数据结构复习纲要

> ---NonoTion

第15章 平衡搜索树

AVL树

定义：

如果搜索树的高度总是O(logn),我们就能保证查找、插入和删除的时间为O(logn)。最坏情况下的高度为O(logn)的树为**平衡树(balanced tree)**。比较流行的一种平衡树是AVL树

一棵空的二叉树是 AVL树;

如果T是一棵非空的二叉树，T_L和T_R分别是其左子树和右子树，那么当T满足以下条件时，T是一棵AVL树:

1)T_L和T_R是AVL树:

2)$|h_L - h_R| \leq 1$，其中h_L和h_R分别是TL和TR的高。

一棵AVL搜索树既是一棵AVL树，也是一棵二叉树

AVL树的一些特征：

- n个元素的AVL树，高度为O(logn)
- 对于大于0的任意一个n，都有一棵AVL树
- AVL搜索树的查找，插入，删除操作的时间复杂度都为O(logn)

AVL树的高度

对一颗高度为h的AVL树，令N_h使其最少的节点数。在最坏情况下，根的一棵子树高度是h-1,另一棵子树的高度是h-2,而且两棵子树都是AVL树，因此有

$$N_h = N_{h-1} + N_{h-2} + 1$$

如果树中有n个节点，那么树的最大高度为 O(logn)

AVL树的描述

AVL一般采用链表描述。为了简化插入和删除操作，为每一个节点增加一个平衡因子bf。节点x的平衡因子$bf(x)$定义为：

x的左子树高度-x的右子树高度，由AVL树的定义可知，平衡因子的取值可能由1，0，-1

AVL搜索树的搜索

用二叉搜索树的搜索即可，搜索时间为O(logn)

AVL搜索树的插入

用二叉搜书树的插入在AVL树中插入一个节点生成一棵搜索树，如果它有一个或更多的节点其平衡因子不再是-1、0、1那么搜索树就是不平衡的，需要移动不平衡子树来恢复平衡

导致不平衡的情形：

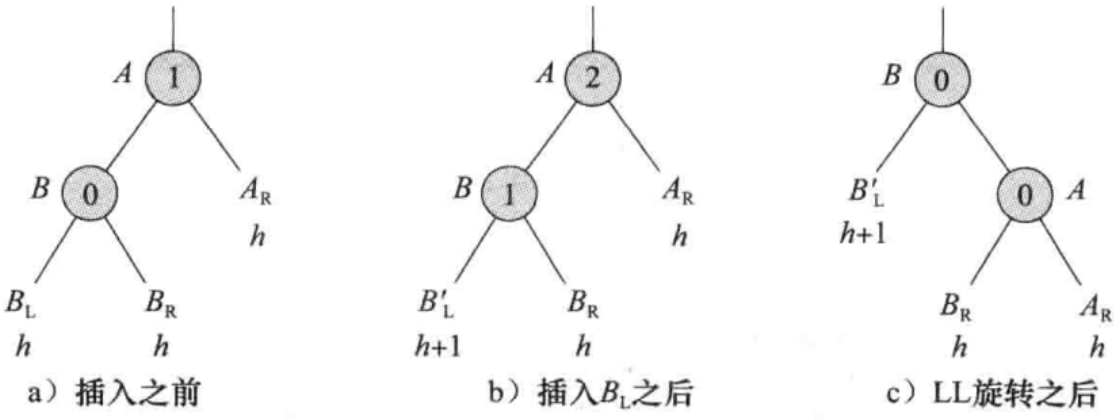
- 插入操作之后，平衡因子从-1变为-2(在X的右子树插入)
- 插入操作之后，平衡因子从1变为2(在X的左子树插入)

不平衡的情况有两种:

1. L型不平衡

又可细分为LL型和LR型不平衡

LL型不平衡

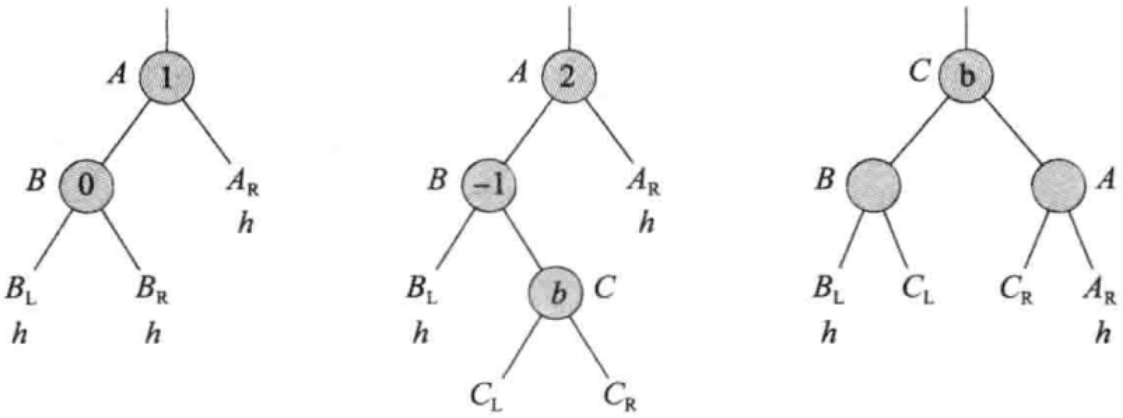


节点内的数字是平衡因子，子树名称下面的是子树高度

图 15-4　LL 旋转

步骤：以B为根节点，A为B右子树的根节点，BL`为B的左子树，BR为A的左子树

LR型不平衡



a）插入之前　　b）插入B_R之后　　c）LR旋转之后

$$b=0 \Rightarrow bf(B)=bf(A)=0 \text{ 旋转后}$$
$$b=1 \Rightarrow bf(B)=0,\ bf(A)=-1 \text{ 旋转后}$$
$$b=-1 \Rightarrow bf(B)=1,\ bf(A)=0 \text{ 旋转后}$$

