斐波那契堆

赵建华

南京大学计算机系

斐波那契堆

- 支持可合并堆
- 某些操作可以在常量的摊销时间内完成
  - Insert，Union，Decrease-Key
  - Extract-Min，Delete仍然是O(lgn)的。

可合并堆

- 支持如下五个操作的堆
  - Make-Heap()
  - Insert(H,x)
  - Minimum(H)
  - Extract-Min(H)
  - Union(H1,H2)

- 附加操作
  - Decrease-Key(H,x,k)
  - Delete(H,x)

斐波那契堆的摊销时间复杂度

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

斐波那契堆的理论和实践

- 理论上，适用于Extract-Min和Delete操作远远少于其它操作的情形。

- 实践上，
  - 常量系数和实现的复杂性使得大部分应用都不使用斐波那契堆。
  - 堆结构都不能够很好支持Search，所以Decrease-Key，Delete的参数实际上要包含指向树中元素的指针（下标）

斐波那契堆的结构

- 有根树的森林
  - 一个节点可以有多个子结点
  - 所有树的子结点在一个双向链表上
  - H.min指向最小的根。

- 满足“最小堆序”
  - 一个结点的Key总是大于等于它的父节点的Key

- 具体表示：
  - 结点包含父节点指针x.p（常量时间内找到父节点）
  - 子节点使用循环双向链表保存，x.left,x.right分别是左右子节点。（常量时间内完成结点删除、链表合并）
  - x.child指向x的子节点的链表
  - x.degree:子节点的个数
  - x.mark：标记
  - H.n：堆的结点个数

堆结构的检查

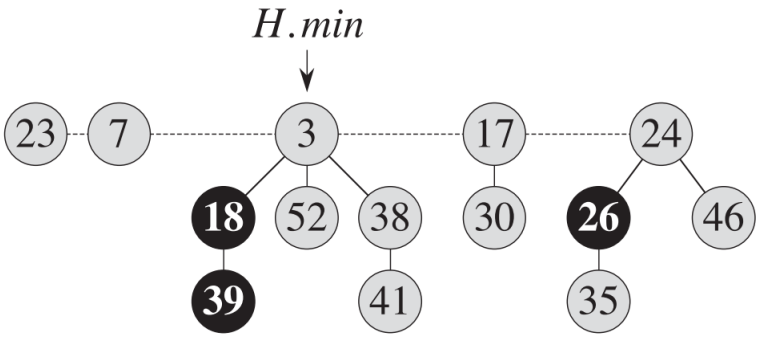
- 在实现一个复杂数据结构的时候，在各种操作之前/之后进行检查可以更快地进行错误定位。

- 斐波那契堆的检查
  - 循环双向链表的检查
  - 父节点指针的检查
  - 最小堆序的检查
  - H.min值的检查
  - D(n)的检查，即堆中结点的子节点数量的上界。

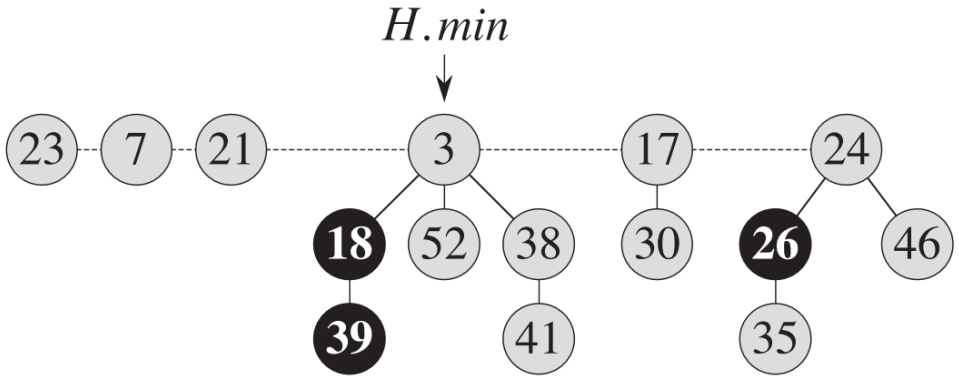
插入操作

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6        create a root list for H containing just x
 7        H.min = x
 8  else insert x into H's root list
 9        if x.key < H.min.key
10              H.min = x
11  H.n = H.n + 1
```

插入示例


H.min
23
7
3
17
24
18
52
38
30
26
46
39
41
35

(a)


H.min
23
7
21
3
17
24
18
52
38
30
26
46
39
41
35

(b)

Union的实现

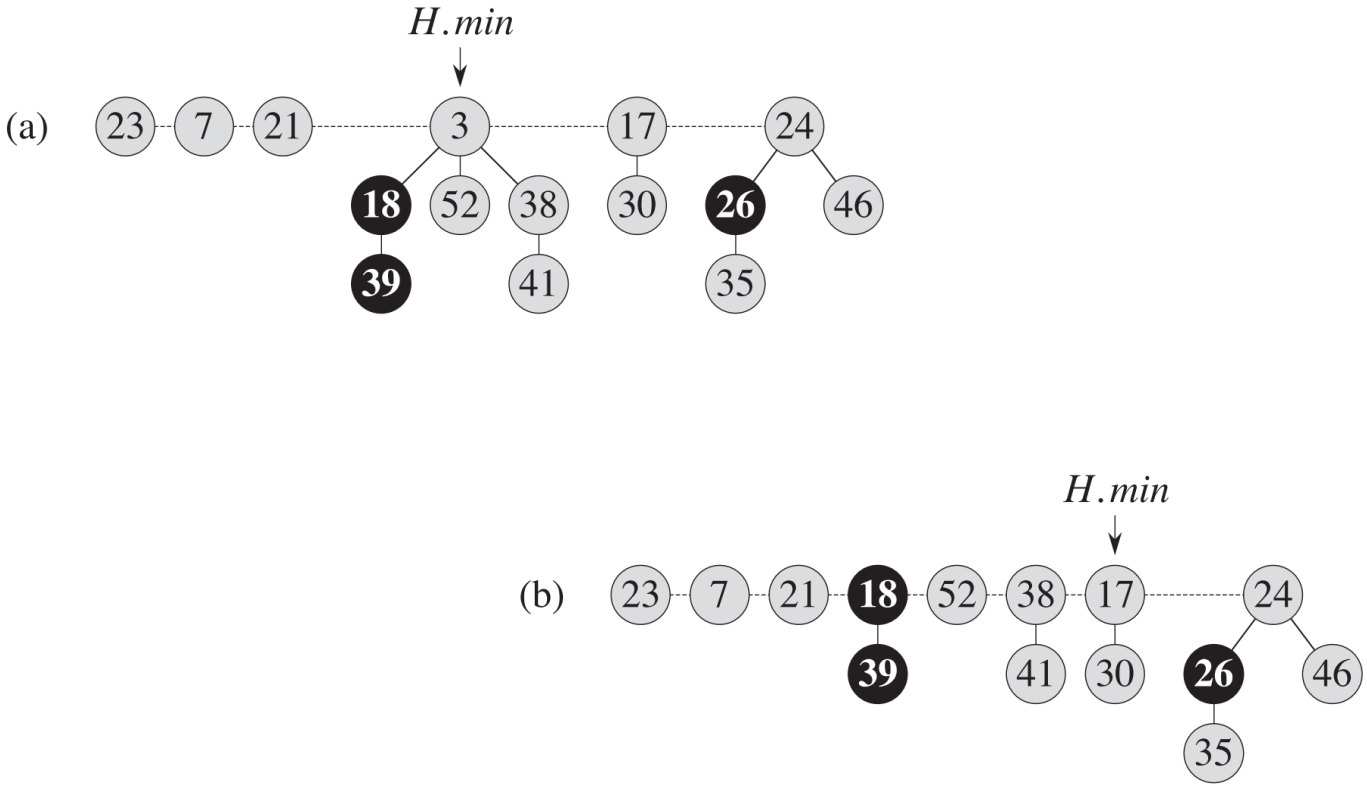
FIB-HEAP-UNION(H_1, H_2)

1 H = MAKE-FIB-HEAP()
2 $H.min = H_1.min$
3 concatenate the root list of H_2 with the root list of H
4 **if** ($H_1.min$ == NIL) or ($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
5 $H.min = H_2.min$
6 $H.n = H_1.n + H_2.n$
7 **return** H

Extract-Min的实现

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3       for each child x of z
 4            add x to the root list of H
 5            x.p = NIL
 6       remove z from the root list of H
 7       if z == z.right
 8            H.min = NIL
 9       else H.min = z.right
10            CONSOLIDATE(H)
11       H.n = H.n − 1
12  return z
```

