线性表

基本概念

1. 除“第一个”和“最后一个”元素外，每个元素都只有一个前驱和一个后继
2. 同一线性表中的元素都必须是同一类型的数据
3. 每个元素在表中的位置仅取决于它的序号
4. 可在表的任意位置进行插入和删除操作

线性表是逻辑的数据结构，要求元素之间的前驱和后继保持唯一性

线性表的顺序表示与实现

使用一组 物理内存地址连续 的存储单元依次存储线性表的数据元素。

知道顺序表的基址和数据元素，即可通过下标直接访问顺序表的任意元素

- 优点：

1. 不需要附加空间
2. 随机存取任意一个元素

- 缺点：

1. 很难估计所需空间的大小
2. 开始就要分配足够大的一片连续的内存空间
3. 更新（插入/删除）代价大 $O(n)$

线性表的链式存储与实现

用指针实现了用 物理上不相邻 的存储单元存放 逻辑上相邻 的一组元素

链表有带头结点的和不带头节点的链表，头节点不存储数据

单链表

- 特点：

1. 用一组 任意 的存储单元存储线性表中的数据元素
2. 通过指针保存直接后继元素的存储地址来表示数据元素之间的逻辑关系
3. 通过头指针（或者首节点）给出线性链表
4. 链表中节点空间是 动态分配 的
5. 插入、删除操作通过修改节点的指针实现
6. 只能顺序存取 元素，不能直接存取元素

循环链表

与单链表不同的是判空的条件从“后继是否为空”变成了 “后继是否为头节点”

循环链表能从任意节点开始找到任意节点

双向链表

相比单链表多了一个指向前驱的指针域

- 优点：
  1. 不需要提前分配空间
- 缺点：
  1. 查找需要循环
  2. 插入删除简单
  3. 需要附加空间

应用

一元多项式的表示与相加

用单链表来记录系数（扩展数据域记录指数），

栈

栈的表示与实现

栈是仅能在表头、表尾进行插入、删除操作的线性表，后进先出（LIFO）

顺序结构

空栈：栈顶指针==栈空间基址

栈满：栈顶指针 - 栈基址 == 栈大小

注意进栈操作，若栈满，则需要追加空间

链式结构

空战：栈顶指针为NULL

不存在栈满，可以随时拓展

栈的应用

数制转换

例如十进制数1348转换八进制：

1348　　1348/8 = 168　　1348 mod 8 = 4

168　　168/4 = 21　　168 mod 8 = 0

21　　21/8 = 2　　21 mod 8 = 3

2　　2/8 = 0　　2 mod 8 = 2

最后得到的八进制数为2304

显然用栈存储 (LIFO) 八进制数可以比较简易的完成输出任务

迷宫求解

是一种奇葩的 "DFS" 求解的方式，不过多描述

行编辑程序

原理就是将用户的输入放在栈顶，以最高的效率处理用户的删改操作

表达式求值

核心思想：

构造两个栈，一个是用于保存运算符的**OPTR**栈，一个是用于保存操作数或结果的**OPND**栈。其中OPTR栈底元素置#，OPND默认空栈。

如果是操作数，则直接进入OPND栈；如果是运算符，则先与OPTR栈顶元素比较优先级，若栈顶元素优先级较低，则直接将符号入栈；如果优先级相等（括号），则直接脱括号；如果栈顶元素优先级较高，则不断使OPTR栈顶元素出栈并进行运算，直到栈空或者栈顶元素优先更低为之，再把符号入栈。

优先级表：

	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	=
#	<	<	<	<	<		=

优先级最高的是 ），最低的是 （

栈与递归

函数在递归调用下，需要在内存中开辟一个运行栈，用以动态分配内存，以处理运行数据。

用作动态存储数据的存储区一般可以分为栈存储和堆存储。

队列

队列的表示与实现

队列是仅能在表头进行删除，表尾进行插入的线性表，先进先出（FIFO）。

顺序结构——循环队列

由基址，队头指针，队尾指针组成。队尾指针指向的是最后一个记录的后一块空间

带来问题：如果出队时队头指针直接后移，则会出现“假溢出”的情况，即表中还有空间，数据插入时就会溢出。为解决这一问题，可以固定队首，每次出队都使所有元素向着基址移动，但这样会增加时间支出。更好的解决方法是采用循环队列。

把队列设想为环形，让队尾接在队首之前。每当向队列中插入一个数据时，都把队尾指针变为（队尾指针+1）mod MAXSIZE。

队空：队头指针 == 队尾指针

队满：队头指针 == （队尾指针+1）mod MAXSIZE 实际上就是队头指针==下次要插入的位置

链式结构

队空：队头指针 == 队尾指针

队列的应用

模拟离散事件

诸如排队、争夺资源之类的问题

数组

数组的概念

数量固定，数据类型相同的元素组合在一起。通过数组下标可以直接访问数组中的元素。

数组的存储与实现

一般分为先行后列和先列后行两种映射方法

$Loc(a_{ij}) = Loc(a_{00}) + (N \times i + j) \times L$

上面的计算公式是针对从0开始，M行N列，每个数据元素长度为L的数组

稀疏矩阵

一般用三元组，再保存行数、列数和非零元素数来进行压缩保存

稀疏矩阵的转置

按列从小到大扫描nu次三元组，将扫描到的元素行号与列号互换后加入转置后的三元组。时间复杂度$O(nu \times tu)$。

为降低时间复杂度，引入num和$cpos$数组分别保存原矩阵中第col列的非零元数量和第col列第一个非零元在转置后的三元组的位置。时间复杂度为$O(2nu + 2tu)$

