第8章 优先队列

第8章 优先队列

学习要点:

- 理解以集合为基础的抽象数据类型优先队列
- 理解用字典实现优先队列的方法
- 理解优先级树和堆的概念
- 掌握用数组实现堆的方法
- 理解以集合为基础的抽象数据类型可并优先队列
- 理解左偏树的定义和概念
- 掌握用左偏树实现可并优先队列的方法
- 掌握堆排序算法

返回章目录

8.1 优先队列的定义

优先队列的原型

- 排队上车，老弱病残者优先上车
- 排队候诊，危急病人优先就诊
- 洗相馆为顾客洗照片，加钱加急者优先洗
- 分时操作系统运行程序，小程序优先
- 贪心算法对解分量的选择，按元素的某种特征值，大(或小)的优先
- 在一个集合中搜索，按元素的某种特征值，大(或小)的优先

……处理或服务时只关心对象中谁的优先级最高，通常的队列是一种优先队列最先到者优先级最高

8.1 优先队列的定义

优先队列也是一个以集合为基础的抽象数据类型。

优先队列中的每一个元素都有一个优先级值。优先队列中元素**x**的优先级值记为**p(x)**，它可以是一个实数，也可以是一个一般的全序集中的元素。优先级值用来表示该元素出列的优先级。

约定优先级值小的优先级高。亦可约定优先级值大的优先级高。

优先队列支持的基本运算有：

(1)Min(H)：返回优先队列**H**中具有最小优先级的元素。

(2)Insert(x, H)：将元素**x**插入优先队列**H**。

(3)DeleteMin(H)：删除并返回优先队列H中具有最小优先级的元素。

返回章目录

8.2 用字典实现优先队列

(1)优先队列与字典的相似性与区别：

- 优先队列中元素的优先级值可以看作是字典中元素的线性序值。
- 在字典中，不同的元素具有不同的线性序值，其插入运算仅当要插入元素**x**的线性序值与当前字典中所有元素的线性序值都不同时才执行。
- 对于优先队列来说，不同的元素可以有相同的优先级值。因此，优先队列的插入运算即使在当前优先队列中存在与要插入元素**x**有相同的优先级值的元素时，也要执行元素**x**的插入。

8.2 用字典实现优先队列

(2)由于优先队列与字典的相似性,所有实现字典的方法都可用于实现优先队列。

- 用有序链表实现优先队列;(**Insert**低效)
- 用二叉搜索树实现优先队列;(**Insert,DeleteMin, Min**均低效)
- 用AVL树实现优先队列;(逻辑复杂)
- 用无序链表实现优先队列;(**DeleteMin, Min**均低效)

……

都有缺点。原因在于没有考虑到优先队列的特性。

返回章目录

8.3 用优先级树实现优先队列

1. 优先队列的特征:

- **DeleteMin**和**Min**只关心优先级最高的元素
- **Insert**的元素不要求全局的序关系

因此实现优先队列的结构只要求方便**DeleteMin**和**Min**,而对**Insert**也只要求不给结构的维护带来太大的麻烦。

根据这两个特征,人们发明了优先级树。

8.3 用优先级树实现优先队列

2. 优先级树的概念

优先级树是满足下面的优先级性质的二叉树：

(1)树中每一结点存储一个元素。

(2)任一结点中存储的元素的优先级值不大(小)于其儿子结点中存储的元素的优先级值即父结点的优先级不低于其儿子结点的优先级。

换句话说，越接近根的结点中的元素的优先级越高，越方便被访问，因为根最方便被访问。

相应的优先级树称为极小(大)化优先级树。

返回章目录

8.4 用堆实现优先队列

用优先级树实现优先队列仍有不足：

- **Insert(x, H)和DeleteMin(H)后对结构的维护，在最坏情况下，仍需O(h)=O(n)。**
- **如果让优先级树近似满, 从而h=[log n],达到最小, 那么，在最坏情况下，Min(H)将只需O(1)，Insert(x,H)和DeleteMin(H)后对结构的维护只需O(log n)。**

- 因而引入堆的概念并用堆来实现优先队列。

(1)堆的概念：

如果一棵优先级树是一棵近似满二叉树，那么，这棵具有优先级性质的近似满二叉树(外形像堆)就叫做堆。

(2)用堆实现优先队列：

Min(H)、Insert(x, H)和DeleteMin(H)运算的实现

8.4 用数组表示堆从而实现优先队列

(1) 用数组表示堆：

从1开始对堆的结点从根开始自上而下逐层、每层从左到右进行编号，然后让结点中的元素按编号在数组A中与下标对号入座。

(2) 用数组表示堆的优点：

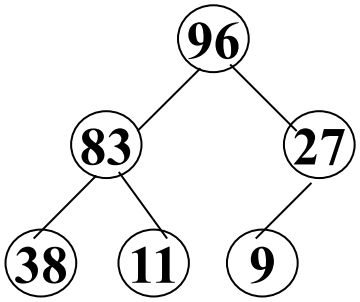
- 存储紧凑，空间利用率高
- 父子关系简单清晰:存放在**A[i]**的是结点**i**的元素，**A[2i]**和**A[2i+1]**分别是结点**i**的左和右儿子结点**2i**和**2i+1**的元素， **A[i/2]**是结点**i**的父结点的元素

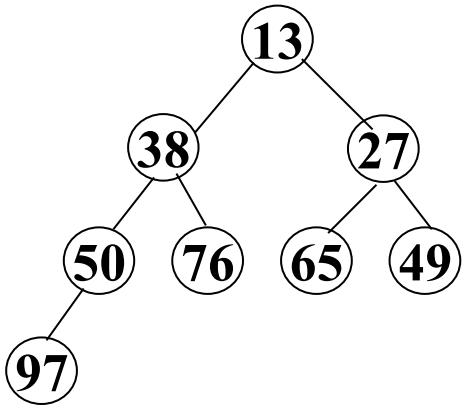
堆排序算法

- 堆的定义：n个元素的序列(k1,k2,……kn)，当且仅当满足下列关系时，称之为堆。

$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i}+1 \end{cases} \quad 或 \quad \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i}+1 \end{cases} \qquad (i=1,2,\ldots\ldots\lfloor n/2 \rfloor)$$

例1（96，83，27，38，11，9） 例2（13，38，27，50，76，65，49，97）

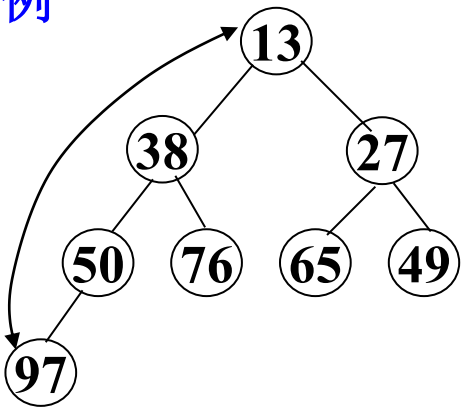


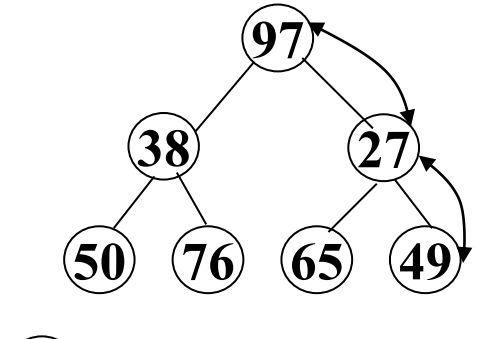


可将堆序列看成近似满二叉树，则堆顶元素（近似满二叉树的根）必为序列中n个元素的最小值或最大值

- **堆排序算法基本思想：**将无序序列建成一个堆，得到关键字最小（或最大）的记录；输出堆顶的最小（大）值后，使剩余的n-1个元素重又建成一个堆，则可得到n个元素的次小值；重复执行，得到一个有序序列，这个过程叫堆排序。
- **堆排序需解决的两个问题：**
  - 如何由一个无序序列建成一个堆？
  - 如何在输出堆顶元素之后，调整剩余元素，使之成为一个新的堆？
- **第二个问题解决方法——筛选**
  - 方法：输出堆顶元素之后，以堆中最后一个元素替代之；然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”。

