

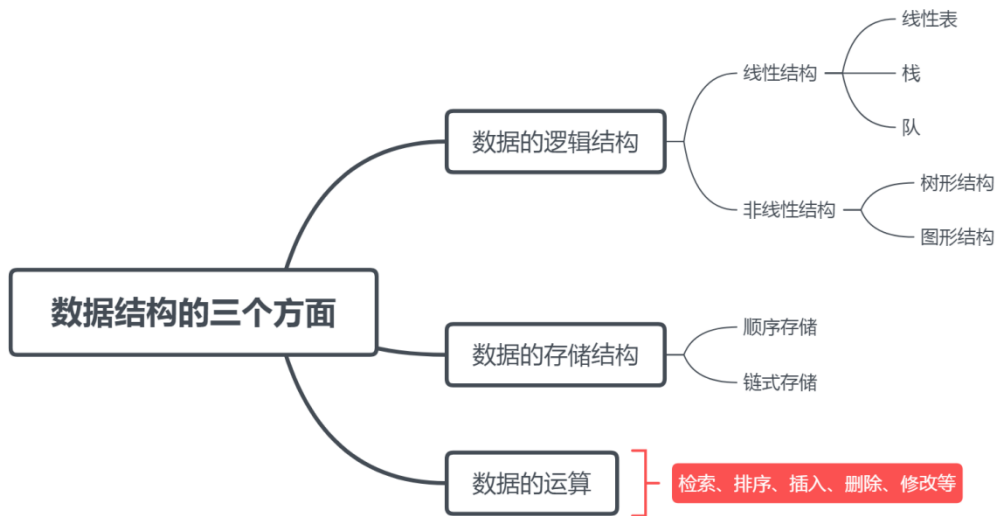
# 数据结构与算法

Author: 皓宇 QAQ

## 目录

数据结构与算法概念解析.....	2
一、数据结构之队列.....	4
二、数据结构之数组.....	12
三、数据结构之栈 .....	17
四、数据结构之树 .....	24
五、数据结构之二叉树.....	32
六、数据结构之各种树.....	39
JAVA 中的算法总结 .....	42
一、选择排序算法 .....	42
二、冒泡排序算法 .....	44
三、直接插入排序算法.....	47
四、快速排序算法 .....	52
五、希尔排序算法 .....	55
六、归并排序算法 .....	58
七、堆排序算法 .....	61
八、基数排序算法 .....	64
算法复杂度总结: .....	66

# 数据结构与算法概念解析



数据之间的相互关系称为**逻辑结构**。通常分为四类基本结构：

**集合结构**：结构中的数据元素除了同属于一种类型外，别无其它关系。

**线性结构**：结构中的数据元素之间存在一对一的关系。

**树型结构**：结构中的数据元素之间存在一对多的关系。

**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。

数据结构在计算机中有两种不同的存储方法：

**顺序存储结构**：用数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系。

**链式存储结构**：在每一个数据元素中增加一个存放地址的指针，用此指针来表示数据元素之间的逻辑关系。

## 时间复杂度

一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为  $T(n)$

在刚才提到的时间频度中， $n$  称为问题的规模，当  $n$  不断变化时，时间频度  $T(n)$  也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。

一般情况下，算法中基本操作重复执行的次数是问题规模  $n$  的某个函数，用  $T(n)$  表示，若有某个辅助函数  $f(n)$ ，使得当  $n$  趋近于无穷大时， $T(n)/f(n)$  的极限值为不等于零的常数，则称  $f(n)$  是  $T(n)$  的同数量级函数。记作  $T(n)=O(f(n))$ ，称  $O(f(n))$  为算法的渐进时间复杂度，简称时间复杂度。

有时候，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同，如在冒泡排序中，输入数据有序而无序，其结果是不一样的。此时，我们计算平均值。

常见的算法的时间 复杂度之间的关系为：

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2n) < O(n!) < O(n^n)$$

### 实例 1

```

-----
sum=0;                // (1)
  for(i=1;i<=n;i++)    // (2)
    for(j=1;j<=n;j++)  // (3)
      sum++;           // (4)
-----

```

语句 (1) 执行 1 次，  
 语句 (2) 执行  $n$  次  
 语句 (3) 执行  $n^2$  次  
 语句 (4) 执行  $n^2$  次  
 $T(n) = 1+n+2n^2 = O(n^2)$

### 实例 2

```

-----
a=0; b=1;             // (1)
for (i=1;i<=n;i++)    // (2)
{
    s=a+b;             // (3)
    b=a;               // (4)
    a=s;               // (5)
}
-----

```

语句 (1) 执行 1 次，  
 语句 (2) 执行  $n$  次  
 语句 (3)、(4)、(5) 执行  $n$  次  
 $T(n) = 1+4n = O(n)$

### 实例 3

-----

```

i=1;           // (1)
while (i<=n)
    i=i*2;      // (2)

```

-----  
 语句 (1) 的频度是 1,  
 设语句 2 的频度是  $f(n)$ , 则:  $2^{f(n)} \leq n; f(n) \leq \log_2 n$   
 取最大值  $f(n) = \log_2 n$ ,  
 $T(n) = O(\log_2 n)$

## 空间复杂度

空间复杂度: 算法所需存储空间的度量, 记作:

$$S(n) = O(f(n))$$

其中  $n$  为问题的规模。

一个算法在计算机存储器上所占用的存储空间, 包括存储算法本身所占用的存储空间, 算法的输入输出数据所占用的存储空间和算法在运行过程中临时占用的存储空间这三个方面。如果额外空间相对于输入数据量来说是个常数, 则称此算法是原地工作。

算法的输入输出数据所占用的存储空间是由要解决的问题决定的, 是通过参数表由调用函数传递而来的, 它不随本算法的不同而改变。存储算法本身所占用的存储空间与算法书写的长短成正比, 要压缩这方面的存储空间, 就必须编写出较短的算法。

## 一、数据结构之队列

队列是一种在一端进行插入, 而在另一端进行删除的线性表。

- 1、队列的插入端称为队尾; 队列的删除端称为队头。(好比火车进隧道)
- 2、队列的插入操作称为入队 (push), 删除操作称为出队 (pop)。

队列就像一列进入隧道的火车, 隧道就是队列, 火车车厢就是元素, 进入隧道就是从隧道的这头 (队尾) 插入元素, 出隧道就是从隧道的另一头 (队头) 删除元素。所以是“先进先出”的特点。

**存储结构:** 类似栈有顺序队和链式队两种。

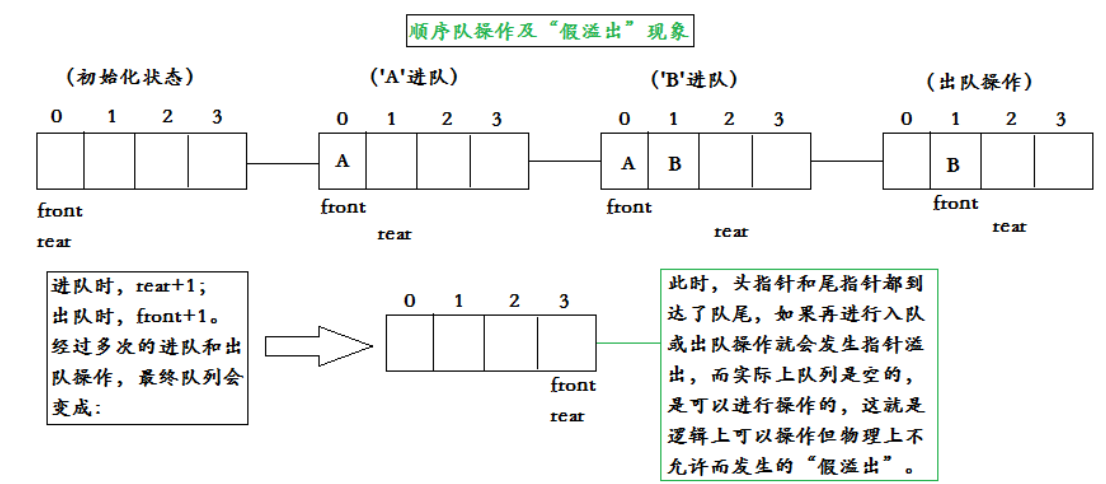
**java 实现** 我们可以围绕栈的 4 个元素来实现队列:

2 状态: 是否队空; 是否队满。

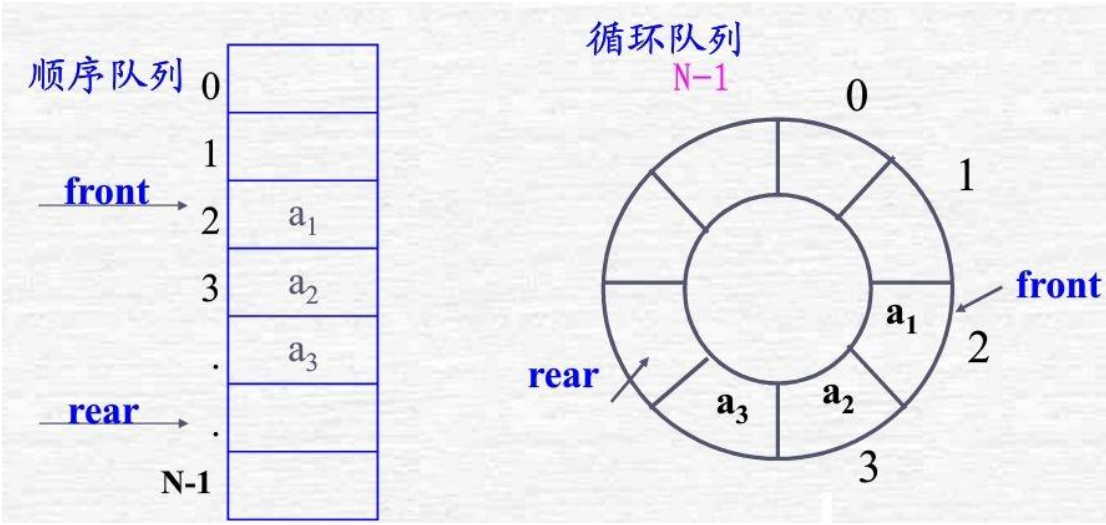
2 操作: 进队 push; 出队 pop。

顺序队的实现

顺序队列的实现也可以使用数组来完成，同栈的实现一样，只是栈是在同一端进行压栈和进栈操作，而队列是在一端做 push，另一端做 pop 操作。可以看一下下面的队列操作示意图：



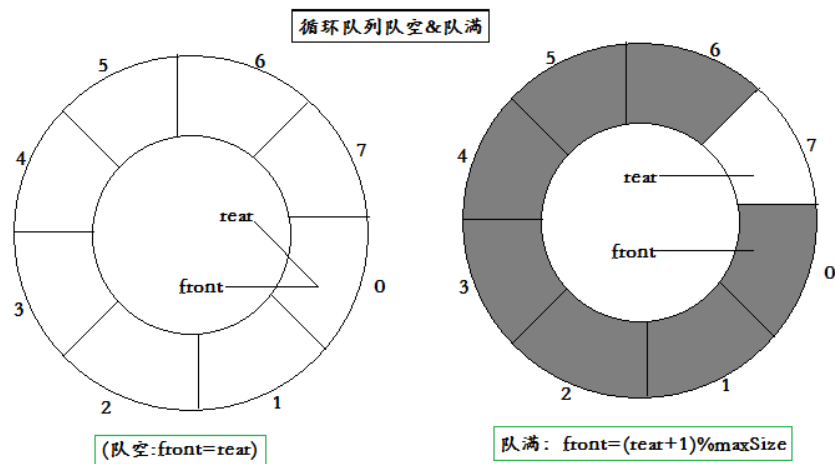
我们在实现顺序栈时使用头指针“front”和尾指针“rear”分别进行出队和入队操作，但普通的队列如上图所示，会发生“假溢出”现象，所以我们通常将数组弄成一个环状，即队头和队尾相连，这样就形成了“**循环队列**”，同时也解决了“假溢出”现象。循环队列是改进版的顺序队列。



对于普通队列的 push 或 pop 我们只需要对尾指针或头指针进行自增操作即可，但是循环队列我们就不能单纯的进行自增，当 front 或 rear=maxSize-1 时我们就不能进行自增操作了，比如一个队列尾长度为 4 的数组 datas[4]，那么当 front 或 rear 需要在 0, 1, 2, 3 之间进行循环“推进”，以此达到循环队列的效果。所以我们可以使用  $rear = (rear + 1) \% maxSize$  ；  $front = (front + 1) \% maxSize$  ；公式进行指针计算。

需要注意的是：队空状态的条件为：front = rear。而如果整个队列全部存满数据那么，队满的条件也是 front = rear；所以循环队列需要损失一个存储

空间，如下图：



解决了这些问题我们就可以很轻松地实现循环队列了：

-----  
`package test;`

```
public class SqQueue<T>{  
    private T[] datas;//使用数组作为队列的容器  
    private int maxSize;//队列的容量  
    private int front;//头指针  
    private int rear;//尾指针  
  
    //初始化队列  
    public SqQueue(int maxSize){  
        if(maxSize<1){  
            maxSize = 1;  
        }  
        this.maxSize = maxSize;  
        this.front = 0;  
        this.rear = 0;  
        this.datas = (T[])new Object[this.maxSize];  
    }  
  
    //两个状态:队空&队满  
    public boolean isNull(){  
        if(this.front == this.rear)  
            return true;  
        else  
            return false;  
    }  
  
    public boolean isFull(){
```

```

        if ((rear+1)%this.maxSize==front)
            return true;
        else
            return false;
    }

    //初始化队列
    public void initQueue(){
        this.front = 0;
        this.rear = 0;
    }

    //两个操作:进队&出队
    public boolean push(T data){
        if(isFull())
            return false;//队满则无法进队
        else{
            datas[rear] = data;//进队
            rear = (rear+1) % maxSize;//队尾指针+1.
            return true;
        }
    }

    public T pop(){
        if(isNull())
            return null;//对空无法出队
        else{
            T popData = datas[front++];//出队
            front = (front+1) % maxSize;//队头指针+1
            return popData;
        }
    }

    //get()
    public T[] getDatas() {
        return datas;
    }

    public int getMaxSize() {
        return maxSize;
    }

    public int getFront() {
        return front;
    }
}

```

```

    public int getRear() {
        return rear;
    }
}

```

测试:

```

package test;

import org.junit.Test;

public class testQueue {

    @Test
    public void fun() {
        SqQueue<Character> queue = new SqQueue<Character>(4);

        //判断
        System.out.println("队列是否为空: "+queue.isNull());

        //入队 A,B,C
        queue.push('A');
        queue.push('B');
        queue.push('C');

        System.out.println("队列是否为满: "+queue.isFull());

        //出队
        Character data = queue.pop();
        System.out.println("出队: "+data);
    }
}

```

运行结果:



```

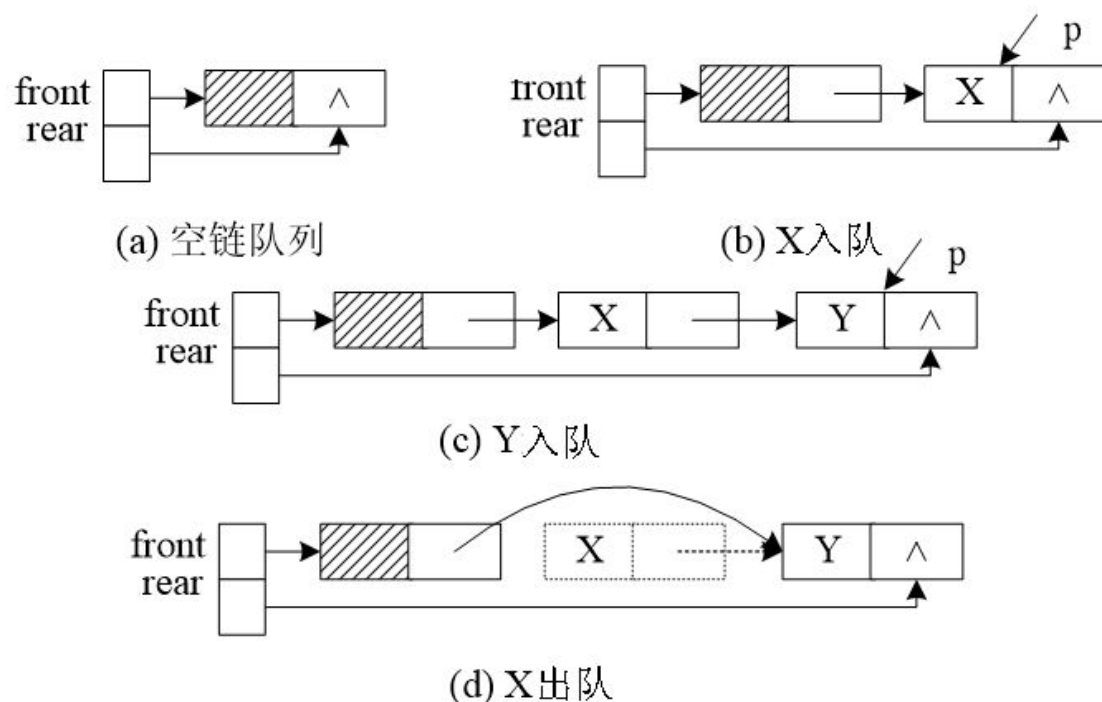
Servers Problems Tasks Web Browser Project Migration Debug
<terminated> testQueue [JUnit] G:\MyEclipse Professional\binary\com.sun.java.jdk.
队列是否为空: true
队列是否为满: true
出队: A

