- 第8章
- 一棵树，有n_1个度为1的节点，n_2个度为2 的节点，……，n_m个度为m的节点。有多少个叶节点？

  - 总结点数　n= n_0+ n_1+n_2+……+n_m
  - 总分支数 e=n−1= n_0+ n_1+n_2+……+n_m−1
  - =m*n_m +(m−1)*n_{m-1}+……+2*n_2+ n_1
  - 则有：

$$n_0 = (\sum_{i=2}^{m} (i-1) * n_i) + 1$$

- 8. 编写公式化描述的二叉树的前序遍历程序。假设二叉树的元素存储在数组a 中，其中,Last 用于保存树中最后一个元素的位置。当位置i 中没有元素时，a [i] = 0。给出该程序的时间复杂性。

```
template <class T>
void PreOrder(T a[], int last, int i)
   {// 前序遍历根节点在a[i]的 公式化描述的二叉树
     if (i <= last && a[i]) {
            Visit(a, i);    //访问根节点
             PreOrder(a, last, 2*i);  // 前序遍历左子树
             PreOrder(a, last, 2*i+1);  //前序遍历右子树
           }
      }
```

时间复杂度 O(n) , n为树中节点个数。

9. 按中序遍历方法完成练习8

```
template <class T>
void InOrder(T a[], int last, int i)
{//中序遍历根节点在a[i]的 公式化描述的二叉树.
   if (i <= last && a[i]) {
      InOrder(a, last, 2*i);    //中序遍历左子树
      Visit(a, i);              //访问根节点
      InOrder(a, last, 2*i+1);  //中序遍历右子树
     }
}
```

10. 按后序遍历方法完成练习8

```
template <class T>
void PostOrder(T a[], int last, int i)
{//后序遍历根节点在a[i]的 公式化描述的二叉树.
   if (i <= last && a[i]) {
      PostOrder (a, last, 2*i);    //后序遍历左子树
      PostOrder (a, last, 2*i+1);  //后序遍历右子树
      Visit(a, i);                 //访问根节点
      }
}
```

11. 按逐层遍历方法完成练习8

```
template <class T>
void LevelOrder(T a[], int last)
{//层序遍历公式化描述的二叉树.
   LinkedQueue<int> Q;
   if (!last) return; // 空树
   int i = 1; // 从根开始
```

```
while (true) {
      Visit(a, i);
      // 将孩子存入队列
      if (2*i <= last && a[2*i])
         Q.Add(2*i);       // 左孩子入队列
      if (2*i+1 <= last && a[2*i+1])
         Q.Add(2*i+1);  //右孩子入队列
      // 得到下一个要访问的结点
      try {Q.Delete(i);}
      catch (OutOfBounds) {return;}
      }
 }
```

- 13．编写两个C++函数，复制用BinaryTreeNode模板结构描述的二叉树t。第一个函数按前序遍历整棵树，第二个按后序遍历。两个函数所需要到的递归栈空间有什么不同?

前序：

```
template <class T>
BinaryTreeNode<T>* PreCopy(BinaryTreeNode<T> *t)
{// 使用前序遍历复制一颗二叉树
 // 返回指向新树根节点的指针
   if (!t) // t 是空树
      return 0;
   BinaryTreeNode<T>  *root;

   root=new BinaryTreeNode<T>(t->data);
   root->LeftChild=PreCopy(t->LeftChild);
   root->RightChild=PreCopy(t->RightChild);

  return root;
}
```

后序：

```
template <class T>
BinaryTreeNode<T>* PostCopy(BinaryTreeNode<T> *t)
{// 使用后序遍历复制一颗二叉树
 // 返回指向新树根节点的指针
   if (!t) // t是空树
      return 0;
   BinaryTreeNode<T> *left, *right, *root;
   left = PostCopy(t->LeftChild);
   right = PostCopy(t->RightChild);
   root = new BinaryTreeNode<T> (t->data, left, right);}
   return root;
}
```

两个函数所需要到的递归栈空间: O(*h*),h是被复制的二叉树的高度。

15. 编写一个函数删除二叉树t (提示：采用后序遍历方法)。
假设t 是一链表树，调用C + +函数delete释放节点空间。

```
template <class T>
void Erase(BinaryTreeNode<T> * &t)
{// 删除*t中所有的节点.
 // 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;              // 删除根节点
      t = 0;
      }
}
```

- 19. 设*t* 是数据域类型为int 的二叉树，每个节点的数据都不相同。数据域的前序和中序排列是否可唯一地确定这棵二叉树? 如果能，给出一函数来构造此二叉树。指出函数的时间复杂性。

```
Template <class T>
BinaryTreeNode<T> *CreateBinary(T *Prelist, T *Inlist, int n) {
// Prelist是二叉树的前序序列, Inlist是二叉树的中序序列,函数返回二叉树根指针
   If （n= =0） return NULL;
   Int  k=0;
   While(Prelist[0]!=Inlist[k]) k++;
   BinaryTreeNode<T> *t=new BinaryTreeNode<T> (Prelist[0]);
   t->LeftChild= CreateBinary(Prelist+1, Inlist, k);
   //从前序Prelist+1开始对中序的0~k-1左子序列的k个元素递归建立左子树
   t->RightChild= CreateBinary(Prelist+k+1, Inlist+k+1, n-k-1);
//从前序Prelist+k+1开始对中序的k+1~n-1右子序列的n-k-1个元素递归建立右子树
   return t;
}
```

- 31. 1)扩充抽象数据类型*BinaryTree*，增加*Compare*(*X*)操作，用来比较该二叉树与二叉树*X*。若两棵二叉树相同返回true，否则返回false。
- 2)扩充C++类BinaryTree，增加用于二叉树比较的共享成员函数，并测试代码。

在C++类BinaryTree中增加public函数

```
bool   Compare(const BinaryTree<T>& t)
        {return Compare(root, t.root);}
```

在C++类BinaryTree中增加private函数

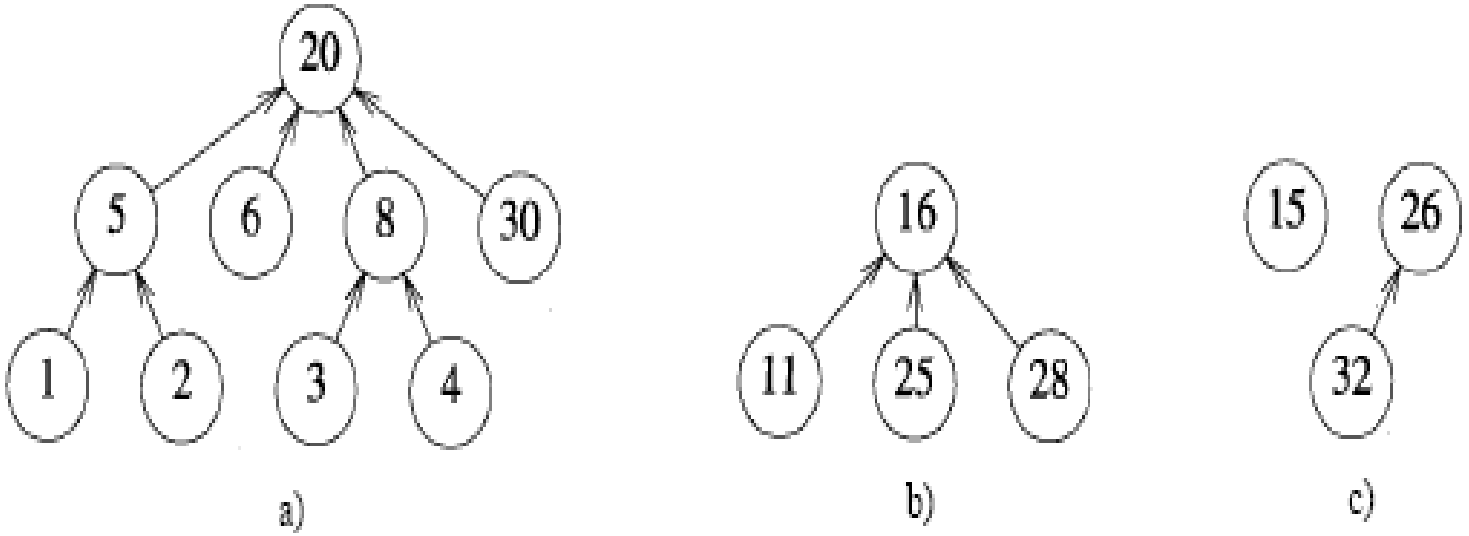
```
template <class T>
bool BinaryTree<T>::Compare(BinaryTreeNode<T> *t,
    BinaryTreeNode<T> *u)
{// 使用前序遍历比较两棵二叉树t 和 u.
// 当t 和 u相同返回true，否则返回false。

if (!t && !u) return true;  // 两棵都是空树

if (!t || !u) return false; // 一棵是空树

//两棵都不为空
if (t->data != u->data) return false; // 数据不同
return  Compare(t->LeftChild, u->LeftChild)   &&
          Compare(t->RightChild, u->RightChild); }
```