Extract-Min示例

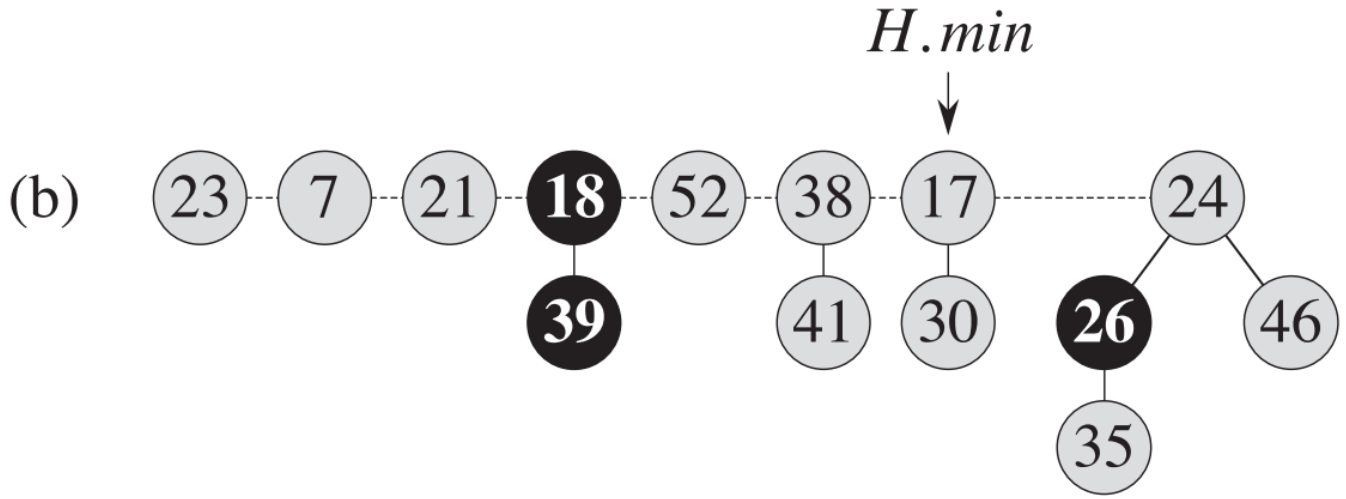


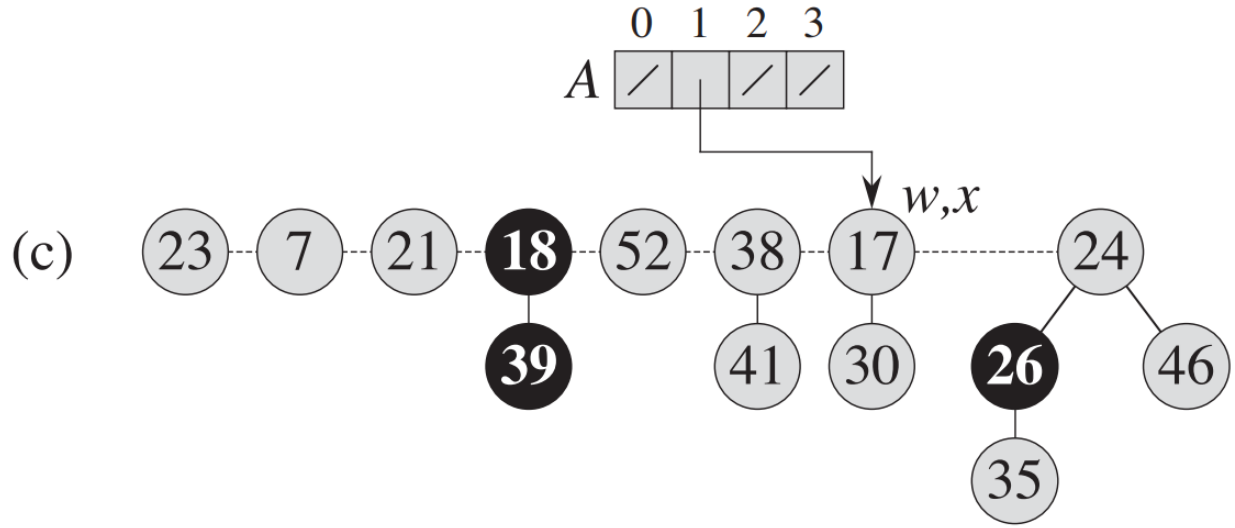
CONSOLIDATE(H)

- 通过合并相同Degree的树来降低斐波那契堆的树的数量
  - 如果x和y具有相同的Degree，且x.key <= y.key，则y成为x的子节点
  - x的度数变成x.degree + 1