2. R型不平衡

又可细分为RL型和RR型不平衡

与L型不平衡是镜像的

总的说来AVL搜索树的插入步骤可以归纳为

1. 沿着根节点开始的路径，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。如果找到具有相同关键字的元素，那么插入失败，终止算法
2. 如果在步骤1种的A节点不存在，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
3. 如果bf(A)=1并且将新节点插入A的右子树种或bf(A)=-1并且新节点插入到左子树，那么A的平衡因子是0。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
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平衡因子做相应的修改

AVL搜索树的删除

设q是被删除节点的父节点

- 如果q的新平衡因子是0，那么它的高度减1，这时需要改变它的父节点的平衡因子，也有可能要改变其他祖先节点的平衡因子
- 如果q的新平衡因子是1或-1，那么它的高度与删除前相同，且无需改变其祖先节点的平衡因子值

- 如果q的新平衡因子是-2或2，那么树在q处是不平衡的

从q到根节点的路径上，找到第一个平衡因子为-2或2的节点，记作A。如何恢复A的平衡，需要按类型而定

- 删除发生在A的左子树为L型，否则为R型

找到A最高的子树，进行旋转，若最高的子树为左子树，R-1,R0,R1

若最高子树为右子树，L-1,L0,L1

看课本即可

B-树

m叉搜索树

定义 15-2　*m* 叉搜索树（*m*-way search tree）可以是一棵空树。如果非空，它必须满足以下特征：

1）在相应的扩充搜索树中（即用外部节点替换空指针之后所得到的搜索树），每个内部节点最多可以有 m 个孩子以及 $1 \sim m-1$ 个元素（外部节点不含元素和孩子）。

2）每一个含有 p 个元素的节点都有 $p+1$ 个孩子。

3）对任意一个含有 p 个元素的节点，设 $k_1, \cdots, k_p$ 分别是这些元素的关键字。这些元素顺序排列，即 $k_1<k_2<\cdots<k_p$。设 $c_0, c_1, \cdots, c_p$ 是该节点的 $p+1$ 个孩子。在以 c_0 为根的子树中，元素的关键字小于 k_1；在以 C_p 为根的子树中，元素的关键字大于 k_p；在以 c_i 为根的子树中，元素的关键字大于 k_i 而小于 k_{i+1}，其中 $1 \leqslant i<p$。

m叉搜索树的搜索、插入、删除看课本即可

m叉搜索树的高

一棵高度为h的m叉搜索树(不含外部节点)最少有h个元素(每层一个节点，每个节点包含一个元素)，最多有m^h-1个元素。

在高度为h的m叉搜索树中，元素个数在h到m^h-1之间，所以一棵n元素的m叉搜索树的高度是$log_m^{(n+1)}$~n

m阶B-树

m阶B-树是一棵m叉搜索树。如果B-树非空，那么相应的扩展树满足下面特征：

- 根节点至少有两个孩子
- 除根节点外，所有内部节点至少有$\lceil m/2 \rceil$个孩子
- 所有外部节点在同一层

B-树的高度

设T是一棵高度为h的m阶B-树。令d=$\lceil m/2 \rceil$，n是T的元素个数，则

1. $2d^{h-1}-1 \leq n \leq m^h-1$
2. $log_m^{n+1} \leq h \leq log_d^{\frac{n+1}{2}}+1$

B-树的搜索

与m叉树相同

B-树的插入

在B-树种插入一个新元素，首先要在B-树种搜索关键字与之相同的元素。如果存在这样的元素，那么插入失败，因为在B-树的元素种不允许有重复的关键字。如果不存在这样的元素，比那可以将元素插入在搜索路径中所遇到的最后一个内部节点中。如果节点饱和，插入一个新元素时，需要分裂该节点

B-树的删除

1. 该元素位于叶子节点，直接删除
2. 该元素位于非叶子节点，转换为情况1

第16章 图

无权图的描述

邻接矩阵

一个n顶点图G=(V,E)的邻接矩阵是一个nxn矩阵，其中每个元素是0或1.

如果G是一个无向图，则其中元素定义如下：

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

如果G是有向图，那么其中的元素定义如下：

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

一些结论:

- n个顶点的无向图，有A(i,i)=0
- 无向图的邻接矩阵是对称的,A(i,j)=A(j,i)
- 对于n顶点的无向图，$\sum_{j=1}^{n}A(i,j)=\sum_{j=1}^{n}A(j,i)=d_i$($d_i$是顶点的度)
- 有向图中，$\sum_{j=1}^{n}A(i,j)=d_i^{out}\quad\sum_{j=1}^{n}A(j,i)=d_i^{in}$

将邻接链表映射到数组

- 使用映射A(i,j)=1,当且仅当a(i,j)=true　将nxn的图映射到一个(n+1)(n+1)的布尔型数组 需要$(n+1)^2$个字节
- 映射A(i,j)=1，当且仅当a(i-1,j-1)=true　将nxn的图映射到一个nxn的布尔型数组，需要n^2个字节
- 进而还可以吧对角线的元素去掉
- 进而还可以只存储无向图的上三角或下三角部分　仅需$(n^2-n)/2$字节

