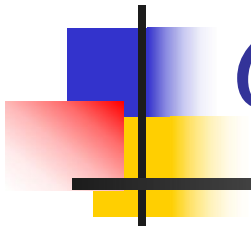
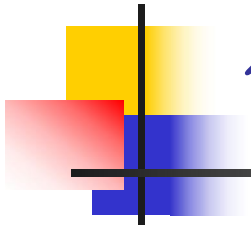


6.7 优先队列



优先队列

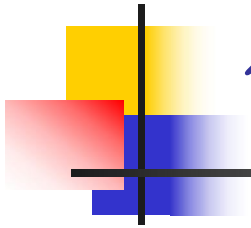
- 不同于先进先出队列的另一种队列
- 每次出队—best数据元素—**优先权最高或最小**的数据元素
- 优先队列是0个或多个元素的集合,每个元素都有一个优先权或值
- 对优先队列执行的操作有:

  1) 查找

  2) 入队--插入一个新元素

  3) 出队--删除

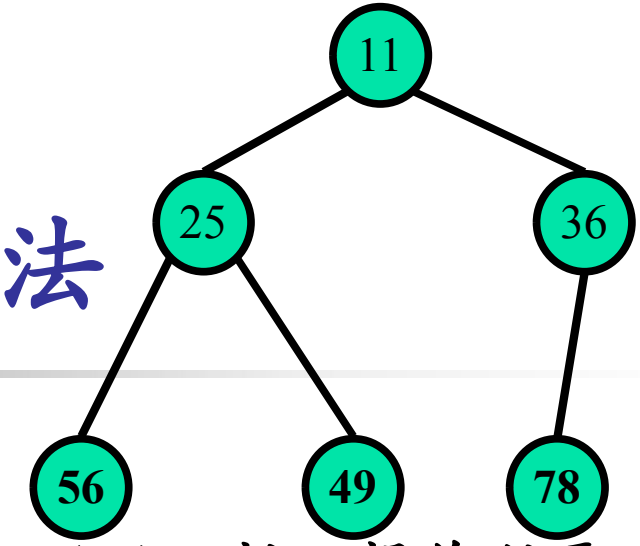
  4) increase-key(decrease-key)—修改队列中数据元素的权值



优先队列

- 在最小优先队列(min priority queue)中：
  - 查找操作用来搜索优先权最小的元素
  - 删除操作用来删除该元素
- 对于最大优先队列(max priority queue)：
  - 查找操作用来搜索优先权最大的元素
  - 删除操作用来删除该元素
- 优先权队列中的元素可以有相同的优先权

优先队列实现方法



- 有序线性表

  (1) 数组： 查找、删除时间均为O(1)；插入操作所需时间为O(n).

  (2) 链表：查找、删除时间均为O(1)；插入操作所需时间为O(n).

- 堆：偏序完全二叉树，根结点的值为max（min）
  - 优先队列中含有n个元素，堆的高度O(log n)
  - 查找O(1)，删除、插入O(logn)
  - increase- key(decrease-key) O(logn)
- pairing forest

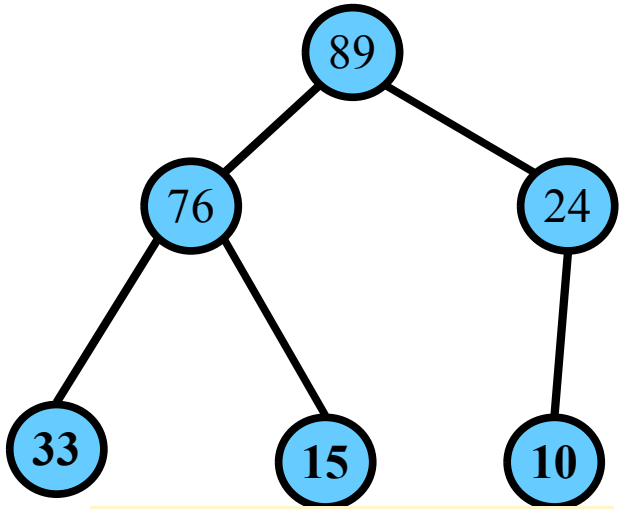
优先队列—示例中以数据元素的优先权代表数据元素

优先队列—示例中以数据元素的优先权代表数据元素

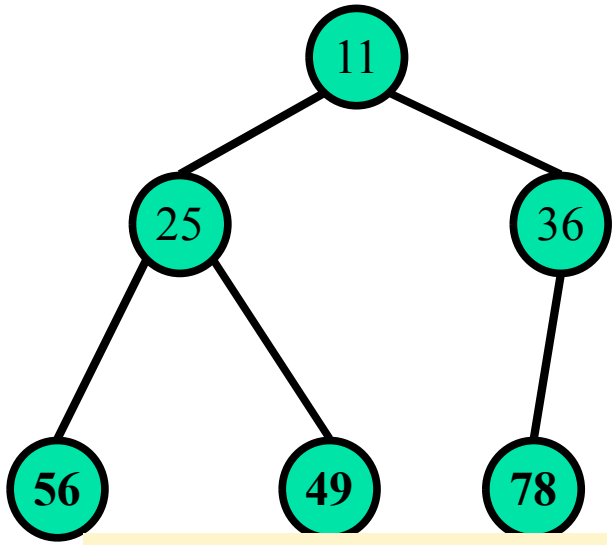
优先队列实现方法--堆

0	1	2	3	4	5	6	7
	89	76	24	33	15	10	

0	1	2	3	4	5	6	7
	11	25	36	56	49	78	



最大优先队列—大顶堆

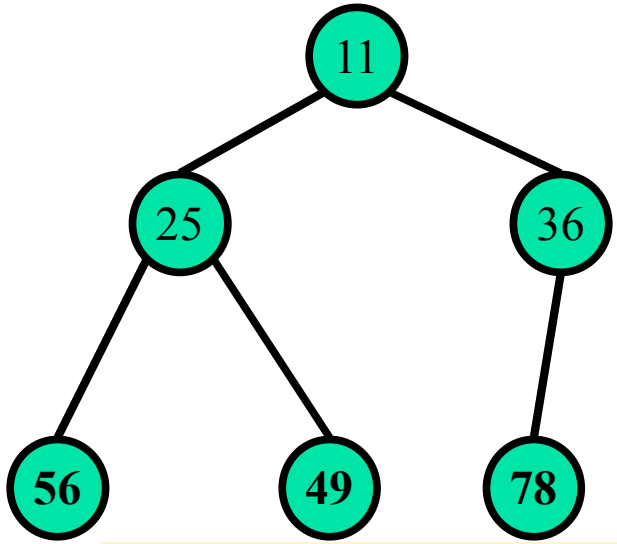


最小优先队列—小顶堆

优先队列实现方法--堆

- 入队操作--将优先权为9的数据元素入队，O(log*n*)

