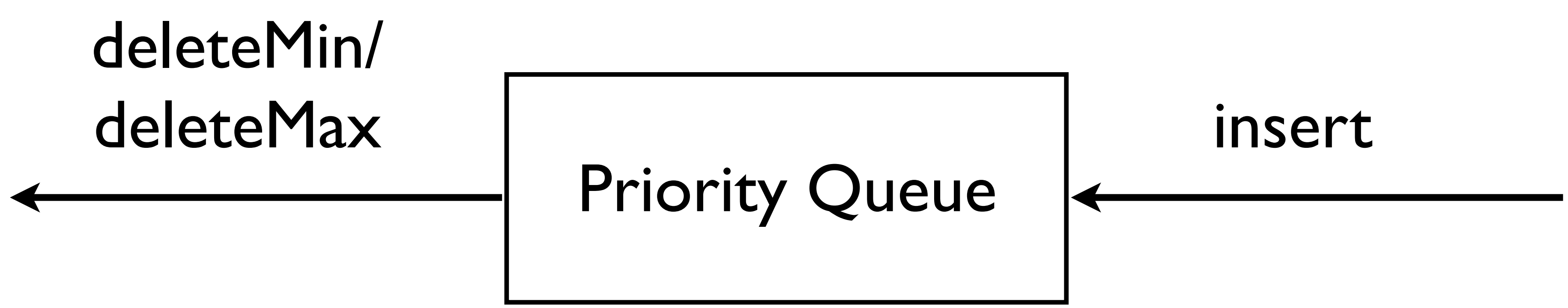
# Data Structure:

# Heap

# priority queue (heap)

- the element to be deleted is the one with the highest (or lowest) priority
- priority queue Q supports
  - insert (x, Q)
  - y = pop(Q) (=deleteMin(Q) or deleteMax(Q))

- priority queue is used for scheduling



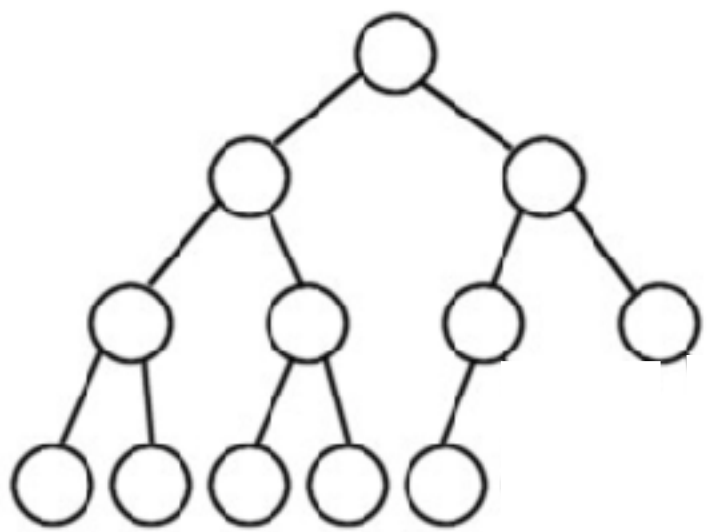

unordered linear list

heap

# binary (min) heap

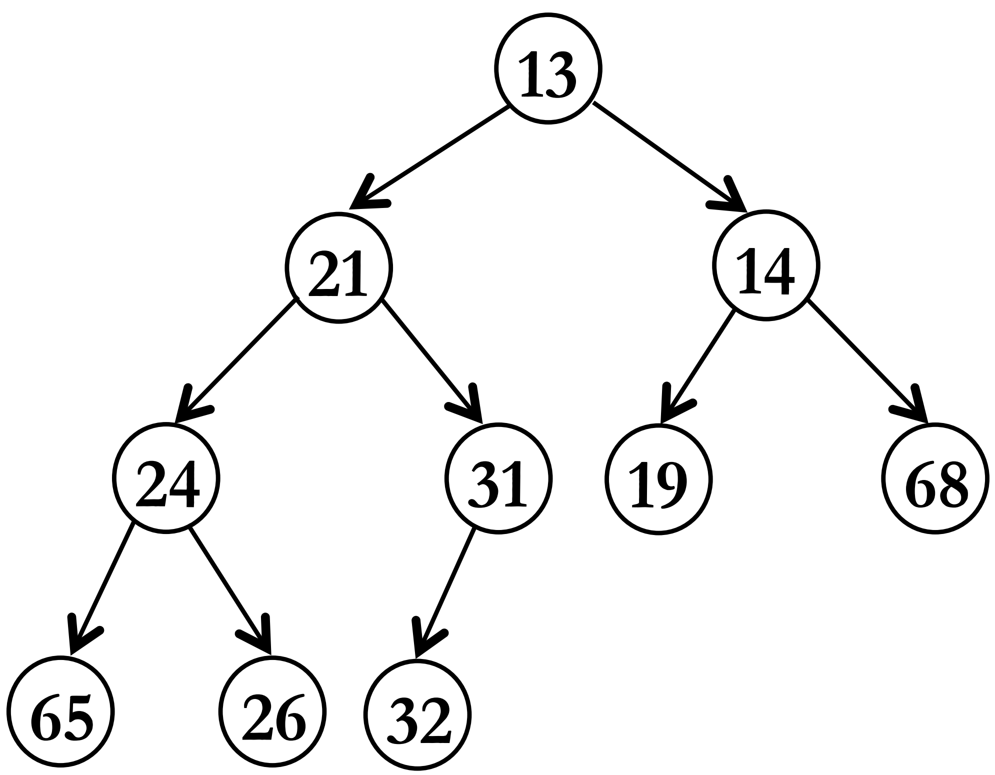
- a min heap is a complete binary tree and partially ordered tree in which the key value in each node is no larger than the key values in its children