使用邻接矩阵时，确定邻接于一个给定节点的集合需要用时$\Theta(n)$,增加一条边或删除一条边用时$\Theta(1)$

邻接链表

一个顶点i的邻接表是一个线性表，它包含所有邻接于顶点i的顶点，在一个图的邻接表描述中，图的每一个顶点都有一个邻接表，当邻接表为链表时，就是邻接链表

我们使用类型为链表的数组aList来描述所有链表，aList[i].firstNode指向顶点i的邻接表的第一个顶点。如果x指向链表aList[i]的一个顶点，那么(i,x->element)是图的一条边，其中element的数据类型是int

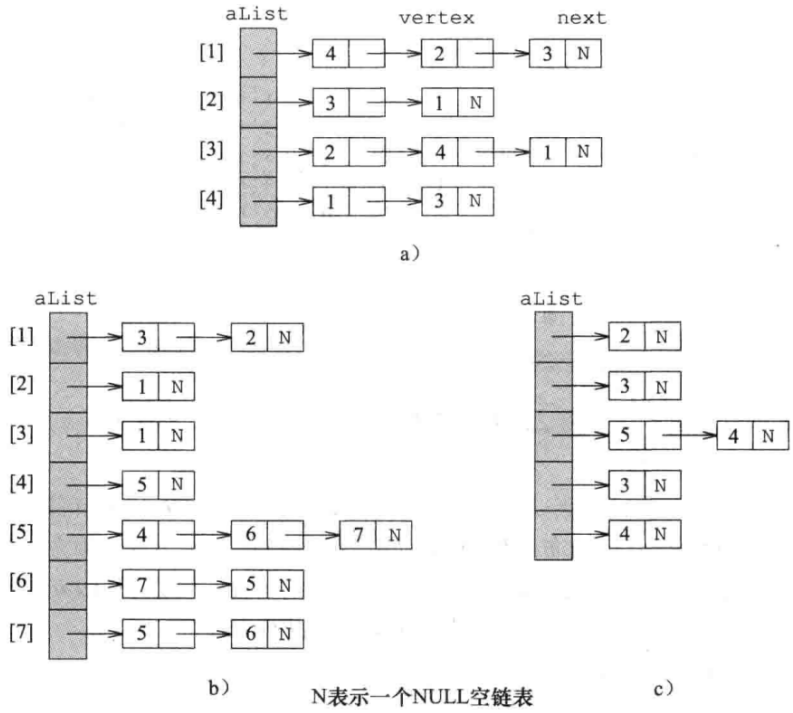


图 16-11　图 16-1 的邻接链表

空间复杂度分析

一个指针和一个整数各需4个字节，因此用一个邻接链表描述一个有n个顶点的图需要8(n+1)字节存储n+1个firstNode指针和aList链表的listSize域，4x2xm个字节存储m个链表节点，m是边数

当e远远小于n^2时，邻接链表比邻接矩阵需要更少的空间

时间复杂度分析

确定邻接于顶点i的顶点需要用时$\Theta(\text{邻接于顶点}i\text{的顶点数})$

插入或删除一条边(i,j) 对于无向图用时$\Theta(d_i + d_j)$

对于有向图用时$\Theta(d_i^{out})$

邻接数组

用二维不规则数组存储图

比邻接链表少用4m字节，因为不需要next指针域

渐进时间复杂度与邻接链表法相同，但实际上大部分图的操作，邻接数组要比邻接链表快

加权图的描述

将无权图的描述进行简单扩充即可得到加权图的描述。用成本邻接矩阵C描述加权图。C(i,j)表示边(i,j)的权值，使用方法和邻接矩阵的使用方法意义。在这种方法中，需要给不存在的边赋一个值，一般是一个很大的值，在实现代码中，用noEdge代表这个值

链表的元素有两个域vertex和weight，就可以从无权图的邻接链表得到加权图的邻接链表。

图的遍历

广度优先搜索

这种从一个顶点开始，搜索所有可到达顶点的方法叫做**广度优先搜索**。这种搜索方法可使用队列实现

```
virtual void bfs(int v,int reach[],int label)
{//广度优先搜索。reach[i]用来标记从顶点v所有可到达的顶点
    arrayQueue<int> q(10);
    reach[v]=label;
    q.push(v);
    while(!q.empty())
    {
        int w=q.front;
        q.pop();
        vertexIterator<T> *iw=iterator(w);
        int u;
        while((u=iw->next())!=0)
```

```
            if(reach[u]==0)
            {
                q.push(u);
                reach[u]=label;
            }
        delete iw;
    }
}
```

为邻接矩阵法定制的BFS代码

```
void bfs(int v,int reach[],int label)
{
    arrayQueue<int> q(10);
    reach[v]=label;
    q.push(v);
    while(!q.empty())
    {
        //从队列中删除一个有标记的顶点
        int w=q.front();
        q.pop();
        for(int u=1;u<=n;u++)
        {
            if(a[w][u]!=noedge&&reach[w][u]==0)
            {
                q.push[u];
                reach[w][u]=label;
            }
        }
    }
}
```

时间复杂度分析

如果顶点v邻接的顶点有s个

则O(sn)

为邻接链表法定制的BFS代码

```
void bfs(int v,int reach[],int label)
{
    arrayQueue<int> q(10);
    reach[v]=label;
    q.push(v);
    while(!q.empty())
```

```
    {
        int w=q.front();
        q.pop();
        for(chainNode<int>*u =aList[u].firstNode;u!=NULL;u=u->next)
        {
            if(reach[u->element]==0)
            {
                q.push(u->element);
                reach[u->element]=label;
            }
        }
    }
}
```