0	1	2	3	4	5	6	7
	11	25	36	56	49	78	



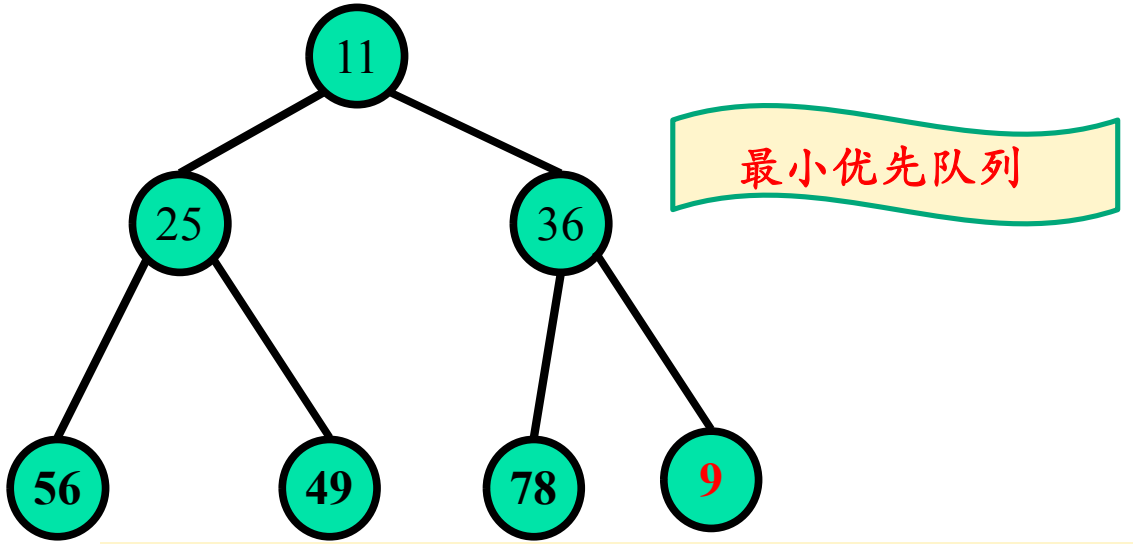
最小优先队列

优先队列—示例中以数据元素的优先权代表数据元素

优先队列实现方法--堆

- 入队操作--将优先权为9的数据元素入队，O(log*n*)

0	1	2	3	4	5	6	7
	11	25	36	56	49	78	9

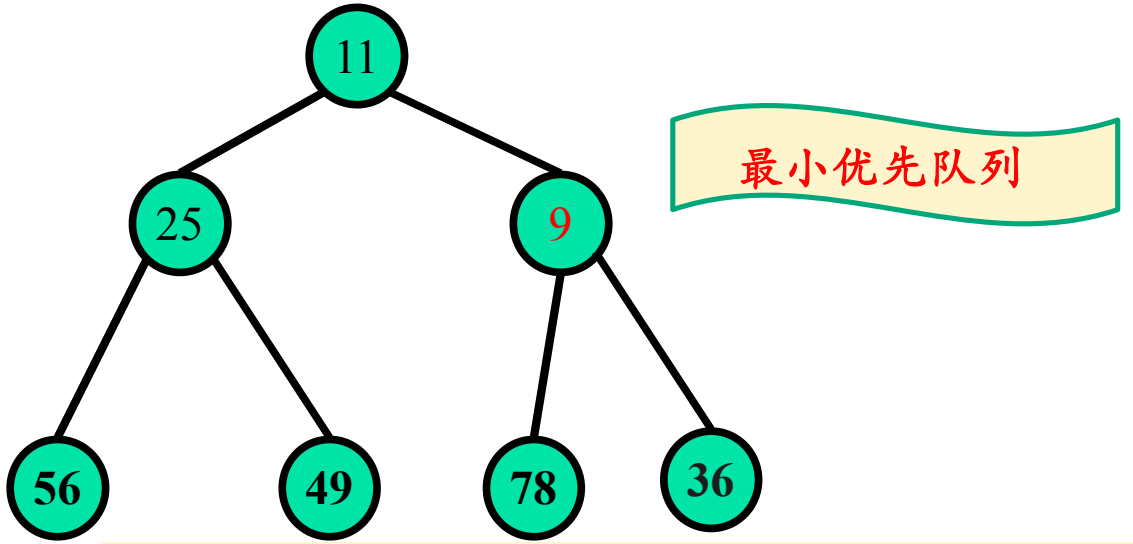


优先队列—示例中以数据元素的优先权代表数据元素

优先队列实现方法--堆

- 入队操作--将优先权为9的数据元素入队，O(log*n*)

0	1	2	3	4	5	6	7
	11	25	9	56	49	78	36

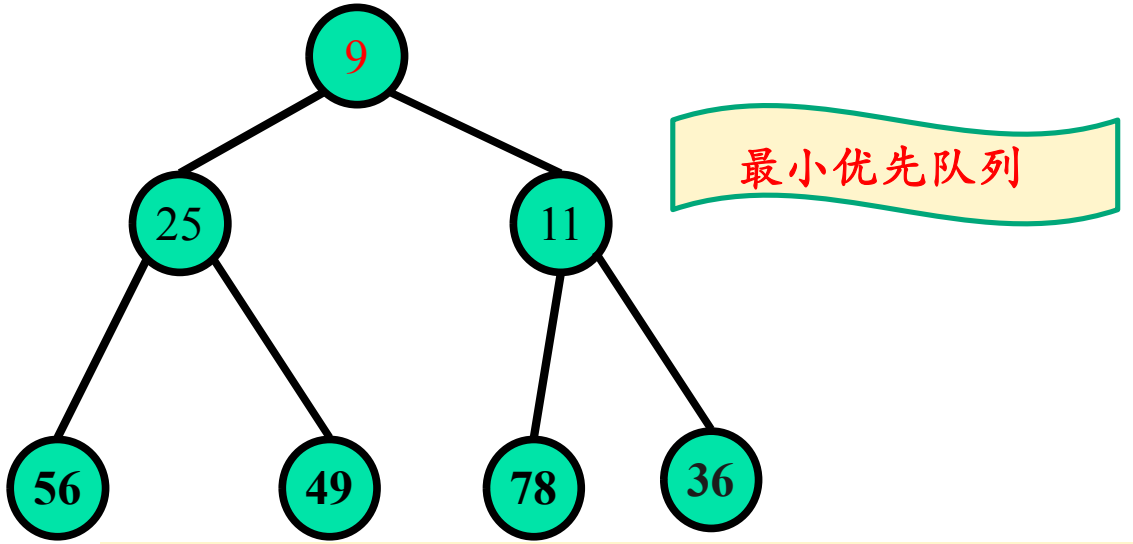


优先队列—示例中以数据元素的优先权代表数据元素

优先队列实现方法--堆

- 入队操作--将优先权为9的数据元素入队，O(log*n*)

