数据结构

作者

王　娜　　眭思悦

梁嘉朗　　梁鸿坭

胡　啸　　傅显坤

苏庆栋　　董作岭

刘　元　　任腾龙

赵乾皓　　高　萁

审核校对：王渊钧　排版：沈珈莹

版本号：V2.0

此版本包括数据结构第 2 版 2、3、5-14、16-18 章节的内容

同时向 1.0 版本的作者聂义莹、鹿敏宽、孙吉鹏、杜泽林、张晓敏、鲍伟、徐卫霞致敬

修订时间：2020.1

山软智库-让知识回归平凡

第2章 程序性能分析

2.1 什么是程序性能

1.程序性能：运行这个程序所需要的内存和时间的多少。（空间复杂度和时间复杂度）
（1）**空间复杂度**：运行程序所需内存的大小。
（2）**时间复杂度**：运行程序所需要的时间。

2.2 空间复杂度

2.2.1 空间复杂度的组成

空间复杂度由指令空间+数据空间+环境栈空间组成。
1.指令空间：编译之后的程序指令所需要的存储空间。
指令空间的数量取决于如下因素：
（1）把程序转换成机器代码的编译器。（编译器不同所需空间可能也不同）
（2）在编译时的编译器选项。（覆盖选项可减少程序空间）
（3）目标计算机。
2.数据空间：所有常量和变量值所需要的存储空间。
数据空间由两部分构成：
（1）常量和简单变量所需的存储空间。
（2）动态数组和动态类实例等动态对象所需存储空间。
不同类型的数据占用的内存也不相同。
3.**环境栈空间**：环境栈用来保存暂停的函数和方法在恢复运行时所需要的信息。（尤其是递归函数）
每当一个函数被调用时，下面的数据将被保存在环境栈中：
（1）返回地址
（2）正在调用的函数所有局部变量的值以及形式参数的值（仅对递归函数而言）
4.程序所需空间可分成两部分：c（固定部分）+S_p（可变部分）
（1）固定部分：独立于实例特征。通常包括指令空间、简单变量空间和常量空间等。
（2）可变部分：由动态分配空间和递归栈空间构成。前者某种程度上依赖实例特征，后者主要依赖实例特征。
递归栈空间：递归函数所需要的栈空间。主要依赖于局部变量和形式参数所需空间以及递归的最大深度（递归调用的最大层次）。
在分析程序的空间复杂度时，我们主要计算S_p。

2.2.2 举例

在计算S_p（实例特征）的时候，我们主要观察程序中有无递归函数（没有递归函数一般为0）。当存在递归函数时，首先确定递归函数中的形式参数和局部变量以及返回地址的空间（一般假定返回地址空间为4字节）。再计算递归深度，相乘即可得到递归栈空间的

大小。

注意：

（1）根据程序实际，递归深度可能有不同情况，不能忽略（如例2-5）。

（2）要检查递归函数中是否有定义局部变量（如例2-6）。

（3）每次都要加上返回地址的空间。

2.3 时间复杂度

2.3.1 时间复杂度的组成

一个程序所需要的时间是编译时间+运行时间。我们主要关注程序的运行时间，用t_p（实例特征）表示。

1.估算程序运行时间的方法：

（1）找出一个或多个关键操作(对时间复杂度影响最大)，确定它们的执行时间。

（2）确定程序总的步数。

2.3.2 操作计数

选取几个书上的例子讨论选择关键操作估算时间复杂度的方法。

1.例2-7（最大元素）：选择的操作为数组元素之间的比较操作，比较次数为max{n-1,0}。

2.例2-8（多项式求值）：选择的操作为加法和乘法的次数。利用Horner法则可以将时间复杂度简化为n次加法和n次乘法。

3.例2-11（选择排序）：选择的操作为数组元素的比较操作和移动操作。比较次数为(n-1)n/2，移动次数为3(n-1)（每一轮找出的最大元素与未排序序列的最后一个元素交换，通过辅助变量{t=a;a=b;b=t;}进行三次移动操作，一共要进行n-1轮交换）。

在基于比较的排序算法中，选择的操作*通常*为比较和移动。

2.3.3 最好、最坏和平均操作计数

操作计数不总是由实例特征唯一确定，与实际的数据情况也有关，所以要分析最好最坏两种操作计数。

1.最好操作计数：所有程序实例操作计数中，最小操作计数。

2.最坏操作计数：所有程序实例操作计数中，最大操作计数。

下面结合例子分析计算最好、最坏和平均操作计数的过程。

例2-13（顺序搜索）：选择的操作x与数组元素的比较次数。顺序搜索有成功、不成功两种结果。查找不成功时比较次数为n。若查找成功，最少比较次数为1，最多比较次数为n。

为了估算平均操作计数，假定所有数组元素都不同且每个元素被查找概率相同。则在查找成功时平均操作计数为：(1+2+…+n)/n=(n+1)/2。

2.3.4 步数

操作计数方法只针对选定的操作而忽视了其他操作，另一种估算时间复杂度的方法——步数方法则对所有操作部分进行统计。
步数也有最好、最坏、平均步数。
1.**程序步**：一个程序步可以定义为一个语法或语义上的程序片段，该片段的执行时间独立于实例特征（与实例特征无关，如100次加法、return a+b+b*c可以视为一步，但n次加法不能视为一个程序步，其中n为实例特征）。
2.确定步数的方法：
（1）在程序中创建一个全局变量stepCount（其初值为0）。可以把stepCount的增值语句嵌入原程序，每当原程序或函数中的一条语句被执行时，就为stepCount累加上该语句所需要的步数。
注意：for语句的每一次条件判断、if语句和return语句都算一个程序步，不能忽略。详见程序2-16、2-17、2-18。
（2）剖析法：建立步数表，确定每条语句每次执行所需要的步数（s/e,steps per execution）和该语句总的执行次数（频率），相乘得到每条语句的总步数。最后所有语句相加就是整个程序步数。详见之后的例题。
s/e：语句每次执行所需要的步数，即执行该语句产生的stepCount值的变化量。与步数有很大差别。
3.分析递归函数步数的方法：得到递推方程，并反复替换求解。具体过程见程序2-18的分析。

第3章 渐进记法

3.2 渐进记法

3.2.1 大*O*记法

1.常用项排序：1<logn<n<nlogn<n^2<n^3<2^n<n!。
2.f(n)=*O*(g(n))(读作"f(n)is big oh of g(n)"):f(n)渐进小于或等于g(n)，g(n)是f(n)的最小上限。当且仅当存在常数c>0和n_0，使得对于所有的n≥n_0，有f(n)≤cg(n)。
注意：
（1）除了f(n)=0以外，函数g(n)习惯上是令f(n)=*O*(g(n))为真的最小单位项（系数为1）。因此常见的是3n+3=*O*(n)，而不是3n+3=*O*(n^2)，尽管后者也是正确的。（2）在渐进复杂度分析中，要确定一个最大单位项以表示复杂度。如执行步数为$3n^2$+6nlogn+7n+5的程序，渐进复杂度为*O*(n^2)。
（3）f(n)=*O*g(n)并不等价于*O*(g(n))=f(n)。实际上，*O*(g(n))=f(n)是无意义的。在这里，使用符号=是不确切的，因为=通常表示相等关系。可以通过把符号=读作“是”而不是“等于”来避免这种矛盾。

3.2.2 渐进记法*Ω*和*Θ*

1.f(n)=Ω(g(n))(读作"f(n)is Ω of g(n)"):f(n)渐进大于或等于g(n)，g(n)是f(n)的最大下限。当且仅当存在常数c>0和n_0，使得对于所有的n≥n_0，有f(n)≥cg(n)。

2.f(n)=Θ(g(n))(读作"f(n)is Θ of g(n)"):f(n)渐进等于g(n)。当且仅当存在常数c_1>0,c_2>0和n_0，使得对于所有的n≥n_0，有c_1g(n)≤f(n)≤c_2g(n)。

3.3 渐进数学（可选）

3.3.1 大O记法

1.有用的结论：

（1）如果f(n)=$a_m n^m$+…+a_1n+a_0且a_m>0,那么f(n)=O(n^m)。

（2）加法规则：T(n)=T_1(n)+T_2(n)=O(g_1(n)+g_2(n))=O(max(g_1(n),g_2(n)))。

（3）乘法规则：T(n)=T_1(n)T_2(n)=O(g_1(n))*O(g_2(n))=O(max(g_1(n)g_2(n)))。

2.大O比率定理：假如函数f(n)和g(n)有极限$\lim_{n\to\infty}$f(n)/g(n)存在，那么关系式f(n)=O(g(n))成立，当且仅当存在常数c，使$\lim_{n\to\infty}$f(n)/g(n)≤c。

3.3.2 Ω记法

1.有用的结论：如果f(n)=$a_m n^m$+…+a_1n+a_0且a_m>0,那么f(n)=Ω(n^m)。

2.Ω比率定理：假如函数f(n)和g(n)有极限$\lim_{n\to\infty}$g(n)/f(n)存在，那么关系式f(n)=Ω(g(n))成立，当且仅当存在常数c，使$\lim_{n\to\infty}$g(n)/f(n)≤c。

3.3.3 Θ记法

1.有用的结论：如果f(n)=$a_m n^m$+…+a_1n+a_0且a_m>0,那么f(n)=Θ(n^m)。

2.Θ比率定理：假如函数f(n)和g(n)有极限$\lim_{n\to\infty}$f(n)/g(n)和$\lim_{n\to\infty}$g(n)/f(n)存在，那么关系式f(n)=Θ(g(n))成立，当且仅当存在常数c，使$\lim_{n\to\infty}$f(n)/g(n)≤c及$\lim_{n\to\infty}$g(n)/f(n)≤c。

3.3.4 小o记法

f(n)=o(g(n))：当且仅当f(n)=O(g(n))且f(n)≠Ω(g(n))。

3.3.5 特性

下面的定理可用于渐进记法的计算：

（1）存在某个n_0，使得对于任何n≥n_0，有(logn)x<$(\log n)^{x+\varepsilon}$。

（2）存在某个n_0，使得对于任何n≥n_0，有(logn)x<n^{ε}。

（3）存在某个n_0，使得对于任何n≥n_0，有n^x<$n^{x+\varepsilon}$。

（4）对于任意实数y，存在某个n_0，使得对于任何n≥n_0，有$n^x(\log n)^y$<$n^{x+\varepsilon}$。

（5）存在某个n_0，使得对于任何n≥n_0，有n^x<2^n。

3.4 复杂度分析举例

渐进复杂度的分析：先确定每一条语句或一组语句的渐进复杂度，再把他们加起来（利用加法规则）。
请仔细阅读本节例题。

写在第2、3章后面——经典算法

第2、3章在叙述如何分析程序性能的过程中，也介绍了很多经典算法。掌握这些算法是很有必要的。
1.搜索算法——顺序搜索、折半搜索
（1）顺序搜索：从左至右遍历数组中所有元素并逐一比较，时间复杂度为$O(n)$。（P43程序2-1）
（2）**折半搜索**：在有序数组中查找元素。
主要思路：
left和right表示搜索端的左右两个端点。
搜索过程从x与数组[left:right]中间元素的比较开始，利用while循环：
如果x等于中间元素，则查找结束；
如果x小于中间元素，因为数组是有序的，那么仅需要查找数组的左半部分，所以right被修改为middle-1；
如果x大于中间元素，因为数组是有序的，那么仅需要查找数组的右半部分，所以left被修改为middle-1。
因为每次迭代以减半比例缩小搜索范围，所以时间复杂度为$O(\log n)$。（P77程序3-1）
（3）顺序搜索与折半搜索比较：从时间复杂度可以看出，顺序搜索相较于折半搜索是一种低效的搜索算法。但是，折半搜索必须加上数组是有序数组的前提，有一定的局限性。
2.排序算法——名次排序、选择排序、冒泡排序、插入排序
（1）名次排序：根据数组元素在序列中的名次进行排序的算法，有利用附加数组和原地重排两种方法。
a.利用附加数组：首先调用new新建数组，再根据名次将原数组中的元素移到新建数组的相应位置，最后把在新建数组中排序好的元素移回原数组。时间复杂度为$O(n^2)$。（P46程序2-6）
b.原地重排：从数组的第一个元素开始检查，如果名次等于索引，那么检查下一个元素；如果名次不等于索引，那么将该元素a[i]与其正确位置上的元素交换，同时，交换两个元素对应的名次。在此索引i处重复操作直到名次等于索引。时间复杂度为$O(n^2)$。（P50程序2-11）
c.利用附加数组与原地重排比较：原地重排比利用附加数组需要更多的移动，因此最坏执行时间也增加了；但利用附加数组比原地重排占用更多内存。
（2）选择排序：每次寻找出[0,n]中最大的元素并把这个最大的元素移到[n-1]，每次循环n–。时间复杂度为$O(n^2)$。（P47程序2-7）
（3）冒泡排序：每次冒泡依次将相邻元素中的较大元与较小元位置交换，使得较小元排在较大元前面（如果有序则不交换），得到[0,n]的最大元，每次循环n–。时间复杂度为

$O(n^2)$。（P48程序2-8、2-9）

（4）插入排序：因为只有一个元素的数组是有序数组，所以对n个元素的数组可以从第一个元素构成的单元数组开始。以有序数组中的插入算法为基础，不断地插入第i个元素得到长度为i的有序数组。如此下去最终得到n个元素的有序数组。时间复杂度为$O(n^2)$。（P52、53程序2-14、2-15）

第5章 线性表——数组描述

本章概述

数组描述方法将元素存储在一个数组里，用一个数学公式来确定每个元素存储的位置，即在数组中的索引。这是最简单的一种存储方式，所有元素依次存储在一片连续的存储空间里，而这就是常说的顺序表。

5.1 数据对象和数据结构

数据对象：一组实例或者值。

数据结构：是一个数据对象，同时这个对象的实例以及构成实例的元素都存在着联系，这些联系都是由相关函数来规定的。

我们研究数据结构实际上是研究数据对象的描述以及相关函数的具体实现。

5.2线性表数据结构

线性表（linear list），它的每一个实例都是元素的有序集合，其实例形式为：(e0,e1,…,en-1)，其中n 是有穷自然数。

ei 是表中的元素，i是ei的索引

n是表的长度或大小，当n = 0 时，表为空

n>0时，e0是第0个元素或首元素，en-1是最后一个元素

e0先于e1, e1先于e2,……如此等等.

5.2.1 抽象数据类型LinearList

一个线性表可以用一个抽象数据类型（ADT）来说明（包括实例和操作）。
在学完堆栈和队列后才发现ADT真的太重要了！好好看看这些概念吧!

5.2.3抽象类linearList

c++支持两种类：抽象类和具体类。抽象类里有纯虚函数，用0作为初始值

形式：
virtual function(class)=0;

5.3数组描述：

5.3.1描述：

所谓的公式化描述就是利用数组随机访问的特性组织数据，将数据连续的存放在数组中，利用数组的下标快速的访问。

location（i） = i-1；

优点：因为使用了数组，随意访问具有随意性，可以在常量时间内访问第 k 个元素。
缺点：数组的特点在于长度不可修改，所以对于扩大存储容量来说，公式化描述的线性表是很困难的；其次由于在删除和插入时保证数据的连续性，不得不移动元素，这是需要消耗时间的。

5.3.2 变长一位数组

思路：建立一个具有新长度的数组，把原来数组a的元素复制到新的数组中，最后改变数组a的值，使它能够引用新数组。

一般方法：数组倍增。

```
template<class T>
void changeLength1D(T* &a,int oldlen,int newlen)
{
   if(newlen<0) throw illegalParameterValue("error newlen");
   T *temp=new T[newlen];
   int number=min(oldlen,newlen);
   copy(a,a+number,temp);
   delete []a;
   a=temp;
}
```

> 时间复杂度：O(n)

5.3.3类arrayList

arrayList是抽象类linearList的派生类。

> 构造函数

```
template<class T>
arrayList<T>::arrayList(int initialCapacity)
{ //构造函数
  if(initialCapacity<1)
   {ostringstream s;
    s<<“initialCapacity =”<< initialCapacity <<“Must be >0”;
```

```
    throw illegalParameterValue(s.str());
    }
  arrayLength = initialCapacity;
  element = new T [arrayLength];
  listSize = 0;
}
```

> 时间复杂性 : O(1); 当T是用户自定义类型时，是 O(initialCapacity);

> 复制构造函数

```
template<class T>
arrayList<T>::arrayList(const arrayList<T>& theList)
{   //复制构造函数
    arrayLength = theList.arrayLength;
    listSize = theList. listSize;
  element = new T [arrayLength];
  copy(theList.element, theList.element+ listSize, element);
    }
```

> 时间复杂性 : O(n) n:复制的线性表的大小

> 查找:

```
template<class T>
int arrayList<T>::indexOf(const T& theElement) const
{
   //返回元素theElement第一次出现时的索引
   // 如果theElement不存在，则返回-1
   //查找元素theElement
int  theIndex=(int) (find(element, element+listSize, theElement)-element);
   //确定元素theElement是否找到
    if (theIndex ==listSize)  return -1;    //没有找到
      else   return theIndex;
    }
```

> 时间复杂性 : O(listSize)

> 删除：

```
template<class T>
void arrayList<T>::erase(int theIndex)
{  //删除表中索引为theIndex的元素，索引大于theIndex的元素其索引值减1
   //如果索引为theIndex元素不存在，则抛出异常.
  checkIndex(theIndex);
   //索引大于theIndex的元素向前(左)移动一个位置
   copy(element+theIndex+1, element+listSize, element+theIndex);
    element[--listSize].~T();
}
```

> 时间复杂性 : O(listSize-theIndex)

> 插入：

```
template<class T>
void arrayList<T>::insert(int theIndex, const T& theElement)
{//在索引theIndex处插入元素theElement;
 //如果theIndex无效，则引发异常
    if (theIndex<0 || theIndex>listSize) {……}
 //如果数组已满，则数组长度倍增.
 if (listSize==arrayLength)
   {changeLength1D(element, arrayLength, 2*arrayLength);
    arrayLength*=2;
    }
    //索引大于等于theIndex的元素向后(右)移动一个位置
  copy_backward(element+theIndex, element+listSize; element+listSize+1);
    element[theIndex]=theElement;
    listSize++;
}
```

> 时间复杂性 : O(listSize)

5.3.4C++迭代器

迭代器，就是指针，指向一个对象的元素。

顾名思义，迭代器可以逐个访问对象的所有元素。

> 使用数组迭代器：

```
int main()
{
int x[3]={0,1,2};
for(int * y=x;y!=x+3;y++)
   cout<<*y<<" ";
cout<<endl;
return 0;
}
```

虽然我们对迭代器了解得不深，但是迭代器非常容易推广，以至于可以输出任何具有迭代器的对象的元素。

5.4 vector的描述

vector是STL提供的基于数组的类，它不仅具有类arrayList 的所有功能，还有很多其他方法。
如：数组长度是按照需要动态增加的。

(代码见课本107.108页，了解即可)

第6章 线性表——链表描述

本章概述

这一章我们打破固有的数组存储思维，用链表描述数据。在链式描述中，线性表中的元素在内存中的存储位置是随机的。每个元素都有一个明确的指针指向线性表的下一个元素的位置。

6.1 单向链表

6.1.1描述

链表描述就是使用一个标记记录下每一个节点的下一个节点的地址，每次通过一个节点的 link 找到下一个节点。

对于链表要熟悉掌握查找，插入删除的操作代码。一般代码中都会涉及几种情况：

• 判断 k 是否合法（k<0 或 1，或者是 k 是否超出上界）；

• 判断链表是否为空；

• 找到第 k 个元素进行操作；

• 操作时注意几种情况：

①是否在头结点操作；

②是否在尾结点操作；

③操作结束后链表的情况（是否为空等）；

6.1.2结构chainNode

chainNode是为节点定义的数据类型，数据成员就是数据域element和链域next

```
template<class T>
struct chainNode
{
   T element;
   chainNode<t> * next;
   chainNode(){}
   chainNode(const T& element){
      this->element=element;
   }
   chianNode(const T& element,chianNode<T>* next){
      this->element=element;
```

```
        this->next=next;
    }
}
```

6.1.3 类chain

> 构造函数

```
chain(int initialCapacity = 10);
```

> 复制构造函数

```
template<class T>
chain<T>::chain (const chain<T>& theList)
{   //复制构造函数
    listSize = theList.listSize;
   //链表theList为空
   if(listSize ==0)
    {firstNode=NULL;
     return;
    }
   //链表theList不为空
chainNode<T>*  sourceNode=theList.firstNode;
   //要复制的theList中的节点
firstNode=new chainNode<T>(sourceNode->element);
   //复制theList中的首元素
sourceNode= sourceNode->next;
chainNode<T>*  targetNode=firstNode;
   //当前链表*this的最后一个节点
while(sourceNode!=NULL)
   {//复制剩余元素
   targetNode->next=new chainNode<T>
   (sourceNode->element);
   targetNode=targetNode->next;
   sourceNode= sourceNode->next;
   }
   targetNode->next=NULL;
}
```

> 时间复杂性 : Q(theList.listSize)

> 析构函数

```
template<class T>
chain<T>::~Chain()
{    //链表的析构函数，删除链表中的所有节点
  while (firstNode)
  {  //删除首节点
   chainNode<T>* next = firstNode->next;
    delete firstNode;
    firstNode = next;
    }
}
```

> 时间复杂性 : Q(listSize)

> 返回索引:

```
template<class T>
T& chain<T>::get(int theIndex) const
{    // 返回索引为theIndex的元素
        //若此元素不存在，则抛出异常
      checkIndex(theIndex);
        //移动到所需要的节点
    chainNode<T> *currentNode = firstNode;
     for (int i=0;i<theIndex;i++)
        currentNode = currentNode->next; //移向下一个节点
     return currentNode->element;
}
```

> 时间复杂性 : O(theIndex)

> 查找:

```
template<class T>
int chain<T>::indexOf(const T& theElement) const
{ //返回元素theElement首次出现时的索引,如果theElement不存在,则返回-1
  //查找元素
  chainNode<T> *currentNode = firstNode;
```

```
   int index = 0; // currentNode的索引
   while (currentNode!=NULL &&
              currentNode->element != theElement)
         { currentNode = currentNode->next;
            index++ ;
           }
   if (currentNode==NULL) return -1;
   return  index;
}
```

> 时间复杂性 : O(listSize)

> 删除:

```
template<class T>
void chain<T>::erase(int theIndex)
{  //删除表中索引为theIndex的元素
   //如果元素不存在，则抛出异常.
  checkIndex(theIndex);
   //索引有效，需要找要删除的元素节点
 chainNode<T> *deleteNode = firstNode;
 if (theIndex = = 0)
     {//删除表中首节点
     deleteNode = firstNode;
     firstNode = firstNode->next;
     }
 else { //用指针p指向第theIndex-1个节点
         chainNode<T> *p = firstNode;
         for (int i = 0; i<theIndex-1; i++)
                        p=p->next;
         deleteNode = p->next;
          p->next = p->next ->next;//删除deleteNode指向的节点
        }
 listSize--;
 delete deleteNode;
}
```

> 时间复杂性 : O(theIndex)

> 插入:

```
template<class T>
void chain<T>::insert(int theIndex, const T& theElement)
{
    //在索引theIndex处插入元素theElement;
    //如果theIndex无效，则引发异常
    if (theIndex<0 || theIndex>listSize) {……}
    //在链表表头插入
  if (theIndex=0)
    firstNode=new chainNode<T>(theElement, firstNode);
 else
   {//寻找新元素的前驱(第theIndex-1个元素 )
    chainNode<T> *p = firstNode;
    for (int i = 0; i < theIndex-1; i++)
            p = p->next;
   //将p移动至第theIndex-1个元素
    //在p之后插入
   p->next=new chainNode<T>(theElement, p->next);
  }
 listSize++;
}
```

> 时间复杂性 : O(theIndex)

> 输出重载:

```
template<class T>
void chain<T>::Output(ostream& out) const
{//将链表元素送至输出流 .
for (chainNode<T>* currentNode = firstNode;
          currentNode!=NULL;
          currentNode = currentNode->next)
   out << currentNode->element << " ";
}
```

> 重载<<

```
template<class T>
ostream& operator<<(ostream& out, const chain<T>& x)
{x.Output(out); return out;}
```

> 时间复杂性 : Q(listSize)

在单链表的基础上扩充单向循环链表，双向链表，双向循环链表等，本质上的操作都和单链表类似，注意将每种情况都考虑到就好。

优点：由于两个物理节点之间在存储上并没有什么实质的联系，可以是不连续的存储，所以链表的最大的就是它的存储是没有限制的。只要有足够的空间链表就可以继续扩充。在插入和删除方面，因为只需要修改 link 指针的指向就可以实现，所以不需要大量的移动元素，相比于公式化描述减少了时间复杂度。

缺点：相比于公式化描述的链表而言，因为多了一个要保存的下一节点的地址，所以要付出更多的空间。其次由于不是连续存储，所以只能依靠某一节点寻找下一节点，这就导致了查询的时间的消耗。因为插入和删除实际上是基于查询的基础上的，所以这就导致了插入和删除的复杂度并没有想象中的那么快。

6.2 循环链表和头结点

循环链表比单链表更加简洁高效

思路：
把线性表描述成一个单向循环链表（singly linked circular list），或简称循环链表（circular list），而不是一个单向链表；
在链表的前部增加一个附加的节点，称之为头节点（ header node）。

> 构造函数

```
template < class T >
circularListwithHeader<T>:: circularListwithHeader()
{ //构造函数
    headerNode = new chainNode<T>();
    headerNode ->next = headerNode;
    listSize=0;
}
```

> 在循环料表中搜索：

```
template<class T>
int circularListwithHeader<T>::indexOf(const T& theElement) const
{//返回元素theElement首次出现时的索引,如果theElement不存在,则返回-1
//将元素theElement放入头节点
 headerNode->element= theElement;
```

```
 //在链表中搜索元素
  chainNode<T> *currentNode = headerNode->next;
   int index = 0; // currentNode的索引
   while (currentNode->element != theElement)
         {
            currentNode = currentNode->next;
            index++ ;
           }
   //确定元素theElement是否存在
   if (currentNode==headerNode) return -1;
   else  return  index;
}
```

> 时间复杂性 : O(listSize)

6.3 双向链表

为了能快速找到一个节点的前驱，可以在单链表中的节点中增加一个指针域指向它的前驱.

> 双向链表的插入删除操作:

```
#include<iostream>
using namespace std;
template<class T>
class DoubleNode{
    friend Double<T>;
    private:
        T data;
        DoubleNode<T>*left,*right;
};
template<class T>
class Double{
    public:
        Double(){LeftEnd=RightEnd=0;}
        Double<T>& Delete(int k,T& x);
        Double<T>& Insert(int k,const T& x);
    private:
        DoubleNode<T>*LeftEnd,*RightEnd;
};
template<class T>
Double<T>& Double<T>::Delete(int k,T& x)
{
```

```
    DoubleNode<T>* p=LeftEnd;
    if(k<1||!LeftEnd) {exit(0);}//k小于1或链表为空则结束程序
    if(k==1){
        LeftEnd=LeftEnd->right;
    }else{
        DoubleNode<T>* cur=LeftEnd;
        for(int index=1;index<k&&cur;index++)//找到要删除的结点
        {
            cur=cur->right;
        }
        if(!cur) {exit(0);}//若k大于链表长度则结束程序
        cur->left->right=cur->right;
        cur->right->left=cur->left;
        delete cur;
    }
    return *this;
}

template<class T>
Double<T>& Double<T>::Insert(int k,const T& x)
{
    if(k<0) {exit(0);}//k小于0结束程序
    DoubleNode<T> *y=new DoubleNode<T>;
    DoubleNode<T> *cur=LeftEnd;
    for(int index=1;index<k&&cur;index++)
    {
        cur=cur->right;
    }
    if(k>0&&!cur) {exit(0);}//k大于链表长度则结束程序
    if(k)
    {
        y->left=cur;
        cur->right->left=y;
        y->right=cur->right;
        cur->right=y;
    }else{
        LeftEnd->left=y;
        y->right=LeftEnd;
        LeftEnd=y;
    }
    return *this;
}
```

6.5应用：

6.5.1箱子排序；

> 实现思想:
> 元素被分配到箱子时，使用相同的物理节点
> 把箱子中的元素收集时，把箱子链接起来

```
>template<class T>
void chain<T>::binSort(int range)
{  // 对链表中的节点排序
   //创建并初始化箱子
   chainNode<T> **bottom, **top;
   bottom = new chainNode<T>* [range + 1];
   top = new chainNode<T>* [range + 1];
    for (int b = 0; b <= range; b++)
              bottom[b] = NULL;
//把链表节点分配到相应箱子中
for (; firstNode; firstNode = firstNode->next)
    {// 将首节点firstNode添加到箱子中
       int theBin = firstNode->element; //元素类型转换到整型int
       if (bottom[theBin]==NULL) {//箱子为空
            bottom[theBin] = top[theBin] = firstNode;
       else //箱子非空, 放到箱子中top[theBin]之后的位置
         {
           top[theBin]->next = firstNode;
           top[theBin] = firstNode;
         }
     }
//把箱子中的节点收集到有序链表
chainNode<T> *y = NULL;
for (int theBin = 0; theBin <= range; theBin++)
     if (bottom[theBin]!= NULL) {//箱子非空
           if (y = NULL)    // 第一个非空的箱子
              firstNode = bottom[theBin];
           else //不是第一个非空的箱子
              y->next = bottom[theBin];
           y = top[theBin];
if  (y!= NULL)    y->next = NULL;
delete [ ] bottom;
delete [ ] top;
}
```

> 时间复杂性：Q(n+range)

6.5.2基数排序;

> 思路：
> 把数按照某种基数r分解为数字，然后对数字进行箱子排序.

6.5.4并查集

等价类中关键的两个方法就是 find 和 union（查找某一元素的所在等价类，以及合并两个等价类）。

> 实现思想：
> 针对每个等价类设立一个相应的链表——等价类链表
> 每个元素都在一个等价类链表中
> initialize ： 为每个元素设置一个只拥有该元素的链表
> Unite：合并两个链表
> Find：查找元素所在的链表

链表实现的等价类是将每一个等价类用一条链表来描述，链表的头就是该等价类的等价元，链表中的每个节点要保存的是 E（所属等价类），size（等价类大小实际使用时只要修改链表的第一个元素的 size 就可以了），link（下一节点的地址）这三个信息。两个等价类的合并实际就是两个链表的合并。在合并（i，j）时把较短的一条链中的节点所在的等价类E，全部修改为 j，然后将 size 修改，最后是将第二条链的首部接到第一条链的二号位置，将第一条链的二号位置之后的链接到第二条链的尾部（这样做是有原因的，很多人可能想的直接把第二条链接在第一条链的后面，这样做就需要找到第一条链的最后一个位置，这是需要时间的，所以不如从头插入；而将一号链的二号位置及之后的节点接到第二条链是可以接受的，因为我们在修改第二条链的等价类时就已经找到了最后一个节点的位置。）

第7章 数组和矩阵

本章概述

在实际应用中，数据通常以表的形式出现，例如我们经常用到的数组。尽管用数组来描述表是一种最直观，最自然的方式，但是直观自然的代价是程序运行时间和空间的增加。
因此，我们在本章探讨：数据表如何采用自定义的描述方式，来提高程序的效率。我们把精力放在如何将数据表更有效地存储在内存中，并能方便地提取数据表中的元素。
我们这里所说的数据表，大部分时间指的是矩阵。毕竟矩阵在计算机领域占有举足轻重的地位。矩阵经常用二维数组来描述，想要了解矩阵，就先要了解数组的**存储结构**。

7.1 数组

7.1.1 数组的定义

数组是由n（n≥1）个**相同类型**的数据元素构成的有限序列，每个数据元素称为一个**数组元素**，每个元素都是(索引，值)的数对组合，显然每个数对的索引都不相同。
数组一旦被定义，其维数和每一维的长度就不再改变。因此，除了数组的初始化和销毁外，数组只会有**存取元素**和**修改元素**的操作。
补充一点**数组和线性表的关系**：数组是线性表的推广。一维数组可以视为一个线性表；二维数组可以是视为元素为线性表的线性表，以此类推。

7.1.2 数组的索引

了解数组我们先从索引开始。数组的索引（下标）具有如下形式：

> $[i_1] [i_2][i_3]\cdots[i_k]$

例如一个三维整形数组score，可用如下语句创建：

> int score[u_1][u_2][u_3]

关于下标的范围我们就不说了，我们可以看到该数组最多可容纳**n= u_1 * u_2 * u_3 个值。根据每个整数占4个字节，我们可以得出该数组存储空间大小为4n**个字节。
另外，数组的存储空间是连续的。如果数组存储空间的起始字节位置为**start**，那该空间将一直延伸到地址为**start+sizeof(score)-1**的字节。

7.1.3 数组的映射

以一维数组A[0…n-1]为例,其映射关系为：

> $A(a_i) = A(a_0) + i (0 \le i < n)$

对于多维数组，我们有两种映射方法：**行主映射**和**列主映射**。
我们先假设二维数组**map[u_1][u_2]**，有u_1行和u_2列。
行主映射的基本思想是：先存储行号较小的元素，行号相等先存储列号较小的元素，对应的映射函数是：

> $map(i_1, i_2) = i_1 * u_2 + i_2$

类似的，列主映射的基本思想是：先存储列号较小的元素，列号相等的行号较小的元素，对应的映射函数是：

> $map(i_1, i_2) = i_2 * u_1 + i_1$

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]

图4-1　int score[3][6]的索引排列表

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

a)

0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

b)

图4-2　二维数组的映射

a) 行主映射　b) 列主映射

三维和维数更大的数组映射思想也是遵循上面两种映射方法，相信这些知识都不难理解。

7.2 矩阵

我们已经在大一接触过矩阵。一个**m×n**的矩阵，是一个m行、n列的表，其中m和n是矩阵的维数。

7.2.1 类matrix

一个**rows×cols**的整形矩阵**M**课用如下的二维整数数组来描述：

> int x[rows][cols];

值得注意的是，对于矩阵来说，矩阵的行和列是从1开始数起的。因此，矩阵元素M（i，j）是对应于二维数组的x[i-1][j-1]。可是，如果使用这种描述方法并不直观。
在这里，我们用的是另一种矩阵描述方法：把二维数组定义为

> int x[rows+1][cols+1]

对这个二维数组，我们对形式为[0][*]和[*][0]的元素弃之不用，这样矩阵的索引就可对应二维数组的索引。

我们来看矩阵类matrix的声明：

```
template <class T>
class matrix {
    friend ostream operater << (ostream &, const matrix<T>&);
public:
    //矩阵的构造函数、复制构造函数以及析构函数↓
    matrix(int theRows = 0, int theColumns = 0);
    matrix(const martrix<T>&);
    ~matrix() { delete[] element; }
    int rows() const { return theRows; }
    int columns() const { return theColumns; }
    T& operator() (int i, int j) const;
        //矩阵最常见的操作有：矩阵转置、矩阵加、矩阵乘。
        //我们要重载矩阵的算术操作符
    matrix<T>& operator=(const matrix<T>&) ;
    matrix<T> operator+() const;
    matrix<T> operator+(const matrix<T>&) const;
    matrix<T> operator-() const;
    matrix<T> operator-(const matrix<T>&) const;
    matrix<T> operator*(const matrix<T>&) const;
    matrix<T>& operator+=(const T&) ;
private:
    int theRows,                //矩阵的行数
        theColumns;         //矩阵的列数
```

```
    T *element;                 //类matrix用一个一位数组存储
};
```

> 矩阵类matrix的构造函数和复制构造函数

```
//矩阵类matrix的构造函数的具体实现
template<class T>
matrix<T> ::matrix(int thwRows, int theColumns) {
        //检验行数和列数的有效性：这一步在后面的诸多方法都有安排上
    //保证矩阵的行数列数规范，是后面一切操作的前提
    if (theRows < 0 || theColumns < 0)
        throw illegalParameterValue
           ( "Rows and columns must be >=0");
    if ((theRows == 0 || theColumns == 0)
                && (theRows != 0 || theColumns != 0))
        throw illegalParameterValue(
               "Either both or neither
               rows and columns should be zero");
        //创建矩阵
    this->theRows = theRows;
    this->theColumns = theColumns;
    element = new T(theRows *theColumns);

}
//矩阵类matrix的复制构造函数
template<class T>
matrix<T> & matrix<T> ::operator=(const matrix<T> &m){
        //既然是复制构造函数那就是把形参里的（&m）复制给（*this）
    if(this! =&m){
        //当然不能自己复制自己了
        delete[] element;
        theRows = m.theRows;
        theColumns = m.theColumns;
        element = new T{ theRows * theColumns };
        //这里每复制一个元素，就执行一次copy（）
        copy(m.elementl,
             m.elementl + theRows * theColumns ,element);

    }
    return *this;
}
```

> 时间复杂度均为O（theRows *theColumns）

为了使矩阵的操作更简便，我们对（）操作符进行重载，这样我们可以用左右括号来表示矩阵的索引。

> 矩阵类matrix对()操作符的重载

```
template <class T>
T& matrix<T> ::operator() (int i ,int j) const{
        //返回元素element的引用，这个引用既可以用来取值，也可以用来赋值
    if (i<1 || i>theRows || j<1 || j>theColumns)
        throw matrixIndexOutOfBounds();
    return element[（i - 1） * theColumns + j - 1]];

}
```

接下来就是矩阵的加法和乘法

> 矩阵加法

```
template<class T>
matrix<T> matrix<T>::operator+ (const matrix<T>& m) const {
        //返回矩阵w=（*this）+m
    if (theRows != m.theRows || theColumns != m.theColumns) {
        throw matrixSizeMismtach();
    }
        //生成结果矩阵
    matrix<T> w(theRows, theColumns);
    for (int i = 0; i < theRows; i++)
        w.element[i] = element[i] + m.element[i];
    return w;
}
```

> 时间复杂度：O（theRows *theColumns）

> 矩阵乘法

```
template <class T>
matrix<T> matrix<T>::operator *(const matrix<T> &m) const {
        //返回结果矩阵w=（*this）*m
```

```
    //该方法的整体思路跟笔算矩阵乘法相同:
    //即矩阵*this 的某一行，依次乘矩阵m的每一列
    //这样就可以得出结果矩阵w某一行的全部元素
    //接着再用矩阵*this 的下一行，依次乘矩阵m的每一列
    //这样类推即可得到完整的结果矩阵w
if (theColums != m.theRows)
    throws matrixSizeMismatch();
    //初始化结果矩阵w
matrix<T> w(theRows, m.theColumns);
//这里定义*this 、矩阵m和结果矩阵w的游标
//全赋值为0，目的是从矩阵的（1，1）开始定位
int ct = 0, cm = 0, cw = 0;

for (int i = 1; i <= theRows; i++) {

    for (int j = 1; j <= m.theColumns; j++) {

        T sum = element[ct] * m.element[cm];
        for (int k = 2; k <= theColumns; k++) {
            // *this中第i行的下一项
            ct++;
            //矩阵m中第j列的下一项
            cm += m.theColumns;
            sum += element[ct] * element[cm];
        }
        //存储在结果矩阵w（i，j）中
        //由于结果矩阵w也是用数组，按行主映射存放数据的
        //因此这里用[cw++]即可移动到下一个存储位置
        w.element[cw++] = sum;
        /*最里层循环结束，说明算完了结果矩阵w中的一个元素
        此时ct指向的是*this矩阵某行的最后一个元素
        cm指向的是m矩阵某一列的最后一个元素
        现在的任务是:
        把ct移回改行的起始位置
        因此ct =ct-（theColumns-1）就不难理解了
        把cm移到下一列的起始位置
        由于j在上一次循环过后已经自加一次了
        因此cm=j即可指向矩阵m的下一列的起始位置
        */
        ct -= theColumns - 1;
        cm = j;
    }
    /*当第二层循环结束，意味着结果矩阵w的第j行已经全部算完了
    现在我们要算结果矩阵w的下一行
    具体做法是:
    *this矩阵的下一行，分别称矩阵m的每一列
    因此ct =ct+theColums就是要指向下一行
```

```
            cm=0意味着指向矩阵m的m（i，j）
            */
            ct += theColumns;
            cm = 0;
        }
        /*
        当第三层循环结束，显然意味着结果矩阵w的每一个元素我们已经算好了*/
        return w;
    }
```

7.3 特殊矩阵

方阵是行数和列数相同的矩阵。正如下图所展示的，我们接下来讲的特殊矩阵都是方阵，分别是**对角矩阵**、**三对角矩阵**、**三角矩阵**、**对称矩阵**。
特殊矩阵中，一些相同的矩阵元素或零元素的分布有一定的规律性，因此，我们要研究如何将这些特殊矩阵**压缩存储**，以节省存储空间。

2 0 0 0	2 1 0 0	2 0 0 0	2 1 3 0	2 4 6 0
0 1 0 0	3 1 3 0	5 1 0 0	0 1 3 8	4 1 9 5
0 0 4 0	0 5 2 7	0 3 1 0	0 0 1 6	6 9 4 7
0 0 0 6	0 0 9 0	4 2 7 0	0 0 0 0	0 5 7 0
a)	b)	c)	d)	e)