```

## 链队的实现



如图所示：



链队的实现很简单，只要理解了链表的操作和队列的特点即可。

```
package test;
```

```
public class LinkQueue<T>{
    private QNode<T> front;//队头指针
    private QNode<T> rear;//队尾指针
    private int maxSize;//为了便于操作，使用这个变量表示链队的数据容量

    //初始化
    public LinkQueue(){
        this.front = new QNode<T>();
        this.rear = new QNode<T>();
        this.maxSize = 0;
    }

    //初始化队列
    public void initQueue(){
        front.next = null;
        rear.next = null;
        maxSize = 0;
    }

    //队空判断
    public boolean isNull(){
        if(front.next==null || rear.next==null)
```

```

        return true;
    else
        return false;
}

//进队
public void push(QNode<T> node) {
    if(isNull()) {
        //第一次
        front.next = node;
        rear.next = node;
        maxSize++;
    }
    else{
        node.next = front.next;
        front.next = node;
        maxSize++;
    }
}

//出队
public QNode<T> pop() {
    if(isNull())
        return null; //队为空时，无法出队
    else if(maxSize==1) {
        //队只有一个元素时直接初始化即可
        QNode<T> node = front.next;
        initQueue();
        return node;
    }
    else{
        //准备工作
        QNode<T> p = front; //使用 p 指针来遍历队列
        for(int i=1; i<maxSize-1; i++)
            p = p.next;
        //pop
        QNode<T> node = rear.next;
        rear.next = p.next;
        maxSize--;
        return node;
    }
}
}

```

```

//链队结点
class QNode<T>{
    private T data;//数据域
    public QNode<T> next;//指针域

    //初始化 1
    public QNode() {
        this.data = null;
        this.next = null;
    }
    //初始化 2
    public QNode(T data) {
        this.data = data;
        this.next = null;
    }

    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
}

```

-----  
测试:

```

package test;

import org.junit.Test;

public class testQueue1 {

    @Test
    public void fun() {
        LinkQueue<Integer> lq = new LinkQueue<Integer>();

        System.out.println("队列是否为空: "+lq.isNull());

        //依次插入 1、2、3、4
        lq.push(new QNode<Integer>(1));
        lq.push(new QNode<Integer>(2));
        lq.push(new QNode<Integer>(3));
        lq.push(new QNode<Integer>(4));
    }
}

```

```

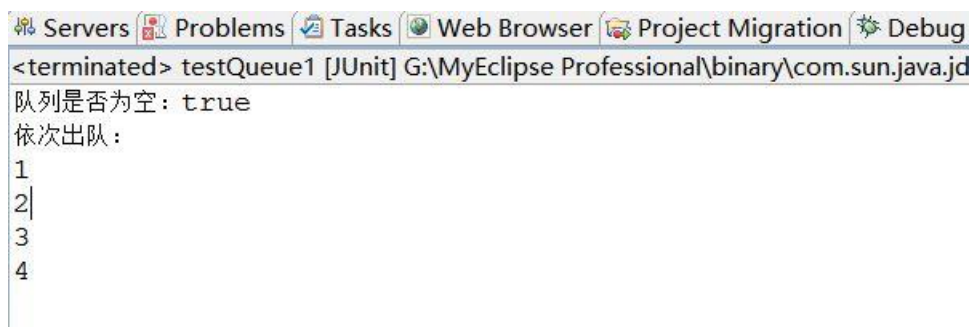
//依次出队
System.out.println("依次出队: ");
while(!lq.isNull()){
    System.out.println(lq.pop().getData());
}

}

}

```

运行结果:



```

<terminated> testQueue1 [JUnit] G:\MyEclipse Professional\binary\com.sun.java.jd
队列是否为空: true
依次出队:
1
2
3
4

```

## 二、数据结构之数组

数组是应用最广泛的一种数据结构，常常被植入到编程语言中，作为基本数据类型使用，因此，在一些教材中，数组并没有被当做一种数据结构单独拿出来讲解（其实数组就是一段连续的内存，即使在物理内存中不是连续的，在逻辑上肯定是连续的）。其实没必要在概念上做纠缠，数组可以当做学习数据结构的敲门砖，以此为基础，了解数据结构的基本概念以及构建方法。

数据结构不仅是数据的容器，还要提供对数据的操作方法，比如检索、插入、删除、排序等。

### 无序数组

下面我们建立一个类，对数组的检索、插入、删除、打印操作进行封装，简便起见，我们假设数组中没有重复值（实际上数组可以包含重复值）

```

public class Array {

    private String [] strArray;
    private int length = 0;        //数组元素个数

    //构造方法，传入数组最大长度
    public Array(int max){
        strArray = new String [max];
    }
}

```

```

//检测数组是否包含某个元素，如果存在返回其下标，不存在则返回-1
public int contains(String target){
    int index = -1;
    for(int i=0;i<length;i++){
        if(strArray[i].equals(target)){
            index = i;
            break;
        }
    }
    return index;
}

//插入
public void insert(String elem) {
    strArray[length] = elem;
    length++;
}

//删除某个指定的元素值，删除成功则返回 true，否则返回 false
public boolean delete(String target){
    int index = -1;
    if((index = contains(target)) != -1){
        for(int i=index;i<length-1;i++){
            //删除元素之后的所有元素前移一位
            strArray[i] = strArray[i+1];
        }
        length--;
        return true;
    }else{
        return false;
    }
}

//列出所有元素
public void display(){
    for(int i=0;i<length;i++){
        System.out.print(strArray[i]+"\\t");
    }
}

}

```

-----  
**无序数组的优点：** 插入快，如果知道下标，可以很快的存取

无序数组的缺点：查找慢，删除慢，大小固定。

## 有序数组

所谓的有序数组就是指数组中的元素是按一定规则排列的，其好处就是在根据元素值查找时可以使用二分查找，查找效率要比无序数组高很多，在数据量很大时更加明显。当然缺点也显而易见，当插入一个元素时，首先要判断该元素应该插入的下标，然后对该下标之后的所有元素后移一位，才能进行插入，这无疑增加了很大的开销。

因此，有序数组适用于查找频繁，而插入、删除操作较少的情况。

有序数组的封装类如下，为了方便，我们依然假设数组中是没有重复值的，并且数据是按照由小到大的顺序排列的。

```
-----
public class OrderArray {
    private int[] intArray;
    private int length = 0; // 数组元素个数

    // 构造方法，传入数组最大长度
    public OrderArray(int max) {
        intArray = new int[max];
    }

    // 用二分查找法定位某个元素，如果存在返回其下标，不存在则返回-1
    public int find(int target) {
        int lowerBound = 0; // 搜索段最小元素的小标
        int upperBound = length - 1; // 搜索段最大元素的下标
        int curIn; // 当前检测元素的下标

        if (upperBound < 0) { // 如果数组为空，直接返回-1
            return -1;
        }

        while (true) {
            curIn = (lowerBound + upperBound) / 2;

            if (target == intArray[curIn]) {
                return curIn;
            } else if (curIn == lowerBound) { // 在当前下标与搜索段的最小下标重合时，代表搜索段中只包含 1 个或 2 个元素
                // 既然走到该分支，证明上一个 if 分支不满足，即目标元素不等于低位元素
            }
        }
    }
}
```

```

        if (target == intArray[upperBound]) { // 等于高位元素，返回
            return upperBound;
        } else { // 高位元素也不等于目标元素，证明数组中没有该元素，
搜索结束
            return -1;
        }
    } else { // 搜索段中的元素至少有三个，且当前元素不等于目标元素
        if (intArray[curIn] < target) {
            // 如果当前元素小于目标元素，则将下一个搜索段的最小下
标置为当前元素的下标
            lowerBound = curIn;
        } else {
            // 如果当前元素大于目标元素，则将下一个搜索段的最大下
标置为当前元素的下标
            upperBound = curIn;
        }
    }
}
}
}

```

```

// 插入
public void insert(int elem) {
    int location = 0;

    // 判断应插入位置的下标
    for (; location < length; location++) {
        if (intArray[location] > elem)
            break;
    }
    // System.out.println(location);
    // 将插入下标之后的所有元素后移一位
    for (int i = length; i > location; i--) {
        intArray[i] = intArray[i - 1];
    }

    // 插入元素
    intArray[location] = elem;

    length++;
}

```

```

// 删除某个指定的元素值，删除成功则返回 true，否则返回 false
public boolean delete(int target) {
    int index = -1;

```

```

        if ((index = find(target)) != -1) {
            for (int i = index; i < length - 1; i++) {
                // 删除元素之后的所有元素前移一位
                intArray[i] = intArray[i + 1];
            }
            length--;
            return true;
        } else {
            return false;
        }
    }

// 列出所有元素
public void display() {
    for (int i = 0; i < length; i++) {
        System.out.print(intArray[i] + "\t");
    }
    System.out.println();
}

public static void main(String[] args) {
    OrderArray orderArray = new OrderArray(4);

    orderArray.insert(3);
    orderArray.insert(4);
    orderArray.insert(6);
    orderArray.insert(8);

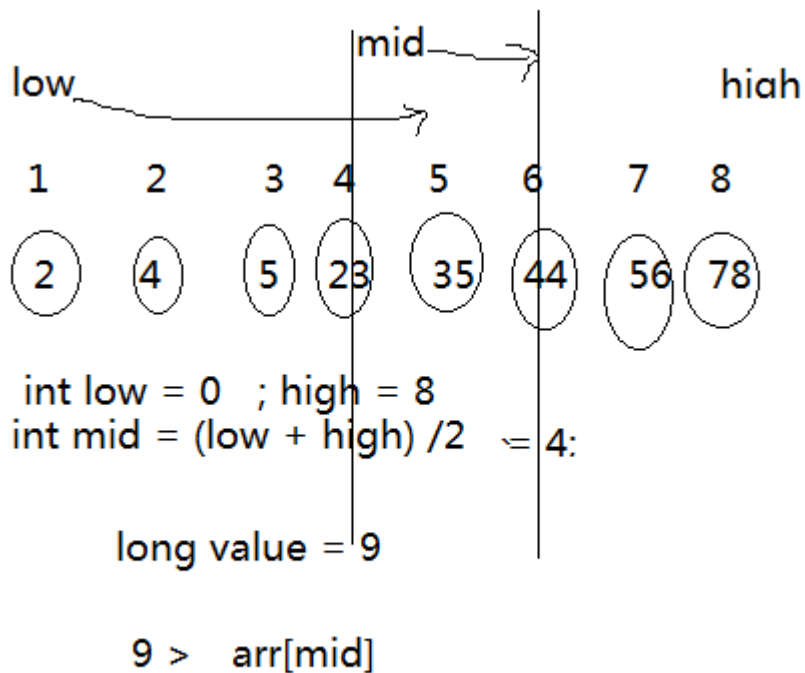
    int i = orderArray.find(8);
    System.out.println("在队列中的位置是" + i);
}
}

```

-----

有序数组最大的优势就是可以提高查找元素的效率，在上例中，find 方法使用了二分查找法，该算法的示意图如下：





这个方法在一开始设置变量 `lowerBound` 和 `upperBound` 指向数组的第一个和最后一个非空数据项。通过设置这些变量可以确定查找的范围。然后再 `while` 循环中，当前的下标 `curIn` 被设置为这个范围的中间值

如果 `curIn` 就是我们要找的数据项，则返回下标，如果不是，就要分两种情况来考虑：如果 `curIn` 指向的数据项比我们要找的数据小，则证明该元素只可能在 `curIn` 和 `upperBound` 之间，即数组后半段中（数组是从小到大排列的），下轮要从后半段检索；如果 `curIn` 指向的数据项比我们要找的数据大，则证明该元素只可能在 `lowerBound` 和 `curIn` 之间，下一轮要在前半段中检索按照上面的方法迭代检索，直到结束

**有序数组的优点：** 查找效率高

**有序数组的缺点：** 删除和插入慢，大小固定

### 三、数据结构之栈

#### 概念

栈是一种只允许在一端进行插入或删除的线性表。

- 1、栈的操作端通常被称为栈顶，另一端被称为栈底。
- 2、栈的插入操作称为进栈（压栈|`push`）；栈删除操作称为出栈（弹栈|`pop`）。

#### 特点

栈就像一个杯子，我们只能从杯口放和取，所以栈中的元素是“先进后出”的特点。

#### 存储结构

顺序存储的栈称为顺序栈；链式存储的栈称为链式栈。

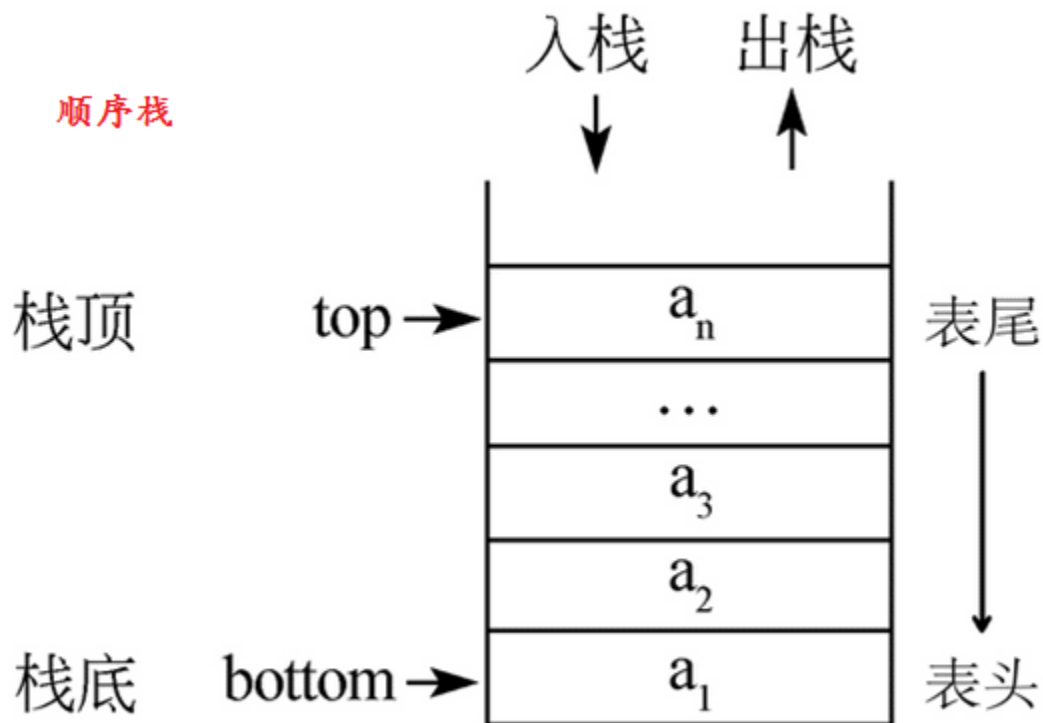
### java 实现

我们可以围绕栈的 4 个元素来实现栈：

2 状态：是否栈空；是否栈满。

2 操作：压栈 push；进栈 pop。

### 顺序栈的实现



```
-----  
package test;  
/**  
 *  
 * @author 皓宇 QAQ  
 * @date 2019 年 3 月 1 日  
 * @Description TODO:  
 * 顺序栈(SqStack)一般用数组来实现，主要有四个元素：2 状态 2 操作。  
 * 2 状态：栈空？；栈满？  
 * 2 操作：压栈 push；弹栈 pop。  
 * @param <T>  
 */  
public class SqStack<T> {  
    private T data[]; // 用数组表示栈元素  
    private int maxSize; // 栈空间大小 (常量)  
    private int top; // 栈顶指针 (指向栈顶元素)
```

```

@SuppressWarnings("unchecked")
public SqStack(int maxSize){
    this.maxSize = maxSize;
    this.data = (T[]) new Object[maxSize]; //泛型数组不能直接 new
创建, 需要使用 Object 来创建 (其实一开始也可以直接使用 Object 来代替泛型)
    this.top = -1; //有的书中使用 0, 但这样会占用一个内存
}

//判断栈是否为空
public boolean isEmpty(){
    boolean flag = this.top <= -1 ? true : false;
    return flag;
}

//判断是否栈满
public boolean isFull(){
    boolean flag = this.top == this.maxSize - 1 ? true : false;
    return flag;
}

//压栈
public boolean push(T vaule){
    if(isFull()){
        //栈满
        return false;
    }else{
        data[++top] = vaule; //栈顶指针加 1 并赋值
        return true;
    }
}

//弹栈
public T pop(){
    if(isEmpty()){
        //栈为空
        return null;
    }else{
        T value = data[top]; //取出栈顶元素
        --top; //栈顶指针-1
        return value;
    }
}