0	1	2	3	4	5	6	7
	9	25	11	56	49	78	36

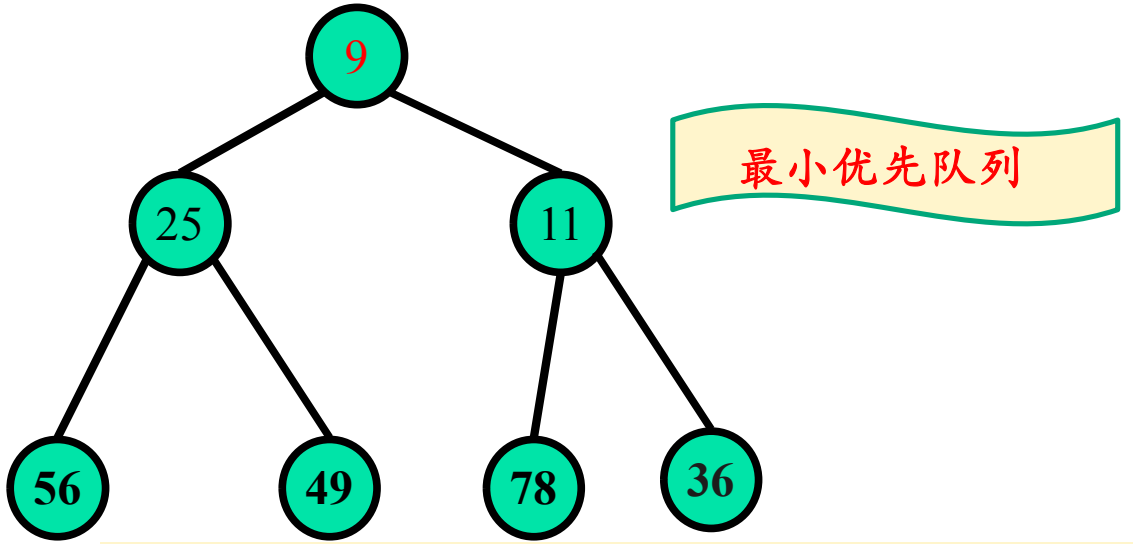


优先队列—示例中以数据元素的优先权代表数据元素

优先队列实现方法--堆

- 出队操作：O(log*n*)

0	1	2	3	4	5	6	7
	9	25	11	56	49	78	36

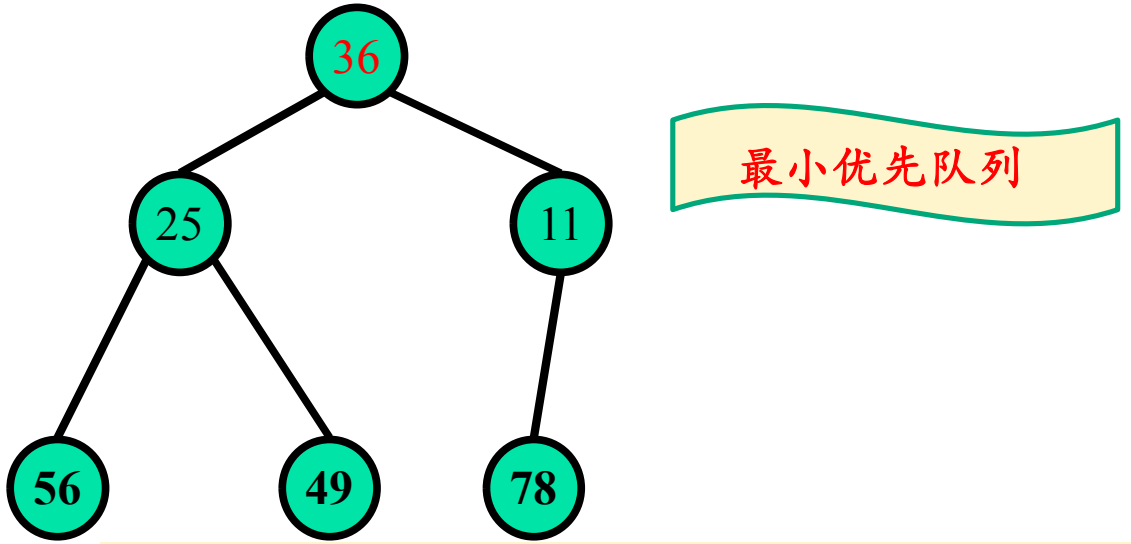


优先队列—示例中以数据元素的优先权代表数据元素

优先队列实现方法--堆

- 出队操作：O(log*n*)

0	1	2	3	4	5	6	7
	36	25	11	56	49	78	36

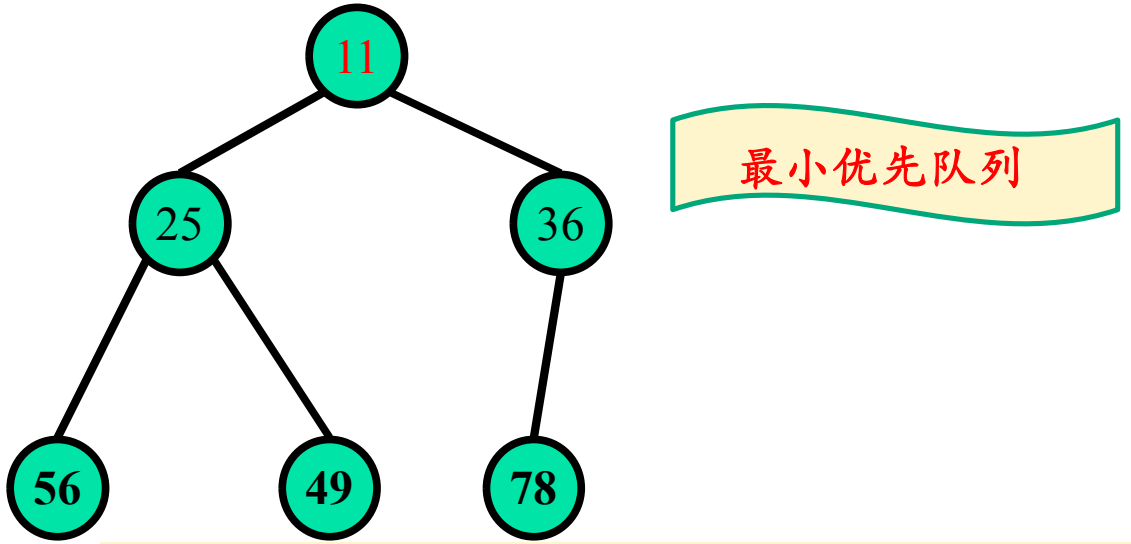


优先队列—示例中以数据元素的优先权代表数据元素

优先队列实现方法--堆

- 出队操作：O(log*n*)

0	1	2	3	4	5	6	7
	11	25	36	56	49	78	36



优先队列—示例中以数据元素的优先权代表数据元素

最小优先队列

小顶堆----decrease-key操作

- decrease-key（ Elem [ ] H, int id, float k）
- 找到数据元素（id），降低其优先权为k，保证小顶堆的性质
- 在堆中查找一个数据元素?

