

# **Leftist Heaps and Skew Heaps**

# Leftist Heaps



Target: Speed up merging in  $O(N)$ .

🌀 **Heap: Structure Property + Order Property**

**Discussion 5:** How fast can we merge two heaps if we simply use the original heap structure?

## Leftist Heap:

Order Property – the same

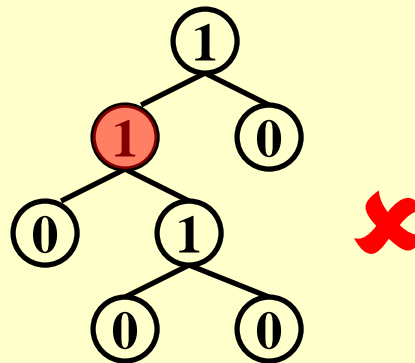
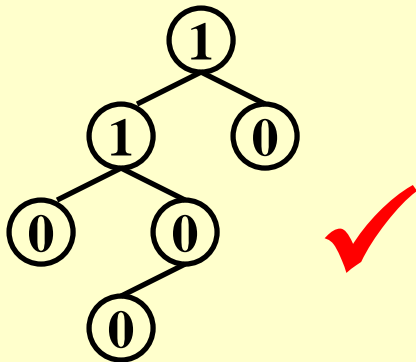
Structure Property – binary tree, but *unbalanced*

【Definition】 The **null path length**,  $Npl(X)$ , of any node  $X$  is the length of the shortest path from  $X$  to a node without two children. Define  $Npl(NULL) = -1$ .

**Note:**

$$Npl(X) = \min \{ Npl(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】 The **leftist heap property** is that for every node  $X$  in the heap, the null path length of the **left** child is **at least as large as** that of the **right** child.



The tree is biased to get deep toward the *left*.

**【Theorem】** A leftist tree with  $r$  nodes on the right path must have at least  $2^r - 1$  nodes.

**Proof:** By induction on p. 162.

**Discussion 6:** How long is the right path of a leftist tree of  $N$  nodes? What does this conclusion mean to us?

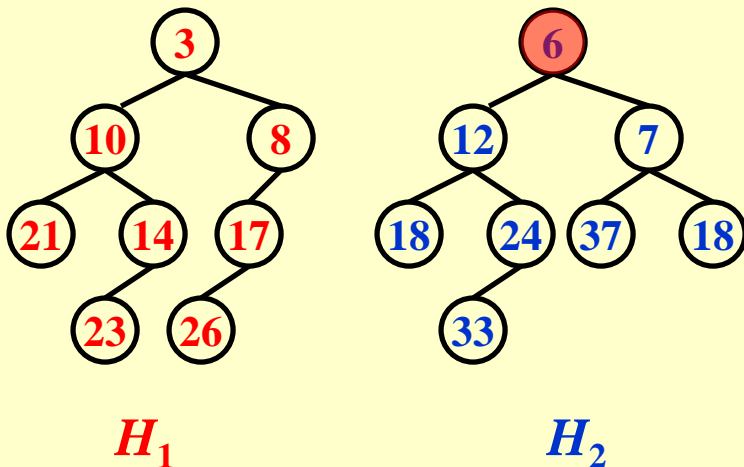
**Trouble makers:** Insert and Merge

**Note:** Insertion is merely a special case of merging.

## Declaration:

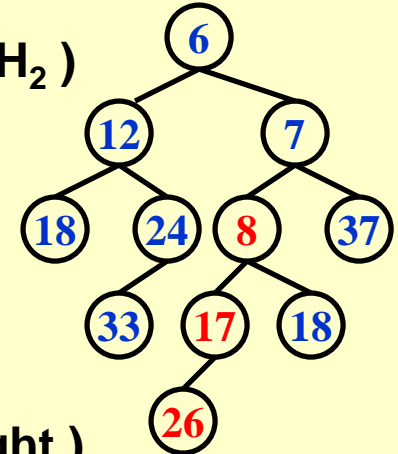
```
struct TreeNode
{
    ElementType    Element;
    PriorityQueue  Left;
    PriorityQueue  Right;
    int           Npl;
};
```