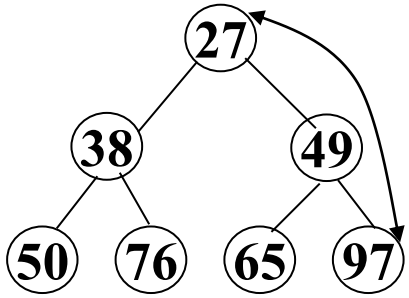
例





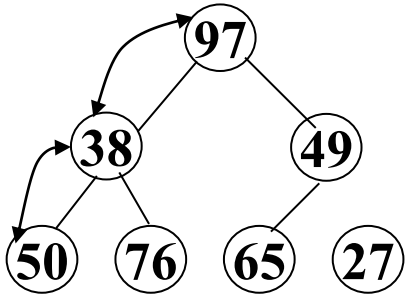
13

输出：13



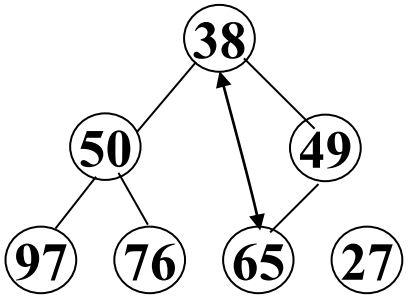
13

输出：13



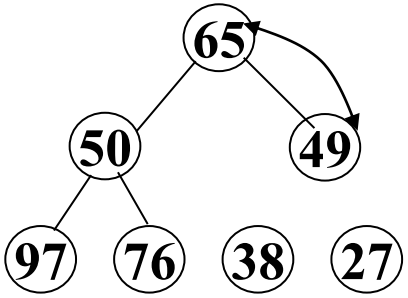
13

输出：13 27



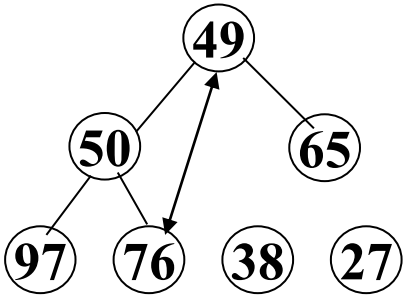
13

输出：13 27



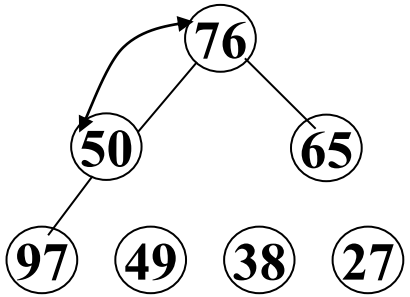
13

输出：13 27 38



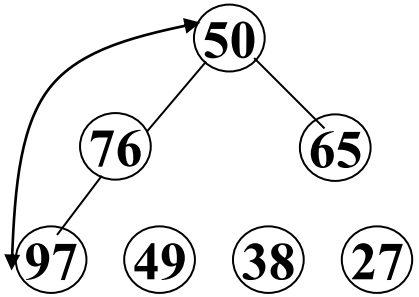
13

输出：13 27 38



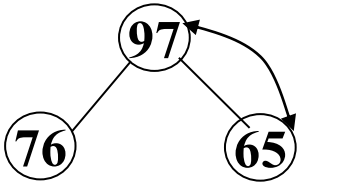
13

输出：13 27 38 49



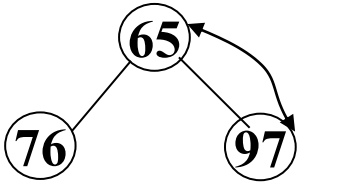
13

输出：13 27 38 49



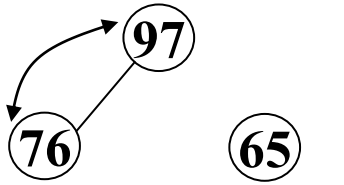
13

输出：13 27 38 49 50



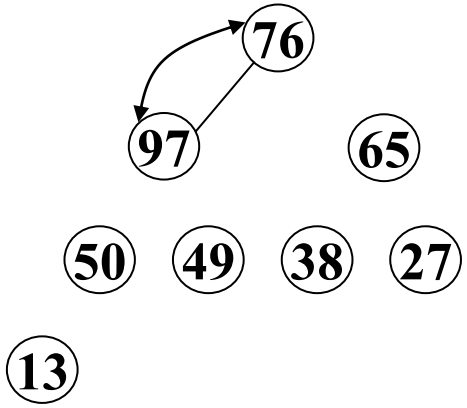
13

输出：13 27 38 49 50

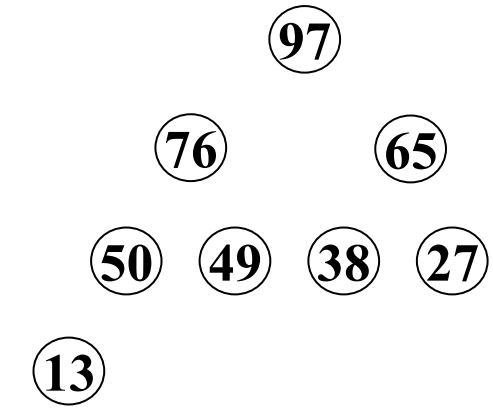


13

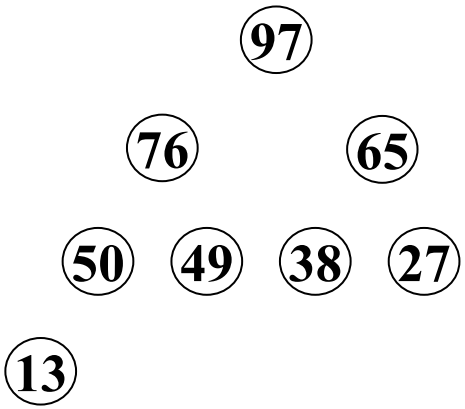
输出：13 27 38 49 50 65



输出：13 27 38 49 50 65



输出：13 27 38 49 50 65 76



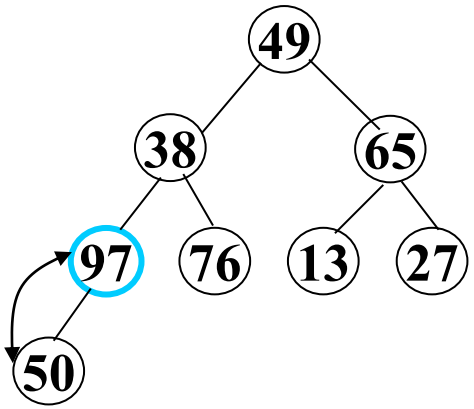
输出：13 27 38 49 50 65 76 97

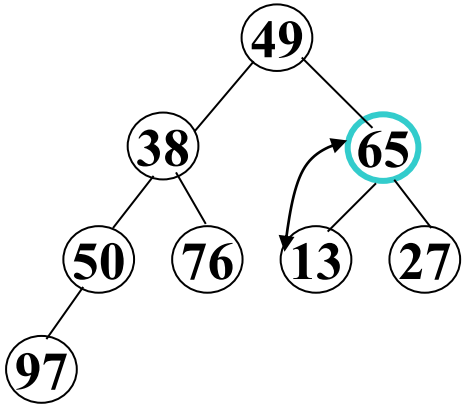
- 算法描述

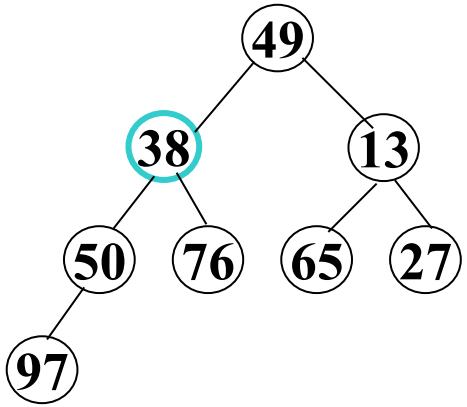
```
int sift(T r[],int k,int m)
{ int i,j;
  T x;
  i=k;  x=r[i]; j=2*i;
  while(j<=m)
  {  if((j<m)&&(r[j]>r[j+1]))
     j++; //这里判断r[i]的右儿子r[j+1]是否存在，以及比较左右儿子的大小
     if(x>r[j])
     {  r[i]=r[j];
        i=j;
        j*=2;
     }
     else  break;
  }
  r[i]=x;
}
```

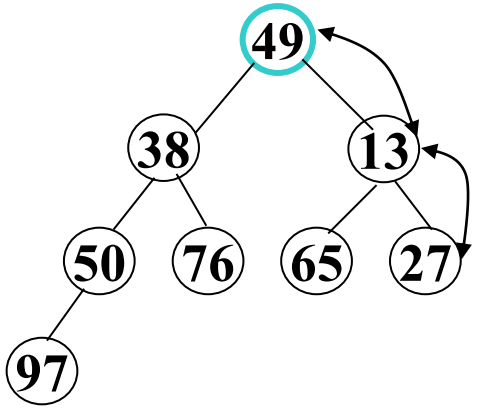
- 第一个问题解决方法
  - 方法：从无序序列的第⌊n/2⌋个元素（即此无序序列对应的完全二叉树的最后一个非终端结点）起，至第一个元素止，进行反复筛选

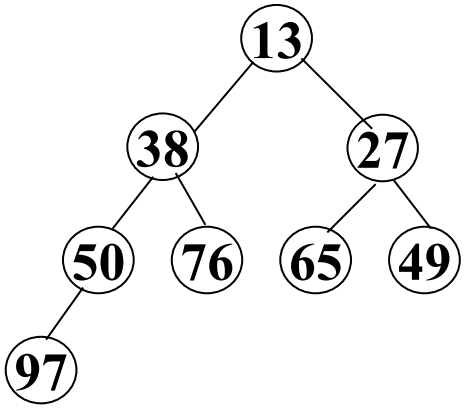
例 含8个元素的无序序列（49，38，65，97，76，13，27，50）











- 算法描述

```
void heapsort(T r[],int n)
{  int i;
   T x;
   for(i=n/2;i>=1;i--)
      sift(r,i,n);
   for(i=n;i>=2;i--)
   {  x=r[1];
      r[1]=r[i];
      r[i]=x;
      sift(r,1,i-1);
   }
}
```

