02.算法设计目标与时间复杂度与空间复杂度

2.1.算法的时间复杂度

O()函数

表示算法的时间效率与算法所处理的数据元素个数n函数关系的最常用函数是O()函数。

**定义：**

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。

2.2.空间复杂度

空间复杂度是指算法在运行期间所需要的内存空间的数量级。

由于大部分算法的空间复杂度问题并不严重，并且算法的空间复杂度分析方法和算法的时间复杂度分析方法基本相同，所以一般数据结构只讨论算法的时间复杂度，不讨论算法的空间复杂度。

2.3.冒泡排序

冒泡排序算法的时间复杂度T(n)=O(n2)

|  |
| --- |
| public class BubbleSort {  //冒泡排序算法，通过flag变量判断排序是否提前结束。  public static void BubbleSort(int arr[])  {  int n = arr.length;  int i,j,temp,flag;  flag = 1;//flag:1表示排序没有结束，0表示排序已经结束  for(i=1;i<n&&flag==1;i++)  {  for(j=0;j<n-i;j++)  {  flag = 0;//如果以后的循环不改变flag的值，说明没有发生数组元素的交换  //也就是说，这个数组已经排好序了。所有就可以提前退出循环。  if(arr[j]>arr[j+1])  {  flag = 1;  temp = arr[j];  arr[j]=arr[j+1];  arr[j+1]=temp;  }  }  }  }    public static void main(String[] args)  {  int arr[]={34,67,32,19,8,20,17,44,67,80,42,57,38};  BubbleSort.BubbleSort(arr);  System.out.println("-----排序结果------");  for(int i=0;i<arr.length;i++)  {  System.out.print(arr[i]+" ");  }  }  } |

03.线性结构与顺序表的实现与应用

3.1.线性结构

**线性结构定义**

**如果一个数据元素序列满足：**

（1）除第一个和最后一个数据元素外，每个数据元素只有一个前驱数据元素和一个后继数据元素；

（2）第一个数据元素没有前驱数据元素；

（3）最后一个数据元素没有后继数据元素。

3.2.线性表抽象数据类型

**线性表抽象数据类型**

数据集合可以表示为a0,a1,a2,...an-1,每个数据元素的数据类型可以是任意的类型。

操作集合包括如下：

1.求元素个数

2.插入

3.删除

4.查找

5.判断是否为空

|  |
| --- |
| **package** stringpro;  //线性表接口  **public** **interface** List {  // 获得线性表长度  **public** **int** size();  // 判断线性表是否为空  **public** **boolean** isEmpty();  // 插入元素  **public** **void** insert(**int** index, Object obj) **throws** Exception;  // 删除元素  **public** **void** delete(**int** index) **throws** Exception;  // 获取指定位置的元素  **public** Object get(**int** index) **throws** Exception;  } |

3.3.顺序表

计算机有两种基本的存储结构：顺序结构、离散结构。使用顺序结构实现的线性表称为顺序表。

**顺序表效率分析**

顺序表插入和删除一个元素的时间复杂度为O(n)。

顺序表支持随机访问，顺序表读取一个元素的时间复杂度为O(1)。

顺序表的优点是：支持随机访问，空间利用率高。

顺序表的缺点是：大小固定，插入和删除元素需要移动大量的数据。

|  |  |
| --- | --- |
| **package** stringpro;  **public** **class** SequenceList **implements** List {  **final** **int** defaultSize = 10;// 默认的顺序表的最大长度  **int** maxSize;// 最大长度  **int** size;// 当前长度  Object[] listArray;// 对象数组  **public** SequenceList() {  init(defaultSize);  }  **public** SequenceList(**int** size) {  init(size);  }  **private** **void** init(**int** size) {//顺序表的初始化方法  maxSize = size;  **this**.size = 0;  listArray = **new** Object[size];  }  @Override  **public** **void** delete(**int** index) **throws** Exception{  **if** (isEmpty()) {  **throw** **new** Exception("顺序表为空，无法删除！");  }  **if** (index < 0 || index > size - 1) {  **throw** **new** Exception("参数错误！");  }  // 移动元素  **for** (**int** j = index; j < size - 1; j++) {  listArray[j] = listArray[j + 1];  }  size--;  } | @Override  **public** Object get(**int** index) **throws** Exception {  **if** (index < 0 || index >= size) {  **throw** **new** Exception("参数错误！");  }  **return** listArray[index];  }  @Override  **public** **void** insert(**int** index, Object obj) **throws** Exception {  // 线性表已满  **if** (size == maxSize) {  **throw** **new** Exception("顺序表已满，无法插入！");  }  // 插入位置编号是否合法  **if** (index < 0 || index > size) {  **throw** **new** Exception("参数错误！");  }  // 移动元素  **for** (**int** j = size; j > index; j--) {  listArray[j] = listArray[j - 1];  }  listArray[index] = obj;  size++;  }  @Override  **public** **boolean** isEmpty() {  **return** size == 0;  }  @Override  **public** **int** size() {  **return** size;  }  } |

3.4.顺序表应用

设计一个顺序表，可以保存100个学生的资料，保存以下三个学生的资料，并打印输出

|  |  |
| --- | --- |
| **package** stringpro;  //学生类  **public** **class** Students {  **private** String sid;// 学号  **private** String name;// 姓名  **private** String gender;// 性别  **private** **int** age;// 年龄    **public** Students()  {    }    **public** Students(String sid,String name,String gender,**int** age)  {  **this**.sid = sid;  **this**.name = name;  **this**.gender = gender;  **this**.age =age;  }    **public** String toString()  {  **return** "学号："+**this**.getSid()+" 姓名："+**this**.getName()+" 性别："+**this**.getGender()+" 年龄:"+**this**.getAge();  }  Get/Setter  。。。  } | **package** stringpro;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  SequenceList list = **new** SequenceList(100);  **try** {  list.insert(list.size, **new** Students("S0001", "张三", "男", 18));  list.insert(list.size, **new** Students("S0002", "李四", "男", 19));  list.insert(list.size, **new** Students("S0003", "王五", "女", 21));  **for** (**int** i = 0; i < list.size; i++) {  System.***out***.println(list.get(i));  }  } **catch** (Exception ex) {  ex.printStackTrace();  }  }  } |
| 学号：S0001 姓名：张三 性别：男 年龄:18  学号：S0002 姓名：李四 性别：男 年龄:19  学号：S0003 姓名：王五 性别：女 年龄:21 | |

04.单向链表以及单向链表的应用

4.1.链表结构

* 链式存储结构是基于指针实现的。我们把一个数据元素和一个指针称为结点。
* 链式存储结构是用指针把相互直接关联的结点（即直接前驱结点或直接后继结点）链接起来。链式存储结构的线性表称为链表。

4.2.链表类型

* 根据链表的构造方式的不同可以分为：

1.单向链表

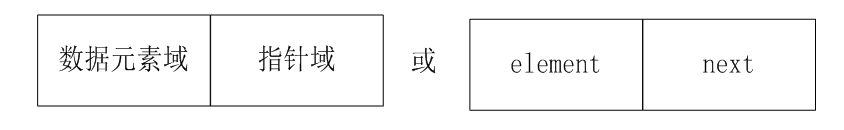
2.单向循环链表

3.双向循环链表

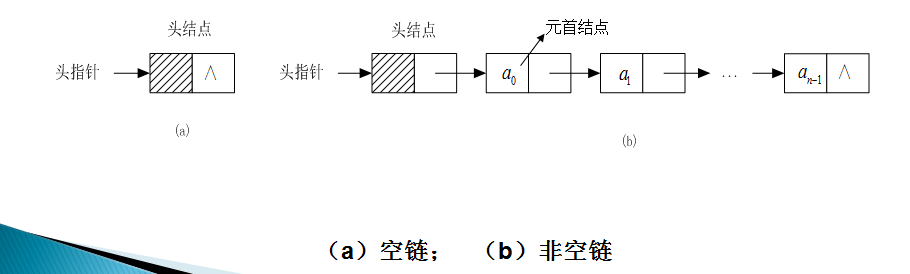
4.3.单链表

**单链表结构**

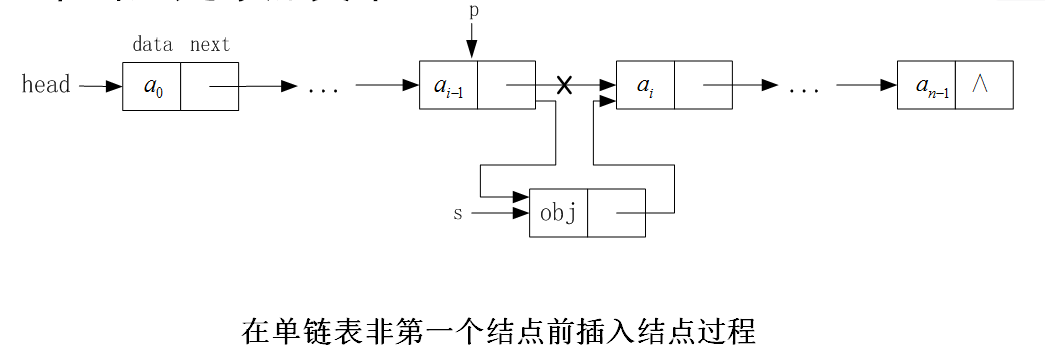
* 单链表是构成链表的每个结点只有一个指向直接后继结点的指针。
* 单链表的表示方法：
* 单链表中每个结点的结构：



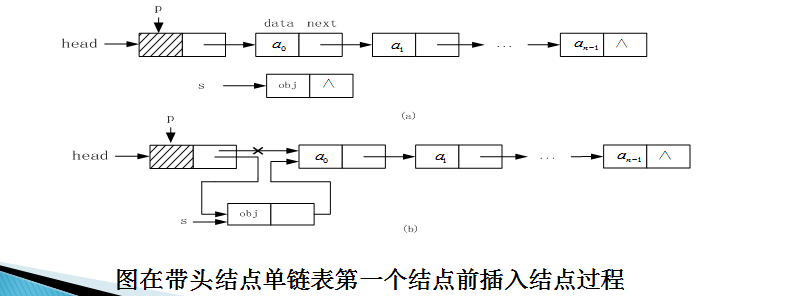
* 单链表有带头结点结构和不带头结点结构两种。我们把指向单链表的指针称为单链表的头指针。头指针所指的不存放数据元素的第一个结点称作头结点。存放第一个数据元素的结点称作第一个数据元素结点，或称首元结点。



* 从线性表的定义可知，线性表要求允许在任意位置进行插入和删除。当选用带头结点的单链表时，插入和删除操作的实现方法比不用带头结点单链表的实现方法简单。
* 设头指针用head表示，在单链表中任意结点（但不是第一个数据元素结点）前插入一个新结点的方法如图2-6所示。算法实现时，首先把插入位置定位在要插入结点的前一个结点位置，然后把s表示的新结点插入单链表中。



* 要在第一个数据元素结点前插入一个新结点，若采用**不带头结点**的单链表结构，则结点插入后，头指针head就要等于新插入结点s，这和在非第一个数据元素结点前插入结点时的情况不同。另外，还有一些不同情况需要考虑。因此，算法对这两种情况就要分别设计实现方法。
* 而如果采用**带头结点**的单链表结构，算法实现时，p指向头结点，改变的是p指针的next指针的值，而头指针head的值不变。因此，算法实现方法比较简单。在带头结点单链表中第一个数据元素结点前插入一个新结点的过程如图所示。



4.4.结点类

* 单链表是由一个一个结点组成的，因此，要设计单链表类，必须先设计结点类。结点类的成员变量有两个：一个是数据元素，另一个是表示下一个结点的对象引用（即指针）。
* 设计操作：

1.头结点的初始化

2.非头结点的构造

3.获取该结点指向的下个结点

4.设置该结点指向的下个结点

5.设置该结点的数据

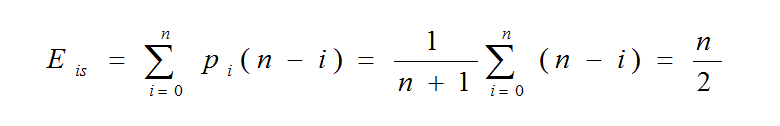
6.获取该结点的数据

4.5.单链表类

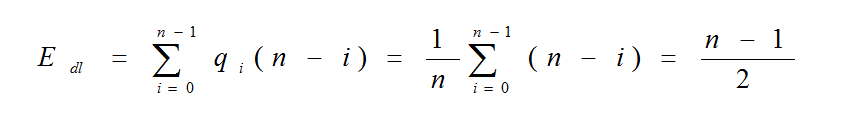
* 单链表类的成员变量至少要有两个：一个是头指针，另一个是单链表中的数据元素个数。但是，如果再增加一个表示单链表当前结点位置的成员变量，则有些成员函数的设计将更加方便。

4.6.单链表的效率分析

* 在单链表的任何位置上插入数据元素的概率相等时，在单链表中插入一个数据元素时比较数据元素的平均次数为：



* 删除单链表的一个数据元素时比较数据元素的平均次数为：



* 因此，单链表插入和删除操作的时间复杂度均为O（n）。另外，单链表取数据元素操作的时间复杂度也为O（n）。

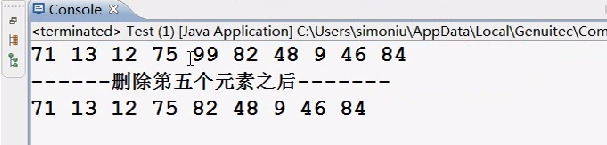
4.7.顺序表和单链表的比较

* 顺序表的主要优点是**支持随机读取**，以及内存空间利用效率高；顺序表的主要缺点是需要预先给出数组的最大数据元素个数，而这通常很难准确作到。当实际的数据元素个数超过了预先给出的个数，会发生异常。另外，顺序表插入和删除操作时需要移动较多的数据元素。
* 和顺序表相比，**单链表的主要优点是不需要预先给出数据元素的最大个数。另外，单链表插入和删除操作时不需要移动数据元素**。
* **单链表的主要缺点是每个结点中要有一个指针，因此单链表的空间利用率略低于顺序表的**。另外，单链表不支持随机读取，单链表取数据元素操作的时间复杂度为O（n）；而顺序表支持随机读取，顺序表取数据元素操作的时间复杂度为O（1）。

4.8.单链表的应用

* 使用单链表建立一个线性表，依次输入十个0-99之间的随机数，删除第5个元素，打印输出该线性表。

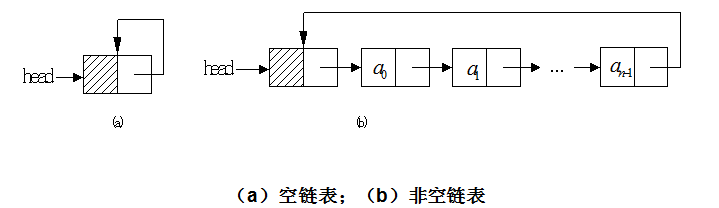
|  |  |
| --- | --- |
| //线性表接口  **public** **interface** List {  // 获得线性表长度  **public** **int** size();  // 判断线性表是否为空  **public** **boolean** isEmpty();  // 插入元素  **public** **void** insert(**int** index, Object obj) **throws** Exception;  // 删除元素  **public** **void** delete(**int** index) **throws** Exception;  // 获取指定位置的元素  **public** Object get(**int** index) **throws** Exception;  } | //单向链表类  **public** **class** LinkList **implements** List {  Node head; // 头指针  Node current;// 当前结点对象  **int** size;// 结点个数  // 初始化一个空链表  **public** LinkList() {  // 初始化头结点，让头指针指向头结点。并且让当前结点对象等于头结点。  **this**.head = current = **new** Node(**null**);  **this**.size = 0;// 单向链表，初始长度为零。  }  // 定位函数，实现当前操作对象的前一个结点，也就是让当前结点对象定位到要操作结点的前一个结点。  **public** **void** index(**int** index) **throws** Exception {  **if** (index < -1 || index > size - 1) {  **throw** **new** Exception("参数错误！");  }  // 说明在头结点之后操作。  **if** (index == -1)  **return**;  current = head.next;  **int** j = 0;// 循环变量  **while** (current != **null** && j < index) {  current = current.next;  j++;  }  }  @Override  **public** **void** delete(**int** index) **throws** Exception {  // 判断链表是否为空  **if** (isEmpty()) {  **throw** **new** Exception("链表为空，无法删除！");  }  **if** (index < 0 || index > size) {  **throw** **new** Exception("参数错误！");  }  index(index - 1);// 定位到要操作结点的前一个结点对象。  current.setNext(current.next.next);  size--;  }  @Override  **public** Object get(**int** index) **throws** Exception {  **if** (index < -1 || index > size - 1) {  **throw** **new** Exception("参数非法！");  }  index(index);  **return** current.getElement();  }  @Override  **public** **void** insert(**int** index, Object obj) **throws** Exception {  **if** (index < 0 || index > size) {  **throw** **new** Exception("参数错误！");  }  index(index - 1);// 定位到要操作结点的前一个结点对象。  current.setNext(**new** Node(obj, current.next));  size++;  }  @Override  **public** **boolean** isEmpty() {  **return** size == 0;  }  @Override  **public** **int** size() {  **return** **this**.size;  }  } |
| //结点类  **public** **class** Node {  Object element; // 数据域  Node next; // 指针域  // 头结点的构造方法  **public** Node(Node nextval) {  **this**.next = nextval;  }  // 非头结点的构造方法  **public** Node(Object obj, Node nextval) {  **this**.element = obj;  **this**.next = nextval;  }  // 获得当前结点的后继结点  **public** Node getNext() {  **return** **this**.next;  }  // 获得当前的数据域的值  **public** Object getElement() {  **return** **this**.element;  }  // 设置当前结点的指针域  **public** **void** setNext(Node nextval) {  **this**.next = nextval;  }  // 设置当前结点的数据域  **public** **void** setElement(Object obj) {  **this**.element = obj;  }  **public** String toString() {  **return** **this**.element.toString();  }  } |
| **public** **class** Test {  **public** **static** **void** main(String[] args) **throws** Exception {  LinkList list = **new** LinkList();  **for** (**int** i = 0; i < 10; i++) {  **int** temp = ((**int**) (Math.*random*() \* 100)) % 100;  list.insert(i, temp);  System.***out***.print(temp + " ");  }  list.delete(4);  System.***out***.println("\n------删除第五个元素之后-------");  **for** (**int** i = 0; i < list.size; i++) {  System.***out***.print(list.get(i) + " ");  }  }  } |



05.循环链表仿真链表以及循环链表应用

5.1.单向循环链表

* 单向循环链表是单链表的另一种形式，其结构特点是链表中最后一个结点的指针不再是结束标记，而是指向整个链表的第一个结点，从而使单链表形成一个环。和单链表相比，循环单链表的长处是从链尾到链头比较方便。当要处理的数据元素序列具有环型结构特点时，适合于采用循环单链表。
* 和单链表相同，循环单链表也有带头结点结构和不带头结点结构两种，带头结点的循环单链表实现插入和删除操作时，算法实现较为方便。
* 带头结点的循环单链表结构如下：



* 带头结点的循环单链表的操作实现方法和带头结点的单链表的操作实现方法类同，差别仅在于：

（1）在构造函数中，要加一条head.next = head 语句，把初始时的带头结点的循环单链表设计成图2-11（a）所示的状态。

（2）在index(i)成员函数中，把循环结束判断条件current != null改为current != head。

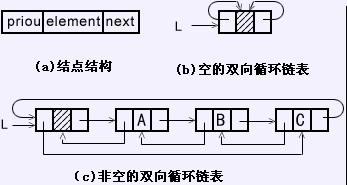
|  |  |
| --- | --- |
| //单向链表类  public class CycleLinkList implements List {  Node head; //头指针  Node current;//当前结点对象  int size;//结点个数    //初始化一个空链表  public CycleLinkList()  {  //初始化头结点，让头指针指向头结点。并且让当前结点对象等于头结点。  this.head = current = new Node(null);  this.size =0;//单向链表，初始长度为零。  this.head.next = head;  }    //定位函数，实现当前操作对象的前一个结点，也就是让当前结点对象定位到要操作结点的前一个结点。  public void index(int index) throws Exception  {  if(index <-1 || index > size -1)  {  throw new Exception("参数错误！");  }  //说明在头结点之后操作。  if(index==-1)  return;  current = head.next;  int j=0;//循环变量  while(current != head&&j<index)  {  current = current.next;  j++;  }    }    @Override  public void delete(int index) throws Exception {  // TODO Auto-generated method stub  //判断链表是否为空  if(isEmpty())  {  throw new Exception("链表为空，无法删除！");  }  if(index <0 ||index >size)  {  throw new Exception("参数错误！");  }  index(index-1);//定位到要操作结点的前一个结点对象。  current.setNext(current.next.next);  size--;  }  @Override  public Object get(int index) throws Exception {  // TODO Auto-generated method stub  if(index <-1 || index >size-1)  {  throw new Exception("参数非法！");  }  index(index);    return current.getElement();  }  @Override  public void insert(int index, Object obj) throws Exception {  // TODO Auto-generated method stub  if(index <0 ||index >size)  {  throw new Exception("参数错误！");  }  index(index-1);//定位到要操作结点的前一个结点对象。  current.setNext(new Node(obj,current.next));  size++;  }  @Override  public boolean isEmpty() {  // TODO Auto-generated method stub  return size==0;  }  @Override  public int size() {  // TODO Auto-generated method stub  return this.size;  }  } | //线性表接口  public interface List {  //获得线性表长度  public int size();  //判断线性表是否为空  public boolean isEmpty();  //插入元素  public void insert(int index,Object obj) throws Exception;  //删除元素  public void delete(int index) throws Exception;  //获取指定位置的元素  public Object get(int index) throws Exception;  } |
| //结点类  public class Node {    Object element; //数据域  Node next; //指针域    //头结点的构造方法  public Node(Node nextval)  {  this.next = nextval;  }    //非头结点的构造方法  public Node(Object obj,Node nextval)  {  this.element = obj;  this.next = nextval;  }    //获得当前结点的后继结点  public Node getNext()  {  return this.next;  }    //获得当前的数据域的值  public Object getElement()  {  return this.element;  }    //设置当前结点的指针域  public void setNext(Node nextval)  {  this.next = nextval;  }    //设置当前结点的数据域  public void setElement(Object obj)  {  this.element = obj;  }    public String toString()  {  return this.element.toString();  }  } |

5.2.双向循环链表

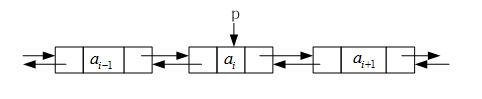
* 双向链表是每个结点除后继指针外还有一个前驱指针。和单链表类同，双向链表也有带头结点结构和不带头结点结构两种，带头结点的双向链表更为常用；另外，双向链表也可以有循环和非循环两种结构，循环结构的双向链表更为常用。
* 在双向链表中，每个结点包括三个域，分别是element域、next域和prior域，其中element域为数据元素域，next域为指向后继结点的对象引用，prior域为指向前驱结点的对象引用。下图为双向链表结点的图示结构。



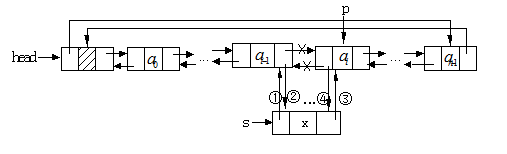
* 如下图是带头结点的循环双向链表的图示结构。循环双向链表的next和prior各自构成自己的循环单链表。



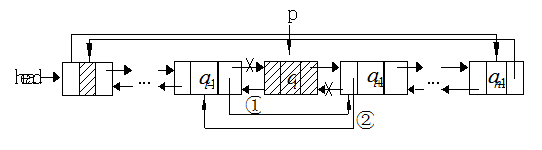
* 在双向链表中，有如下关系：设对象引用p表示双向链表中的第i个结点，则p.next表示第i+1个结点，p.next.prior仍表示第i个结点，即p.next.prior == p；同样地，p.prior表示第i-1个结点，p.prior.next仍表示第i个结点，即p.prior.next == p。下图是双向链表上述关系的图示。



