# 数据结构

数据结构包括**线性结构**和**非线性结构**；

## 线性结构

1. 线性结构是最常见的数据结构，特点是数据元素之间存在一对一的线性关系；
2. 线性结构有两种不同的存储结构，顺序存储结构和链式存储结构：

顺序存储的线性表成为顺序表，顺序表的存储元素是连续的，如数组：依次往下保存；

链式存储的线性表成为链表，链表中的存储元素不一定是连续的，元素节点中存放数据元素以及相邻元素的地址信息；

1. 线性结构常见的有数组、队列、链表和栈；

## 非线性结构

非线性结构包括：二维数组、多维数组、广义表、树结构、图结构

# 稀疏数组和队列

## 稀疏数组

当一个数组中大部分元素都为0，或者为同一值的数组时，可以使用稀疏数组来保存该数据；

1. 稀疏数组的处理方法：
2. 记录数组一共有几行几列，有多少个不同的值
3. 把具有不同值的元素的队列及值记录在一个规模数组中，从而缩小程序的规模；

第一列数据表示：6行7列8个值

PS：数组下标是从0开始的；

EG：需求保存五子棋中下棋的位置

1. 二维数组转为稀疏数组的思路：
2. 遍历原始的二维数组，得到有效数据的个数sum；
3. 根据sum创建稀疏数组sparseArray int[sum+1][3];
4. 将二维数组的有效数据存入稀疏数组；
5. 稀疏数组转为二维数组：
6. 先读取稀疏数组第一行，根据第一行创建原始的二位数组：

**int[][]** origin = **new int[**sparse**[**0**][**0**]][**sparse**[**0**][**1**]]**;

1. 读取稀疏数组最后几行的数据，并赋予原始的二位数组；

## 队列

1. 队列的介绍
2. 队列是一个有序列表，可以用数组或者链表来实现；
3. 遵循先入先出的原则；
4. 数组模拟队列：



maxSize为该队列的最大容量；front和rear作为两个指针，记录队列的前后端指标，初始位置是-1；

存入数据后，rear值为存放值的最大下标，front为已取出值下标+1；

存数据rear加，front不变；取数据rear不变，front加；

思路：

1. addQueue 添加队列
2. 当rear<maxSize-1时数据存入rear为下标的元素中，然后rear上移++；
3. 当rear=maxSize-1，数组存满；
4. fetchQueue 获取队列
5. 先将front指针上移+1，取出对应下标的值；
6. 如果front=rear，则队列取完；
7. 数组模拟环形队列

思路：

1. front：初始值0，指向队列的第一个元素的当前位置;
2. rear： 初始值0，指向队列后一个元素的后面一个位置，为了空出来一个空间做约定，发现要不没办法判断队列是否满，因为取模实现循环，判断只能使用==。所以导致实际队列存放的数据是数组maxSize-1;
3. 队列为空的条件：front==rear；
4. 队列为满的条件：(rear+1)%maxSize==front；
5. 队列中的有效数据的个数：(rear+ maxSize - front)%maxSize

# 链表(Linked List)

链表是有序的列表，内存中存储模型如下:

小结：

1. 链表是以节点存储的；
2. 每个节点包含data域和next域：指向下一个节点；
3. 链表的各个节点不是连续存储的，即next域记录内存地址不是连续的；
4. 链表分为带头节点的链表和无头节点的链表；
5. 头指针记录第一个数据的地址；
6. 第一个数据的地址在data域获取数据，在next域获取下一个数据的地址；依次往后，直到next域内容为null；

## 单链表

单链表：带有头结点，逻辑结构如下：



最后一个的next域为null，这只是逻辑结构，实际在内存中a1，a2...是不连续的；

1. 依次添加Node
2. 先创建一个head头节点，作用是表示单链表的头。初始情况下头节点的next域为null；
3. 后面每添加一个节点，就直接加入到链表的最后

此时需要遍历找到最后一个节点，头节点都是不能动的，需要引入一个临时节点tempNode来承接数据；

1. 然后tempNode一直赋予next，直到next == null;

CODE

节点的数据结构：

**class** HeroNode**{  
 public int no**;  
 **public** String **name**;  
 //自己当时理解的是存的地址的具体信息  
 //其实存的就是对象 他本身就是一个地址  
 **public** HeroNode **next**;  
 **public** HeroNode**(int** no, String name**) {  
 this**.**no** = no;  
 **this**.**name** = name;  
 **}  
}**

添加节点：

**private** HeroNode **headNode** = **new** HeroNode**(**0,**"")**; //头节点  
**public void** addNode**(**HeroNode newNode**){** HeroNode temp = **headNode**;  
 **while (true) {** //标志找到了当前队列的最后一个元素  
 **if (**temp.**next** == **null) {** temp.**next** = newNode;  
 **break**;  
 **}** //否则就一直往后走  
 //以为将当前元素的内存地址赋予了temp，那么更改temp就是更改对应的LinkedList的最后一个元素  
 temp = temp.**next**;   
 **}  
}**

