# 数据结构

数据结构(data structure)是带有结构特性的数据元素的集合。它研究的是数据的[逻辑结构](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E7%BB%93%E6%9E%84/9663235" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/_blank)和数据的物理结构以及它们之间的相互关系，并对这种结构定义相适应的运算，设计出相应的算法，并确保经过这些运算以后所得到的新结构仍保持原来的结构类型。简而言之，数据结构是相互之间存在一种或多种特定关系的数据元素的集合，即带“结构”的数据元素的集合。“结构”就是指数据元素之间存在的关系，分为逻辑结构和存储结构

* 数据：所有能被输入到计算机中，且能被计算机处理的符号的集合。是计算机操作的对象的总称。
* 数据元素：数据（集合）中的一个“个体”，数据及结构中讨论的基本单位
* 数据项：数据的不可分割的最小单位。一个数据元素可由若干个数据项组成。
* 数据类型：在一种程序设计语言中，变量所具有的数据种类。整型、浮点型、字符型等等
* 逻辑结构：数据之间的相互关系。

1. 集合 结构中的数据元素除了同属于一种类型外，别无其它关系。
2. 线性结构 数据元素之间一对一的关系
3. 树形结构 数据元素之间一对多的关系
4. 图状结构或网状结构 结构中的数据元素之间存在多对多的关系

* 物理结构/存储结构：数据在计算机中的表示。物理结构是描述数据具体在内存中的存储（如：顺序结构、链式结构、索引结构、哈希结构）等
* 在数据结构中,从逻辑上可以将其分为线性结构和非线性结构
* 数据结构的基本操作的设置的最重要的准则是,实现应用程序与存储结构的独立。实现应用程序是“逻辑结构”，存储的是“物理结构”。逻辑结构主要是对该结构操作的设定，物理结构是描述数据具体在内存中的存储（如：顺序结构、链式结构、索引结构、希哈结构）等。
* 顺序存储结构中，线性表的逻辑顺序和物理顺序总是一致的。但在链式存储结构中，线性表的逻辑顺序和物理顺序一般是不同的。
* 常用算法：数据结构研究的内容：就是如何按一定的逻辑结构，把数据组织起来，并选择适当的存储表示方法把逻辑结构组织好的数据存储到计算机的存储器里。研究的目的是为了更有效的处理数据，提高数据运算效率。数据的运算是定义在数据的逻辑结构上，但运算的具体实现要在存储结构上进行。一般有以下几种常用运算：

1. 检索。检索就是在数据结构里查找满足一定条件的节点。一般是给定一个某字段的值，找具有该字段值的节点。
2. 插入。往数据结构晕增加新的节点。
3. 删除。把指定的结点从数据结构中去掉。
4. 更新。改变指定节点的一个或多个字段的值。
5. 排序。把节点按某种指定的顺序重新排列。例如递增或递减。

## 数组(Array)

### 概念

所谓数组，是有序的元素序列。 [1]  若将有限个类型相同的变量的集合命名，那么这个名称为数组名。组成数组的各个变量称为数组的分量，也称为数组的元素，有时也称为下标变量。用于区分数组的各个元素的数字编号称为下标。数组是在程序设计中，为了处理方便， 把具有相同类型的若干元素按无序的形式组织起来的一种形式。 [1]  这些无序排列的同类数据元素的集合称为数组。数组是用于储存多个相同类型数据的集合。数组是最基本的数据结构，很多其他的数据结构都要靠数组来实现。

java中数组

数组也是除了基本类型以外的引用类型，默认继承了Object类，所以在数组使用之前一定要开辟空间（实例化），如果使用了没有开辟空间的数组，则一定会出现 NullPointerException 异常信息，当我们数组采用动态初始化开辟空间后，数组里面的每一个元素都是该数组对应数据类型的默认值，在 Java 中提供有一种动态取得数组长度的方式：数组名称.length。

动态初始化

数据类型 [] 数组名称 = new 数据类型[长度]；

数组静态初始化

|  |  |
| --- | --- |
| 简化格式： | 数据类型 数组名称 = {值, 值,…} |
| 完整格式： | 数据类型 数组名称 = new 数据类型[] {值, 值,…} |

### 优缺点

优点   
1、按照索引查询元素速度快   
2、按照索引遍历数组方便

缺点：  
1、数组的大小固定后就无法扩容了   
2、数组只能存储一种类型的数据   
3、添加，删除的操作慢，因为要移动其他的元素。

### 适用场景

频繁查询，对存储空间要求不大，很少增加和删除的情况。

### 排序算法

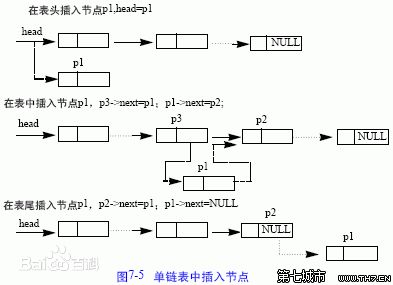


java.util.Arrays:java自带对数组的处理工具类，如排序、搜索等算法的实现。

## 链表( Linked List)

### 概念

链表是物理存储单元上非连续的、非顺序的存储结构，数据元素的逻辑顺序是通过链表的指针地址实现，每个元素包含两个结点，一个是存储元素的数据域 (内存空间),另一个是指向下一个结点地址的指针域。根据指针的指向，链表能形成不同的结构，例如单链表，双向链表，循环链表等。