- 36. 森林（forest）是一棵或多棵树的集合。在树的二叉树描述中，根没有右孩子。由此可以用二叉树来描述有*m* 棵树的森林。首先得到树林中每棵树的二叉树描述，然后，第*i* 棵作为第*i*- 1棵树的右子树，画出图8 - 15中有4棵树的森林的二叉树描述，同时还有图8 - 17和图8 - 18的二棵树的森林。



- 已知一棵二叉树的中序、后序序列分别如下：
- 中序：D C E F B H G A K J L I M
- 后序：D F E C H G B K L J M I A
- 要求：
- ⑴ 画出该二叉树；
- ⑵ 写出该二叉树的先序序列。
- (3)画出该二叉树对应的森林；
-

- 第**9**章
- 8．在MaxHeap类中加入一个共享成员函数ChangeMax(x)，将当前最大元素改为元素x，x的值可以大于或小于当前最大元素的值，与删除最大元素的操作一样，代码沿着从根开始的一条向下的路径遍历。执行过程中当最大堆为空时，引发一个OutOfBounds异常，代码的复杂性应为O(log*n*)，其中*n*是最大堆中元素个数，证明这个结论。

```
Template<class T>
MaxHeap<T>& MaxHeap<T>::ChangeMax(T &x){
    If(currentSize==0) throw OutOfBounds();
    //将当前最大元素改为元素x
    Heap[1]=x;
    //重构最大堆
    int i=1;
    ci=2;
    while(ci<=CurrentSize){
        if(ci<CurrentSize&&heap[ci]<heap[ci+1]) ci++;
        heap[i]=heap[ci];
        i=ci;
        ci*=2;
        }
    return *this;
}
```

- 9. 由于在删除操作（见程序9 - 4）中重新插入最大堆的元素y是从堆的底部移出的那个元素，我们期望它仍插在接近底部的地方。重新写一个DeleteMax函数，让根节点的空位置先移到叶节点，然后y通过这个叶节点往上找到它的合适位置。通过实验比较新代码是否比旧代码执行速度快。

```
template<class T>
MaxHeap<T>& MaxHeap<T>::DeleteMax(T& x)
{// 从堆中删除最大元素并存入x
// 检查堆是否为空
 if (CurrentSize == 0)  throw OutOfBounds(); // 空
 x = heap[1]; // 最大元素
//重构堆
 T y = heap[CurrentSize--]; // 最后一个元素
```

```
//让根节点的空位置先移到叶节点
int i = 1; // 堆的当前位置
ci = 2; //  i的孩子
while (ci <= CurrentSize)
    { // heap[ci] 是 i的较大孩子
    if (ci < CurrentSize && heap[ci] < heap[ci+1])
       ci++;       // 移动较大孩子到heap[i]
     heap[i] = heap[ci]; // 孩子上移一层
     i = ci
     ci *= 2;
    }
```

```
 i = ci/2;   // 空位置在heap[i], 从这里开始插入y
while (i != 1 && y > heap[i/2])
   { // 不能将y 放入 heap[i]
     heap[i] = heap[i/2]; // 元素下移
      i /= 2;                  // i移到父节点
   }
heap[i] = y;
return *this;
}
```

- 16. 一种稳定的排序算法是指具有相同值的记录在排序前与排序后具有相同的顺序。假设记录3与记录10的关键值相同，对于一个稳定排序，排序后记录3仍在记录10的前面。请问堆排序是稳定排序吗？插入排序呢？

  堆排序不是稳定排序
  插入排序是稳定排序

不稳定： 直接选择排序，快速排序，堆排序.
稳定： 直接插入排序，气泡排序，归并排序，基数排序

- 23．run是一个有序元素序列。假设两个run可以在(r+s)时间内合并为一个run，其中r与s分别为要合并的两个run的长度。通过不断地合并两个run，可将n个不同长度的run最终合并成一个。解释如何运用霍夫曼树来合并n个run，以使时间开销最小。

- 一组序列的关键码为：

  {28、19、27、49、56、12、10、25}

  该序列是否是堆？若不是，写出建立的初始堆（最大堆）。

  利用堆排序的方法对该序列进行非递减排列，给出堆排序的主要过程。

  删除一元素后所得到的堆结构。

  给出在初始堆中插入一新元素60后的堆结构。

- 在一段文字中，7个常用汉字及出现频度如下：

  的　地　于　个　和　是　有

  20　19　18　17　15　10　1

要求：

⑴ 画出对应的Huffman树；

⑵ 求出每个汉字的Huffman编码。

- 第11章
- 10. 二叉搜索树可用来对n个元素进行排序。编写一个排序过程，首先将n 个元素a [1 : n]插入到一棵空的二叉搜索树中，然后对树进行中序遍历，并将元素按序放入数组a 中。为简单起见，假设a 中的数是互不相同的。将此过程的平均运行时间与插入排序和堆排序进行比较。

```
template <class T>;
Void puteleinarray (BinaryTreeNone<T>  *t)
        {a[k]= t->data;  k++;}
void Sort(T a[], int n)
    {// 对a[1:n] 进行排序
    // 创建一个二叉搜索树
    int k=1;
    BSTree<T>   bst;
    for (int i=1; i <= n; i++)
    bst.Insert（a[i]）;
    bst.Inorder(puteleinarray);
    }
```