```

```
}
```

测试:

```
package test;

import org.junit.Test;

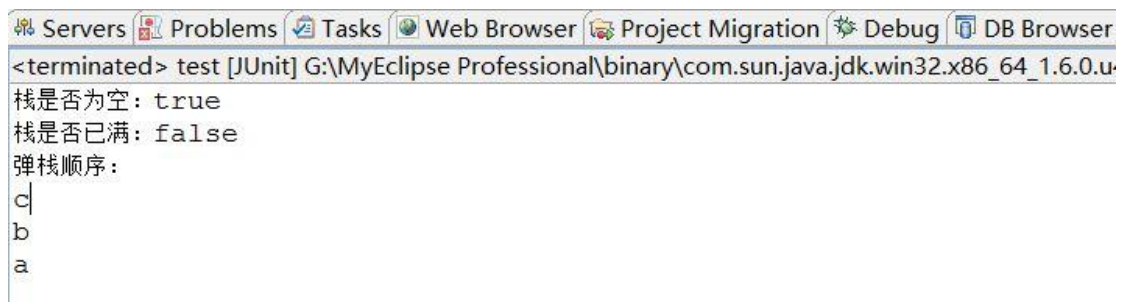
public class test {

    @Test
    public void fun() {
        //初始化栈(char 类型)
        SqStack<Character> stack = new SqStack<Character>(10);

        //2 状态
        System.out.println("栈是否为空: "+stack.isNull());
        System.out.println("栈是否已满: "+stack.isFull());

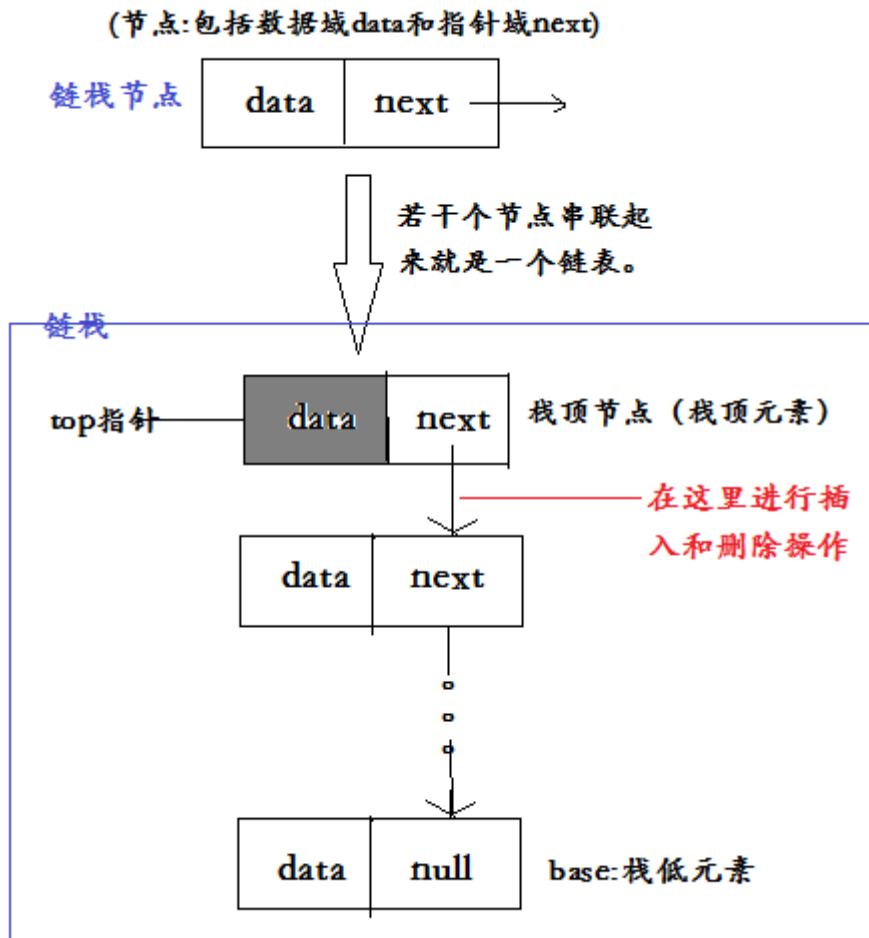
        //2 操作
        //依次压栈
        stack.push('a');
        stack.push('b');
        stack.push('c');
        //依次弹栈
        System.out.println("弹栈顺序: ");
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

运行结果:



```
<terminated> test [JUnit] G:\MyEclipse Professional\binary\com.sun.java.jdk.win32.x86_64_1.6.0.u
栈是否为空: true
栈是否已满: false
弹栈顺序:
c
b
a
```

链式栈的实现:



```
package test;
```

```
/**
 * @author 皓宇 QAQ
 * @date 2019 年 3 月 1 日
 * @Description TODO:尹海鹏是煞笔
 * 链栈(LinkStack)用链表来实现,主要有四个元素: 2 状态 2 操作。
 * 2 状态: 栈空? ; 栈满 (逻辑上永远都不会栈满, 除非你的电脑没内存了^_^)。
 * 2 操作: 压栈 push; 弹栈 pop。
 * @param <T>
 */
class LinkStack<T>{
    //栈顶节点
    private LinkNode<T> top;

    //初始化 1
    public LinkStack(){
        this.top = new LinkNode<T>();
    }
}
```

```

//初始化栈
public void initStack(){
    this.top.setData(null);
    this.top.setNext(null);
}

//是否栈空
public boolean isNull(){
    boolean flag = top.getNext()==null?true:false;
    return flag;
}

//压栈
public void push(LinkNode<T> node){
    if(isNull()){
        //栈空，即第一次插入
        top.setNext(node);
        node.setNext(null); //该句可以省略(首次插入的元素为栈底元素)
    }else{
        node.setNext(top.getNext());
        top.setNext(node);
    }
}

//弹栈
public LinkNode<T> pop(){
    if(isNull()){
        //栈空无法弹栈
        return null;
    }else{
        LinkNode<T> delNode = top.getNext(); //取出删除节点
        top.setNext(top.getNext().getNext()); //删除节点
        return delNode;
    }
}
}

```

//链式栈节点（外部类实现，也可以使用内部类）

```

class LinkNode<T>{
    private T data; //数据域
    private LinkNode<T> next; //指针域
}

```

```

//初始化 1
public ListNode() {
    this.data = null;
    this.next = null;
}

//初始化 2
public ListNode(T data) {
    super();
    this.data = data;
    this.next = null;
}

public T getData() {
    return data;
}

public void setData(T data) {
    this.data = data;
}

public ListNode<T> getNext() {
    return next;
}

public void setNext(ListNode<T> next) {
    this.next = next;
}
}

-----
测试:
-----
package test;

import org.junit.Test;

public class test {

    @Test
    public void fun() {
        LinkStack<Character> ls = new LinkStack<Character>();

        //1 状态
        System.out.println("栈是否为空: "+ls.isNull());

        //2 操作
        //依次压栈
        ls.push(new ListNode<Character>('a'));
    }
}

```

```

ls.push(new LinkNode<Character>('b'));
ls.push(new LinkNode<Character>('c'));

//依次弹栈
System.out.println("弹栈顺序: ");
System.out.println(ls.pop().getData());
System.out.println(ls.pop().getData());
System.out.println(ls.pop().getData());
}
}

```

运行结果:

```

Servers Problems Tasks Web Browser Project Migration Debug DB Browser
<terminated> test [JUnit] G:\MyEclipse Professional\binary\com.sun.java.jdk.win32.x86_64_1.6.0.u4
栈是否为空: true
弹栈顺序:
c
b
a
|

