WASHINGTON STATE
UNIVERSITY

CPTS 223 Advanced Data Structure C/C++

Heaps

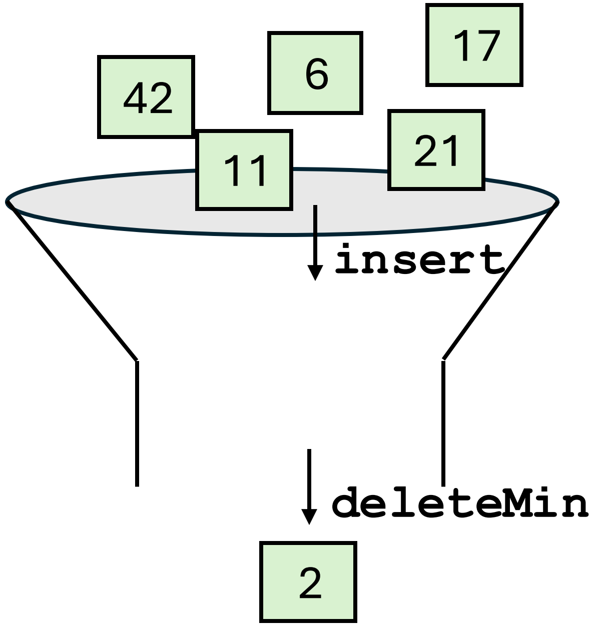
Motivation

- Queues are a standard mechanism for ordering tasks on a first-come, first-served basis
- However, some tasks may be more important or timely than others (higher priority)
- Priority queues
  - Store tasks using a partial ordering based on priority
  - Ensure highest priority task at head of queue
- Heaps are the underlying data structure of priority queues

Applications

- Operating system scheduling
  - Process jobs by priority
- Graph algorithms
  - Find shortest path
- Event simulation
  - Look up next event to happen

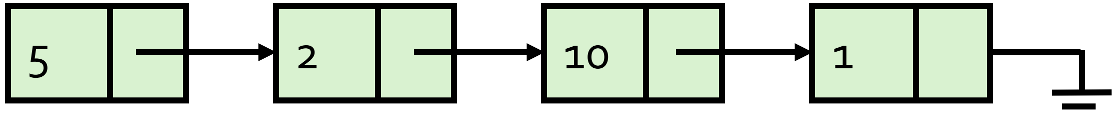
Using priority queues



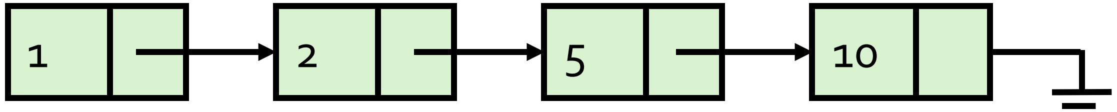
- Main operations
  - **`insert`** (i.e., enqueue)
    - Dynamic insert
    - specification of a priority level (0-high, 1,2.. Low)
  - **`deleteMin`** (i.e., dequeue)
    - Finds the current minimum element (read: “highest priority”) in the queue, deletes it from the queue, and returns it
- Performance goal is for operations to be “fast”

Priority queues: simple options

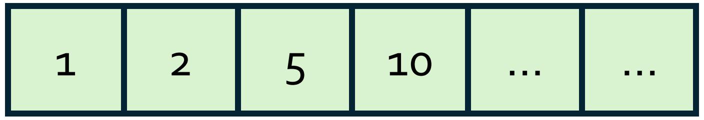
- Unordered linked list
  - O(1) insert
  - O(n) deleteMin



- Ordered linked list
  - O(n) insert
  - O(1) deleteMin



- Ordered array
  - O(lg(n) + n) insert
  - O(n) deleteMin



Priority queues: simple options

- BST
  - Θ(lg(n)) average-case for insert and deleteMin
  - O(n) worse-case for insert and deleteMin
- Self-based BST
  - O(lg(n)) worse-case for findMin, insert and delteMin
  - Complicated implementation
    - AVL trees: height maintenance for every node
    - R-B trees: complicated implementations for many cases
      - Bottom-up procedure
      - Top-down procedure

Search, insert and delete any data cost O(lg(n)): overkill

Can we do better for findMin and deleteMin?

Time complexity per operation

	findMin	insert	deleteMin	merge
Binary heap	O(1)	O(log(n)) worst-case O(1) amortized for buildHeap	O(log(n))	O(n)
Leftist heap	O(1)	O(log(n))	O(log(n))	O(log(n))
Skew heap	O(1)	O(log(n))	O(log(n))	O(log(n))
Binomial heap	O(1)	O(log(n)) worst-case O(1) amortized for sequence of n inserts	O(log(n))	O(log(n))
Fibonacci heap	O(1)	O(1)	O(log(n))	O(1)

Binary heap

- A binary heap is a binary tree with two properties
    - Structure property
    - Heap-order property

Binary heap: structure property

- A binary heap is a complete binary tree
  - Each level (except possibly the bottom most level) is completely filled
  - The bottom most level may be partially filled (from left to right)
- Height of a complete binary tree with N elements is $\lfloor \mathrm{log\_2}(N) \rfloor$

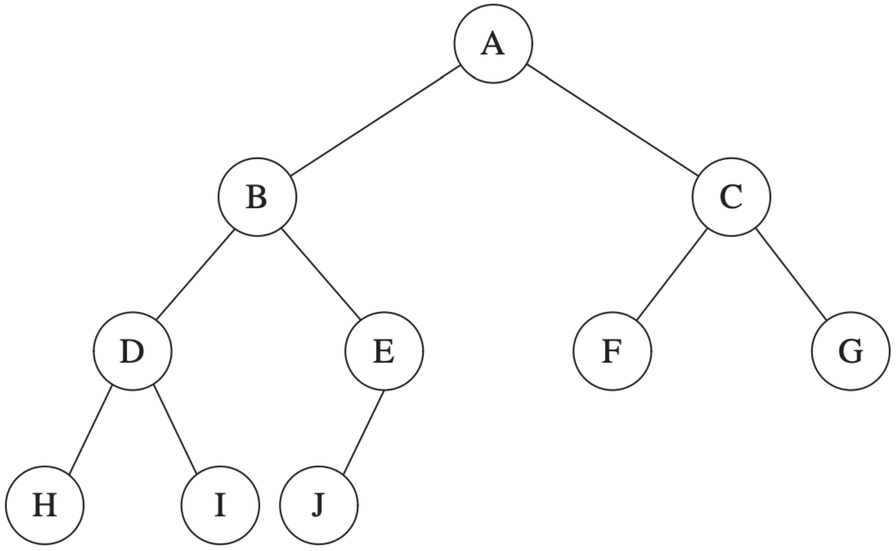


Figure 6.2 A complete binary tree

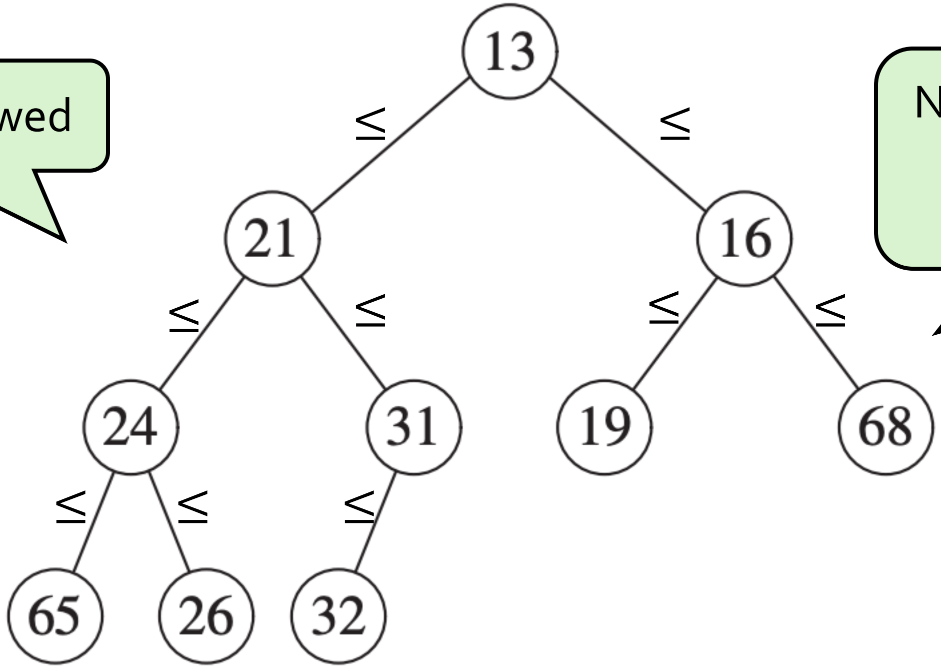
With a simple array representation

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

Binary heap: heap-order property

- Heap-order property (for a “MinHeap”)
  - For every node X, key(parent(X)) ≤ key(X)
  - Except root node, which has no parent
- Thus, minimum key always at root
- Alternatively, for a “MaxHeap”, always keep the maximum key at the root
- Insert and deleteMin must maintain heap-order property

Binary heap: heap-order property



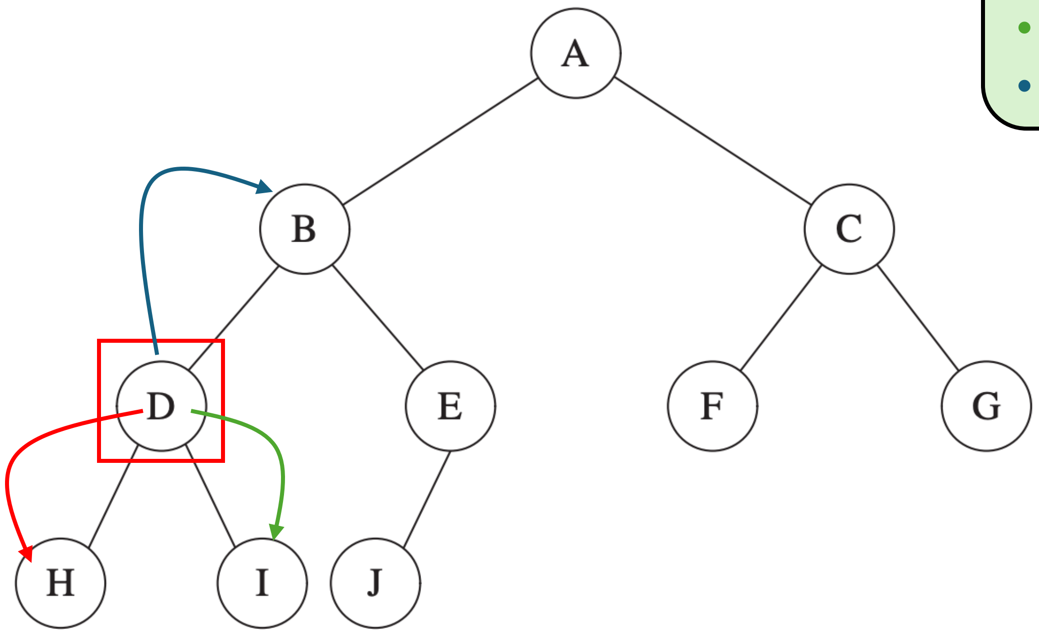
Duplicates allowed

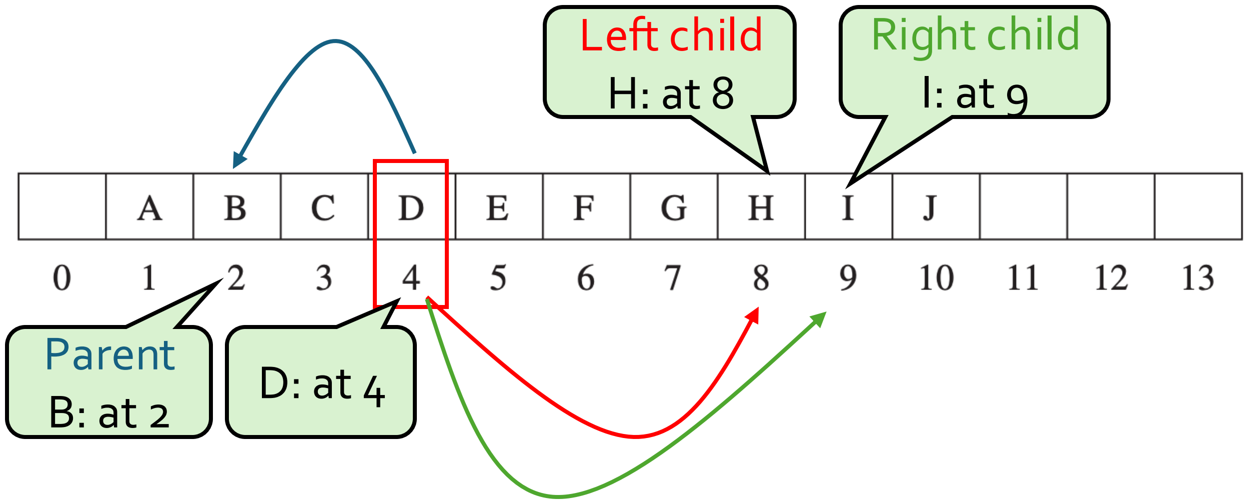
No order implied for elements that do not share ancestor-descendant relation