### 算法实现

自定义单链表实现 jdk内置实现:LinkedList部分源码链表实现（双链表）

### 优缺点

优点

链表是很常用的一种数据结构，不需要初始化容量，可以任意加减元素；

添加或者删除元素时只需要改变前后两个元素结点的指针域指向地址即可，所以添加，删除很快；

缺点

因为含有大量的指针域，占用空间较大；

查找元素需要遍历链表来查找，非常耗时。

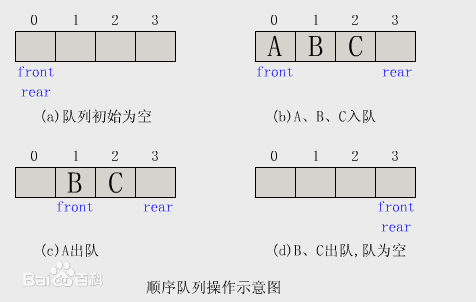
### 适用场景

数据量较小，需要频繁增加，删除操作的场景

## 队列(Queue)

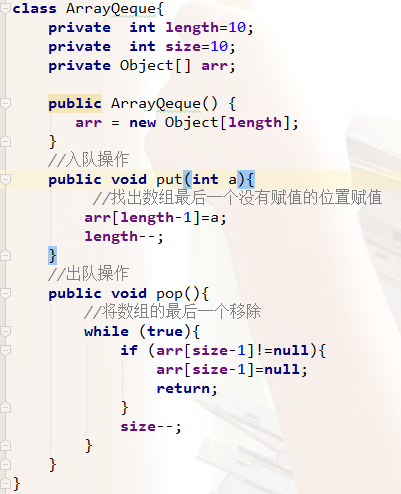
### 概念

队列是一种特殊的[线性表](https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E8%A1%A8" \t "https://baike.baidu.com/item/%E9%98%9F%E5%88%97/_blank)，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。队列的数据元素又称为队列元素。在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先（FIFO—first in first out）[线性表](https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E8%A1%A8" \t "https://baike.baidu.com/item/%E9%98%9F%E5%88%97/_blank)。



### 算法实现

数组实现、链表实现，以数组实现示例



jdk中有对应的Qeque接口，不同的实现方式对应不同的类。

### 使用场景

因为队列先进先出的特点，在多线程阻塞队列管理中非常适用。

## 散列表(Hash)

### Hash

一般翻译做散列、杂凑，或音译为哈希，是把任意长度的[输入](https://baike.baidu.com/item/%E8%BE%93%E5%85%A5/5481954" \t "https://baike.baidu.com/item/Hash/_blank)（又叫做预映射pre-image）通过散列算法变换成固定长度的[输出](https://baike.baidu.com/item/%E8%BE%93%E5%87%BA/11056752" \t "https://baike.baidu.com/item/Hash/_blank)，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的[消息摘要](https://baike.baidu.com/item/%E6%B6%88%E6%81%AF%E6%91%98%E8%A6%81/4547744" \t "https://baike.baidu.com/item/Hash/_blank)的函数。

### [哈](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E8%A1%A8/10027933" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)希表概念

Hash table，也叫[散列表](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E8%A1%A8/10027933" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)，是根据关键码值(Key value)而直接进行访问的[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/1450" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做[散列函数](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B0/2366288" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)，存放记录的[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84/3794097" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)叫做[散列表](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E8%A1%A8/10027933" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)。给定表M，存在函数f(key)，对任意给定的关键字值key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表M为哈希(Hash）表，函数f(key)为哈希(Hash) 函数。

### 哈希表优点

在实现编程中，常常面临着两个问题：存储和查询。存储和查询的效率往往决定了整个程序的效率。而我们常见存储数据的数据结构比如线性表，树等。数据在结构中的位置都是不明确的，当我们在这些数据结构中要查询一个数据，都避免不了的执行查询算法，去遍历数据结构，拿关键字和结构中的数据一一比较，从而得到想要的数据。整个查询的效率就决定了程序的执行效率，查询效率又依赖查找过程中所进行的比较次数。所以我们就希望能不能不通过比较就能获得我们想要的结果呢?答案是有的，不经过任何比较，一次存储便能取得所查记录。但这就必须在记录的存储位置和它的关键字之间建立一个确定的对应关系f。使得每个关键字和结构中一个唯一的存储位置相对应。这个关系就是我们所说的哈希函数f（x）。在这个思想上建立起来的表就称为哈希表。

### 哈希表理解

关键字(Key):哈希表是通过一个信息来查找另一个信息，将这两个信息在哈希表从形成映射关系，而关键字则是我们要提供的信息

值(Value):值是我们想要获取到的信息，如map.put(key,value)

哈希函数:f(x)哈希函数是用来构成哈希表的工具，也是哈希表的核心思想，是关键字和对应数据的存储位置的一个映射关系，通过把关键字代入哈希函数中进行计算，可以得到关键字所对应的数据在哈希表中的存储位置即哈希地址。

哈希地址:=f(key)哈希地址记录的是我们所需要的数据在哈希表中的存储位置，哈希地址只是表示查找表中的存储位置，不是实际的物理存储位置

关键字、值、哈希函数、哈希地址、哈希表之间的关系

哈希表是通过哈希函数来构建的，我们把哈希函数想象成数学中的函数f()。而函数中的X就是关键字Key，既f(x)。然后将关键字带入一个公式中，如f（x） = x \* 2 + 3 。然后经过运算就可以求出一个值，这个值表示要查询的数据(Value)在哈希表中的存储位置，也就是哈希地址。而记录这整个Key-Value信息的表就是哈希表。