- complete tree
  - every level of tree is completely filled, with the exception of the bottom level, which is filled from left to right

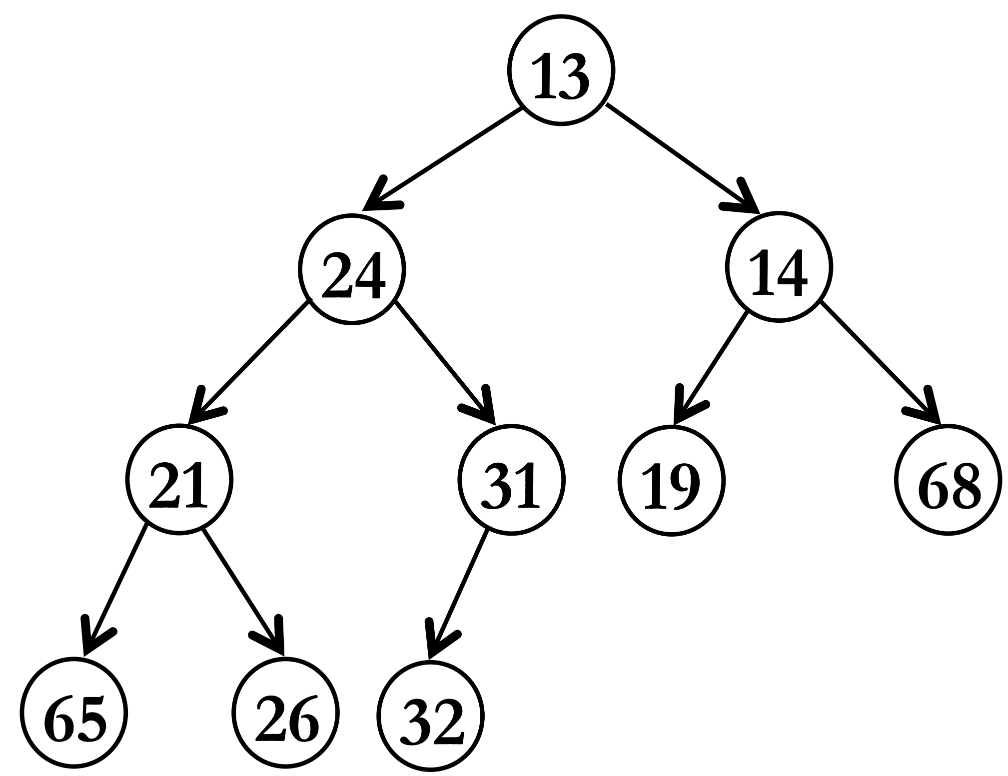


# binary (min) heap

- partially ordered tree
  - the key of each internal node is less than or equal to the keys of its children
  - the smallest element should be at the root