- 11. 编写一个从二叉搜索树中删除最大元素的函数，函数的时间复杂性必须是O (*h*)，其中*h*是二叉搜索树的高度。

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::DeleteMax(E& e)
{// 删除关键值最大的元素，存入e.
// 如果树空，则引发OutOfBounds异常.
if (!root) throw OutOfBounds();  // 空树
// p指向最大元素
BinaryTreeNode<E>   *p = root, // 搜索指针
                    *pp = 0;   // p的父结点
while (p->RightChild)//寻找最右下的元素
      {  pp = p;      p = p->RightChild; }
e = p->data;  // 存最大元素   /
/ 从树中删除p
// p 最多有一个孩子
 if (p == root)   root = p->LeftChild;
          else pp->RightChild = p->LeftChild;
delete p;
 return *this;}
```

- 31．如果每个节点占用2个磁盘块并且需要2次磁盘访问才能搜索出来，那么在一棵2*m*序B-树的搜索过程中需要的最大磁盘访问次数是多少？将该次数与节点大小占1个磁盘块的*m* 序B-树的磁盘访问次数相比较，并论述节点大小大于磁盘块大小时的优点。

- 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：
- $\log_m(n+1) \leqslant h \leqslant \log_d((n+1)/2)+1$
-
- 在一棵2m序的B－树中，因为最坏情况下，树的高度是 $\log_m((n+1)/2) + 1$。因为每一个节点的搜索需要2次磁盘访问，所以最大磁盘访问次数是$2\log_m((n+1)/2) + 2$.