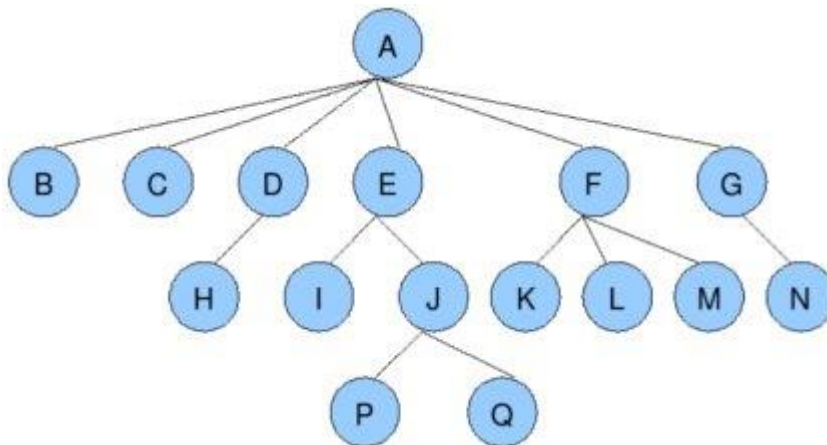
```

## 栈的应用

栈结构是很基本的一种数据结构，所以栈的应用也很常见，根据栈结构“先进后出”的特点，我们可以在很多场景中使用栈，下面我们就是使用上面我们已经实现的栈进行一些常见的应用：十进制转 N 进制、行编辑器、校验括号是否匹配、中缀表达式转后缀表达式、表达式求值等。

## 四、数据结构之树

树结构可以说是我们日常生活中“树”的一种抽象，树只有一个根，其余都是树的枝叶，数据结构中的树类似一颗“倒立”的树，如下：





这颗树是由若干结点（A-Q）组成，所以树是由唯一的根结点和互不相干的若干子树组成。树的定义：由  $n$  ( $n \geq 0$ ) 个有限结点组成一个具有层次关系的集合。

当结点数  $n=0$  时，称为空树，这是一种特殊状态。由上可知，树的定义是递归的，即在整颗树中，若干子树也是树的定义。

### 树的基本术语

**结点：**A-Q 都是结点，结点不仅包含了数据元素，还包含了指向子树的分支信息，如 A 结点就包含了 6 个分支信息。（类似链式存储中的结点，既有数据域，又有指针域）。

**结点的度：**结点拥有的子树个数（或分支个数）。如 A 结点的度为 6，B 结点的度为 0。

**树的度：**树中各个度的最大值。如该树的度就是 A 结点的度，为 6。

**叶子结点：**度为 0 的结点。如 B、C、H、I、K、L、M、N、P、Q 都是树的叶子结点。

**孩子：**结点子树的根结点称为孩子。如 E 的孩子 I 和 J。（所以叶子结点莫有孩子）

**双亲：**与孩子的定义对应，如 E 的孩子 I 和 J，那么反过来，I 和 J 的双亲就是 E。

**兄弟：**同一双亲的孩子互相称为兄弟，比如 P 和 Q。

**祖先：**从根结点到某结点的路径上的所有结点，都是该结点的祖先，如路径：A-E-J-Q，那么 Q 的结点的祖先就是 A、E、J。

**子孙：**和祖先对应，以某结点作为根结点的所有子树的结点都是该结点的子孙，如路径：A-E-J-Q，那么 A 的子孙就是 E、J、Q。

**层次：**从根开始为第一层，根的孩子为第二层，根的孩子孩子为第三层，依次类推。可以理解为家族树中的辈份概念，所有该树有 4 层。

**树的高度：**或称为树的深度，表示树中结点的最大层次即为树的高度，例如该树的高度为 4。（注意和树的度加以区分）。

**结点的高度和深度：**两个相反地概念，结点的高度是从叶子结点开启计算层数，而结点的深度是从根结点开始计算层数。例如 B 的高度为：3，深度为 2。

**堂兄弟：**双亲在同一层的结点互称为堂兄弟。

**有序树：**树中结点的子树从左往右是有次序的，不能交换，这样的树称为有序树。（类似兄弟有长幼之分，兄必须在左，弟必须在右）。

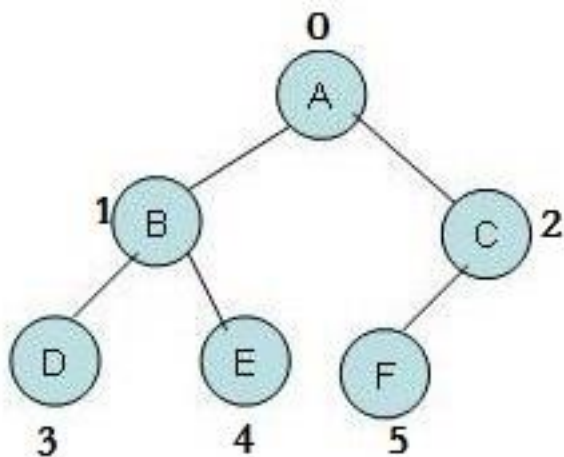
**无序树：**与有序树对应，子树的左右顺序可以交换的树。

**丰满树：**即理想平衡树，除了最底层外其他层都是满的（即叶子结点全部在最底层的树）。

**森林：**若干互不相交的树组成的集合称为森林。

### 树的存储结构：

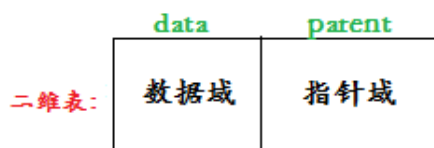
为了方便讲解，我们使用下图作为案例树进行说明。



### 双亲存储结构:

我们利用树的双亲唯一的这个特点，存储每个结点时只需要存储自身数据和双亲即可完成整颗树的存储，这就是双亲存储的原理。双亲存储结构是一种树的顺序存储结构，使用二维数组即可实现。在存储时我们先从上往下、从左往右依次为树结点标上序号，这个序号也代表数组的角标，这个角标就用来表示这个结点的索引。

#### 双亲存储结构



由于双亲结点是唯一的。所以  
**数据域**表示结点的数据元素；  
**指针域**指向双亲结点。

二维数组实现



data	parent	下标
A	-1	[0]
B	0	[1]
C	0	[2]
D	1	[3]
E	1	[4]
F	2	[5]
		[max]

### 孩子存储结构:

除了围绕双亲来存储我们还可以**围绕孩子来存储树**，但是树中每个结点可能有多棵子树，可以考虑用多重链表，即每个结点有多个指针域，其中每个指针指向一棵子树的根结点，我们把这种方法叫做多重链表表示法(或说孩子存储法)。

孩子存储结构的结点设计如下:

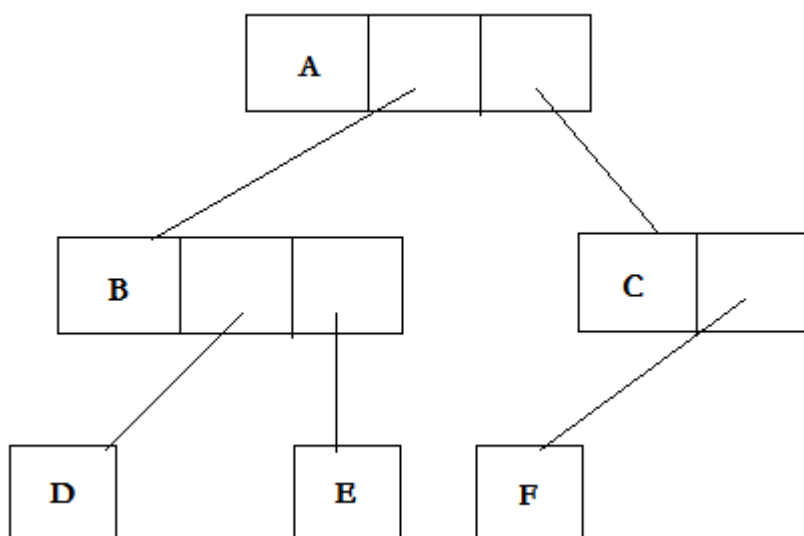
### 孩子存储结构的节点设计

data	child1	child2	child3	.....	childn
数据域		指针域：指向孩子节点			

对于结点的设计，有两种方案可选：

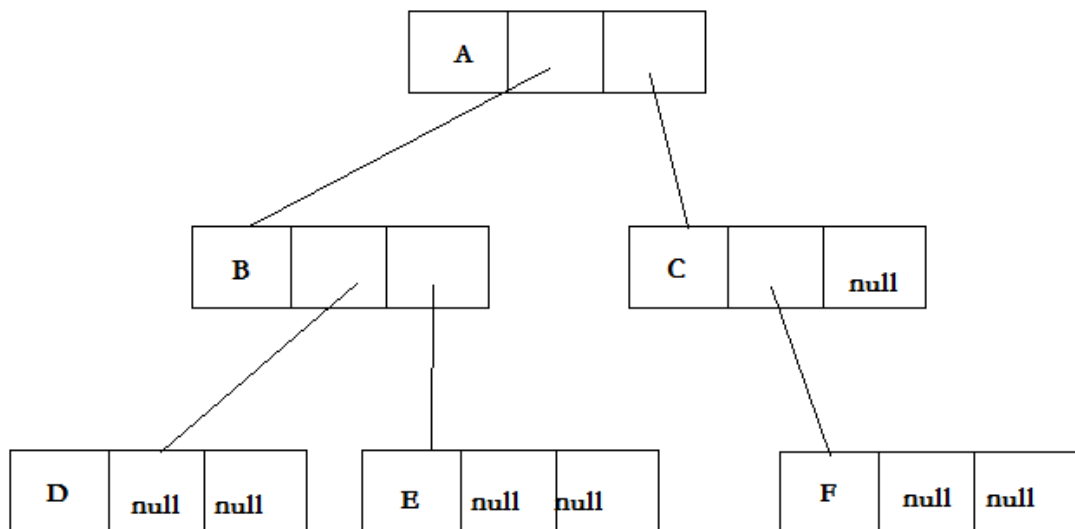
方案一：每个结点都设计对应的结构。

如 A 结点有两个孩子，所以 A 的指针域就设计为两个；C 结点只有一个孩子，所以指针域就设计为 1 个；以此类推。上图案例的树就可以设计为：



缺点：这样设计有个很明显的缺点，如果有很多结点的结构都不一样，那么就要设计多种结点结构来保存结点信息，这样不利于统一管理。

方案二：针对方案一的缺点，我们应该将树的**结点结构统一**（统一的关键在于**指针域数量的统一**），比如 A 有两个孩子，而 C 有一个孩子，那么我们需要使用一样的结构保存 A 和 C 就需要设计至少两个指针域的结点。依次类推，我们要使用一个全部结点都能使用的结构，那么我们就需要将指针域的数量设计为最多孩子的结点的度（孩子的数量）。即该树的度作为指针域的数量。所以我们将案例树的结点设计为 2 个指针域。

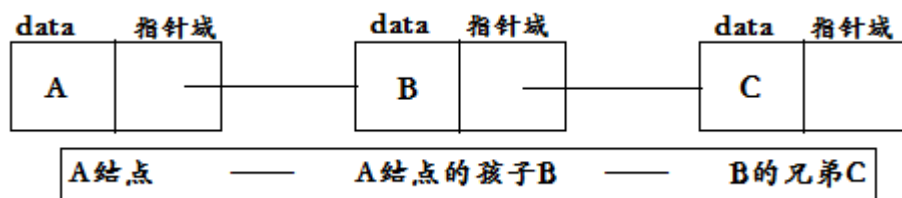


**缺点：**这样设计可以统一管理结点，但是当结点之间的度相差太大时是否会浪费存储空间，比如 A 结点有 1 个孩子，B 结点有 10 个孩子，那么按照 10 个指针域的设计，A 结点就会浪费 9 个指针域。

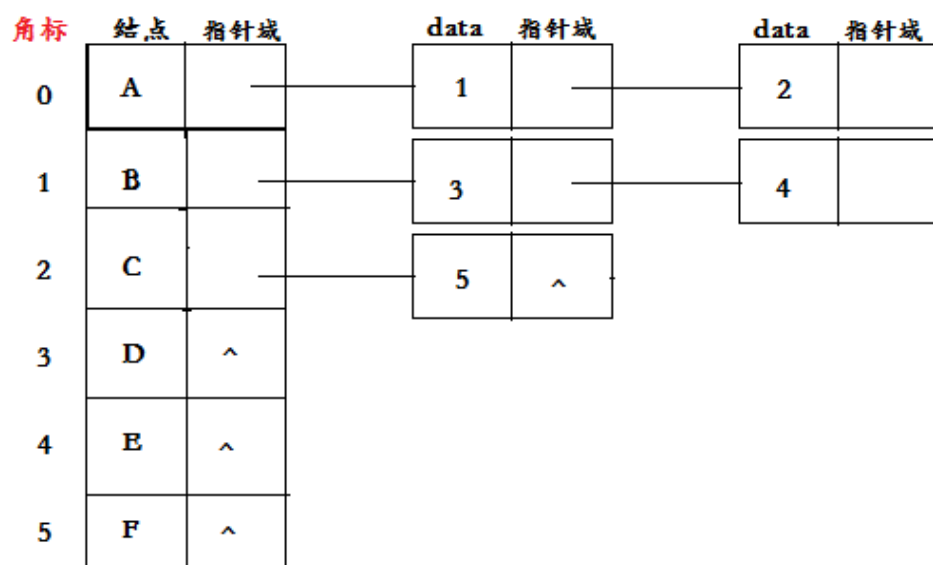
方案三（重要）：换一种思维。

首先树可以由两个关系表示出来：**结点——孩子——兄弟**。（双亲与孩子之间的关系；孩子与孩子之间的兄弟关系）。所以这两个关系都可以用“数据域——指针域”的设计来表示。并且每一个结点都可以这样来设计：结点——孩子——兄弟。

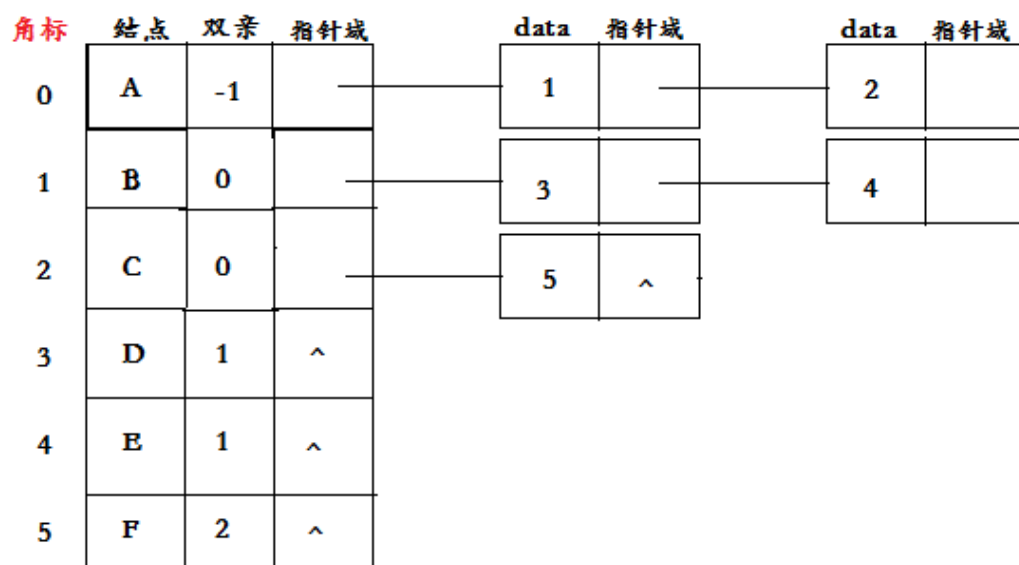
例如 A 结点即它的孩子们就可以表示为：



依次类推，我们可以将所有结点都用这种“双亲——孩子——兄弟”的结构表示出来。但这样如果有 n 个结点就会得到 n 个单链表，而这 n 个单链表之间都没有联系，所以我们按照还是按照从上到下、从左到右的顺序为各个结点排序，并用这些下标表示各个结点的位置，这样就可以将这些分散的单链表联系起来。如图：



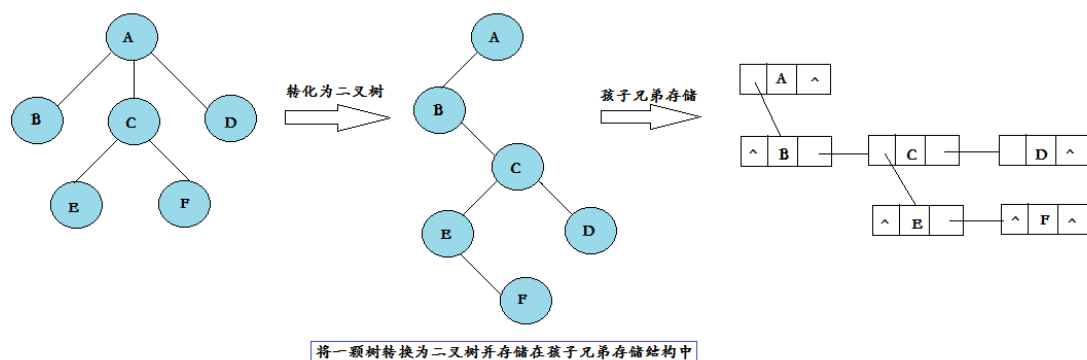
改进：该方案解决了结点结构不统一和空间浪费的问题，但任然可以进行改进，注意方案三的设计，我们将双亲和孩子串联起来，知道了某一结点，很容易就知道它的孩子，但如果要找到该结点的双亲需要进行遍历比较麻烦，而我们知道双亲是唯一的，所以我们可以将新增一个指针域指向双亲，所以改进结构就变成了：结点——双亲——孩子——兄弟。



### 3. 孩子兄弟存储结构（重要）

这种存储方式是将树转换为二叉树然后再进行孩子兄弟存储。孩子兄弟存储结构与二叉树的关系十分密切，使用也比较广泛，因为这种存储结构能够把树或森林转换为二叉树。

（由于案例树已经是二叉树，这里我使用下图的树作为新的案例树）：

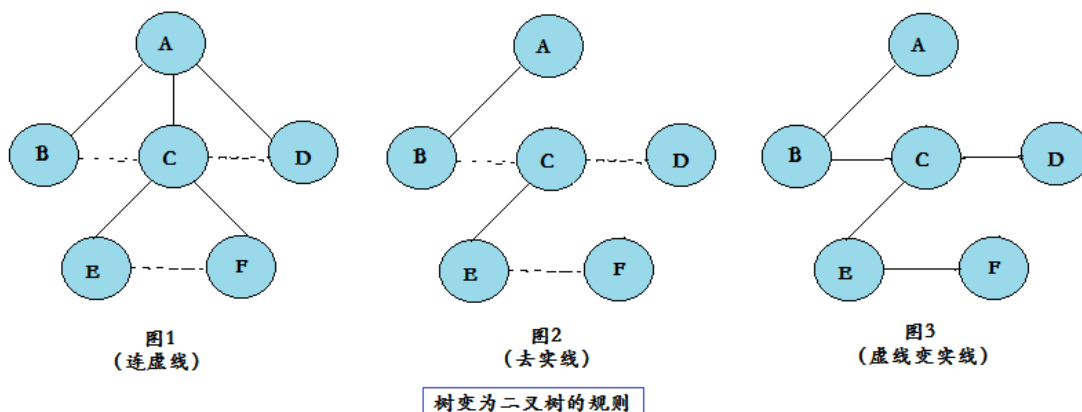


## 树和森林转化为二叉树

### 1. 普通树转化为二叉树

将普通树转换为二叉树的规则：

- 1) 将同一层的孩子结点用虚线串起来（如图 1）
- 2) 将每个结点的分支（实线）除了最左边的第一条线全部剪掉（如图 2）
- 3) 将虚线变为实线就形成了二叉树。（如图 3）



孩子兄弟存储结构其实是利用二叉树的规范的优点，先将树转化为二叉树再进行存储问题就变得简单多了。

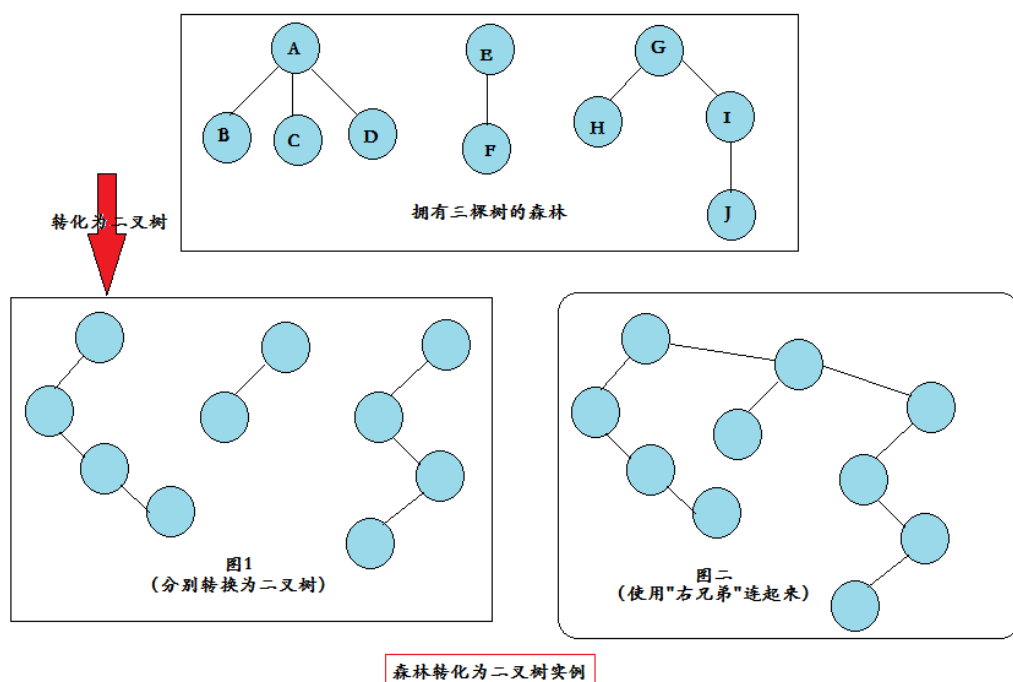
### 2. 森林转换为二叉树

如果你懂了普通树转换为二叉树的方法，那么森林转换为二叉树也就变得十分简单了。

注意普通树转换成的二叉树有一个特点：**根结点没有右兄弟**（这里说的右兄弟是根结点的孩子的兄弟）。而森林是由若干不相连的树组成的，所以我们将森林里的没颗树转化为二叉树，然后利用“没有右兄弟”这个特性，第二颗树的根结点作为第一颗树的右兄弟，第三颗树作为第二颗树的右兄弟，依次类推。这样就将所有分散的二叉树连接成为一颗二叉树。

将森林转换为二叉树的规则：

- 1) 分别将森林里的树转化为二叉树（如图 1）
- 2) 从左往右，右边的树的结点作为左边树结点的右兄弟连接起来即可。



## 树和森林的遍历

我们常说的树的三种遍历方式，其实是二叉树的三种遍历方式：前序遍历、中序遍历和后序遍历。

注意，普通树的遍历只有先序遍历和后序遍历两种（即普通树没有中序遍历）；而森林的遍历方式只有先序遍历和中序遍历两种（即森林没有后序遍历）。不要混淆。由于树和森林都可以转换为二叉树来存储，所以我们一般只需要重点掌握二叉树的三种遍历方式就可以了。

### 1、三者之间遍历的联系：

1) 树转化为二叉树，那么树的先序遍历对应二叉树的先序遍历；树的后序遍历对应二叉树的中序遍历。

2) 森林转化为二叉树，那么森林的先序遍历对应二叉树的先序遍历；树的中序遍历对应二叉树的中序遍历。

2、遍历规则（注意，由于树的定义本身就存在递归，所以遍历是也存在递归）

**树的先序遍历：**先访问根结点，再依次先序遍历根结点的每颗子树。

**树的后序遍历：**先依次访问根结点的子树，再访问根结点。

（森林是由树组成的，所以森林的遍历是在树的遍历继承之上的）

**森林的先序遍历：**先访问森林第一棵树的根结点，先序遍历第一颗树的子树，然后再先序遍历森林中除第一颗树外的其他树。（其实就是从左往右分别对每棵树进行先序遍历的效果）。

**森林的中序遍历：**中序遍历第一颗树根结点的子树，访问第一颗树的根结点，然后再中序遍历森林中除第一颗树外的其他树。（其实就是从左往右分别对每棵树进行后序遍历的效果）。

### 3、结论

1、森林的遍历其实就是若干树从左往右进行（先序或后序）遍历的结果。（所以其实森林的中序遍历其实是树后序遍历的结果，我们称为中序遍历而不是后序遍历是因为，后序遍历遇到根结点表示遍历结束，而森林中的根结点不止一个，遇到根结点可能并没有结束，所以不宜称为后序遍历）。

2、那为什么普通树没有中序遍历呢？因为普通树的孩子为若干个，不像二叉树刚好分为“左根右”，根结点正好符合中序遍历的“中间位置”，而普通树如果有中序遍历，那么这个根结点什么时候遍历（即放在哪个子树之间）算是中呢？所以，**树没有中序遍历**。