- 算法描述
  - 时间复杂度：最坏情况下T(n)=O(nlogn)
  - 空间复杂度：S(n)=O(n)

返回章目录

8.5 可并优先队列

可并优先队列也是一个以集合为基础的抽象数据类型。除了必须支持优先队列的Insert和DeleteMin运算外，可并优先队列还支持2个不同优先队列的合并运算Concatenate。

用堆来实现优先队列，可在O(logn)时间内支持同一优先队列中的基本运算。但合并2个不同优先队列的效率不高。下面讨论的左偏树结构不但能在O(logn)时间内支持同一优先队列中的基本运算，而且还能有效地支持2个不同优先队列的合并运算Concatenate。

8.5 可并优先队列

8.5.1 左偏树的定义

左偏树是一类特殊的优先级树。与优先级树类似，左偏树也有极小化左偏树与极大化左偏树之分。为了确定起见，下面所讨论的左偏树均为极小化左偏树。常用的左偏树有左偏高树和左偏重树2种不同类型。顾名思义，左偏高树的左子树偏高，而左偏重树的左子树偏重。下面给出其严格定义。

若将二叉树结点中的空指针看作是指向一个空结点，则称这类空结点为二叉树的前端结点。并规定所有前端结点的高度（重量）为0。

对于二叉树中任意一个结点x，递归地定义其高度s(x)为：

$s(x) = \min \{ s(L), s(R) \} + 1$

其中L和R分别是结点x的左儿子结点和右儿子结点。

8.5 可并优先队列

8.5.1 左偏树的定义

一棵优先级树是一棵左偏高树，当且仅当在该树的每个内结点处，其左儿子结点的高（s值）大于或等于其右儿子结点的高（s值）。