- 在一棵m序的B－树中，在最坏情况下，树的高度是$\log_d((n+1)/2) + 1$, 其中 d=⌈*m*/2⌉因为每个节点的搜索需要一次磁盘访问，所以最大磁盘访问次数是
- $\log_d((n+1)/2) + 1 = \log_m((n+1)/2) * (\log m / \log d) + 1$
-
- 因为$(\log m / \log d) <= 2$ ， $m >= 3$，考虑搜索复杂度，用m序B－树比用2m序的B－树要好。

- 32. 删除一个*m* 序B-树中非树叶节点的元素需要的最大磁盘访问次数是多少？

- $h \leqslant \log_d((n+1)/2)+1$; $d=\lceil m/2 \rceil$
- 删除一个m序B－树中非树叶节点的元素需要的最大磁盘访问次数：3h
- $3\log_d((n+1)/2)+3$;

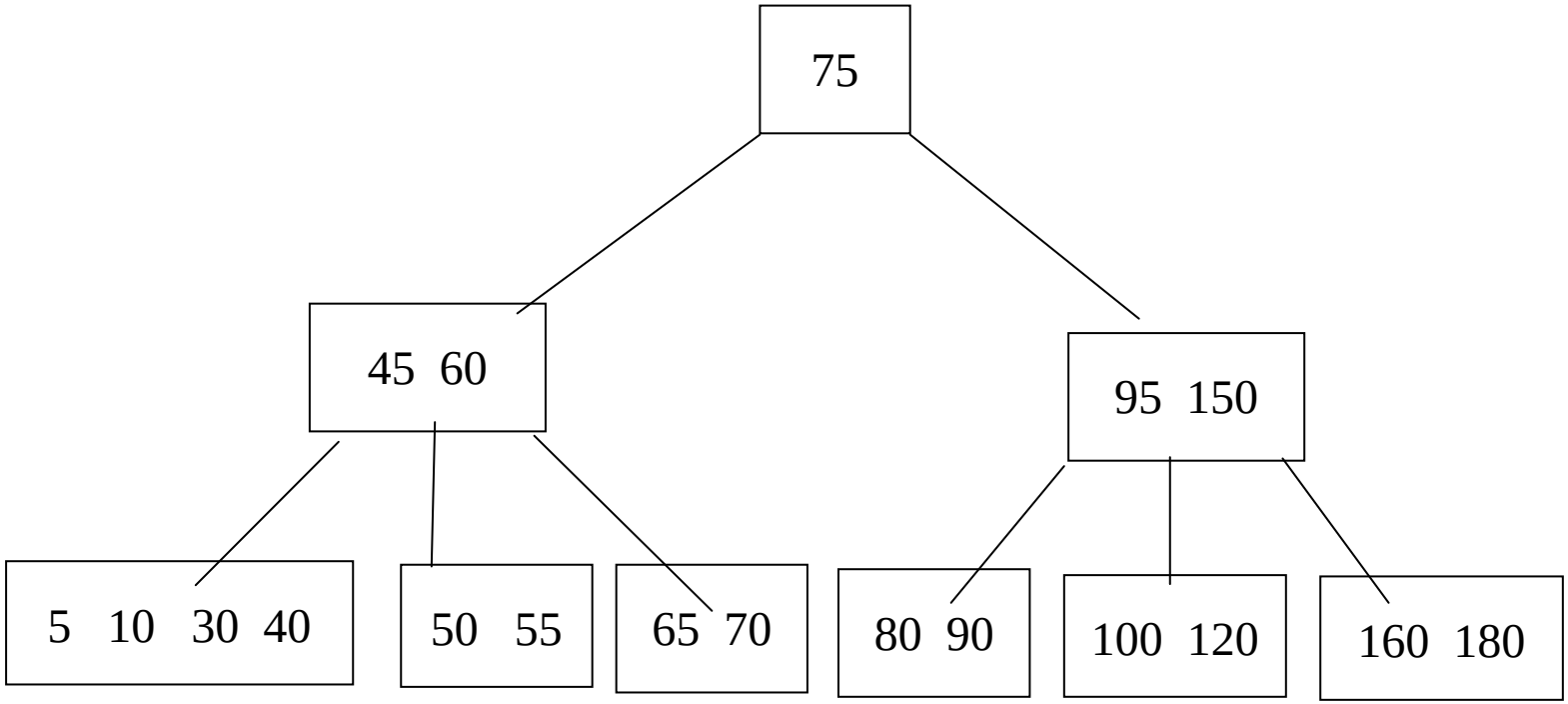
- 输入一个正整数序列{100，50，280，450，66，200，30，260}，建立一棵二叉搜索树，要求：

  ⑴ 画出该二叉搜索树；并指出该二叉树是否为平衡二叉树？

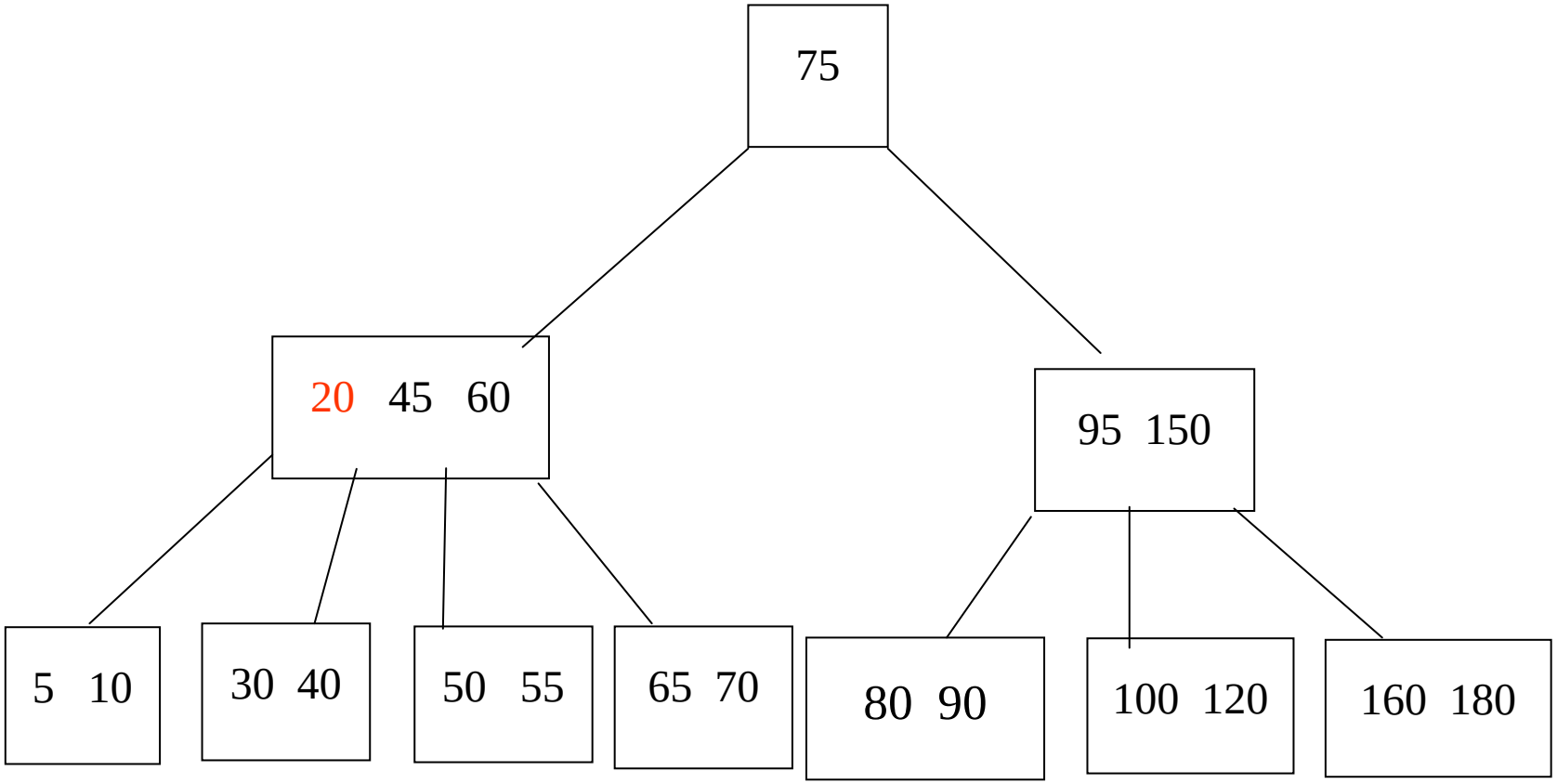
  ⑵ 画出删除结点280后的二叉搜索树。

- 将关键码16，3，7，11，9，26，18，14，15依次插入到一棵初始为空的AVL树中，画出每插入一个关键码后的AVL树。

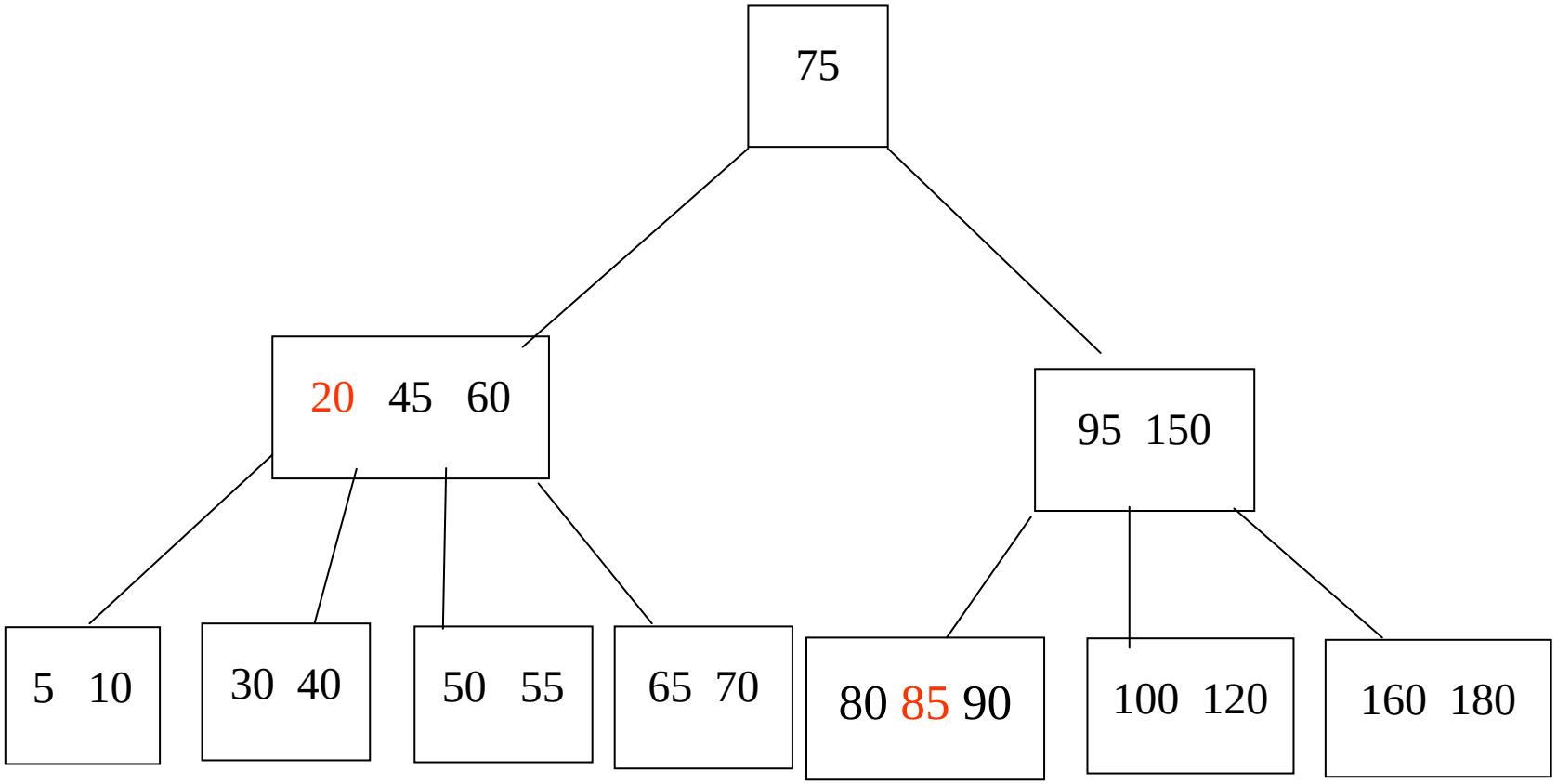
- 在下列5序B-树中首先依次插入关键字20，85，然后依次删除关键字120，75，10，画出每次插入或删除元素后的B-树。



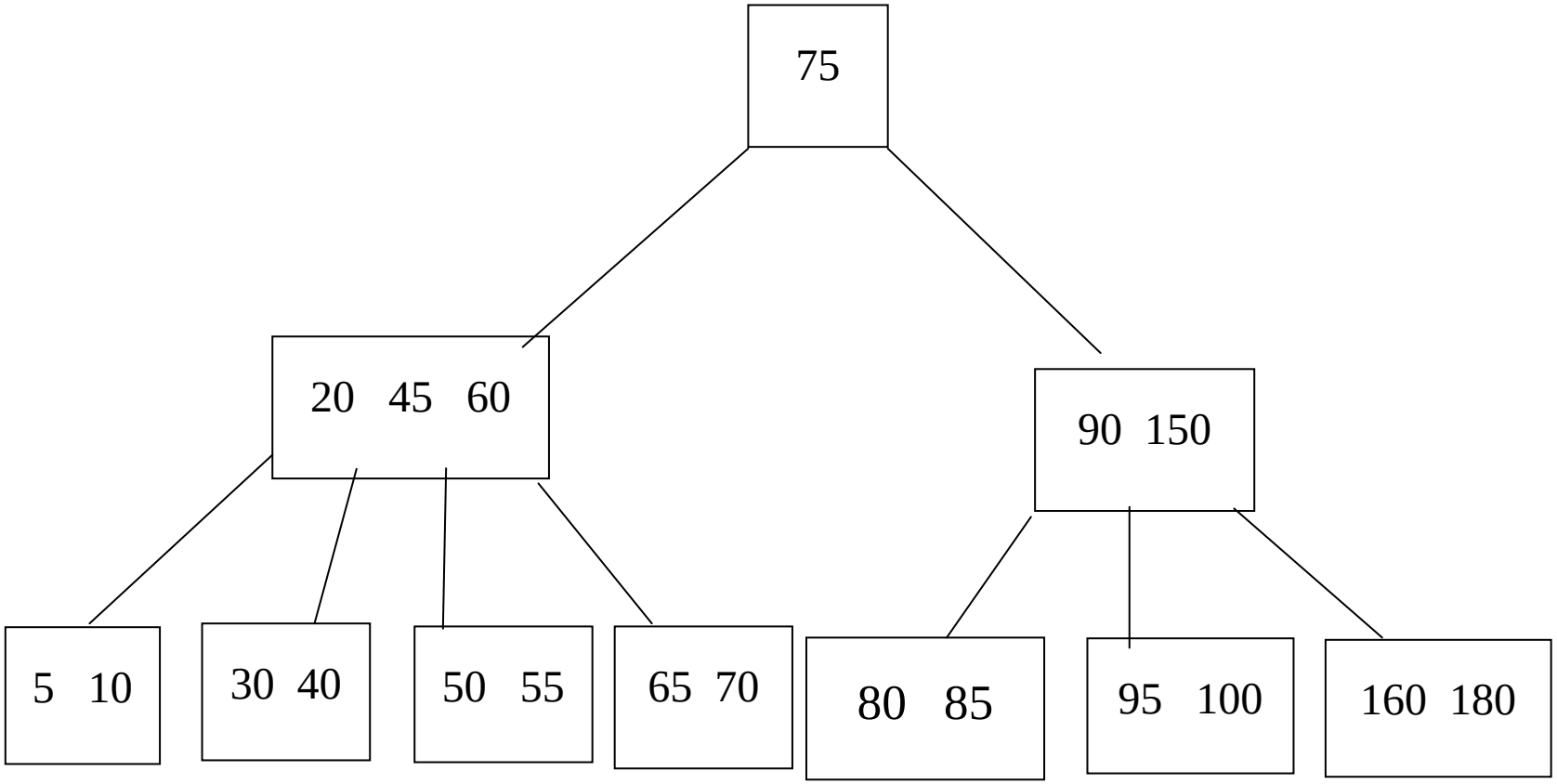
- 插入20



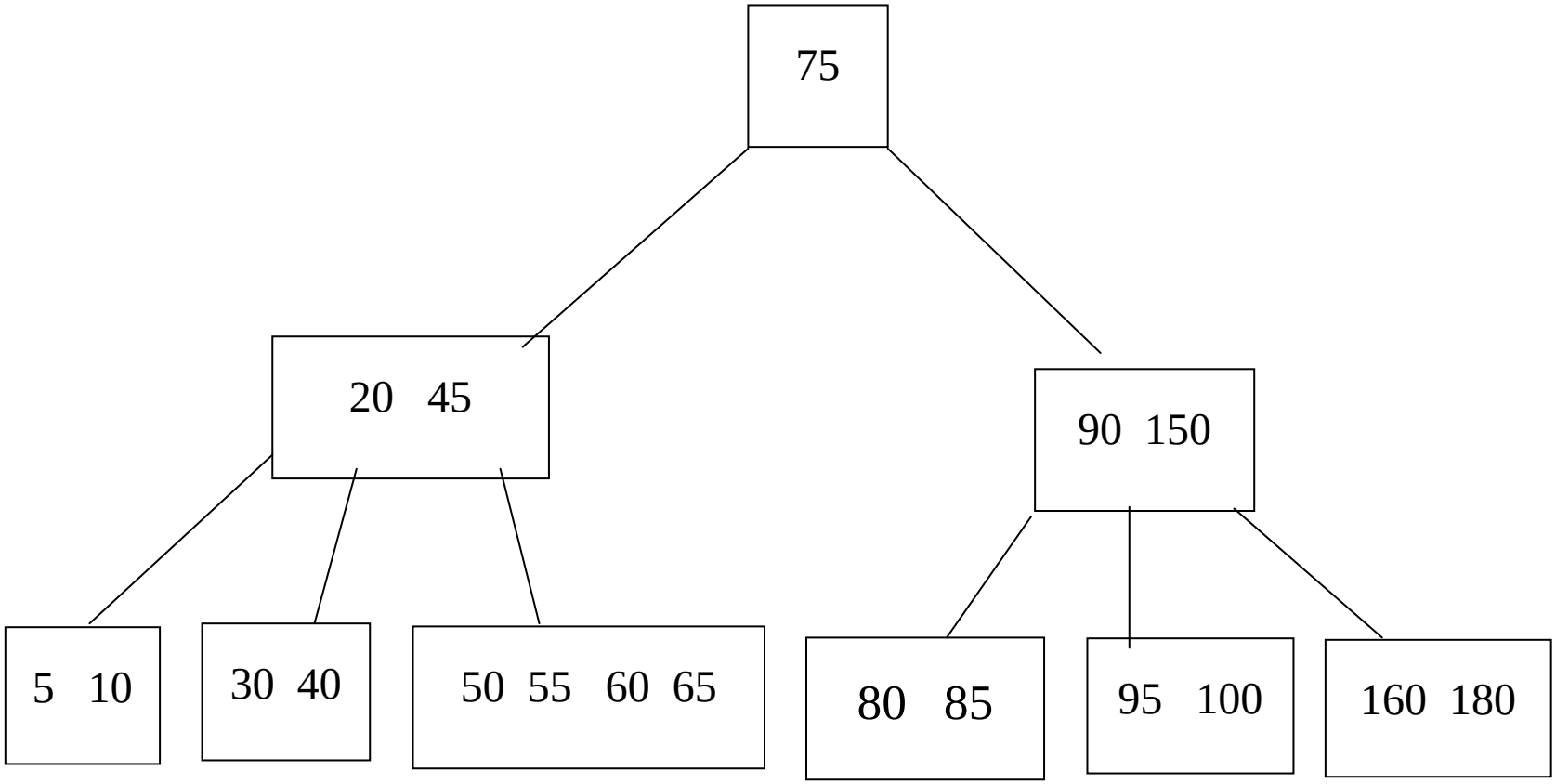