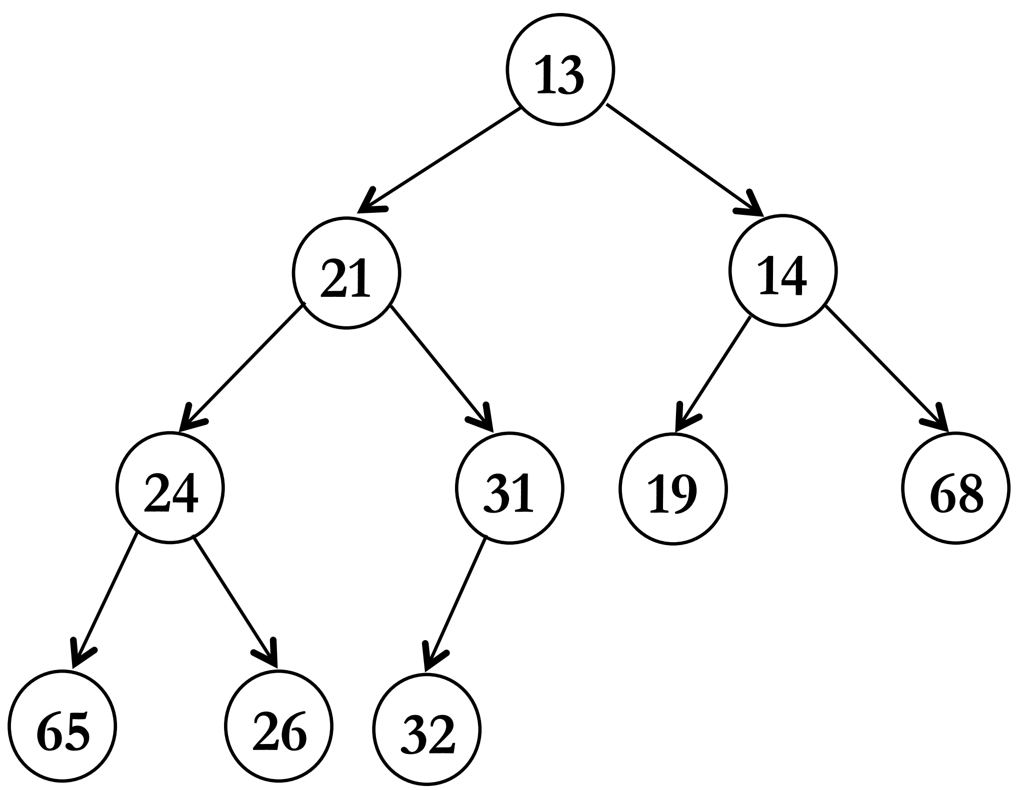

?




?

# binary heap

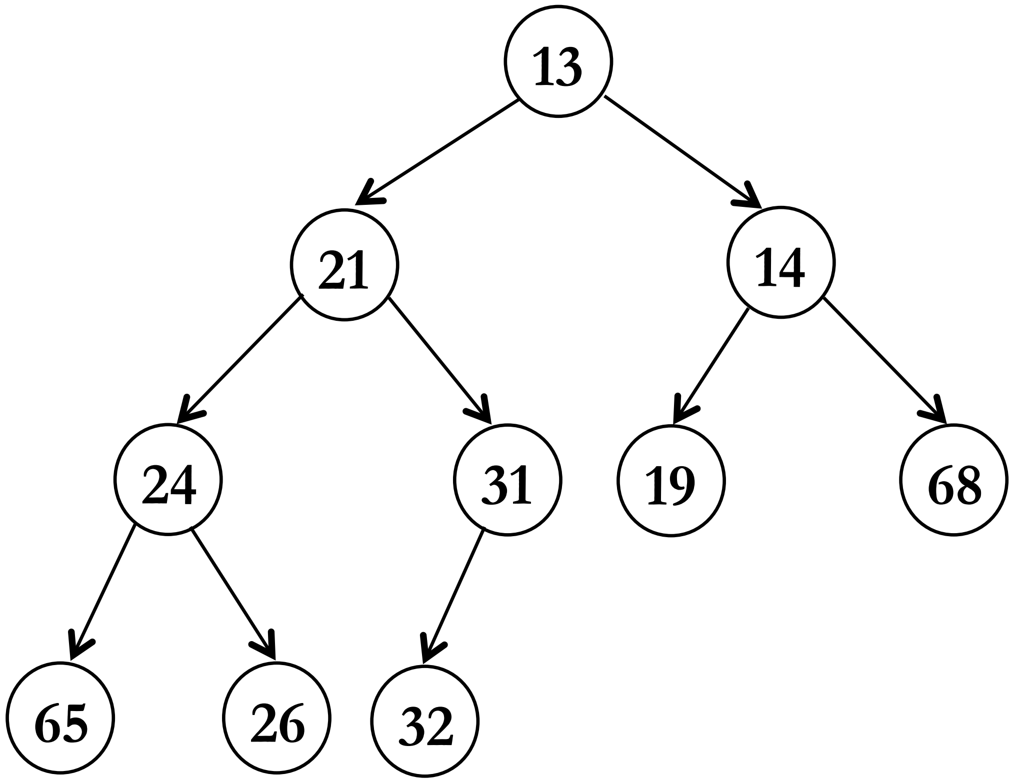
- binary heap can be stored in array since it is a complete tree




- left_child(i) = 2i (when 2i <= n)
- right_child(i) = 2i + 1 (when 2i+1 <=n)
- parent(i) = floor(i/2)   (when i>=2)

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| array | | 13 | 21 | 14 | 24 | 31 | 19 | 68 | 65 | 26 | 32 | |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

# binary heap

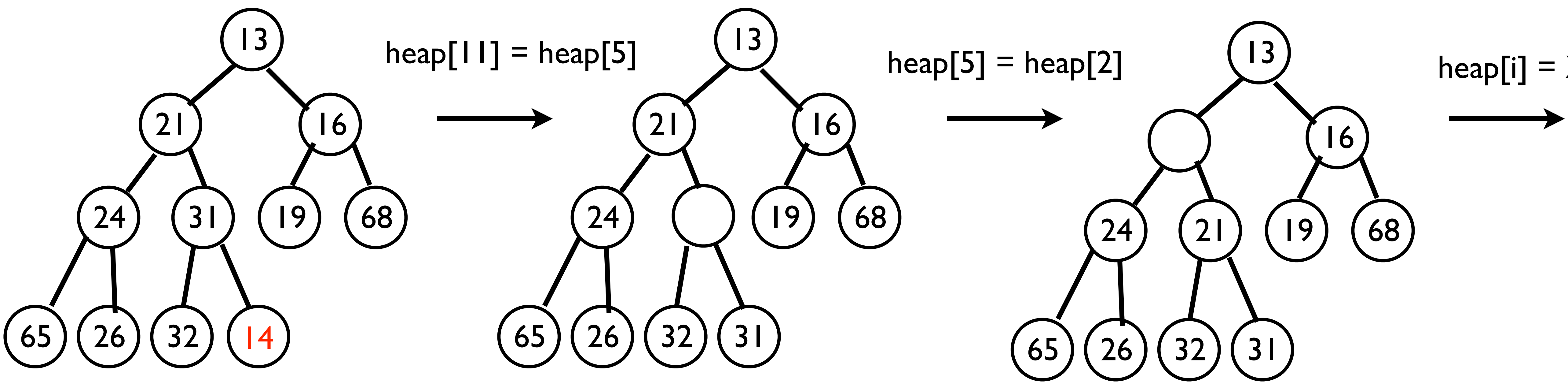



```
struct HeapStruct
{
   int Capacity;   // max heap capacity
   int Size;       // current heap size
   ElementType *Elements;
};
```