1. 按顺序添加Node



1. 首先定位位置，仍然是需要借助临时节点：需要找到新节点的前一个节点，这样将新节点插入该节点后面即可；
2. 定位条件：
3. temp.next == null 即新节点是位于最后一个节点的位置；
4. temp.**next**.**no** > newNode.**no** temp刚好是新节点的前一个节点
5. newNode.next = temp.next;
6. temp.next = newNode;
7. 单链表面试题：
8. 求链表中有效节点的个数，参数为头节点

遍历即可，头节点要去掉

1. 查找倒数第K个节点

获取链表总长度length，然后遍历得到第length-k+1个节点；

1. 单链表翻转

思路：

1. 先定义一个headNode来承接翻转后获取的链表；
2. 从头到尾遍历原链表，每遍历一个，就放在新链表的headNode后面；

PS：此处需要单独保存原节点的next，如不然，因为需要将新链表headNode.next赋给该找到的节点的next域，这样这导致该节点.next.next的丢失

代码：

HeroNode current = headNode.**next**;  
HeroNode curNext;  
HeroNode newHeadNode = **new** HeroNode**(**0,**"")**;  
//自己之前没有保存current.next 导致current被重新赋值后其next的丢失  
//注意此处涉及到内存地址的相互指向  
//所以要注意赋值的形式  
**while (**current != **null){** curNext = current.**next**;  
 current.**next** = newHeadNode.**next**;  
 newHeadNode.**next** = current;  
 current = curNext;  
**}**headNode.**next** = newHeadNode.**next**;

1. 从尾到头打印链表

根据栈这个数据结构的特点：先进后出

遍历放入栈中，然后遍历栈：

Stack**<**HeroNode**>** stack = **new** Stack**<>()**;  
HeroNode temp = headNode.**next**;  
**while (**temp != **null){** stack.add**(**temp**)**;  
 temp = temp.**next**;  
**}  
while (**!stack.empty**()){** System.**out**.println**(**stack.pop**())**;  
**}**

1. 合并两个有序链表并整体保持有序

就是结合3的思路加上有序添加

## 双向链表

带有head域记录前面一个数据的位置

单向链表的缺点：

1. 单向链表的查找只能是一个方向，而双向链表可以向前、向后查找；
2. 单向链表不能自我删除，需要借助辅助节点；而双向链表可以自我删除；

双向链表的数据结构示意：



1. 双向链表的添加：
2. 通过辅助接点找到链表的最后一个节点；
3. temp.next = newNode
4. newNode.pre = temp;

**void** addNode**(**HeroNode newNode**){** HeroNode temp = **this**.**headNode**;  
 **while (**temp.**next** != **null) {** temp = temp.**next**;  
 **}** temp.**next** = newNode;  
 newNode.**pre** = temp;  
**}**

1. 双向链表的修改：

同单向链表

1. 双向链表的删除

双向链表可以自我删除

直接找到待删除节点temp：

temp.pre.next = temp.next;

temp.next.pre = temp.pre;

**void** deleteNode**(**HeroNode delNode**){** HeroNode temp = **this**.**headNode**.**next**;  
 **if (**temp == **null){** System.**out**.println**("该链表为空，无法删除")**;  
 **return**;  
 **}  
 while (true){** //相比单链的一个额外判断 要不就temp.next.pre = temp.pre报空指针  
 **if (**temp.**no** == delNode.**no** && temp.**next** == **null){** System.**out**.println**("待删除节点为最后一个节点")**;  
 temp.**pre**.**next** = temp.**next**;  
 **break**;  
 **}  
 if (**temp.**no** == delNode.**no){** temp.**pre**.**next** = temp.**next**;  
 temp.**next**.**pre** = temp.**pre**;  
 **break**;  
 **}  
 if (**temp.**next** == **null){** System.**out**.println**("未找到待删除的节点")**;  
 **break**;  
 **}** temp = temp.**next**;  
 **}  
}**

1. 双链表的按顺序添加节点：

思路同单节点，关键还在于节点的定位

**void** addNodeByOrder**(**HeroNode newNode**){** HeroNode current = **this**.**headNode**;  
 HeroNode currentNext;  
 HeroNode currentPre;  
 **while (true){  
 if (**current.**next** == **null){** System.**out**.println**("新插入节点为最后一位....")**;  
 newNode.**pre** = current;  
 current.**next** = newNode;  
 **break**;  
 **}  
 if (**current.**next**.**no** > newNode.**no){** currentNext = current.**next**;  
 currentPre = current.**pre**;  
   
 current.**next** = newNode;  
 newNode.**next** = currentNext;  
   
 newNode.**pre** = current;  
 currentPre.**pre** = newNode;  
 **break**;  
 **}** current = current.**next**;  
 **}  
}**

## 环形链表和约瑟夫问题

介绍：一个环形链表，有n个元素，约定编号为k(1<=k<=n)的人从1开始报数，数到m出列，他的下一位继续从1开始数，到m出列，依次直到所有人出列，由此产生一个出队编号的序列；

