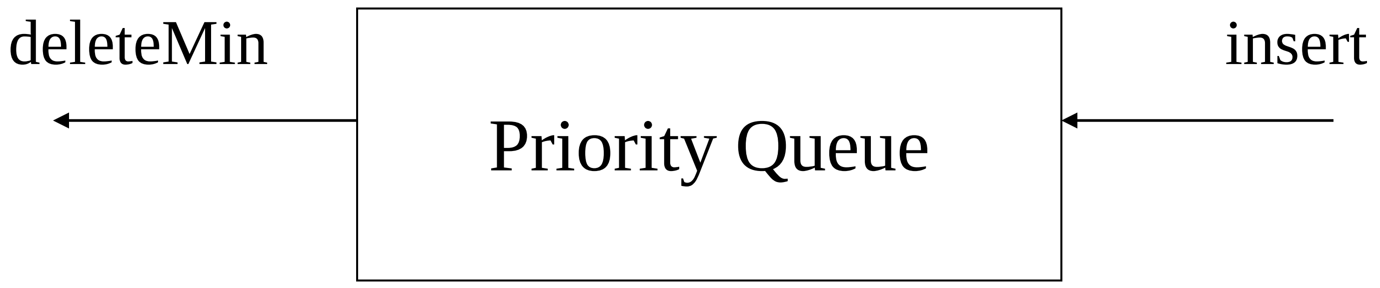
Priority Queues (Heaps)

Priority Queues

- Many applications require that we process records with keys in order, but not necessarily in full sorted order.
- Often we collect a set of items and process the one with the current minimum value.
  - e.g. jobs sent to a printer,
  - Operating system job scheduler in a multi-user environment.
  - Simulation environments
- An appropriate data structure is called a *priority queue*.

Definition

- A priority queue is a data structure that supports two basic operations: insert a new item and remove the minimum item.



Simple Implementations

- A simple linked list:
  - Insertion at the front (O(1)); delete minimum (O(N)), or
  - Keep list sorted; insertion O(N), deleteMin O(1)
- A binary search tree:
  - This gives an O(log N) average for both operations.
  - But BST class supports a lot of operations that are not required.
- An array: Binary Heap
  - Does not require links and will support both operations in O(logN) wost-case time.

Binary Heap

- The binary heap is the classic method used to implement priority queues.
- We use the term **heap** to refer to the binary heap.
- Heap is different from the term heap used in dynamic memory allocation.
- Heap has two properties:
  - Structure property
  - Ordering property

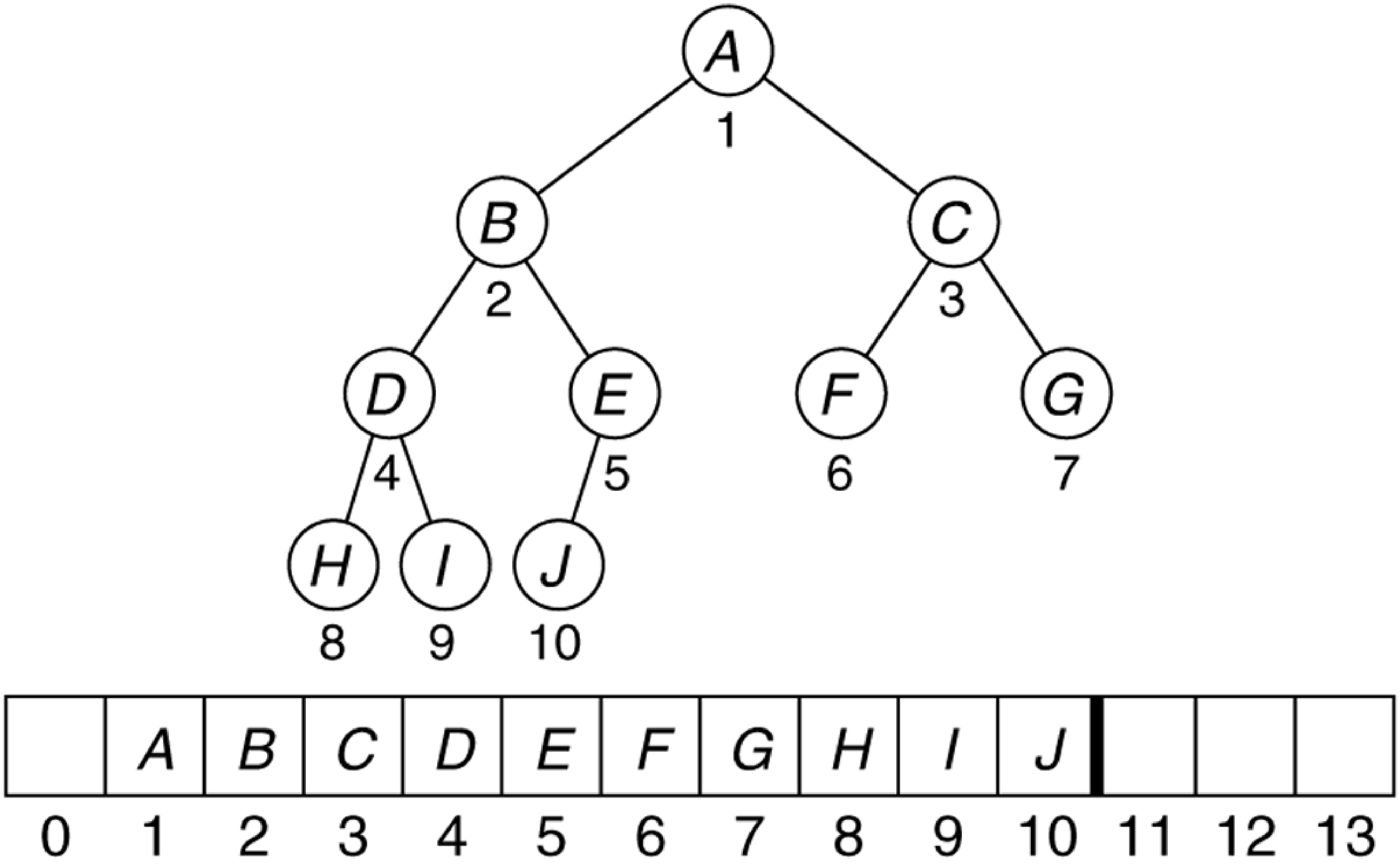
Structure Property

- A **heap** is a *complete binary tree*, represented as an array.
- A **complete binary tree** is a tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Properties of a complete binary tree

- A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes
- The height of a complete binary tree is $\lfloor \log N \rfloor$.
- It can be implemented as an array such that:
  - For any element in array position i :
    - the left child is in position $2i$,
    - the right child is in the cell after the left child $(2i + 1)$, and
    - the parent is in position $\lfloor i/2 \rfloor$.