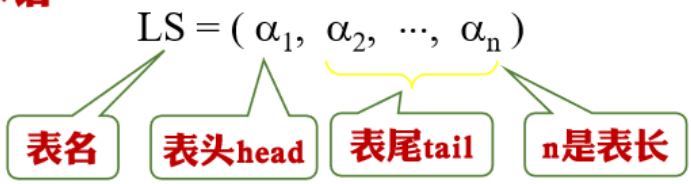
广义表

广义表的定义

$LS = (\alpha_1, \alpha_2, \ldots, \alpha_n)$

其中的α既可以是原子，也可以是广义表

广义表的术语



表头：LS的第一个元素称为表头
表尾：其余元素组成的表称为LS的表尾
表长：为最外层包含元素个数
深度：所含括弧的重数。原子的深度为 0，“空表”的深度为 1。

注意表尾是除了表头的一个表

广义表的存储结构

链表存储，由标志域和一个*union*组成。标志域标志着该节点是原子还是表，若是原子则*union*中只有一个原子结点的值；若是表则包含了指向表头和表尾的指针。

广义表与递归

广义表在定义上就有递归的特性，所以很多操作都可以用递归实现。都写过，这块就不多说了。

树

树的定义与基本术语

树是n个结点的优先集合，在任意一棵非空树中：

1. 有且仅有一个称为根的 *root* 节点

2. 其余结点可分为若干个互不相交的集合，且每个集合本身又是一棵树

显然这是个递归定义，递归在树的操作中也有着广泛的运用

二叉树

二叉树中每个结点最多有两棵子树，每个结点度小于2；左右子树不能颠倒

性质

1. 二叉树的第 i 层上至多有 2^{i-1} 个结点
2. 深度为 k 的二叉树至多有 2^k-1 个结点
3. 设 n_i 为度为 i 的结点的数量，则必有： $n_0=n_2+1$
4. 具有 n 个结点的 完全二叉树 的深度为 log_2n+1

二叉树的表示

可用顺序表示，但比较浪费空间；一般用二叉链表来表示；也可以用三叉链表，加入指向父节点的指针；还有静态二叉链表和双亲链表等

遍历二叉树

遍历

按某种搜索路径访问二叉树中的每个结点，且每个结点仅被访问一次

某个二叉树的遍历总是可以分为：访问根（D）、遍历左子树（L）和右子树（R）

1. 先序遍历： DLR
2. 中序遍历： LDR
3. 后序遍历： LRD

通过递归可以轻松实现上述三种遍历

遍历的非递归算法

用栈记录尚待遍历的结点或子树，以备以后访问

线索二叉树

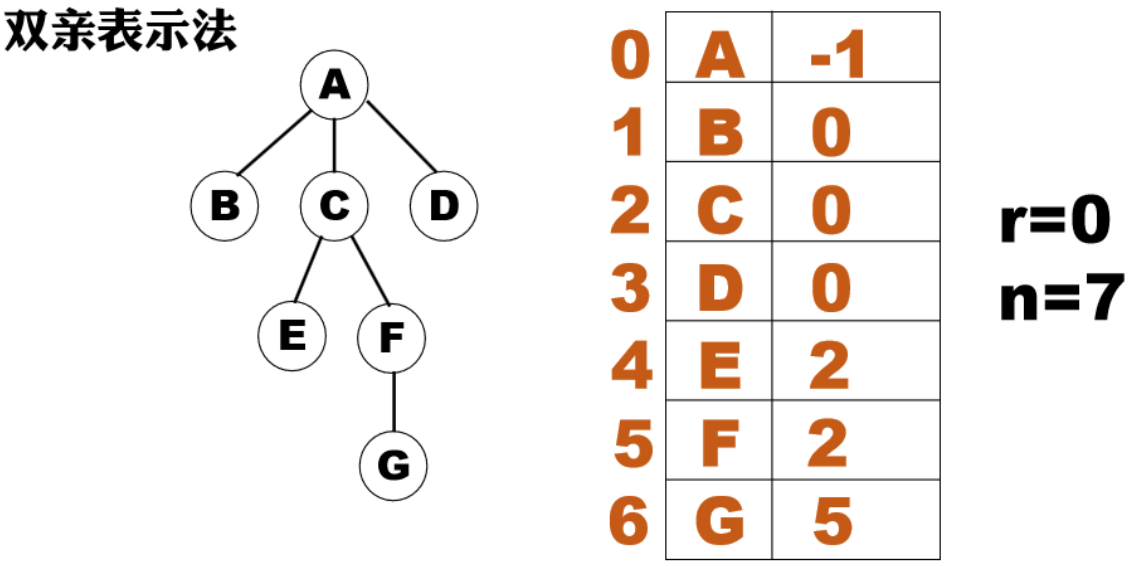
- 二叉树中度不为2的结点会有空的指针域，用空的指针域来保存线索
- 加入两个 tag 用来记录指针域存储的内容（到底是指向左、右孩子还是指向前驱或者后继）
- 指针域指向的前驱或者后继是谁取决于二叉树线索化的方式（前中后序）

