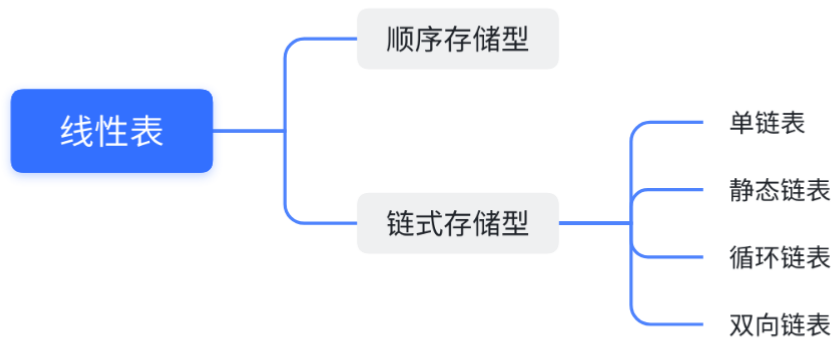
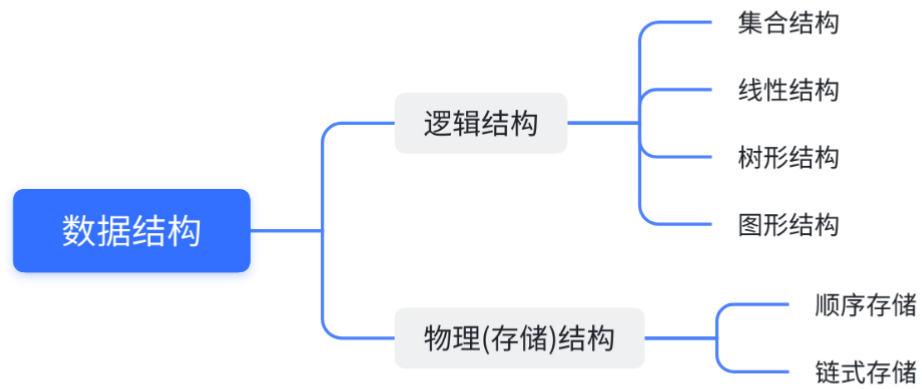


# 数据结构



# 栈和队列

## 中缀转后缀

如果是数字直接输出，是运算符进栈，每当将要进栈的运算符的优先级不大于栈顶的运算符时，栈顶的运算符出栈，当前运算符进栈，如果有括号存在的情况下在右括号进栈时弹出栈顶符号

$9 + (3 - 1) \times 3 + 10 / 2 \rightarrow 9 \ 3 \ 1 \ - \ 3 \ \times \ + \ 10 \ 2 \ / \ +$

