计算机内部的线性存储结构来表示现实中复杂的多分支结构

**Preface：**

人们无法理解他没有经历过的事情。 ——尼采

A picture is worth a thousand works.

任何有难度的知识，都不是那么容易掌握的。

最淡的墨水，胜于最强的记忆。

同一个主题的东西 好几本书参和着读

**绪论 数学逻辑+工程直觉**

DataStructure：相互之间存在关系的数据元素的集合

∴需考虑 **空集, 真子集, 全集**

研究数据结构，就是为了提高数据的处理效率

程序设计中的循环结构，其实是对任意一个元素的操作

**原则：验证来源，反馈结果. 容错性：空？越界？输入合法？**

**逻辑结构：**

1、集合 2、线性结构

3、树：一对多 4、图：多对多

**物理结构（存储结构）：**

1.顺序存储结构（数组ArrayList）

2.链式存储结构（链表LinkedList）三维数组：页/行/列

**抽象数据类型ADT(Abstract Data Type) :** primitive + struct/class

目的：作为解释内存中信息含义的一种手段

存储结构的设计是灵活的，是否合理取决于其结构的运算操作是否合适、方便，时间复杂度.

抽象程度越高，通用性越高，当然越难理解。

——————————————————————————————————

**线性表** 节点与边组织形式（树/图）的基础

Definition: 相同类型的数据元素的有限序列。

**通常，存在一个工作指针（地址或下标）来标记位置**

*诠释：*

1. 序列，即元素有顺序的

首元素head无前驱，尾元素tail无后继，

其他元素有且仅有一个前驱next和后继prev

1. 一般地，有限个相同类型的元素

/\* List Fundamental Function :

\* 创建 初始化 清零 访问 查找 插入 删除 获得长度

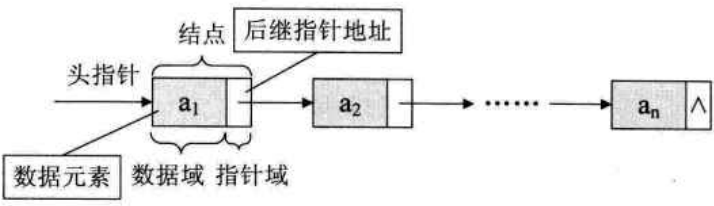
\*/

**头结点head：**

·头结点是为了操作的统一和方便而设定的，第一元素结点前的数据域无效

·有了头结点，对在第一元素结点的插入和删除，就与其他结点的操作统一

·不必要，但头指针是必要的



[拓展思维]

静态链表：用数组（内存空间）模拟链表，

Data -> Data , Index -> Address

分为 有效区 & 备用区，模拟malloc和free（手动模拟分配）

**循环链表Circular Linked List：**尾结点 指向 头结点，的单链表

差异：判断尾结点时，确定是否指向头结点

尾指针（指向尾结点的地址），可将两链表合成

**双向链表Double Linked List：**含前指针和后指针的结点

Template <typename T> Struct Node {

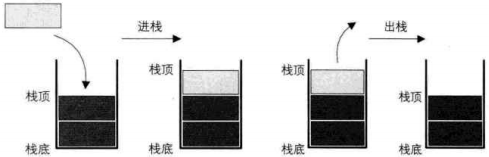
T data;

Node\* prior;

Node\* next;

};

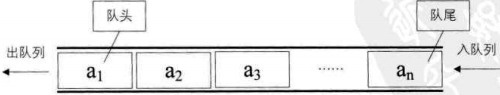
**栈与队列**



栈：仅在表尾插入与删除的线性表,FILO

**应用：**

相邻元素的关系压缩(2048, 祖玛游戏)、记录刚才发生过的事(Undo)、等待后续(中缀表达式求值)



