

数据结构复习指南

For 软件 1001 班

马彬 张力

序

“祝大家都考出好成绩！”

本复习指南整理了数据结构科目所有的重点难点，方便大家复习。直接看数据结构 PPT，未免很耗时，效果又不一定好。如果大家在复习的时候先把指南整体看一遍，相信将会对复习有莫大的帮助。

指南增加了第 0 章：C++极速入门，剩余章节与 PPT 一致。其中：0~4 章为张力撰写（第 0 章 C++极速入门、第 1 章 概论、第 2 章 数组、第 3 章 链表、第 4 章 栈和队列），5~10 章为马彬撰写（第 5 章 递归、第 6 章 树、第 7 章 搜索、第 8 章 图、第 9 章 排序、第 10 章 散列）。每章节后面的习题都选自复习 PPT，解法是我们自己写的，如果有错误或者遗漏，请及时在软件 1001 二群内通知大家，以免由于个人疏漏而导致错误蔓延，谢谢！

最后，祝大家都取得好成绩！

——软件 1001 班 马彬、张力

2012 年 7 月 2 日

<本指南欢迎共享，但请**务必**保留前两页，谢谢！>

第 0 章 C++极速入门

“献给想临时抱住 C++ 佛脚的童鞋们”

1. 文件后缀名

C++文件的后缀名有许多种，较为常见的是.cpp；头文件的后缀名也有很多种，较为常见的是.h。

2. #include

对于#include,这相当于 Java 中的 Import ,作用是引入需要的库文件。
<iostream>中包含了标准输入输出流，我们常用的 **cin** 和 **cout** 函数以及>>和<<运算符都在其中。<iterator>是一个迭代器库，使用
istream_iterator 或者 ostream_iterator 都要把他包含进来。<cstdlib>提供了一些函数和符号常量，如 NULL。

3. #include 后面接< >还是' '?

对于标准库 ,其后应该接< >,如<iostream>;对于自定义的头文件，
则应该用' '，这时候使用的是相对路径。

4. 什么是声明？什么是定义？

简单的来讲：声明就是说明这个类中**有什么类型**，什么名字的变量和什么样的成员函数（当然必须指明函数的返回值，函数名和形参列表），却不用对变量初始化或者具体写出成员函数的实现代码；而变量的初始化或者函数的具体代码实现的过程叫做定义。

5. C++中声明和定义类通常的做法

C++是一种面向对象的编程语言，因此不可避免的要声明和定义类，
我们在 java 中总是在把声明和定义放在一起做，比如刚声明了一个

变量就给它初始化，或者刚声明了一个方法紧接着就在后面跟上 { }，并在其中给出代码实现来定义。虽然 C++ 中也可以这样做，但通常的做法是把类的声明放入头文件中，而把类的定义放入相对应的 .cpp 中。

6. ‘::’ 作用域运算符

:: 表示 :: 右边的操作数可以在 :: 左边的作用域上找到。例如 : A :: function () 表示这个 function () 是类 A 的。

7. 命名空间

编码中，我们总是使用如 std :: cout 这样的形式会使代码冗余，如果一个函数中总是调用一个命名空间中的其他函数，我们可以使用 using 声明来简化代码，语法：using namespace std；这样我们之后就可以直接调用 cout 而省去 std :: 了。

8. Const 关键字

Const 关键字用来使一个变量不可被更改，常用于常量的定义。

9. 引用

引用是一个对象的别名，是一种复合类型，通过在变量名前添加“&”符号来定义，操作引用就是操作对象本身，如 int a = 10; int &b = a; b++; 此时 a 的值已经变为了 11。与指针不同，指针定义后可以变换所指的對象，但引用一经定义，则不能改变与之相关联的对象。

10. C++ 中的函数传参方式

C++ 中的函数，有三种参数传递方式，传值，传引用，传指针。传值时，会在函数执行时为形参分配内存，等待函数执行完毕后，把

形参的值 return 出去赋给指定的变量，再将形参销毁，而传引用时，函数会直接操作引用，也就是直接操作引用所绑定的对象，不会为形参实际分配空间，函数执行完以后也不用再把值赋回去，因为操作的就是变量本身，效率很高。而传指针，类似于传值，会在函数内部产生一个形参作为实参的副本。操作指针形参时，真正的那个指针的值并没有发生变化，你要是不把形参指针的值再赋回去，相当于没有操作。

11. 操作符的重载

比之于 Java 只能重载函数，C++中可以重载操作符显得更为灵活，重载操作符的语法为：返回值类型 operator 关键字 要重载的运算符 (形参列表)，对于现在的我们，认识即可。抱佛脚不要求我们会使用。

12. 析构函数

与 Java 不同，C++采用手动内存管理，因此我们需要手动回收动态分配出去的内存，因此对于 C++类来说，它们还有一类特殊的函数，叫做析构函数，在我们使用 delete 关键字删除一个对象 (确切的说是删除指向这个对象的指针时) 时，析构函数会自动被调用，但并不是所有时候都需要显式的编写析构函数，只有当对象在其生命周期内或构造函数中获取资源时，我们才需编写以释放。

13. 模板类

我敢保证，你总是在 PPT 或者其他地方看见这么一行代码：

```
template <class Type>
```

这叫啥，这叫模板。模板有两种，一个叫模板函数，一个叫模板类，先说模板函数，如果有时候你需要定义一大堆的重载函数，而这些函数仅仅只有形参列表中的参数类型不一样（就是除过类型，形参个数一样，甚至连变量名都一样。），那此时就有一个简单的办法：只写一个函数，而使用一个暂不指出的类型把这些类型都替换掉，等需要用的时候再确定，给啥类型，函数中用 Type 标记的类型就是啥，非常灵活。这样的想法，用于实现函数，就叫函数模板，用于实现类，就叫类模板。

那咋在需要的时候指定类型呢？在函数模板中，编译器会推断出到底是在是用什么类型，我们直接给出实参就可以。而使用类模板时，必须为模板形参显式的指定实参。

语法：类名 <要用的类型> 对象名。

接下来，我将告诉你 C++极速入门的**至关重要**的一句话！你若遵循，技术必定**突飞猛进**！请注意，这个真谛是：

你看了上面的解说，最好**动手**写上一两个 C++程序，哪怕不比 helloworld 复杂多少都行，**否则**的话，你看了上面，基本等于**白看**！

第 1 章 概论

“蛋疼的数据结构.....”

1. 数据结构

要想方便的使用数据，你就需要把他们组织起来，特别是有大量的数据的时候要用的时候，你不得不组织。所以，抛开那些严谨的定义，通俗的来讲：组织数据的方法或者数据被组织起来的**形式**就是数据结构。

2. 分类

数据结构可以分为**线性结构**和**非线性结构**。线性结构也称为线性表，这种结构中的所有数据都按某种顺序排列在一个序列中。而线性表又分为顺序表和链表。非线性结构中各个元素不再保持在一个线性序列中，每个数据元素可能与零或多个其他数据元素发生关系，根据关系不同又分为层次结构和群结构，层次结构如树，群结构如图。

3. ADT

Abstract data type 即抽象数据类型，我们要使用一个非内置的数据类型时，就需要先对它进行设计，进行设计的时候应该关心这个数据类型应该包含那些信息，或者支持那些操作，而不是一开始就关注这个数据类型该如何实现，所以，好的做法是把数据类型抽象一下，把声明和实现分开，设计的时候考虑声明，用的时候在去实现。

4. 算法性能分析和度量

算法的复杂性度量属于事前估计，它可以分为时间复杂度和空间复杂度，常采用**大 O 表示法**来描述。

第 2 章 数组

“伟大的数组”

1. 简介

数组是具有相同类型的若干变量的有序组织形式。通常使用一块连续的内存。数组是线性结构，每个元素都有唯一的前驱和后继（第一个和最后一个元素除外），元素的个数和数组的起始地址必须在分配内存的时候就指定。数组中的元素可以被任意的直接访问，这个随机访问是通过数组的下标来实现的。

2. 一维数组元素地址推算公式

设 α 为该数组的起始地址，数组中每个元素要用 I 的存储空间，则：

$$\text{Addr}[i] = \alpha + i * I$$

3. 顺序表

定义：把线性表中所有的表项按照逻辑顺序依次存储到一块指定地址的连续的存储空间。显然数组是顺序表。顺序表类至少应该支持一下操作：

(1) 插入元素 (2) 移除元素 (3) 查找某元素的先驱 (4) 查找某元素的后继 (5) 判断是否为空 (6) 判断是否为满 (7) 按索引得到某一元素。

4. 顺序表的性能分析

主要是分析搜索，插入和删除运算的实现代码的时间复杂度。搜索算法中，设各个表项的搜索概率为 P_i ，找到该表项时数据比较次数为 C_i

搜索的平均比较次数为 ACN (average comparing number) =

$\sum_{i=1}^n P_i * C_i$, 若搜索表项的各项可能性相同, 则 $ACN = \frac{1}{n} \sum_{i=1}^n i =$

$$\frac{1}{n} (1+2+\dots+n) = \frac{1+n}{2}$$

分析顺序表的插入和删除的时间代价主要看循环内的数据移动次数 AMN

(average moving number)

插入 : $AMN = \frac{1}{n+1} \sum_{i=0}^n (n-i)$

若个表项插入概率相等, $AMN = \frac{n}{2}$

删除 : $AMN = \frac{1}{n} \sum_{i=1}^n (n-i)$

若个表项删除概率相等, $AMN = \frac{n-1}{2}$

5. 三角矩阵的存储

对于一个 $n \times n$ 对称方阵来说, 可以只存储它的上三角或者下三角部分来

节省空间, 而其上 (下) 三角矩阵的元素个数为

可以用一个一维数组来存放这些元素 :

a00	a10	a11	a20	a21	a22	a30	a31	a32	a33
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

$$\text{Loc}(i,j) = k;$$

$$k = \begin{cases} i(i+1)/2 + j & i \geq j \\ j(j+1)/2 + i & i < j \end{cases}$$

6. 对角矩阵的存储

$$\left\{ \begin{array}{cccccc} a_{00} & a_{11} & a_{02} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{42} & a_{43} & a_{43} & a_{45} \\ 0 & 0 & 0 & a_{53} & a_{54} & a_{55} \end{array} \right\}$$

a11	a12	a21	a22	a23	a32	a33	a34	an(n-1)	ann
-----	-----	-----	-----	-----	-----	-----	-----	-------	---------	-----

设第 i ($i \neq 1, n$) 行非零元素的个数为 m , 则 $d = \frac{m-1}{2}$

则带状矩阵中，非零元素的个数为： $(2d+1)n - (1+d)d$

7. 稀疏矩阵

使用一个三元组只存储矩阵中的非零元素，可以实现对稀疏矩阵的压缩，三元组： $\langle \text{行}, \text{列}, \text{值} \rangle$

对于一个稀疏矩阵，如果要对其进行转置，应当考虑在其压缩状态下直接对其进行转置，最简单的方法是把三元表内的 row 与 col 呼唤后，再对新的三元表进行 row 主序排序。

这里讲一种方法：假设稀疏矩阵 A 有 Cols 列，则对这个三元组进行 Cols 次扫描，第 k 次扫描是在三元组表中寻找列号为 k 的所有三元组，意味着此三元组应该放在新表中的第 k 行，此时交换该三元组的 row 和 col，再连同 value 一同存入新的三元表中。

为了提高效率，还有一种快速转置的方法，通过一如两个辅助数组来

提高效率 rowSize[]来记录每一列有多少个非零元素 ,rowStart[]来记录每行第一个三元组应该存放在新表中的什么位置

8. String 类的抽象数据结构

由零个或多个字符的顺序排列所组成的数据结构，基本组成元素是单个字符，

S0	S1	S2	S3	S4	Sn-1
----	----	----	----	----	-------	------

一个字符串的连续字符子集叫做子字符串，空字符串不包含任何字符。

实现可增长字符串有三种方法：(1) 创建一个字符缓冲区并且只初始化这个区域的一部分，剩下的缓冲区以后使用。(2) 创建一个新的第三字符串，其容量是另两个字符串的容量之和，并把这两个字符串的内容拷贝进来。(3) 创建一个字符串链表，而不是数组，这种方法是真正地无限制方法，但是需要维持链表

9. 字符串模式匹配

老师 PPT 中提到的有两种模式，一种是朴素匹配，另外一种 KMP 匹配。KMP 匹配算法因为无回溯，效率较高。(PPT 86 页)

10. STL 中的 string

STL 中已经给我们提供了 string 类供我们使用 ,只要#include <string>就可以使用了，此类包含了初始化，串接，获取长度，输入输出等基本操作的支持。string 类提供了字符串高级处理函数的集合。

11. STL 中的 vector

同样。只要#include <vector>就可以使用 STL 中的 vector 了 ,vector

提供了一个安全的对数组的替代，因为它提供成员函数来实现那些更高级的操作。实际上，可以简单的理解 vector 为一个大小可变的数组。

vector 引用的声明语法：

```
vector <type> v;
```

其中 type 叫做泛型，它制定了 vector 中存储的数据类型。我们可以用 vector 声明一个整形矩阵：

```
vector < vector <int> > matrix;
```

vector 有两个构造函数，vector <type> v 构造了一个空 vector，而 vector <int> v(5, 42)构造了一个包含 5 个值为 42 的元素的 vector。

vector 支持对项的随机访问，可以使用如下两种方式：v[]或者 v.at()，并且可以通过 vector.push_back()来向 vector 中添加新项

12. STL 中的 deque

同样。只要#include <deque>就可以使用 STL 中的 deque 了，deque 叫做双向队列，类似于 vector，但支持双向操作，deque 通过.push_front()和.pop_front()提供了在表的两端对元素进行高效插入和删除的操作。deque 占用连续的存储空间,并且在所存储元素的头和尾部又保留了额外的存储空间。

13. 习题:

1. We store a 4×4 symmetric (对称) matrix A into an array B (index start from 0)with row major order Store the lower triangle only, the index of element a[2][3] in B is 4.

把一个 4×4 的对称矩阵 A 存储到一个下标从 0 开始的数组 B 中，使用行主序只存储它的下三角部分，那么 a_{23} 在数组 B 中的索引是多少？

a_{11} a_{12} a_{13} a_{14} 因为是对称矩阵，

a_{21} a_{22} a_{23} a_{24} 所以 $a_{23} = a_{32}$ ， a_{32}

a_{31} a_{32} a_{33} a_{34} 是数组里的第 5 个元素，

a_{41} a_{42} a_{43} a_{44} 因此下标是 4。

2. () The worst-time for removing an element from a sequence list (Big-Oh) is **B** .
- a. $O(1)$ **b. $O(n)$** c. $O(n^2)$ d. $O(n^3)$

从线性表中移除一个元素，最坏情况下的时间复杂度是多少？

考虑最坏情况：移除顺序表的第一个元素，则后面所有元素都要向前移一位，这是个线性的时间花费，因此时间复杂度为 $O(n)$

3. We can use 3 vector type to store value and **location** of non-zero elements in a sparse matrix.

在稀疏矩阵中，我们可以使用一个三元 vector 来存储非零元的值和 位置。三元组中一个表项有三个变量，分别是 row,col,value。

第 3 章 链表

“程序员们用指针和数据做项链”

1. 简介

考虑到数组的长度固定不变，很可能造成内存浪费，也很可能不够用，不够用时扩展的开销很大，在数组中移除或者插入项的时间是线性的等等缺点，链表诞生了。

链表并不是用连续的存储空间，而是通过指针指向下一个节点的地址来实现**逻辑上的连续**，这样无论在链表的何处插入或者删除一项，所花费的时间都是恒定的，而且链表的大小是无限制的（只要内存够）。

链表的一个节点由它的数据项和指针组成，最后一个节点的指针为空指针，链表会有一个头结点，这个头结点仅仅是一个指向链表第一个节点的指针，使用头结点的目的是简化链表操作的实现，统一空表和非空表的运算。我们可以通过头结点访问整个链表。

链表的缺点是大多数方法需要声明，数组对数据元素的访问速度是链表不可比拟的，因为数组可以做到随机访问，但链表访问中部元素或者后部元素（对单链表来说）会慢的多，原因是链表**只能通过上一项找到下一项**。

2. 单链表的插入和删除元素

插入：①找到第 $i-1$ 个节点，并判断这个插入位置是否合法。

②分配内存给新的节点，并检查内存分配是否成功。

③让新节点的指针指向第 i 个节点

④使第 $i-1$ 个节点的指针指向新节点。

删除：①找到第 $i-1$ 个节点，并判断要删除的位置是否合法。

②使用一个指针指向将被删除的节点。

③让第 $i-1$ 节点的指针指向第 $i+1$ 个节点。

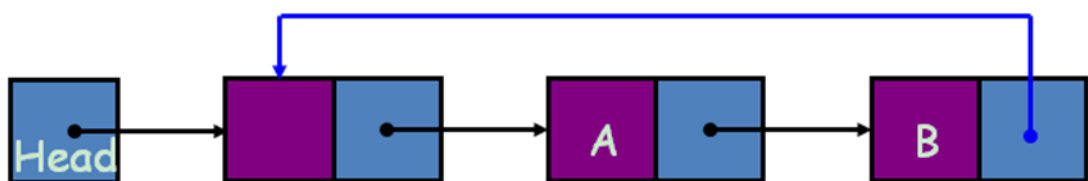
④delete 掉刚才指向那个指向被删除节点的指针。

3. 循环链表

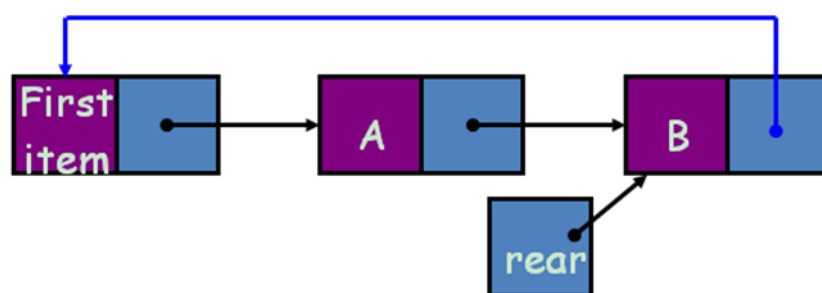
考虑到单链表的缺点，如单链表访问当前节点的先驱难等。因素，在单链表的基础上，又出现了循环链表和双向链表。

循环链表的尾节点的指针域指向头指针，大多数时候，为了在头部插入方便，循环链表不定义头指针而是定义尾指针 rear。

附加头结点型循环链表



使用尾指针 rear 的循环链表

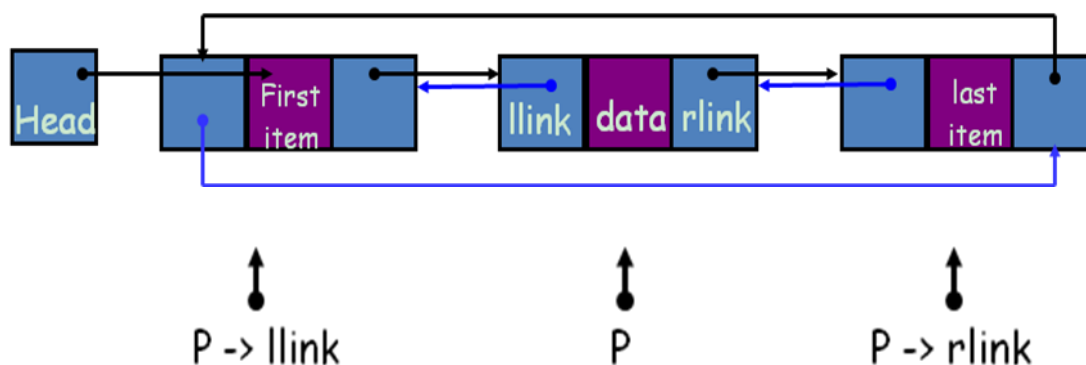


循环链表的应用：约瑟夫问题。

4. 双向链表

双向链表的诞生是为了解决在链表中直接访问前驱的问题，双向链表中每个节点都有两个指针，一个指针指向前驱结点，一个指针指向后

继节点。一个双向链表结构是这样的：



容易发现：

$p == p \rightarrow llink \rightarrow rlink == p \rightarrow rlink \rightarrow llink$