Implementation: arrays

Given element at position i in the array

- leftChild(i) = 2i
- rightChild(i) = 2i + 1
- parent(i) = i/2





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

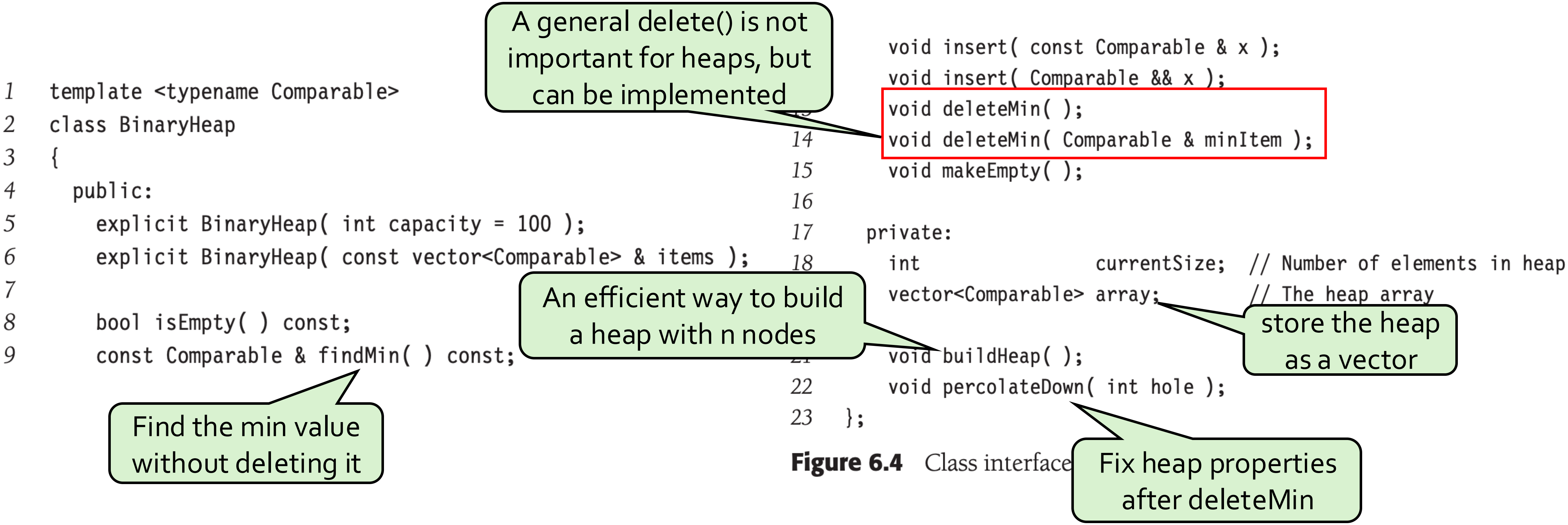
Left child
H: at 8

Right child
I: at 9

Parent
B: at 2

D: at 4

Binary heap: class interface



```
template <typename Comparable>
class BinaryHeap
{
  public:
    explicit BinaryHeap( int capacity = 100 );
    explicit BinaryHeap( const vector<Comparable> & items );

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

  private:
    int                currentSize;  // Number of elements in heap
    vector<Comparable> array;        // The heap array

    void buildHeap( );
    void percolateDown( int hole );
};
```

Figure 6.4 Class interface

A general delete() is not important for heaps, but can be implemented

Find the min value without deleting it

An efficient way to build a heap with n nodes

store the heap as a vector

Fix heap properties after deleteMin

Binary heap: insert

- Insert new element into the heap at the next available slot (“hole”)
  - According to maintaining a complete binary tree
- Then, “percolate” the element up the heap while heap-order property not satisfied

Binary heap: insert

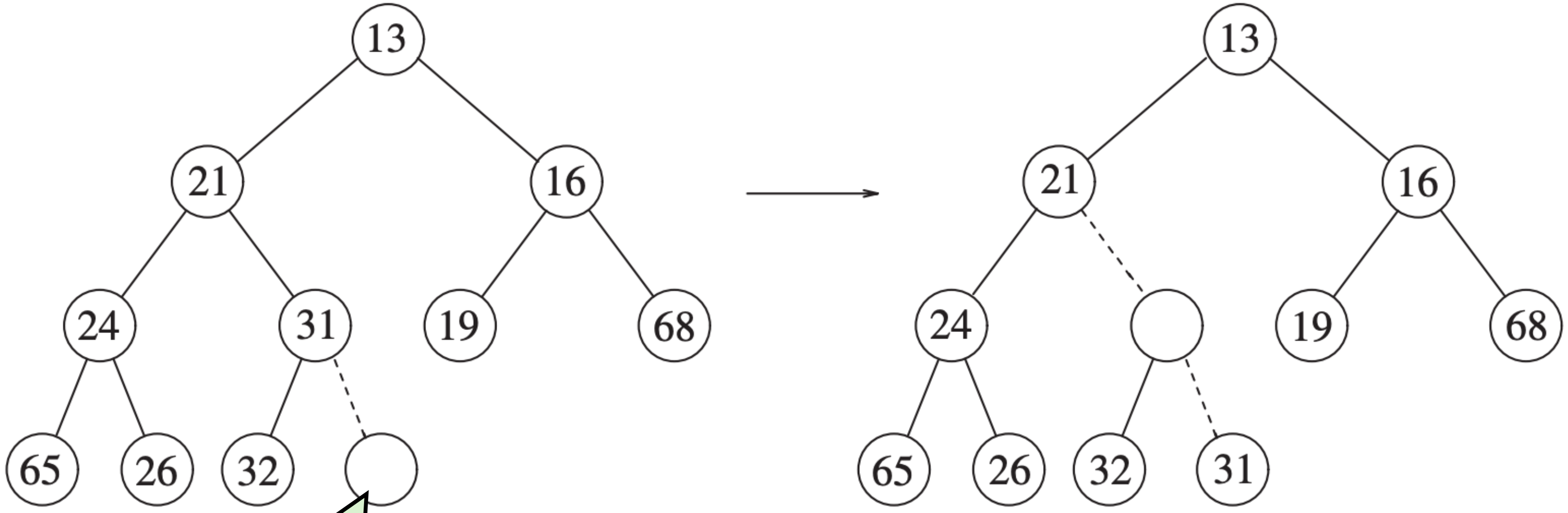


Figure 6.6 A t to insert 14 creating the hole, and bubbling the hole up

Insert new element into the heap at the next available slot ("hole")

Structure property ✓

Binary heap: insert

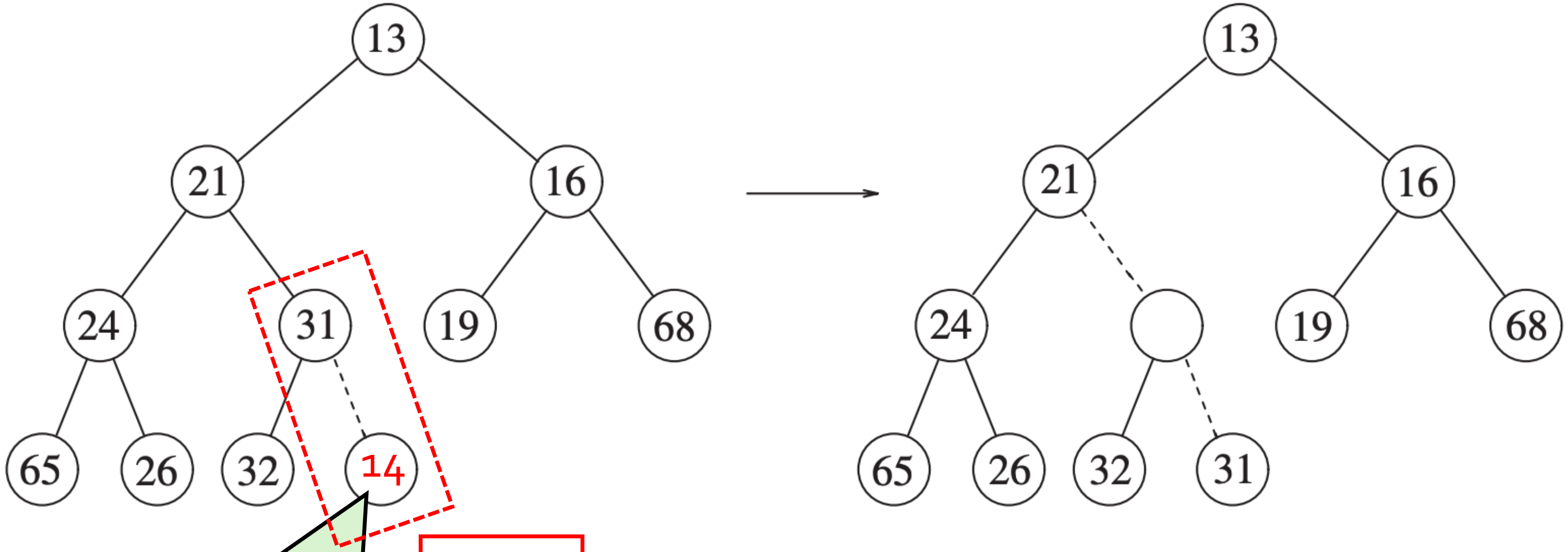
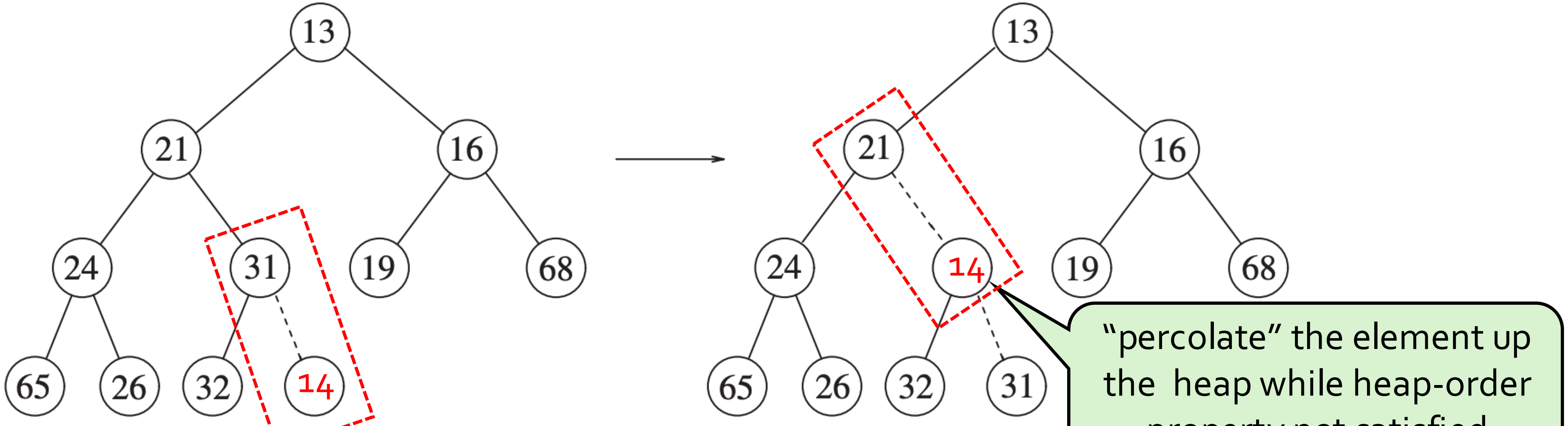


Figure 6.6 A t to insert 14: creating the hole, and bubbling the hole up

Insert new element into the heap at the next available slot ("hole")