二叉树的线索化也可以通过递归实现

树的存储结构

双亲表示法

保存每个结点的父结点的位置，来表示树中结点的关系



孩子表示法

保存数组每个结点的孩子结点的位置

- 多重链表

  类似二叉链表，采用多叉结点。结点分为定长的与不定长的

- 孩子链表

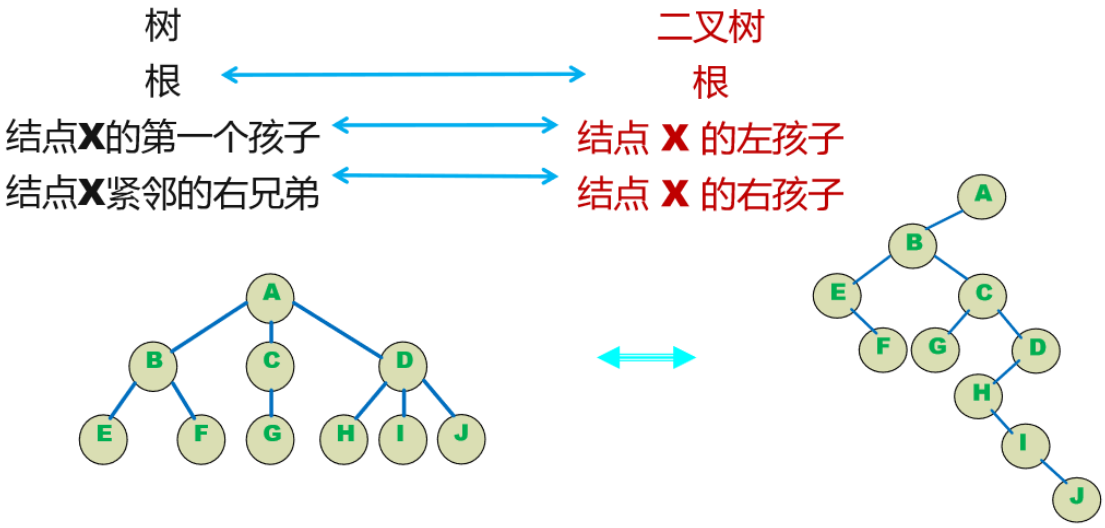
  把树中每个结点的孩子排列起来，连成一条链表存储在该节点后面

孩子兄弟表示法

每个节点有两个指针域，一个指向第一个孩子，一个指向下一个兄弟

树与二叉树

树与二叉树都能用二叉链表存储，以二叉链表做中介，可实现树与二叉树的转换



树与森林

- 森林

  树的集合

森林与二叉树的转换

将森林中树的跟看成兄弟，用树与二叉树转换的方法进行森林与二叉树的转换

广义表建树

广义表的表头存储根节点，表尾存储剩下的子树

最优二叉树（**Huffman**树）

- 路径长度

  从根节点到某一结点的分支数目

- 节点的权

  结点自带的某值

树的带权路径长度（WPL）

树的带权路径长度 = 树中所有叶子节点的带权路径之和

$$WPL=\sum_{k=1}^{n}W_kL_k$$

Huffman树的构造方法

1. 将所有结点并将其构造成只有根节点的树，按根节点权值从大到小排序（优先队列）
2. 取根节点权值最小的两个树作为左右子树合成新的树，新的根结点权值为这两个树根节点权值之和
3. 将新树按根节点权值大小顺序加入优先队列