- 插入85



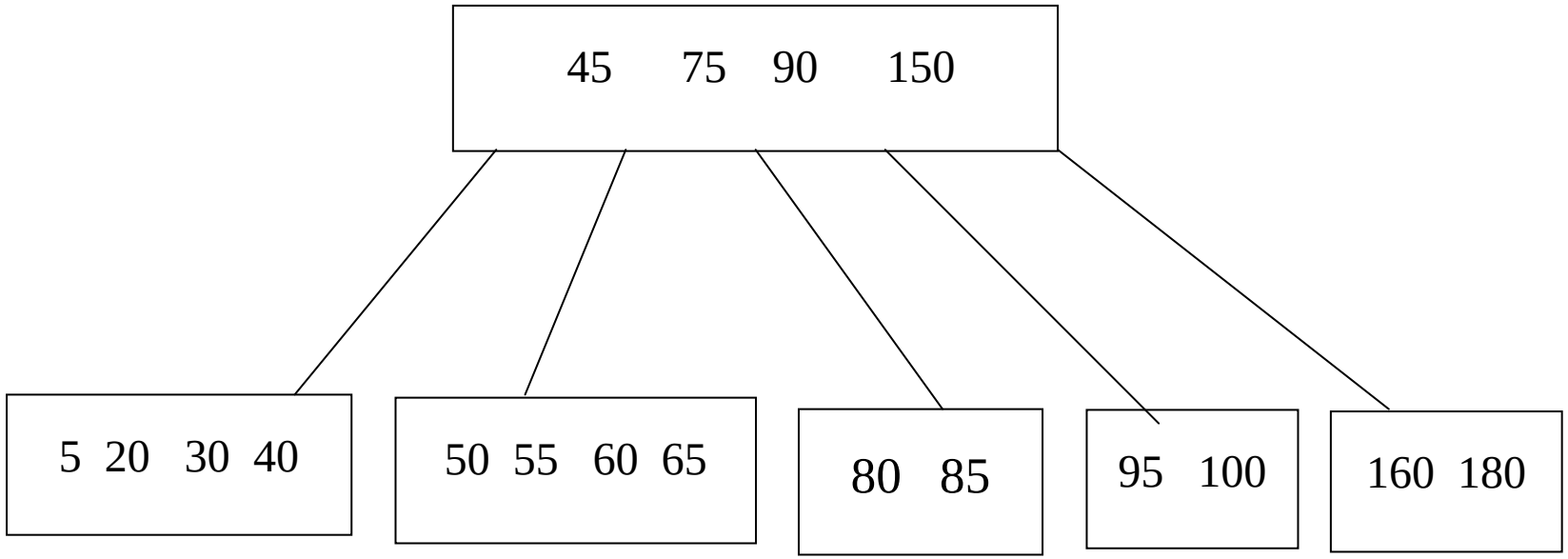
- 删除120



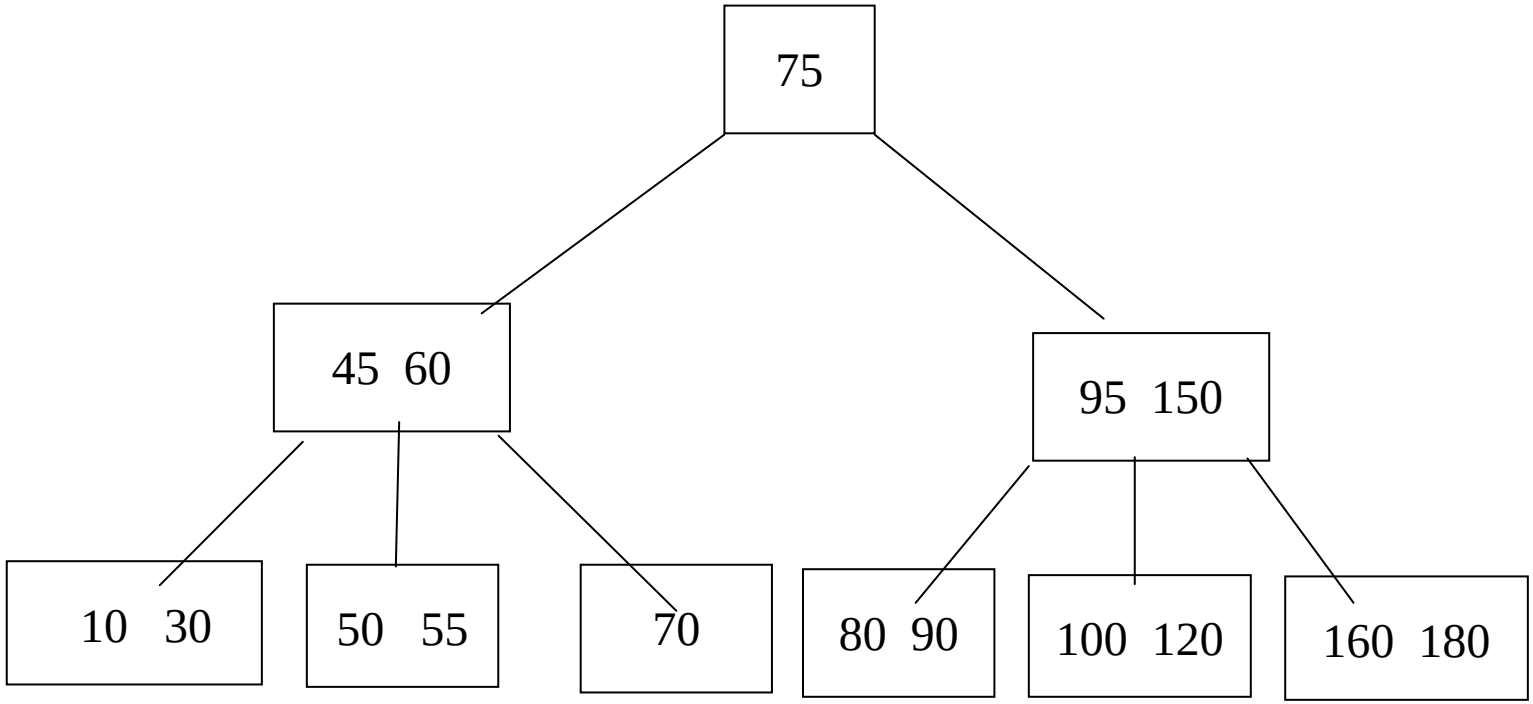
- 删除70



- 删除10



- 在下列3序B-树中首先依次插入关键字20，85，然后依次删除关键字120，75，30，画出每次插入或删除元素后的B-树。



第12章---第15章

1．对于图1 2 - 8的每一个有向图，确定下列各项：

1) 每个顶点的入度。

2) 每个顶点的出度。

3) 邻接于顶点2的顶点集合。

4) 邻接至顶点1的顶点集合。

5) 关联于顶点3的边的集合。

6) 关联至顶点4的边的集合。

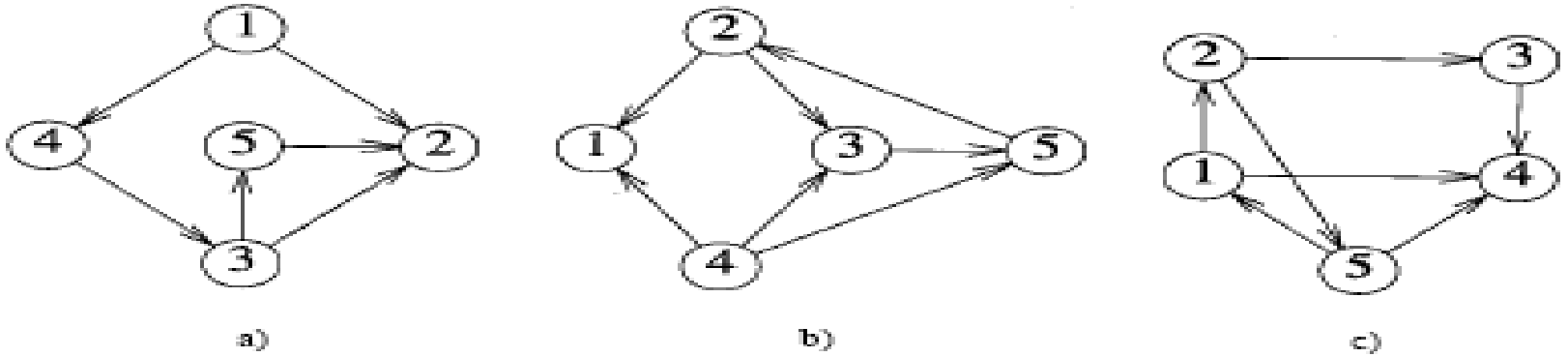


图12-8　有向图

- (1) 图a的入度是： [0,3,1,1,1].
  图b的入度是： [2,1,2,0,2].
  图c的入度是： [1,1,1,3,1].
- (2) 图a的出度是： [2,0,2,1,1].
  图b的出度是： [0,2,1,3,1].
  图c的出度是： [2,2,1,0,2].
- (3)

  a图中顶点2没有邻接于它的顶点

  b图中邻接于顶点2的邻接顶点有顶点1和3

  c图中邻接于顶点2的邻接顶点有顶点3和5
- (4)

  a图中没有邻接至顶点1的邻接顶点

  b图中邻接至顶点1的顶点有 顶点2和4

  c图中邻接至顶点1的顶点有 顶点5

- (5)

  a图中边(3,5) (3,2) 是关联于顶点3的边

  b图中边(3,5) 是关联于顶点3的边

  c图中(3,4) 是关联于顶点3的边

- (6)

  a图中边(1,4) 是关联至顶点4的边

  b图中没有

  c图中边(1,4), (5,4), (3,4) 是关联至顶点3的边

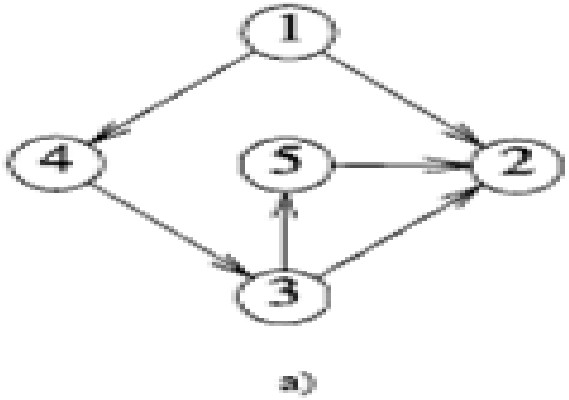