图4-4　4×4矩阵

a) 对角矩阵　b) 三对角矩阵　c) 下三角矩阵　d) 上三角矩阵　e) 对称矩阵

7.3.1 对角矩阵

书本对角矩阵的定义是：**M是一个对角矩阵，当且仅当i≠j时，有M（i，j）=0**
如图，对角矩阵无论是矩阵结构和映射方式都十分直观，理解起来不具有难度。

一、对角矩阵

定义 所有非对角线元素都是0的矩阵称为对角矩阵.

$$A=\begin{bmatrix}-2 & 0 & 0\\ 0 & 3 & 0\\ 0 & 0 & 5\end{bmatrix}.$$

一般对角矩阵

$$A=\begin{bmatrix}a_1 & & & \\ & a_2 & & \\ & & \ddots & \\ & & & a_n\end{bmatrix}.$$

1

> 类diagonalMatrix的声明和构造函数

```
template<class T>
class diagonalMatrix {
public:
    diagonalMatrix(int theN = 10);
    void set(int i, int j, const T&  );
    T get(int i , int j ) const;
    ~diagonalMatrix{ delete[] element; };
private:
    int n;          //矩阵维数
    T *element;     //存储对角矩阵的一维数组
};

template<class T>
diagonalMatrix<T>::diagonalMatrix(int theN) {
        //diagonalMatrix的构造函数
        //构造前先检验theN的值是否有效
    if (theN < 1) {
        throw illegalParameteterValue(
            "Matrix size must be>10");
```

```
    }
    n = theN;
    element = new T[n];
}
```

> 类diagonalMatrix的方法get

```
template<class T>
T diagonalMatrix<T>::get(int i ,int j) const {
   //该方法是要返回矩阵中（i，j）位置上的元素
   //检验i和j的值是否有效
   if (i<1 || j<1 || i>n || j>n)
    throw matrixIndexOutOfBounds();

   if (i == j)
    return element[i - 1];  //若i等于j，该元素为对角线上元素
   else
    return 0;       //非对角线元素
}
```

> 类diagonalMatrix的方法set

```
template<class T>
void diagonalMatrix<T>::set(int i, int j, const T& newValue) {
    if (i<1 || j<1 || i>n || j>n)
        throw matrixIndexOutOfBounds();

    if (i == j)
        element[i - 1] = newValue;
    else
        if (newValue != 0)
            throw illegalParameterValue
                ("nondiagonal element must be zero");
}
```

7.3.2 三对角矩阵

三对角矩阵的定义是：
M是一个三对角矩阵，当且仅当|i-j|>1时，M（i，j）=0

如图，在一个n×n三对角矩阵T中，非0元素排列在如下的三条对角线上：
1、主对角线——i=j。

2、主对角线之下的对角线(称低对角线)——i=j+1。

3、主对角线之上的对角线(称高对角线)——i=j-1。

主对角线及其相邻两侧的一条对角线之外的元素都为0

$$A=\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix}$$

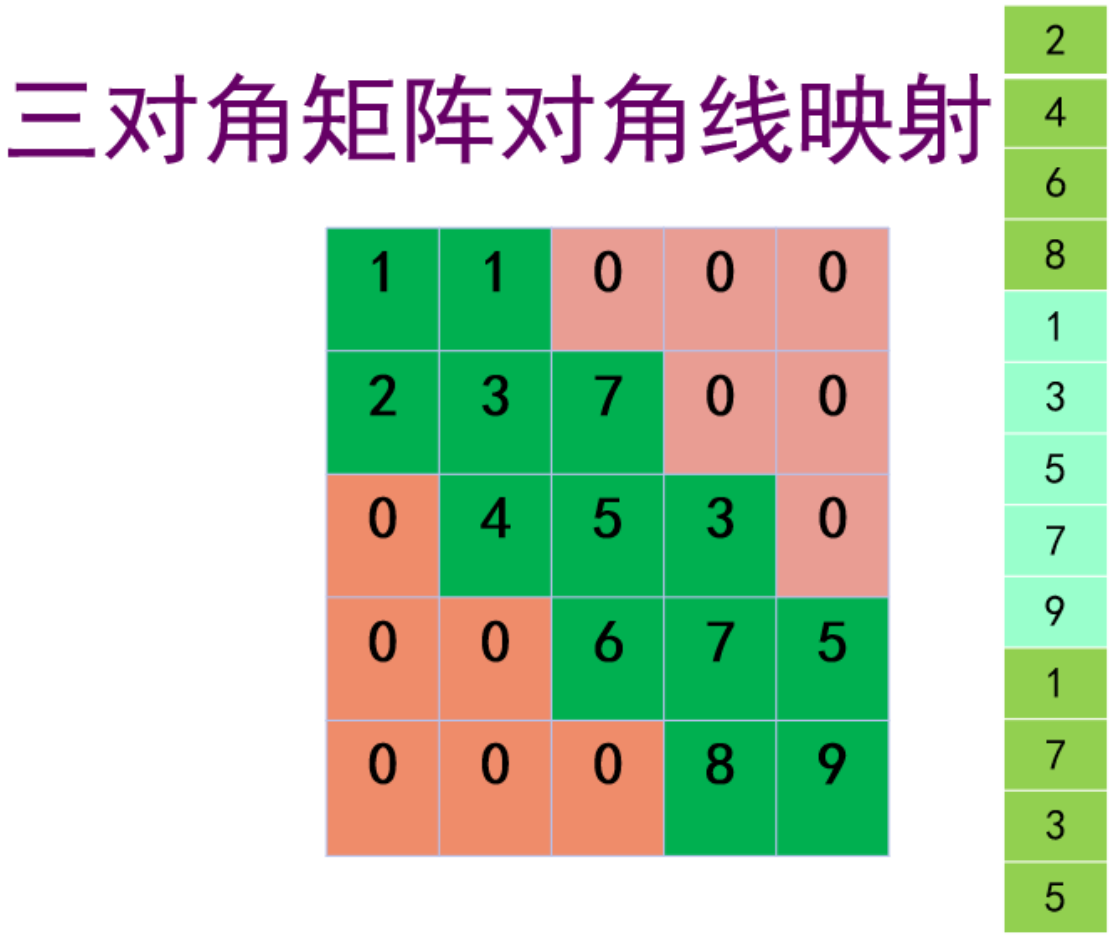
> 三对角矩阵的方法get

对于三对角矩阵的get方法，我们是根据其对角线映射的方法实现的。具体实现方法可以参见下列代码。

```
template<class T>
void tridigonalMatrix<T>::set(int i, int j ) const {
    //返回矩阵中（i，j）位置上的元素
    //先要检验i和j时候有效
    if (i < i || j<1 || i>n || in) {
        throw matrixIndexOutOfBounds();
    }
    //在这个程序中我们是按逐条对角线映射的
    //先是按下对角线，再按中间的煮对角线，最后按上面的上对角线
    //我们可以看到，下对角线和上对角线的元素共有n-1个
    //中对角线的元素有n个
    switch (i - j) {
    case 1:
        //若该元素在对角线上，下标范围为[0,n-2]
        return element[i - 2] ;
    case 0:
        //若元素在主对角线上，元素下标是从n-1开始的
        //因此下标范围是[n-1,2n-2]
        return element[n + i - 2] ;
    case -1:
```

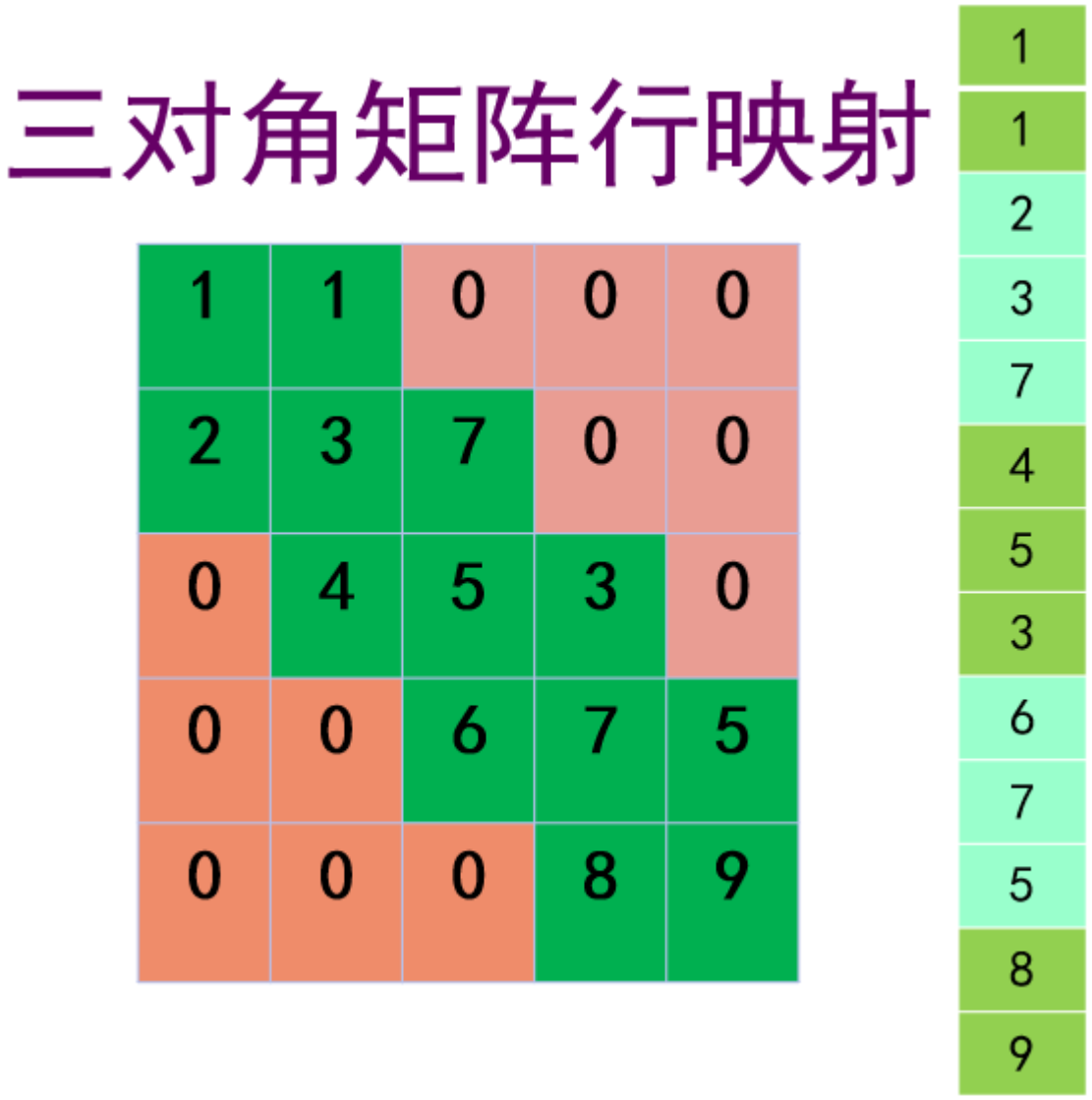
```
            //若元素在上对角线上,前面已经有下对角线和中对角线
            //因此下标范围是[2n-1,3n-2]
            return element[2 * n + i - 2] ;
        default: throw illeagalParameteterValue();
        }

    }
```

下图充分说明了三对角矩阵对角线映射的特征。这里要说明的是，我们下面讲的**矩阵压缩存储只有按行映射和按列映射**两种方式，按对角线映射是没法参加压缩存储的。



由于三对角矩阵除了三条对角线外其余元素都为0，一个绕不开的话题就是**压缩存储**。就是我们可以只存储主对角线及其上、下两侧次对角线上的元素，其他的零元素一律不存储。对一个n×n的三对角方阵A，元素总数有n2个，而其中非零的元素共有3n-2个。因此，存储三对角矩阵时最多只需存储3n-2个元素，节省了存储空间。
那实现压缩存储的途径，就是了解三对角矩阵的**映射方式**，而这就是我们下面要讨论的内容。

我们先来看一下三角矩阵行映射：



上面行映射体现的思想是，给你一个坐标a[i,j]。要先考虑前（i-1）行一共有多少个元素，然后再考虑这个位置是在第i行中的第几个元素。两者加起来就能得到该元素映射的顺序。
三对角矩阵，从第二行开始选中的元素的个数都为3个。对于a[i,j]将要存储的位置k，首先前(i-1)行元素的个数是(i-2)*3 +2(第一行元素的个数为2)，又a[i,j]属于第i行被选中元素的第j-i+1个元素，所以k= (i-2)3 +2 + j-i+1 = 2i+j-3。

实际上，三对角矩阵的**列映射**也是同样的道理：
首先我们得到坐标a[i,j]，我们可以看到该元素在第j列，那么前面j-1列的元素共有（j-2）*3+2个。然后我们再分析，该元素是第j列中的第几个元素，我们经过分析可以得到是第i-j+1个元素。因此我们得到
k=（j-2）*3+2+i-j+1=2j+i-3。

7.3.3 三角矩阵

如图，三角矩阵有两类：**上三角矩阵**和**下三角矩阵**

三、三角形矩阵

定义 主对角线下(上)方的元素全为0的方阵称为上(下)三角矩阵 .

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}, B=\begin{pmatrix} b_{11} & 0 & \cdots & 0 \\ b_{21} & b_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}.$$

$$a_{ij}=0, j<i. \qquad b_{ij}=0, j>i$$

上三角矩阵　　下三角矩阵

4

书本上上三角矩阵的定义是：

M是一个下三角矩阵，当且仅当i<j时，M（i，j） =0

下三角矩阵(lowertriangular)

1	0	0	0	0
2	3	0	0	0
3	6	5	0	0
4	7	9	7	0
5	8	1	2	9

上三角矩阵的定义是：

书本上**M是一个上三角矩阵，当且仅当i>j时，M（i，j） =0**

理解三角矩阵存储结构的重点，在于理解其映射方法。下面，我们将详细分析上三角矩阵和下三角矩阵的映射方法。跟三对角矩阵的压缩存储一样，主要有**行主映射**和**列主映射**。

对于**上三角矩阵**来说，给定一个坐标a[i,j]。

> 如果**按行映射**，我们先算前i-1行有多少个元素，再算该元素在第i行中是第几个元素。
> 经过观察我们发现，从第1行开始，每往下一行，元素个数减一。第一行元素个数为n，到第n-1行，元素个数变为n-i+2个。因此我们用等差数列求和公式，求得前n-1行的元素个数为[n+(n-i+2)]·(i-1)/2。
> 接下来我们看一下该元素在第i行中的顺序，很明显为j-i+1。换句话说在第i行，该元素前面有j-i个非零元素。
> 因此我们算得元素a[i,j]前面一共有(j-i)+[n+(n-i+2)]·(i-1)/2。换句话说，该元素在存储矩阵的数组中，索引就为(j-i)+[n+(n-i+2)]·(i-1)/2。由于公式化简后也很复杂，这里就不展示化简结果了。

> 上三角矩阵的**按列映射**也是同样的道理。前j-1列共有元素j·(j-1)/2个。在第j列，排在该a[i,j]前面的元素有i-1个。因此，该元素的索引为
> (i-1)+[j·(j-1)]/2。

由于上三角矩阵和下三角矩阵在结构上是相似的，因此只要理解了上面所讲述的映射规则，相信下三角矩阵的映射规则也就不难理解了。
对**下三角矩阵**来说，给定一个坐标a[i,j]。

> 如果按行映射，前i-1行共有元素i·(i-1)/2个。在第i行，排在该a[i,j]前面的元素有j-1个。因此，该元素的索引为
> (j-1)+[i·(i-1)]/2。

> 如果按列映射，前j-1列元素共有[n+(n-j+2)]·(j-1)/2。在第j列，排在a[i,j]前面的元素有(i-j)个。因此该元素索引为:
> (i-j)+[n+(n-j+2)]·(j-1)/2

下面是课本给出的代码，该代码体现的是下三角矩阵按行映射的情况下，如何对元素的查找。

> lowerTriangularMtarix的方法set

```
template<class T>
T tridigonalMatrix<T>::get(int i, int j) const {
    //给（i，j）元素符新值
    //惯例检验i和j的值是否合法
    if (i<1 || j<1 || i>n || j>n) {
        throw matrixIndexOutOfBounds();
    }
    //若（i，j）在下三角区域，当且仅当i>=j
    if (i >= j) {
        element[i * (i - 1) / 2 + j - 1] = newValue;
    }
    else
        if (newValue != 0)
            throw illegalParameterValue
           ("elements not in lower triangle must be zero");
}
```

7.3.4 对称矩阵

对称矩阵的定义是:
M是一个对称矩阵，当且仅当对于所有的i和j，M（i，j）=M（j，i）
对称矩阵可以视为下三角矩阵或上三角矩阵，用三角矩阵的表示方法，用一个大小为**n（n+1）/2**的一位数组来表示。

7.4 稀疏矩阵

7.4.1 基本概念

一个m×n的矩阵，如果大多数元素都是0，则称为**稀疏矩阵**，一个矩阵如果不是稀疏的，则称为**稠密矩阵**，至于稀疏矩阵和稠密矩阵之间没有明显的界限。
对于对角矩阵和三对角矩阵这样的稀疏矩阵，他们的非零元素的分布是有规律，且确定的。我们这一部分主要是讨论非零元素分布不均匀的稀疏矩阵。
如图，我们通常是用数组term来存储稀疏矩阵的非零元素。

图4-10 稀疏矩阵及其数组描述

a) 4×8矩阵 b) 数组描述

> 类spareMatrix的头

```
template<class T>
class sparseMatrix {
public :
    //稀疏矩阵的转置
    void transpose(sparseMatrix<T>  &b);
    //稀疏矩阵的相加
    void add(sparseMatrix <T> &b, sparseMatrix <T> &c);
private:
    int rows,                          //矩阵行数
        cols;                         //矩阵列数
```

```
    arrayList<matrixTerm<T>>  terms; //非0项表

};
```

> 重载输出操作符<<

```
template<class T>
ostream & operator <<(ostream & out, sparseMatrix<T> &x) {
        //将x放入输出流
        //这里是输出矩阵的行数、列数、和非零元素
    out << "rows =" << x.rows << "columns =" << x.cols << endl;
    out << "nonzero terms =" << x.terms.size()endl;
        //输出矩阵的非零项，一行一个
    for (arrayList<matrixTerm<T>> ::iterator i = terms.begin();
        i != x.terms.end(); i++) {
        out << "a(" << (*i).row << "," << (*i).col
                    << ")=" << (*i).value << endl;
    }
    return out;
}
```

> 重载输入操作符

```
template<class T>
istream& operator >>(istream& in, sparseMatrix<T> &x) {
    int numberOfTerms;
        //输入稀疏矩阵的特征
    cout << "Enter number of rows,columns ,and #terms" << endl;
    in >> x.rows >> x.cols >> numberOfTerms;

    //设置x.term的大小，确保足够的容量
        x.terms.reset(numberOfTerms);
        //输入项
    matrixTerm<T> mTerm;
    for (int i = ; i < numberOfTerms; i++) {
        cout << "Enter row,column,and value of term"
                    << (i + 1) << endl;
        in >> mTerm.row >> mTerm.col >> mTerm.value;
        x.terms.set(i, mTerm);

    }
    return in;
```

```
}
```

> 稀疏矩阵的转置

```
template<class T>
void sparseMatrix<T> ::transpose(sparseMatrix<T> &b) {
    /*我们都知道矩阵的转置是把元素的行和列互换，
    而这一点在term数组中可以轻易做到
    但稀疏矩阵的转置关键在于：
    矩阵转置后元素的顺序如何确定
    我们来分析一下：
    矩阵转置后，元素的顺序是每一列从上往下数，然后列之间从左往右数
    对稀疏矩阵来说：
    （转置前矩阵每列的非零元素个数） =（转置后每一行的非零元素数）
    而且参照于矩阵行映射的特征。
    */
        //b矩阵是对（*this）向量的转置
        //设置转置矩阵的特征，转置矩阵的行数和列数显然可以看出来的
    b.cols = rows;
    b.rows = cols;
    b.terms.reSet(terms.size());
        //初始化以实现转置
    int* colSize = new int[cols + 1];
    int * rowNext = new int[cols + 1];
        //寻找 *this 中每一列的项的数目
        //colSize[]装的是：原本矩阵每一列的非零元素数目
    for (int i = 1; i <= cols; i++)
        colSize[i] = 0;

    for (arrayList<matrixTerm<T>> ::iterator i = terms.begin();
        i != terms.end(); i++) {
        colSize[(*i).col]++;
    }

        /*rowNext[]里面装的是 b 中每一行的起始点
        也就是说rowNext[i]指的是：
        转置前的元素在第i列，
        那么按理说这个元素要放在转置后矩阵的term数组中
        而这个元素就是放在第rowNext[i]个位置中
        不难想到，rwoNext[]的元素得值是逐个递增的
        */
    rowNext[1] = 0;
    for (int i = 2; i <= cols; i++) {
        rowNext[i] = rowNext[i - 1] + colSize[i - 1];
```

```
    }
        //实施从 *this 到b的转置复制
    martixTerm<T> mTerm;
    for (arrayList<matrixTerm<T>>::iterator i = terms.begin();
             i!= terms.end(); i++) {
                /*这里的j指的是:
        这个元素存储在mterm的位置为rowNext[(*i).col]
        看仔细了↓
        rowNext[(*i).col]的值是先赋给j，再自增
        为什么要自增:
        自增使得下一个原本同一列的元素能排在rowNext[(*i).col]+1
        由于矩阵是行主映射的
        因此同一列的元素，行数大的排在行数小的元素前面
        而矩阵转置之后，原本同一列元素的相对位置是不变的
        这是整个算法的精髓之处
        */
        int j = rowNext[(*i).col]++;
        mTrem.row = (*i).col;
        mTrem.col = (*i).row;
        mTern.value = (*i).value;
        b.terms.set(j, mTerm);
    }
}
```

> 两个稀疏矩阵相加

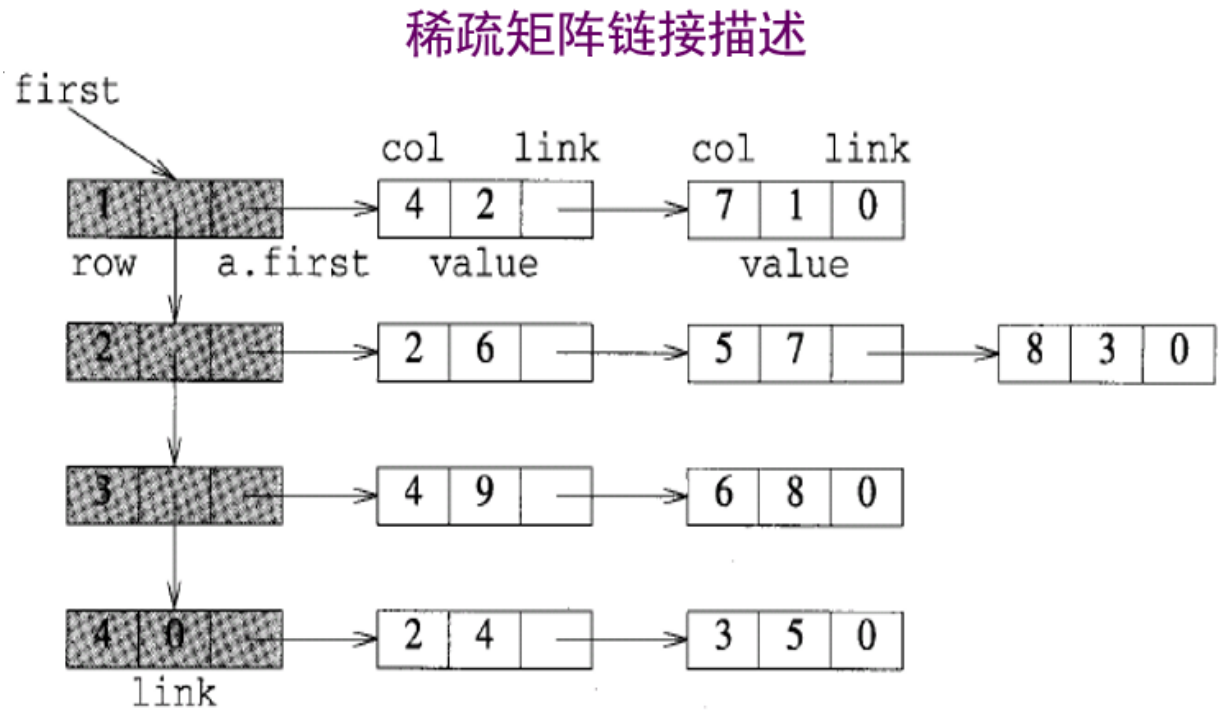
```
template<class T>
void sparseMatrix<T>::add(
    sparseMatrix<T>& b, sparseMatrix<T>& c) {
    /*实际上两个稀疏矩阵相加的整体思路比转置更容易理解:
    实际上是两个矩阵term数组的合并:
    要是有重复的点则相加;
    要是在某一个矩阵中单独出现的点可以直接搬到合并的term数组中
    因为在某个矩阵中单独出现，在另一矩阵相同位置的元素值为0
    */
        //计算 c =(*this) + b
        //先检验两个矩阵可否相加
    if (rows != b.rows || cols != b.cols) {
        throw matrixSizeMismatch();
    }
        //设置结果矩阵c的特征
    c.rows = rows;
    c.cols = cols;
    c.terms.clear();
    int cSize = 0;
```

```
    //定义 *this 和 b 的迭代器
arrayList<matrixTerm<T>>::iterator it = terms.begin();
arrayList<matrixTerm<T>>::iterator ib = b.terms.begin();
arrayList<matrixTerm<T>>::iterator itEnd = terms.end();
arrayList<matrixTerm<T>>::iterator ibEnd = b.terms.end();

    //遍历*this 和 b，把相关的项相加
while (it != itEnd && ib != ibEnd) {
    //tIndex和bIndex表明的是该非零元素在数组中的位置
    int tIndex = (*it).row * cols + (*it).col;
    int bIndex = (*ib).row * cols + (*ib).col;
    /*这里的思想是：
    我们要把矩阵*this和矩阵b的所有非零元素整合到结果矩阵c中
    那矩阵是行主映射的。
    所以位置靠前（即索引较小的元素）会先录入矩阵c
    那么我们要找到矩阵*this 和矩阵b中索引最小的
    下面有三种情况：
    1、待录入索引最小的元素在矩阵*this中
    2、待录入索引最小的元素在矩阵b中
    3、待录入索引最小的元素在矩阵*this和矩阵b中同时出现
    */
    if (tIndex < bIndex) {
        c.terms.insert(cSize++, *it);
        it++;
    }
    else {if (tIndex == bIndex)
                    //若两项在同一个位置
        {
    //这里容易踩坑的一点是：
    //要检验两个相同位置的元素加起来是否为0
    //要是和为0，将不会在矩阵c中显示
    if ((*it).value + (*ib).value != 0) {
    matrixTerm<T> mTerm;
    mTerm.row = (*it).row;
    mTerm.col = (*it).col;
    mTerm.value = (*it).value + (*ib).value;
    c.terms.insert(cSize++, mTerm);
    }
    it++;
    ib++;
     }
    else {
                    //一项在后
    c.terms.insert(cSize++, *ib);
    ib++;
    }
```

```
    }
    }
    //当矩阵*this或者矩阵b的非零元素全部录入后
    //把剩余的全录入矩阵c的terms数组中
    for (; it != itEnd; it++)
        c.terms.insert(cSize, *it);
    for (; ib != ibEnd; ib++)
        c.terms.insert(cSize++, *ib);

    }
}
```

7.4.2 用多个线性表描述稀疏矩阵

用多个线性表描述矩阵，就是把每行的非零元素串接在一起，构成一个链表，这样的链表我们成为**行链表**。
下图为行列表的结构



在这个部分，我们将重点了解，用行列表描述的稀疏矩阵，其转置是如何实现的。

```
template <class T>
void LinkedMatrix<T>::
Transpose(LinkedMatrix<T> &b) const{ //转置*this，并把结果放入b
    b.a.Erase();//删除b中所有节点
    //创建用来收集b中各行元素的箱子
    Chain<CNode<T>> *bin;
```

```
    bin=new Chain<CNode<T>>[cols+1];
    //头节点遍历器
    ChainIterator<HeadNode<T>> p;
    //h指向*this的第一个头节点
    HeadNode<T> *h=p.Initialize(a);
    //把*this的元素复制到箱子中
    while(h){//检查所有的行
        int r=h->row;//行链表中的行数
        //行链表遍历器
        ChainIterator<CNode<T>> q;
        //z指向行链表的第一个节点
        CNode<T> *z=q.Initialize(h->a);
        CNode<T> x;//临时节点
        //*this第r行中的元素变成b中第r列的元素
        x.col=r;
        //检查*this中第r行的所有非0元素
            while(z){//遍历第r行
            x.value=z->value;
            bin[z->col].Append(x);
            z=q.Next();//该行的下一个元素
    }
        h=p.Next();//继续下一行
}
    //设置b的维数
    b.rows=cols;
    b.cols=rows;
    //装配b的头节点链表
    HeadNode<T> H;
    //搜索箱子
    for(int i=1;i<=cols;i++)
        if(!bin[i].IsEmpty()){//转置矩阵的第i行
            H.row=i;
            H.a=bin[i];
            b.a.Append(H);
            bin[i].Zero();//链表置空
            }

    H.a.Zero();//置空H，防止H释放链表
    delete []bin;
}
```

第8章 栈

本章概述

把线性表的插入和删除操作放在同一端进行就能得到栈数据结构。我们可以从相应线性表类派生出栈类，也可以直接创建基于数组和链表的栈类。直接派生的优点是减少编码量，提高程序可靠性，缺点是运行效率低。

8.1 定义和应用

栈：是特殊线性表，插入和删除只能在表的同一端进行，允许插入和删除的这一端栈顶，另一端叫栈底，是一个后进先出的数据结构。

8.2 抽象数据类型

栈的抽象数据类型给出了栈的最基础常用操作.

> C++ 抽象类栈

```
template < class T>
class stack
{
    public:
    virtual ~stack () {}
    virtual bool empty () const = 0;
    //返回true,当且仅当栈为空
    virtual int size () const = 0;
    //返回栈中元素个数
    virtual T & top() = 0;
    //返回栈顶元素的引用
    virtual void pop() = 0;
    //删除栈顶元素
    virtual void push( const T & theElement) = 0;
    //将元素theElement压入栈顶
};
```

8.3 数组描述

把数组线性表的右端定义为栈顶,那么入栈和出栈就是线性表在最好情况下的插入和删除,两个操作的时间都为O(1)。

8.3.1 作为一个派生类实现

从类arrayList和stack派生类derivedArrayStack，其成员函数通过调用基类成员函数实现。

> 一个从类arrayList 派生的数组栈类

```
template <class T>
class derivedArrayStack: private arrayList<T>,public stack <T>
{
    public:
    derivedArrayStack(int initialCapacity = 10)
         :arrayList< T > (initialCapacity) {}
    bool empty () const
         { return arrayList <T>::empty(); }
    int size () const
         { return  arrayList <T> ::size(); }
    T & top()
    {//返回栈顶元素
        if (arrayList <T>::empty())
           throw stackEmpty();
        return get(arrayList <T>::size()-1 );
    }
    void & pop()
    {//删除栈顶元素
        if (allayList <T>::empty())
           throw stackEmpty();
        erase(arrayList <T>:: size()-1);
    }
    void push(const T & theElement)
    {//插入
        insert(allayList <T>::size(),theElement)
    }
};
```

> 时间复杂度：构造函数是O（initialCapacity），插入操作在数组长度不增加时为Θ（1），在增加时为O（stack size）。其他复杂度都是为Θ（1）。

因为get和erase遇到空栈都会抛出异常，所以在top和pop中可以删除对空栈的检查。可以用try—catch结构代替对空栈的检查。

8.3.2 类arrayStack

为了减少不必要的下标检查和往回复制，我们可以定制数组栈。每种操作的时间复杂度和derivedArrayStack的相应操作一样。

> 类arrayStack:

```
template<class T>
class arrayStack : public  stack<T>
{
    public:
       arrayStack(int initialCapacity = 10);
       ~ arrayStack() { delete [] stack; }
       bool empty() const { return stackTop == -1; }
       int size() const { return stackTop+1; }
       T & top();
       void pop();
       void push(const T& theElement);
    private:
       int stackTop;  //当前栈顶的索引值
       int arrayLength;  //栈容量
       T * stack;     //元素数组
};
```

> 构造函数:

```
template < class T >
arrayStack<T> :: arrayStack(int initialCapacity = 10)
{ // 构造函数
    if (initialCapacity < 1) //输出错误信息，抛出异常
    {
        ostringstream s;
        s<<"initialCapacity ="<< initialCapacity <<"Must be >0";
        throw illegalParameterValue(s.str());
    }
    arrayLength = initialCapacity;//初始化
    stack = new T[arrayLength];
    stackTop = -1;//！不是从0开始
}
```

> 得到栈顶元素:

```
template < class T >
T & arrayStack<T> :: top ()
    {
    if (stackTop == -1)//当栈为空时
```

```
            throw StackEmpty();
     return stack[ stackTop ];
     }
```

> 删除栈顶元素：

```
template<class T>
void arrayStack<T> :: pop()
{
   if (stackTop == -1)
      throw StackEmpty();
   stack [stackTop--].~T();
   // ！调用T的析构函数，而不是栈的析构函数，然后stackTop-1
}
```

> 在栈顶插入：

```
template<class T>
void arrayStack<T>:: push(const T & theElement)
 {
   //如果栈已满，则容量加倍.
   if (stackTop == arrayLength-1)
   {
        changeLength1D(stack, arrayLength, 2*arrayLength);
        arrayLength*=2;
   }
     //在栈顶插入元素
     stack[++stackTop] = theElement;
}
```

8.4 链表描述

用链表描述，如果链表右端作为栈顶，每次top、pop、push操作都需要从左往右找到最后一个节点，时间复杂度为O（size()),如果把链表左端作为栈顶，则每次top、pop、push操作的时间复杂度为Θ（1），所以我们选择链表左端作为栈顶。

8.4.1 类derivedLinkyStack

①将类derivedArrayStack中的private arrayList替换为private chain,②用名称derivedLinkyStack替代derivedArrayStack，③insert和erase的调用中索引实参改为0 。

8.4.2 类linkStack

为提高性能，我们可以定制链表栈。

> 定制链表栈：

```
template<class T>
class LinkedStack : public  stack<T>
{
    public:
        LinkedStack(int initialCapacity = 10)
            {stackTop=NULL; stackSize=0; }
        ~ LinkedStack();
        bool empty() const { return stackSize == 0; }
        int size() const { return stackSize; }
        T & top();
        void pop();
        void push(const T & theElement);
    private:
        chainNode<T>* stackTop;  //栈顶指针
        int  stackSize;     //栈中元素个数
};
```

> 析构函数：

```
template <class T>
LinkedStack<T>::~LinkedStack()
{//删除栈中所有节点
   while(stackTop!=NULL)
   {//删除栈顶节点
      chainNode <T> * nextNode = stackTop->next;
      delete stackNode;
      stackNode = nextNode;
   }
}
```

> 时间复杂度：O（sackSize）

> 得到栈顶元素：

```
template <class T>
T & LinkedStack<T> ::top()
{
    if(stackSize == 0)
      throw  stackEmpty();
    return stackTop->Element;
}
```

> 时间复杂度：Θ (1)

> 删除栈顶节点：

```
template <class T>
void LinkedStack<T> ::pop()
{
    if(stackSize == 0)
      throw  stackEmpty();
    chainNode <T> * nextNode = stackTop->next;
    delete stackTop;
    stackTop = nextNode;
    stackSize--;
}
```

> 时间复杂度：Θ (1)

> 在栈顶插入：

```
template <class T>
void LinkedStack<T> ::push(const T & theElement)
{
    stackTop = new chainNode (theElement,stackTop);
    stackSize++;
}
```

> 时间复杂度：Θ (1)

8.5应用

某些应用满足LIFO方式，可以用栈数据结构解决。

8.5.1 括号匹配

> 目的：输入一个字符串，输出匹配的括号和不匹配的括号
> 思路：从左到右扫描字符串，扫描到左括号保存到栈中，每当扫描一个右括号，就将它与栈顶的左括号（如果存在）相匹配，并将匹配到的左括号从栈顶删除。

```
void PrintMatchedPairs (string expr)
{// 括号匹配
    arrayStack<int> s;
    int length = (int) expr.size(); //也可以用strlen（expr）
    // 扫描表达式expr ，寻找( 、)
    for ( int i = 0; i<length; i++)
    {
        if ( expr.at(i) == ' ( ' )  //at是STL中的函数
            s.push(i);  //保存的是索引值
        else if (expr.at(i) ==' ) ' )
            try
            { //从栈中删除匹配的左括号，输出匹配括号的索引值
              cout << s.top() <<'   ' << i << endl;
              s.pop();
            }
            catch (stackEmpty)
            { //栈空，没有匹配的左括号
              cout<<"No match for right parenthesis"<<" at "<<i<<endl;
            }
}
    //右括号匹配结束后，栈不为空，栈中所剩下的左括号都是未匹配的
    while( !s.empty() )
   {
        cout << "No match for left parenthesis at " << s.top() < endl;
        s.pop();
   }
}
```

> 时间复杂度：O(n),n为字符串长度

8.5.2 汉诺塔

> 目的：假设有n个碟子和三座塔。初始时所有碟子从小到大堆在塔1上，我们要把碟子都移动到塔2，每次移动一个，而且任何时候都不能把大碟子压在小碟子上。

> 思路：利用递归，先把塔1上面n-1个碟子移到塔3，再把最大的碟子从塔1移到塔2，最后把n-1个碟子从塔3移到塔2。

> 第一种实现方法：输出碟子从塔1到塔2的移动次序

```
void towersOfHanoi ( int n, int x, int y, int z )
{//把n 个碟子从塔x 移动到塔y，可借助于塔z
    if (n > 0)
    {
        towersOfHanoi(n-1, x,z,y);  //把塔x上面n-1个碟子移到塔z
        cout << "Move top disk from tower " << x
            <<" to top of  tower " << y << endl;
        //把最大的碟子从塔x移到塔y
        towersOfHanoi(n-l, z, y, x);
        //把n-1个碟子从塔z移到塔y
    }
}
```

> 时间复杂度：$\Theta(2^n)$

> 第二中实现方法：要显示每次移动后三座塔的布局，所以要在内存中保存这些布局。因为碟子的移动是按照LIFO方式进行的，所以可以将每座塔表示为一个栈，因为数组栈运行速度比链表栈快，所以我们采用数组栈。

```
//全局变量，tower[1:3]表示三个塔
arrayStack<int> tower[4];
void moveAndShow ( int n, int x, int y, int z);//函数声明
void towersOfHanoi(int n)
{ // 函数moveAndShow的预处理程序
    for (int d = n; d > 0; d--) // 初始化
    tower[1].push(d); // 把碟子d从大到小放到塔1上
   //把塔1上的n个碟子移动到塔2上，借助于塔3的帮助
    moveAndShow(n, 1, 2, 3);
}
void moveAndShow(int n, int x, int y, int z)
{ // 把n 个碟子从塔x 移动到塔y，可借助于塔z
    if (n > 0)
    {
        moveAndShow(n-l, x, z, y);//把塔x上面n-1个碟子移到塔z
        int d= tower[x].top(); // 碟子编号
        tower[x].pop(); //从x中移走一个碟子
```

```
        tower[y].push(d); //把这个碟子放到y 上
        showState(); //显示塔的布局
        moveAndShow(n-1, z, y, x);
        }//把n-1个碟子从塔z移到塔y
}
```

> 时间复杂度：$\Theta(2^n)$

8.5.3 列车车厢重排

> 目的：借助缓冲轨道，将入轨道上非有序车厢，按照车厢号递增方式移到出轨道。
> 思路：从前至后检查入轨道上的车厢，如果正在检查的车厢是满足排列要求的下一节车厢，直接把它移到出轨道（此时缓冲轨道上可能也有满足条件的车厢，将其移到出轨道），否则将它移到缓冲轨道。
> 对缓冲轨道选择的基本要求是：编号为u的车厢应该直接进入的缓冲轨道其顶部车厢是大于u的最小者，即最小上限。因为缓冲轨道符合LIFO方式，所以可以数组栈表示缓冲轨道。