## Merge (recursive version):



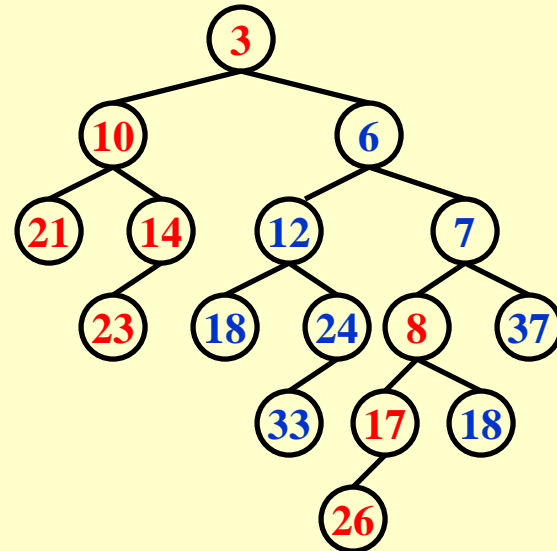
### Step 1:

Merge(  $H_1 \rightarrow \text{Right}$ ,  $H_2$  )



### Step 2:

Attach(  $H_2$ ,  $H_1 \rightarrow \text{Right}$  )



### Step 3:

Swap( $H_1 \rightarrow \text{Right}$ ,  $H_1 \rightarrow \text{Left}$  )  
if necessary

```

PriorityQueue Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL ) return H2;
    if ( H2 == NULL ) return H1;
    if ( H1->Element < H2->Element ) return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}

```

```

static PriorityQueue
Merge1( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1->Left == NULL )    /* single node */
        H1->Left = H2;        /* H1->Right is already NULL
                               and H1->Npl is already 0 */

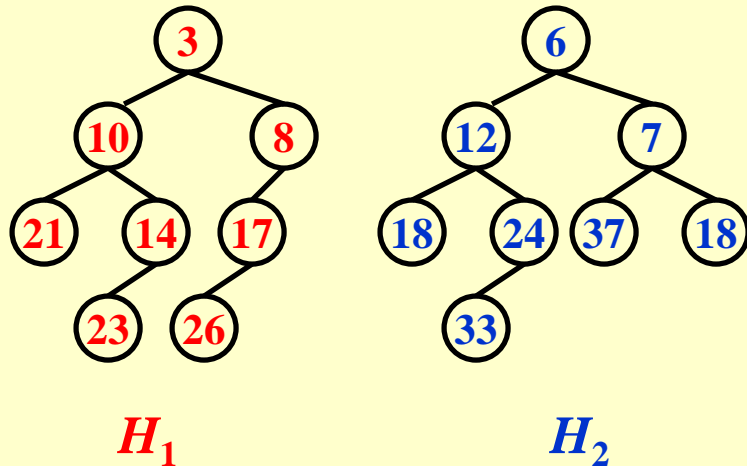
    else {
        H1->Right = Merge( H1->Right, H2 );    /* Step 1 & 2 */
        if ( H1->Left->Npl < H1->Right->Npl )
            SwapChildren( H1 );                /* Step 3 */
        H1->Npl = H1->Right->Npl + 1;
    } /* end else */
    return H1;
}

```

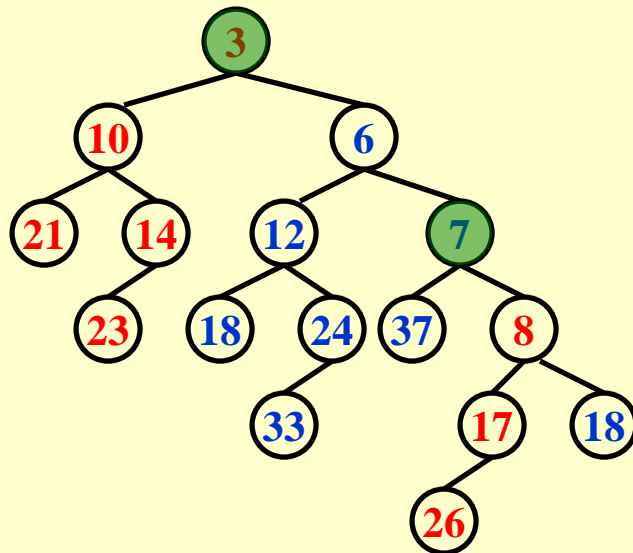
$T_p = O(\log N)$

What if  $Npl$  is NOT updated?