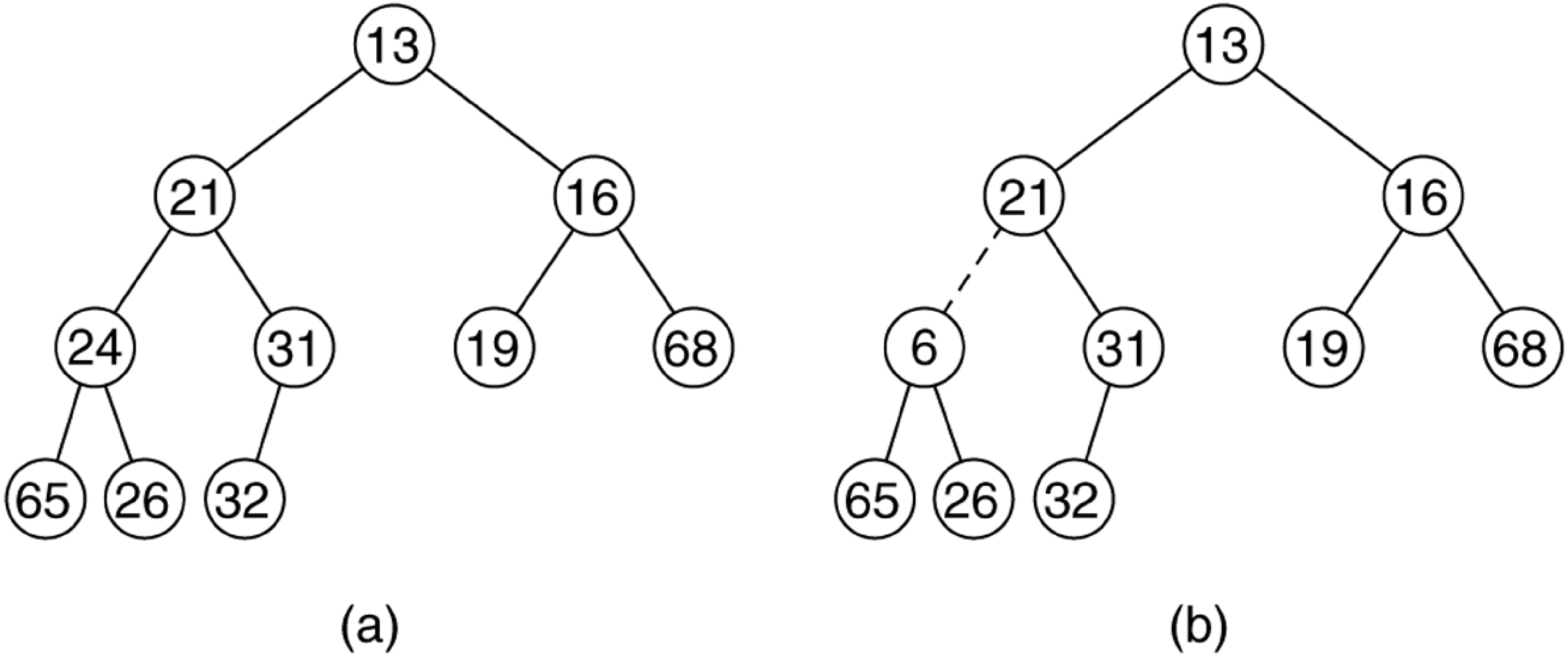
Figure 21.1
A complete binary tree and its array representation



Heap-Order Property

- In a heap, for every node X with parent P, the key in P is smaller than or equal to the key in X.
- Thus the minimum element is always at the root.
  - Thus we get the extra operation findMin in constant time.
- A **max heap** supports access of the maximum element instead of the minimum, by changing the heap property slightly.

Figure 21.3
Two complete trees: (a) a heap; (b) not a heap



Binary Heap Class

```
template <class Comparable>
class BinaryHeap
{
  public:
    BinaryHeap( int capacity = 100 );
    bool isEmpty( ) const;
    const Comparable & findMin( ) const;

    void insert( const Comparable & x );
    void deleteMin( );
    void deleteMin( Comparable & minItem );
    void makeEmpty( );

  private:
    int theSize;  // Number of elements in heap
    vector<Comparable> array;   // The heap array
    void buildHeap( );
    void percolateDown( int hole );
};
```

Basic Heap Operations: Insert

- To insert an element X into the heap:
  - We create a hole in the next available location.
  - If X can be placed there without violating the heap property, then we do so and are done.
  - Otherwise
    - we bubble up the hole toward the root by sliding the element in the hole's parent down.
    - We continue this until X can be placed in the hole.
- This general strategy is known as a *percolate up*.