删除节点的关键代码：

$current \rightarrow rlink \rightarrow llink = current \rightarrow llink;$

$current \rightarrow llink \rightarrow rlink = current \rightarrow rlink;$

在 $current$ 前驱方向插入节点的关键代码：

$newnode \rightarrow llink = current \rightarrow llink;$

$newnode \rightarrow rlink = current;$

$current \rightarrow llink = newnode;$

$newnode \rightarrow llink \rightarrow rlink = newnode;$

5. 静态链表

如果我们不是动态的为每一个节点分配内存，而是先分配好一个数组（每项含有两个值，一个存值，一个存指针），并把数组的下标看做地址，那么我们可以得到一个叫做静态链表的东西。如图：

地址（下标） 数据 指针

0	NULL	7
---	------	---

1	周	4
2	钱	5
3	王	NULL
4	吴	6
5	孙	8
6	郑	3
7	赵	2
8	李	1

6. 习题:

1. Which of the following is true

- ① we can random access an element in array.
- ② we can random access an element in linked list.
- ③ we cant random access both in array and linked list.
- ④ all above is wrong

答案：①。数组时可以通过下标做到随机访问的，也就是对其中的任意一个元素的访问，不需要依赖于对其他元素的访问，但是链表是做不到这一点的，除过头结点，要想访问链表中的任何一个元素，必须先访问它的先驱。

2. Write a function to insert an element into a single linked list with header node.

解题思路：PPT 上有这个方法的代码，这里就不占篇幅了。

第 4 章 栈和队列

“食堂打饭要是后来先买，那一定会非常有意思”

1. 简介

栈可以定义为只允许在表的**末端**进行插入和删除的线性表，允许插入和删除的一端叫做**栈顶**，另一端则叫做**栈底**，当栈中没有数据元素时，栈成为空栈。栈也叫后进先出的线性表。栈应该提供以下操作：(1)检查是否为空；(2)检查是否为满；(3)返回栈顶的数据元素；(4)将一个新的数据元素压入栈中；(5)将栈顶的数据元素弹出；(6)展示栈中所有的数据元素。

2. 栈的实现

任何列表的实现都可以用来实现栈：例如数组或者链表，基于数组表示的栈叫做**顺序栈**，基于链表表示的栈叫做**链式栈**。

3. 进栈

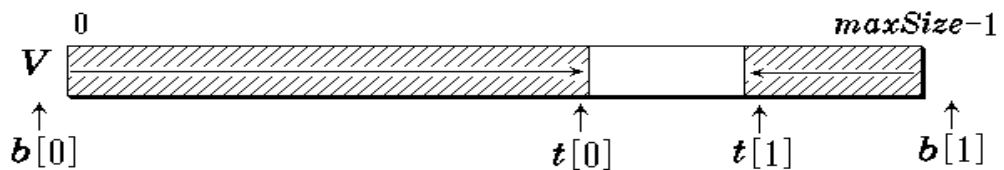
进栈时，首先要判断栈内是否还有剩余空间，对于一个栈来说，栈的最后允许存储的位置是 $\text{MaxSize}-1$ ，如果栈顶指针已经 $\text{top} == \text{MaxSize}-1$ ，则说明所有位置均已使用，栈已满，这时若再有新的数据元素进栈，则会发生溢出，程序会转入溢出处理；若 $\text{top} < \text{MaxSize}-1$ ，则先让栈顶指针加 1，指到可加以加入新元素的位置，再按栈顶所指到的位置将新元素插入。

4. 出栈

如果出栈时发现栈是空栈，即 $\text{top} = -1$ ，则退栈操作将执行空栈处理，若当前 $\text{top} \geq 0$ ，则将栈顶元素弹出，并将栈顶指针减 1。读取栈顶元素值的函数 $\text{getTop}(T\&)$ 与退栈函数 $\text{Pop}(T\&)$ 的区别在于：前者没有改变栈顶指针的值，后者改变了栈顶指针的值。

5. 双向栈

程序中往往同时存在几个栈，而各个栈所需的运行空间是动态变化的，有的栈膨胀的快，有的栈可能还会有许多空闲空间，因此给几个栈分配同样大小的空间并不明智，因此产生了多个栈共享栈空间的思想。这里只介绍双向栈。



双向栈定义了一个足够大的栈空间，，空间的两端是两个栈的栈底，两个栈的栈顶都向中间延伸，直至两个栈顶相遇时，才认为栈发生了溢出，但由于这里的栈是使用数组表示的，那么在栈顶压入一个数据实际上是在数组中的某个位置插入了数据，当数据量很大时，这种插入操作的开销非常大，因此下面介绍链式栈。

6. 链式栈

采用链式栈来表示一个栈，便于节点的插入与删除，链式栈的栈顶在表头，新节点的插入和栈顶节点的删除表头进行。

7. 栈的应用

(1) 括号匹配 (2) 表达式计算

8. 队列简介

队列是另一种限定存取位置的线性表。它只允许在表的**一端**插入，在**另一端**删除，允许插入的一端叫做队尾，允许删除的一端叫做队头。队列具有**先进先出**的特性。

队列的存储表示也有两种方式；一种是基于数组的存储表示，叫做**顺序**

队列，另一种是基于链表的存储表示，叫做链式队列。

9. 顺序队列

利用一个一维数组作为队列的存储结构，并设两个指针 front 和 rear。在队列刚刚建立的时候，对队列进行初始化 $\text{front} = \text{rear} = 0$ ，每当加入一个新元素的时候，先将元素添加到 rear 所指的地方，然后再让 rear 指针进 1。当要退出队头元素的时候，先记录此元素的值，然后再将 front 指针进 1。然后返回记录下来的。

由于刚才所讲的队列可能会产生一种假溢出的问题，即 front 前面还有空间可用。所以设计了循环队列来弥补这一缺陷。

循环队列把数组的前端和后端连起来，形成一个环形的表，front 和 rear 指针进到 $\text{MaxSize}-1$ 后，再进 1 就回到了 0。这可以用取余算法来实现。

队头指针进 1： $\text{front} = (\text{front} + 1) \% \text{MaxSize}$ ；

队尾指针进 1： $\text{rear} = (\text{rear} + 1) \% \text{MaxSize}$ ；

可以用 $\text{front} == \text{rear}$ 来判断队列是否为空，用 $(\text{rear} + 1) \% \text{MaxSize} ==$ 来判断队列是否为满，即 rear 若指到 front 的前一个位置，队列即满，这实际上空出了一个位置用来区分满与空，循环队列最多只能放下 $\text{MaxSize}-1$ 个元素。

10. 链式队列

队列的队头指针指向单链表的第一个节点，队尾指针指向单链表的最后一个节点。用单链表表示的链式队列特别适合与数据元素变动比较大的情况，而且不存在队列满而溢出的情况。

11. 习题

1.(**D**)In a circular queue we can distinguish(区分) empty queues from full queues by _____ .

- A using a gap in the array
- B incrementing queue positions by 2 instead of 1
- C keeping a count of the number of elements
- D a and c

- A 在数组中使用一个空元素作为间隔
- B 增加两个队列位置而不是一个
- C 保持对元素的个数的计数
- D a 和 c**

A 正确 ,因为循环队列的普遍做法是当 rear 指向 front 的前一个位置时 ,就认为队列满了 , 因此会空出一个位置 , 故正确。B 我不明白他说的啥意思
C 显然正确 , 元素个数达到所能存储的最大值就满了么 , 个数为零就是空的么 , 因此选 D

2. A **stack** is a list where removal and addition occur at the same end . Frequently known a LIFO (Last-In-First-Out) structure.

添加和移除都发生在列表的同一端 , 并且是后进先出的结构 , 这显然是栈么。

3 . Consider the following function to balance symbols stored in string exp that includes parentheses (圆括号) and numbers. Please fill in blank.

```
#include <stack>

using namespace std;

int matching(string &exp) {

//exp is a pointer to a string to check

    int state = 1,i=0;

    char e;

    stack <char> s;

    while ( i<exp.length() && state )

        switch (exp[i]) {

            case '(':

                s.push('(') ;

                i++;

                break;

            case ')':

                if ( !s.empty() ) {

                    s.pop(); i++; }

                else

                    state = 0;           //an error occurs

                break;
```

default:

i++; break;

} //end of while

if (s.empty()&&state) return 1;

else return 0;

第 5 章 递归 (Recursion)

“要理解递归，你必须先理解递归”

1. 什么是递归？为什么需要递归？

Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first

有时候，解决一个问题的最佳途径是先着手解决这个问题的规模情形。

Recursion is a technique that solves a problem by solving a smaller problem of the same type

递归是一种技巧，它通过解决同类型的小问题来完成问题的解决。

2. 递归函数

来看一个递归函数的例子：

```
Int Function(int n) {  
    If (n==1 || n==2) return 1; //(1)  
    Return Function(n-1)+Function(n-2); //(2)  
}
```

<1> 该函数只接收一个参数 n，返回 Fibonacci 数列第 n 项的值。

<2> 该函数用到了递归的设计，因为求 Fibonacci 数列第 n 项的值这个问题，可以分割为同类型的子问题：求其第 n-1 项和第 n-2 项的值，并求和。

<3> 其中(1)被称为 base case，是因为它很容易被计算。(2)被称为 general (recursive) case，它类似于一种递推。

<4> 每种递归算法至少要各有一个 base case 和 general case。

<5> 如果没有 base case , 递归函数将导致栈溢出(stack overflow)。

3. 递归与迭代(Iteration)

<1> Iteration can be used in place of recursion

迭代可以用来代替递归

An iterative algorithm uses a *looping structure*

一个迭代算法使用循环结构。

A recursive algorithm uses a *branching structure*

一个递归算法使用分支结构。

<2> Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions

比起迭代 , 递归通常都是低效率的 , 在时间和空间上都是。

<3> Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

递归可以简化一个问题的解决 , 通常都会带来更简短、更通俗易懂的源代码。

<4> (Nearly) every recursively defined problem can be solved iteratively, iterative optimization can be implemented after recursive design

(几乎)每一个以递归方式定义的问题都可以用迭代方式解决 , 在递归设计之后可以实现迭代优化。

4. 递归的应用

<1> 分治法(divide and conquer)

Eg:归并排序，快速排序。

<2> 回溯法(backtracking)

Eg:求解八皇后问题。

5. 递归的算法复杂度分析

<1> $T(n)=C_1 \cdot T(n/2)+C_2 \cdot n$ 型(C_1 、 C_2 均为常数)

每次递归数据规模折半,所以是 $\log n$ 趟,每趟又有 n 次(只看数量级,所以 $C_2 \cdot n$ 等价于 n),所以此类型递归的算法复杂度为 $O(n \log n)$ 。

<2> $T(n)=C_1 \cdot T(n-1)+C_2 \cdot n$ 型(C_1 、 C_2 均为常数)

每次递归数据规模减一,所以是 n 趟,每趟又有 n 次(只看数量级,所以 $C_2 \cdot n$ 等价于 n),所以此类型递归的算法复杂度为 $O(n^2)$ 。

6. 习题

<1> $T(n) = 2T(n/2) + cn$ $T(n) = T(n-1) + cn$

$T(n) = O(?)$ $n \log n$ $T(n) = O(?)$ n^2

<2> (**B**) A recursive function can cause an infinite sequence of function calls if _____.

一个递归函数能导致无数的函数调用如果_____

A. the problem size is halved at each step

问题规模在每一步减半。

B. the termination condition is missing

终止条件丢失。

C. no useful incremental computation is done in each step

在每一步没有有用的增量计算。

D. the problem size is positive

问题规模是确定的。

第 6 章 树 (Tree)

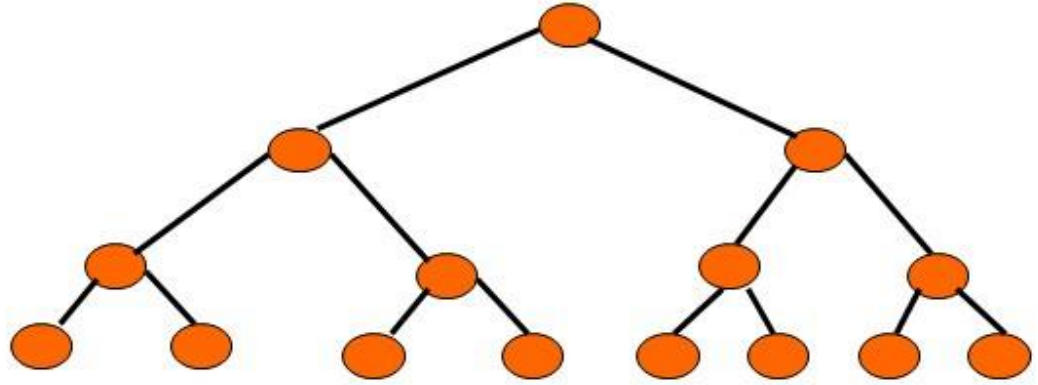
“植树节到了，大家一起种一棵平衡二叉搜索树吧”

1. 关于树的一些术语

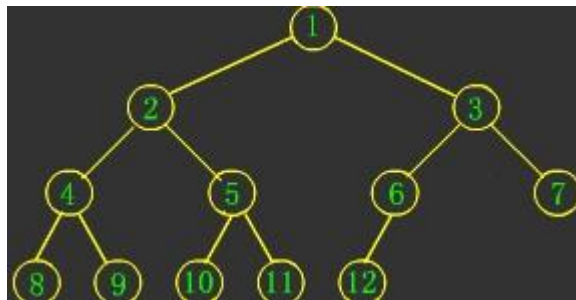
- <1> 根 (Root) :没有双亲的结点。
- <2> 兄弟 (Siblings) :有共同双亲的结点。
- <3> 内结点 (Internal node) :至少有一个孩子的结点。
- <4> 外结点 (External node) 或叶子 (leaf) :没有孩子的结点。
- <5> 一个结点的祖先 (Ancestors) : 双亲、双亲的双亲.....
- <6> 一个结点的后裔 (Descendant) :孩子、孩子的孩子.....
- <7> 一个结点的深度 (Depth) :其祖先的数目。
- <8> 一棵树的高度 (Height) :树中任意结点中最大的深度。
- <9> 一个结点的度 (Degree) :其孩子的数目。
- <10> 一棵树的度 : 树中任意结点中最大的度。
- <11> 子树 (Subtree) :包含指定结点和它的后裔的一棵树。

2. 二叉树 (Binary Tree)

- <1> 二叉树是度为 2 的树，即每个结点最多有两个孩子。
- <2> 二叉树内结点的两个孩子分别被称作左孩子和右孩子。
- <3> 满二叉树 (Full binary tree) 是具有 $2^{k+1}-1$ 个结点高度为 k 的二叉树。(即每层都是满的，如下图所示)



<4> 完全二叉树 (Complete binary tree) 是指一棵高度为 k 的二叉树 ,
除第 k 层外 , 其他各层的结点树都达到最大 , 第 k 层所有结点都连续集中
在最左边。(如下图所示)



3. 二叉树的遍历

<1> 先序遍历 (Preorder traversal)

In an preorder traversal a node is visited before its left subtree and
right subtree

在先序遍历中 , 每个结点在它的左子树和右子树之前被访问。

遍历顺序 : 当前结点->左子树->右子树。

Algorithm PreOrder(v)

if(v 非空) {

 visit(v) //标记访问结点 v

 if LeftChild (v) //如果 v 有左孩子

```

        PreOrder (leftChild (v)) //递归遍历左子树

        if rightChild(v) //如果 v 有右孩子

            PreOrder (rightChild (v)) //递归遍历右子树

    }

```

<2> 中序遍历 (Inorder traversal)

In an inorder traversal a node is visited after its left subtree and before its right subtree

在中序遍历中，每个结点在它的左子树之后和右子树之前被访问。

遍历顺序：**左子树->当前结点->右子树**。

Algorithm InOrder(v)

```

if(v 非空) {

    if LeftChild (v) //如果 v 有左孩子

        InOrder (leftChild (v)) //递归遍历左子树

    visit(v) //标记访问结点 v

    if rightChild(v) //如果 v 有右孩子

        InOrder (rightChild (v)) //递归遍历右子树

}

```

<3> 后序遍历 (Postorder traversal)

In an postorder traversal a node is visited after its left subtree and right subtree

在后序遍历中，每个结点在它的左子树和右子树之后被访问。

遍历顺序：**左子树->右子树->当前结点**。

Algorithm PostOrder(v)

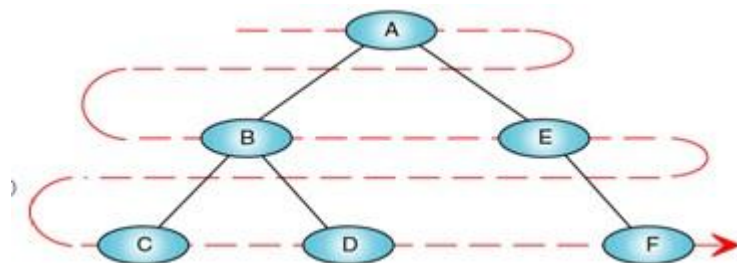
```
if(v 非空) {  
    if LeftChild (v) //如果 v 有左孩子  
        PostOrder (leftChild (v)) //递归遍历左子树  
    if rightChild(v) //如果 v 有右孩子  
        PostOrder (rightChild (v)) //递归遍历右子树  
    visit(v) //标记访问结点 v  
}
```

以上三种遍历 ,均为二叉树的**深度优先遍历**(Depth-First Traversals, DFS)

<4> 广度优先遍历 (Breadth-First Traversals , BFS)

BFS 也称作层次遍历，一般用**队列**来实现层次遍历，遍历过程如图

所示：



关键代码：

```
current = q.DeQueue ( ); //退队  
if ( current->leftChild != NULL ) //左子女
```



```
q.Enqueue ( current→leftChild);    //进队列  
  
if ( current→rightChild != NULL )    //右子女  
  
q.Enqueue ( current→rightchild );    //进队列
```

4. 二叉树的线索化 (Thread)