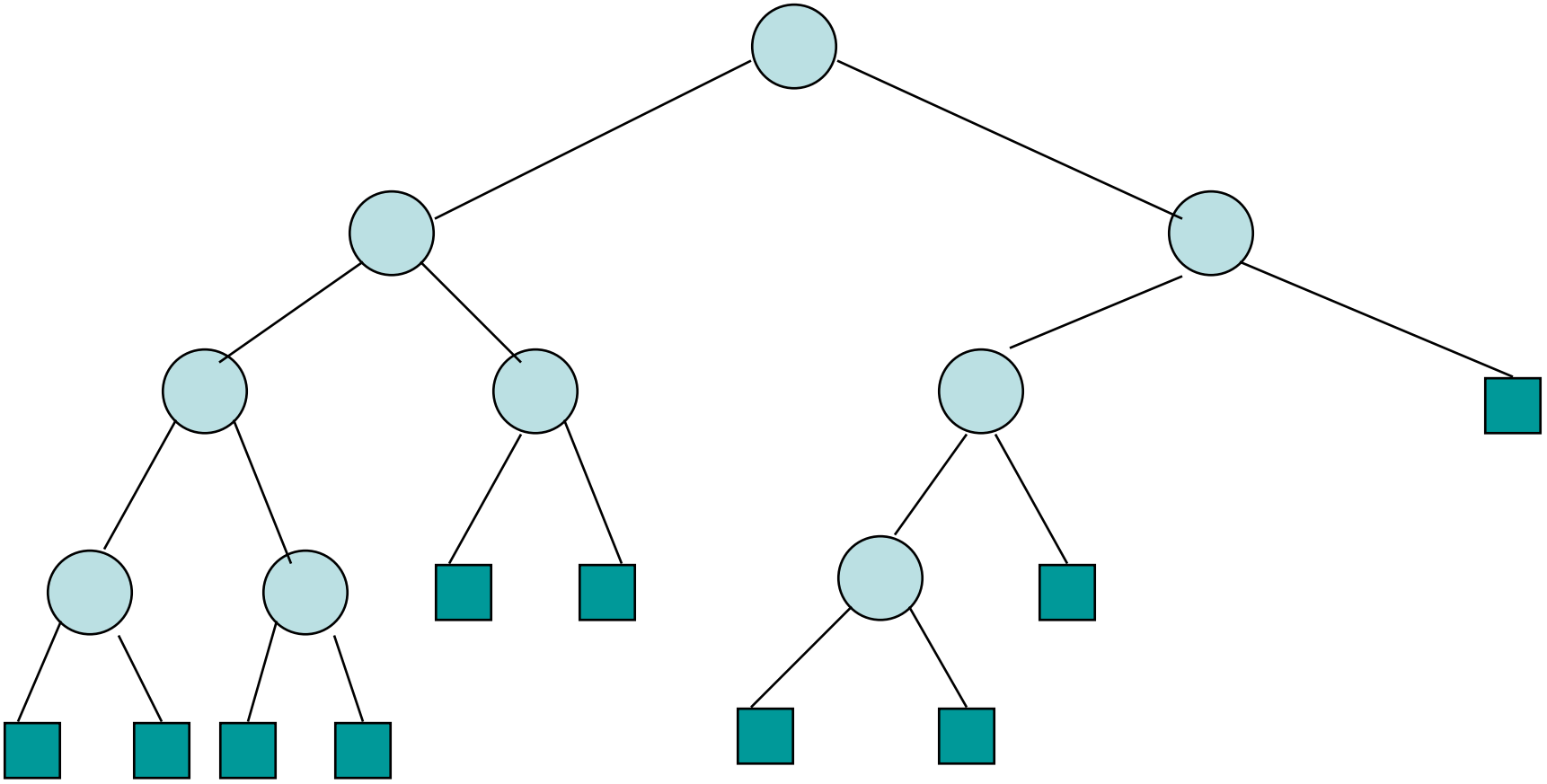
对于二叉树中任意一个结点x，其重量w(x)递归地定义为：

w(x) = w(L) + w(R) + 1

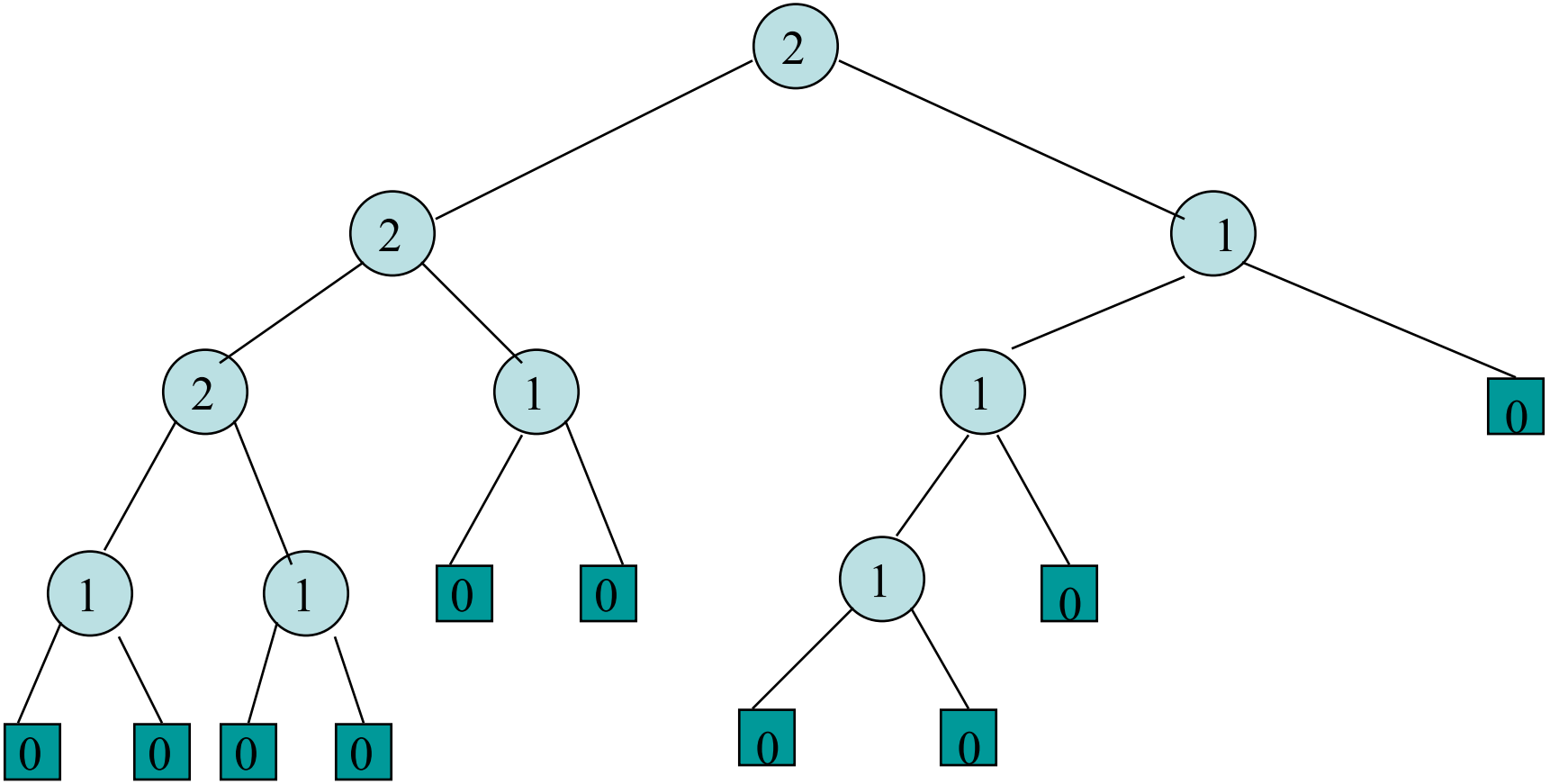
其中L和R分别是结点x的左儿子结点和右儿子结点。

一棵优先级树是一棵左偏重树，当且仅当在该树的每个内结点处，其左儿子结点的重（w值）大于或等于其右儿子结点的重（w值）。

A Leftist Tree



s() Values Example



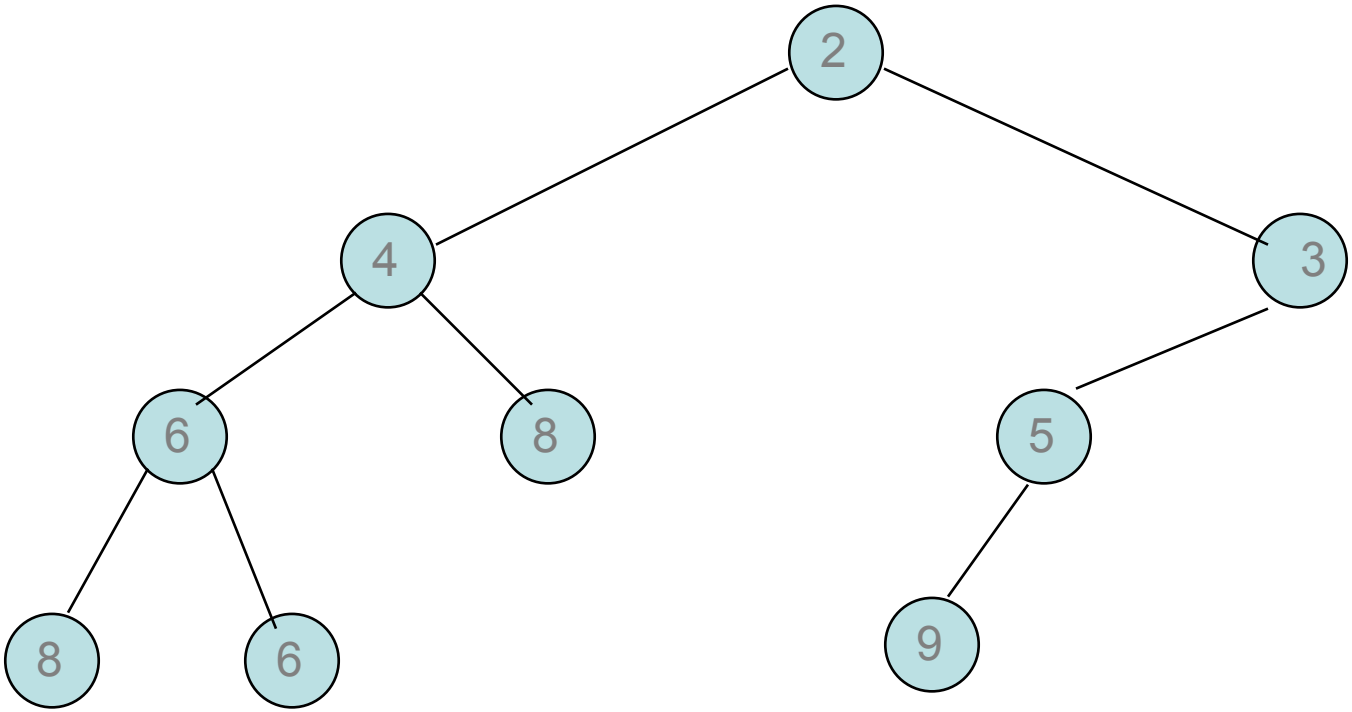
8.5 可并优先队列

8.5.2 左偏高树的性质

左偏高树具有下面性质：
设x是一棵左偏高树的任意一个内结点，则
(1)以x为根的子树中至少有2^s(x) - 1个结点。
(2)如果以x为根的子树中有m个结点，则s(x)的值不超过log(m+1)。
(3)从x出发的最右路经的长度恰为s(x)。

返回章目录

A Min Leftist Tree



Some Min Leftist Tree Operations

Insert

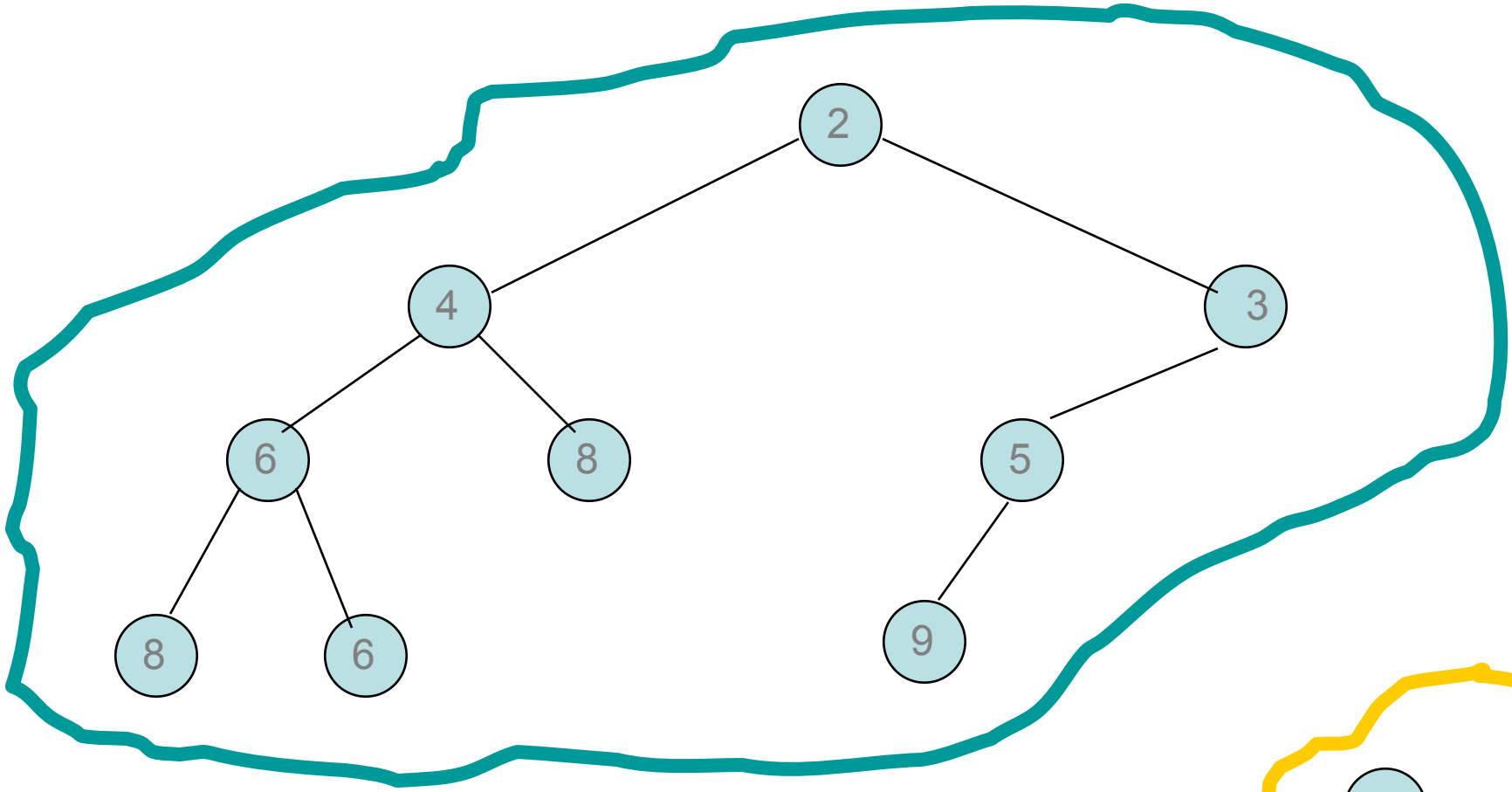
DeleteMin()

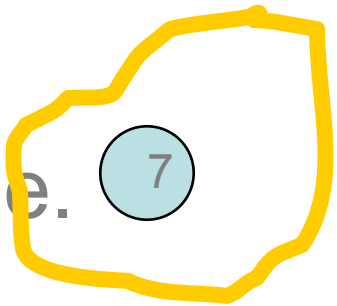
meld()

initialize()

Insert() and DeleteMin() use meld().