Figure 21.7
Attempt to insert 14, creating the hole and bubbling the hole up

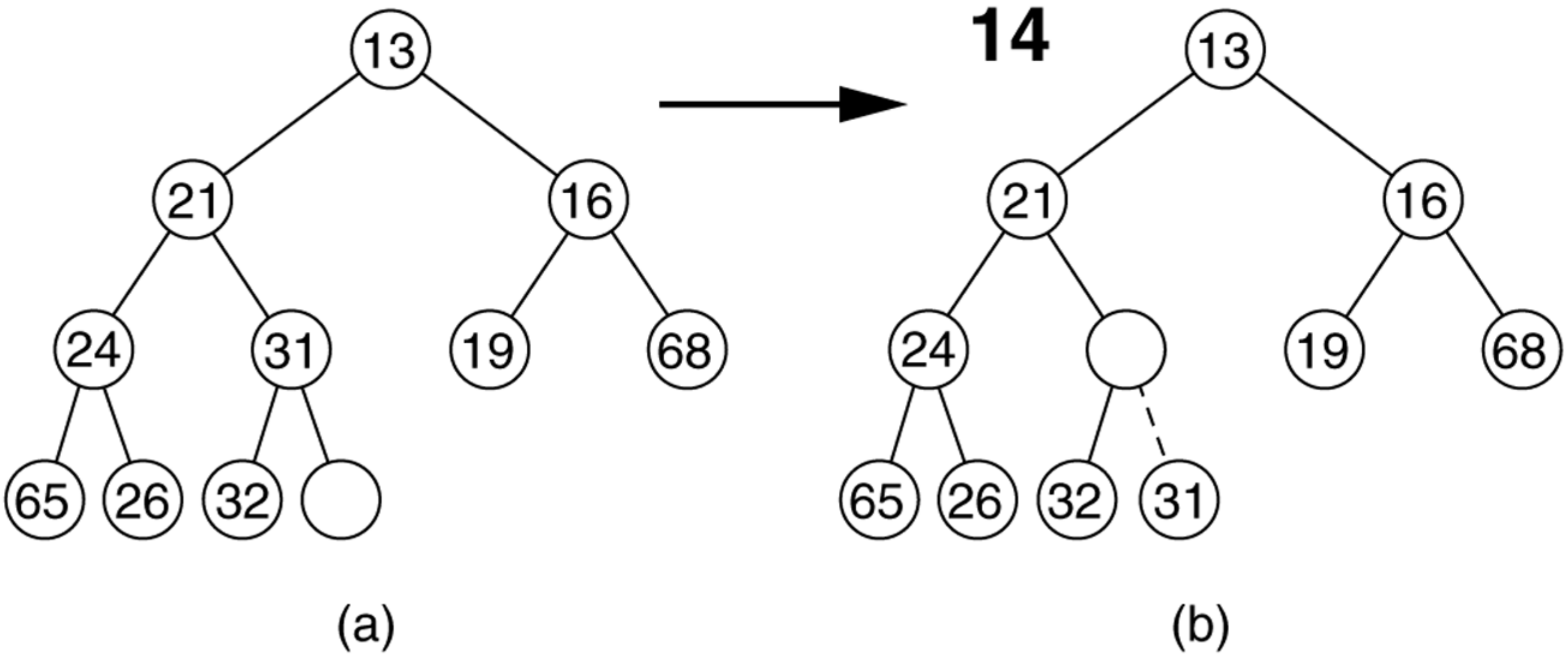
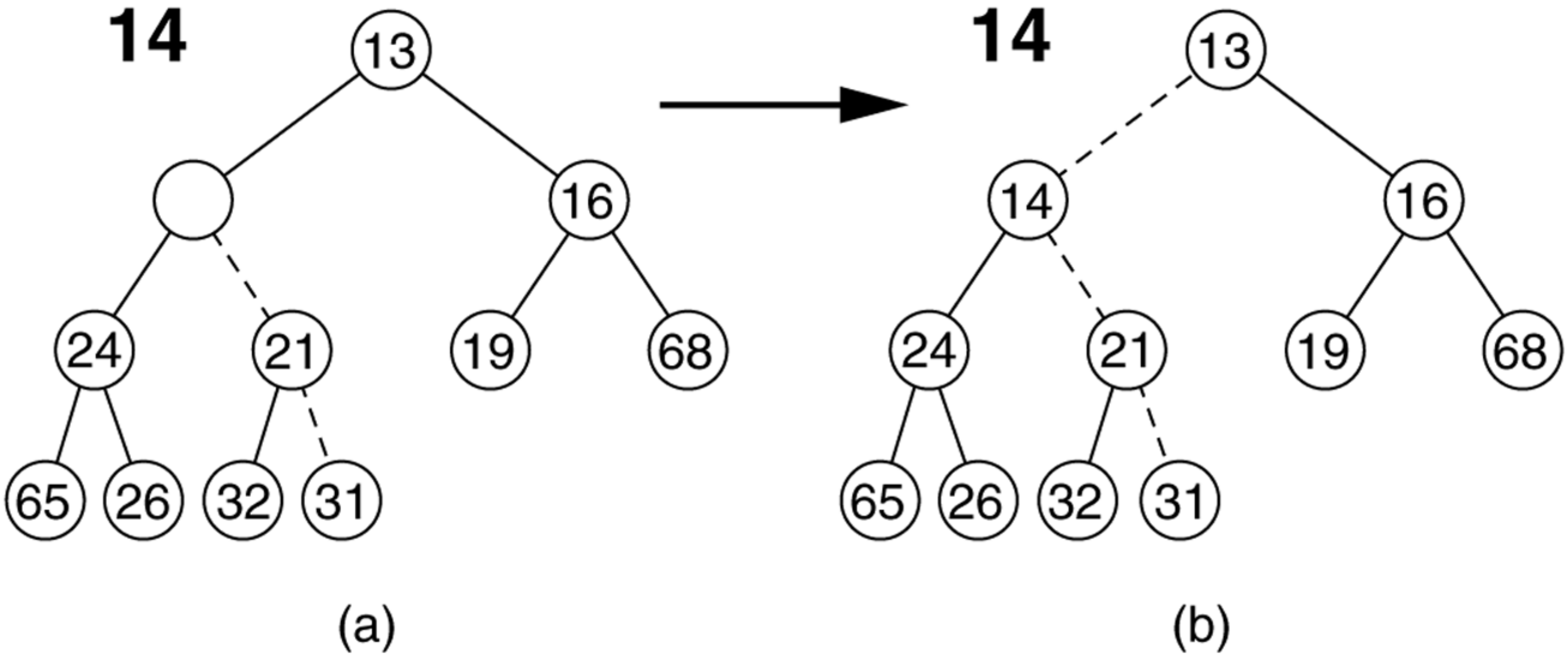


Figure 21.8

The remaining two steps required to insert 14 in the original heap shown in Figure 21.7