<1> 关于线索化

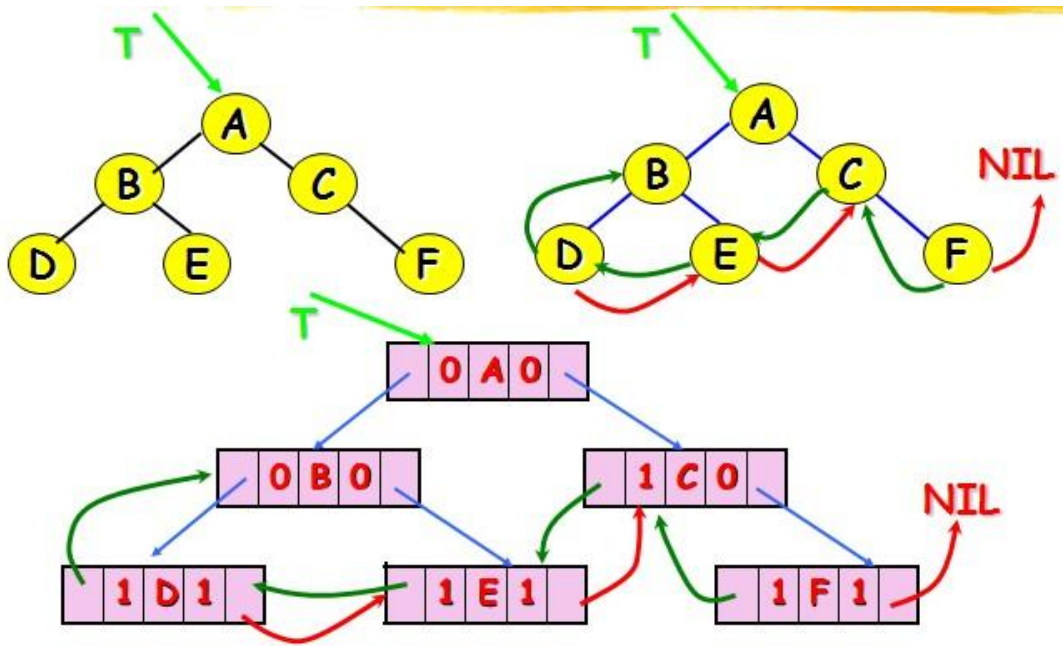
二叉树浪费了许多空间，每个根节点都有两个空指针。我们可以利用这些空指针来帮助我们遍历二叉树，可以用这些空指针指向遍历顺序的后继结点，这个过程就叫做线索化。

<2> 三种线索化

根据遍历种类的不同，线索化分为：先序线索化、中序线索化、后序线索化。

<3> 线索化的具体过程 (只以先序线索化为例说明)

如下图所示：



先写出**先序遍历顺序**：ABDECF

红箭头表示指向后继结点，绿箭头表示指向前驱结点。

每个有空指针位置的结点，右侧连接红箭头，左侧连接绿箭头。

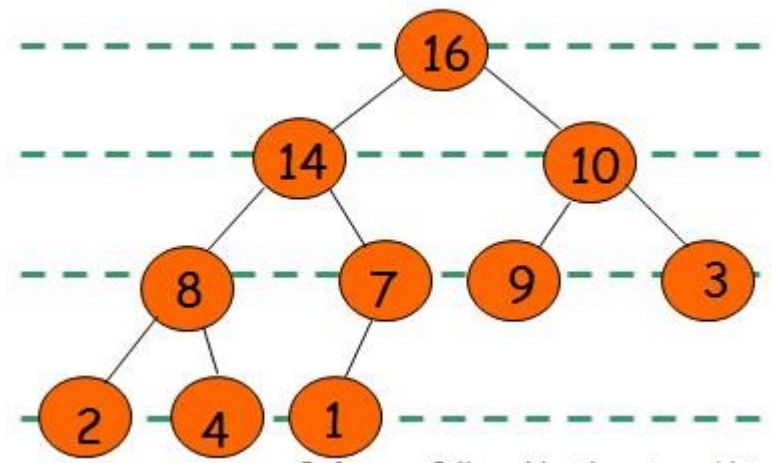
每个结点最多连接两个结点，如果没有前驱或者后继，则指向 NIL，表示空。

5. 优先队列 (Priority Queue)

<1> 特点

每次出队的元素是所有元素中**优先级**最高的。

<2> 用二叉堆 (Binary Heap) 实现



如上图所示，二叉堆其实是一棵**完全二叉树**，可以分为大根堆（Max-Heap）和小根堆（Min-Heap），大根堆的任何结点的权值都比其孩子结点的权值大，小根堆相反。

堆的一些基本操作（实现自《算法导论》）：

```
DATA_TYPE PARENT(int i) { //求双亲结点的索引
```

```
    return (int)(i/2);
```

```
}
```

```
DATA_TYPE LEFT (int i) { //求左孩子的索引
```

```
    return 2*i;
```

```
}
```

```
DATA_TYPE RIGHT (int i) { //求右孩子的索引
```

```
    return 2*i+1;
```

```
}
```

```
void MAX_HEAPIFY(DATA_TYPE *A,int i) {
```

```
    //保持大根堆性质
```

```
    int l,r,largest;
```

```

DATA_TYPE temp;

l = LEFT(i);

r = RIGHT(i);

if (l<=HEAP_SIZE && A[l]>A[i]) largest = l;

else largest = i; //选取三个之中权值最大的

if (r<=HEAP_SIZE && A[r]>A[largest]) largest = r;

if (largest != i) {

    temp = A[i];

    A[i] = A[largest];

    A[largest] = temp;

    MAX_HEAPIFY(A,largest); //递归保持堆性质

}

}

void BUILD_MAX_HEAP(DATA_TYPE *A,int len) {

//自底向上建立堆

    HEAP_SIZE = len;

    int i;

    for (i=(int)(len/2);i>=1;i--) {

        MAX_HEAPIFY(A,i);

    }

}

```

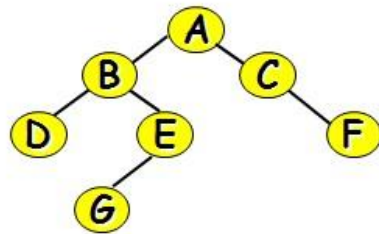
6. 哈夫曼树 (Huffman Tree)

<1> 一些术语

路(path)、路长(path length)、结点的 路长(path length of a node)、

二叉树的路长 (path length of a binary tree)

具体关系如下图：



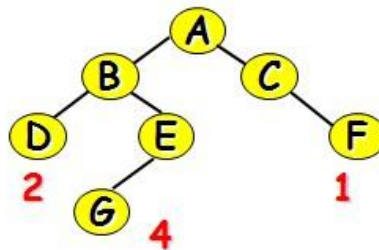
Path from A to G: $\langle A, B \rangle \langle B, E \rangle \langle E, G \rangle$
Pathlength from A to G: 3
Pathlength of node G: 3
Pathlength of a tree: $2 + 3 + 2 = 7$

结点的权值 (Weight of a node)、带权结点的 路长 (Path length of a

node with weight)、带权树的路长 (Path length of a tree with

weights)、总权值 $WPL = \sum w_i \times L_i$ 。

具体关系如下图：

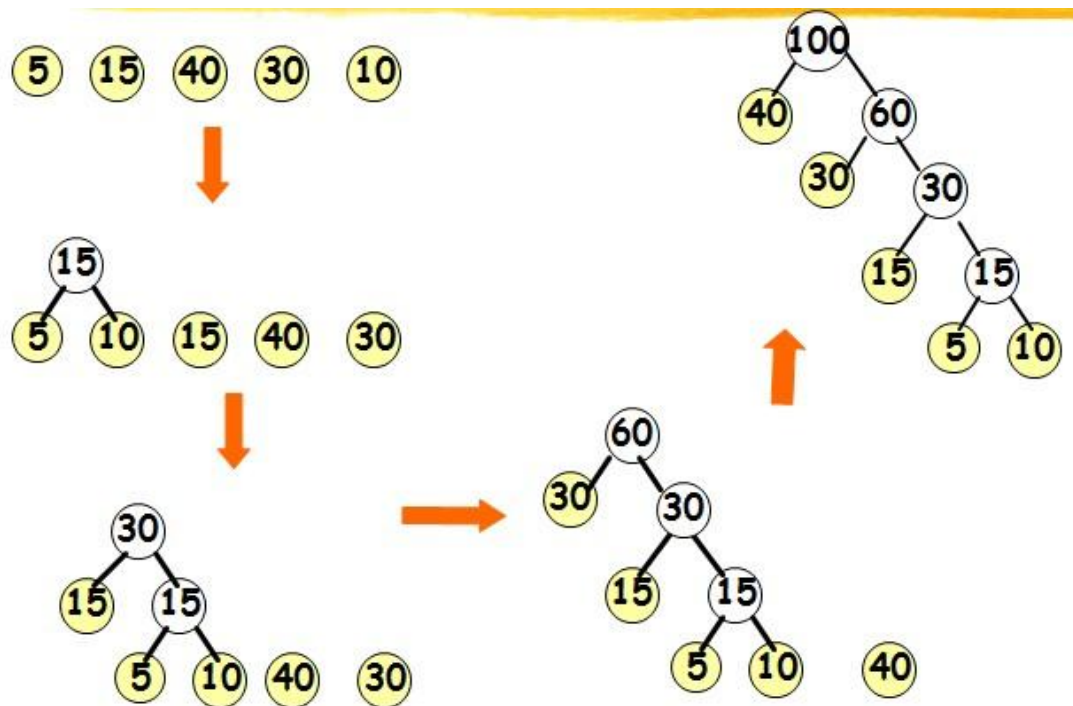


Path length of node G with weight:
 $3 \times 4 = 12$
Path length of the tree with weights:
 $2 \times 2 + 3 \times 4 + 2 \times 1 = 18$

<2> 哈夫曼树的构造

具体算法及证明请参考《离散数学》和《算法导论》，此处用简单图

示说明问题：

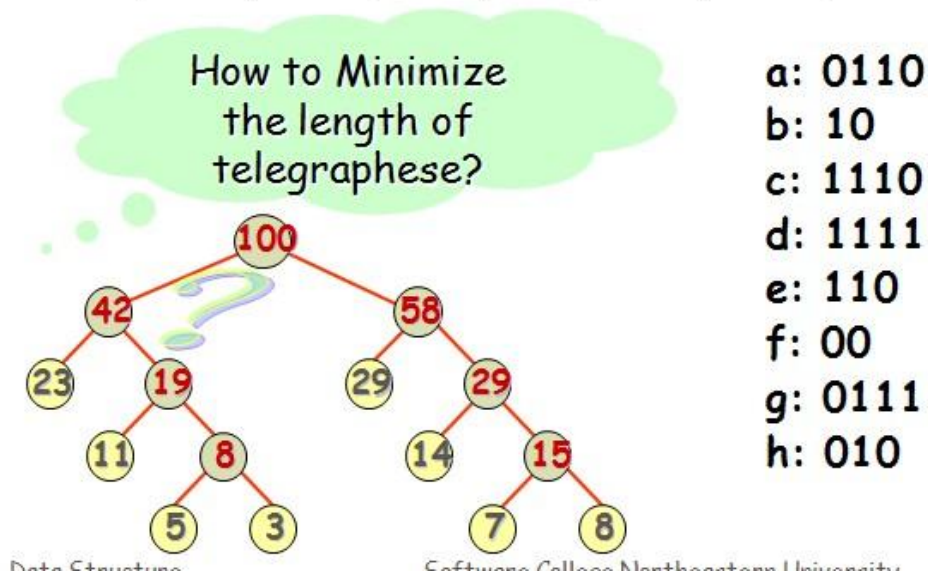


每次从待选点集中选出两个权值最小的点，作为左右儿子，其双亲结点的权值等于二者之和，并把双亲结点作为新节点加入待选点集，原来选出的两个点从待选点集中移除。

<3> 哈夫曼编码 (Huffman Code)

先将欲编码的字符集按照概率分布建立哈夫曼树，左子树代表 0，右子树代表 1，逐个编码即可。具体如下图：

a	b	c	d	e	f	g	h
0.05	0.29	0.07	0.08	0.14	0.23	0.03	0.11



7. 习题

<1> () There is a binary tree whose elements are characters.

Preorder list of the binary tree is “ABECDFGHIJ” and inorder list of the binary tree is “EBCDAFHIGJ”. Postorder traversal sequence of the binary tree is EDCBIHJGFA.

现有一棵元素是字符的二叉树，先序遍历的顺序是“ABECDFGHIJ”并且中序遍历的顺序是“EBCDAFHIGJ”。那么这棵二叉树的后序遍历顺序是

EDCBIHJGFA.

解题思路：从先序遍历入手，第一个被访问的元素 A 肯定是树根，再到中序遍历顺序中找 A，以 A 为界把序列分而为二，则左边 EBCD 是以 A 为根左子树的元素，FHIGJ 是以 A 为根右子树的元素。再看先序遍历顺序，到 B，则 B 为根，用 B 划分 EBCD，得到左子树是 E，右子树是 CD。

同理递归得到所有划分，便可画出整棵树，再进行后序遍历写出顺序即可。

<2> () The full binary tree with height 4 has C nodes.

a. 15 b. 16 c. 31 d. 32

高度为 4 的满二叉树有 C 个结点。

解题思路：熟悉满二叉树定义即可， $\text{结点数} = 2^{(\text{高度}+1)} - 1$ 。

<3> There are $(n+1)/2$ leaf nodes in a full binary tree with n nodes.

拥有 n 个结点的满二叉树有 $(n+1)/2$ 个叶节点。

解题思路：熟悉满二叉树结构即可直接得出。

<4> Show the results of inserting 53,17,78,09,45,65,87 each , one at a time, in a initially empty max heap (大根堆)

写出依次将 53,17,78,09,45,65,87 插入一个空的初始化的大根堆里的结果。

解题思路：依次插入，每插入一个节点，保持一次堆性质。

<5> Write a function lenFarthestTwoDegree that returns the length of a farthest path from the root to a two degree node. Use the following function header:

`int lenFarthestTwoDegree(BiTree T)`

Please write the definition of binary tree and implement the function.If

Binary Tree is NULL,return -1., return 0. If the root is the node we need

编写一个函数 lenFarthestTwoDegree 返回根结点到最远的双度结点(即

左右孩子都有) 的距离。使用以下函数头：

```
int lenFarthestTwoDegree(BiTree T)
```

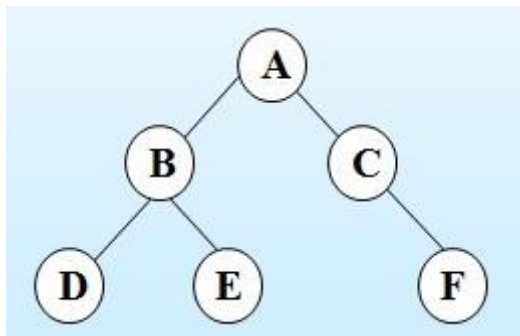
请写出二叉树的定义并且实现这个函数。如果二叉树为空，返回-1,。如果根结点即是所求 (最远的二度结点)，则返回 0.

解题思路：直接 DFS 整棵树 (先序遍历即可)，维护一个最大的距离和这个最远的二度结点，每深入一层，距离便加一。

方法还有很多，BFS 也行。

<6> write the sequence of preorder,postorder traversals and add inorder threads in the tree.

写出这棵树的先序、中序和后序遍历顺序，并且将树中序线索化。



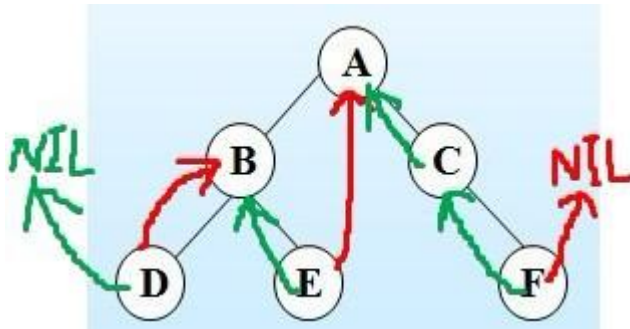
解题思路：

先序遍历：ABDECF

中序遍历：DBEACF

后序遍历：DEBFCA

中序线索化：



红箭头和绿箭头的定义见<二叉树的线索化>

<7> Write efficient functions (and give their Big-Oh running times) that take a pointer to a binary tree root T and compute:

- The number of leaves of T

写一个高效的函数 (并且给出它的大 O 运行时间) 来让一个指针指向二叉树的根 T 并计算 :

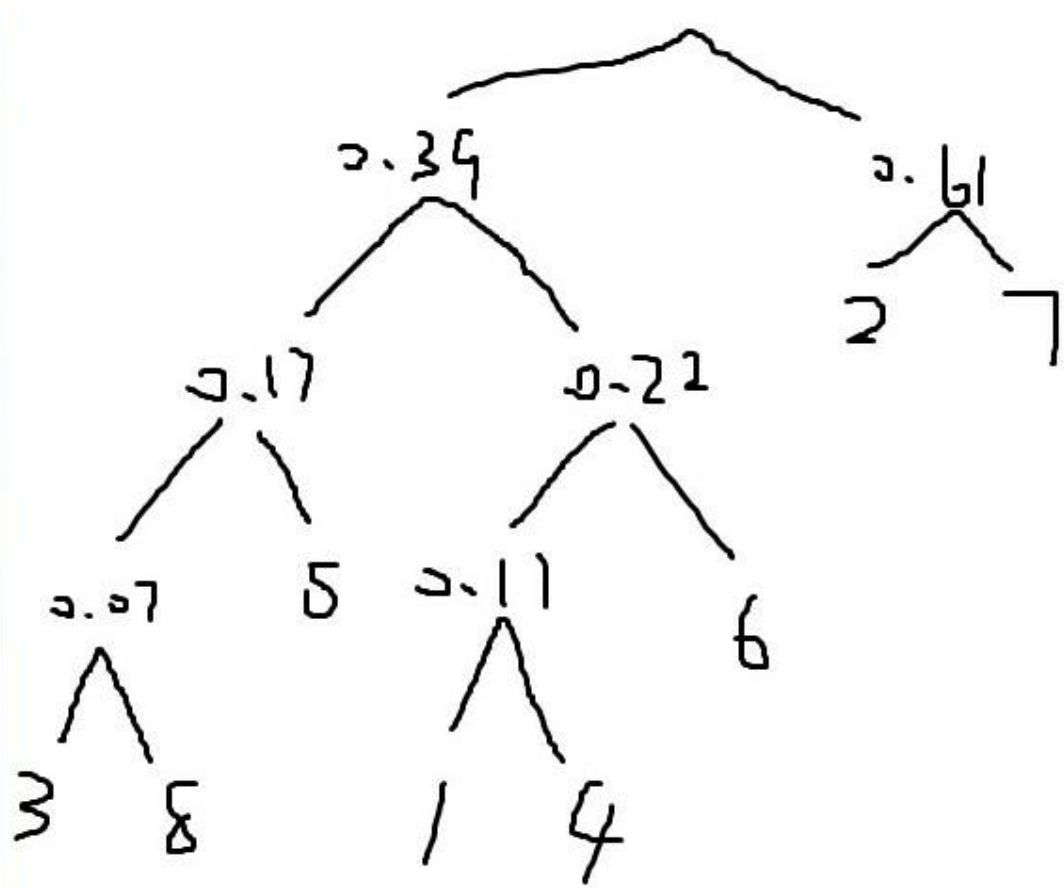
- T 的叶子结点的数目

解题思路 : 直接 DFS , 每次走到尽头 , 统计数目加一。

<8> Build a Huffman tree and determine Huffman code when the probability distribution(概率分布) over the 8 alphabets (c1, c2, c3, c4, c5, c6, c7, c8) is (0.05, 0.25, 0.03, 0.06, 0.10, 0.11, 0.36, 0.04

建立一棵哈夫曼树并且确定其哈夫曼编码 , 8 个字母 (c1, c2, c3, c4, c5, c6, c7, c8) 的概率分布分别为 (0.05, 0.25, 0.03, 0.06, 0.10, 0.11, 0.36, 0.04)。

解题思路 : 根据哈夫曼树建立方法建立哈夫曼树 :



哈夫曼编码为：

C1	C2	C3	C4	C5	C6	C7	C8
0100	10	0000	0101	001	011	11	0001

第 7 章 搜索 (Searching)

“百度一下，你就知道”

1. 一些术语

<1> 搜索表 (Search Table) : 装有待搜索元素的集合。

<2> 静态搜索表 (static search table) : 表中元素不会改变的搜索表。

<3> 动态搜索表 (dynamic search table) : 表中元素可能被改变的搜索表。

<4> 关键字 (key) : 一个或多个对象属性构成的组，用以唯一标识一个对象。

<5> 平均查找长度 (Average Searching Length , ASL) :