* 循环双向链表的插入过程如下图所示。图中的指针p表示要插入结点的位置，s表示要插入的结点，①、②、③、④表示实现插入过程的步骤。



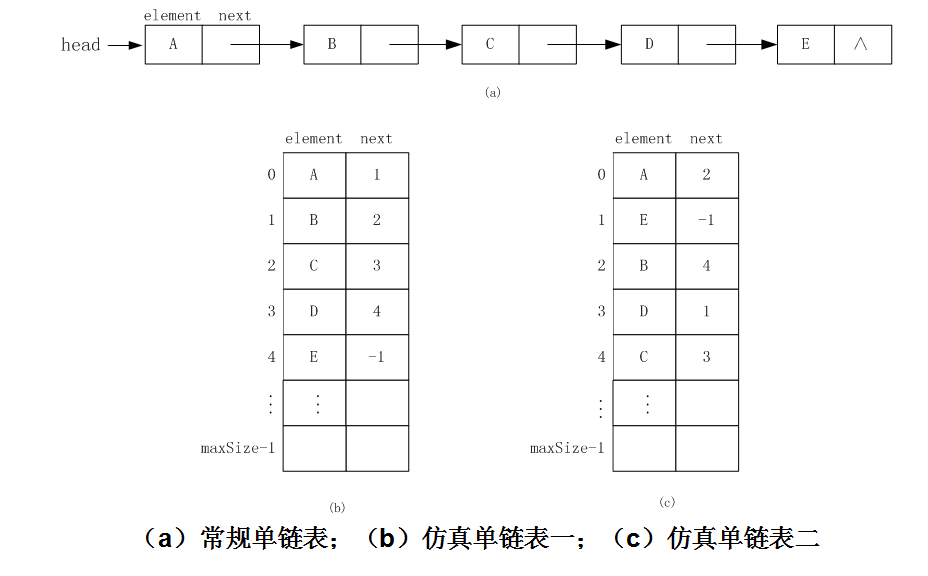
* 循环双向链表的删除过程如下图所示。图中的指针p表示要插入结点的位置，①、②表示实现删除过程的步骤。



|  |  |
| --- | --- |
| //单向链表类  public class DoubleCycleLinkList implements List {  Node head; //头指针  Node current;//当前结点对象  int size;//结点个数    //初始化一个空链表  public DoubleCycleLinkList()  {  //初始化头结点，让头指针指向头结点。并且让当前结点对象等于头结点。  this.head = current = new Node(null);  this.size =0;//单向链表，初始长度为零。  this.head.next = head;  this.head.prior = head;  }    //定位函数，实现当前操作对象的前一个结点，也就是让当前结点对象定位到要操作结点的前一个结点。  public void index(int index) throws Exception  {  if(index <-1 || index > size -1)  {  throw new Exception("参数错误！");  }  //说明在头结点之后操作。  if(index==-1)  return;  current = head.next;  int j=0;//循环变量  while(current != head&&j<index)  {  current = current.next;  j++;  }    }    @Override  public void delete(int index) throws Exception {  // TODO Auto-generated method stub  //判断链表是否为空  if(isEmpty())  {  throw new Exception("链表为空，无法删除！");  }  if(index <0 ||index >size)  {  throw new Exception("参数错误！");  }  index(index-1);//定位到要操作结点的前一个结点对象。  current.setNext(current.next.next);  current.next.setPrior(current);  size--;  }  @Override  public Object get(int index) throws Exception {  // TODO Auto-generated method stub  if(index <-1 || index >size-1)  {  throw new Exception("参数非法！");  }  index(index);    return current.getElement();  }  @Override  public void insert(int index, Object obj) throws Exception {  // TODO Auto-generated method stub  if(index <0 ||index >size)  {  throw new Exception("参数错误！");  }  index(index-1);//定位到要操作结点的前一个结点对象。  current.setNext(new Node(obj,current.next));  current.next.setPrior(current);  current.next.next.setPrior(current.next);    size++;  }  @Override  public boolean isEmpty() {  // TODO Auto-generated method stub  return size==0;  }  @Override  public int size() {  // TODO Auto-generated method stub  return this.size;  }  } | //线性表接口  public interface List {  //获得线性表长度  public int size();  //判断线性表是否为空  public boolean isEmpty();  //插入元素  public void insert(int index,Object obj) throws Exception;  //删除元素  public void delete(int index) throws Exception;  //获取指定位置的元素  public Object get(int index) throws Exception;  } |
| //结点类  public class Node {    Object element; //数据域  Node next; //后继指针域  Node prior; //前驱指针域  //头结点的构造方法  public Node(Node nextval)  {  this.next = nextval;  }    //非头结点的构造方法  public Node(Object obj,Node nextval)  {  this.element = obj;  this.next = nextval;  }    //获得当前结点的后继结点  public Node getNext()  {  return this.next;  }    //获得当前结点的前驱结点  public Node getPrior()  {  return this.prior;  }  //获得当前的数据域的值  public Object getElement()  {  return this.element;  }    //设置当前结点的后继指针域  public void setNext(Node nextval)  {  this.next = nextval;  }    //设置当前结点的前驱指针域  public void setPrior(Node priorval)  {  this.prior = priorval;  }    //设置当前结点的数据域  public void setElement(Object obj)  {  this.element = obj;  }    public String toString()  {  return this.element.toString();  }  } |
| public class Test {  /\*\*  \* @param args  \*/  public static void main(String[] args) throws Exception {  // TODO Auto-generated method stub  DoubleCycleLinkList list = new DoubleCycleLinkList();  for(int i=0;i<10;i++)  {  int temp = ((int)(Math.random()\*100))%100;  list.insert(i, temp);  System.out.print(temp+" ");  }  list.delete(4);  System.out.println("\n------删除第五个元素之后-------");  for(int i=0;i<list.size;i++)  {  System.out.print(list.get(i)+" ");  }  }  } |

5.3.仿真链表

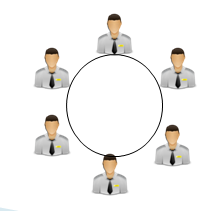
* 在链式存储结构中，我们实现数据元素之间的次序关系依靠指针。我们也可以用数组来构造仿真链表。方法是在数组中增加一个（或两个）int类型的变量域，这些变量用来表示后一个（或前一个）数据元素在数组中的下标。我们把这些int类型变量构造的指针称为仿真指针。这样，就可以用仿真指针构造仿真的单链表（或仿真的双向链表）。



5.4.循环链表的应用

* 编写击鼓传花小游戏。

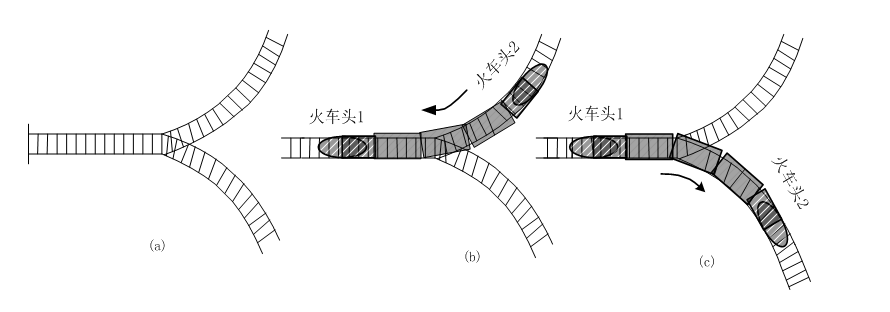
游戏规则：N个人围成一个圈，从第一个人开始传花，当数到M时，该人退出游戏，直到剩下最后一个人。



06.栈的基本概念以及顺序栈的应用

6.1.堆栈的基本概念

* 堆栈（也简称作栈）是一种特殊的线性表，堆栈的数据元素以及数据元素间的逻辑关系和线性表完全相同，其差别是线性表允许在任意位置进行插入和删除操作，而堆栈只允许在固定一端进行插入和删除操作。
* 堆栈中允许进行插入和删除操作的一端称为栈顶，另一端称为栈底。堆栈的插入和删除操作通常称为进栈或入栈，堆栈的删除操作通常称为出栈或退栈。
* 从输入和输出数据元素的位置关系看，堆栈的功能和一种火车调度装置的功能类同。



6.2.堆栈抽象数据类型

* 数据集合

堆栈的数据集合可以表示为a0,a1,…,an-1，每个数据元素的数据类型可以是任意的类类型。

* 操作集合

（1）入栈push(obj)：把数据元素obj插入堆栈。

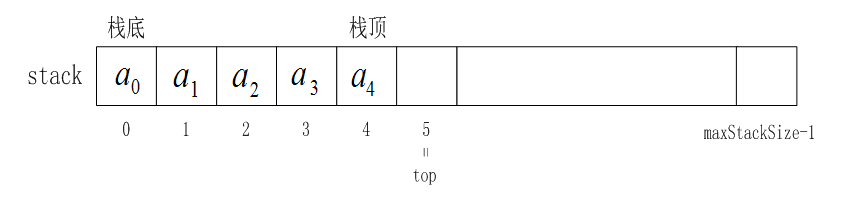
（2）出栈pop()：出栈, 删除的数据元素由函数返回。

（3）取栈顶数据元素getTop()：取堆栈当前栈顶的数据元素并由函数返回。

（4）非空否notEmpty()：若堆栈非空则函数返回true，否则函数返回false。

6.3.顺序栈

* 顺序存储结构的堆栈称作顺序堆栈。
* 顺序堆栈的存储结构示意图如图所示。



6.4.顺序栈实现

* 设计Stack接口
* 实现SeqStack类

|  |  |
| --- | --- |
| //顺序栈  public class SequenceStack implements Stack {    Object[] stack; //对象数组  final int defaultSize =10; //默认最大长度  int top; //栈顶位置  int maxSize; //最大长度    public SequenceStack()  {  init(defaultSize);  }    public SequenceStack(int size)  {  init(size);  }    public void init(int size)  {  this.maxSize = size;  top =0;  stack = new Object[size];  }    @Override  public Object getTop() throws Exception {  // TODO Auto-generated method stub  if(isEmpty())  {  throw new Exception("堆栈为空！");  }      return stack[top-1];  }  @Override  public boolean isEmpty() {  // TODO Auto-generated method stub  return top==0;  }  @Override  public Object pop() throws Exception {  // TODO Auto-generated method stub  if(isEmpty())  {  throw new Exception("堆栈为空！");  }  top--;    return stack[top];  }  @Override  public void push(Object obj) throws Exception {  // TODO Auto-generated method stub  //首先判断栈是否已满  if(top == maxSize)  {  throw new Exception("堆栈已满！");  }  stack[top]=obj;  top++;  }  } | //栈接口  public interface Stack {  //入栈  public void push(Object obj) throws Exception;  //出栈  public Object pop() throws Exception;  //获得栈顶元素  public Object getTop() throws Exception;  //判断栈是否为空  public boolean isEmpty();  } |
| public class Test {  public static void main(String[] args) throws Exception {  // TODO Auto-generated method stub  SequenceStack stack = new SequenceStack(10);    Scanner in = new Scanner(System.in);  int temp;  for(int i=0;i<10;i++)  {  System.out.println("请输入第"+(i+1)+"个整数：");  temp = in.nextInt();  stack.push(temp);  }    while(!stack.isEmpty())  {  System.out.println(stack.pop());  }    }  } |
| public class TestStack {  public static void main(String[] args) {  String s1 = "china";  String s2 = "china";  //凡是用new关键创建的对象，都是在堆内存中分配空间。  String s3 = new String("china");    //凡是new出来的对象，绝对是不同的两个对象。  String s4 = new String("china");    System.out.println(s1==s2);  System.out.println(s1==s3);  System.out.println(s3==s4);  System.out.println(s3.equals(s4));    System.out.println("\n-----------------\n");  /\*String很特殊，重写从父类继承过来的hashCode方法，使得两个  \*字符串如果内容相等，那么hashCode也相等。  \*\*/  System.out.println(s3.hashCode());  System.out.println(s4.hashCode());    System.out.println("\n-----------------\n");  System.out.println(System.identityHashCode(s3));  System.out.println(System.identityHashCode(s4));    System.out.println("\n-----------------\n");  System.out.println(s1.hashCode());  System.out.println(s2.hashCode());    System.out.println("\n-----------------\n");  System.out.println(System.identityHashCode(s1));  System.out.println(System.identityHashCode(s2));    }  } |

6.5.顺序栈的应用

* 设计一个顺序栈，从键盘输入十个整数压进栈，然后再弹出栈，并打印出栈序列。

6.6.Java中栈与堆的区别

* 栈(stack):是一个先进后出的数据结构,通常用于保存方法(函数)中的参数,局部变量。在java中,所有基本类型和引用类型都在栈中存储。栈中数据的生存空间一般在当前scopes内(就是由{...}括起来的区域)。
* 堆(heap):是一个可动态申请的内存空间(其记录空闲内存空间的链表由操作系统维护),C中的malloc语句所产生的内存空间就在堆中。在java中,所有使用new xxx()构造出来的对象都在堆中存储,当垃圾回收器检测到某对象未被引用,则自动销毁该对象。所以,理论上说java中对象的生存空间是没有限制的,只要有引用类型指向它,则它就可以在任意地方被使用。

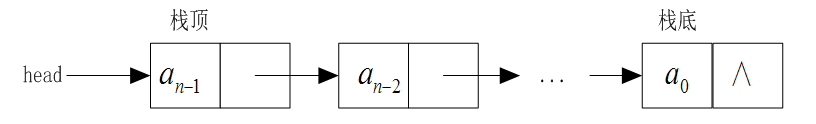
6.7.hashCode与对象之间的关系

* 如果两个对象的hashCode不相同，那么这两个对象肯定也不同。
* 如果两个对象的hashCode相同，那么这两个对象有可能相同，也有可能不同。
* 判断两个对象是否相同，只有一种方法，就是使用==符号。

07.链式堆栈以及栈的应用

7.1.链式堆栈

* 链式存储结构的堆栈称作链式堆栈。
* 与单链表相同，链式堆栈也是由一个个结点组成的，每个结点由两个域组成，一个是存放数据元素的数据元素域element，另一个是存放指向下一个结点的对象引用（即指针）域next。
* 堆栈有两端，插入数据元素和删除数据元素的一端为栈顶，另一端为栈底。链式堆栈都设计成把靠近堆栈头head的一端定义为栈顶。
* 依次向链式堆栈入栈数据元素a0, a1, a2, ..., an-1后，链式堆栈的示意图如图所示。



* 设计链式堆栈

7.2.堆栈的应用

* 堆栈是各种软件系统中应用最广泛的数据结构之一。括号匹配和表达式计算是编译软件中的基本问题，其软件设计中都需要使用堆栈。

括号匹配问题

表达式计算问题

* 括号匹配问题

假设算术表达式中包含圆括号，方括号，和花括号三种类型。使用栈数据结构编写一个算法判断表达式中括号是否正确匹配，并设计一个主函数测试。

比如：{a+[b+(c\*a)/(d-e)]} 正确

([a+b)-(c\*e)]+{a+b} 错误

* 括号匹配有四种情况：

1.左右括号匹配次序不正确

2.右括号多于左括号

3.左括号多于右括号

4.匹配正确

* 表达式计算

3+(6-4/2)\*5=23

其后缀表达式为：3642/-5\*+# (#符号为结束符)

使用链式堆栈，设计一个算法计算表达式：A+(B-C/D)\*E

08.中缀表达式转换后缀表达式算法

8.1.表达式的三种形式

* 中缀表达式：运算符放在两个运算对象中间，如：(2+1)\*3
* 后缀表达式：不包含括号，运算符放在两个运算对象的后面，所有的计算按运算符出现的顺序，严格从左向右进行（不再考虑运算符的优先规则，如：2 1 + 3 \*
* 前缀表达式：同后缀表达式一样，不包含括号，运算符放在两个运算对象的前面，如：\* + 2 1 3

8.2.中缀表达式与后缀表达式转换算法

* 将中缀表达式转换为后缀表达式：

(1)当读到数字直接送至输出队列中；

(2)当读到运算符t时：

a.将栈中所有优先级高于或等于t的运算符弹出，送到输出队列中；

这句话不好理解，可以说成这样，从栈顶开始，依次弹出比当前处理的运算符优先级高的运算符，直到一个比它优先级低的或者遇到了一个左括号就停止。

b.t进栈；

（3）读到左括号时总是将它压入栈中；

（4）读到右括号时，将靠近栈顶的第一个左括号上面的运算符全部依次弹出，送至输出队列后，再丢弃左括号；

（5）中缀表达式全部读完后，若栈中仍有运算符，将其送到输出队列中。

* 中缀表达式：3+(2-5)\*6/3 转换为后缀表达式的过程：

后缀表达式 栈

3

3 +

3 +(

32 +(

32 +(-

325 +(-

325- +

325- +\*

325-6 +\*

325-6\* +/

325-6\*3 +/

325-6\*3/+

最终后缀表达式为：325-6\*3/+

* 运用后缀表达式进行计算：

(1)建立一个栈S；

(2)从左到右读后缀表达式，读到数字就将它转换为数值压入栈S中，读到运算符则从栈中依次弹出两个数分别到Y和X，然后以“X 运算符 Y”的形式计算机出结果，再压加栈S中；

(3)如果后缀表达式未读完，就重复上面过程，最后输出栈顶的数值则为结束。

* 3+(2-5)\*6/3=-3 ,其后缀表达式为：325-6\*3/+
* 运算过程如下：

栈 运算

3 2 5 325入栈

3 2-5=-3

3 -3 运算结果进栈

3 -3 6

3 -3\*6=-18

3 -18 3 -18/3=-6

3 -6 3+(-6)=-3

-3

8.3.JDK Stack类使用。

8.4.使用泛型

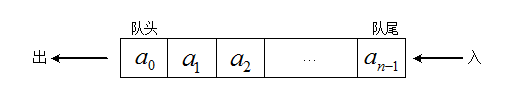
8.5.中缀表达式转换为后缀表达式算法设计

09.队列以及顺序循环队列的应用

9.1.队列

**队列基本概念**

* 队列（简称作队，Queue）也是一种特殊的线性表，队列的数据元素以及数据元素间的逻辑关系和线性表完全相同，其差别是线性表允许在任意位置插入和删除，而队列只允许在其一端进行插入操作在其另一端进行删除操作。
* 队列中允许进行插入操作的一端称为队尾，允许进行删除操作的一端称为队头。队列的插入操作通常称作入队列，队列的删除操作通常称作出队列。
* 下图是一个依次向队列中插入数据元素a0,a1,...,an-1后的示意图，其中，a0是当前队头数据元素，an-1是当前队尾数据元素。



9.2.队列抽象数据类型

1 数据集合

队列的数据集合可以表示为a0,a1,…,an-1，每个数据元素的数据类型可以是任意的类型。

2 操作集合

（1）入队列append(obj)：把数据元素obj插入队尾。

（2）出队列delete()：把队头数据元素删除并由函数返回。

（3）取队头数据元素getFront()：取队头数据元素并由函数返回。

（4）非空否isEmpty()：非空否。若队列非空，则函数返回false，否则函数返回true。

**队列抽象数据类型的Java接口定义如下：**

public interface Queue{

public void append(Object obj) throws Exception;

public Object delete() throws Exception;

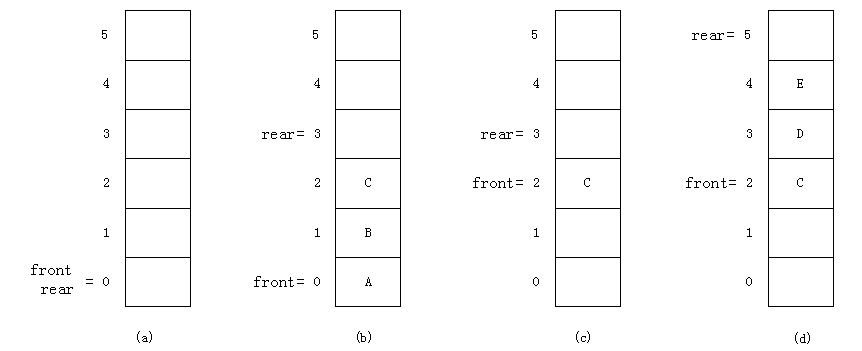
public Object getFront() throws Exception;

public boolean isEmpty();

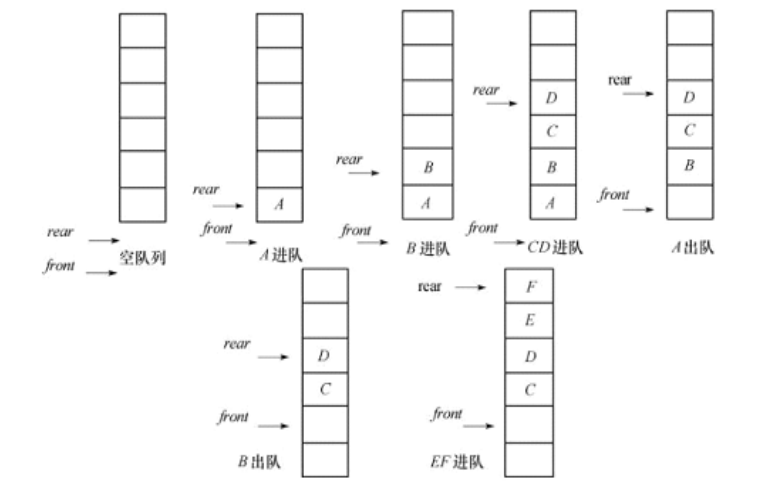
}

9.3.顺序队列

* 顺序队列的存储结构
* 下图是一个有6个存储空间的顺序队列的动态示意图，图中front指示队头，rear指示队尾。



* 顺序队列的"假溢出"现象



9.4.顺序循环队列

* 假溢出是由于队尾rear的值和队头front的值不能由所定义数组下界值自动转为数组上界值而产生的。因此，解决的方法是把顺序队列所使用的存储空间构造成一个逻辑上首尾相连的循环队列( Circular Queue)。
* 当rear和front达到maxSize-1后，再加1就自动到0。这样，就不会出现顺序队列数组的头部已空出许多存储空间，但队尾却因数组下标越界而引起溢出的假溢出问题。
* 解决这个问题的方法有三种：

(1)设计一个布尔变量以判断队列的空和满；

(2)少用一个存储空间。

(3)设计一个计数器，统计队列中得元素个数。

* 设计一个布尔变量以判断队列的空和满；

添加一个标志位。设标志位为tag，初始时置tag=0；每当入队列操作成功就置tag=1；每当出队列操作成功就置tag=0。则队列空的判断条件为：

rear == front && tag==0

队列满的判断条件为：

rear = = front && tag= =1

* 少用一个存储空间。

当少用一个存储空间时，以队尾rear加1等于队头 front为队列满的判断条件，即队列满的判断条件此时为：

(rear + 1) % maxSize == front

队列空的判断条件仍然为：

rear = = front

* 设计一个计数器，统计队列中得元素个数。

添加一个计数器。设计数器为count，初始时置count=0；每当入队列操作成功就使count加1；每当出队列操作成功就使count减1。这样，该计数器不仅具有计数功能，而且还具有像标志位一样的标志作用，则此时队列空的判断条件为：

count == 0

队列满的判断条件为：

count > 0 && rear == front

* 顺序循环队列的实现

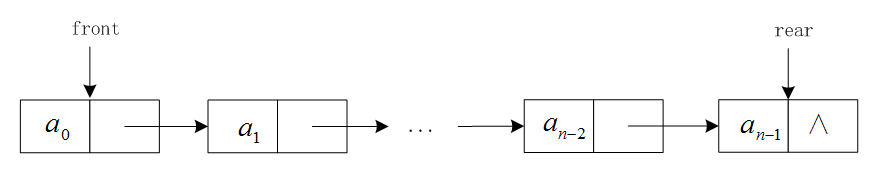
9.5.消息队列应用

* 实例：使用顺序循环队列和多线程实现一个排队买票的例子。
* 生产者（等候买票）
* 消费者 (买票离开)

10.链式队列以及优先级队列应用

10.1.链式队列

* 链式存储结构的队列称作链式队列。
* 链式队列的存储结构如下图所示



* 链式队列的实现

10.2.链式队列的应用

* 编写一个判断一个字符串是否是回文的算法。
* 思路：设字符数组str中存放了要判断的字符串。把字符数组中的字符逐个分别存入一个队列和栈中，然后逐个出队和出栈比较出队的字符与出栈的字符是否相同，若全部相等则该字符串为回文。

10.3.优先级队列

* 优先级队列是带有优先级的队列。

用顺序存储结构实现的优先级队列称作顺序优先级队列。

用链式存储结构存储的优先级队列称作链式优先级队列。

10.4.顺序优先级队列

* 顺序优先级队列和顺序循环队列相比主要有两点不同：

（1）对于顺序优先级队列来说，出队列操作不是把队头数据元素出队列，而是把队列中优先级最高的数据元素出队列。

（2）对于顺序优先级队列来说，数据元素由两部分组成，一部分是原先意义上的数据元素，另一部分是优先级。通常设计优先级为int类型的数值，并规定数值越小优先级越高。

10.5.顺序优先级队列的实现

* 设计顺序优先级队列（分为两个类）
* 数据元素类
* 优先级队列类

10.6.优先级队列的应用

* 设计一个程序模仿操作系统的进程管理问题。进程服务按优先级高的先服务，优先级相同的先到先服务的原则管理。
* 模仿数据包含两个部分：进程编号和优先级。如下有五个进程：

1 30

2 20

3 40

4 20

5 0 ----------优先级最高，先服务

11.串的基本概念与串存储结构

11.1.串基本概念

* 串（也称作字符串）是由n（n≥0）个字符组成的有限序列。
* 一个串中任意个连续的字符组成的子序列称为该串的子串。 包含子串的串称为该子串的主串。
* 一个字符在一个串中的位置序号（为大于等于0的正整数）称为该字符在串中的位置。当且仅当这两个串的值完全相等时，称这两个串相等。

11.2.串抽象数据类型

* 数据集合：串的数据集合可以表示为字符序列s0, s1,… , sn-1，每个数据元素的数据类型为字符类型。
* 操作集合：

（1）取字符charAt(index) ：取index下标的字符返回。

（2）求长度length()：返回串的长度。

（3）比较compareTo(anotherString)：比较当前对象串和串anotherString的Unicode码值的大小。

（4）取子串substring(beginIndex, endIndex)：取当前对象串中从beginIndex下标开始、至endIndex下标的前一下标止的子串。

（5）连接concat(str)：把串str连接到当前对象串的末尾。

（6）插入子串insert(str, pos)：在当前对象串的第pos个字符前插入子串str。

（7）删除子串delete(beginIndex, endIndex)：删除当前对象串中从beginIndex下标开始、至endIndex下标的前一下标止的子串 。

（8）输出串值myPrint()：输出当前对象的串值。

//建议重写Object类的toString()方法。使用toString()代替

（9）查找子串index(subStr, start)：在当前对象串的start下标开始，查找是否存在子串subStr。

11.3.串存储结构

* 串的顺序存储结构

串的顺序存储结构就是用字符类型数组存放串的所有字符。表示串的长度通常有两种方法：

（1）设置一个串的长度参数。

（2）在串值的末尾添加结束标记。

* 串值长度的第一种表示方法下，串的成员变量应包括如下两项：

char[] value;

int count;