Insert Operation

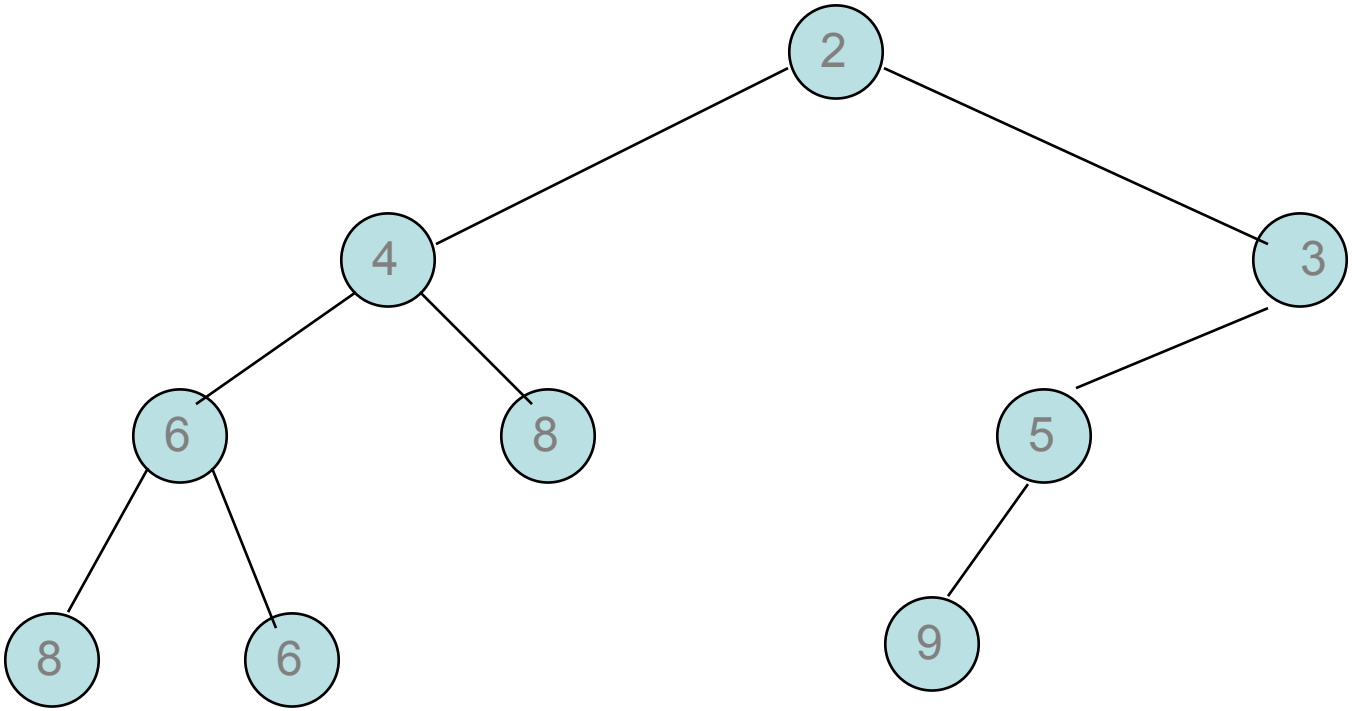




Create a single node min leftist tree.

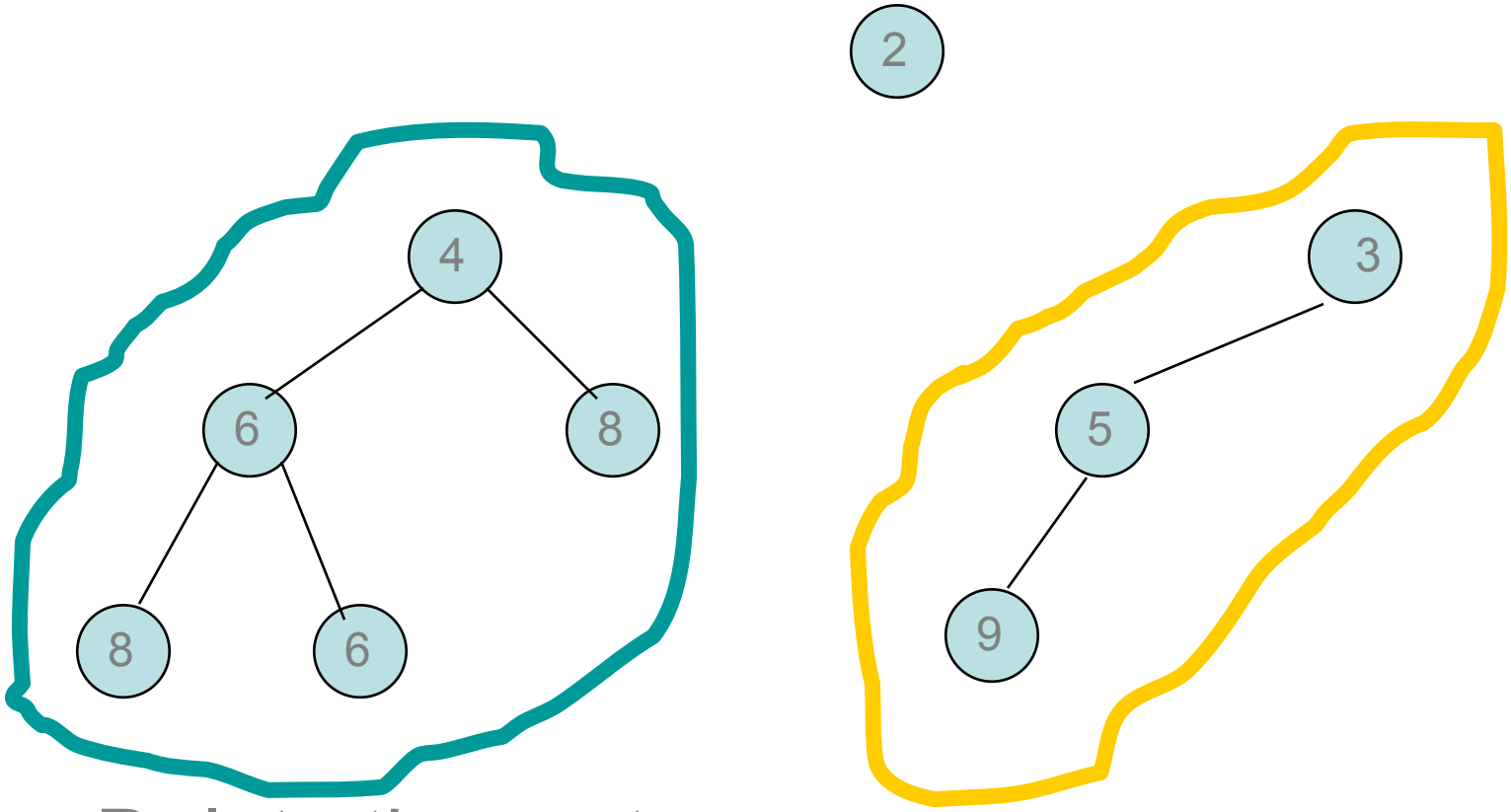
Meld the two min leftist trees.

Delete Min



Delete the root.

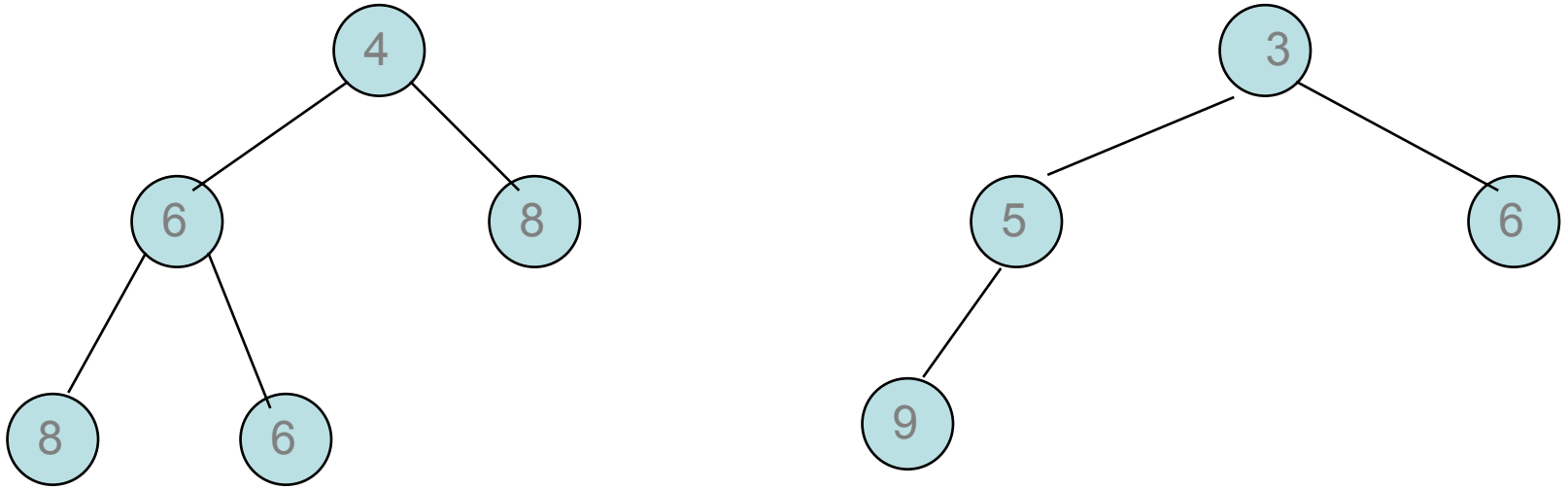
Delete Min



Delete the root.