# insertion

insertion of 14

x=14




i=11
heap[floor(i/2)] $\leq$ X ?

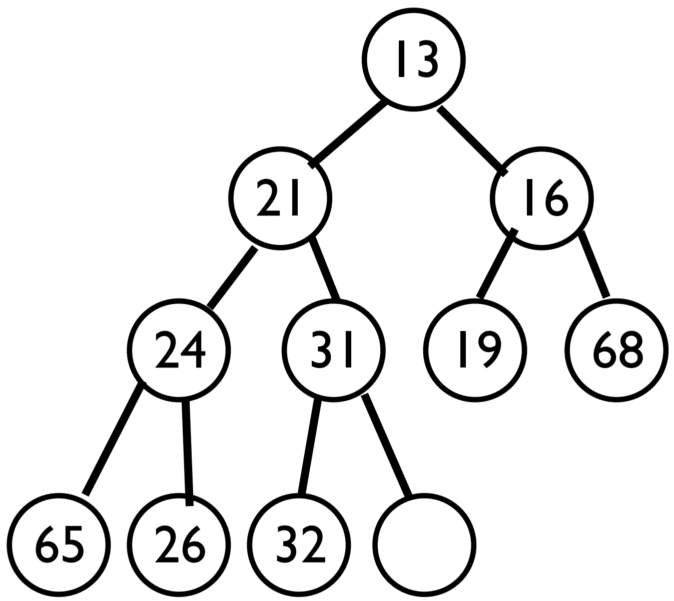
i=5
heap[floor(i/2)] $\leq$ X ?

i=2
heap[floor(i/2)] $\leq$ X ?

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 13 | 21 | 16 | 24 | 31 | 19 | 68 | 65 | 26 | 32 | 14 |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

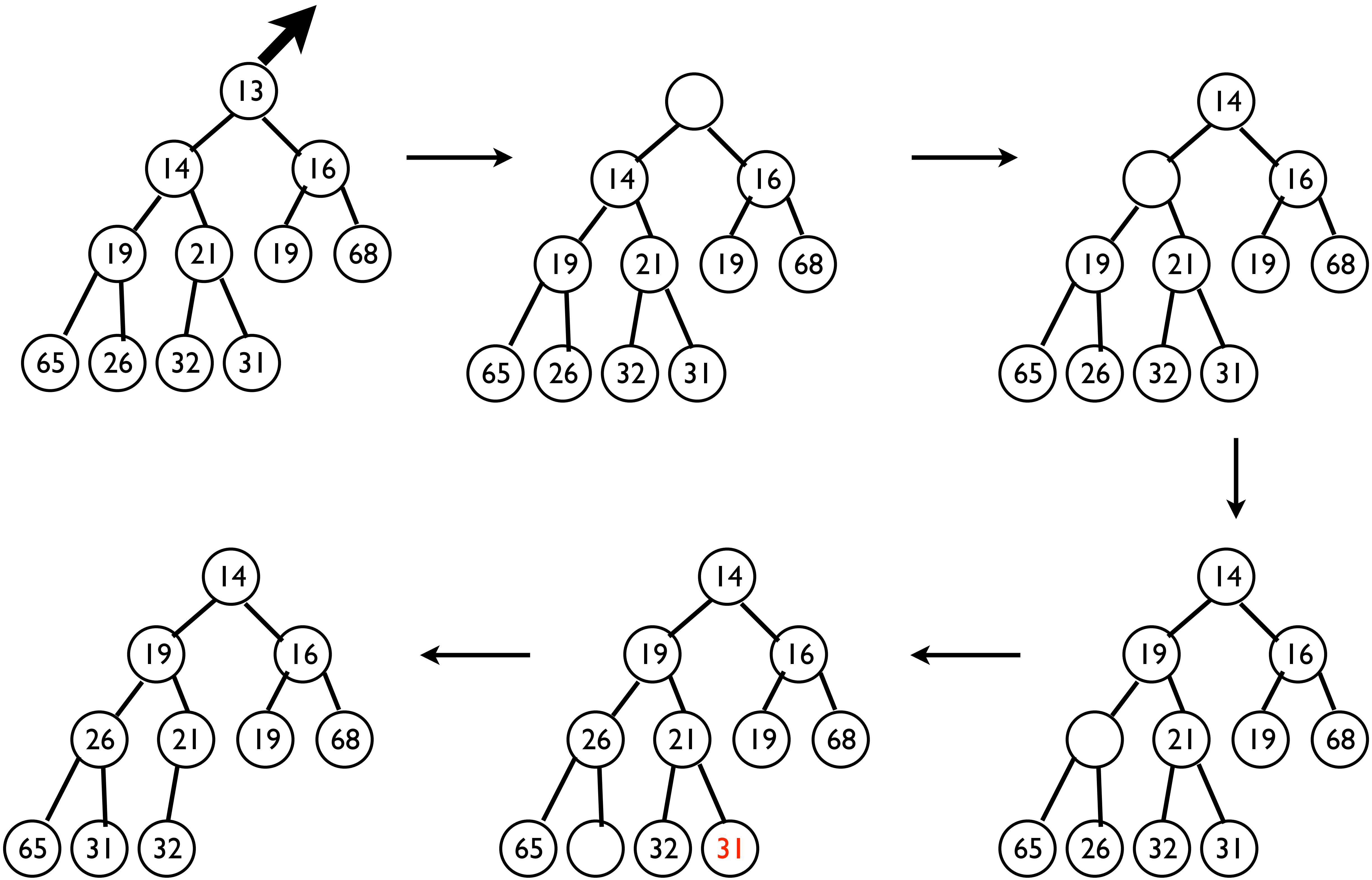
# insertion

```
void Insert( ElementType X, PriorityQueue H )
{

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

   /*percolating up*/
   for( i = ++H->size;    H->Elements[ i/2 ] > X;    i /= 2 )
             H->Elements[ i ] = H->Elements[ i/2 ];

   H->Elements[ i ] = X;
}
```