其中，value为存储串值的字符类型数组名，count表示串值的长度。

* 串的链式存储结构
* 串的链式存储结构就是把串值分别存放在构成链表的若干个结点的数据元素域上。 有单字符结点链和块链两种。
* 单字符结点链就是每个结点的数据元素域只包括一个字符。
* 块链就是每个结点的数据元素域包括若干个字符。

11.4.MyString类实现

设计MyString类

12.MyString与MyStringBuffer实现

12.1.MyString类

* MyString类实现

12.2.MyStringBuffer类

* MyStringBuffer与MyString的不同之处是：对于MyString类的连接，插入和删除子串成员函数都是不改变原对象的串值。但对于MyStringBuffer类，连接，插入和删除子串的成员函数都改变了原对象的串值。
* MyStringBuffer类实现

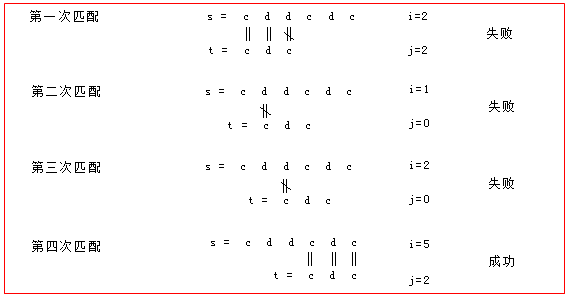
13.Brute-Force算法与KMP算法

13.1.串的模式匹配操作

* 在字符串匹配问题中，我们期待察看S串中是否含有串T（模式串）。其中串S被称为主串，串T被称为子串。
* 如果在主串中查找到子串，则称为模式匹配成功，返回模式串的第一个字符在主串中出现的位置。
* 如果在主串中未找到子串，则称为模式匹配失败，返回-1。
* Brute-Force与KMP算法是两种最经典的模式匹配算法。

13.2.Brute-Force算法

* 也称简单匹配算法，其基本思路是：从目标串s=”s0s1…sn-1”的第一个字符开始和模式串t=”t0t1…tm-1”中的第一个字符比较，若相等，则继续逐个比较后续字符，否则，从目标串s的第2个字符开始重新与模式串t的第一个字符进行比较，依次类推，若从目标串s的第i个字符开始，每个字符依次和模式串t中的对应字符相等，则匹配成功，该算法返回i;否则匹配失败，返回-1。
* **设主串s=“cddcdc”,模式串t=“cdc”,模式匹配过程如图：**

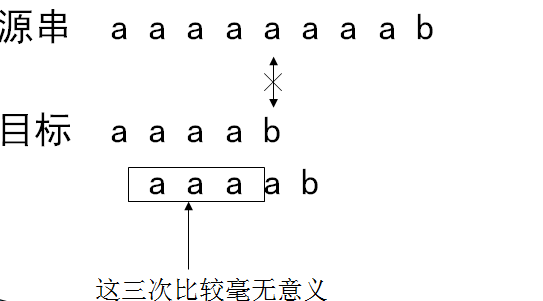


|  |
| --- |
| public static int bruteforce(String source, String sub) {  int j = 0, i = 0,index=-1;  while (i < source.length() && j < sub.length()) {  if (source.charAt(i) == sub.charAt(j))  {  i++; j++;  } else {  // 使i回退到下一个字符，应为子串的前面j向可能匹配成功，而第j+1项失败，所以 i=i-j+1  i = i - j + 1;  j = 0;  }  }  if (j == sub.length()) {  index = i - sub.length();  } else {  index = -1;  }  return index;  } |

* Brute-Force算法总结 ：

该方法的优点是：算法简单明朗，便于实现记忆。

缺点是：进行了回溯，效率不高，而这些都是没有必要的。比如下图：



13.3.KMP算法

* Brute-Force被称为简单模式匹配算法 。KMP算法是它的改进算法。
* 一种改进的字符串匹配算法，由D.E.Knuth与V.R.Pratt和J.H.Morris同时发现，因此称之为KMP算法。此算法可以在O(n+m)的时间数量级上完成串的模式匹配操作，其基本思想是：每当匹配过程中出现字符串比较不等时，不需回溯指针，而是利用已经得到的“部分匹配”结果将模式向右“滑动”尽可能远的一段距离，继续进行比较。
* 模式串求最大真子串

**Max{ k | 0<k<j 且‘p0p1……pk-1’= ‘pj-kpj-k+1……pj-1’ } 当此集合不为空时**

next[j]=

**-1 当j=0时；**

**0 其他情况**

* 例如：求 aaaab next[j]。

模式 a a a a b

j 0 1 2 3 4

next[j] -1 0 1 2 3

* 例如：求abcabcaaa,next[j]。

模式 a b c a b c a a a

j 0 1 2 3 4 5 6 7 8

next[j] -1 0 0 0 1 2 3 4 1

* 计算模式串的next[j]
* 当Si≠Tj,若模式串存在最大真子串，可将模式串T按照k=next[j]的值向右滑动，然后比较Si和Tk，若仍有Si≠Tk,则模式串T按照新的k=next[j]的值向右滑动后比较。这样的过程一直进行到k=next[k]=0,此时若Si≠T0,则模式串T不再向右滑动，随后比较Si+1和T0。
* 例如：主串为"aaaaaaab",子串为"aaaab",求采用KMP的模式匹配过程。

模式 a a a a b

j 0 1 2 3 4

next[j] -1 0 1 2 3

i 0 1 2 3 4 5 6 7

S： a a a a a a a b

j 0 1 2 3 4

T： a a a a b

a a a a b

a a a a b

a a a a b

* 实现KMP算法

14.对象数组以及MyVector类实现

14.1.数组

* 数组是n（n≥1）个相同数据类型的数据元素a0,a1,a2,...,an-1构成的占用一块地址连续的内存单元的有限集合。

14.2.数组的实现

* 数组通常以字节为内部计数单位。对一个有n个数据元素的一维数组，设a0是下标为0的数组元素，Loc(a0)是a0的内存单元地址，k是每个数据元素所需的字节个数，则数组中任一数据元素ai的内存单元地址Loc(ai)可由下面公式求出：

Loc(ai) = Loc(a0) + i × k ( 0≤i＜n )

* 对一个m行n列的二维数组，设a00是行下标和列下标均为0的数组元素，Loc(a00)是a00的存储地址，k是每个数据元素所需的字节个数，则数组中任一数据元素aij的内存单元地址Loc(aij)可由下面公式求出：

Loc(aij) = Loc(a00) + (i × n + j) × k

( 0≤i＜m, 0≤j＜n）

14.3.二维数组的顺序存储结构

图6-2%20%20二维数组的顺序存储结构

14.4数组抽象数据类型

* 数据集合：数组的数据集合可以表示为a0, a1, a2, ..., an-1，且限定数组元素必须存储在地址连续的内存单元中。
* 操作集合：

（1）分配内存空间acclocate()：为数组分配用户所需的内存空间。

（2）取数组长度getLength()：取数组的长度。

（3）存数组元素set(i, x)：把数据元素x存入下标为i的数组中。其约束条件为：0≤i≤getLength()-1。

（4）取数组元素get(i)：取出数组中下标为i的数据元素。其约束条件为：0≤i≤getLength()-1。

14.5Java语言支持的数组功能

* 基本数据类型的数组

由于数组是非常基础的程序设计语言要素，所以Java语言设计实现了数组功能。Java语言（以及大部分高级程序设计语言）支持的数组操作有：

（1）分配内存空间

（2）获得数组长度

（3）存数组元素

（4）取数组元素

* 对象数组

除了可以定义基本数据类型的数组外，Java语言还可以定义对象数组。



14.6.向量类

* Java语言只直接支持上述基本的数组操作。如果程序开始时定义的数组长度为10，且数组中已经存放了若干数据元素，要在程序运行过程中扩充数组长度为20，且把数组中原先存放的数据元素原样保存，则系统不提供直接支持，需要应用程序自己实现。
* 为了扩充数组功能，Java类库还定义了Vector类。要说明的是，国内的大部分教材和科技书籍都把Vector类翻译为向量类，但这里的向量和数学上的向量概念完全不同。
* 向量类Vector扩充了数组的功能，提供了自动扩充数组长度、且把数组中原先存放的数据元素原样保存的功能。Vector类在java.util包中。

14.7.集合

* 集合（Set）是具有某种相似特性的事物的全体。换一种说法，也可以说，集合是某种具有相同数据类型的数据元素全体。
* 如果一个数据元素x在一个集合A中，则说数据元素x属于集合A；如果一个数据元素x不在一个集合A中，就说数据元素x不属于集合A。
* 如果集合A中的所有数据元素都在集合B中，则说集合B包含集合A。
* 集合的运算主要有三种：两个集合的并A∪B、两个集合的交A∩B、两个集合的差A-B。
* 没有一个数据元素的集合称做空集合。

14.8.集合抽象数据类型

* 数据集合：数据元素集合可以表示为{a0, a1, a2, ..., an-1}，每个数据元素的数据类型可以是任意的类类型。
* 操作集合：

（1）添加add(obj)：在集合中添加数据元素obj。

（2）删除remove(obj)：删除集合中的数据元素obj。

（3）属于contain(obj)：数据元素obj是否属于集合。是则返回true,否则返回false。

（4）包含include(otherSet)：当前对象集合是否包含集合otherSet。是则返回true,否则返回false。

（5）相等eqauls(otherSet)：当前对象集合是否和集合otherSet相等。是则返回true,否则返回false。

（6）数据元素个数size()：返回集合中的数据元素个数。

（7）集合空否isEmpty()：若集合空返回true，否则返回false。

14.9.集合类

* 集合的特点是数据元素无序且不重复。集合类既可以基于向量类来实现，也可以用其他方法实现。常用的另一种实现方法是基于哈希表来实现。
* 基于向量类的集合类实现方法:

1 MyVector类增加的成员函数

2 集合类MySet

3 集合类MySet的测试

15.MySet类实现以及彩票机选算法实现

15.1.MySet类的实现

* MySet类实现

15.2.Java集合框架

* Java集合框架介绍
* List：有序的，可以重复
* Set:无序的，不能重复
* Map：键值对的方式储存

15.3.集合框架的应用

* 编写双色球号码机选生成算法。
* 双色球投注区分为红球号码区和蓝球号码区，红球号码范围为01～33，蓝球号码范围为01～16。双色球每期从33个红球中开出6个号码，从16个蓝球中开出1个号码作为中奖号码，双色球玩法即是竞猜开奖号码的6个红球号码和1个蓝球号码，顺序不限 。
* 编写双色球号码机选生成算法。
* 双色球投注区分为红球号码区和蓝球号码区，红球号码范围为01～33，蓝球号码范围为01～16。双色球每期从33个红球中开出6个号码，从16个蓝球中开出1个号码作为中奖号码，双色球玩法即是竞猜开奖号码的6个红球号码和1个蓝球号码，顺序不限 。

16.矩阵类与对称矩阵的压缩算法

16.1.矩阵类

* 矩阵是工程设计中经常使用的数学工具。
* 矩阵的运算主要有矩阵加、矩阵减、矩阵乘、矩阵转置、矩阵求逆等。
* 矩阵用两维数组处理最为方便。
* 二维数组存储结构。
* 设计矩阵类

16.2.特殊矩阵

* 特殊矩阵是指这样一类矩阵，其中有许多值相同的元素或有许多零元素，且值相同的元素或零元素的分布有一定规律。一般采用二维数组来存储矩阵元素。但是，对于特殊矩阵，可以通过找出矩阵中所有值相同元素的数学映射公式，只存储相同元素的一个副本，从而达到压缩存储数据量的目的。

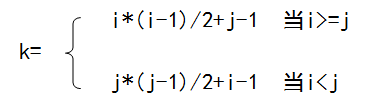
16.3.特殊矩阵的压缩存储

* 只存储相同矩阵元素的一个副本。此种压缩存储方法是：找出特殊矩阵数据元素的分布规律，只存储相同矩阵元素的一个副本。
* n阶对称矩阵的压缩存储对应关系

aij=aji 1<=i<=n,1<=j<=n

元素个数m = n\*(n+1)/2

* 打印对称矩阵第i行,第j列的元素，与一维数组的下标关系为：



* 采用不等长的二维数组
* Java语言支持不等长的二维数组，对于n阶对称矩阵，也可以通过只申请存储下三角（或上三角）矩阵元素所需的二维数组，来达到压缩存储的目的。
* 不等长的二维数组结构



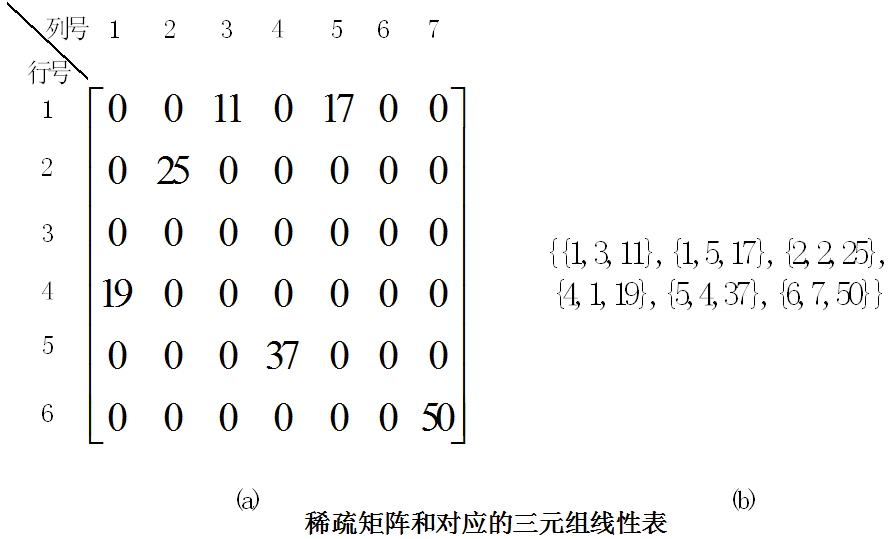
17.稀疏矩阵和三元组稀疏矩阵压缩算法

17.1.稀疏矩阵

* 对一个m×n的矩阵，设s为矩阵元素个数的总和，有s=m\*n，设t为矩阵中非零元素个数的总和，满足t＜＜s的矩阵称作稀疏矩阵。符号“＜＜”读作小于小于。简单说，稀疏矩阵就是非零元素个数远远小于元素个数的矩阵。相对于稀疏矩阵来说，一个不稀疏的矩阵也称作稠密矩阵。

17.2.稀疏矩阵的压缩存储

* 稀疏矩阵的压缩存储方法，是只存储矩阵中的非零元素。
* 稀疏矩阵中每个非零元素及其对应的行下标和列下标构成一个三元组，稀疏矩阵中所有这样的三元组构成一个以三元组为数据元素的线性表。

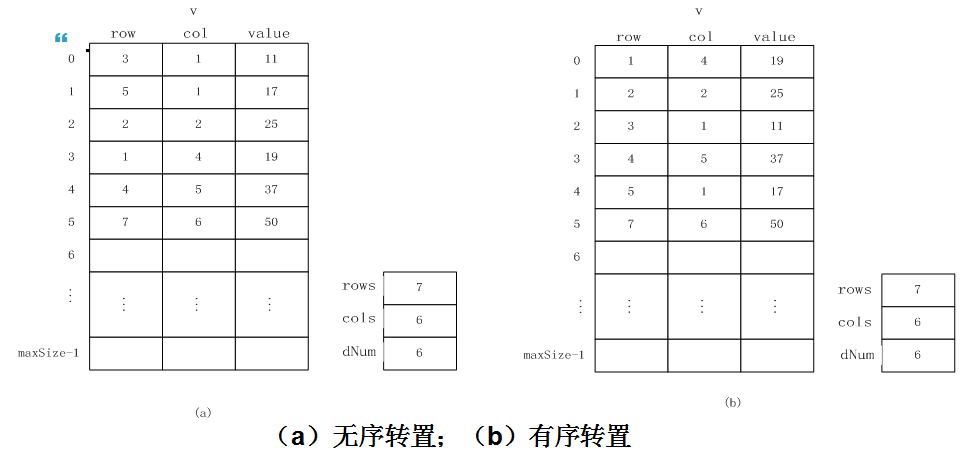


* 稀疏矩阵的压缩存储结构主要有三元组的数组结构存储和三元组的链表结构存储两大类型。三元组的数组结构存储就是把稀疏矩阵的所有三元组按某种规则存储在一个一维数组中。三元组的链表结构存储就是把稀疏矩阵的所有三元组存储在一个链表中。

17.3.数组结构的稀疏矩阵类

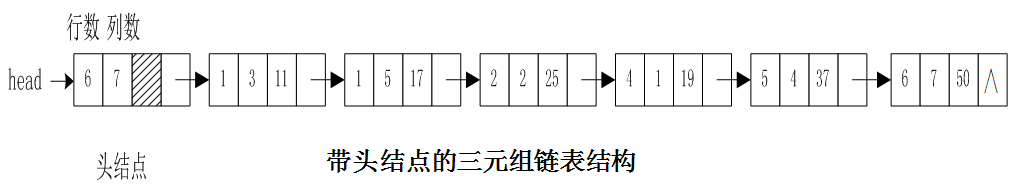
* 三元组的数组结构存储，就是把所有三元组存储在一个数组中。

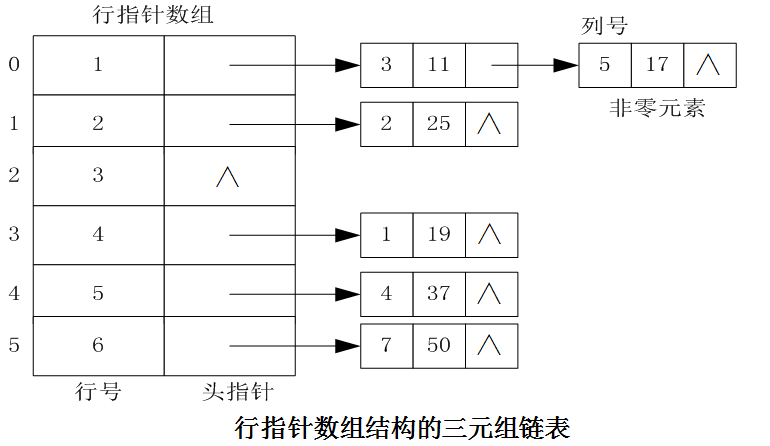




17.4.链式结构稀疏矩阵

* 稀疏矩阵的所有三元组也可采用链表结构存储。用链表存储的稀疏矩阵三元组简称三元组链表。在三元组链表中每个结点的数据域由稀疏矩阵非零元的行号、列号和元素值组成。





18.递归算法与递归算法应用

18.1.递归算法概念

若一个算法直接地或间接地调用自己本身，则称这个算法是递归算法。

问题的定义是递归的

例如：阶乘函数的定义

1 当n=1时

n! =

n\*(n-1)! 当n>1时

18.2.递归算法的设计方法

* 适宜于用递归算法求解的问题的充分必要条件是：

（1）问题具有某种可借用的类同自身的子问题描述的性质

（2）某一有限步的子问题（也称作本原问题）有直接的解存在。

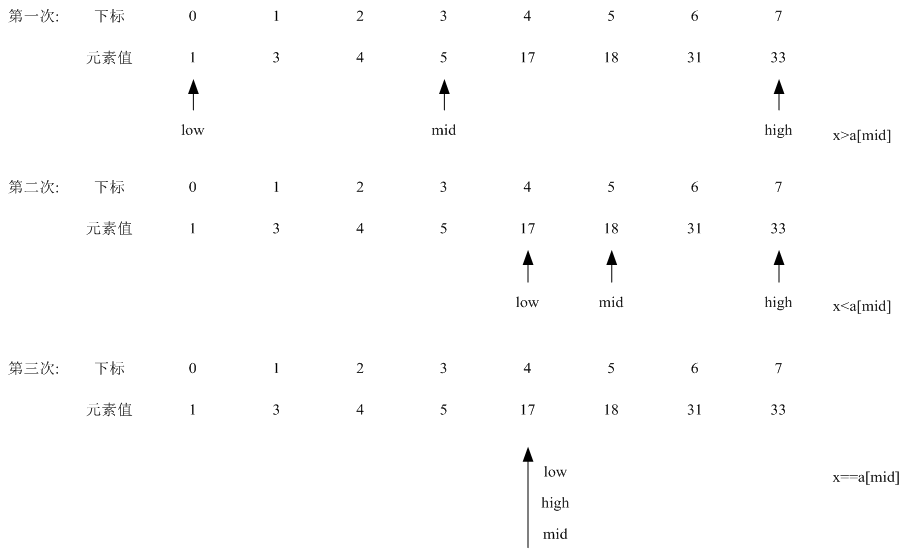
* 当一个问题存在上述两个基本要素时，设计该问题的递归算法的方法是：

（1）把对原问题的求解表示成对子问题求解的形式。

（2）设计递归出口。

18.3.递归算法应用

* 实例1：求n!问题。
* 实例2：设计折半查找递归算法。



* 实例3：求波列纳契数列前N项之和。

从第三项起，每项是前两项之和。

1、1、2、3、5、8、13.....

* 实例4：求两个正整数的最大公约数。

18.4.递归过程和运行时栈

递归函数的执行过程具有三个特点：

（1）函数名相同；

（2）不断地自调用；

（3）最后被调用的函数要最先被返回。

系统用于保存递归函数调用信息的堆栈称作运行时栈。

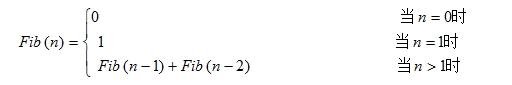
每一层递归调用所需保存的信息构成运行时栈的一个工作记录

栈顶的工作记录保存的是当前调用函数的信息，所以栈顶的工作记录也称为活动记录。

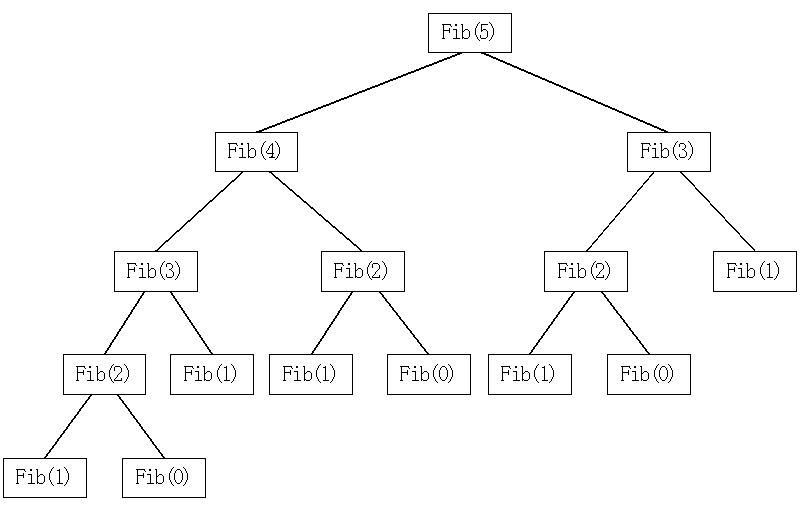
19.递归转换为非递归算法以及迷宫算法

19.1.递归算法的效率分析

* 我们以斐波那契数列递归函数的执行效率为例来讨论递归算法的执行效率问题。
* 斐波那契数列Fib(n)的递推定义是：



* **fib(5)的递归调用树**



19.2.递归算法转换为非递归算法

* 一般来说，如下两种情况的递归算法可转化为非递归算法：

（1）存在不借助堆栈的循环结构的非递归算法，如阶乘计算问题、斐波那契数列的计算问题、折半查找问题等

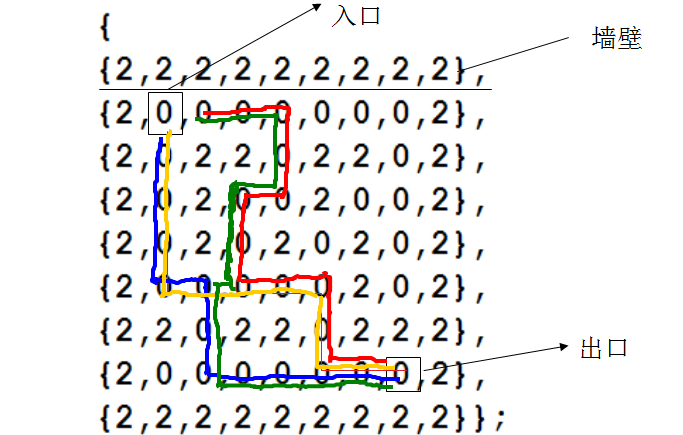
（2）存在借助堆栈的循环结构的非递归算法。所有递归算法都可以借助堆栈转换成循环结构的非递归算法。

* 实例：将求n的阶乘的递归算法转换为使用循环的非递归算法实现。

回溯法及设计举例

* 回溯法的基本思想是：对一个包括有很多结点，每个结点有若干个搜索分支的问题，把原问题分解为对若干个子问题求解的算法。当搜索到某个结点、发现无法再继续搜索下去时，就让搜索过程回溯（即退回）到该结点的前一结点，继续搜索这个结点的其他尚未搜索过的分支；如果发现这个结点也无法再继续搜索下去时，就让搜索过程回溯到这个结点的前一结点继续这样的搜索过程；这样的搜索过程一直进行到搜索到问题的解或搜索完了全部可搜索分支没有解存在为止。