我们现在把一个抽象的哈希表具体表现成下面的形式，可以说下面就是一个哈希表。我们把它想象成一本电话簿，该电话簿用拼音首字母来区分数据，数据记录了具体的人名和其电话号码。换成哈希表的概念，就是说ABCD是哈希地址，用于记录值所存储的位置。人名为查询关键字(Key)。电话号码是我们需要获得的数据(Value)。

为了简化流程且易懂，我们这里假设了哈希地址就是关键字的拼音首字母大写，那么哈希函数就是f（艾力）= A，f(包三) = B等…。我们通过将关键字(包三)代入计算公式(哈希函数)中，得到包三的电话号码所在的位置，既哈希地址(B)。那我们就能通过位置直接获得我们想要的数据，而不需要遍历比较。

| 哈希地址 | Key and Value |
| --- | --- |
| A | 艾力 13912345678 |
| B | 包三 15823457890 |
| C | 成五 15823457890 |

### 哈希冲突

哈希冲突就是key1!=key2.但key1和key2所对应的数据的存储位置都一致，既哈希地址一致。这就是哈希冲突。我们看来下面的哈希表：

| 哈希地址 | Key and Value |
| --- | --- |
| A | 艾力 13912345678 ；爱丽丝 15823787890 |
| B | 包三 15823457890 |

如上表我们可以看到，艾力和爱丽丝两个人的拼音首字母大写都是A,这就意味着这两个人的电话号码都存储在哈希表(电话簿)的A区，这就存在冲突了。所以在构建哈希表中，最为重要的核心就是哈希函数的设计。哈希冲突只能尽可能减少，是不能完全避免的

处理冲突

1. [开放寻址法](https://baike.baidu.com/item/%E5%BC%80%E6%94%BE%E5%AF%BB%E5%9D%80%E6%B3%95" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)：Hi=(H(key) + di) MOD m,i=1,2，…，k(k<=m-1），其中H(key）为[散列函数](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B0" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)，m为[散列表](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E8%A1%A8" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)长，di为增量序列，可有下列三种取法：

1.1. di=1,2,3，…，m-1，称线性探测再散列；

1.2. di=1^2,-1^2,2^2,-2^2，⑶^2，…，±（k)^2,(k<=m/2）称二次探测再散列；

1.3. di=[伪随机数](https://baike.baidu.com/item/%E4%BC%AA%E9%9A%8F%E6%9C%BA%E6%95%B0" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)序列，称伪随机探测再散列。

2. 再[散列法](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E6%B3%95" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)：Hi=RHi(key),i=1,2，…，k RHi均是不同的[散列函数](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B0" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)，即在同义词产生地址冲突时计算另一个散列函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算时间。

3. 链地址法（拉链法）,java中的HashMap就是采用这种方法

4. 建立一个公共溢出区

### 哈希函数构造方法

如何判断一个哈希函数的优劣：

* 能否将关键字均匀影射到哈希空间上
* 有无好的解决冲突的方法
* 计算哈希函数是否简单高效

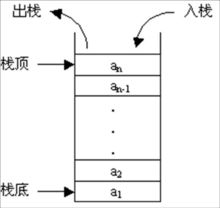
常见构建哈希函数的六个方法

* 直接定值法：即H(key)=key或H(key) = a·key + b
* 数字分析法：找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址
* 平方取中法：当无法确定关键字中哪几位分布较均匀时，可以先求出关键字的平方值，然后按需要取平方值的中间几位作为哈希地址
* 折叠法：将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址
* 除留余数法：选择一[随机函数](https://baike.baidu.com/item/%E9%9A%8F%E6%9C%BA%E5%87%BD%E6%95%B0" \t "https://baike.baidu.com/item/%E5%93%88%E5%B8%8C%E8%A1%A8/_blank)，取关键字的随机值作为散列地址，通常用于关键字长度不同的场合。
* 随机数法：f (key) = random (key)

java中Object的hashCode就是一个哈希函数，方法是本地方法，具体用的什么哈希函数综合了上面几种算法，String、Integer等则重写了hashCode，可以查看源码得知。

## 栈( Stack)

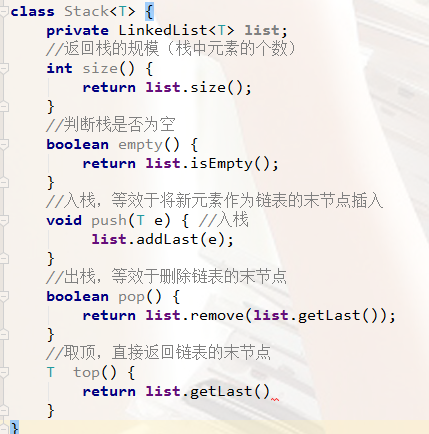
### 概念

栈是一种特殊的线性表，仅能在线性表的一端操作，栈顶允许操作，栈底不允许操作。 栈的特点是：先进后出，或者说是后进先出，从栈顶放入元素的操作叫入栈，取出元素叫出栈。   