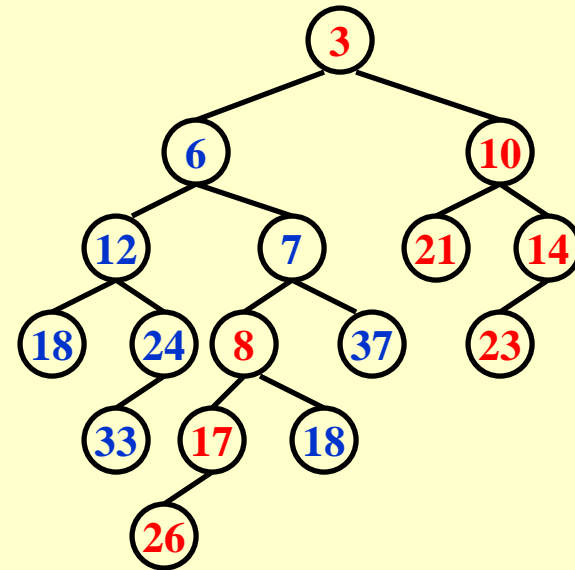
## ☞ Merge (iterative version):



**Step 1:** Sort the right paths without changing their left children



**Step 2:** Swap children if necessary



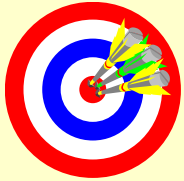
## ☞ DeleteMin:

**Step 1:** Delete the root

**Step 2:** Merge the two subtrees

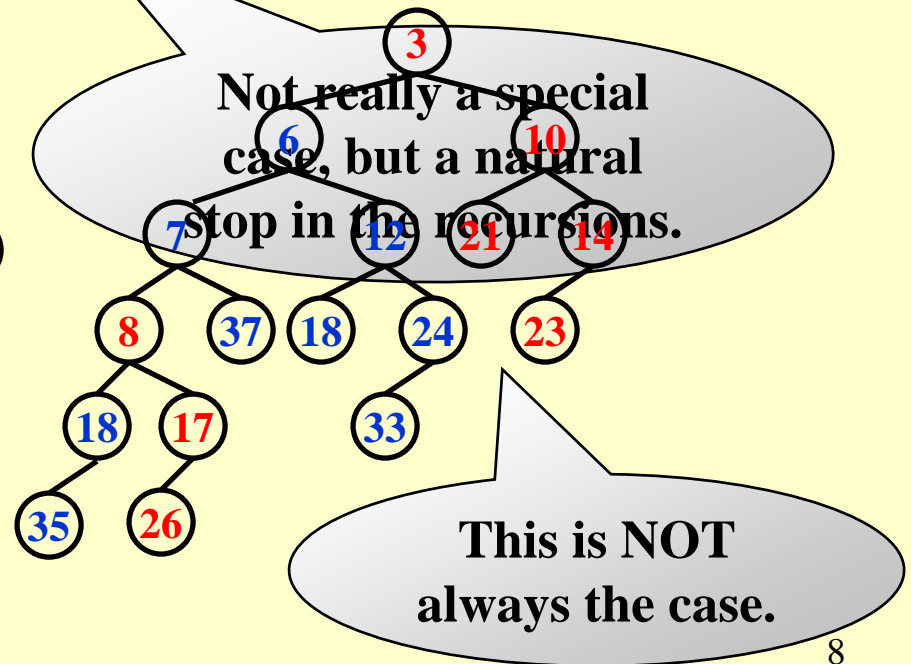
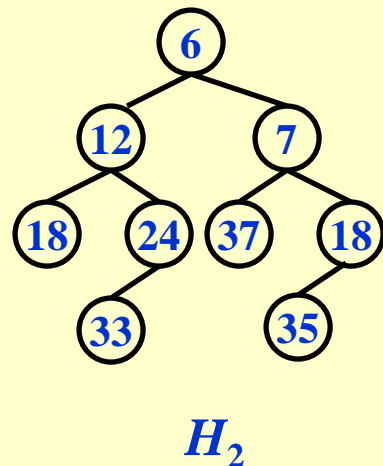
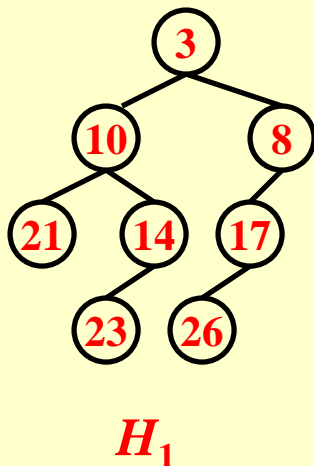
$$T_p = O(\log N)$$