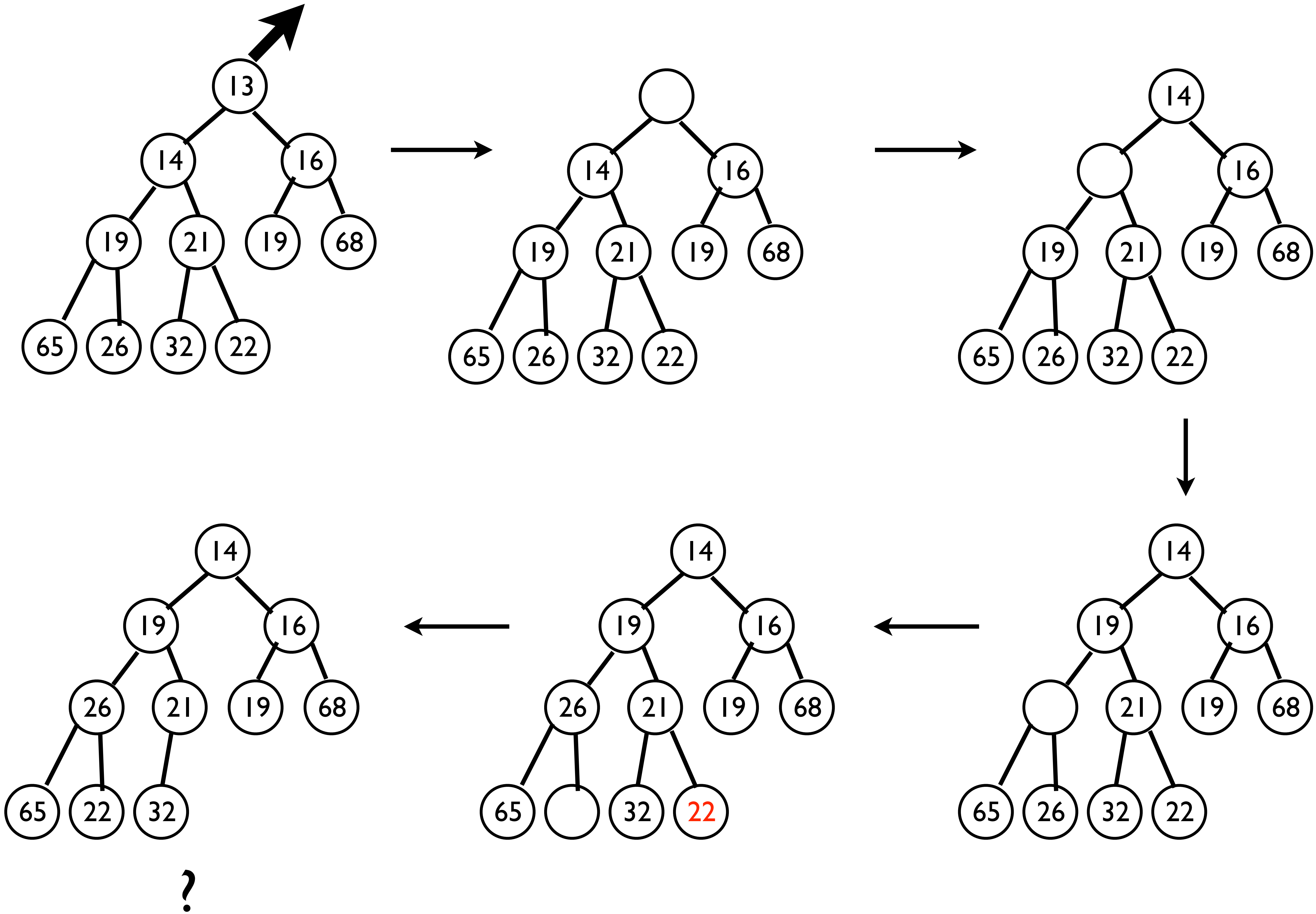



the complexity of the insertion function is $O(\log_2 n)$

# DeleteMin: a possible way?

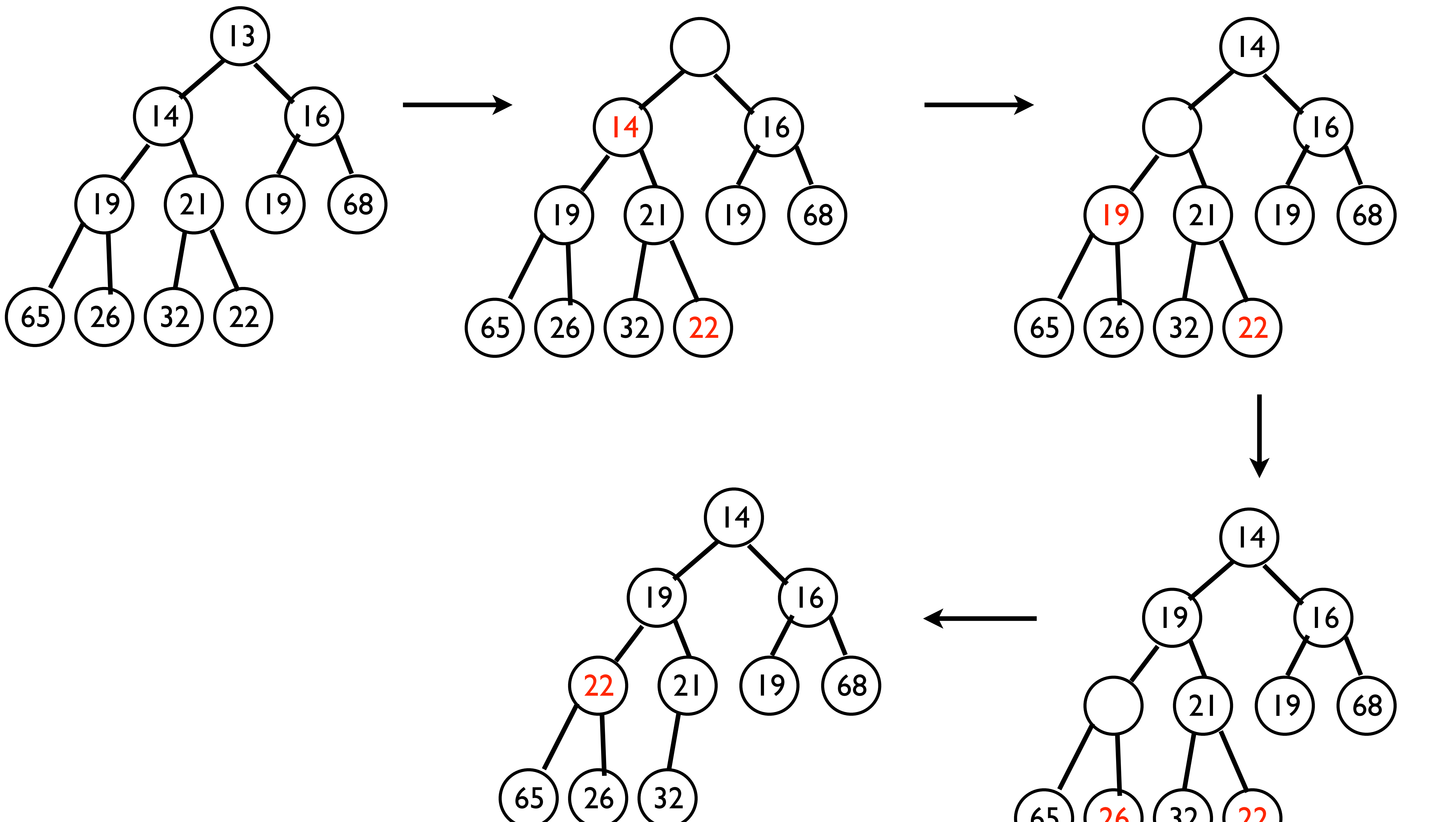


# DeleteMin: what if?



# DeleteMin

- choose the smaller one between H->Elements[ LChild ] and H->Elements[ RChild ]
- choose the smaller one between LastElement and H->Elements[ Child ]