1. 构建单向环形链表的思路：
2. 先构件第一个节点，让first指向自己形成一个环形；
3. 后续每添加一个节点，就把该节点加入到已有的环形链表中；
4. 单向环形链表的遍历
5. 先让指针（辅助变量）currentNode指想first节点；
6. 然后一个个遍历直到currentNode.next == first;
7. 约瑟夫问题思路：
8. 需要创建一个辅助指针变量pre，事先指向环形量表的最后一个节点，记录要数的变量的前一个变量；

PS：报数前需要先移动k-1次至指定位置；

1. 报数时，first和pre同时迁移m-1次；
2. 此时first节点出圈：

**first** = **first**.**next**;  
helper.**next** = **first**;

原first就会没有引用而被删除

# 数据结构 --栈 stack

## 栈的介绍

1. 栈是一个先入后出(FILO -- First In Last Out)的有序列表；
2. 栈是限制限制线性表中的元素的插入和删除只能在线性表的同一端进行的特殊线性表。允许插入和删除的一段为变化的一端，成为栈顶(Top)；另一端为固定的一端，称为栈底(Bottom)；
3. 最先放入栈的元素放入栈底，最后放入栈的元素放入栈顶，而删除元素刚好相反，最后放入的先被删除，最先放入的后被删除；
4. 出栈成为pop，入栈为push；

## 栈的应用场景：

1. 子程序的调用：在调往子程序之前，会先将下个指令的地址存入栈堆中，直到子程序执行完后再将地址去除，以回到原来的程序中；
2. 处理递归调用：和子程序的调用类似，只是除了储存下一个指令的地址外，也将参数、区域变量扥数据存入栈堆中；
3. 表达式的转换[中缀转为后缀表达式]与求值；
4. 二叉树的遍历；
5. 图形的深度优先(depth-first)搜索法；

## 用数组实现栈的模拟

1. 定义一个top来表示栈顶，初始化为-1；
2. 入栈：有数据加入到栈，top++;stack[top]=data;
3. 出栈：value=stack[top];top--;

## 栈实现计算器：

1. 通过一个index来遍历需要计算的表达式；
2. 如果是一个数字则直接入数栈；
3. 如果是一个符号则分情况考虑：
4. 如果当前符号栈为空，则可以直接入栈；
5. 如果符号栈不为空，则进行比较：

如果当前的操作符优先级小于于或者等于栈中的操作符(\* /操作大于+-)，就需要去数栈中pop出两个数字；

如果当前的操作符优先级大于栈中的操作符，就直接放入符号栈；

1. 扫描件完毕，就顺序从数栈和符号栈pop相应数和操作符并运行；
2. 最后数栈中只有一个数字，就是计算结果；

# 前缀、中缀、后缀表达式（逆波兰拨打时）

## 前缀表达式(波兰表达式)

前缀表达式的运算符位于操作数之前；

EG：(3+4)\*5-6 对应的前缀表达式是 - \* + 3 4 5 6

从右至左扫描表达式，遇到数字压入栈堆，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算(栈顶元素和次顶的元素)，并将结果入栈：重复上述过程指导表达式的最左端，最后计算的值即为表达式的结果；

## 中缀表达式

中缀表达式就是常见的运算表达式，如(3+4)\*5-6

中缀表达式对计算机不好操作，一般转为后缀表达式进行计算；

## 后缀表达式(逆波兰表达式)

运算符位于操作数之后

EG：(3+4)\*5-6 对应的后缀表达式是 3 4 + 5 \* 6 -

从左至右扫描表达式，遇到数字时，将数字压入栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的运算（次顶元素和栈顶元素），并将结果入栈；重复上述操作直到最右端；

## 中缀表达式转换为后缀表达式

1. 初始化两个栈，运算符栈s1和存储中间计算结果的栈s2；
2. 从左至右扫描中缀表达式；
3. 遇到操作数时，将其压入s2；
4. 遇到运算符时，比较其与s1栈顶运算符的优先级(可以用一个enum来做个int的优先级映射)：
5. 若s1为空，或者栈顶运算符为左括号(,则直接将此运算符入栈；
6. 若优先级比栈顶运算符的高，也将运算符压入s1；(只有高才执行，同级不执行)
7. 否则将s1栈顶的运算符弹出并压入s2中，再次转到4-1步骤中与s1中新的栈顶运算符进行比较；
8. 遇到括号时：
9. 如果是左括号(,则直接压入s1；
10. 如果是右括号),则依次弹出s1栈顶的运算符，并压入s2直到遇到左括号为止，此时将这一对括号丢弃
11. 重复步骤2到5，直到表达式的最右端；
12. 将s1中的运算符依次弹出并压入s2；
13. 依次弹出s2中的元素并输出，结果的逆序即为中缀表达式对应的后缀表达式；

# 递归 recursion

## 递归调用规则