# Skew Heaps -- a simple version of the leftist heaps



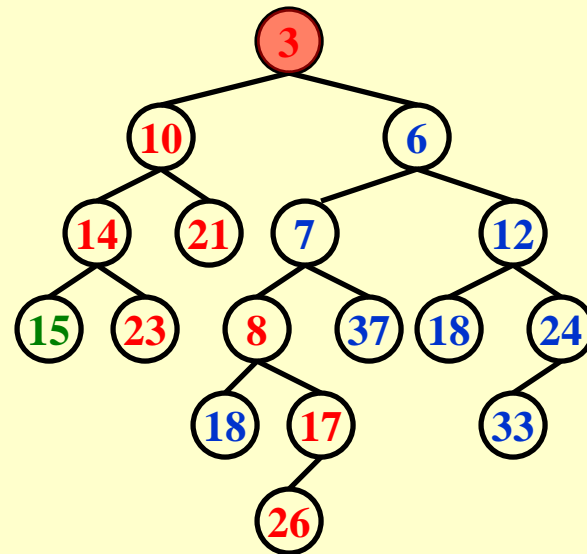
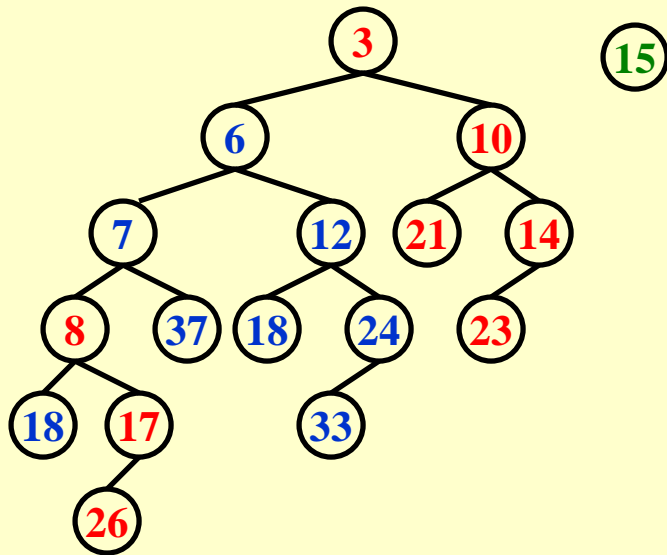
Target: Any  $M$  consecutive operations take at most  $O(M \log N)$  time.

👉 Merge: **Always** swap the left and right children except that the **largest** of all the nodes on the right paths does not have its children swapped. **No Npl.**

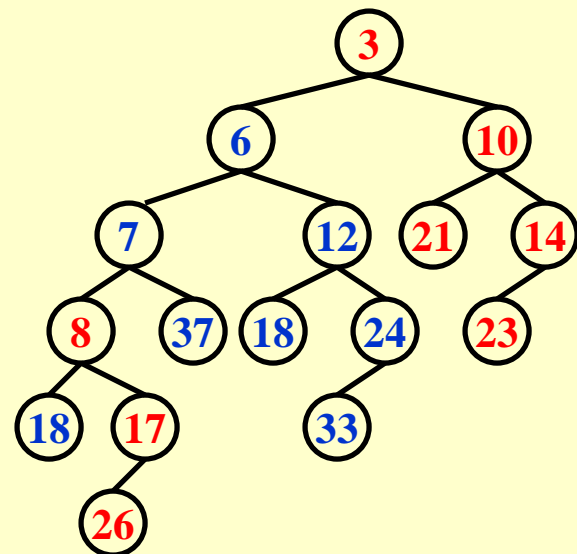
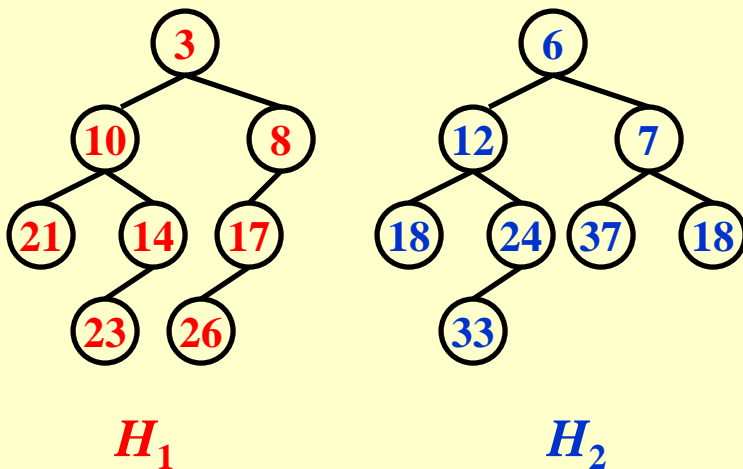




