# 数据结构

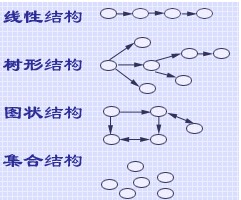
## 数据结构

编程的本质就是对数据（信息以数据的形式而存在）的处理，实际编程中不得不处理大量数据，因此实际动手编程之前必须先分析处理这些数据，处理数据之间存在的关系。

现实的数据元素之间有着纷繁复杂的逻辑关系，需要采用合适的物理结构来存储这些数据，并以此为基础对这些数据进行相应的操作。同时，还要分析这些数据结构在时间、空间上的开销的优劣。这种专门研究应用程序中数据之间逻辑关系、存储方式及其操作的学问就是数据结构。

## 逻辑结构

* **Data-Structure = (D,R)**
* **数据元素之间存在的关联关系被称为数据的逻辑结构。**归纳起来，应用程序中的数据大致有如下四种基本的逻辑结构：
* **集合：**数据元素之间只有“同属于一个集合”的关系
* **线性关系：**数据元素之间存在一个对一个的关系
* **树形结构：**数据元素之间存在一个对多个的关系
* **图状结构或网状结构：**数据元素之间存在多个对多个的关系



# 数组的常用算法

## 求数组元素的最大值、最小值、平均数、总和等

//数组最大值

**public** **int** getMax(**int**[] arr) {

**int** max = arr[0];

**for** (**int** i = 1; i < arr.length; i++) {

**if** (max < arr[i]) {

max = arr[i];

}

}

**return** max;

}

//数组的最小值

**public** **int** getMin(**int**[] arr) {

**int** min = arr[0];

**for** (**int** i = 1; i < arr.length; i++) {

**if** (min > arr[i]) {

min = arr[i];

}

}

**return** min;

}

// 遍历数组元素

**public** **void** printArray(**int**[] arr) {

System.*out*.print("[");

**for** (**int** i = 0; i < arr.length; i++) {

System.*out*.print(arr[i] + "\t");

}

System.*out*.println("]");

}

//数组平均数

**public** **int** avg(**int**[] arr) {

**int** sum = getSum(arr);

**return** sum / arr.length;

}

// 数组的总和

**public** **int** getSum(**int**[] arr) {

**int** sum = 0;

**for** (**int** i = 0; i < arr.length; i++) {

sum += arr[i];

}

**return** sum;

}

## 数组的复制、反转

// 数组的反转

**public** **int**[] reverse(**int**[] arr) {

**for** (**int** x = 0, y = arr.length - 1; x < y; x++, y--) {

**int** temp = arr[x];

arr[x] = arr[y];

arr[y] = temp;

}

**return** arr;

}

//数组的复制

**public** **int**[] copy(**int**[] arr) {

// int[] arr1 = arr;

// return arr1;

**int**[] arr1 = **new** **int**[arr.length];

**for** (**int** i = 0; i < arr.length; i++) {

arr1[i] = arr[i];

}

**return** arr1;

}

## 数组元素的排序

// 数组的排序

**public** **void** sort(**int**[] arr, String desc) {

**if** (desc == "asc") { //从小到大

**for** (**int** i = 0; i < arr.length - 1; i++) {

**for** (**int** j = 0; j < arr.length - 1 - i; j++) {

**if** (arr[j] > arr[j + 1]) {

swap(arr,j,j+1);

}

}

}

} **else** **if** (desc == "desc") { //从大到小

**for** (**int** i = 0; i < arr.length - 1; i++) {

**for** (**int** j = 0; j < arr.length - 1 - i; j++) {

**if** (arr[j] < arr[j + 1]) {

swap(arr,j,j+1);

}

}

}

} **else** {

System.*out*.println("你输入的排序方式有误");

}

}

**public** **void** swap(**int**[] arr,**int** i,**int** j){

**int** temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

# 排序与查找

## 直接排序

案例二：使用直接排序对数组进行排序：

/\*

选择排序。

以一个角标的元素和其他元素进行比较。

在内循环第一次结束，最值出现的头角标位置上。

\*/

**public** **static** **void** selectSort(**int**[] arr)

{

**for**(**int** x=0; x<arr.length-1; x++)

{

**for**(**int** y=x+1; y<arr.length; y++)//为什么y的初始化值是 x+1？ 因为每一次比较，

//都用x角标上的元素和下一个元素进 行比较。

{

**if**(arr[x]>arr[y])

{

**int** temp = arr[x];

arr[x] = arr[y];

arr[y] = temp;