$ASL = \sum P_i C_i$ ($i=1,2,3,\dots,n$)。其中 P_i 为查找表中第 i 个数据元素的概率， C_i 为找到第 i 个数据元素时已经比较过的次数。

2. 顺序搜索 (Sequential Searching)

<1> 存储

搜索表：装有无序元素的数组。

<2> 算法

顺序遍历数组，直到发现一个合适的结果匹配 (查找成功)，或者一直到达数组的结尾 (查找失败)，或者可以通过在结尾设立哨兵值来判断失败。

<3> 优点和缺点

优点：

算法简单。

搜索表中的元素无序排序。

缺点：

搜索过程浪费大量时间，时间复杂度 $O(n)$ 。

3. 二分搜索 (Binary Search)

<1> 说明

二分搜索使用分治法的策略。

要求搜索表中的元素**必须有序**。

<2> 存储

搜索表：装有有序元素的**数组**。

<3> 算法

记当前要搜索的元素位于搜索表索引 left 与 right 之间，每次取其中点 **$mid=(left+right)/2$** 与要搜索的元素 e 的 key 值进行比较，若成功匹配，则查找成功，返回。若 $e.key > table[mid].key$ ，则 **$left=mid+1$** (范围折半)；若 $e.key < table[mid].key$ ，则 **$right=mid-1$** (范围折半)。若 $left > right$ ，算法终止，表示查找失败。

<4> 优点和缺点

优点：

算法高效，时间复杂度 $O(\log n)$ 。

缺点：

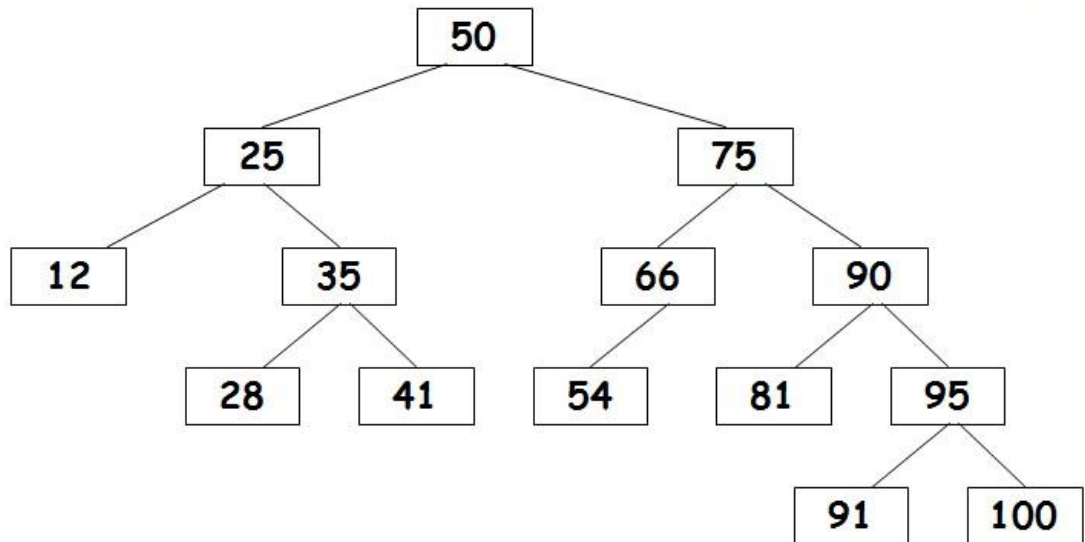
需要保证搜索表中的元素是有序的。

4. 二叉搜索树 (Binary Search Tree , BST)

<1> 定义

对于任何树中结点，其权值总是不小于其左子树中任一元素权值，并且总是不大于其右子树中任一元素权值的二叉树。

如下图所示：



5. AVL 树 (Adelson-Velskii and Landis Tree , Balancing Binary Search Tree)

<1> 一些术语

一个结点的平衡因子 (Balance factor)：以该结点为根的左子树高度减去右子树高度。

<2> AVL 树的性质

任何结点的平衡因子的绝对值不大于 1.

空子树的高度记为-1.

AVL 树是二叉搜索树，满足其所有性质。

<3> 插入元素和旋转保持平衡

插入操作可能导致某些结点的平衡因子变为 2 或-2.

只有处在插入点到根结点之间路径上的结点的高度才有可能发生改变。

所以在每次插入操作后，从插入点沿着路返回根结点，依次更新其高度。

如果一个新的平衡因子是 2 或 -2，则需要通过**旋转 (rotation)** 结点来调整整棵树的平衡。

<4> 插入元素的情形分类

共有 2 类 4 种插入情形：

第 1 类：外部情形 (Outside Cases)，需要单旋转 (single rotation)

第 1 种：插入左子树的左孩子 (LL)。

第 2 种：插入右子树的右孩子 (RR)。

第 2 类：内部情形 (Inside Cases)，需要双旋转 (double rotation)

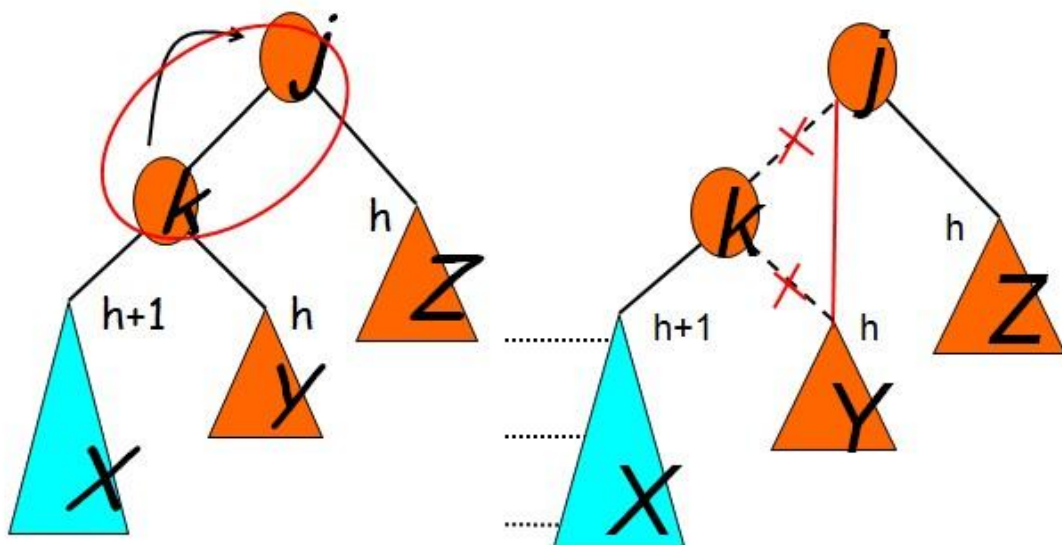
第 1 种：插入右子树的左孩子 (RL)。

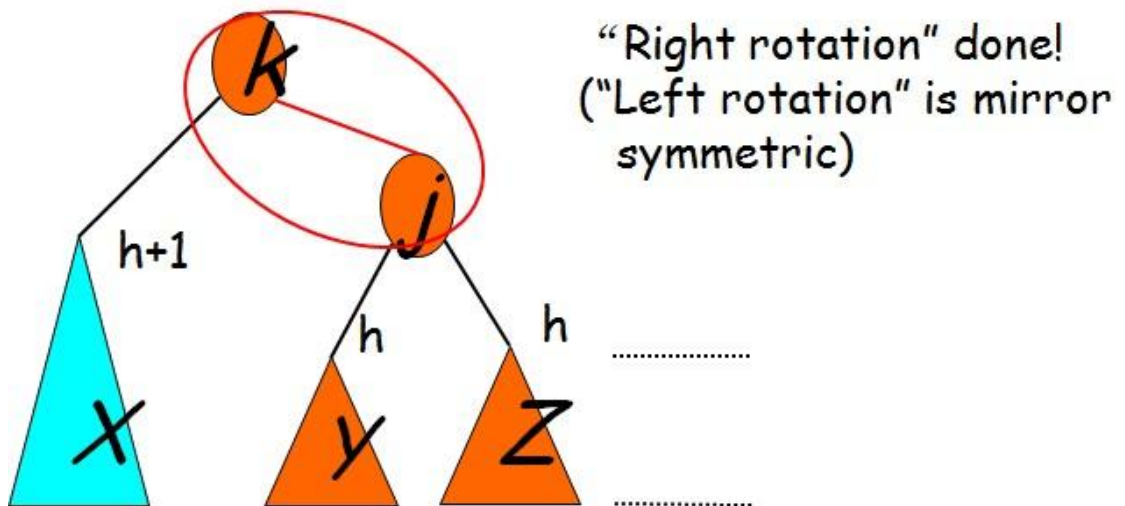
第 2 种：插入左子树的右孩子 (LR)。

<5> 插入情形对应的旋转算法

LL：只做一次右旋。

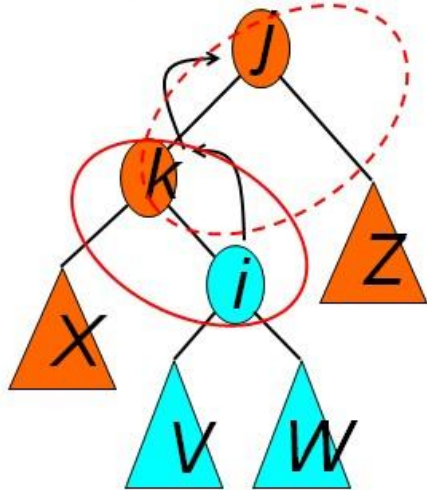
Do a "right rotation"



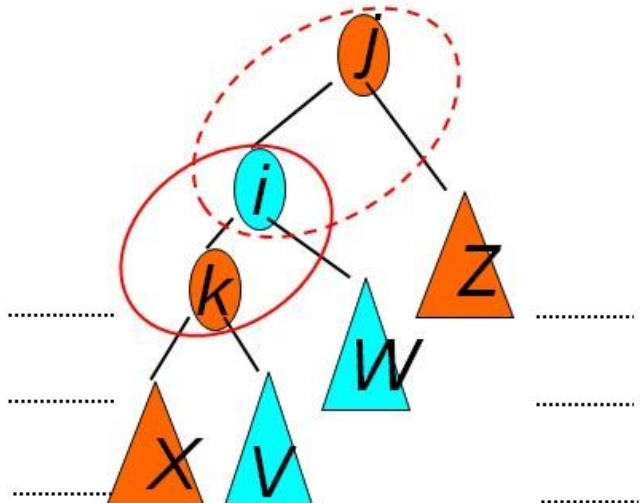


LR：进行左右双旋，即先左旋，后右旋。

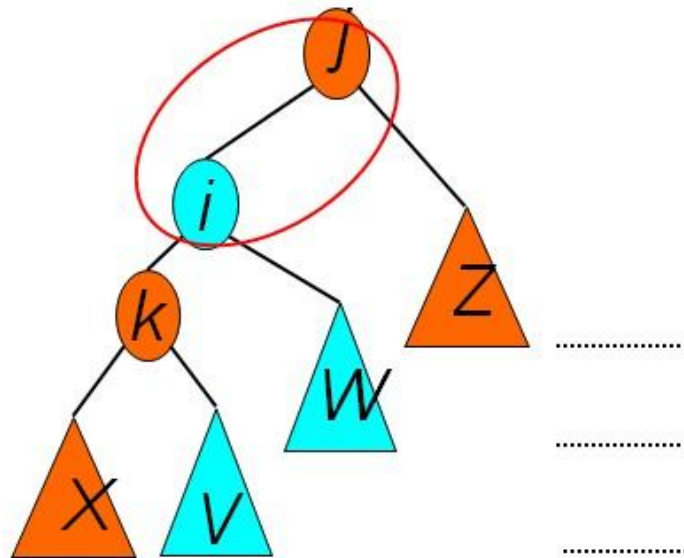
We will do a **left-right**
 “double rotation”...



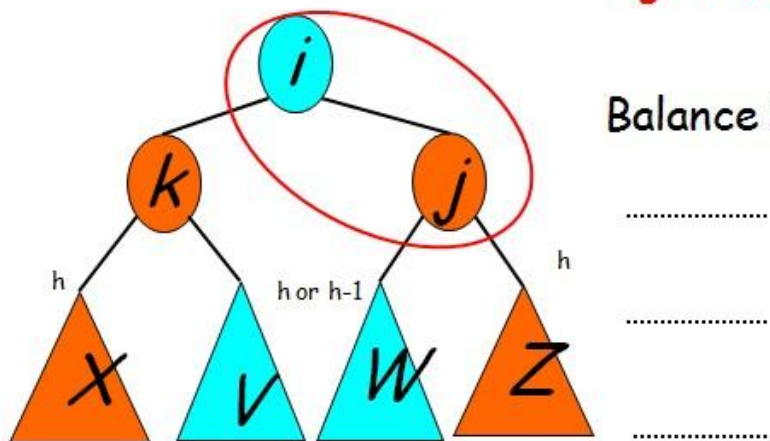
left rotation complete



Now do a right rotation



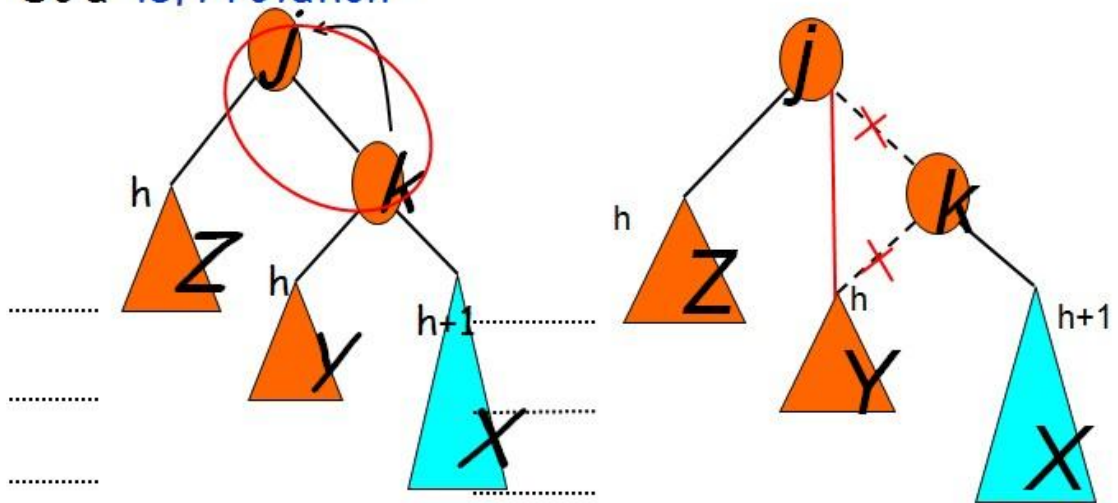
right rotation complete



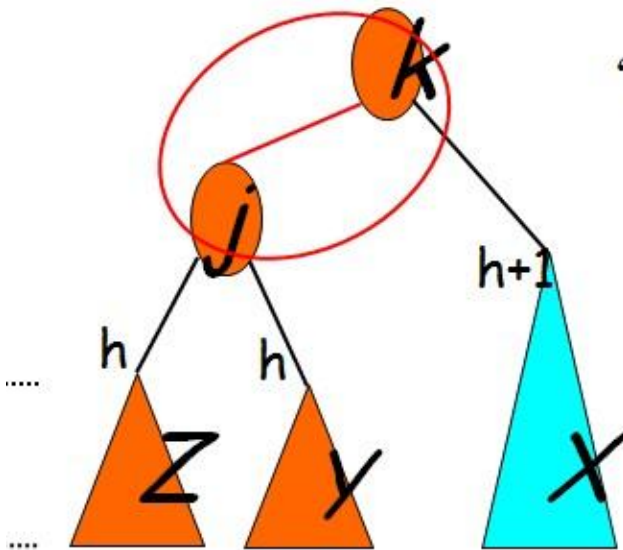
Balance has been restored

RR : 只做一次左旋。

Do a "left rotation"



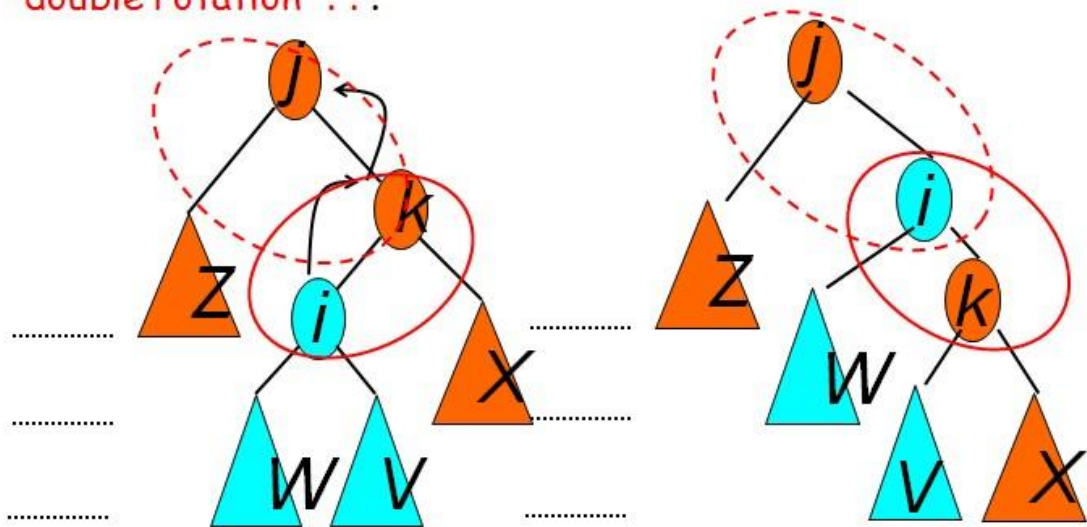
"left rotation" done!



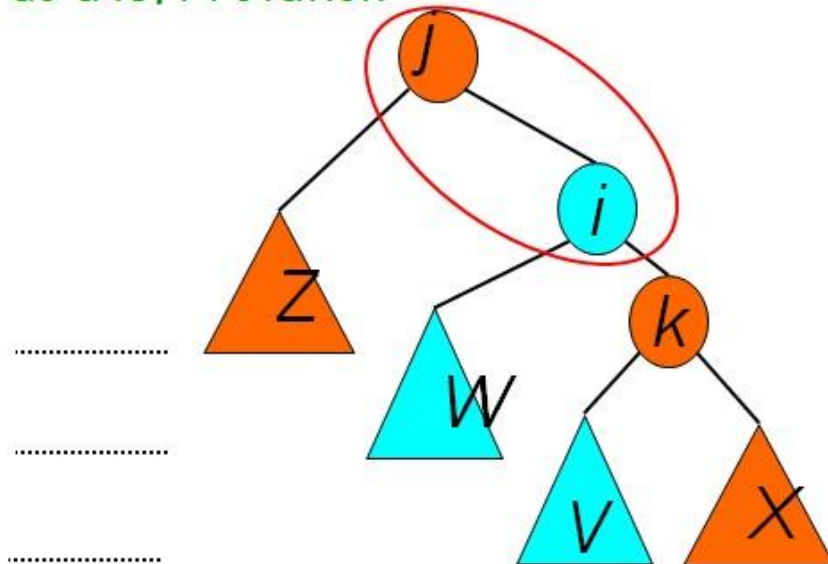
RL：进行右左双旋，即先右旋，再左旋。

We will do a right-left
"double rotation" ...

right rotation complete

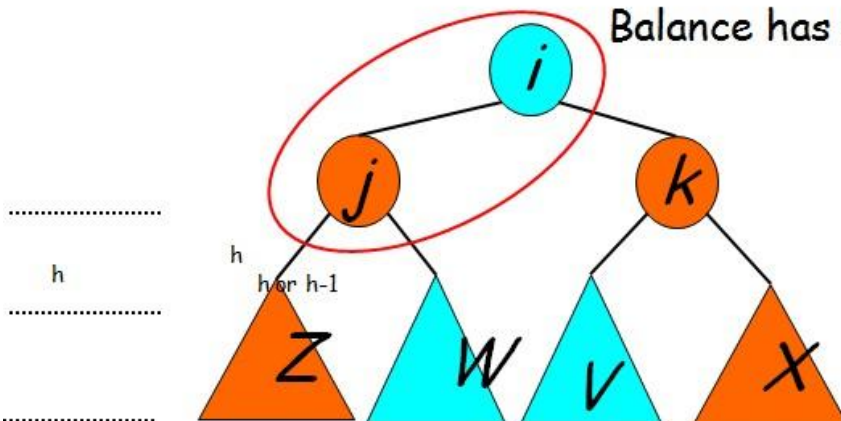


Now do a left rotation



left rotation complete

Balance has been restored



6. 习题

<1> (**B**) Searching in an unsorted list can be made faster by using _____ .