19.3.迷宫问题算法



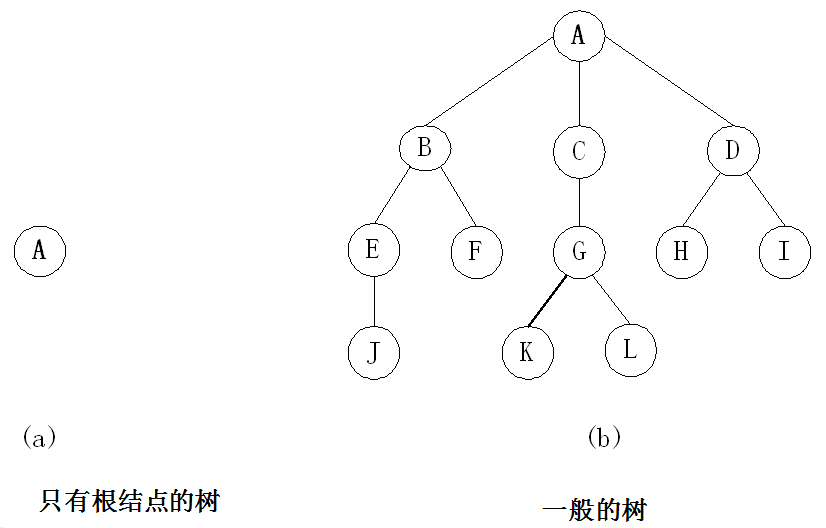
20.二叉树的基本概念以及设计二叉树类

20.1.树的概念

* 树是由n（n≥0）个结点构成的满足以下条件的结点集合：

（1）当n>0时，有一个特殊的结点称为根结点，根结点没有前驱结点；

（2）当n>1时，除根结点外的其他结点被分成m（m>0）个互不相交的集合T1, T2,…, Tm，其中每一个集合Ti（1≤i≤m）本身又是一棵结构和树结构类同的子树。



20.2.树的基本概念

* 结点：结点由数据元素和构造数据元素之间关系的指针组成。
* 结点的度：结点所拥有的子树的个数称为该结点的度。
* 叶结点：度为0的结点称为叶结点，叶结点也称作终端结点。
* 分支结点：度不为0的结点称为分支结点，分支结点也称 作非终端结点。
* 孩子结点：树中一个结点的子树的根结点称作这个结点的孩子结点。
* 双亲结点：若树中某结点有孩子结点，则这个结点就称作它的孩子结点的双亲结点。
* 兄弟结点：具有相同的双亲结点的结点称为兄弟结点。
* 树的度：树中所有结点的度的最大值称为该树的度。
* 结点的层次：从根结点到树中某结点所经路径上的分支数。
* 树的深度：树中所有结点的层次的最大值称为该树的深度。
* 无序树：树中任意一个结点的各孩子结点的排列没有严格次序的树称为无序树。
* 有序树：树中任意一个结点的各孩子结点的排列有严格次序的树称为有序树。
* 森林：m（m≥0）棵树的集合称为森林。

20.3.树的抽象数据类型

* 数据集合 ：树的结点集合，每个结点由数据元素和构造数据元素之间关系的指针组成。

操作集合：

（1）双亲结点parent()

（2）左孩子结点leftChild()

（3）右兄弟结点rightSibling()

（4）遍历树traverse(vs)

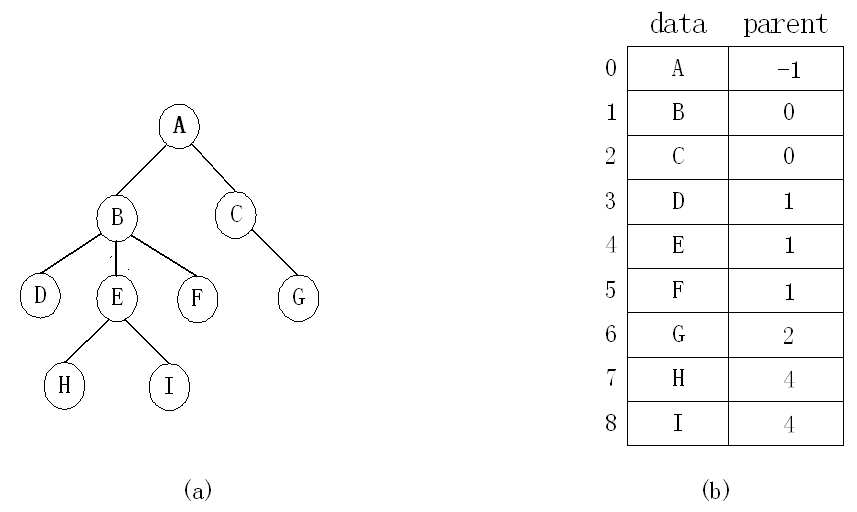
20.4.树的存储结构

* 双亲表示法

双亲表示法就是用指针表示出每个结点的双亲结点。

对于使用仿真指针的双亲表示法来说，每个结点应有两个域，一个是数据元素域，另一个是指示其双亲结点在数组中下标序号的仿真指针域。

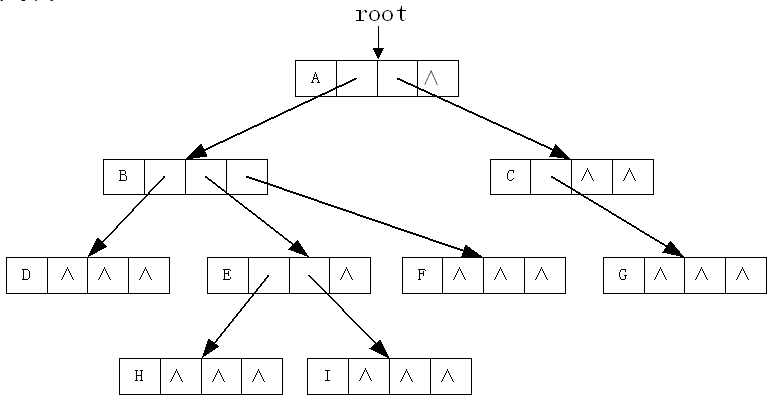
* 树及其使用仿真指针的双亲表示法



* 孩子表示法

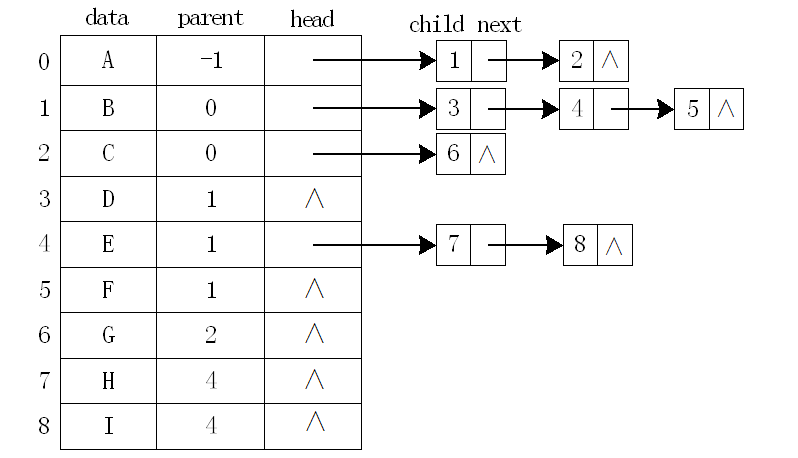
孩子表示法就是用指针表示出每个结点的孩子结点。

常规指针的孩子表示法



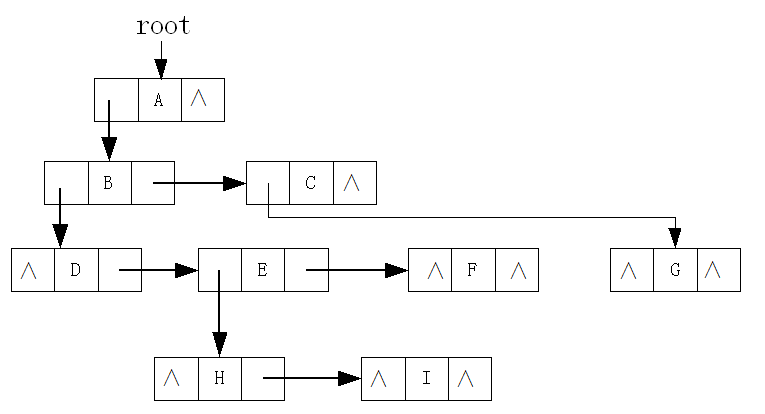
* 双亲孩子表示法

双亲孩子表示法就是用指针既表示出每个结点的双亲结点，也表示出每个结点的孩子结点。



* 孩子兄弟表示法

孩子兄弟表示法就是用指针既表示出每个结点的孩子结点，也表示出每个结点的兄弟结点。



20.5.二叉树

* 二叉树的定义

二叉树是n（n≥0）个结点构成的、每个结点最多只有两个子树的有序树。

满二叉树：在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上。

完全二叉树：如果一棵具有n个结点的二叉树的逻辑结构与满二叉树的前n个结点的逻辑结构相同。

* 两棵不同的二叉树
* 满二叉树和完全二叉树

20.6.二叉树抽象数据类型

* 数据集合：二叉树的结点集合，每个结点由数据元素和构造数据元素之间关系的指针组成。
* 操作集合：

（1）双亲结点parent()：

（2）左孩子结点leftChild()

（3）右孩子结点rightSibling()

（4）左插入结点insertLeftNode(x)

（5）右插入结点insertRightNode(x)：

（6）左删除子树deleteLeftTree()

（7）右删除子树deleteRightTree()

（8）遍历二叉树traverse(vs)

20.7.二叉树的性质

* 性质1 若规定根结点的层次为0，则一棵非空二叉树的第i层上最多有2i（i≥0）个结点。
* 性质2 若规定空二叉树树的深度为-1(即根结点的深度为0)，则深度为k的二叉树的最大结点数是2k+1-1（k≥-1）个。
* 性质3 具有n个结点的完全二叉树的深度k为不超过log2(n+1)-1的最大整数。
* 性质4 对于一棵非空的二叉树，如果叶结点个数为n0，度为2的结点数为n2，则有n0= n2+1。
* 性质5 对于具有n个结点的完全二叉树，如果按照从上至下和从左至右的顺序对所有结点从0开始顺序编号，则对于序号为i的结点，有：

（1）如果i>0，则序号为i结点的双亲结点的序号为 (i-1)/2（“/”表示整除）；如果i=0，则序号为i结点为根结点，无双亲结点。

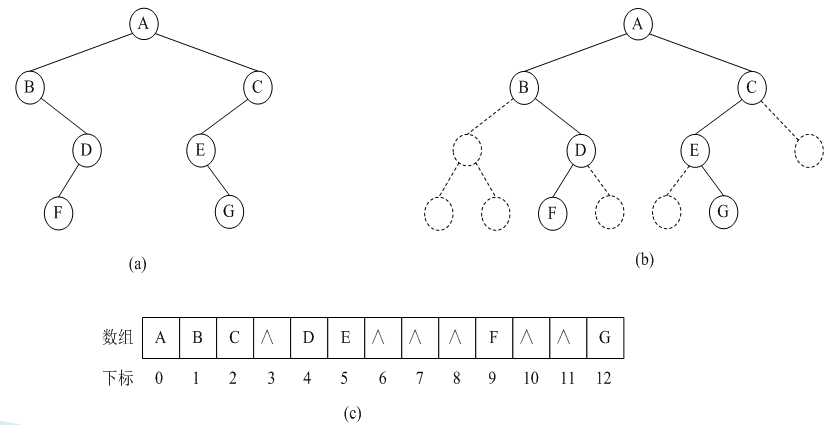
（2）如果2×i+1<n，则序号为i结点的左孩子结点的序号为2×i+1；如果2×i+1≥n，则序号为i结点无左孩子结点。

（3）如果2×i+2<n，则序号为i结点的右孩子结点的序号为2×i+2；如果2×i+2≥n，则序号为i结点无右孩子结点。

20.8.二叉树的存储结构

* 二叉树的顺序存储结构

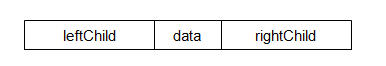
非完全二叉树的顺序存储结构如下图



* 二叉树的链式存储结构

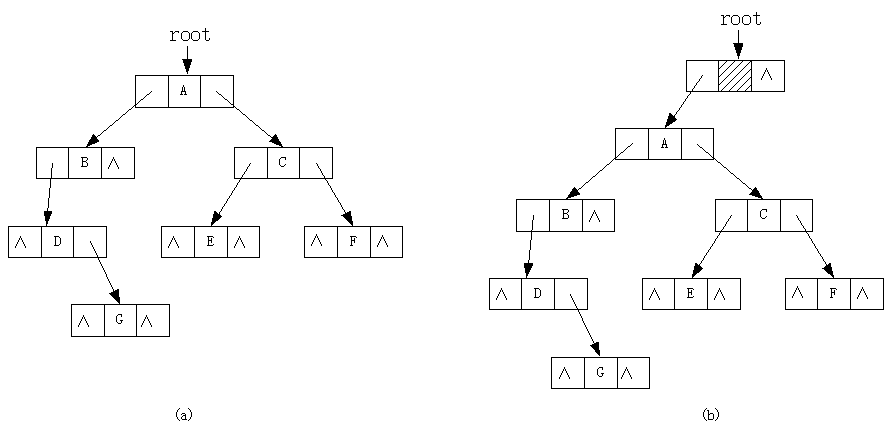
二叉树的链式存储结构是用指针建立二叉树中结点之间的关系。

二叉链存储结构的每个结点包含三个域 。



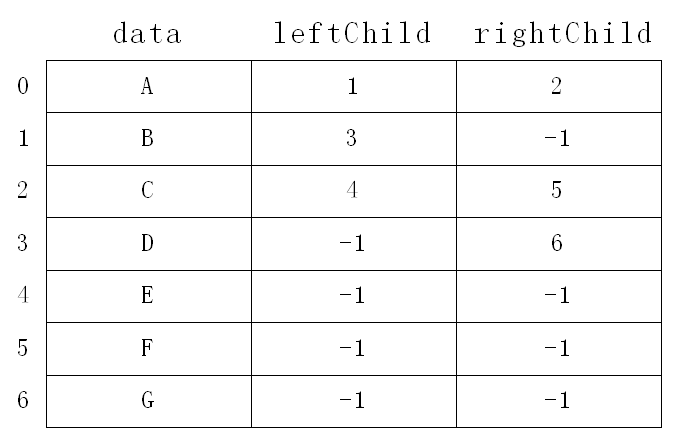
* 二叉链存储结构的二叉树

（a）不带头结点的二叉树 （b）带头结点的二叉树



* 二叉树的仿真指针存储结构

二叉树的仿真指针存储结构是用数组存储二叉树中的结点 。



20.9.以结点类为基础的二叉树设计

* 设计**二叉树结点类。**

21.二叉树的遍历算法

21.1.二叉树的遍历

* 一棵二叉树由三部分组成：根结点、左子树和右子树 。
* 根据遍历算法访问根结点的次序，我们介绍三种遍历算法分别为前序遍历（DLR）、中序遍历（LDR）和后序遍历（LRD）。

21.2.前序遍历

* 前序遍历（DLR）递归算法为：

若二叉树为空则算法结束；否则：

（1）访问根结点；

（2）前序遍历根结点的左子树；

（3）前序遍历根结点的右子树。

21.3.中序遍历

* 中序遍历（LDR）递归算法为：

若二叉树为空则算法结束；否则：

（1）中序遍历根结点的左子树；

（2）访问根结点；

（3）中序遍历根结点的右子树。

21.4.后序遍历

* 后序遍历（LRD）递归算法为：

若二叉树为空则算法结束；否则：

（1）后序遍历根结点的左子树；

（2）后序遍历根结点的右子树；

（3）访问根结点。

除前序、中序和后序遍历算法外，二叉树还有层序遍历。层序遍历的要求是：按二叉树的层序次序（即从根结点层至叶结点层），同一层中按先左子树再右子树的次序遍历二叉树。

21.5.层次遍历

* 二叉树的层序遍历算法如下：

（1）初始化设置一个队列；

（2）把根结点指针入队列；

（3）当队列非空时，循环执行步骤（3.a）到步骤（3.c）；

（3.a）出队列取得当前队头结点，访问该结点；

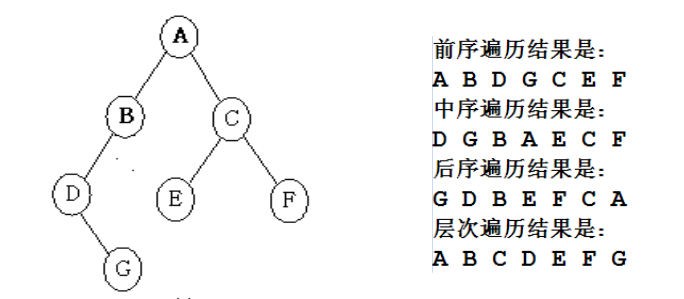
（3.b）若该结点的左孩子结点非空，则将该结点的左孩子结点指针入队列；

（3.c）若该结点的右孩子结点非空，则将该结点的右孩子结点指针入队列；

（4）结束。

21.6.设计遍历类

* 设计Traverse遍历类
* 分别实现前序、中序、后序、层次遍历



21.7.二叉树遍历应用

* 打印二叉树
* 查找数据元素

22.二叉树的游标遍历算法

22.1.非递归的二叉树遍历算法

* 非递归的二叉树前序遍历算法如下：

（1）初始化设置一个堆栈；

（2）把根结点指针入栈；

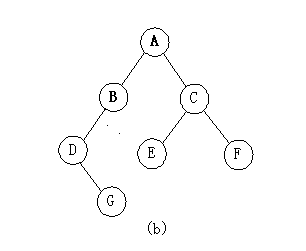
（3）当堆栈非空时，循环执行步骤（3.a）到步骤（3.c）；

（3.a）出栈取得栈顶结点，访问该结点；

（3.b）若该结点的右孩子结点非空，则将该结点的右孩子 结点指针入栈；

（3.c）若该结点的左孩子结点非空，则将该结点的左孩子结点指针入栈；

（4）结束。



对于上图所示的二叉树，非递归的二叉树前序遍历算法的执行过程 如下页图所示：



22.2.二叉树的游标遍历

* 二叉树的游标遍历是指，在规定了一棵二叉树的遍历方法后，每次只访问当前结点的数据元素值，然后使当前结点为当前结点的后继结点，直到到达二叉树的最后一个结点为止。
* 二叉树游标类
* 要满足分步遍历操作的需要，首先要有可分解的遍历算法，并要分解这样的遍历算法为几个子算法。然后再把这样的子算法设计成几个控制遍历过程的成员函数。这样，应用程序就可通过这几个成员函数，方便地对二叉树进行分步遍历操作。本节我们所讨论的就是这样的一种类，我们把它称作二叉树游标类。
* 二叉树中序游标类

非递归的二叉树中序遍历算法如下：

（1）设置一个堆栈并初始化；

（2）使结点对象引用t等于二叉树根指针，如t非空令结束标记为0；否则为1；

（3）当t的左孩子结点不空时循环；否则转向步骤（4）；

（3.1）把t入堆栈；

（3.2）t等于t的左孩子结点；

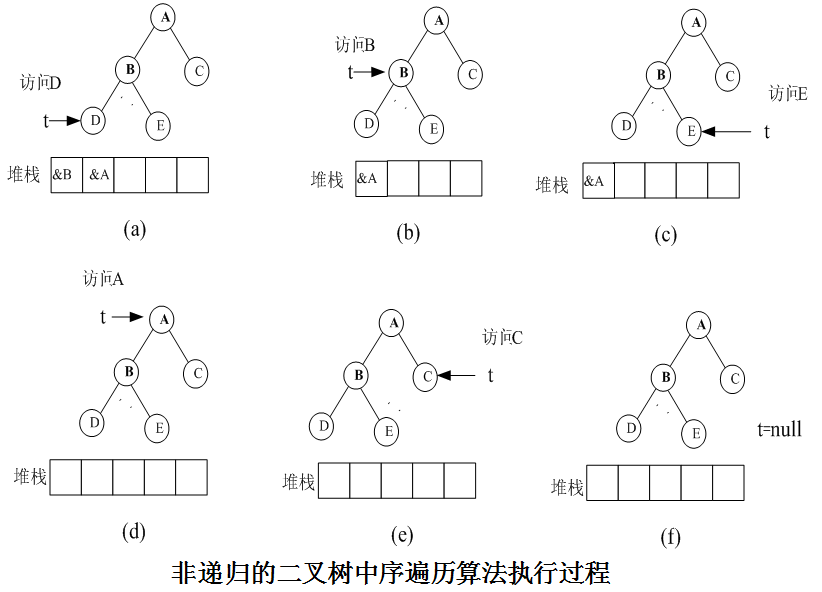
（4）如果t为空则令结束标记为1；

（5）如果结束标记为1转步骤（8）；否则继续执行；

（6）访问t结点；

（7）如果t的右孩子结点非空，则使t等于t的右孩子结点，转到步骤（3）；否则如果堆栈不空，则退栈使t等于栈顶结点，转向步骤（5）；否则令结束标记为1，转向步骤（5）；

（8）算法结束。

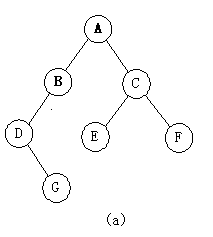


* 二叉树层序游标类
* 二叉树层序遍历的次序是根结点，根结点的左孩子结点，根结点的右孩子结点，根结点的左孩子结点的左孩子结点，根结点的左孩子结点的右孩子结点，如此等等，一直到最下层最右边的结点为止。
* 把它用在分步层序遍历二叉树上，需要把算法过程分解为几个子过程，由reset()成员函数、endOfBiTree()成员函数和next()成员函数来分别完成。
* reset()成员函数完成使当前结点等于根结点 ；
* next()成员函数完成让当前结点等于当前结点的下一个结点。
* 设计MyBiTreeIterator类
* 分别实现二叉树遍历的非递归算法
* 前序遍历
* 中序遍历
* 后序遍历
* 层次遍历

23.线索二叉树算法和翻转二叉树算法

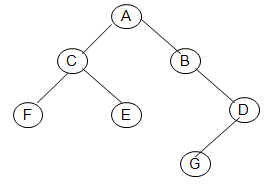
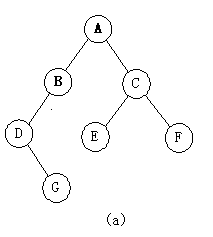
23.1.线索二叉树

* 把结点中指向前驱结点和后继结点的指针称为线索。在二叉树的结点上加上线索的二叉树称作线索二叉树。对二叉树以某种方法（如前序、中序或后序方法）遍历使其变为线索二叉树的过程称作按该方法对二叉树进行的线索化。



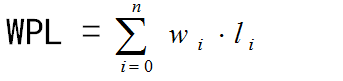
23.2.翻转二叉树

* 翻转二叉树,就是把二叉树所有非叶节点的左右子树位置交换.用递归很容易就可以实现,非递归时,只要利用栈,也不难实现。



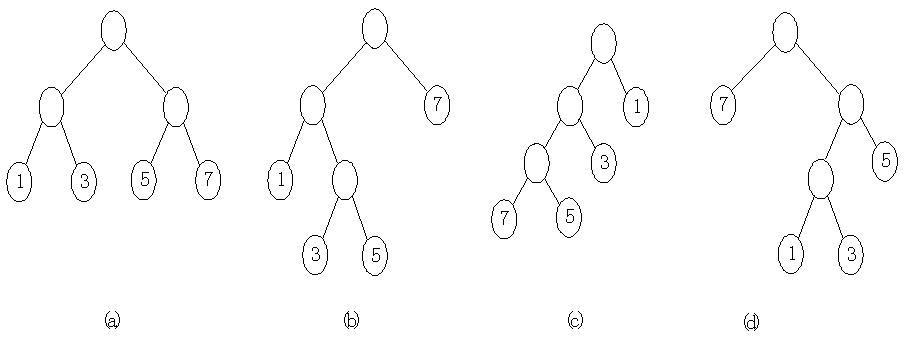
24.哈夫曼树构造算法与哈夫曼编码算法

24.1.哈夫曼树的基本概念

* 在一棵二叉树中，定义从A结点到B结点所经过的分支序列叫做从A结点到B结点的路径。
* 从A结点到B结点所经过的分支个数叫做从A结点到B结点的路径长度
* 从二叉树的根结点到二叉树中所有叶结点的路径长度之和称作该二叉树的路径长度。
* 如果二叉树中的叶结点都带有权值，我们可以把这个定义加以推广。设二叉树有n个带权值的叶结点，定义从二叉树的根结点到二叉树中所有叶结点的路径长度与相应叶结点权值的乘积之和为该二叉树的带权路径长度（WPL），即：
* 
* wi为第i个叶结点的权值，li为从根结点到第i个叶结点的路径长度。