jdk中有Stack类实现栈的相关算法。

### 算法实现

以链表实现



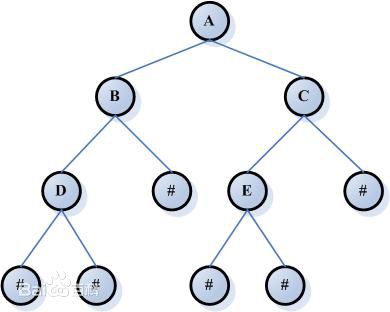
jdk中有Stack类实现。

### 应用场景

先进后出的特性，所以栈常应用于实现递归功能方面的场景，如ctrl+z撤销就是一个栈的应用

## 树( Tree)

树状图是一种[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/1450" \t "https://baike.baidu.com/item/%E6%A0%91/_blank)，它是由n（n>=1）个有限结点组成一个具有层次关系的[集合](https://baike.baidu.com/item/%E9%9B%86%E5%90%88" \t "https://baike.baidu.com/item/%E6%A0%91/_blank)。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。每个结点有零个或多个子结点；没有父结点的结点称为根结点；每一个非根结点有且只有一个父结点；除了根结点外，每个子结点可以分为多个不相交的子树



各种树的算法是很复杂的，不必钻牛角尖，关键要懂各种树的概念、用途即可，如果要手写算法，那别去了。

### 二叉排序树

又称[二叉查找树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%9F%A5%E6%89%BE%E6%A0%91/7077965" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)，二叉排序树或者是一棵空树，或者是具有下列性质的[二叉树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)：

（1）若左子树不空，则左子树上所有节点的值均小于它的[根节](https://baike.baidu.com/item/%E6%A0%B9%E8%8A%82" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)点的值；

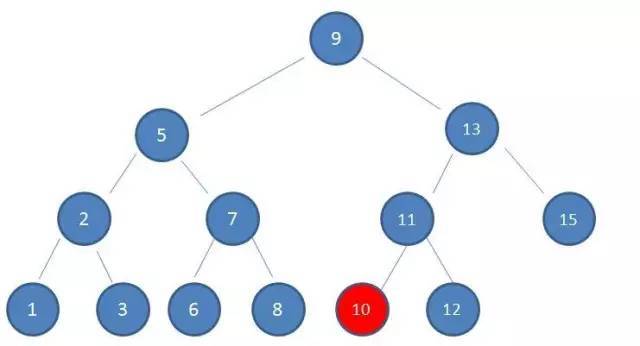
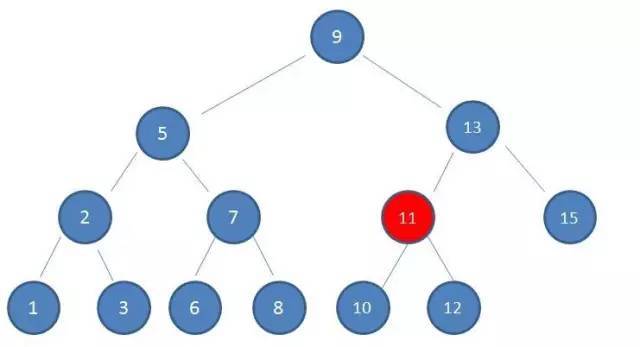
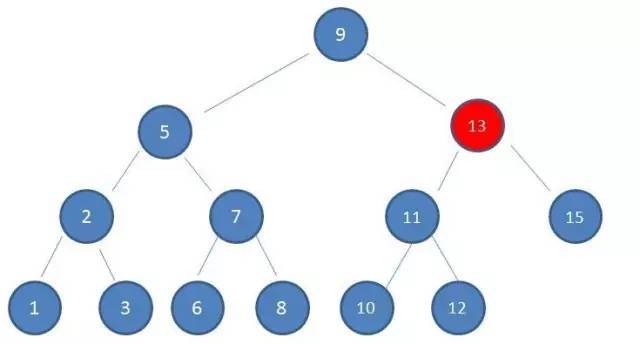
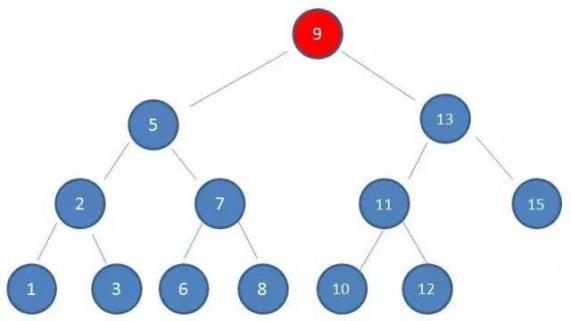
（2）若右子树不空，则右子树上所有节点的值均大于它的根节点的值；