时间复杂度分析

O($\sum_i d_i^{out}$)

深度优先搜索

与二叉树的前序遍历很相似

```
void dfs(int v,int reach[],int label)
{
    graph<T>::reach=reach;
    graph<T>::label=label;
    rDfs(v);
}

void rDfs(v)
{
    reach[v]=label;
    vertexIterator<T> *iv=iterator(v);
    int u;
    while((u=iv->next())!=0)
    {
        if(reach[u]==0)
            rDfs(u);
    }
    delete iv;
}
```

复杂度分析

DFS和BFS具有相同的时间和空间复杂性

应用

寻找一条路径

用深度优先搜索或广度优先搜索，寻找一条从源点sourceNode到终点theDestination的路径。注意要记录这条路径的每个节点

前序方法dfs

```
int *findPath(int theSource,int theDestination)
{
    int n=numberOfVertices;
    path=new int [n+1];
    path[1]=theSource;
    length=1;
    destination=theDestination;
    reach=new int [n+1];
    for(int i=1;i<=n;i++)
        reach[i]=0;
    if(theSource==theDestination||rFindPath(theSource))
        path[0]=length-1;
    else
    {
        delete[] path;
        path=NULL;
    }
    delete reach;
    return path;
}
bool rFindPath(int s)
{
    reach[s]=1;
    vertexIterator<T>* is=iterator(s);
    int u;
    while((u=is->next())!=0)
    {
        if(reach[u]==0)
            path[++length]=u;
        if(u==destination||rFindPath(u))
            return true;
        length--;
    }
    delete is;
```

```
    return false;
}
```

FindPath首先初始化 graph的静态数据成员:destination, path,length和 reach。算法实际上调用了保护性方法 graph::rFindPath，这个方法在路径不存在时，返回false。

方法graph::rFindPath对DFS 做了两点修改:

1)一到达路径终点，rFindPath就停止。

2)rFindPath把从源点theSource 到当前顶点u 的路径上的顶点都记录在数组 path 中。

rFind寻找的不是最短路径，如果需要找到最短路径，需要用BFS代替DFS

连通图及其构成

从任意一个顶点开始BFS或DFS，然后检验是否所有顶点都被标记为已到达顶点，从而可以判断一个无向图是否连通。连通的概念仅是对无向图定义的

```
bool connected()
{
    if(directed())
        throw undefinedMethod("graph::connected() not defined for directed graph");
    int n=numberOfVertices;
    reach=new [n+1];
    for(int i=1;i<=n;i++)
        reach[i]=0;
     dfs(1,reach,1);
    for(int i=1;i<=n;i++)
    {
        if(reach[i]==0)
            return false;
    }
    return true;
}
```

在一个无向图中，从一个顶点i可到达的顶点集合 C与连接 C的任意两个顶点的边称为**连通构件(connected component)。**

构件标记问题(component-labeling problem)是指对无向图的顶点进行标记，使得2个顶点具有相同的标记，当且仅
当它们属于同一构件。

```
int labelComponents(int c[])
```

```
{
    if(directed())
        throw undefinedMethod("graph::labelComponents() not defined for directed graph");
    int n=numberOfVertices;
    for(int i=1;i<=n;i++)
        c[i]=0;//令所有顶点都是非构件
    label=0;
    for(int i=1;i<=n;i++)
    {
        if(c[i]==0)
        {
            label++;
            bfs(i,c,label);
        }
    }
    return label;
}
```

生成树

在一个具有n个顶点的连通无向图或网中，如果从任一个顶点开始进行 BFS，那么从定
理16-1可知，所有顶点都将被加上标记。在graph::bfs(见序16-4)的内层while 循环中正好有 n-1个顶点到达。在该循环中，当到达一个新顶点时，到达的边是(w,u)。这样得
到的边集有 n-1条边，且它包含一条从v到其他每个顶点的路径，这条路径构成了一个连通
子图，该子图即为G的生成树。
广度优先生成树(breadth-first spanning tree)是按BFS所得到的生成树。

