Data Structures and Algorithms

- Heap -

School of Electrical Engineering
Korea University

Heaps(Priority Queues)

- FIFO Queue 에서의 문제점
  - Shortest job first 정책필요
  - incoming job에 대해 동적(dynamic) 재구성 필요

-> priority queue

- Implementation – list, array, binary tree

Model

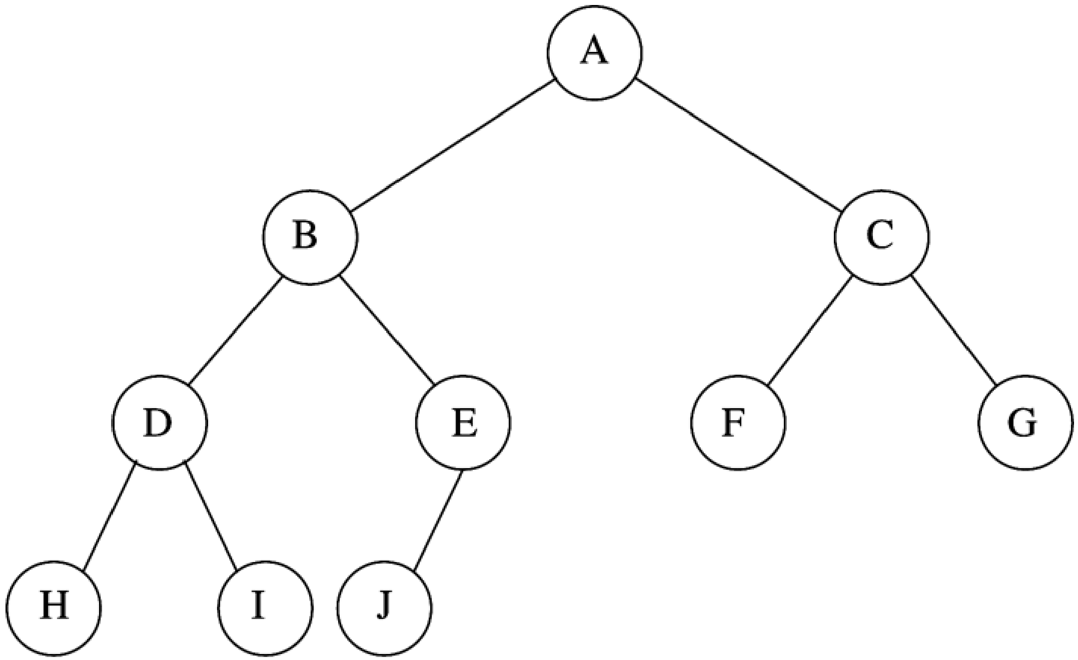
- Allows at least following two operations:
  - *Insert*
  - *DeleteMin*

- *DeleteMin* returns and removes the minimum element in the heap

What is a *min*-heap?

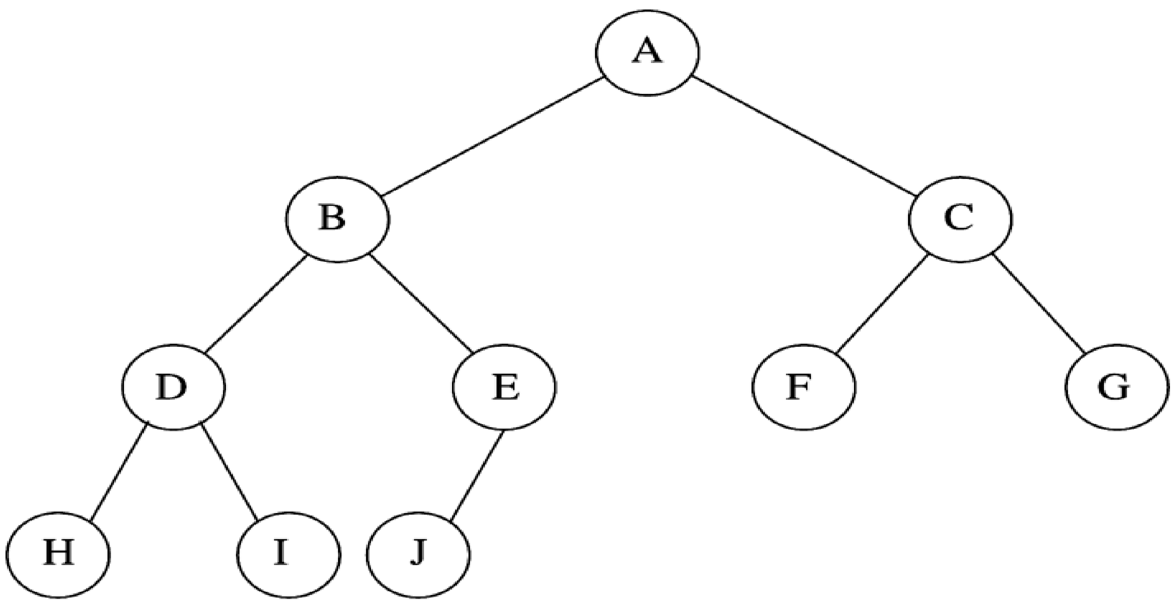
- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  1) Heap order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  2) Complete binary tree: let h be the height of the heap, then
     - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
     - at depth h, the nodes are filled from left to right

Heap: complete binary tree



- A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes

Array implementation



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Min heap

- *heap order* property:

$$ord(\text{parent}_\text{i}) < ord(\text{left\_child\_of\_parent}_\text{i}),$$

$$ord(\text{parent}_\text{i}) < ord(\text{right\_child\_of\_parent}_\text{i}),$$

for all nodes in the tree.

- tree implementation on array *a[1:N]*

$a[2i]$: left child of $a[i]$

$a[2i+1]$: right child of $a[i]$

Declaration for priority queue

```
        PriorityQueue
        Initialize( int MaxElements )
        {
            PriorityQueue H;

/* 1*/      if( MaxElements < MinPQSize )
/* 2*/          Error( "Priority queue size is too small" );

/* 3*/      H = malloc( sizeof( struct HeapStruct ) );
/* 4*/      if( H == NULL )
/* 5*/          FatalError( "Out of space!!!" );

            /* Allocate the array plus one extra for sentinel */
/* 6*/      H->Elements = malloc( ( MaxElements + 1 )
                                    * sizeof( ElementType ) );
/* 7*/      if( H->Elements == NULL )
/* 8*/          FatalError( "Out of space!!!" );

/* 9*/      H->Capacity = MaxElements;
/*10*/      H->Size = 0;
/*11*/      H->Elements[ 0 ] = MinData;

/*12*/      return H;
        }
```

Heap Order Property

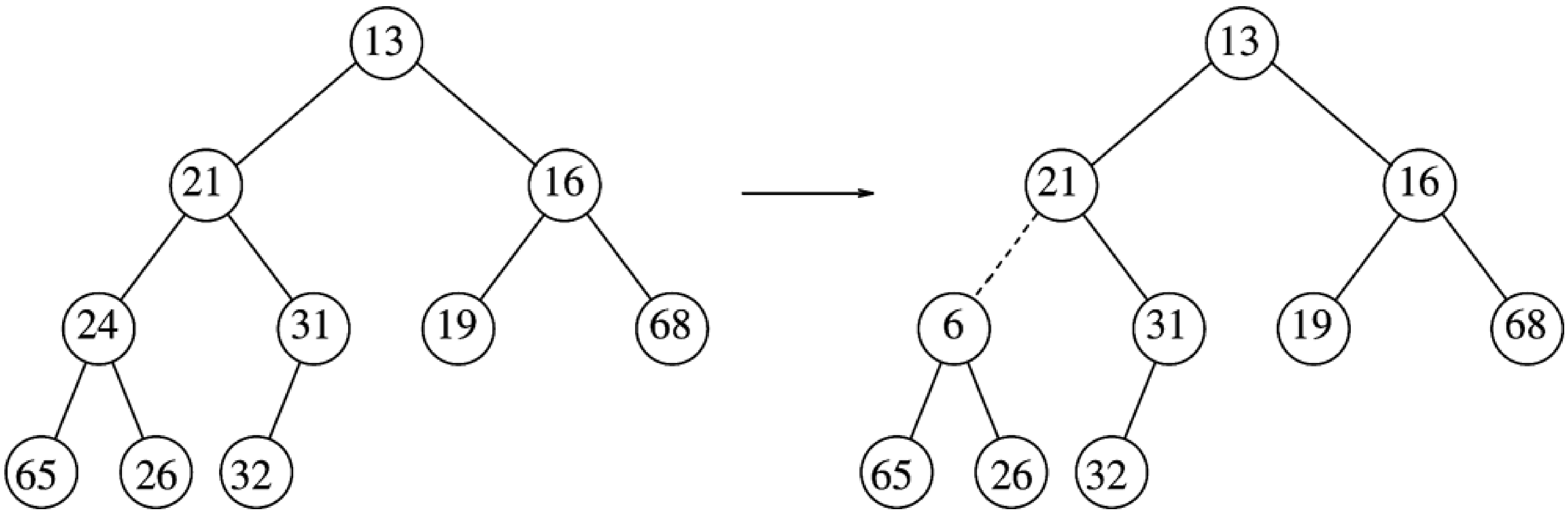


Figure 6.5 Two complete trees(only the left tree is a heap)

Array Implementation

- We can represent a heap with n keys by means of a vector of length $n+1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i+1$
- Links between nodes are not explicitly stored
- The cell of at rank $\mathbf{0}$ is not used
- ***Insert*** corresponds to inserting at rank $n+1$
- ***DeleteMin*** corresponds to removing at rank *1*

Height of a Heap

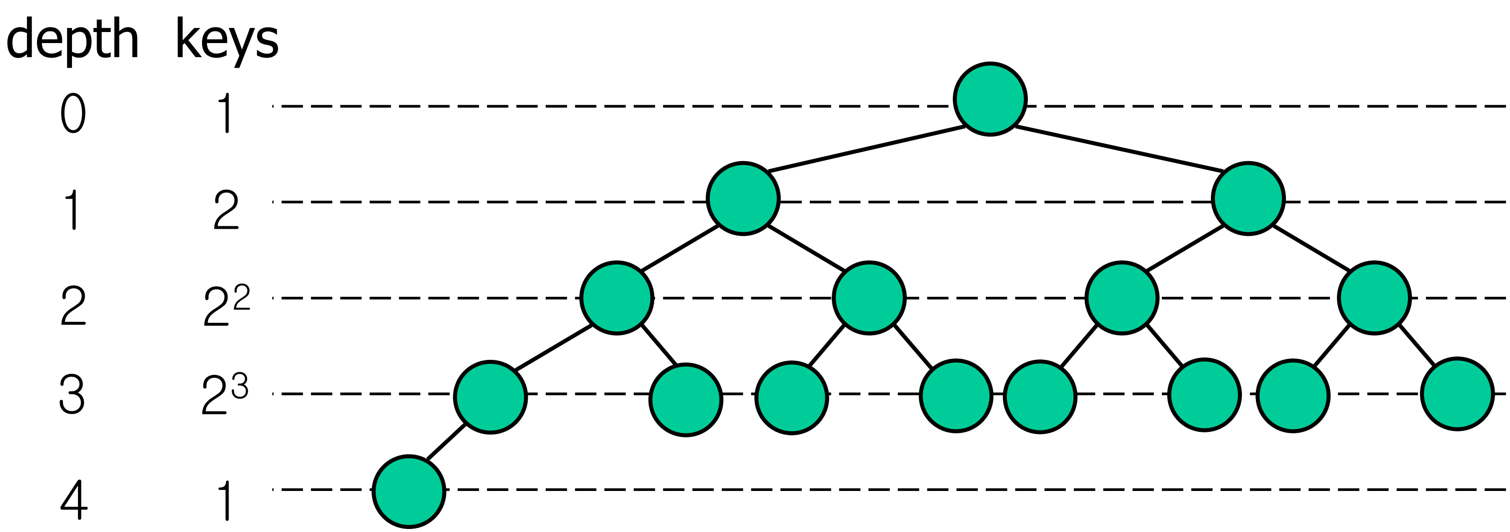
Theorem: A heap storing n keys has height $O(\log n)$