A.binary search

B.a sentinel (哨兵) at the end of the list

C.linked list to store the elements

D.a and c

通过采取下列哪种措施，可以使一个针对无序序列的搜索加快速度？

A . 二分搜索

B . 在序列末端设置哨兵

C. 用链表来存放元素

D. A 和 C

解题思路：因为序列无序，故二分搜索无法使用，排除 A、D。

只能使用顺序搜索，所以用链表来存储元素并不会提高访问效率，排除 C。

<2> Insert the following keys to a AVL Tree:

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11

Show the results of the AVL tree

将下列值插入 AVL 树中：

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11

画出 AVL 树的结果。

解题思路：

依次插入结点，每次插入都要使树满足二叉搜索树。

若在一次插入之后平衡因子变为 2 或 -2，则需要旋转调整（根据插入的不同分类：LL、LR、RL、RR, 执行不同的调整策略）。

详细步骤：

a). 插入 3

3

b). 插入 7



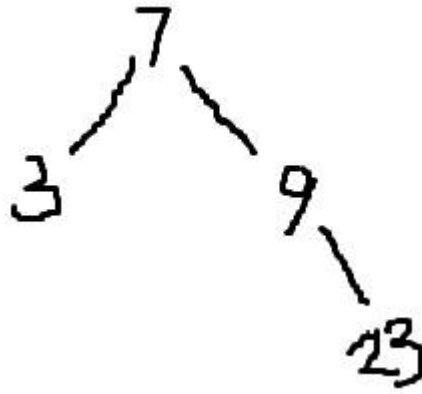
c). 插入 9



结点 3 的平衡因子变为了 -2，需要调整，插入方式是 RR，进行左旋。



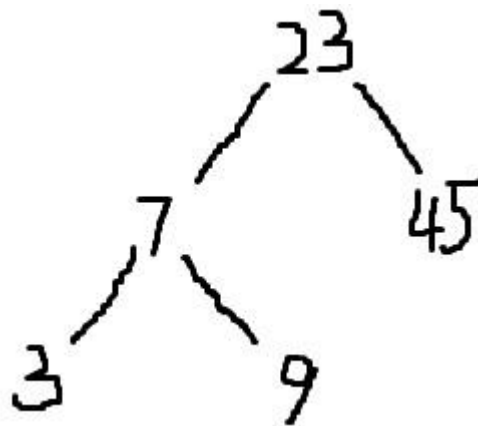
d). 插入 23



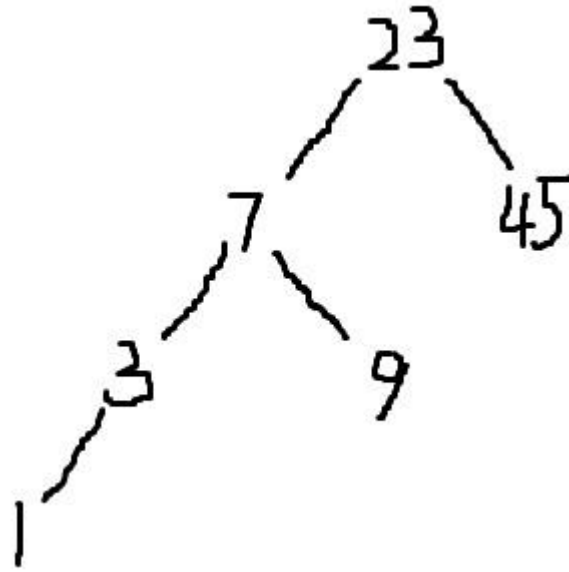
e). 插入 45



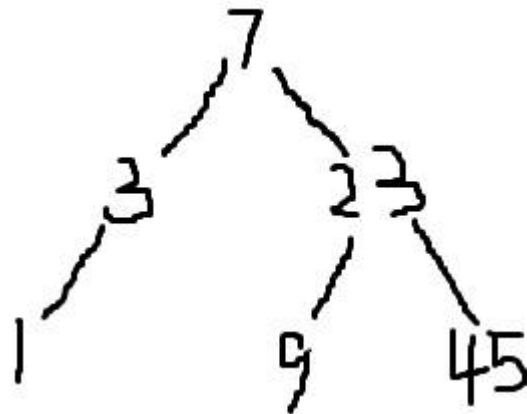
结点 9 的平衡因子变为了 -2，需要调整，相对 9 的插入方式是 RR，进行左旋。



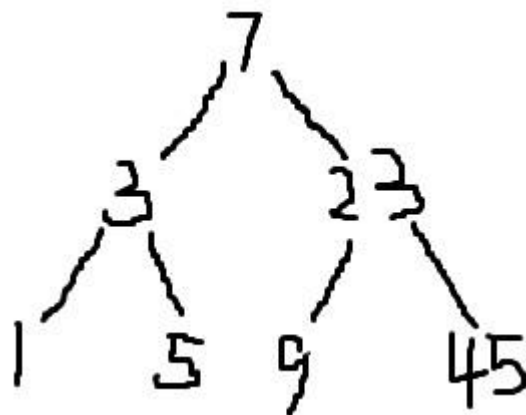
f). 插入 1



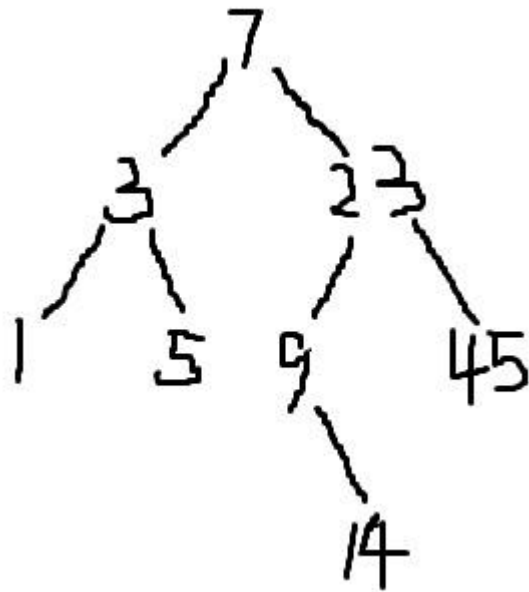
结点 7 的平衡因子变为了 2，需要调整，相对 7 的插入方式是 LL，进行右旋。



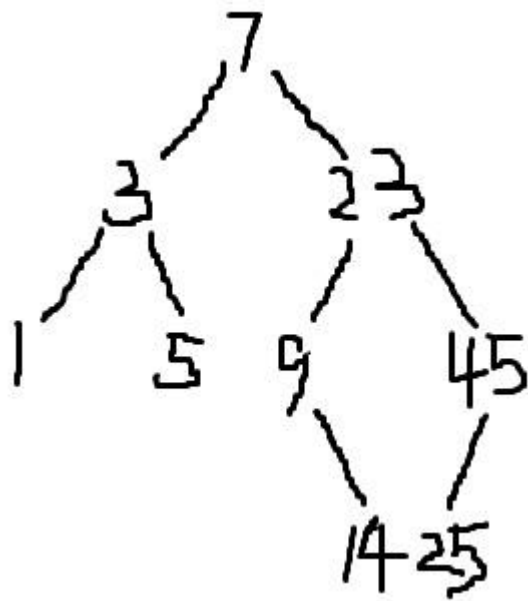
g). 插入 5



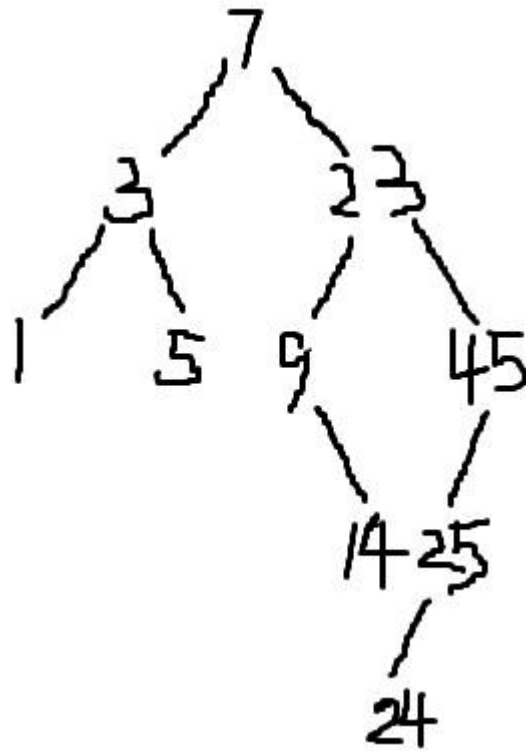
h). 插入 14



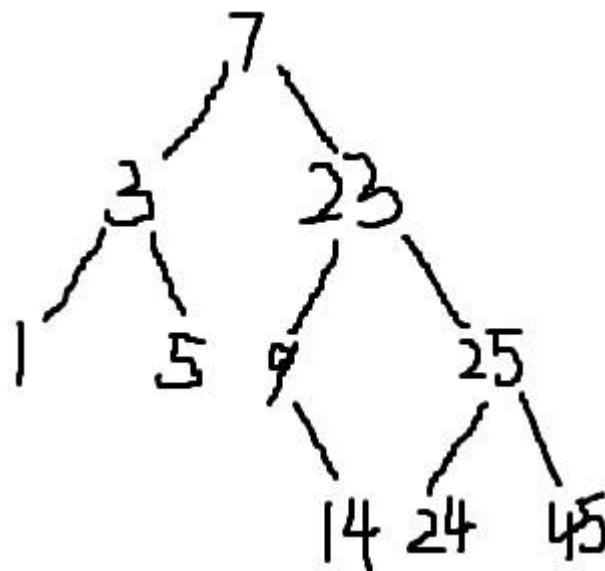
i). 插入 25



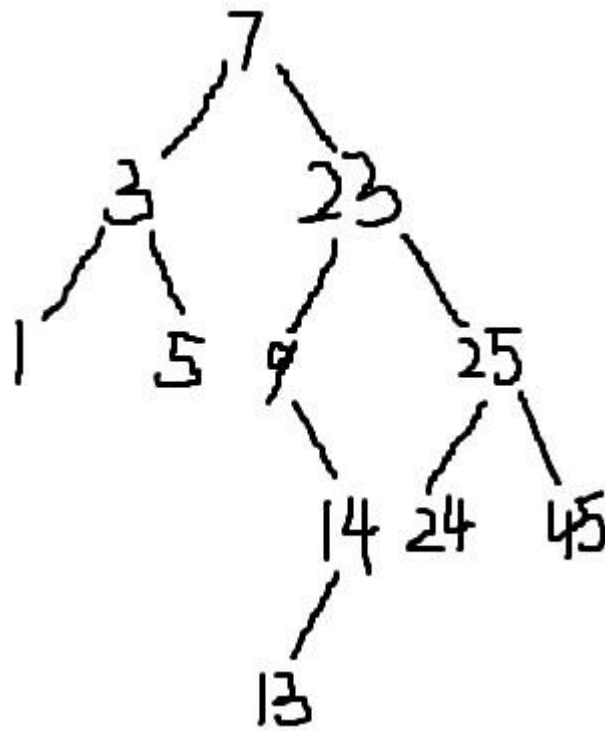
j). 插入 24



结点 45 的平衡因子变为了 2，需要调整，相对结点 45 的插入方式是 LL，进行右旋。

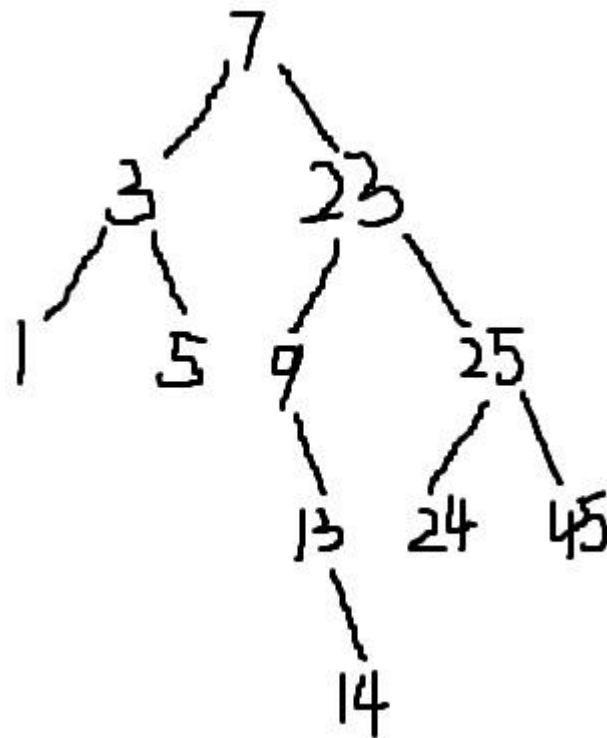


k). 插入 13

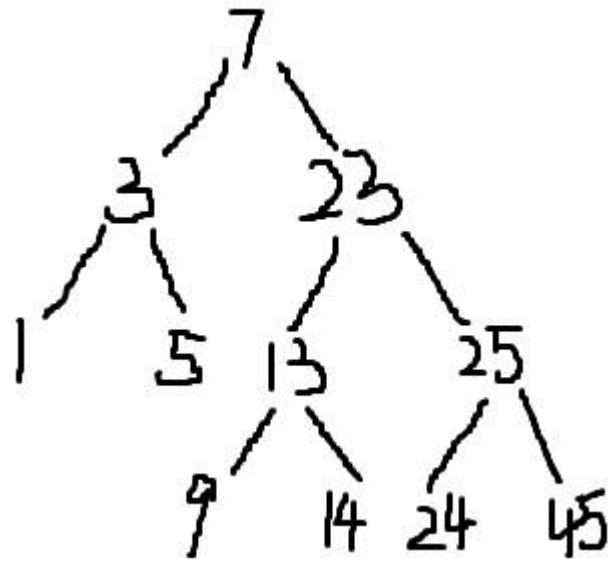


结点 9 的平衡因子变为了-2，需要调整，相对结点 9 的插入方式是 RL，进行先右旋后左旋。

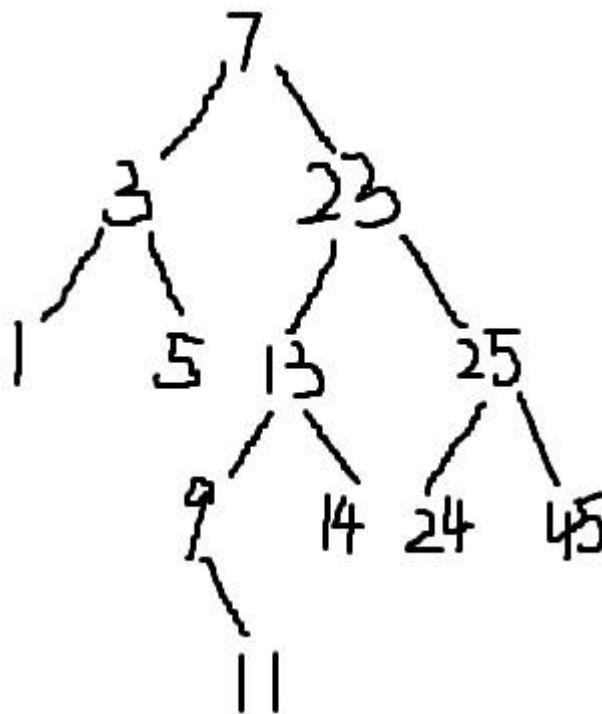
右旋结果：



左旋结果：

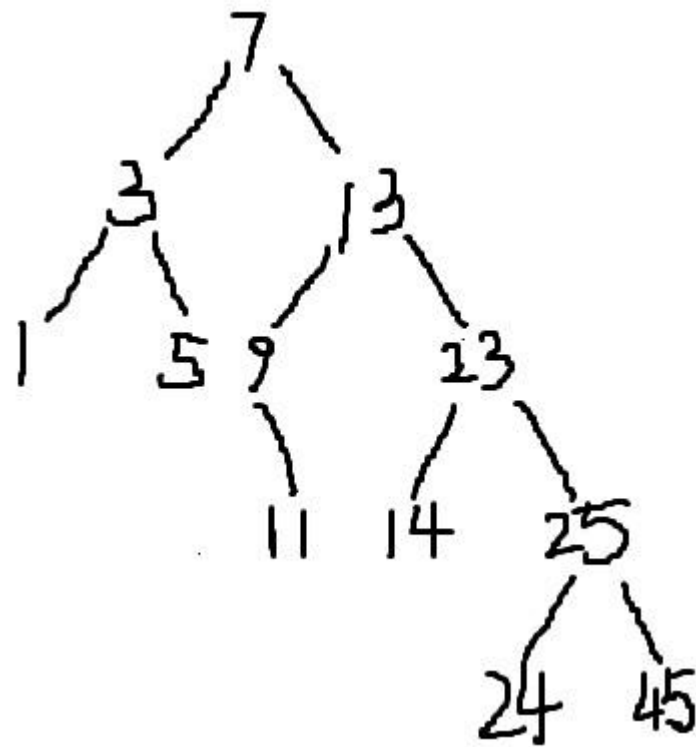


1). 插入 11

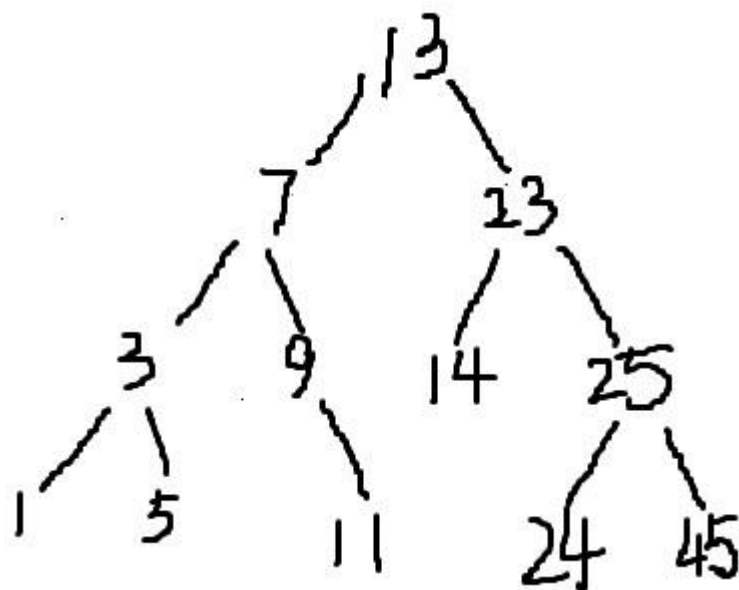


结点 7 的平衡因子变为了-2 ,需要调整 ,相对结 7 的插入方式是 RL ,
进行先右旋后左旋。

右旋结果：



左旋结果：



至此，本题 AVL 树构造完毕。

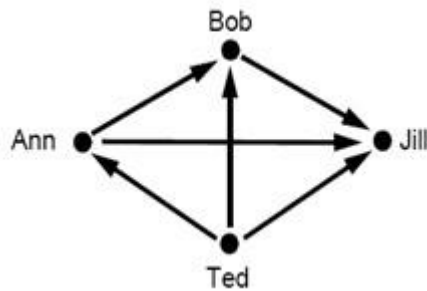
(手画了两个小时，图丑求别黑)