用DFS方法得到的生成树叫做**深度优先生成树(depth-first spanningtree)**

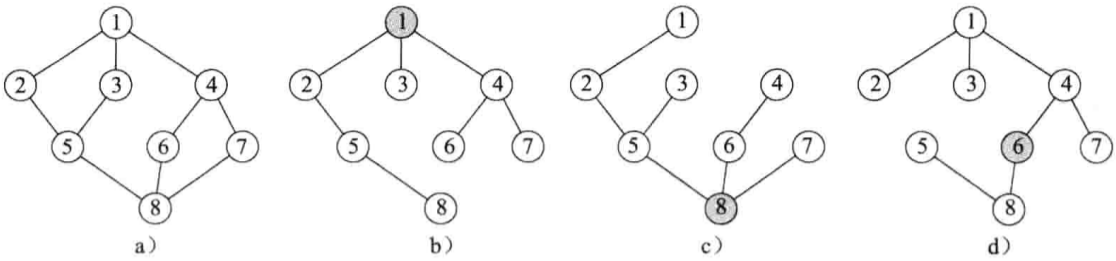


图 16-20　图及其一些广度优先生成树

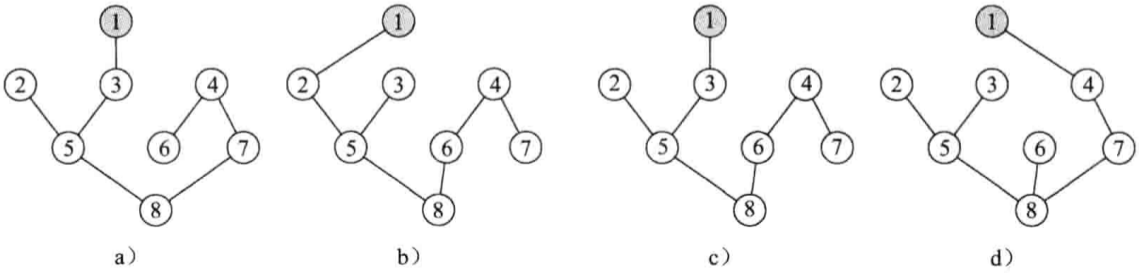


图 16-21　图 16-20a 的一些深度优先生成树

第17章 贪婪算法

在**贪婪算法**中，我们要逐步构造一个最优解。每一步，我们都要在一定的标准下，作出一个最优决策，且这个决策在后续的步骤中都不可更改。做出决策所依据的标准叫做**贪婪准则**

在有些情况下，贪婪算法都能得到最优解，但也有只能得到近似最优解的情况，通常不会与最优解偏差过大

货箱装载问题

有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样，但货箱的重量都各不相同。设第i 个货箱的重量为wi（1≤i≤n），而货船的最大载重量为c。

目的：是在货船上装入最多的货箱。

贪婪准则：从剩下的货箱中，选择重量最小的货箱

```
void containerLoading(container *c,int capacity,int
numberOfContainers,int *x)
{
    int n=numberOfContainers;
    heapSort(c,numberOfContainers);
    for(int i=1;i<=n,i++)
    {
        x[i]=0;
    }
```

```
    for(int i=1;i<=n&&c[i].weight<=capacity;i++)
    {
        x[c[i].id]=1;
        capacity-=c[i].weight;//剩余容量
    }
}
```

时间复杂度O(nlogn)

0/1背包问题

在0/1背包问题中，需对容量为c 的背包进行装载。从n 个物品中选取装入背包的物品，每件物品i 的重量为wi ，价值为pi 。

可行的背包装载：背包中物品的总重量不能超过背包的容量

约束条件为$\sum wixi <= c和xi \in [0,1](1 \leq i \leq n)$。

最佳装载是指所装入的物品价值最高，即取得最大值。

贪婪准则：从剩余的物品中，选出可以装入背包的价值最大的物品。

利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。

这种策略不能保证得到最优解。

拓扑排序

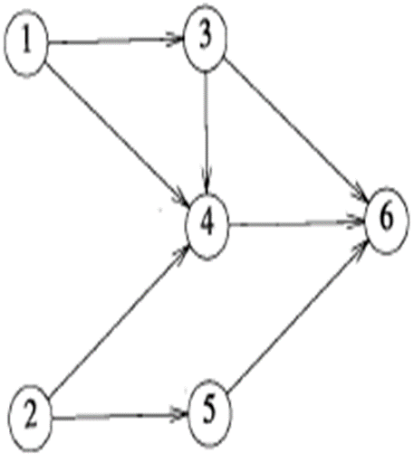
一个复杂的工程通常可以分解成一组简单任务(活动)的集合，完成这些简单任务意味着整个工程的完成。

任务之间具有先后关系。

顶点活动网络(AOV—Activity on vertex network)：任务的集合以及任务的先后顺序

顶点：表示任务(活动)

有向边(i, j)：表示任务间先后关系——任务j 开始前任务i 必须完成。



拓扑序列(Topological orders/topological sequences):

满足：对于在任务的有向图中的任一边 (i,j)，在序列中任务*i* 在任务*j* 的前面

拓扑排序(Topological Sorting):

根据任务的有向图建立拓扑序列的过程

```
bool topologicalOrder(int *theOrder)
{ //求有向图中顶点的拓扑序列;如果找到了一个拓扑序列，则返回true，此时，在theOrder[0:n-1]中记录拓扑序列;如果不存在拓扑序列，则返回false
      ……//确定图是有向图
int n=numberofVertices();
//计算入度
int *inDegree = new int [n+1];
fill(indegree+1, indegree+n+1, 0); //初始化
for (i=1; i<=n; i++) {// i的出边
vertexIterator<T> *ii=iterator(i);
int u ;
while ((u=ii->next())!=0) {
inDegree[u]++;}
}

//把入度为 0 的顶点压入栈
arrayStack<int> stack;
for (i = 1; i <= n; i++)
if (!inDegree[i]) stack.push(i);
// 生成拓扑序列
j = 0; // 数组theOrder 的索引
while (!stack.empty()) // 从堆栈中选择
    {int nextVertex= stack.top(); // 从栈中提取下一个顶点
stack.pop();
theOrder[j++] = nextVertex;
//更新nextVertex邻接到的顶点的入度
```

```
vertexIterator<T> *inextVertex =iterator(nextVertex);
int u;
while (u= inextVertex->next())!=0)
{ inDegree[u]--;
  If   (inDegree[u]==0)    stack.push(u);
}
}

return (j == n);
}
```

•使用(耗费)邻接矩阵描述：

$\Theta(n^2)$

•使用邻接链表描述:

$\Theta(n+e)$

单源最短路径(Dijkstra算法)

对于给定的源顶点sourceVertex，需找出从它到图中其他任意顶点(称为目的)的最短路径。

假设:

边的长度(耗费) >= 0.