> 函数railroad：

```
//全局变量
arrayStack<int> * track;  //缓冲轨道数组，数组类型是数组栈
int numberOfCars;         //车箱数
int numberOfTracks;       //缓冲轨道数
int smallestCar;          //在缓冲轨道中编号最小的车厢
int itsTrack;             //停靠最小编号车厢的缓冲轨道

bool railRoad ( int inputOrder[], int theNumberOfCars, int theNumberOfTracks )
{//从初始顺序开始重排车厢

    numberOfCars = theNumberOfCars;
    numberOfTracks = theNumberOfTracks;
    //创建用于缓冲轨道的栈
    track = new arrayStack <int> [numberOfTrack+1];
    int nextCarToOutput=1;      //当前能进入出轨道的车厢
    smallestCar = numberOfCars +1;  //初始化，取一个比最大车厢号大的值

    //重排车厢
    for( int i =1; i<=numberOfCars; i++)
    {
       if( inputOrder[i] == nextCarToOutput)
       {//满足排列要求的下一节车厢，直接移到出轨道
            cout<<"Move car"<<inputOrder[i]
```

```
                << "from input track to output track"<<endl;
            nextCarToOutput++;

            while(smallestCar == nextCarToOutput)
            {//缓冲轨道上满足出轨条件的车厢，将其移到出轨道
                outputFromHoldingTrack();
                nextCarToOutput++;
            }
        }
         else
         //将车厢inputOrder[i]移到一个缓冲轨道
            if(!putInHoldingTrack(inputOrder[i]))
               return false;
    }
    return true;
}
```

> 时间复杂度：O（numberOfTracks*numberOfCars）

> 函数outputFromHoldingTrack：

```
void outputFromHoldingTrack()
{//把编号最小的车厢从缓冲轨道移到出轨道

    track[itsTrack].pop(); //删除编号最小车厢
    cout <<"Move car"<<smallestCar<<"from holding track"
         <<itsTrack <<"to output track"<<endl;

    //检查所有栈的栈顶，寻找编号最小的车厢和它所属的栈
    smallestCar = numberOfCar + 2;
    //因为不能确定编号最小的车厢编号，所以取一个较大值，方便寻找
    for( int i=1; i< = numberOfTracks; i++)
       if( !track[i].empty() && ( track[i].top() < smallestCar) )
       {
           smallestCar = track[i].top();
           itsTrack = i;
       }
}
```

> 时间复杂度：O（numberOfTracks）

函数putInHoldingTrack:

```
bool putInHoldingTrack(int c)
{//将车厢c移到一个缓冲轨道。返回false，当且仅当没有可用的缓冲轨道

    //为车厢c寻找最合适的缓冲轨道
    //初始化
    int bestTrack = 0;         //最适合的轨道
    bestTop = numberOfCars + 1;
    //因为不能确定编号最适合的轨道，所以最适合轨道的顶部车厢号取一个较大值，方便寻找

    //扫描缓冲轨道
    for (int i =1; i <=numberOfTracks; i++)
       if(!track[i].empty())
       {//缓冲轨道不为空
           int topCar = track[i].top();
           if( c<topCar && topCar < bestCar)
           {//缓冲轨道i具有编号更小的车厢，同时满足比c编号大，且又是比c编号大的车厢号
               bestCar = topCar;
               bestTrack = i;
           }
       }
       //缓冲轨道为空，必须先考虑有车厢的轨道，因为空轨道要留给比所有轨道顶部车厢号都大
        else if(bestTrack = 0) bestTrack = i;

    //没有满足条件的缓冲轨道
    if(bestTrack == 0)   return false;

    //把车厢c移到轨道bestTrack
    track[bestTrack].push(c);
    cout << "Move car " <<c<< "from input track "
         << "to holding track "<< bestTrack <<endl;

    //如果需要更新smallestCar 和itsTrack
    if( c < smallestCar )
    {
        smallestCar = c;
        itsTrack = bestTrack;
    }
    return true;
}
```

时间复杂度：O（numberOfTracks）

8.5.6 迷宫老鼠

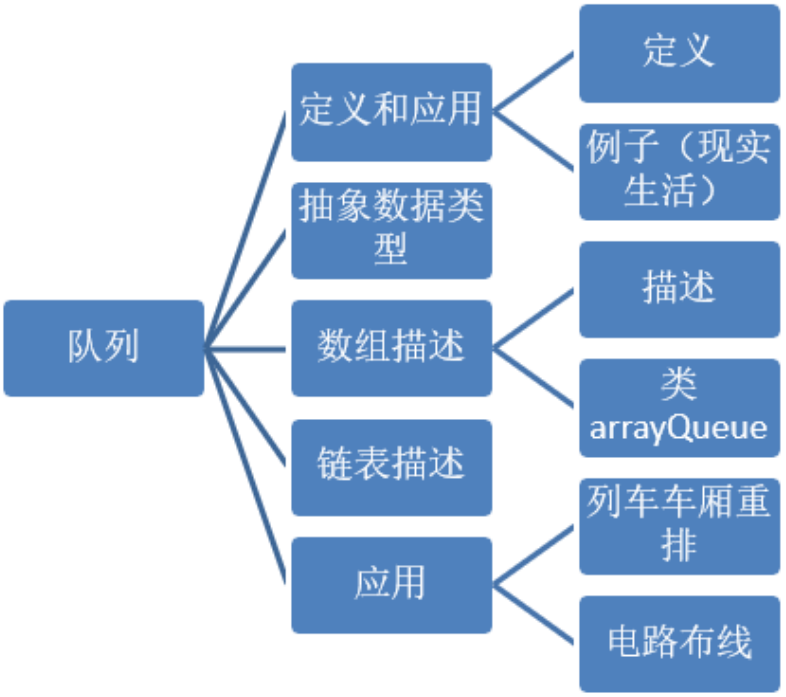
> 目的：寻找一条从入口到出口的路径。
> 思路：a.用 n×m 的矩阵来描述迷宫,在位置(i,j)处有一个障碍物时其值为1，否则其值为0。
> b.为了避免处理内部位置和边界位置时存在差别，在迷宫的周围增加一圈障碍物。
> c.如果当前位置不是迷宫出口，则在当前位置上放置障碍物，以便阻止搜索过程又绕回到这个位置。
> d.用系统的方式来确定从当前位置要向哪一个相邻位置移动。
> e.检查相邻的位置中是否有空闲的，如果有就移动到这个新的相邻位置上，然后从这个位置开始搜索通往出口的路径。如果不成功，选择另一个相邻的空闲位置，并从它开始搜索通往出口的路径。如果相邻的位置中没有空闲的，则回退到上一位置。
> f.为了方便移动，在进入新的相邻位置之前，把当前位置保存在一个堆栈中。在堆栈中始终包含从入口到当前位置的路径。
> g.如果所有相邻的空闲位置都已经被探索过，并且未能找到路径，则表明在迷宫中不存在从入口到出口的路径。

```
bool findPath()
{//寻找一条从入口（1，1）到出口（n，n）的路径
 //如果找到，返回true，否则返回false

    //动态建立栈用来保存路径，position类中有数据成员col，row
    path = new arrayStack <position>;

    //初始化偏移量
    position offset[4];
    offset[0].row = 0;  offset[0].col = 1;  //向右
    offset[1].row = 1;  offset[1].col = 0;  //向下
    offset[2].row = 0;  offset[1].col = -1; //向左
    offset[1].row = -1;  offset[1].col = 0; //向上

    //初始化迷宫外围的障碍墙
    for( int i = 0; i<= size +1; i++)
    {
        maze[0][[i]] = maze[size+1][i] = 1; //底部和顶部
        maze[i][0] = maze[i][sie+1] = 1;    //左和右
    }


    position here;  //当前位置
    here.row = 1;   //在起点
    here.col = 1;
    maze [1][1] = 1;  //在起点放上障碍物，防止再次回到起点
    int option = 0;   //下一步要往哪个方向走，0~3分别对应右下左上
    int lastOption = 3;
```

```
    //寻找一条路径
    while( here.row != size || here.col != size)
    {//还没有到达出口
        int r,c;  //临时row，col
        while(option <= lastOption)
        {//查看往哪个方向走没有障碍物
            r = here.row + offset[option].row;
            c = here.col + offset[option].col;
            if( maze[r][c] == 0 ) break;
            //有相邻位置没有障碍物，就到这个位置，跳出循环
            option++;
        }

        if( option <= lastOption)
        {//有相邻位置可走
            path->push(here);//把当前位置放入栈
            here.row = r;    //移到可行的相邻位置
            here.col = c;
            maze[r][c] = 1;  //放上障碍物，防止重复访问
            option = 0;   //！为寻找下一个位置准备
        }
        else
        {//如果没有相邻的一步可走，返回上一步
            if(path->empty())
               return false;
               //链表为空，无路可返，即不存在这样一条路径，结束循环
            position next = path->top();
            path->pop();
            //返回到上一步，要把上一步从栈中删了，直到确定它有可行的下一步再放入栈

            if(next.row == here.row)
              option = 2 + next.col - here.col;
              //上一步和当前位置行数相同，返回到上一步后可能要向下走或者向上走
            else option = 3 + next.row - here.row;
            //上一步和当前位置列数相同，返回到上一步后只能要向左走，
            //否则只能返回到上上步，如果存在上上步的话
            //因为按照移动规则，第一次进入下一步，一定是先考虑向右走。
            here = next;
        }
    }
    return true;
}
```

> 时间复杂度：O（size^2）= O（m^2）

第9章 队列



联系：

- 队列和栈一样，都是一种特殊的线性表。

区别：

- 栈的插入和删除在线性表一端进行，后进先出线性表。
- 队列的插入和删除在线性表两端进行，先进先出线性表。

> 为了执行的效率，我们把队列设计成一个基类，而且分别采用数组描述和链表描述。

9.1 定义和应用

定义：

队列是一个线性表，其插入和删除操作分别在表的不同端进行。插入元素的那一端称为队尾(back或rear)，删除元素的那一端称为队首(front)。

应用：

类似于商场购物结账、贩卖机添加货物等现实生活的例子。

9.2 抽象数据类型

队列的抽象数据类型描述 ADT9-1

```
抽象数据类型queue
{
    实例
    元素的有序表，一端称为队首，另一端称为队尾
    操作
    empty();//返回true，当且仅当队列为空，否则返回false
    size();//返回队列中元素个数
    front();//返回队列头元素
    back();//返回队列尾元素
    pop();//删除队列首元素
    push(x);//把元素x加入队尾
}
```

参看课本[^1]p207
P.S：该类型的函数名称和C++的STL容器类队列的函数名称相同

队列的抽象数据类型描述 ADT9-1

```
template<class T>
class queue
{
    public:
        virtual ~queue(){}
        virtual bool empty() const = 0; //返回true，当且仅当队列为空
        virtual int size() const = 0; //返回队列中元素个数
        virtual T& front() = 0; //返回头元素的引用
        virtual T& back() = 0; //删除尾元素的引用
        virtual void pop() = 0; //删除首元素
        virtual void push(const T& theElement) = 0; //把元素theElement加入队尾
};
```

参看课本[^2]p207

9.3 数组描述

9.3.1 描述

用环形数组来表示队列：
原因：在数组长度不变的情况下，插入和删除操作的最坏时间复杂度均为Θ(1)，并且充分利用数组空间。

公式：location(i)=(location(队列首元素)+i)%arrayLength
队首：queueFront沿逆时针方向指向队列首元素的下一个位置。
队尾：queueBack指向队列尾元素。
插入一个元素：在队尾插入，queueBack指向新插入元素。
删除一个元素：在队首删除，queueFront沿顺时针方向移动一位。
队列为空：当且仅当queueFront=queueBack时队列为空。
队列为满：因为队列满时也满足queueFront=queueBack，所以环形队列不能插满。在向一个队列插入一个元素之前，先要判断这次操作是否会让队列变满。如果是，就先把数组长度加倍，然后执行插入操作。
队列元素：最多为arrayLength-1。

9.3.2 类arrayQueue

类arrayQueue用公式location(i)=(location(队列首元素)+i)%arrayLength把队列映射到一个一维数组。数据成员是queueFront,queueBack和queue。

> 除了插入和删除操作之外，所有方法都与类arrayStack的对应方法相似，这些相似的代码可以从本书web网站得到。

程序9-2 在队列中插入一个元素

```
template<class T>
void arrayQueue<T>::push(const T& theElement)
{//把元素theElement加入队列
    //如果需要，则增加数组长度
    if((theBack + 1) % arrayLength == theFront)
    {//加倍数组长度
        //此处是数组长度加倍的代码，具体看程序9-3
    }
    //把元素theElement插入队列的尾部
    queueBack = (queueBack + 1) % arrayLength;//让queueBack指向队列下一个位置
    queue[queueBack] = theElement;//把该元素赋给queueBack
}
```

参看课本[^3]p209

详细解释：
首先判断队首的下一位是否等于队首，
如果是则说明队列已经满了，需要长度加倍。

如果不是，就让queueBack指向下一个位置。
然后将要插入的元素赋给queueBack。

程序9-3 数组队列长度加倍

```
//分配新的数组空间
T* newQueue = new T[2 * arrayLength];

//把原数组元素复制到新数组
int start = (theFront + 1) % arrayLength;
if(start < 2)
    //没有形成环
    copy(queue + start,queue + start +arrayLength - 1,newQueue);
else
{//队列形成环
    copy(queue + start,queue + arrayLength,newQueue);
    copy(queue,queue + theBack +1,newQueue + arrayLength - start);
}

//设置新队列的首和尾的元素位置
theFront = 2 * arrayLength - 1;
theBack = arrayLength - 2;//队列长度arrayLength-1
arrayLength *= 2;
delete[]queue;
queue = newQueue;
```

参看课本[^4]p211

详细解释：
建立一个新的队列，该队列长度为原队列二倍。
先判断能否成环，不能成环则直接复制。
如果队列可以成环，则先复制queueFront以后直到拉直队列末尾到新队列的开头。
再将其余元素复制到队列之后的位置。
设置新队列的首尾元素位置。
删除原有队列，将新队列赋给原有队列。

程序9-4 从队列中删除一个元素

```
void pop()
{//删除队列首元素
    if(theFront == theBack)
```

```
        throw queueEmpty()
    theFront = (theFront + 1) % arrayLength;
    queue[theFront].~T();}//给T析构
}
```

参看课本[^5]p211

详细解释：
首先判断队列是否为空。
如果为空则抛出异常。
如果不为空，则让theFront指向下一个位置，即带有元素的位置。
用析构函数删除该位置的元素。

- 队列构造函数的复杂度：
  T为基本数据类型时，复杂度为O(1)。
  T为用户定义的类型时，复杂度为O(initialCapacity)
- 队列操作函数的复杂度：
  empty、size、front、back、pop的复杂度均为Θ(1)。
- 插入函数的复杂度：
  队列容量不加倍的时候为Θ(1)。
  队列容量加倍的时候为Θ(queue size)。
  调用m次，复杂度为O(m)。

9.4 链表描述

采用从头到尾的链接方向（即theFront指向链表头，theBack指向链表尾）：

- 插入：
  从头到尾链表方向：theFront指向链表头，theBack指向链表尾。theBack插入，所以在链表尾添加一个指向新插入元素的指针。同时theBack后移。
  从尾到头链表方向：theFront指向链表尾，theBack指向链表尾。theBack插入，所以在插入元素后添加一个指针指向链表头。同时theBack前移。
- 删除：
  从头到尾链表方向：theFront指向链表头，theBack指向链表尾。theFront删除，所以在链表头删除第一个元素并且删除第一个元素指向第二个元素的指针，将theFront后移。
  从尾到头链表方向：theFront指向链表尾，theBack指向链表头。theFront删除，所以在链表尾删除最后一个元素并且删除前一个元素指向最后元素的指针。但是由于这里是单向链表，所以theFront无法一次性指向前一个元素，只能从第一个开始往后查找，比从头到尾链表方向复杂。

程序9-5 链队列中的插入和删除方法

```
template<class T>
void linkedQueue<T>::push(const T& theElement)
{//把元素theElement插到队尾
    //申请新元素节点
    chainNode<T>* newNode = new chainNode<T>(theElement,NULL);
    //把新节点插到队尾
    if(queueSize == 0)
        queueFront = newNode;//队列空
    else
        queueBack -> next = newNode;//队列不空
    queueBack = newNode;

    queueSize++;
}

template<class T>
void linkedQueue<T>::pop()
{//删除首元素
    if(queueFront == NULL)
        throw queueEmpty();

    chainNode<T>* nextNode = queueFront -> next;
    delete queueFront;
    queueFront = nextNode;
    queueSize--;
}
```

参看课本[^6]p213

详细解释:

- 插入新元素:
  申请新元素节点，将元素赋给新元素节点，下一个节点指针指向NULL。
  如果queueSize为零的话，说明队列为空，让queueFront指向新节点。
  如果queueSize不为零，就让队列最后一个元素的下一个指针指向newNode。
  将queueBack指向newNode。
  将queueSize加一。
- 删除新元素:
  如果queueFront为NULL,说明队列为空，抛出异常。
  如果queueFront不为NULL，说明队列不空，将queueFront的下一个指针指向链表

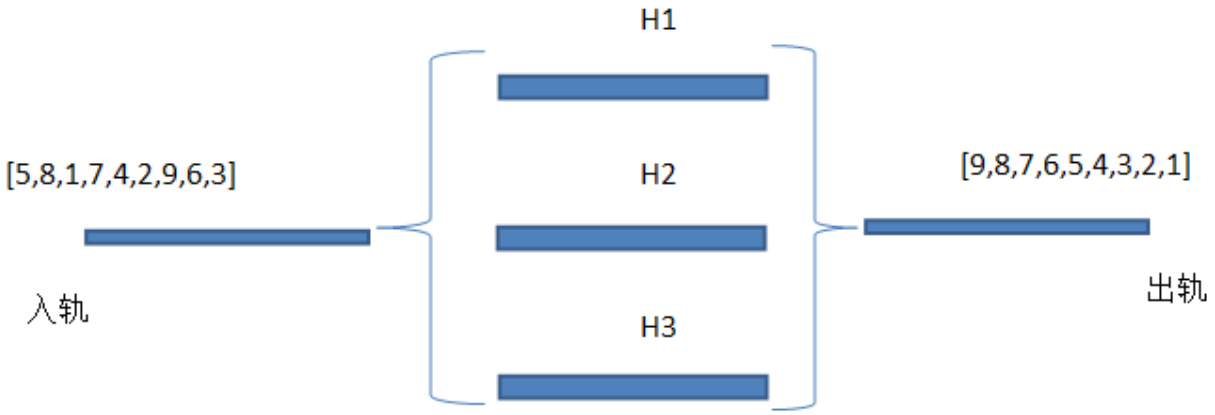
的nextNode。
删除首元素。
让queueFront指向nextNode.
将queueSize减一。

复杂度：都为Θ(1)。

9.5 应用

9.5.1 列车车厢重排

问题描述：



求解策略：
当一节车厢c进入缓冲轨道时候，依据以下原则选择缓冲轨道：

- 缓冲轨道上已有的车厢其编号均小于c；
- 如果有多个缓冲轨道都满足这一条件，则选择左端车厢编号最大的缓冲轨道；
- 否则选择一个空的缓冲轨道（如果有的话）

代码实现：

程序9-6 输出车厢

```
void outputFromHoldingTrack()
{   //将编号最小的车厢从缓冲轨道移到出轨道
    //从栈itsTrack中删除编号最小的车厢
    track[itsTrack].pop();
    cout << "Move car " << smallestCar << " from holding track " << itsTrack <<

    //检查所有栈的栈顶，寻找编号最小的车厢和它所属的栈
```

```
    smallestCar = numberOfCars + 2;
    for(int i = 1;i <= numberOfTracks;i++)
        if(!track[i].empty() && track[i].front() < smallestCar)
        {
            smallestCar = track[i].front();
            itsTrack = i;
        }
}
```

参看课本[^7]p215

整体解释：
先把确认最小编号的车厢删除，然后将该车厢编号输出，相当于移动到出轨道。
由于之前最小编号车厢已经移动到出轨道，所以重新寻找当前最小编号车厢，并且将车厢编号和所在缓冲轨道记录下来。
详细解释：
首先将需要移到出轨道的车厢编号在原有的缓冲轨道删除。（因为最小编号的车厢一定在队首，所以直接删除即可）
smallestCar是最小车厢的编号，itsTrack是现在所在的缓冲轨道。
用if语句寻遍所有缓冲轨道。
如果缓冲轨道不为空并且队首车厢编号小于现有最小编号，就把队首车厢编号赋给最小编号，并且记录下当前缓冲轨道的编号。

程序9-7 把车厢调入一个缓冲轨道

```
bool putInHoldingTrack(int c)
{//将车厢c移到一个缓冲轨道。返回false，当且仅当没有可用的缓冲轨道
    //为车厢c寻找最适合的缓冲轨道
    //初始化
    int bestTrack = 0,//目前最合适的缓冲轨道
        bestLast = 0;//取bestTrack中最后的车厢

    //扫描缓冲轨道
    for(int i = 1; i <= numberOfTracks;i++)
        if(!track[i].empty())
        {//缓冲轨道i不空
            int lastCar = track[i].back();
            if(c > lastCar && lastCar > bestLast)
            {
                //缓冲轨道i的尾部具有编号更大的车厢
                bestLast = lastCar;
                bestTrack = i;
```

```
            }
        }
        else//缓冲轨道i为空
            if(bestTrack == 0)
                bestTrack = i;

        if(bestTrack == 0)
            return false;

        //把车厢c移到轨道bestTrack
        track[bestTrack].push(c);
        cout << "Move car " << c << "from input track " << "to holding track " <

        //如果需要，更新smallestCar和itsTrack
        if(c < smallestCar)
        {
            smallestCar = c;
            itsTrack = bestTrack;
        }

        return true;
    }
}
```

参看课本[^8]p215

整体解释:
先寻遍所有缓冲轨道，确定最适合移入的缓冲轨道。然后用push移入，同时更新所需要的数据。
详细解释:
bestTrack表示目前最合适移入的缓冲轨道编号。
bestLast表示当前最合适移入缓冲轨道中最后车厢的编号。
初始化，使得这两个数值都为零。
寻遍所有缓冲轨道。
如果缓冲轨道不为空，就把该缓冲轨道最后车厢的编号赋给lastCar。如果现在要放入的车厢编号大于lastCar,说明这个缓冲轨道可以考虑。并且lastCar大于最合适缓冲轨道中最后车厢的编号，说明这个缓冲轨道比之前寻找的缓冲轨道更加合适。就把末尾车厢编号赋给bestCar, 并且将该缓冲轨道的编号赋给bestTrack。
如果缓冲轨道为空，并且最适合的缓冲轨道编号为零，说明之前没找到合适的缓冲轨道，就把该空车厢视为最适合的缓冲轨道。
如果寻遍之后最适合的缓冲轨道编号为零，说明所有缓冲轨道都不合适，返回false。
把现在的车厢推入已找到的最适合缓冲轨道中。

如果需要，更新所有车厢当前最小编号以及最小编号所在轨道。
如果当前新加入缓冲轨道的车厢小于最小编号，就把最小编号改为刚加入车厢的编号，并且记录下当前缓冲轨道。（这种情况只在车厢移到空轨道可能发生）
找到合适轨道，成功移入，返回true。

9.5.2 电路布线

问题描述：
在迷宫中找到最短路径。
把布线的区域画成一个网格，该网格用一个二维数组来存储，其中通路用0值表示，墙壁用1值表示，从该区域内的某一点向另一点布线，求布线的最短路径与路径长度。
求解策略：
把起点的值设置为2（与0和1区分开），然后把它四周走的通的点赋值为3，然后把值为3的点的四周走得通的点赋值为4，依次类推，直到走不下去或者赋值到终点，然后从终点按照数值依次减小的路线往回走即可找到最短路径。实现该操作需要队列这种数据结构，因为它有先进先出的功能。

代码实现：

程序9-8 寻找电路布线的路径

```
bool findPath()
{//寻找从始点到终点的最短路径
 //找到时，返回true，否则返回false

    if((start.row == finish.row) && (start.col == finish.col))
    {//始点等于终点
        pathLength = 0;
        return true;
    }

    //初始化变量
    position offset[4];
    offset[0].row = 0;offset[0].col = 1;//右
    offset[1].row = 1;offset[1].col = 0;//下
    offset[2].row = 0;offset[2].col = -1;//左
    offset[3].row = -1;offset[3].col = 0;//上

    //初始化网格四周的障碍物
    for(int i = 0;i <= size + 1;i++)
    {
        grid[0][i] = grid[size + 1][i] = 1;//底部和顶部
        grid[i][0] = grid[i][size + 1] = 1;//左边和右边
```

```
}

position here = start;
grid[start.row][start.col] = 2;//标记
int numOfNbrs = 4;//一个方格的相邻位置数

//对可达到的位置做标记
arrayQueue<position> q;
position nbr;
do
{//给相邻位置做标记
    for(int i = 0;i < numOfNbrs;i++)
    {//检查相邻位置
        nbr.row = here.row + offset[i].row;
        nbr.col = here.col + offset[i].col;
        if(grid[nbr.row][nbr.col]==0)
        {//对不可标记的nbr做标记
            grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
            if((nbr.row == finish.row) && (nbr.col == finish.col))
                break; //标记完成
            //把后者插入队列
            q.push(nbr);
        }
    }

//是否到达终点？
if((nbr.row == finish.row) && (nbr.col == finish.col))
    break;

//终点不可到达，可以移到nbr吗？
if(q.empty())
    return false;//路径不存在
here = q.front();//取下一个位置
q.pop();
}while(true);

//构造路径
pathLength = grid[finish.row][finish.col] - 2;
path = new position[pathLength];

//从终点回溯
here = finish;
for(int j = pathLength - 1;j >= 0;j--)
{
    path[j] = here;
    //寻找祖先的位置
    for(int i = 0;i < numOfNbrs;i++)
```

```
        {
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if(grid[nbr.row][nbr.col] == j + 2)
                break;
        }
    here = nbr;//移向祖先
    }
    return true;
}
```

参看课本[^9]p219

整体解释：
先从起点开始移动到终点，挨个对每一个相邻网格标记上应该有的数字，直到终点。则根据终点的标记可以知道最短路径长度。再从终点回溯，每次减一，得到最短路径。
详细解释：
如果起点和终点一致，说明已经到达，最短路径为零。
初始化偏移量，使得元素每次前后左右移动一格。
初始化网格四周的障碍物，全部设置为1。
将起始点标记为2.（为了区分开障碍物，所以起始点设置为从2开始）
检查相邻的四个网格。
如果相邻网格标记为零，说明没有走过，则将相邻网格的标记在现有基础上加1，并且插入队列。
如果现在点的坐标等于终点坐标，说明到达。
如果队列为空，说明没有可以到达的路径，返回false。
找到终点后，根据终点标记，减去2即为最短路径长度。
从终点回溯寻找最短路径。

第 10 章 跳表和散列

本章概述

1.引入跳表的必要性

之前我们学过在n个元素的有序数组上的折半查找所需要的时间是O（logn），但是在有序链表上查询的时间为O（n）。为了提高有序链表的查找性能，可以在全部或部分节点上增加额外的指针。这种增加了额外向前指针的链表叫做**跳表**。

2.散列

散列是用来查找、插入、删除的另一种随机方法。与跳表相比它把操作时间提高到Θ（1），但最坏情况下的时间仍是Θ（n）。

3.应用

本章有一个关于散列的重要应用—利用**LZW方法**实现文本的压缩和解压缩。

10.1字典

字典的定义：
字典是由一些形如 （k，v）的数对所组成的集合，其中k是**关键字**，v是与关键字k对应的**值**。任意两个 数对，其关键值都不相等。

10.3线性表描述

字典的线性表描述：
关键字从左到右依次增大，搜索的时间复杂度为 O（logn），插入操作的 时间复杂度为 O（n） （查找+搬移（O（n））），删除操作的时间复杂度为 O（n） （查 找+删除（O（1））+搬移）。以下的三种方法的实现实现代码都是以链表为基础实现的。

搜索的实现代码如下：

```
template<class K, class E>
pair<const K,E>* sortedChain<K,E>::find(const K& theKey) const
{
   pairNode<K,E>* currentNode = firstNode;

   // 寻找关键字是theKey的数对
   while (currentNode != NULL &&
          currentNode->element.first != theKey)
      currentNode = currentNode->next;
```

```
    // 判断是否匹配
    if (currentNode != NULL && currentNode->element.first == theKey)

        return &currentNode->element;

    // 当无匹配的数对时返回为空
    return NULL;
}
```

插入的实现代码如下：

```
template<class K, class E>
void sortedChain<K,E>::insert(const pair<const K, E>& thePair)
{
    pairNode<K,E> *p = firstNode,
                  *tp = NULL; // tp trails p

    // 移动指针tp，使thePair可以插在tp的后面
    while (p != NULL && p->element.first < thePair.first)
    {
        tp = p;
        p = p->next;
    }

    // 检查是否有匹配数对
    if (p != NULL && p->element.first == thePair.first)
    {// 替换旧值
        p->element.second = thePair.second;
        return;
    }

    // 没有匹配的数对，为thePair创建新节点
    pairNode<K,E> *newNode = new pairNode<K,E>(thePair, p);

    // 在tp之后插入新节点
    if (tp == NULL) firstNode = newNode;
    else tp->next = newNode;

    dSize++;
    return;
}
```

删除的实现代码如下：

```
template<class K, class E>
void sortedChain<K,E>::erase(const K& theKey)
{
   pairNode<K,E> *p = firstNode,
                 *tp = NULL; // tp trails p

   // 寻找关键字为theKey的数对
   while (p != NULL && p->element.first < theKey)
   {
      tp = p;
      p = p->next;
   }

   // 确定是否匹配
   if (p != NULL && p->element.first == theKey)
   {
      // 从链表中删除p
      if (tp == NULL) firstNode = p->next; //如果p是头结点
      else tp->next = p->next;

      delete p;
      dSize--;
   }
}
```

10.4跳表（这一部分比较难，可选）

（1）原理：

由于链表描述的特性，我们没有办法像公式化描述一样二分搜索，但是如果在链表中添加 若干的指针（例如增加一个指向中间元素的指针），我们就可以利用二分搜索的思想减少 比较的次数。将时间复杂度缩小到 logn 的级别。

（2）i级链表：

对n个数对而言，0级链表包含所有数对，1级链表每2个数对取一个，2级链表每四个数对取一个，则i级链表每2^i个数对取一个。一个数对属于**i级链表**，当且仅当它属于0~i级链表，但不属于i+1级链表（元素在**最高**可以在 i 级链上，它属于i级链表）。

（3）maxLevel：

由于跳表级的分配是随机产生的，这种方法有一个潜在的缺点—就是某些数对被分配的基数可能特别大，远远超过了$\log_{1/p}N$，其中N为字典数对的最大预期数目。为避免这种情况，设定一个级数的上限maxLevel，最大值为：**$\log_{1/p}N-1$；**

（4）用skipList描述跳表的复杂度：

当字典有n个数对时，查找、插入、删除操作的时间复杂度均为*O（n+maxLevel）*。尽管在最坏的情况下跳表的性能较差，但是跳表查找、插入、删除的**平均复杂度**的均为*O（logn）*。

至于空间复杂度，在最坏的情况下每个记录都可能是maxLevel级，都需要maxLevel+1个指针的空间。但是和时间复杂度相同的是在平均的情况下，指针域是$n\sum_i p^i=n/（1-p）$。（**p：**在规则的链表中i-1级链表的数对个数与i级链表的数对个数之比是一个分数p。）

10.5散列：

10.5.1 理想散列：

字典的另一种表示方法是**散列**。它用一个散列函数（也称哈希函数）把字典的数对映射到一个**散列表**的具体位置。如果数对p的关键字是k，散列函数是*f*，那么在理想的情况下，p在散列表中的位置为*f（k）*。在理想情况下，查找、插入、删除操作的时间均为Θ（1）。

10.5.2 散列函数和散列表：

1.桶和起始桶：散列表的每一个位置叫一个**桶**，对关键字为k的数对，*f（k）*是**起始桶**，桶的数量等于散列表的大小。

2.除法散列函数：在多种散列函数中，最常见的是除法散列函数，形式如下：*f（k）*=k%D;其中k是关键字，D是散列表的大小。

3.冲突和溢出：当两个不同关键字所对应的起始桶相同时，就是**冲突**发生了。如果存储桶没有空间储存一个新的数对，就是**溢出**发生了。

4.选择散列函数的除数D：理想的D是一个素数，并且又不能被小于20的数整除。

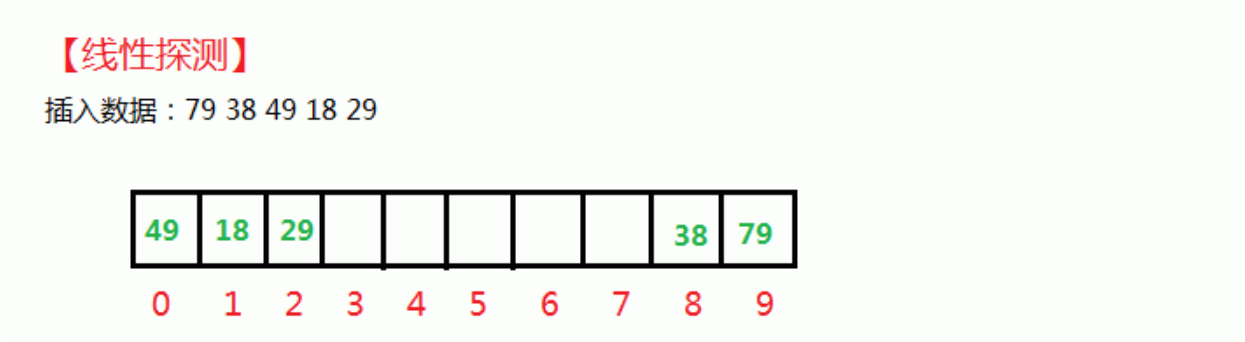
10.5.3线性探查：

当每个桶只能储存一个数对时，冲突和溢出就同时发生了，那么应当将数据放到哪呢？最常见的解决方法就是线性探查法。

HashTable的数据成员和构造函数

```
//散列表数据成员
      pair<const K, E>** table;  // 散列表
```

```
    hash<K> hash;               // 把类型k映射到一个非整数
    int dSize;                  // 字典中数对个数
    int divisor;                // 散列函数除数
//构造函数
template<class K, class E>
hashTable<K,E>::hashTable(int theDivisor)
{
   divisor = theDivisor;
   dSize = 0;

   // 分配和初始化散列表数组
   table = new pair<const K, E>* [divisor];
   for (int i = 0; i < divisor; i++)
      table[i] = NULL;
}
```



插入：图中D是10，由于每个桶只能装一个数据，插入49时就发生了冲突，这时就要往下探查找到下一个可用的桶。

HashTable<K,E>::insert

```
template<class K, class E>
void hashTable<K,E>::insert(const pair<const K, E>& thePair)
{
   //把数对thePair插入字典，若存在相同的关键字的数对，则覆盖
   // 搜索散列表，查找匹配的数对
   int b = search(thePair.first);

   // 查找匹配的数对是否存在
   if (table[b] == NULL)
   {
      table[b] = new pair<const K,E> (thePair);
      dSize++;
   }
   else
   {
      if (table[b]->first == thePair.first)
```

```
        {
            table[b]->second = thePair.second;
        }
        else // 表满
            throw hashTableFull();
    }
}
```

查找：假设要查找关键字为k的数对，首先搜索起始桶*f（k）*，然后将散列当作环表继续搜索下一个桶，直到有以下情况之一发生1）找到对应元素，2）找到一个空桶，3）再次找到 f(k)桶。对于后两种情况 表示没有关键字为 k 的元素。

HashTable<K,E>::search

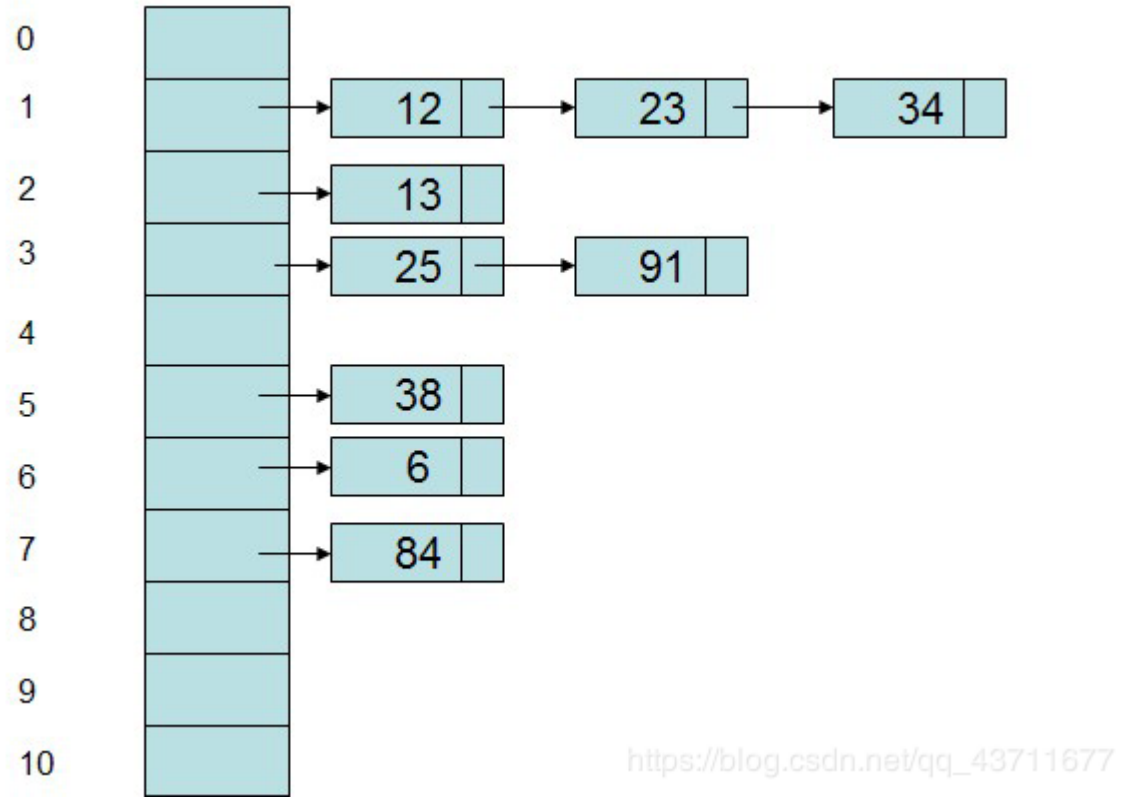
```
template<class K, class E>
int hashTable<K,E>::search(const K& theKey) const
{
   int i = (int) hash(theKey) % divisor;  // 起始桶
   int j = i;     // 从起始桶开始
   do
   {
      if (table[j] == NULL || table[j]->first == theKey)
         return j;
      j = (j + 1) % divisor;  // 下一个桶
   } while (j != i);          // 能回到起始桶?

   return j;  // 表满
}
```

删除：删除的时候可能会带来某些元素无法被查询到，这个时候为了保持散列的特性，我们需要适当地移动元素来维持。也可以不移动元素而是采用布尔值标记桶的状态 。——详见P251.

10.5.4 链式散列：

如果散列表的每一个桶都可以容纳无限个记录，那么溢出问题就不存在了。实现这个目标的方法之一就是给散列表的每个位置配置一个有序链表。

如图是一个除数为11的散列表：

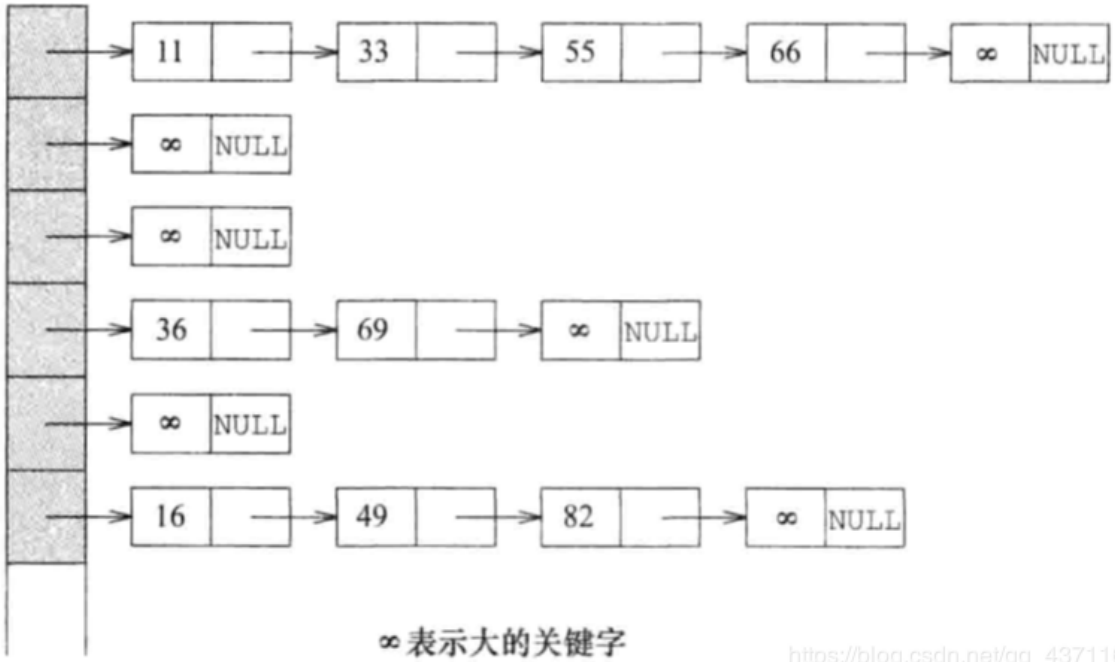


如何插入：通过哈希函数找到 f(k)个桶，然后将元素添加到链表中。由于每次插入都 要进行一遍查询，所以有序的链表会更加有效。
如何删除：和链表的操作类似。只不过是在第 f(k)个桶的有序链表上进行操作。

实现的代码如下：

```
pair<const K, E>* find(const K& theKey) const
    {return table[hash(theKey) % divisor].find(theKey);}

 void insert(const pair<const K, E>& thePair)
 {
    int homeBucket = (int) hash(thePair.first) % divisor;
    int homeSize = table[homeBucket].size();
    table[homeBucket].insert(thePair);
    if (table[homeBucket].size() > homeSize)
       dSize++;
 }

 void erase(const K& theKey)
    {table[hash(theKey) % divisor].erase(theKey);}
```

优化：在每个链表的末尾增加一个无穷大的标记（也可以所有的链表共用一个无穷大 标记。）这样相当于一个哨兵，减少了每次循环的判断次数。
实现如图：



与线性探查和跳表的比较，这一部分较难了解即可，具体的请看课本P256-P257。

10.6 一个应用——文本压缩

为减少一个文本文件的磁盘空间，常常需要把给文本文件压缩编码后储存。本节采用的是Lempel、Ziv和Welch所开发的而技术，设计对文本文件进行压缩和解压缩的C++代码。这种技术称为LZW方法。

10.6.1 LZW压缩：

LZW 压缩方法把文本的字符串映射为数字编码。首先，为该文本文件中所有可能出现的字母分别分配一个代码。例如，要压缩的文本文件为：aaabbbbbbaabaaba。

字符串由 a 和 b 组成。为 a 分配代码 0，为 b 分配代码 1。字符串和编码的映射关系储存在一个数对字典中。LZW 压缩器不断地在输 入文件中寻找在字典中出现的最长的前缀 p，并输出其相应的代码。

编码过程可以总结为：①找出余下字符串中在自字典内出现的最长前缀（设为 A），输出字符串对应的编码②为字符串（A+下一位）分配编码，写入字典。 解码过程可以总结为：初始化字典后①在字典中读取 key②根据法则创建新的 key 和 value③index 偏移，继续进行解码。

最后得到的字典数对如下图：

code	0	1	2	3	4	5	6	7
key	a	b	aa	aab	bb	bbb	bbba	aaba

10.6.2 LZW压缩的实现

代码实现——P261-P264.