# 【Example】 Insert 15



👉 Merge (iterative version):



### Note:

- ☞ Skew heaps have the advantage that **no extra space** is required to maintain path lengths and **no tests** are required to determine when to swap children.
- ☞ It is an open problem to determine precisely the **expected right path length** of both leftist and skew heaps.

# Amortized Analysis for Skew Heaps

Insert & Delete are just **Merge**

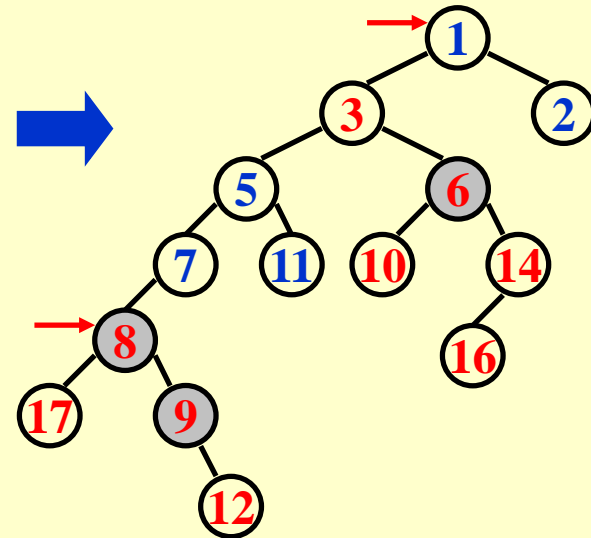
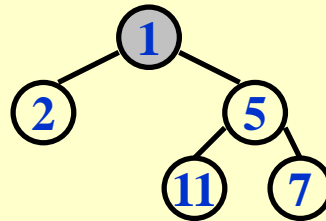
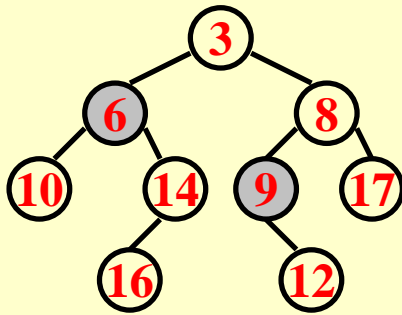
$$T_{amortized} = O(\log N) ?$$

$D_i$  = the root of the resulting tree



$\Phi(D_i)$  = number of *heavy* nodes

**【Definition】** A node  $p$  is *heavy* if the number of descendants of  $p$ 's right subtree is at least half of the number of descendants of  $p$ , and *light* otherwise. Note that the number of descendants of a node includes the node itself.



The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$$H_i : l_i + h_i \quad (i = 1, 2) \quad \longrightarrow \quad T_{\text{worst}} = l_1 + h_1 + l_2 + h_2$$

Along the right path

$$\text{Before merge: } \Phi_0 = h_1 + h_2 + h \quad T_{\text{amortized}} = T_{\text{worst}} + \Phi_N - \Phi_0$$

$$\text{After merge: } \Phi_N \leq l_1 + l_2 + h \leq 2(l_1 + l_2)$$

$$l = O(\log N) \quad \longrightarrow \quad T_{\text{amortized}} = O(\log N)$$

## Reference:

**Data Structure and Algorithm Analysis in C (2<sup>nd</sup> Edition):**  
**Ch.5, p.161-169; Ch.11, p.435-437; *M.A.Weiss* 著、**  
**陈越改编, 人民邮电出版社, 2005**