## 五、数据结构之二叉树

### 一、二叉树的定义

如果你知道树的定义（有限个结点组成的具有层次关系的集合），那么就很好理解二叉树了。定义：二叉树是  $n (n \geq 0)$  个结点的有限集，二叉树是每个结点最多有两个子树的树结构，它由一个根结点及左子树和右子树组成。（这里的左子树和右子树也是二叉树）。

值得注意的是，二叉树和“度至多为 2 的有序树”几乎一样，但，二叉树不是树的特殊情形。具体分析如下

### 二、二叉树为何不是特殊的树

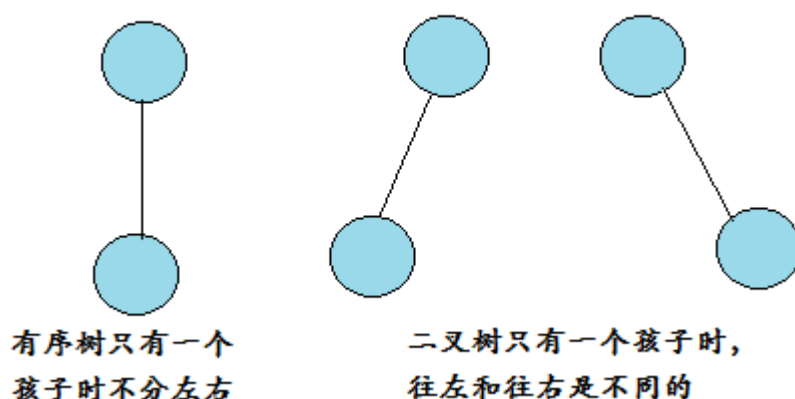
#### 1、二叉树与无序树不同

二叉树的子树有左右之分，不能颠倒。无序树的子树无左右之分。

#### 2、二叉树与有序树也不同（关键）

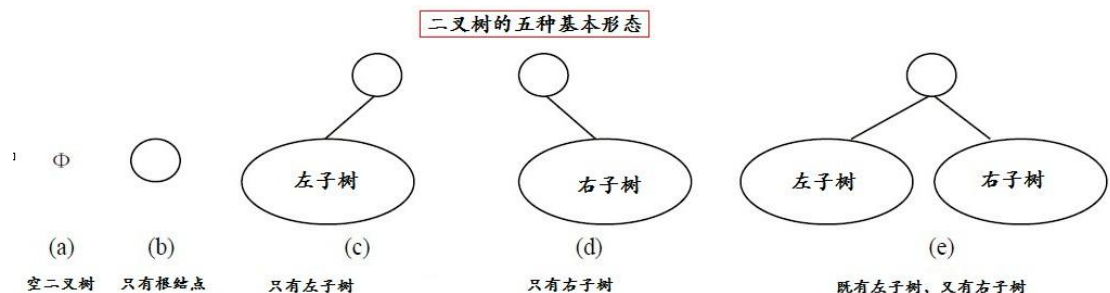
当有序树有两个子树时，确实可以看做一颗二叉树，但当只有一个子树时，就没有了左右之分，如图所示：

有序树和二叉树不同之处



### 三、二叉树的五种基本状态

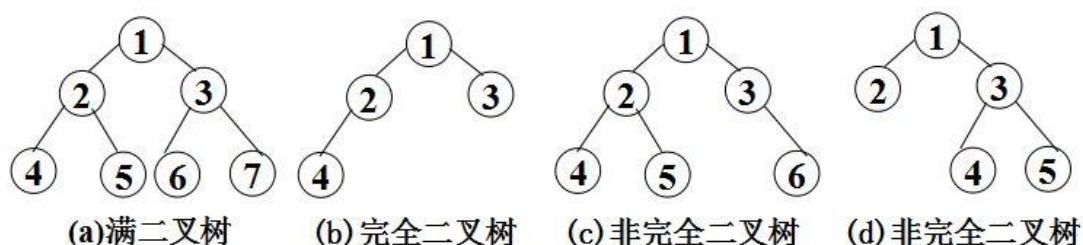




#### 四、二叉树相关术语

**满二叉树：**所有叶子结点全部集中在最后一层，这样的二叉树称为满二叉树。  
 （注意：国内的定义是每一层的结点都达到最大值时才算是满二叉树；而国际定义为，不存在度为1的结点，即结点的度要么为2要么为0，这样的二叉树就称为满二叉树。这两种概念完全不同，既然在国内，我们就默认第一种定义就好。）

**完全二叉树：**如果将一颗深度为K的二叉树按从上到下、从左到右的顺序进行编号，如果各结点的编号与深度为K的满二叉树相同位置的编号完全对应，那么这就是一颗完全二叉树。如图所示：



#### 五、二叉树的主要性质

二叉树的性质是基于它的结构而得来的，这些性质不必死记，使用到再查询或者自己根据二叉树结构进行推理即可。

**性质 1：**非空二叉树的叶子结点数等于双分支结点数加 1。

证明：设二叉树的叶子结点数为  $X$ ，单分支结点数为  $Y$ ，双分支结点数为  $Z$ 。  
 则 **总结点数** =  $X + Y + Z$ ，**总分支数** =  $Y + 2Z$ 。

由于二叉树除了根结点外其他结点都有唯一的分支指向它，所以总分支数 = 总结点数 - 1。

结合三个方程：**总分支数** = **总结点数** - 1，即  $Y + 2Z = X + Y + Z - 1$ 。化简得到  $X = Z + 1$ 。即叶子结点数等于双分支结点数加 1。

**性质 2：**在二叉树的第  $i$  层上最多有  $2^{i-1}$  个结点 ( $i \geq 1$ )。

证明：二叉树结点最多的情况即为满二叉树的情况，由于是满二叉树，每个结点都有两个孩子，所以下一层是上一层的 2 倍，构成了公比为 2 的**等比数列**，而第一层只有根结点，所以首项是 1。所以二叉树的第  $i$  层上最多有  $2^{i-1}$  个结点。

**性质 3：**高度（或深度）为  $K$  的二叉树最多有  $2^k - 1$  个结点 ( $K \geq 1$ )。

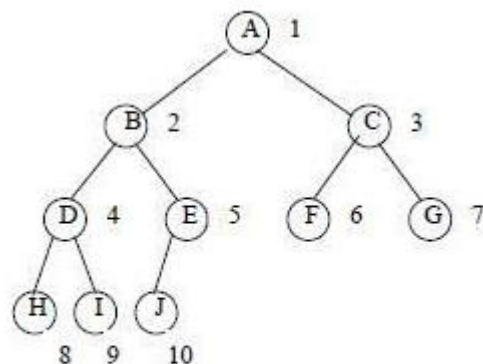
证明：本性质其实就是性质 2 中描述的等比数列的前项和的问题。

**性质 4：**若对含  $n$  个结点的完全二叉树从上到下且从左至右进行 1 至  $n$  的编号，则对完全二叉树中任意一个编号为  $i$  的结点：

(1) 若  $i=1$ ，则该结点是二叉树的根，无双亲，否则，编号为  $[i/2]$  的结点为其双亲结点；

(2) 若  $2i > n$ ，则该结点无左孩子， 否则，编号为  $2i$  的结点为其左孩子结点；

(3) 若  $2i+1 > n$ ，则该结点无右孩子结点， 否则，编号为  $2i+1$  的结点为其右孩子结点。



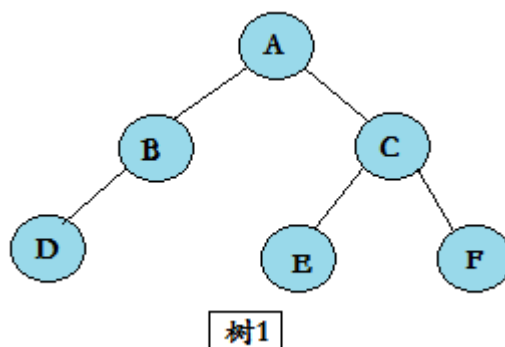
一颗完全二叉树

**性质 5:** 具有  $n$  个结点的完全二叉树的深度为  $\lceil \log_2 n \rceil + 1$  或者  $\lceil \log_2 (n+1) \rceil$ ，其中  $\lceil \log_2 n \rceil + 1$  是向下取整， $\lceil \log_2 (n+1) \rceil$  是向上取整。

**性质 6:** Catalan 函数性质：给定  $n$  个结点，能构成  $H(n)$  种结构不同的树。 $H(n) = c(2n, n) / (n+1)$ 。

## 六、二叉树的存储结构

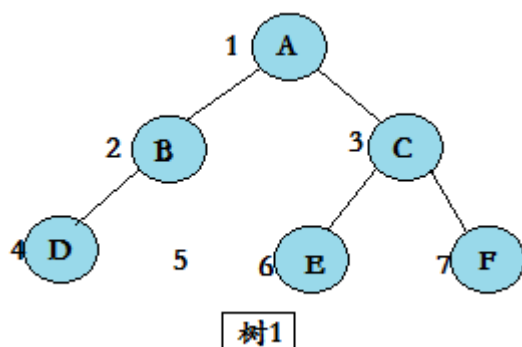
为了方便说明，我们使用下图树 1 作为案例树。



### 1、顺序存储

顺序存储是使用一个**数组**来存储二叉树，我们一般将二叉树按照性质 4 的做法，即从上到下且从左至右进行 1 至  $n$  的编号，然后编号与数组下标对应，按照编号依次将对应的结点信息存储数组中即可。

第一步：给二叉树编号：



注意，编号 5 的位置是没有结点的，但是我们这样编号的目的是为了更好的应用性质 4，而性质 4 描述的是完全二叉树，所以我们编号时要将普通二叉树看做完全二叉树来进行编号，所以即使编号 5 即使没有结点也需要进行编号。

第二步：按编号存储到数组 BTree[] 中

使用数组存储二叉树

结点数据:		A	B	C	D		E	F
角标/编号:	0	1	2	3	4	5	6	7

第三步：按照性质 4 的规律取元素

性质 4 主要是描述结点的编号和双亲编号或孩子编号之间的数学关系，即我们知道了一个结点的编号，那么它的双亲编号和孩子孩子我们都能计算得到并从数组中取出来。

例如，结合性质 4 我们来计算编号 3 的双亲和孩子：选取出编号 3 即 BTree[3]，就可以知道编号 3 的元素为 C；双亲结点编号 =  $3/2 = 1 \geq 1$ ，所以 C 结点的双亲为 BTree[1] 即 A；左孩子编号 =  $3*2 = 6 \leq 7$ ，所以 C 的左孩子为 BTree[6] = E；右孩子编号 =  $3*2+1 = 7 \leq 7$ ，所以 C 的右孩子为 Btree[7] = F。

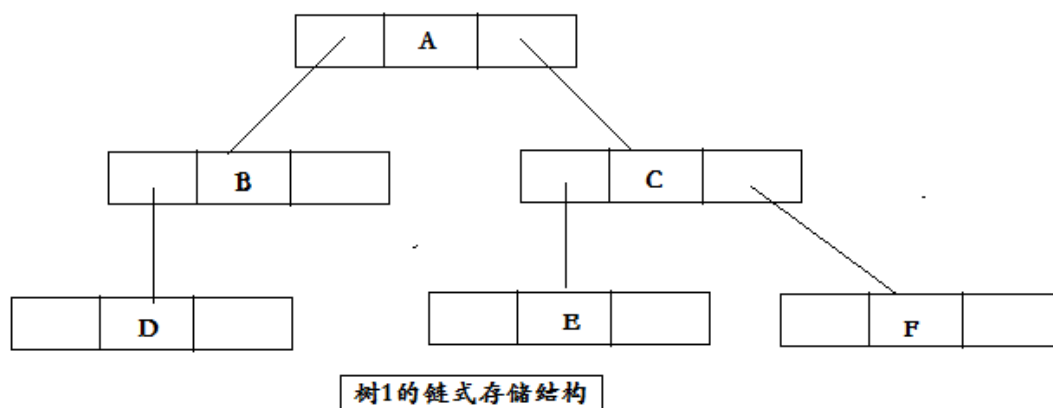
结论：像编号 5 这种情况会占用存储空间，所以这种存储方式最适合用于存储完全二叉树，而存储一般的二叉树则会浪费大量空间。

## 2、链式存储

根据二叉树的结构，我们使用下面的链式结点来存储一个二叉树结点。

lchild	data	rchild
左孩子	数据域	右孩子

所以树 1 对应的链式存储结构为：



## 七、二叉树的遍历算法

根据二叉树的结构特点：一般二叉树由左子树、根结点和右子树组成。这三个元素：左子树（L）、根结点（N）、右子树（R）有 6 种中排列组合，即 NLR、LNR、LRN、NRL、RNL、RLN。而从左往右和从右往左这种遍历顺序是对称结构的，采用一种顺序即可，所以二叉树按照三个元素的排列顺序遍历就形成了：NLR（先序遍历）、LNR（中序遍历）和 LRN（后序遍历）。

ps: 二叉树的这三种遍历要用递归的思想去理解。

**先序遍历（NLR）：根左右**

- 1) 访问根结点
- 2) 先序遍历左子树
- 3) 先序遍历右子树

**中序遍历（LNR）：左根右**

- 1) 中序遍历左子树
- 2) 访问根结点
- 3) 中序遍历右子树

**后序遍历（LRN）：左右根**

- 1) 后序遍历左子树
- 2) 后序遍历右子树
- 3) 访问根结点

**层次遍历**

层次遍历比较简单，即按照从上往下、从左往右一层一层遍历即可。层次遍历是现实，如果遍历的是顺序存储（数组存储）的二叉树，由于存储的时候就是按照从上往下从左往右的顺序存储的，直接按顺序取出即可；如果是链式存储的二叉树，需要使用一个循环队列进行操作：先将根结点入队，当前结点是队头结点，将其出队并访问，如果当前结点的左结点不为空将左结点入队，如果当前结点的右结点不为空将其入队。所以出队顺序也是从左到右依次出队。

## 八、二叉树的基本应用

### 1、二叉排序树（Binary Sort Tree）

二叉排序树，又称二叉查找树（Binary Search Tree），亦称二叉搜索树。

#### 1、定义

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：

- (1) 若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值；

(2) 若右子树不空, 则右子树上所有结点的值均大于或等于它的根结点的值;

(3) 左、右子树也分别为二叉排序树;

ps: 根据二叉排序树的定义, 如果对二叉排序树进行中序遍历, 那么遍历的结果就是一个递增的序列。

## 2、查找关键字

二叉排序树的主要功能就是查找。首先将需要查找的序列排序后存储到二叉排序树中, 那么要查找的关键字要么在左子树, 要么在根结点, 要么在右子树, 所以我们首先将要查找的关键字与根结点做比较, 相等则查找成功。小于根结点则递归查找左子树, 大于根结点则递归查找右子树, 直到出现相等情况则查找成功, 否则查找失败。(该查找过程与折半查找类似)

## 3、插入关键字

插入操作主要是对查找不成功的排序二叉树, 即如果关键字查找不成功, 那么我们就需要将查找不成功的关键字插入查找不成功的位置。所以我们只需要将查找算法进行修改就能实现插入操作

## 4、删除关键字

在删除关键字结点时, 需要注意的是, 在删除后我们需要保持二叉排序树的特性。

在二叉排序树删去一个结点, 分三种情况讨论: (设被删除结点为  $p$ ,  $p$  的双亲结点为  $f$ )

1. 若  $p$  结点为叶子结点, 即  $PL$ (左子树) 和  $PR$ (右子树) 均为空树。由于删去叶子结点不破坏整棵树的结构, 则可以直接删除此子结点。

2. 若  $p$  结点只有左子树  $PL$  或右子树  $PR$ , 此时只要令  $PL$  或  $PR$  直接成为其双亲结点  $f$  的左子树 (当  $p$  是左子树) 或右子树 (当  $p$  是右子树) 即可, 作此修改也不破坏二叉排序树的特性。

3. 若  $p$  结点的左子树和右子树均不空。这种情况可以转换为情况 1 或 2 然后再按照 1 或 2 的方法来解决。有两种转换方法:

其一是找到  $p$  结点的左子树的最右边的结点  $r$  (沿着  $p$  的左子树的根结点的右指针一直走, 其实就是找到左子树的最大值), 用  $r$  结点替换  $p$  结点, 然后再删除  $r$  结点即可。