10.6.3 LZW解压缩：

解压缩时，每次输入一个代码，然后把代码替换为相应的文本。可以分为两种情况1）代码p在字典中.2）代码p不在字典中。两种处理方式并不相同。

1.代码p在字典中的情形

当p在字典中时，找到与p对应的文本text(p)输出。由压缩器的原理可知，代码q出现在p之前，且text(q)时q对应的文本，则压缩器会为文本text(q)和文本text(p)的第一个字符fc(p)所连接构成的文本text(q)fc(p)分配一个新的代码。**于是在字典中插入数对（下一个代码，text(q)fc(p)）。**

2.代码p不在字典中的情形

只有在文本段中的形式为text(q)text(q)fc(q)和text(p)=text(q)fc(q),且相应的压缩文件段为qp（其中q是文本text（q）的代码，p是文本text（p）的代码）的时候，在qp的解压缩过程中，当q被text（q）代替之后，代码p在字典中不会有对应的文本，而这个文本应该是text（q）fc（q），其中q是在p前的代码。

10.6.4LZW解压缩的实现

代码实现——P265-P268.

第11章 二叉树和其他树

本章概述

本章我们将介绍数据结构世界里的“树”。
主要涵盖以下内容：

- 树和二叉树的术语，如根、叶子、孩子、父母、兄弟、级、高度、深度和度。
- 二叉树的数组及链表表示。
- 4种常用的二叉树遍历方法：前序遍历、中序遍历、后序遍历和层次遍历。

本章的应用部分给出了树的两个应用。第一个应用是关于在一个树形分布的网络中设置**信号调节器**。第二个应用是在线等价类问题。在线等价类问题在本章中又被称为**合并搜索问题**。利用树来解决等价类问题要比链表解决方案高效得多。

11.1 树

树的定义：
树(tree)是一个**非空的**有限元素的集合，其中一个元素为**根**(root)，余下的元素（如果有的话）组成t的**子树**(subtree)。

根：
在树这种层次数据中最高层的元素是**根**(root)。

孩子：
一个结点含有的子树的根结点称为该结点的**孩子**(children)结点。

父母：
若一个结点含有子结点，则这个结点称为其子结点的**父母**(parent)。树根是树中唯一一个没有父母的元素。

叶子：
在树中没有孩子的节点称为**叶子**(leaf)。

兄弟：
具有相同父结点的结点互称为**兄弟**(sibling)结点。

级：
树根是1**级**(level)，其孩子(如果有)为2级，孩子的孩子是3级，等等。

> 部分资料中树的级号从0开始，这时树的级是0级。

高度、深度：
一棵树的**高度**(height)或**深度**(depth)是树中级的个数。

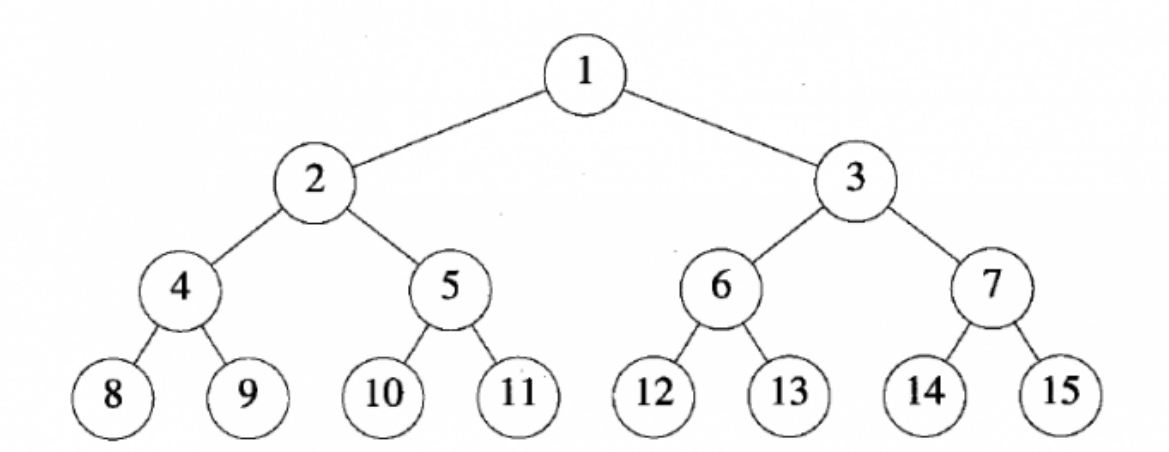
度：
一个**元素的度**(degree of an element)是指其孩子的个数。叶节点的度为0。**一颗树的度**(degree of an element)是其元素的度的最大值。

11.2 二叉树

二叉树的定义：
二叉树(binary tree)是有限个元素的集合(可以为空)。当二叉树非空时其中有一个称为根的元素，余下的元素(如果有的话)被组成2个二叉树，分别称为左子树和右子树。

> 二叉树和树的根本区别:
>
> - 二叉树可以为空，但树不能为空。
> - 二叉树中每个元素都恰好有两棵子树(其中一个或两个可能为空)。而树中每个元素可有若干子树。
> - 在二叉树中每个元素的子树都是有序的，可以用左、右子树来区别。而树的子树间是无序的。

与树相似地，二叉树也是根节点在顶部。二叉树左（右）子树中的元素画在根的左（右）下方，在每个元素和其子节点间有一条边。

> 下面以一棵二叉树为例，说明前两节的相关概念：
>
> 
>
>
> - “1”为根
> - 以“2”和“3”为根的两棵树是“1”的子树
> - “4” “5” 是“2”的子节点(孩子)
> - “6”是“12”、“13”的父节点

> - 8~15没有孩子，成为“叶子”节点
> - “2”与“3”为“兄弟”关系，同理“12”与“13”也是
> - 这棵树为4“级”
> - 这棵树“高度”（或称“深度”）为4
> - 图中除叶节点外，其余节点的“度”均为2（这是由于我们举得例子特殊，为“二叉树”）

11.3 二叉树的特性

特性1：包含$n(n>0)$个元素的二叉树边数为$n-1$。
证明:二叉树中每个元素(除了根节点)有且只有一个父节点。在子节点与父节点间有且只有一条边，因此边数为$n-1$。

特性2：若二叉树的高度为$h(h\geq 0)$，则该二叉树最少有h个元素，最多有2^h-1个元素。
证明：因为每一层最少要有1个元素，因此元素数最少为h。每元素最多有2个子节点，则第i层节点元素最多为2^{i-1}个$(i>0)$。$h=0$时，元素的总数为0，也就是2^0-1。当$h>0$时，元素的总数不会超过$\sum_{i=1}^{h}2^{i-1}=2^h-1$。

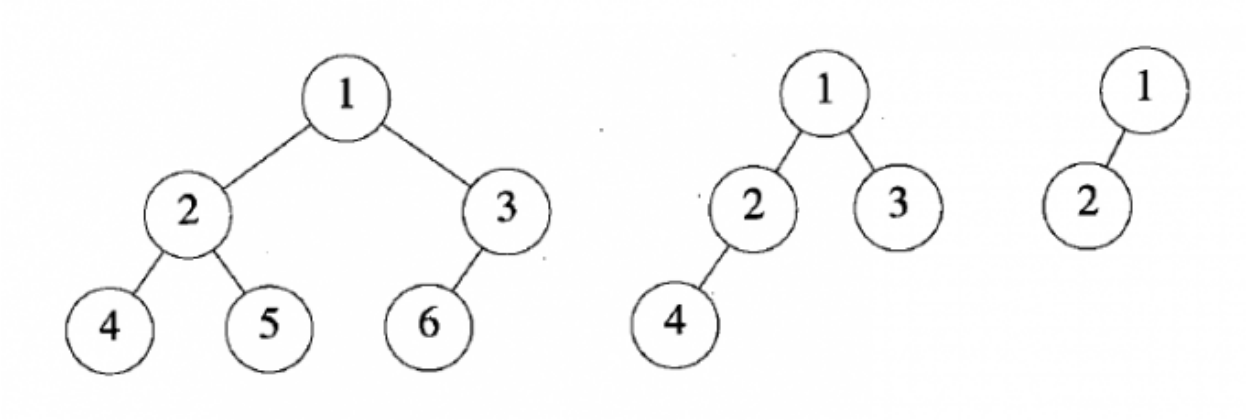
特性3：包含$n(n>0)$个元素的二叉树的高度最大为n，最小为$\lceil \log_2(n+1)\rceil$。
证明：因为每层至少有一个元素，因此高度不会超过n。由特性2可以得知高度为h的二叉树最多有2^h-1个元素。因为$n\leq 2^h-1$，因此$h\geq \log_2(n+1)$。由于h是整数，所以$h\geq \lceil \log_2(n+1)\rceil$。

特性4：设完全二叉树的一元素其编号为$i,1\leq i\leq n$。有以下关系成立：
1)当$i=1$时，该元素为二叉树的根。若$i>1$，则该元素父节点的编号为 $\lfloor \frac{i}{2}\rfloor$。
2)当$2i>n$时，该元素无左孩子。否则，其左孩子的编号为 $2i$。
3)若$2i+1>n$，该元素无右孩子。否则，其右孩子编号为$2i+1$。

满二叉树:
当高度为h的二叉树恰好有2^h-1个元素时，称其为**满二叉树**(full binary tree)

完全二叉树:
假设对高度为 h的满二叉树中的元素按从第上到下，从左到右的顺序从 1到$2h-1$进行编号。假设从满二叉树中删除k个编号为2^h-i的元素，$1\leq i\leq k\leq 2^h$，所得到的二叉树

被称为**完全二叉树**(complete binary tree)。如下图三个完全二叉树举例。

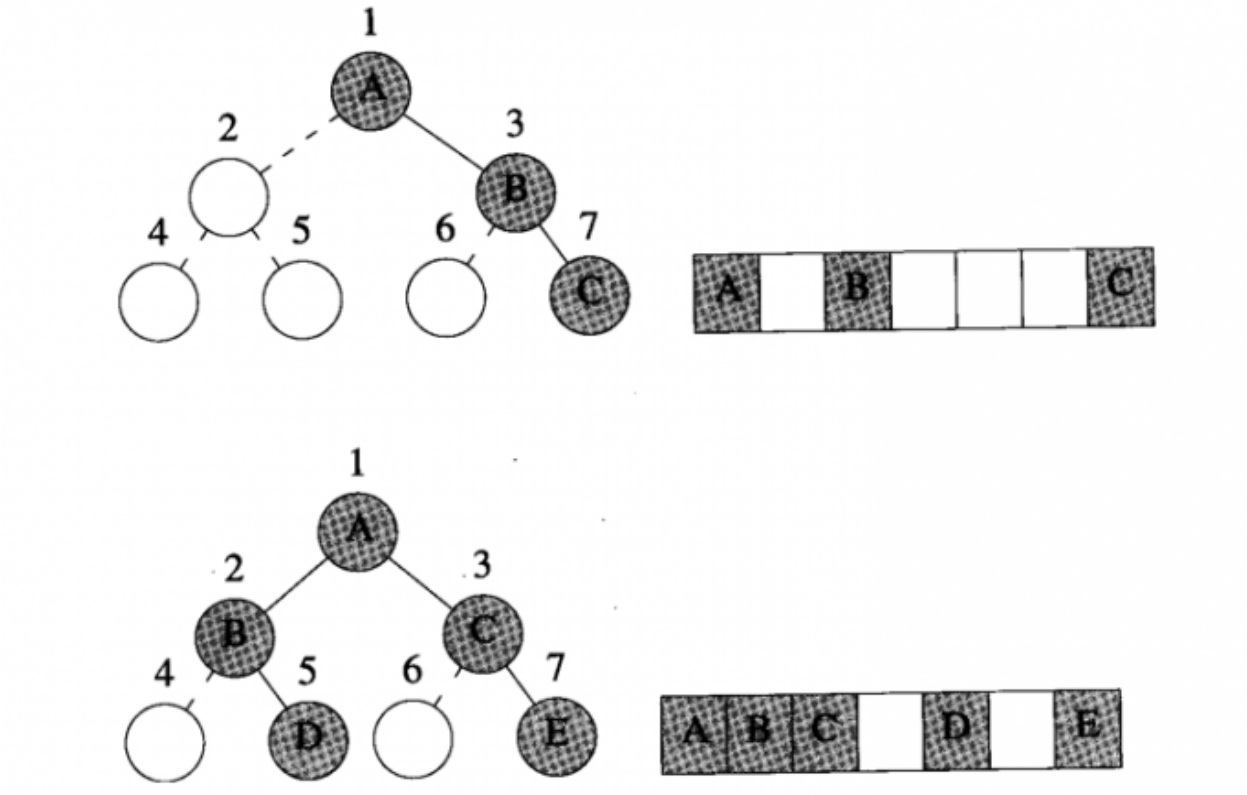


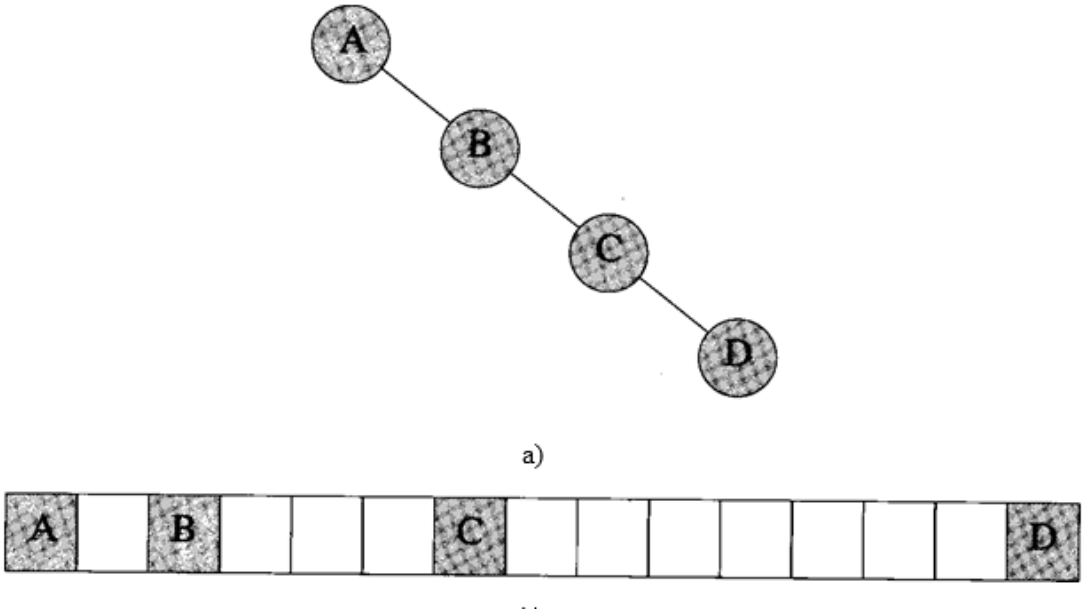
> 满二叉树是完全二叉树的一个特例。有n个元素的完全二叉树，其高度为$\lceil \log_2 (n+1) \rceil$

11.4 二叉树的描述

11.4.1 数组描述

二叉树的数组描述利用了特性4。二叉树可以作为缺少了部分元素的完全二叉树。在示意图中，没有涂阴影的圈表示缺少的元素。所有的元素(包括缺少的元素)按前面介绍的方法编号。然后将二叉树的元素存储在数组中。**缺少的元素**由白圈和方格描述。



> 当缺少很多元素时，这种描述方法非常浪费空间。当每个节点都是其他节点的右孩子时，存储空间达到最大。如下图：
>
> 
>
>
> 这种类型的二叉树称为**右斜二叉树**(right-skewed binary tree)。

> 当缺少的元素数目比较少时， 这种描述方法很有效。

11.4.2 链表描述

二叉树的每个元素用一个节点表示。节点包含两个指针域：leftChild和rightChild，另外还含有一个element域。

链表二叉树节点的结构体实现：

```
template <class T>
struct binaryTreeNode
{
    T element;
    binaryTreeNode<T>* leftChild;
    binaryTreeNode<T>* rightChild;
    binaryTreeNode() {
        leftChild = rightChild = NULL;

    }
    binaryTreeNode(const T& theElement) {
        element(theElement);
        leftChild = rightChild = NULL;
    }
```

```
    binaryTreeNode(const T& theElement, binaryTreeNode* theLeftChild, binaryTreeNo
        element(theElement);
        leftChild = theLeftChild;
        rightChild = theRightChild;
    }
};
```

链表二叉树的节点类：

```
template <class T>
class BinaryTreeNode {
    friend void  Visit(BinaryTreeNode<T>*);
    friend void InOrder(BinaryTreeNode<T>*);
    friend void PreOrder(BinaryTreeNode<T>*);
    friend void PostOrder(BinaryTreeNode<T>*);
    friend void LevelOrder(BinaryTreeNode<T>*);
    friend void main(void);
    public:
        BinaryTreeNode() {
            LeftChild = RightChild = 0;
        }
        BinaryTreeNode(const T& e)
        {
            data = e;
            LeftChild = RightChild = 0;
        }
        BinaryTreeNode(const T& e, BinaryTreeNode* l,BinaryTreeNode* r)
        {
            data = e;
            LeftChild = l;
            RightChild = r;
        }
    private:
        T data;//
        BinaryTreeNode<T>* LeftChild, /*左子树*/ * RightChild; // 右子树
};
```

11.5 二叉树的操作

二叉树的常用操作有：

- 得到二叉树的高度
- 得到二叉树的元素数目
- 复制一棵二叉树
- 打印一棵二叉树

- 比较两棵二叉树是否一样
- 删除二叉树

这些操作的核心是怎么遍历二叉树。在二叉树的**遍历**(traversal)中，每个元素遍历一次，在遍历时，可对正在进行遍历的节点进行打印元素等操作。在下一节中我们将进一步讨论遍历算法。

求二叉树的高度：

> 从二叉树深度的定义可知，二叉树的高度应为其左、右子树高度的**较（最）大值**加1。由此，需先分别求得左、右子树的高度，算法中“访问结点”的操作为:求得左、右子树高度的最大值，然后加1。

```
int Depth (BinaryTreeNode<T> *t ){
    // 返回二叉树的高度
    int depthval=0;
   if (!t) depthval = 0;
   else {
        //递归求左子树的高度
        int depthLeft = Depth( t->lchild );
        //求右子树的高度
        int depthRight= Depth( t->rchild );
        if (depthLeft>depthRight){
            depthval = 1+depthLeft;
        }
        else depthval = 1+depthRight;
   }
   return depthval;
}
```

求二叉树的元素个数：
与求二叉树的深度的过程类似，二叉树的元素个数应为左、右子树元素个数的和加“1”（正访问的节点本身）。

```
int Count (BinaryTreeNode <T> *t){
//返回指针T所指二叉树中所有结点个数
    if (!t) return 0;
    if (!t->lchild && !t->rchild) return 1;
    else{
        //计算左子树的节点数
        int m = Count ( t->lchild);
        //计算右子树
        int n = Count ( t->rchild);
        return (m+n+1);   //返回加和并加1
```

```
    }
}
```

复制二叉树举例：

```
template <class T>
BinaryTreeNode<T>* PreCopy(BinaryTreeNode<T> *t)
{
 //前序复制二叉树
    if (!t)  return 0;
    BinaryTreeNode<T> *root;
    //复制根节点的
    root = new BinaryTreeNode<T> (t->data);
    //复制左子树
    root->LeftChild = PreCopy(t->LeftChild);
    //复制右子树
    root->RightChild = PreCopy(t->RightChild);
    return root;  //返回
}
```

删除二叉树：

```
template <class T>
void Erase(BinaryTreeNode<T> * t)
{   //使用后序遍历
   if (t) {
      Erase(t->LeftChild);   //删除左子树
      Erase(t->RightChild);  //删除右子树
      delete t;              //删除
      t = 0;
    }
}
```

11.6 二叉树的遍历

对于二叉树的遍历有四种常用的方法：

- 前序遍历
- 中序遍历
- 后序遍历
- 层次遍历

前序遍历的代码实现：

```
template <class T>
void PreOrder(BinaryTreeNode<T> *t)
{  // 对*t进行前序遍历
    if (t) {
        Visit(t);   // 访问根节点
        PreOrder(t->LeftChild); // 前序遍历左子树
        PreOrder(t->RightChild);// 前序遍历右子树
    }
}
```

> 易发现，该遍历方法使用了“递归”。

中序遍历的代码实现：

```
template <class T>
void InOrder(BinaryTreeNode<T> *t)
{  // 对*t进行前序遍历
    if (t) {
        PreOrder(t->LeftChild); // 前序遍历左子树
        Visit(t);   // 访问根节点
        PreOrder(t->RightChild);// 前序遍历右子树
    }
}
```

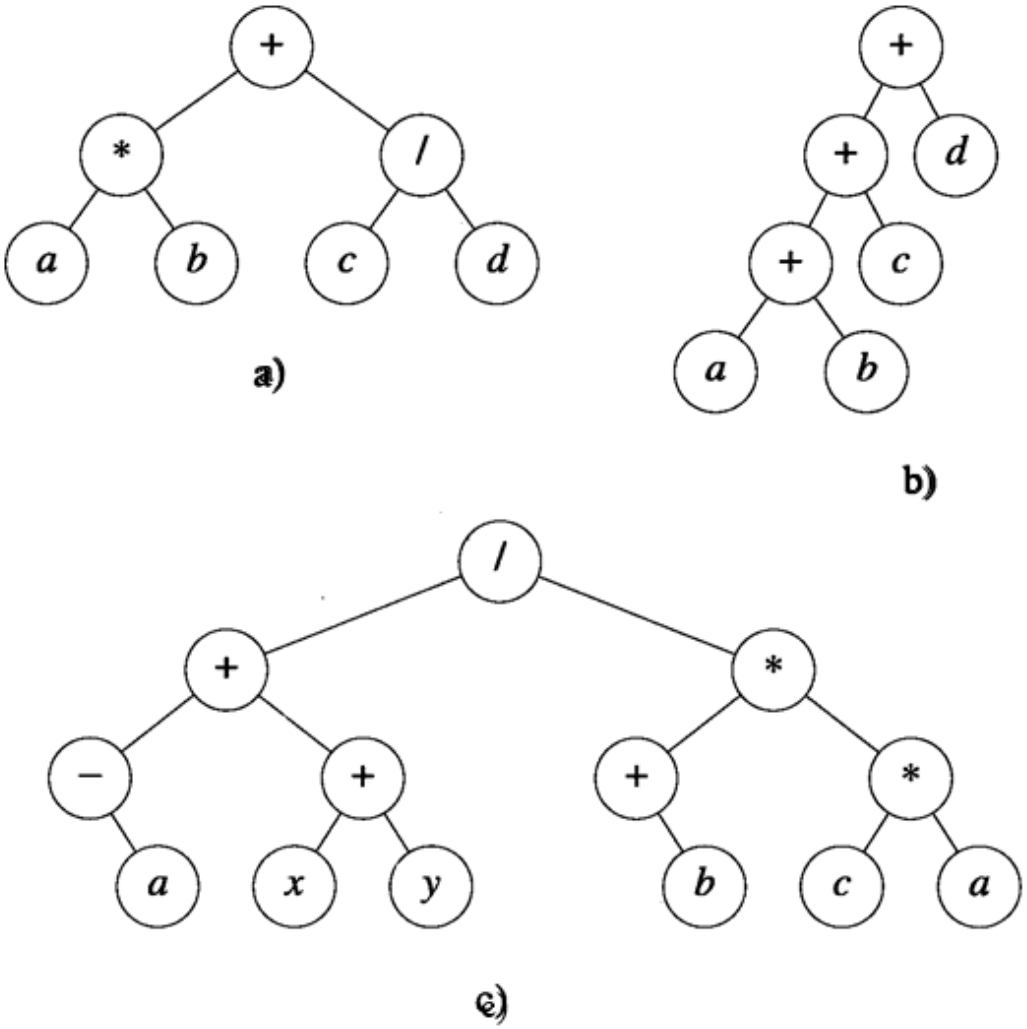
后序遍历的代码实现：

```
template <class T>
void PostOrder(BinaryTreeNode<T> *t)
{  // 对*t进行前序遍历
    if (t) {
        PreOrder(t->LeftChild); // 前序遍历左子树
        PreOrder(t->RightChild);// 前序遍历右子树
        Visit(t);   // 访问根节点
    }
}
```

一个visit函数的例子：

```
template <class T>
void Visit(binaryTreeNode<T>* x){
    //访问x的element域
    cout<<x->element<<' ';
}
```

对数学表达式树的前序、中序、后序遍历的结果分别成为数学表达式的前缀、中缀、后缀形式。下图举例数学表达式树：



其前序、中序、后序遍历的结果如下：

	a)	b)	c)
前序	+*ab/cd	+++abcd	/+-a+xy*+b*ca
中序	a*b+c/d	a+b+c+d	−a+x+y/+b*c*a
后序	ab*cd/+	ab+c+d+	a−xy++b+ca**/

中缀(infix)形式是我们日常使用的书写形式。但是中缀形式可能会产生一些歧义。例如$x+y*z$究竟是“$(x+y)*z$”呢，还是“$x+(y*z)$”呢?

为了避免歧义，我们可以对操作符规定优先级，“乘、除的优先级高于加、减”等，然后用优先级规则分析中缀表达式。如果有需要，还可以通过使用“括号”来起到规范优先级的作用。

在**完全括号化**的中缀表达式中，每个操作符和相应的操作数都用一对括号括起来。如$((x)+(y))$，$((x)+((y)*(z)))$和$(((x)+(y))*((y)+(z)))*(w))$。

输出完全括号化的中缀表达式：

```
template <class T>
void Infix(BinaryTreeNode<T> *t){
    // 输出表达式的中缀形式
    if (t) {
        cout << '(';
        Infix(t->LeftChild);    // 左操作数
        cout << t->data;    // 操作符
        Infix(t->RightChild);   // 右操作数
        cout << ')';
    }
}
```

在后缀(postfix)表达式中，每个操作符跟在操作数之后，操作数按从左到右的顺序出现。在前缀(prefix)表达式中，操作符位于操作数之前。

> 在前缀和后缀表达式中不会存在歧义。

层次遍历的代码实现：

```
template<class T>
void levelOrder(binaryTreeNode<T> *t){
    //'层次遍历二叉树*t
    arrayQueue<binaryTreeNode<T>*> q;
    while(t!=NULL){
        //树非空时进入循环
        visit(t);   //首先访问根节点
        //如果有的话，将t的孩子插入队列
        if(t->leftChild!=NULL){
            q.push(t->leftChild);
        }
        if(t->rightChild!=NULL){
            q.push(t->rightChild);
        }
        //提取下一个要访问的节点
        try{    //访问队列首元素
            t=q.front();
            //不为空，则front()返回值指向队首元素指针
        }
        catch(queueEmpty){
            return;
        }
        q.pop();
```

```
    }
}
```

> 这四种遍历算法的空间和时间复杂性均为$O(n)$(设二叉树中元素数目为 n)。

> 若二叉树中各节点的值均不相同，则由二叉树的**前序序列和中序序列**，或由其**后序序列和中序序列**均能唯一地确定一棵二叉树，但由**前序序列和后序序列**却**不一定**能唯一地确定一棵二叉树。

11.7 抽象数据类型 BinaryTree

```
抽象数据类型BinaryTree
{
实例
    元素集合；
    如果不空，则被划分为根节点、左子树和右子树；
    每个子树仍是一个二叉树
操作
    Create ()：创建一个空的二叉树；
    IsEmpty：如果二叉树为空，则返回true ，否则返回false
    Root(x)：取x为根节点；如果操作失败，则返回 false，否则返回true
    MakeTree(root，left，right)：创建一个二叉树，root作为根节点，left作为左子树，right
    BreakTree(root，left，right)：拆分二叉树
    PreOrder：前序遍历
    InOrder：中序遍历
    PostOrder：后序遍历
    LevelOrder：逐层遍历
}
```

二叉树的抽象类：

```
template<class T>
class binaryTree{
    public :
        virtual ~binaryTree(){}
        virtual bool empty()const=0;
        virtual int size()const=0;
        virtual void preOrder(void (*)(T*))=0;
        virtual void inOrder(void (*)(T*))=0;
        virtual void postOrder(void (*)(T*))=0;
        virtual void levelOrder(void (*)(T*))=0;
};
```

> 遍历函数的参数类型 void (*)(T *)是一种函数类型，这种函数类型的返回值是void，它的参数类型是 T *。

二叉树抽象数据类型的C++定义:

```
template<class T>
class BinaryTree {
    public:
        BinaryTree() { root = 0; };
        ~BinaryTree() {};
        bool IsEmpty()const { return((root) ? false : true); }
        bool Root(T& x) const;
        void MakeTree(const T& element, BinaryTree<T>& left, BinaryTree<T>& right)
        void BreakTree(T& element, BinaryTree<T>& left , BinaryTree<T>& right);
        void PreOrder(void(*Visit) (BinaryTreeNode<T>* u)){
        PreOrder(Visit, root);
        }
        void Inorder(void (*Visit) (BinaryTreeNode<T>* u)){
        Inorder(Visit, root);
        }
        void PostOrder(void(*Visit) (BinaryTreeNode<T>* u))
        {Postorder(Visit, root);
        }
        void LevelOrder(void(*Visit) (BinaryTreeNode<T>* u));
    private:
        BinaryTreeNode<T>* root; // 根节点指针
        void PreOrder(void (*Visit) (BinaryTreeNode<T>* u), BinaryTreeNode<T>* t);
        void InOrder(void(*Visit) (BinaryTreeNode<T>* u), BinaryTreeNode<T>* t);
        void PostOrder(void(*Visit) (BinaryTreeNode<T>* u), BinaryTreeNode<T>* t);
};
```

共享成员函数的实现:

```
template<class T>
bool BinaryTree<T>::Root(T& x) const
{
// 取根节点的data域，放入x
// 如果没有根节点，则返回false
    if (root) {
        x = root->data;
        return true;
    }
    else return false; // 没有根节点
}
```

```
template<class T>
void BinaryTree<T>::MakeTree(const T& element, BinaryTree<T>& left, BinaryTree<T>&
    // 将left, right和element 合并成一棵新树
    // left, right和this必须是不同的树
    // 创建新树
    root = new BinaryTreeNode<T> (element, left.root, right.root);
    // 阻止访问left和right
    left.root = right.root = 0;
}

template<class T>
void BinaryTree<T>::BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>& righ
    // left, right和this必须是不同的树
    // 检查树是否为空
    if (!root) throw BadInput(); // 空树
    // 分解树
    element = root->data;
    left.root = root->LeftChild;
    right.root = root->RightChild;
    delete root;
    root = 0;
}

template<class T>
// 前序遍历
void BinaryTree<T>::PreOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T>
    if(t){
        Visit(t);
        PreOrder(Visit, t->LeftChild);
        PreOrder(Visit, t->RightChild);
    }
}

template <class T>
// 中序遍历
void BinaryTree<int>::InOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T
    if (t) {
        InOrder(Visit, t->LeftChild);
        Visit(t);
        InOrder(Visit, t->RightChild);
    }
}

template <class T>
// 后序遍历
void BinaryTree<T>::PostOrder(void(*Visit)(BinaryTreeNode<T> *u), BinaryTreeNode<T
    if (t) {
        PostOrder(Visit, t->LeftChild);
```

```
        PostOrder(Visit, t->RightChild);
        Visit(t);
    }
}

template<class T>// 逐层遍历
void BinaryTree<T>::LevelOrder(void(*Visit)(BinaryTreeNode<T> *u))
{
    LinkedQueue<BinaryTreeNode<T>*> Q;
    BinaryTreeNode<T> *t;
    t = root; while (t) {
        Visit(t);
        if (t->LeftChild) Q.Add(t->LeftChild);
        if (t->RightChild) Q.Add(t->RightChild);
        try {Q.Delete(t);}
        catch (OutOfBounds) {return;}
    }
}
```

11.8 类linkedBinaryTree

类linkedBinaryTree是抽象类binaryTree的派生类，节点类型是binaryTreeNode（见11.4.2）。下面展示linkedBinaryTree的数据成员和访问方法。

```
template<class T>
class linkedBinaryTree:public binaryTree<binaryTreeNode<E>> {
    public :
        linkedBinaryTree() { root = NULL; treeSize = 0; }
        ~linkedBinaryTree() { return(); };
        bool  empty()const { return treeSize == 0; }
        int size()const { return treeSize; }
        void preOrder(void (*theVisit)(binaryTreeNode<E>*)) {
            visit = theVisit; preOrder(root);
        }
        void inOrder(void (*theVisit)(binaryTreeNode<E>*)) {
            visit = theVisit; inOrder(root);
        }
        void postOrder(void (*theVisit)(binaryTreeNode<E>*)) {
            visit = theVisit; postOrder(root);
        }
        void levelOrder(void (*)(binaryTreeNoode<E>*));
        //erase方法利用静态方法dispose作为函数指针visit方法，删除二叉树所有节点。
        void erase() {
            postOrder(dispose);
            root = NULL;
            treeSize = 0;
        }
```

```
    private:
        binaryTreeNode<E>* root;
        int treeSize;
        static void (*visit)(binaryTreeNode<E>*);
        static void preOrder(binaryTreeNode<E>*t);
        static void inOrder(binaryTreeNode<E>* t);
        static void postOrder(binaryTreeNode<E>* t);
        //静态方法dispose删除一个节点
        static void dispose(binaryTreeNode<E>* t) { delete t; }
};
```

> 类linkedBinaryTree有两个数据成员实例root和treeSize。有一个静态数据程序成员visit，它是一个函数指针，这个函数指针返回值类型是void，参数是binaryTreeNode类型的指针。

11.9 应用

11.9.1 设置信号放大器

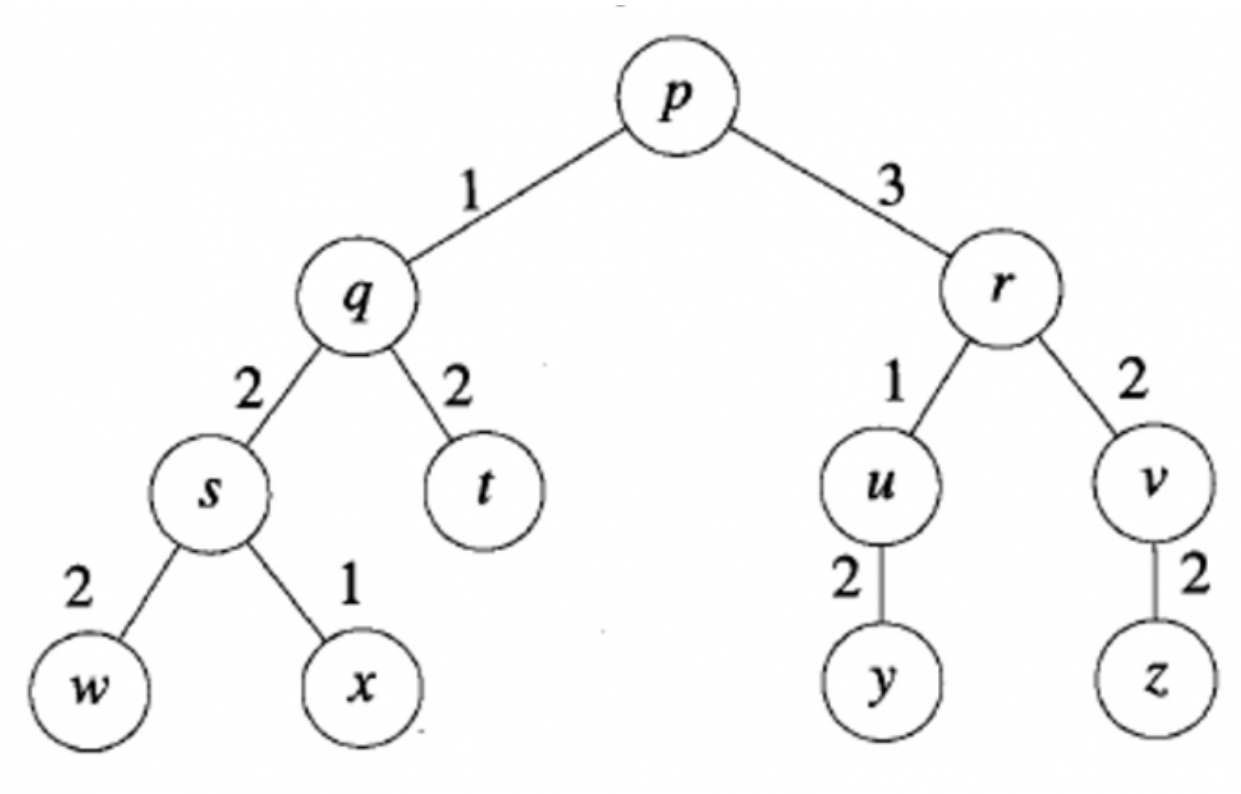
1. 问题描述

信号（signal）指称通过管道网络输送的资源(汽油、天然气、电力等)。当信号在网络中传输时，其性能的某一个或几个方面可能会有所损失或衰减。在信号从信号源到消耗点传输过程中，仅能容忍**一定范围内的**信号衰减。

为了保证信号衰减不超过**容忍值**（tolerance），应在网络中合适的位置放置**信号放大器**（signal booster）。信号放大器可以增加信号的压强或电压使其与源端的相同。

为简化问题，设分布网络是一树形结构，源是树的根。树中的每一节点(除了根)表示一个可以用来放置放大器的子节点。信号从一个节点流向其子节点。每条边上标出从父节点到子节点的信号衰减量。

如下图举例：



2. 求解策略

设degradeFromParent(i)表示节点i与其父节点间的衰减量。例如，degradeFromParent(w) = 2，degradeFromParent (p)=0。

在以 i 为根的子树中，从节点 i 到子树的每个叶子都有一个衰减量，将衰减量中的最大值记为degradeToLeaf(i)。若 i 是叶节点则degradeToLeaf(i)=0。

对于非叶子节点，有如下方程：

$$degradeToLeaf(i) = \max_{j是i的一个孩子}\{degradeToLeaf(j) + degradeFromParent(j)\}$$

因此，degradeToLeaf(s)=2。要计算节点的degradeToLeaf值，必须先计算其子节点的degradeToLeaf值。因而必须遍历整棵树，先访问子节点然后访问父节点。

按照上面的方法，在计算degradeToLeaf值的过程中，遇到一节点i，且其有一子节点j 满足degradeToLeaf(j)+degradeFromParent(j)>容忍值。若不在j 处放置放大器，则从i 节点到叶节点的信号衰减量将会超过容忍值，即使在 i 处放置放大器也是如此。