没有路径的长度< 0.

贪婪准则：从一条最短路径还没有到达的顶点中，选择一个可以产生最短路径的目的顶点，也就是按路径长度的递增顺序产生最短路径

使用数组predecessor，令predecessor[i]是从源顶点到达顶点i前面的哪个顶点

便于按长度递增顺序产生最短路径，我们定义distanceFromSource[i]是在已生成的最短路径商扩展一条最短边而从到达顶点i时这条最短边的长度

算法思想:

1. 初始化distanceFromSource,predecessor数组，确定从源顶点可以直接到达的顶点以及其距离，并将其放入newReachableVertices数组内
2. 从newReachableVertices数组中找到下一个最小可以到达的顶点v
3. 检验可以通过v顶点可以到达的顶点距离是否比distanceFromSource短，如果是，更新数组，对于没有访问过的顶点，要将其插入newReachableVertices数组内

4. 重复2，3步，直到没有可以访问的新顶点

```
void shortestPaths(int sourceVertex,T *distanceFromSource,int *predecessor)
{
    if（souceVertex<1||sourceVertex>n)
        throw illegalParameterValue("Invalid source vertex");
    graphChain<int> newReachableVertices;
    //初始化
    for(int i=1;i<=n;i++)
    {
        distanceFromSource[i]=a[sourceVertex][i];
        if(distanceFromSource[i]==noEdge)
            predecessor[i]=-1;
        else
        {
            predecessor[i]=sourceVertex;
            newReachableVertices.insert(0,i);
        }
    }
    distanceFromSource[sourceVertex]=0;
    predecessor[sourceVertex]=0;//源顶点没有前驱
    //更新distanceFromSource和predecessor
    while（! newReachableVertices.empty())
    {//还存在更多的路径
        //寻找distanceFromSource值最小的，还未到达的顶点v
        chain<int>::iterator iNewReachableVertices=newReachableVertices.begin();
        chain<int>::iterator theEnd=newReachableVertices.end();
        int v=*iNewReachableVertices;
        iNewReachableVertices++;
        while(iNewReachableVertices!=theEnd)
        {
            int w=*iNewReachableVertices;
            iNewReachableVertices++;
            if(distanceFromSource[w]<distanceFromSource[v])
                v=w;
        }
        //下一条最短路径是达到顶点v
        //从newReachableVertices删除顶点v，然后更新distanceFromSource
        newReachableVertices.eraseElement(v);
        for(int j=1;j<=n;j++)
        {
```

```
            if(a[v][j]!=noEdge&&
(predecessor[j]==-1||distanceFromSource[j]>distanceFromSource[v
]+a[v][j]))
            {
                distanceFromSource[j]=distanceFromSource[v]+a[v]
[j];
                if(predecessor[j]==-1)
                    newReachableVertices(0,j);
                predecessor[j]=v;
            }
          }
      }
}
```

复杂性分析 $O(n^2)$

最小成本生成树

在n个顶点的无向网格G中，每棵生成树都刚好有n-1条边，现在的问题是如何选择n-1条边使它形成G的最小生成树

限制条件：所有的边构成一个生成树。

优化函数：子集中所有边的权值之和。

最小生成树

具有*n*个顶点的无向(连通)网络G的每个生成树刚好具有*n*-1条边。

最小耗费生成树问题是用某种方法选择n-1条边使它们形成*G*的**最小生成树**。

三种求解该问题的贪婪思想：

Kruscal算法

算法思想：

开始，初始化含有 n个顶点 0条边的森林.

Kruskal算法所使用的贪婪准则是：从剩下的边中选择一条**不会产生环路**的**具有最小耗费**的**边**加入已选择的边的集合中。

Kruskal算法分*e*步(*e*是网络中边的数目)。

按耗费递增的顺序来考虑这*e*条边，每次考虑一条边。

当考虑某条边时，若将其加入到已选边的集合中会出现环路，则将其抛弃，否则，将它选入。

```
伪代码：
初始化所选边集T=空集
E为网格中的边集
while(E!=空集&&|T|!=n-1)
{
    从E中找出耗费最小的边(U,V);
    if(T加入(u,v)后不成环) 加入(u,v);
}
if(|T|=n-1) T是最小耗费生成树
else 网格不是连通的，不能找到生成树
```

数据结构选择：边集E 小根堆，被选边集用数组来表示

复杂度分析 设有n个顶点，e条边

使用并查集

初始化：O(n)

find操作的次数最多为2*e*，Unite操作的次数最多为*n*-1(若网络是连通的，则刚好是*n*-1次)。

比O(n+e)稍大一点。

使用边的最小堆，按耗费递增的顺序来考虑*e*条边：O(eloge).

$O(n + eloge)$

Prim算法

贪婪准则：从剩余的边中，选择一条成本最小的边，并且把它加入已选的边集中形成一棵树

```
伪代码：
令T是已入选的边集，初始化T=空集
令TV是已在树中的顶点集，TV={1}
令E是网格的边集
while(E!=空&&|T|!=n-1)
{
令(u,v)是一条成本最小的边，且u∈TV,v不属于TV
if(没有这样的边) break;
E=E-{u,v};
把边{u,v}加入T
把顶点v加入TV
}
if(|T|=n-1) T是一棵最小生成树
```

```
14  else 网格不连通，没有最小生成树
```

Solin算法

Kruscal算法和Prim算法的结合，从多个点并发生成最小生成树

算法思想.

从含有 n个顶点的森林开始.