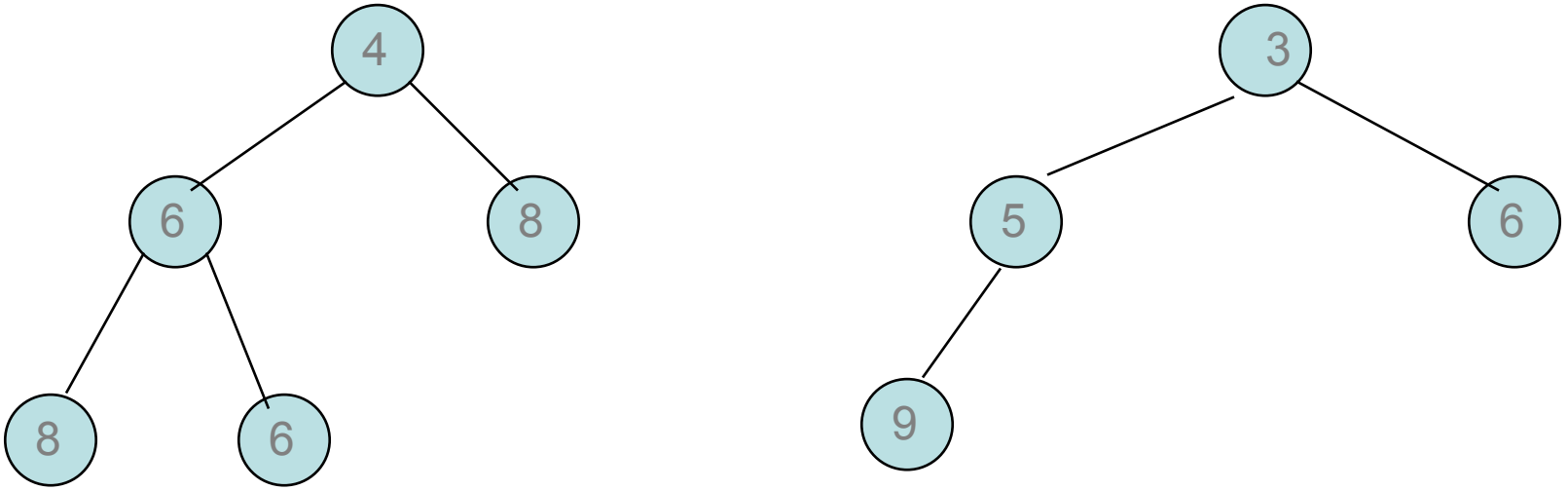
Meld the two subtrees.

Meld Two Min Leftist Trees



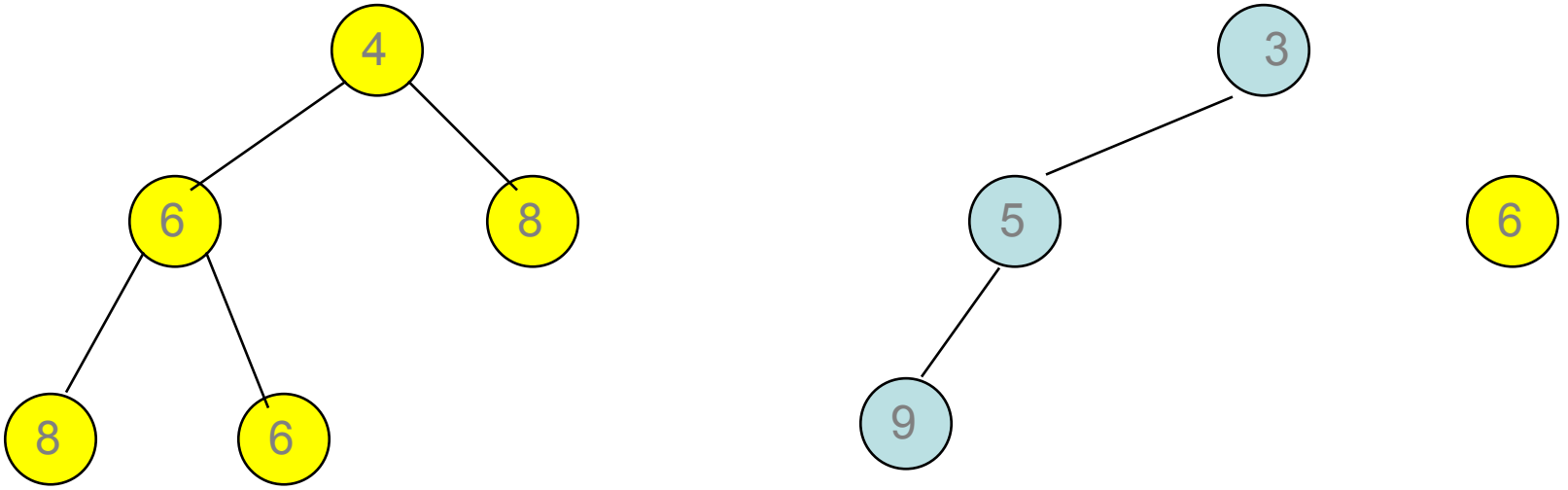
Traverse only the rightmost paths so as to get logarithmic performance.

Meld Two Min Leftist Trees



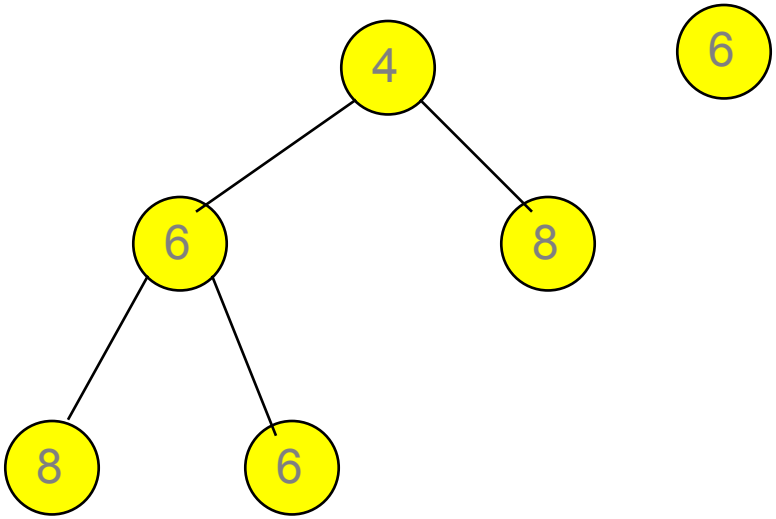
Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

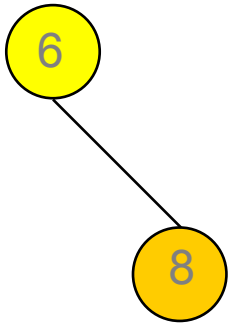
Meld Two Min Leftist Trees

Meld right subtree of tree with smaller root and all of other tree.

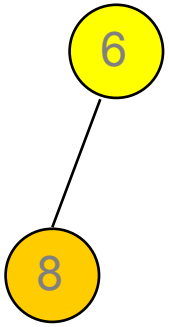
Right subtree of 6 is empty. So, result of melding right subtree of tree with smaller root and other tree is the other tree.

Meld Two Min Leftist Trees

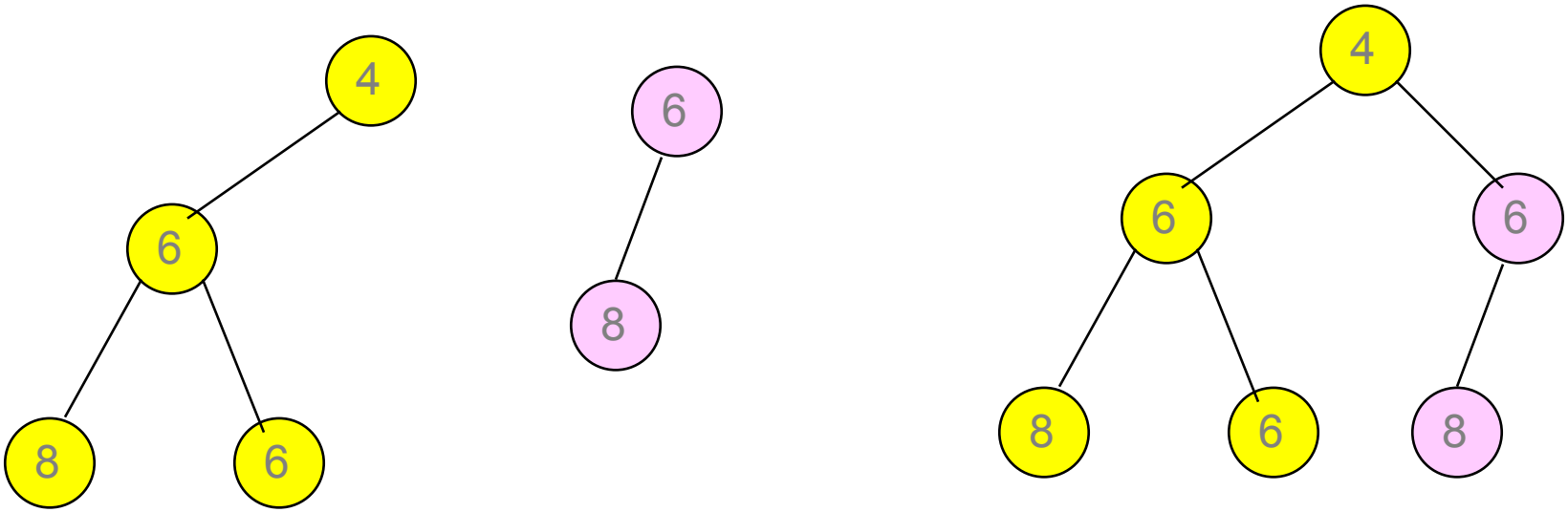
Make melded subtree right subtree of smaller root.