下面我们给出放置放大器的伪代码：

```
degradeToLeaf ( i )=0;
for (i 的每个孩子 j){
    if ((degradeToLeaf(j )+degradeFromParent(j))>tolerance){
```

```
            在 j 放置放大器;
            degradeToLeaf(i)=max{degradeToLeaf(i),degradeFromParent(j)};
        }
        else{
            degradeToLeaf(i)=max{degradeToLeaf(i),degradeToLeaf(j)+degradeFromParent(
        }
    }
```

定理 按上述算法放置的放大器最少。
证明详见P286

3. C++实现
   结构booster：

```
struct booster {
    int degradeToLeaf,degradeFromParent;
    //到达叶子时的衰减量，从父节点出发的衰减量
    bool boosterHere;
    void Output(ostream & out) const{
        out<<boosterHere<<' '<<degradeToLeaf<<' '<<degradeFromParent<<' ';
        }
};
// 重载操作符 <<
    ostream& operator<<(ostream& out, booster x){
        x.Output(out);
        return out;
    }
```

放置放大器并计算degradeToLeaf：

```
void placeboosters(binaryTreeNode<booster> *x){
    //计算*x的衰减量，如果衰减量超出了容忍值，则设置放大器
    x->element.degradeToLeaf=0;
    //计算左子树的衰减量，若大于容忍值，在x左孩子处放置放大器
    binaryTreeNode<booster> *y = x->leftChild;

    if (y!=NULL) {
        // 从左孩子来计算
        int degradation = y->element.degradeToLeaf + y->element.degradeFromParent
        if (degradation > tolerance){
            //在y处放一个放大器
            y->element.boosterHere = true;
            x->element.degradeToLeaf=y->element.degradeFromParent;
        }
        else x->element.degradeToLeaf = degradation;
    }
```

```
    //计算右子树的衰减量，若大于容忍值，在x右孩子处放置放大器
    y = x->rightChild;
    if (y!=NULL) {
        // x有非空右子树
        int degradation = y->element.degradeToLeaf + y->element.degradeFromParent;
        if (degradation > tolerance){
            //在y处放一个放大器
            y->element.boosterHere = true;
            degradation = y->element.degradeFromParent;
        }
        if (x->element.degradeToLeaf<degradation ){
            x->element.degradeToLeaf =degradation;
        }
    }
}
```

11.9.2 并查集

1.问题描述

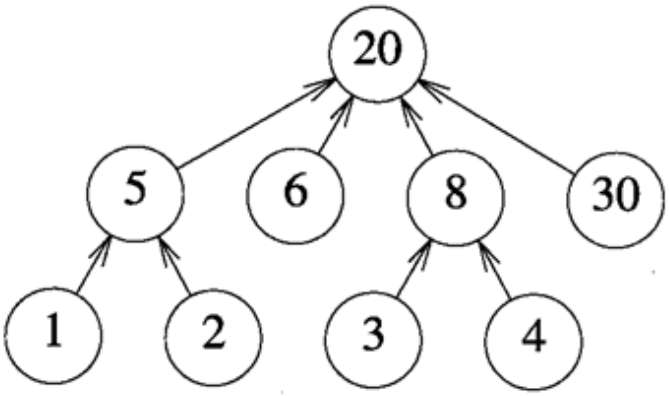
前面我们已经讨论过并查集问题。这一节我们将采用另一种方案来解决并查集问题，其中每个集合（类）被表示为一棵树。

2.集合的树形描述

任何一个集合S都可以描述为一棵具有 |S|个节点的树，一个节点代表一个元素。任何一个元素可以作为根元素。每个非根节点都有一个指针指向其父节点。这样设计的原因是查找操作需要向上搜索一棵树。

3.求解策略

- 把每一个集合表示为一棵树。
- 根元素作为集合标识符。因为一个集合只有一个根，所以当且仅当 i 和 j 同属于一个集合时，举例：



find(1)的返回值是20；find(3)的返回值是20。

- 为了确定某元素属于哪个集合，我们沿着节点找父节点向上移动，直到根节点为止。
- 在合并时，为了把两个树集合合并，我们让一个树成为另一棵树的子树。

4.C++实现

我们使用树的链表描述，每个节点必须有一个parent域，但不必有孩子域。根节点的parent域被置为0。因为没有索引为0的节点，所以parent为0表示不指向任何节点（即为空链）。

```
void initialize(int numberOfElements){
    // 初始化，每个类/树有一个元素
    parent = new int[numberOfElements+1];
    for (int e = 1; e <= numberOfElements; e++) parent[e] = 0;
}

int find(int theElement){
    //返回包含e的树的根节点,也就是集合标志符
    while (parent[theElement]!=0)
        theElement = parent[theElement];// 上移一层,寻找集合标志符
    return theElement;
}

void unite(int rootA, int rootB)
{// 将根为 rootA 和rootB 的两棵树进行合并
    parent[rootB] = rootA;
}
```

5.性能分析

构造函数的时间性能是O(numberOfElements);查找函数find的时间性能是$O(h)$,其中h是树的高度。合并函数unite的时间性能是 $\Theta(1)$。

- 在并查集这个应用中，我们不关心一个操作需要多少时间，我们关心整个操作需要的时间。
- 假设一系列操作要执行u 次合并和f 次查找。因为每次合并前都必须执行两次查找，因此可假设 f > u。
- 每次合并所需时间为$\Theta(1)$。
- 而每次查找所需时间由树的高度决定。每一次查找需花费$\Theta(q)$时间，其中q是在执行查找之前所进行的合并操作的次数。
- 一系列操作的时间为$O(fu)$。

6.合并函数的性能改进

在对树i 和树j 进行合并操作时，可以通过使用重量规则或高度规则来提高算法的性能。

> **重量规则**：若根为 i 的树的节点数少于根为 j 的树的节点数，则将j 作为i 的父节点。否则，将i 作为j 的父节点。
> **高度规则**：若根为i 的树的高度小于根为 j 的树的高度，则将j 作为i 的父节点，否则将i 作为j 的父节点。

注意：同样是合并两棵树，根据不同规则合并的结果可能是不一样的。

定义结构unionFindNode

```
struct unionFindNode{
    bool root;//是根时，才为真
    int parent; //若为真，表示树重量，否则是父节点的指针
    unionFindNode(){
        parent=1;
        root=true;
    }
}
```

重量规则合并：

```
void initialize(int numberOfElements){
    // 初始化，每个类/树有一个元素
    node=new unionFindNode[numberOfElements+1];
}

int find(int theElement){
    //返回元素所在树的根
    while (!node[theElement].root)
        theElement = node[theElement].parent;// 上移一层
    return theElement;
```

```
}

void unite(int rootA, int rootB)
{// 将根为 rootA 和rootB 的两棵树进行合并
    if(node[rootA].parent<node[rootB].parent){
        //树rootA成为树rootB的子树
        node[rootB].parent+=node[rootA].parent;
        //重量加和
        node[rootA].root=false;
        //不再为根节点
        node[rootA].parent=rootB;
        //链接到B上
    }
    else{
         //树rootB成为树rootA的子树
        node[rootA].parent+=node[rootB].parent;
        node[rootB].root=false;
        node[rootB].parent=rootA;
    }
}
```

> **重量规则引理**:
> 假设从单元素集合出发，用重量规则进行合并操作。若按此方法构建有p个节点的树t，则t 的高度最多为$\lfloor \log_2 p \rfloor + 1$。

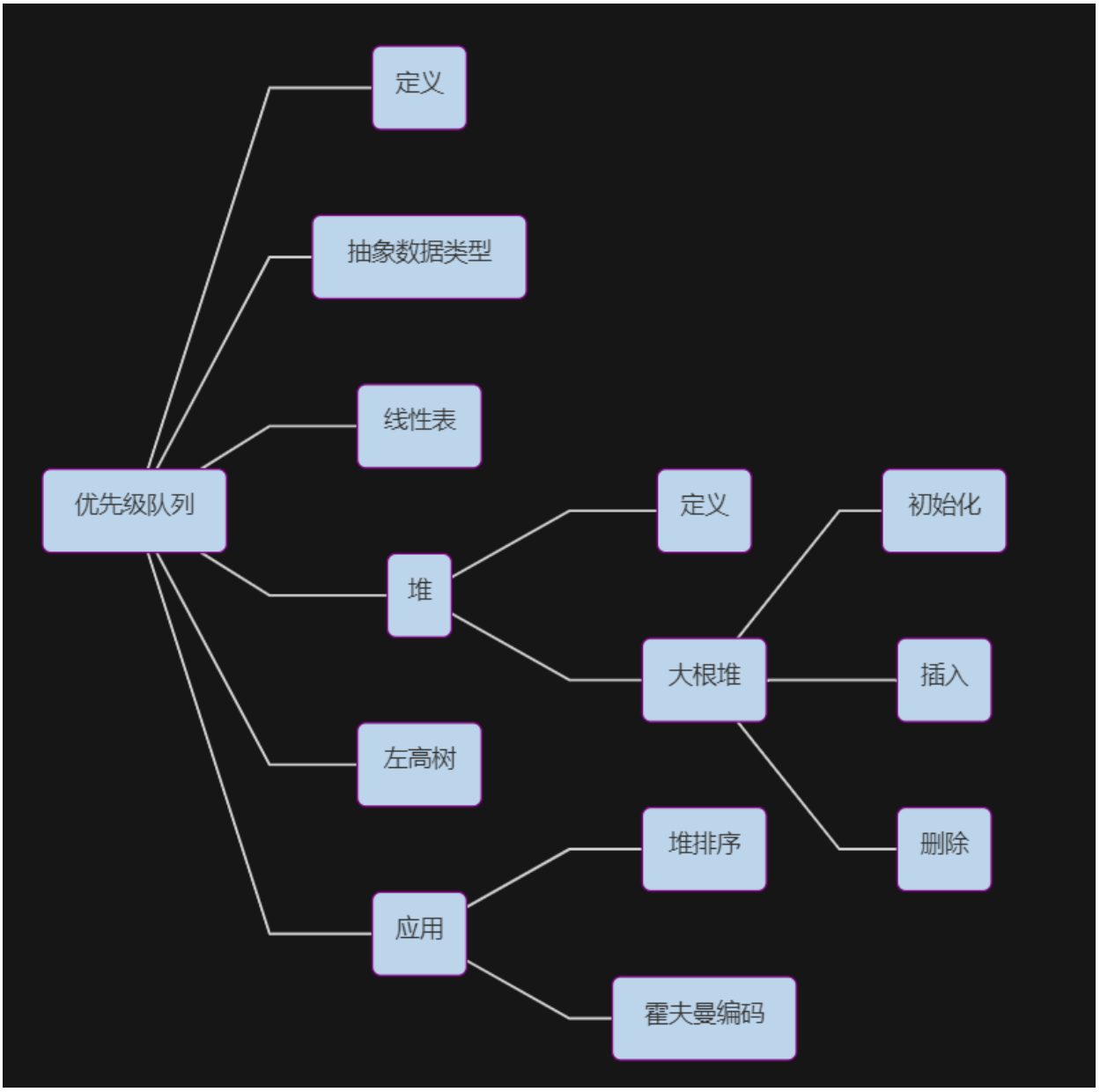
7.查找函数的性能改进
进一步改进查找函数在最坏情况下的性能，方法是:
缩短从元素到根的查找路径。

路径压缩(path compression)过程，实现起来有三种不同途径:

- 路径紧缩(path compaction)
  从要查找的节点到根的路径上所有节点的指针都改为指向根节点。虽然路径紧凑增加了单个查找操作的时间，但它减少了此后查找操作的时间。
- 路径分割(path splitting)
  改变从e 到根节点路径上每个节点（除了根和其子节点）的 parent指针，使其指向各自的祖父节点。
- 路径对折(path halving)
  改变从e 到根节点路径上每隔一个节点（除了根和其子节点）的 parent 域， 使其指向各自的祖父节点。在路径对折中指针改变数仅为路径分割中的一半。同样，在路径对折中只考虑从e 到根节点的一条路径就够了。

第12章 优先级队列

本章概述



- 与第9章的队列结构不同，优先队列中元素出队列的顺序由元素的优先级决定。从优先队列中删除元素是根据优先权高或低的次序，而不是元素进入队列的次序。

12.1 定义

- 优先队列是0个或多个元素的集合，每个元素都有一个优先权或值。
- 对优先队列执行的操作有1) 查找；2) 插入一个新元素；3) 删除
- 两种优先级队列：
  最小优先队列：“查找/删除”操作用来“搜索/删除”优先权最小的元素

最大优先队列：“查找/删除”操作用来“搜索/删除”优先权最大的元素

12.2 抽象数据类型

- 最大优先队列的抽象数据类型描述

```
抽象数据类型MaxPriorityQueue
{
  实例
    有限的元素集合，每个元素都有一个优先权
  操作
    Create()：创建一个空的优先队列
    Size()：返回队列中的元素数目
    Max()：返回具有最大优先权的元素
    Insert(x)：将x 插入队列
    DeleteMax(x)：从队列中删除具有最大优先权的元素，并将该元素返回至 x
}
```

- 抽象类maxPriorityQueue

```
template<class T>
class maxPriorityQueue
{
    public:
        virtual ~maxPriorityQueue(){}
        virtual bool empty() const = 0; //返回true，当且仅当队列为空
        virtual int size() const = 0; //返回队列中元素个数
        virtual T& top() = 0; //返回优先级最大的元素的引用
        virtual void pop() = 0; //删除首元素
        virtual void push(const T& theElement) = 0; //把元素theElement加入队尾
};
```

- 最小优先队列的抽象数据类型描述与之类似，只需将最大改为最小即可

12.3 线性表

采用无序线性表描述最大优先队列

- 公式化描述（利用公式Location(i)=i-1）
  插入：表的右端末尾执行，时间：Θ(1)；
  删除：查找优先权最大的元素，时间：Θ(n)；
- 使用链表
  插入：在链头执行，时间：Θ(1)；

删除：Θ(n)；

采用有序线性表描述最大优先队列

- 公式化描述（利用公式Location(i)=i-1，元素按递增次序排列)
  插入： O(n)；删除：O(1)；
  使用链表（按递减次序排列）
  插入： O(n)；删除：O(1)；

12.4 堆

12.4.1 定义

- [最大树（最小树）] 每个节点的值都大于（小于）或等于其子节点（如果有的话）值的树
- [大根堆（小根堆）] 最大（最小）的完全二叉树

12.4.2 类maxHeap

- 下面的程序给出了最大堆的类定义。n是私有成员，代表目前堆中元素的个数；MaxSize是堆的最大容量heap为存贮堆元素的数组，省缺堆的大小为10个元素。

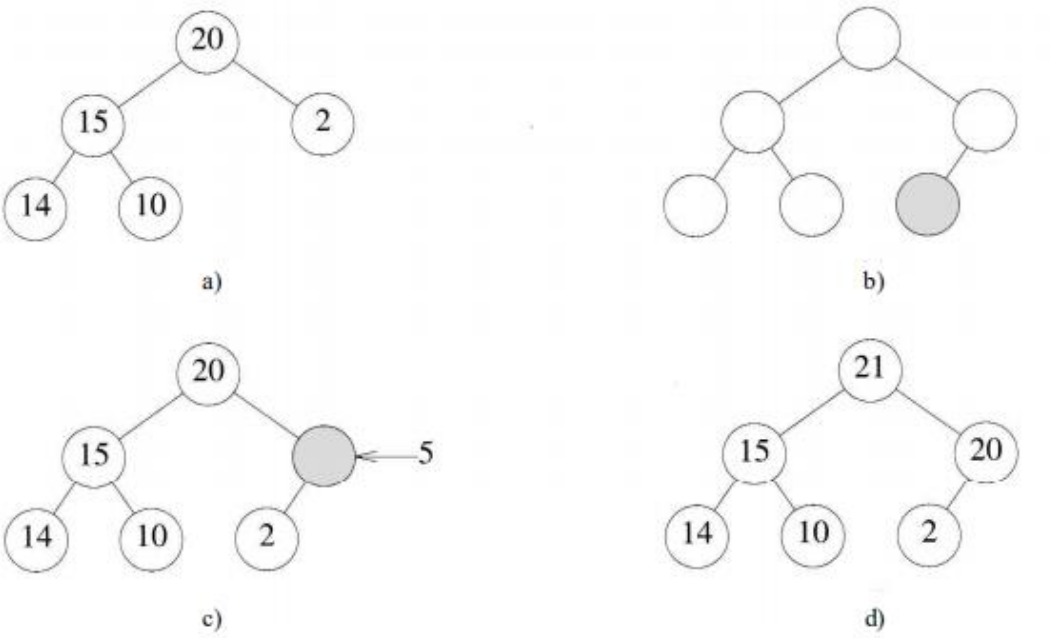
```
template<class T>
class MaxHeap {
public:
  MaxHeap(int MaxHeapSize = 10);
  ~MaxHeap() {delete [] heap;}
  int Size() const {return CurrentSize;}
  T Max()
  {
      if (CurrentSize == 0) throw OutOfBounds();
      return heap[1];
  }
  MaxHeap<T>& Insert(const T& x);
  MaxHeap<T>& DeleteMax(T& x);
  void Initialize(T a[], int size, int ArraySize);
private:
  int CurrentSize, MaxSize;
  T *heap; // 元素数组
} ;
```

- MaxHeap的构造函数

```
template<class T>
MaxHeap<T>::MaxHeap(int MaxHeapSize)
{// 构造函数
  MaxSize = MaxHeapSize;
  heap = new T[MaxSize+1];
  CurrentSize = 0;
}
```

12.4.3 大根堆的插入

- 插入到最后位置再做调整



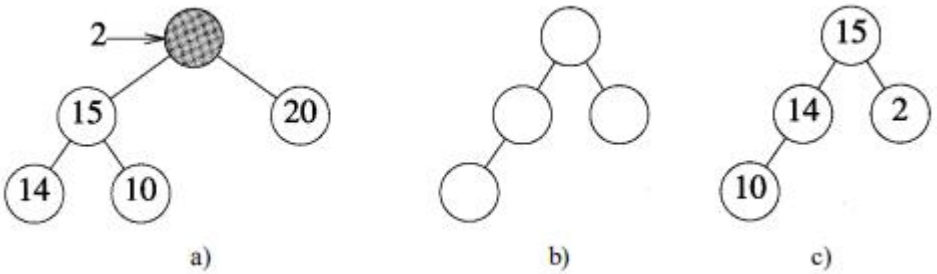
- 插入策略从叶到根只有单一路径，每一层的工作需耗时Θ(1)因此实现策略的时间复杂性为O(height)=O(logn)

```
template<class T>
MaxHeap<T>& MaxHeap<T>::Insert(const T& x)
{// 把 x 插入到最大堆中
  if (CurrentSize == MaxSize)
    throw NoMem(); // 没有足够空间
//为 x寻找应插入位置
// i 从新的叶节点开始，并沿着树上升
  int i = ++CurrentSize;
  while (i != 1 && x > heap[i/2]) {
  // 不能够把 x 放入 h e a p [ i ]
  heap[i] = heap[i/2]; // 将元素下移
  i /= 2; // 移向父节点
  }
```

```
  heap[i] = x;
  return *this;
}
```

12.4.4 大根堆的删除

- 从堆中删除一个元素后，先将最后的元素填充到该位置，然后再按大根堆插入时的方法进行调整



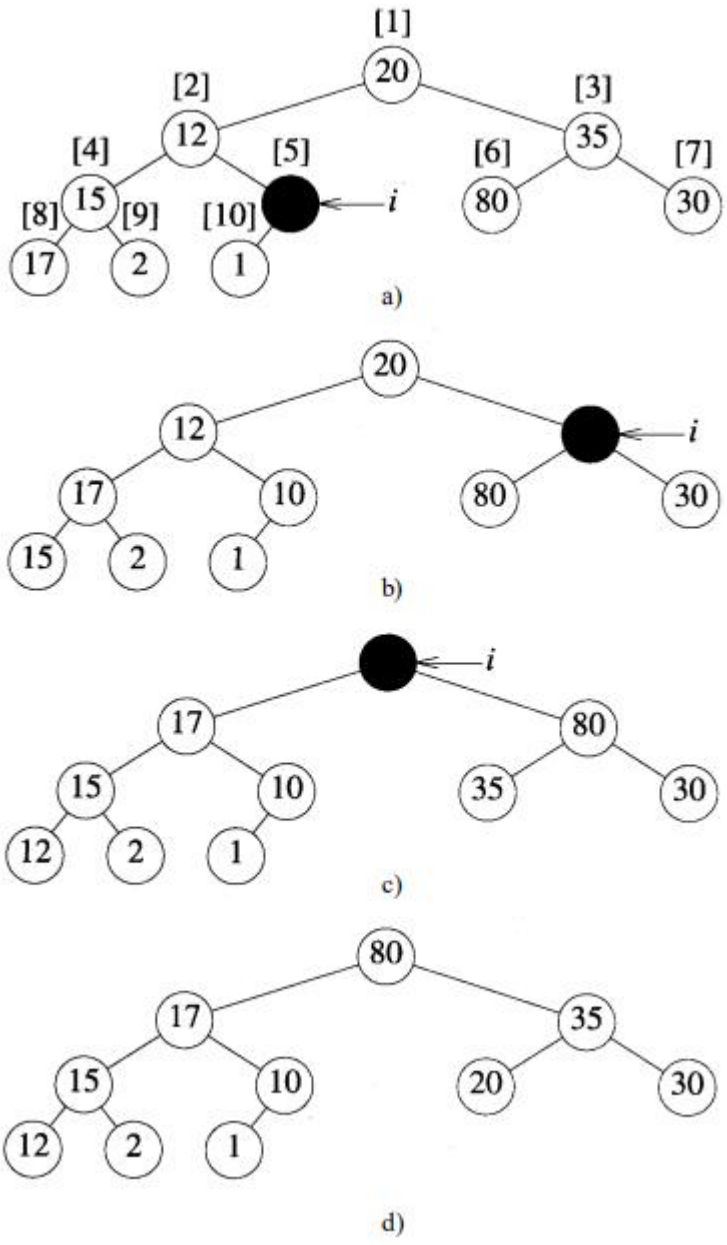
- 删除策略产生了一条从堆的根节点到叶节点的单一路径，每层工作需耗时Θ(1)因此实现此策略的时间复杂性为O(height)=O(logn)

```
template<class T>
MaxHeap<T>& MaxHeap<T>::DeleteMax(T& x)
{// 将最大元素放入 x ，并从堆中删除最大元素
// 检查堆是否为空
  if (CurrentSize == 0)
    throw OutOfBounds(); // 队列空
  x = heap[1]; // 最大元素
  // 重构堆
  T y = heap[CurrentSize--]; // 最后一个元素
  // 从根开始，为y 寻找合适的位置
  int i = 1, // 堆的当前节点
  ci = 2; // i的孩子
  while (ci <= CurrentSize) {
    // heap[ci] 应是 i的较大的孩子
    if (ci < CurrentSize &&
      heap[ci] < heap[ci+1])
      ci++;
    // 能把 y 放入h e a p [ i ]吗?
    if (y >= heap[ci])
     break; // 能
    // 不能
    heap[i] = heap[ci]; // 将孩子上移
    i = ci; //下移一层
    ci *= 2;
```

```
  }
  heap[i] = y;
  return *this;
}
```

12.4.5 大根堆的初始化

- 从第一个具有孩子的节点开始，这个元素在数组中的位置为i=[n/2]，如果以这个元素为根的子树已是大根堆，则此时不需调整，否则必须调整子树使之成为堆。
  随后，继续检查以i-1,i-2等节点为根的子树，直到检查到整个二叉树的根节点（其位置为1）



- 初始化一个非空大根堆

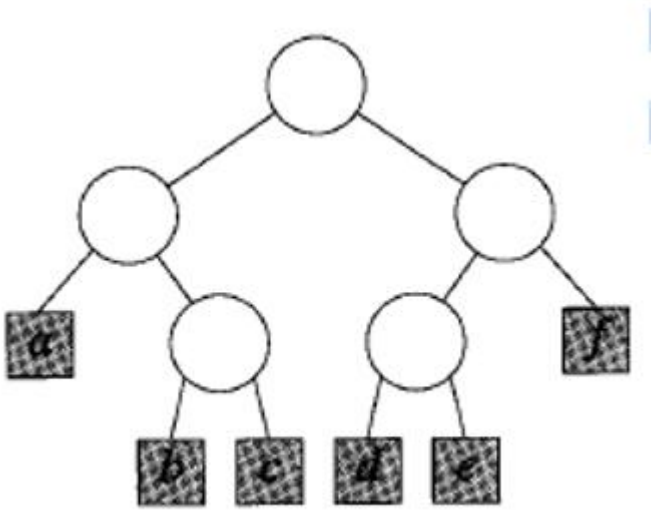
```
template<class T>
void MaxHeap<T>::Initialize(T a[], int size, int ArraySize)
{// 把最大堆初始化为数组 a .
  delete [] heap;
  heap = a;
  CurrentSize = size;
  MaxSize = ArraySize;
  // 产生一个最大堆
  for (int i = CurrentSize/2; i >= 1; i--) {
    T y = heap[i]; // 子树的根
    // 寻找放置 y的位置
    int c = 2*i; // c的父节点是y的目标位置
    while (c <= CurrentSize)
    {
    // heap[c] 应是较大的同胞节点
    if (c < CurrentSize &&heap[c] < heap[c+1])
      c++;
    // 能把 y 放入h e a p [ c / 2 ]吗?
    if (y >= heap[c])
      break; // 能
    // 不能
    heap[c/2] = heap[c]; // 将孩子上移
    c *= 2; // 下移一层
    }
  heap[c/2] = y;
  }
}
```

12.5 左高树

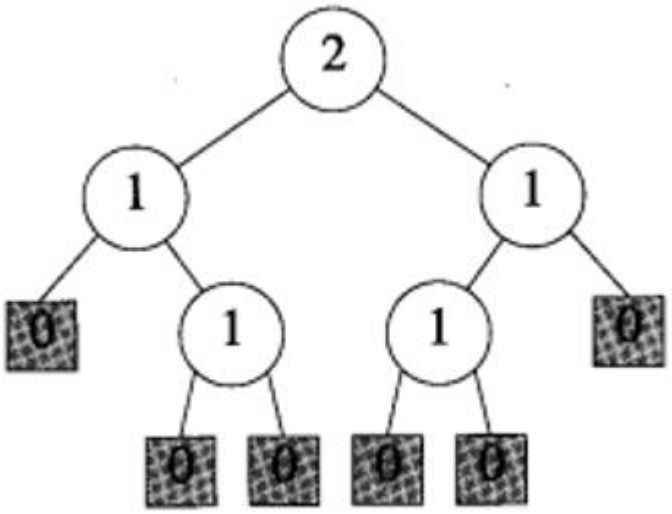
12.5.1 高度优先与宽度优先的最大及最小左高树

- [扩充二叉树]
  1）外部节点(External node) –代替树中的空子树的节点
  2）内部节点(Internal node) – 具有非空子树的节点

3）扩充二叉树(Extended binary tree) –增加了外部节点的二叉树



- 令s(x)为从节点x 到它的子树的外部节点的所有路径中最短的一条
  1）若x是外部节点，则s的值为0；
  2）若x为内部节点，则它的s值是： min{s(L), s®} + 1
  其中L与R分别为x 的左右孩子



- [高度优先左高树] 当且仅当一棵二叉树的任何一个内部节点，其左孩子的 s 值大于等于右孩子的s 值时，该二叉树为高度优先左高树（height-biased leftist tree, HBLT）

- 令x 为一个HBLT的内部节点，则
  1)以x为根的子树的节点数目至少为2s(x)-1
  2)若子树x有m个节点，s(x)最多为$\log_2(m+1)$
  3)通过最右路径（即，此路径是从 x 开始沿右孩子移动）从x 到达外部节点的路径长

度为
s(x)。

- [最大HBLT] 即同时又是最大树的HBLT;
  [最小HBLT] 即同时又是最小树的HBLT.

12.5.2 类maxHblt

- 最大HBLT的每个节点均需要 data、LeftChild、RightChild和s四个域，相应的节点类为HBLTNode。在HBLTNode的构造函数中要求提供data和s，同时将LeftChild和RightChild置为0。

```
template <class T>

class HBLTNode {
friend MaxHBLT < T > ;
public:
  HBLTNode(const T& e, const int sh)
  {
    data = e;
    s = sh;
    LeftChild = RightChild = 0;
  }
private:
  int s; // 节点的s 值
  T data;
  HBLTNode<T> *LeftChild, *RightChild;
} ;
```

- MaxHBLT类

```
template<class T>
class MaxHBLT {
public:
  MaxHBLT() {root = 0;}
  ~MaxHBLT() {Free(root);}
  T Max() {if (!root) throw OutOfBounds();
      return root->data;}
  MaxHBLT<T>& Insert(const T& x);
  MaxHBLT<T>& DeleteMax(T& x);
  MaxHBLT<T>& Meld(MaxHBLT<T>& x) {
    Meld (root , x.root) ;
    x.root = 0;
    return *this;}
```

```
  void Initialize(T a[], int n);
private:
  void Free(HBLTNode<T> *t);
  void Meld(HBLTNode<T>* &x, HBLTNode<T>* y);
  HBLTNode<T> *root; // 指向树根的指针
} ;
```

12.5.3 最大HBLT的合并

- 合并策略用递归来实现(令A、B为需要合并的两棵最大HBLT):

  1)

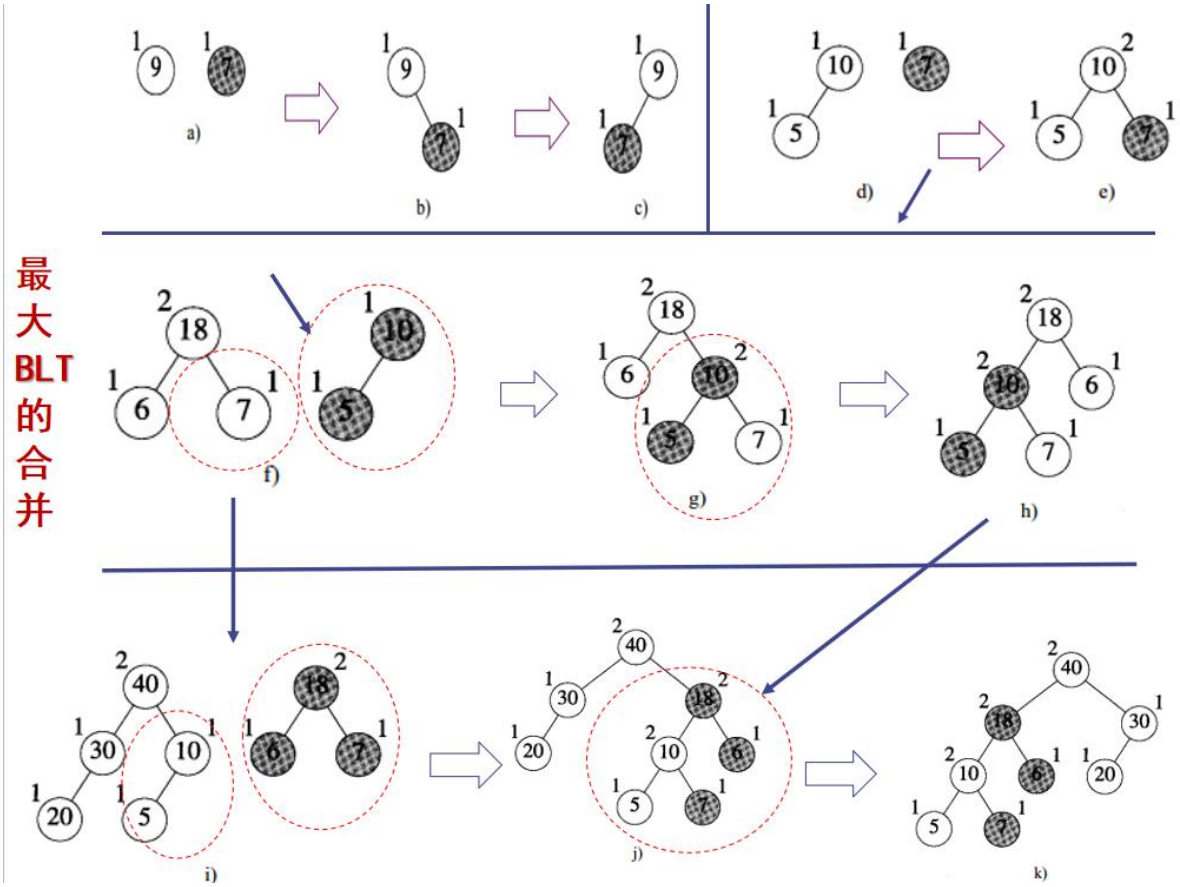
  如果其中一个为空，则将另一个作为合并的结果，因此可以假设两者均不为空。

  2)

  为实现合并，先比较两个根元素，较大者作为合并后的HBLT的根。假定A具有较大的根，且其左子树为L，C是由A的右子树与B合并而成的HBLT。

  3)

  A与B合并所得结果即是以A的根为根，L与C为左右子树的最大HBLT。如果L的s值小于C的s值，则C为左子树，否则L为左子树。




```
template<class T>
void MaxHBLT < T > : : M e l d ( H B LTNode<T>* &x, HBLTNode<T>* y)
{// 合并两棵根分别为* x和 * y的左高树
```

```
// 返回指向新根 x的指针
  if (!y) return; // y 为空
  if (!x) // x为空
  {x = y;
    return; }
// x和y 均为空
  if (x->data < y->data) Swap(x,y);
  // 现在 x->data >= y->data
  Meld(x->RightChild,y ) ;
  if (!x->LeftChild) {// 左子树为空
// 交换子树
  x->LeftChild = x->RightChild;
  x->RightChild = 0;
  x->s = 1;}
  else {// 检查是否需要交换子树
    if (x->LeftChild->s < x->RightChild->s)
    Swap(x->LeftChild,x->RightChild);
  x->s = x->RightChild->s + 1;}
}
```

12.5.4 最大HBLT的插入、删除

- 最大HBLT的插入操作可借助于最大HBLT的合并操作来完成。
  1)
  假设将元素x插入到名为H的最大HBLT中
  2)
  建造一棵仅有一个元素x的最大HBLT
  3)
  然后将它与H进行合并
  4)
  结果得到的最大HBLT将包括H中的全部元素及元素x
  5)
  因此插入操作只需先建立一棵仅包含欲插入元素的HBLT，然后将它与原来的HBLT合并即可

- 删除操作可以通过删除根元素并对两个子树进行合并来实现
  根是最大元素
  如果根被删除
  将留下分别以其左右孩子为根的两棵HBLT的子树。
  将这两棵最大HBLT合并到一起，便得到包含除删除元素外所有元素的最大HBLT。

(详见p308-312)

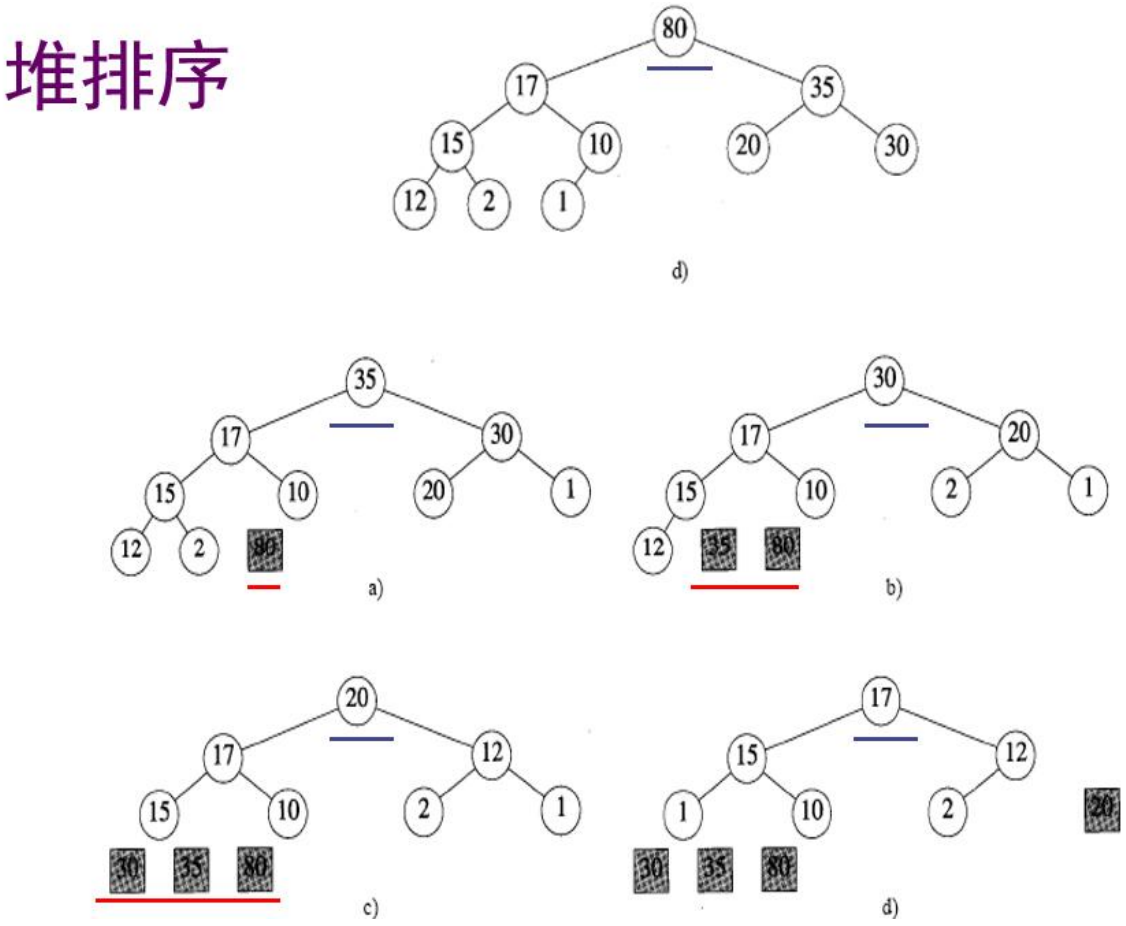
12.6 应用

12.6.1 堆排序

- 可利用堆来实现n个元素的排序，每个元素的关键值为优先权

- 堆排序算法：
  1）将要排序的n个元素初始化为一个最大(小)堆.
  2）每次从堆中提取（即删除）元素。
  如果使用最大堆,各元素将按非递减次序排列。
  如果使用最小堆,各元素将按非递增次序排列。

- 时间复杂度：
  初始化所需要的时间为(n)，每次删除所需要的时间为O(logn)，因此总时间为O(nlogn)。

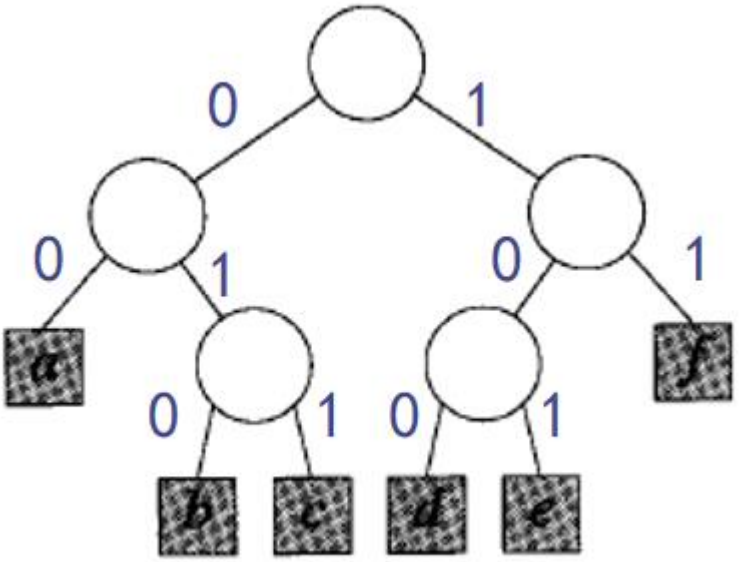


（详见p313、314）

12.6.2 霍夫曼编码

- 霍夫曼编码
  可以利用扩充二叉树来派生一个实现可变长编码的特殊类，该类满足上述前缀性

质，被称为霍夫曼编码。
在扩充二叉树中，可对从根到外部节点的路径进行编码，方法是向左孩子移动时取0，向右孩子移动时取1

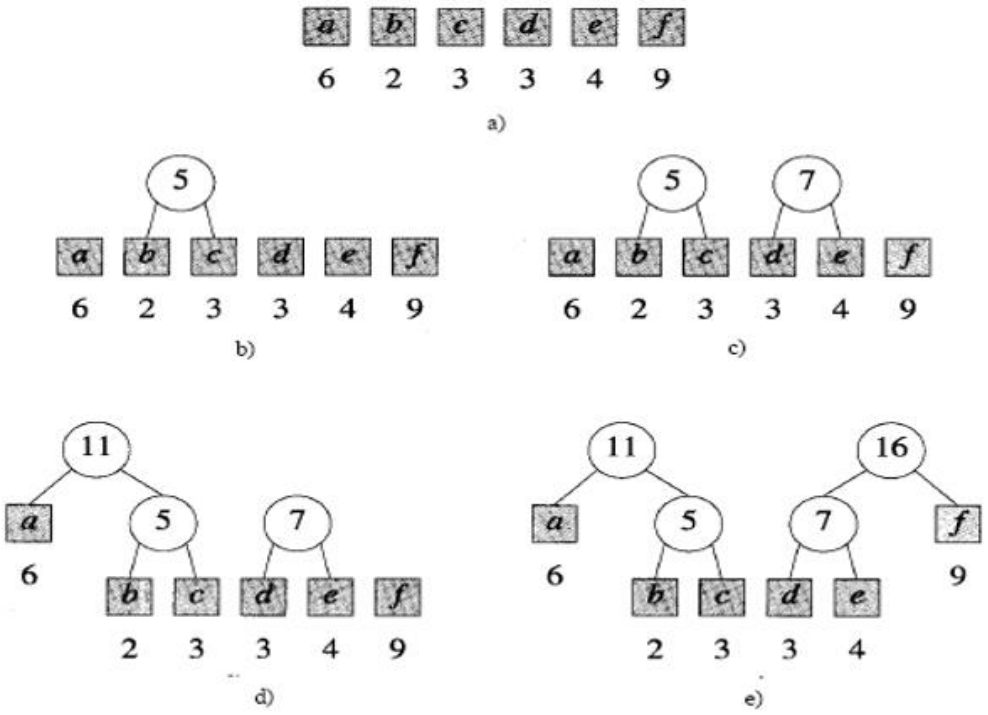


- 霍夫曼树:
  加权路径长度达到最小的扩充二叉树即为霍夫曼树。
  在霍夫曼树中，权值大的结点离根最近。

- 构造霍夫曼树

  - 为了构造霍夫曼树，首先从仅含一个外部节点的二叉树集合开始，每个外部节点代表字符串中一个不同的字符，其权重等于该字符的频率
  - 此后不断地从集合中选择两棵具有最小权重的二叉树，并把它们合并成一棵新的二叉树，合并方法是把这两棵二叉树分别作为左右子树，然后增加一个新的根节点。新二叉树的权重为两棵子树的权重之和。