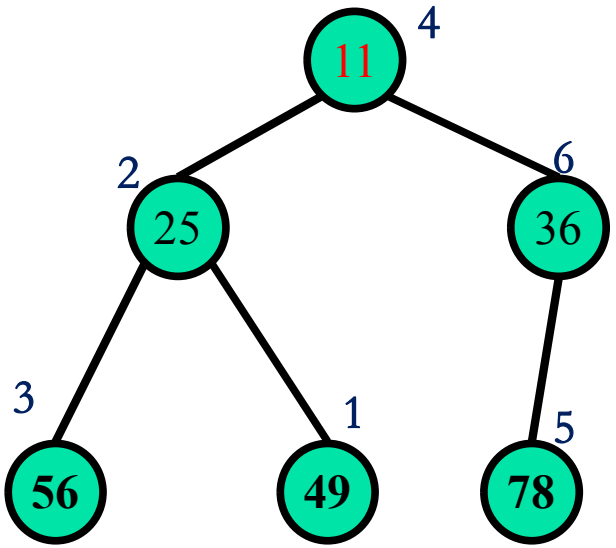
为提高查找速度，建立辅助数据结构
xref[i]:表示id=i的数据元素在堆中是否存在，若存在，给出其在堆中的位置

优先队列实现方法--堆

- decrease-key（Elem [ ] H, int id, float k）

堆

	0	1	2	3	4	5	6	7
priority		11	25	36	56	49	78	
id		4	2	6	3	1	5	



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

辅助数据结构xref[i]:表示id=i的数据元素在堆中是否存在，若存在，存放其在堆中的位置

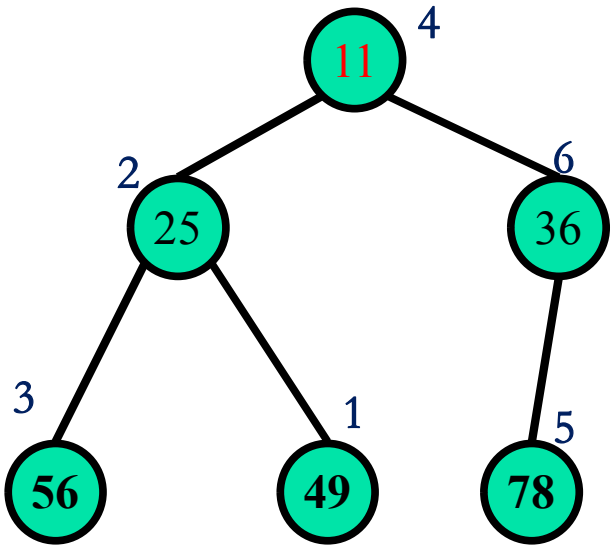
decrease-key（Elem [ ] H, int id, float k）
p=xref[id];
H[p]. priority=k;
……采用bubble up重新调整成一个堆