图

图的术语与定义

图G是由点集$V(G)$和边集$E(G)$组成的，记为$G=(V,E)$

图分为有向图和无向图

- 邻接点

边的两个顶点

- 关联边

若边$e=(v,u)$，则称顶点v、u关联边e

- 度

顶点有 度 ，入度和出度。度 = 入度 + 出度

- 路径、回路

路径就是通路，回路就是回到起点的通路

- 连通图

若对任何两个顶点v,u都存在从v到u的路径，则称G是连通图（强连通图）。

- 子图

一个图G_1中所有的点与边都在图G中，则称G_1是G的子图

- 连通分量

无向图G的极大连通子图称为图G的连通分量。
对于一个连通分量，加入图G中任何不在该子图的顶点加入，子图不再连通

连通图只能有一个连通分量就是G本身，非连通图可以有多个连通分量

- 强连通分量

有向图D的极大强连通子图称为D的强连通分量

- 极小连通子图

在该子图删除任何一条边，都会使子图不再连通

- 生成树

包含无向图G所有顶点的极小连通子图称为G的生成树。生成树中显然不能有回路

图的存储结构

邻接矩阵

采用数组，下表表示顶点，值表示是否有边

邻接表

是一种链式存储结构，适用于边稀疏的图

用线性链表存储以同一顶点为起点的弧

十字链表（有向图）

分为顶点结点和弧结点

弧从弧尾指向弧头

- 顶点结点

*data*域；指向以该顶点为弧头的第一个弧结点（射入）；指向以该顶点为弧尾的第一个弧结点（射出）

- 弧结点

存储弧尾、弧头在表头数组中的位置；指向弧头相同的下一条弧；指向弧尾相同的下一条弧

邻接多重表（无向图）

- 顶点结点

*data*域；指向第一条依附于该顶点的边

- 边结点

标志域（标记是否被搜索过）；该边依附的两个顶点在表头数组中位置；两个指针，分别指向依附于左右两个顶点的下一条边

图的遍历

访问图中的所有顶点，并且使图中的每个顶点仅被访问一次

基本方法是DFS与BFS

图的最小生成树（MST）

问题的提出：用最小生成树可以解决建立通信网络最小代价的问题

Prim算法

将点集V分为U和$V-U$，初始时U为空，通过不断向U中添加结点与边来得到最小生成树

需要一个数组$closeEdge$来存储$V-U$中的顶点到U中的顶点的距离最小的边

1. 若 U 为空，则任选一节点加入 U
2. 选出 $closeEdge$ 中距离最小的那条边，将将该边和对应的 $V-U$ 中的结点加入 U
3. 更新 $closeEdge$ 数组
4. 重复以上步骤

时间复杂度为$O(n^2)$，适合处理边稠密的网的最小生成树

Kruskal算法

与$Prim$算法不同，$Kruskal$算法是选定了所有的点之后一个一个的添加边

1. 初始时最小生成树就包含图的所有顶点，每个顶点视为一棵子树
2. 选取关联的两个 顶点不在同一子树 且 权值最小 的边加入到最小生成树中
3. 重复以上步骤

时间复杂度为$O(eloge)$，适合处理边稀疏的网的最小生成树

有向无环图

拓扑排序

*AOV*网——用顶点表示活动

拓扑排序的图不能有环

- 算法

1. 在有向图中选一个入度为0的顶点且输出
2. 从图中删除该节点和所有以该结点为起点的弧
3. 重复前两步，直至所有顶点均输出或图中不存在入度为0的结点为止

关键路径

*AOE*网——用边表示活动，是一个带权的有向无环图

为求解某*AOE*网的关键路径，现在给出如下几个数组：

- $e[i]$，弧（活动）的最早开始时间
- $l[i]$，弧（活动）的最迟开始时间
- $ve[i]$，顶点（事件）的最早开始时间
- $vl[i]$，顶点（事件）的最迟开始时间

当$e[i]=l[i]$时，弧i显然是关键路径上的一条弧

几个数组间的关系

对于顶点（事件）i与弧（活动）$<i,j>$，标号为k，有$e[k]=ve[i]$

对与顶点（事件）j与弧（活动）$<i,j>$，标号为k，有$l[k]=vl[j]-time[k]$

计算ve、vl

1. 从源点开始，设 $ve[0]=0$，之后按拓扑顺序，
   $ve[j]=MAX\{ve[i]+time[<i,j>]\}$，其中 i 是任意有弧直接指向 j 的点
2. 从汇点开始，设 $vl[n-1]=ve[n-1]$，按逆拓扑顺序，
   $vl[i]=min\{vl[j]-time[<i,j>]\}$，其中 j 是 i 任意有弧直接指向的点

算法

1. 拓扑排序
2. 计算 ve、vl
3. 得到 e、l
4. 得出关键路径

最短路径

单源最短路径——Dijkstra算法

基本思想：按长度递增的顺序求解最短路径

算法

将图中所有顶点分为2组，第1组包含已求得最短路径的顶点，第2组包含尚未球的最短路径的顶点

1. 每次从第2组中选择与源点距离最小的顶点，加入第一组
2. 然后更新所有点到源点的距离
3. 不断重复，直至把所有顶点都加进第1组

是一个不断更新一个一维数组的过程。同时也可以添加记录最短路径的步骤

时间复杂度：$O(n^2)$

局限

不能解决含有负权边的图

每一对顶点之间的最短路径——Floyd算法

基本思想：逐个顶点进行试探

算法

1. 以邻接矩阵 S 的形式编码图 G ，对角线为0，不存在弧的置 ∞ ；用矩阵 P 中的元素 $p[i][j]$ 记录从点 i 到点 j 所经过的顶点；用矩阵 D 存储最短路径，矩阵 D 最初与 S 相等
2. 逐步尝试在原直接路径中 增加中间顶点 ，若加入中间点后路径变短，则修改值；否则维持原值
3. 所有顶点试探完毕

时间复杂度为$O(n^3)$

局限

不能解决含有负权回路的图

查找

基本概念

查找表

由同一类型的数据元素或记录构成的集合

- 查找表的基本操作

1. 查询
2. 检索
3. 插入
4. 删去

静态查找

在指定的表中查询某一个“特定”的数据元素是否存在，检索某一个“特定元素”的各种属性

动态查找

在查找的过程中同时插入表中不存在的数据元素，或从查找表中删除已存在的某个数据元素

查找性能

一般计算查找成功时的平均查找长度与查找失败时的比较次数

平均查找长度

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

P_i为查找表中第i个记录的概率

C_i为找到该记录是，曾和给定值比较过的关键字的个数

定义

$EQ(a,b)$：$a == b$

$LT(a,b)$：$a < b$

$LQ(a,b)$：$a \leq b$

静态查找表

顺序查找（顺序表）

在等概率查找的情况下$P_i = \frac{1}{n}$，$C_i = n - i + 1$

查找成功时$ASL = \frac{n+1}{2}$

折半查找（有序表）

还是等概率查找，$P_i = \frac{1}{n}$

假定有序表的长度 $n=2^h-1$，即等效的排序树有h层

查找成功时$ASL=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum_{j=1}^{h}j2^{j-1}\approx log_2(n+1)+1$