队列：在表尾插入, 表头删除的线性表,FIFO

**队列的顺序存储结构：**

循环队列, 删除不用O(n), 避免“假溢出”,

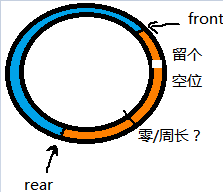
实现：

front指针标记首元素，rear指针标记待插入位置。

定义一个变量count，记录结点数，以判断空 | 满

有周期性，可把此结构看作给定半径的圆环，利用余数。

length = (rear – front + capacity) % capacity



（余数：指从零到该点 抛开周期性后 的长度 相对于 周长 还剩多长）

**队列的链式存储结构：**

Front -> 头结点、rear -> 尾元素

入队： 尾插入。Rear->next = new T;

出列： 头删除。Front->next = front -> next > next;

**应用：**

排队论

**集合Set**

元素间不存在关系，重点是存在性问题。

但在实际过程中，往往添加偏序关系来方便操作（线性表 或 并查集……）

如：装有课程的集合中，会加以编号，来排序。

多重集合Multiset：元素可重复出现的集合。（元素自带计数器，便可实现）

**实践：**

连续的内存空间(数组) 来实现简易集合：

Structure: boolean flag[n] # 标记是否忽略

T element[n] # 元素的容器

1. 删除元素(子集)： 把flag标记逻辑取反
2. 添加元素： 先判断是否重复、再插入到尾部

**应用于** 排列组合、Dijistra最短路 等问题的子步骤

**并查集DisjointSet**

不相交集（等价类）的数据组织方式（树，牵一发而动全身）

（parent-link）

+array 记录当前下标的元素对应的父节点位置，规定根节点 记录的是-1

-find() 查找所在等价类的根节点

-union() 合并不相交集

**合并优化：**

为了降低树的深度，by size/ by height（顺便记录树的 节点个数 或 高度 作为判断的依据）

**路径压缩：**

使每个节点的父节点 变为 根节点

(兼容性)by-size √ ， by-height？

∵路径的压缩，不改变大小，可能改变高度

**散列表HashTable**

数组是通过整数下标进行索引，而字典是通过关键码（key）进行索引

描述的是key-value映射的数据结构，best case：单射且满射

**hash函数** postion = hash(key)

信息指纹：唯一的标识(ID), 如：MD5、SHA-1

根据特征，把一大段信息有损压缩成一个点，且尽量互不重复。

**要求:** 运算简单、均匀、少冲突、利用率高

**考虑：**关键字长度、计算散列地址的时间、散列表长、关键字的分布、查找频率

1 线性同余： h(key) = key mod capacity + delta

关键在于容量capacity的选取

（不含20以内的质因子的合数 && 不太接近2t的质数）

2 直接定址：（线性函数）



3 乘法取整：h(key) = floor(m\*(\*key mod 1))

4 预定义映射表

·数字分析：抽取部分来计算位置

·平方取中：平方后选取中间部分的数字

·折叠法：将数字分隔成若干个位数相等的部分，然后运算（相加）

字符串-> "word" // 假定编码为12 8 6 15

① hash("word") = 0xC86F

② hash("word") = 12+8+6+15 = 41 = 0x0029

**散列冲突：Collision**

不同的数据记录映射到同一个位置 or 不是同义词却需要争夺一个位置

**解决方案： (关键区分)** 把冲突元素存储在数组之外还是之内

1 开放地址法(探测法) h = （hashCode(k) + d(i)）mod capacity

生成增量序列，沿着此序列，逐个位置探测，直到发现一个闲置的位置

增量序列:

线性探测：di=i

二次探测：di=12、-12、…、q2、-q2，相当于双向寻找空位

2 链接法

对冲突位置建立同义词的链表

3 公共冲突区

将所有冲突元素存储于另外一个表中

4 换个哈希函数

**性能分析Performance**

装载因子load factor = 0.72为比较好

哈希函数hash function

冲突的处理collision resolution

**Dictionary字典 / Map 映射**

The fundamental operation for a dictionary is finding a record that matches a given key. 存储key-value pair，可实现查找、删插等功能