其二是找到  $p$  结点的右子树的最左边的结点  $r$  (沿着  $p$  的右子树的根结点的左指针一直走, 其实就是找到右子树的最小值), 用  $r$  结点替换  $p$  结点, 然后再删除  $r$  结点即可。

## 2、平衡二叉树

平衡二叉树又称为 AVL 树, 是一种特殊的二叉排序树, 即左右两个子树高度之差不超过 1, 并且左右两个子树都是平衡二叉树的二叉排序树称为平衡二叉树。

为什么要构造平衡二叉树呢? 对于一般的二叉排序树, 其期望高度 (即为一棵平衡树时) 为  $\log_2 n$ , 其各操作的时间复杂度 ( $O(\log_2 n)$ ) 同时也由此而决定。由于 AVL 树的左右子树高度之差不超过 1, 其高度一般都良好地维持在  $O(\log(n))$ , 大大降低了操作的时间复杂度。

平衡二叉树的实现算法: 关键在于左右子树的平衡。具体的算法有: 红黑树、AVL 算法、Treap、伸展树、SBT 来实现。有兴趣的可以自行搜索, 这里就不再描

述。

### 3、赫夫曼树及赫夫曼编码

赫夫曼树又叫做最优二叉树，特点是带权路径长度最短。

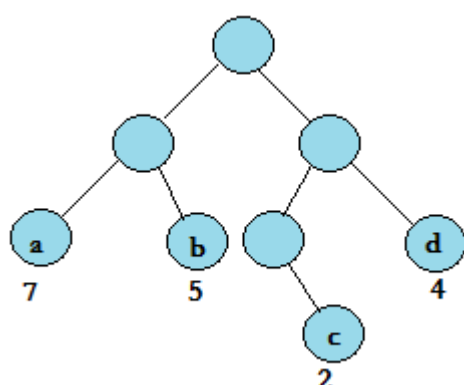
#### 1、相关术语

**路径和路径长度：**在一棵树中，从一个结点往下可以达到的孩子或孙子结点之间的通路，称为路径。通路中分支的数目称为**路径长度**。

**结点的权及带权路径长度：**若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。结点的带权路径长度为：从根结点到该结点之间的路径长度与该结点的权的乘积。

**树的带权路径长度：**树的带权路径长度规定为所有叶子结点的带权路径长度之和，记为WPL。

如下图有4个叶子结点的二叉树：



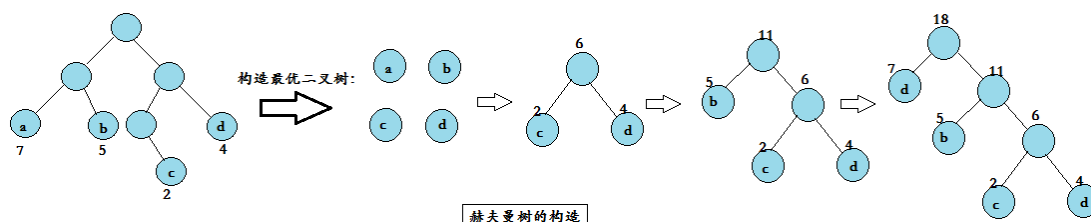
则这棵树的  $WPL = 2*7+2*5+3*2+2*4 = 38$ 。

#### 2、赫夫曼树的构造

假设有  $n$  个权值，则构造出的哈夫曼树有  $n$  个叶子结点。  $n$  个权值分别设为  $w_1, w_2, \dots, w_n$ ，则哈夫曼树的构造规则为：

- (1) 将  $w_1, w_2, \dots, w_n$  看成是有  $n$  棵树的森林(每棵树仅有一个结点)；
- (2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；
- (3) 从森林中删除选取的两棵树，并将新树加入森林；
- (4) 重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

以上面的二叉树为例：首先有4个叶子结点 a、b、c、d，权值分别为7、5、2、4。则：



普构造得到的赫夫曼树的带权路径长度是最小的， $WPL = 1*7+2*5+3*2+4*4 = 35$ 。

#### 3、赫夫曼树的特点

- 1) 权值越大的结点离根结点越近。

2) 树中没有度为 1 的结点。这类树又称为正则 (严格) 二叉树。

4、赫夫曼树的应用：赫夫曼编码

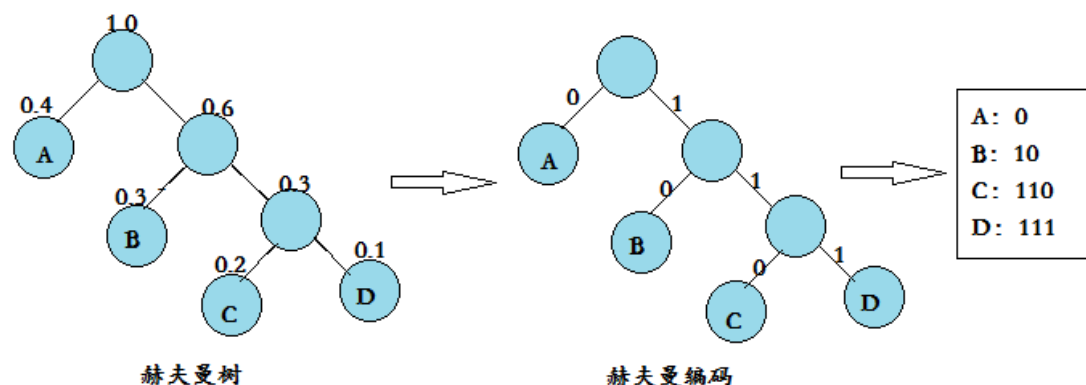
赫夫曼编码是一种编码方式。例如我们需要发送有 A、B、C、D 这 4 个字符组成的一些文字，如果分别用 00, 01, 10, 11 来代表要传送的 A、B、C、D。则需要发送“ADA”就编码为：001100。

我们希望编码的长度能够越短越好，上面的编码方式长度显然是与字符个数成正比的，不能满足需求，所以赫夫曼想到了设置不同的编码长度来表达不同字符，那么让出现概率越高的字符设置编码长度越短即可。假设 A、B、C、D 出现的概率分别为 0.4、0.3、0.2、0.1。那么 A、B、C、D 的编码就可以分别用 0、00、01、1 来表示。那么 ADA 的编码就是“010”，但“CA”的编码也是“010”，所以我们在设置不同长度的编码时需要使用前缀编码 (即任一编码都不是其他编码的前缀，这样就不会产生歧义了)。所以 A、B、C、D 的编码修改为：0、10、110、111 即可。

赫夫曼编码就是长度可变 (编码长度最短) 的前缀编码。其实，赫夫曼编码的构造就是赫夫曼树的构造过程：即字符相当于叶子结点；字符出现的概率相当于结点的权值；由于构造的赫夫曼树权值越大的结点离根结点越近，所以字符出现概率越大的字符离根结点越近，路径也就越短。

所以，A、B、C、D (0.4、0.3、0.2、0.1) 构造的赫夫曼树过程如下：

- 1) 构造赫夫曼树。
- 2) 约定左分支表示 0，右分支表示 1。
- 3) 从根结点到字符结点的路径组成的编码就是该字符的赫夫曼编码。



## 六、数据结构之各种树

### 一、平衡二叉树之 AVL 树

#### 1、定义

AVL 树是最先发明的自平衡二叉查找树。所以 AVL 树是一种二叉平衡树，即 AVL 树是二叉平衡树的一种实现。

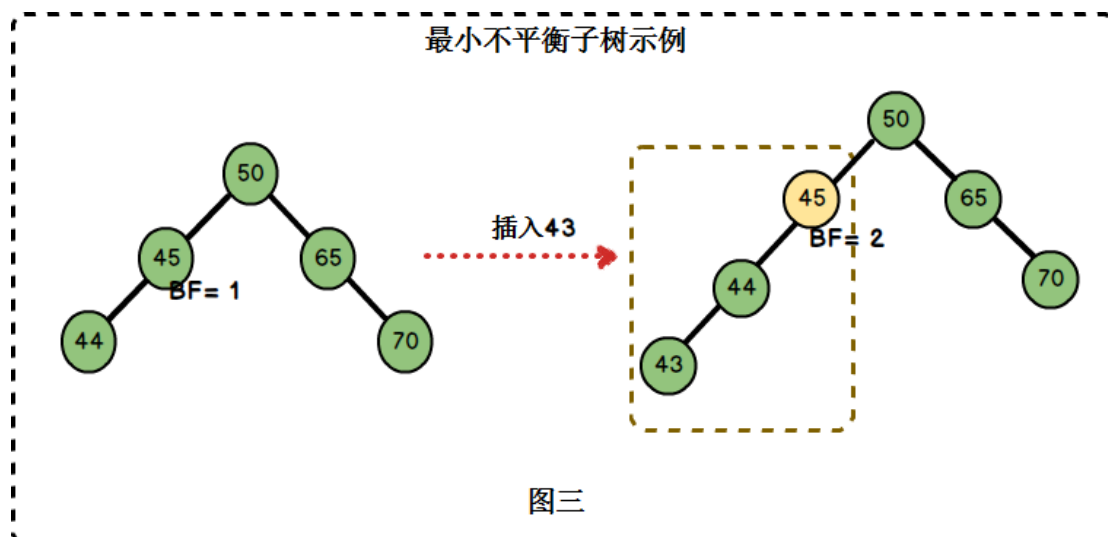
ps: AVL 树既然是二叉平衡树，那么必须满足：1) 是二次排序树；2) 左右子树相对高度差不大于 1。

#### 2、相关概念

**平衡因子**：将二叉树上结点的左子树高度减去右子树高度的值称为该结点的平衡因子 BF (Balance Factor)。

**最小不平衡子树**：距离插入结点最近的，且以平衡因子绝对值大于 1 的结点为根的子树。





在图三中，左边二叉树的结点 45 的  $BF = 1$ ，插入结点 43 后，结点 45 的  $BF = 2$ 。结点 45 是距离插入点 43 最近的  $BF$  不在  $[-1, 1]$  范围内的结点，因此以结点 45 为根的子树为最小不平衡子树。

#### 4、AVL 树的操作

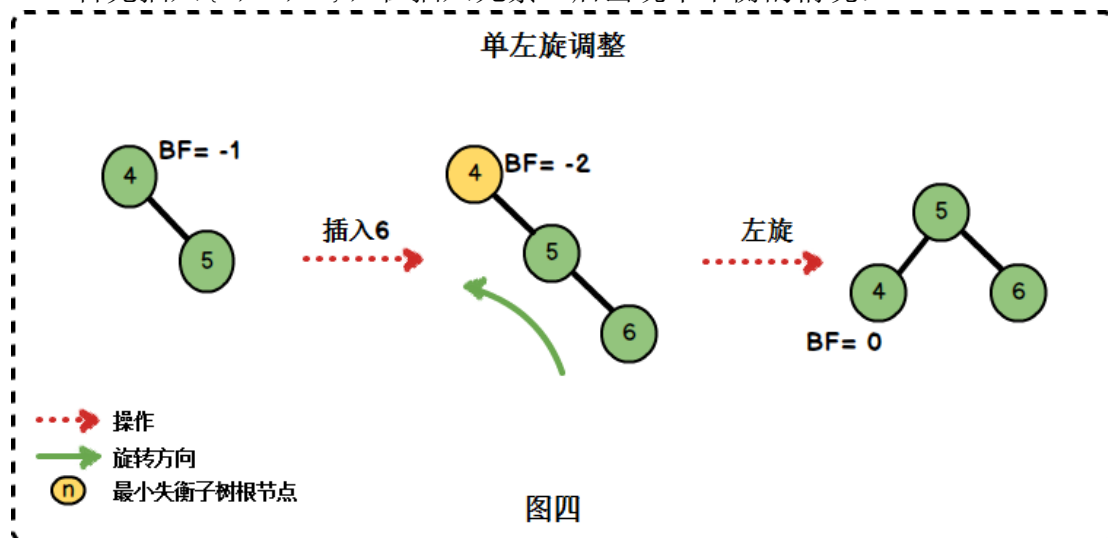
AVL 树具有一般二叉树的查找、插入和删除等基本操作，但 AVL 树作为自平衡二叉查找树，需要保证其平衡因子不大于 1，而插入或删除操作都可能会打破树的平衡，所以 AVL 树的特殊操作是进行旋转操作，以此来调整树的不平衡子树。

##### 1) 旋转操作

为了方便说明，假设我们要为数组  $a[] = \{4, 5, 6, 3, 2, 8, 7, 0, 1\}$  构建一棵 AVL 树。

##### 情况一：左单旋转

首先插入  $\{4, 5, 6\}$ ，在插入元素 6 后出现不平衡的情况：



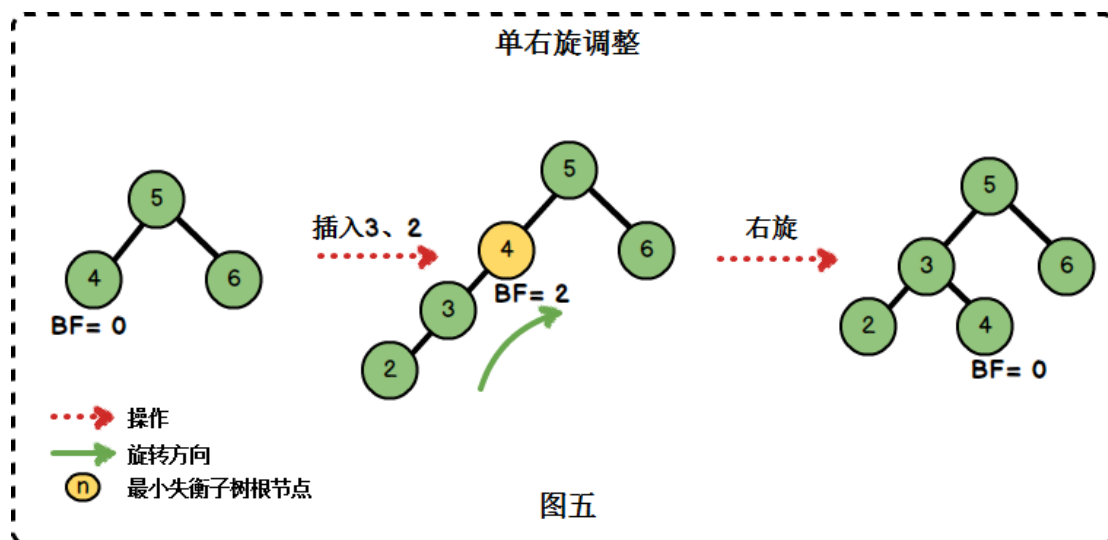
当我们在右子树插入右孩子导致 AVL 失衡时，我们需要进行单左旋调整。旋转围绕最小失衡子树的根节点进行。

在删除新节点时也有可能会出现需要单左旋的情况。

##### 情况二：右单旋转

我们继续插入元素  $\{3, 2\}$ ，此时二叉树为：



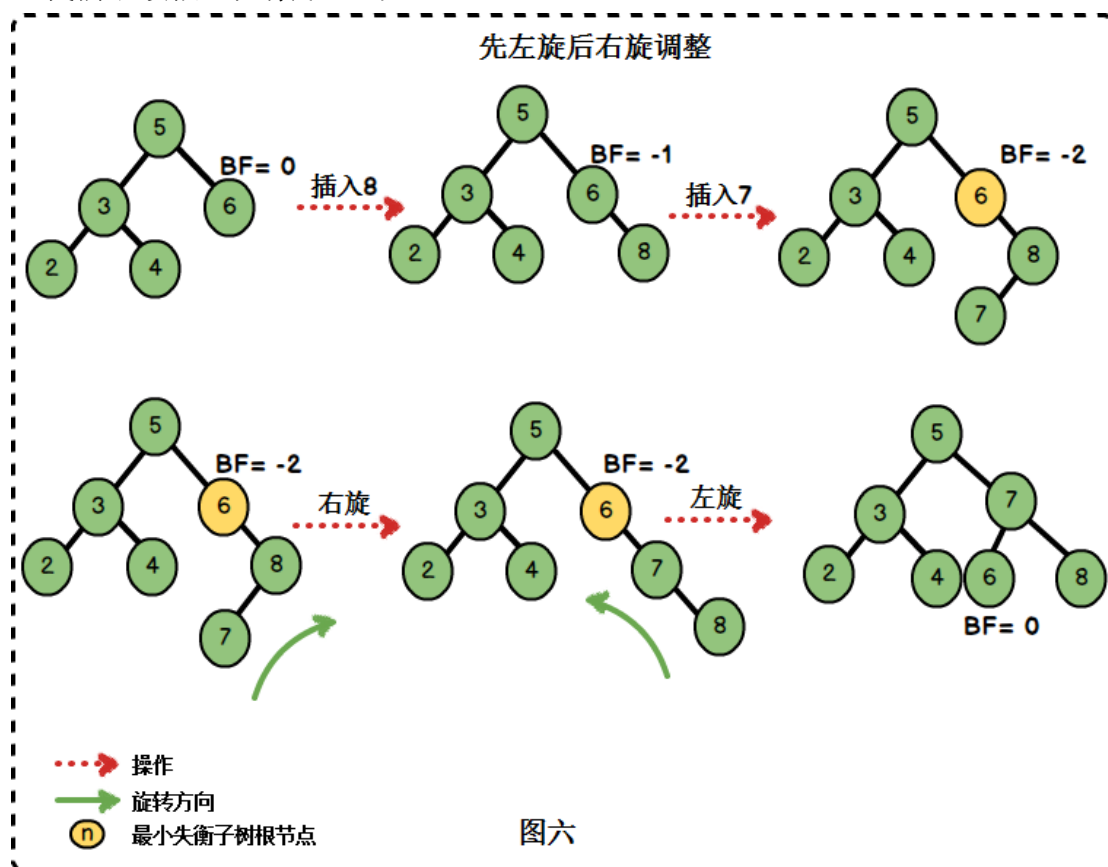


此时的插入情况是“在左子树上插入左孩子导致 AVL 树失衡”，我们需要进行单右旋调整。

### 情况三：先左旋后右旋

需要进行两次旋转的原因是第一次旋转后，AVL 树仍旧处于不平衡的状态，第二次旋转再次进行调整。

我们继续插入元素 {8, 7}：

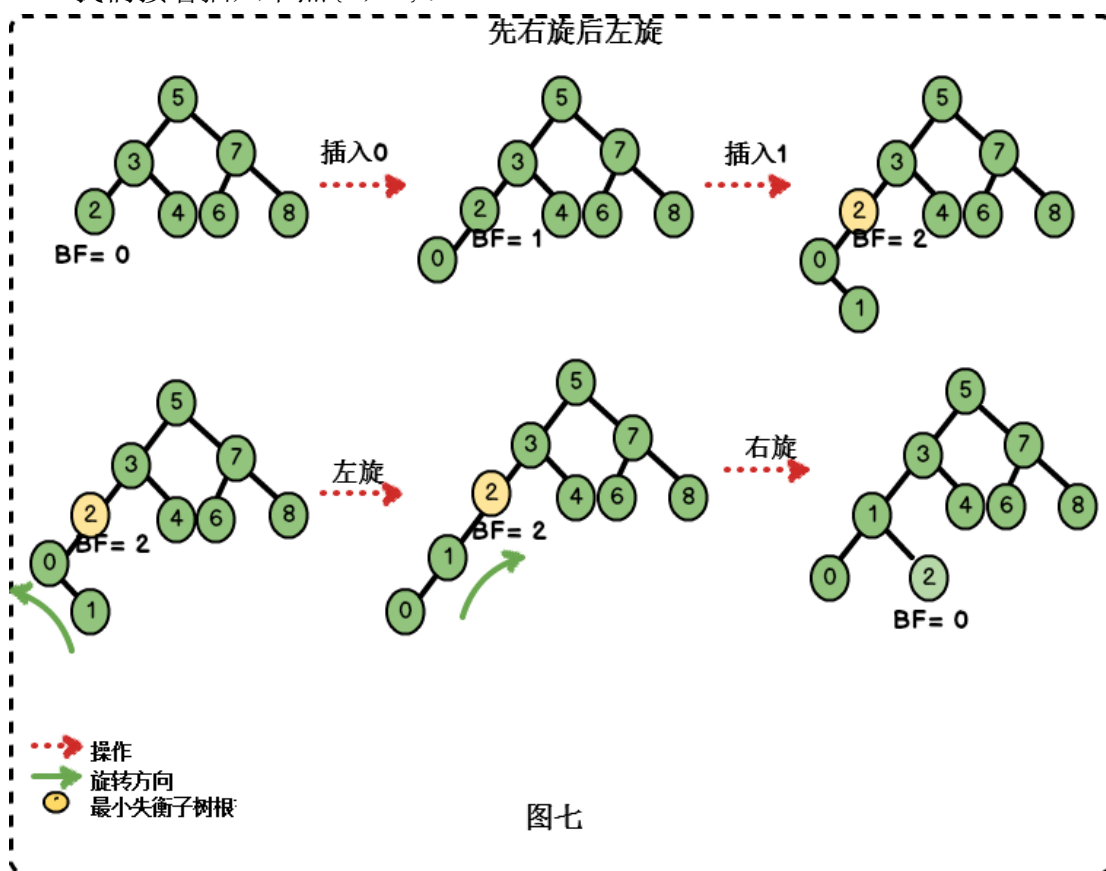


这种情况，总结起来就是“在右子树上插入左孩子导致 AVL 树失衡”，此时我们需要进行先右旋后左旋的调整。

#### 情况四：先左旋后右旋

根据对称性原理，当我们“在左子树上插入右孩子导致 AVL 树失衡”，此时我们需要进行先左旋后右旋的调整。如果你不理解接着看图。

我们接着插入节点{0, 1}：



总结：四种失衡调整

类型	使用情形
单左旋	在左子树插入左孩子节点，使得平衡因子绝对值由1增至2
单右旋	在右子树插入右孩子节点，使得平衡因子绝对值由1增至2
先左旋后右旋	在左子树插入右孩子节点，使得平衡因子绝对值由1增至2
先右旋后左旋	在右子树插入左孩子节点，使得平衡因子绝对值由1增至2

#### 2) 其他操作

其他操作如插入、删除、查找、遍历与一般二叉树的操作类似，不同之处在于如果插入或删除操作导致树失衡，需要进行旋转操作进行恢复平衡。

## JAVA 中的算法总结

author: 皓宇 QAQ

### 一、选择排序算法

1. 原理：每一趟从待排序的记录中选出最小的元素，顺序放在已排好序的序列最后，直到全部记录排序完毕。也就是：每一趟在  $n-i+1$  ( $i=1, 2, \dots, n-1$ ) 个记录中选取关键字最小的记录作为有序序列中第  $i$  个记录。基于此思想的算法主要有简单选择排序、树型选择排序和堆排序。

2. 简单选择排序的基本思想：给定数组：int[] arr={里面 n 个数据}；第 1 趟排序，在待排序数据 arr[1]~arr[n] 中选出最小的数据，将它与 arr[1] 交换；第 2 趟，在待排序数据 arr[2]~arr[n] 中选出最小的数据，将它与 arr[2] 交换；以此类推，第  $i$  趟在待排序数据 arr[i]~arr[n] 中选出最小的数据，将它与 arr[i] 交换，直到全部排序完成。

3. 举例：数组 int[] arr={5, 2, 8, 4, 9, 1};

---

第一趟排序： 原始数据： 5      2      8      4      9      1  
最小数据 1，把 1 放在首位，也就是 1 和 5 互换位置，  
排序结果： 1      2      8      4      9      5

---

第二趟排序：  
第 1 以外的数据 {2      8      4      9      5} 进行比较，2 最小，  
排序结果： 1      2      8      4      9      5

---

第三趟排序：  
除 1、2 以外的数据 {8      4      9      5} 进行比较，4 最小，8 和 4 交换  
排序结果： 1      2      4      8      9      5

---

第四趟排序：  
除第 1、2、4 以外的其他数据 {8      9      5} 进行比较，5 最小，8 和 5 交换  
排序结果： 1      2      4      5      9      8

---

第五趟排序：  
除第 1、2、4、5 以外的其他数据 {9      8} 进行比较，8 最小，8 和 9 交换  
排序结果： 1      2      4      5      8      9

---

注：每一趟排序获得最小数的方法：for 循环进行比较，定义一个第三个变量 temp，首先前两个数比较，把较小的数放在 temp 中，然后用 temp 再去跟剩下的数据比较，如果出现比 temp 小的数据，就用它代替 temp 中原有的数据。

Java 代码实现：

---