将C_i用数据所在的层数计算，每层有2^{j-1}个数，访问到第j层需要j层

分块查找

8.1 静态查找表

查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

排序二叉树

与堆排序中的堆很像，但是二叉排序树的 左子树所有节点 < 根节点 < 右子树所有节点

因为排序二叉树的这种特性，使查找某个元素变得十分简单

排序二叉树的插入

- 时机

  在二叉排序树查找不成功时

- 特点

一定是先添加叶子节点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子结点或右孩子结点

排序二叉树的删除

需要考虑删除的结点是叶子节点还是一个子树的根节点

1. 若删除的是叶子结点**p**

   直接修改父节点的指针即可

2. 删除的结点p只有左子树或右子树

   直接将子树替换被删除结点p

3. 删除的结点p的左子树和右子树均非空

   思想：找到左子树中最大的结点然后把右子树插在那个结点的右孩子处
   沿着p的左子树一直取右子树直至找到右子树为空的结点，把p的右子树插在那个结点上，之后用p的左子树代替p

排序二叉树的特点

- 含有n个节点的二叉排序树不唯一，与插入顺序有关
- 删除某个节点后，二叉排序树要重组
- 没有对树的深度进行控制

平衡二叉树（AVL树）

每个结点左右子树高度差绝对值不超过1的排序二叉树

定义平衡因子BF = 左子树深度 - 右子树深度，可见每一个结点都有平衡因子

若一个结点$|BF| > 1$，则称这个结点不平衡

平衡二叉树的平衡

设离插入结点最近的不平衡节点为A。

有四种不同的平衡操作

LL型

插入结点在A左子树的左子树下

单向右旋平衡处理

RR型

插入节点在A右子树的右子树下

单向左旋平衡处理

LR型

插入节点在A左子树的右子树下

先左后右双向旋转平衡处理

RL型

插入结点在A右子树的左子树下

先右后左双向旋转平衡处理

平衡二叉树的插入

是一种递归算法

1. 若AVL树为空则插入新结点e作为根节点
2. 若e的关键字与根节点相等，则不必插入
3. 若e的关键字小于根节点的关键字，且左子树中不存在与e关键字相同的节点，则将e插入到左子树上。若插入后左子树深度+1，则分情况处理：
    - 根节点平衡因子为-1：将根节点平衡因子改为0即可
    - 根节点平衡因子为0：将根节点平衡因子改为1，树的深度+1
    - 根节点的平衡因子为1：
        - 若左子树根节点平衡因子为1，进行单向右旋处理。之后将根节点与右子树根节点平衡因子置0，树的深度不变

- 若左子树根节点平衡因子为-1，进行先左后右两次旋转。之后修改左右子树根节点的平衡因子，树的深度不变

4. 若e的关键字大于根节点的关键字。。。。。。

B树与B+树

B树与其查找

产生原因

排序二叉树只能用于规模较小的数据查找（内存中）

而当数据涉及到了外存，则查找的主要时间消耗就变成了对外存的访问次数

基本思想

采用分块技术

将外存分为若干个固定大小的页块，另外开辟多个缓冲区，大小与一个页块相同。

每次内外存交换一个页块的内容，以减少访问外村的次数

B树的结构

一棵平衡的多路查找树

1. 每个节点至多有m棵子树
2. 若根节点不是叶子节点，则至少有两棵子树
3. 除根之外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树
4. 所有非终端结点都要包含以下数据

$$(n, a_0, k_1, a_1, k_2, a_2, \cdots\cdots, k_n, a_n)$$

其中 k_i 为关键字，且 $k_i<k_{i+1}$；a_i 为指向子树根结点的指针，且指针 a_{i-1} 所指子树中所有结点的关键字均小于 k_i，a_n 所指子树中所有结点的关键字均大于 k_n。n 为关键字个数，有 n+1棵子树。

5. 所有子树的叶子节点均在同一层，且不带信息

B树的查找

从根节点出发，沿指针搜索结点和在结点内进行顺序或折半查找，两个过程交叉进行。

实质就是先在一个结点内搜索，没搜索到就找到可能含有该记录的下一个结点，如此重复

- 若查找成功，返回指向该记录的指针
- 若查找失败，返回插入位置

对于含有N个关键字的B树来说，进行查找时涉及的节点数不超过$log_{\lceil m/2 \rceil}(\frac{N+1}{2})+1$

B树的插入

B树建树的过程就是一个不断插入结点的过程

1. 首先在最底层 的某个非终端结点添加一个关键字
2. 若插入后该结点含有的关键字少于m，则插入完成；若不少于m，则要进行分裂，添加新的结点
3. 一般来说，分裂的方式是把关键字不少于m结点的中间的记录抬到上面