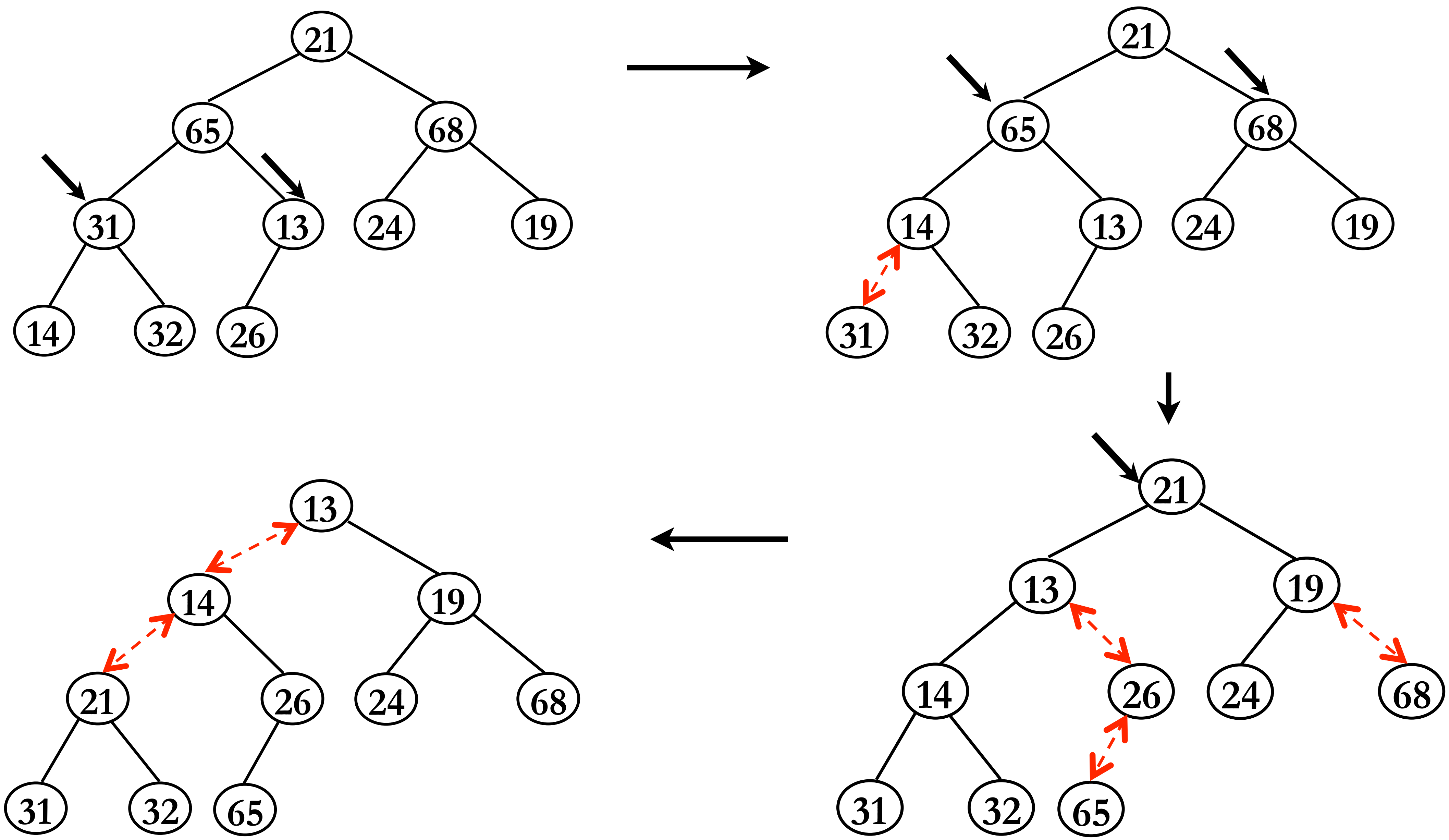
# DeleteMin

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

        MinElement = H->Elements[ 1 ];
        LastElement = H->Elements[ H->Size-- ];

       /*percolating down*/
       for( i = 1; i*2 <= H->Size; i = Child )
       {
            Child = i * 2;
            if( Child != H->Size && H->Elements[ Child + 1 ] < H->Elements[ Child ] )
                    Child++;

            if( LastElement > H->Elements[ Child ] )
                    H->Elements[ i ] = H->Elements[ Child ];
           else
                    break;
       }