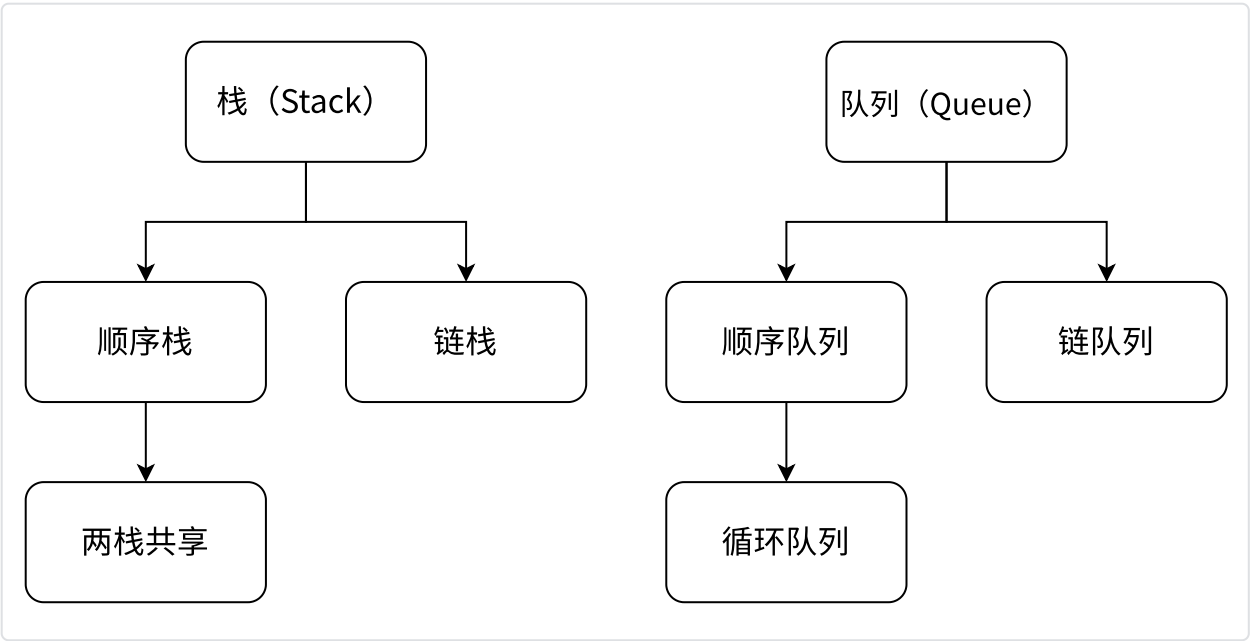
## 循环队列

判断循环队列队满	$(rear + 1) \% QueueSize = front$
计算队列长度	$(rear - front + QueueSize) \% QueueSize$

## 弊端解决

栈和队列都可以使用线性表的顺序存储结构进行实现，但都存在一定的弊端

- 数组形式的栈，大小是固定的，初始化一个较大的数组又会造成空间的浪费，可以使用一个数组的两端作为栈底存储相同数据类型的数据（数据总量必须是固定的，且对应关系多为此消彼长）
- 队列的出队后需要移动数据（不移动数据采用队首后移的方式会很快使用完空间），因此可以采用循环队列



# 串

# KMP匹配

区别于传统模式匹配，传统的模式匹配中查找 `str` 是否包含子串 `subStr` 时，要遍历 `str` 的每一个字符作为 `subStr` 的首字符，如果相等继续比较 `subStr.length()` 长度的字符，遇到不相等，判断下一个 `str` 的字符，以此循环。

而KMP主要就是省去了二次比较的过程，如"asdfgh"和"asdv"进行比较，当index=3时，'f'≠'v'，同时"asdv"中的'a'不等于后边的任何一个字符，所以之前判断相等的"sd"就无需进行比较，直接将"asdv"的首字符与index=3的字符进行比较，省去中间的判断以减少复杂度

所以KMP算法的核心就是 `sunStr` 中字符的重复度，这里用 `next[]` 数组进行表示，通过计算初始化后将 `next[]` 添加到传统的匹配算法当中，主要改动就是在遇到字符不匹配的情况时，让下一轮的比较在合适的地方开始

Index	0	1	2	3	4	5
String	a	b	c	a	b	c
next[]	-1	0	0	0	1	2

## Java

```
1  /**
2   * 计算next[]数组
3   *
4   * @param p 匹配串
5   * @return
6   */
7  private static int[] getNext(String p) {
8      int len = p.length();
9      int[] next = new int[len];
10     next[0] = -1;
11     int i = 0, k = -1;
12     while (i < len - 1) {
13         // p[k]表示前缀, p[i]表示后缀
14         if (k == -1 || p.charAt(k) == p.charAt(i)) {
15             ++k;
16             ++i;
17             next[i] = k;
18         } else {
19             k = next[k];
20         }
21     }
22     return next;
23 }
```

## Java

```
1  /**
2   * KMP实现代码
3   *  $O(m+n)$ 
4   * @param s 目标串
5   * @param p 模式串
6   * @return 如果匹配成功, 返回下标, 否则返回-1
7   */
8  private static int kmpSearch(String s, String p) {
9      int sLen = s.length();
10     int pLen = p.length();
11     if (sLen < pLen) {
12         return -1;
13     }
14
15     int[] next = getNext(p);
16     // matching:  $O(n)$ 
17     int i = 0, j = 0;
18     while (i < sLen && j < pLen) {
19         //①如果  $j = -1$ , 或者当前字符匹配成功 (即  $S[i] == P[j]$ ), 都令  $i++$ ,  $j++$ 
20         if (j == -1 || s.charAt(i) == p.charAt(j)) {
21             i++;
22             j++;
23         } else {
24             //②如果  $j != -1$ , 且当前字符匹配失败 (即  $S[i] != P[j]$ ), 则令  $i$  不变,  $j = ne$ 
25             //xt[j]
26             //next[j]即为j所对应的next值
27             j = next[j];
28         }
29     }
30     if (j == pLen) {
31         return i - j;
32     } else {
33         return -1;
34     }
```