Proof: (we apply the complete binary tree property)

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$

Height of a Heap

- Since there are $\mathbf{2}^{i}$ keys at depth $\boldsymbol{i} = \mathbf{0, \ldots, } \boldsymbol{h} - \mathbf{1}$ and at least one key at depth $\boldsymbol{h}$, we have $\boldsymbol{n} \geq \mathbf{1 + 2 + 4 + \ldots + 2}^{\boldsymbol{h}-\mathbf{1}} \mathbf{+ 1}$
- Thus, $\boldsymbol{n} \geq \mathbf{2}^{\boldsymbol{h}}$, i.e., $\boldsymbol{h} \leq \mathbf{log}\ \boldsymbol{n}$



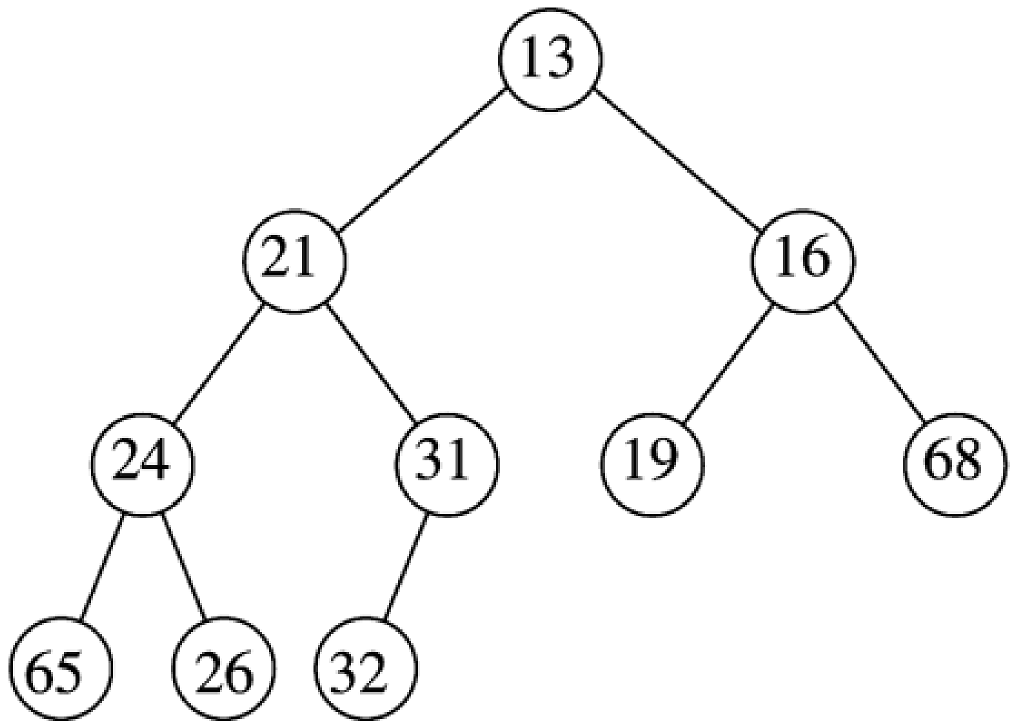
Heap operations

- ***Insert*** (up-heap) – insert a new key with a hole at the last location of the heap: $O(\log N)$
- ***DeleteMin*** (down-heap) – min. of the heap is replaced with the last element of heap, then heapify it: $O(\log N)$
- ***Delete*** – remove a specified node, then heapify it: $O(\log N)$
- ***BuildHeap*** – Create a heap with N input keys $O(N \log N)$

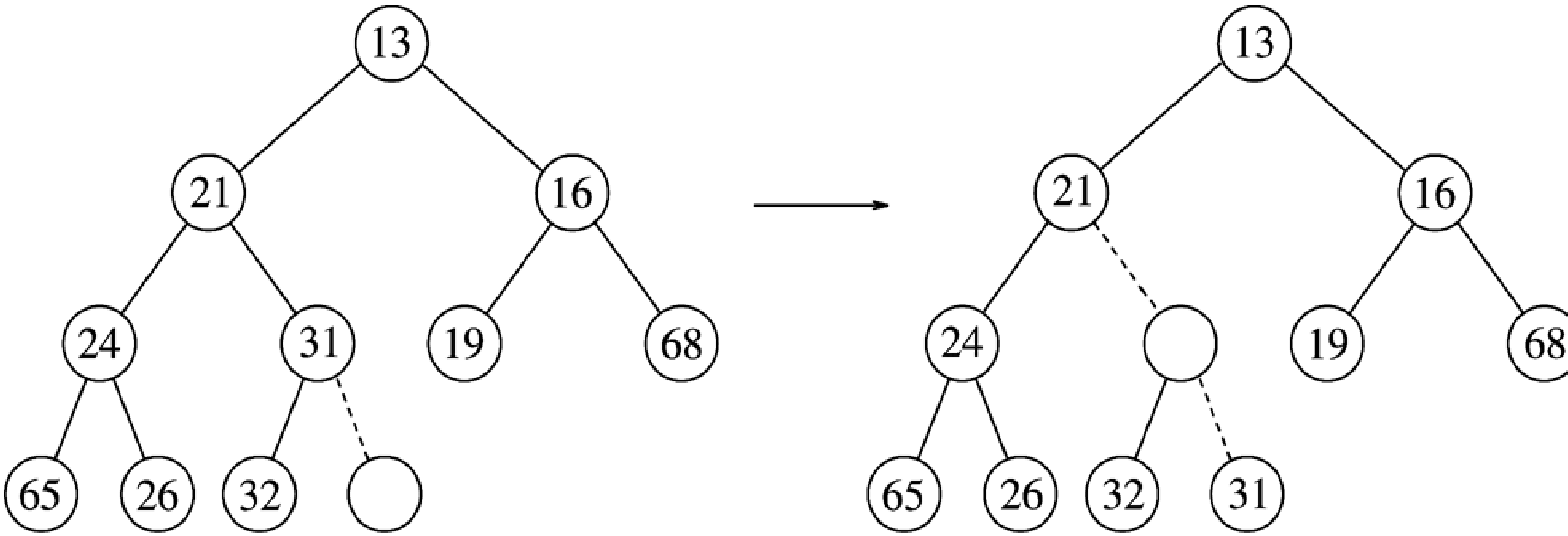
Insertion into a Heap

- ***Insert*** operation of the priority queue ADT corresponds to the insertion of a key $\boldsymbol{k}$ to the heap
- The insertion algorithm consists of the following steps
  - Find the insertion node z (the new last node)
  - Restore the heap-order property (known as ***percolate up***)

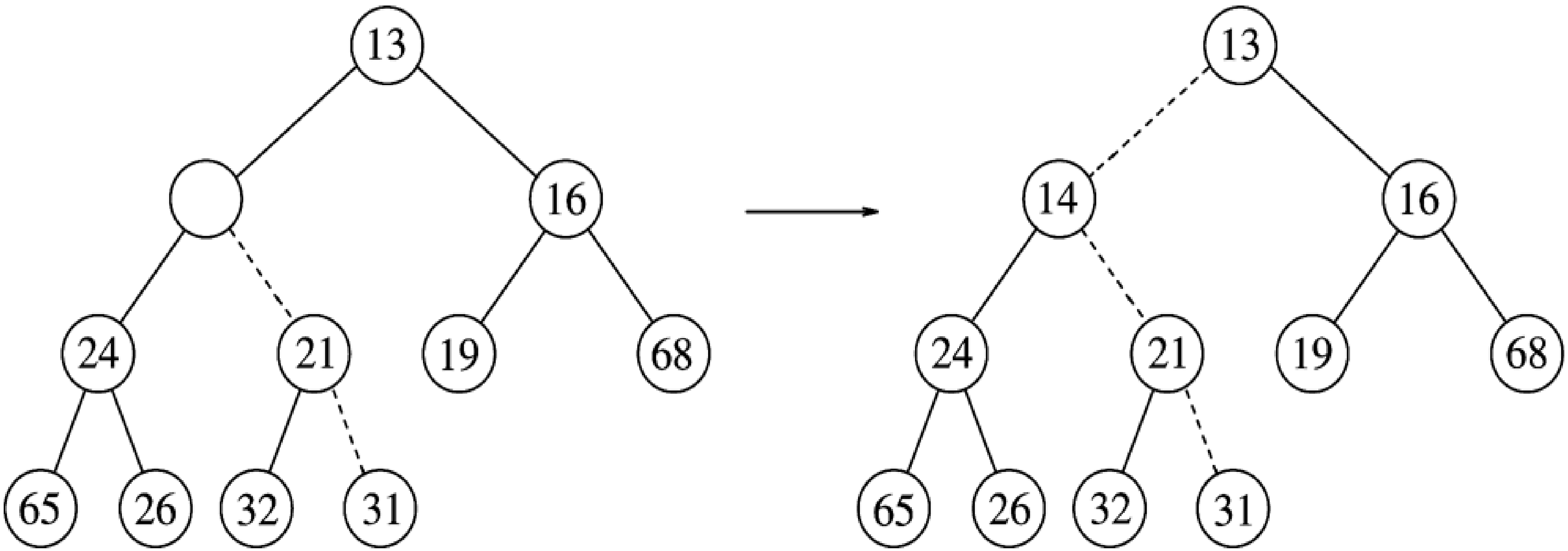
Example: Insert 14



Example: Insert 14



Example: Insert 14



Insert Algorithm

```
/* H->Element[ 0 ] is a sentinel */

void
Insert( ElementType X, PriorityQueue H )
{
    int i;

    if( IsFull( H ) )
    {
        Error( "Priority queue is full" );
        return;
    }

    for( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
        H->Elements[ i ] = H->Elements[ i / 2 ];
    H->Elements[ i ] = X;
}
```

DeleteMin

- DeleteMin operation of the priority queue ADT corresponds to the removal of the root key from the heap

- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node ***w***
  - Restore the heap-order property (known as ***percolate down***)