- 这个过程可一直持续到仅剩下一棵树为止。



(详见p317-320)

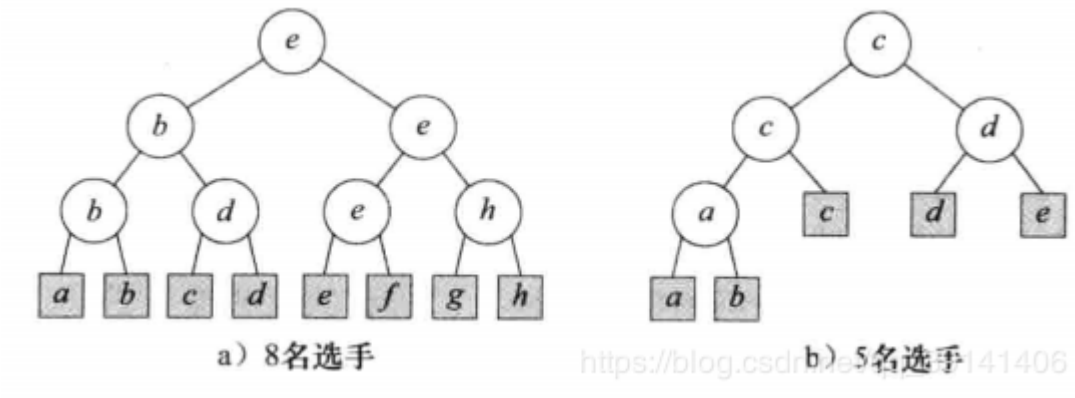
第13章 竞赛树

本章概述

本章将重点学习两种竞赛树——赢者树和输者树。赢者树相对而言更符合我们的值观印象，输者树则实现起来更高效。

13.1 赢者树

假定在一次比赛中有n名选手，每位选手只要输一次球就被淘汰，则最后必然存在一个胜利者。他们的赛程图就可以视为一棵赢者树（以我最喜欢的NBA季后赛为例，假如把他们的胜场看作比分，便可以把他们的赛程图视作一棵完整的赢者树）



13.1.1 定义

赢者树：有n个选手的赢者树是一个完全二叉树，有n个外部节点和n-1个内部节点，每个内部节点记录的是该节点比赛的赢者。

注意：现实的竞赛所对应的树不一定都是完全二叉树，但是用完全二叉树能使比赛的场次最少。

分类：我们假设获胜的条件取决于得分的比较，那么就会有两种可能：**得分最小的选手获胜**和**得分最大的选手**获胜。所以我们的赢者树也就分为最大赢者树和最小赢者树。

我们用左孩子对应的选手代表赢家节点。

优点：当一个选手分数改变时，修改竞赛树比较容易。重构赢者树需耗时$(log_2 n)$

实现——WinnerTree

在定义抽象数据类型WinnerTree时，我们需要假设选手数是固定的，也就是说选手本身不是赢者树的组成部分。那么对于一棵赢者树，就需要定义这样几个操作：创建空的赢者树，用n名选手初始化赢者树，返回赢者，在从选手i到根的路径上重新比赛。

抽象数据类型 *WinnerTree*
{
　　实例
　　　　完全二叉树，每一个节点指向比赛胜者；外部节点表示参赛者
　　操作
　　　initialize(*a*): 为数组 *a* 的参赛者初始化胜者树
　　　　winner(): 返回锦标赛胜者
　　　　rePlay(*i*): 在参赛者 *i* 改变之后重赛
}

假设我们使用完全二叉树的公式化描述来定义赢者树，假设有一棵有n名选手的赢者树，需要$n-1$个内部节点$t[1:n-1]$。选手（或外部节点）用数组$e[1:n]$表示，因此$t[i]$

是数组player的一个索引，类型为int。在赢者树的节点i对应比赛中，$t[i]$代表赢者。为了实现这种对应关系，我们必须能够确定外部节点$e[i]$的父节点$[p]$。当外部节点的个数为n时，内部节点的个数为$n-1$。最底层最左端的内部节点，其编号为2^s，且$s=[log_2(n-1)]$。因此，最底层内部节点的个数是$n-2^s$，最底层外部节点个数$LowExt+1$是这个数的2倍。倒数第二层最左端的外部节点号为$LowExt+1$。令$offset=2^{s+1}-1$。对于任何一个外部节点e[i]，其父节点t[p]由以下公式给出：

$$p=\begin{cases}(i+offset)/2 & i<=lowExt.\\(i-lowExt+n-1)/2 & i>lowExt\end{cases}$$

以下给出一个赢者树的具体实现方式

```
#include<iostream>

using namespace std;

template <class T>
class winnerTree
{
public:
    //构造函数和析构函数
    winnerTree(T* thePlayer, int theNumberOfPlayer)
    {
        tree = NULL;
        initializer(thePlayer, theNumberOfPlayer);
    }
    ~winnerTree(){ delete[] tree; }

    //使用数组thePlayer初始化一个赢者树
    void initializer(T *, int );
    //返回比赛最后赢者的下标
    int winner() const { return tree[1]; }
    //返回比赛中某一环节的比赛的赢者
    int winner(int i) const { return (i < numberofPlayer) ? tree[i] : 0; }
    //因改变某一个player的值进而重赛
    void rePlay(int);
    //输出
    void output() const;

private:
    T* player;//参赛成员组成的数组
    int numberofPlayer;//参赛成员的数量
    int* tree;//赢者树，数组表示，赢者树只包含比赛结果，节点表示赢者的下标

    int lowExt;//用于生成赢者树，表示最底层外部节点的个数
    int offset;//用于生成赢者树，=2^log(n-1) - 1，详见解析
```

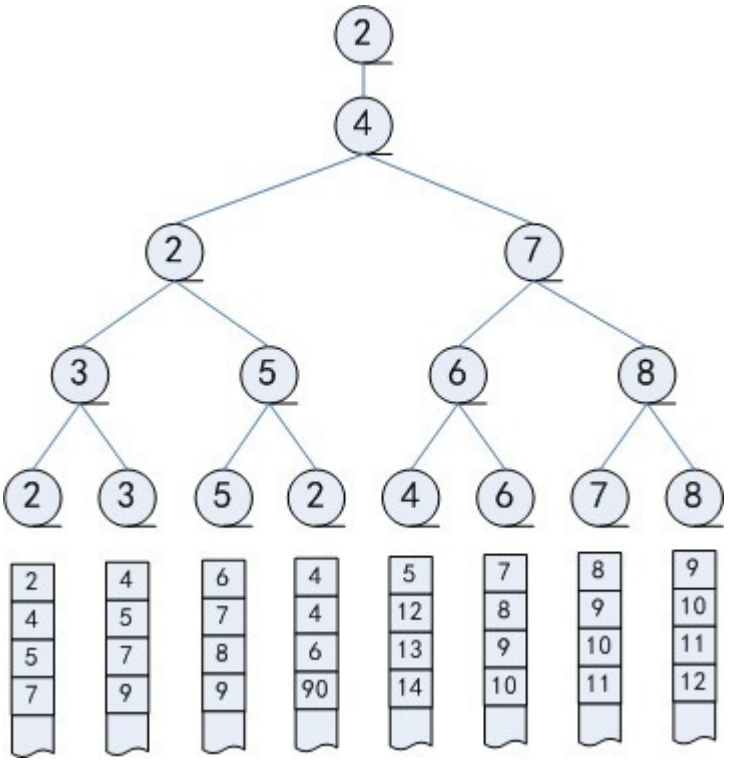
```
    void play(int, int, int);//初始化需要用到,
};

//使用数组thePlayer初始化为一个赢者树
template<class T>
void winnerTree<T>::initializer(T *thePlayer, int theNumberOfPlayers)
{
    int n = theNumberOfPlayers;
    if (n < 2)
    {
        cerr << "Must have at least 2 players"; exit(0);
    }

    player = thePlayer;
    numberofPlayer = n;
    delete[] tree;
    tree = new int[n];//n个外部节点，需要n-1内部节点，而数组0空置不用，所以需要n的空

    int s;
    for (s = 1; 2 * s <= n - 1; s += s);//最底层最左端的内部节点编号s = 2^log (n-1
    lowExt = 2 * (n - s);//最底层外部节点个数=2*最底层内部节点个数，最底层内部节点个
    offset = 2 * s - 1;//令offset=2*s-1
    //对于任何一个外部节点player[i]，其父节点tree[p]满足:
    //当i<=lowExt时，p=(i+offset)/2;  当i>lowExt时，p=(i-lowExt+n-1)/2;

    int i;
    //对最底层的外部节点比赛，从2，4，6。。开始知道判断
    for (i = 2; i <= lowExt; i += 2)
        play((offset + i) / 2, i - 1, i);
    //处理剩余的外部节点
    if (n % 2 == 1)
    {//如果外部节点n为奇数，需要特殊处理
        play(n / 2, tree[n - 1], lowExt + 1);
        i = lowExt + 3;
    }
    else//如果外部节点n为偶数
        i = lowExt + 2;
    for (; i <= n; i += 2)
        play((i - lowExt + n - 1) / 2, i - 1, i);
}

//play操作，从最开始的底层进行比赛，直到完成所有比赛决胜出最后赢者
template<class T>
void winnerTree<T>::play(int p, int leftchild, int rightchild)
{//从tree[p]开始的比赛，一直往上冒泡，直至根节点
    tree[p] = (player[leftchild] <= player[rightchild]) ? leftchild : rightchilc
    while (p % 2 == 1 && p > 1)
```

```
    {//p作为右孩纸出现，才继续往上走，因为最开始play中也是由右孩纸判断的
        tree[p / 2] = (player[tree[p - 1]] <= player[tree[p]]) ? tree[p - 1] : t
        p /= 2;
    }
}

//当thePlayer的值改变时，需要重赛,我们可以打乱然后重新初始化，但一般不这么做
template<class T>
void winnerTree<T>::rePlay(int thePlayer)
{
    int n = this->numberofPlayer;
    if (thePlayer <= 0 || thePlayer > n)
    {
        cerr << "Player index is illegal"; exit(0);
    }
    int matchNode,leftchid, rightchild;//下一个需要重赛的节点以及他的孩纸

    //找出第一个需要重赛的节点以及他的左右孩纸
    if (thePlayer <= lowExt)
    {
        matchNode = (offset + thePlayer) / 2;
        leftchid = 2 * matchNode - offset;
        rightchild = leftchid + 1;
    }
    else
    {
        matchNode = (thePlayer - lowExt + n - 1) / 2;
        if (2 * matchNode == n - 1)
        {
            leftchid = tree[2 * matchNode];
            rightchild = thePlayer;
        }
        else
        {
            leftchid = 2 * matchNode - n + 1 + lowExt;
            rightchild = leftchid + 1;
        }
    }
    tree[matchNode] = (player[leftchid] <= player[rightchild]) ? leftchid : righ

    //第二个需要重赛的节点中的特殊情况，即包含内部和外部节点
    if (matchNode == n - 1 && n % 2 == 1)
    {
        matchNode /= 2;
        tree[matchNode] = (player[tree[n - 1]] <= player[lowExt + 1]) ? tree[n -
    }
    //剩余的需要重赛的节点处理
```

```
    matchNode /= 2;
    for (; matchNode >= 1; matchNode /= 2)
        tree[matchNode] = (player[tree[2 * matchNode]] <= player[tree[2 * matchN
                           tree[2 * matchNode] : tree[2 * matchNode + 1];
}

//输出
template<class T>
void winnerTree<T>::output() const
{
    cout << "number of players  = " << numberofPlayer
        << " lowExt = " << lowExt
        << " offset = " << offset << endl;
    cout << "complete winner tree pointers are" << endl;
    for (int i = 1; i < numberofPlayer; i++)
        cout << tree[i] << ' ';
    cout << endl;
}
```

13.2 输者树

输者树是赢者树的一种变体。在输者树中，用父结点记录其左右子结点进行比赛的败者，而让胜者参加下一轮的比赛。输者树的根结点记录的是败者，需要加一个结点来记录整个比赛的胜利者。采用输者树可以简化重构的过程。



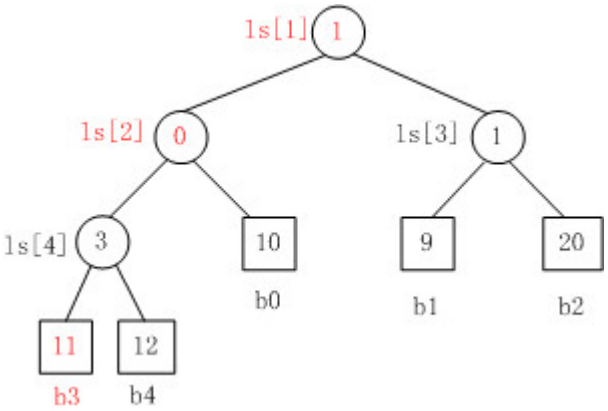
13.3 堆，胜者树，败者树的比较

相同点:

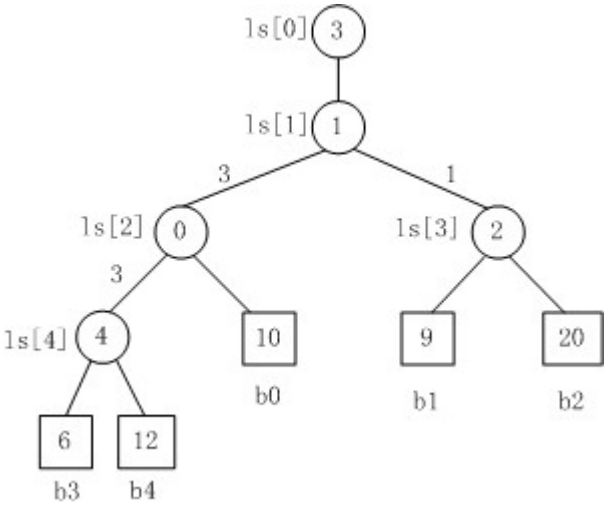
首先它们三个的相同点就是在于：空间和时间复杂度都是一样的。调整一次的时间复杂度都是O(logN)的。
所以这道题用堆来做，跟用败者树来做并没有本质上的算法复杂度量级上的差别。

不同点:

其实一开始就是只有堆来完成多路归并的，但是人们发现堆每次取出最小值之后，把最后一个数放到堆顶，调整堆的时候，每次都要选出父节点的两个孩子节点的最小值，然后再用孩子节点的最小值和父节点进行比较，所以每调整一层需要比较两次。
这时人们想能否简化比较过程，这时就有了胜者树



这样每次比较只用跟自己的兄弟节点进行比较就好，所以用胜者树可以比堆少一半的比较次数。
而胜者树在节点上升的时候首选需要获得父节点，然后再获得兄弟节点，然后再比较。
这时人们又想能否再次减少比较次数，于是就有了败者树



在使用败者树的时候，每个新元素上升时，只需要获得父节点并比较即可。
所以总的来说，减少了访存的时间。

第14章 搜索树

本章概述

二叉搜索树相关知识以及对其进行的部分查找，插入，删除操作

14.1 二叉搜索树

二叉搜索树又称二叉排序树，它或者是一棵空树，或者是具有以下性质的二叉树：

- 每个元素有一个关键值，并且没有任意两个元素有相同的关键值；所有的关键都是唯一的
- 若根节点的左子树不为空，则左子树上所有节点的值都小于根节点的值
- 若根节点的右子树不为空，则右子树上所有节点的值都大于根节点的值
- 根节点的左右子树也分别为二叉搜索树

带索引的二叉搜索树：它是在每个节点中添加一个LeftSize域，该域的值是该节点左子树的元素个数加1。

14.2二叉搜索树的部分操作

我们首先要给出二叉搜索树和带索引的抽象数据类型描述

抽象数据类型 *bsTree*
{
    实例
        二叉树，每一个节点都有一个数对，其中一个成员是关键字，另一个成员是数值；所有关键字都不相同；
        任何一个节点的左子树的关键字小于该节点的关键字；右子树的关键字大于该节点的关键字
    操作
    find(*k*): 返回关键字为 *k* 的数对
  insert(*p*): 插入数对 *p*
   erase(*k*): 删除关键字为 *k* 的数对
  ascend(): 按关键字升序输出所有数对
}

抽象数据类型 *IndexedBSTree*

{

实例

与 bsTree 的实例相同，只是每一个节点还有一个 leftSize 域

操作

find(*k*): 返回关键字为 *k* 的数对

get(*index*): 返回第 *index* 个数对

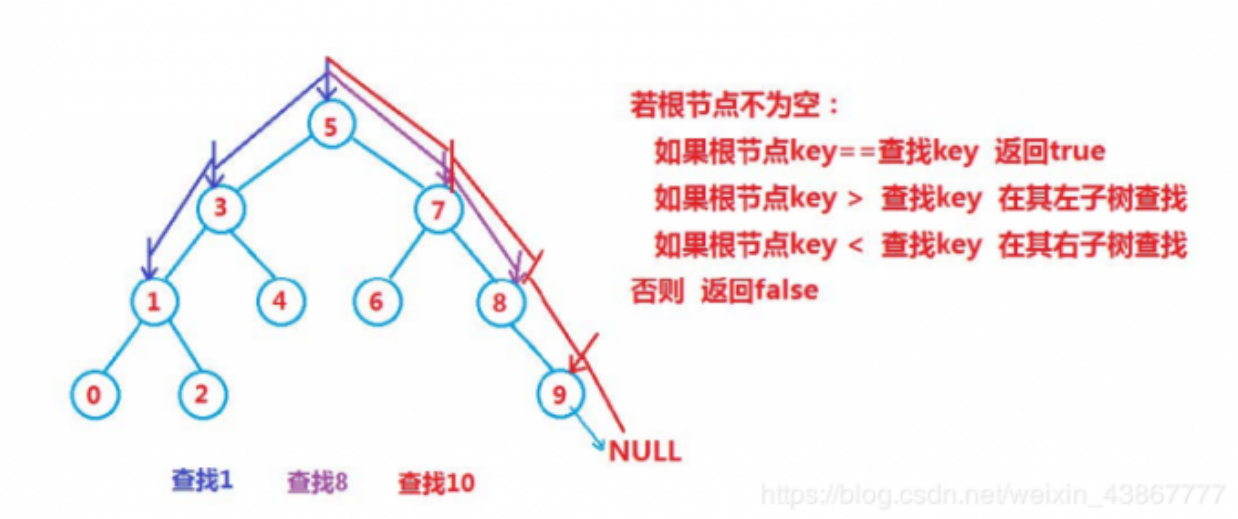
insert(*p*): 插入数对 *p*

erase(*k*): 删除关键字为 *k* 的数对

ascend(): 按关键字升序输出所有数对

}

14.2.1 搜索



通过上图，我们很容易的可以想到，查询最小元素，就是使用递归从根节点开始，一直递归左孩子，直到一个节点的左孩子为null。这样我们就找到了最小节点。同理，查询最大值也是一样的。该过程时间复杂性为$O(h)$

程序 14-4 二叉搜索树的查找

```
template<class K, class E>
pair<const K, E>* binarySearchTree<K,E>::find(const K& theKey) const
{// 返回值是匹配数对的指针
 // 如果没有匹配的数对，返回值为 NULL
     // p 从根节点开始搜索，寻找关键字等于 theKey 的一个元素
    binaryTreeNode<pair<const K, E> > *p = root;
    while (p != NULL)
      // 检查元素 p->element
      if (theKey < p->element.first)
         p = p->leftChild;
      else
         if (theKey > p->element.first)
            p = p->rightChild;
         else                        // 找到匹配的元素
            return &p->element;

   // 无匹配的数对
   return NULL;
}
```

14.2.2 删除

对删除来说，我们需要考虑包含被删除元素的节点p的四种情况：

- 要删除的节点无孩子节点
- 要删除的节点只有左孩子节点
- 要删除的节点只有右孩子节点
- 要删除的节点有左、右孩子节点

对于第一种情况，我们无需过多考虑，只需要把它的父节点的左孩子域置为零，然后删除该节点即可

至于剩下的三种：

- 第二种情况：删除该节点且使被删除节点的双亲结点指向被删除结点的左孩子结点
- 第三种情况：删除该节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
- 第四种情况：在它的右子树中寻找中序下的第一个结点（关键码最小），用它的值填补到被删除结点中，再来处理该结点的删除问题 / 右子树中找最小值，左子树中找最大值

下面是具体的代码实现：

程序 14-6 二叉搜索树的删除

```
template<class K, class E>
void binarySearchTree<K,E>::erase(const K& theKey)
{// 删除其关键字等于 theKey 的数对

   // 查找关键字为 theKey 的节点
   binaryTreeNode<pair<const K, E> > *p = root,
                                     *pp = NULL;

   while (p != NULL && p->element.first != theKey)
   {// p 移到它的一个孩子节点
      pp = p;
      if (theKey < p->element.first)
         p = p->leftChild;
      else
         p = p->rightChild;
   }
   if (p == NULL)
      return; // 不存在与关键字 theKey 匹配的数对

   // 重新组织树结构
   // 当 p 有两个孩子时的处理
   if (p->leftChild != NULL && p->rightChild != NULL)
   {// 两个孩子
      // 转化为空或只有一个孩子
      // 在 p 的左子树中寻找最大元素
      binaryTreeNode<pair<const K, E> > *s = p->leftChild,
                                        *ps = p;        // s 的双亲
      while (s->rightChild != NULL)
      {// 移到最大的元素
         ps = s;
         s = s->rightChild;
      }

      // 将最大元素 s 移到 p, 但不是简单的移动
      // p->element = s->element , 因为 key 是常量
      binaryTreeNode<pair<const K, E> > *q =
         new binaryTreeNode<pair<const K, E> >
             (s->element, p->leftChild, p->rightChild);
      if (pp == NULL)
         root = q;
      else if (p == pp->leftChild)
              pp->leftChild = q;
           else
              pp->rightChild = q;
      if (ps == p) pp = q;
      else pp = ps;
      delete p;
      p = s;
   }

   // p 最多有一个孩子
   // 把孩子指针存放在 c
   binaryTreeNode<pair<const K, E> > *c;
   if (p->leftChild != NULL)
      c = p->leftChild;
   else
      c = p->rightChild;

   // 删除 p
   if (p == root)
      root = c;
   else
   {// p 是 pp 的左孩子还是右孩子 ?
      if (p == pp->leftChild)
```

```
            pp->leftChild = c;
```

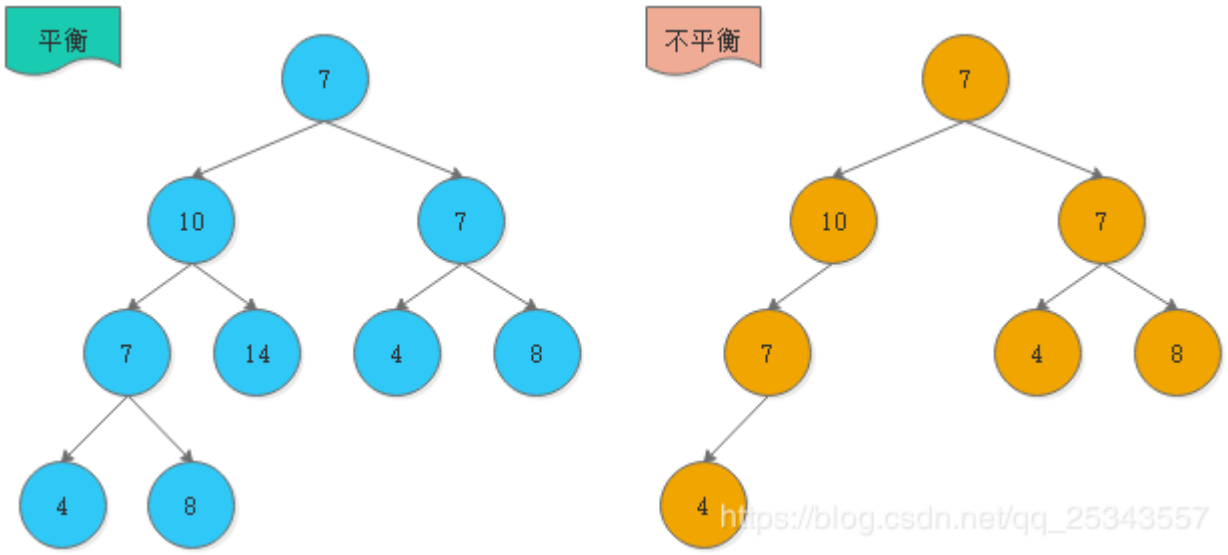
```
            else pp->rightChild = c;
        }
        treeSize--;
        delete p;
    }
```

14.2.3 二叉树的高度

二叉树的平均高度是$O(logn)$每一个树的平均操作时间也是$O(logn)$

14.3 AVL树

AVL树具有以下性质：
它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。



平衡因子
某结点的左子树与右子树的高度(深度)差即为该结点的平衡因子（BF,Balance Factor）。平衡二叉树上所有结点的平衡因子只可能是 -1，0 或 1。如果某一结点的平衡因子绝对值大于1则说明此树不是平衡二叉树。为了方便计算每一结点的平衡因子我们可以为每个节点赋予height这一属性，表示此节点的高度。

下面给出一个简单的AVL树实现

```
/**
 * AVLTree是BST，所以节点值必须是可比较的
```

```
 */
public class AvlTree<E extends Comparable<E>>{
    private class Node{
        public E e;
        public Node left;
        public Node right;
        public int height;

        public Node(E e){
            this.e = e;
            this.left = null;
            this.right = null;
            this.height = 1;
        }
    }

    private Node root;
    private int size;

    public AvlTree(){
        root=null;
        size=0;
    }

    //获取某一结点的高度
    private int getHeight(Node node){
        if(node==null){
            return 0;
        }
        return node.height;
    }

    public int getSize(){
        return size;
    }

    public boolean isEmpty(){
        return size == 0;
    }

    /**
     * 获取节点的平衡因子
     * @param node
     * @return
     */
    private int getBalanceFactor(Node node){
        if(node==null){
```

```
            return 0;
        }
        return getHeight(node.left)-getHeight(node.right);
    }

    //判断树是否为平衡二叉树
    public boolean isBalanced(){
        return isBalanced(root);
    }

    private boolean isBalanced(Node node){
        if(node==null){
            return true;
        }
        int balanceFactory = Math.abs(getBalanceFactor(node));
        if(balanceFactory>1){
            return false;
        }
        return isBalanced(node.left)&&isBalanced(node.right);
    }
}
```

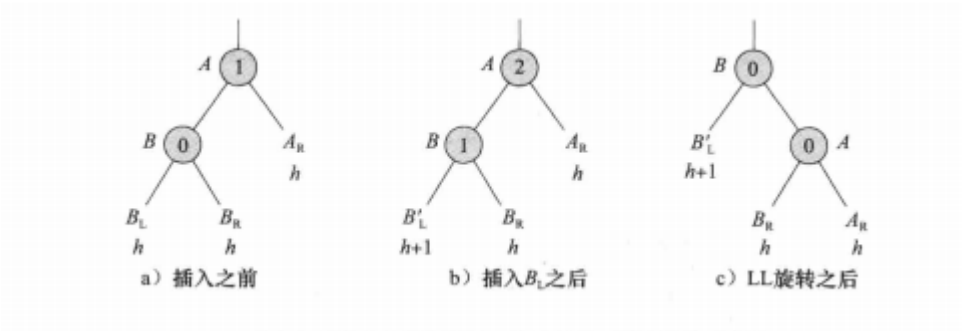
14.3.1 AVL树的相关操作

插入:

往平衡二叉树中添加节点很可能会导致二叉树失去平衡，所以我们需要在每次插入节点后进行平衡的维护操作。插入节点破坏平衡性有如下四种情况:

LL（右旋）

LL的意思是向左子树（L）的左孩子（L）中插入新节点后导致不平衡，这种情况下需要右旋操作。



LR

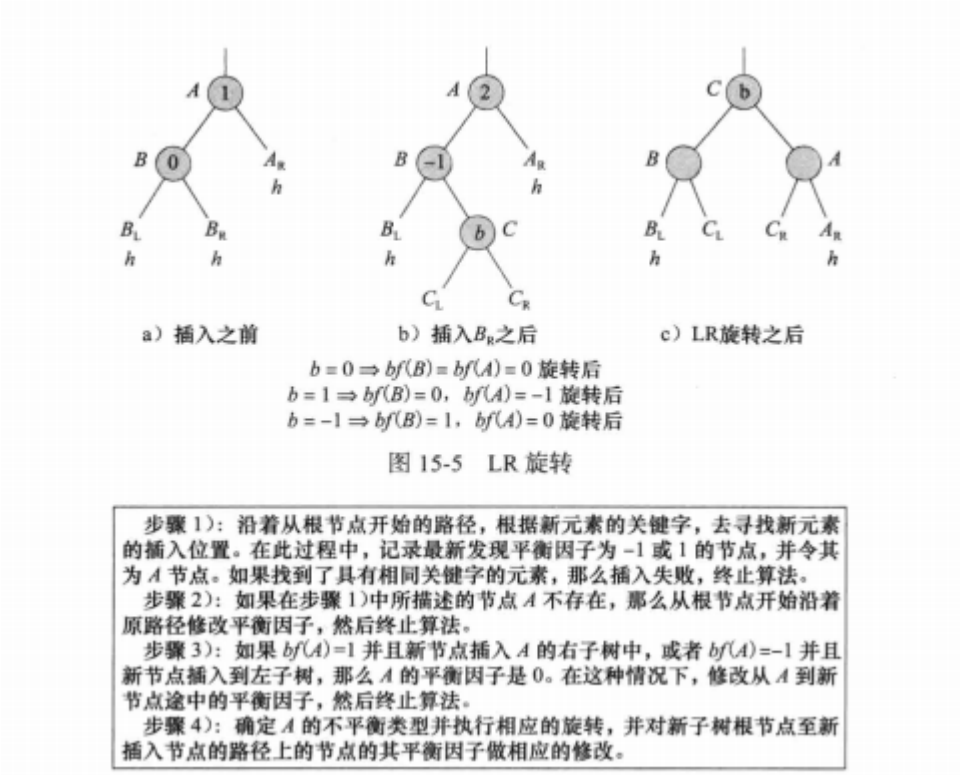


图 15-5　LR 旋转

步骤 1）：沿着从根节点开始的路径，根据新元素的关键字，去寻找新元素的插入位置。在此过程中，记录最新发现平衡因子为 -1 或 1 的节点，并令其为 A 节点。如果找到了具有相同关键字的元素，那么插入失败，终止算法。
步骤 2）：如果在步骤 1）中所描述的节点 A 不存在，那么从根节点开始沿着原路径修改平衡因子，然后终止算法。
步骤 3）：如果 $bf(A)=1$ 并且新节点插入 A 的右子树中，或者 $bf(A)=-1$ 并且新节点插入到左子树，那么 A 的平衡因子是 0。在这种情况下，修改从 A 到新节点途中的平衡因子，然后终止算法。
步骤 4）：确定 A 的不平衡类型并执行相应的旋转，并对新子树根节点至新插入节点的路径上的节点的其平衡因子做相应的修改。

删除

从AVL树中删除可以通过把要删除的节点向下旋转成一个叶子节点，接着直接剪除这个叶子节点来完成。因为在旋转成叶子节点期间最多有$logn$个节点被旋转，而每次 AVL 旋转耗费恒定的时间，删除处理在整体上耗费 $O(logn)$ 时间。我们可以把AVL树的删除操作理解为需要进行平衡维护工作的二叉搜索树的节点删除（如果二叉搜索树的删除也不会呢？）

欢迎查找往期关于二叉搜索树的知识见解

查找

在AVL树中查找同在一般BST完全一样的进行，所以耗费$O(logn)$ 时间，因为AVL树总是保持平衡的。不需要特殊的准备，树的结构不会由于查询而改变。（这是与伸展树查找相对立的，它会因为查找而变更树结构。）

下面是我在百度百科上看到的完整的各个操作执行代码（大家在遇到不会的难题的时候，百度也会是大家的好帮手哟）

```
#include <iostream>
#include <math.h>
#include <stdlib.h> //建立一个整数类型

using namespace std;
```

```
#define MAXSIZE 512

typedef struct obj_n_t {
    int obj_key;
} obj_node_t; //建立树结点的基本机构
typedef struct tree_n_t {
    int key;
    struct tree_n_t *left, *right;
    int height;
} tree_node_t; //建立堆栈

tree_node_t *stack[MAXSIZE]; //warning!the tree can contain 256 leaves at most!
int i = 0; //堆栈计数器 //堆栈清空
void stack_clear() {
    while (i != 0) {
        stack[i - 1] = NULL;
        i--;
    }
}

//堆栈为空
int stack_empty() {
    return (i == 0);
}

//入栈函数
int push(tree_node_t *node) {
    if (i < MAXSIZE) {
        stack[i++] = node;
        return (0);
    } else
        return (-1);
}

//出栈函数
tree_node_t *pop() {
    if (i > 0)
        return (stack[--i]);
    else
        return (0);
}

//建立get_node函数，用于动态分配内存空间
tree_node_t *get_node() {
    tree_node_t *tmp;
    tmp = (tree_node_t *) malloc(sizeof(tree_node_t));
    return (tmp);
```

```
}

//建立return_node函数，用于释放内存
void return_node(tree_node_t *free_node) {
    free(free_node);
}

//建立左旋转函数
void left_rotation(tree_node_t *node) {
    tree_node_t *tmp;
    int tmp_key;
    tmp = node->left;
    tmp_key = node->key;
    node->left = node->right;
    node->key = node->right->key;
    node->right = node->left->right;
    node->left->right = node->left->left;
    node->left->left = tmp;
    node->left->key = tmp_key;
}

//建立右旋转函数
void right_rotation(tree_node_t *node) {
    tree_node_t *tmp;
    int tmp_key;
    tmp = node->right;
    tmp_key = node->key;
    node->right = node->left;
    node->key = node->left->key;
    node->left = node->right->left;
    node->right->left = node->right->right;
    node->right->right = tmp;
    node->right->key = tmp_key;
}

int rebalance(tree_node_t *node) {
    int finished = 0;
    while (!stack_empty() && !finished) {
        int tmp_height, old_height;
        node = pop(); //back to the root along the search path
        old_height = node->height;
        if (node->left->height - node->right->height == 2) {
            if (node->left->left->height - node->right->height == 1) {
                right_rotation(node);
                node->right->height = node->right->left->height + 1;
                node->height = node->right->height + 1;
            } else {
```

```
                left_rotation(node->left);
                right_rotation(node);
                tmp_height = node->left->left->height;
                node->left->height = tmp_height + 1;
                node->right->height = tmp_height + 1;
                node->height = tmp_height + 2;
            }
        } else if (node->left->height - node->right->height == -2) {
            if (node->right->right->height - node->left->height == 1) {
                left_rotation(node);
                node->left->height = node->left->right->height + 1;
                node->height = node->left->height + 1;
            } else {
                right_rotation(node->right);
                left_rotation(node);
                tmp_height = node->right->right->height;
                node->left->height = tmp_height + 1;
                node->right->height = tmp_height + 1;
                node->height = tmp_height + 2;
            }
        } else {
            if (node->left->height > node->right->height)
                node->height = node->left->height + 1;
            else
                node->height = node->right->height + 1;
        }
        if (node->height == old_height)
            finished = 1;
    }
    stack_clear();
    return (0);
}

//建立creat_tree函数，用于建立一颗空树
tree_node_t *creat_tree() {
    tree_node_t *root;
    root = get_node();
    root->left = root->right = NULL;
    root->height = 0;
    return (root); //build up an empty tree.the first insert bases on the empty
}

//建立find函数，用于查找一个对象
obj_node_t *find(tree_node_t *tree, int query_key) {
    tree_node_t *tmp;
    if (tree->left == NULL)
        return (NULL);
```

```
    else {
        tmp = tree;
        while (tmp->right != NULL) {
            if (query_key < tmp->key)
                tmp = tmp->left;
            else
                tmp = tmp->right;
        }
        if (tmp->key == query_key)
            return ((obj_node_t *) tmp->left);
        else
            return (NULL);
    }
}

//建立插入函数
int insert(tree_node_t *tree, obj_node_t *new_obj) {
    tree_node_t *tmp;
    int query_key, new_key;
    query_key = new_key = new_obj->obj_key;
    if (tree->left == NULL) {
        tree->left = (tree_node_t *) new_obj;
        tree->key = new_key;
        tree->height = 0;
        tree->right = NULL;
    } else {
        stack_clear();
        tmp = tree;
        while (tmp->right != NULL) {
//use stack to remember the path from root to the position at which the new obje
//then after inserting,we can rebalance from the parrent node of the leaf which
            push(tmp);
            if (query_key < tmp->key)
                tmp = tmp->left;
            else
                tmp = tmp->right;
        }
        if (tmp->key == query_key)
            return (-1);
        else {
            tree_node_t *old_leaf, *new_leaf;
//It must allocate 2 node space in memory.
//One for the new one,another for the old one.
//the previous node becomes the parrent of the new node.
//when we delete a leaf,it will free two node memory spaces as well.
            old_leaf = get_node();
            old_leaf->left = tmp->left;
```

```
            old_leaf->key = tmp->key;
            old_leaf->right = NULL;
            old_leaf->height = 0;
            new_leaf = get_node();
            new_leaf->left = (tree_node_t *) new_obj;
            new_leaf->key = new_key;
            new_leaf->right = NULL;
            new_leaf->height = 0;
            if (tmp->key < new_key) {
                tmp->left = old_leaf;
                tmp->right = new_leaf;
                tmp->key = new_key;
            } else {
                tmp->left = new_leaf;
                tmp->right = old_leaf;
            }
            tmp->height = 1;
        }
    }
    rebalance(tmp);
    return (0);
}

//建立删除函数
int del(tree_node_t *tree, int key) {
    tree_node_t *tmp, *upper, *other;
    if (tree->left == NULL)
        return (-1);
    else if (tree->right == NULL) {
        if (tree->key == key) {
            tree->left = NULL;
            return (0);
        } else
            return (-1);
    } else {
        tmp = tree;
        stack_clear();
        while (tmp->right != NULL) {
            upper = tmp;
            push(upper);
            if (key < tmp->key) {
                tmp = upper->left;
                other = upper->right;
            } else {
                tmp = upper->right;
                other = upper->left;
            }
```

```
        }
        if (tmp->key != key)
            return (-1);
        else {
            upper->key = other->key;
            upper->left = other->left;
            upper->right = other->right;
            upper->height = upper->height - 1;
            return_node(tmp);
            return_node(other);
            rebalance(pop());
//here must pop,then rebalance can run from the parrent of upper,because upper h
            return (0);
        }
    }
}

//建立测试遍历函数
int travel(tree_node_t *tree) {
    stack_clear();
    if (tree->left == NULL)
        push(tree);
    else if (tree->left != NULL) {
        int m = 0;
        push(tree);
        while (i != m) {
            if (stack[m]->left != NULL && stack[m]->right != NULL) {
                push(stack[m]->left);
                push(stack[m]->right);
            }
            m++;
        }
    }
    return (0);
}

//建立测试函数
int test_structure(tree_node_t *tree) {
    travel(tree);
    int state = -1;
    while (!stack_empty()) {
        --i;
        if (tree->right == NULL && tree->height == 0)//this statement is leaf,bu
            state = 0;
        else if (tree->left != NULL && tree->right != NULL) {
            if (abs(tree->height - tree->height) <= 1)
                state = 0;
```

```
            else {
                state = -1;
                stack_clear();
            }
        }
    }
    stack_clear();
    return (state);
}

//建立remove_tree函数
int remove_tree(tree_node_t *tree) {
    travel(tree);
    if (stack_empty())
        return (-1);
    else {
        while (!stack_empty()) {
            return_node(pop());
        }
        return (0);
    }
}

int main() {
    tree_node_t *atree = NULL;
    obj_node_t obj[256], *f; //MAXSIZE=n(number of leaf)+(n-1) number of node
    int n, j = 0;
    cout << "Now Let's start this program! There is no tree in memory.\n";
    int item;
    while (item != 0) {
        cout << "\nRoot address = " << atree << "\n";
        cout << "\n1.Create a tree\n";
        cout << "\n2.Insert a int type object\n";
        cout << "\n3.Test the structure of the tree\n";
        cout << "\n4.Find a object\n";
        cout << "\n6.Delete a object\n";
        cout << "\n7.Remove the Tree\n";
        cout << "\n0.Exit\n";
        cout << "\nPlease select:";
        cin >> item;
        cout << "\n\n\n";
        switch (item) {
            case 1: {
                atree = creat_tree();
                cout << "\nA new empty tree has been built up!\n";
                break;
            }
```

```
case 2: {
    if (atree != NULL)
        while (n != 3458) {
            cout << "Please insert a new object.\nOnly one object ev
            cin >> n;
            if (n != 3458) {
                obj[j].obj_key = n;
                if (insert(atree, &obj[j]) == 0) {
                    j++;
                    cout << "Integer " << n << " has been input!\n\r
                } else
                    cout << "\n\nInsert failed!\n\n";
            }
        }
    else
        cout << "\n\nNo tree in memory,insert Fail!\n\n";
    break;
}
case 3: {
    if (atree != NULL) {
        n = test_structure(atree);
        if (n == -1)
            cout << "\n\nIt's not a correct AVL tree.\n\n";
        if (n == 0)
            cout << "\n\nIt's a AVL tree\n\n";
    } else
        cout << "\n\nNo tree in memory,Test Fail!\n\n";
    break;
}
case 4: {
    if (atree != NULL) {
        cout << "\n\nWhat do you want to find? : ";
        cin >> n;
        f = find(atree, n);
        if (f == NULL) {
            cout << "\n\nSorry," << n << " can't be found!\n\n";
        } else {
            cout << "\n\nObject " << f->obj_key << " has been found!
        }
    } else
        cout << "\n\nNo tree in memory,Find Fail!\n\n";
    break;
}
case 5: {
    if (atree != NULL) {
        travel(atree);
        for (int count = 0; count < i; count++) {
```

```
                        cout << " " << stack[count]->key << ",";
                    }
                } else
                    cout << "\n\nNo tree in memory,Travel Fail!\n\n";
                break;
            }
            case 6: {
                if (atree != NULL) {
                    cout << "\n\nWhich object do you want to delete?\n\n";
                    cin >> n;
                    if (del(atree, n) == 0) {
                        cout << "\n\n" << n << " has been deleted!\n\n";
                    } else
                        cout << "\n\nNo this object\n\n";
                } else
                    cout << "\n\nNo tree in memory,Delete Fail!\n\n";
                break;
            }
            case 7: {
                if (atree != NULL) {
                    remove_tree(atree);
                    cout << "\n\nThe Tree has been removed!\n\n";
                    atree = NULL;
                } else
                    cout << "\n\nNo tree in memory,Removing Fail!\n\n";
            }
            default:
                cout << "\n\nNo this operation!\n\n";
        }
        n = 0;
    }
}
```