- (7)

  a图中没有有向环路

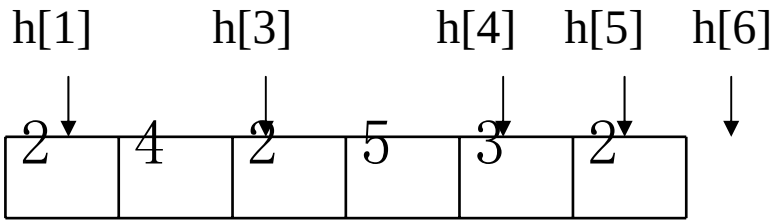
  b图中环路：2，3，5，2 长度是3

  c图中环路：1，2，5，1 长度是3

- 10. 请为图(12 – 5)和12-8a 提供下列描述：
- 1) 邻接矩阵。
- 2) 邻接压缩表。
- 3) 邻接链表。



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



- 23. 给出相应于图12-1a 和b 中耗费邻接矩阵的网络邻接链表。

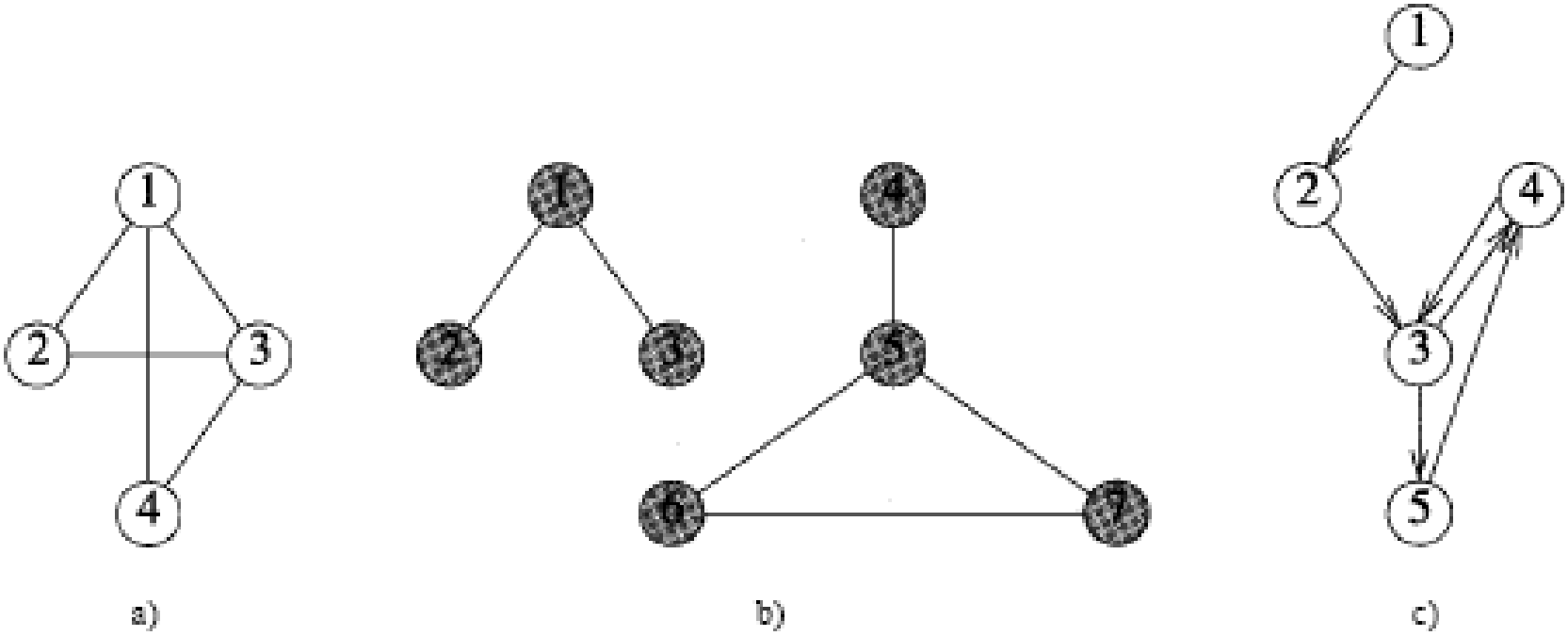


图12-1　图

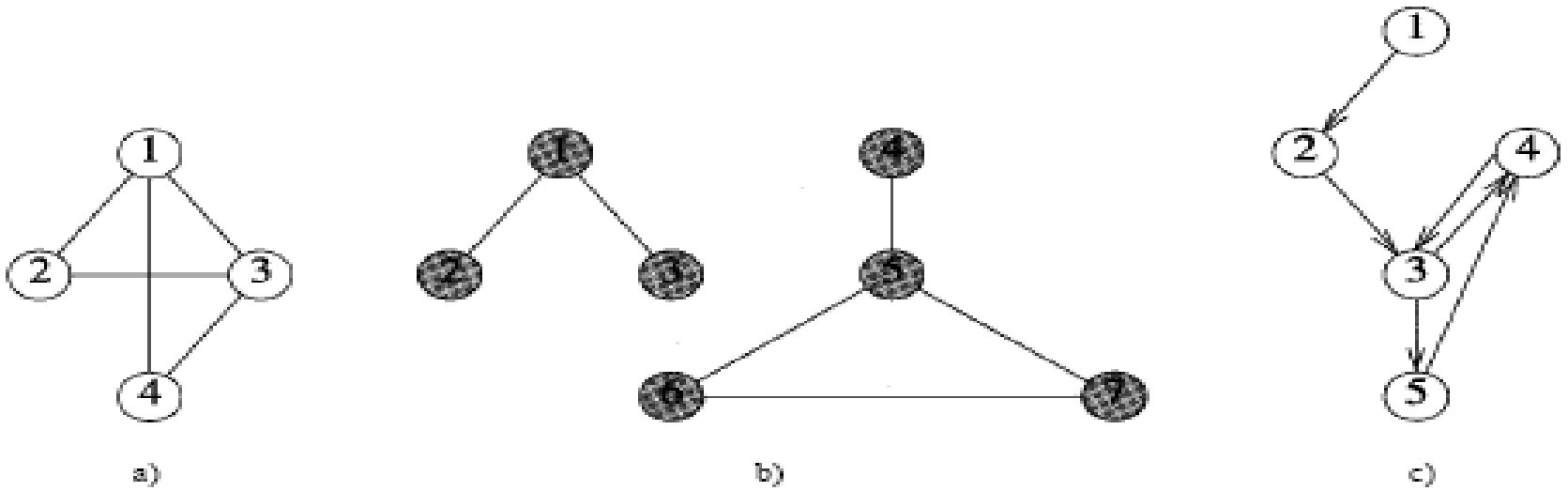


图12-1　图

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

a)

	1	2	3	4	5	6	7
1	∞	9	5	∞	∞	∞	∞
2	9	∞	∞	∞	∞	∞	∞
3	5	∞	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞
5	∞	∞	∞	3	∞	6	4
6	∞	∞	∞	∞	6	∞	1
7	∞	∞	∞	∞	4	1	∞

b)

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

c)

图12-13　图12-1对应的可能的耗费邻接矩阵

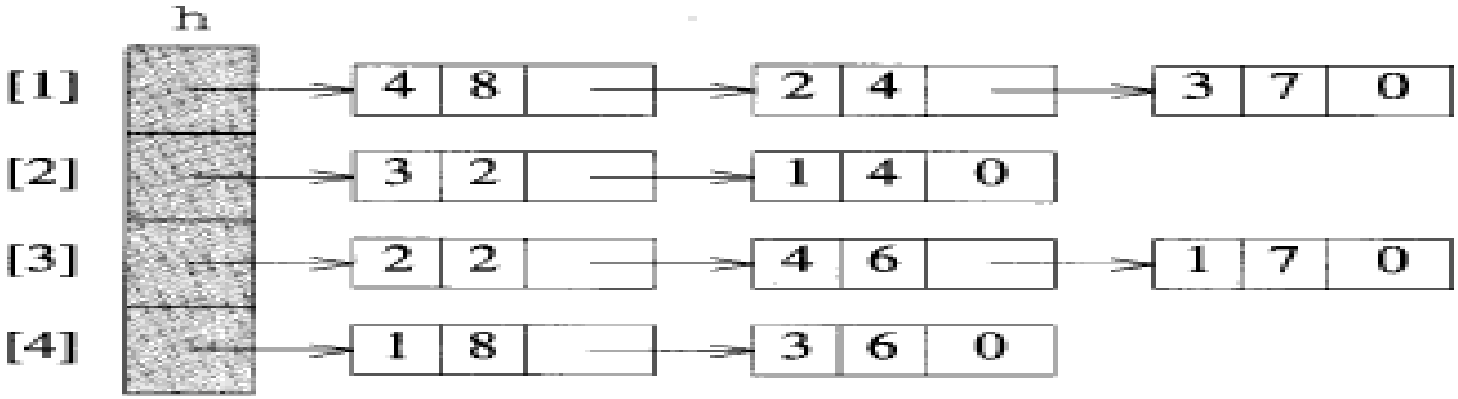
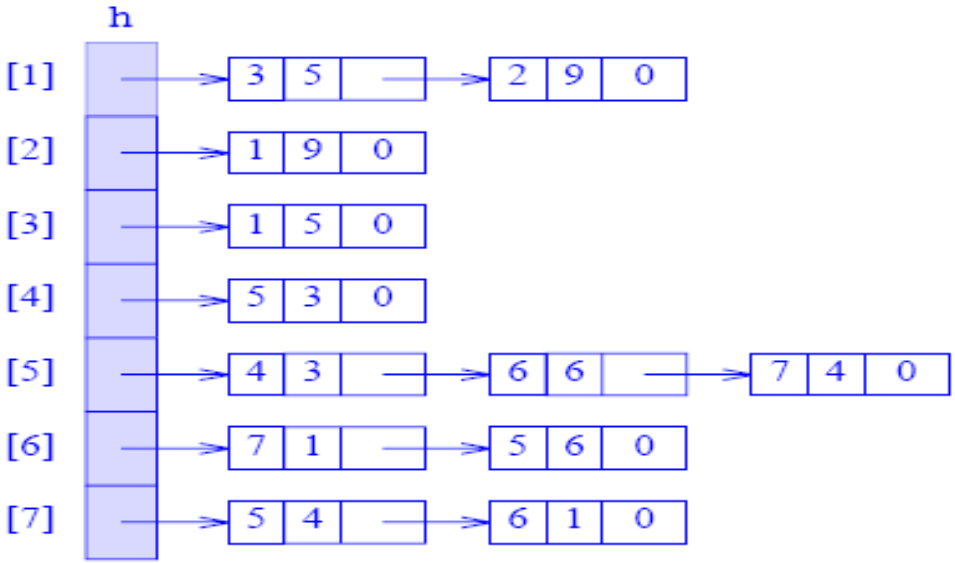
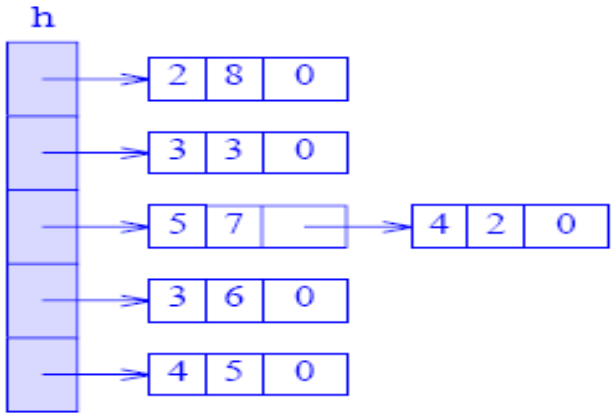


图12-14　图12-13a对应的网络的邻接链表