优先队列实现方法--堆

- decrease-key（Elem [ ] H, int id, float k）;//id=3,k=8

堆

	0	1	2	3	4	5	6	7
priority		11	25	36	56	49	78	
id		4	2	6	3	1	5	



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

辅助数据结构xref[i]:表示id=i的数据元素在堆中是否存在，若存在，存放其在堆中的位置

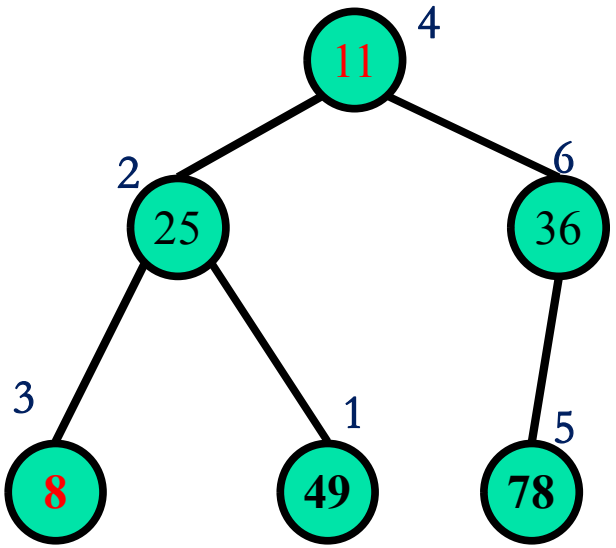
P=xref[3]=4

优先队列实现方法--堆

- decrease-key（Elem [ ] H, int id, float k）;//id=3,k=8

堆

	0	1	2	3	4	5	6	7
priority		11	25	36	8	49	78	
id		4	2	6	3	1	5	



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

辅助数据结构xref[i]:表示id=i的数据元素在堆中是否存在，若存在，存放其在堆中的位置

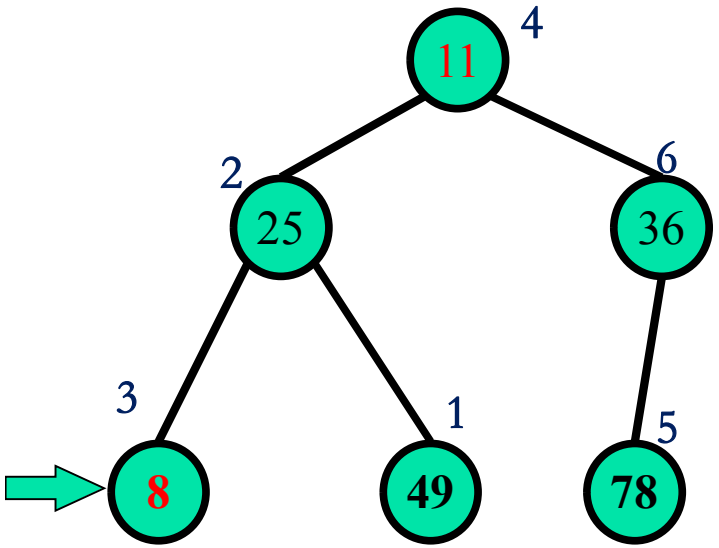
P=xref[3]=4
H[4]. priority=8

优先队列实现方法--堆

- decrease-key（ Elem [ ] H, int id, float k）;//id=3,k=8

堆

	0	1	2	3	4	5	6	7
priority		11	25	36	8	49	78	
id		4	2	6	3	1	5	



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

辅助数据结构xref[i]:表示id=i的数据元素在堆中是否存在，若存在，存放其在堆中的位置

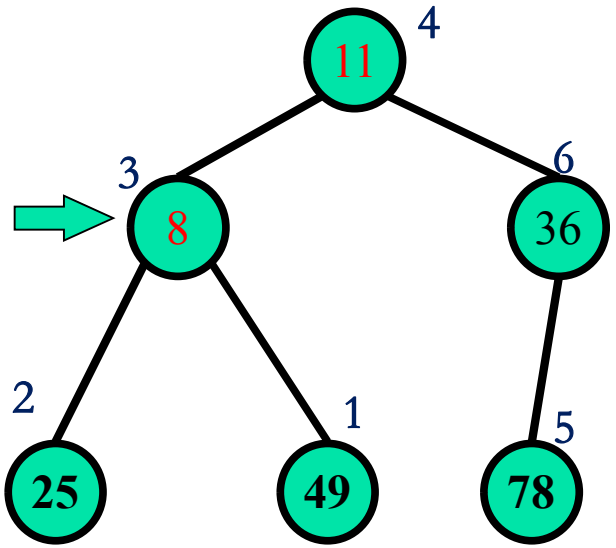
P=xref[3]=4
H[4]. priority=8
Bubble up

优先队列实现方法--堆

- decrease-key（Elem [ ] H, int id, float k）;//id=3,k=8

堆

	0	1	2	3	4	5	6	7
priority		11	8	36	25	49	78	
id		4	3	6	2	1	5	



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

辅助数据结构xref[i]:表示id=i的数据元素在堆中是否存在，若存在，存放其在堆中的位置

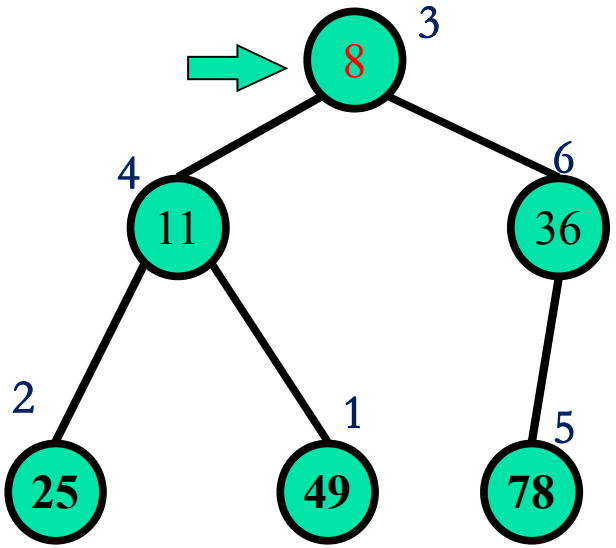
P=xref[3]=4
H[4]. priority=8
Bubble up

优先队列实现方法--堆

- decrease-key（ Elem [ ] H, int id, float k）;//id=3,k=8

堆

	0	1	2	3	4	5	6	7
priority		8	11	36	25	49	78	
id		3	4	6	2	1	5	



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

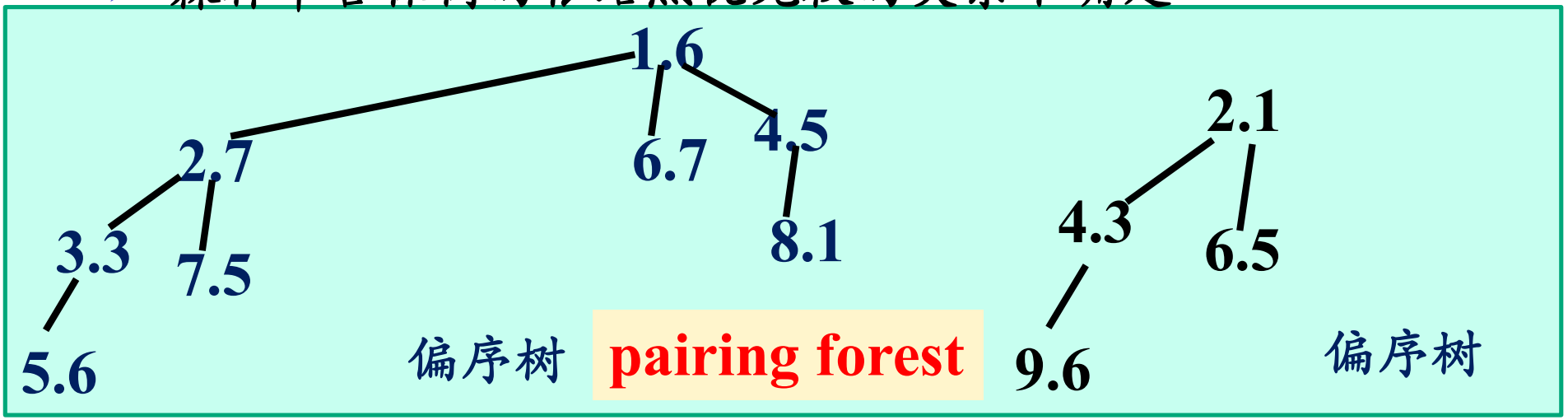
辅助数据结构xref[i]:表示id=i的数据元素在堆中是否存在，若存在，存放其在堆中的位置

P=xref[3]=4
H[4]. priority=8
Bubble up

id的管理?

优先队列实现方法-- pairing forest

- 目的：提高decrease-key操作的性能，同时不增加其他操作的代价
- 定义：优先队列中的数据元素存放于**若干**偏序树中—构成偏序森林
  - 偏序树：树中每个结点的优先权小于其孩子结点的优先权，**树根结点的优先权是树中优先权最小的数据元素**
  - 森林中各棵树的根结点优先权的关系不确定

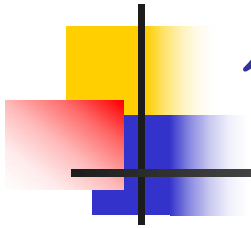


优先队列实现方法-- pairing forest

- 入队x：生成只有一个根结点的树，且根结点的优先权为x，将其加入当前的pairing forest
  - O(1)，入队操作使得森林中增加一棵新树
- 出队（删除）：
  - 若当前pairing forest中只有一棵树，则将其根结点的数据元素出队（删除），其子树森林为出队操作后的*pairing forest*。
  - 若当前pairing forest中存在k>1棵树，从k棵树中选根结点优先权最小的数据元素出队（删除）

优先队列实现方法-- pairing forest

- 从k棵树中选根结点优先权最小的数据元素—k-1次比较
- 锦标赛方法：森林中的树两两比较，根结点优先权大的作为根结点优先权小的子树----重复这个操作知道森林中只有一棵树，该树根结点的优先权即是优先级最小的