Heap-order property ✗

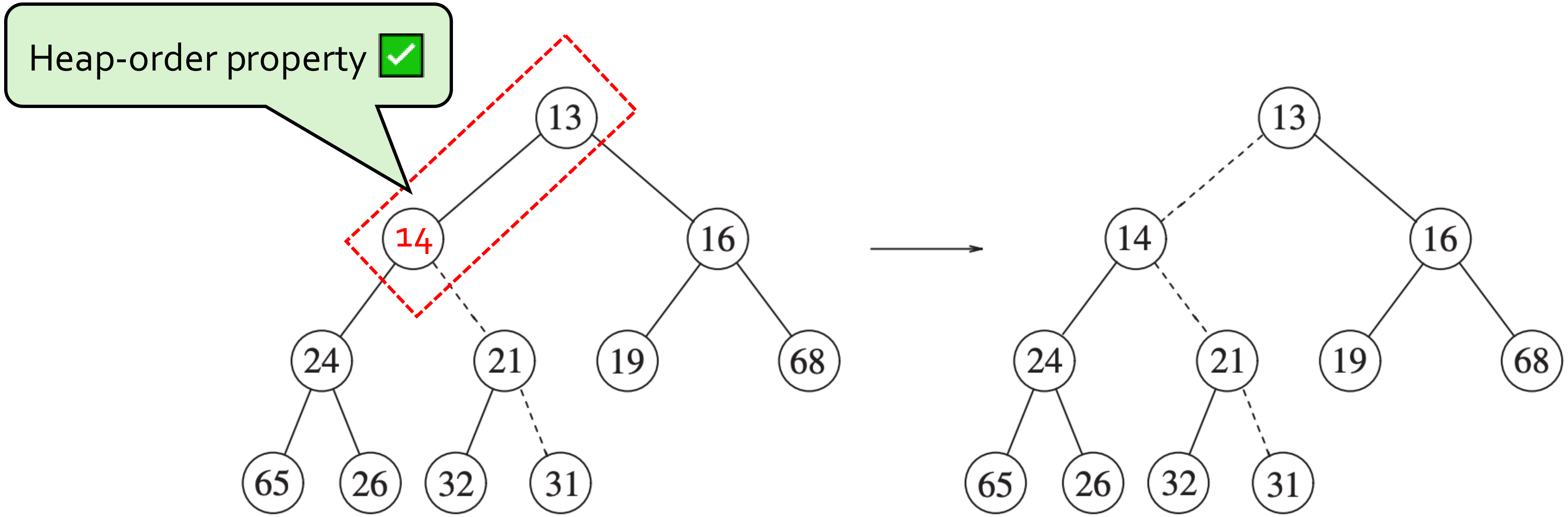
Binary heap: insert



"percolate" the element up the heap while heap-order property not satisfied

Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

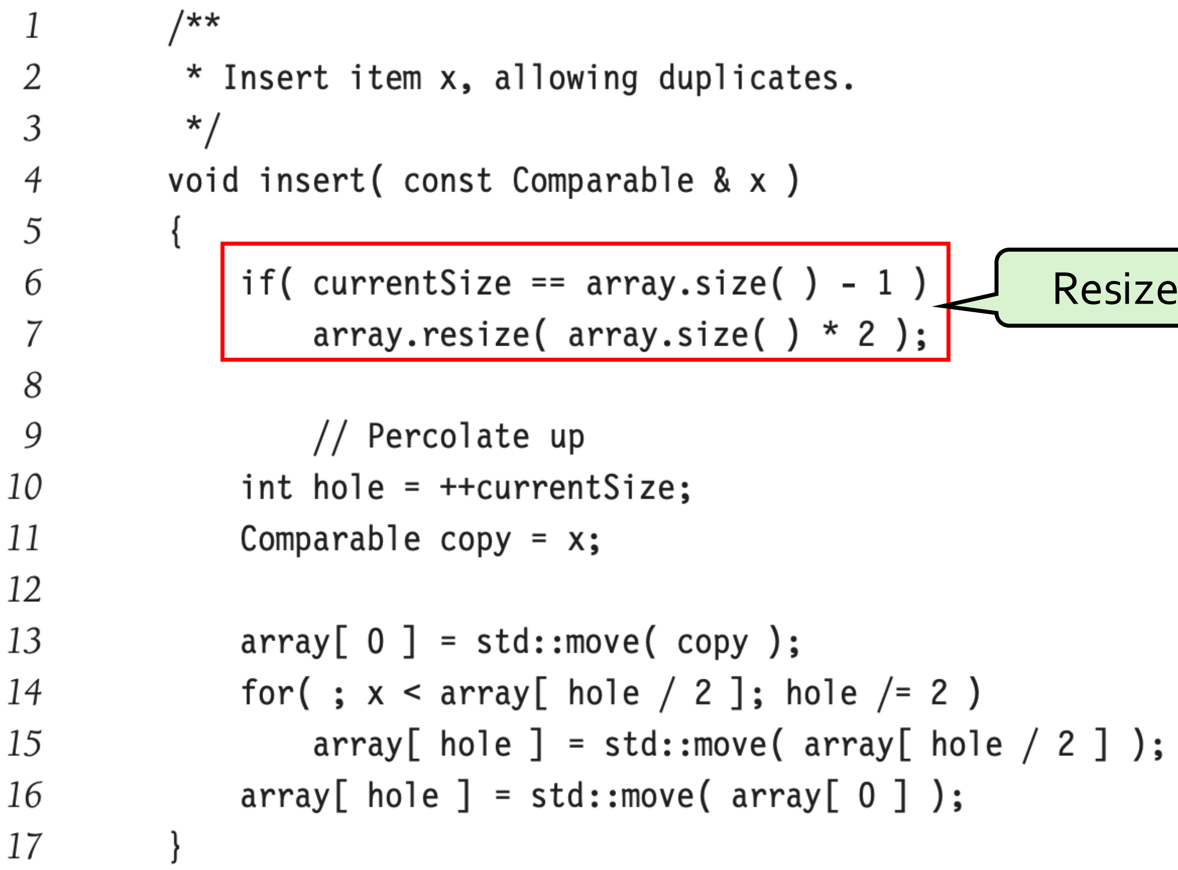
Binary heap: insert



Heap-order property ✓

Figure 6.7 The remaining two steps to insert 14 in previous heap

Heap insert: implementation

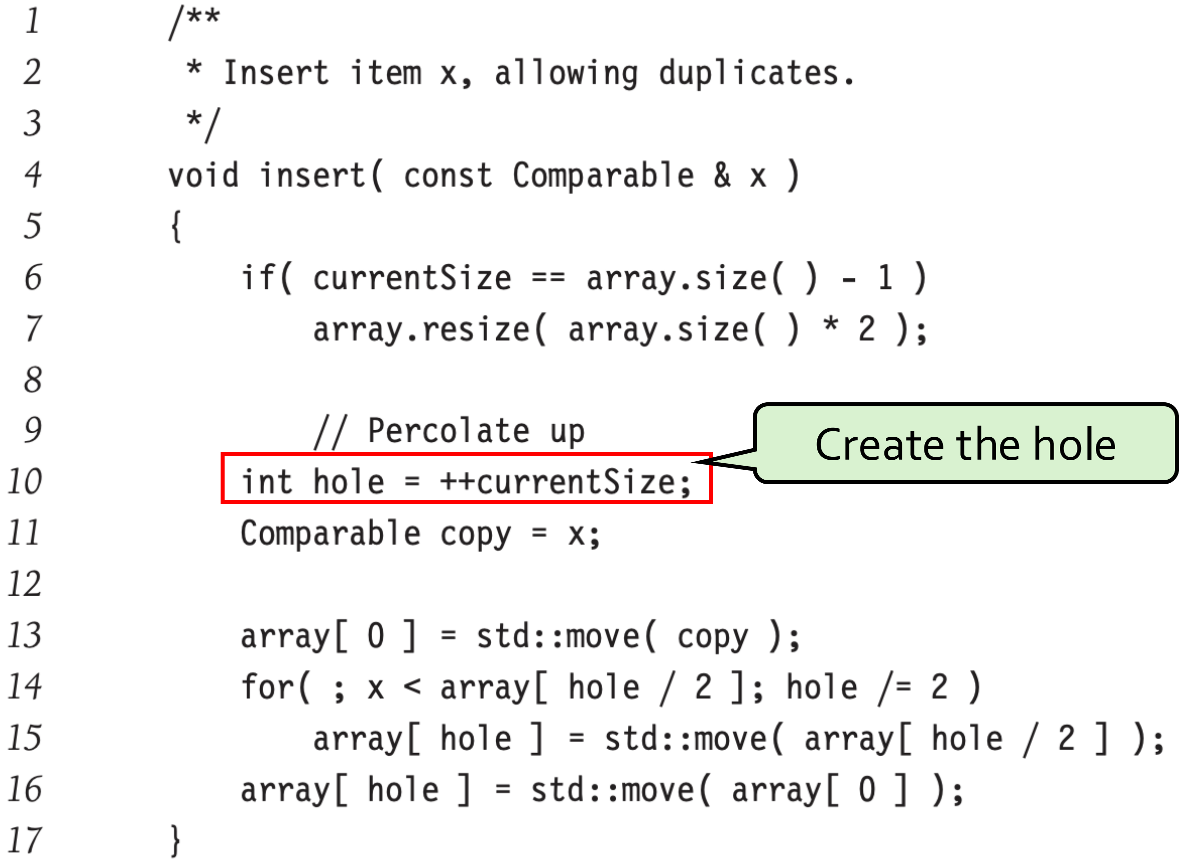


```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );

        // Percolate up
    int hole = ++currentSize;
    Comparable copy = x;

    array[ 0 ] = std::move( copy );
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = std::move( array[ hole / 2 ] );
    array[ hole ] = std::move( array[ 0 ] );
}
```

Resize the array

Figure 6.8 Procedure to insert into a binary heap

Heap insert: implementation

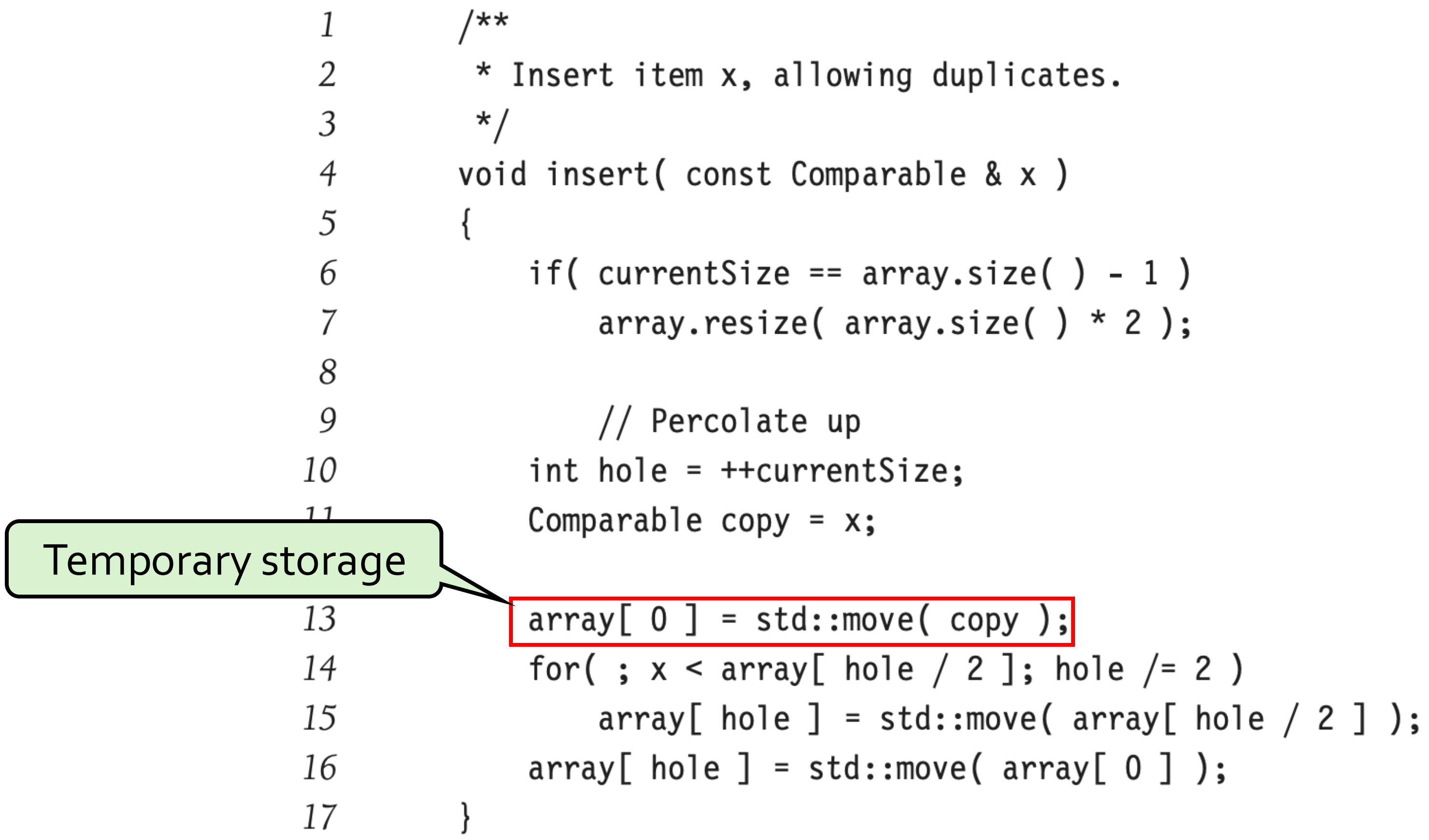


```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );

        // Percolate up
    int hole = ++currentSize;
    Comparable copy = x;

    array[ 0 ] = std::move( copy );
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = std::move( array[ hole / 2 ] );
    array[ hole ] = std::move( array[ 0 ] );
}
```

Create the hole

Figure 6.8 Procedure to insert into a binary heap

Heap insert: implementation

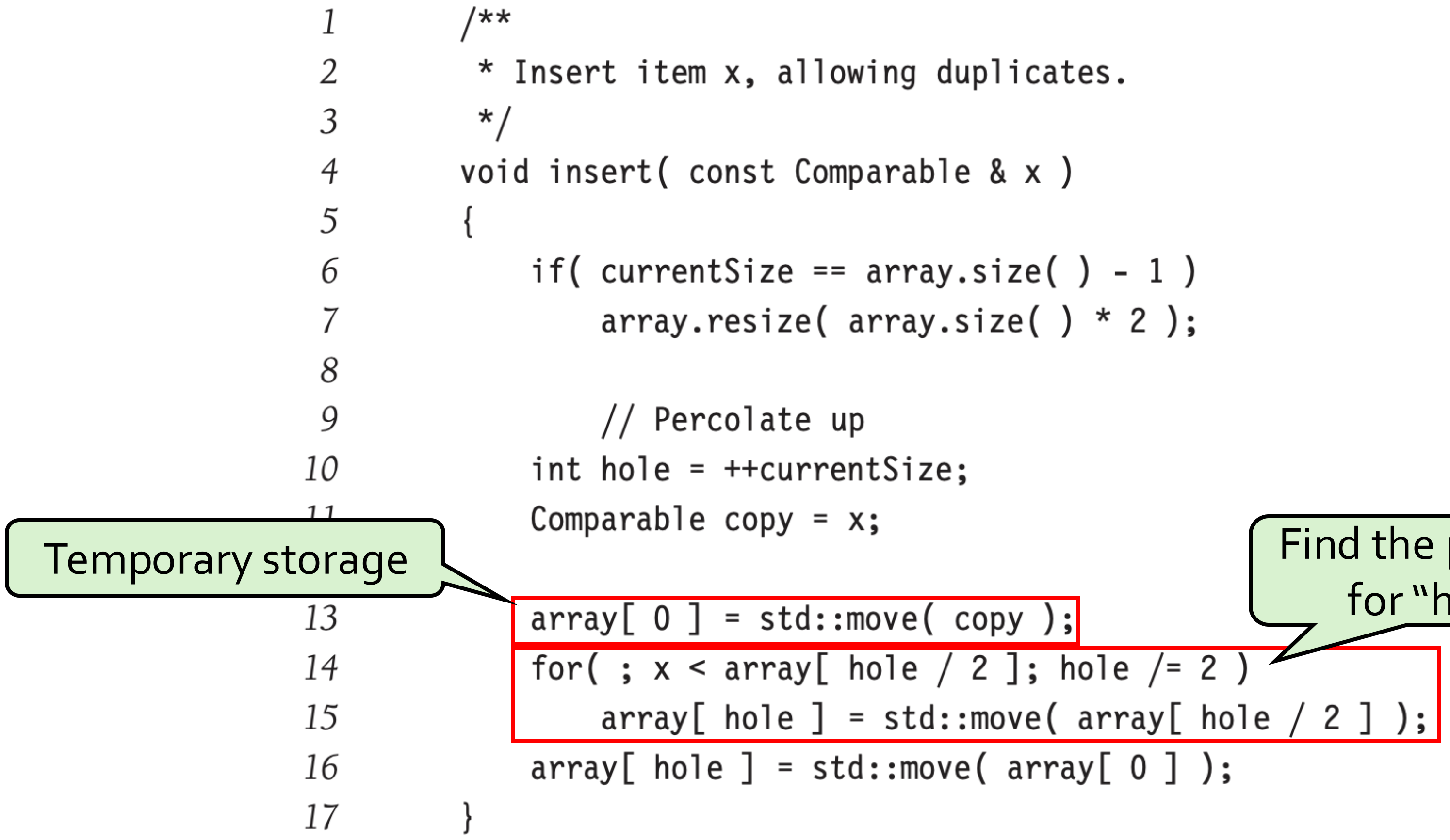


```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );

        // Percolate up
    int hole = ++currentSize;
    Comparable copy = x;

    array[ 0 ] = std::move( copy );
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = std::move( array[ hole / 2 ] );
    array[ hole ] = std::move( array[ 0 ] );
}
```