每一步中为森林中的每棵树选择一条边，这条边刚好有一个顶点在树中且边的代价最小。将所选择的边加入要创建的生成树中。

一个森林中的两棵树可选择同一条边。

当有多条边具有相同的耗费时，两棵树可选择与它们相连的不同的边。

丢弃重复的边和构成环路的边。

直到仅剩下一棵树或没有剩余的边可供选择时算法终止。

第18章 分而治之

算法思想：

1. 将问题分解成两个或多个更小的问题
2. 分别解决小问题
3. 把各小问题的解答组合起来，即可得到原来问题的解

归并排序

算法思想：把n个元素按非递减顺序排列。若n为1，则算法终止；否则，将序列划分为k个子序列，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。

二路划分：将n个元素的序列仅仅划分为两个子序列

几种划分思想：

- 把前n-1个元素放到第一个子序列中，最后一个元素放入第二个字序列中，在第一个子序列排序后，使用插入函数将两个子序列归并。这就是插入排序的递归形式时间复杂度$O(n^2)$
- 将最大的元素放入第一个子序列中，剩下n-1个元素放入第二个子序列中，第二个子序列排序后，将第一个子序列放到第二个子序列后。如果使用冒泡函数来寻找关键字的最大值，那么这种划分就是冒泡排序的递归形式；如果使用Max函数来寻找关键字的最大值，这种划分就是选择排序的递归形式。两种方法的时间复杂度都是$O(n^2)$

- 平衡分割法的情况：

  A:含有**n/k** 个元素

  B:其余的元素

  递归地使用分而治之方法对*A*和*B*进行排序

  将排好序的*A*和*B*归并为一个集合。

可以证明，在两个较小实例大小接近相等时，算法的时间复杂度最小，最好、最坏、平均复杂度均为$\Theta(nlogn)$

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

  从a归并到b，从b归并到a，其实消除了从b到a的复制过程

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

为了完成排序代码，需要函数mergePass，这个函数仅用于确定归并子序列的左右边界。实际归并由函数merge完成

```
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;
    while(i<=n-2*segmentSize)
    {
        merge(x,y,i,i+segmentSize-1,i+2*segmentSize-1);
        i+=segmentSize*2;
```

```
    }
  //少于两个满数据段
    if(i+segmentSize<n)
        merge(x,y,i,i+segmentSize-1,n-1);//剩余两个数据段
    else//剩余一个数据段
        for(int j=i;j<n;j++)
        {
            y[j]=x[j];
        }
}
```

```
template<class T>
    void merge(T c[],T d[],int startOfFirst,int endOfFirst,int endOfSecond)
{//把两个数据段从c归并到d
    int first=startOfFirst;
    second=endOfFist+1;
    result=startOfFirst;//用于归并数据段的索引
    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];
}
```

自然归并排序

首先从左至右扫描序列元素，如果位置i的元素比位置i+1大，则位置i就是一个分割点，以此确定输入序列中以及存在的有序段。

然后归并这些有序段，直到剩下一个有序段

最好情况 序列已经有序 $O(n)$

最坏情况 被认为与直接归并排序相同 $O(nlogn)$

只有在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

快速排序

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，并且排序后不需要归并。middle的元素为**支点(pivot)**或**分割元素(partitioning element)**

简单描述:

- 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。使左段元素关键字不大于支点，右段元素关键字不小于支点
- 对左段进行快速排序
- 对右段进行快速排序

快速排序函数quickSort把数组a的最大元素移动到数组的最右端，然后调用递归函数quickSort执行排序。

驱动程序

```
template<class T>
    void quickSort(T a[],int n)
{
    if(n<=1) return;
    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
    if(leftEnd>=rightEnd) return;

    int leftCursor=leftEnd,
    rightCursor=rightEnd+1;
    T pivot=a[leftEnd];
    //将位于左侧不小于支点的元素或位于右侧不大于支点的元素交换
    while(true)
    {
        do
        {//寻找左侧不小于pivot的元素
```

```
            leftCursor++;
        }while(leftCursor<pivot);
        do
        {//寻找右侧不大于pivot的元素
            rightCursor--;
        }while(rightCursor>pivot);
        if(leftCursor>rightCursor) break;
        swap(a[leftCursor],a[rightCursor]);
    }
    a[leftEnd]=a[rightCursor];
    a[rightCursor]=pivot;

    quickSort(a,0,rightCursor-1);//对左侧的数段进行排序
    quickSort(a,rightCursor+1,rightEnd);//对右侧的数段进行排序
}
```

时间复杂度

最坏情况下，left段总是空 $\Theta(n^2)$

最好情况下，right段和left段元素个数大致相同 $\Theta(nlogn)$

平均复杂度也是 $\Theta(nlogn)$

三值取中快速排序

在三元素a[leftEnd],a[rightEnd],a[(leftEnd+rightEnd)/2]中选取大小居中的元素作为支点元素

各种排序算法的时间复杂度

方法	最坏复杂性	平均复杂性
冒泡排序	n^2	n^2
基数排序	n	n
插入排序	n^2	n^2
选择排序	n^2	n^2
堆排序	nlogn	nlogn
归并排序	nlogn	nlogn
快速排序	n^2	nlogn

选择

从n元素数组a[0：n-1]中找出第k小的元素

思路：

对a进行排序，a[n-k]就是第k小的元素，使用快速排序的话 平均复杂度为$\Theta(nlogn)$

通过修改上面的快速排序的代码，可以获得较快的求解方法

预处理程序

```
template<class T>
    T quickSort(T a[],int n,int k)
{
    if(k<1||k>n)
    throw illegalParameterValue("k must be between 1 and n");
    int max=indexOfMax(a,n);
    swap(a[n-1],a[max]);
    return select(a,0,n-1,k);
}
```