（3）左、右子树也分别为二叉排序树；

（4）没有键值相等的节点。



查看根节点9，由于10 > 9，因此查看右孩子13 ，由于10 < 13，因此查看左孩子11，由于10 < 11，因此查看左孩子10，发现10正是要查找的节点

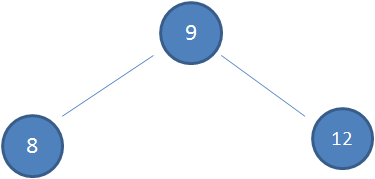




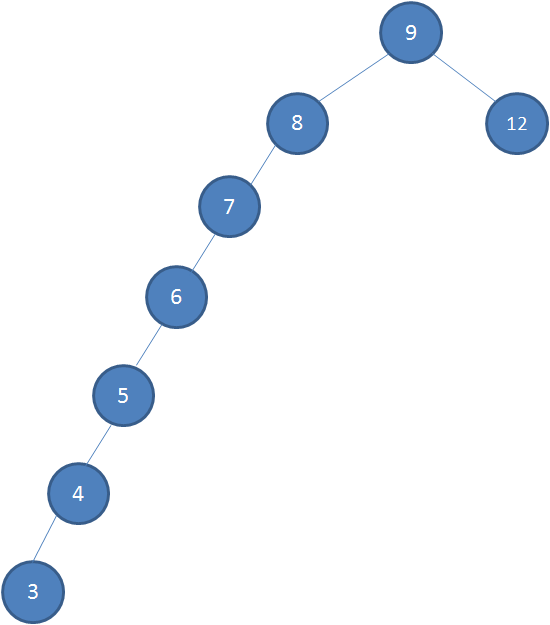




假设初始的二叉查找树只有三个节点，根节点值为9，左孩子值为8，右孩子值为12：



接下来我们依次插入如下五个节点：7,6,5,4,3。依照二叉查找树的特性，结果会变成这样





### 红黑树

红黑树（Red Black Tree） 是一种自平衡二叉查找树，是在[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA" \t "https://baike.baidu.com/item/%E7%BA%A2%E9%BB%91%E6%A0%91/_blank)科学中用到的一种[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/1450" \t "https://baike.baidu.com/item/%E7%BA%A2%E9%BB%91%E6%A0%91/_blank)，典型的用途是实现[关联数组](https://baike.baidu.com/item/%E5%85%B3%E8%81%94%E6%95%B0%E7%BB%84/3317025" \t "https://baike.baidu.com/item/%E7%BA%A2%E9%BB%91%E6%A0%91/_blank)。



1.节点是红色或黑色。

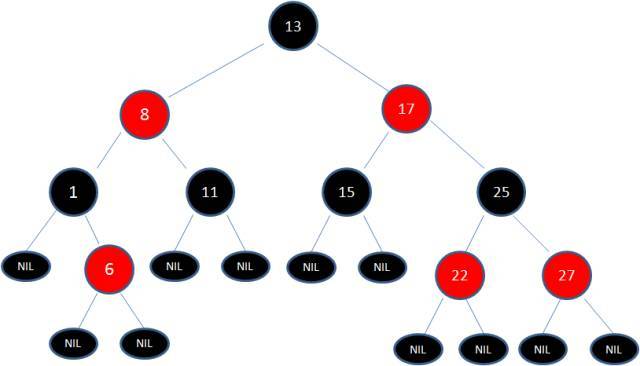
2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

1. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

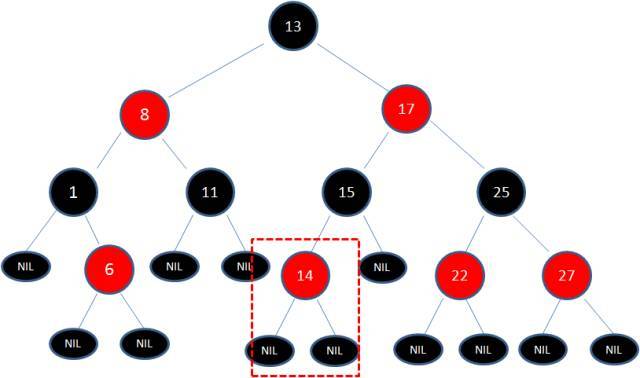
图中这棵树，就是一颗典型的红黑树：



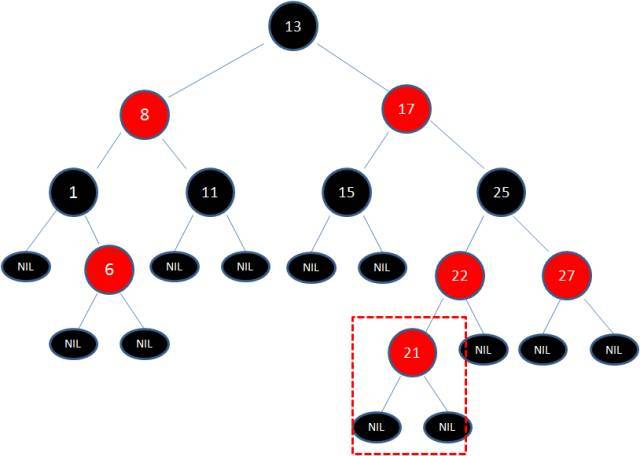


什么情况下会破坏红黑树的规则，什么情况下不会破坏规则呢？我们举两个简单的栗子：

1.向原红黑树插入值为14的新节点：



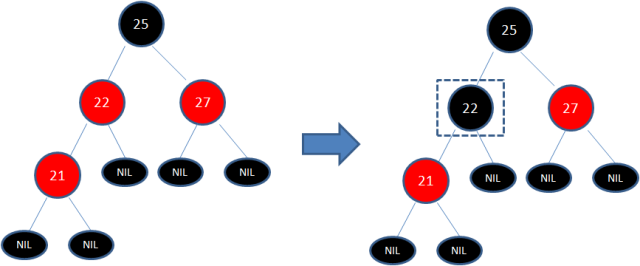
2.向原红黑树插入值为21的新节点：



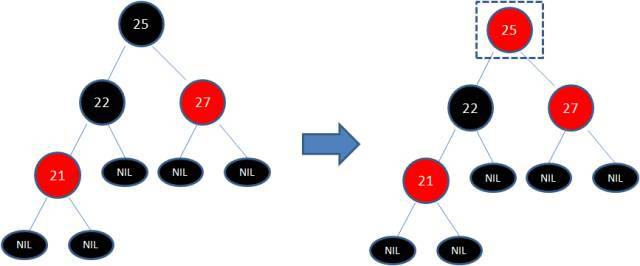
由于父节点22是红色节点，因此这种情况打破了红黑树的规则4（每个红色节点的两个子节点都是黑色），必须进行调整，使之重新符合红黑树的规则。