Swap left and right subtree if s(left) < s(right).



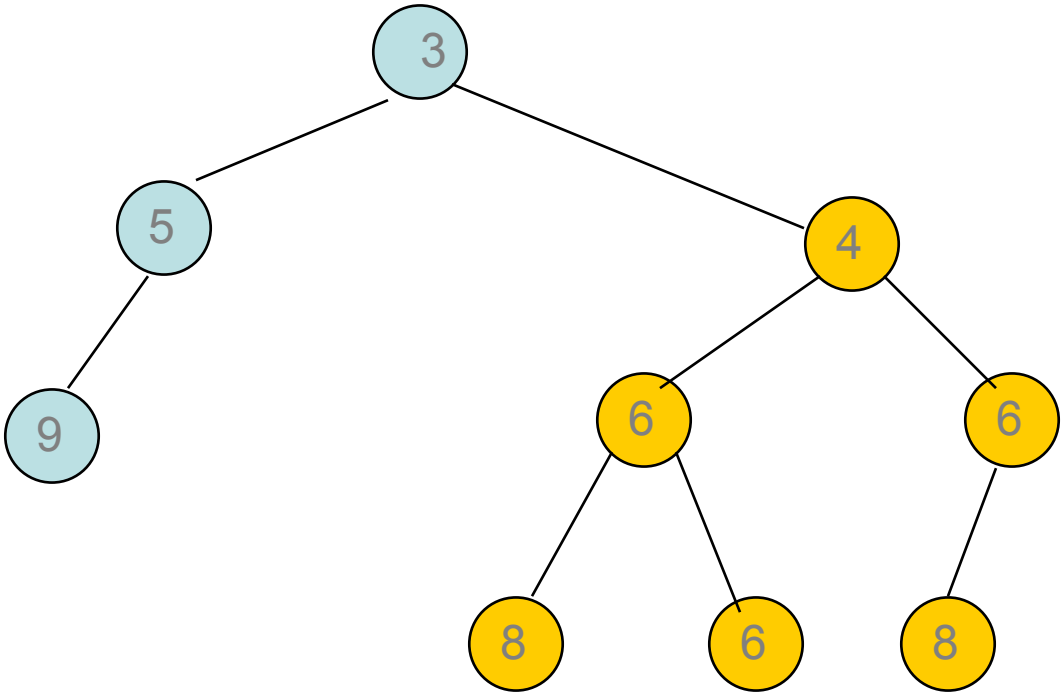
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if s(left) < s(right).

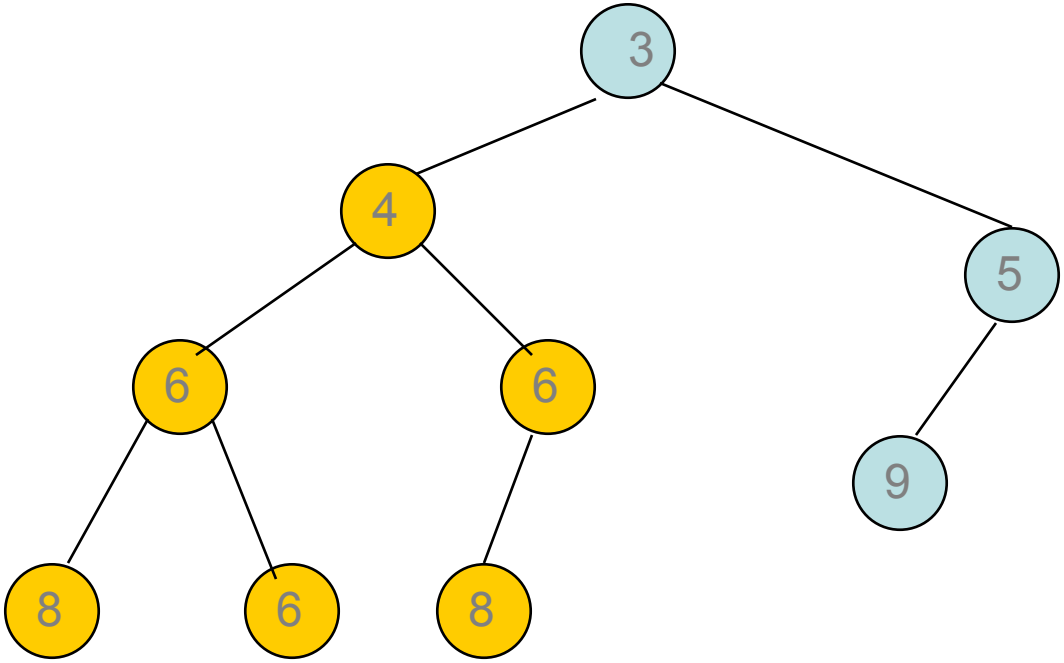
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if s(left) < s(right).

Meld Two Min Leftist Trees



Initializing In O(n) Time

- Create n single-node min leftist trees and place them in a FIFO queue.
- Repeatedly Delete two min leftist trees from the FIFO queue, meld them, and Insert the resulting min leftist tree into the FIFO queue.
- The process terminates when only 1 min leftist tree remains in the FIFO queue.
- Analysis is the same as for heap initialization.

8.6 优先队列的应用

- 哈夫曼树(Huffman)——带权路径长度最短的树
  - 定义
    - 路径：从树中一个结点到另一个结点之间的分支构成这两个结点间的路径
    - 路径长度：路径上的分支数
    - 树的路径长度：从树根到每一个结点的路径长度之和
    - 树的带权路径长度：树中所有带权结点的路径长度之和

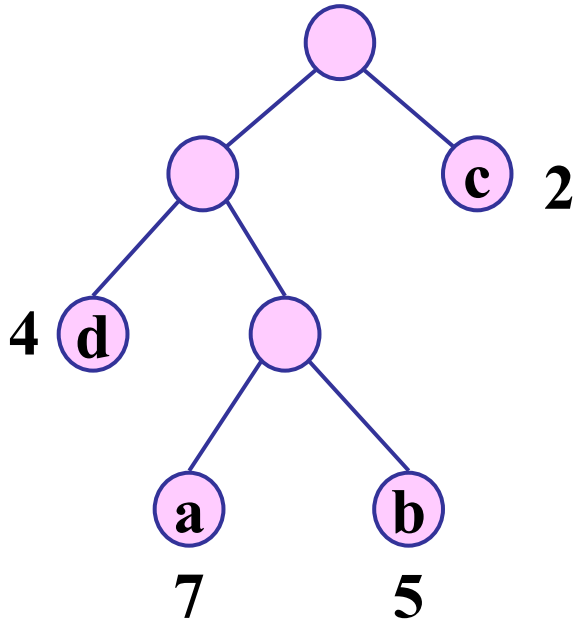
      记作：$wpl = \sum_{k=1}^{n} w_k l_k$

      其中：w_k — 权值

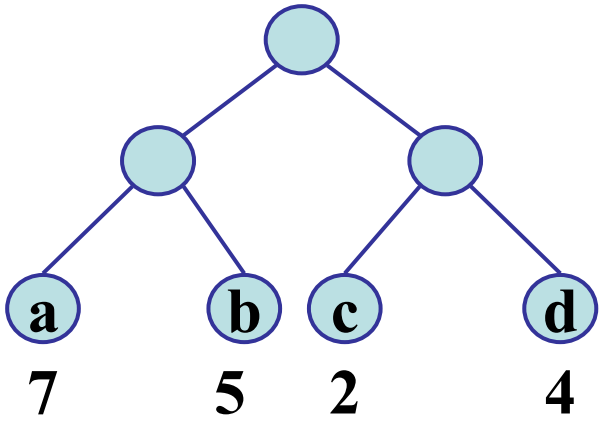
      l_k — 结点到根的路径长度
    - **Huffman树——设有n个权值{w1,w2,……wn}，构造一棵有n个叶子结点的二叉树，每个叶子的权值为wi,则wpl最小的二叉树叫Huffman树**