Temporary storage

Figure 6.8 Procedure to insert into a binary heap

Heap insert: implementation



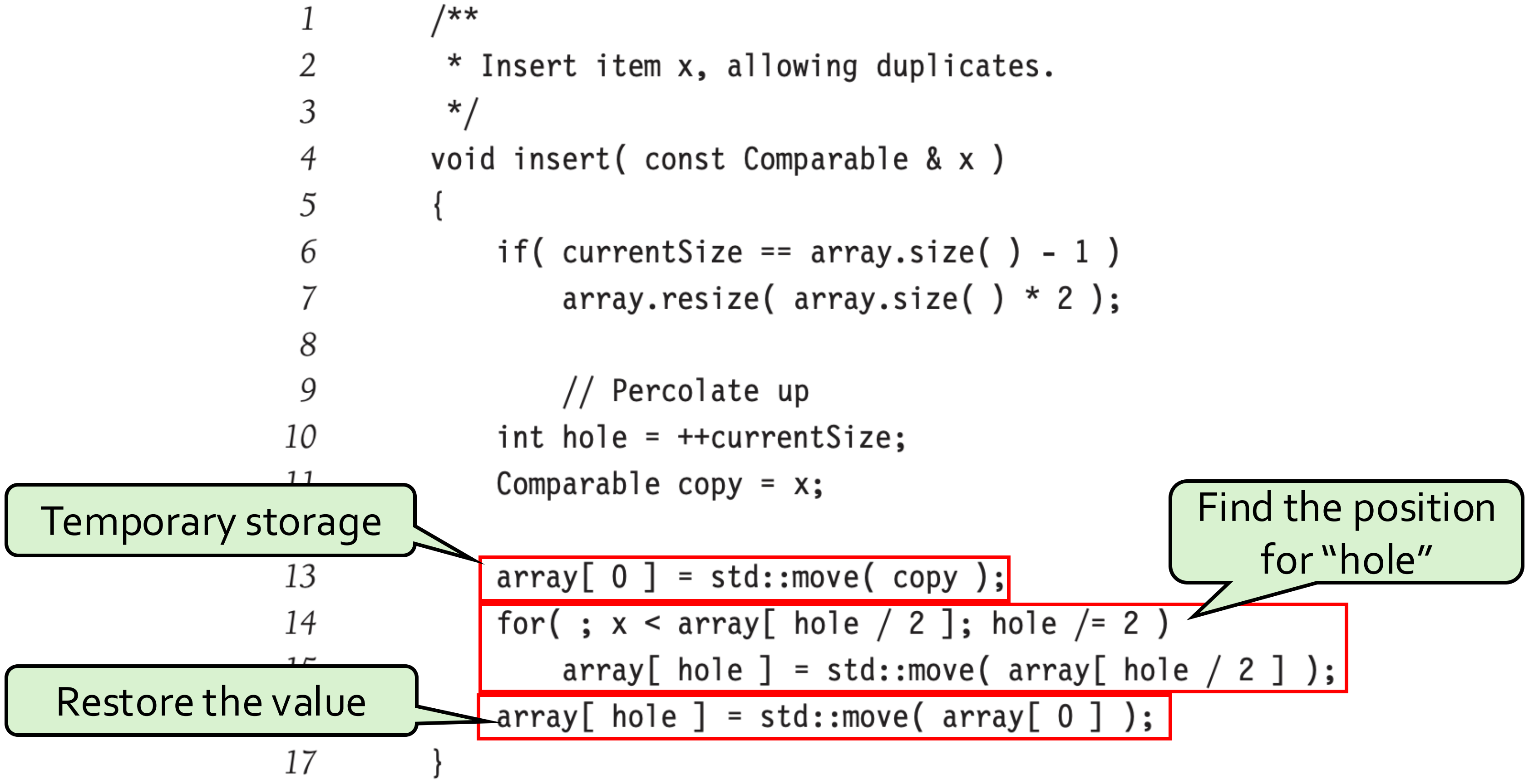
```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );

        // Percolate up
    int hole = ++currentSize;
    Comparable copy = x;

    array[ 0 ] = std::move( copy );
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = std::move( array[ hole / 2 ] );
    array[ hole ] = std::move( array[ 0 ] );
}
```

Temporary storage

Find the position for “hole”

Figure 6.8 Procedure to insert into a binary heap

Heap insert: implementation



```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );

        // Percolate up
    int hole = ++currentSize;
    Comparable copy = x;

    array[ 0 ] = std::move( copy );
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = std::move( array[ hole / 2 ] );
    array[ hole ] = std::move( array[ 0 ] );
}
```

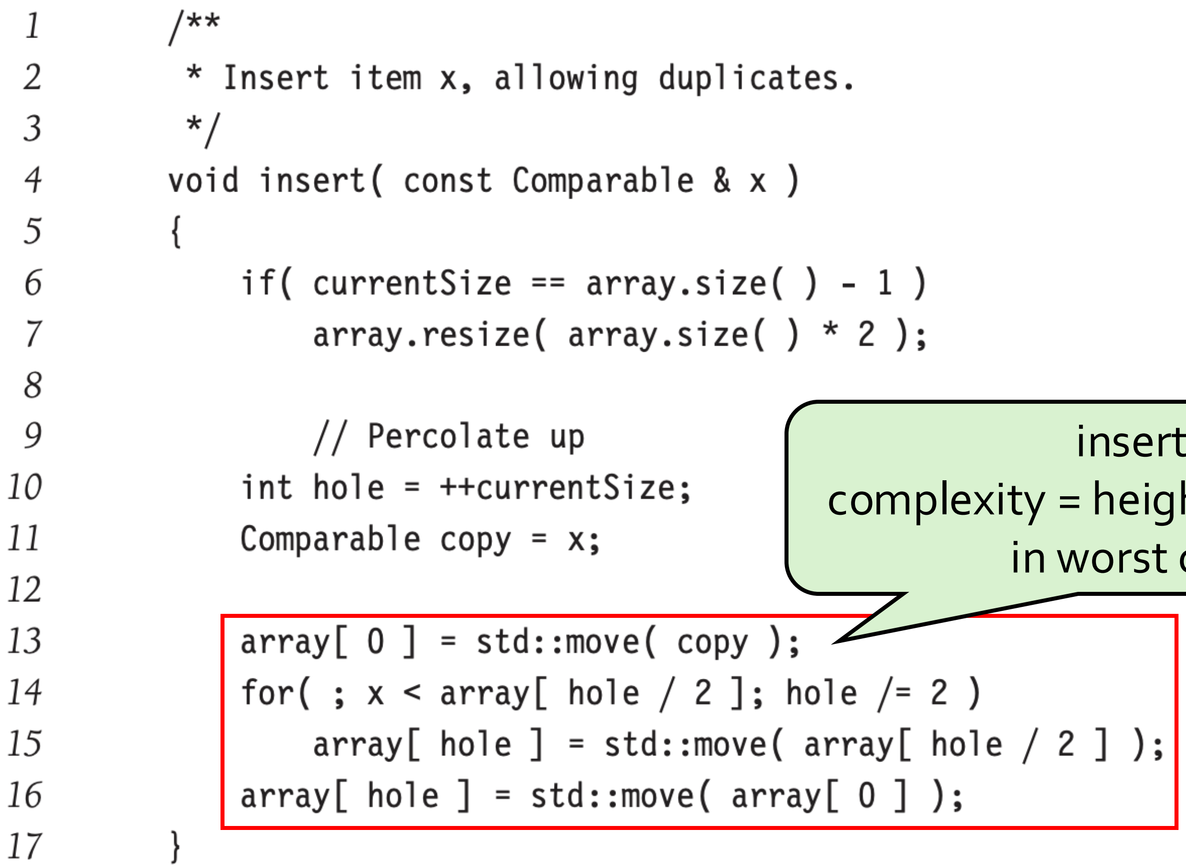
Temporary storage

Find the position for "hole"

Restore the value

Figure 6.8 Procedure to insert into a binary heap

Heap insert: implementation



```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );

        // Percolate up
    int hole = ++currentSize;
    Comparable copy = x;

    array[ 0 ] = std::move( copy );
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = std::move( array[ hole / 2 ] );
    array[ hole ] = std::move( array[ 0 ] );
}
```

insert:
complexity = height = O(log(N))
in worst case

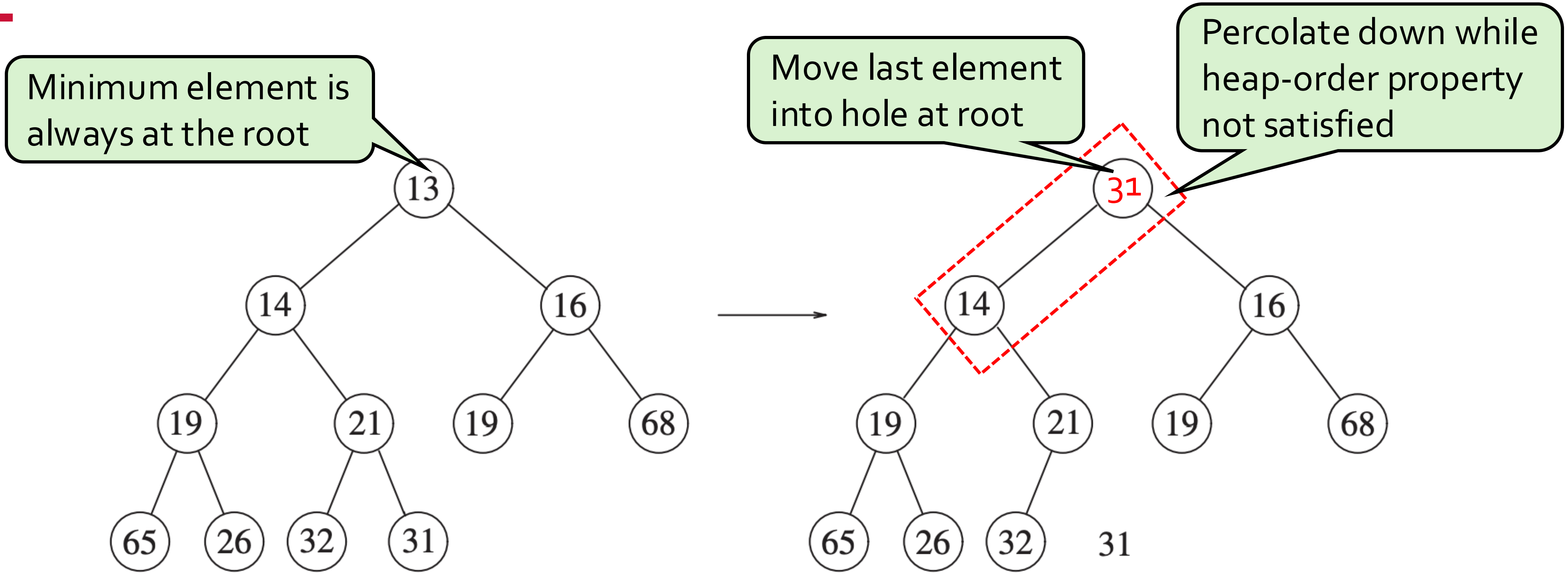
Figure 6.8 Procedure to insert into a binary heap

Heap deleteMin

So findMin requires O(1) in worst case

- Minimum element is always at the root
- Heap decreases by one in size
- Move last element into hole at root
- Percolate down while heap-order property not satisfied