Insert procedure

```
// Insert item x into the priority queue, maintaining heap
   order.
// Duplicates are allowed.
template <class Comparable>
void BinaryHeap<Comparable>::insert( const Comparable & x )
{
    array[ 0 ] = x;   // initialize sentinel
    if( theSize + 1 == array.size( ) )
        array.resize( array.size( ) * 2 + 1 );

    // Percolate up
    int hole = ++theSize;
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```

Delete Minimum

- ➢ **deleteMin** is handled in a similar manner as insertion:
- Remove the minimum; a hole is created at the root.
- The last element X must move somewhere in the heap.
  - If X can be placed in the hole then we are done.
  - Otherwise,
    - We slide the smaller of the hole's children into the hole, thus pushing the hole one level down.
    - We repeat this until X can be placed in the hole.
- ➢ deleteMin is logarithmic in both the worst and average cases.

Figure 21.10

Creation of the hole at the root

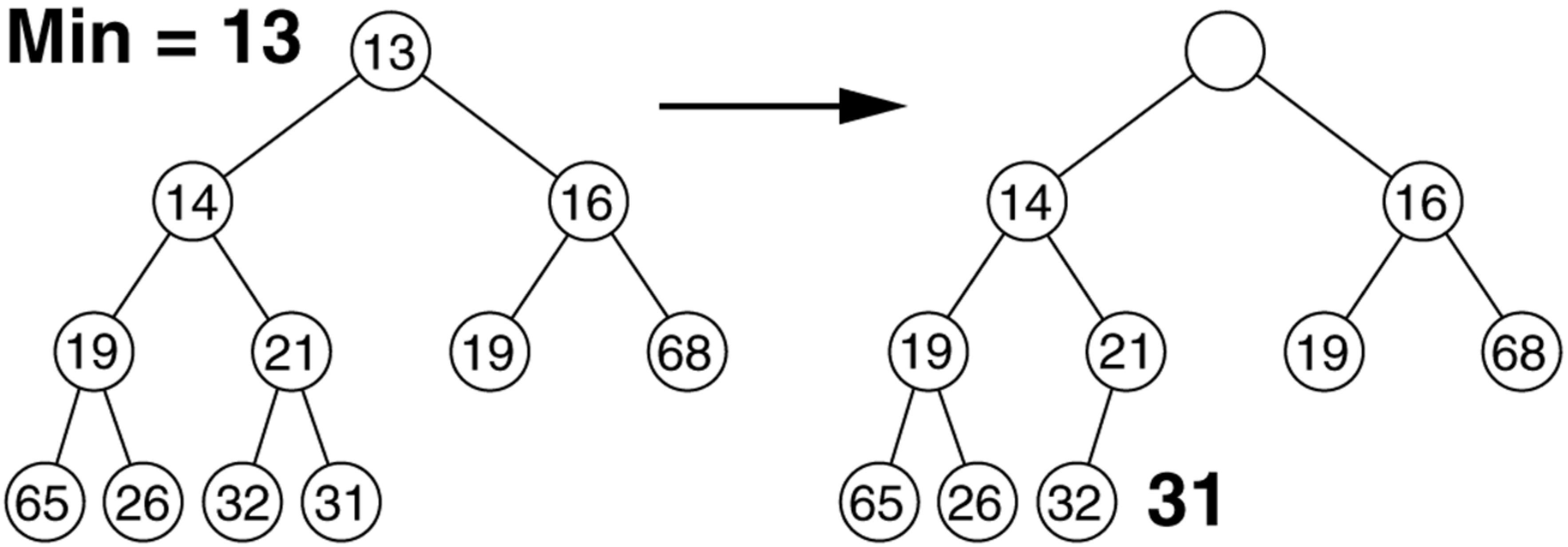


Figure 21.11
The next two steps in the deleteMin operation

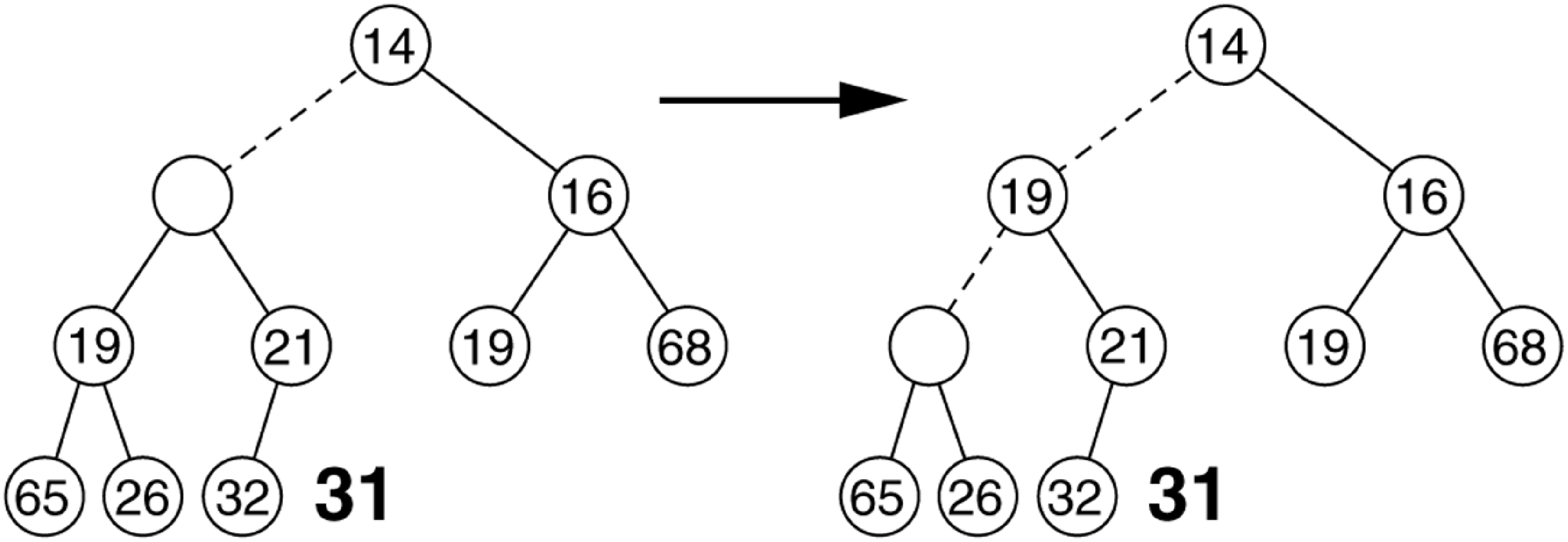
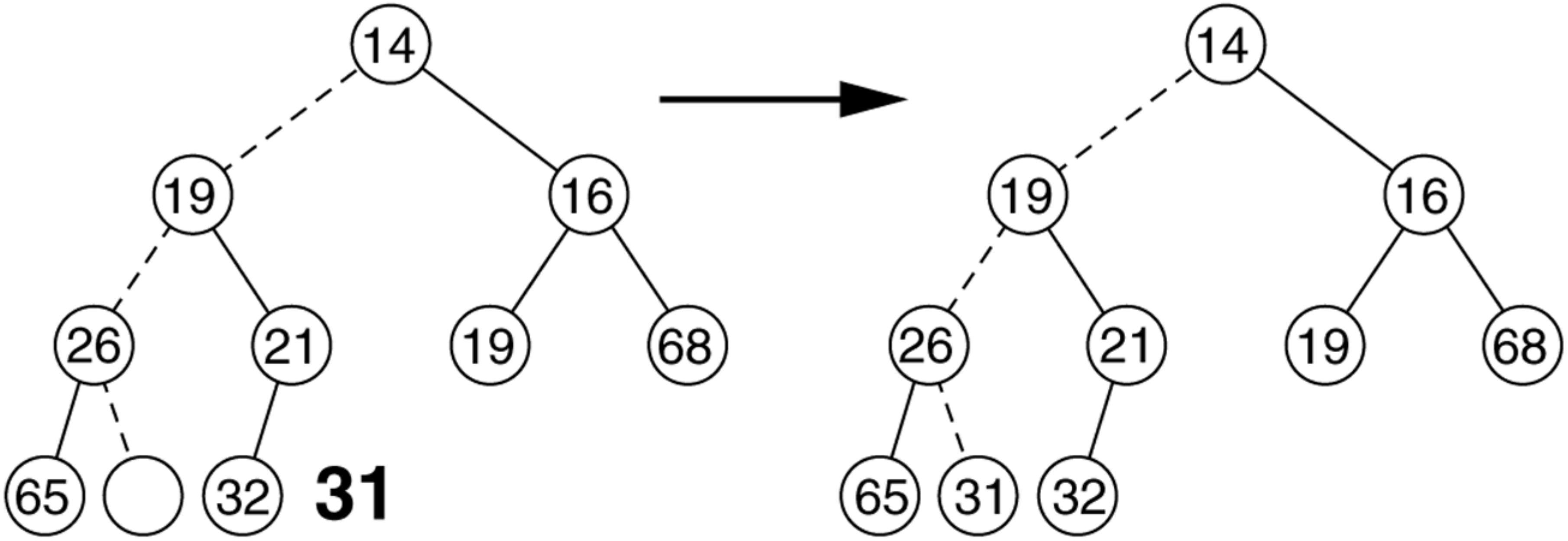


Figure 21.12

The last two steps in the deleteMin operation



deleteMin procedure