Example: DeleteMin

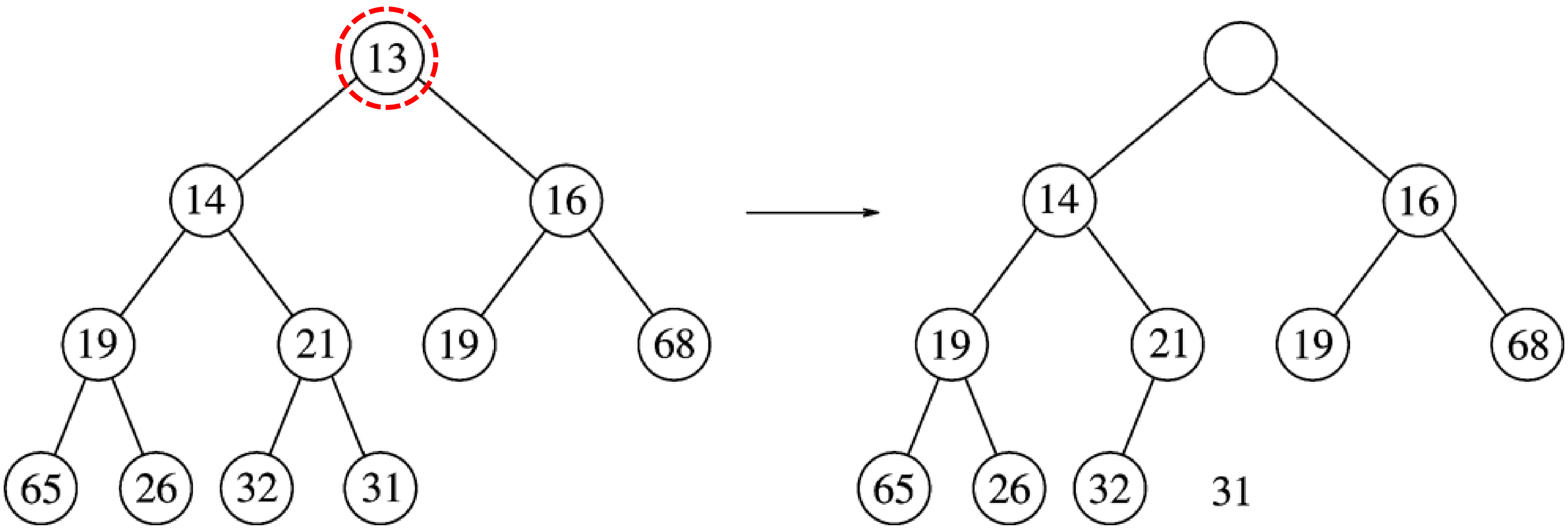


Figure 6.9 Creation of the hole at the root

Example: DeleteMin

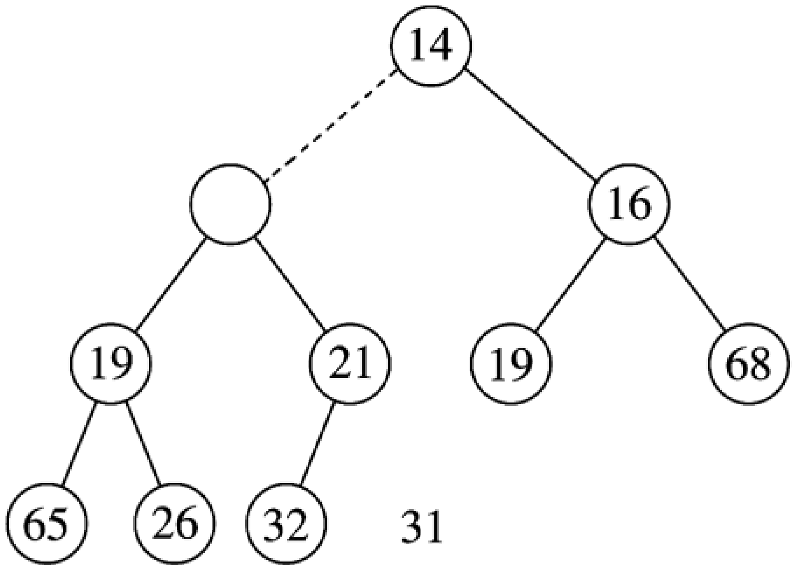


Figure 6.10 Next two step in DeleteMin

Example: DeleteMin

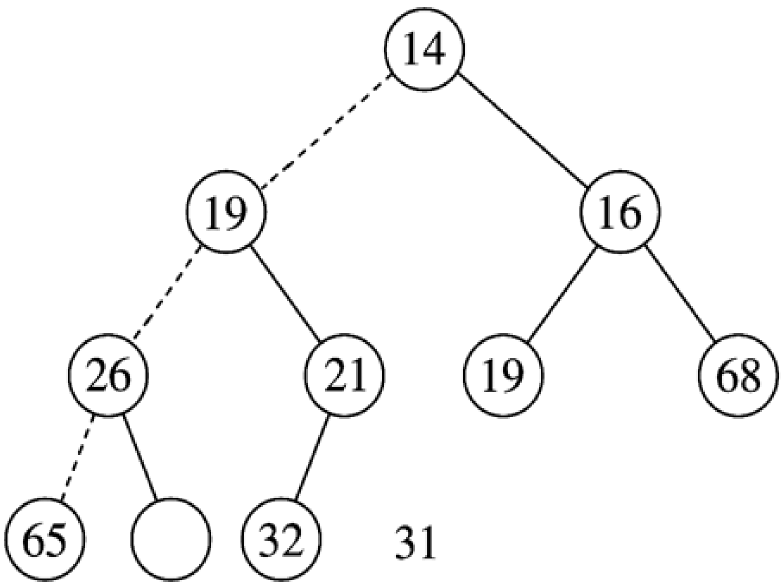


Figure 6.11 Last two steps in DeleteMin

```
        ElementType
        DeleteMin( PriorityQueue H )
        {
            int i, Child;
            ElementType MinElement, LastElement;

/* 1*/      if( IsEmpty( H ) )
            {
/* 2*/          Error( "Priority queue is empty" );
/* 3*/          return H->Elements[ 0 ];
            }
/* 4*/      MinElement = H->Elements[ 1 ];
/* 5*/      LastElement = H->Elements[ H->Size-- ];

/* 6*/      for( i = 1; i * 2 <= H->Size; i = Child )
            {
                /* Find smaller child */
/* 7*/          Child = i * 2;
/* 8*/          if( Child != H->Size && H->Elements[ Child + 1 ]
/* 9*/                                < H->Elements[ Child ] )
/*10*/              Child++;

                /* Percolate one level */
/*11*/          if( LastElement > H->Elements[ Child ] )
/*12*/              H->Elements[ i ] = H->Elements[ Child ];
                else
/*13*/              break;
            }
/*14*/      H->Elements[ i ] = LastElement;
/*15*/      return MinElement;
        }
```

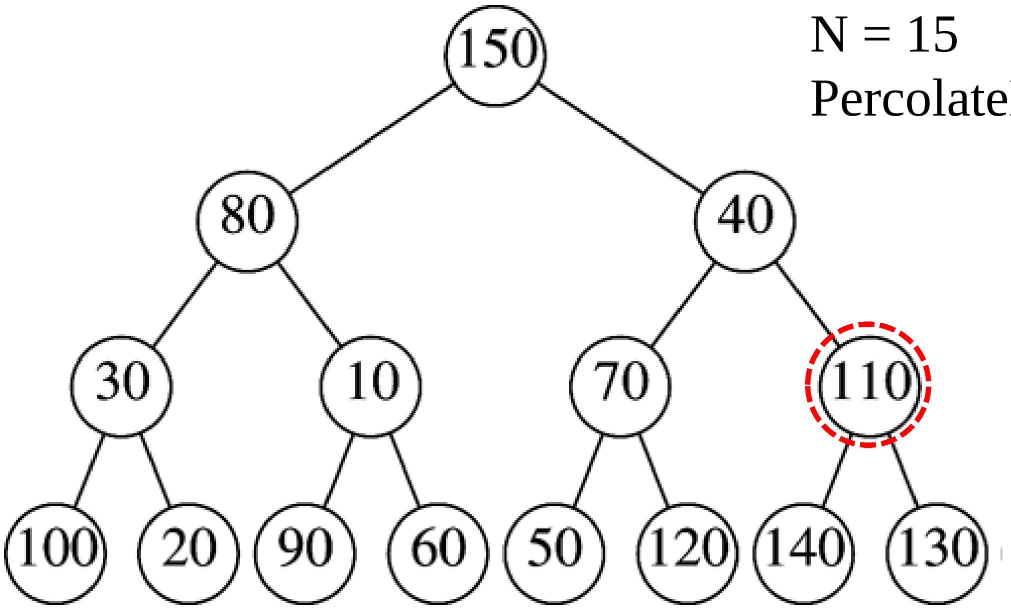
BuildHeap operation

- Takes as input *N* keys and places them into an empty heap
- Can be done with *N* successive Insert operations, which takes $O(N * \log N)$ worst time.
- A solution is to place the *N* keys into the tree in any order, maintaining the structure property.
- Create a heap-ordered tree using the following algorithm

BuildHeap operation

Figure 6.14 Sketch of *BuildHeap*

```
for( i = N / 2; i > 0; i-- )
    PercolateDown( i );
```



N = 15
PercolateDown(7)

BuildHeap operation

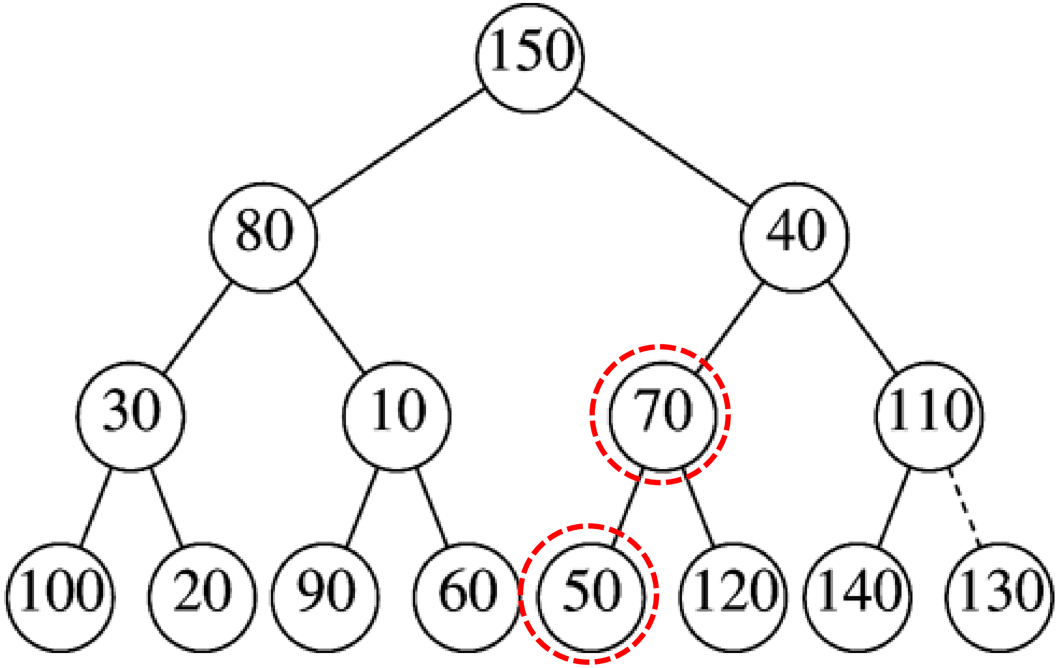


Figure 6.15 After PercolateDown(7)