24.2.哈夫曼树

* 具有相同叶结点和不同带权路径长度的二叉树



（a）WPL = 1×2+3×2+5×2+7×2 = 32

（b）WPL = 1×2+3×3+5×3+7×1 = 33

（c）WPL = 7×3+5×3+3×2+1×1 = 43

（d）WPL = 1×3+3×3+5×2+7×1 = 29

24.3.哈夫曼树构造算法

* 根据哈夫曼树的定义，要使一棵二叉树的带权路径长度WPL值最小，必须使权值越大的叶结点越靠近根结点。哈夫曼提出的构造哈夫曼树构造算法为：

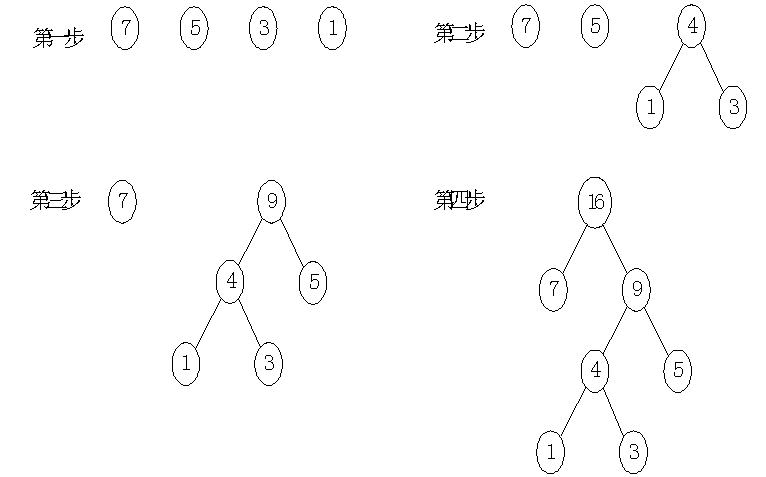
（1）由给定的n个权值{w1,w2,…,wn}构造n棵只有根 结点的二叉树，从而得到一个二叉树森林F={T1,T2,…,Tn}。

（2）在二叉树森林F中选取根结点的权值最小和次小的两棵二叉树作为新的二叉树的左右子树构造新的二叉树，新的二叉树的根结点权值为左右子树根结点权值之和。

（3）在二叉树森林F中删除作为新二叉树左右子树的两棵二叉树，将新二叉树加入到二叉树森林F中。

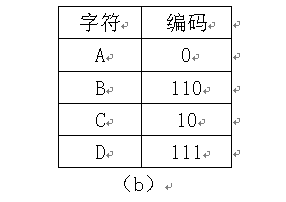
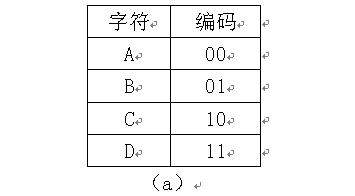
（4）重复步骤（2）和（3），当二叉树森林F中只剩下一棵二叉树时，这棵二叉树就是所构造的哈夫曼树。

对于一组给定的叶结点，设它们的权值集合为{7,5,3,1}，按哈夫曼树构造算法对此集合构造哈夫曼树的过程如图所示。



24.4.哈夫曼编码问题

* 在数据通讯中，经常需要将传送的文字转换为二进制 字符0和1组成的二进制串，我们称这个过程为编码。
* 例如要传送的电文为ABACCDA
* 不同的编码方案



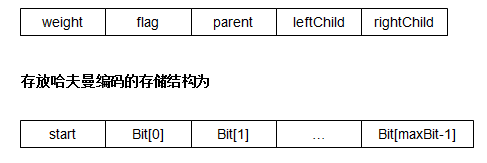
* 哈夫曼树可用于构造代码总长度最短的编码方案。具体构造方法如下：
* 设需要编码的字符集合为{d1,d2,…,dn}，各个字符在电文中出现的次数集合为{w1,w2,…,wn}，以d1,d2,…,dn作为叶结点，以w1,w2,…,wn作为各叶结点的权值构造一棵二叉树，规定哈夫曼树中的左分支为0，右分支为1，则从根结点到每个叶结点所经过的分支对应的0和1组成的序列便为该结点对应字符的编码。这样的代码总长度最短的不等长编码称之为哈夫曼编码。



24.5.哈夫曼编码的软件设计

哈夫曼编码的数据结构设计

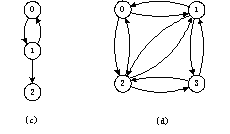
设计哈夫曼树的结点存储结构为双亲孩子存储结构。



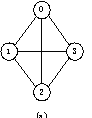
25.图的概念以及图的邻接矩阵类实现

25.1.图的基本概念

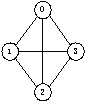
* 图 是由结点集合及结点间的关系集合组成的一种数据结构。
* 结点和边 图中的顶点称作结点，图中的第i个结点记做vi。
* 有向图 在有向图中，结点对＜x ,y＞是有序的，结点对＜x,y＞称为从结点x到结点y的一条有向边，因此，＜x,y＞与＜y,x＞是两条不同的边。有向图中的结点对＜x,y＞用一对尖括号括起来，x是有向边的始点，y是有向边的终点，有向图中的边也称作弧。



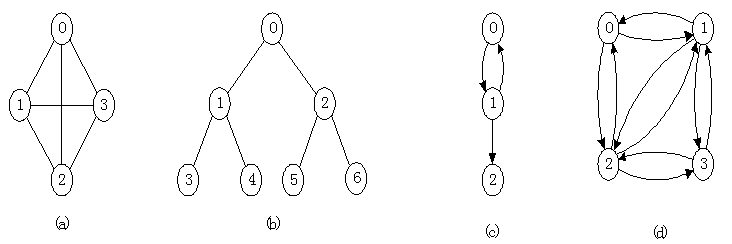
* 无向图 在无向图中，结点对（x,y）是无序的，结点对（x,y）称为与结点x和结点y相关联的一条边。（x,y）等价于＜x,y＞和＜y,x＞。
* 完全图 在有n个结点的无向图中，若有n(n-1)/2条边，即任意两个结点之间有且只有一条边，则称此图为无向完全图。在有n个结点的有向图中，若有n(n-1)条边，即任意两个结点之间有且只有方向相反的两条边，则称此图为有向完全图。



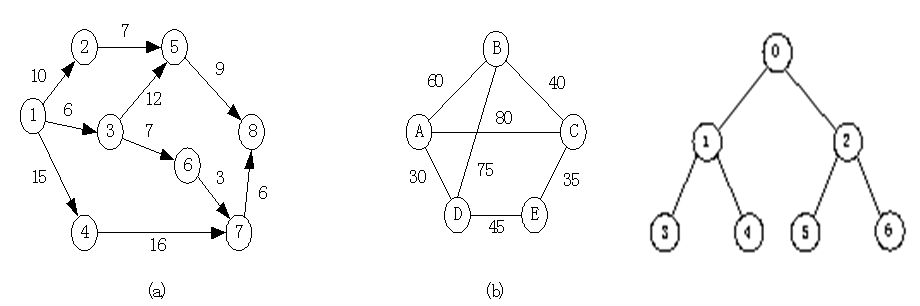
* 邻接结点 在无向图G中，若（u,v）是E(G)中的一条边，则称u和v互为邻接结点，并称边（u,v）依附于结点u和v。在有向图G中，若＜u,v＞是E(G)中的一条边，则称结点u邻接到结点v，结点v邻接自结点u，并称边＜u,v＞和结点u和结点v相关联。



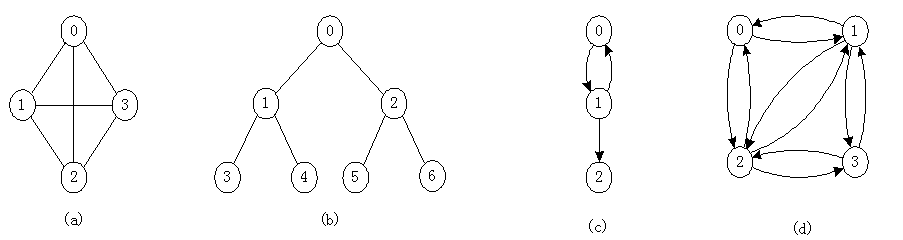
* 结点的度 结点v的度是与它相关联的边的条数，记作TD(v)。
* 路径 在图G=(V,E)中，若从结点vi出发有一组边使可到达结点vj，则称结点vi到结点vj的结点序列为从结点vi到结点vj的路径



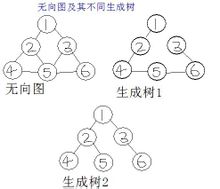
* 权 有些图的边附带有数据信息，这些附带的数据信息称为权。第i条边的权用符号wi表示。
* 路径长度 对于不带权的图，一条路径的路径长度是指该路径上的边的条数；对于带权的图，一条路径的路径长度是指该路径上各个边权值的总和。



* 子图 设有图G1={V1,E1}和图G2={V2,E2}，若V2V1且E2E1，则称图G2是图G1的子图。
* 连通图和强连通图 在无向图中，若从结点vi到结点vj有路径，则称结点vi和结点vj是连通的。如果图中任意一对结点都是连通的，则称该图是连通图。在有向图中，若对于任意一对结点vi和结点vj（vi≠vj）都存在路径，则称图G是强连通图。



* 最小生成树 一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图联通的最少的边。（n-1）条边。



25.2.图的抽象数据类型

* 数据集合：由一组结点集合{vi}和一组边{ej}集合组成。当为带权图时每条边上权wj还构成权集合{wj}。

操作集合：

（1）初始化initiate(n)：

（2）插入结点 insertVertex(vertex)：

（3）插入边insertEdge(v1, v2, weight)：

（4）删除边deleteEdge(v1, v2)：

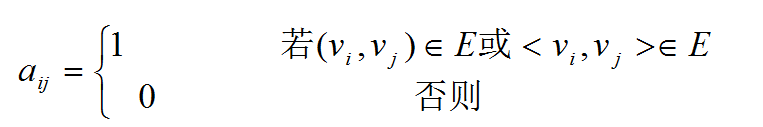
（5）删除结点deleteVertex(vertex)：

（6）第一个邻接结点getFirstVex(v)：

（7）下一个邻接结点getNextVex(int v1, v2)：

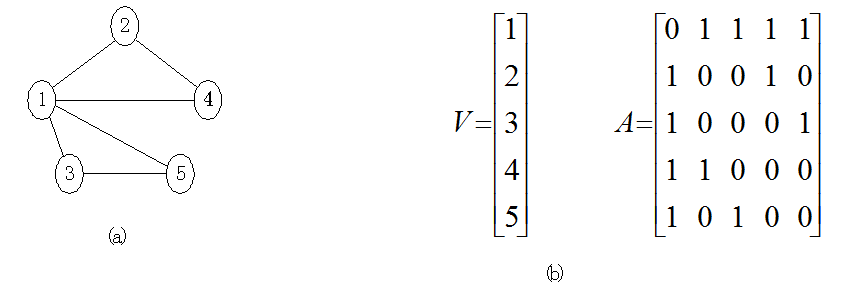
（8）遍历depthFirstSearch(vs)：

25.3.图的存储结构

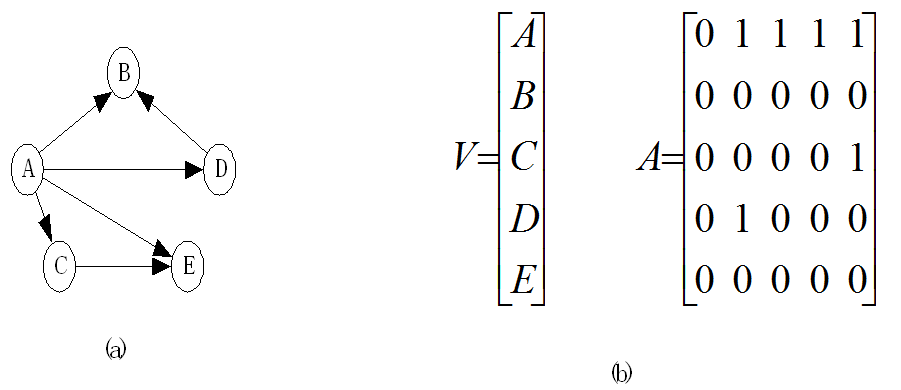
* 图的邻接矩阵存储结构
* 假设图G=(V,E)有n个结点，即V={v0,v1,…,vn-1}，E可用如下形式的矩阵A描述，对于A中的每一个元素aij，满足：
* 
* 由于矩阵A中的元素aij表示了结点vi和结点vj之间边的关系，或者说，A中的元素aij表示了结点vi和结点vj（0≤j≤n-1）的邻接关系，所以矩阵A称作邻接矩阵。

25.4.图的邻接矩阵存储结构

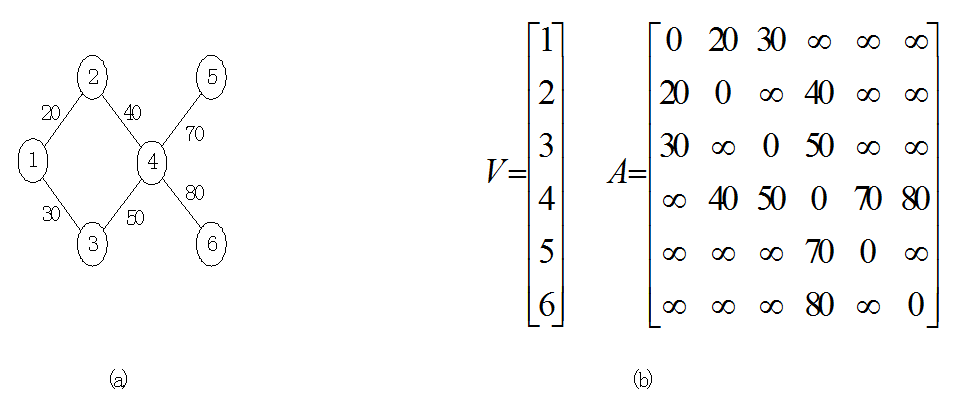
* 无向图及其邻接矩阵



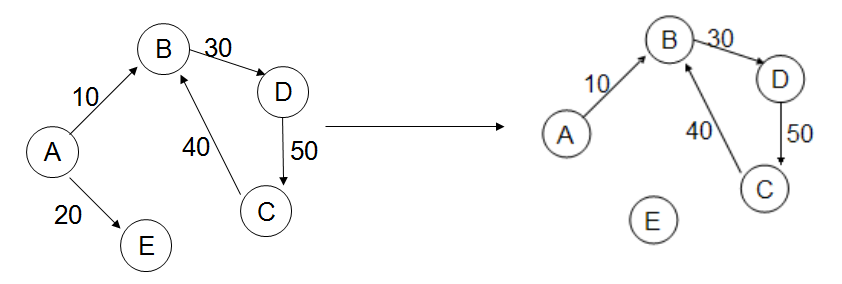
* 有向图及其邻接矩阵



* 带权图及其邻接矩阵



* 邻接矩阵图类设计



26.图的广度和深度优先遍历算法

26.1.图的遍历

* 图的遍历算法设计需要考虑三个问题：

（1）图的特点是没有首尾之分，所以算法的参数要指定访问的第一个结点；

（2）对图的遍历路径有可能构成一个回路，从而造成死循环，所以算法设计要考虑遍历路径可能出现的死循环问题；

（3）一个结点可能和若干个结点都是邻接结点，要使一个结点的所有邻接结点按照某种次序被访问。

26.2.图的深度优先遍历算法

* 连通图的深度优先遍历递归算法为：

（1）访问结点v并标记结点v为已访问；

（2）查找结点v的第一个邻接结点w；

（3）若结点v的邻接结点w存在，则继续执行，否则算法结束；

（4）若结点w尚未被访问则深度优先搜索递归访问结点w；

（5）查找结点v的w邻接结点的下一个邻接结点w，转到步骤（3）。

26.3.图的广度优先遍历算法

* 连通图的广度优先遍历算法为：

（1）访问初始结点v并标记结点v为已访问；

（2）结点v入队列；

（3）当队列非空时则继续执行，否则算法结束；

（4）出队列取得队头结点u；

（5）查找结点u的第一个邻接结点w；

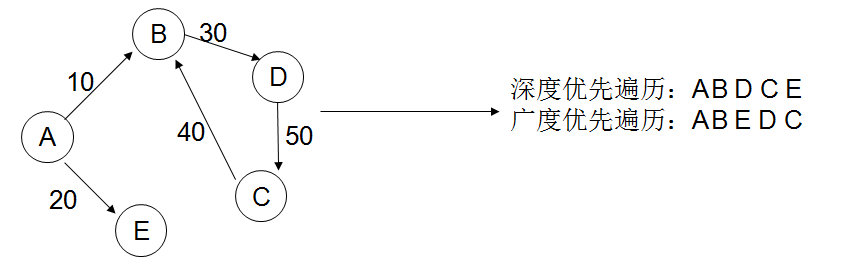
（6）若结点u的邻接结点w不存在，则转到步骤（3），否则循环执行，

（6.1）若结点w尚未被访问，则访问结点w，并标记结点w为已访问；

（6.2）结点w入队列；

（6.3）查找结点u的w邻接结点后的下一个邻接结点w，转到步骤（6）。

26.4.图的遍历



26.5.遍历算法设计

|  |
| --- |
| private void depthFirstSearch(int v, boolean[] visited, Visit vs) throws Exception{  //连通图以v为初始结点序号、访问操作为vs的深度优先遍历  //数组visited标记了相应结点是否已访问过，0表示未访问，1表示已访问  vs.print(getValue(v)); //访问该结点  visited[v] = true; //置已访问标记  int w = getFirstNeighbor(v);//取第一个邻接结点  while(w != -1)  {//当邻接结点存在时循环  if(!visited[w])//如果没有访问过  {  depthFirstSearch(w, visited, vs); //以w为初始结点递归遍历  }  w = getNextNeighbor(v, w);//取下一个邻接结点  }  }  private void broadFirstSearch(int v, boolean[] visited, Visit vs) throws Exception{  //数组visited标记了相应结点是否已访问过，0表示未访问，1表示已访问  int u, w;  Queue queue = new LinkedList();//创建顺序队列queue  vs.print(getValue(v));//访问结点v  visited[v] = true; //置已访问标记  queue.add(new Integer(v)); //结点v入队列  while(! queue.isEmpty())  {//队列非空时循环  u = ((Integer)queue.remove()).intValue();//出队列  w = getFirstNeighbor(u);//取结点u的第一个邻接结点  while(w != - 1)  { //当邻接结点存在时循环  if(! visited[w])  { //若该结点没有访问过  vs.print(getValue(w));//访问结点w  visited[w] = true;//置已访问标记  queue.add(new Integer(w)); //结点w入队列  }//取结点u的邻接结点w的下一个邻接结点  w = getNextNeighbor(u, w);  }  }  } |

26.6.非连通图的遍历

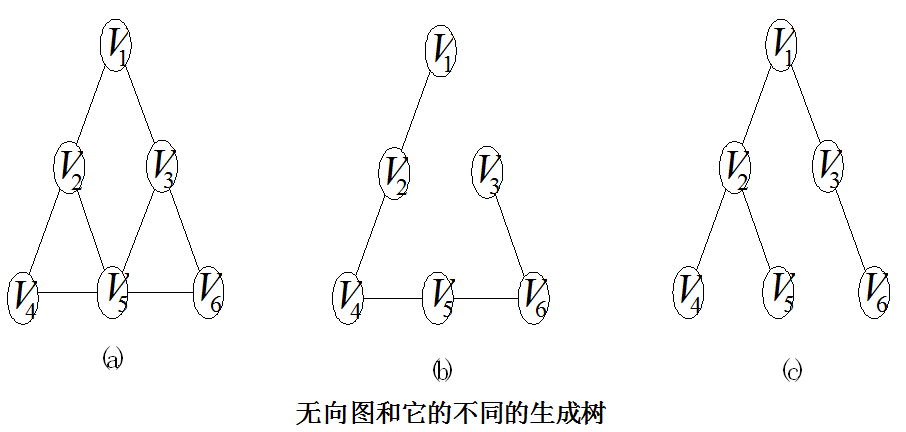
* 对于连通图，从图的任意一个结点开始深度或广度优先遍历，一定可以访问图中的所有结点。但对于非连通图，从图的任意一个结点开始深度或广度优先遍历，并不能访问图中的所有结点。对于非连通图，从图的任意一个结点开始深度或广度优先遍历只能访问和初始结点连通的所有结点。

27.最小生成树及特里姆和克鲁斯卡尔算法

27.1.最小生成树

* 一个有n个结点的连通图的生成树是原图的极小连通子图，它包含原图中的所有n个结点，并且有保持图连通的最少的边。

如果无向连通图是一个带权图，那么它的所有生成树中必有一棵边的权值总和最小的生成树，我们称这棵生成树为最小代价生成树，简称最小生成树。



* 从最小生成树的定义可知，构造有n个结点的无向连通带权图的最小生成树,必须满足以下三条：

（1）构造的最小生成树必须包括n个结点；

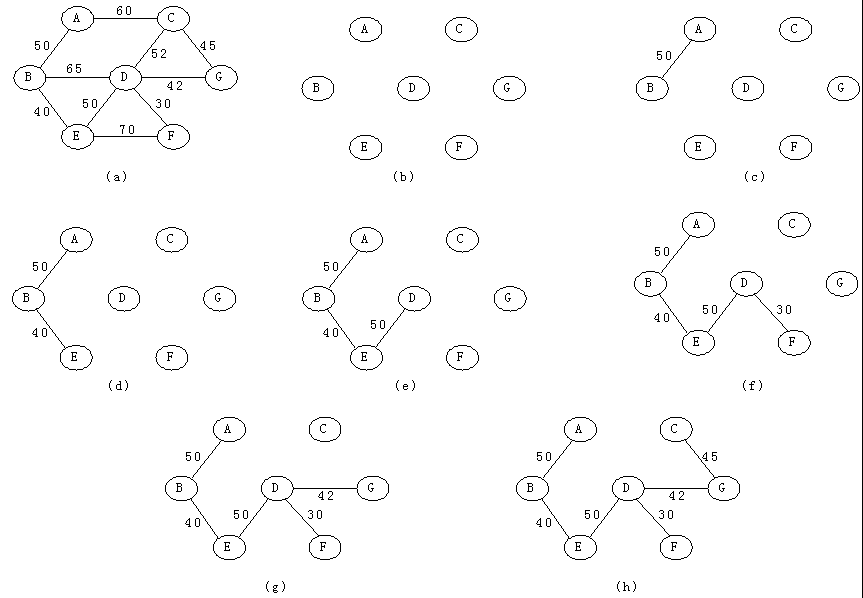
（2）构造的最小生成树中有且只有n-1条边；

（3）构造的最小生成树中不存在回路。

构造最小生成树的方法有许多种，典型的构造方法有两种，一种称作普里姆（Prim）算法，另一种称作克鲁斯卡尔（Kruskal）算法。

27.2.普里姆算法

* 假设G=(V,E)为一个带权图，其中V为带权图中结点的集合，E为带权图中边的权值集合。设置两个新的集合U和T，其中U用于存放带权图G的最小生成树的结点的集合，T用于存放带权图G的最小生成树的权值的集合。
* 普里姆算法思想是：令集合U的初值为U={u0}（即假设构造最小生成树时从结点u0开始），集合T的初值为T={}。从所有结点u∈U和结点v∈V-U的带权边中选出具有最小权值的边(u,v)，将结点v加入集合U中，将边(u,v) 加入集合T中。如此不断重复，当U=V时则最小生成树构造完毕。此时集合U中存放着最小生成树结点的集合，集合T中存放着最小生成树边的权值集合。



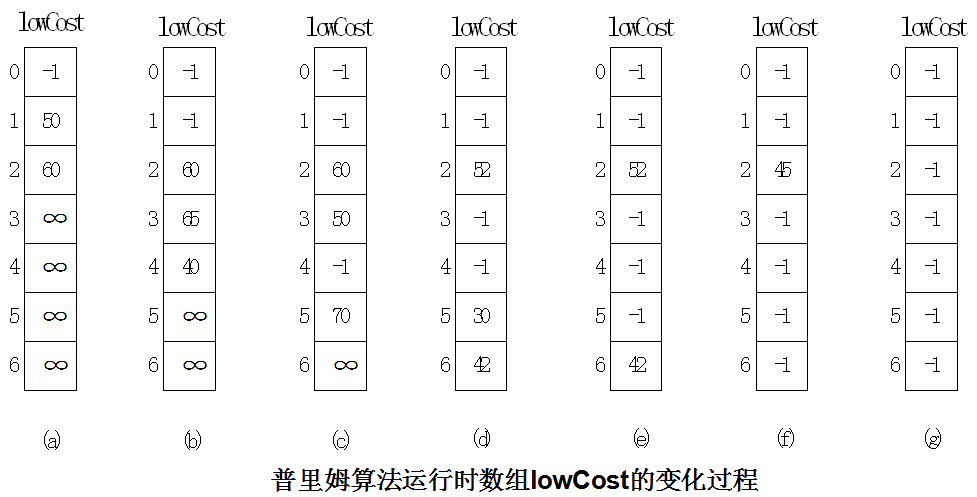
27.3.普里姆函数设计

* 这里我们令当弧头结点等于弧尾结点时权值等于0。
* 函数的参数设计：

普里姆函数应有两个参数：

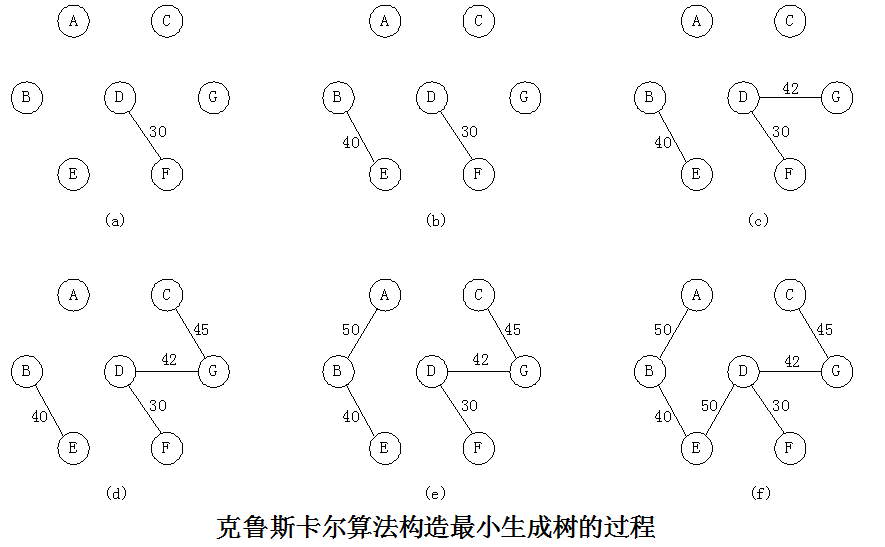
一个参数是图g，这里图g定义为邻接矩阵存储结构图类的对象；

另一个参数是通过函数得到的最小生成树的结点数据和相应结点的边的权值数据closeVertex。



27.4.克鲁斯卡尔算法

* 克鲁斯卡尔算法是：设无向连通带权图G=(V,E)，其中V为结点的集合，E为边的集合。设带权图G的最小生成树T由结点集合和边的集合构成，其初值为T=(V,{})，即初始时最小生成树T只由带权图G中的结点集合组成，各结点之间没有一条边。这样，最小生成树T中的各个结点各自构成一个连通分量。然后，按照边的权值递增的顺序考察带权图G中的边集合E中的各条边。若被考察的边的两个结点属于T的两个不同的连通分量，则将此边加入到最小生成树T，同时把两个连通分量连接为一个连通分量；若被考察的边的两个结点属于T的同一个连通分量，则将此边舍去。如此下去，当T中的连通分量个数为1时，T中的该连通分量即为带权图G的一棵最小生成树。