```
// Remove the smallest item from the priority queue.
// Throw UnderflowException if empty.
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin( )
{
    if( isEmpty( ) )
        throw UnderflowException( );

    array[ 1 ] = array[ theSize-- ];
    percolateDown( 1 );
}
```

```
// Internal method to percolate down in the heap.
// hole is the index at which the percolate begins.
template <class Comparable>
void BinaryHeap<Comparable>::percolateDown( int hole )
{
  int child;
  Comparable tmp = array[ hole ];

  for( ; hole * 2 <= theSize; hole = child )
  {
    child = hole * 2;
    if( child != theSize && array[child + 1] < array[child])
       child++;
    if( array[ child ] < tmp )
       array[ hole ] = array[ child ];
    else
       break;
  }
  array[ hole ] = tmp;
}
```

Building a Heap

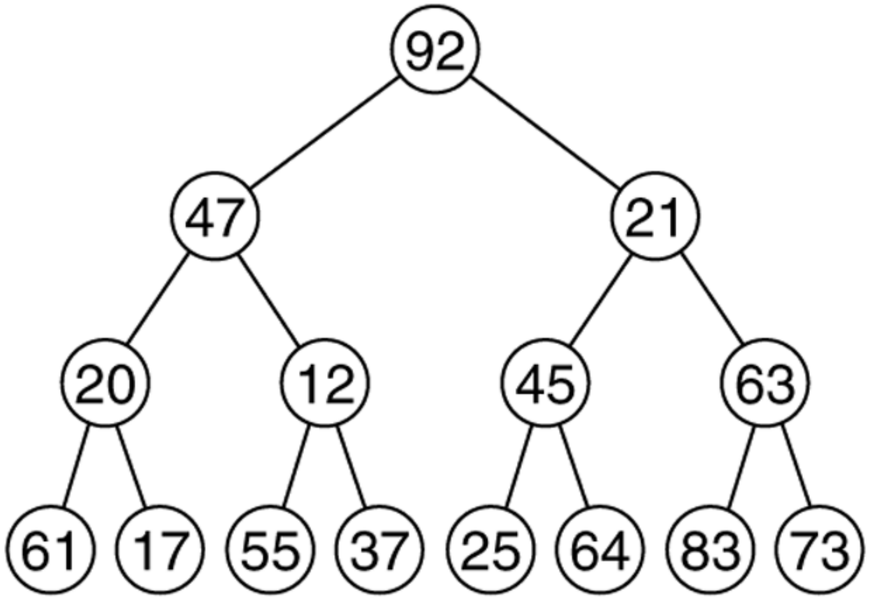
- Take as input *N* items and place them into an empty heap.
- Obviously this can be done with *N* successive inserts: *O(NlogN)* worst case.
- However `buildHeap` operation can be done in linear time (*O(N)*) by applying a percolate down routine to nodes in reverse level order.

buildHeap method

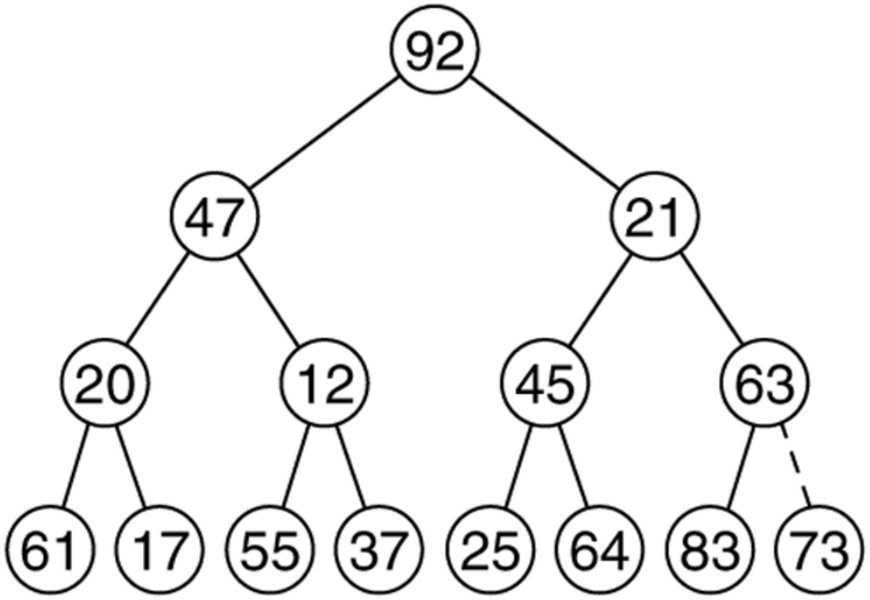
```
// Establish heap-order property from an arbitrary
// arrangement of items. Runs in linear time.
template <class Comparable>
void BinaryHeap<Comparable>::buildHeap( )
{
    for( int i = theSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

Figure 21.17

Implementation of the linear-time buildHeap method



(a) Initial heap

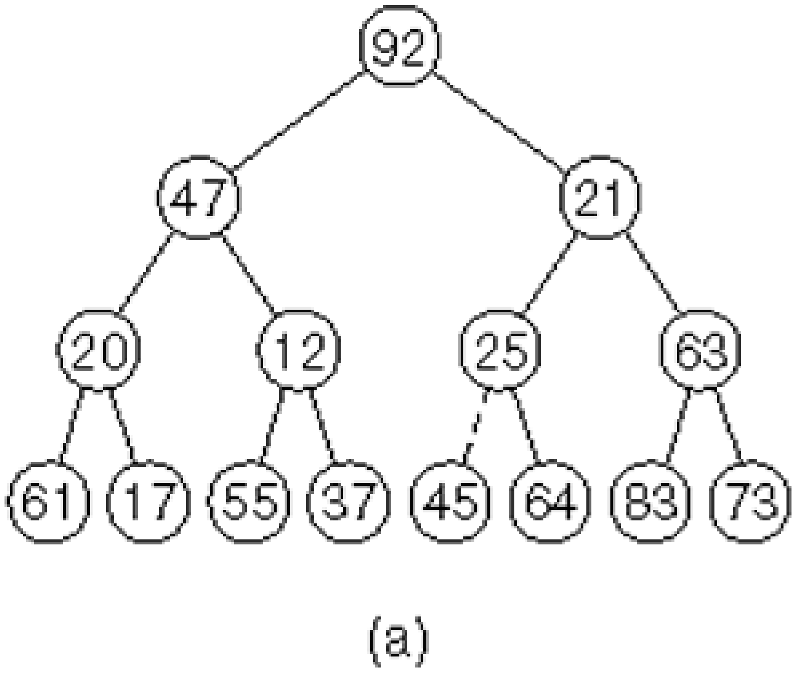


(b)

After percolatedown(7)

Figure 21.18

(a) After percolateDown(6); (b) after percolateDown(5)



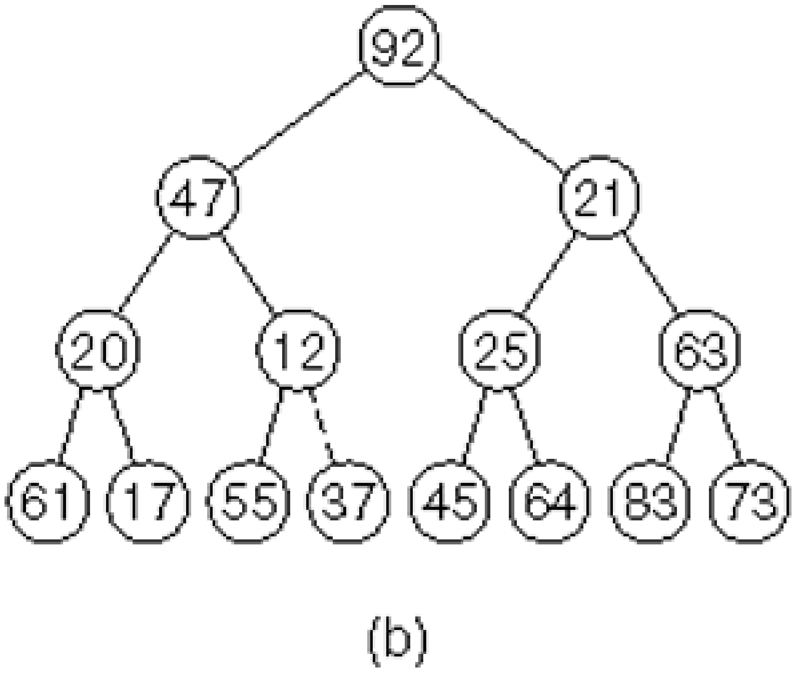


Figure 21.19

(a) After percolateDown(4); (b) after percolateDown(3)