Heap deleteMin



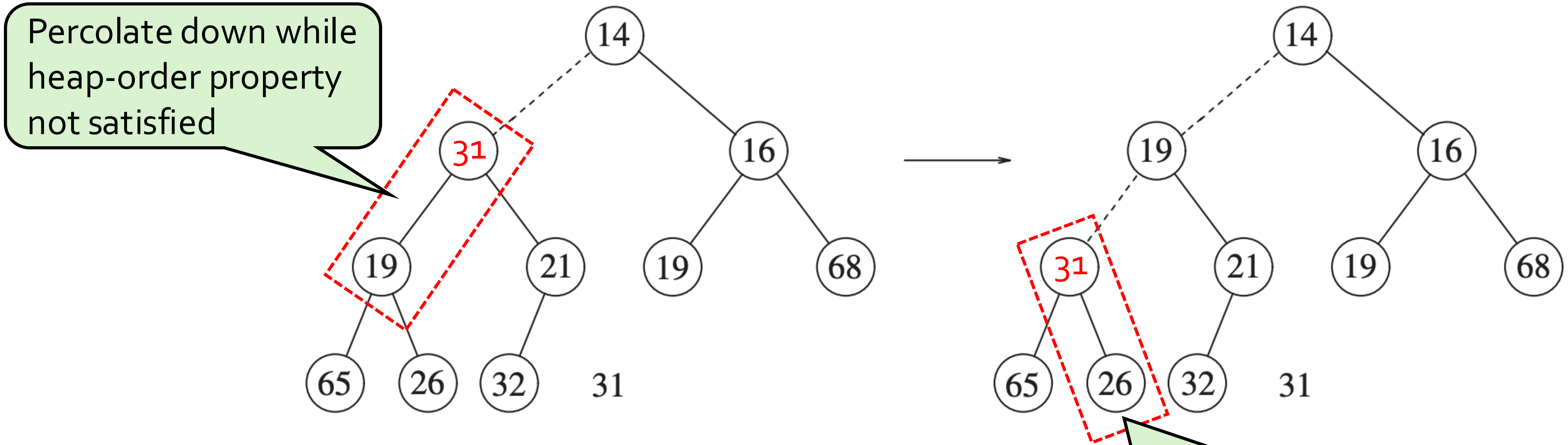
Minimum element is always at the root

Move last element into hole at root

Percolate down while heap-order property not satisfied

Figure 6.9 Creation of the hole at the root

Heap deleteMin



Percolate down while heap-order property not satisfied

Percolate down while heap-order property not satisfied

Figure 6.10 Next two steps in `deleteMin`

Heap deleteMin

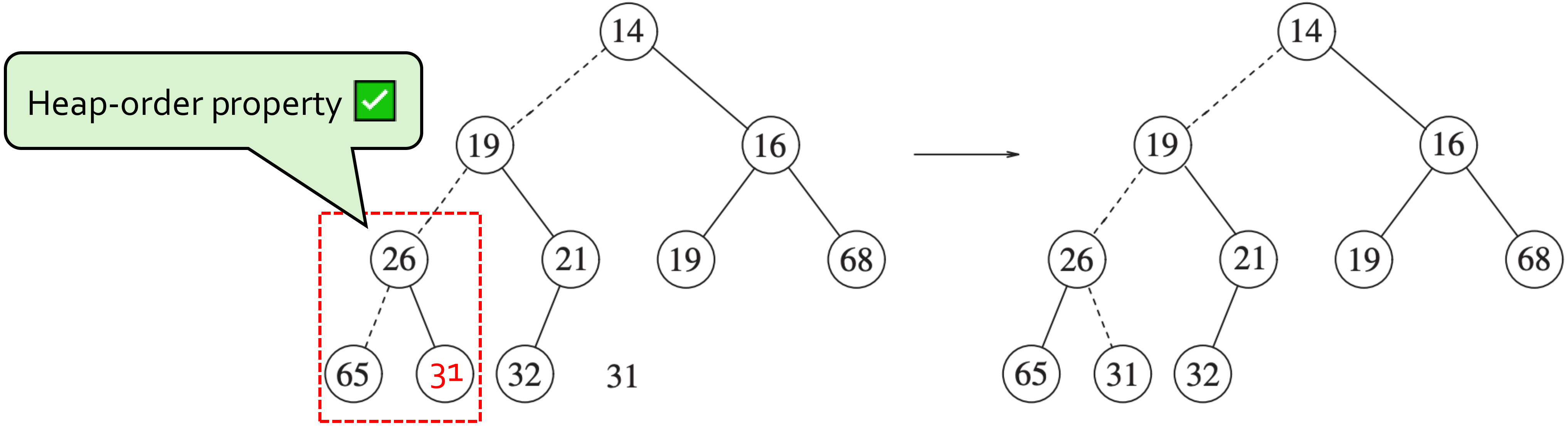

Heap-order property ✅

Figure 6.11 Last two steps in `deleteMin`

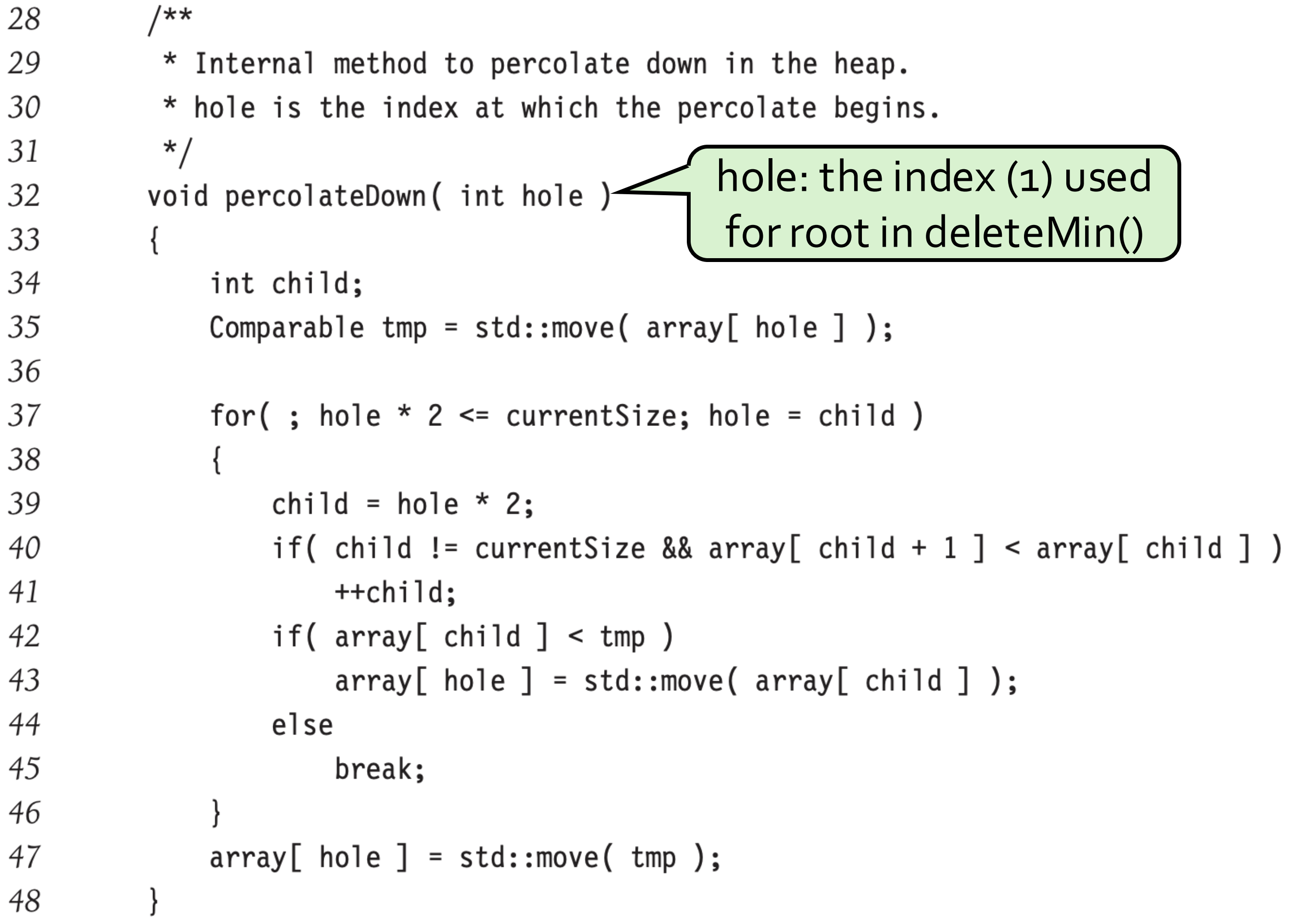
Heap deleteMin

```
/**
 * Remove the minimum item.
 * Throws UnderflowException if empty.
 */
void deleteMin( )
{
    if( isEmpty( ) )
        throw UnderflowException{ };

    array[ 1 ] = std::move( array[ currentSize-- ] );
    percolateDown( 1 );
}
```

```
/**
 * Remove the minimum item and place it in minItem.
 * Throws UnderflowException if empty.
 */
void deleteMin( Comparable & minItem )
{
    if( isEmpty( ) )
        throw UnderflowException{ };

    minItem = std::move( array[ 1 ] );
    array[ 1 ] = std::move( array[ currentSize-- ] );
    percolateDown( 1 );
}
```