```
package test;

/* 选择排序算法实现代码
 * author 皓宇 QAQ
 */

public class StraightSelectionSort {
    public static void main(String[] args) {
        int[] arr={1,3,2,45,65,33,12};
        System.out.println("交换之前：");
```

```

    for(int num:arr){
        System.out.print(num+" ");
    }
    //选择排序的优化
    for(int i = 0; i < arr.length - 1; i++) { // 做第 i 趟排序
        int min = i;
        for(int j = i + 1; j < arr.length; j++){ // 选最小的记录
            if(arr[j] < arr[min]){
                min = j; //记下目前找到的最小值所在的位置
            }
        }
        //在内层循环结束，也就是找到本轮循环的最小的数以后，再进行交换
        if(i != min){ //交换 a[i] 和 a[min]
            int temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
        }
    }
    System.out.println();
    System.out.println("交换后: ");
    for(int num:arr){
        System.out.print(num+" ");
    }
}
}

```

运行结果：

```

<terminated> StraightSelectionSort (1) [Java Application] G:\MyEclipse Professional\binary\
交换之前：
1 3 2 45 65 33 12
交换后：
1 2 3 12 33 45 65 |

```

选择排序的时间复杂度：简单选择排序的比较次数与序列的初始排序无关。 假设待排序的序列有  $N$  个元素，则比较次数永远都是  $N(N-1)/2$ 。而移动次数与序列的初始排序有关。当序列正序时，移动次数最少，为 0。当序列反序时，移动次数最多，为  $3N(N-1)/2$ 。所以简单排序的时间复杂度为  $O(N^2)$ 。

## 二、冒泡排序算法

**原理：**比较两个相邻的元素，将值大的元素交换至右端。

**思路：**依次比较相邻的两个数，将小数放在前面，大数放在后面。即在第一趟：首先比较第 1 个和第 2 个数，将小数放前，大数放后。然后比较第 2 个数和第 3 个数，将小数放前，大数放后，如此继续，直至比较最后两个数，将小数放前，大数放后。重复第一趟步骤，直至全部排序完成。

第一趟比较完成后，最后一个数一定是数组中最大的一个数，所以第二趟比较的时候最后一个数不参与比较；

第二趟比较完成后，倒数第二个数也一定是数组中第二大的数，所以第三趟比较的时候最后两个数不参与比较；

依次类推，每一趟比较次数-1...

举例说明：要排序数组：int[] arr={6,3,8,2,9,1};

第一趟排序：

第一次排序：6 和 3 比较，6 大于 3，交换位置： 3 6 8 2 9 1  
第二次排序：6 和 8 比较，6 小于 8，不交换位置： 3 6 8 2 9 1  
第三次排序：8 和 2 比较，8 大于 2，交换位置： 3 6 2 8 9 1  
第四次排序：8 和 9 比较，8 小于 9，不交换位置： 3 6 2 8 9 1  
第五次排序：9 和 1 比较：9 大于 1，交换位置： 3 6 2 8 1 9  
第一趟总共进行了 5 次比较， 排序结果： 3 6 2 8 1 9

-----

第二趟排序：

第一次排序：3 和 6 比较，3 小于 6，不交换位置： 3 6 2 8 1 9  
第二次排序：6 和 2 比较，6 大于 2，交换位置： 3 2 6 8 1 9  
第三次排序：6 和 8 比较，6 大于 8，不交换位置： 3 2 6 8 1 9  
第四次排序：8 和 1 比较，8 大于 1，交换位置： 3 2 6 1 8 9  
第二趟总共进行了 4 次比较， 排序结果： 3 2 6 1 8 9

-----

第三趟排序：

第一次排序：3 和 2 比较，3 大于 2，交换位置： 2 3 6 1 8 9  
第二次排序：3 和 6 比较，3 小于 6，不交换位置： 2 3 6 1 8 9  
第三次排序：6 和 1 比较，6 大于 1，交换位置： 2 3 1 6 8 9  
第二趟总共进行了 3 次比较， 排序结果： 2 3 1 6 8 9

-----

第四趟排序：

第一次排序：2 和 3 比较，2 小于 3，不交换位置： 2 3 1 6 8 9  
第二次排序：3 和 1 比较，3 大于 1，交换位置： 2 1 3 6 8 9  
第二趟总共进行了 2 次比较， 排序结果： 2 1 3 6 8 9

-----

第五趟排序：

第一次排序：2 和 1 比较，2 大于 1，交换位置： 1 2 3 6 8 9  
第二趟总共进行了 1 次比较， 排序结果： 1 2 3 6 8 9

-----

最终结果：1 2 3 6 8 9

-----

由此可见：N 个数字要排序完成，总共进行 N-1 趟排序，每 i 趟的排序次数为(N-i)

次，所以可以用双重循环语句，外层控制循环多少趟，内层控制每一趟的循环次数，即

```
-----  
for(int i=1;i<arr.length;i++){  
  
    for(int j=1;j<arr.length-i;j++){  
  
        //交换位置  
  
    }  
}-----
```

**冒泡排序的优点：**每进行一趟排序，就会少比较一次，因为每进行一趟排序都会找出一个较大值。如上例：第一趟比较之后，排在最后的一个数一定是最大的一个数，第二趟排序的时候，只需要比较除了最后一个数以外的其他的数，同样也能找出一个最大的数排在参与第二趟比较的数后面，第三趟比较的时候，只需要比较除了最后两个数以外的其他的数，以此类推.....也就是说，没进行一趟比较，每一趟少比较一次，一定程度上减少了算法的量。

用时间复杂度来说：

1.如果我们的数据正序，只需要走一趟即可完成排序。所需的比较次数  $C$  和记录移动次数  $M$  均达到最小值，即： $C_{min}=n-1;M_{min}=0$ ；所以，冒泡排序最好的时间复杂度为  $O(n)$ 。

2.如果很不幸我们的数据是反序的，则需要进行  $n-1$  趟排序。每趟排序要进行  $n-i$  次比较( $1 \leq i \leq n-1$ )，且每次比较都必须移动记录三次来达到交换记录位置。在这种情况下，比较和移动次数均达到最大值：

$$C_{max} = \frac{n(n-1)}{2} = O(n^2)$$
$$M_{max} = \frac{3n(n-1)}{2} = O(n^2)$$

冒泡排序的最坏时间复杂度为： $O(n^2)$ 。

综上所述：**冒泡排序总的平均时间复杂度为： $O(n^2)$ 。**

**Java 代码实现：**

```
-----  
package test;  
  
/* 冒泡排序算法实现代码  
 * author: 皓宇 QAQ  
 */  
  
public class BubbleSort{  
    public static void main(String[] args) {  
        int arr[] = {12,6,45,3,1,9};  
        System.out.println("排序前数组为:");  
        for(int num:arr){  
            System.out.print(num+" ");  
        }  
    }  
}
```

```

    }
    for(int i=0;i<arr.length-1;i++){//外层循环控制排序趟数
        for(int j=0;j<arr.length-1-i;j++){//内层循环控制每一趟排
序多少次
            if(arr[j]>arr[j+1]){
                int temp= arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    System.out.println();
    System.out.println("排序后数组为: ");
    for(int num:arr){
        System.out.print(num+" ");
    }
}
}

```

---

运行结果：

```

<terminated> BubbleSort [Java Application] G:\MyEclipse Professional\binary\com.sun.java.jdk.v
排序前数组为：
12 6 45 3 1 9
排序后数组为：
1 3 6 9 12 45

```

### 三、直接插入排序算法

基本思想：

把  $n$  个待排序的元素看成一个有序表和一个无序表，开始时有序表中只有一个元素，无序表中有  $n-1$  个元素；排序过程即每次从无序表中取出第一个元素，将它插入到有序表中，使之成为新的有序表，重复  $n-1$  次完成整个排序过程。

实例：

0. 初始状态 3, 1, 5, 7, 2, 4, 9, 6 (共 8 个数)

有序表：3；无序表：1, 5, 7, 2, 4, 9, 6

1. 第一次循环，从无序表中取出第一个数 1，把它插入到有序表中，使新的数列依旧有序

有序表：1, 3；无序表：5, 7, 2, 4, 9, 6

2. 第二次循环，从无序表中取出第一个数 5，把它插入到有序表中，使新的数列依旧有序

有序表：1, 3, 5; 无序表：7, 2, 4, 9, 6

3. 第三次循环，从无序表中取出第一个数 7，把它插入到有序表中，使新的数列依旧有序

有序表：1, 3, 5, 7; 无序表：2, 4, 9, 6

4. 第四次循环，从无序表中取出第一个数 2，把它插入到有序表中，使新的数列依旧有序

有序表：1, 2, 3, 5, 7; 无序表：4, 9, 6

5. 第五次循环，从无序表中取出第一个数 4，把它插入到有序表中，使新的数列依旧有序

有序表：1, 2, 3, 4, 5, 7; 无序表：9, 6

6. 第六次循环，从无序表中取出第一个数 9，把它插入到有序表中，使新的数列依旧有序

有序表：1, 2, 3, 4, 5, 7, 9; 无序表：6

7. 第七次循环，从无序表中取出第一个数 6，把它插入到有序表中，使新的数列依旧有序

有序表：1, 2, 3, 4, 5, 6, 7, 9; 无序表：(空)

**Java 代码实现：**

```
package test;

/**
 * 直接插入排序算法实现
 * @author 皓宇 QAQ
 */
public class InsertSort {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int a[] = {45,64,15,2,31,16,9,7};
        new InsertSort().insertSort(a);
    }

    private void insertSort(int[] a) {
        // TODO Auto-generated method stub
        System.out.println("—————直接插入排序算法—————");
        int n = a.length;
        int i, j;
        for (i = 1; i < n; i++) {
            //temp 为本次循环待插入有序列表中的数
            int temp = a[i];
            // 寻找 temp 插入有序列表的正确位置
            for (j = i - 1; j >= 0 && a[j] > temp; j--) {
                //元素后移，为插入 temp 做准备
                a[j + 1] = a[j];
            }
        }
    }
}
```



```

        //插入 temp
        a[j+1] = temp;
        print(a,n,i);
    }
    printResult(a,n);
}

/**
 * 打印排序的最终结果
 * @param a
 * @param n
 */
private void printResult(int[] a, int n){
    System.out.print("最终排序结果: ");
    for(int j=0;j<n;j++){
        System.out.print(" "+a[j]);
    }
    System.out.println();
}

/**
 * 打印排序的每次循环的结果
 * @param a
 * @param n
 * @param i
 */
private void print(int[] a, int n, int i) {
    // TODO Auto-generated method stub
    System.out.print("第"+i+"次: ");
    for(int j=0;j<n;j++){
        System.out.print(" "+a[j]);
    }
    System.out.println();
}
}

```

---

运行结果:

```
Servers Problems Tasks Web Browser Project Migration Debug C
<terminated> InsertSort [Java Application] G:\MyEclipse Professional\binary\com.sun.ja
直接插入排序算法
第1次: 45 64 15 2 31 16 9 7
第2次: 15 45 64 2 31 16 9 7
第3次: 2 15 45 64 31 16 9 7
第4次: 2 15 31 45 64 16 9 7
第5次: 2 15 16 31 45 64 9 7
第6次: 2 9 15 16 31 45 64 7
第7次: 2 7 9 15 16 31 45 64
最终排序结果: 2 7 9 15 16 31 45 64
```

## 直接插入排序算法的优化：折半查找/二分查找

基本思想：

折半插入算法是对直接插入排序算法的改进，排序原理同直接插入算法：

把  $n$  个待排序的元素看成一个有序表和一个无序表，开始时有序表中只有一个元素，无序表中有  $n-1$  个元素；排序过程即每次从无序表中取出第一个元素，将它插入到有序表中，使之成为新的有序表，重复  $n-1$  次完成整个排序过程。

与直接插入算法的区别在于：在有序表中寻找待排序数据的正确位置时，使用了折半查找/二分查找。

Java 代码实现：

```
package test;

/**
 * 折半插入排序算法实现
 * @author 皓宇 QAQ
 */
public class InsertTwoSort {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int a[] = {64,45,15,2,31,16,9,7};
        new InsertTwoSort().binaryInsertSort(a);
    }

    private void binaryInsertSort(int[] a) {
        // TODO Auto-generated method stub
        System.out.println("-----折半
插入排序算法-----");
        int n = a.length;
        int i,j;
        for (i=1;i<n;i++){
            //temp 为本次循环待插入有序列表中的数
            int temp = a[i];
            int low=0;
```

```

        int high=i-1;
        //寻找 temp 插入有序列表的正确位置，使用二分查找法
        while(low <= high){
            //有序数组的中间坐标，此时用于二分查找，减少查找次数
            int mid = (low+high)/2;
            //若有序数组的中间元素大于待排序元素，则有序序列向中间元素之前搜索，否则向后搜索
            if(a[mid]>temp){
                high = mid-1;
            }else{
                low = mid+1;
            }
        }

        for(j=i-1;j>=low;j--){
            //元素后移，为插入 temp 做准备
            a[j+1] = a[j];
        }
        //插入 temp
        a[low] = temp;
        //打印每次循环的结果
        print(a,n,i);
    }

    //打印排序结果
    printResult(a,n);
}

/**
 * 打印排序的最终结果
 * @param a
 * @param n
 */
private void printResult(int[] a, int n){
    System.out.print("最终排序结果: ");
    for(int j=0;j<n;j++){
        System.out.print(" "+a[j]);
    }
    System.out.println();
}

/**
 * 打印排序的每次循环的结果
 * @param a
 * @param n
 * @param i
 */

```

```

private void print(int[] a, int n, int i) {
    // TODO Auto-generated method stub
    System.out.print("第"+i+"次: ");
    for(int j=0;j<n;j++){
        System.out.print(" "+a[j]);
    }
    System.out.println();
}
}

```

运行结果:

```

<terminated> InsertTwoSort [Java Application] G:\MyEclipse Professional\binary\com.sun.j
折半插入排序算法
第1次: 45 64 15 2 31 16 9 7
第2次: 15 45 64 2 31 16 9 7
第3次: 2 15 45 64 31 16 9 7
第4次: 2 15 31 45 64 16 9 7
第5次: 2 15 16 31 45 64 9 7
第6次: 2 9 15 16 31 45 64 7
第7次: 2 7 9 15 16 31 45 64
最终排序结果: 2 7 9 15 16 31 45 64

```

## 四、快速排序算法

**基本思想:**

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

1. 设置 low=0, high=N-1。
2. 选择一个基准元素赋值给 temp, 即 temp=a[low]。
3. 从 high 开始向前搜索，即由后开始向前搜索 (high--), 找到第一个小于 temp 的值，将 a[high] 和 a[low] 交换。
4. 从 low 开始向前后搜索，即由前开始向后搜索 (low++), 找到第一个大于 temp 的值，将 a[high] 和 a[low] 交换。
5. 重复第 3 步和第 4 步，直到 low==high, 3, 4 步中，若没找到符合条件的值，执行 high-- 或 low++，直到找到为止。进行交换时 low 和 high 的位置不变。当 low==high 时循环结束。

基准点的选取：固定切分、随机切分、三取样切分。

快速排序是不稳定的

快速排序在序列元素很少时，效率比较低。因此，元素较少时，可选择使用插入排序。

**Java 实现代码:**

```

-----

package test;

import java.util.Arrays;

/**
 * 快速排序算法实现
 * @author 皓宇 QAQ
 */
public class QuickSort {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        new QuickSort().run();
    }

    public void run() {

        int a[] = {64,45,15,2,31,16,9,7};
        int n = a.length;
        System.out.println("-----快速
排序算法-----");
        System.out.println("排序前:");
        System.out.println(Arrays.toString(a));
        if(n>0){
            quickSort(a, 0, n - 1);
        }
        System.out.println("排序后:");
        for(int i=0;i<a.length;i++){

            System.out.print(a[i]+" ");
        }
        System.out.println();
    }

    /**
     * 快速排序
     * 快速排序    时间复杂度为  $O(N\log N)$  .
     */
    private void quickSort(int[] a,int low,int high){
        if (low < high) {
            /**
             * 将数组 a 一分为二
             */
            int middle = getMiddle(a, low, high);
            /**

```