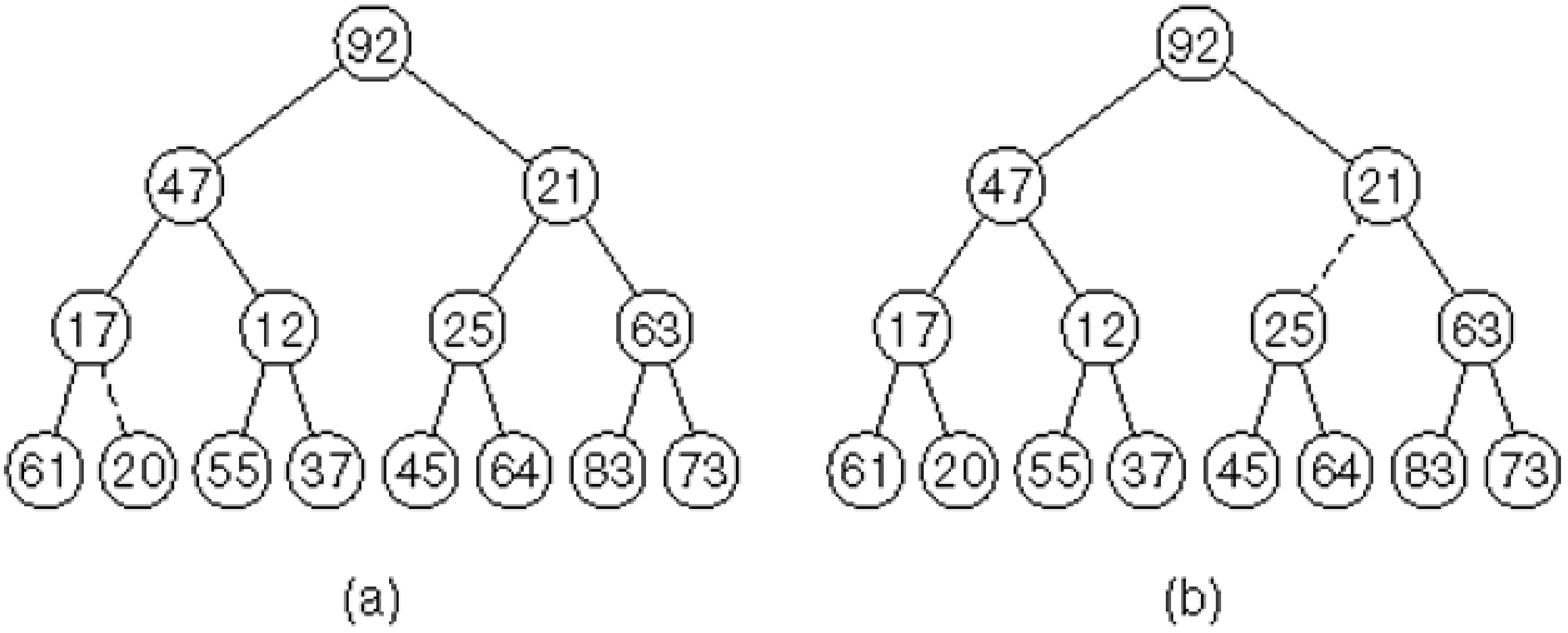
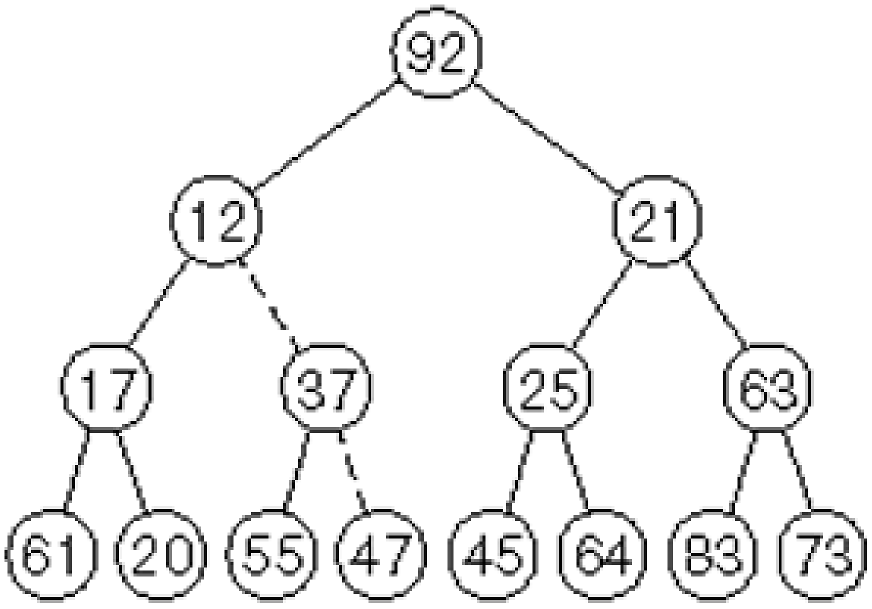
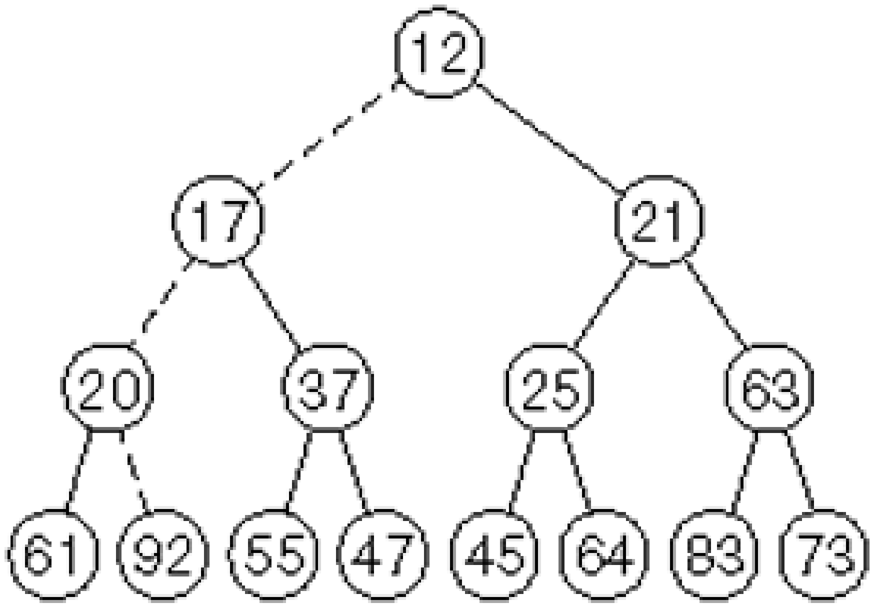


Figure 21.20

(a) After percolateDown(2); (b) after percolateDown(1) and buildHeap terminates



(a)



(b)

Analysis of **`buildHeap`**

- The linear time bound of `buildHeap`, can be shown by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines.
- For the perfect binary tree of height h containing $N = 2^{h+1} – 1$ nodes, the sum of the heights of the nodes is $N – H – 1$.
- Thus it is $O(N)$.