在某些特定的情况下 `next[]` 的数组效率就没那么高了, 如 `str="aaaasdf"` 和 `subStr="aaaaaz"` 的情况。因为 `subStr` 的 `[1,2,3,4]` 都与首位相等, 这时候可以直接将 `next[{1,2,3,4}]` 的值用 `next[first]` 进行取代, 由此引出 `nextval[]` 数组, 该数组的思

想就是若某一位(n)字符的 `next[n]` 的值与指向的m位字符(m=`next[n]`)相等，则 `nextval[n]` 就继承 `nextval[m]`，否则继承自身的 `next[]` 值

Index	0	1	2	3	4	5
String	a	b	c	a	b	c
next[]	-1	0	0	0	1	2
nextval[]	-1	0	0	-1	0	0

Java

```
1  /**
2   * Table building: O(m)
3   * 优化的next数组
4   * @param p 匹配串
5   * @return
6   */
7  private static int[] getNext2(String p) {
8      int len = p.length();
9      int[] next = new int[len];
10     next[0] = -1;
11     int i = 0, k = -1;
12     while (i < len - 1) {
13         // p[k]表示前缀，p[i]表示后缀
14         if (k == -1 || p.charAt(i) == p.charAt(k)) {
15             ++k;
16             ++i;
17             if (p.charAt(i) != p.charAt(k)) {
18                 next[i] = k;
19             } else {
20                 // 因为不能出现p[i] = p[next[i]]，所以当出现时需要继续递归，k = next
21                 // [k] = next[next[k]]
22                 next[i] = next[k];
23             }
24         } else {
25             k = next[k];
26         }
27     }
28     return next;
29 }
```

# 树

树分为分支节点（包括根节点）与叶子节点，其中分支节点（除根节点外）又叫做内部节点，节点的度就是其所含子树的个数，而树的度就是所有节点中度的最大值

从根节点开始，每延伸一层子树就代表一层，最大的层数就是树的深度

线性结构与树结构的最大区别就是中间节点（中间元素），线性表只有一个前驱一个后继，而树中有一个双亲多个孩子

## 树的存储

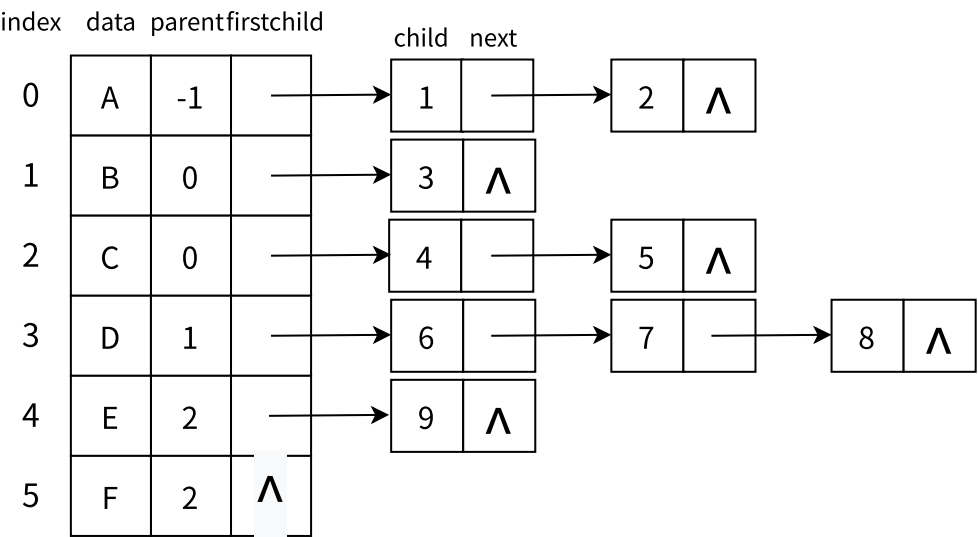
### 1. 双亲表示法

双亲表示法的核心为除根节点外任何节点都有自己的双亲，可以定义节点结构：包含数据和双亲在链表中的地址，根据操作重点的不同，还可以在节点结构中扩展孩子属性、兄弟属性等