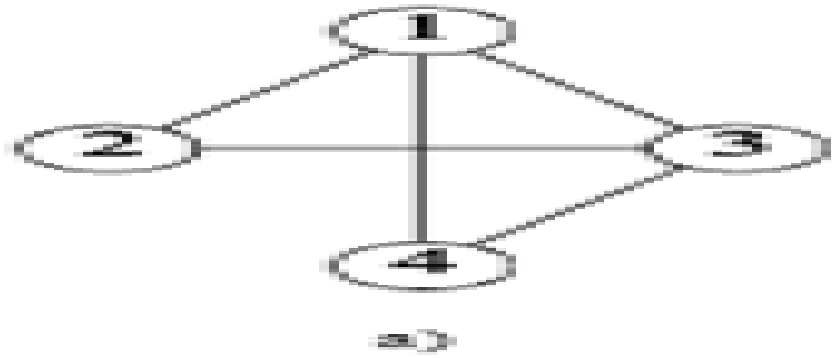


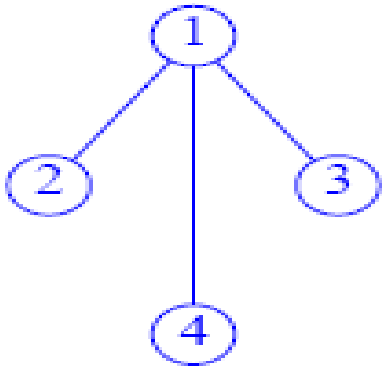
(b)



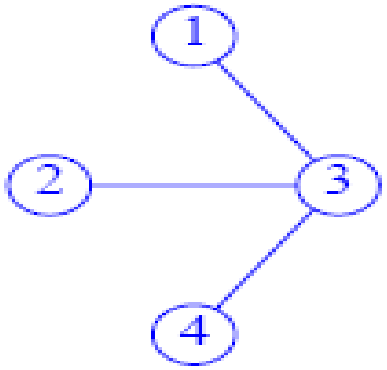
(c)

- 45. 根据图12 - 1a，完成以下练习：
- 1) 从顶点1开始产生一个宽度优先生成树。
- 2) 从顶点3开始产生一个宽度优先生成树。
- 3) 从顶点1开始产生一个深度优先生成树。
- 4) 从顶点3开始产生一个深度优先生成树。

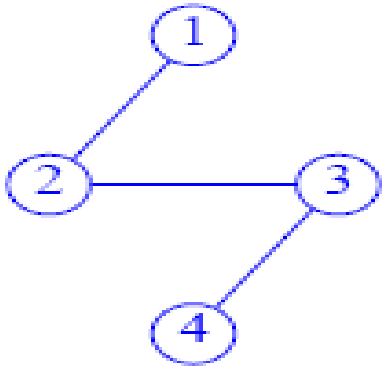




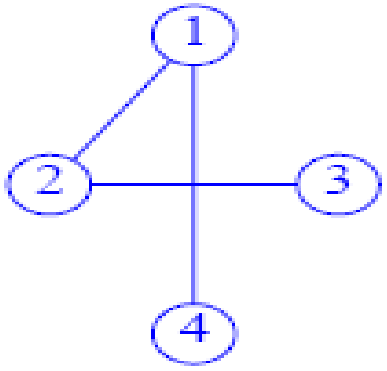
(a)



(b)



(c)



(d)

- 48．编写共享成员Network::Cycle( )，用于确定网络中是否存在一个（有向）环路。可基于DFS或BFS来实现。

算法思想：
在DFS过程中，如果发现遍历到一个已经被访问的节点，则说明图中存在环路。

state [i] = 0：顶点i未被访问
state [i] = 1：顶点i已被访问，但其子孙节点未被访问
state [i] = 2：顶点i及其子孙节点已访问完

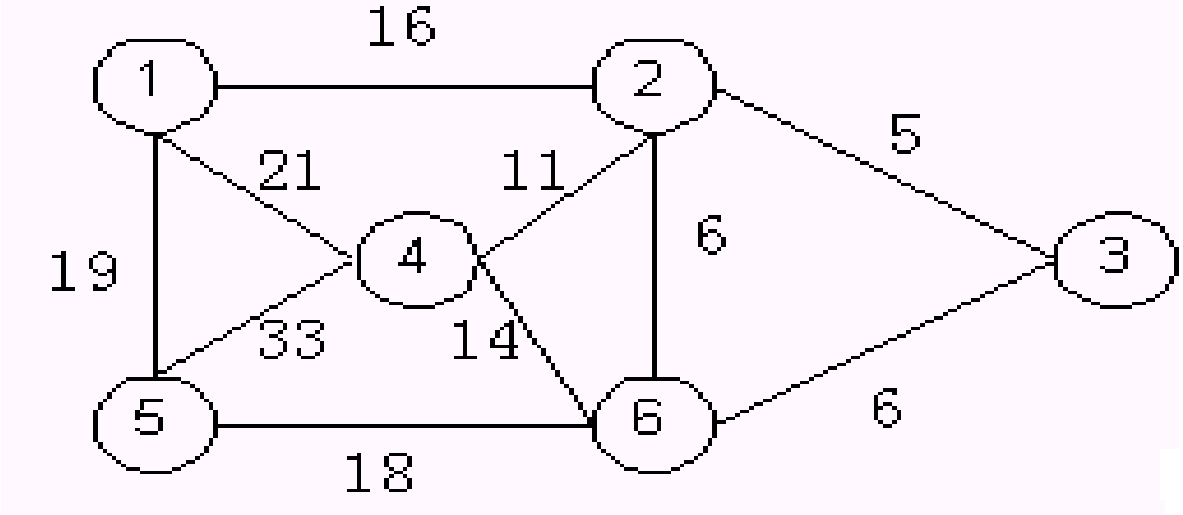
```
bool Network::cycle();
{//确定网络中是否存在一个（有向）环路。
int n = Vertices( ) ;
InitializePos(); // 遍历器
int * state = new int [n+1];
for (int i = 1; v <= n; i++)
      state [i] = 0;
for (int v = 1; i <= n; i++)
   if（state[v] = =0）
            if (dfs_FromVertex (v))  return  true;