}

}

}

}

## 冒泡排序

案例三：冒泡排序

/\*

冒泡排序。

比较方式：相邻两个元素进行比较。如果满足条件就进行位置置换。

原理：内循环结束一次，最值出现在尾角标位置。

\*/

**public** **static** **void** bubbleSort(**int**[] arr)

{

**for**(**int** x=0; x<arr.length-1; x++)

{

**for**(**int** y=0; y<arr.length-x-1; y++)//-x:让每次参与比较的元减。

//-1:避免角标越界。

{

**if**(arr[y]>arr[y+1])

{

**int** temp = arr[y];

arr[y] = arr[y+1];

arr[y+1] = temp;

}

}

}

}

## 折半查找(二分法)

案例四：

/\*

为了提高查找效率，可使用折半查找的方式，注意：这种查找只对有序的数组有效。

这种方式也成为二分查找法。

\*/

**public** **static** **int** halfSeach(**int**[] arr,**int** key)

{

**int** min,mid,max;

min = 0;

max = arr.length-1;

mid = (max+min)/2;

**while**(arr[mid]!=key)

{

**if**(key>arr[mid])

min = mid + 1;

**else** **if**(key<arr[mid])

max = mid - 1;

**if**(min>max)

**return** -1;

mid = (max+min)/2;

}

**return** mid;

}

案例五：数组翻转

/\*

反转其实就是头角标和尾角标的元素进行位置的置换，

然后在让头角标自增。尾角标自减。

当头角标<尾角标时，可以进行置换的动作。

\*/

**public** **static** **void** reverseArray(**int**[] arr)

{

**for**(**int** start=0,end=arr.length-1; start<end; start++,end--)

{

swap(arr,start,end);

}

}

//对数组的元素进行位置的置换。

**public** **static** **void** swap(**int**[] arr,**int** a,**int** b)

{

**int** temp = arr[a];

arr[a] = arr[b];

arr[b] = temp;

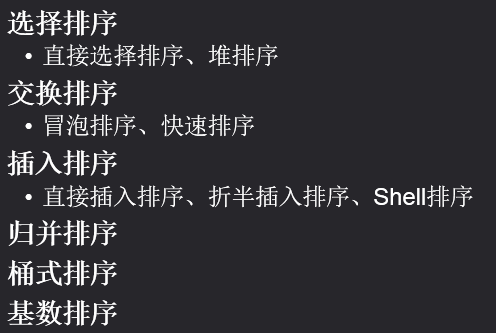
}

# 排序算法

## 排序算法优劣

* **排序：**假设含有n个记录的序列为{R1，R2，...,Rn},其相应的关键字序列为{K1，K2，...,Kn}。将这些记录重新排序为{Ri1,Ri2,...,Rin},使得相应的关键字值满足条Ki1<=Ki2<=...<=Kin,这样的一种操作称为排序。
* 通常来说，排序的目的是快速查找。
* **衡量排序算法的优劣：**
* 1.时间复杂度：分析关键字的比较次数和记录的移动次数
* 2.空间复杂度：分析排序算法中需要多少辅助内存
* 3.稳定性：若两个记录A和B的关键字值相等，但排序后A、B的先后次序保持不变，则称这种排序算法是稳定的。

## 常用排序



## 选择排序

在要排序的一组数中，选出最小的一个数与第一个位置的数交换；然后在剩下的数当中再找最小的与第二个位置的数交换，如此循环到倒数第二个数和最后一个数比较为止。

public static void selectSort(int[] arr) {

for(int i=0; i<arr.length; i++){

int k = i; // 待确定的位置

for(int j=arr.length-1; j>i; j--){

if(arr[j] < arr[k]){

k = j; // 记录第i个最小值的位置

}

}

// 将第i个位置的元素与此轮找到的最小的值交换位置

int temp = arr[i];

arr[i] = arr[k];

arr[k] = temp;

}

}

## 冒泡排序

比较相邻两个元素的大小，如果后一个元素比前面一个元素小，则交换两个元素的位置。一轮之后最大的元素会沉到底部。小的元素会逐渐往上浮，故称为冒泡排序。

private static void bubbleSort(int[] a) {

int temp = 0; // 起始比较位置

int size = a.length; // 数组的元素个数

// 外层for循环控制比较的轮次

for(int i=0; i<size-1; i++){

// 内存for循环比较相邻的两个元素

for(int j=0; j<size-1-i; j++){

// 如果前面一个元素比后面一个元素大，则交换两个元素的位置

if (a[j] > a[j+1]) {

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

}

## 插入排序

* 每步将一个待排序的记录，按其顺序码大小插入到前面已经排序的字序列的合适位置（从后向前找到合适位置后），直到全部插入排序完为止。

public static void insertSort(int[] arr) {

int size = arr.length;

int temp = 0 ;

int j = 0;

for(int i = 0; i < size; i++){

temp = arr[i];

// 假如temp比前面的值小，则将前面的值后移

for(j = i; j > 0 && temp < arr[j-1]; j --){

arr[j] = arr[j-1];

}

arr[j] = temp;

}

}