14.4 B-树

B树是一种平衡的多路搜索树， 结点最大的孩子数目称为B树的阶。一个m阶B树具有如下属性:

1.定义任意非叶子结点最多只有M个儿子；且M>2；

2.根结点的儿子数为[2, M]（即：如果根节点不是叶节点，则最少有两个孩子）；

3.除根结点以外的非叶子结点的儿子数为[M/2, M]；

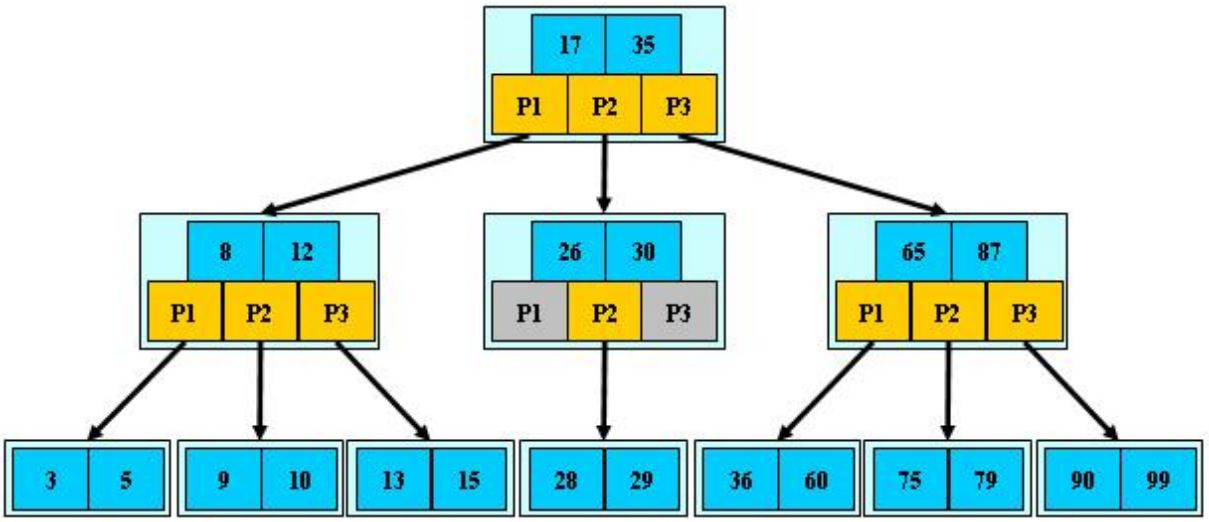
4.每个结点存放至少M/2-1（取上整）和至多M-1个关键字；（至少2个关键字）

5.非叶子结点的关键字个数=指向儿子的指针个数-1；

6.非叶子结点的关键字：K[1]，K[2]，…，K[M-1]；且K[i] < K[i+1]；

7.非叶子结点的指针：P[1]，P[2]，…，P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指

8.所有叶子结点位于同一层；



B-树的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；

B-树的特性：

1.关键字集合分布在整颗树中；

2.任何一个关键字出现且只出现在一个结点中；

3.搜索有可能在非叶子结点结束；

4.其搜索性能等价于在关键字全集内做一次二分查找；

5.自动层次控制；

B-树的高度：

设T是一棵高度为h的m序B-树，d=[m/2]且n是T中的元素个数，则：

$$2d^{h-1} - 1 \leq n \leq m^{h-1}$$

$$log_m(n+1) \le h \le log_d(n+1/2)+1$$

B-树的搜索：

B-树的搜索算法与m叉搜索树的搜索算法相同。在搜索过程中，从根至外部节点上的所有内部节点都有可能被搜索到，因此，磁盘访问次数最多是h。

B-树的插入:

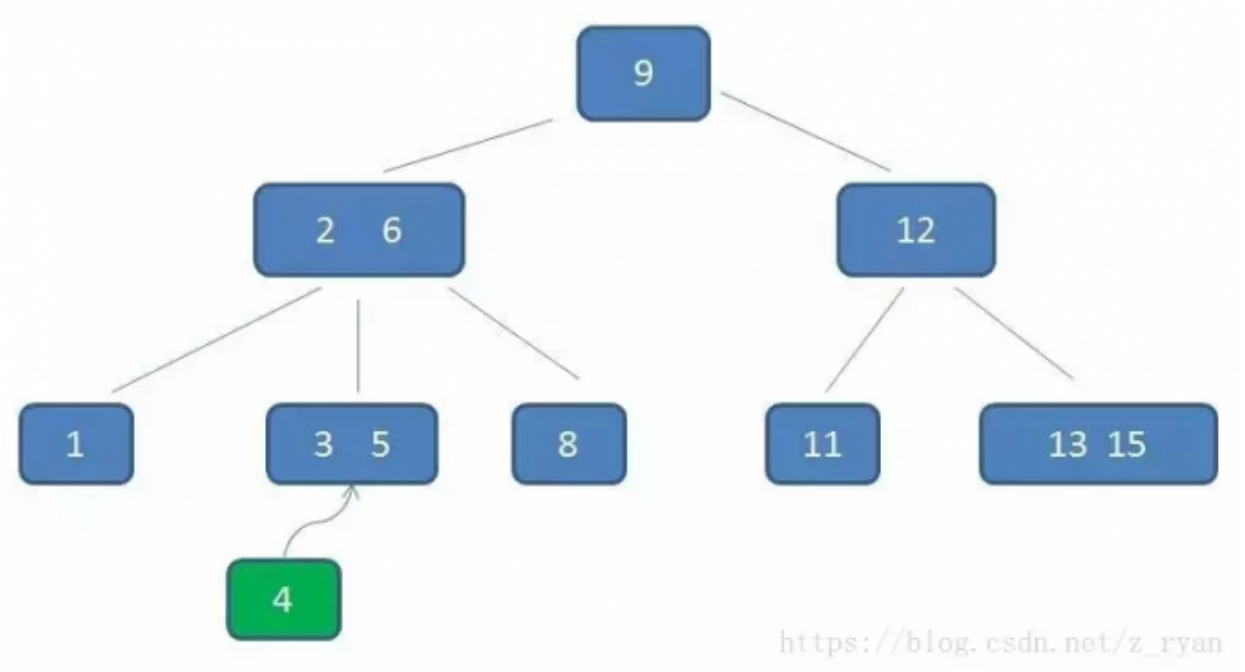
对高度为 k 的m阶B树，新结点一般是插在叶子层。通过检索可以确定关键码应插入的结点位置。然后分两种情况讨论：

1、 若该结点中关键码个数小于m-1，则直接插入即可。

2、 若该结点中关键码个数等于m-1，则将引起结点的分裂。以中间关键码为界将结点一分为二，产生一个新结点，并把中间关键码插入到父结点(k -1层)中

重复上述工作，最坏情况一直分裂到根结点，建立一个新的根结点，整个B树增加一层。

例如：在下面的B树中插入key：4



第一步：检索key插入的节点位置如上图所示，在3,5之间;

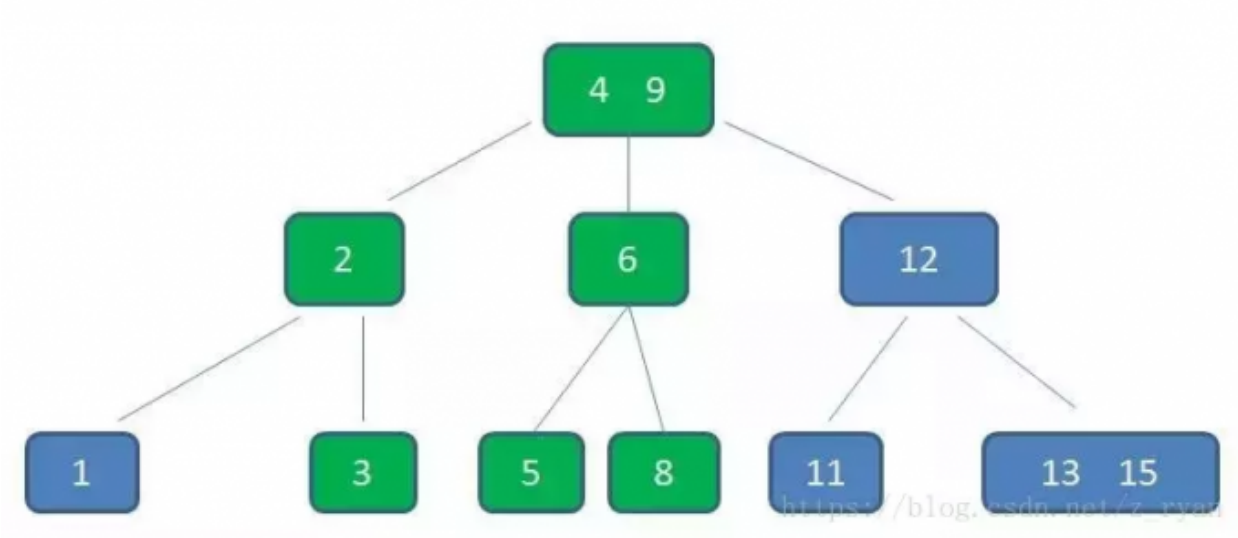
第二步：判断节点中的关键码个数:

节点3，5已经是两元素节点，无法再增加（已经 = 3-1）。父亲节点 2， 6 也是两元素节点，也无法再增加。根节点9是单元素节点，可以升级为两元素节点。；

第三步：结点分裂：

拆分节点3，5与节点2，6，让根节点9升级为两元素节点4，9。节点6独立为根节点的第二个孩子。

最终结果如下图：虽然插入比较麻烦，但是这也能确保 B 树是一个自平衡的树。



B-树的删除

删除操作是指，根据key删除记录，如果B树中的记录中不存对应key的记录，则删除失败。

1）如果当前需要删除的key位于非叶子结点上，则用后继key（这里的后继key均指后继记录的意思）覆盖要删除的key，然后在后继key所在的子支中删除该后继key。此时后继key一定位于叶子结点上，这个过程和二叉搜索树删除结点的方式类似。删除这个记录后执行第2步

2）该结点key个数大于等于Math.ceil(m/2)-1，结束删除操作，否则执行第3步。

3）如果兄弟结点key个数大于Math.ceil(m/2)-1，则父结点中的key下移到该结点，兄弟结点中的一个key上移，删除操作结束。

否则，将父结点中的key下移与当前结点及它的兄弟结点中的key合并，形成一个新的结点。原父结点中的key的两个孩子指针就变成了一个孩子指针，指向这个新结点。然后当前结点的指针指向父结点，重复上第2步。

有些结点它可能即有左兄弟，又有右兄弟，那么我们任意选择一个兄弟结点进行操作即可。

第16章 图

本章概述

图可以解决的问题很多，本章主要包括无向图、有向图和加权图得到创建及其插入删除等操作，以及图的描述方法(邻接矩阵、矩阵邻接表和邻接链表)、遍历方法（广度优先搜索、深度优先搜索）和相关简单算法（寻找图的路径、寻找无向图的连通构件和寻找连通无向图的生成树）

16.1 基本概念

图是有限集V和E的有序对，即**G=(V,E)**

（1）其中V的元素称为**顶点**（也称**节点(nodes)或点(points)**）

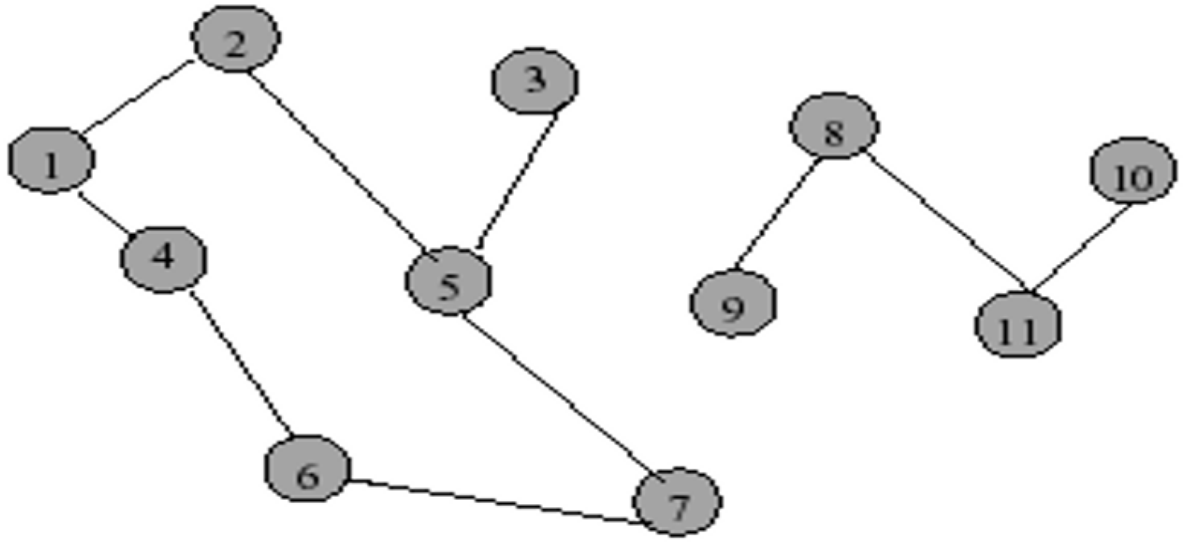
（2）E中的每一条**边**连接V中两个不同的顶点。边也叫作**弧**(arcs)或**连线**(lines) 。可以用(i.j)，其中*i*和*j*是*E*所连接的两个顶点

无向边(undirected edge): (*i*,*j*)和(*j*,i)是一样的

有向边(directed edge): (*i*,*j*)和(*j*,*i*)是不同的

通常把无向图称为图，有向图称为有向图

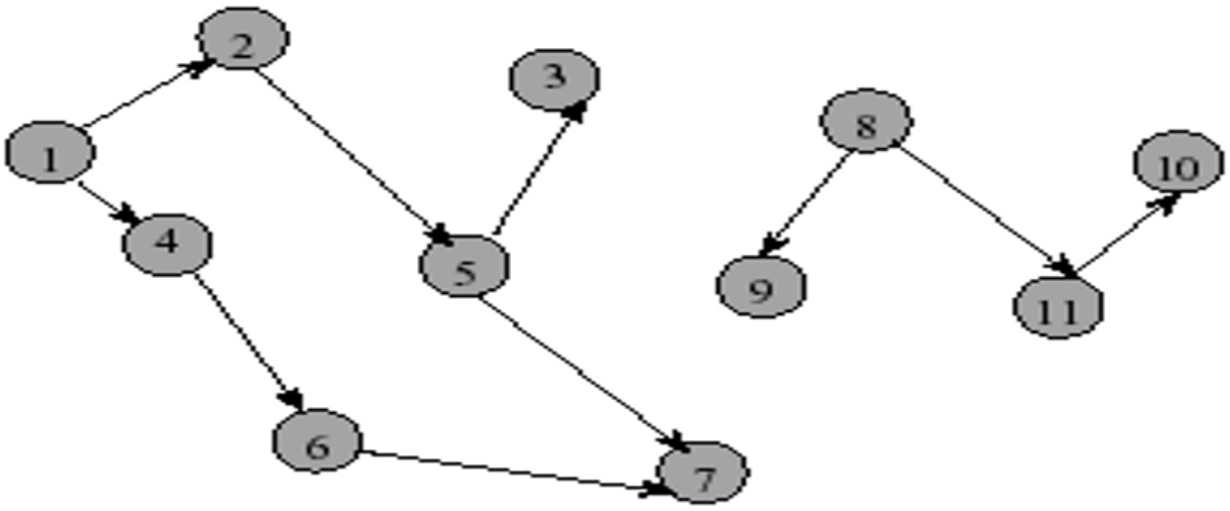
无向图中所有边都为无向边



当(i,j)是图中的边时，顶点i和j是**邻接的**(adjacent)。(j是i的**邻接点；** i是j的邻接点)

边(i,j)**关联于**(incident)顶点i和j

有向图中所有边都为有向边



有向边(i, j) 是关联至(incident to)顶点j 而关联于(incident from)顶点i。
顶点i邻接至(adjacent to)顶点j，顶点j邻接于(adjacent from)顶点i。

网络(Network):加权有向图(Weight digraph)或加权无向图(Weight graph)，每一条边都赋予一个权值或耗费。

所有图都可以看作网络的一种特殊情况：一个无向(有向)图可以被看作是一个所有边具有相同权的无向(有向)网络。

16.2 应用和更多概念

1.路径问题

路径：当且仅当对于每一个j(1≤j≤k)，边(i_j,i_{j+1})都在E中时，顶点序列P=i_1, i_2, i_3., …i_k,是图或有向图G= (V, E)中一条从i_1到i_k 的**路径。**

简单路径：除第一个和最后一个顶点以外，路径中其他所有顶点均不同

路径的长度：路径上所有边的长度之和

2.生成树

连通图：设*G*= (*V*,*E*)是一个无向图，当且仅当*G*中每一对顶点之间有一条路径时，可认为*G*是连通图(Connected Graph)

H是图G的子图(subgraph)的充要条件是，H的顶点和边的集合是G的顶点和边的集合的子集

环路(Cycle/回路):起始节点与结束节点是同一节点的简单路径

环路(Cycle/回路):起始节点与结束节点是同一节点的简单路径

树(Tree):**没有环路**的**无向连通图**是一棵树

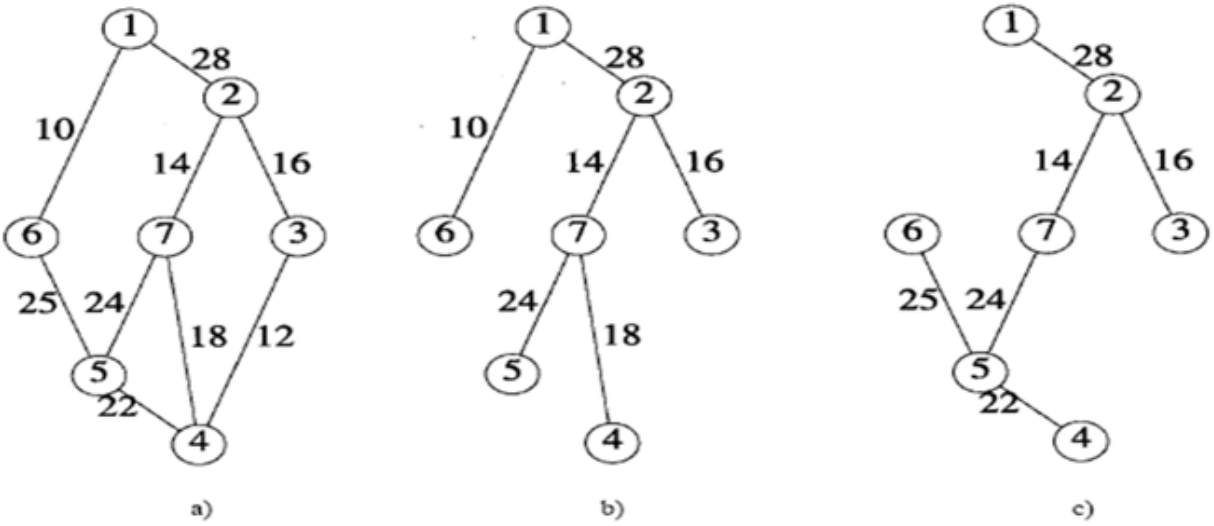
树的耗费(cost of tree/树的代价)： 树的耗费是所有边的耗费(weights/costs)之和

图G的生成树(Spanning Tree of G):一棵包含G中所有顶点并且是G的子图的树是G的生成树

（1）一个n节点的连通图必须至少有n-1条边

（2）如果图G有 n 个顶点, 那么图G的生成树有 n 个顶点和有n-1条边

生成树示例：



图a的两棵生成树:图b和图c

图b生成树耗费 = 100;.图c生成树耗费 = 129

最小耗费生成树（最小代价生成树）： 生成树耗费（代价）达到最小的生成树

16.3 特性

度：设G是一个**无向图**，**顶点i的度**(degree)di是与顶点i相连的边的个数

完全(无向)图：一个具有n 个顶点， n(n-1)/2 条边的图是一个完全(无向)图

入度：设*G*是一个**有向图**，**顶点**i的入度**(in-degree) **d_i^{in} 是指关联至顶点i的边的数量

出度：顶点i **的出度**(out-degree) d_i^{out} 是指关联于该顶点的边的数量

完全有向图：一个具有n个顶点,，n(n-1)条边的图是一个完全有向图

16.3.1特性一

设G=(V,E) 是一无向图.令 |V|=n; |E| =e; di=顶点i的度，则

（1）

$$\sum_{i=1}^{n} d_i = 2e$$

(2)

$$0 \leq e \leq n(n-1)/2$$

16.3.2特性二

设G=(V,E) 是一有向图.令|V|=n; |E| =e

(1)

$$0 \leq e \leq n(n-1)$$

(2)

$$\sum_{i=1}^{n} d_i^{in} = \sum_{i=1}^{n} d_i^{out} = e$$

16.4 抽象数据类型graph

graph：无向图、有向图、加权无向图、加权有向图

```
抽象数据类型  graph
{    实例
         顶点集合V和边集合E。
     操作:
     numberOfVertices(): 返回图中顶点的数目
     numberOfEdges(): 返回图中边的数目
     ExistsEdge (i,j): 如果边(i,j)存在,返回true，否则返回false
     insertEdge (theEdge): 向图中添加边theEdge
     eraseEdge (i,j): 删除边(i,j)
     degree(i): 返回顶点i的度，只能用于无向图
     inDegree(i): 返回顶点i的入度
     outDegree(i): 返回顶点i的出度
}
```

```
template <class T>
class  graph
{public:
     ……
     //ADT方法操作:
     virtual int numberOfVertices() const=0;
     virtual int numberOfEdges() const=0;
     virtual bool existsEdge (int,int) const=0;
     virtual void insertEdge (edge<T> *) const=0;
     virtual void eraseEdge (int,int) const=0;
     virtual int degree(int) const=0;
```

```
    virtual int inDegree(int) const=0;
    virtual int outDegree(int) const=0;
    //其他方法
    virtual bool directed() const=0;//当且仅当是有向图时，返回值是true
    virtual bool weighted() const=0; //当且仅当是加权图时，返回值是true
    virtual vertexIterator<T>* iterator(int) = 0;//访问指定顶点的邻接顶点
}
```

16.5无权图的描述

16.5.1 邻接矩阵

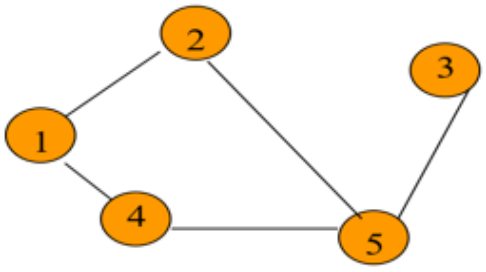
无权图G=(V,E)，令|V|=n; 假设： V={1,2,3,…,n}

G的邻接矩阵：n×n矩阵A

1.G为无向图时

$$A(i,j)\begin{cases}1 & \text{如果}(i,j)\in E\text{或者}(j,i)\in E\\ 0 & \text{其他}\end{cases}$$

对于n顶点的无向图，有



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

(1)

$$A(i,i)=0, 1\le i\le n$$

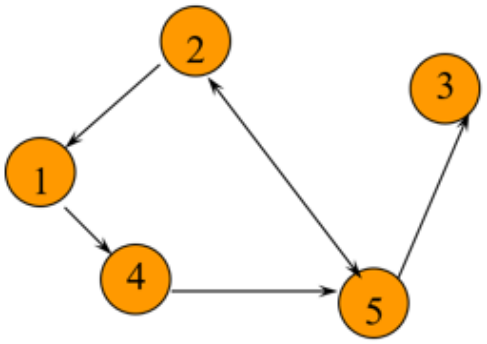
(2)

$$\text{邻接矩阵是对称的，即}(A(i,j)=A(j,i)，\ 1\le i\le n，\ 1\le j\le n)$$

(3)

$$\sum_{j=1}^{n}A(i,j)=\sum_{j=1}^{n}A(j,i)=d_i$$

2.G为有向图时



	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	1
3	0	0	0	0	0
4	0	0	0	0	1
5	0	1	1	0	0

$$A(i,j)\begin{cases}1, & \text{如果}(i,j)\in E\\ 0, & \text{其他}\end{cases}$$

$$\sum_{j=1}^{n}A(i,j)=d_i^{out},\sum_{j=1}^{n}A(j,i)=d_i^{in};1\le i\le n$$

3.邻接矩阵的存储

（1）使用(n+1)×(n+1)的布尔型数组a，映射A(i,j)=1，当且仅当a[i] [j]=true，需要$(n+1)^2$字节空间

（2）减少存储空间方法：

采用n×n数组a[n] [n]，映射A(i,j)=1，当且仅当a[i-1] [j-1]=true，需要n^2字节，比前一种减少了2n+1个字节

不储存所有对角线元素(都是零)，减少n个字节空间

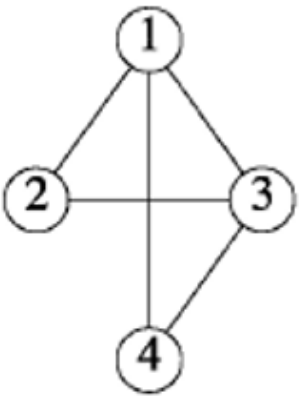
以上减少存储空间的办法，代码容易出错，某些操作代价大.

（3）对无向图，只需要存储上三角(或下三角)的元素

16.5.2 邻接链表

顶点i的邻接表(adjacency list)：是一个邻接于顶点i 的顶点的线性表,包含了顶点i的所有邻接点

1.无向图邻接链表

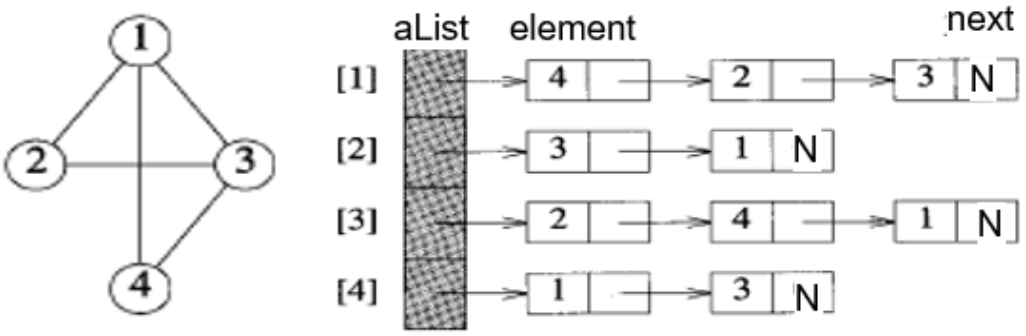


顶点1的邻接表 = (2，3,4)

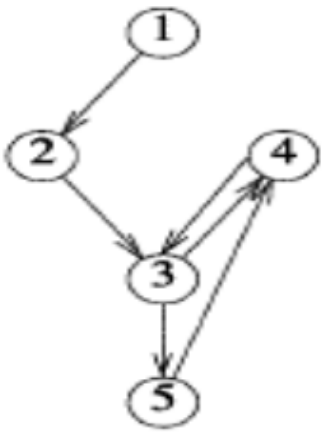
顶点2的邻接表 = (1,3)

顶点3的邻接表 = (1，2，4)

顶点4的邻接表 = (1,3)



2.有向图邻接链表



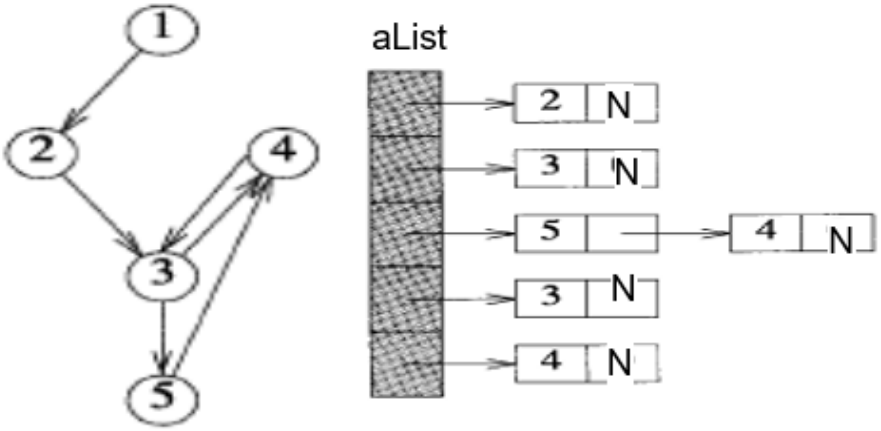
顶点1的邻接表 = (2)

顶点2的邻接表 = (3)

顶点3的邻接表 = (5,4)

顶点4的邻接表 = (3)

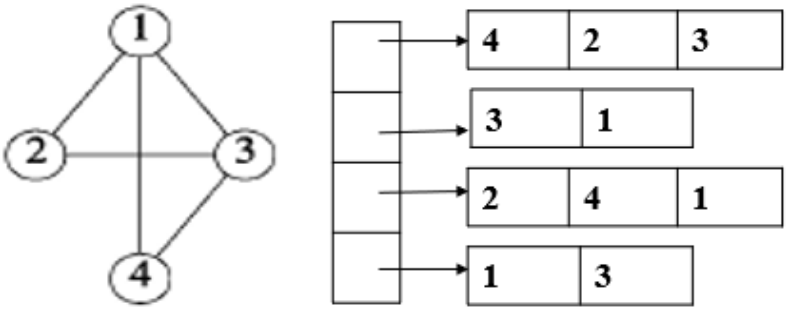
顶点5的邻接表 = (4)



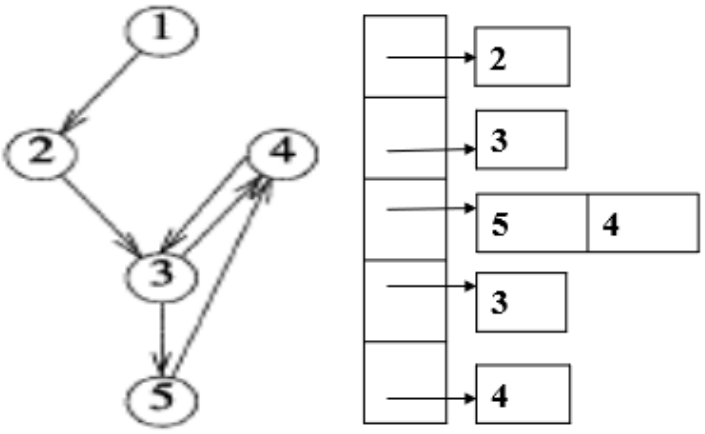
16.5.3 邻接数组

在邻接数组中，每个顶点的邻接表使用一个数组描述

1.无向图邻接链表



2.有向图邻接链表

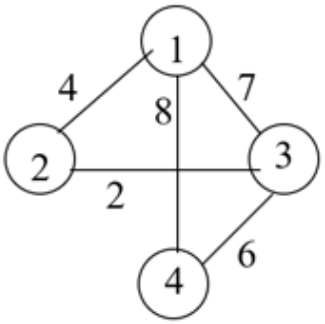


16.6 加权图的描述

1.G是一加权无向图

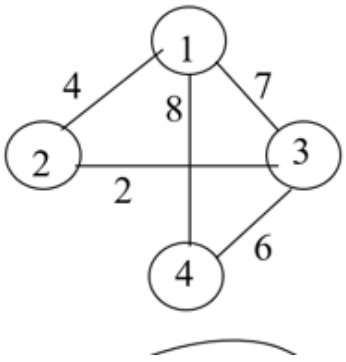
（1）耗费/代价/成本(cost)邻接矩阵：

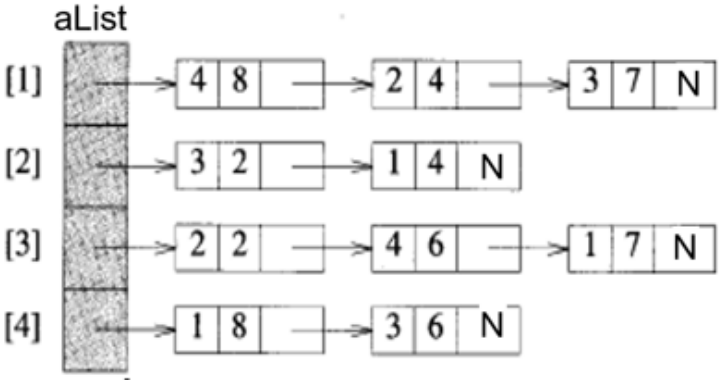
$$C(i,j)\begin{cases}\text{边}(i,j)\text{的耗费(或权)} & \text{如果}(i,j)\in E\text{或}(j,i)\in E\\ -\text{或}\infty & \text{其他}\end{cases}$$



	1	2	3	4
1	∞	4	7	8
2	4	∞	2	∞
3	7	2	∞	6
4	8	∞	6	∞

（2）加权图邻接链表描述

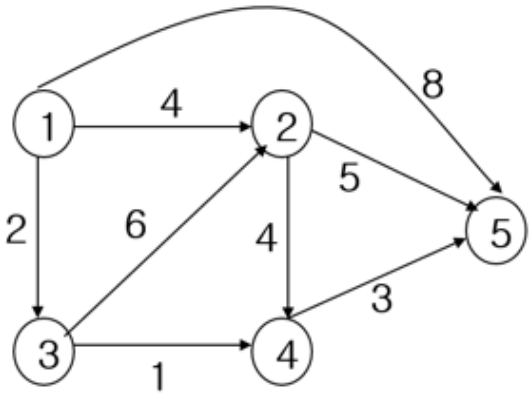




2.G是一加权有向图

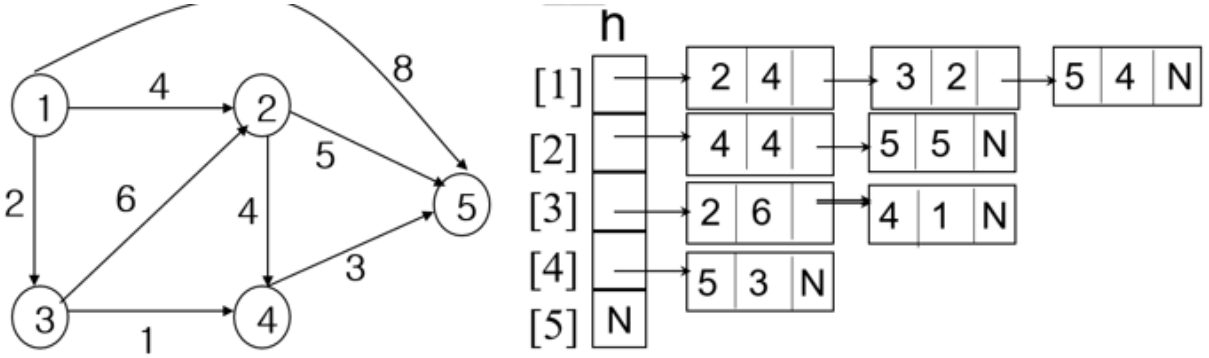
（1）耗费/代价/成本(cost)邻接矩阵：

$$C(i,j)\begin{cases}\text{边}(i,j)\text{的耗费(或权)} & \text{如果}(i,j)\in E\\ -\text{或}\infty & \text{其他}\end{cases}$$



$$\begin{bmatrix}\infty & 4 & 2 & \infty & 8\\ \infty & \infty & \infty & 4 & 5\\ \infty & 6 & \infty & 1 & \infty\\ \infty & \infty & \infty & \infty & 3\\ \infty & \infty & \infty & \infty & \infty\end{bmatrix}$$