第 8 章 图 (Graph)

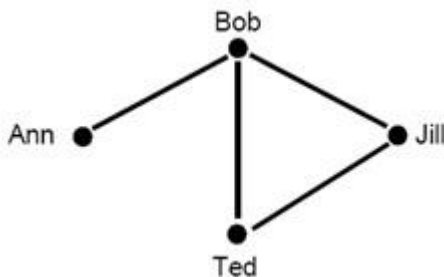
“图论是一门优雅的艺术”

1. 一些术语

<1> 有向图 (directed graph): 所有边单向导通。



<2> 无向图 (undirected graph): 所有边双向导通。



<3> 完全图 (complete graph): 任何两个结点之间都有边直连可达。

一个 n 个结点的无向完全图具有 $n(n-1)/2$ 条边。

一个 n 个结点的有向完全图具有 $n(n-1)$ 条边。(二倍关系)

<4> 权 (weight):

有时候边具有权，此时该图便被称作带权图 (weighted graph)，有向无向均可。

<5> 子图 (Subgraph)

设 V 和 E 分别是图 G 的结点集和边集，令 V' 为 V 的子集， E' 为 E 的子

集且 E' 中任何边两端的结点都属于 V' ,则称由 V' 和 E' 构成的新图 G' 为原图 G 的一个子图。

<6> 结点的度 (degree)

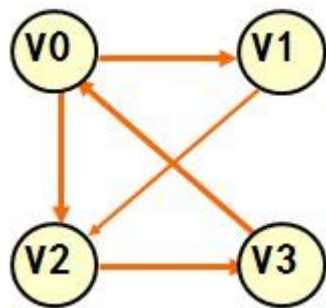
与一个结点相连的边的数目称为该结点的度。

在有向图中 ,从该结点射出的边的数目称为该结点的出度(Out-degree) ,

向该结点射入的边的数目称为该结点的入度 (In-degree)。

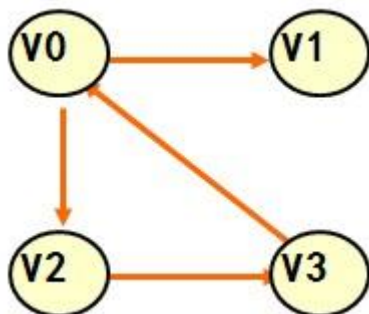
<7> 强连通图 (strongly connected graph) :

一个强连通图是指一个有向图 , 它的所有结点与其他结点都有路可通。



<8> 弱连通图 (weakly connected graph) :

一个弱连通图是指一个有向图 , 它并不满足强连通 , 但是如果把所有有向边都看成无向边 , 它就是连通的。

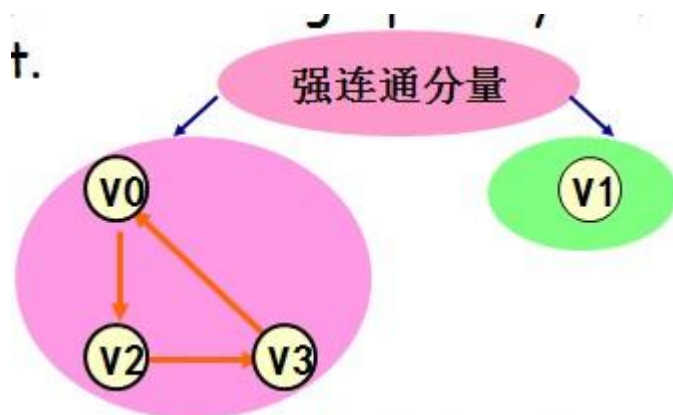


<9> 强连通分量 (strongly connected component) :

一个有向图的强连通分量是指一个最大的结点集合 , 满足 : 集合内任意

一个点到集合内任意其他点**总有路可通**。

一个强连通图只有一个强连通分量，即其点集 V 。



<10> 无向图的生成树 (spanning tree) :

设 G 是一个无向连通图，其一个**包含所有结点的连通无环子图**即为 G 的一棵生成树。

每棵具有 n 个结点的生成树都一定有 $n-1$ 条边。

如果我们在棵生成树上任意加一条边，都将会产生一个环。

<11> 最小生成树 (minimal spanning tree, MST) :

边权值总和最小的生成树称作一个无向图的最小生成树。

2. 图的存储方式

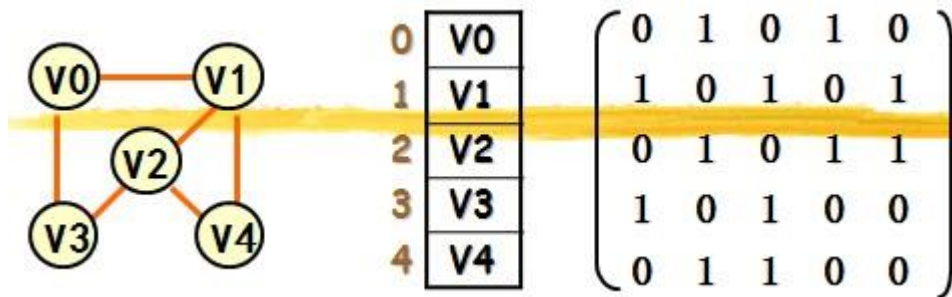
<1> 邻接矩阵 (adjacency matrix) :

定义一个 $n \times n$ 的矩阵 A ，结点从 $0 \sim n-1$ 标记。(n 为图的结点数目) :

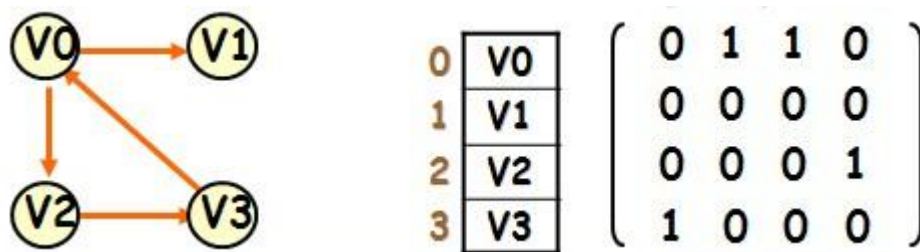
$$A.Edge[i][j] = \begin{cases} 1, & \text{if } \langle i, j \rangle \in E \text{ or } (i, j) \in E \\ 0, & \text{otherwise} \end{cases}$$

如果边有权值 ($W(i,j)$ 表示连接结点 i 和 j 的边的权)，则上面定义中的 1 可替换为 $W(i,j)$ 。

无向图的邻接矩阵表示 : (对称阵)

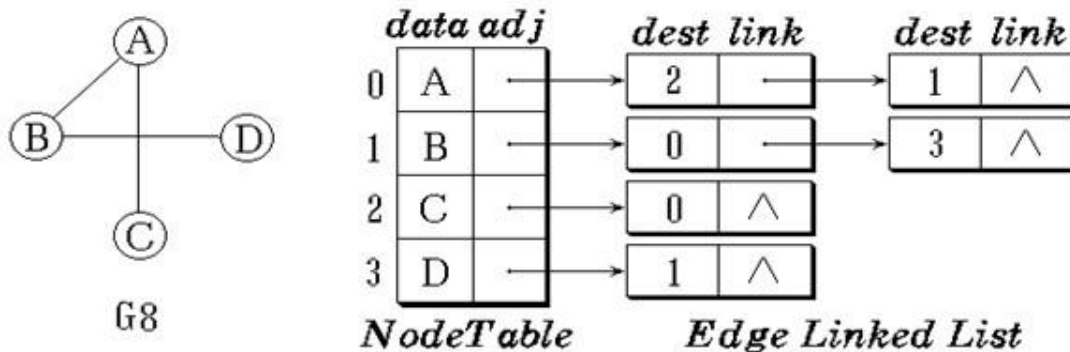


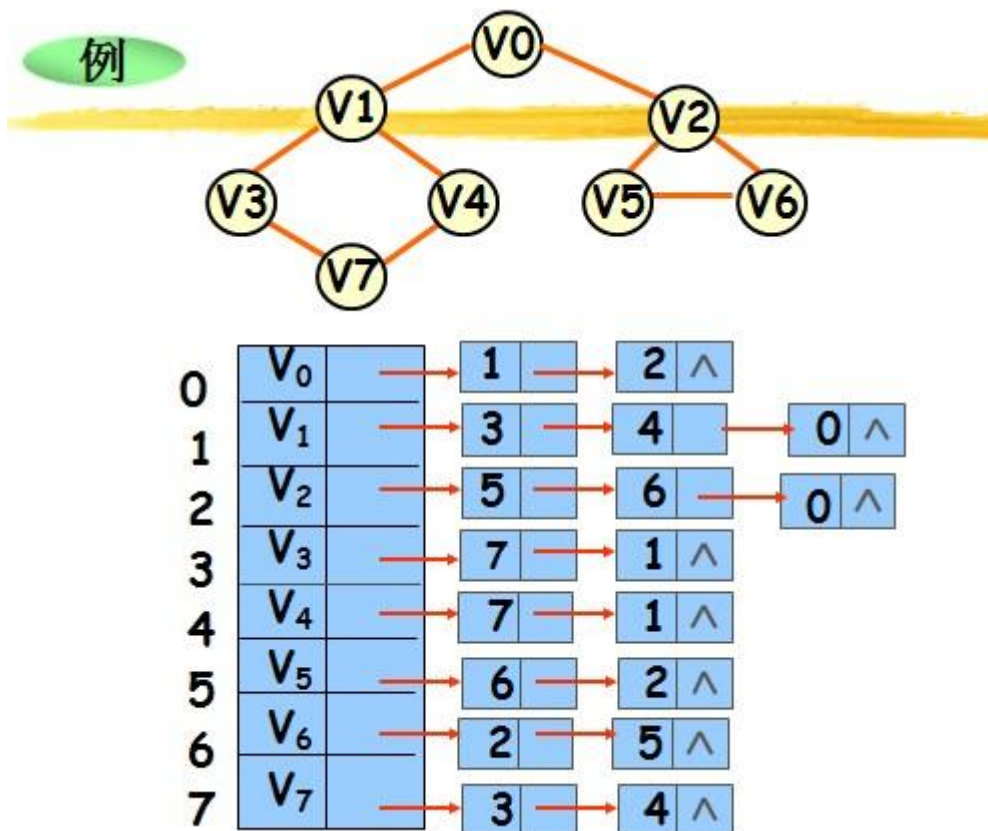
有向图的邻接矩阵表示：



<2> 邻接表 (adjacency list):

对于任何一个结点，维护一个所有与它相邻的结点的链表。



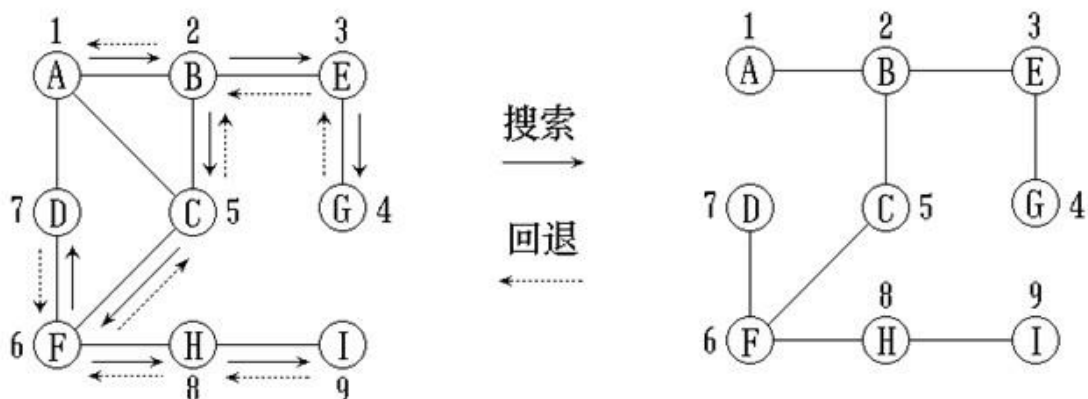


3. 图的遍历

<1> 深度优先遍历 (Depth-First Traversal , DFS):

假定现有一个无向图 G ，所有的结点初始化为未访问 (unvisited)。深度优先遍历选择一个未访问的结点 v 作为起始结点，然后标记 v 为已访问 (visited)。接着递归遍历 v 周围未访问的结点。

example

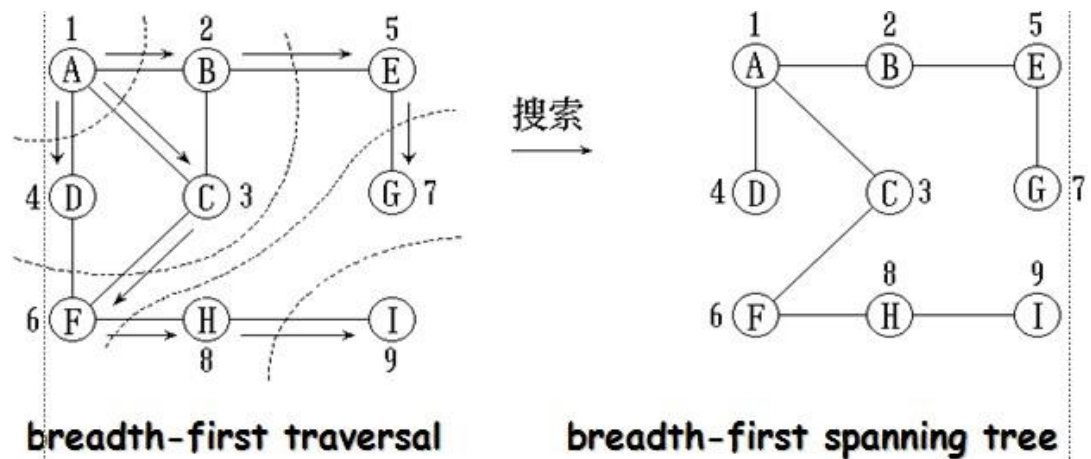


<2> 广度优先遍历 (Breadth-First Traversal , BFS) :

先访问结点 v_i 。

访问 v_i 的所有邻居结点。

从这些结点继续，直到所有结点访问完毕。



4. 一些图论算法

<1> 求无向图的最小生成树：

克鲁斯卡尔算法 (Kruskal's Algorithm) :

将图的所有边按权值从小到大排序，每次选取权值最小的边加入生成树，注意不能形成环。直到所有点均有边连接到，算法结束。

在判断是否形成环时候用到了并查集 (Union-Find set)，初始状态每个结点都处在不同集合，每次加边的时候要判断边的两个结点 u 和 v 是否处在不同集合，如果处在不同集合则将边加入生成树，并且合并 u 和 v 所在的两个集合。如果 u 和 v 本来处在统一集合，则放弃该边。

伪代码：

```
For( $i=0; i < \text{EdgeNum}; i++$ ) { //边已经按权值从小到大排好序
```

```
    Int  $u = G.\text{edge}[i].u$ ; //边的左结点
```

```
    Int  $v = G.\text{edge}[i].v$ ; //边的右结点
```

```

    Int x = F.Find(u); //查找 u 所在的集合

    Int y = F.Find(v); //查找 v 所在的集合

    If(x!=y) { //若不在同一集合

        F.Union(u,v); //合并两个集合

        MST.Insert(G.edge[i]); //将该边加入生成树

        Count++; //生成树边数目加一

        If(Count>=n-1) break; //边数已够

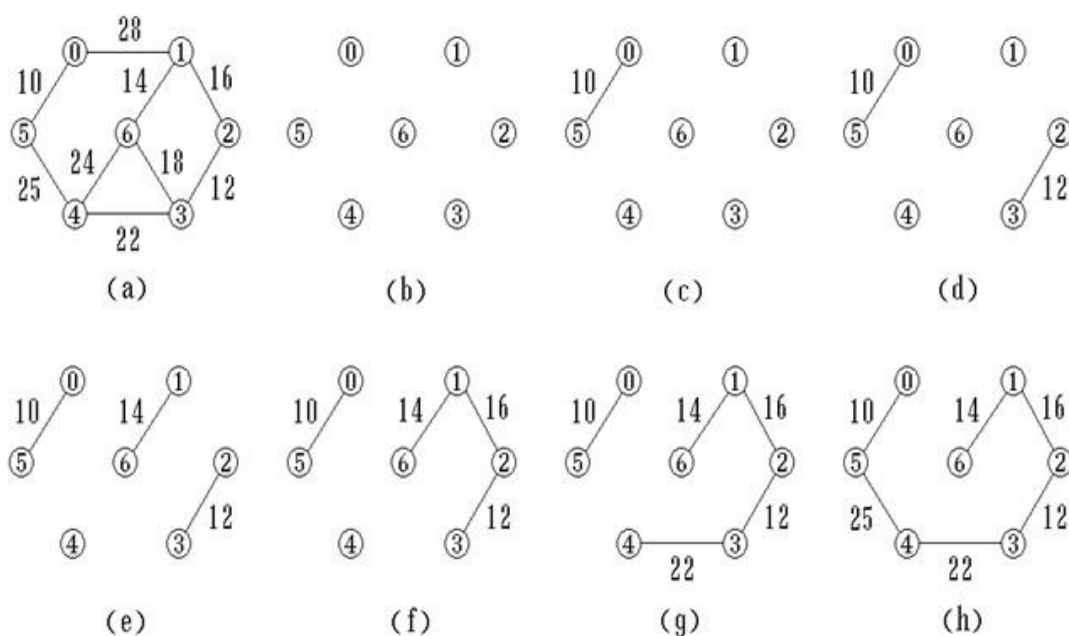
    }

}

```

算法过程如下图所示：

Sequence of edges added by Kruskal's algorithm



普里姆算法 (Prim's Algorithm)：

选取一个初始结点放入集合 U 中，算法让集合 U 慢慢“成长”成为一棵生成树，每次加一条边。

在每一步中，找一条最短的边，满足其一端在 U 中，另一端在结点集合 $V-U$ 中，把这条边加入 U 。重复上述步骤直到 $U=V$ 。

伪代码：

$U.Insert(v_0);$ //初始化集合 U

While ($U \neq V$) { // $U=V$ 时算法完成

$Edge\ e = getShortestEdge();$ //选取满足要求的最短边

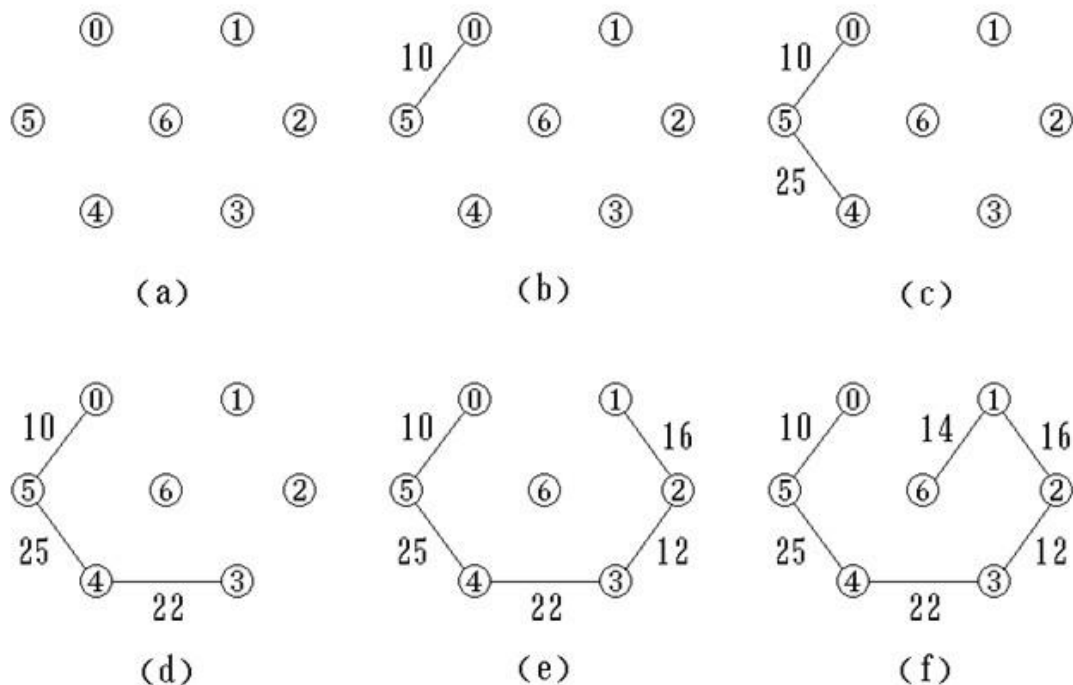
$U.Insert(e.v);$ //加入另一端的结点

$U.AddEdge(e);$ //加入该边

}

算法过程如下图所示：

Sequences of edges added by Prim's algorithm.