### 2. 孩子表示法

将每个孩子节点用链表的形式进行存储，附着在其双亲节点的后边

双亲表示法方便查找双亲，但找孩子需要遍历，孩子表示法反之，于是产生了双亲孩子表示法



孩子兄弟法的结构使用 `[data|firstchild|rightsib]` 结构进行存储，具有方便查找孩子的优点，同时将不管多复杂的树都转换为一颗二叉树

## 二叉树

完全二叉树是指任一编号和其在满二叉树上的位置在同一层，符合以下特点：

- 叶子节点只在最后两层

- 最底层的叶子节点一定在左侧连续位置
- 倒数第二层叶子节点一定在右侧连续位置
- 度为1的节点只能是只有左孩子，右孩子不能单独出现
- 同样节点数量的二叉树，完全二叉树的深度最小

## 二叉树性质

1. 二叉树的第*i*层最多有  $2^{i-1}$  个节点
2. 深度为*k*的二叉树最多有  $2^k - 1$  个节点
3. 任意二叉树，叶子节点有  $n_0$  个，度为2的节点有  $n_2$  个，则  $n_0 = n_2 + 1$
4. 具有*n*个节点的二叉树具有  $\lfloor \log_2 n \rfloor + 1$  ， 其实是根据性质2反推得来
5. 对于一颗有*n*个节点的完全二叉树（其深度为  $\lfloor \log_2 n \rfloor + 1$  ）的节点按层序编号，对任一节点*i*（ $1 \leq i \leq n$ ）有：
  - a. 如果*i*=1,则节点*i*是二叉树的根；*i*>1，其双亲为  $\lfloor i/2 \rfloor$
  - b. 如果  $2 \times i > n$  则节点*i*一定是叶子节点，否则其左孩子为  $2 \times i$
  - c. 如果  $2 \times i + 1 > n$  ，则节点*i*没有有孩子，否则右节点为  $2 \times i + 1$

## 二叉树存储

二叉树存储完全可以用数组实现，只要下表具有对应关系即可（不存在的节点，置空跳过即可），但如果是一棵右斜树，那样会极大的浪费空间，所以数组的形式用来存储完全二叉树最合适

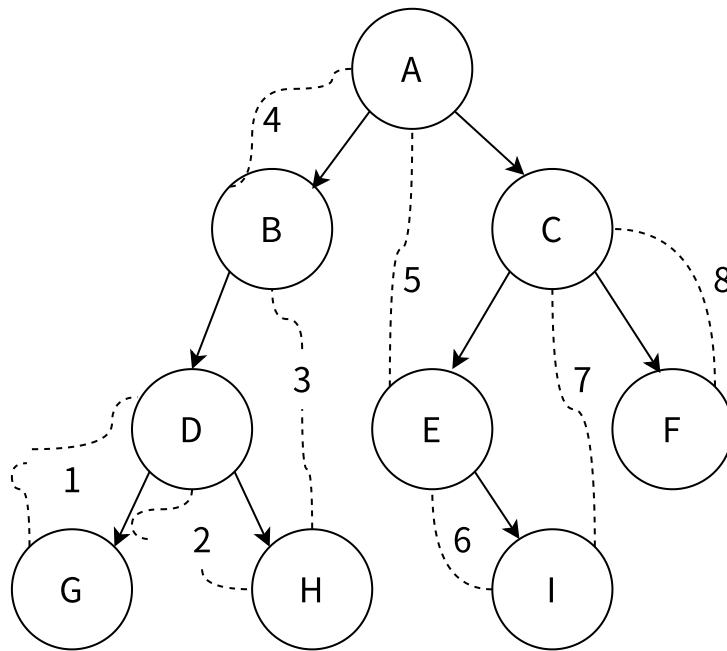
树的存储使用最多的还是链式存储结构，抽象为：  $[lchild|data|rchild]$

## 二叉树的遍历

方法	实现
前序	根左右
中序	左根右
后序	左右根



## 中序遍历



## 线索二叉树

线索二叉树就是节约每个节点的空指针域，使节点结构  $[lchild|data|rchild]$  中空闲的  $lchild$  指向前驱，空闲的  $rchild$  指向后继，这样就节约了指针域为空时的浪费，但这样就无法判断其到底是指向孩子还是前驱或后继，所以将节点机构改为  $[ltag|lchild|data|rchild|rtag]$ ，根据  $tag$  的值为 0 或 1 判断其指向孩子还是逻辑节点