Heaps

Heap deleteMin

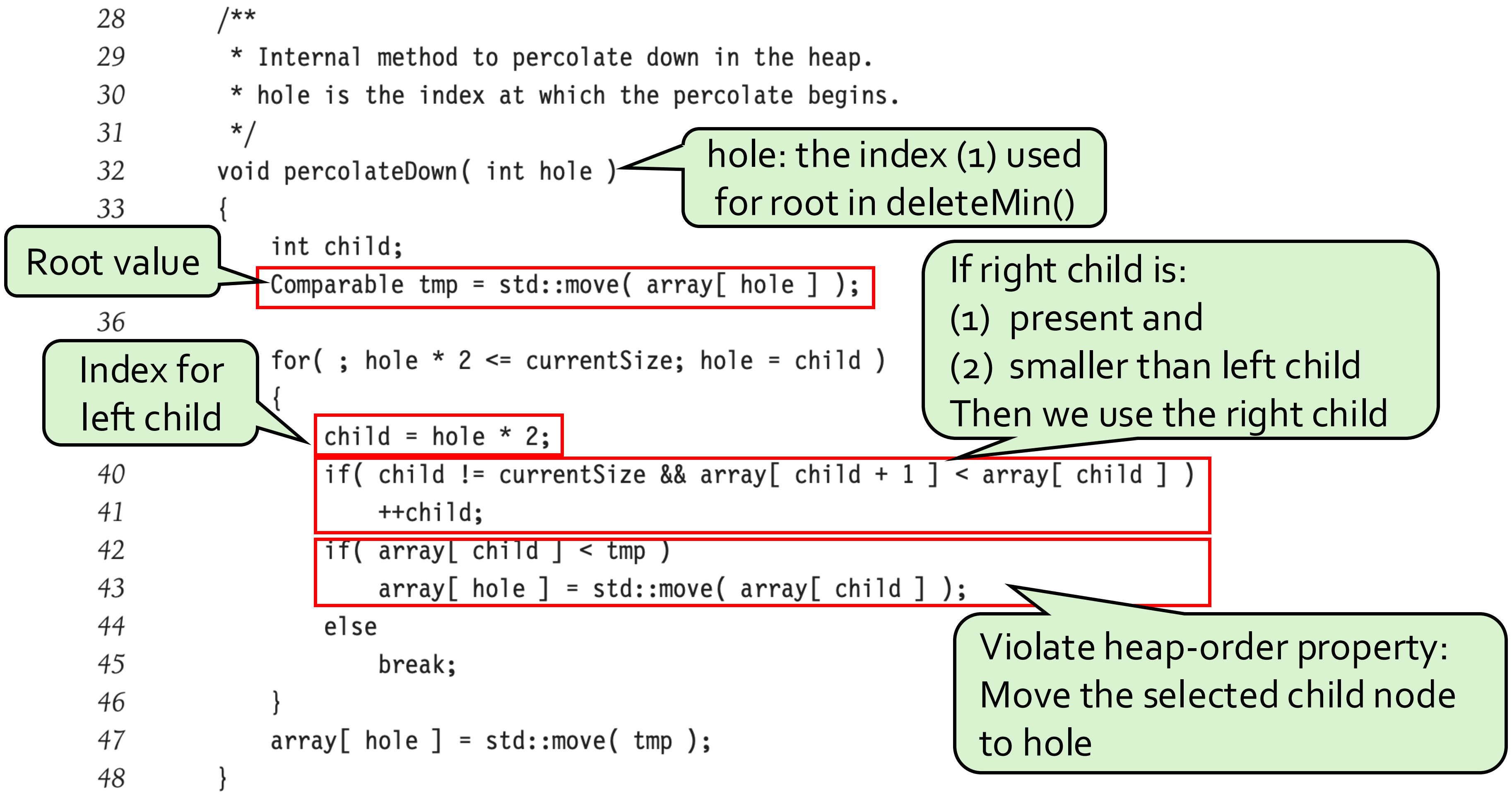


```
/**
 * Internal method to percolate down in the heap.
 * hole is the index at which the percolate begins.
 */
void percolateDown( int hole )
{
    int child;
    Comparable tmp = std::move( array[ hole ] );

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

hole: the index (1) used for root in deleteMin()

Figure 6.12 Method to perform `deleteMin` in a binary heap

Heap deleteMin



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

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

hole: the index (1) used for root in deleteMin()

Root value

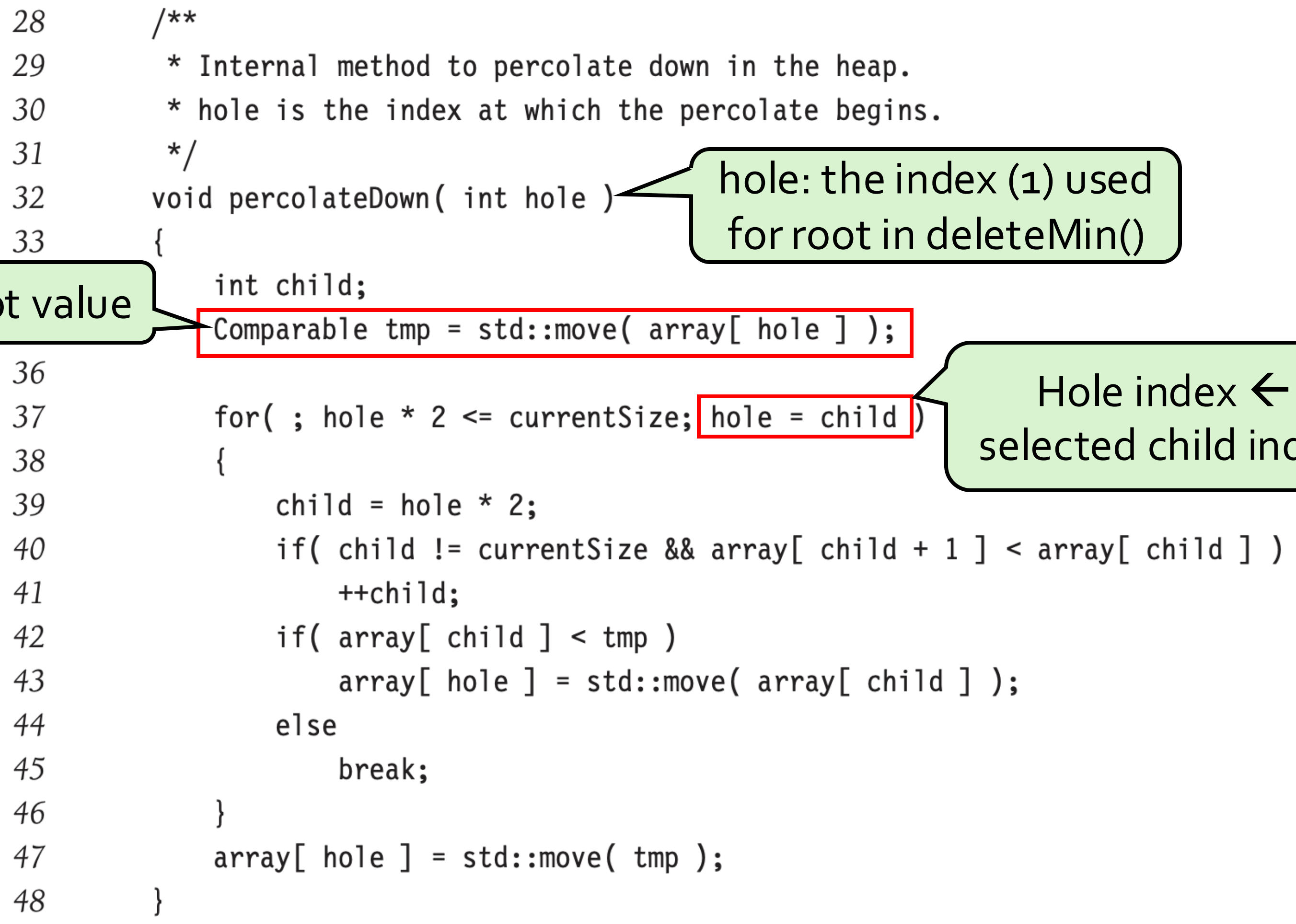
Index for left child

If right child is:
(1) present and
(2) smaller than left child
Then we use the right child

Violate heap-order property: Move the selected child node to hole

Figure 6.12 Method to perform `deleteMin` in a binary heap

Heap deleteMin



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

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

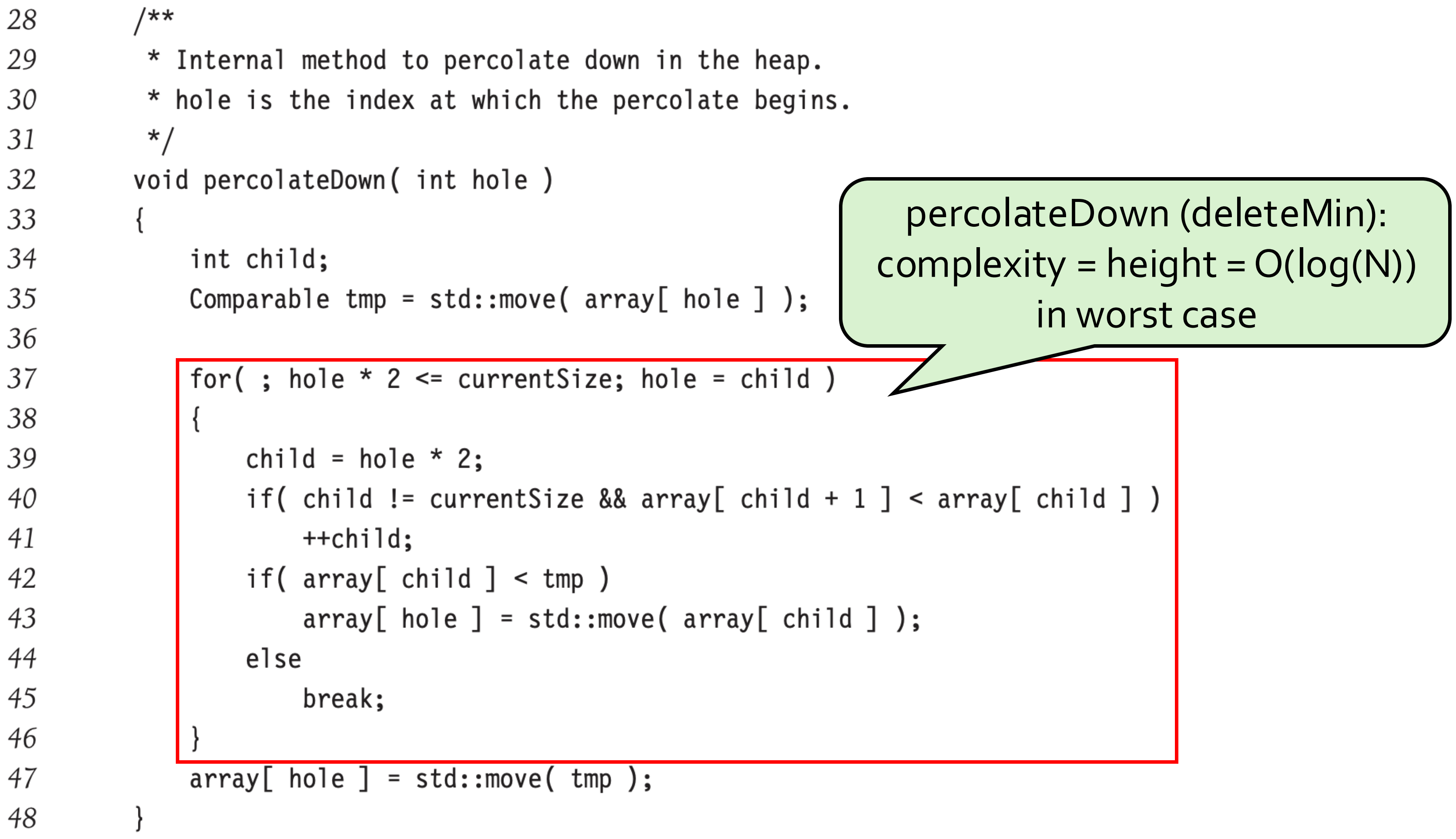
hole: the index (1) used for root in deleteMin()

Root value

Hole index ← selected child index

Figure 6.12 Method to perform `deleteMin` in a binary heap

Heap deleteMin



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

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

percolateDown (deleteMin):
complexity = height = O(log(N))
in worst case

Figure 6.12 Method to perform `deleteMin` in a binary heap

Other heap operations

- `decreaseKey`(p,v)
  - Lowers the current value of item p to new priority value v
  - Need to percolate up
  - E.g., promote a job
- `increaseKey`(p,v)
  - Increases the current value of item p to new priority value v
  - Need to percolate down