**应用：**

简单高效的查询容器

**字符串**

Definition:零到多个字符组成的有限序列。

元素全是字符的线性表，因其常用，特殊处理

**子串**：主串中任意个数的连续字符组成的子序列

**串的比较**：（实例：查英文字典）

遍历寻找第一个不同的字符，比较其编码对应的数值。(字典序)

**基本操作**：

复制、拼接、子串、替换、查找

获得长度、插入、删除、清空、赋值

**树Tree**(层次结构)

**定义:**

非空树中，1）有且仅有一个**根结点** 2）其余结点可分为互不相交的有限集，其中每个集合本身又是一棵树，被称为根的**子树**（递归定义决定了递归解法）

**树中还是树，只不过，我们把空树作为最小单位的树（基线条件），只有根节点的树看成节点（简单解决的问题）**

结点拥有的子树数目，叫**度degree**。度为0的点，叫**叶节点leaf**。

internal内点：非叶节点

外部路径长度([external path length](http://soj.sysu.edu.cn/show_problem.php?pid=1011&cid=1624))： 所有叶节点的高度之和

树中结点的最大层次，叫**深度depth**

森林：多个不相交的树的集合

n个节点的树，含有n-1条边

full tree：正则树（每个节点分支数一致）

**缩进表示: (XML/JSON**)

A\_\_\_\_\_\_\_\_\_\_

B\_\_\_\_\_\_\_

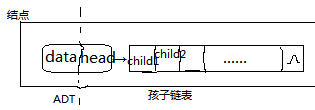
C\_\_\_\_\_\_\_

D\_\_\_

**双亲表示法：Parent**

规定根结点无父节点，其余结点数据类型中包含其父节点的地址

**孩子表示法：Child** (邻接表)



节点除了数据外，还附带着其孩子的地址。

**二叉树(Binary Tree)**

引导问题：大家猜一个商品的价格，100元以内，只能猜7次，每次选择后，只被告知“太高”|“太低”.（实际上，就是二分查找法）

递归定义注定了递归算法：

一个**根节点**下有两棵不相交的左右**二叉子树**.

1. 每个节点最多有2棵子树
2. 左右顺序不可颠倒
3. 即使节点只有一棵子树，也要分左右

**特例**:

1. 斜树（即线性表）

每个节点只有一棵（左/右）子树

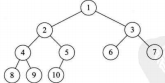
1. 满二叉树

每个分支节点都有左右子树，叶节点都在同一层.