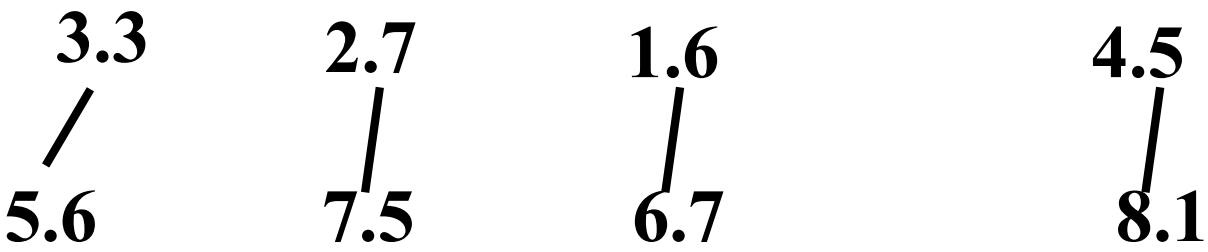


锦标赛法----实现删除操作

5.6　3.3　7.5　2.7　6.7　1.6　4.5　8.1

pairing forest

锦标赛法----实现删除操作

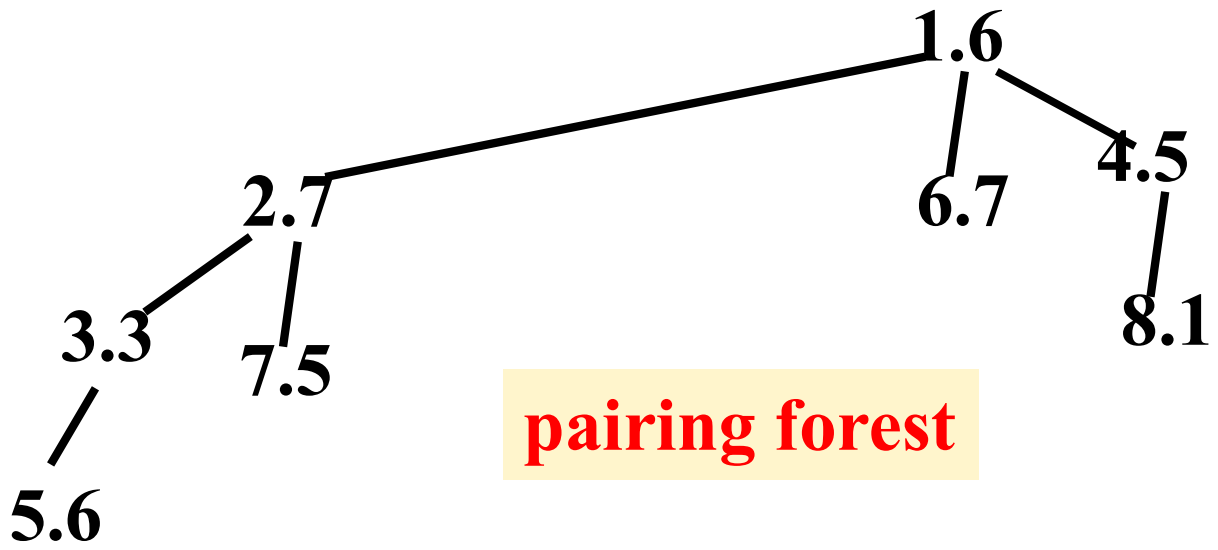


pairing forest

锦标赛法----实现删除操作

pairing forest

锦标赛法----实现删除操作



当森林中包含$n>1$棵树时，出队（删除）操作首先采用锦标赛法通过$n-1$次比较，确定优先权最小的数据----根结点的数据

锦标赛法----实现删除操作

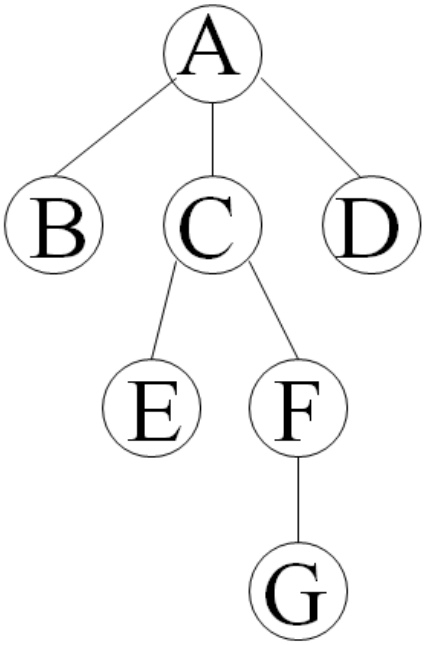
As always. for finding the minimum of k elements (k roots of trees in the initial forest), $k-1$ key comparisons are needed. Since k can be large, this operation can be expensive. However, it is not clear if it can be expensive time after time. For the example in Figure 6.26, the first getMin requires 7 comparisons. But if that element is deleted, there are only three candidates for the next minimum. The exact complexity of operations with this data structure is still not known.