       H->Elements[ i ] = LastElement;
       return MinElement;
}
```

the complexity of the deletion function is $O(\log_2 n)$

# BuildHeap

- Build a Heap containing $n$ keys takes $O(n \log n)$ with consecutive insertions
- But it can take $O(n)$ if they are already in array.
- Starting with the lowest non-leaf node, working back towards root, perform percolating-down on each node of the tree.



# BuildHeap

Let's assume that the tree is complete:

There is one key at level 0, which might sift down $h$ levels.

There are two keys at level 1, which might sift down $h$ – 1 levels

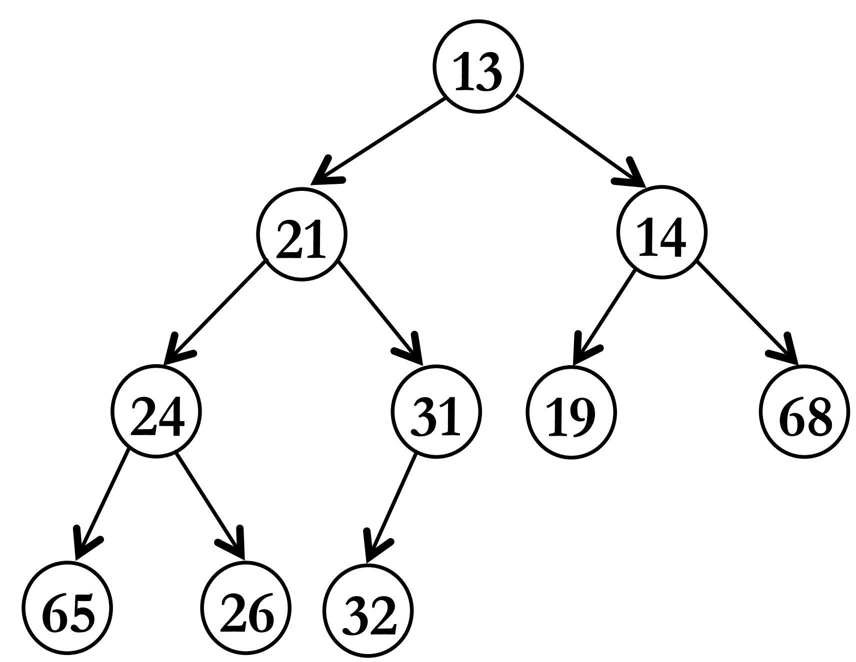
There are four keys at level 2, which might sift down $h$ – 2 levels

...

$$S = h + 2(h-1) + 4(h-2) + \ldots + 2^{h-1}(1)$$

$$2S = 2h + 4(h-1) + 8(h-2) + \ldots + 2^{h-1}(2) + 2^{h}(1)$$

$$\begin{aligned} 2S - S &= -h + (2 + 4 + \ldots + 2^{h-1}) + 2^{h} \\ &= -h - 1 + (1 + 2 + 4 + \ldots + 2^{h-1}) + 2^{h} \\ &= 2^{h} + 2^{h} - (h+2) = 2 \cdot 2^{h} - h - 2 \\ &= 2 \cdot 2^{\log n} - \log n - 2 \le 2n \end{aligned}$$



# heap sort

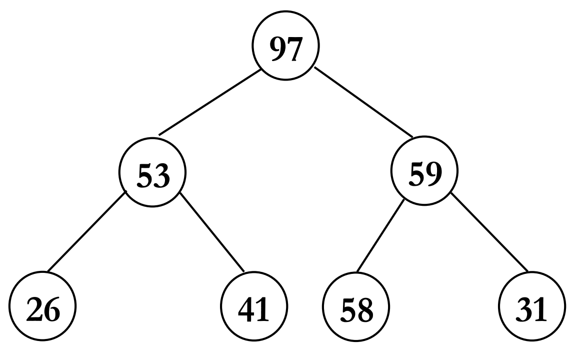
- building binary heap of n elements: O(n)
- DeleteMin operation n times: O(n log n)
- need extra space to save the sorted list: use the last cell in the previous heap

```
void HeapSort (ElementType A[], int N)
{
      int   i;
      for (i = N/2;   i >0;   i --)       /* Build Heap */
             PercDown (A,  i,  N);

      for (i = N;  i > 0;   i --)
      {
              Swap(&A[1],  &A[i] );     /*DeleteMax */
              PercDown(A, 1, i-1);
      }
}
```

# heap sort (by increasing order with max heap)

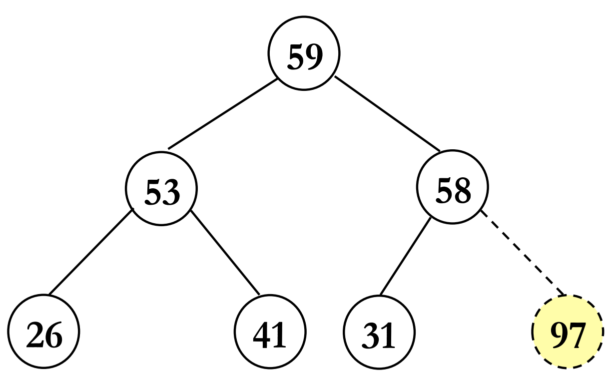
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 31 | 41 | 59 | 26 | 53 | 58 | 97 | | |

BuildHeap

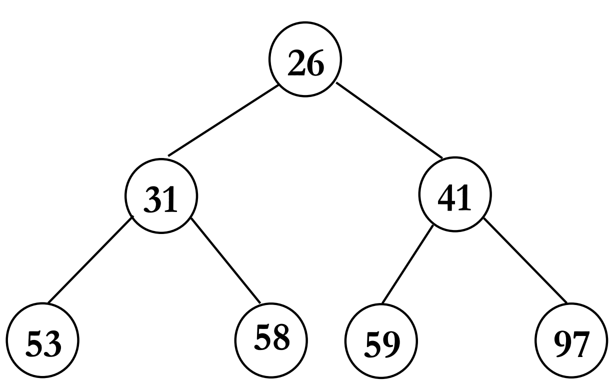


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 97 | 53 | 59 | 26 | 41 | 58 | 31 | | |

DeleteMax



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 59 | 53 | 58 | 26 | 41 | 31 | 97 | | |



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 26 | 31 | 41 | 53 | 58 | 59 | 97 | | |