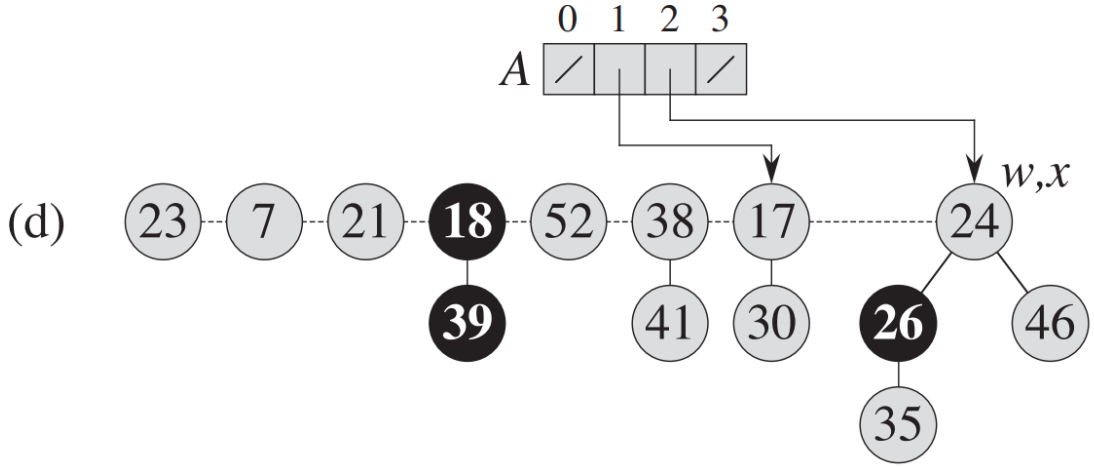
- 如何快速实现？

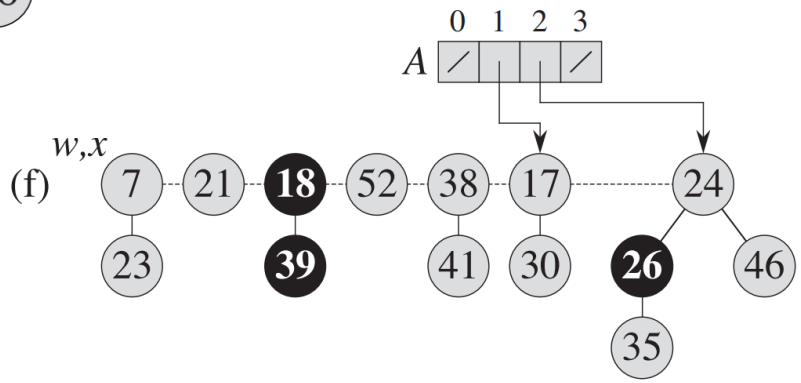
CONSOLIDATE示例（1）

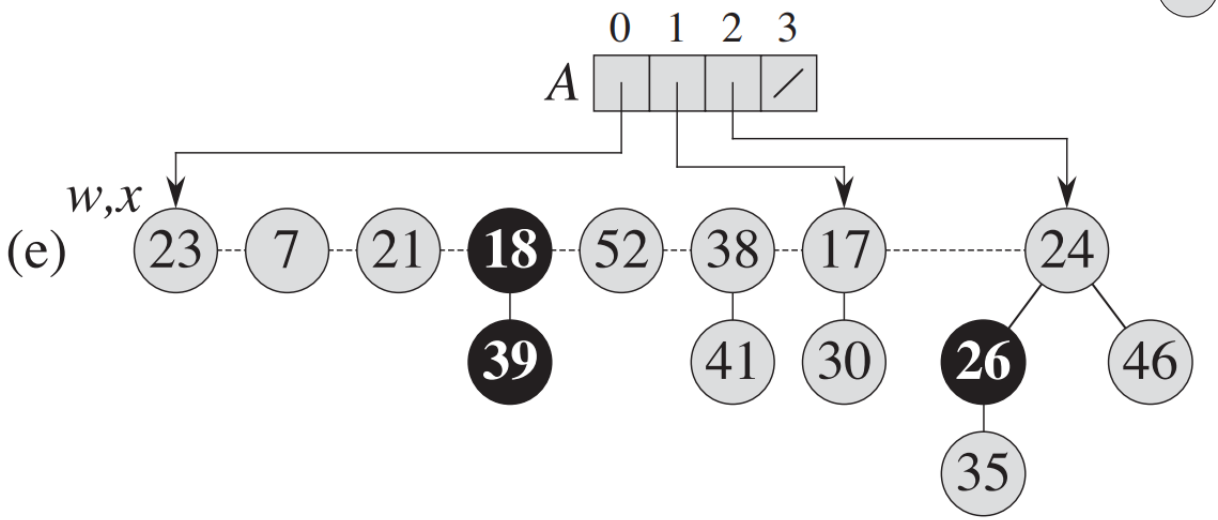




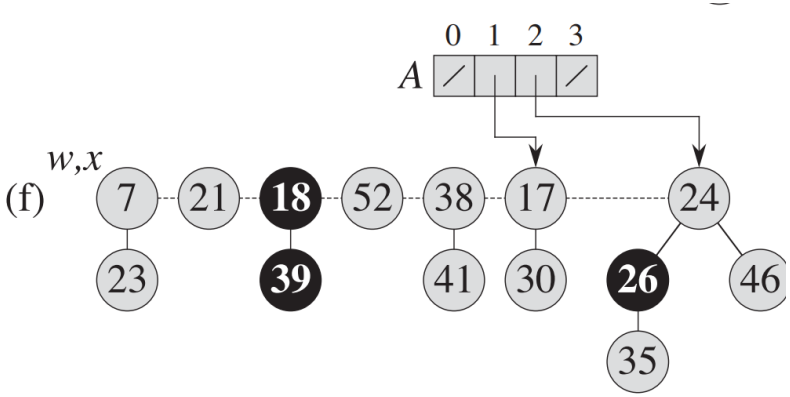
CONSOLIDATE示例（2）

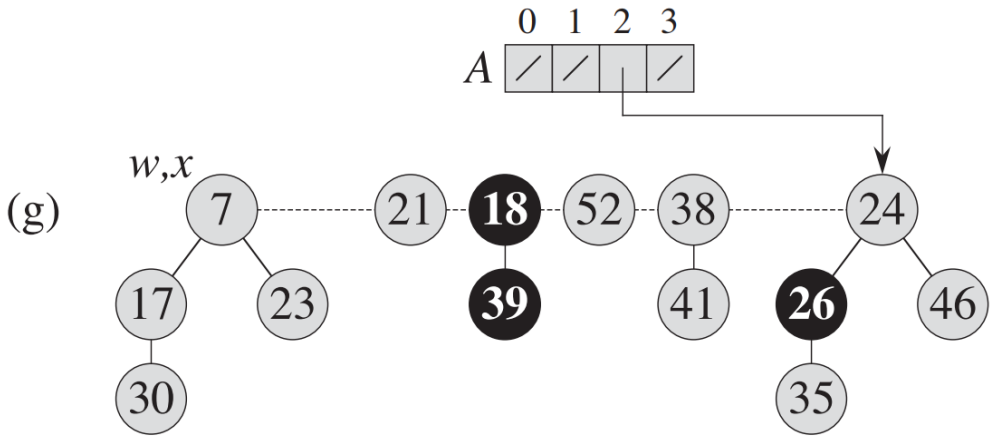


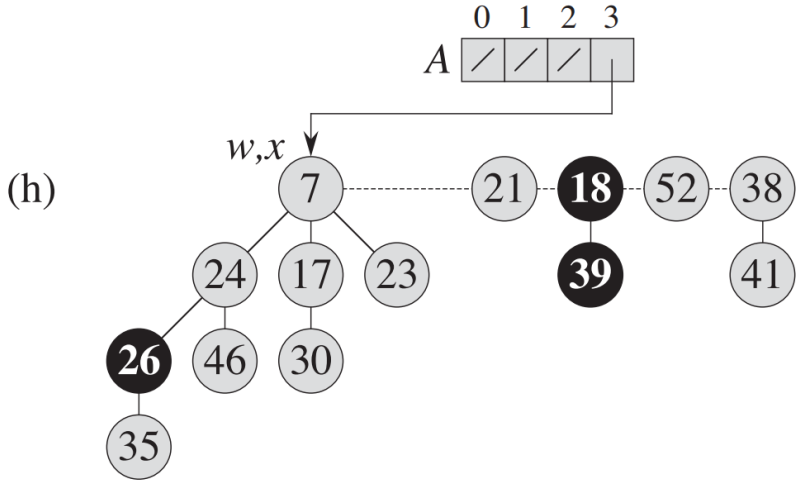




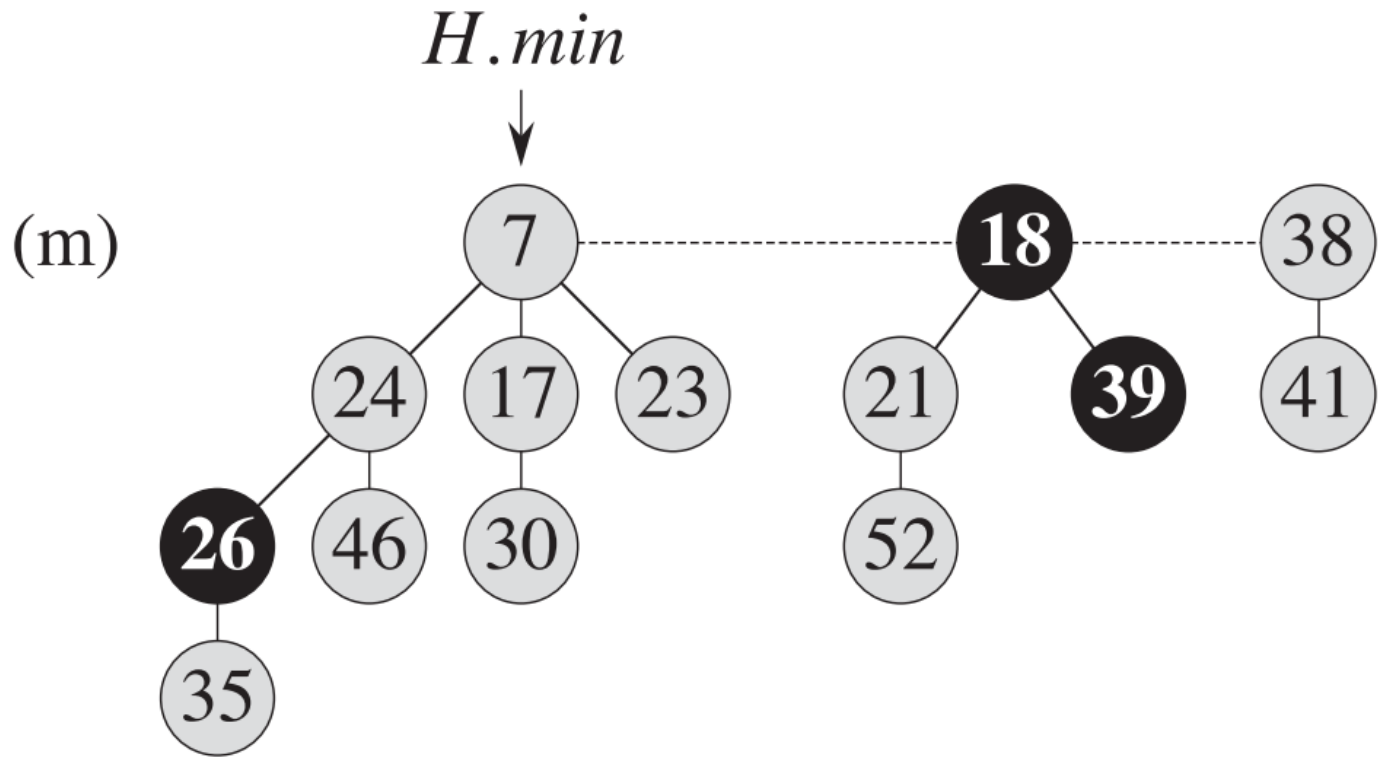
CONSOLIDATE示例（3）







继续扫描，…


(m)
H.min
7
18
38
24
17
23
21
39
41
26
46
30
52
35

```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]          // another node with the same degree as x
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = A[i]
```

```
FIB-HEAP-LINK(H, y, x)
1  remove y from the root list of H
2  make y a child of x, incrementing x.degree
3  y.mark = FALSE
```

复杂性分析

- 假设D(n)表示具有n个结点的斐波那契堆中的结点的最大度数

- CONSOLIDATE中，
  - 第一个循环和最后一个循环需要O(D(n))时间。
  - 4-14行的循环：
    - 外层循环总计需要处理D(n)+t(H)次
      - 原来的树的个数是t(H)，同时，最小结点所在的树的子节点有D(n)个。
    - 7-13行的内层while语句的每一次迭代都会减少一个root list中的结点，所以while语句的迭代次数不会多余root list中的根数量。
    - 因此，for语句所需的实际开销时间是O(D(n) + t(H))的。

如何保证合并之后的度数最多是D(n)？

- 势能的变化
  - 原值t(H)+2m(H)
  - 之后是(D(n)+1)+2m(H)
  - 均摊：实际开销时间 + 新势能 – 原势能 = O(D(n))

DecreasingKey(1)

FIB-HEAP-DECREASE-KEY(H, x, k)

```
1  if k > x.key
2        error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6        CUT(H, x, y)
7        CASCADING-CUT(H, y)
8  if x.key < H.min.key
9        H.min = x
```

- 如果X不是根节点
  - 将X从父节点中切割下来
  - 执行级联切割
  - X所在的子树被加入到堆的root list.

DecreasingKey(2)

```
CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE
```

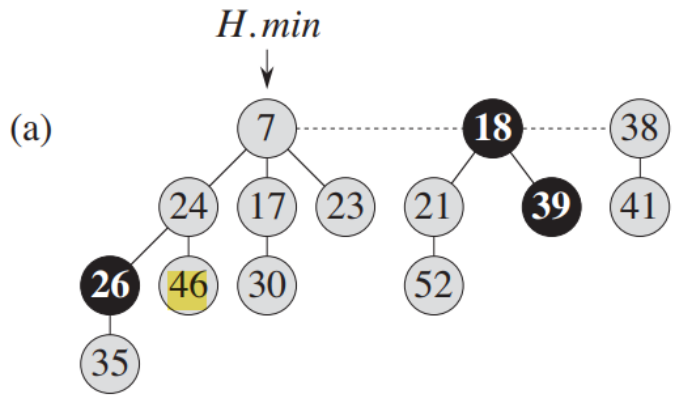
- 如果x不是根节点
  - 将x从父节点中切割下来
  - 执行级联切割
  - X所在的子树被加入到堆的root list.
- 注意x.mark被消除了。

DecreasingKey(3)

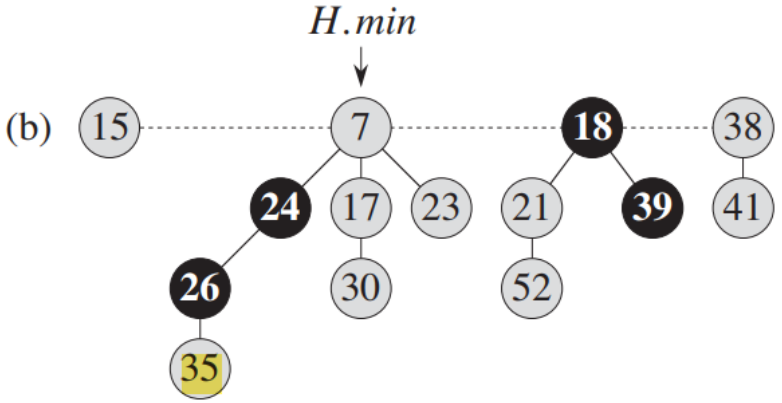
```
CASCADING-CUT(H, y)
1   z = y.p
2   if z ≠ NIL
3       if y.mark == FALSE
4           y.mark = TRUE
5       else CUT(H, y, z)
6           CASCADING-CUT(H, z)
```

- 为什么要进行级联切割？(CASCADING-CUT?)