pairing forest

当森林中包含$k>1$棵树时，出队（删除）操作首先采用锦标赛法通过$k-1$次比较，确定优先权最小的数据----根结点的数据----删除

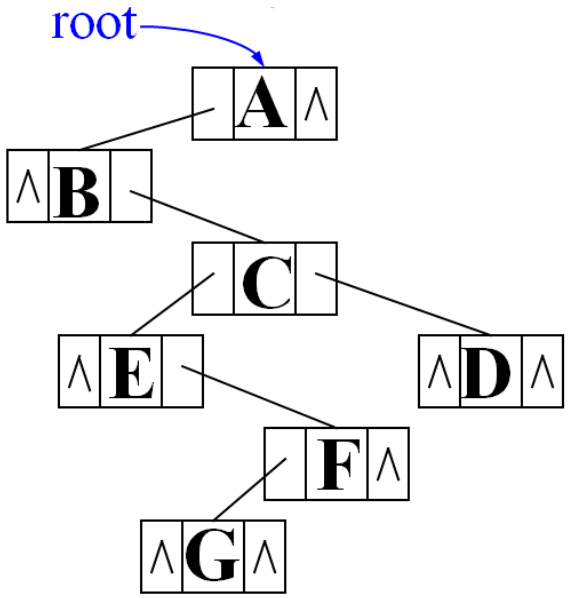
链表存放森林和树

- 孩子兄弟表示法—二叉链表



孩子兄弟表示法存储的是偏序树，树的根结点右指针为空

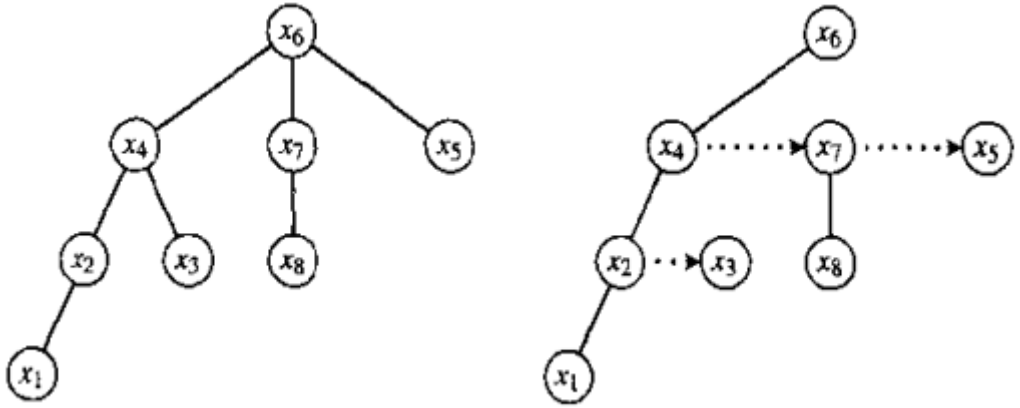
用森林中每棵树的根结点的右指针将森林中的偏序树拉成“单”链表

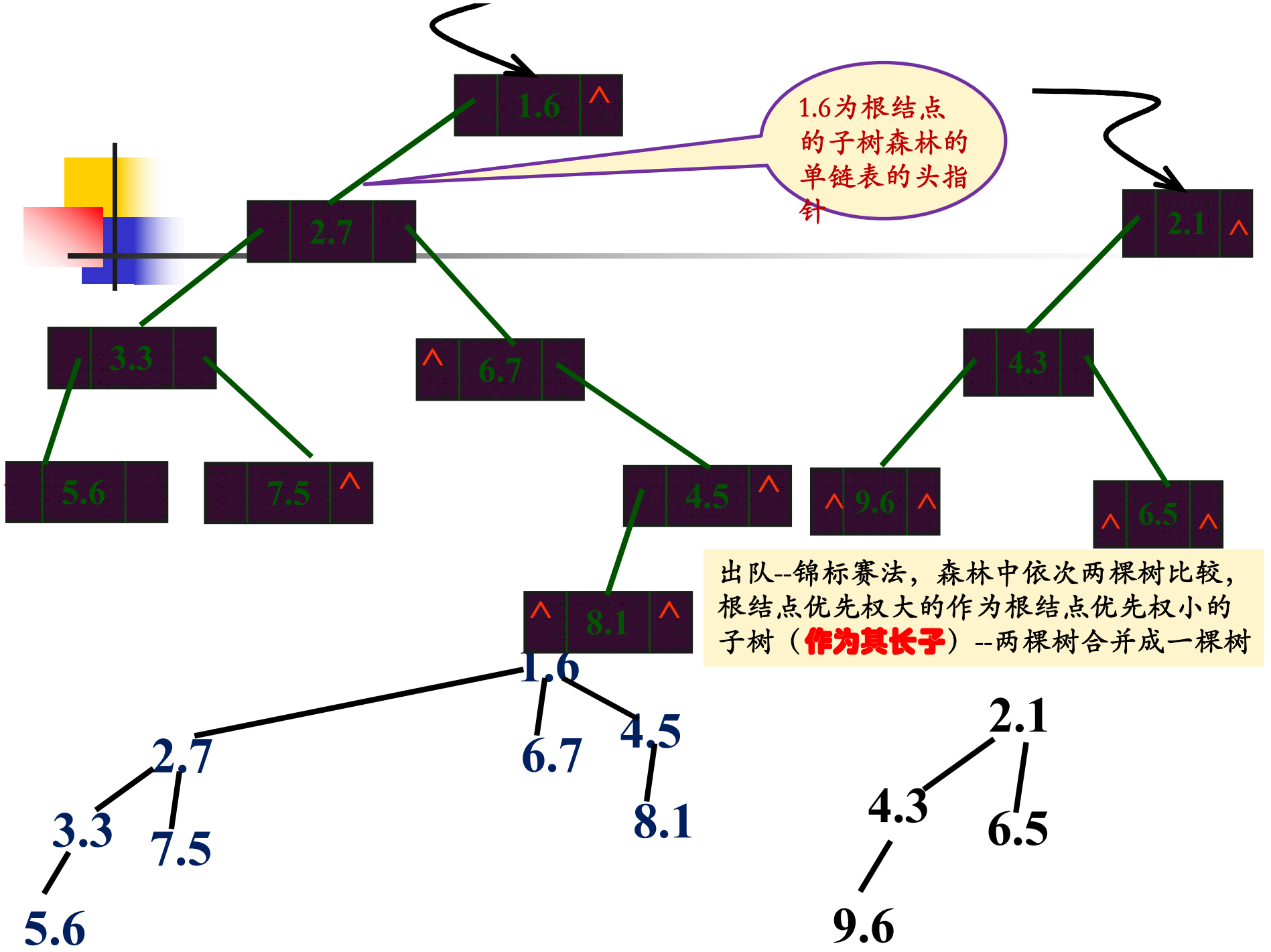


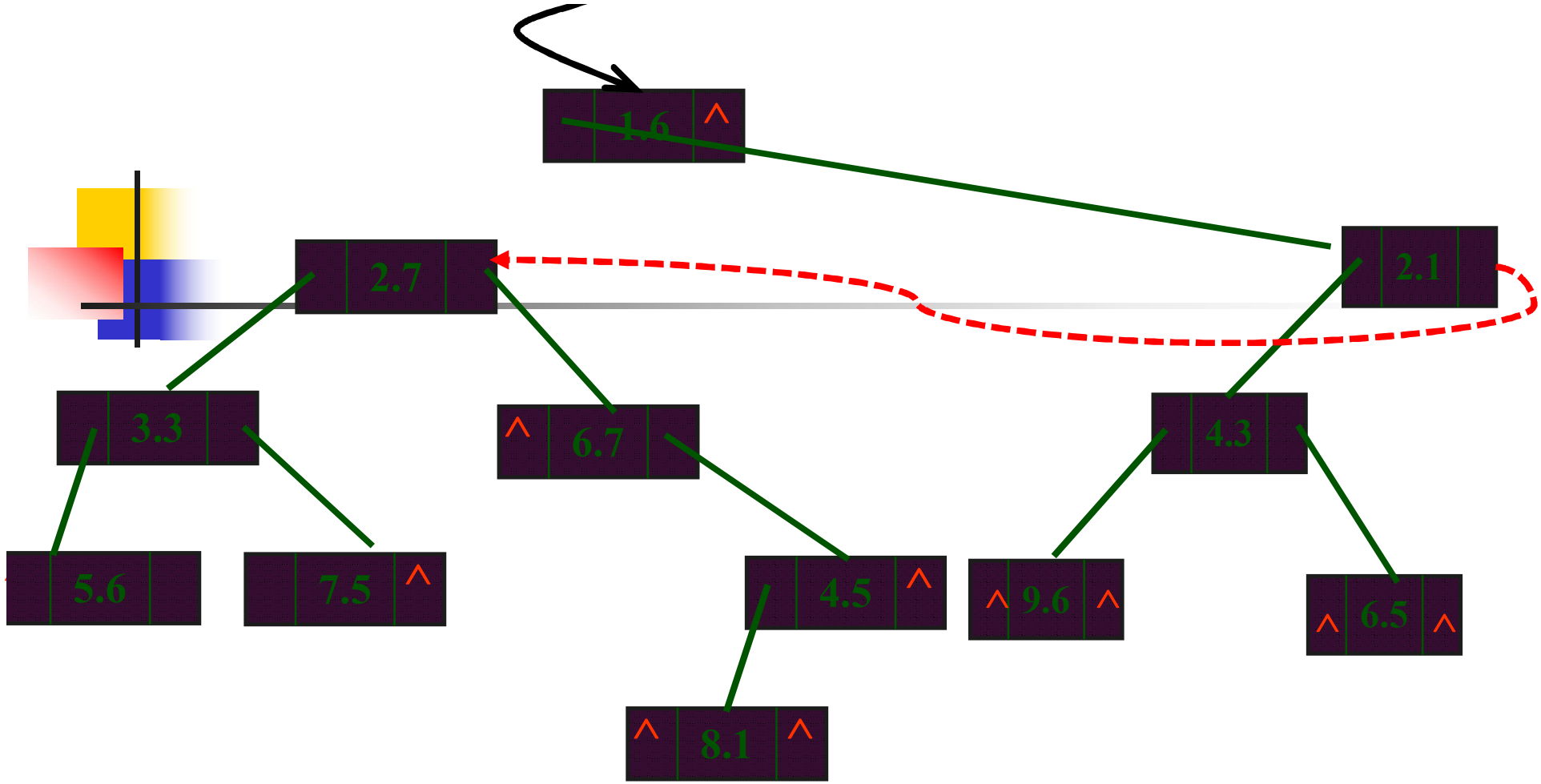
链表存放森林和树

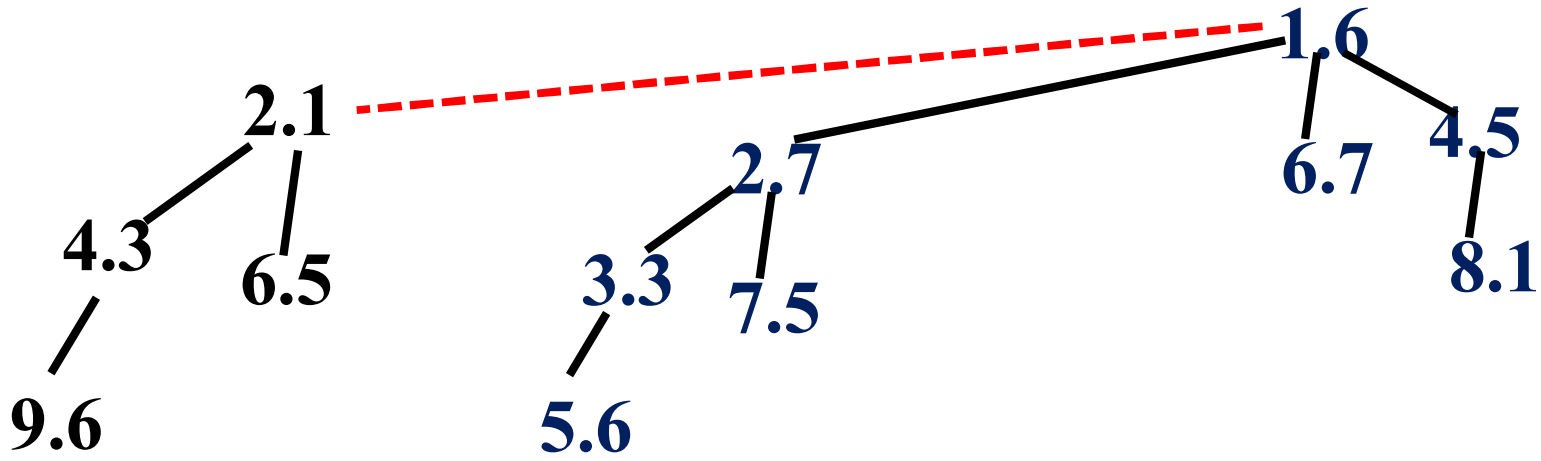
- 孩子兄弟表示法—二叉链表

```
typedef struct Tree {
    float       priority;
    struct Tree  *fc, *nb;
    int id;
    ……;
} Tree;
```




1.6
2.7
2.1
3.3
6.7
4.3
5.6
7.5
4.5
9.6
6.5
8.1
1.6为根结点的子树森林的单链表的头指针
出队--锦标赛法，森林中依次两棵树比较，根结点优先权大的作为根结点优先权小的子树（作为其长子）--两棵树合并成一棵树
1.6
2.7
6.7
4.5
3.3
7.5
8.1
5.6
2.1
4.3
6.5
9.6


1.6
2.7
2.1
3.3
6.7
4.3
5.6
7.5
4.5
9.6
6.5
8.1


2.1
4.3
6.5
9.6
1.6
2.7
6.7
4.5
3.3
7.5
8.1
5.6

链表存放森林和树

- 出队--锦标赛法，森林中依次两棵树比较，根结点优先权大的作为根结点优先权小的子树--两棵树合并成一棵树
- 实现方法：将根结点优先权大的树作为根结点优先级小的树的长子

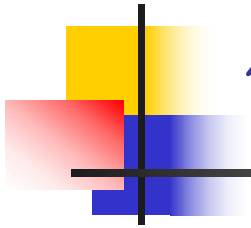
```
Tree pairTree(Tree t1,Tree t2)
{  if(t1.priority<t2.priority)
        { t2.nb=t1.fc; t1.fc=t2; return t1;}
   else
        { t1.nb=t2.fc; t2.fc=t1; return t2;}
 }
```

每一轮锦标赛法的基本操作：两两（树）比较

链表存放森林和树

一轮锦标赛算法

```
Tree pairForest(Tree t)
{   newForest=NULL;
    while (t!=NULL)
    {   t1=t;  t=t.nb; t1.nb= NULL;
        if(t!=NULL)
        {   t2=t;   t=t.nb; t2.nb= NULL;  t3=pairTree(t1,t2);
            t3.nb= newForest; newForest=t3;}
        else {t1.nb= newForest; newForest=t1}