（2）加权图邻接链表描述



16.7 类实现

16.7.1 不同的类

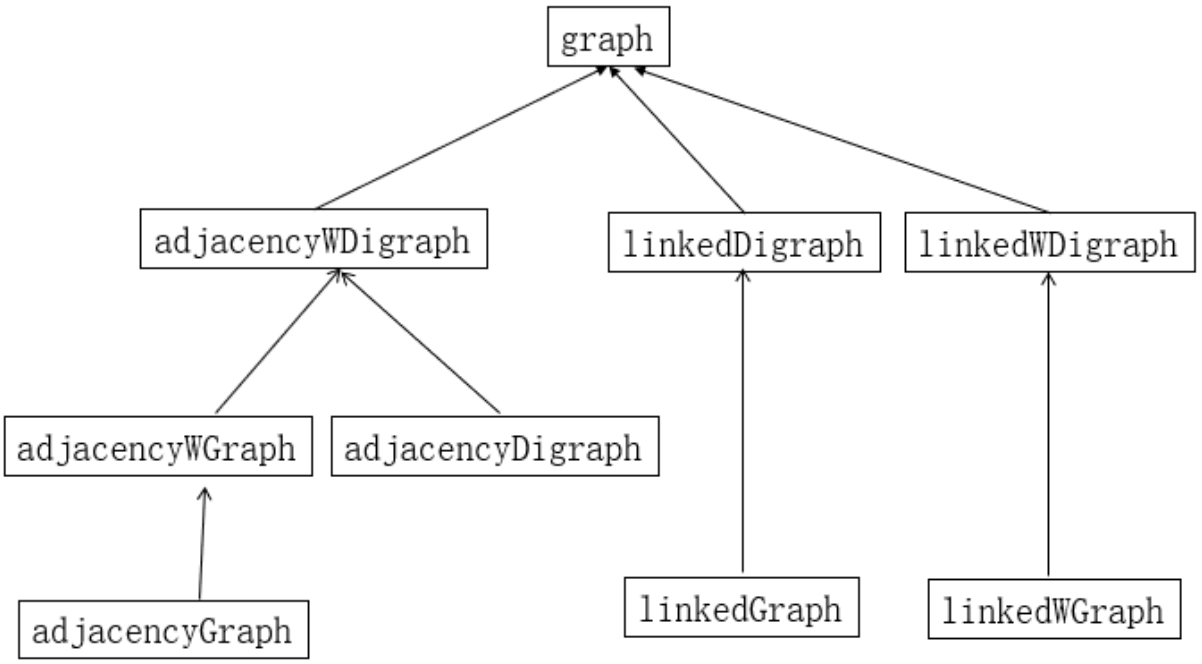
图的描述：邻接矩阵，邻接链表，邻接数组

图的类型：有向图、无向图，加权有向图、加权无向图

主要学习邻接矩阵和邻接链表的各自四种类型的图

无权有向图和无向图可以看作每条边的权是1的加权有向图和无向图

无向图,边(i, j)存在可以看作边(i, j) 和边(j,i) 都存在的有向图



16.7.2 邻接矩阵类

```
template <class T>
class adjacencyWDigraph : public graph<T>
{
    int n;          //顶点数目
    int e;          //边数
    T **a;          //邻接数组
    T noEdge;       //表示不存在的边
    adjacencyWDigraph(int numberOfVertices=0, T theNoEdge=0){//构造函数.
        //确认顶点数的合法性
        if (numberOfVertices < 0)  throw    ……..
        n = numberOfVertices;
        e = 0;
        noEdge = theNoEdge;
        make2dArray(a, n + 1, n + 1);
        for (int i = 1; i <= n; i++)// 初始化邻接矩阵
            fill(a[i], a[i] + n + 1, noEdge);
    }
    ~adjacencyWDigraph() {delete2dArray(a, n + 1);}
    void insertEdge(edge<T> * theEdge){// 在图中插入边theEdge;如果该边已经存在,
        //则theEdge中的权值更新该边权值
        int v1 = theEdge->vertex1();
        int v2 = theEdge->vertex2();
```

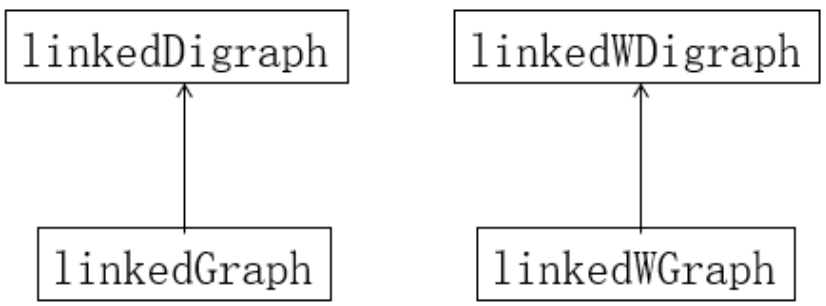
```
        if (v1 < 1 || v2 < 1 || v1 > n || v2 > n || v1 == v2){
            ……//输出出错信息,抛出异常
        }
        if (a[v1][v2] == noEdge)//新的边
            e++;
        a[v1][v2] = theEdge->weight();
    }
    void insertEdge(edge<T> * theEdge){
        int v1 = theEdge->vertex1();
        int v2 = theEdge->vertex2();
        ……..//确认参数有效性
        if (a[v1][v2] == noEdge)
            e++;
        a[v1][v2] = theEdge->weight();
        a[v2][v1] = theEdge->weight();
    }
    void eraseEdge(int i, int j){//删除边(i,j).
        if (i >= 1 && j >= 1 && i <= n && j <= n && a[i][j] != noEdge){
            a[i][j] = noEdge;
            a[j][i] = noEdge;
            e--;
        }
    }
}
```

16.7.3 扩充chain类

增加了eraseElement(theVertex)：删除顶点等于theVertex的元素

16.7.4 链表类

邻接链表描述的无向图(linkedGraph)
邻接链表描述的加权无向图(linkedWGraph)
邻接链表描述的有向图(linkedDigraph)
邻接链表描述的加权有向图(linkedWDigraph)



```
class linkedDigraph : public graph<bool>{
    protected:
        int n;                    // 顶点数目
        int e;                    // 边数目
        graphChain<int> *aList; // 邻接链表
        ……
}
```

16.8 图的遍历

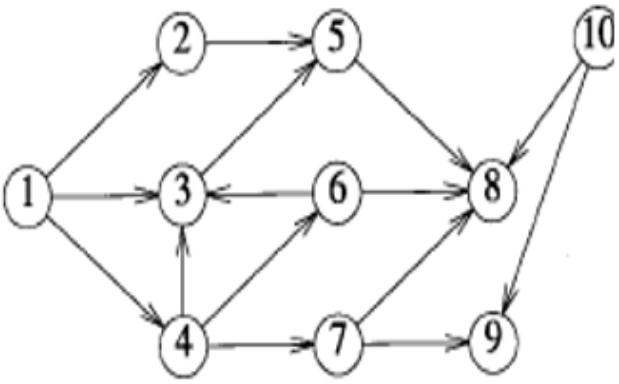
图的许多函数(寻找路径，寻找生成树，判断无向图是否连通等)都要求从一个给定的顶点开始，访问能够到达的所有顶点。

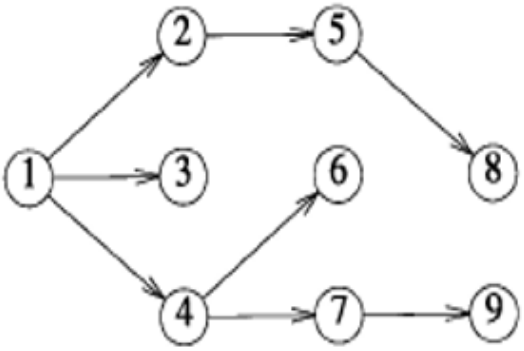
当且仅当存在一条从v 到u 的路径时，顶点v可到达顶点u

图的搜索：从一个已知的顶点开始，搜索所有可以到达的顶点

16.8.1 广度优先搜索

1.搜索顺序





1, { 2, 3, 4}, {5，6，7}， {8，9}

2.伪代码（流程）

```
breadthFirstSearch(v){
    //从顶点v 开始的宽度优先搜索
    把顶点v标记为已到达顶点;
    初始化队列Q，其中仅包含一个元素v;
    while (Q不空){
    从队列中删除顶点w;
    令u为邻接于w的顶点;
        while (u!=NULL){
            if ( u 尚未被标记) {
                把u加入队列;
```

```
                把u标记为已到达顶点
            }
            u = 邻接于w的下一个顶点;
        }
    }
}
```

16.8.2 广度优先搜索的实现

```
void bfs (int v, int reach[], int label){
    //广度优先搜索reach[i] = label用来标记顶点i
    arrayQueue<int> q(10);
    reach[v] = label;
    q.push(v);
    while (!q.empty()){
        int w=q.front();// 获取一个已标记的顶点
        q.pop();//从队列中删除一个已标记过的顶点
        // 标记所有邻接自w的没有到达的顶点
        vertexIterator<T> *iw=iterator(w);//顶点w的迭代器
        int u;
        while ( u = iw->next()!=0)//w的下一个邻接点u
            //访问w的下一个邻接点u
            if (reach[u]==0){//u未到达过
                q.push(u); reach[u] = label;//标记顶点u为已到达
            }
        delete iw;
    }
}
```

为AdjacencyWDigraph定制的BFS的实现

```
void bfs (int v, int reach[], int label){
    //广度优先搜索,reach[i] = label用来标记顶点i
    arrayQueue<int> q(10);
    reach[v] = label;
    q.push(v);
    while (!q.empty()) {
        int w=q.front();//获取一个已标记的顶点
        q.pop(); //从队列中删除一个已标记过的顶点
        //标记所有没有到达的,w的邻接点
        for (int u = 1; u <= n; u++)
            if (a[w][u] != noEdge && reach[u]==0){//u未到达过
                q.push(u);
                reach[u] = label;
            }
    }
}
```

```
为linkedDigraph定制的BFS的实现
void bfs (int v, int reach[], int label){
    //广度优先搜索， reach[i] = label用来标记顶点i
    arrayQueue<int> q(10);
    reach[v] = label;
    q.push(v);
    while (!q.empty()){
        int w=q.front(); //获取一个已标记的顶点
        q.pop(); //从队列中删除一个已标记过的顶点
        //标记所有 没有到达的、邻接自w的顶点
        //使用指针u沿着邻接表进行搜索
        for (ChainNode<int> *u =aList[w].FirstNode;u!=NULL; u=u->next){
            if(reach[u->element]==0) {//一个尚未到达的顶点
                q.push(u->element);
                reach[u->element] = label;
            }
        }
    }
}
```

16.8.3 方法 graph::bfs 的复杂性分析

从顶点v 出发，可到达的每一个顶点都被加上标记，且每个顶点只加入到队列中一次，也只从队列中删除一次。当一个顶点从队列中删除时，需要考察它的邻接点

(1)

当使用邻接矩阵时，它的邻接矩阵中的行只遍历一次，$\Theta(n)$

(2)

当使用邻接链表时，它的邻接链表只遍历一次，Θ(顶点的出度)

总时间

(1)

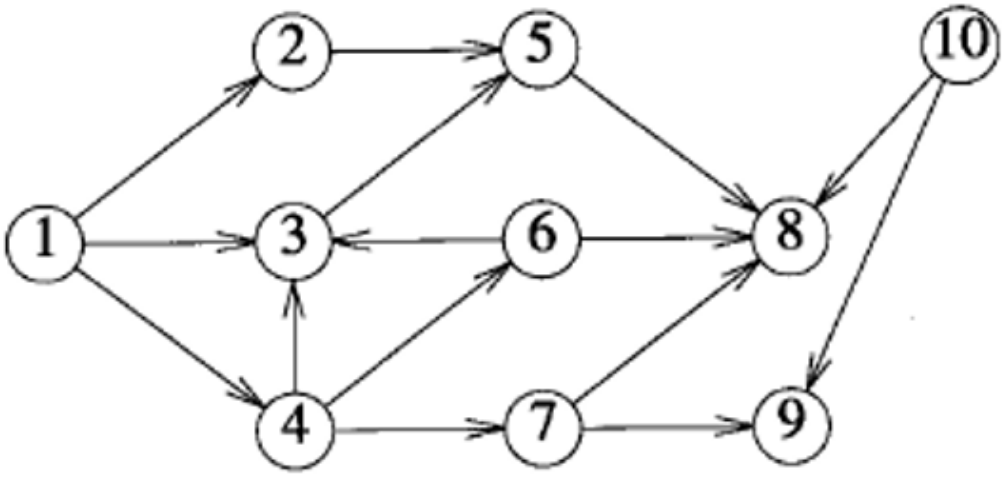
当使用邻接矩阵时，$\Theta(sn)$

(2)

当使用邻接链表时，$\Theta(\sum_{i=1}^{n} d_i^{out})$(对于无向图 / 网络来说，顶点的出度就等于它的度)

16.8.4 深度优先搜索

从顶点v 出发，首先将v 标记为已到达顶点，然后选择一个与v 邻接的尚未到达的顶点u，如果这样的u 不存在，搜索中止。假设这样的u 存在，那么从u 又开始一个新的DFS。当从u 开始的搜索结束时，再选择另外一个与v 邻接的尚未到达的顶点，如果这样的顶点不存在，那么搜索终止。而如果存在这样的顶点，又从这个顶点开始DFS，如此循环下去



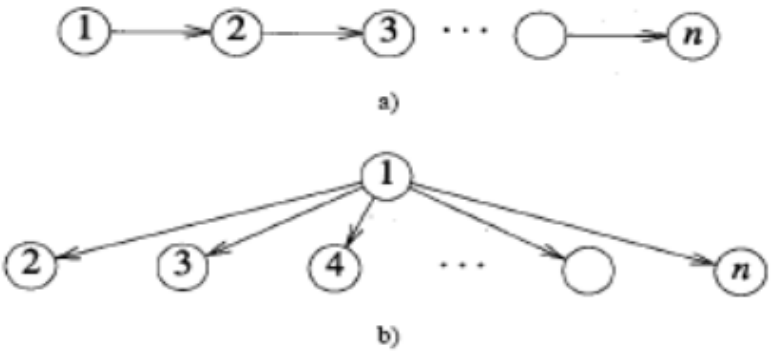
从顶点1开始进行DFS

16.8.5 深度优先搜索的实现

```
void dfs(int v, int reach[], int label){
    // dfs graph的public成员方法
    //深度优先搜索,reach[i] = label用来标记顶点i
    graph<T>::reach=reach;
    graph<T>::label=label;
    rDFs(v);//执行dfs
}
void rDFs(int v){
    //dfs保护成员方法,深度优先搜索递归方法
    reach[v] = label;
    vertexIterator<T> *iv=iterator(v);//顶点v的迭代器
    int u;
    while ( u = iv->next()!=0)  //v的下一个邻接点u
        //访问v的下一个邻接点u
        if (reach[u]==0)   //u未到达过
            rDFs (u);
    delete iv;
}
```

16.8.6 方法 graph::dfs 的复杂性分析

dfs与bfs有相同的时间和空间复杂性



(a) depthFirstSearch(1)的最坏情况(占用空间最大);

breadthFirstSearch(1)的最好情况(占用空间最小)

(b) depthFirstSearch(1)的最好情况

breadthFirstSearch(1)的最坏情况

16.9 应用

16.9.1 寻找一条路径

从顶点theSourse开始搜索(宽度或深度优先)且到达顶点theDestination时终止搜索

路径是一顶点序列，path[0]记录路径中的边数(路径长度) ;用数组 path[1:path[0]+1]记录路径中的顶点; length记录顶点的个数;path[1]= theSourse；path[length]= theDestination

```
bool rFindPath(int s){
    //寻找顶点s到达顶点destination的路径,s不是destination
    //找到一条路径,返回true;否则,返回false
    reach[s] = 1;//将s标记为已到达顶点
    vertexIterator<T>* is = iterator(s);
    int u;
    while ((u = is->next()) != 0){
        // 访问s的一个未到达邻接点
        if (reach[u] == 0){//u未到达
            //移到顶点u
            path[++length] = u;//将u加入路径
            if (u == destination || rFindPath(u))
                return true;
            //从u到destination没有路径
            length--;//从路径中删除u
        }
    }
    delete is;
    return false;
}
```

```
int* findPath(int theSource, int theDestination){
    //寻找一条顶点theSourse到达顶点theDestination的路径
    //返回一个数组path，path[0]表示路径长度
    //path[1]开始表示路径，如果路径不存在，返回NULL
    //为调用递归函数rFindPath(theSourse)初始化
    int n = numberOfVertices();
    path = new int [n + 1];
    path[1] = theSource;    //路径中的第一个顶点
    length = 1;             //当前路径长度+ 1
    destination = theDestination;
    reach = new int [n + 1];
    for (int i = 1; i <= n; i++)
        reach[i] = 0;
    //搜索路径
    if (theSource == theDestination || rFindPath(theSource))
        //找到一条路径
        path[0] = length - 1;
    else{//路径不存在
        delete [] path;
        path = NULL;
    }
    delete [] reach;
    return path;
}
```

16.9.2 连通图及其构成

一个无向图是连通图当且仅当从任意顶点开始执行DFS或BFS,所有顶点被标记为已到达顶点(任意两个顶点u和v之间存在路径)

判断无向图是否是连通图的实现

```
bool connected(){
    //当且仅当图是连通的,则返回true
    if (directed()) throw  ……//如果图不是无向图,抛出异常
    int n =numberofVertices();//图中顶点数
    //置所有顶点为未到达顶点
    int *reach = new int [n+1];
    for (int i = 1; i <= n; i++)
        reach[i] = 0;
    //对从顶点1出发可到达的顶点进行标记
    dfs(1, reach, 1);
    //检查是否所有顶点都已经被标记
    for (int i = 1; i <= n; i++)
        if (reach[i]==0) return false;
    return true;
}
```

连通构件：从顶点i可到达的顶点的集合C与连接C中顶点的边称为连通构件(connected component)

构件标记问题：给无向图中的顶点做标记，两个顶点具有相同的标记，当且仅当 两个顶点属于同一个构件

标记连通构件的实现

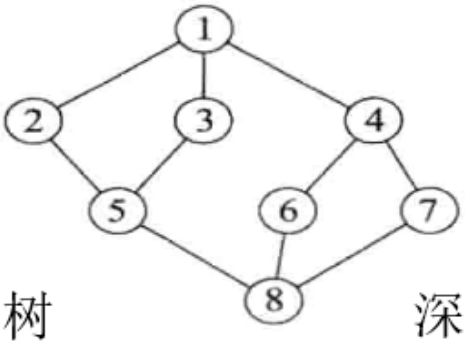
```
int labelcomponents(int c[]){
    //构件标识, 返回构件的数目,并用c[1:n]表示构件编号
    if (directed()) throw  ……//如果图不是无向图，抛出异常
    int n = numberofVertices();//图中顶点数
    //初始时，所有顶点都不属于任何构件
    for (int i = 1; i <= n; i++)
        c[i] = 0;
    int label = 0; //最后一个构件的编号
    //识别构件
    for (int i = 1; i <= n; i++)
        if (c[i]==0) //顶点i未到达
        //顶点i属于一个新的构件
        {label++ ;
        bfs(i, c, label);//标记新构件
        }
    return label;
}
```

16.9.3 生成树

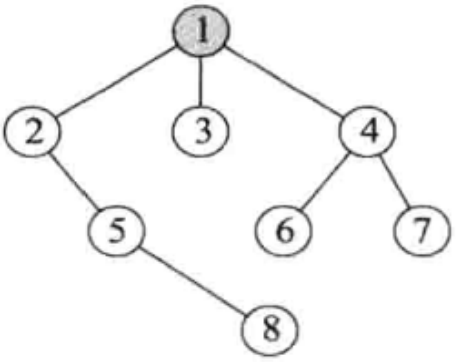
1.在一个n 顶点的**连通无向图**中，如果从任一个顶点开始进行BFS (DFS)，有n - 1个顶点是可到达的

2.用来到达n - 1个顶点的边的数目正好是n - 1

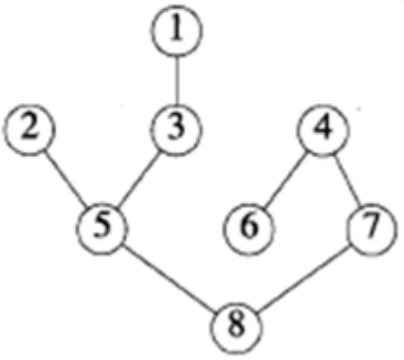
3.用来到达n - 1个顶点的边的集合中包含从v 到图中其他每个顶点的路径，因此它构成了一个连通子图，该子图即为G的生成树。宽度优先生成树(深度优先生成树)



宽度优先生成树



深度优先生成树



第17章 贪心算法

在算法界，有一个古老而神奇的传说——99%的算法都基于四种奇妙的思想(我硬顶的，知道有的对不上哈哈哈哈，但是就那意思)

- 我全都要（贪心算法）
- 走一步看一步（动态规划）
- 俄罗斯套娃（递归）
- 暴力出奇迹（穷举）

于是今天，贪心的我们就来贪心的学习一下这个听起来就很贪心的贪心算法

假设一个问题比较复杂，暂时找不到全局最优解，那么我们可以考虑把原问题拆成几个小问题（分而治之思想），分别求每个小问题的最优解，再把这些“局部最优解”叠起来，就“当作”整个问题的最优解了。

贪心选择是采用从顶向下、以迭代的方法做出相继选择，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。对于一个具体问题，要确定它是否具有贪心选择的性质，我们必须证明每一步所作的贪心选择最终能得到问题的最优解。通常可以首先证明问题的一个整体最优解，是从贪心选择开始的，而且作了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步贪心选择，最终可得到问题的一个整体最优解。

思想

贪心算法的基本思路是从问题的某一个初始解出发一步一步地进行，根据某个优化测度，每一步都要确保能获得局部最优解。每一步只考虑一个数据，他的选取应该满足局部优化的条件。若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中，直到把所有数据枚举完，或者不能再添加算法停止 [3] 。

过程

- 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。

17.2 部分例题举例

0-1背包问题：

有一个背包，背包容量是M=150kg。

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 35kg 30kg 6kg 50kg 40kg 10kg 25kg

价值 10$ 40$ 30$ 50$ 35$ 40$ 30$

这个时候我们就知道每一次放入物品的最佳选择自然是剩余物品中价值最高且质量最小的那一个，我们可以引入一个新的属性：**权重**，即用物品的价值除以质量，所以我们每一次的最佳选择就是将权重值最大的物品放入背包中。

需要注意的是背包问题与0-1背包问题的不同，虽然这两个问题极为相似，但背包问题可以用贪心算法求解，而对于0-1背包问题，贪心选择算法不能得到最优解。因为在0-1背包问题的这种情况下，它无法保证最后能将背包装满，部分闲置的背包空间，使每公斤背包的价值降低了。动态规划算法可以有效的解0-1背包问题。

拓扑排序

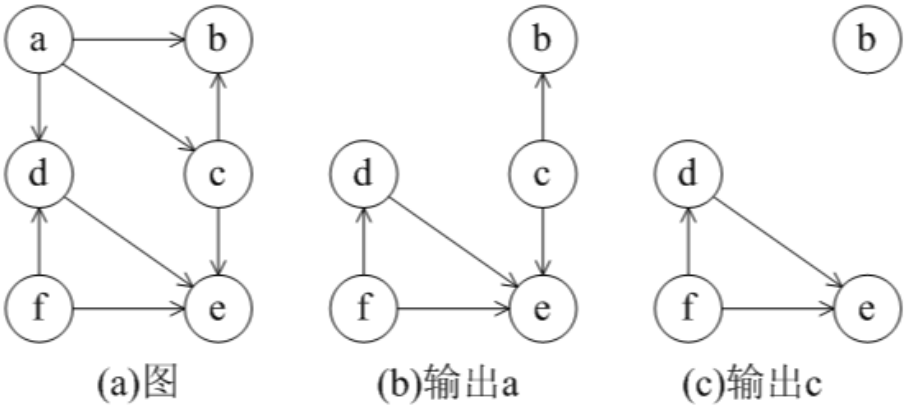
在一个有向图中，对所有的节点进行排序，要求没有一个节点指向它前面的节点。

先统计所有节点的入度，对于入度为0的节点就可以分离出来，然后把这个节点指向的节点的入度减一。

一直做改操作，直到所有的节点都被分离出来。

如果最后不存在入度为0的节点，那就说明有环，不存在拓扑排序，也就是很多题目的无解的情况。

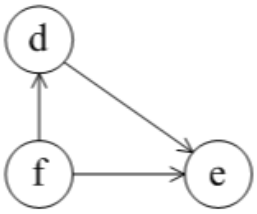
下面是算法的演示过程。



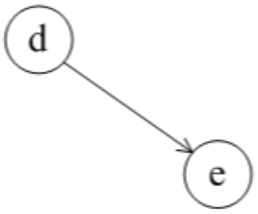
(a)图

(b)输出a

(c)输出c



(d)输出b



(e)输出f

(f)输出d (g)输出e

拓扑排序序列：acbfde

图一

算法的简单代码

```
 //b[]为每个点的入度
for(i=1;i<=n;i++){
   for(j=1;j<=n;j++){
      if(b[j]==0){    //找到一个入度为0的点
        ans=j;
        vis[cnt++]=j;
        b[j]--;
        break;
       }
    }
    for(j=1;j<=n;j++)
        if(a[ans][j]) b[j]--; //与入度为0的点相连的点的入度减一
}
    printf("%d",vis[0]);
    for(i=1;i<cnt;i++) printf(" %d",vis[i]);
    printf("\n");
```

单源最短路径（迪杰斯特拉算法）

给定一个带权有向图 G=(V,E)，其中每条边的权是一个非负实数。再给定 V 中的一个顶点，称为源，计算源到所有顶点的最短距离。

求取过程：

1. 从起点开始，查找所有与起点直接相连的点，并把连接它们的边权值作为起点到它们的距离(有平行边取最小值)。起点到起点的距离为0，其它不与起点直接相连的点距离为无穷大，存入上图所示的表中。

2. 遍历所有点，查找得到的结果中距离最小的那个点，然后再看所有与该点直接相连的点，计算该点到它们的最短距离(即从起点开始，经由该点到达的点的距离)，根据结果更新上图。

3. 不断重复2中过程，直到每一个点都像2中那样计算过一遍。

如果觉得描述抽象，可以点点这里，有一个mooc视频推荐大家食用，下面是URL（https://www.icourse163.org/learn/NJTU-1002530017?tid=1205949204#/learn/content?type=detail&id=1210310564&sm=1）

代码示例

```
#include <bits/stdc++.h>

using namespace std;

const int N=1e5+5;
const int MAX=2147483647;
typedef long long LL;

struct node{        //建图部分还是一样的
    int from;
    int to;
    int value;
    int next;
}edge[2*N];
int head[N];

LL dis[N];
int visited[N];

priority_queue<pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > >vis

int main()
{
    int n,m,s;
    int cnt=0;
    cin>>n>>m>>s;
    for(int i=1;i<=n;i++) dis[i]=MAX;
    for(int i=1;i<=m;i++)
```

```
    {
        cnt++;
        scanf("%d%d%d",&edge[cnt].from,&edge[cnt].to,&edge[cnt].value);
        edge[cnt].next=head[edge[cnt].from];
        head[edge[cnt].from]=cnt;
    }


    dis[s]=0;
    vis.push(make_pair(0,s));     //储存某点距离和该点
                                  //make_pair是c++自带的二元组定义方式,可以通过first
    while(!vis.empty())         //判断所有点是否都查找过一遍
    {
        int start=vis.top().second;     //优先队列队首即最小的元素
        vis.pop();

        if(visited[start]==1) continue;     //如果该点查找过，跳过
        visited[start]=1;

        for(int i=head[start];i!=0;i=edge[i].next)
        {
            if(dis[edge[i].to]>dis[start]+edge[i].value)
            {
                dis[edge[i].to]=dis[start]+edge[i].value;
                vis.push(make_pair(dis[edge[i].to],edge[i].to));
            }
        }
    }
    for(int i=1;i<=n;i++)
    {
        if(i==n) printf("%d\n",dis[i]);
        else printf("%d ",dis[i]);
    }
}
```

最小生成道路

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，并且有保持图连通的最少的边。最小生成树可以用kruskal（克鲁斯卡尔）算法或prim（普里姆）算法求出。

在一给定的无向图G = (V, E) 中，(u, v) 代表连接顶点 u 与顶点 v 的边（即），而 w(u, v) 代表此边的权重，若存在 T 为 E 的子集（即）且为无循环图，使得
的 w(T) 最小，则此 T 为 G 的最小生成树。

这个问题的解决也十分抽象，如果不是很清楚的话，我们也推荐大家去这个mooc看一看，URL（https://www.icourse163.org/learn/NJTU-1002530017?tid=1205949204#/learn/content?type=detail&id=1210310563&cid=1212286444）

下面给出Kruskal算法和Prim算法的思想：

- Kruskal算法：
  假设 WN=(V,{E}) 是一个含有 n 个顶点的连通网，则按照克鲁斯卡尔算法构造最小生成树的过程为：先构造一个只含 n 个顶点，而边集为空的子图，若将该子图中各个顶点看成是各棵树上的根结点，则它是一个含有 n 棵树的一个森林。之后，从网的边集 E 中选取一条权值最小的边，若该条边的两个顶点分属不同的树，则将其加入子图，也就是说，将这两个顶点分别所在的两棵树合成一棵树；反之，若该条边的两个顶点已落在同一棵树上，则不可取，而应该取下一条权值最小的边再试之。依次类推，直至森林中只有一棵树，也即子图中含有 n-1条边为止。

- Prim算法：
  1).输入：一个加权连通图，其中顶点集合为V，边集合为E；
  2).初始化：$V_{new} = x$，其中x为集合V中的任一节点（起始点），$E_{new} =$,为空；
  3).重复下列操作，直到$V_{new} = V$：
  a.在集合E中选取权值最小的边<u, v>，其中u为集合V_{new}中的元素，而v不在V_{new}集合当中，并且v∈V（如果存在有多条满足前述条件即具有相同权值的边，则可任意选取其中之一）；
  b.将v加入集合V_new中，将<u, v>边加入集合Enew中；
  4).输出：使用集合V_{new}和E_{new}来描述所得到的最小生成树。

第18章 分而治之

本章概述

分而治之策略可以用来设计有效的计算机算法，本次主要讲述将分而治之策略应用到排序算法中。分而治之算法把一个问题实例分解为若干个小型而独立的实例，从而可以在并行计算机上执行；那些小型而独立的实例可以在并行计算机的不同处理器上完成。

18.1 算法思想

分而治之方法与软件设计的模块化方法非常相似。一个问题的小实例可以用直接方法求解。而要解决一个问题的大实例，可以1）把它分成两个或多个更小的实例；2）分别解决每个小实例；3）把这些小实例的解组合成原始大实例的解。小实例通常是原问题的实例，可以使用分而治之策略递归求解。

18.2 应用

18.2.1 残缺棋盘

18.2.2 归并排序

1、排序方法
可以应用分而治之法来设计排序算法，把n个元素按非递减顺序排列。这种排序算法常用的结构是:若n为1，则算法终止；否则，将序列划分为k个子序列(k是不小于2的整数)。先对每一个子序列排序，然后将有序子序列归并为一个序列。

假设将n个元素的序列仅仅划分为两个子序列，称之为二路划分。一种二路划分是把前面n-1个元素放到第一个子序列中(称为A),最后一个元素放到第二个子序列中(称为B)。按照这种划分方式对A递归地进行排序。由于B仅含一个元素，所以它已经有序。在A排序后，使用insert函数将A和B归并。把这种排序与插入排序insertionSort比较，可以发现它实际上是插入排序的递归形式。该算法的复杂度为O(n)。

另一种二路划分是将关键字最大的元素放入B,剩余元素放入A。然后对A递归排序。为了将排序后的A和B归并，这时只需要将B附加到A的尾部。如果使用函数Max来寻找关键字最大的元素，那么这种排序算法实际上就是选择排序selectionSort的递归形式。如果使用冒泡函数来寻找关键字最大的元素并把它移到最右边的位置，那么这种排序算法就是冒泡排序bubbleSort的递归形式。这两种排序算法的复杂度均为$0(n^2)$。如果A一旦有序，就终止对A的递归划分，则算法的复杂度为$0(n^2)$。

上述划分方案将n个元素序列划分为两个极不平衡的子序列A和B。A有n-1个元素，而B仅有一个元素。如果划分得平衡一点， 情况会怎样呢?假设A包含n/k个元素，B包含其余的元素。递归地应用分而治之法对A和B进行排序，然后采用一个被称之为归并的过程,将有序子序列A和B归并成一个序列。

分而治之排序算法的伪码：

```
void sort(E,n)
{//对E中的n个元素排序，k是全局变量
    if(n >= k)
    {
        i=n/k;
        j=n-i;
        令A由E的前i个元素组成
        令B由E剩余的j个元素组成
        sort(A,i);
        sort(B,i);
        merge(A,B,E,i,j);//把A和B合并到E
    }
    else
        对E插入排序
}
```

归并排序的平均复杂度为O（nlogn）。

2、C++实现

当 k = 2 时的排序称为归并排序（merge sort），更准确地说，是二路归并排序（two-way merge sort）。

归并排序函数用一个数组a来存储元素序列E，并用a返回排序后的序列。当序列E被划分为两个子序列时，不必把它们分别复制到A和B中，只需简单地记录它们在序列E中的左右边界。然后将排序后的子序列归并到一个新数组b中，最后再将它们复制回a中。

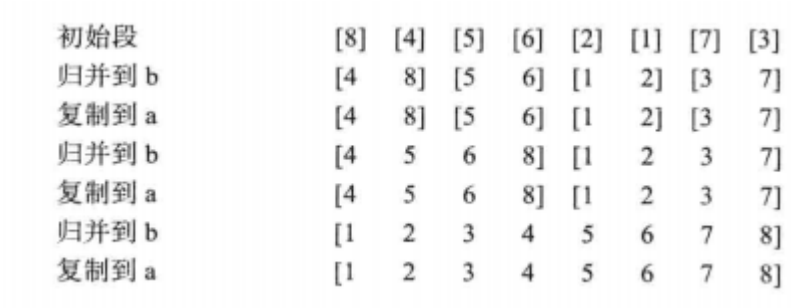
分而治之排序算法的改进：

```
void mergeSort(T *a,int left,int right)
{
    if(left < right)
    {//至少有两个元素
        int middle = (left + right)/2;
        mergeSort(a,left,middle);
        mergeSort(a,middle+1,right);
        merge(a,b,left,middle,right);  //从a到b归并
        copy(b,a,left,right);          //将排序结果复制到a
    }
}
```

可以从很多方面改进上述代码的性能。例如，消除递归。

程序中的递归只是简单地对序列反复划分，直到序列的长度变为1，这时再进行归并。这个过程用n为2的幂来描述会更好。长度为1的子序列被归并为长度为2的有序子序列；长

度为2的子序列被归并为长度为4的有序子序列;这个过程不断地重复直到归并为一个长度为n的序列。
下图是n=8时的归并(和复制)过程，方括号表示一个有序序列左右边界。



初始段	[8]	[4]	[5]	[6]	[2]	[1]	[7]	[3]
归并到 b	[4	8]	[5	6]	[1	2]	[3	7]
复制到 a	[4	8]	[5	6]	[1	2]	[3	7]
归并到 b	[4	5	6	8]	[1	2	3	7]
复制到 a	[4	5	6	8]	[1	2	3	7]
归并到 b	[1	2	3	4	5	6	7	8]
复制到 a	[1	2	3	4	5	6	7	8]

二路归并排序的一种迭代算法是这样的:首先将每两个相邻的大小为1的子序列归并，然后将每两个相邻的大小为2的子序列归并，如此反复，直到只剩下一个有序序列。轮流地将元素从a归并到b,从b归并到a,实际上消除了从b到a的复制过程。

用归并排序法对数组元素a[0:n-1]排序:

```
template <class T>
void mergeSort(T a[],int n)
{//使用归并排序方法对a[0:n-1]排序
    T *b = new T[n];
    int segmentSize = 1;
    while(segmentSize < n)
    {
        mergePass(a,b,n,segmentSize);   //从a到b的归并
        segmentSize += segmentSize;
        mergePass(b,a,n,segmentSize);   //从b到a的归并
        segmentSize += segmentSize;
    }
    delete[] b;
}
```

为了完成排序代码，需要函数mergePass。不过这个函数仅用来确定需要归并的子序列的左右边界。实际的归并是由函数merge完成的。

把相邻的两个数据段从x到y归并:

```
template <class T>
void mergePass(T x[],T y[],int n,int segmentSize)
{//从x到y归并相邻的数据段
    int i = 0;    //下一个数据段的起点
    while(i <= n - 2*segmentSize)
```

```
    {//从x到y归并相邻的数据段
        merge(x,y,i,i + segment-1,i + 2*segmentSize - 1);
        i = i + 2*segmentSize;
    }

    //少于两个满数据段
    if(i + segmentSize < n)
        //剩有两个数据段
        merge(x,y,i+segmentSize - 1,n - 1);
    else
    {
        //只剩一个数据段，复制到y
        for(int j = i;j < n;j++)
            y[j] = x[j];
    }
}
```

把相邻的数据段从c到d归并：

```
template <class T>
void merge(T c[],T d[],int startOdFirst,int endOfFirst,int endOfSecond)
{//把两个相邻数据段从c归并到d
    int first = startOfFirst,       //第一个数据的索引
        second = endOfFirst + 1,    //第二个数据段的索引
        result = startOfFirst;      //归并数据段的索引

    //直到有一个数据段归并到归并段d
    while((first <= endOfFirst) && (second <= endOfSecond))
    {
        if(c[first] <= c[second])
            d[result++] = c[first++];
        else
            d[result++] = c[second++];
    }

    //归并剩余元素
    if(first > endOfFirst)
    {
        for(int q = second;q <= endOfSecond;q++)
            d[result++] = c[q];
    }
    else
    {
        for(int q = first;q <= endOfFirst;q++)
            d[result++] = c[q];
```

```
    }
}
```

3、自然归并排序
上述归并排序函数也称直接归并排序（straight merge sort）。

在自然归并排序中，首先认定在输人序列中已经存在的有序段。

认定的方法是，从左至右扫描序列元素，若位置i的元素比位置 i+1 的元素大，则位置i便是一个分割点。然后归并这些有序段，直到剩下一个有序段。

自然归并排序的最好情况是，输入序列已经有序。自然归并排序只认定了一个有序段，不需要归并，自然归并排序需用时O(n)。

自然归并排序的最坏情况是输人序列按递减顺序排列。最初认定的有序段有n个。这时的归并排序和自然归并排序需要相同的归并次数，但自然归并排序为记录有序段的边界需要更多的时间。因此，在最坏情况下的性能，自然归并排序不如直接归并排序。

只有输入序列确实有很少的有序段时，才建议使用自然归并排序。

18.2.3 快速排序

1、排序方法
用分而治之法可以实现另一种完全不同的排序——快速排序 (quick sort)。把n个元素划分为三段:左段left、中间段middle和右段right。中段仅有一个元素。左段的元素都不大于中间段的元素，右段的元素都不小于中间段的元素。因此可以对left 和right独立排序,并且排序后不用归并。middle 的元素称为支点(pivot)或分割元素(partitioning element)。

快速排序的简单描述：

```
//对a[0:n-1]快速排序
从a[0:n-1]中选择一个元素作为支点，组成中间段。
把剩余元素分为左段left和右段right。使左段的元素关键字都不大于支点关键字，
    右段的元素关键字都不小于支点的关键字
对左段递归快速排序
对右段递归快速排序
最终结果按左段、中间段和右段排列
```

2、C++实现
快速排序函数quickSort把数组a的最大元素移动数组的最右端，然后用递归函数quickSort执行程序。递归函数要求每一个数据段或者其最大元素位于右端，或者其后继元素大于数据段的所有元素，因此，把最大元素移到最右端；如果这个条件不满足，例如，当支点是最大元素时，第一个do循环语句的结果时左索引值大于n-1。

递归快速排序的驱动程序：

```
template <c1ass T>
void quickSort(T a[], int n)
{//对a[0:n-1]快速排序
    if (n <= 1) return;
    //把最大的元素移到数组右端
    int max = indexOfMax(a,n) ;
    swap(a[n - 1], a[max]) ;
    quickSort(a, 0, n - 2);
}
```

递归快速排序函数：

```
template <class T>
void quickSort(T a[],int leftEnd,int rightEnd)
{//对a[leftEnd:rightEnd]排序，a[rightEnd+1] >= a[leftEnd:rightEnd]
    if (leftEnd >= rightEnd) return;
    int leftCursor = leftEnd,        //从左到右移动的索引
        rightCursor = rightEnd + 1;  //从右到左移动的索引
    T Pivot = a[leftEnd];

    //将位于左侧不小于支点的元素和位于右侧不大于支点的元素交换
    while (true)
    {
        do
        {//寻找左侧不小于支点的元素
            leftCursor++;
        } while (a[leftCursor] < pivot);

        do
        {//寻找右侧不大于支点的元素
            rightCursor--;
        } while (a[rightCursor] >pivot) ;

        if (leftCursor > rightCursor) break;//没有找到交换的元素对
        swap(a[leftCursor], a[rightCursor]);
    }

    //放置支点
    alleftEnd] = a[rightCursor];
    a[rightCursor] = pivot;
    quickSort (a,leftEnd,rightCursor - 1) ;    //对左侧的数段排序
    quickSort(a, rightCursor + 1, rightEnd) ;  //对右侧的数段排序
}
```

3、复杂度分析

快速排序的平均复杂度时O(nlogn)，其中n为排序元素的个数。

4、三值取中快速排序

对有序的输入序列实施快速排序，却表现出最坏情况下的时间性能。也就是说，一个快速排序算法，它对有序表排序比对无序表排序要慢，这是让人痛苦的问题。不过，我们可以解决这个问题，同时提高快速排序的平均性能，方法是根据三值取中规则(median-of-threerule)选择支点元素。

三值取中快速排序(median-of-three quick sort)在三元素a[leftEnd]、a[(leftEnd+rightEnd)/2]和a[rightEnd]中选择大小居中的中值元素作为支点元素。为此，最简单的做法是将中值位置上的元素与元素a[leftEnd]交换，然后继续使用递归快速排序的驱动程序。在递归快速排序函数中，如果a[rightEnd]是中值元素，则将a[leftEnd]和a[rightEnd]交换，然后再将a[leftEnd]选为支点元素，而其他代码不动。

使用三值取中规则对输入有序的表进行快速排序，用时为O(nlogn)。而且不会出现一个数据段为空的情况，除非有关键字相同的元素。也就是说，使用三值取中规则可以保证左右两个数据段的长度更均衡。