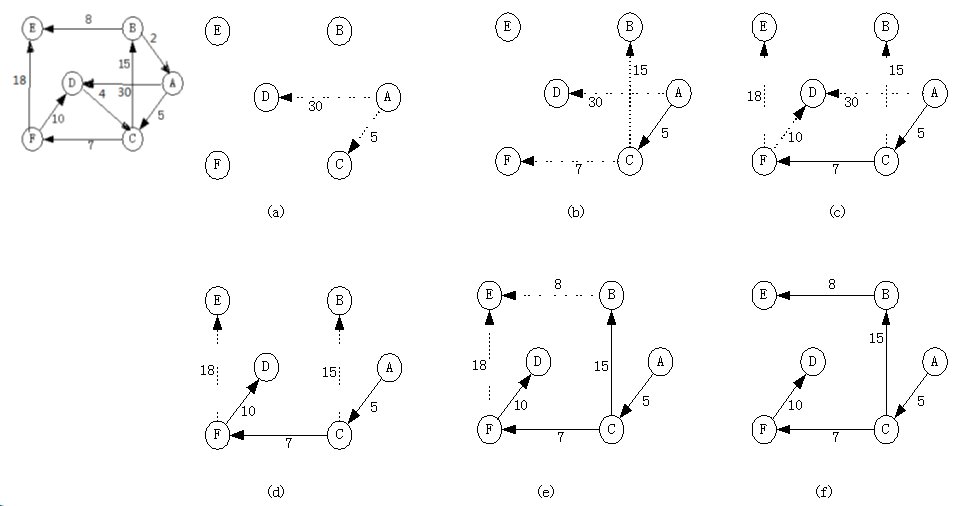
28.最短路径与狄克斯特拉算法与佛洛依德算法

28.1.最短路径

* 在一个图中，若从一个结点到另一个结点存在着路径，定义路径长度为一条路径上所经过的边的数目。图中从一个结点到另一个结点可能存在着多条路径，我们把路径长度最短的那条路径叫做最短路径，其路径长度叫做最短路径长度或最短距离.
* 在一个带权图中，若从一个结点到另一个结点存在着一条路径，则称该路径上所经过边的权值之和为该路径上的带权路径长度。带权图中从一个结点到另一个结点可能存在着多条路径，我们把带权路径长度值最小的那条路径也叫做最短路径，其带权路径长度也叫做最短路径长度或最短距离。

28.2.狄克斯特拉算法

* 从一个点到其余各结点的最短路径。
* 设置两个结点的集合S和T，集合S中存放已找到最短路径的结点，集合T中存放当前还未找到最短路径的结点。初始状态时，集合S中只包含源点，设为v0，然后从集合T中选择到源点v0路径长度最短的结点u加入到集合S中，集合S中每加入一个新的结点u都要修改源点v0到集合T中剩余结点的当前最短路径长度值，集合T中各结点的新的当前最短路径长度值，为原来的当前最短路径长度值与从源点过结点u到达该结点的路径长度中的较小者。此过程不断重复，直到集合T中的结点全部加入到集合S 中为止。 下图为意示例：



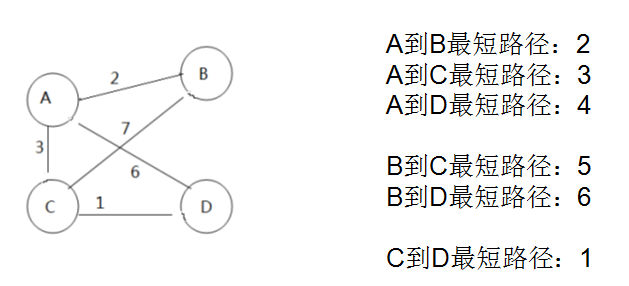
28.3.弗洛伊德算法

* 每对结点之间的最短路径。
* 设矩阵cost用来存放带权有向图G的权值，即矩阵元素cost[i][j]中存放着下标为i的结点到下标为j的结点之间的权值，可以通过递推构造一个矩阵序列A0,A1,A2,……,AN来求每对结点之间的最短路径。初始时有，A0[i][j]=cost[i][j]。当已经求出Ak,要递推求解Ak+1时，可分两种情况来考虑：一种情况是该路径不经过下标为k+1的结点，此时该路径长度与从结点vi到结点vj的路径上所经过的结点下标不大于k的最短路径长度相同；另一种情况是该路径经过下标为k+1的结点，此时该路径可分为两段，一段是从结点vi到结点vk+1的最短路径，另一段是从结点vk+1到结点vj的最短路径，此时的最短路径长度等于这两段最短路径长度之和。这两种情况中的路径长度较小者，就是要求的从结点vi到结点vj的路径上所经过的结点下标不大于k+1的最短路径长度。
* 弗洛伊德算法的算法思想可用如下递推公式描述：

A0[i][j]=cost[i][j]

Ak+1[i][j]=min{Ak[i][j], Ak[i][k+1]+Ak[k+1][j]} （0≤k≤n-1）

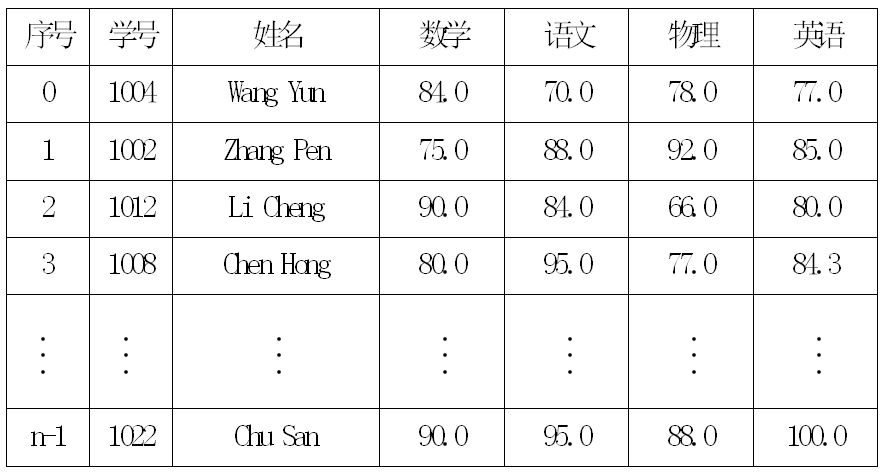
也就是说，初始时，A0[i][j]=cost[i][j]，然后进行递推，每递推一次，从结点vi到结点vj的最短路径上就多考虑了一个经过的中间结点，这样，经过n次递推后得到的An[i][j]就是考虑了经过图中所有结点情况下的从结点vi到结点vj的最短路径长度。



29.插入选择交换排序算法

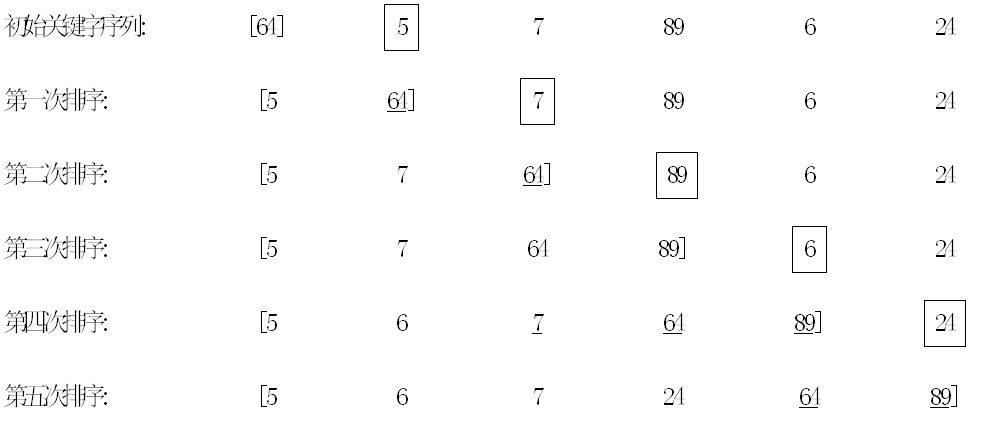
29.1.排序的基本概念

* 排序是对数据元素序列建立某种有序排列的过程。
* 关键字是要排序的数据元素集合中的一个域，排序是以关键字为基准进行的。
* 关键字分主关键字和次关键字两种。对要排序的数据元素集合来说，如果关键字满足数据元素值不同时该关键字的值也一定不同，这样的关键字称为主关键字。
* 不满足主关键字定义的关键字称为次关键字。



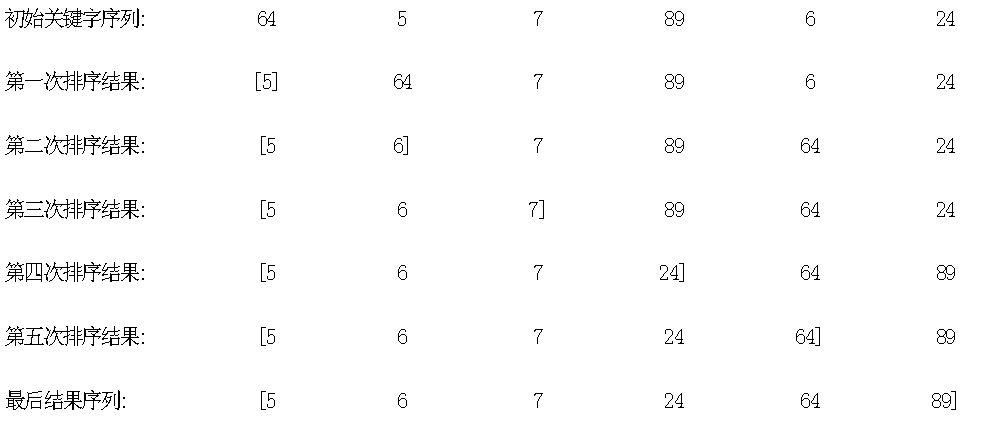
29.2.插入排序

* 直接插入排序的基本思想是：顺序地把待排序的数据元素按其值的大小插入到已排序数据元素子集合的适当位置。子集合的数据元素个数从只有一个数据元素开始逐次增大。当子集合大小最终和集合大小相同时排序完毕。



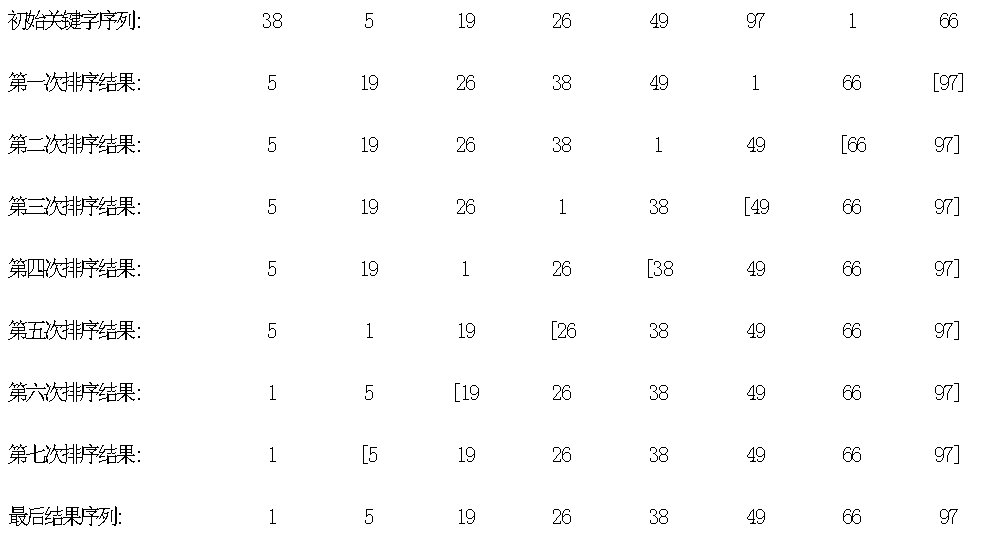
29.3.选择排序

* 选择排序的基本思想是：每次从待排序的数据元素集合中选取最小（或最大）的数据元素放到数据元素集合的最前（或最后），数据元素集合不断缩小，当数据元素集合为空时排序过程结束。常用的选择排序有直接选择排序和堆排序两种。堆排序是一种基于完全二叉树的排序。



29.4.交换排序

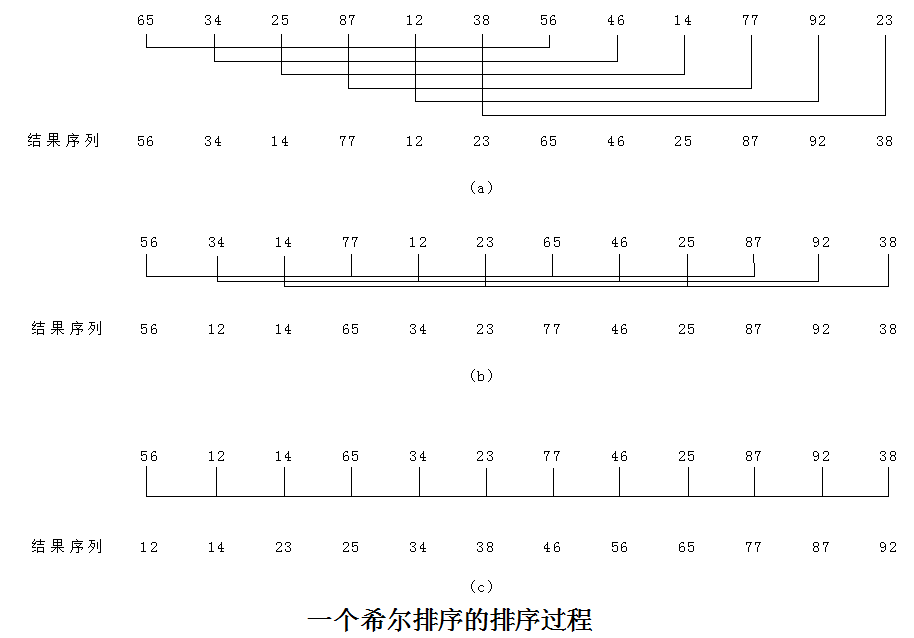
* 冒泡排序的基本思想是：设数组a中存放了n个数据元素，循环进行n-1趟如下的排序过程：第1趟时，依次比较相临两个数据元素a[i]和a[i+1]（i = 0,1,2,…,n-2）,若为逆序，即a[i]>a[i+1]，则交换两个数据元素，否则不交换，这样数值最大的数据元素将被放置在a[n-1]中。第2趟时，循环次数减1，即数据元素个数为n-1，操作方法和第1趟的类似，这样整个n个数据元素集合中数值次大的数据元素将被放置在a[n-2]中。当第n-1趟结束时，整个n个数据元素集合中次小的数据元素将被放置在a[1]中，a[0]中放置了最小的数据元素。



30.希尔排序快速排序和堆排序算法

30.1.希尔排序

* 希尔排序的基本思想是：把待排序的数据元素分成若干个小组，对同一小组内的数据元素用直接插入法排序；小组的个数逐次缩小；当完成了所有数据元素都在一个组内的排序后排序过程结束。希尔排序又称作缩小增量排序。

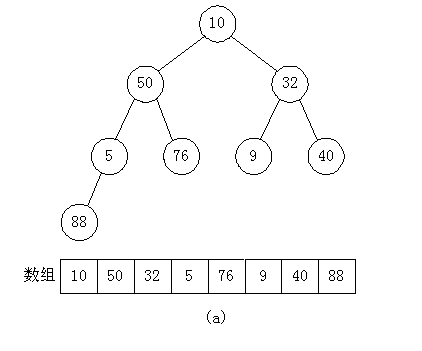
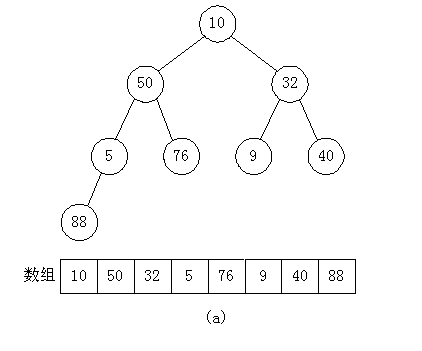


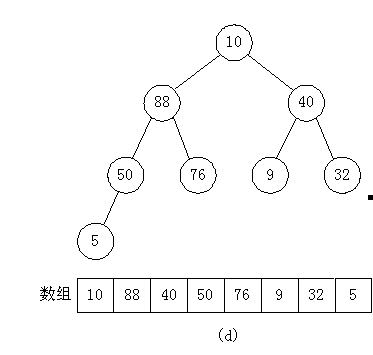
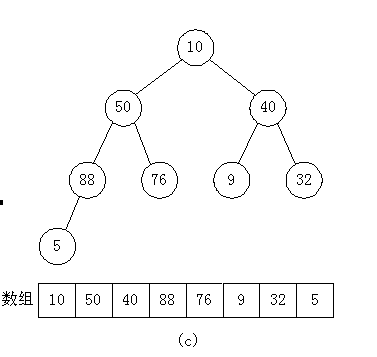
30.2.快速排序

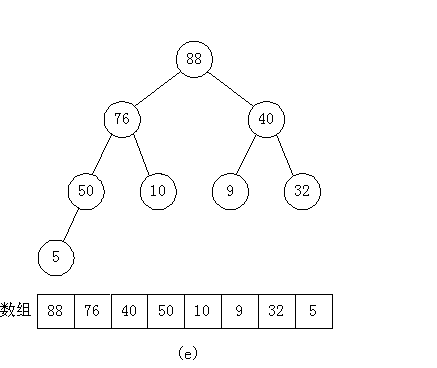
* 快速排序的基本思想：通过一趟排序将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分关键字小，则分别对这两部分继续进行排序，直到整个序列有序。把整个序列看做一个数组，把第零个位置看做中轴，和最后一个比，如果比它小交换，比它大不做任何处理；交换了以后再和小的那端比，比它小不交换，比他大交换。这样循环往复，一趟排序完成，左边就是比中轴小的，右边就是比中轴大的，然后再用分治法，分别对这两个独立的数组进行排序。

30.3.堆排序

* 在直接选择排序中，放在数组中的n个数据元素排成一个线性序列（即线性结构），要从有n个数据元素的数组中选择出一个最小的数据元素需要比较n-1次。如果能把待排序的数据元素集合构成一个完全二叉树结构，则每次选择出一个最大（或最小）的数据元素只需比较完全二叉树的高度次，即log2n次，则排序算法的时间复杂度就是O(nlog2n)。这就是堆排序的基本思想。
* 最大堆的定义如下：
* 设数组a中存放了n个数据元素，数组下标从0开始，如果当数组下标2i+1<n时有：a[i]≥a[2i+1]；如果当数组下标2i+2<n时有：a[i]≥a[2i+2]，则这样的数据结构称为最大堆。
* 最小堆的定义如下：
* 设数组a中存放了n个数据元素，数组下标从0开始，如果当数组下标2i+1<n时有：a[i]≤a[2i+1]；如果当数组下标2i+2<n时有：a[i]≤a[2i+2]，则这样的数据结构称为最小堆。
* 下图是完全二叉树调整为最大堆的过程:



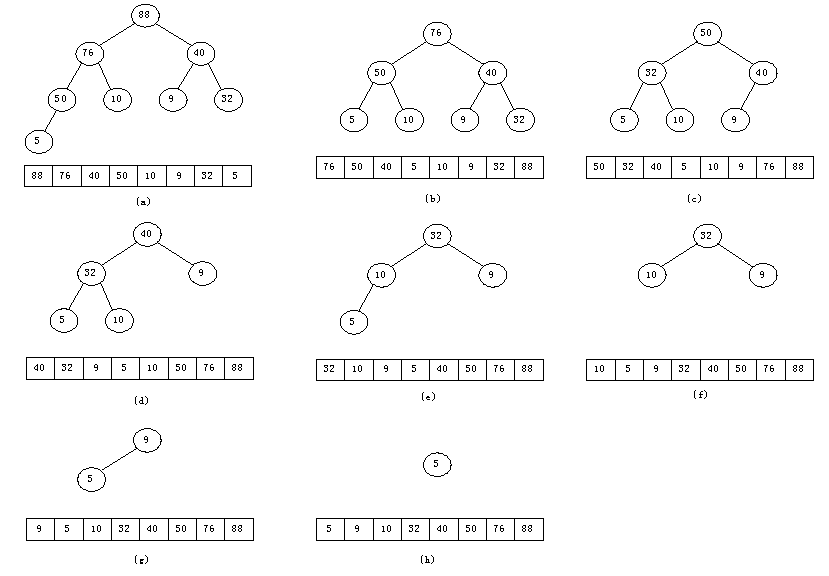




初始状态/调整结点5后 /调整结点32后 /调整结点50后 **/调整结点10后**

30.4.堆排序算法

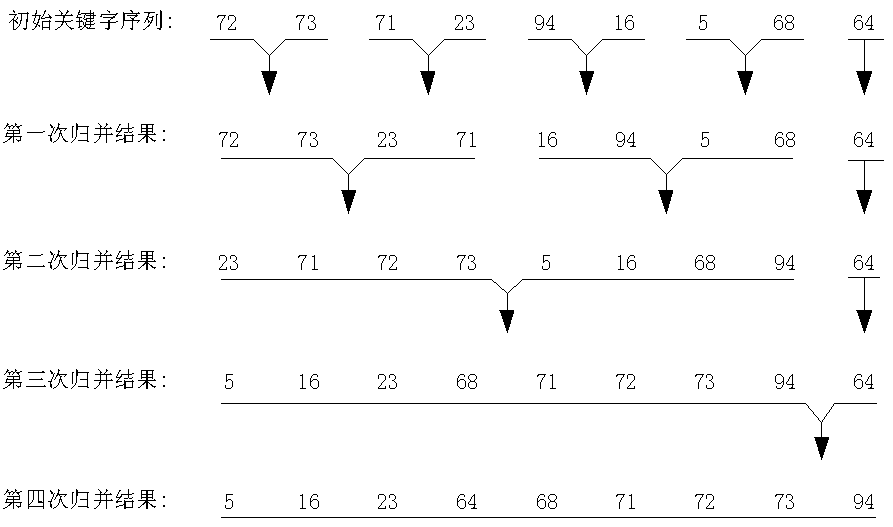
* 堆排序的基本思想是：循环执行如下过程直到数组为空：（1）把堆顶a[0]元素（最大元素）和当前最大堆的最后一个元素交换；（2）最大堆元素个数减1；（3）调整根结点使之满足最大堆的定义。
* 下图为一个堆排序算法的排序过程：



31.归并和基数排序算法及排序稳定性和性能比较

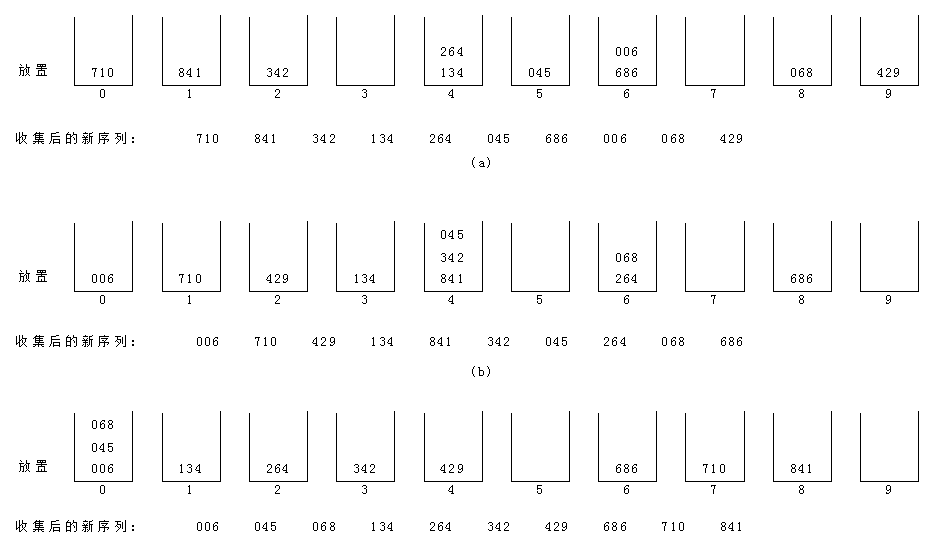
31.1.归并排序

* 归并排序主要是二路归并排序。二路归并排序的基本思想是：设数组a中存放了n个数据元素，初始时我们把它们看成是n个长度为1的有序子数组，然后从第一个子数组开始，把相临的子数组两两合并，得到n/2个（若n/2为小数则上取整）长度为2的新的有序子数组（当n为奇数时最后一个新的有序子数组的长度为1）；对这些新的有序子数组再两两归并；如此重复，直到得到一个长度为n的有序数组为止。多于二路的归并排序方法和二路归并排序方法类同。