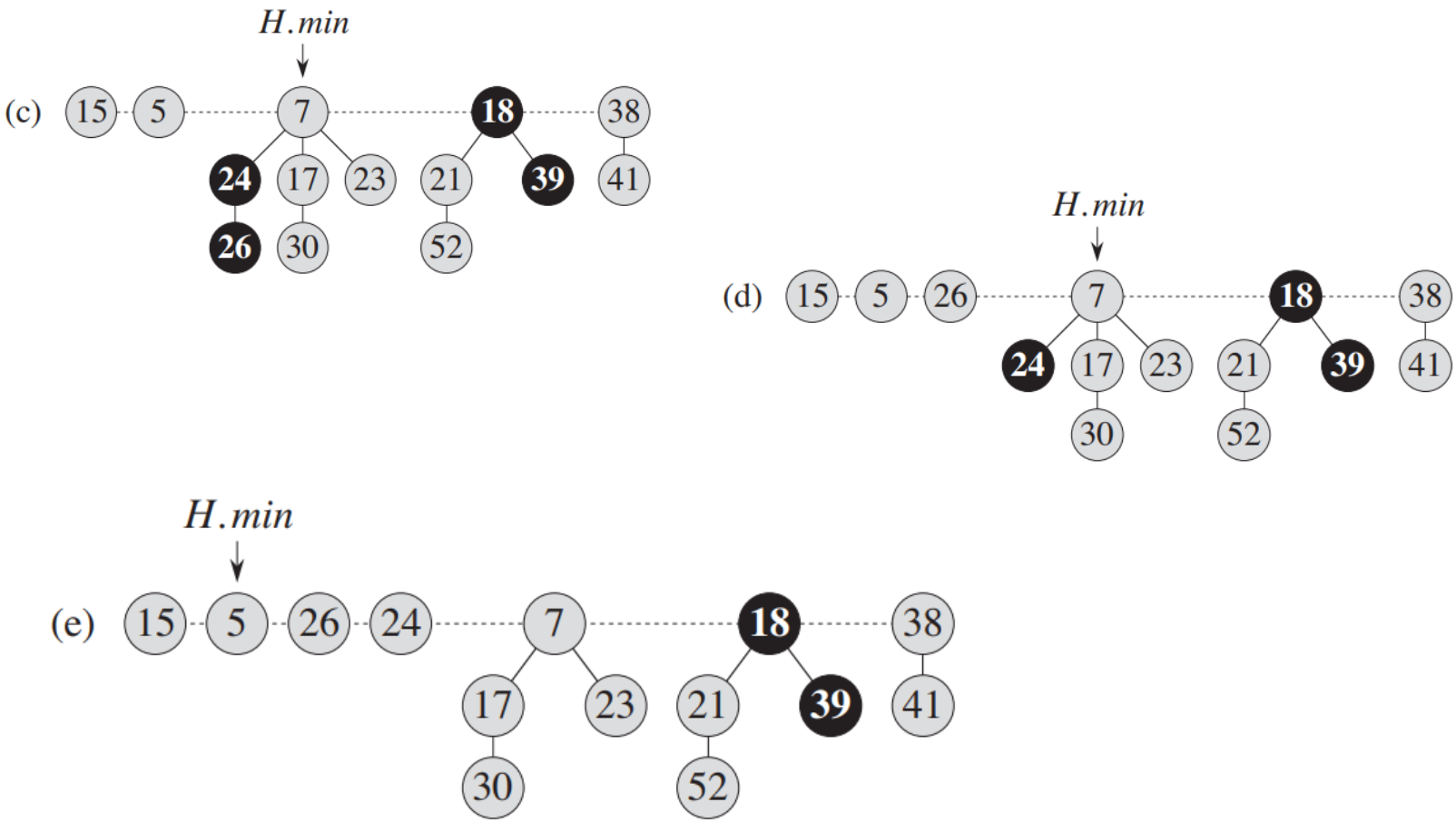
DecreaseKey演示（1）



Key 46 降低到 15



DecreaseKey演示（2）



Deleting a node

FIB-HEAP-DELETE(H, x)

1 FIB-HEAP-DECREASE-KEY$(H, x, -\infty)$
2 FIB-HEAP-EXTRACT-MIN(H)

复杂性分析——D(n)

- D(n)的上界是O(lg n)

$$D(n) \le \lfloor \log_\phi n \rfloor$$

$$\phi = (1 + \sqrt{5})/2 = 1.61803\ldots .$$

- 定义：
  - Size(x)：以x为根的子树的结点数量
  - Size(x)是x.degree的指数

引理19.1：子结点的度数

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, \ldots, k$.

Proof Obviously, $y_1.degree \geq 0$.

For $i \geq 2$, we note that when y_i was linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were children of x, and so we must have had $x.degree \geq i - 1$. Because node y_i is linked to x (by CONSOLIDATE) only if $x.degree = y_i.degree$, we must have also had $y_i.degree \geq i - 1$ at that time. Since then, node y_i has lost at most one child, since it would have been cut from x (by CASCADING-CUT) if it had lost two children. We conclude that $y_i.degree \geq i - 2$. ■

按照成为x子节点的顺序， x的第i个子节点的度数大于等于i-2
一个结点成为子节点后，最多被删除一个子节点（删除2个子节点后，会变成根结点）

斐波那契数的性质

$$F_k = \begin{cases} 0 & \text{if } k = 0\,, \\ 1 & \text{if } k = 1\,, \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2\,. \end{cases}$$

Lemma 19.2

For all integers $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i\ .$$

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fibonacci number satisfies $F_{k+2} \geq \phi^k$.

Size和Degree的关系（1）

Lemma 19.3

For all integers $k \geq 0$, the $(k+2)$nd Fibonacci number satisfies $F_{k+2} \geq \phi^k$.

Proof Let s_k denote the minimum possible size of any node of degree k in any Fibonacci heap. T
s at most size(x)
and, because addi
's size, the value
of s_k increases m
in any Fibonacci
heap, such that z
$\leq$ size(x), we
compute a lower
nd on s_k. As in
Lemma 19.1, let y
der in which they
were linked to z. T
r the first child y_1
(for which size(y_1

$$\begin{aligned}
\text{size}(x) &\geq s_k \\
&\geq 2 + \sum_{i=2}^{k} s_{y_i.degree} \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2} ,
\end{aligned}$$

Size和Degree的关系（2）

$$
\begin{aligned}
s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
&\geq 2 + \sum_{i=2}^{k} F_i \\
&= 1 + \sum_{i=0}^{k} F_i \\
&= F_{k+2} && \text{(by Lemma 19.2)} \\
&\geq \phi^k && \text{(by Lemma 19.3) .}
\end{aligned}
$$

Thus, we have shown that $\text{size}(x) \geq s_k \geq F_{k+2} \geq \phi^k$.

左式堆

赵建华

南京大学计算机系

左式堆的特点

- 有序性：每个结点中的key小于其子结点的key
- 结构：不是平衡的树，实际上通常非常不平衡
  - 左子树可能比右子树高很多
- 能够支持高效的堆合并操作
  - 两个堆的合并操作可以在O(lgN)中完成
  - 堆的插入、删除都可以转化称为堆的合并操作

左式堆的定义

- 零路径长(Null Path Length，NPL)
  - 对于结点X，Npl(X)定义为从X到一个只有零个或一个子结点的最短路径的长度。

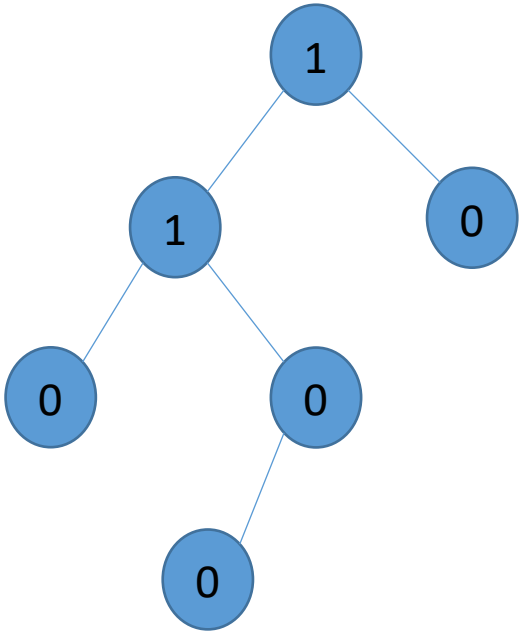
- Npl(null)等于-1
- Npl(x) = MIN(Npl(x->left), Npl(x->right)) + 1