1. 完全二叉树（逐层从左往右填满节点, n2+1=n0）

节点编号与同深度的满二叉树节点编号一致，斜树之上，满树未满。

（同节点数的二叉树中，深度最小，递归执行次数最小）



相同节点数的二叉树，完全二叉树深度最小。

**性质：**

1 在第i层，至多有2i-1个结点（根节点在第1层）

2 深度为k的二叉树，至多有2k – 1个结点（等比数列的求和）

3 叶节点的个数 = 度为2的节点的个数 + 1

4 具有n个节点的完全二叉树深度 [log2n] + 1 （满树的个数公式逆推）

5 完全二叉树，假定从上到下，从左到右依次顺序编号1~n，

对任一节点i，当i = 1, 则为根节点，i > 1, 其双亲为节点(i / 2), 若存在孩子，则2i为长子，2i+1为兄弟

**完全二叉树的顺序存储结构**

从根节点开始，从左到右依次编号1~n，通过二倍关系来表示父子关系。

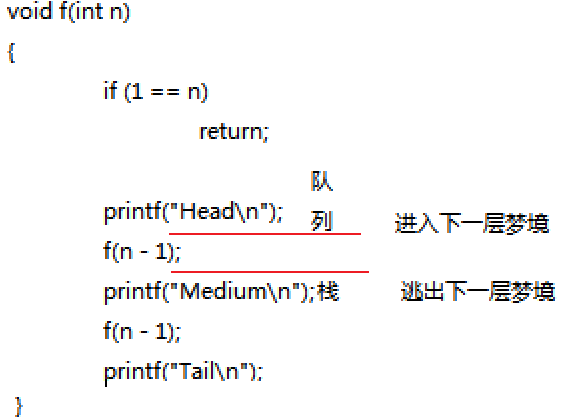
**二叉树的链式存储结构**

节点数据类型，数据域+指针域（指向孩子/双亲）

**二叉树的遍历：**

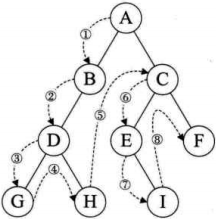
[不断缩小树的规模, 叶节点就是只有根节点的树,无子空间,递归界限无效]

递归的过程就是树，无分支为止。



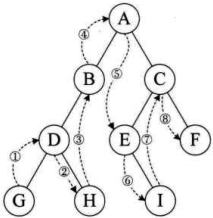
**前序遍历()**

若树非空，则访问根节点，再前序遍历左子树，最后前序遍历右子树



**中序遍历（横向的相对顺序）**

若树非空，则中序遍历左子树，再访问根节点，最后中序遍历右子树



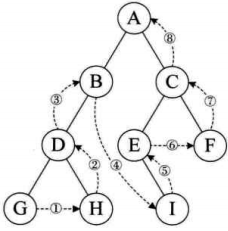
**后序遍历**

若树非空，则后序遍历左子树，再后序遍历右子树，最后访问根节点

**逆波兰表达式：**

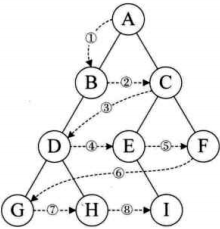
根节点为操作符，子树为操作数，后序遍历的序列。

∵计算机中function：add(a, b)总是需要先确定操作，再处理数据



**层序遍历**

若树非空，则从上到下逐层遍历，同一层中，从左到右。



**非递归遍历：**

模拟递归返回，保存上一级的地址于栈中，push、pop来模拟递归

**二叉树的建立**

准备工作：对无分支的结点，人为设定虚结点. 方便制定递归界限。

递归前序遍历的思路，进行建树。

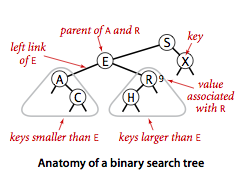
**线索二叉树：**某种次序遍历得到的序列

为了解决普通二叉树节点中空指针域的闲置浪费问题，且提高遍历效率，ThreadedBinaryTree应运而生，以某种次序遍历使其线索化，成为双向循环链表.

**二叉查找树(Binary Sort Tree)**

【变小向左, 变大向右】

左二叉排序子树任意节点的值均小于根节点的值，右二叉排序子树任意节点的值均大于根节点的值。



**查找：** 二分法

**插入：** 查找到山穷水尽时，补一个

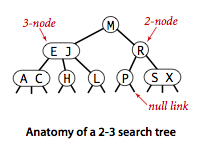
**删除：** 1、叶节点：直接删除

2、有1个子树的节点： 子树根节点顶上

3、有2个子树的节点： 右子树的最小值顶上 或 左子树的最大值顶上

**红黑树(Red Black Tree)**

src：<https://algs4.cs.princeton.edu/33balanced/>



definition A 2-3 search tree is a tree that either is empty or:

①A 2-node, with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys

②A 3-node, with two keys (and associated values) and three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys and a right link to a 2-3 search tree with larger keys.