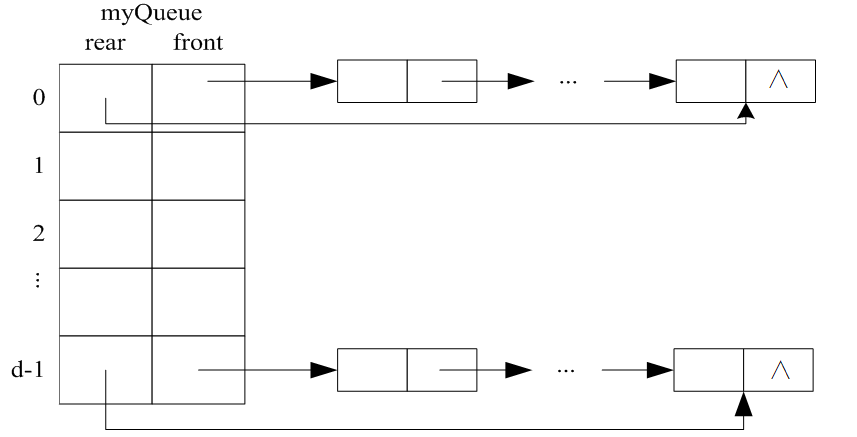


31.2.基数排序

* 基数排序算法的基本思想是：设待排序的数据元素是m位d进制整数（不足m位的在高位补0），设置d个桶，令其编号分别为0,1,2,…,d-1。首先按最低位（即个位）的数值依次把各数据元素放到相应的桶中，然后按照桶号从小到大和进入桶中数据元素的先后次序收集分配在各桶中的数据元素，这样就形成了数据元素集合的一个新的排列，我们称这样的一次排序过程为一次基数排序；再对一次基数排序得到的数据元素序列按次低位（即十位）的数值依次把各数据元素放到相应的桶中，然后按照桶号从小到大和进入桶中数据元素的先后次序收集分配在各桶中的数据元素；这样的过程重复进行，当完成了第m次基数排序后，就得到了排好序的数据元素序列。
* **基数排序算法的排序过程**



* 链式队列的基数排序算法存储结构



31.3.排序的稳定性

* 假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，ri=rj，且ri在rj之前，而在排序后的序列中，ri仍在rj之前，则称这种排序算法是稳定的；否则称为不稳定的。
* 快速排序、希尔排序、堆排序、直接选择排序不是稳定的排序算法，而基数排序、冒泡排序、直接插入排序、归并排序是稳定的排序算法。
* (1)冒泡排序

冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，我想你是不会再无 聊地把他们俩交换一下的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

* (2)选择排序

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第n-1个元素，第n个元素不用选择了，因为只剩下它一个最大的元素了。那么，在一趟选择，如果当前元素比一个元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么 交换后稳定性就被破坏了。比较拗口，举个例子，序列5 8 5 2 9， 我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

31.4.各种排序算法的性能比较

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 排序方法 | 最好时间 | 平均时间 | 最坏时间 | 辅助空间 | 稳定性 |
| 直接插入排序 | O(n) | O(n2) | O(n2) | O(1) | 稳定 |
| 希尔排序 |  | O(n1.3) |  | O(1) | 不稳定 |
| 直接选择排序 | O(n2) | O(n2) | O(n2) | O(1) | 稳定 |
| 堆排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(1) | 不稳定 |
| 冒泡排序 | O(n) | O(n2) | O(n2) | O(1) | 稳定 |
| 快速排序 | O(nlog2n) | O(nlog2n) | O(n2) | O(log2n) | 不稳定 |
| 归并排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(n) | 稳定 |
| 基数排序 | O(mn) | O(mn) | O(mn) | O(n) | 稳定 |

32.查找基本概念和静态查找算法

32.1.查找的基本概念

* 查找是在数据元素集合中查找是否存在关键字等于某个给定数据元素关键字的过程。
* 主关键字是能够唯一区分各个不同数据元素的关键字 。
* 次关键字通常不能惟一区分各个不同数据元素。以主关键字进行的查找是最经常、也是最主要的查找。
* 查找可分为静态查找和动态查找 ：
* 静态查找是指只在数据元素集合中查找是否存在某个给定的数据元素。
* 动态查找除包括静态查找的要求外，还包括在查找过程中同时插入数据元素集合中不存在的数据元素，或者从数据元素集合中删除已存在的某个数据元素的要求。
* 衡量查找算法效率的最主要标准是平均查找长度。
* 平均查找长度是指查找过程所需进行的比较次数的平均值。 数学定义为：



32.2.静态查找

* 静态查找主要有无序序列、有序序列和索引结构三种情况。
* 在无序序列中查找：在一个无序序列中查找某个数据元素是否存在的算法思想是：从数组的一端开始，用给定数据元素逐个和数组中各数据元素比较，若在数组中查找到要查找的数据元素，则查找成功；否则查找失败。
* 顺序查找算法设计。
* 设要查找的数据元素在数据元素集合中出现的概率均相等，则该算法查找成功时的平均查找长度ASL成功为：
* 算法查找失败时的平均查找长度ASL失败为：

32.3.二分查找算法

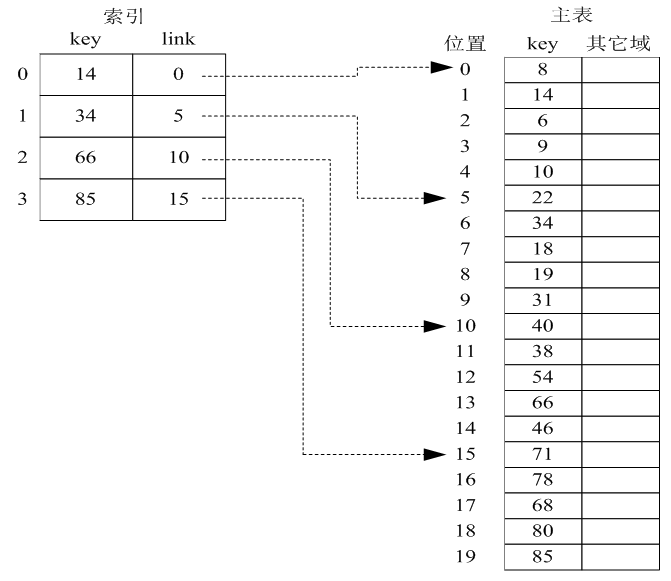
* 有序序列的二分查找算法的基本思想是：
* 在一个查找区间中，确定出查找区间的中心下标，用待查找数据元素和中心下标上的数据元素比较，若两者相等则查找成功；否则若前者小于后者则把查找区间定为原查找区间的前半段继续这样的过程；若前者大于后者则把查找区间定为原查找区间的后半段继续这样的过程。
* 二分查找算法设计。
* 当每个数据元素的查找概率相等时，二分查找算法查找成功时的平均查找长度为：
* 二分查找算法查找失败时的平均查找长度为；

32.4.索引的概念

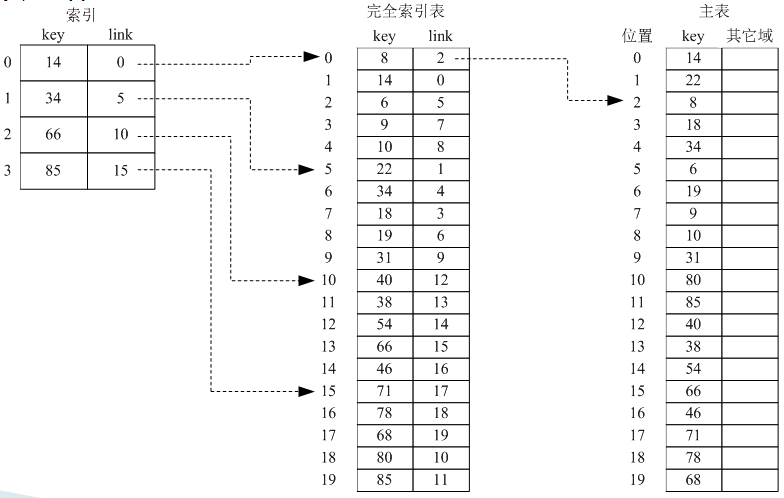
* 索引：把一个关键码与它对应的记录相关联的过程。索引由若干索引项构成，索引项至少应包含关键码和关键码对应的记录在存储器中的位置等信息。

32.5.索引查找

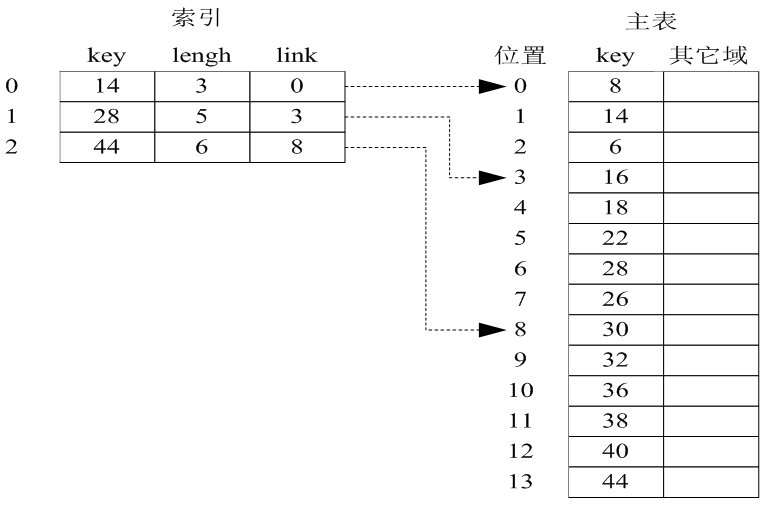
* 把要在其上建立索引的数据元素序列（通常采用数组保存）称作主表。
* 主表中存放着数据元素的全部信息。
* 索引中只保存主表中要查找数据元素的关键字和索引信息。
* 下页图是一个主表和一个按关键字key建立的索引的结构图。



* 把和主表项完全相同，但只包含索引关键字和该数据元素在主表中位置信息的索引称作主表的完全索引表。下图为带完全索引表的索引表结构图:



* 等长索引是指索引中的每个索引项对应主表中的数据元素个数是相等的。
* 索引中的索引项对应主表中的数据元素个数也可以是不相等的，这种索引结构称作不等长索引结构。
* 下图为不等长索引表结构图：



* 带索引结构的数据元素查找算法的比较次数由两部分组成：
* 一部分是在索引上查找的比较次数 ；
* 一部分是在主表中的某个子表中进行查找的比较次数。
* 假设索引的长度为m，主表中每个子表的长度为s，并假设在索引上和在主表上均采用顺序查找算法，则整个查找算法的平均查找长度为：



33.二叉排序树和B树索引算法

33.1.动态查找

* 静态查找：数据集合稳定，不需要添加，删除元素的查找操作。
* 动态查找：数据集合在查找的过程中需要添加或删除元素。

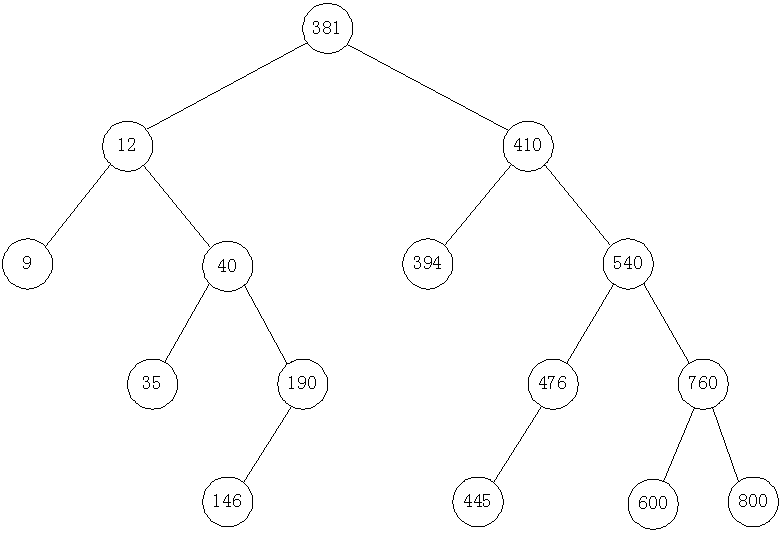
33.2.二叉排序树

* 二叉排序树也称做二叉查找树。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）若左子树非空，则左子树上所有结点的数据元素值均小于根结点的数据元素值；

（2）若右子树非空，则右子树上所有结点的数据元素值均大于或等于根结点的数据元素值；

（3）左右子树也均为二叉排序树。

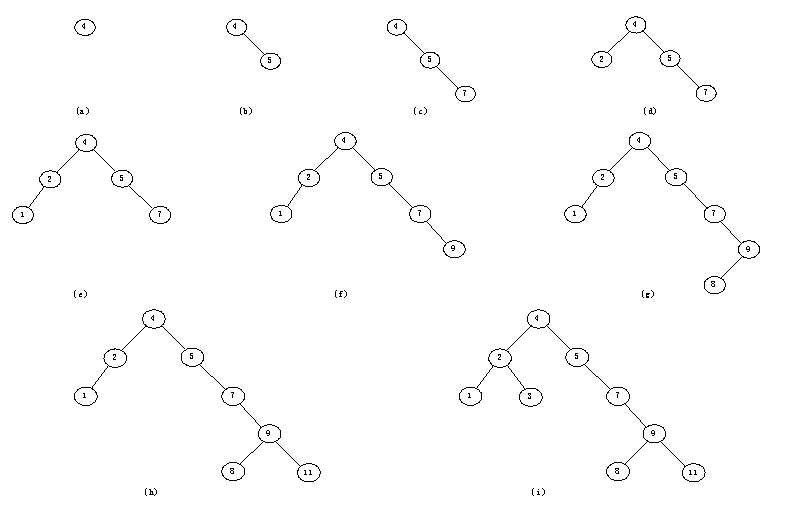


33.3.二叉排序树的查找算法

|  |
| --- |
| **public BiTreeNode find(int item){**  **if(root != null)**  **{**  **BiTreeNode temp = root;**  **while(temp != null)**  **{**  **if(temp.getData() == item)**  **return temp;**  **if(temp.getData() < item)**  **temp = temp.getRight();**  **else**  **temp = temp.getLeft();**  **}**  **}**  **return null;**  **}** |

int[] a = {4, 5, 7, 2, 1, 9, 8, 11, 3};

**二叉排序树的插入**



* 二叉排序树的删除算法 ：

删除操作的要求是：首先查找数据元素是否在二叉排序树中存在，若不存在则结束；若存在则按下面四种情况分别进行不同的删除操作。这四种情况是 ：

（1）要删除结点无孩子结点；

（2）要删除结点只有左孩子结点；

（3）要删除结点只有右孩子结点；

（4）要删除结点有左右孩子结点。

* 对于上述四种不同情况，相应的删除方法是：

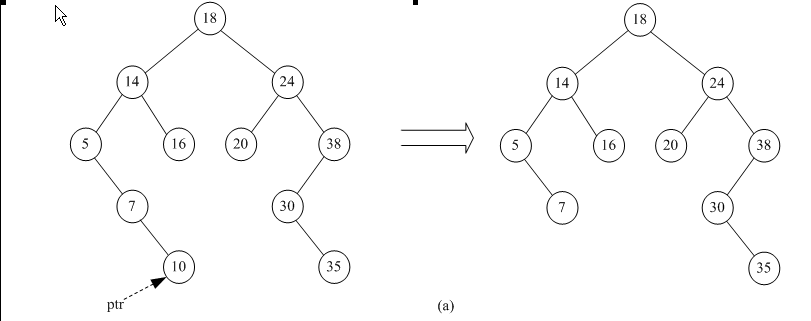
（1）要删除结点无孩子结点时，直接删除该结点。

（2）要删除结点只有左孩子结点时，删除该结点且使被删除结点的双亲结点指向被删除结点的左孩子结点。

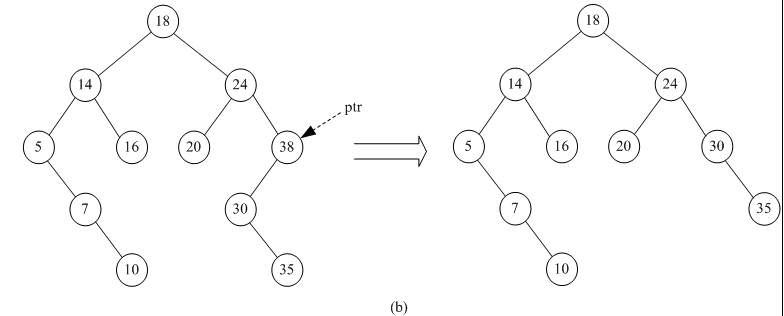
（3）要删除结点只有右孩子结点时，删除该结点且使被删除结点的双亲结点指向被删除结点的右孩子结点。

（4）要删除结点有左右孩子结点时，分如下三步完成：首先寻找数据元素值大于要删除结点数据元素关键字的最小值，然后把该最小值拷贝到要删除的结点上；最后删除该最小值所在的结点。

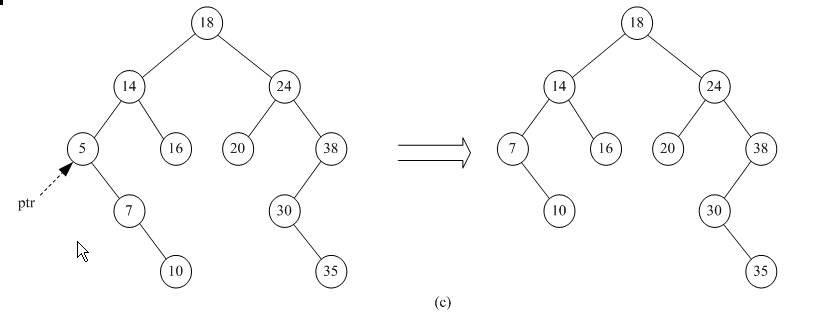
* 删除结点无孩子结点



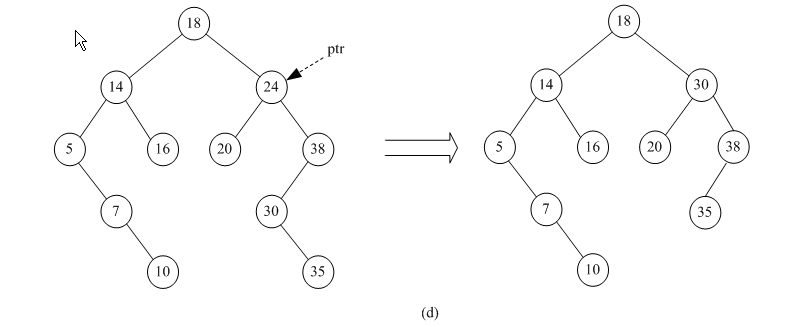
* 删除结点只有左孩子结点



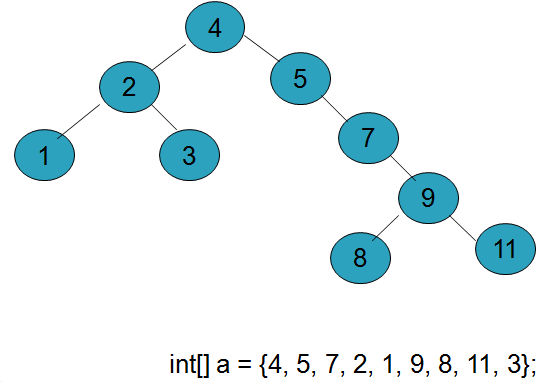
* 删除结点只有右孩子结点



* 删除结点有左右孩子结点



* 二叉排序树算法设计
* 二叉树结点类
* 二叉树类
* 测试类



33.4.二叉排序树的性能分析

* 对有n个结点的二叉排序树来说，若每个数据元素的查找概率相等，则二叉排序树平均查找长度是结点深度的函数，即：



* 若每个数据元素的查找概率相等，则二叉排序树查找成功的平均查找长度为：



* 当二叉排序树是一棵单分支退化树时，查找成功的平均查找长度和有序数组的平均查找长度相同，即为：

33.5.B\_树

* B\_树是一种平衡多叉排序树。

B\_树中所有结点的孩子结点个数的最大值称为B\_树的阶。一棵m阶的B\_树或者是一棵空树，或者是满足下列要的m阶的B\_树：

（1）树中每个结点至多有m个孩子结点。

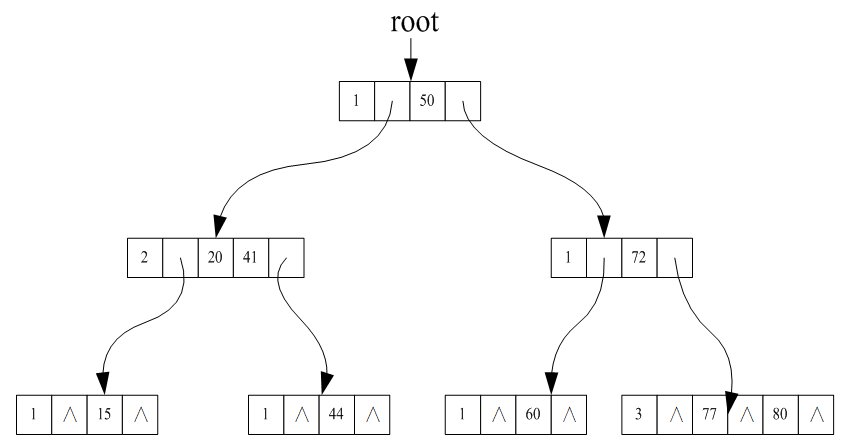
（2）除根结点外，其他结点至少有[m/2]个孩子结点。

（3）若根结点不是叶结点，则根结点至少有两个孩子结点

（4）每个结点的结构为：

（5）所有叶结点都在二叉排序树的同一层上。

* 一棵3阶B\_树的示例



33.6.B\_树的查找算法

* 在B\_树上查找数据元素（确切说是查找数据元素的关键字）key的方法为：将 key与根结点的Ki逐个顺序比较：

（1）若key=Ki则查找成功 ；

（2）若key<K1则沿着指针P0所指的子树继续查找；

（3）若Ki<key<Ki+1则沿着指针Pi所指的子树继续查找

（4）若key>Kn则沿着指针Pn所指的子树继续查找 。

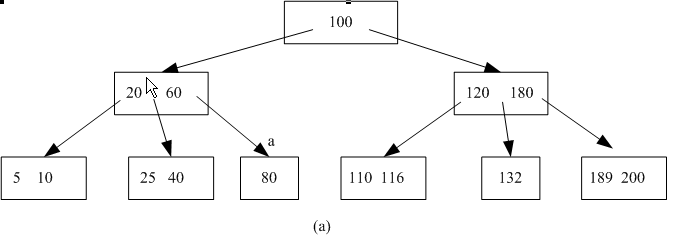
33.7.B\_树的插入算法

将数据元素key插入到B\_树的过程分两步完成：

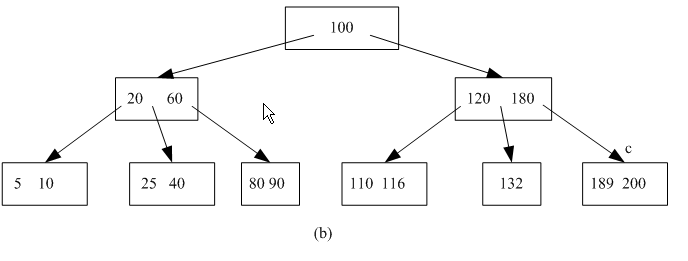
（1）利用B\_树的查找算法找出该数据元素结点应该插入的结点位置；

（2）判断要插入的结点是否还有空位置，即判断该结点是否满足n<m-1，若该结点满足n<m-1，说明该结点还有空位置，直接把数据元素key插入到该结点的合适位置上；若该结点有n=m-1，说明该结点已没有空位置，要插入就要分裂该结点。