return false;
}
```

```
dfs_FromVertex (Vertex v)
{
state[v]= 1;
int  u=Begin(v);
while (u)
   { if  (state[u]= =1)  return true
      else  if  (state[u]= =0)
               if (dfs_FromVertex (u))  return  true;
     u=NextVertex(v);}

state[v]= 2;
return  false;
}
```

- 分别用深度优先搜索和宽度优先搜索遍历下图所示的无向图，给出以1为起点的顶点访问序列（同一个顶点的多个邻接点，按数字顺序访问），给出一棵深度优先生成树和宽度优先生成树。

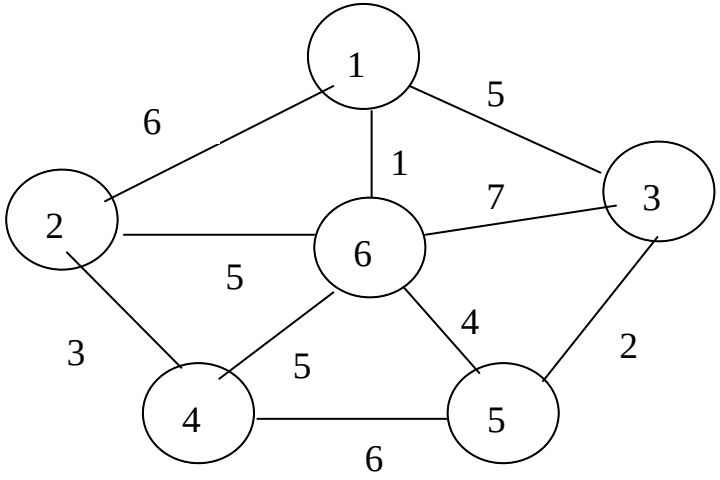


- 已知图G的邻接矩阵如下所示：

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- 由邻接矩阵画出相应的图G；图中所有顶点是否都在它的拓扑有序序列中？
- 如果要使此图成为完全图，还需增加几条边？

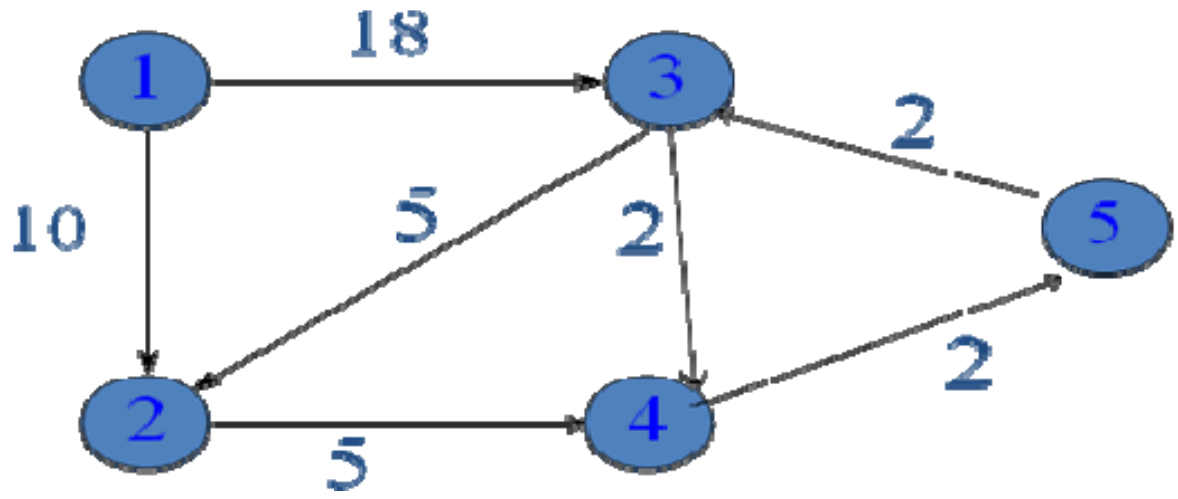
- 描述图的普里姆算法和克鲁斯卡尔算法；分别用普里姆算法和克鲁斯卡尔算法求出下图的一棵最小代价生成树。给出构造最小生成树的过程.



- 有如下的网络邻接矩阵，画出该图；给出图的邻接链表表示；画出一棵最小生成树。

∞	17	∞	∞	20	22
17	∞	6	7	∞	12
∞	6	∞	11	∞	∞
∞	7	11	∞	19	15
20	∞	∞	19	∞	34
22	12	∞	15	34	∞

- 以下图为例，
- （1）按Dijkstra算法计算从源点1到其它各个顶点的最短路径和最短路径长度。
- （2）使用Floyd算法计算各对顶点之间的最短路径。



- K=0

	1	2	3	4	5
1	0	10	18	∞	∞
2	∞	0	∞	5	∞
3	∞	5	0	2	∞
4	∞	∞	∞	0	2
5	∞	∞	2	∞	0

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

- K=1

	1	2	3	4	5
1	0	10	18	∞	∞
2	∞	0	∞	5	∞
3	∞	5	0	2	∞
4	∞	∞	∞	0	2
5	∞	∞	2	∞	0

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

- K=2

	1	2	3	4	5
1	0	10	18	15	∞
2	∞	0	∞	5	∞
3	∞	5	0	2	∞
4	∞	∞	∞	0	2
5	∞	∞	2	∞	0

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

- K=3

	1	2	3	4	5
1	0	10	18	15	∞
2	∞	0	∞	5	∞
3	∞	5	0	2	∞
4	∞	∞	∞	0	2
5	∞	7	2	4	0

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

- K=4

	1	2	3	4	5
1	0	10	18	15	17
2	∞	0	∞	5	7
3	∞	5	0	2	4
4	∞	∞	∞	0	2
5	∞	7	2	4	0

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

- K=5

	1	2	3	4	5
1	0	10	18	15	17
2	∞	0	9	5	7
3	∞	5	0	2	4
4	∞	9	4	0	2
5	∞	7	2	4	0

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

- 设要将序列〈12，2，16，30，28，10，16*，20，6，18〉中的关键码按字母序的升序进行排序，写出：
- （1）分别给出冒泡排序、插入排序、选择排序、基数排序、归并排序、以第一个元素为支点的快速排序第一趟结束时的序列（或每趟排序后的结果）；
- （2）堆排序初始建堆的结果，堆排序的过程。