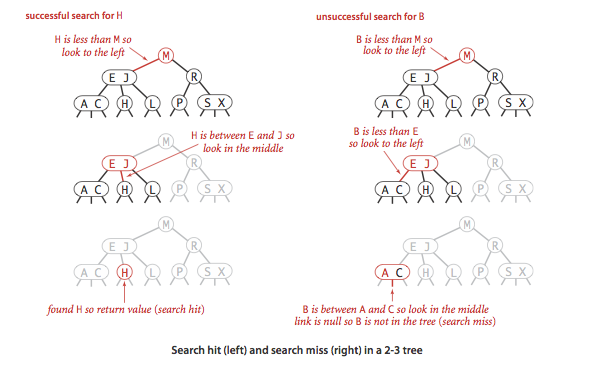
平衡定义：A perfectly balanced 2-3 search tree is one whose null links are all the same distance from the root.

个人推论：

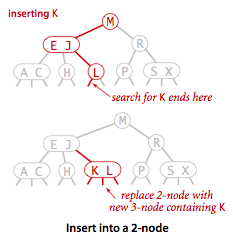
1.任一节点的link要么全null，要么全指向。所有叶节点在同一层。因为如果存在部分非空的link，则存在不同层的null link到根节点的距离不同，违反平衡定义。

2.新增的key都插入于叶节点。因为如果插入的是中间节点，则null link到root的距离违反平衡定义。

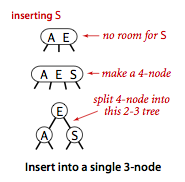
search搜索：



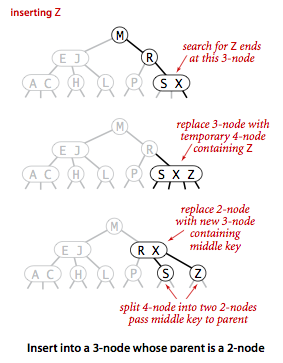
* 1. Insert into a 2-node



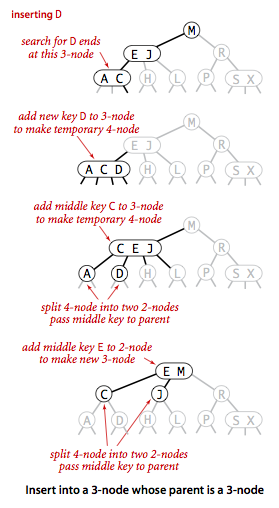
(demo) Insert into a tree consisting of a single 3-node



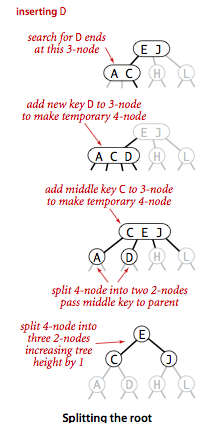
* 1. Insert into a 3-node whose parent is a 2-node



* 1. Insert into a 3-node whose parent is a 3-node



* 1. Splitting the root



1. 维护平衡结构的思路：提拔middle，连接left和right
2. Search and insert operations in a 2-3 tree with N keys are guaranteed to visit at most lgN nodes

**堆Heap(优先队列)**

complete tree，节点逐层堆积

根节点是最值点（max-heap、min-heap），带有左右子堆

本质：堆的根节点是最值，其子堆的根节点是等待成为根节点的

(C++STL中的优先队列priority\_queue)

**堆的过滤(percolate)**

下滤： 与最大的子节点交换，保证交换后的根节点大于子节点，直到不能再下降

上滤： 不断与上面比较，直到不能再上升（上位）

**构造**： 循环或递归，都是由规模最小的堆（节点）开始建立

O(n)的make-heap是 for i=size[array]/2 to 1 do ： maxHeapify(array, i); 【自下而上】

而O(nlogn)的make-heap是 for i = 1 to n do ： insert(array, array[i]); 【自上而下】

删除堆顶：与堆尾值交换，然后更换后的堆顶不断下滤

插入堆尾：不断上滤

（小优化：先设定需调整的节点为空，通过覆盖来调整位置，最后填上数据）

d叉堆：完全d叉树，可降低树的深度