BuildHeap operation

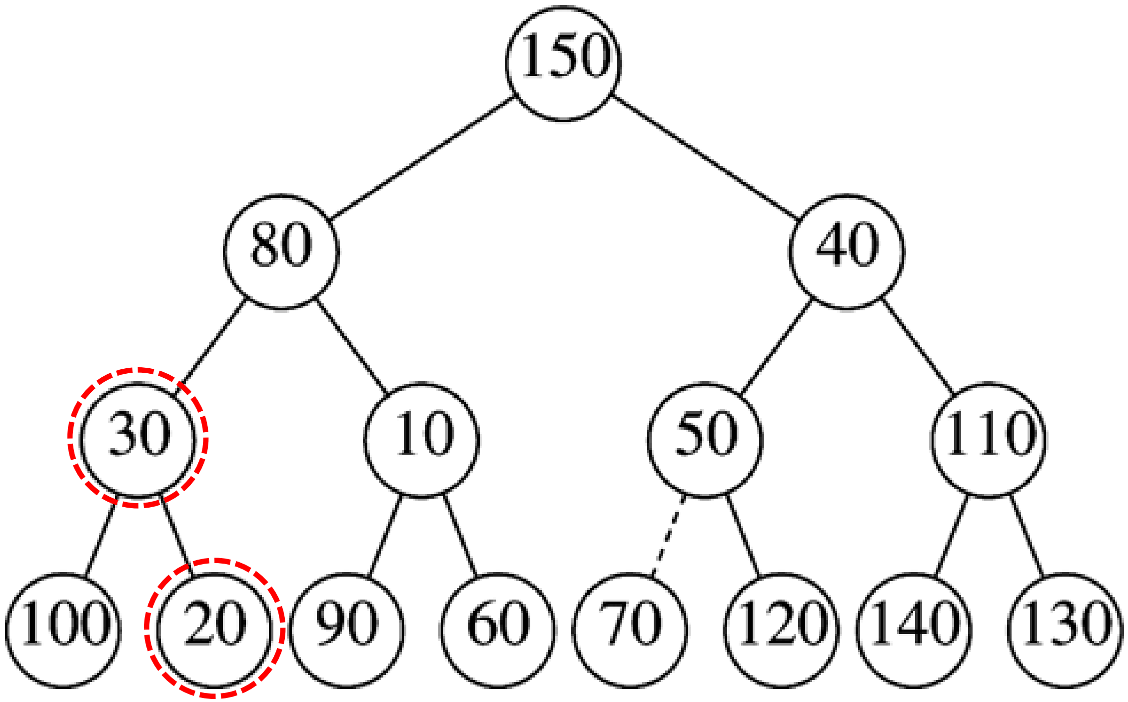


Figure 6.16 After PercolateDown(6)

BuildHeap operation

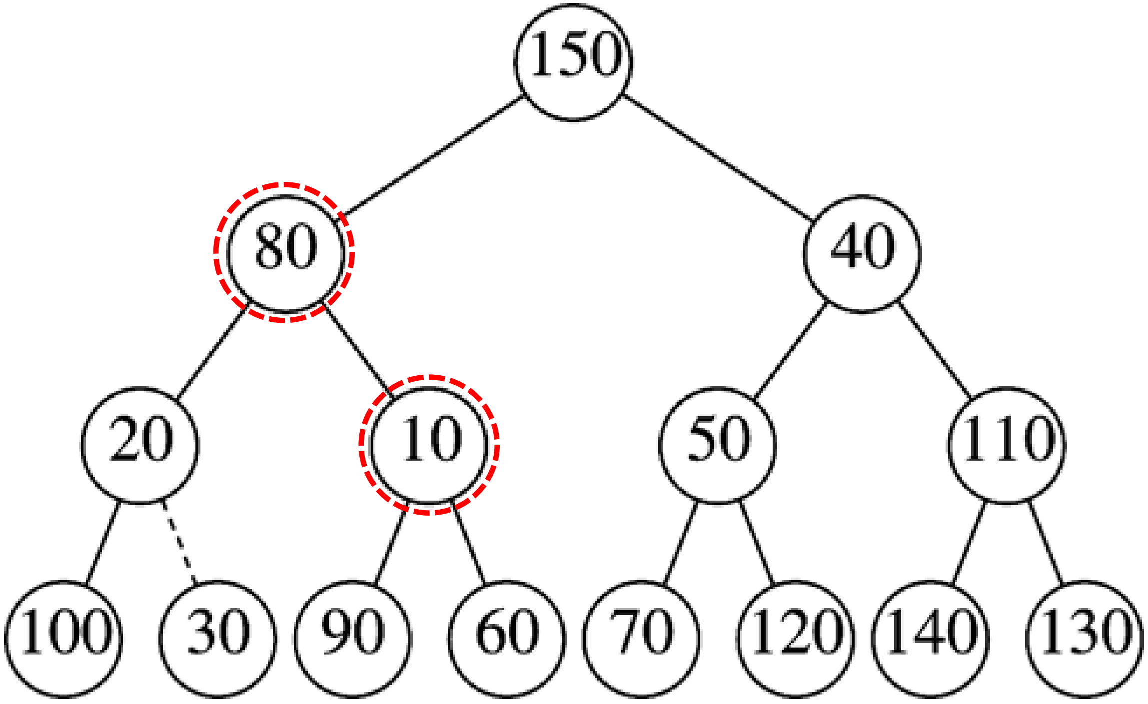


Figure 6.17 After PercolateDown(4)

BuildHeap operation

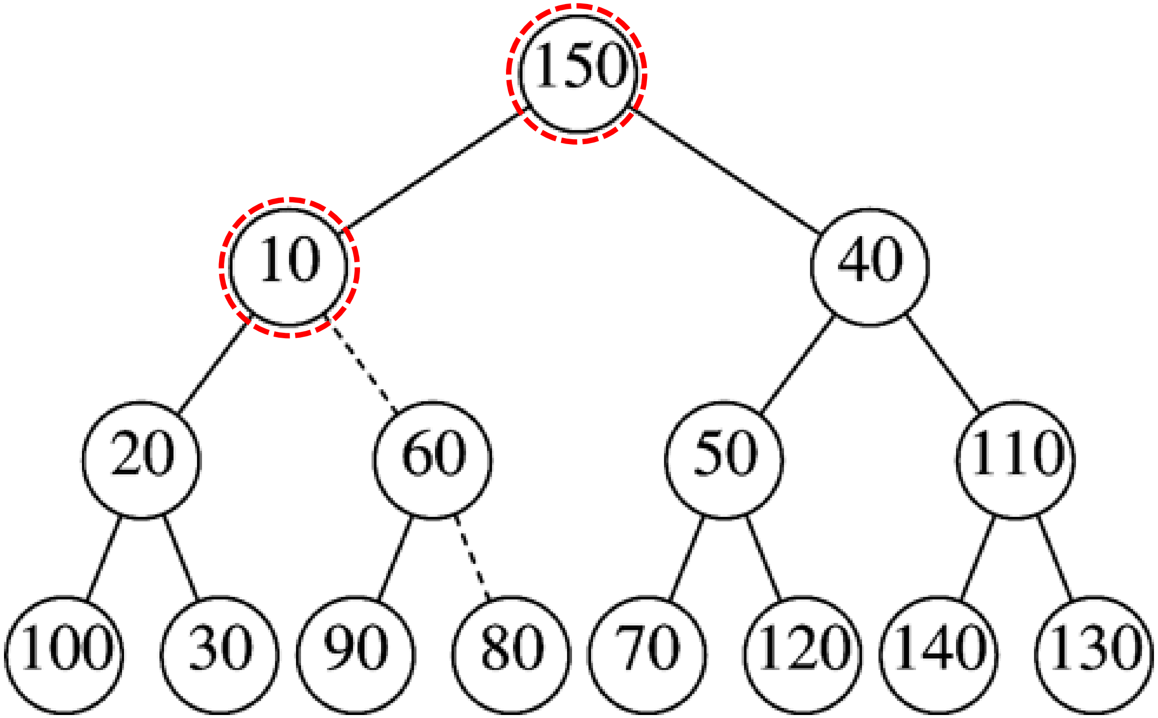


Figure 6.18 After PercolateDown(2)

BuildHeap operation

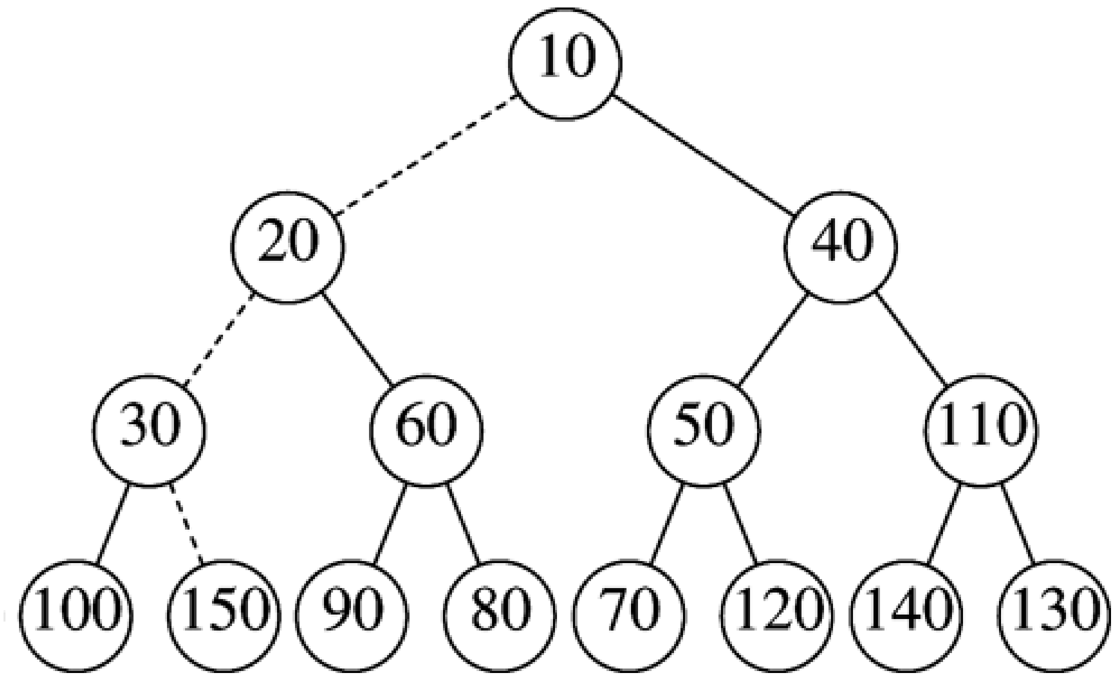
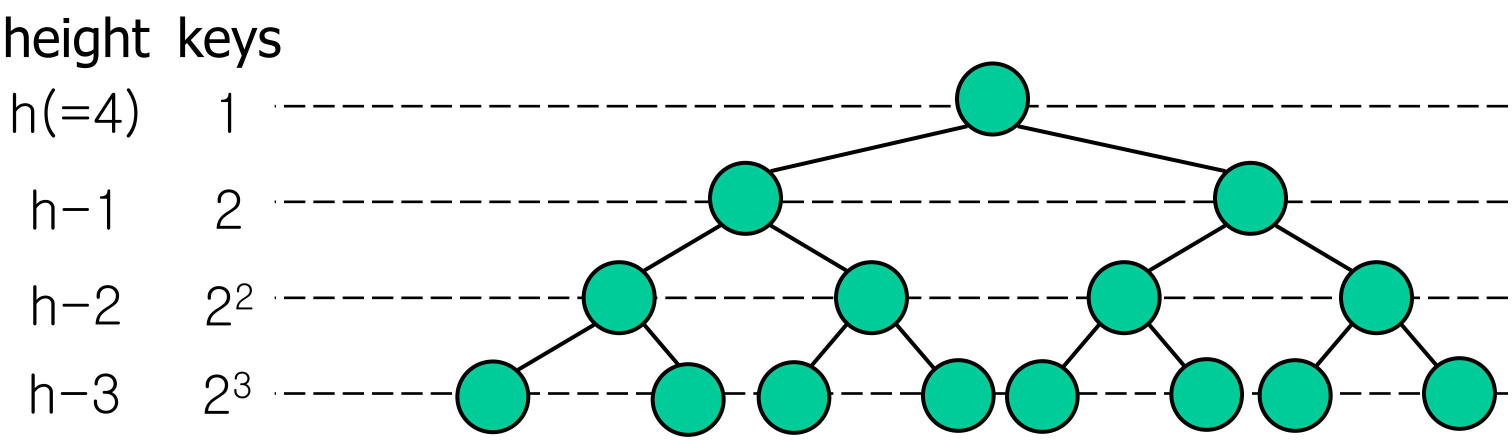


Figure 6.18 after PercolateDown(1)