B+树

构造

1. 有n棵子树的结点中包含n个关键字
2. 所有叶子节点包含了全部的关键字和指向这些关键字的指针，且从小到大顺序链接
3. 所有的非终端结点都只是索引

与B树的区别

B树搜索到了就可以停止，**B+树必须一直搜索到底**

B+树不只有指向根节点的指针，还有指向叶子结点链头的指针

哈希表

基本思想

一般的查找，效率与比较次数成负相关

而如果建立一种关键字与存储位置的对应关系，使得关键字不需要比较，直接通过这种对应关系就能找到该存储位置，就能大大提高查找的效率

这种对应关系称为**Hash函数**，映像的过程称为散列，所得存储地址称为哈希地址或散列地址

除了查找以外，Hash函数对于不同的关键字可能得到相同的地址，这就产生了冲突。这种冲突难以避免，所以一个哈希表还要想办法设定一个好的解决冲突的办法。

哈希表的构造方法

1. 直接定址法

   去关键字或关键字的某个线性函数值为哈希地址

2. 数字分析法

   去关键字的若干 数位 组成哈希地址

3. 平方取中法

   取关键字平方后的中间几位为哈希地址

4. 折叠法

   将关键字分割成位数相同的几部分，然后取这几部分的叠加和作为哈希地址

5. 除留余数法

$H(key) = key \ mod \ p$

6. 随机数法

   $H(key) = random(key)$

解决冲突的方法

开放定址法

$H_i = (H(key) + d_i) \ MOD \ m \quad i = 1, 2, \dots, k \ (k \leq m - 1)$

根据d_i取法不同，再散列的结果也不同：

1. $d_i = 1, 2, \dots m - 1$，线性探测再散列

   相当于是发生冲突时，一直把记录向后移动到第一个空存储位置时存下

2. $d_i = 1^2, -1^2, 2^2, -2^2, .., \pm k^2$，二次探测再散列

   相当于是向右试一试，向左试一试，平方递增

3. $d_i =$ 伪随机序列，伪随机探测再散列

   就是再次摇随机数

再哈希法

$H_i = RH_i(key)$

一直哈希到不发生冲突位置

增加了计算的时间

链地址法

将所有哈希地址相同的记录链接在同一链表中

公共溢出区

建立一个溢出表（相对于基本表），所有与基本表中关键字为同义词的记录直接无脑填入溢出表

哈希表的查找

1. 根据哈希函数求得哈希地址
2. 若该位置没有记录则查找失败
3. 否则比较关键字，若相等则查找成功
4. 若不相等则根据处理冲突的方法查找下一关键字

哈希表的查找性能

定义负载因子 $\alpha = n/m$。其中n是填入表中的结点数，m是散列表的空间大小

查找成功时

- 线性探测再散列

$$S_{nl} \approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

- 二次探测、随机、再哈希

$$S_{nr} \approx -\frac{1}{\alpha}ln(1-\alpha)$$

- 链地址

$$S_{nc} \approx 1 + \frac{\alpha}{2}$$

查找失败时

- 线性探测再散列

$$U_{nl} \approx \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$$

- 二次探测、随机、再哈希

$$U_{nr} \approx -\frac{1}{1-\alpha}$$

- 链地址

$$U_{nc} \approx \alpha + e^{-\alpha}$$

内部排序

简单插入排序

将一个已知的长度为k的有序序列变为长度为$k+1$的有序序列，如此反复直至全部有序

1. 查找 $R[k+1]$ 的插入位置i
2. 将i之后的记录全部后移一位