```

        * 将小于基准元素的数据进行递归排序
        */
        quickSort(a, low, middle - 1);
    /**
        * 将大于基准元素的数据进行递归排序
        */
        quickSort(a, middle + 1, high);
    }
}

public int getMiddle(int[] list, int low, int high) {
    /**
        * 数组的第一个数为基准元素
        */
        int temp = list[low];
        while (low < high) {
            while (low < high && list[high] > temp) {
                high--;
            }
            /**
                * 比基准小的数据移到低端
                */
            list[low] = list[high];
            while (low < high && list[low] < temp) {
                low++;
            }
            /**
                * 比基准大的记录移到高端
                */
            list[high] = list[low];
        }
        /**
            * 此时 low == high
            */
        list[low] = temp;
        return low;
    }
}

```

---

```

Servers Problems Tasks Web Browser Project Migration Debug DB Bro
<terminated> QuickSort [Java Application] G:\MyEclipse Professional\binary\com.sun.java.jdk
快速排序算法
排序前：
[64, 45, 15, 2, 31, 16, 9, 7]
排序后：
2 7 9 15 16 31 45 64

```

## 五、希尔排序算法

### 基本思想：

希尔排序的实质就是分组插入排序，又称缩小增量法。

将整个无序序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序时，再对全体元素进行一次直接插入排序。

因为直接插入排序在元素基本有序的情况下，效率是很高的，因此希尔排序在时间效率上有很大提高。

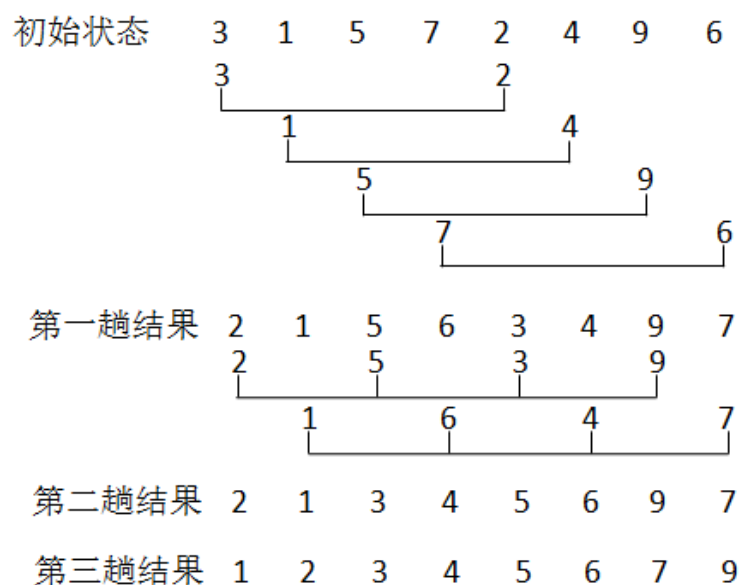
### 实例：

无序序列：int a[] = {3, 1, 5, 7, 2, 4, 9, 6};

第一趟时：n=8; gap=n/2=4; 把整个序列共分成了 4 个子序列 {3, 2}、{1, 4}、{5, 9}、{7, 6}

第二趟时：gap=gap/2=2; 把整个序列共分成了 2 个子序列 {2, 5, 3, 9}、{1, 6, 4, 7}

第三趟时：对整个序列进行直接插入排序



希尔排序是不稳定的

### Java 实现代码：

```
package test;
```

```

import java.util.Arrays;
/**
 * 希尔排序算法实现
 * @author 皓宇 QAQ
 */
public class ShellSort {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        new ShellSort().run();
    }

    private void run() {
        // TODO Auto-generated method stub
        int a[] = {64,45,15,2,31,16,9,7};
        System.out.println("—————希尔排序算法—————");
        System.out.println("排序前:");
        System.out.println(Arrays.toString(a));
        //shellSort(a);
        System.out.println("排序规律动态变化:");
        shellSort2(a);
        printResult(a,a.length);
        System.out.println("排序后:");
        for(int i=0;i<a.length;i++){

            System.out.print(a[i]+" ");
        }
    }
    /**
     * 希尔排序 (缩小增量法) 属于插入类排序
     * 不稳定
     * @param a
     */
    private void shellSort(int[] a){
        int n=a.length;
        int gap=n/2;
        while(gap>=1){
            for(int i=gap;i<a.length;i++){
                int j=0;
                int temp = a[i];
                for(j=i-gap;j>=0 && temp<a[j];j=j-gap){
                    a[j+gap] = a[j];
                }
            }
        }
    }
}

```



```

        a[j+gap] = temp;
    }
    printResult(a,a.length);
    gap = gap/2;
}
}
/**
 * 严格按照定义来写的希尔排序
 * @param a
 */
private void shellSort2(int[] a){
    int n=a.length;
    int i,j,k,gap;
    for(gap=n/2;gap>0;gap/=2){
        for(i=0;i<gap;i++){
            for(j=i+gap;j<n;j+=gap){
                int temp = a[j];
                for(k=j-gap;k>=0 && a[k]>temp;k-=gap){
                    a[k+gap]=a[k];
                }
                a[k+gap]=temp;
            }
        }
        printResult(a,a.length);
    }
}

private void printResult(int[] a, int n){
    for(int j=0;j<n;j++){
        System.out.print(" "+a[j]);
    }
    System.out.println();
}
}

```

---

运行结果:

```
Servers Problems Tasks Web Browser Project Migration Debug DB Browser
<terminated> ShellSort [Java Application] G:\MyEclipse Professional\binary\com.sun.java.jdk.win32.
希尔排序算法

排序前：
[64, 45, 15, 2, 31, 16, 9, 7]
排序规律动态变化：
31 16 9 2 64 45 15 7
9 2 15 7 31 16 64 45
2 7 9 15 16 31 45 64
2 7 9 15 16 31 45 64
排序后：
2 7 9 15 16 31 45 64
<
```

## 六、归并排序算法

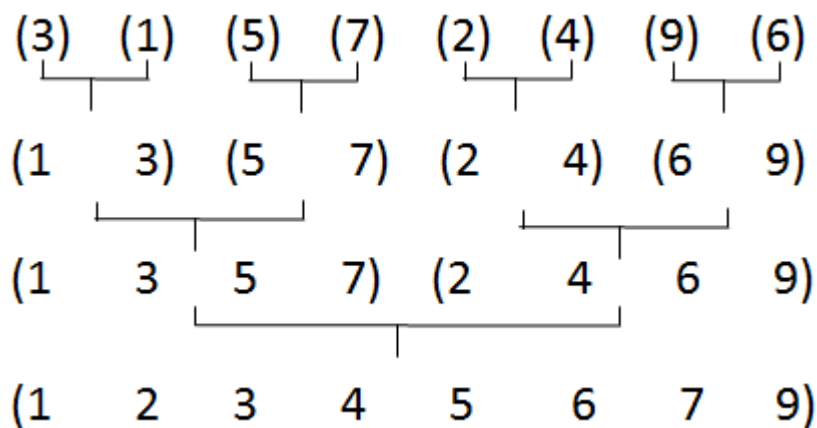
### 基本思想：

归并排序法是分治法的典型实例，分为分割和归并两部分。

把一个数组分为大小相近的子数组（分割），分别把子数组排好序后，通过合成一个大的排好序的数组（归并）。

### 实例：

先分割成每个子序列只有一个元素，然后再两两归并。



归并排序是稳定的排序算法，时间复杂度为： $O(N\log N)$ 。

### Java 实现代码：

```
package test;

import java.util.Arrays;
/**
 * 归并排序算法实现
 * @author 皓宇 QAQ
 */
public class MergeSort {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

```

        new MergeSort().run();
    }
    public void run() {

        int a[] = {64,45,15,2,31,16,9,7};
        System.out.println("-----归并
排序算法-----");
        System.out.println("排序前:");
        System.out.println(Arrays.toString(a));
        System.out.println("排序后:");
        int len = a.length;
        sort(a,0,len-1);
        for(int i=0;i<len;i++){
            System.out.print(a[i]+" ");
        }
    }
    /**
     * 归并排序 稳定的
     * 基本思想
     * 将两个或两个以上有序表合并成一个新的有序表
     * 即把待排序序列分成若干个子序列，每个子序列是有序的，然后在把有序子序
     列合并为整体有序序列
     */
    public void sort(int arr[],int low,int high){
        if(low == high){
            return;
        }
        int mid = (low+high)/2;
        if(low<high){
            /**
             * 对左边排序
             */
            sort(arr,low,mid);
            /**
             * 对右边排序
             */
            sort(arr,mid+1,high);
            /**
             * 左右归并
             */
            merge(arr,low,mid,high);
        }
    }
    public void merge(int[] arr,int low,int mid,int high){

```

```

int [] temp = new int[high-low+1];
/**
 * 左指针
 */
int i = low;
/**
 * 右指针
 */
int j = mid+1;
int k=0;
/**
 * 先把较小的数移到新数组中
 */
while(i<=mid && j<=high){
    if(arr[i]<arr[j]){
        temp[k++] = arr[i++];
    }else{
        temp[k++] = arr[j++];
    }
}
/**
 * 把左边剩余的数移入新数组
 */
while(i<=mid){
    temp[k++] = arr[i++];
}
/**
 * 把右边剩余的数移入新数组
 */
while(j<=high){
    temp[k++] = arr[j++];
}
/**
 * 用新数组中的数覆盖 arr 数组
 */
for(int k2=0;k2<temp.length;k2++){
    arr[k2+low] = temp[k2];
}
}
}

```

---

运行结果:

```

Servers Problems Tasks Web Browser Project Migration Debug DB Bro
<terminated> MergeSort [Java Application] G:\MyEclipse Professional\binary\com.sun.java.j
-----归并排序算法-----
排序前：
[64, 45, 15, 2, 31, 16, 9, 7]
排序后：
2 7 9 15 16 31 45 64

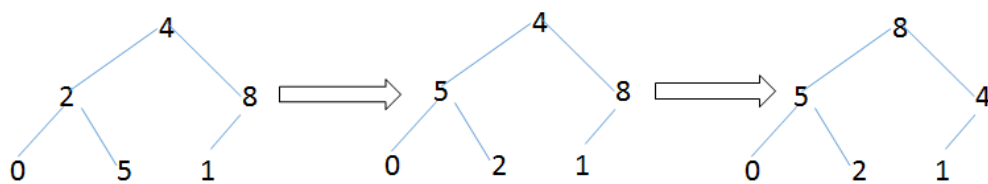
```

## 七、堆排序算法

堆的定义如下：n 个元素的序列 {k<sub>1</sub>, k<sub>2</sub>, ..., k<sub>n</sub>} 当且仅当满足一下条件时，称之为堆。

$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad or \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad (i = 1, 2, \dots, \lfloor n/2 \rfloor)$$

可以将堆看做是一个完全二叉树。并且，每个结点的值都大于等于其左右孩子结点的值，称为大顶堆；或者每个结点的值都小于等于其左右孩子结点的值，称为小顶堆。



如何将二叉树调整为堆：从二叉树最后一个非叶子结点(其在数组中的序号(数组从1开始计数)一般为数组的长度除以2)开始，将结点上的值依次和左右子树中的值做比较，若子树上的值大于结点值，则将其交换。对二叉树上所有的非叶子结点执行上述操作，最终将二叉树调整成为了一个大顶堆。  
[https://blog.csdn.net/liang\\_gu](https://blog.csdn.net/liang_gu)

堆排序(Heap Sort)是利用堆进行排序的方法。其基本思想为：将待排序列构造成为一个大顶堆(或小顶堆)，整个序列的最大值(或最小值)就是堆顶的根结点，将根节点的值和堆数组的末尾元素交换，此时末尾元素就是最大值(或最小值)，然后将剩余的 n-1 个序列重新构造成为一个堆，这样就会得到 n 个元素中的次大值(或次小值)，如此反复执行，最终得到一个有序序列。

Java 实现代码：

```

package test;

import java.util.Arrays;
/* 堆排序算法实现代码
 * author: 皓宇 QAQ
 */
public class HeapSort{
    public static void main(String[] args)

```

```

{
    new HeapSort().run();
}
public void run() {
    int a[] = {64,45,75,2,31,99,9,7};
    int len=a.length;
    System.out.println("—————堆排
序算法—————");
    System.out.println("排序前:");
    System.out.println(Arrays.toString(a));
    /**
     * 循环建堆
     */
    for(int i=0;i<len-1;i++){
        /**
         * 建堆，建一次最大堆，寻到一个待排序序列的最大数
         */
        buildMaxHeap(a, len-1-i);
        /**
         * 交换堆顶（待排序序列最大数）和最后一个元素
         */
        swap(a,0, len-1-i);
    }
    System.out.println("排序后:");
    for(int j=0;j<len;j++){
        System.out.print(a[j]+" ");
    }
}
/**
 * 对数组 从 0 到 lastIndex 建大顶堆
 */
public void buildMaxHeap(int[] arr, int lastIndex){
    /**
     * 从最后一个节点（lastIndex）的父节点开始
     */
    for(int i=(lastIndex-1)/2;i>=0;i--){
        /**
         * k 保存正在判断的节点
         */
        int k=i;
        /**
         * 如果当前 k 节点的子节点存在
         */
        while(k*2+1<=lastIndex){

```

```

    /**
     * k 节点的左子节点的索引
     */
    int biggerIndex=2*k+1;
    /**
     * 如果 k 节点的左子节点 biggerIndex 小于 lastIndex，即
    biggerIndex+1 代表的 k 节点的右子节点存在
     */
    if(biggerIndex<lastIndex) {
        /**
         * 若果右子节点的值较大
         */
        if(arr[biggerIndex]<arr[biggerIndex+1]) {
            /**
             * biggerIndex 总是记录较大子节点的索引
             */
            biggerIndex++;
        }
    }
    /**
     * 如果 k 节点的值小于其较大的子节点的值，交换他们
     */
    if(arr[k]<arr[biggerIndex]) {
        swap(arr,k,biggerIndex);
        /**
         * 将 biggerIndex 赋予 k，开始 while 循环的下一次循环，
        重新保证 k 节点的值大于其左右子节点的值
         */
        k=biggerIndex;
    }else{
        /**
         * 当前判断结点 k（父结点），大于他的两个子节点时，跳出
        while 循环
         */
        break;
    }
}

}

/**
 * 交换下标为 i、j 的两个元素
 */
private void swap(int[] data, int i, int j) {
    int tmp=data[i];

```

```

        data[i]=data[j];
        data[j]=tmp;
    }
}

```

运行结果:



## 八、基数排序算法

基数排序也无需元素间相互比较和相互交换位置，类似计数排序和桶排序，我们需要对元素进行“分类”。基数排序的关键就是使用基数字 0-9 进行数字排序，需要实现分配与收集的过程。分配：我们都知道任何数字都是 0-9 组成的，所以我们先设置 10 个“桶”，分别代表 0-9 十个数字，我们从个位开始对元素的进行分配。然后收集，第一遍能保证序列中的个位数变为有序。然后再对收集的数列按照十位数字进行分配，这次能保证序列十位数和个位数有序。依次类推，进行分配和收集的重复操作，直至最高位，然后实现整体排序。（桶排序方法自行百度。我桶排序和基数排序理解少。。。简单说下）

**基本思想：**类似于桶式排序，我们需要给待排序记录准备 10 个桶，为什么是 10 个？？因为一个数的任何一位上，其数字大小都位于 0~9 之间，因此采用 10 个桶，桶的编号分别为 0, 1, 2, 3, 4... 9，对应待排序记录中每个数相应位的数值，基数排序也是因此而得名。我们先根据待排序记录的每个数的个位来决定让其加入哪个桶中。

**Java 实现代码：**

```

package test;
import java.util.Arrays;
/* 堆排序算法实现代码
 * author: 皓宇 QAQ
 */
public class RadixSort{

    public static void sort(int[] number, int d) //d 表示最大的数有多少位
    {
        int k = 0;
        int n = 1;

```



```

        int m = 1; //控制键值排序依据在哪一位
        int[][] temp = new int[10][number.length]; //数组的第一维表示可能的余数 0-9
        int[] order = new int[10]; //数组 orderp[i] 用来表示该位是 i 的数的个数
        while (m <= d)
        {
            for (int i = 0; i < number.length; i++)
            {
                int lsd = ((number[i] / n) % 10);
                temp[lsd][order[lsd]] = number[i];
                order[lsd]++;
            }
            for (int i = 0; i < 10; i++)
            {
                if (order[i] != 0)
                    for (int j = 0; j < order[i]; j++)
                    {
                        number[k] = temp[i][j];
                        k++;
                    }
                order[i] = 0;
            }
            n *= 10;
            k = 0;
            m++;
        }
    }

    public static void main(String[] args)
    {
        int[] data =
        {73, 22, 93, 43, 55, 14, 28, 65, 39, 81, 33, 100};
        System.out.println("—————基数
排序算法—————");
        System.out.println("排序前:");
        System.out.println(Arrays.toString(data));
        System.out.println("排序后:");
        RadixSort.sort(data, 3);
        for (int i = 0; i < data.length; i++)
        {
            System.out.print(data[i] + " ");
        }
    }
}

```

运行结果:

```

Servers Problems Tasks Web Browser Project Migration Debug DB Brow
<terminated> RadixSort [Java Application] G:\MyEclipse Professional\binary\com.sun.java.jdk.v
基数排序算法
排序前:
[73, 22, 93, 43, 55, 14, 28, 65, 39, 81, 33, 100]
排序后:
14 22 28 33 39 43 55 65 73 81 93 100

```

算法复杂度总结:

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$	不稳定
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定