Theorem

For the perfect binary tree of height h containing $2^{h+1} - 1$ *nodes, the sum of the heights of the nodes is* $2^{h+1} - 1 - (h + 1)$



d-Heaps

- Like a binary heap except that all nodes have d children

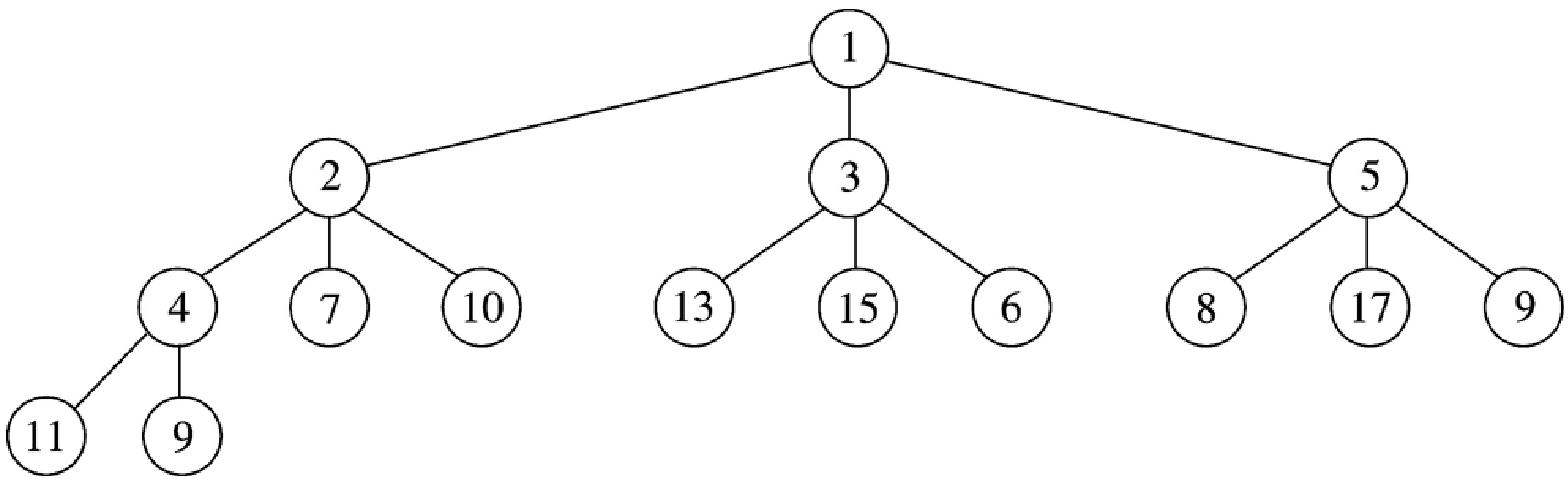


Figure 6.19 A sample 3-heap

Properties of d-Heaps

- Much shallower than a binary heap
- Running time of *Inserts* is $O(\log_d N)$
- For *DeleteMin*, the minimum of $\boldsymbol{d}$ children must be found
  - Could be expensive for large $\boldsymbol{d}$
  - Requires $\boldsymbol{d}-1$ comparisons
  - $O(\boldsymbol{d} * \log_d N)$
- When implemented on an array, $\boldsymbol{d}$ is a power of 2 for bit shift for division & multiplication

Merging

- Combining two heaps into one
- A few ways of implementing heaps so that the running time of a Merge is $O(\log N)$
  - Leftist Heaps
  - Skew Heaps
  - Binomial Queues

Leftist Heap

- Is a binary tree
- Has both a structural property and an ordering property.
- Ordering property is same as ordinary heap ordering property
- Structural property is different: is not perfectly balanced, but actually attempts to be unbalanced.
- Defined using the null path length

Null path length

- Npl(X) of any node X: the length of the shortest path from X to a node without two children.
- Npl(X) = 0 when X is a node with zero or one child
- Npl(NULL) = −1.
- Npl(X) = 1 + Npl(Y) where Y is a child of X with a minimum null path length.

Leftist heap property

- For a node X in the heap, $Npl(LC_X) >= Npl(RC_X)$ where LC_X and RC_X are left child and right child of X, respectively

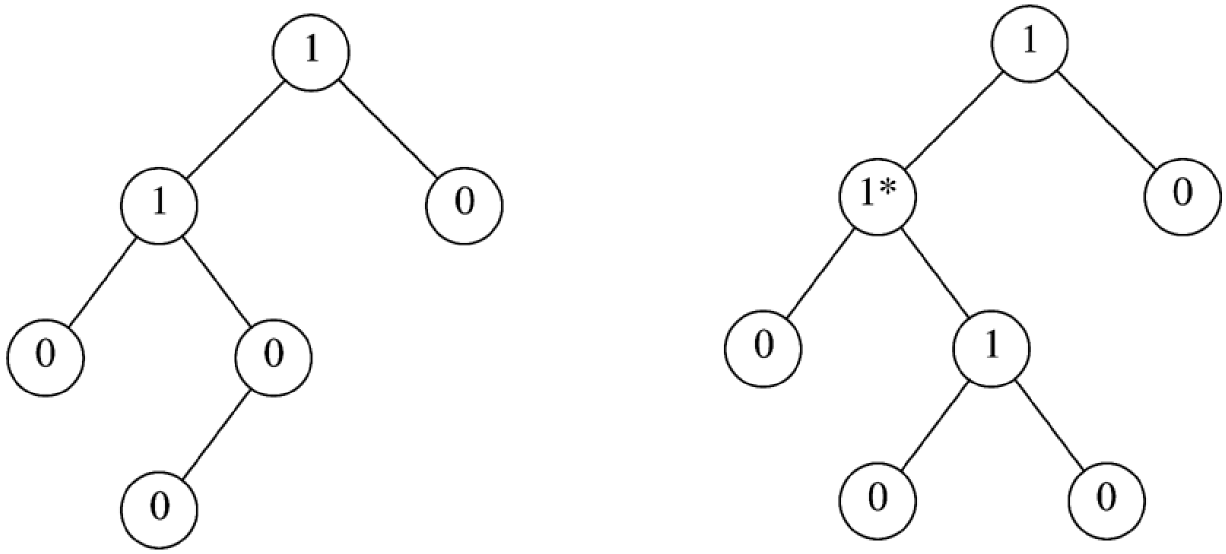


Figure 6.20 Null path lengths for two trees; only the left tree is leftist

Observations

- The tree is unbalanced and biases deeply toward the left.
- The tree has a left deep paths while the right path ought to be short

(Theorem 6.2)

A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes

=> proof by induction… (you try!!!)

Type declaration

```
struct TreeNode;
typedef struct TreeNode *PriorityQueue;

/* Minimal set of priority queue operations */
/* Note that nodes will be shared among several */
/* leftist heaps after a merge; the user must */
/* make sure to not use the old leftist heaps */

PriorityQueue Initialize( void );
ElementType FindMin( PriorityQueue H );
int IsEmpty( PriorityQueue H );
PriorityQueue Merge( PriorityQueue H1, PriorityQueue H2 );

#define Insert( X, H ) ( H = Insert1( ( X ), H ) )
/* DeleteMin macro is left as an exercise */

PriorityQueue Insert1( ElementType X, PriorityQueue H );
PriorityQueue DeleteMin1( PriorityQueue H );
```

Driving Routine for Merging

```
        /* Place in implementation file */
        struct TreeNode
        {
            ElementType   Element;
            PriorityQueue Left;
            PriorityQueue Right;
            int           Npl;
        };

        PriorityQueue
        Merge( PriorityQueue H1, PriorityQueue H2 )
        {
/* 1*/      if( H1 == NULL )
/* 2*/          return H2;
/* 3*/      if( H2 == NULL )
/* 4*/          return H1;
/* 5*/      if( H1->Element < H2->Element )
/* 6*/          return Merge1( H1, H2 );
            else
/* 7*/          return Merge1( H2, H1 );
        }
```

Actual Merging Routine

```
        static PriorityQueue
        Merge1( PriorityQueue H1, PriorityQueue H2 )
        {
/* 1*/      if( H1->Left == NULL )  /* Single node */
/* 2*/          H1->Left = H2;      /* H1->Right is already NULL,
                                       H1->Npl is already 0 */
            else
            {
/* 3*/          H1->Right = Merge( H1->Right, H2 );
/* 4*/          if( H1->Left->Npl < H1->Right->Npl )
/* 5*/              SwapChildren( H1 );

/* 6*/          H1->Npl = H1->Right->Npl + 1;
            }
/* 7*/      return H1;
        }
```