## 哈夫曼树

思想就是权值最高的节点找到它的路径长度就越短，即频率越高越好找

带权路径长度就是权值与其路径长度的积之和

构造哈夫曼树：将权值递增排序后，每次取出最小的两个来构造二叉树，将它俩的和作为双亲的权值放入排序中，重复上一步

## 图

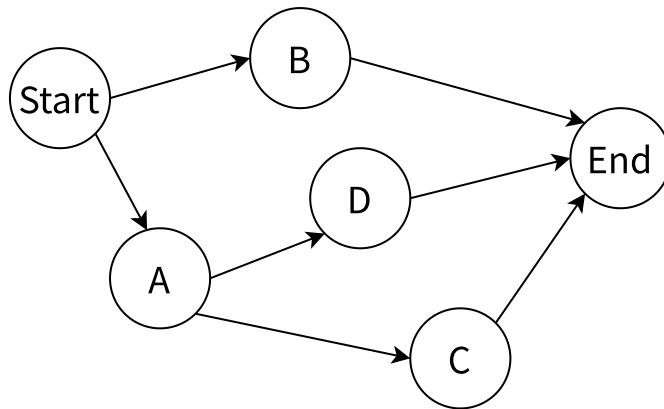
### 图的存储

1. 邻接矩阵：使用一维数组存储顶点，二维数组存储边
2. 邻接表：主要解决邻接矩阵中二维数组存储的边较少时对空间的浪费，将点之间的关系用链表进行存储

逆邻接表：方便表示当前顶点是哪些结点的弧头

3. 十字链表：将邻接表和逆邻接表结合使用，主要应用于有向图
4. 邻接多重表：与邻接表的区别就是将每条边存储的两端顶点用一个结点进行表示，方便对无向图的边操作（添加、删除）
5. 边集数组：使用两个一维数组分别存储顶点信息和边关系

## BFS遍历



Java

```
1 import java.util.HashMap;
2 import java.util.LinkedList;
3 import java.util.Queue;
4
5 public class Test {
6
7     static HashMap<Character, LinkedList<Character>> graph;
8     static HashMap<Character, Integer> distance;
9
10    //bfs
11    private static void bfs(HashMap<Character, LinkedList<Character>> graph, HashMap<Character, Integer> distance, char start) {
12        Queue<Character> queue = new LinkedList<Character>();
13        queue.add(start); //将s作为起始顶点加入队列
14        distance.put(start, 0); //distance表是用来记录每一个图节点到顶点的距离
15        int i = 0;
16        while (!queue.isEmpty()) //队列不为空就一直遍历
17        {
18            char top = queue.poll(); //取出队首元素
19            i++;
20            System.out.println("The " + i + "th element:" + top + " Distance from S is:" + distance.get(top));
```

```

21         int d = distance.get(top) + 1; //得出其周边还未被访问的节点的距离，例如：d
    (S->A)就是0+1=1
22         for (Character c : graph.get(top)) {
23             if (!distance.containsKey(c)) //如果distance中还没有该元素说明还没有
    被访问
24             {
25                 distance.put(c, d); //将新访问到的元素与其和顶点相距的距离信息
    存入 distance
26                 queue.offer(c); //将新访问到的元素入队
27             }
28         }
29     }
30 }
31
32 public static void main(String[] args) {
33     // s顶点的邻接表
34     LinkedList<Character> list_s = new LinkedList<Character>();
35     list_s.add('A');
36     list_s.add('B');
37     LinkedList<Character> list_a = new LinkedList<Character>();
38     list_a.add('C');
39     list_a.add('D');
40     LinkedList<Character> list_b = new LinkedList<Character>();
41     list_b.add('D');
42     list_b.add('E');
43     LinkedList<Character> list_c = new LinkedList<Character>();
44     list_c.add('E');
45     LinkedList<Character> list_d = new LinkedList<Character>();
46     list_c.add('E');
47
48     //构造图
49     graph = new HashMap<Character, LinkedList<Character>>();
50     graph.put('S', list_s);
51     graph.put('A', list_a);
52     graph.put('B', list_b);
53     graph.put('C', list_c);
54     graph.put('D', list_d);
55     graph.put('E', new LinkedList<Character>());
56
57     //调用
58     distance = new HashMap<Character, Integer>();
59     bfs(graph, distance, 'S');
60 }
61 }