  - E.g., demote a job
- `remove`(p)
  - First, decreaseKey(p,-∞)
  - Then, deleteMin
  - E.g., abort/cancel a job

A smaller key: higher priority

A larger key: lower priority

Time complexity for three functions: O(lg(n))

Build a heap

- N successive inserts
  - Each insert:
    - O(1) average [1]
    - O(log(N)) worst-case
  - Total time complexity
    - O(N) average
    - O(N log(N)) worst-case
- A better method buildHeap(): O(N) worst-case

[1] Porter, Thomas, and Istvan Simon. "Random insertion into a priority queue structure." *IEEE Transactions on Software Engineering* 3 (1975): 292-298.

Build a heap

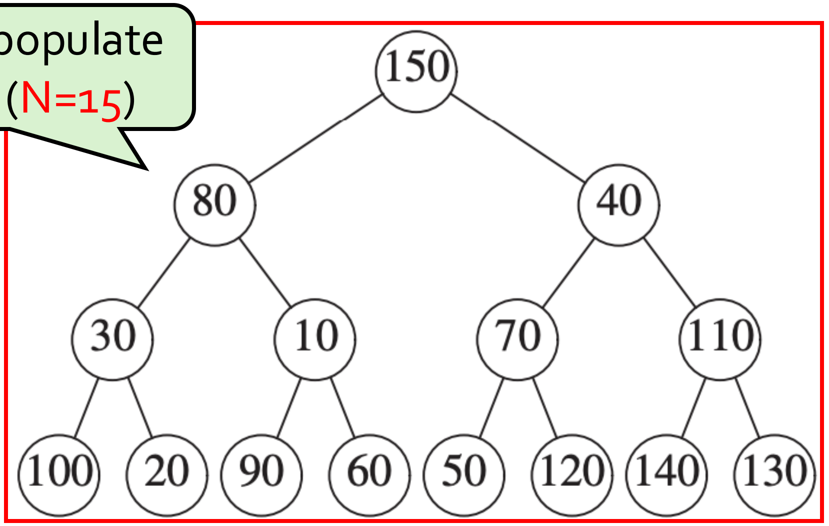
- buildHeap():
- Randomly populate initial heap with structure property
- Perform a percolate-down from each internal node (from element H[size/2] to H[1])
    - → To take care of heap-order property

Heaps

Build a heap

Insert: { 150, 80, 40, 10, 70, 110, 30, 120, 140, 60, 50, 130, 100, 20, 90 }

Randomly populate initial heap (N=15)



Array implementation

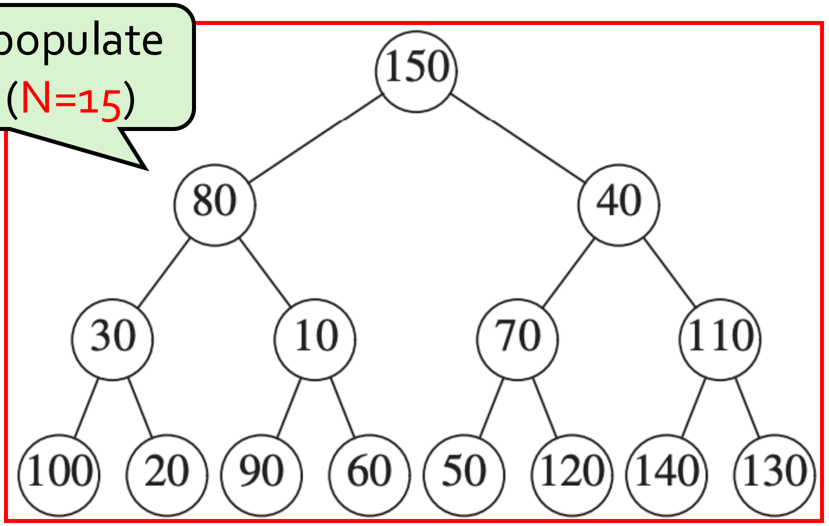
	150	80	40	30	10	70	110	100	…
0	1	2	3	4	5	6	7	8	

Build a heap

Insert: { 150, 80, 40, 10, 70, 110, 30, 120, 140, 60, 50, 130, 100, 20, 90 }

structure property ✓

Randomly populate initial heap (N=15)



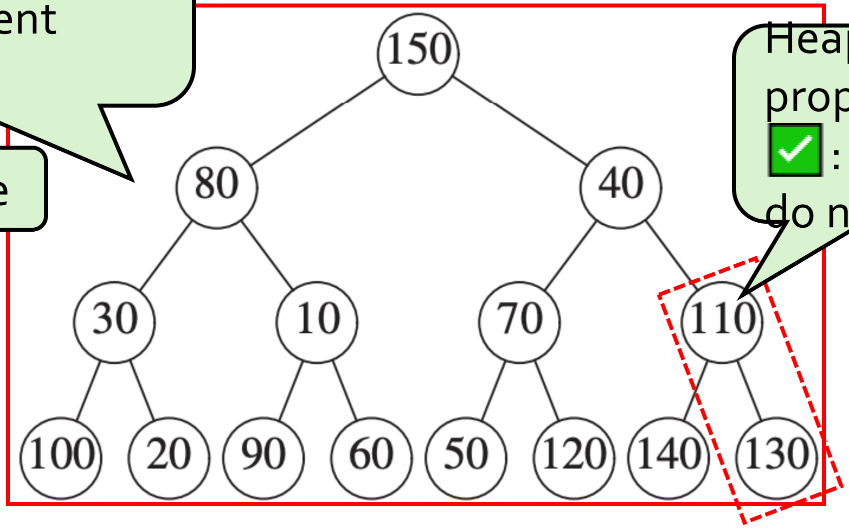
Array implementation

	150	80	40	30	10	70	110	100	…
0	1	2	3	4	5	6	7	8	

Build a heap

Perform a percolate-down from each internal node (from element H[size/2] to H[1])

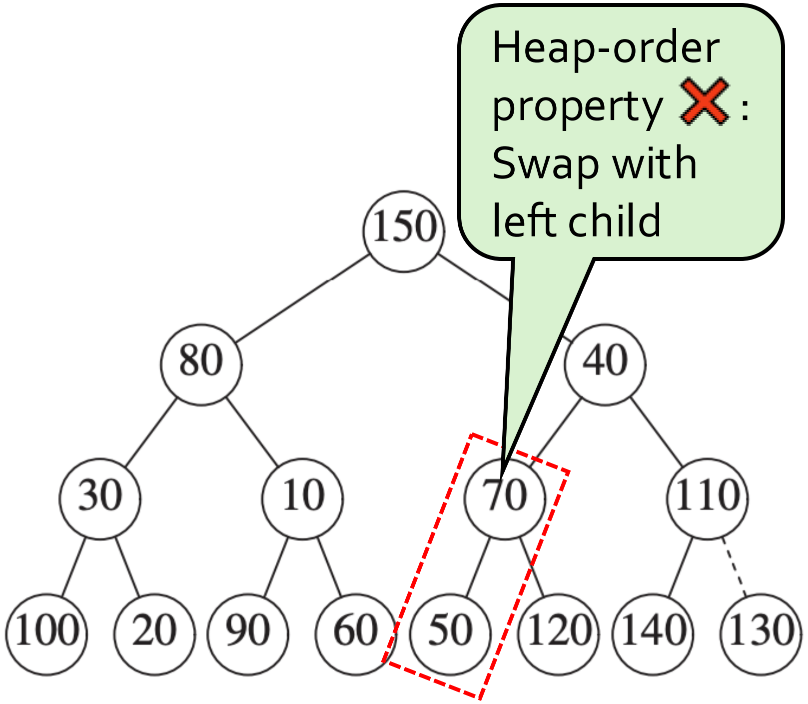
Last internal node



Heap-order property ✓:
do nothing

	150	80	40	30	10	70	110	100	...
0	1	2	3	4	5	6	7	8	

Build a heap

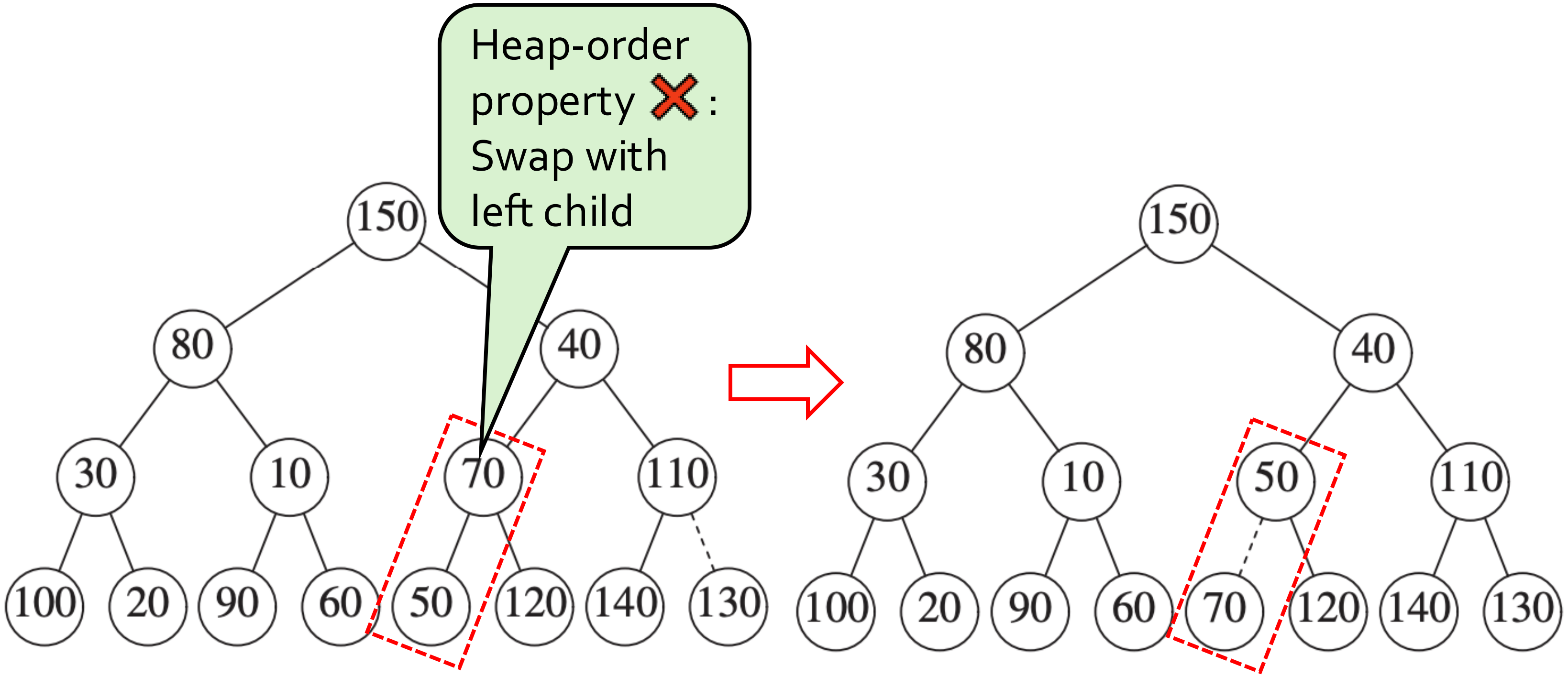


Heap-order property ✗: Swap with left child

	150	80	40	30	10	70	110	100	…