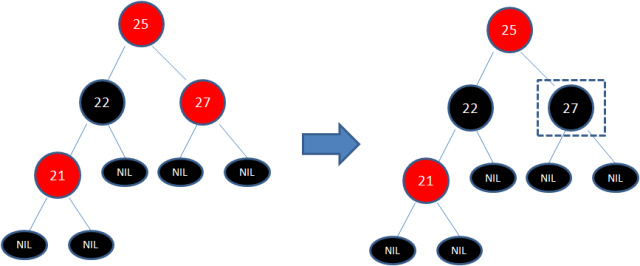
变色：为了重新符合红黑树的规则，尝试把红色节点变为黑色，或者把黑色节点变为红色。下图所表示的是红黑树的一部分，需要注意节点25并非根节点。因为节点21和节点22连续出现了红色，不符合规则4，所以把节点22从红色变成黑色：



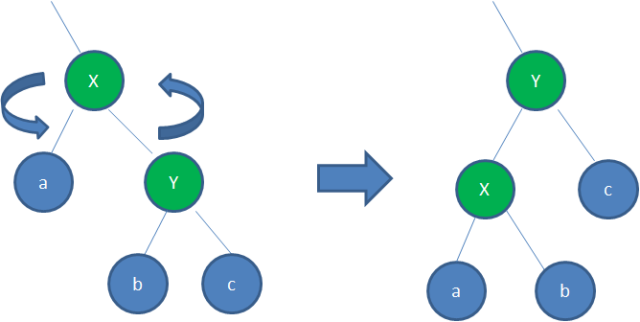
但这样并不算完，因为凭空多出的黑色节点打破了规则5，所以发生连锁反应，需要继续把节点25从黑色变成红色：



此时仍然没有结束，因为节点25和节点27又形成了两个连续的红色节点，需要继续把节点27从红色变成黑色：

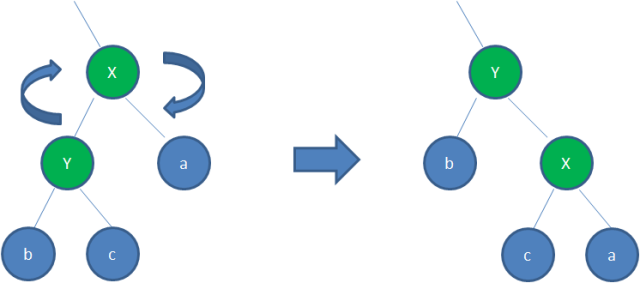


左旋转:逆时针旋转红黑树的两个节点，使得父节点被自己的右孩子取代，而自己成为自己的左孩子。说起来很怪异，大家看下图：



图中，身为右孩子的Y取代了X的位置，而X变成了自己的左孩子。此为左旋转。

右旋转：顺时针旋转红黑树的两个节点，使得父节点被自己的左孩子取代，而自己成为自己的右孩子。大家看下图：





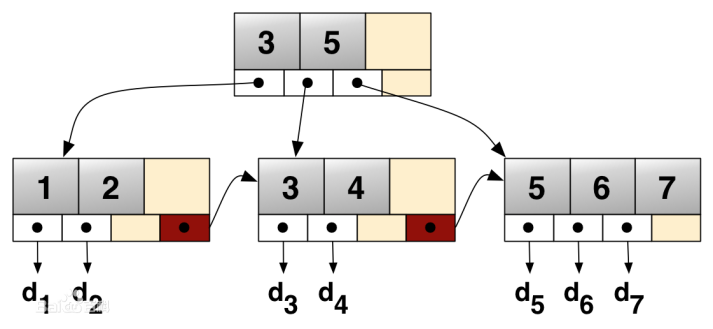
### B+树

B+ 树是一种树数据结构，通常用于[数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93" \t "https://baike.baidu.com/item/B+%E6%A0%91/_blank)和[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "https://baike.baidu.com/item/B+%E6%A0%91/_blank)的[文件系统](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F" \t "https://baike.baidu.com/item/B+%E6%A0%91/_blank)中。B+ 树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+ 树元素自底向上插入，这与[二叉树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91" \t "https://baike.baidu.com/item/B+%E6%A0%91/_blank)恰好相反。B+树是B树的一种变形形式，B+树上的叶子结点存储关键字以及相应记录的地址，叶子结点以上各层作为索引使用。一棵m阶的B+树定义如下

(1)每个结点至多有m个子女;

(2)除根结点外,每个结点至少有[m/2]个子女，根结点至少有两个子女;