3. 将移动之前的 $R[k+1]$ 复制到i处

时间复杂度$O(n^2)$，空间复杂度$O(1)$，是 稳定 的排序

根据查找插入位置的不同可以分为以下几种排序方式：

直接插入排序

将R[0]置为哨兵，从R[k]处向前顺序查找$R[k+1]$的插入位置

折半插入排序

仍置$R[0]$为哨兵，查找插入位置时采取折半的策略

希尔排序

对数据先做“宏观”调整，再做“微观调整”。

1. 选取 增量d ，对数据分组，在各组内直接进行插入排序
2. 逐次减少增量 d ，作若干次步骤1，使待排记录基本有序
3. 对全部记录进行一次顺序插入排序

下面给出一个分组方式的例子：

对于记录$R[1]$，$R[2],\ldots,R[10]$，当d为5时，$R[1],R[6]$为一组，$R[2]$，$R[7]$为一组，以此类推

简单选择排序

堆排序

基本内容

对于堆来说，节点$R[2i]$和$R[2i+1]$是节点$R[i]$的子节点

父节点小于等于子节点的堆叫小顶堆，父节点大于等于子节点的堆叫小顶堆

堆排序的一般步骤（大顶堆）：

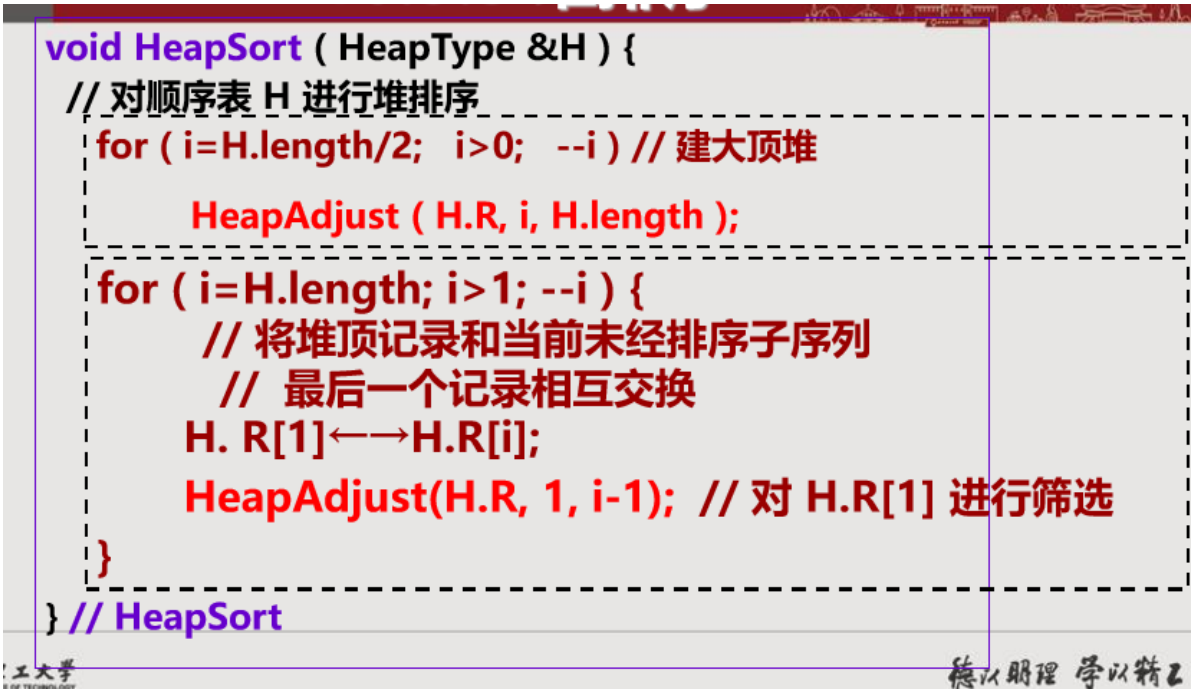
1. 建立初始大顶堆
2. 最大堆堆顶 $R[1]$ 具有最大的排序码，将 $R[1]$ 与 $R[n]$ 对调，将最大的交换到最后
3. 前面的 $n-1$ 个对象，使用堆调整算法重新建立最大堆，使具有最大排序码的记录上浮到 $R[1]$ 位置，称为 筛选
4. 重复执行步骤2、3直至排序完成

由上面的步骤可以看出堆排序是一个在建立大顶堆后不断维护大顶堆的过程，建立和维护大顶堆则需要一个 筛选 将非大顶堆转换成大顶堆，下面将给出这种筛选算法的介绍

筛选

对于$R[s\ldots n]$中的记录，除$R[s]$以外均满足堆，现在要使$R[s\ldots n]$满足堆

1. 使用一个变量rc暂存$R[s]$
2. 从s的左孩子开始进行筛选，直到记录$R[n]$
    1. 找出 $R[s]$ 的两个子节点中的较大的节点
    2. 若 $R[s]$ 比较大的子树节点大则可以直接结束筛选过程
    3. 否则将较大的子树节点记录上移， s 重新指向大子树根节点
3. 将rc中保存的记录插入到最终的s位置



最差的 时间复杂度为$O(nlogn)$，空间复杂度为$O(1)$，不稳定排序

冒泡排序

快速排序

基本思想

1. 通过一趟排序将待排记录分割成两部分
2. 一部分记录的关键字比另一部分小
3. 选择一个关键字作为分割标准，成为 *pivot*

算法过程

1. 置*low*指针于第一个记录，*high*指针为最后一个记录，将第一个记录设为*pivot*
2. 从表的两端交替扫描，直至两个指针相遇
    1. 先从高端扫描，找到一个比 *pivot* 键值小的记录，将 *high* 指针指向该记录位置，将 *low* 指针指向的内容与 *high* 指针指向的内容互换
    2. 再从低端扫描，找到一个比 *pivot* 键值大的记录，将 *low* 指针指向该记录位置，将 *low* 指针指向的内容与 *high* 指针指向的内容互换

3. 将$pivot$移动到low指针位置，返回该位置
4. 对返回的位置的左右两侧片段分别再做步骤1，2，3，直至所有记录返回

平均时间复杂度$O(nlogn)$，平均空间复杂度$O(logn)$，不稳定

改进的快速排序——三平均分区法

尽量将$pivot$取在中间位置

1. 取 $left$、$center$、$right$ 指针分别指向顺序表的左端、中间、和右端
2. 将 $left$、$center$、$right$ 指针指向的三个记录排序
3. 把指针 $rigth-1$ 所指向的记录设为 $pivot$ ，并与最中间的记录互换，然后执行快速排序