Heapsort

- The priority queue can be used to sort *N* items by
  - inserting every item into a binary heap and
  - extracting every item by calling `deleteMin` *N* times, thus sorting the result.
- An algorithm based on this idea is *heapsort*.
- It is an *O(N logN)* worst-case sorting algorithm.

Heapsort

- The main problem with this algorithm is that it uses an extra array for the items exiting the heap.
- We can avoid this problem as follows:
  - After each deleteMin, the heap shrinks by 1.
  - Thus the cell that was last in the heap can be used to store the element that was just deleted.
  - Using this strategy, after the last deleteMin, the array will contain all elements in *decreasing* order.
- If we want them in *increasing* order we must use a *max heap*.

Figure 21.25

Max heap after the buildHeap phase for the input sequence 59,36,58,21,41,97,31,16,26,53

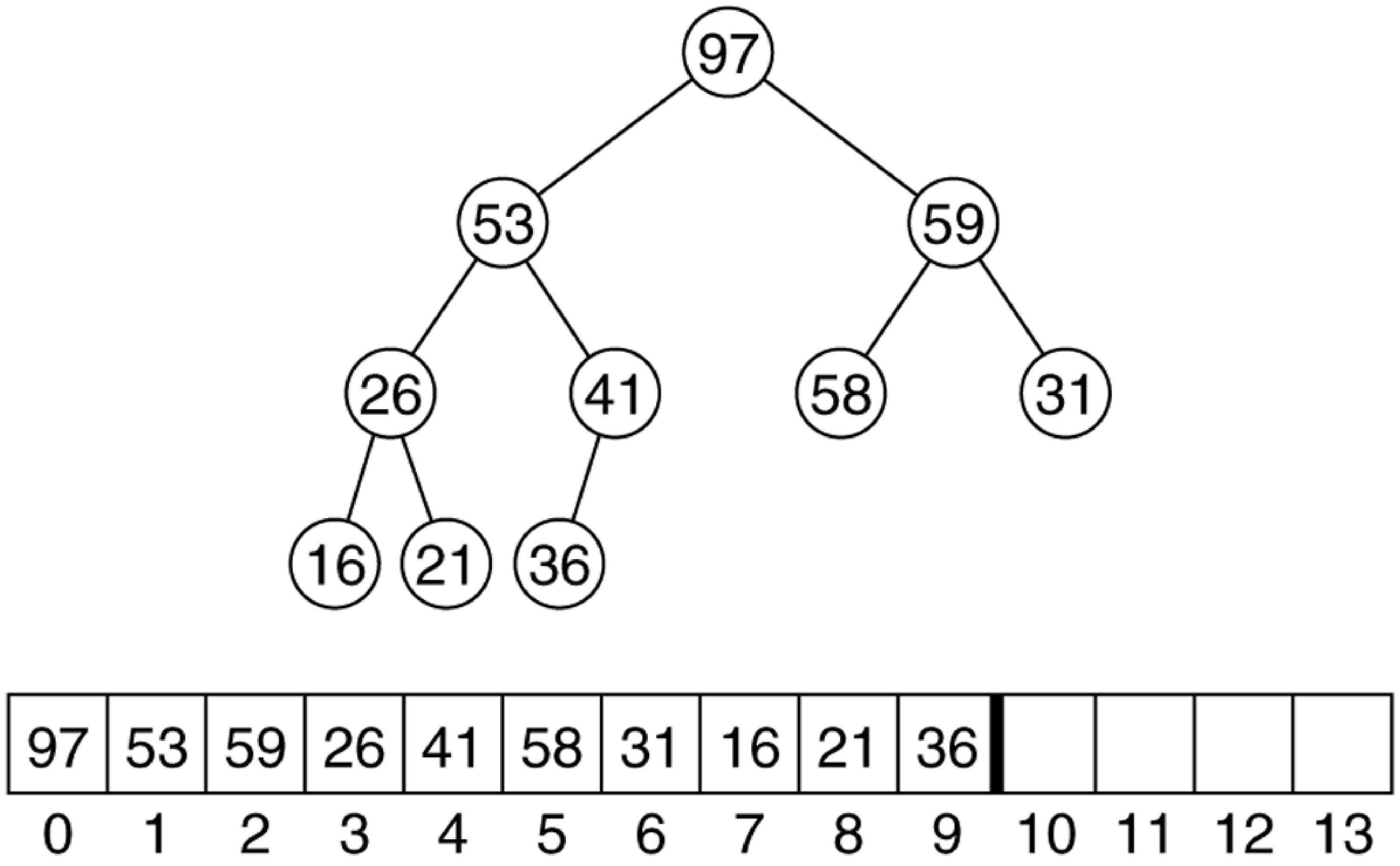


Figure 21.26

Heap after the first deleteMax operation

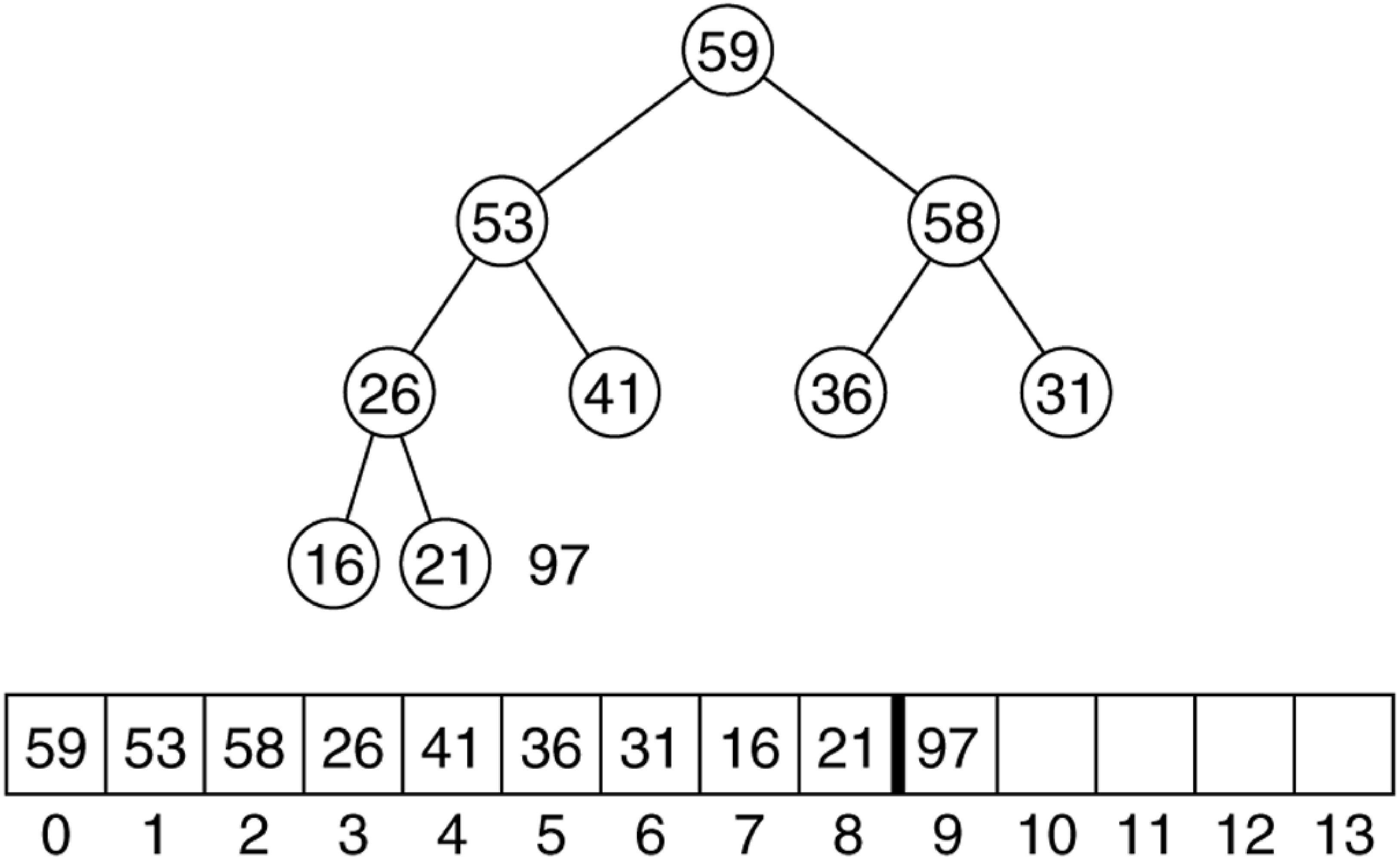
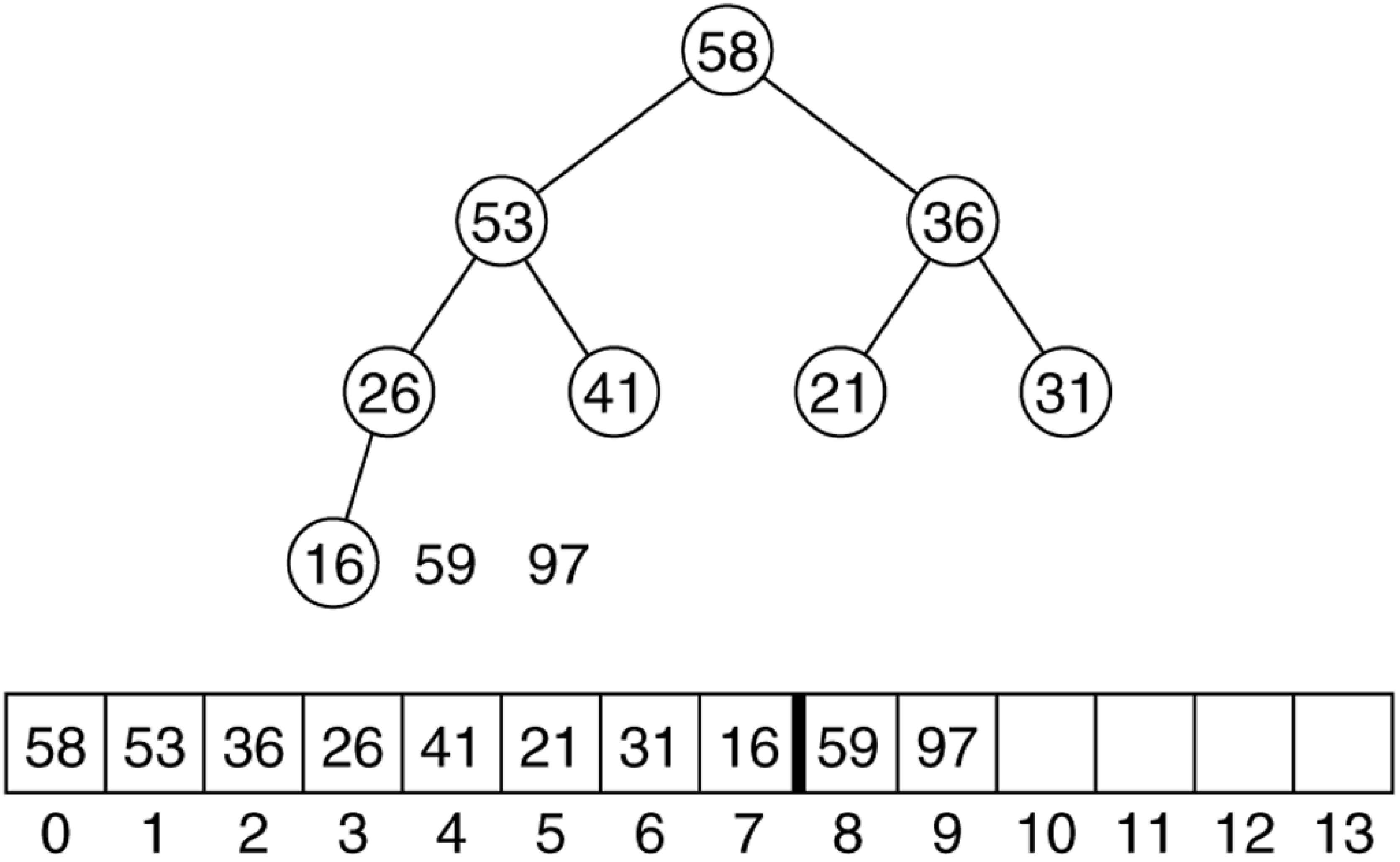


Figure 21.27

Heap after the second deleteMax operation



Implementation

- In the implementation of heapsort, the ADT BinaryHeap is not used.
  - Everything is done in an array.
- The root is stored in position 0.
- Thus there are some minor changes in the code:
  - Since we use max heap, the logic of comparisons is changed from > to <.
  - For a node in position *i*, the parent is in (*i*-1)/2, the left child is in 2*i*+1 and right child is next to left child.
  - Percolating down needs the current heap size which is lowered by 1 at every deletion.

The `heapsort` routine

```
/**
 * Standard heapsort.
 */
template <class Comparable>
void heapsort( vector<Comparable> & a )
{
  for( int i = a.size( ) / 2; i >= 0; i-- )  /* buildHeap */
    percDown( a, i, a.size( ) );
  for( int j = a.size( ) - 1; j > 0; j-- )
  {
     swap( a[ 0 ], a[ j ] );                          /* deleteMax */
     percDown( a, 0, j );
  }
}
```

Analysis of Heapsort

- It is an O(N log N) algorithm.
  - First phase: Build heap O(N)
  - Second phase: N deleteMax operations: O(NlogN).
- Detailed analysis shows that, the average case for heapsort is poorer than quick sort.
  - Quicksort's worst case however is far worse.
- An average case analysis of heapsort is very complicated, but empirical studies show that there is little difference between the average and worst cases.
  - Heapsort usually takes about twice as long as quicksort.
  - Heapsort therefore should be regarded as something of an insurance policy:
  - On average, it is more costly, but it avoids the possibility of $O(N^2)$.