- 对于只有0个或1个子结点的结点，其NPL为0

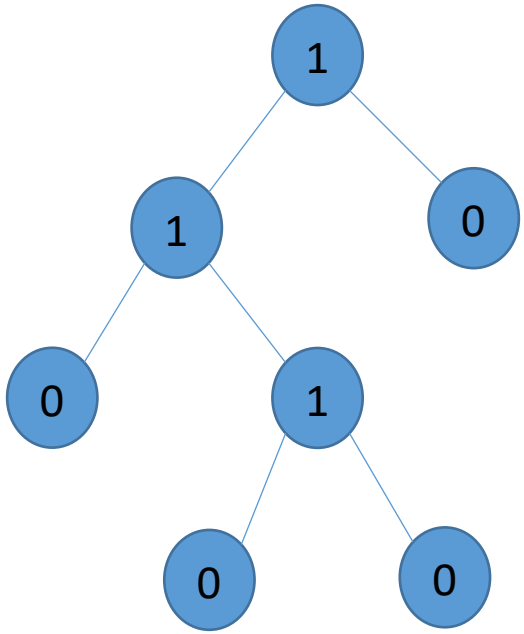
- 左式堆的定义：对于堆中每个结点X，其左子结点的Npl大于等于其右子结点的Npl，那么它称为左式堆

- 对于左式堆中的每个结点X，Npl(X) = Npl(X->right) + 1

NPL的例子



左式堆



左式堆的性质

- 如果一个左式堆的最右路径上有r个结点，那么该堆至少有2^r-1个结点
- 证明：数学归纳法
  - 当r=1时，至少有1（即2^1-1）个结点，成立
  - 假设r=1,2,…,k时成立，那么当r=k+1时，
    - 根结点的右子树的右路径含有k个结点
    - 根结点的左子树的右路径含有至少k个结点（因为根结点的左子结点的NPL大于等于根结点的右子结点的NPL）
    - 根据假设：
      - 根结点的右子树至少有2^k-1个结点，
      - 根结点的左子树至少有2^k-1个结点（可以在左子树的右路径上找到NPL等于k的结点N，根据归纳假设以N为根的子树至少有2^k-1个结点）
    - 因此，左式堆至少有$2^{k+1}-1$个结点

N个结点的左式堆的右路径最多含有log(N+1)个结点

左式堆的操作

- 合并
  - 即将两个左式堆合并称为一个左式堆；
  - 是左式堆的基本操作

- 插入：转化成为一个左式堆和一个单结点左式堆的合并

- 删除最小元素：转化称为原左式堆的左右子树（都是左式堆）的合并

左式堆的合并（1）

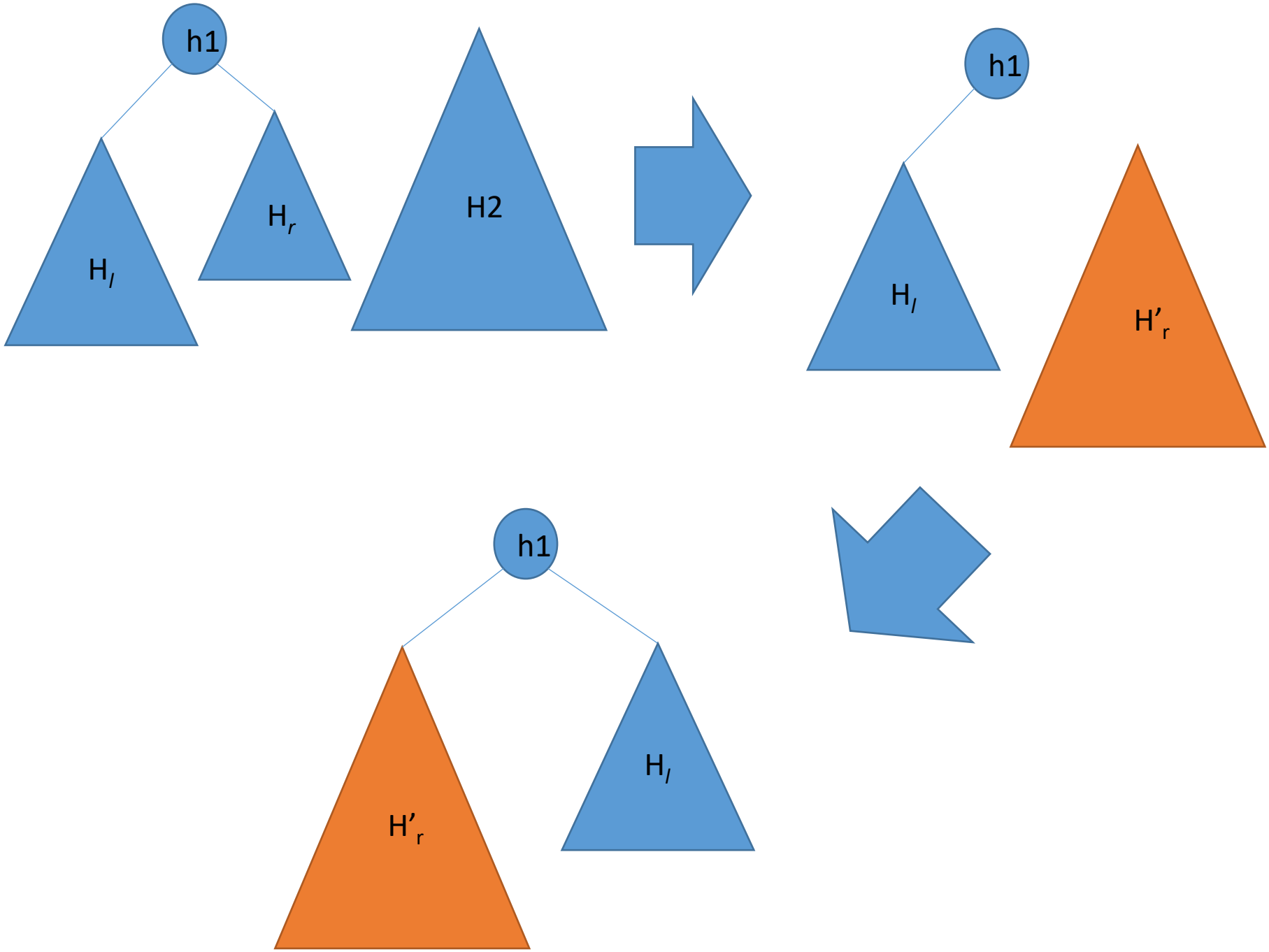
- Merge(H1,H2)
  - 将左式堆H1和H2合并，并返回一个左式堆

- 基本情况
  - H1是空堆，返回H2
  - H2是空堆，返回H1

  - H1和H2都不是空堆？

左式堆的合并（2）

- 当H1和H2都不是空堆时，最小值是H1和H2的根结点中较小的
- 如果H1的根较小
  - 将H1的右子树H_r（H1的比较矮的子树）和H2合并（递归调用），得到合并后的左式堆H'_r
  - 假设H1的原左子树为H_l，
    - 如H'_r的NPL小于等于H_l的NPL，则返回以H1的根结点为根，H_l和H'_r为左右子树的左式堆（需要重新计算根结点的Npl值）
    - 如H'_r的NPL大于H_l的NPL，则返回以H1的根结点为根，H'_r和H_l为左右子树的左式堆（需要重新计算根结点的Npl值）
- 如果H2的根较小
  - …