(3)有k个子女的结点必有k个关键字。



mysql索引用的是B+树的原理

理解

我们在MySQL中的数据一般是放在磁盘中的，读取数据的时候肯定会有访问磁盘的操作，磁盘中有两个机械运动的部分，分别是盘片旋转和磁臂移动。盘片旋转就是我们市面上所提到的多少转每分钟，而磁盘移动则是在盘片旋转到指定位置以后，移动磁臂后开始进行数据的读写。那么这就存在一个定位到磁盘中的块的过程，而定位是磁盘的存取中花费时间比较大的一块，毕竟机械运动花费的时候要远远大于电子运动的时间。当大规模数据存储到磁盘中的时候，显然定位是一个非常花费时间的过程，但是我们可以通过B树进行优化，提高磁盘读取时定位的效率。

为什么B类树可以进行优化呢？我们可以根据B类树的特点，构造一个多阶的B类树，然后在尽量多的在结点上存储相关的信息，保证层数尽量的少，以便后面我们可以更快的找到信息，磁盘的I/O操作也少一些，而且B类树是平衡树，每个结点到叶子结点的高度都是相同，这也保证了每个查询是稳定的。

总的来说，B/B+树是为了磁盘或其它存储设备而设计的一种平衡多路查找树(相对于二叉，B树每个内节点有多个分支)，与红黑树相比，在相同的的节点的情况下，一颗B/B+树的高度远远小于红黑树的高度(在下面B/B+树的性能分析中会提到)。B/B+树上操作的时间通常由存取磁盘的时间和CPU计算时间这两部分构成，而CPU的速度非常快，所以B树的操作效率取决于访问磁盘的次数，关键字总数相同的情况下B树的高度越小，磁盘I/O所花的时间越少。

具体用B+树的原因

1、 B+树的磁盘读写代价更低：B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。

2、B+树的查询效率更加稳定：由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

3、由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

PS：有人认为数据库索引采用B+树的主要原因是：B树在提高了IO性能的同时并没有解决元素遍历的我效率低下的问题，正是为了解决这个问题，B+树应用而生。B+树只需要去遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作或者说效率太低。

## 堆(Heap)

### 概念

堆(Heap)是计算机科学中一类特殊的数据结构的统称。堆通常是一个可以被看做一棵[完全二叉树](https://baike.baidu.com/item/%E5%AE%8C%E5%85%A8%E4%BA%8C%E5%8F%89%E6%A0%91/7773232" \t "https://baike.baidu.com/item/%E5%A0%86/_blank)的数组对象。

* 堆中某个节点的值总是不大于或不小于其父节点的值
* 堆总是一棵完全二叉树

a.最大堆：

           ①根的值大于左右子树的值   ②子树也是最大堆

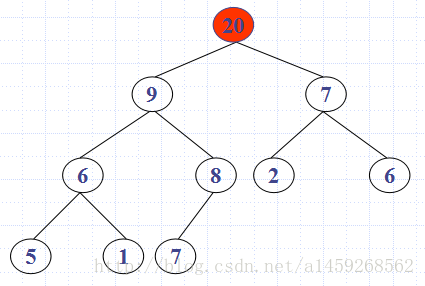
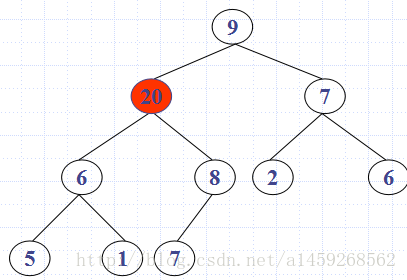
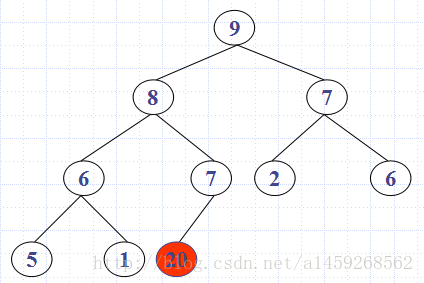
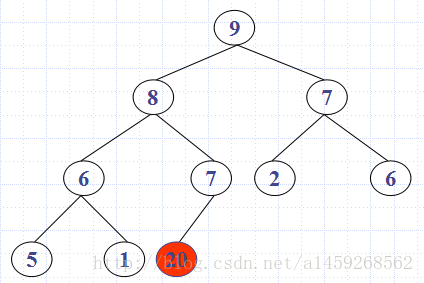
b.最小堆：