Figure 6.27 Actual routine to merge leftist heaps

Leftist Heap Merge

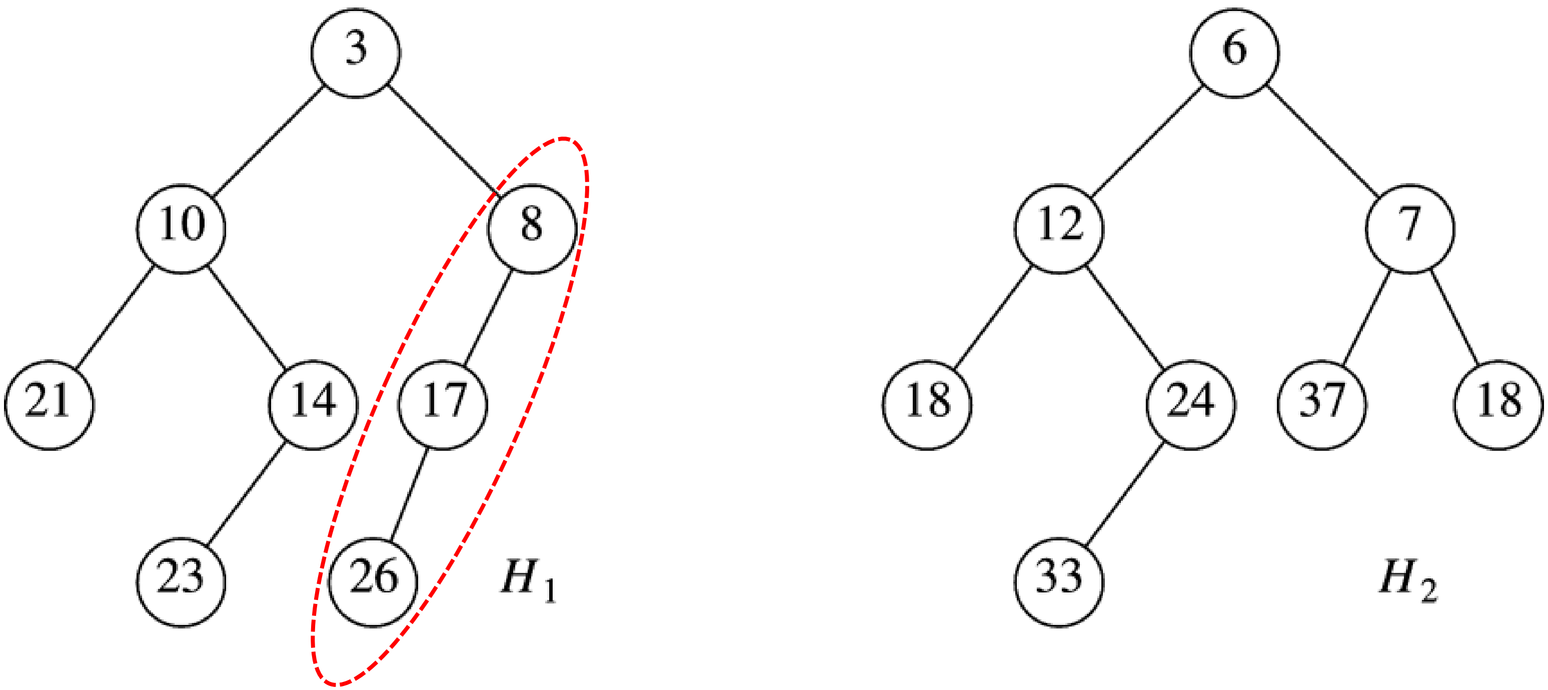


Figure 6.21 Two leftist heaps H_1 and H_2

Leftist Heap Merge

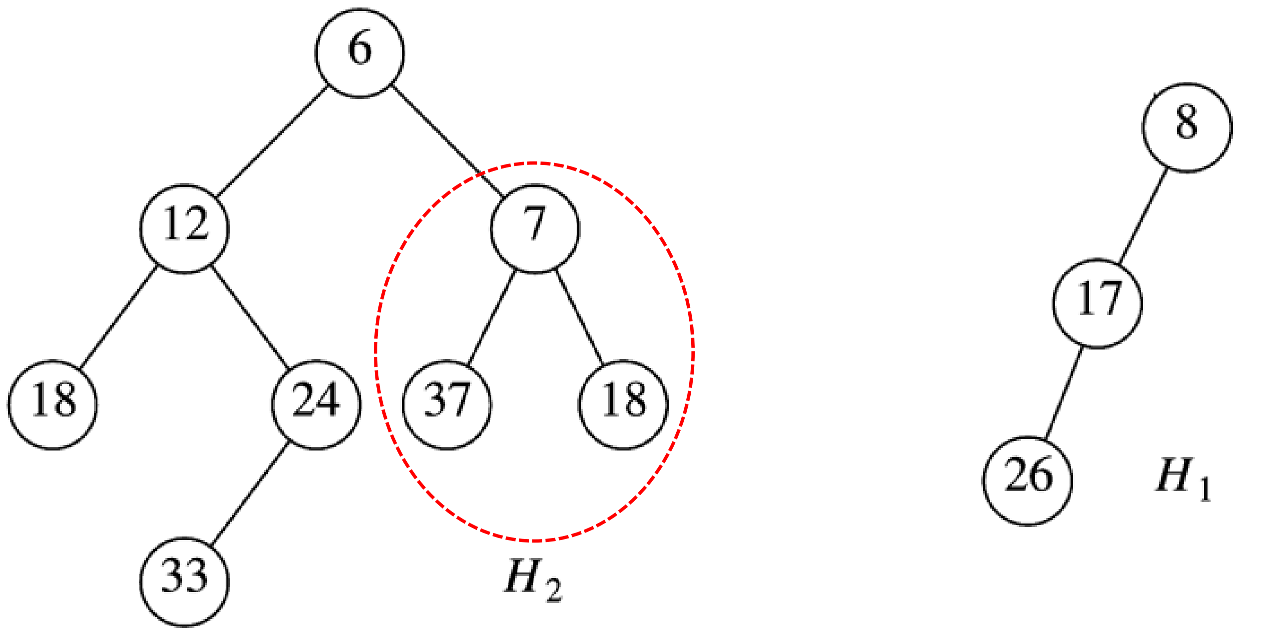


Figure 6.21 Two leftist heaps H_2 and right subheap of H_1

Leftist Heap Merge

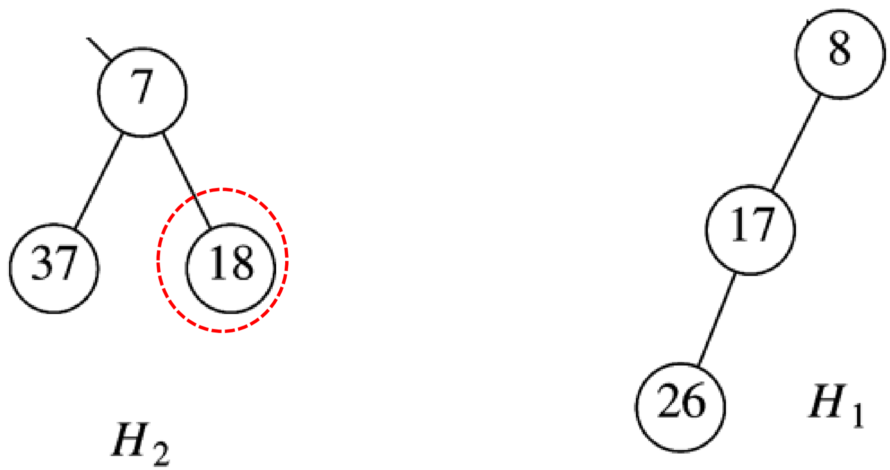


Figure 6.21 Two right subheaps of H_1 and H_2

Leftist Heap Merge

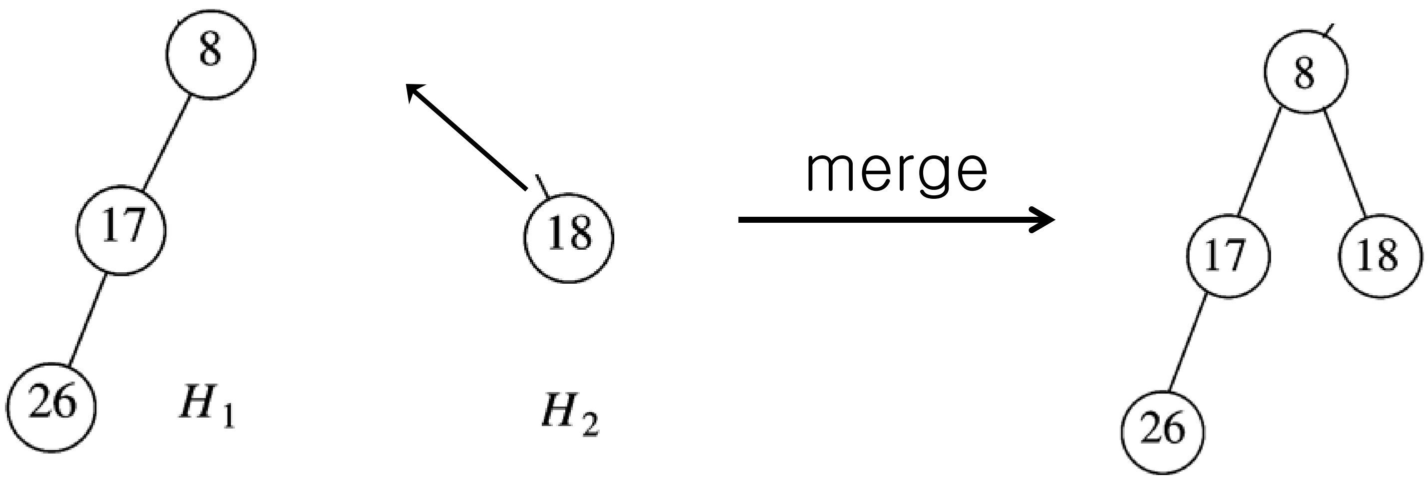


Figure 6.21 Right subheap continued

Leftist Heap Merge

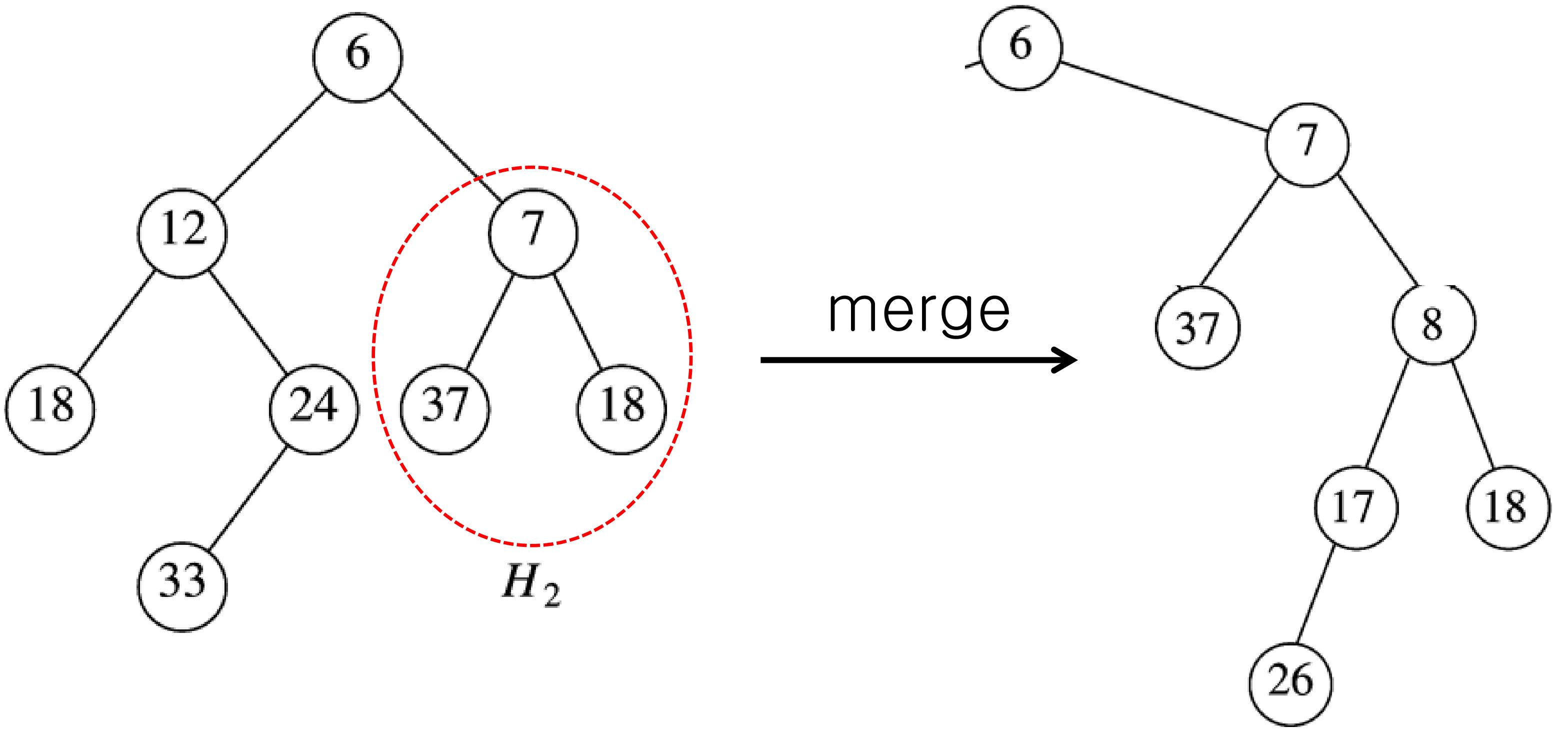


Figure 6.22 Result of merging H_2 with H_1's right subheap

Leftist Heap Merge

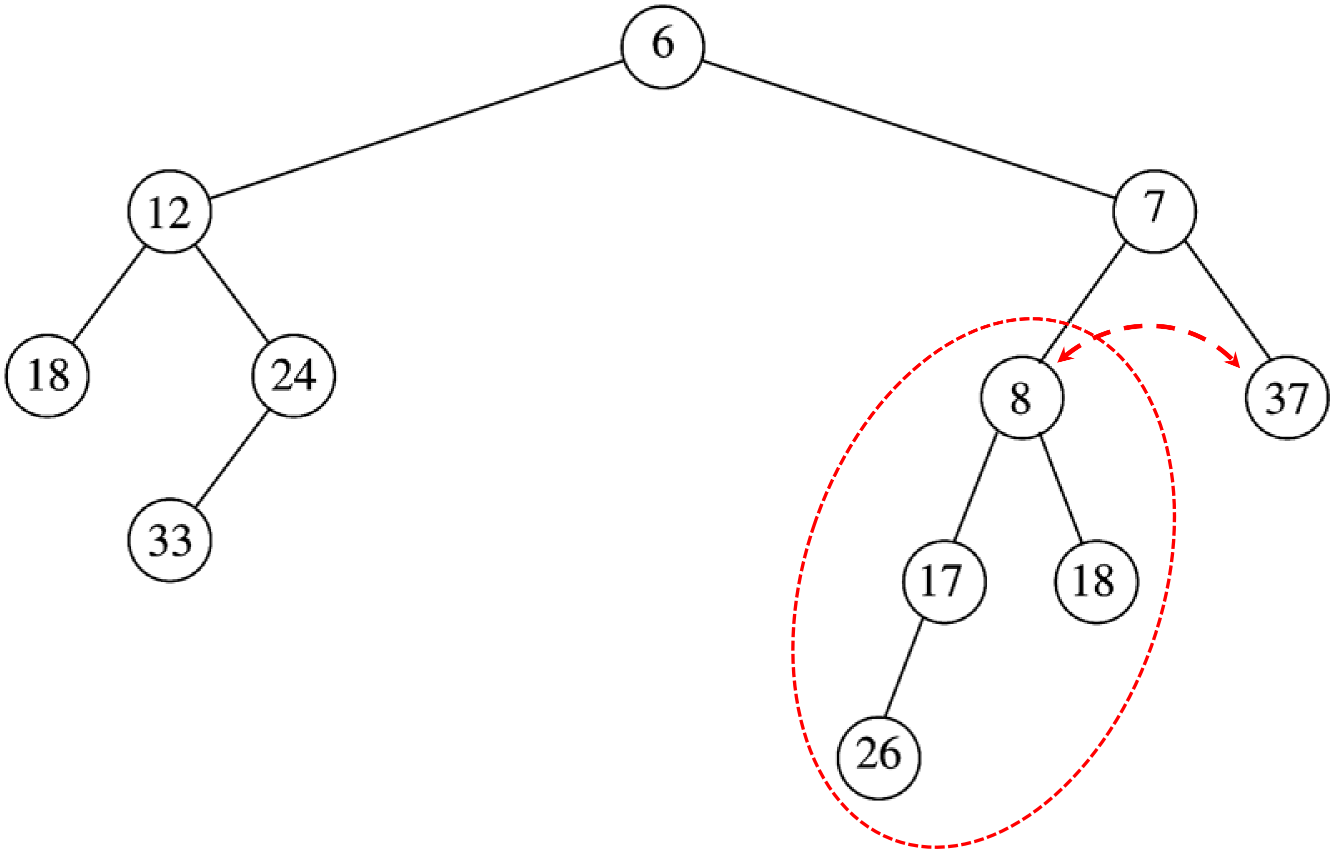


Figure 6.22 Result of merging H_2 with H_1's right subheap

Leftist Heap Merge

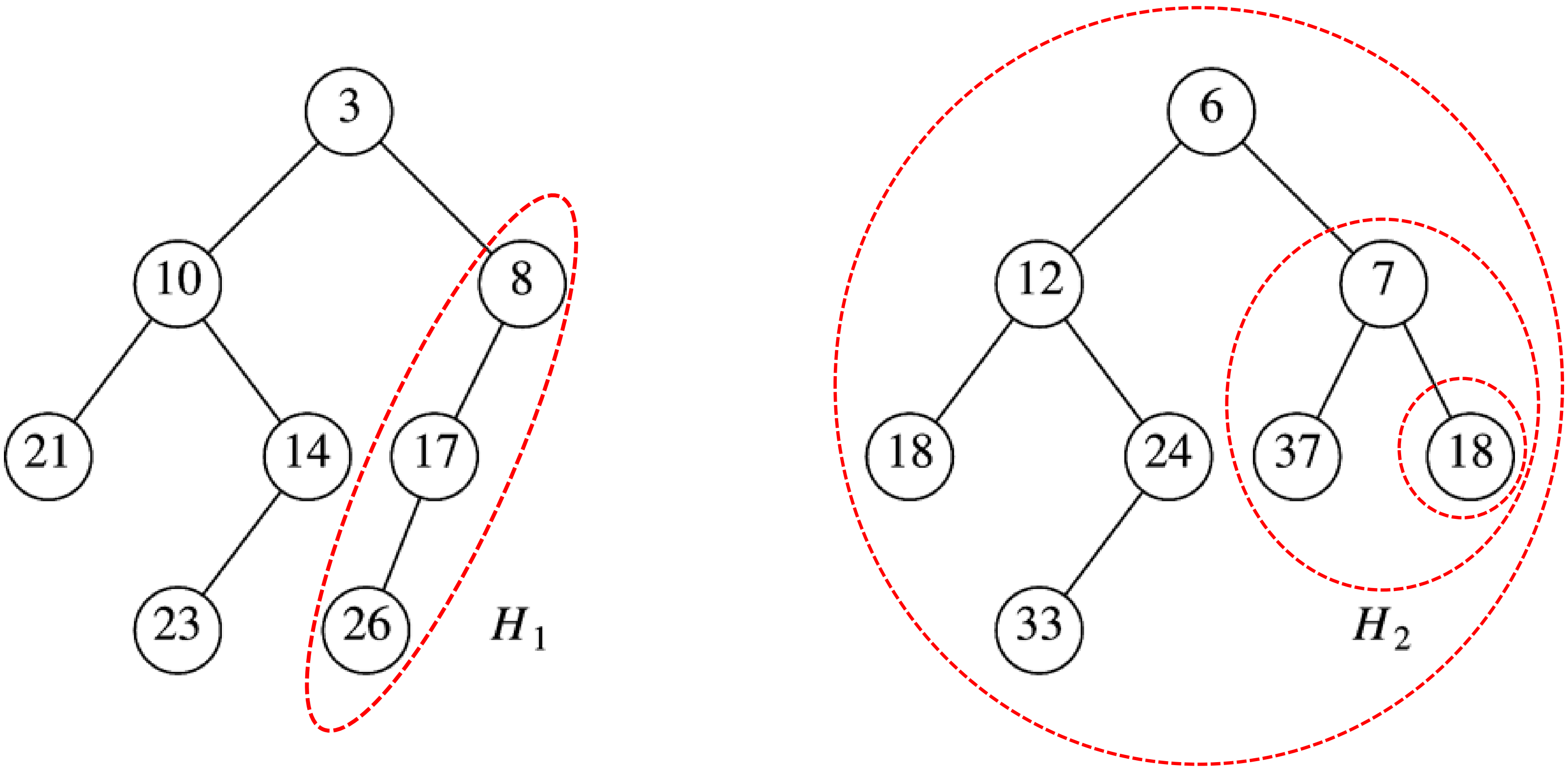