例 有4个结点，权值分别为7，5，2，4，构造有4个叶子结点的二叉树

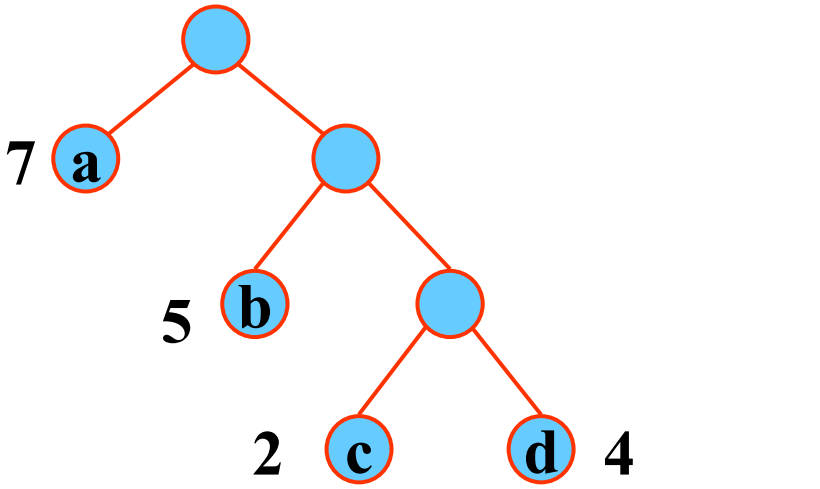
$$WPL = \sum_{k=1}^{n} W_K L_K$$



WPL=7*3+5*3+2*1+4*2=46



WPL=7*2+5*2+2*2+4*2=36



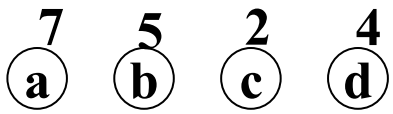
WPL=7*1+5*2+2*3+4*3=35

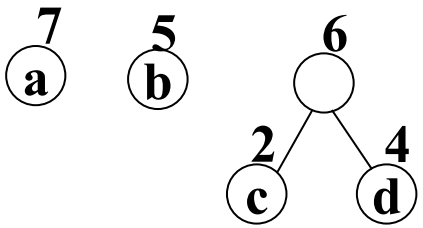
- 构造Huffman树的方法——Huffman算法
  - 构造Huffman树步骤
    - 根据给定的n个权值{w1,w2,……wn}，构造n棵只有根结点的二叉树，令其权值为wj
    - 在森林中选取两棵根结点权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和
    - 在森林中删除这两棵树，同时将新得到的二叉树加入森林中
    - 重复上述两步，直到只含一棵树为止，这棵树即哈夫曼树

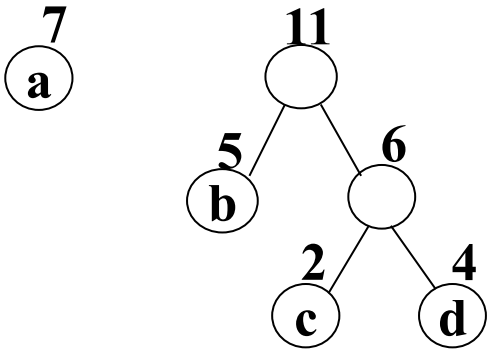
字符：	a	e	i	s	t	sp	nl
权值：	10	15	12	3	4	13	1

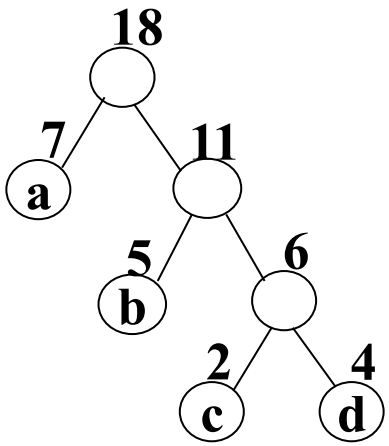
压缩与哈夫曼树

例

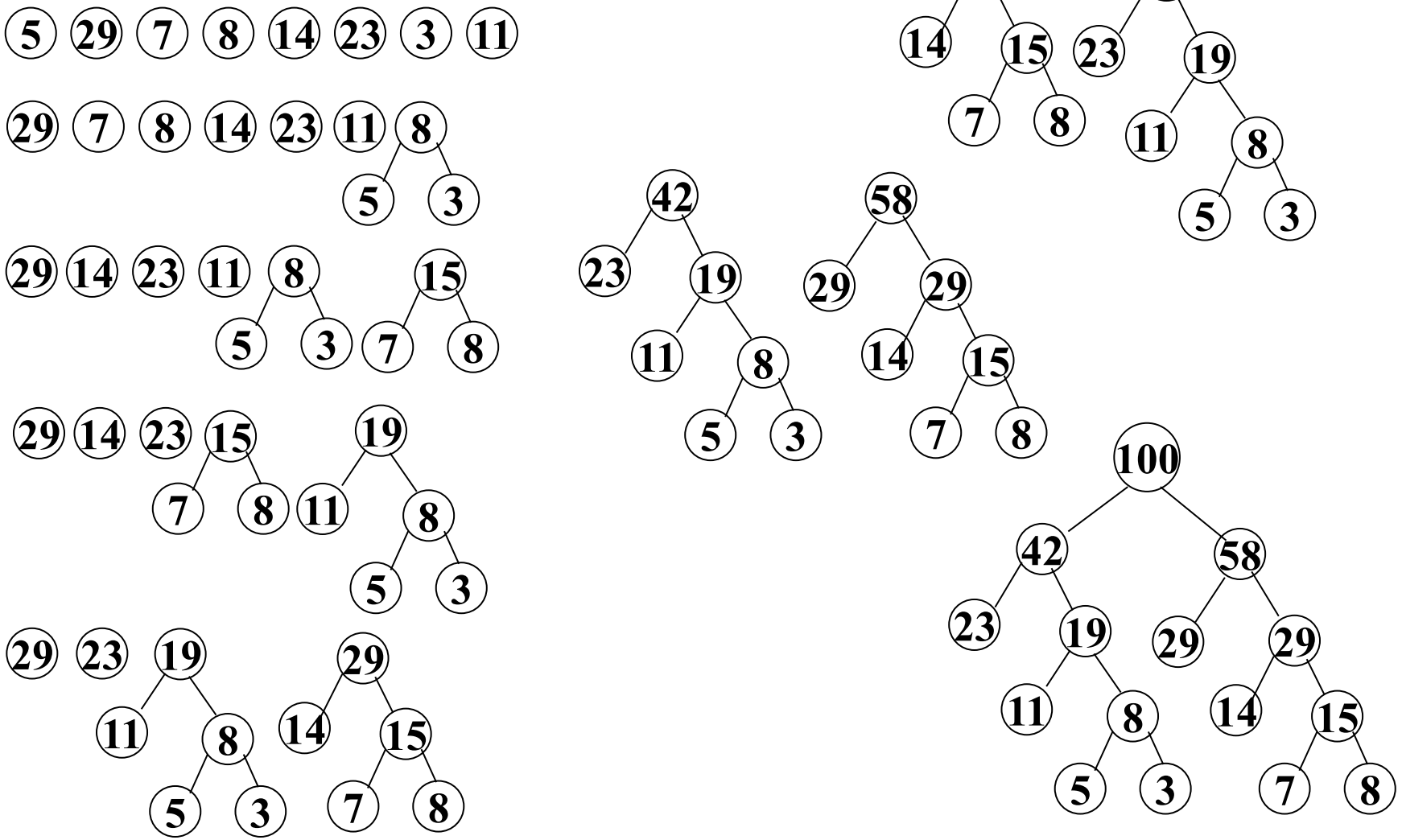








例：w={5, 29, 7, 8, 14, 23, 3, 11}



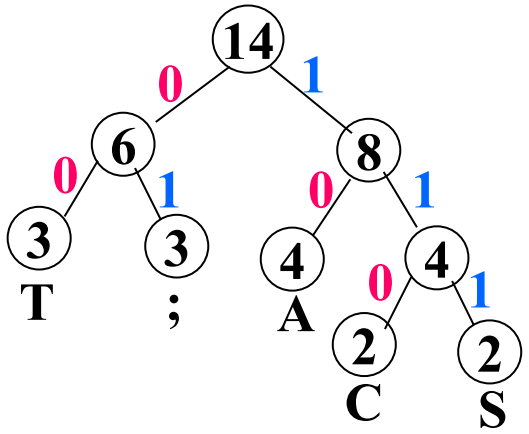
- Huffman算法实现

  - 一棵有n个叶子结点的Huffman树有2n-1个结点
  - 采用顺序存储结构——一维结构数组
  - 结点类型定义

```
typedef  struct
{  int data;
   int pa,lc,rc;
}JD;
```

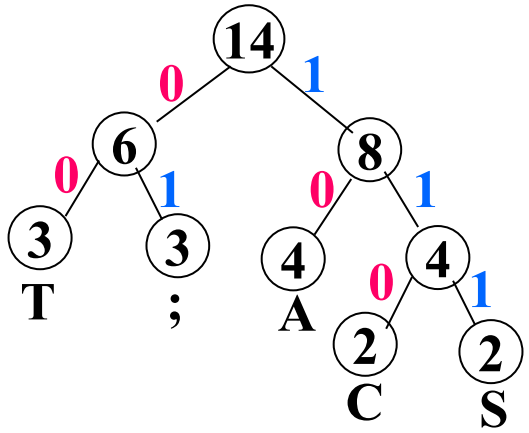
- **Huffman编码：数据通信用的二进制编码**
  - 思想：根据字符出现频率编码，使电文总长最短
  - 编码：根据字符出现频率构造Huffman树，然后将树中结点引向其左孩子的分支标“0”，引向其右孩子的分支标“1”；每个字符的编码即为从根到每个叶子的路径上得到的0、1序列

例 要传输的字符集 D={C,A,S,T, ; }
字符出现频率 w={2,4,2,3,3}



T： 00
; ： 01
A： 10
C： 110
S： 111

- 译码：从Huffman树根开始，从待译码电文中逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束



T： 00
; ： 01
A： 10
C： 110
S： 111

例 电文是{CAS;CAT;SAT;AT}
其编码 “1101011101110100001111100011000”
电文为“1101000”
译文只能是“CAT”

THE END