归并排序

算法过程

1. 将长度为 n 的序列看作为 n 个长度为1的小序列
2. 两两合并，得到 $\lfloor n/2 \rfloor$ 个长度为1或2的小序列
3. 不断两两合并，直至得到长度为 n 的有序序列为止

共进行$\lfloor logn \rfloor$趟归并，每趟n个记录，所以时间复杂度为$O(nlogn)$，空间复杂度为$O(n)$

基数排序

借助多关键字排序的方法对单关键字排序

外部排序

基于外部存储设备或文件的排序技术就是外部排序

基本思想

1. 建立用于外排序的 内存缓冲区 。根据他们的大小将输入文件划分为若干段，用某种内部排序方法对各段进行排序。这些经过排序的段叫做初始归并段或初始顺串。当他们 生成后就被写到外存中 去
2. 仿照内排序所介绍过的归并模式，把第一阶段生成的初始归并段加以归并，一趟趟地扩大归并段和减少归并段个数，直至最后归并成一个大归并段

2路平衡归并树

假设有$u=10000$个记录，分为$m=10$个段，每段有1000个记录。

假设每个物理块能容纳200个记录，则每段有5个物理块

1. 对各段内部进行排序
2. 第一趟归并，10个段归并成5个段
3. 第二趟归并，5（4+1）段归并成3（2+1）段
4. 第三趟归并，3（2+1）段归并成2（1+1）段
5. 第四趟，归为一段

所需时间：

$t_{ES}=m*t_{IS}+d*t_{IO}+s*u*t_{mg}=$ 内部排序总时间 + 外存读写总时间 + 内部归并总时间，s为归并趟数

因为$t_{IO}>>t_{mg}$，所以应减少外存读写次数d

为了减小读写次数d，应该减小总归并趟数s，又$s=\lceil log_k m \rceil$

所以要么增加归并路数**k**，要么减少初始段数**m**

多路平衡归并排序——归并的方法

做内部k路归并时，在k个对象中选择最小者，需要顺序比较k-1次。

每趟归并n个对象需要做$(n-1)(k-1)$次比较

则s趟归并总共需要比较次数为：$\lceil log_k m \rceil (k-1)(n-1)$

归并所需的时间反而随着k的增加而增加了，为避免这种情况，采用败者树

败者树

败者树是一棵完全二叉树，其中：

1. 每个叶节点存放各归并过程中当前参加比较的对象
2. 每个非叶节点 记忆两个子结点中对象关键码大的结点即败者

冠军就是根节点再上面的结点，每次比较结束后都输出冠军

初始化

败者树的初始化就是让k个叶子结点指向k路的最小结点即可

调整

当败者树输出了冠军，此时需要加入新的记录并做调整

将新的结点不断向上拿比较、更新即可

置换选择排序——生成初始归并段的方法

掌握基本思想即可

解决的问题：如何再不增加内存的情况下减少段数m

构造初始排序段

1. 从输入文件FI中把 w 个对象读入内存中
2. 利用败者树在 w 中选择一个关键码最小的对象记为 $MINMAX$

3. 将 $MINMAX$ 记录写到输出文件FO中
4. 若FI未读完，则从FI读入下一个对象到内存中
5. 从内存中所有比 $MINMAX$ 大的记录中，选择一个最先的作为新的 $MINMAX$
6. 重复3~5，知道内存中选不出新的 $MINMAX$ 为止
7. 重复2~6直到内存为空

具体：

FO	WA (6)	FI
空	空	51,49, 39, 46, 38, 29, 14, 61, 15, 30, 1, 48, 52, 3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76
空	51, 49, 39, 46, 38, 29	14, 61, 15, 30, 1, 48, 52, 3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76
29	51, 49, 39, 46, 38, 14	61, 15, 30, 1, 48, 52, 3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76
29,38	51, 49, 39, 46, 61, 14	15, 30, 1, 48, 52, 3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76
29,38,39	51, 49, 15, 46, 61, 14	30, 1, 48, 52, 3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76
29,38,39,46	51, 49, 15, 30, 61, 14	1, 48, 52, 3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76
29,38,39,46,49	51, 1, 15, 30, 61, 14	48, 52, 3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76
29,38,39,46,49,51	48, 1, 15, 30, 61, 14	52, 3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76
29,38,39,46,49,51,61	48, 1, 15, 30, 52, 14	3, 63, 27, 4, 13, 89, 24, 46, 58, 33, 76

这样得到的每一段都是都是排好序的

最佳归并树

掌握构造方法

缘由

- 由置换-选择生成的所得的初始归并段，其各段长度不等对平衡归并有什么影响？

将每个归并段的长度看作归并树的叶结点，那么归并树的带权路径长度WPL最小时，外存读写次数就最少，所需时间就越少。

所以我们在构造归并数时需要尽可能的使WPL小，参照Huffman树

基本概念

归并树是描述归并过程的k叉树，只有度为0和度为k的结点。

但最佳归并树可以有度不为k和0的结点

构造方法

为了使归并树称为一棵正则k叉树，可能需要补入空归并段

设需要增加u个空归并段，则：

$$u = \begin{cases} 0, & (m-1)mod(k-1) == 0 \\ k-1-(m-1)mod(k-1)，其他 \end{cases}$$

之后按照Huffman树的构造方式进行构造即可