<2> 求最短路 (Shortest Path)

求正权图单源最短路的迪杰斯特拉算法 (Dijkstra algorithm)：

维护一个顶点集合 S ，集合中所有点到源点（Source vertex）的最短路已知。初始化的时候， S 中只包含源点。在每一步中，我们向 S 中添加一个在剩余图中的顶点 v ，它到源点的距离尽可能小。

伪代码：

$S.Insert(源点); //初始化$

$For(i=0; i < VertexNum; i++) \{$

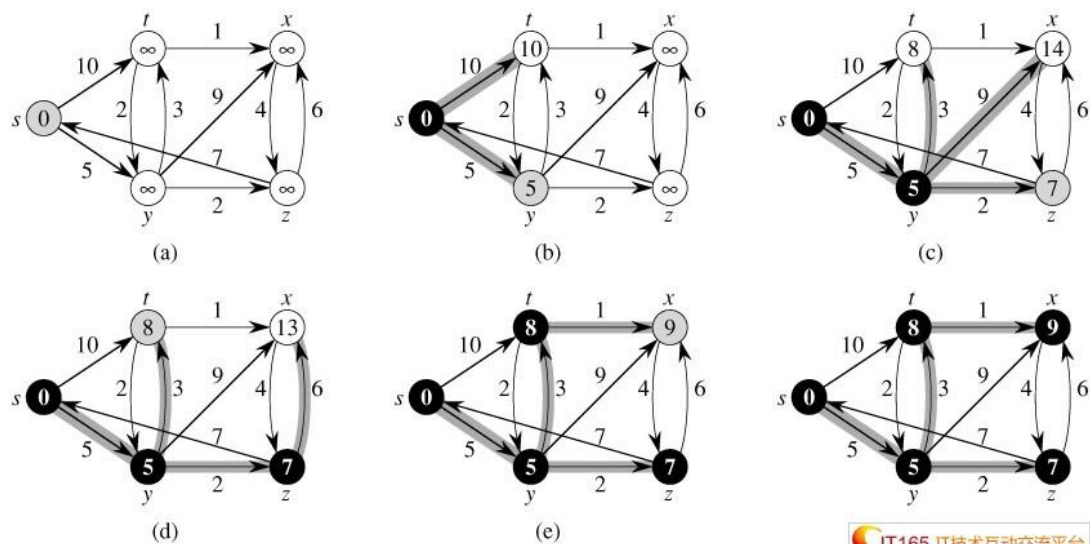
$\quad Int\ x = \text{到源点距离最小并且未访问过的点的标号};$

$\quad Visited[x]=true;$

$\quad \text{更新 } S \text{ 中所有点到源点的距离} = \min(\text{原值}, \text{该点到 } x \text{ 的距离} + x \text{ 到源点的距离});$

$\}$

算法过程如下图所示：



<3> 拓扑排序 (Topological sort) :

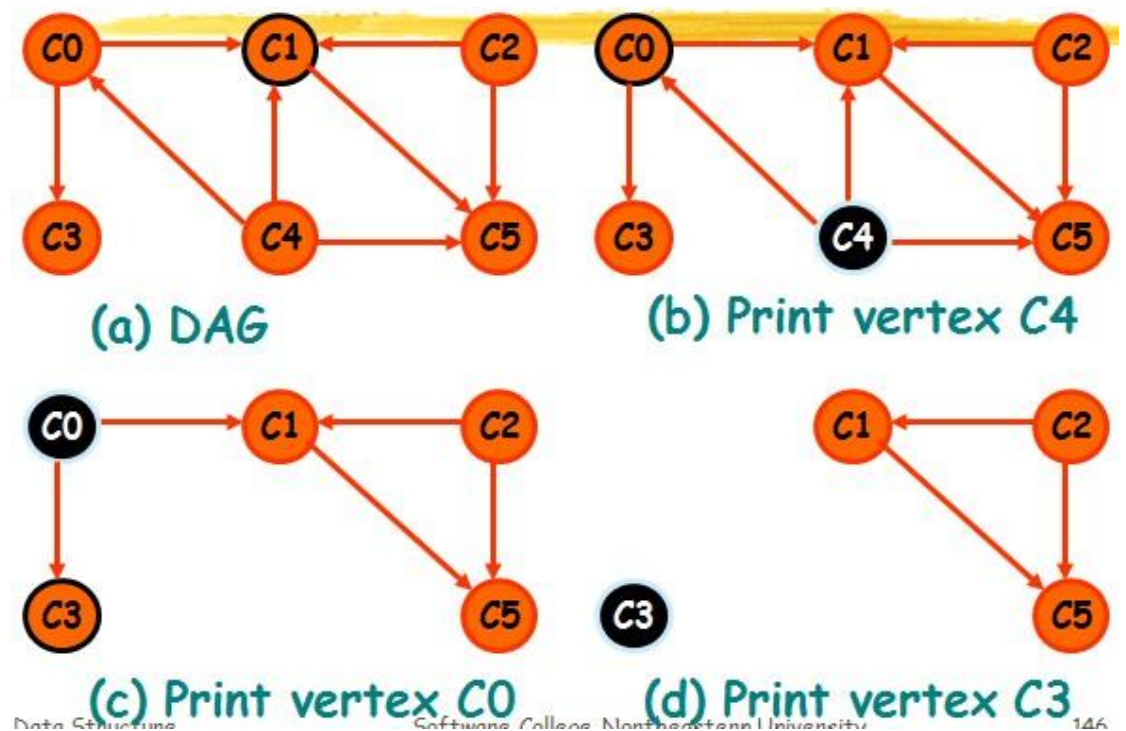
拓扑排序是一个过程，它指定一个有向无环图 (DAG) 的顶点构成一个线性序列。满足：若顶点 i 和顶点 j 之间有边，那么 i 在线性序列中会出

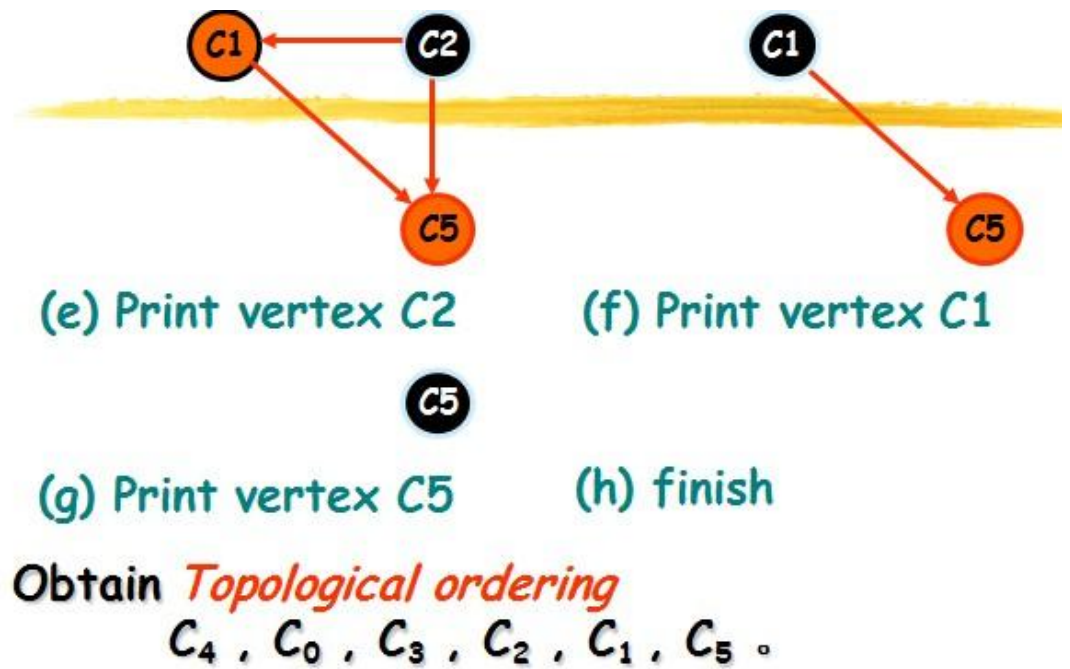
现在 j 的前面。

重复下列步骤直到图为空：

- 选择一个入度为零的顶点。
- 将此结点加入序列。
- 若入度为零的顶点均已加入,则在原图中将这些结点及其射出的所有边移除。

算法过程如下图所示：





5. 习题

- <1> (B) Suppose there are 3 edges in an undirected graph G, If we represent graph G with a adjacency matrix, How many “1”s are there in the matrix?

a.3 b. 6 c. 1 d. 9

假定在一个无向图 G 中有 3 条边 ,如果我们用邻接矩阵来表示图 G ,
 那么在矩阵中有多少个“1” ?

解题思路：若有无向边 $edge[i][j]$ ($i \neq j$), 则 $matrix[i][j]$ 和 $matrix[j][i]$ 均为 1. 因为有 3 条边，故有 6 个位置为“1”。

<2> There are at least n-1 edges in a connected graph with n vertices.

在一个有 n 个结点的连通图中至少有 n-1 条边。

解题思路：要求最少的边，且保证连通性，即求其生成树的边数，故为 n-1。

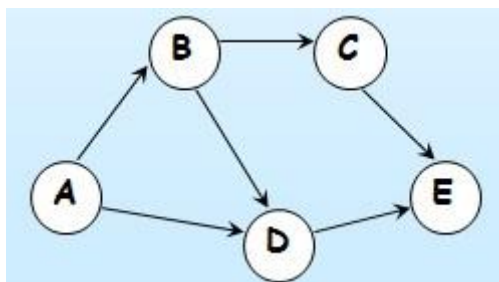
<3> Dijkstra's algorithm may be used to find the shortest path in a graph that contains edges with non-negative weights.

迪杰斯特拉算法可以用来寻找图中边权 非负 的最短路。

<4> Graph G shown in Fig 2 is a directed graph, please describe G with adjacency matrix and write the orders of breadth first traversal and depth first traversal.

如 Fig 2 所示是一个有向图 G，请写出图 G 的邻接矩阵表示并且写出图 G 的广度优先遍历次序和深度优先遍历次序。

Figure 2 . Directed Graph G



解题思路：

邻接矩阵表示：

A 0 1 0 1 0

B 0 0 1 1 0

C 0 0 0 0 1

D 0 0 0 0 1

E 0 0 0 0 0

广度优先遍历次序 (字典序最小): ABDCE

深度优先遍历次序 (字典序最小): ABCED

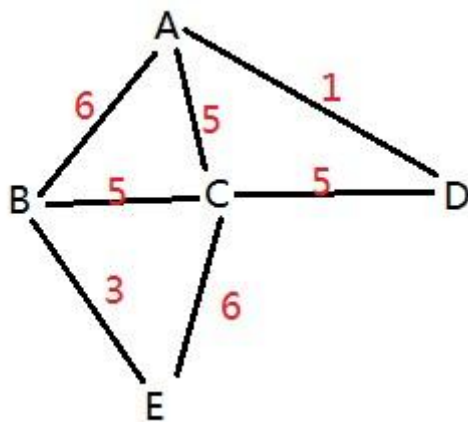
<5> The adjacency matrix of graph G is shown below. There are 5

vertices in the graph. Describe the connected components and minimum spanning tree.

图 G 的邻接矩阵表示如下所示。在图中有 5 个顶点。求其连通分量和最小生成树。

A	∞	6	5	1	∞
B	6	∞	5	∞	3
C	5	5	∞	5	6
D	1	∞	5	∞	∞
E	∞	3	6	∞	∞

解题思路：



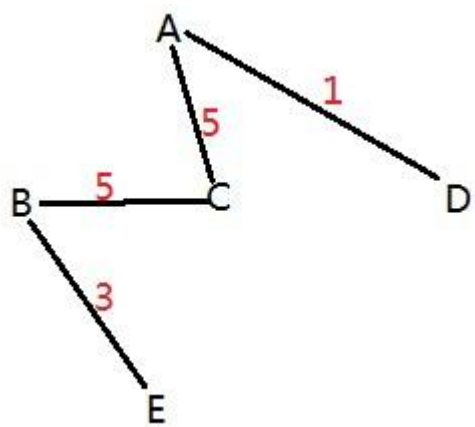
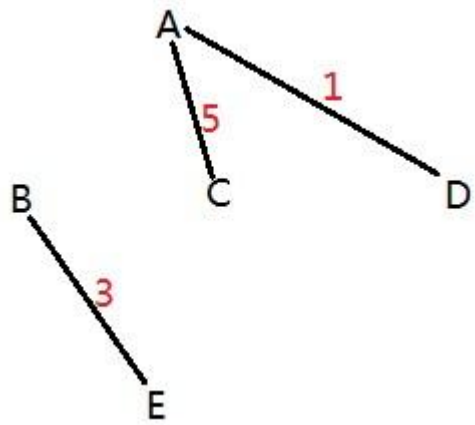
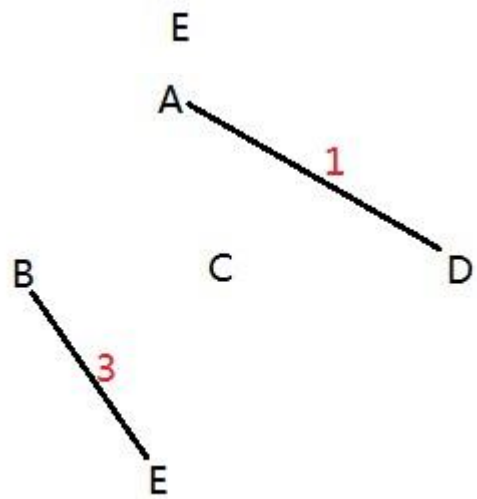
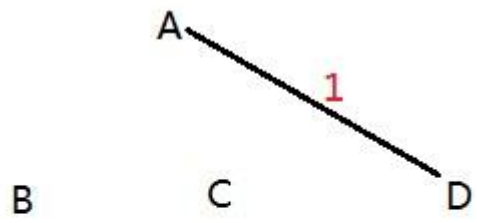
求连通分量：

通过邻接矩阵画出原图，可以看出原图是连通图，故连通分量为

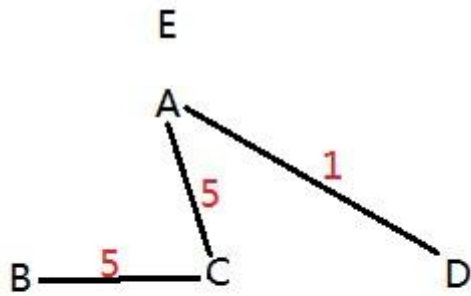
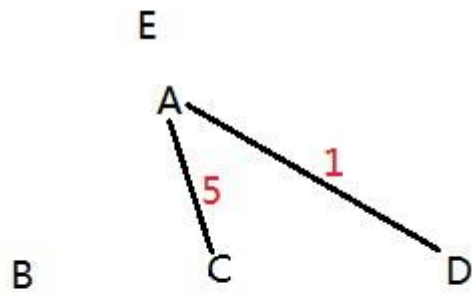
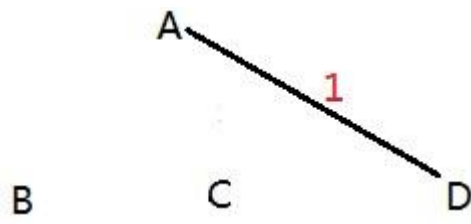
$S=\{A,B,C,D,E\}$ 。

求最小生成树：

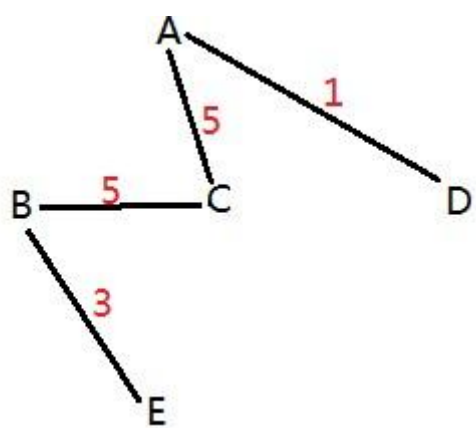
a). 克鲁斯卡尔算法：



b). 普里姆算法：



E



第 9 章 排序 (Sorting)

“研表究明，汉顺字序并不定一影阅响读”

1. 插入排序 (Insertion sort)

插入排序每次都将元素移动到**适当的位置**上。

如下图所示：

Example: $n=8$, a set of records to be sorted

49, 38, 65, 97, 76, 13, 27, 49

Process of simple insertion Sorting:

初始关键字: (49) 38 65 97 76 13 27 49

i=2 **(38)** (38 49) 65 97 76 13 27 49

i=3 **(65)** (38 49 65) 97 76 13 27 49

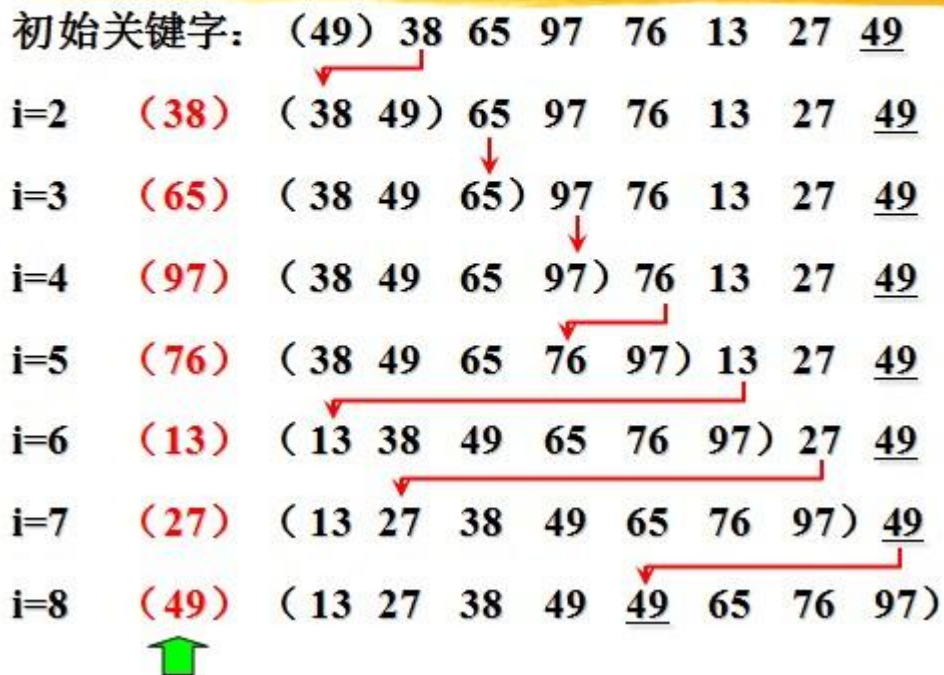
i=4 **(97)** (38 49 65 97) 76 13 27 49

i=5 **(76)** (38 49 65 76 97) 13 27 49

i=6 **(13)** (13 38 49 65 76 97) 27 49

i=7 **(27)** (13 27 38 49 65 76 97) 49

i=8 **(49)** (13 27 38 49 49 65 76 97)