    }
    return   newForest;  }
```



链表存放森林和树

- Tree getMin(Tree forest)

  { while(forest.nb!=NULL) forest=pairForest(forest);

  return forest.id;}

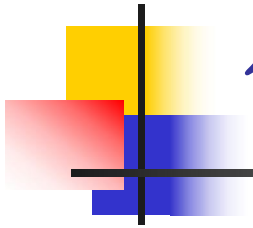
- Insert
- DeleteMin

decrease-key（Tree T, int id, float k）

- 首先查找id数据元素
- 辅助数据结构xref[i]:表示id=i的数据元素在pairing forest中是否存在，若存在，其存放的地址

```
decreaseKey(pq, v, w)  // OUTLINE
    Create newNode with id = v and priority = w.
    Tree  oldTree = xref[v];
    PairingNode  oldNode = root(oldTree);
    Tree  newTree = buildTree(newNode, children(oldTree));
    xref[v] = newTree;
    oldNode.id = −1;  // This tree is obsolete.
    pq.forest = cons(newTree, pq.forest);
```

优先队列实现方法



- 优先队列与栈和队列类似，可以将元素保存其中，可以访问和弹出
- 优先队列的的特点是存入其中的每项数据都另外附有一个数值，表示其优先成度，即优先级。
- 优先队列保证，在任何时候访问或弹出的，总是当时在这个结构里保存的所有元素中优先级最高（低）的 。
- 一般队列有FIFO特性,普通队列的弹出操作与数据元素进队的次序有关，优先队列的弹出操作与元素(数据)进入队列的次序无关，与优先级有关

优先队列实现方法

- 操作系统采用优先队列进行作业调度
- 服务行业的VIP服务：

1. 我们有一个每日交易时段生成股票报告的应用程序，需要处理大量数据并且花费很多处理时间。客户向这个应用程序发送请求时，实际上就进入了队列。我们需要首先处理优先客户再处理普通用户
2. 一个电商网站搞特卖或抢购，用户登录下单提交后，用普通队列是FIFO的，这里有个需求是，用户会员级别高的，可以优先抢购到商品，可能这个时间段的级别较高的会员用户下单时间在普通用户之后，这个时候使用优先队列代替普通队列，基本能满足我们的需求。

优先队列实现方法

- 有序线性表

(1) 数组： 查找、删除时间均为O(1)；插入操作所需时间为O(n).

(2) 链表：查找、删除时间均为O(1)；插入操作所需时间为O(n).

- 堆：偏序完全二叉树，根结点的值为max（min）

- 优先队列中含有n个元素，堆的高度O(log n)
- 查找O(1)，删除、插入O(logn)
- increase- key(decrease-key) O(logn)

- pairing forest

- 查找、删除
- 插入O(1)
- increase- key(decrease-key) O(1)