内将局部的黑高度恢复到与全局一致，只好将 p 的左右孩子黑高度恢复一致。染红 s 即可。需要回溯向上。遇到双黑就要处理。

3) s 是红色：在 p 处做 *zig/zag*，令 s 达到 p 的高度。令 s 变成黑色，p 变成红色，然后再用 *1* 或者 *2.1* 处理。

9 词典

9.1 Hashing

hashfunction: 通常取模大质数

冲突的解决：

1）开放散列：多槽位，公共溢出区，独立链

2）封闭散列：线性试探（一旦冲突，试探紧邻后面的桶）、平方试探。

平方试探：单项版在装填因子小于一半时一定可以查询到。

双向平方试探：取表长为大质数 M 满足 $M = 4k + 3, k \in \mathbb{N}$，那么一定可以查询到。

9.2 桶排序

桶排序：可以在 $O(n)$ 时间内完成对 n 个数的排序，需要 $O(n)$ 的额外空间。需要利用很好的 hashfunction 将值域进行压缩。

基数排序：按照某种进制的高位到低位，不断利用桶排序进行排序。桶排序是一种稳定的排序算法。

9.3 跳转表

跳转表：若干个链表叠起来。

插入元素时，每层有一半的概率将当前塔向上增长。查找和删除均从最上面的链表开始，查询时间复杂度是 $O(\log n)$.

10 图

10.1 广度优先搜索

维护一个队列，开始时将源点入队。取出队顶元素并设置访问时间。考察结点的所有相邻结点。如果未发现，则加入队列中，将点设置为 DISCOVERED，边设置为 TREE。

如果结点已经所 DISCOVERED 或者 VISITED，那么将边设置为 CROSS。

结点的所有相邻结点访问完成时候，设置为 VISITED。

10.2 深度优先搜索

进入结点时：将结点设置为 DISCOVERED，然后依次访问与它相邻的结点。如果相邻结点未访问，设置为 TREE 边，如果相邻结点是 DISCOVERED，设置为 BACKWARD 边，不然则设置 CROSS 边。

所有相邻结点访问结束后，设置该结点为 VISITED。

边的分类：TREE 树边，BACKWARD 指向祖先结点，CROSS 边：指向更早访问的非祖先结点。

10.3 拓扑排序

时间复杂度：$O(n+e)$

算法 10.1 (零入度算法)**.** *初始状态：将所有零入度结点入队。*

从队中拿一个结点，将与它相邻，入度为 1 的结点入队，然后删除结点。

重复直至队空。

算法 10.2 (零出度算法)**.** *在深度优先搜索中，从结点返回时，将当前结点入栈。深度优先搜索结束后，倒序输出所有元素即可。*

11 图应用

可以使用优先级队列来维护结点权重。

Dijkstra 算法：$p(u)$ $w(v,u)+p(v)$ 来松弛 $p(u)$。$O((n+e)\log n)$.

Prim：$p(u)$ $w(v,u), v\in$ 已经选入的点。$O((n+e)\log n)$.

Kruskal：使用并查集，从最短边到最长边依次选择。$O(e\log e)$.

12 优先级队列

12.1 完全二叉堆

使用向量来维护即可。

插入：插入在最后一个位置，逐层与父亲结点比较交换。

删除：删去根结点之后，用最后一个结点替换，然后逐层与孩子比较交换。

12.2 胜者树

胜者树：父亲结点记录的是两个孩子中优胜孩子的权重。

按照胜者排序：沿着根节点找到胜利者对应的叶子结点，将该结点失效后沿着路径进行深度为 $\log n$ 的重赛。可以另取一个树，其结点记录的败者。

12.3 左式堆

定义 12.1. *$npl(v)$：v 子树中最近 $NULL$ 结点的高度。*

$npl(NULL)=0, npl(x)=1+\min\{npl(x.lc), npl(x.rc)\}$

定义 12.2 (左式堆). *对任何内部结点 v，都有 $npl(v.lc) \geq npl(v.rc)$.*

算法 12.1 (左式堆的合并). *取两个指针 a, b 指向两个堆的根（大根堆）。*

让 $a > b$，不然交换 a, b。$a \leftarrow a.rc$, 考察 a 与 b 的大小，令 $a > b$，然后将 a 接在上一轮的空处，再令 $a \leftarrow a.rc$。

插入：单个元素也可以看做是左式堆。

删除：删去根结点之后，合并 $r.lc, r.rc$。

13 串

13.1 KMP

算法 13.1 (next 的构建). *$t = -1, j = 0$.*

如果 $P[t] = P[j]$，那么 $N[j+1] = t+1$(如果 $P[j+1] \neq N[j+1]$ 如此，不然则 $N[j+1] = N[t+1]$), $t++, j++$, 不然则 $t = N[t]$

构建的依据：

定理 13.1. *$P[j+1] = P[next[j]+1] \iff next[j+1] = next[j]+1$*

当不满足该条件时候，只需令 $t \leftarrow next[t]$ 即可。时间复杂度：$O(n+m)$

13.2 BM 算法：BC 策略

定义 13.1. *$bc['X'] = j$(j 是'X' 最后出现的位置), -1('X' 没有出现过)*

搜索：从后向前搜索，遇到不匹配的则向后移动，使用 bc 表的位置来使得当前位置可以对应上。

Note：不做反向移动，如果需要反向移动，则正向移动一个位置。

时间复杂度：最好 $O(n/m)$，最坏 $O(nm)$

13.3 trie

pp.1216 用来统计词频

算法 13.2 (快速排序). *每次随机选取一个 $pivot$，将比 $pivot$ 大的归在右边，小的归在左边，然后二分地递归下去。*

平均时间复杂度：$O(n \log n)$

快速选择第 k 个元素：通过类似快速排序的方式，每次减掉一半待选择元素，可以在平均 $O(n)$ 时间内完成。

LinearSelect：将大序列划分成若干小序列，小序列排序，选出中位数之后选择众多序列中位数的中位数。然后类似快速排序，划分 L/E/G，递归地进行下去。

快速选择众数（超过一半的数）：在前缀中，如果元素 x 出现了恰好一半次数，那么

1）后面的众数 $m = x$，则 m 是众数。

2）后面的众数 $m = x$，则 m 有可能是众数。

不断向后缩小规模，可以选出众数 candidate，再 $O(n)$ 验证一次即可。