Figure 6.21 Two leftist heaps H_1 and H_2

Leftist Heap Merge

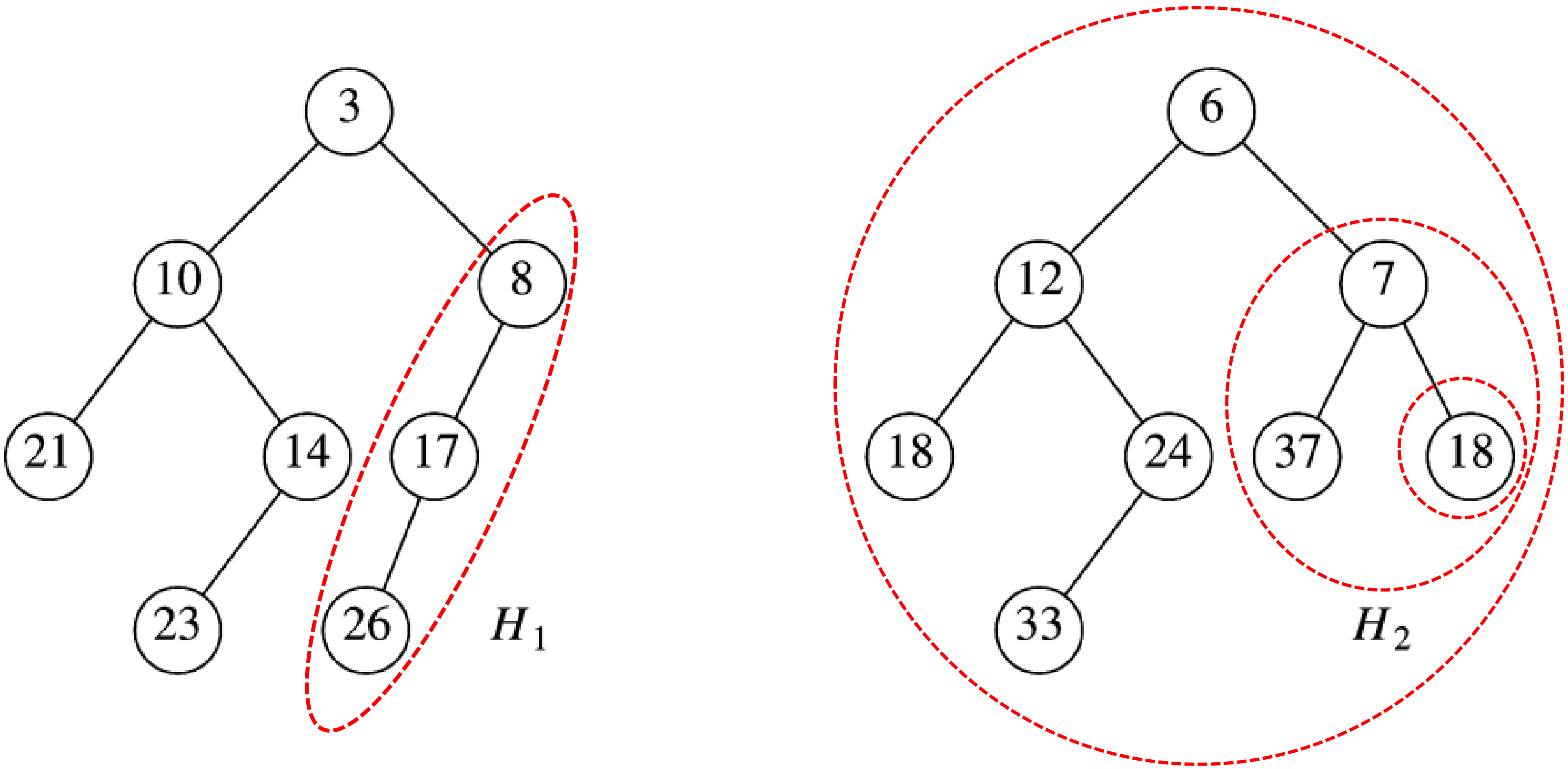


Figure 6.21 Two leftist heaps H_1 and H_2

Leftist Heap Merge

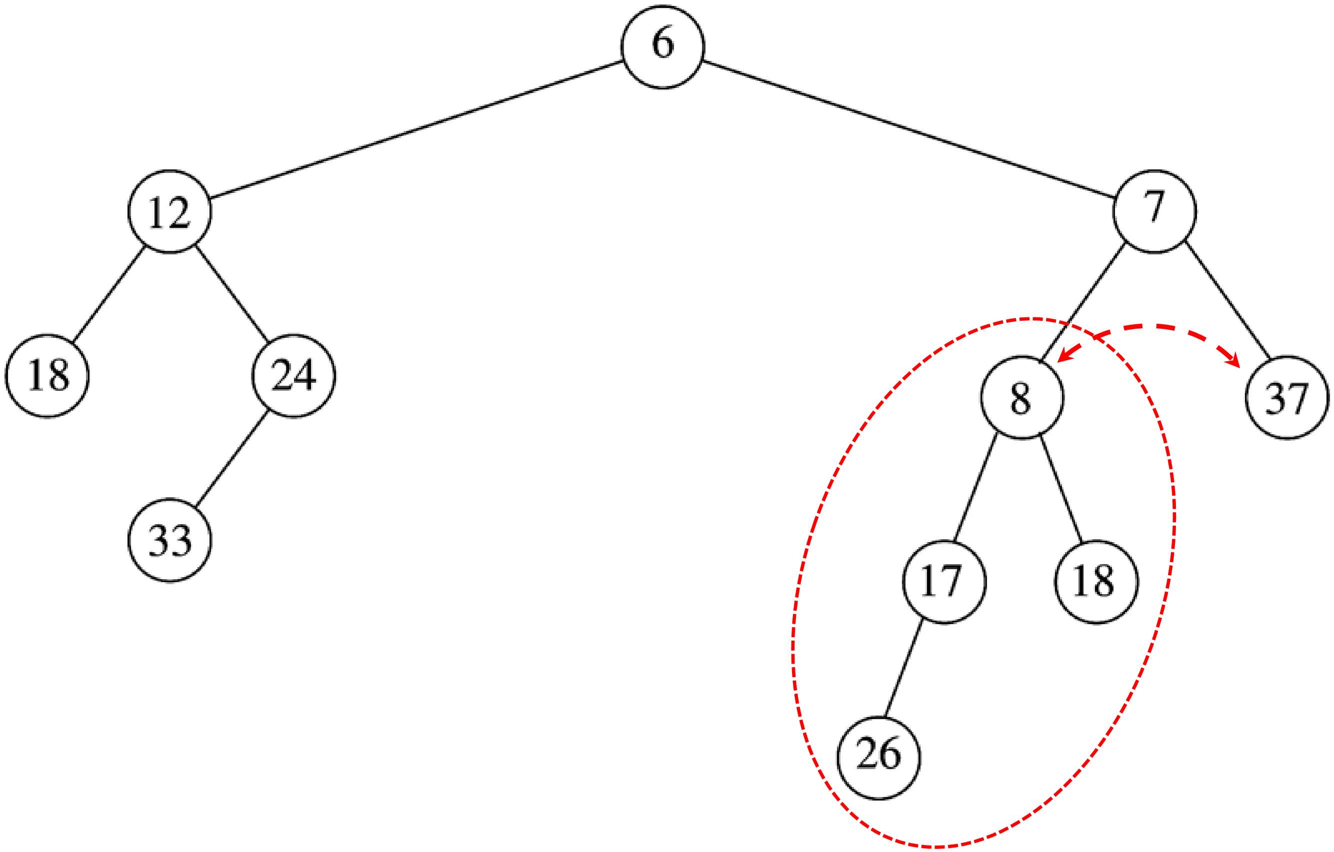


Figure 6.22 Result of merging H_2 with H_1's right subheap

Leftist Heap Merge

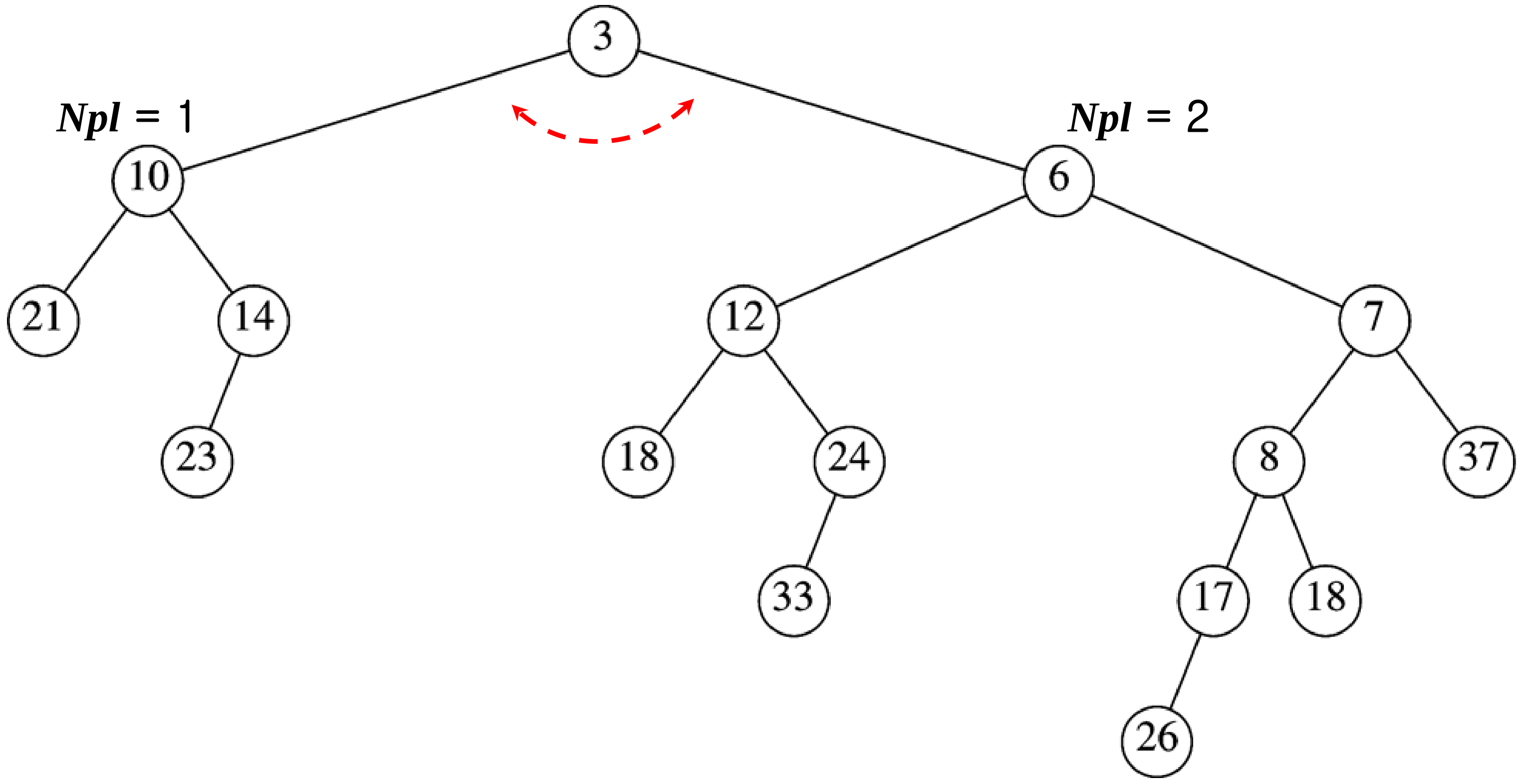


Figure 6.23 Result of attaching leftist heap of previous figure as H_1's right child

Leftist Heap Merge

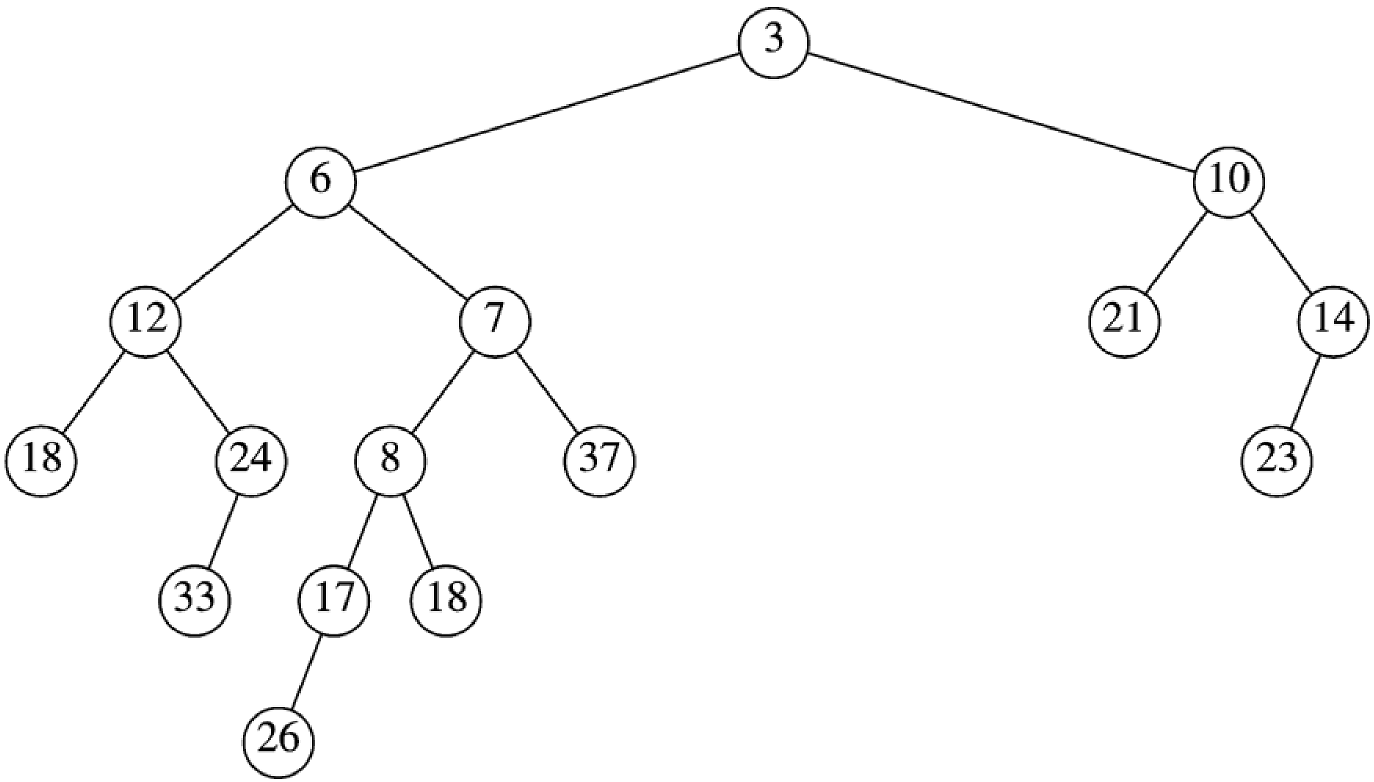


Figure 6.24 Result of swapping children H_1's root

Leftist Heap Insertion

```
PriorityQueue
Insert1( ElementType X, PriorityQueue H )
{
    PriorityQueue SingleNode;

    SingleNode = malloc( sizeof( struct TreeNode ) );
    if( SingleNode == NULL )
        FatalError( "Out of space!!!" );
    else
    {
        SingleNode->Element = X; SingleNode->Npl = 0;
        SingleNode->Left = SingleNode->Right = NULL;
        H = Merge( SingleNode, H );
    }
    return H;
}
```

Figure 6.29 Insertion routine for leftist heaps

Leftist Heap DeleteMin

```
        /* DeleteMin1 returns the new tree; */
        /* To get the minimum, use FindMin */
        /* This is for convenience */

        PriorityQueue
        DeleteMin1( PriorityQueue H )
        {
            PriorityQueue LeftHeap, RightHeap;

/* 1*/      if( IsEmpty( H ) )
            {
/* 2*/          Error( "Priority queue is empty" );
/* 3*/          return H;
            }

/* 4*/      LeftHeap = H->Left;
/* 5*/      RightHeap = H->Right;
/* 6*/      free( H );
/* 7*/      return Merge( LeftHeap, RightHeap );
        }
```

Figure 6.30 *DeleteMin* routine for leftist heaps