h1
H_l
H_r
H2
h1
H_l
H'_r
h1
H'_r
H_l

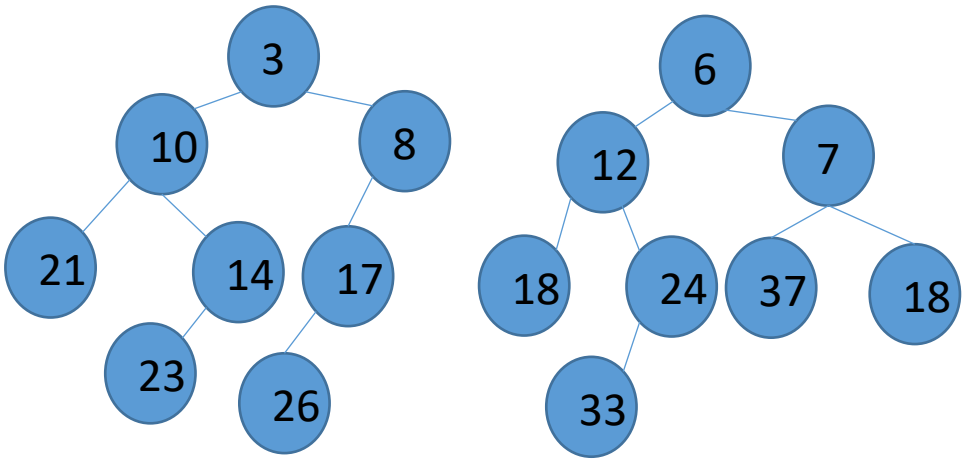
左式堆的合并（3）

- //当H1和H2都不是空堆时

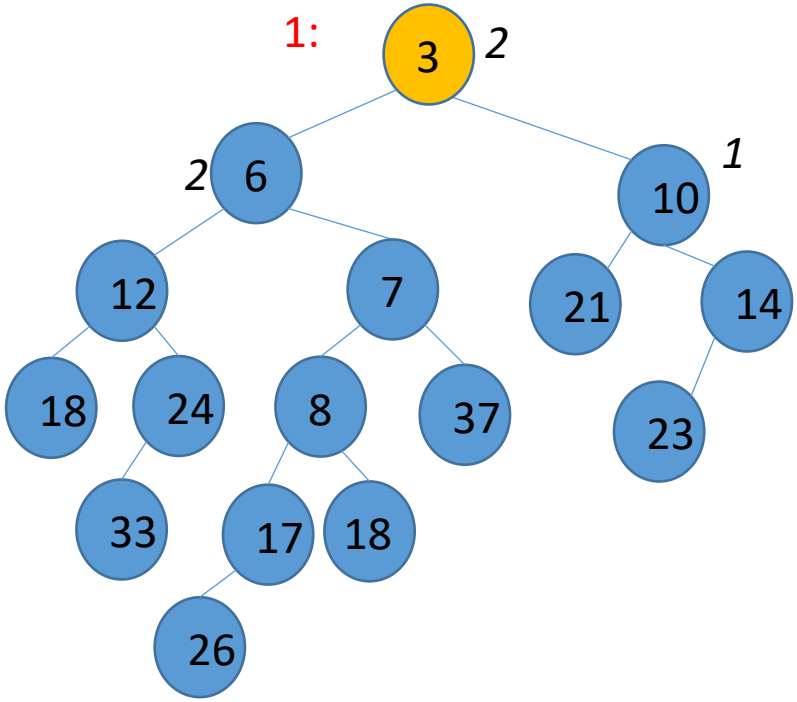
```
…
if(h1.data <= h2.data)
{
        Node * hr1 = Merge(h1->right, h2);
        if(NPL(h1->left) < NPL(hr1->npl))
        {/*对换*/          h1->right = h1->left; h1->right = hr1;          }
        else
                h1->right = hr1;

        h1->npl = NPL(h1->right) + 1;
        return h1;
}
…
```

合并的实例



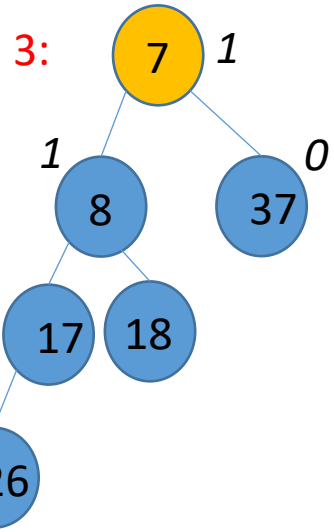
1:



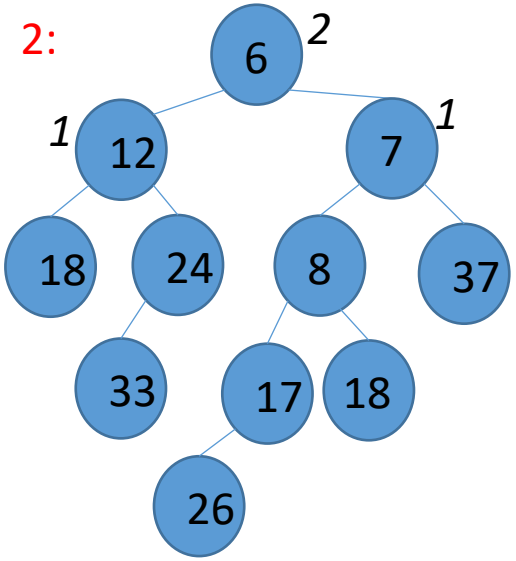
//表示以该结点为根的子堆

1. 递归：3和6合并
2. 递归：8和6合并
3. 递归：8和7合并
4. 递归：8和18合并
5. 递归：8的右子树(null)和18合并

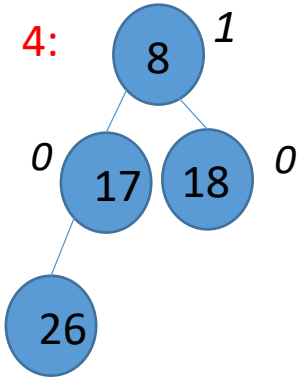
3:



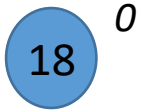
2:



4:



5:

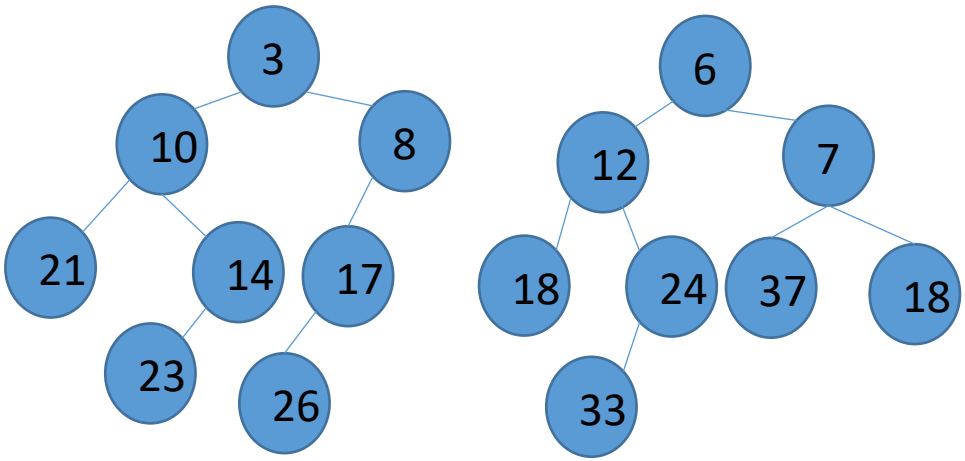


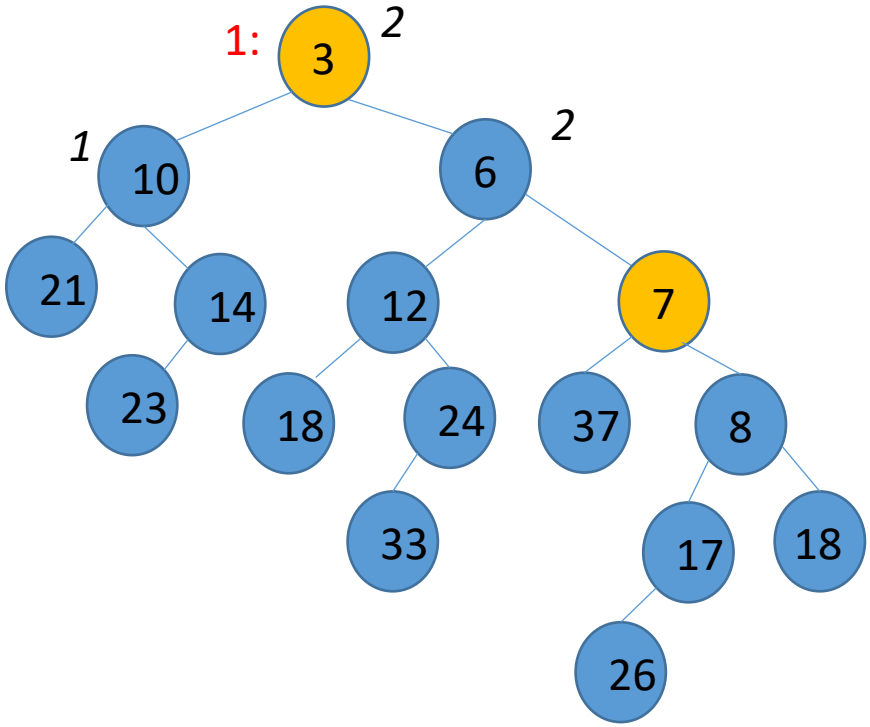
左式堆的合并（不对换）

- 合并得到的不是左式堆，但是仍然满足序性质：每个结点小于等于其子节点

```
if(h1.data <= h2.data)
{
        h1->right = Merge(h1->right, h2);
        h1->npl = (NPL(h1->left) <  NPL(h1->right)) ?
                                NPL(h1->left->npl) + 1
                :               NPL(h1->right) + 1
        return h1;
}
…
```

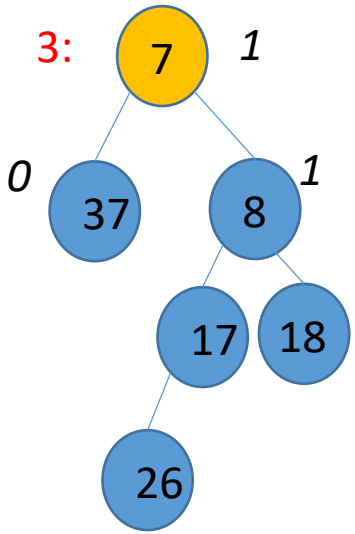
合并的实例(不对换)