```

dfs和bfs的时间复杂度是相同的，具体使用根据情况的不同，单纯找目标用dfs，求最优解用bfs

## 最小生成树

### Prim算法

是一种增量算法，每次选择已连接结点的最小边，每一步都是在前面的基础上进行的

### Kruskal算法

将边进行递增排序，按顺序对顶点进行连接，若加入某条边时会形成回路就将其舍弃

## 最短路径

### Dijkstra算法

从一个顶点到其余各顶点的最短路径，具体实现是将顶点分为两组，一组是最短路径结点（最开始是空的），每判断下一个邻接的最短结点就放入

### Floyd算法

对任意顶点  $(i, j)$ ，如果  $A[i][j] > A[i][v] + A[v][j]$ ，则将  $A[i][j]$  更新为  $A[i][v] + A[v][j]$  的值，同时其路径  $Path[i][j]$  改为  $V$

其构造使用三层循环嵌套：中转顶点包含首尾顶点

## 拓扑排序

区别于最短路径，针对没有回路（环）的图进行的最短求解，如AOV图（Activity On Vertex Network），即图中某个顶点的进行必须取决于前一顶点的完成，而这里的最短路径其实就是对无环有向图求一个拓扑序列

基本思想：找到入度为0的顶点输出，同时删除以此顶点为尾的弧，重复此步骤

## 关键路径

在AOV图的基础上对每条弧增加权值表示活动持续的时间，称为AOE图（Activity On Edge Network）  
判断关键路径就是判断当前活动的持续时间是否等于给定时间（没有空闲）

## 查找

查找分为静态和动态，所谓动态就是在查找时可能会进行插入未存在数据、删除数据等改变查找表的操作

## 有序情况

### 插值查找

基于二分查找，当关键字分布比较均匀时，根据其值在数组中的分布位置，将折半范围改为自适应的  $mid$ ，但在关键字分布不均匀的情况下效率未必好于折半查找，分割点确定代码为：

$$mid = low + (high - low) * (findVal - arr[low]) / (arr[high] - arr[low]);$$

### 斐波那契法

斐波那契数列中越往后的相邻两数其比无限接近于黄金分割率，而这种查找也就是根据黄金分割率来进行分割的，分割点确定代码为： $mid = low + F[k - 1] - 1$

虽然斐波那契数列的确定代码只有简单的加减运算，但不代表一定快于另两种查找，因为折半的算法完全可以使用位运算替代除法

## 索引查找

现实中的数据多是无序的，但可以建一个有序的索引来对数据进行查找，而索引也分为线性、树形和多级，而线性索引就是将索引项集合组织为线性结构

- 稠密索引：将数据中的每一条记录对应一个索引项，其缺点就是数据越多索引项越大
- 分块索引：将数据集进行分块，为每块建立索引，需要实现两个条件：块内无序、块间有序
- 倒排索引：根据不同数据项中含有的次关键字进行返回数据项（搜索引擎常用）

## 二叉排序树

将二叉排序树按中序遍历就是一个有序序列，在动态查找中，其有效提高了查找和插入删除关键字的速度

如果删除二叉排序树的一个同时具有左右子树的结点，需要找到其子树中大于其左结点且小于右结点的结点将其进行替换

### 平衡二叉树

在二叉排序树中的查找，其最坏情况也不过是树的深度，但如果一个树左右重量相差过大很难保证其复杂度接近于  $O(\log_n)$ ，所以平衡二叉树（AVL树）就是保证每一棵左右子树的深度相近（绝对值不超过1）

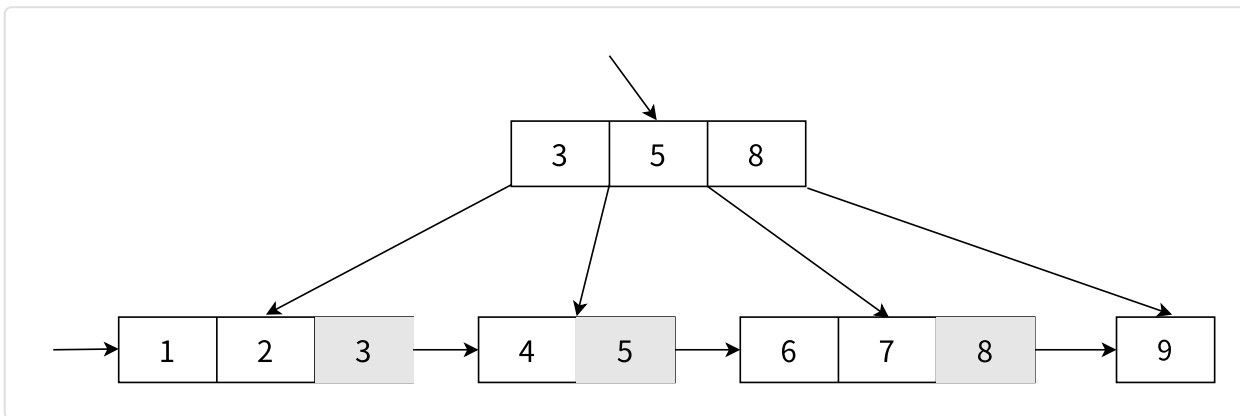
其构建过程需要根据BF值（BF = 左结点 - 右节点）进行旋转，大于1右旋，小于-1左旋，符号不相等时先旋转一次使符号相同后反向旋转