寻找第k小的元素的递归函数

```
template<class T>
T select(T a[],int leftEnd,int rightEnd,int k)
{//寻找第k大的元素
    if(leftEnd>=rightEnd) return a[leftEnd];

    int leftCursor=leftEnd,
    rightCursor=rightEnd+1;
    T pivot=a[leftEnd];
    //将位于左侧不小于支点的元素或位于右侧不大于支点的元素交换
    while(true)
    {
        do
        {//寻找左侧不小于pivot的元素
            leftCursor++;
        }while(leftCursor<pivot);
        do
        {//寻找右侧不大于pivot的元素
            rightCursor--;
        }while(rightCursor>pivot);
        if(leftCursor>rightCursor) break;//交换的一对元素没有找到
        swap(a[leftCursor],a[rightCursor]);
    }
```

```
    if(rightCursor-leftEnd+1==k)
        return pivot;
    a[leftEnd]=a[rightCursor];
    a[rightCursor]=pivot;
    //对一个数据段调用递归
    if(rightCursor-leftEnd+1<k)
    {
        return select(a,rightCursor+1,rightEnd,k-rightCursor+leftEnd-1);//在右段中寻找第k-支点是第几大的
    }
    else return select(a,leftEnd,rightCursor-1,k)//左段中寻找第k大的点
}
```

第19章 动态规划

在动态规划中，要考察一系列决策，已确定最优决策序列是否包含最优决策子序列。

当最优决策包含最优决策子序列时，可以建立**动态规划递归方程**，它可以帮助我们高效的解决问题

0/1背包问题

在0/1背包问题中，需对容量为c 的背包进行装载。从n 个物品中选取装入背包的物品，每件物品i 的重量为wi ，价值为pi

假设

$$f(i,y)表示剩余容量为y，剩余物品为i,i+1,\ldots,n的背包问题的最优解的值$$

$$f(n,y)=\begin{cases}p_n & \mathrm{y}\geq w_n\\ 0 & 0\leq y<w_n\end{cases}$$

$$f(i,y)=\begin{cases}max(f(i+1,y),f(i+1,y-w_i)+p_i & \mathrm{y}\geq w_i\\ f(i+1,y) & 0\leq y<w_i\end{cases}$$

无论第一次的选择是什么，接下来的选择一定是当前状态下的最优解，我们称此为**最优原则(principle of optimality)**

所有顶点对最短路径

1. **问题**

在n个顶点的有向图G中，寻找每一对顶点之间的最短路径，即对于每对顶点(i,j)，需要寻找从i到j的最短路径及从j到i的最短路径，对于无向图，这两条路径是一条。
对一个n个顶点的图，需寻找p =n(n-1)条最短路径。

2. **动态规划公式**

假定图可以由权值为负的边，但不能由带权长度为负值的环路。因此，每一对顶点(i,j)总有一条不含环路的最短路径

假设G有n个顶点，c(i,j,k)表示从顶点i到j的一条最短路径长度，其中间顶点的编号不大于k。

$$C(i,j,0)=\begin{cases}a[i][j] & (a[i][j]\text{是邻接耗费矩阵})\\ 0 & \mathrm{i}=j\\ +\infty & \mathrm{no}Edge\end{cases}$$

$$C(i,j,k)=\begin{cases}c(i,j,k-1) & \text{该路径中不含顶点}k\\ min(c(i,j,k-1),c(i,k,k-1)+c(k,j,k-1)) & \text{该路径中含顶点}k\end{cases}$$

3. **Floyd算法伪代码**

```
//初始化c
for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
        c[i][j][0]=a[i][j];
for (int k = 1; k <= n; k++)
   for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
         if (c[i][k][k-1] + c[k][j][k-1]< c[i][j][k-1] )
                     c[i][j][k] = c[i][k][k-1] + c[k][j][k-1]
          else     c[i][j][k] = c[i][j][k-1]

```

若用c(i,j) 代替c(i,j,k) ,最后所得的*c(i,j)* 之值将等于*c(i,j,n)* 值

c和kay值的计算

kay：从i到j的最短路径中最大的k值

```
template<class T>
    void allPairs(T **c,int **kay)
{
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=n;j++)
        {
            c[i][j]=a[i][j];
            kay[i][j]=0;
        }
    }
```

```
    for(int i=1;i<=n;i++)
        c[i][i]=0;
    for(int k=1;k<=n;k++)
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
            {
             if(c[i][k]!=noEdge&&c[k][j]!=noEdge&&(c[i][j]=noEdge||c[i][k]+c[k][j]<c[i][j]))
                 c[i][j]=c[i][k]+c[k][j];
                k[i][j]=k;
            }
}
```

输出最短路径

```
template<class T>
    void outputPath(T **c,int **kay,T noEdge,int i,int j)
{
    if(c[i][j]==noEdge)
        cout<<"There is no path from "<<i<<" to "<<j<<endl;
    else
    {
        cout<<"The path is"<<i<<" ";
        outputPath(kay,i,j);
        cout<<endl;
    }
}
void outputPath(int **kay,int i,int j)
{
    if(i==j)
        return;
    if(k[i][j]==0)
        cout<<j<<" ";
    else
    {
        outputPath(kay,i,k[i][j]);
        outputPath(kay,kay[i][j],j);
    }
}
```