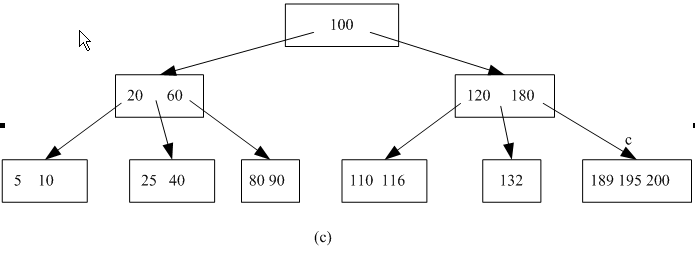
* 初始状态



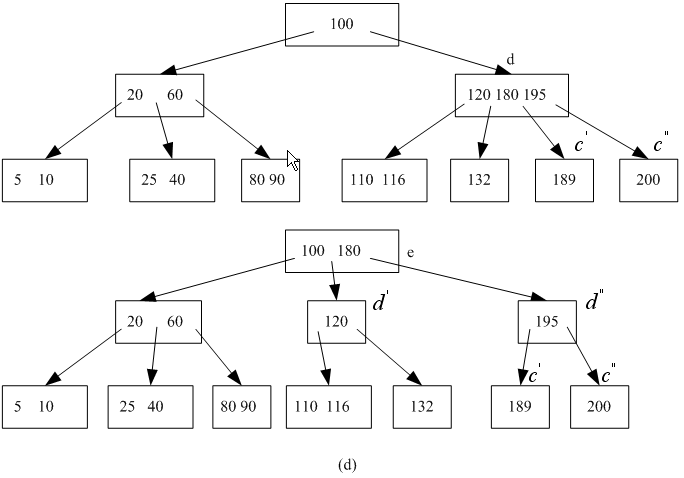
* 插入90后的状态



* 插入195后结点分裂前的状态



* **插入195后结点的分裂过程**



33.8.B\_树的删除

* 在B\_树上删除数据元素key的过程分两步完成：

（1）利用B\_树的查找算法找出该数据元素所在的结点；

（2）在结点上删除数据元素key分两种情况：一种是在叶结点上删除数据元素；另一种是在非叶结点上删除数据元素。

在非叶结点上删除数据元素的算法思想为：假设要删除一个结点的数据元素Ki（1≤i≤n），首先寻找该结点Pi所指子树中的最小数据元素Kmin，然后用Kmin覆盖要删除的数据元素Ki，最后再以指针Pi所指结点为根结点查找并删除Kmin。

* 在B\_树的叶结点上删除数据元素共有以下三种情况：

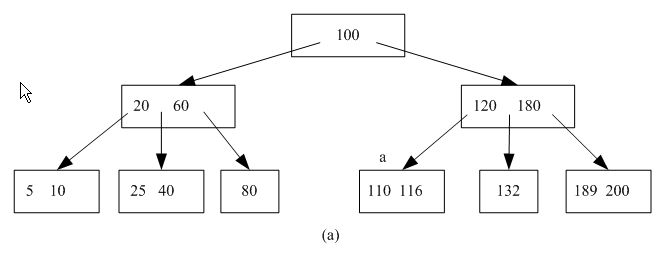
（a）要删除数据元素结点的数据元素个数n>[m/2]-1

（b）要删除数据元素结点的数据元素个数n=[m/2]-1

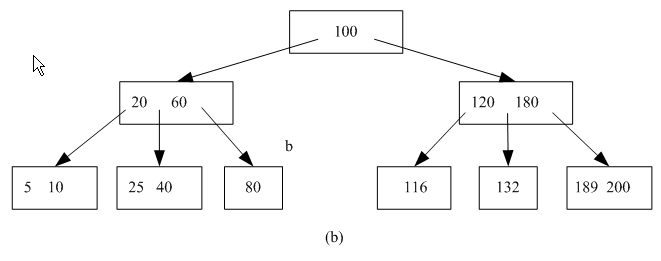
（c）假如要删除数据元素结点的数据元素个数n=[m/2]-1并且该结点的左和右兄弟结点（如果存在的话）中数据元素个数n均等于[m/2]-1

下面几幅图是在3阶B\_树上进行删除操作的示例

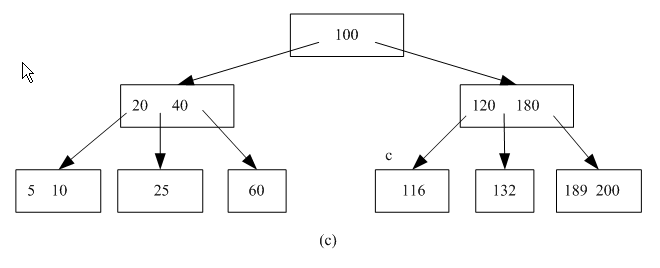
* 初始状态



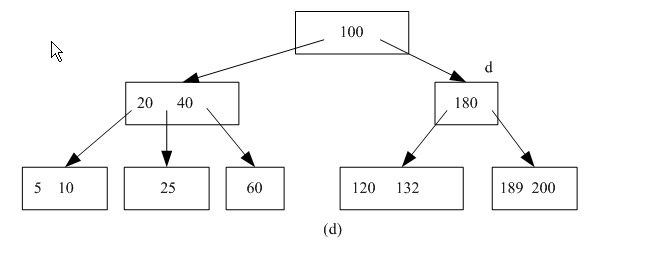
* 删去110后的状态



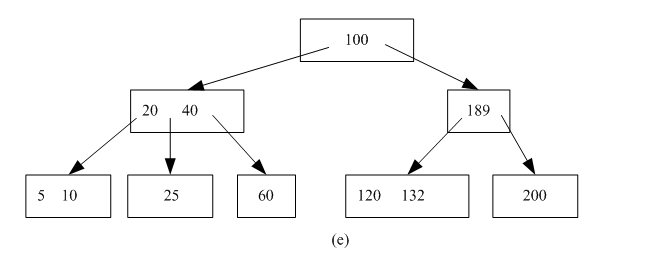
* 删去80后的状态



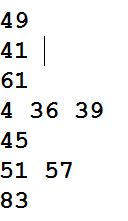
* 删去116后的状态



* 删去180后的状态



33.9.B\_树算法设计



34.HashTable算法概述与设计

34.1.哈希表

哈希表主要是构造一个映射函数，该函数以数据元素为自变量，函数值即为数据元素在内存中的存储位置。通常把这样的映射函数称为哈希函数h(x)。因此可以说，哈希表是通过哈希函数h(x)来确定数据元素x存放位置h(x)的一种特殊存储结构。

34.2.哈希表的构造方法

* 构造哈希表的方法是：设要存储的数据元素个数为 n，设置一个长度为m（m≥n）的连续内存单元（即数组），分别以每个数据元素的关键字Ki（0≤i≤n-1）为自变量，以哈希函数h(Ki)值为该数据元素在数组中的下标值存储该数据元素。
* 把构造哈希表时Ki≠Kj（i≠j）,但h(Ki)=h(Kj)的现象称作哈希冲突。
* 设计一个好的哈希函数，使得尽量避免哈希冲突，以及哈希冲突发生后，如何解决哈希冲突，就成了建立哈希表的两个关键问题。
* 哈希冲突主要与三个因素有关：

（1）与装填因子a有关。

（2）与所采用的哈希函数有关。

（3）与解决哈希冲突的哈希冲突函数有关。

34.3.哈希函数构造方法

* 设要存放的数据元素有n个，存放数据元素的数组个数为m，哈希函数的设计目标，就是要使通过哈希函数得到的n个数据元素的哈希地址 。
* 1 除留余数法

除留余数法是用数据元素的关键字K除以哈希表长度m所得的余数作为哈希地址的方法。除留余数法的哈希函数h(K)为：

h(K) = K mod m

* 2 直接定址法

直接定址法是以数据元素的关键字K本身或关键字加上某个数值常量C作为哈希函数的方法。直接定址法的哈希函数h(K)为：

h(K) = K + C

* 3 数字分析法

数字分析法是取数据元素关键字中某些取值较均匀的数字位构造哈希函数的方法。它只适合于所有关键字值已知的情况。

34.4.哈希冲突解决方法

解决哈希冲突的方法主要有开放定址法和链表法两大类。

* 开放定址法

开放定址法是一类以发生哈希冲突的哈希地址为自变量、通过某种哈希冲突函数得到一个新的空闲的哈希地址的方法。开放定址法的哈希冲突函数通常是一组。

* 线性探查法

线性探查法是从发生哈希冲突的地址d开始，依次探查d的下一个地址，直到找到一个空闲单元为止。线性探查法的数学递推描述公式为：



* 平方探查法

设发生哈希冲突的地址为d，则平方探查法的探查序列为：d+2, d+21, d+22, …。平方探查法的数学递推描述公式为：



* 伪随机数法

设发生哈希冲突的地址为d，则伪随机数法的探查序列为d+Ri。Ri为一伪随机数序列的第i个数值。伪随机数法的数学递推描述公式为：



* 链表法

链表法解决哈希冲突的基本思想是：如果没有发生哈希冲突，则直接在该地址保存该数据元素；如果发生了哈希冲突，则把发生哈希冲突的数据元素保存在另外的单链表中。

用链表法解决哈希冲突通常有两种方法：

第一种方法是为发生哈希冲突的不同的同义词建立不同的单链表。

第二种方法是为所有发生哈希冲突的数据元素建立一个单链表。

* 用链表法解决冲突的哈希表示例

35.MD5加密算法原理与应用

35.1.MD5原理

* MD5的全称是Message-digest Algorithm 5（信息-摘要算法），用于确保信息传输完整一致。在90年代初由MIT Laboratory for Computer Science和RSA Data Security Inc,的Ronald L. Rivest开发出来，经MD2、MD3和MD4发展而来。它的作用是让大容量信息在用数字签名软件签署私人[密钥](http://baike.baidu.com/view/934.htm)前被"压缩"成一种保密的格式（就是把一个任意长度的字节串变换成一定长的大整数）。
* MD5一度被广泛应用于计算机安全领域。但由于近年来MD5的弱点不断被发现，以及当今计算机运算能力的不断提升，现在已经可能人为构造出两个具有相同MD5校验值的信息，使本算法不再适合现今的安全领域。目前，MD5算法因其普遍、稳定、快速的特点，仍广泛应用于普通数据的错误检查领域。例如在一些BitTorrent下载中，软件将通过计算MD5检验下载到的文件片段的完整性。
* MD5算法较老，散列长度固定为128比特，随着计算机运算能力提高，更快地找到“碰撞”是有可能的。因此，在安全要求高的场合不应再使用MD5。
* 步骤1：

在MD5算法中，首先需要对信息进行填充，使其字节长度对512求余的结果等于448。因此，信息的字节长度（Bits Length）将被扩展至N\*512+448，即N\*64+56个字节（Bytes），N为一个正整数。填充的方法如下，在信息的后面填充一个1和无数个0，直到满足上面的条件时才停止用0对信息的填充。

然后，在在这个结果后面附加一个以64位二进制表示的填充前信息长度。经过这两步的处理，现在的信息字节长度=N\*512+448+64=(N+1)\*512，即长度恰好是512的整数倍。这样做的原因是为满足后面处理中对信息长度的要求。

MD5中有四个32位被称作链接变量（Chaining Variable）的整数参数，他们分别为：A=0x01234567，B=0x89abcdef，C=0xfedcba98，D=0x76543210。

当设置好这四个链接变量后，就开始进入算法的四轮循环运算。循环的次数是信息中512位信息分组的数目。

将上面四个链接变量复制到另外四个变量中：A到a，B到b，C到c，D到d。

* 步骤2：

主循环有四轮，每轮循环都很相似。第一轮进行16次操作。每次操作对a、b、c和d中的其中三个作一次非线性函数运算，然后将所得结果加上第四个变量，文本的一个子分组和一个常数。再将所得结果向右环移一个不定的数，并加上a、b、c或d中之一。最后用该结果取代a、b、c或d中之一。

以一下是每次操作中用到的四个非线性函数（每轮一个）。

　　F(X,Y,Z) =(X&Y)|((~X)&Z)

　　G(X,Y,Z) =(X&Z)|(Y&(~Z))

　　H(X,Y,Z) =X^Y^Z

　　I(X,Y,Z)=Y^(X|(~Z))

　　（&是与，|是或，~是非，^是异或）

假设Mj表示消息的第j个子分组（从0到15），

FF(a, b, c, d, Mj, s, ti)

表示 a = b + ((a + (F(b, c, d) + Mj + ti) << s

GG(a, b, c, d, Mj, s, ti)

表示 a = b + ((a + (G(b, c, d) + Mj + ti) << s

HH(a, b, c, d, Mj, s, ti)

表示 a = b + ((a + (H(b, c, d) + Mj + ti) << s

II(a, b, c, d, Mj, s, ti)

表示 a = b + ((a + (I(b, c, d) + Mj + ti) << s

* 这四轮（64步）：

第一轮

　　FF(a, b, c, d, M0, 7, 0xd76aa478)

　　FF(d, a, b, c, M1, 12, 0xe8c7b756)

　　FF(c, d, a, b, M2, 17, 0x242070db)

　　FF(b, c, d, a, M3, 22, 0xc1bdceee)

　　FF(a, b, c, d, M4, 7, 0xf57c0faf)

　　FF(d, a, b, c, M5, 12, 0x4787c62a)

　　FF(c, d, a, b, M6, 17, 0xa8304613)

　　FF(b, c, d, a, M7, 22, 0xfd469501)

　　FF(a, b, c, d, M8, 7, 0x698098d8)

　　FF(d, a, b, c, M9, 12, 0x8b44f7af)

　　FF(c, d, a, b, M10, 17, 0xffff5bb1)

　　FF(b, c, d, a, M11, 22, 0x895cd7be)

　　FF(a, b, c, d, M12, 7, 0x6b901122)

　　FF(d, a, b, c, M13, 12, 0xfd987193)

　　FF(c, d, a, b, M14, 17, 0xa679438e)

FF(b, c, d, a, M15, 22, 0x49b40821)

第二轮

　　GG(a, b, c, d, M1, 5, 0xf61e2562)

　　GG(d, a, b, c, M6, 9, 0xc040b340)

　　GG(c, d, a, b, M11, 14, 0x265e5a51)

　　GG(b, c, d, a, M0, 20, 0xe9b6c7aa)

　　GG(a, b, c, d, M5, 5, 0xd62f105d)

　　GG(d, a, b, c, M10, 9, 0x02441453)

　　GG(c, d, a, b, M15, 14, 0xd8a1e681)

　　GG(b, c, d, a, M4, 20, 0xe7d3fbc8)

　　GG(a, b, c, d, M9, 5, 0x21e1cde6)

　　GG(d, a, b, c, M14, 9, 0xc33707d6)

　　GG(c, d, a, b, M3, 14, 0xf4d50d87)

　　GG(b, c, d, a, M8, 20, 0x455a14ed)

　　GG(a, b, c, d, M13, 5, 0xa9e3e905)

　　GG(d, a, b, c, M2, 9, 0xfcefa3f8)

　　GG(c, d, a, b, M7, 14, 0x676f02d9)

　　GG(b, c, d, a, M12, 20, 0x8d2a4c8a)

第三轮

　　HH(a, b, c, d, M5, 4, 0xfffa3942)

　　HH(d, a, b, c, M8, 11, 0x8771f681)

　　HH(c, d, a, b, M11, 16, 0x6d9d6122)

　　HH(b, c, d, a, M14, 23, 0xfde5380c)

　　HH(a, b, c, d, M1, 4, 0xa4beea44)

　　HH(d, a, b, c, M4, 11, 0x4bdecfa9)

　　HH(c, d, a, b, M7, 16, 0xf6bb4b60)

　　HH(b, c, d, a, M10, 23, 0xbebfbc70)

　　HH(a, b, c, d, M13, 4, 0x289b7ec6)

　　HH(d, a, b, c, M0, 11, 0xeaa127fa)

　　HH(c, d, a, b, M3, 16, 0xd4ef3085)

　　HH(b, c, d, a, M6, 23, 0x04881d05)

　　HH(a, b, c, d, M9, 4, 0xd9d4d039)

　　HH(d, a, b, c, M12, 11, 0xe6db99e5)

　　HH(c, d, a, b, M15, 16, 0x1fa27cf8)

　　HH(b, c, d, a, M2, 23, 0xc4ac5665)

第四轮

　　II(a, b, c, d, M0, 6, 0xf4292244)

　　II(d, a, b, c, M7, 10, 0x432aff97)

　　II(c, d, a, b, M14, 15, 0xab9423a7)

　　II(b, c, d, a, M5, 21, 0xfc93a039)

　　II(a, b, c, d, M12, 6, 0x655b59c3)

　　II(d, a, b, c, M3, 10, 0x8f0ccc92)

　　II(c, d, a, b, M10, 15, 0xffeff47d)

　　II(b, c, d, a, M1, 21, 0x85845dd1)

　　II(a, b, c, d, M8, 6, 0x6fa87e4f)

　　II(d, a, b, c, M15, 10, 0xfe2ce6e0)

　　II(c, d, a, b, M6, 15, 0xa3014314)

　　II(b, c, d, a, M13, 21, 0x4e0811a1)

　　II(a, b, c, d, M4, 6, 0xf7537e82)

　　II(d, a, b, c, M11, 10, 0xbd3af235)

　　II(c, d, a, b, M2, 15, 0x2ad7d2bb)

　　II(b, c, d, a, M9, 21, 0xeb86d391)

* 常数ti可以如下选择：

　　在第i步中，ti是4294967296\*abs(sin(i))的整数部分，i的单位是弧度。(4294967296等于2的32次方)。所有这些完成之后，将A、B、C、D分别加上a、b、c、d。然后用下一分组数据继续运行算法，最后的输出是a、b、c和d的级联。而S参数可以任意指定。

从理论上来说，反向推导MD5是不可行的。

35.2.MD5算法设计

* java MD5算法设计

35.3.MD5算法应用

* 使用MD5实现用户注册与登录

36.贪心算法以及应用

36.1.贪心算法

* 贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。贪心算法不是对所有问题都能得到整体最优解，但对范围相当广泛的许多问题他能产生整体最优解或者是整体最优解的近似解。
* 贪婪算法（Greedy algorithm）是一种对某些求最优解问题的更简单、更迅速的设计技术。用贪婪法设计算法的特点是一步一步地进行，常以当前情况为基础根据某个优化测度作最优选择，而不考虑各种可能的整体情况，它省去了为找最优解要穷尽所有可能而必须耗费的大量时间，它采用自顶向下，以迭代的方法做出相继的贪心选择，每做一次贪心选择就将所求问题简化为一个规模更小的子问题，通过每一步贪心选择，可得到问题的一个最优解，虽然每一步上都要保证能获得局部最优解，但由此产生的全局解有时不一定是最优的，所以贪婪法不要回溯。

36.2.贪心算法的特性

⑴ 有一个以最优方式来解决的问题。为了构造问题的解决方案，有一个候选的对象的集合：比如不同面值的硬币。

⑵ 随着算法的进行，将积累起其它两个集合：一个包含已经被考虑过并被选出的候选对象，另一个包含已经被考虑过但被丢弃的候选对象。

⑶ 有一个函数来检查一个候选对象的集合是否提供了问题的解答。该函数不考虑此时的解决方法是否最优。

⑷ 还有一个函数检查是否一个候选对象的集合是可行的，也即是否可能往该集合上添加更多的候选对象以获得一个解。和上一个函数一样，此时不考虑解决方法的最优性。

⑸ 选择函数可以指出哪一个剩余的候选对象最有希望构成问题的解。

⑹ 最后，目标函数给出解的值。

36.3.贪心算法的步骤

建立数学模型来描述问题。

把求解的问题分成若干个子问题。

对每一子问题求解，得到子问题的局部最优解。

把子问题的解局部最优解合成原来解问题的一个解。

36.4.贪心算法实例分析

* 背包问题：背包容量是M=150。有7个物品，物品不可以分割成任意大小。要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

物品 A B C D E F G

重量 35 30 60 50 40 10 25

价值 10 40 30 50 35 40 30

* 根据贪心的策略，每次挑选价值最大的物品装入背包，得到的结果是否最优？

每次挑选所占重量最小的物品装入是否能得到最优解？

每次选取单位重量价值最大的物品，是否能成为解本题的策略?

* 贪心策略：选取价值最大者。

W=30

物品：A B C

重量：28 12 12

价值：30 20 20

根据策略，首先选取物品A，接下来就无法再选取了，可是，选取B、C则更好。

* 贪心策略：选取重量最小。

W=30

物品：A B C

重量：28 12 12

价值：30 10 10

它的反例与第一种策略的反例差不多。

* 贪心策略：选取单位重量价值最大的物品。

W=30

物品：A B C

重量：28 20 10

价值：28 20 10

根据策略，三种物品单位重量价值一样，程序无法依据现有策略作出判断，如果选择A，则答案错误。

W=40

物品：A B C

重量：25 20 15

价值：25 20 15

对于选取单位重量价值最大的物品这个策略，可以再加一条优化的规则：对于单位重量价值一样的，则优先选择重量小的！这样，上面的反例就解决了。

本题是个DP问题，用贪心法并不一定可以求得最优解，以后了解了动态规划算法后本题就有了新的解法。

36.5.Java贪心算法实例分析

有十一个快件，每个快件有送货的起始时间，要求用最少的车，运送完所有的快件。

快件1:1:00-4:00

快件2:3:00-5:00

快件3:0:00-6:00

快件4:5:00-7:00

快件5:3:00-8:00

快件6:5:00-9:00

快件7:6:00-10:00

快件8:8:00-11:00

快件9:8:00-12:00

快件10:2:00-13:00

快件11:12:00-14:00

* 设计货车类
* 设计快件类
* 设计排序类
* 设计贪心算法类

37.动态规划算法与0-1背包问题

37.1.动态规划算法简介

动态规划算法与分治法类似，其基本思想是将待求解问题分解成若干个子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划法求解的问题，经分解得到的子问题往往不是互相独立的，若用分治法解这类问题，则分解得到的子问题数目太多，以至于最后解决原问题需要耗费过多的时间。动态规划法又和贪婪算法有部分相同，在动态规划中，可将一个问题的解决方案视为一系列决策的结果。

37.2.动态规划算法的设计步骤

描述最优解的结构特征；

递归地定义一个最优解的值；

自底向上计算一个最优解的值；

从已计算的信息中构造一个最优解；

根据最优解写出算法。

37.3.动态规划与贪心算法的比较

动态规划和贪心算法都是一种递推算法 ，均有局部最优解来推导全局最优解。

不同点：

贪心算法：

1.贪心算法中，作出的每步贪心决策都无法改变，因为贪心策略是由上一步的最优解推导下一步的最优解，而上一部之前的最优解则不作保留。

2.贪心法正确的条件是：每一步的最优解一定包含上一步的最优解。

动态规划算法：

1.全局最优解中一定包含某个局部最优解，但不一定包含前一个局部最优解，因此需要记录之前的所有最优解

2.动态规划的关键是状态转移方程，即如何由以求出的局部最优解来推导全局最优解

3.边界条件：即最简单的，可以直接得出的局部最优解。

相同点是：两种方法都利用了最优子结构特征。

不同点是：在贪婪算法中，每采用一次贪婪准则便做出一个不可撤回的决策，而在动态规划中，还要考察每个最优决策序列中是否包含一个最优子序列。

易错误处：当贪心算法满足全局最优时，可能我们试图使用动态规划求解，但前者更有效；当实事上只有动态规划才能求解时，错误地使用了贪心法。

37.4.0-1背包问题在两种算法上体现

物品个数：3

W=10

物品：A B C

重量：3 4 5

价值：4 5 6

采用贪心算法：总是让余下单位价值量最大的物品装入背包中，直至背包不能够再装入物品。

最优解：A B 总价值：9

物品个数：3

W=10

物品：A B C

重量：3 4 5

价值：4 5 6

采用动态规划算法：

1.只装一个物品 C 总价值：6

2.只装两个物品 B C 总价值：11

3.只装三个物品 无解

最优解： B C 总价值：11

37.5.找零问题在两种算法上体现

比如中国的货币，只看元，有1元、2元、5元、10元20、50、100。

如果我要找16元，可以拿16个1元，8个2元，但是怎么找张数最少呢？

如果用贪心算，就是我每一次拿那张可能拿的最大的。

 比如16，我第一次拿20拿不起，拿10元，OK，剩下6元，再拿个5元，剩下1元   也就是3张   10、5、1。

  每次拿能拿的最大的，就是贪心法。

但是一定注意，贪心得到的并不是最优解，也就是说用贪心不一定是拿的最少的张数 。 贪心只能得到一个比较好的解，而且贪心算法很好想得到。   再注意，为什么我们的钱可以用贪心呢？因为我们国家的钱的大小设计，正好可以使得贪心算法算出来的是最优解（一般是个国家的钱币都应该这么设计）。如果设计成别的样子情况就不同了。

比如某国的钱币分为：   1元、3元、4元

 如果要找6元钱 ，怎么找？

贪心算法的话 ，先拿4，再拿两个1，一共3张钱 。

 实际最优呢？ 两张3元就够了。

37.6.贪心与动态规划对比

求最优解的问题，从根本上说是一种对解空间的遍历。最直接的暴力分析（穷举分析）容易得到，最优解的解空间通常都是以指数阶增长，因此暴力穷举都是不可行的。最优解问题大部分都可以拆分成一个个的子问题，把解空间的遍历视作对子问题树的遍历，则以某种形式对树整个的遍历一遍就可以求出最优解。

贪心和动态规划本质上是对子问题树的一种修剪。两种算法要求问题都具有的一个性质就是“子问题最优性”。即，组成最优解的每一个子问题的解，对于这个子问题本身肯定也是最优的。如果以自顶向下的方向看问题树（原问题作根），则，我们每次只需要向下遍历代表最优解的子树就可以保证会得到整体的最优解。形象一点说，可以简单的用一个值（最优值）代表整个子树，而不用去求出这个子树所可能代表的所有值。

37.7.动态规划算法应用

0-1背包问题的实现。

问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。

采用对象建模形式

背包建模

背包问题求解类

背包问题测试类

38.算法应用举例之随机发牌算法

算法应用举例

需求分析：

52张牌随机发给4个玩家，每个玩家手中13张牌。

* 利用java的Collection.shuffle();对集合里的元素打乱次序。
* 使用servlet实现界面显示四个玩家的牌。