## 多路查找树

每个结点可以存储多个元素，可以有多个孩子，元素之间有某种特定的排序

其出现主要就是为了应对内存无法处理过多的数据，需要搭配硬盘使用，这时的时间复杂度计算就需要将对硬盘读取的时间和读取的次数考虑进来，而使用树结构进行存储就不得不对较大的数据量设计较多的度和高度，使得访问次数增多

- 2-3树和2-3-4树：要么同时有相应数量的孩子要么就没有孩子
- B树：平衡的多路查找树，也叫B-树、B\_树，其中结点最大的孩子数目称为树的阶（order），其减少了必须访问结点和数据块的数量以提高性能，这种数据结构就是为了内外存的数据交互准备的
- B+树：B树的缺陷就是中序遍历时可能会在不同的硬盘页面进行访问，造成一个页面被访问了多次，B树中的每个元素在树中只出现一次，可能在叶子结点也可能在分支结点，而在B+树上，出现在分支结点中的元素会被当做它们在该分支结点位置的中序后继者（叶子节点）中再次列出，每一个叶子结点都会保存一个指向后一叶子结点的指针



随机查找时，即便在分支结点找到了也只是索引而已，还是需要到存储关键字的叶子结点上；如果顺序查找则只需要从叶子结点出发不经过分支；特别适合带范围的查找：找到起始关键字后直接使用叶子结点进行查找到结束关键字。插入和删除都是在叶子结点上进行

## 散列查找

散列查找基于散列存储，根据查找的对应关系确定位置： $Station = f(key)$ ，当数据集中含有不唯一的關鍵字时无法使用（查找性别“男”）、范围查找无法使用

几种散列构造算法：

- 直接定址法：适用于数据规模小且关键字连续的情况下
- 数字分析法：抽取关键字的一部分进行运算操作（反转、叠加、环位移）  
适用于关键字位数较大或事先知道关键字分布且若干位分布均匀的情况
- 平方取中法：算出关键字的平方，选择特定的其中几位  
不清楚分布且位数不是很大的情况
- 折叠法：分割相加或其他操作  
不清楚分布且位数较大


- 除留余数法：选择合适的  $P$ ，进行取余  $f(key) = key \% P$  ( $P \leq length$ ) 其中  $P$  通常选择为小于等于表长的最大质数
- 随机数法：适合关键字长度不相等的情况

## 处理冲突

- 开放定址法：遇到冲突就存在后面的空位置，容易产生堆积（本来无需争夺地址的关键字的原本位置被抢占），使用随机种子产生的随机数计算合适的偏移量
- 再散列函数法：准备多个函数，等前位置被占用使用另一个函数
- 链地址法：在相同位置使用链进行存储
- 公共溢出区：将溢出关键字存放到另一个溢出表中
- 

### 十大经典排序算法(动图演示)\_HK\_John的博客-CSDN博客

十大经典排序算法(动图演示)0、算法概述0.1 算法分类十种常见排序算法可以分为两大类:非线性时间比较类排序:通过比较来决定元素间的相对次序,由于其时间复杂度不能突破 $O(n\log n)$ ,因此称为非线性时间比较类排...

 [https://blog.csdn.net/Hk\\_john/article/details/79888992?utm\\_medium=distribute.pc\\_relevant.none-task-blog-baid...](https://blog.csdn.net/Hk_john/article/details/79888992?utm_medium=distribute.pc_relevant.none-task-blog-baid...)