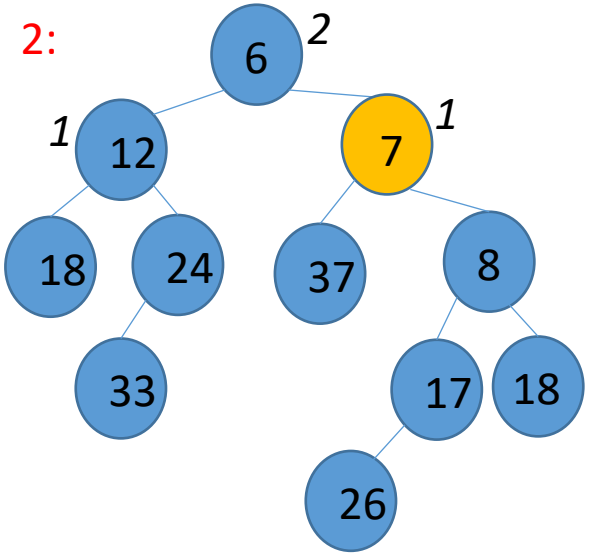


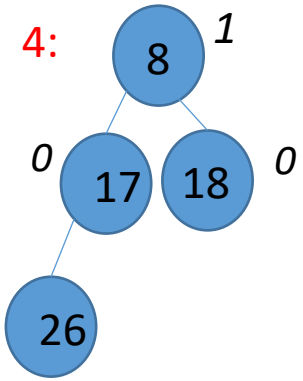


//表示以该结点为根的子堆

1. 递归：3和6合并
2. 递归：8和6合并
3. 递归：8和7合并
4. 递归：8和18合并
5. 递归：8的右子树(null)和18合并

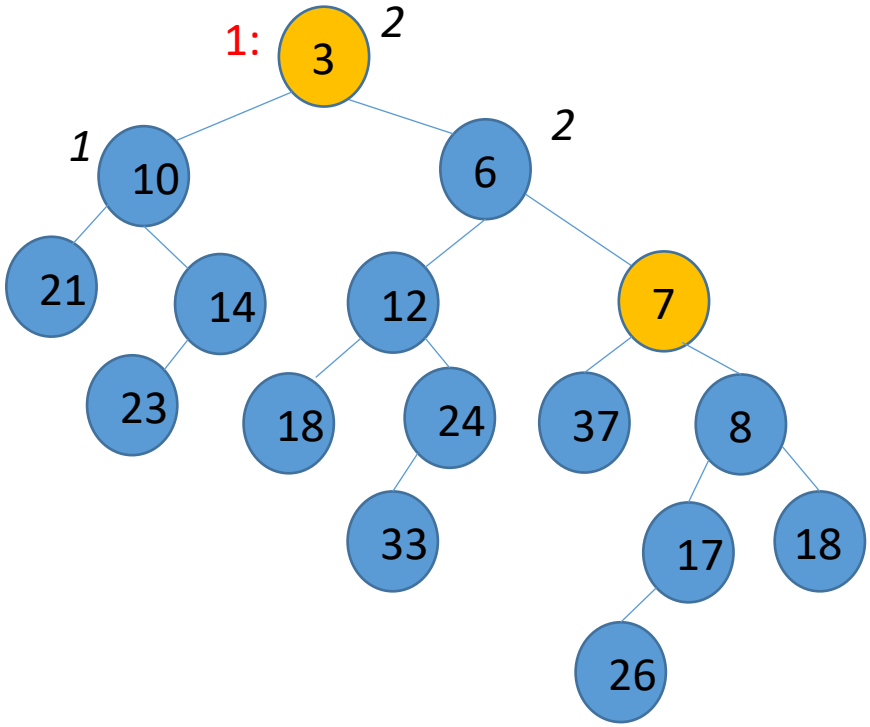






合并的实例(不对换)

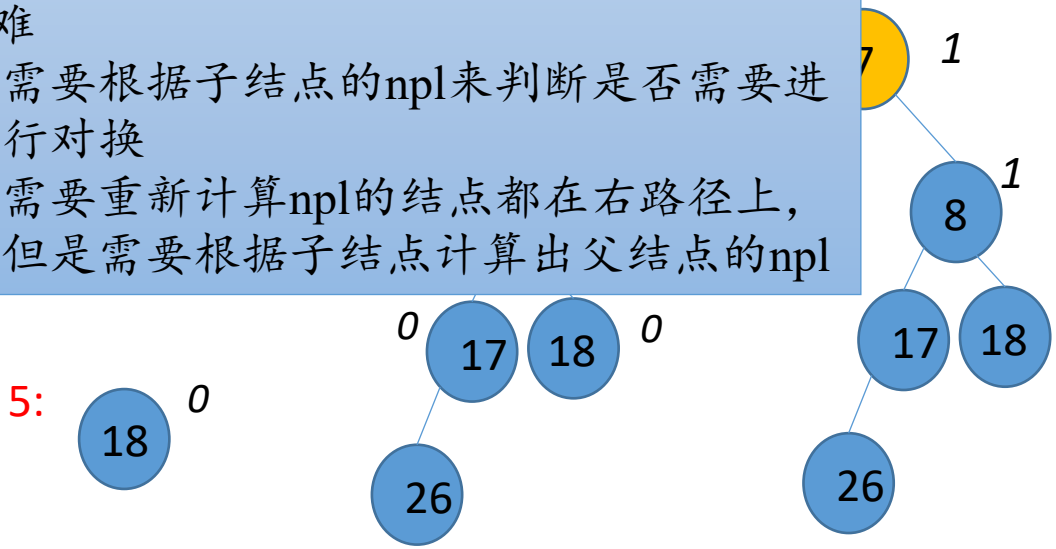
- 合并后的右路径是原来两棵树的右路径的Merge，并且右路径上的结点仍然是从小到达排序的。
- 右路径上的各个结点的左子树没有改变
- 不满足左式堆要求的结点都在右路径上，也就是说，只有右路径上结点才可能需要进行对换子节点。
- 对换一个结点的左右子树，不影响其它结点的NPL

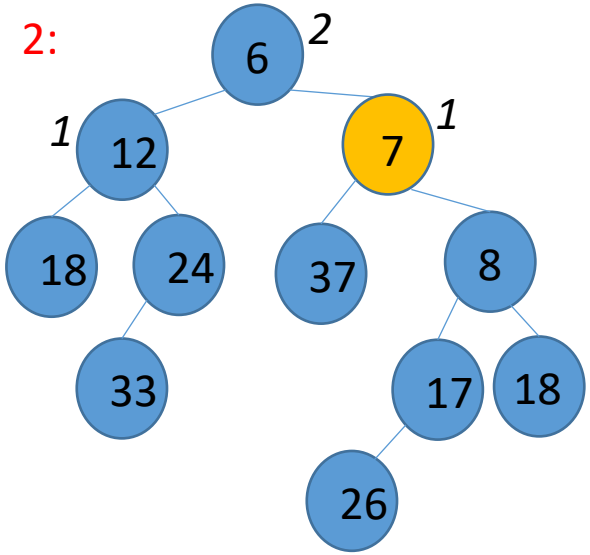


2. 递归：8和6合并

困难

- 需要根据子结点的npl来判断是否需要进行对换
- 需要重新计算npl的结点都在右路径上，但是需要根据子结点计算出父结点的npl





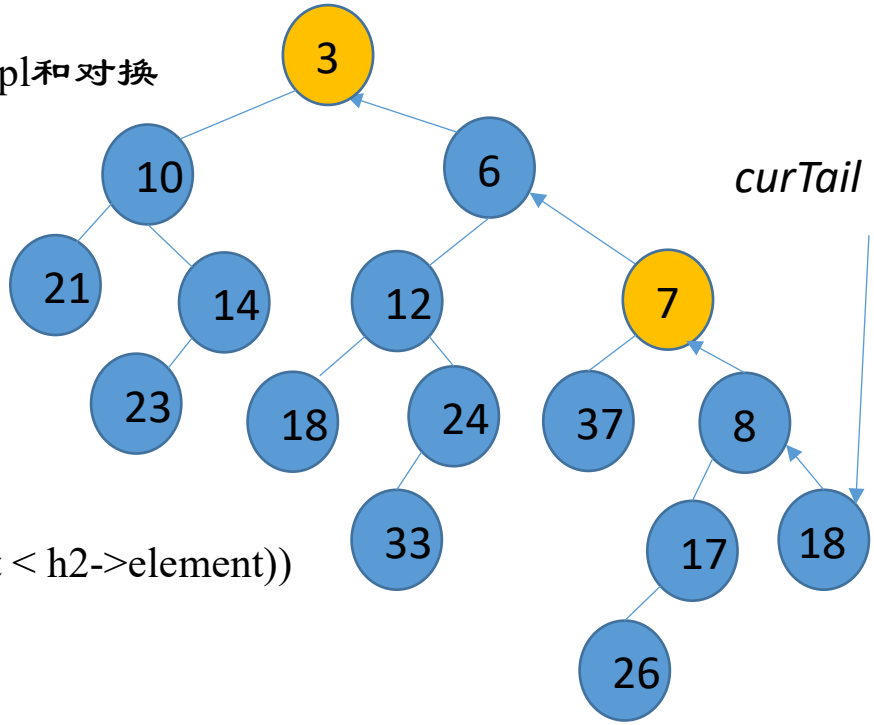
左式堆的迭代合并（1）

- 首先Merge两个左式堆的右路径
- 重新计算右路径上的各个结点的npl
  - 可以暂时使用right指针来计算父亲结点
- 对右路径上的结点，根据其子结点的npl决定是否对换

左式堆的迭代合并（2）

```
//合并右路径
//实际上是两个单链表的合并
//1、单链表的right对应于单链表的next
//2、合并得到的链表是逆向的：以方便重新计算npl和对换

Node * MergeRight(Node* h1, Node *h2)
{
      Node *curTail = null;
       while(h1 != null || h2 != null)
        {
             Node *next;
            if(h2 == null || (h1 != null && h1->element < h2->element))
                    next = h1;
             else
                    next = h2;
             next->right = curTail;
             curTail = next;
        }
        return curTail;
}
```