           ①根的值小于左右子树的值   ②子树也是最小堆

### 堆的实现

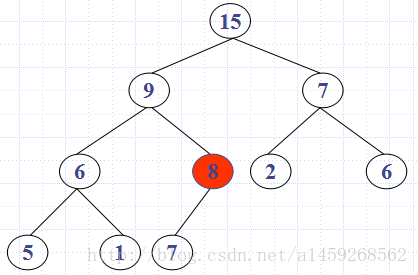
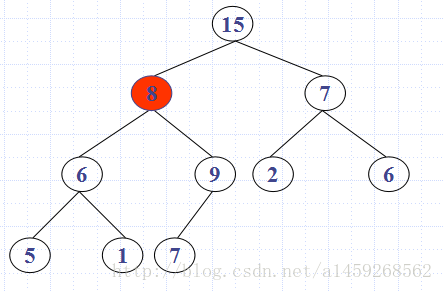
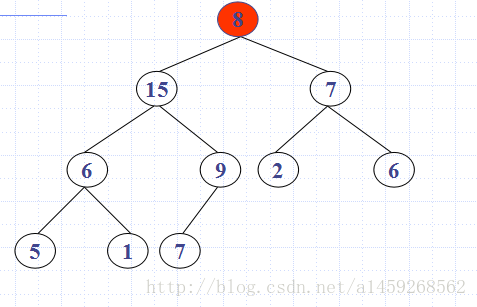
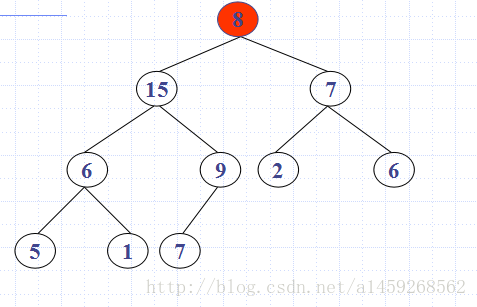
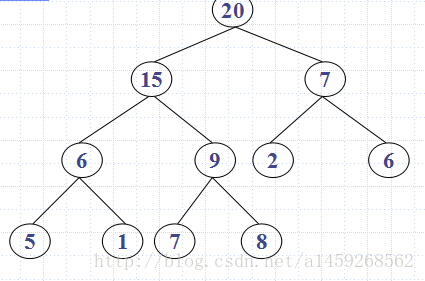
1.插入（Insert）

以最大堆为栗子，插入其实就是把插入结点放在堆最后面，然后与父亲比较，如果父亲值小于它，那么它就和父亲结点交换位置，重复该过程，直到插入节点遇到一个值比它大的父亲或者它成为树根结点，以最大堆为栗子，在堆中插入值为20的结点（红色结点代表新进入）



2.删除（Pop）

删除就是删除最大堆中的最大值或者最小堆中的最小值，也就是树根以删除最大堆树根为例子，删除其实就是整个堆中少了一个结点，不妨把位于最后面的一个结点当成新的树根，然后与它左右孩子比较，与值最大并且值大于它的孩子进行交换（好好读这句话），直到它的孩子都是小于它的，或者变成树叶，最大堆为例，删除树根20



### 应用场景

高效排序：堆排序

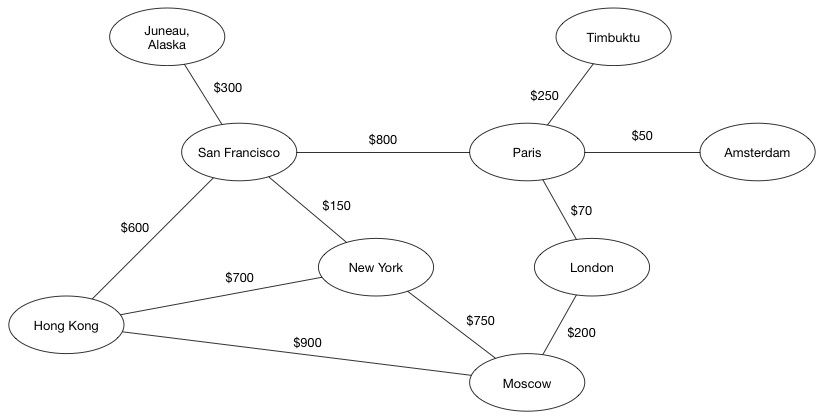
## 图(Graph)

### 概念

线性表和树两类数据结构，线性表中的元素是“一对一”的关系，树中的元素是“一对多”的关系，本章所述的图结构中的元素则是“多对多”的关系。图（Graph）是一种复杂的非线性结构，在图结构中，每个元素都可以有零个或多个前驱，也可以有零个或多个后继，也就是说，元素之间的关系是任意的。

示例

一个代表航线的图，各个城市就是顶点，航线就是边。那么边的权重可以是飞行时间，或者机票价格，通过一张图更容易建模分析理解，这就是用图的好处。



### 顶点和边

理论上，图就是一堆顶点和边对象而已，但是怎么在代码中来描述呢？有两种主要的方法：邻接列表和邻接矩阵。

邻接列表：在邻接列表实现中，每一个顶点会存储一个从它这里开始的边的列表。比如，如果顶点A 有一条边到B、C和D，那么A的列表中会有3条边



邻接列表只描述了指向外部的边。A 有一条边到B，但是B没有边到A，所以 A没有出现在B的邻接列表中。查找两个顶点之间的边或者权重会比较费时，因为遍历邻接列表直到找到为止。

邻接矩阵：在邻接矩阵实现中，由行和列都表示顶点，由两个顶点所决定的矩阵对应元素表示这里两个顶点是否相连、如果相连这个值表示的是相连边的权重。例如，如果从顶点A到顶点B有一条权重为 5.6 的边，那么矩阵中第A行第B列的位置的元素值应该是5.6：



往这个图中添加顶点的成本非常昂贵，因为新的矩阵结果必须重新按照新的行/列创建，然后将已有的数据复制到新的矩阵中。所以使用哪一个呢？大多数时候，选择邻接列表是正确的。

图的节点代码表示，无向简单图的节点

public class GraphNode<T> {

T data;

List<GraphNode<T>> neighborList;

boolean visited;

public GraphNode(T data){

this.data = data;

neighborList = new ArrayList<GraphNode<T>>();

visited = false;

}

}

jdk并没有对Graph相关的算法，要根据具体业务实现，所以没有同意模板算法。

### 应用场景

具体业务分析建模，如分析多线程中线程状体机制图。