2. 希尔排序 (Shell sort)

希尔排序是**改进版**的插入排序。

元素的比较间隔不再局限为 1，而是按照规律从大至小。

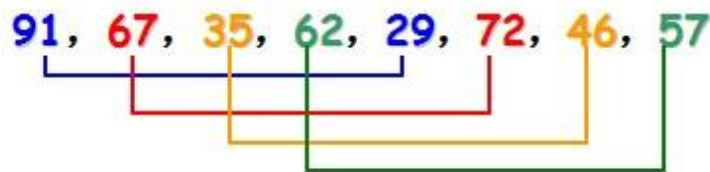
如下图所示：

Example: $n=8$, a set of records to be sorted:

91, 67, 35, 62, 29, 72, 46, 57

Process of shell sorting:

第一趟, 增量 $dk=4$



第一趟排序结果

29, 67, 35, 57, 91, 72, 46, 62

第二趟 增量 $dk=2$



第二趟排序结果

29, 57, 35, 62, 46, 67, 91, 72

第三趟 增量 $dk=1$

(29)	57	35	62	46	67	91	72
(29	57)	35	62	46	67	91	72
(29	35	57)	62	46	67	91	72
(29	35	57	62)	46	67	91	72
(29	35	46	57	62)	67	91	72
(29	35	46	57	62	67)	91	72
(29	35	46	57	62	67	91)	72
(29	35	46	57	62	67	72	91)

3. 冒泡排序 (Bubble sort)

在每一趟中, 比较两个相邻的元素, 若前一个大于后一个, 则交换, 遍历完整个序列, 最大的一个元素会被“挤”到最后面。接着进行下一趟, 会将次大元素放置到倒数第二个位置。直到第 n 趟结束, 算法完成。(n 为

序列中元素个数)

算法流程如下图所示 (只画出了前两趟比较):

Bubble Sort

1	1	23	2	56	9	8	10	100
2	1	2	23	56	9	8	10	100
3	1	2	23	9	56	8	10	100
4	1	2	23	9	8	56	10	100
5	1	2	23	9	8	10	56	100

---- finish the first traversal ----

---- start again ----

1	1	2	23	9	8	10	56	100
2	1	2	9	23	8	10	56	100
3	1	2	9	8	23	10	56	100
4	1	2	9	8	10	23	56	100

---- finish the second traversal ----

---- start again ----

Why Bubble Sort ?

29

4. 快速排序 (QuickSort)

如果待排序的元素只有 0 个或一个，返回。

选择任何一个元素 v (被称作中心点，pivot)。

把剩余的元素分成两个独立的集合， S_1 中的元素小于等于 v ， S_2 中的元

素大于 v 。

返回 $\text{QuickSort}(S_1)$ ，接着是 v ，接着是 $\text{QuickSort}(S_2)$ 。

算法流程如下图所示：

QuickSort example



5	1	4	2	10	3	9	15	12
---	---	---	---	----	---	---	----	----

Pick the middle element as the pivot, i.e., 10

Partition into the two subsets below

5	1	4	2	3	9
---	---	---	---	---	---



15	12
----	----

Sort the subsets

1	2	3	4	5	9
---	---	---	---	---	---

12	15
----	----

Recombine with the pivot



1	2	3	4	5	9	10	12	15
---	---	---	---	---	---	----	----	----

正如它的名字，快速排序在实践中被认为是**最快**的排序算法。

5. 选择排序 (Selection sort)

算法每次从当前剩余序列中找出最小的元素，并与当前剩余序列最前面的元素进行交换。这样进行 n 次**选择**之后，算法结束，得到由小到大排列的有序序列。(n 为序列中元素个数)

算法流程如下图所示：

例如，给定 $n=8$ ，数组R中的8个元素的排序码为：

8, 3, 2, 1, 7, 4, 6, 5

直接选择排序过程：



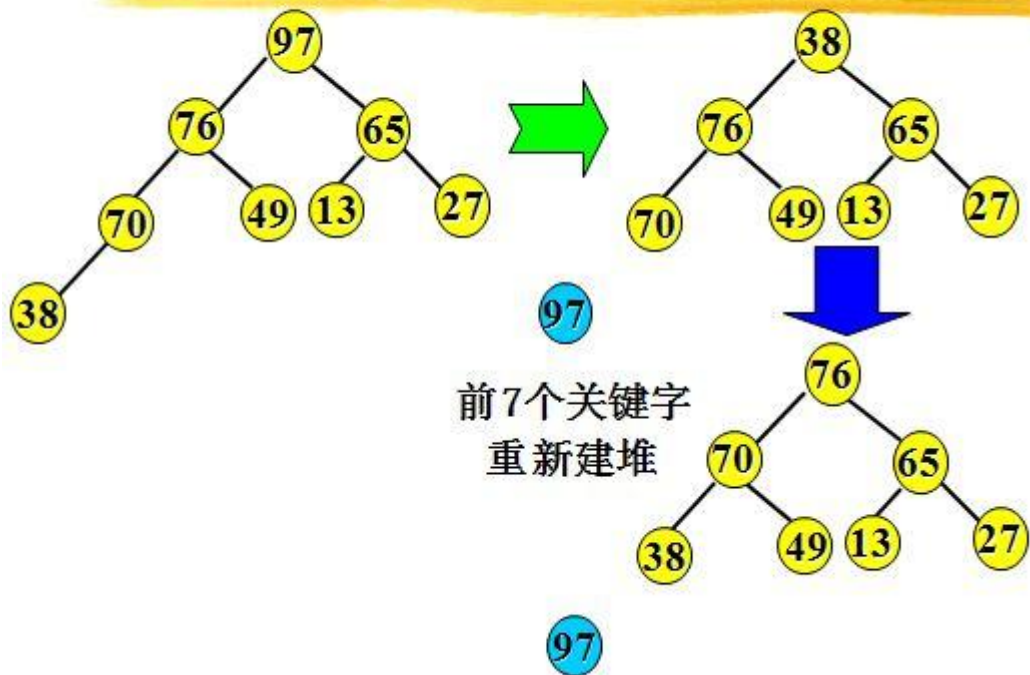
6. 堆排序 (Heapsort)

将序列建立为 **小根堆** (目的是将元素 **从大到小** 排序) 或 **大根堆** (目的是将元素 **从小到大** 排序)。

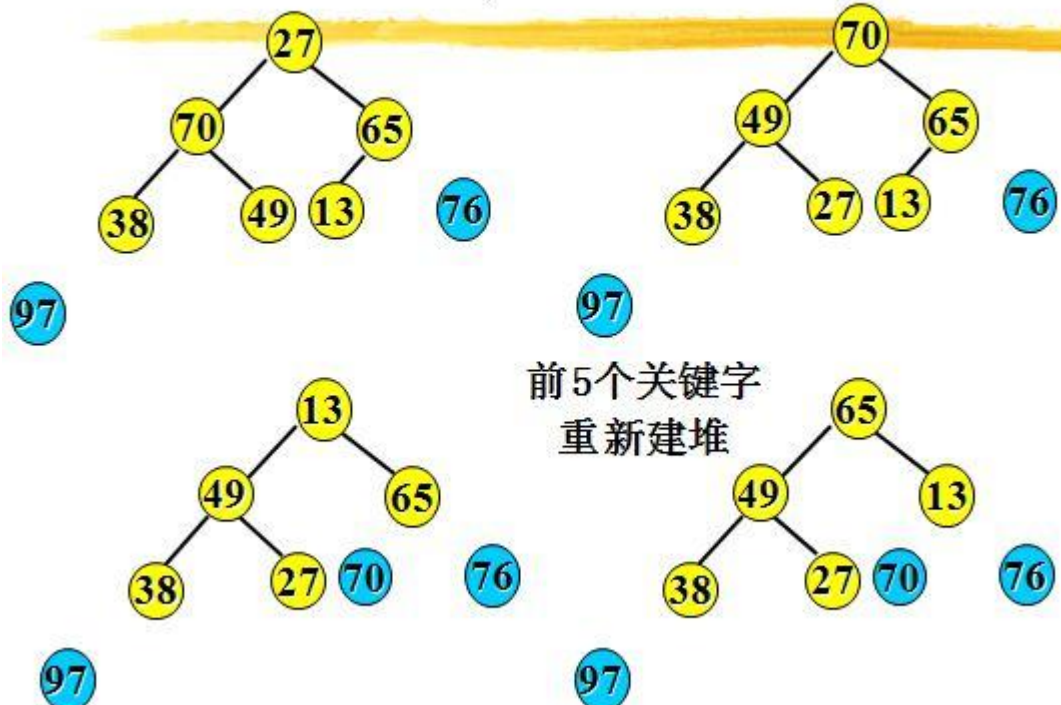
每次选择根元素，将其与最后一个元素进行交换，**堆规模减一**，其余元素重新**原地建堆**。

算法流程如下图所示 (以大根堆为例)：

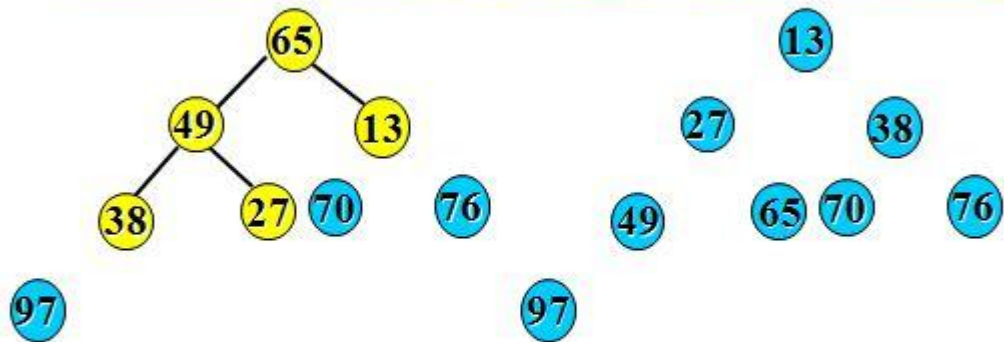
Heapsort: A complete example



Example (cont'd)



Example (cont'd)



数组中的排列方式

13	27	38	49	65	70	76	97
----	----	----	----	----	----	----	----

7. 归并排序 (MergeSort)

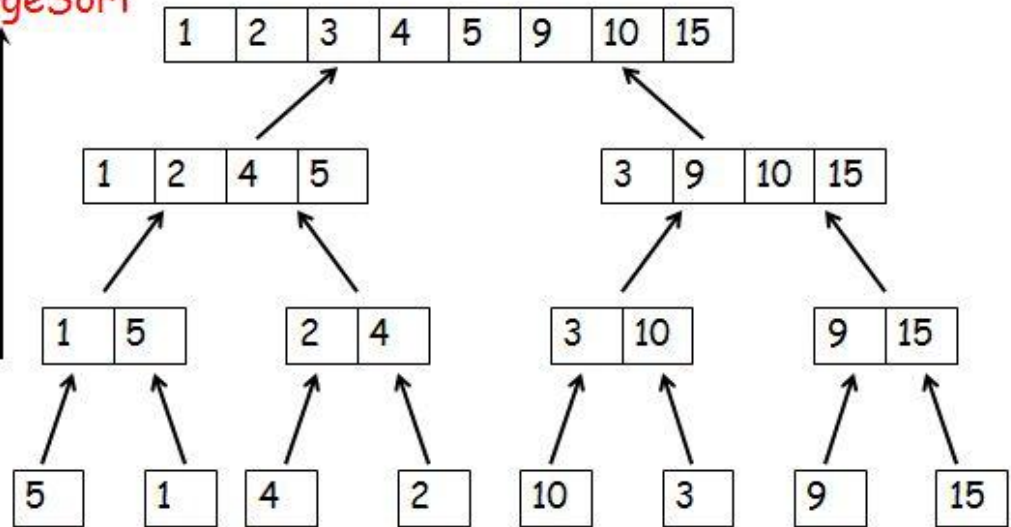
归并排序的思想是分治法。

此处要说的是基于两路归并的归并排序。

两路归并是基于两个有序序列的操作，将其合为一个有序序列，可在线性时间完成。

算法流程如下图所示：

MergeSort



8. 习题

<1> When the input has been sorted ,the running time of insertion sort(Big-Oh) is $O(n)$

当输入是已经排好序时，插入排序的运行时间是(大O表示法) $O(n)$

解题思路：既然排好序了，那么每趟插入的时候无需比较，需要的时间只是遍历序列的时间，所以是线性时间。

<2> We sort the sequence (43 , 02 , 80 , 48 , 26 , 57 , 15 , 73 , 21 , 24 , 66) with shell sort for the first increment 3, the result is 15 02 21

24 26 57 43 66 80 48 73

我们用希尔排序第一次增量为 3 来排序序列 (43 , 02 , 80 , 48 , 26 , 57 , 15 , 73 , 21 , 24 , 66) , 得到的结果将会是

15 02 21 24 26 57 43 66 80 48 73

解题思路：

根据增量为 3，将序列元素分组：

第一组：43 48 15 24 排序之后：15 24 43 48

第二组 : 02 26 73 66 排序之后 : 02 26 66 73

第三组 : 80 57 21 排序之后 : 21 57 80

合并 :

15 02 21 24 26 57 43 66 80 48 73

<3> (D) The fastest sorting algorithm list below is_____.

a. Bubble sort b. Merge sort c. Selection sort d. Quick sort

下列中最快的排序算法是_____.

a. 冒泡排序 b. 归并排序 c. 选择排序 d. 快速排序

解题思路 :A 和 C 都是 $O(n^2)$ 的 ,直接排除。至于 B 和 D ,都是 $O(n\log n)$

的算法 ,但在实践中快速排序通常具有更高的效率。

第 10 章 散列 (Hashing)

“但愿有这么一个散列函数：分数= $\max(60, \text{分数})$ ”

1. 一些术语

<1> 散列 (hash):

或直接音译为“哈希”,就是把任意长度的输入(又叫做预映射 ,pre-image), 通过散列算法, 变成固定长度的输出, 该输出就是散列值。这种转换是一种**压缩映射**, 也就是, 散列值的空间通常小于输入的空间, 不同的输入可能会散列成相同的输出, 而不可能从散列值来唯一的确定输入值。

<2> 散列函数 (hash function):

若结构中存在和关键字 K 相等的记录, 则必定在 $h(K)$ 的存储位置上。由此, 不需比较便可**直接取得**所查记录。称这个对应关系 h 为散列函数。

<3> 散列表 (hash table)

按照散列函数思想建立的表称为散列表。

<4> 冲突 (collision)

对不同的关键字可能得到**同一散列地址**, 即 $K_1 \neq K_2$, 而 $h(K_1)$ 等于 $h(K_2)$, 这种现象称作冲突。

<5> 平均搜索长度 (average search length , ASL)

平均搜索到一个元素所需的比较次数。

2. 散列函数的构造 (除留余数法 , The division method)

$h(k) = k \bmod m$ 。(m 大于等于总元素个数, 通常取素数)

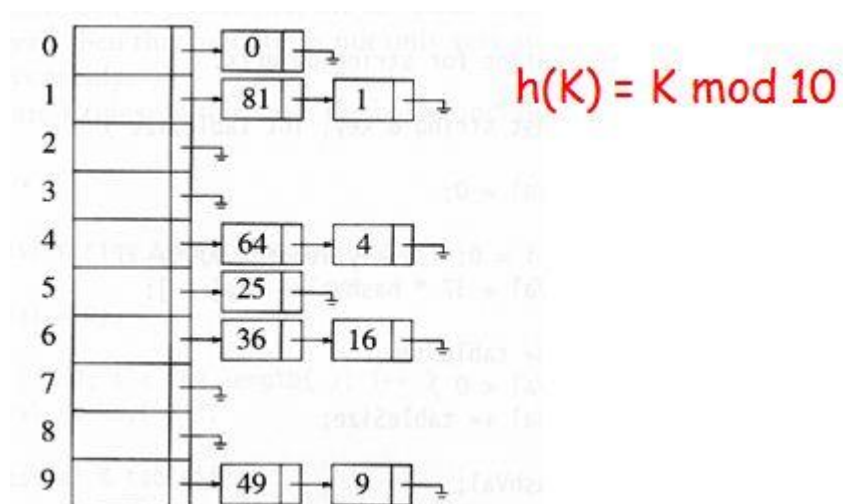
3. 冲突处理 (Collision Handling)

<1> 链地址法 (Separate Chaining)

用一个装有**链表**的表来代替散列表。

为散列值相同的关键字维护一个链表。

如下图所示：



<2> 开放地址法 (Open Addressing)

如果要插入的关键字 K 与已存在的关键字发生冲突，则为 K **重新定位**。

也就是说，我们把 K 存储在一个和 $h(K)$ 不同的地址中。

常用手段：

a). 线性探测 (Linear probing)

若检测到冲突，则**线性**向右遍历，直到找到空位置为止。

$$h_i(k) = (h(k) + i) \bmod m$$

b). 平方探测 (Quadratic probing)

若检测到冲突，则**平方**向右遍历，直到找到空位置为止。

$$h_i(k) = (h(k) + i^2) \bmod m$$

c). 再散列 (Double hashing)

若检测到冲突，则使用第二个散列函数，直到找到空位置为止。

$$h_i(k) = (h(k) + i \cdot h_2(k)) \bmod m$$

4. 习题

<1> The sequence of input keys is shown below:

19, 1, 23, 14, 55, 20, 84, 27, 68, 11, 10, 17

A fixed table size of 19 and a hash function $H(\text{key}) = \text{key} \% 13$, with linear probing(线性探测), fill the table below and compute the average length of successful search.

输入的关键字序列如下所示：

19, 1, 23, 14, 55, 20, 84, 27, 68, 11, 10, 17

表长固定为 19，散列函数为 $H(\text{key}) = \text{key} \% 13$ ，使用线性探测，填写散列表并计算平均成功搜索长度。

解题思路：

每一行代表每一步插入操作，红色代表有冲突，用线性探测找到的新地址。

0	1	2	3	4	5	6	7	8	9	10	11	12
						19						
	1					19						
	1					19				23		
	1	14				19				23		
	1	14	55			19				23		
	1	14	55			19	20			23		

	1	14	55			19	20	84		23		
	1	14	55	27		19	20	84		23		
	1	14	55	27	68	19	20	84		23		
	1	14	55	27	68	19	20	84		23	11	
	1	14	55	27	68	19	20	84		23	11	10
	1	14	55	27	68	19	20	84	17	23	11	10

故散列表为：

0	1	2	3	4	5	6	7	8	9	10	11	12
	1	14	55	27	68	19	20	84	17	23	11	10

每个关键字的成功查找长度：

1	14	55	27	68	19	20	84	17	23	11	10
1	2	1	4	3	1	1	3	6	1	1	3

平均查找长度 $ASL = (1+2+1+4+3+1+1+3+6+1+1+3)/12 = 2.25$