接下来需要：

1、从新计算右路径上的各个npl

2、翻转right指针，各个结点的right要指向它的前驱

3、必要时进行对换

左式堆的迭代合并（3）

```
ReverseAndSwap(Node *endOfRightPath)
{       Node * prev = null;
        Node * cur = endOfRightPath;
        while(cur != null)
        {       Node *next = cur->right;
                cur->right = prev
                if(NPL(cur->right) > NPL(cur->left))
                {   Node * tmp = cur->left;
                    cur->left = cur->right;
                    cur->right = tmp;
                }
                cur->npl = NPL(cur->right) + 1;
                prev = cur; cur = next;
        }
        return prev;
}
```

单次递归到迭代的转换

- 递归形式：

```
rec_func(args)
{
        if(!condition)
        {
                base_case_process;
                return;
        }
        process_before_recursion;
        rec_func(args1);
        process_after_recursion;
}
```

- 执行的过程：
  - 首先不断深入递归：执行process_before_recursion，然后递归调用；
  - 当最后一次递归返回后，不断执行process_after_recursion()；当然，这个时候执行顺序的参数和前面递归的时候相比是逆向的

单次递归到迭代的转换（二）

- 简化假设：rec_func中precess_before_recursion，process_after_recursion的处理仅仅和参数相关，那么使用一个存放参数的栈就可以将这个递归转换成为迭代处理的方法（如涉及局部变量，例如process_after_recursion使用了process_before_recursion中赋值的局部遍历，那么局部变量值也要入栈）
  - 递归调用前的部分转成一个循环代码，循环体包括process_before_recursion，以及将参数压栈的代码。转化方式类似于尾递归转成迭代。
  - 这个循环的后面，是对基础情况进行处理的代码，即base_case_process。
  - 接下来是第二个循环：不断出栈获取参数，并按照参数进行process_after_recursion处理

```
rec_func(args)
{
        if(!condition)
        {
                base_case_process;
                return;
        }
        process_before_recursion;
        rec_func(args1);
        process_after_recursion;
}
```

单次递归到迭代的转换（二）

进一步优化：

- 如果能够找到适当的方式保存栈中的内容，或者能够反推出递归调用的参数序列，那么上面的栈可以省略；
- 如果process_before_recursion很简单（甚至是空），并且参数序列也很简单，第一个循环也可以省略。

- 递归调用前的部分转成一个循环代码，循环体
- 将参数压栈
- 成迭代。
- 行处理的
- 取参数，
- sion处理

```
rec_func(args0)
{       args = args0;stack.push(args);
        while(condition)
        {
                process_before_recursion;
                {stack.push(args); args =  args1; }
        }
        base_case_process;
        while(!stack.isEmpty())
        {
                args = stack.pop();
                process_after_recursion;
        }
}
```

```
rec_func(args)
{
        if(!condition)
        {
                base_case_process;
                return;
        }
        process_before_recursion;
        rec_func(args1);
        process_after_recursion;
}
```

一个简单的例子

计算阶乘

```
int f(int N)
{       if(N == 0)
                return 1;
        else
                return N * f(N - 1);
}
```

```
f(int N)
{
        int ret;
        while(N != 0)
        {       stack.push(N); N = N -1;     }

        ret = 1;

        while(!stack.isEmpty())
        {
                N = stack.pop();
                ret = N *ret;
        }
        return ret;
}
```

因为栈中内容已知：就是从1到N，并且参数的计算过程显而易见，所以第一个循环中不需要入栈操作，可以省略；第二个循环也不需要从栈中获取参数，可以简化，得到：

```
f(int N, int *retAddr)
{
        ret = 1;
        for(int I = i; i <= N; i ++)
                ret = i * *ret;
        return ret;
}
```

就是最基本的求阶乘的方法。

另一个例子：并查集的cFind

递归：

```
int cFind(x)
{
        if(s[x] < 0)
                return x;
        root = find(s[x]);
        s[x] = root;
        return root;
}
```

另一个例子：并查集的cFind

```
int cFind(x)
{
    int ret;
    while(s[x] >= 0)
    {   stack.push(x); x = s[x];  }

    ret = x;

    while(!stack.isEmpty())
    {
        x = stack.pop();
        s[x] = ret;
    }
    return ret;
}
```

另一个例

```
int cFind(x)
{
    int ret;
    while(s[x] >=
    {   stack.pu

    ret = x;

    while(!stack.
    {
        x = stacl
        s[x] = re
    }
    return ret;
}
```

```
//1、第二个循环对栈中每个元素的处理效果和处理的顺序无关；
//2、栈中元素就是路径上的结点，所以不需要栈进行保存，重
//    复遍历该路径即可。但是需要注意遍历时对s[x]的赋值和
//    读取的顺序。
//因此可以简化得到如下的代码。
int cFind(x)
{
    int ret;
    while(s[x] >= 0)
    {   stack.push(x); x = s[x];  }

    ret = x;
    while(s[x] >= 0)
    {   tmp = s[x];
        s[x]  = root;
        x   =  tmp;
    }

    return ret;
}
```

左堆的合并的迭代化（1）

```
Merge(h1, h2)
{       if(h1 == NULL)
                return h2;
        else if(h2 == NULL)
                return h1;
        else if(h1.data <= h2.data)
        {       h1->right = Merge(h1->right, h2);
                h = h1;
        }
        else // h1.data > h2.data
        {       h2->right = Merge(h1, h2->right);
                h = h2;
        }

        if(NPL(h->left) < NPL(h->right))
        {       tmp = h->left; h->left = h->right; h->right = tmp;    }

        h->npl = NPL(h->right) + 1;
        return h;
}
```

复杂一点的原因：

1、有两处递归调用，虽然仍然是单次递归调用；

2、在不同递归调用分支中，有不同的递归调用后处理程序。

左堆的合并的迭代化（2）

```
Merge(h1, h2)
{
    while(h1 != NULL && h2 != NULL)
    {
        if(h1.data <= h2.data)
        {       stack.push(h1, h2);     h1 = h1->right;
        else
        {       stack.push(h1, h2);     h2 = h2->right;
    }

    ret = (h1 == NULL) ?  h2 : h1;                  //base case的返回值

    while(!stack.isEmpty())
    {       (h1, h2) = stack.pop();                 //恢复栈中参数

        h = (h1.data <= h2.data) ? h1 : h2;         //对应于不同递归调用分支的后处理过程。
        h->right = ret;

        if(NPL(h->left) < NPL(h->right))
        {       tmp = h->left; h->left = h->right; h->right = tmp;      }

        h->npl = NPL(h->right) + 1;

        ret = h;
    }

    return ret;
}
```

这里虽然两个参数都入栈，但是在第二个循环中只需要用到其中的一个值：

(h1.data <= h2.data) h1 : h2

因此，我们可以考虑简化栈，得到下一页的代码

左堆的合并的迭代化（3）

```
Merge(h1, h2)
{
    while(h1 != NULL && h2 != NULL)
    {
        if(h1.data <= h2.data)
        {       stack.push(h1);         h1 = h1->right;         }
        else
        {       stack.push(h2);         h2 = h2->right;         }
    }

    ret = (h1 == NULL) ?  h2 : h1;    //base case的返回值

    while(!stack.isEmpty())
    {       h = stack.pop();                        //恢复栈中的值

        h->right = ret;

        if(NPL(h->left) < NPL(h->right))
        {   tmp = h->left; h->left = h->right; h->right = tmp;    }

        h->npl = NPL(h->right) + 1;

        ret = h;
    }

    return ret;
}
```

左堆的合并的迭代化（3）

```
Merge(h1, h2)
{
    while(h1 != NULL && h2 != NULL)
    {
        if(h1.data <= h2.data)
        {       stack.push(h1);         h1 = h1->right;
        else
        {       stack.push(h2);         h2 = h2->right;
    }

    ret = (h1 == NULL) ?  h2 : h1;    //base case的返回值

    while(!stack.isEmpty())
    {       h = stack.pop();                          //恢复栈中的值

        h->right = ret;

        if(NPL(h->left) < NPL(h->right))
        {   tmp = h->left; h->left = h->right; h->right = tmp;

        h->npl = NPL(h->right) + 1;

        ret = h;
    }

    return ret;
}
```

如果我们能够把这里的栈stack用一个紧凑的方式表示处理，可以省略独立的栈。

注意：在第一个循环的两个分支处理中，

h1 = h1->right

或

h2 = h2->right

之后，相应的right字段中的值实际上已经不会被引用了，我们可以把它复用为链表栈的next字段

注意，这个在一般编程时是很差的做法，程序变得难以理解。但是在算法中能省一点是一点，可以省掉一些额外的内存需求

左堆的合并的迭代化（4）

```
Merge(h1, h2)
{       stackTop = NULL;                        //设置栈顶指针，初始值为NULL
        while(h1 != NULL && h2 != NULL)
        {
                if(h1.data <= h2.data)          //下面的红色部分为将tmp压栈，复用了right字段
                {               //stack.push(h1);h1 = h1->right;
                                tmp = h1; h1 = h1->right;       tmp->right = stackTop; stackTop = tmp;    }
                else
                {               //stack.push(h2);h2 = h2->right;
                                tmp = h2; h2 = h2->right;       tmp->right = stackTop; stackTop = tmp;    }
        }

        ret = (h1 == NULL) ?  h2 : h1;                          //base case的返回值

        while(stackTop != NULL)                                                 //(!stack.isEmpty())
        {
                h = stackTop; stackTop = stackTop->right;    //h = stack.pop(); 出栈

                h->right = ret; //这里的right重新变成指向右子节点的指针

                if(NPL(h->left) < NPL(h->right))
                {     tmp = h->left; h->left = h->right; h->right = tmp;    }

                h->npl = NPL(h->right) + 1;

                ret = h;
        }
        return ret;
}
```