0	1	2	3	4	5	6 ←	7	8	

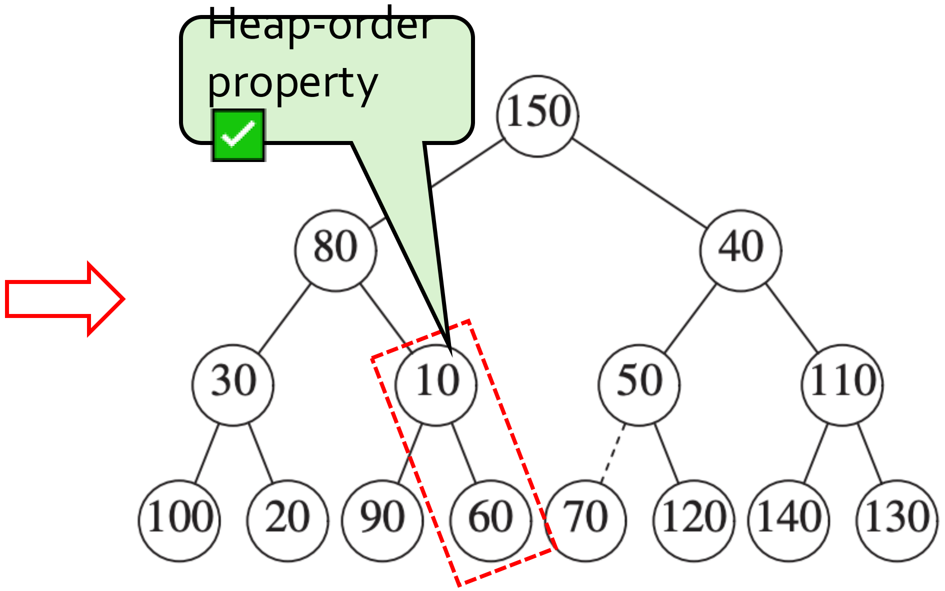
Heaps

Build a heap


Heap-order property ❌: Swap with left child

	150	80	40	30	10	50	110	100	…
0	1	2	3	4	5	6	7	8	

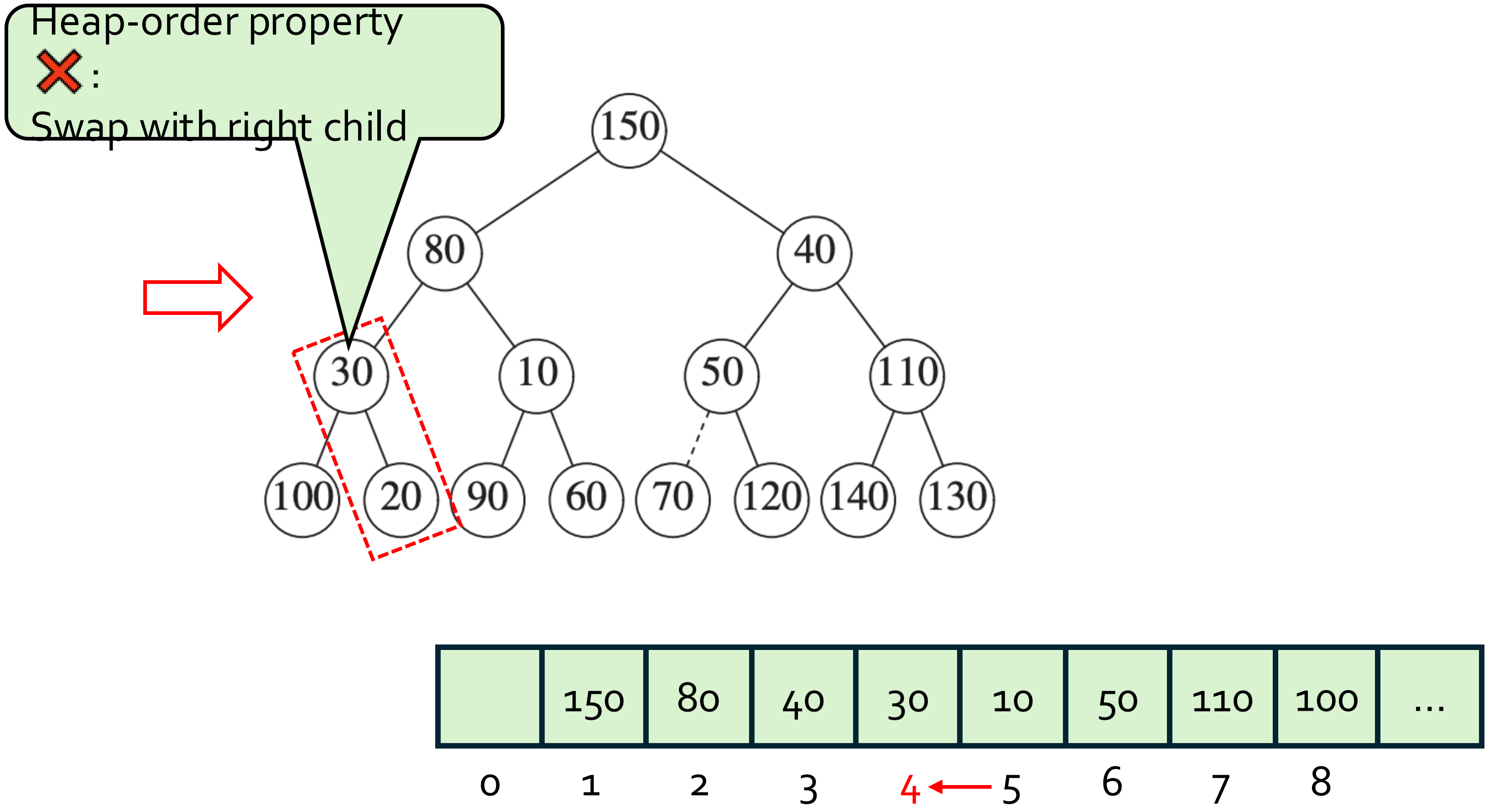
Build a heap



Heap-order property ✓

	150	80	40	30	10	50	110	100	...
0	1	2	3	4	5 ←	6	7	8	

Build a heap

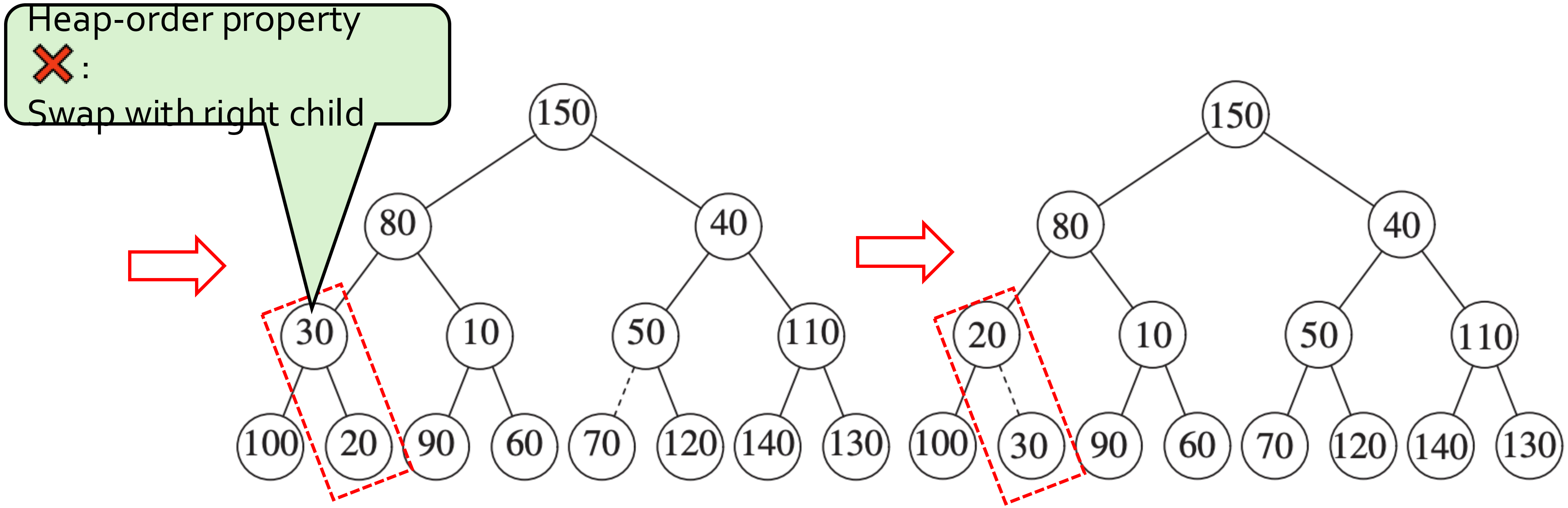


Heap-order property
❌:
Swap with right child

	150	80	40	30	10	50	110	100	...
0	1	2	3	4 ←	5	6	7	8	

Heaps

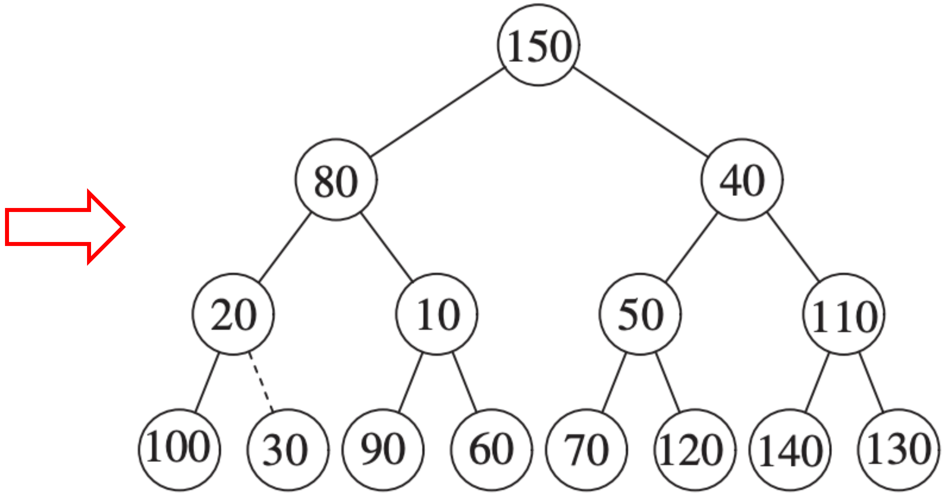
Build a heap



Heap-order property ✗:
Swap with right child

	150	80	40	20	10	50	110	100	…
0	1	2	3	4	5	6	7	8	

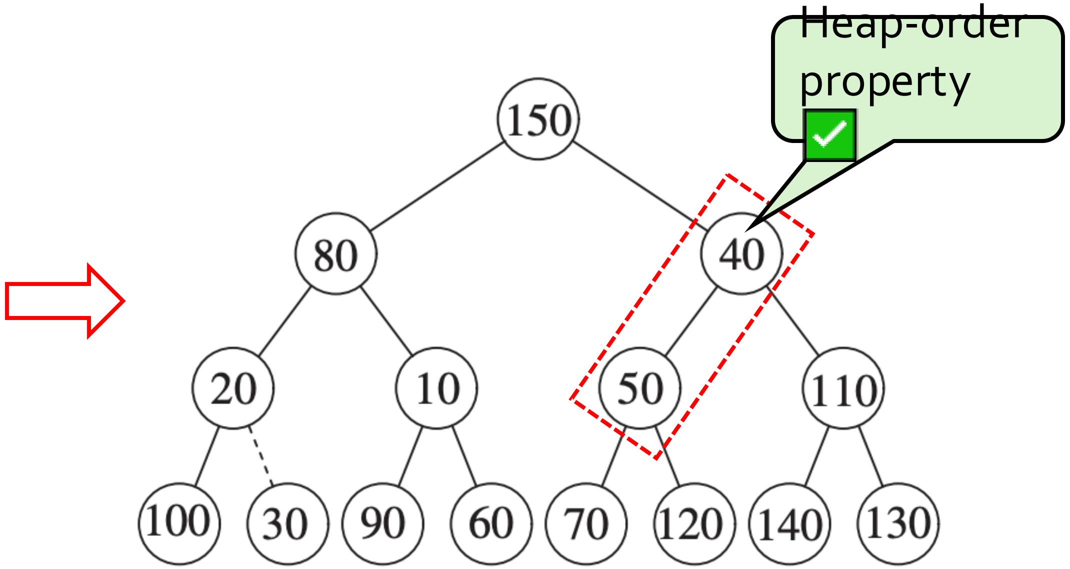
Build a heap



	150	80	40	20	10	50	110	100	…
0	1	2	3	4	5	6	7	8	

Heaps

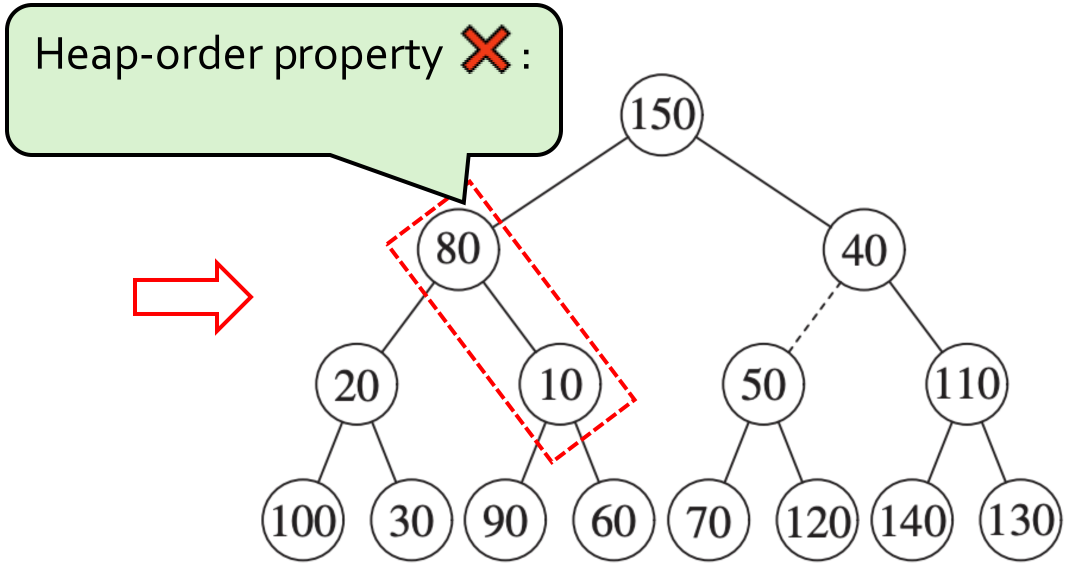
Build a heap



Heap-order property ✓

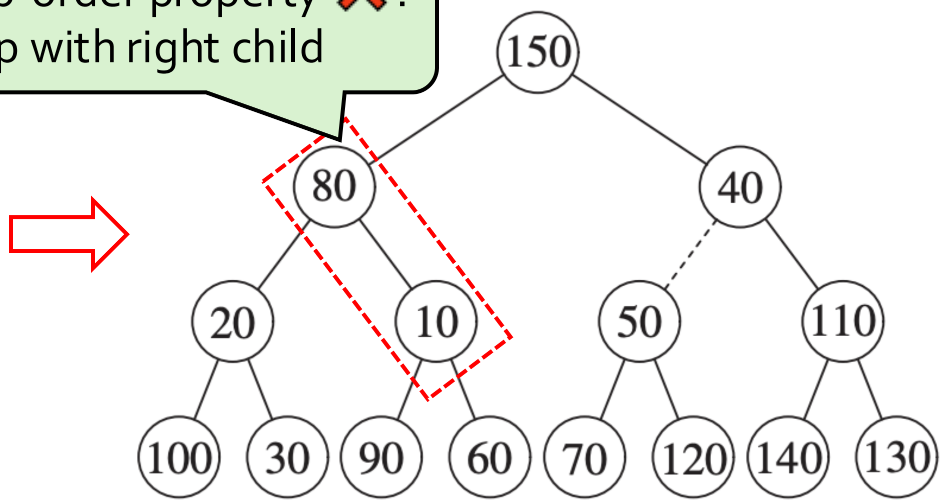
	150	80	40	20	10	50	110	100	...
0	1	2	3 ←	4	5	6	7	8	

Build a heap


Heap-order property ❌:

	150	80	40	20	10	50	110	100	...
0	1	2 ←	3	4	5	6	7	8	

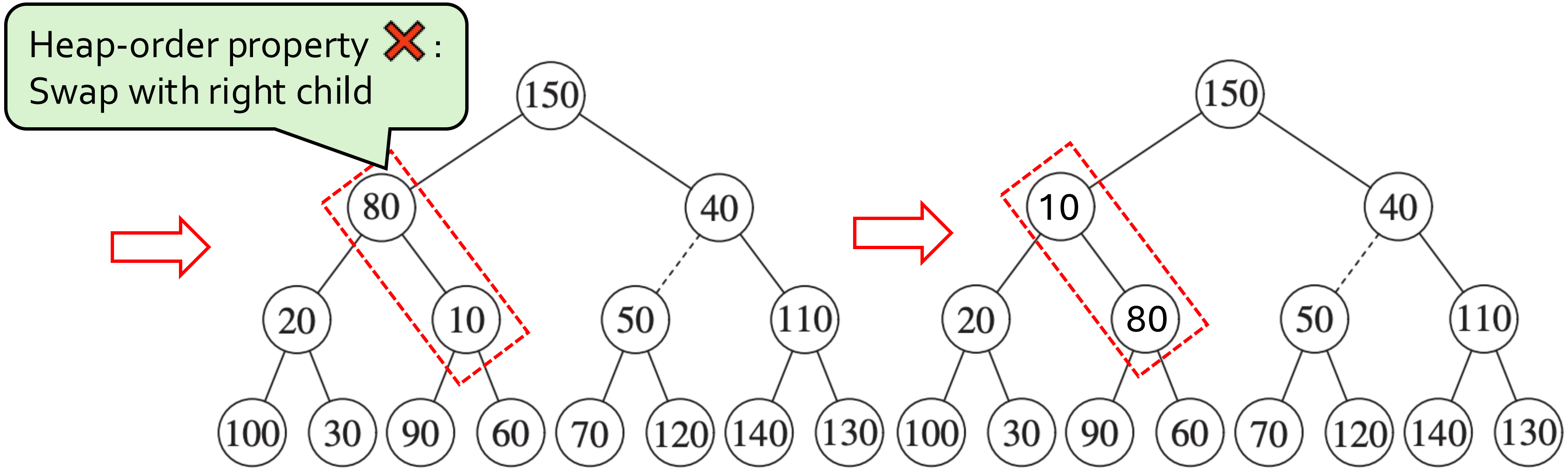
Build a heap



Heap-order property ❌:
Swap with right child

	150	80	40	20	10	50	110	100	…
0	1	2 ←	3	4	5	6	7	8	

Build a heap

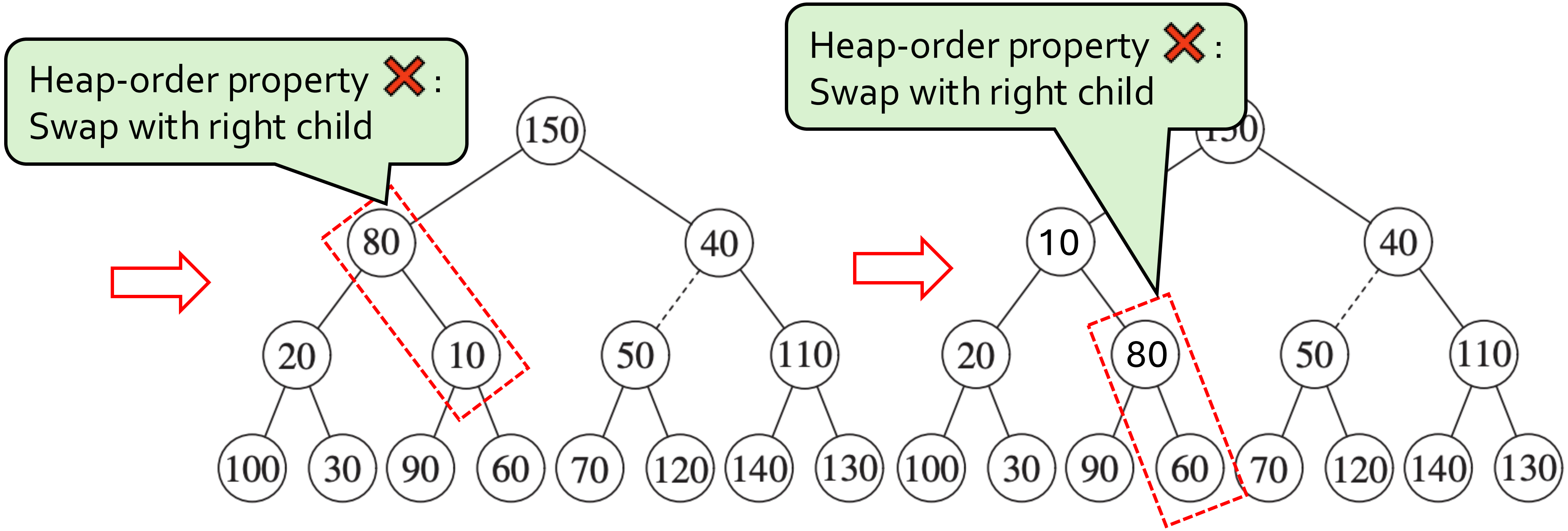


Heap-order property ✗:
Swap with right child

	150	10	40	20	80	50	110	100	...
0	1	2	3	4	5	6	7	8	

Heaps

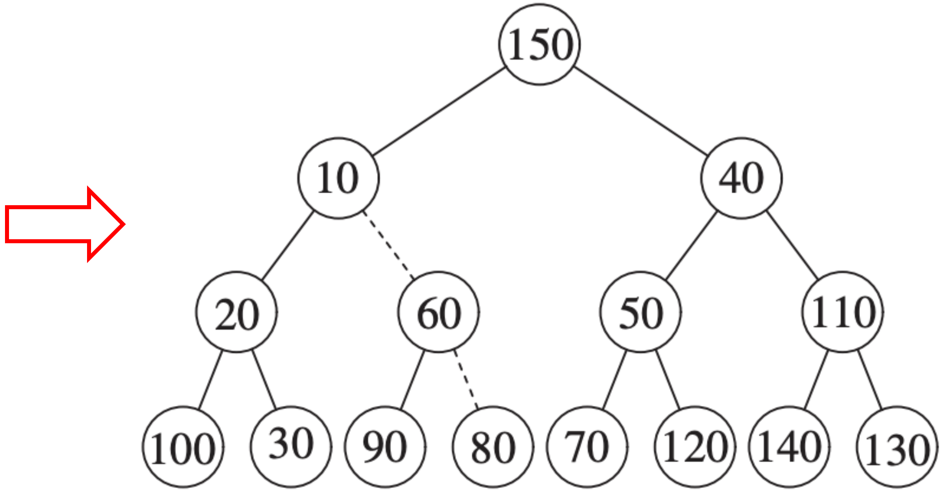
Build a heap


Heap-order property ❌: Swap with right child

Heap-order property ❌: Swap with right child

	150	10	40	20	80	50	110	100	…
0	1	2	3	4	5	6	7	8	

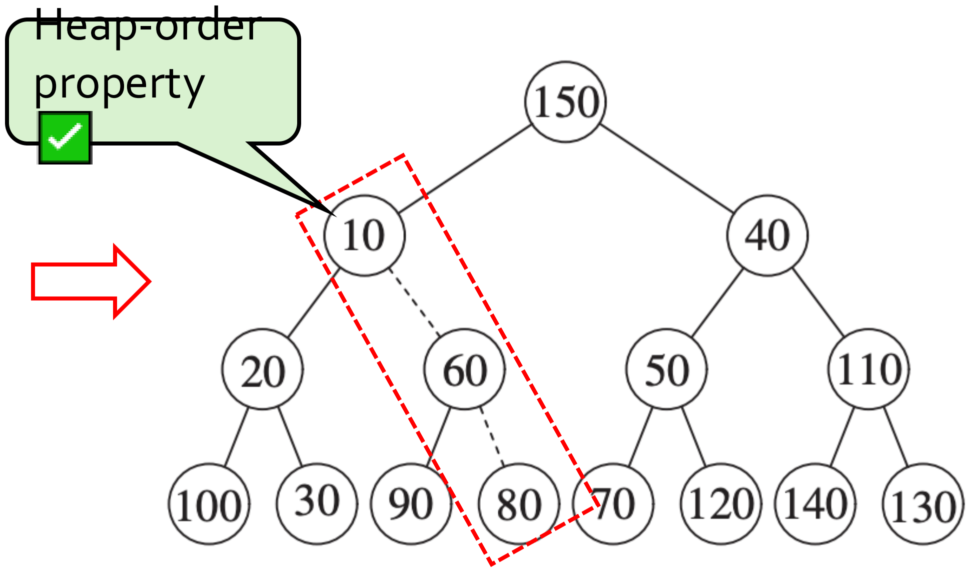
Build a heap



	150	10	40	20	60	50	110	100	…
0	1	2	3	4	5	6	7	8	

Heaps

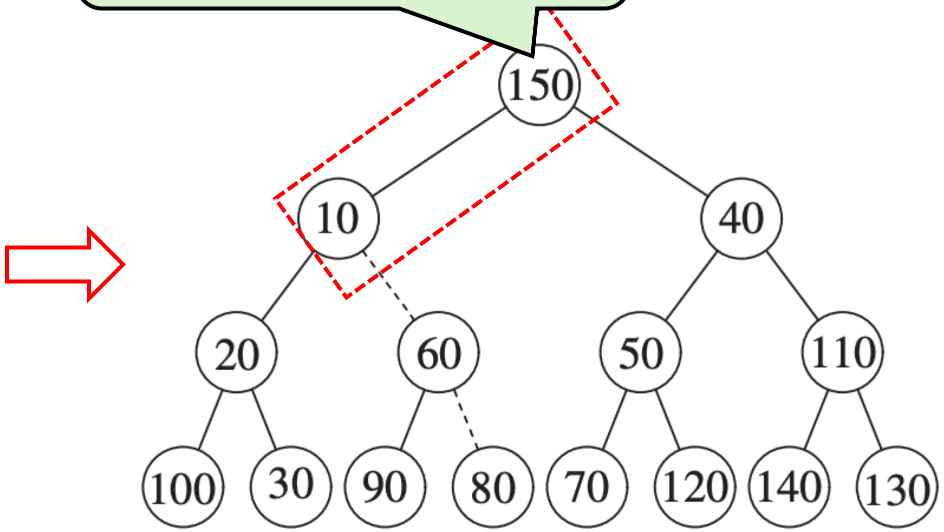
Build a heap



	150	10	40	20	60	50	110	100	...
0	1	2	3	4	5	6	7	8	

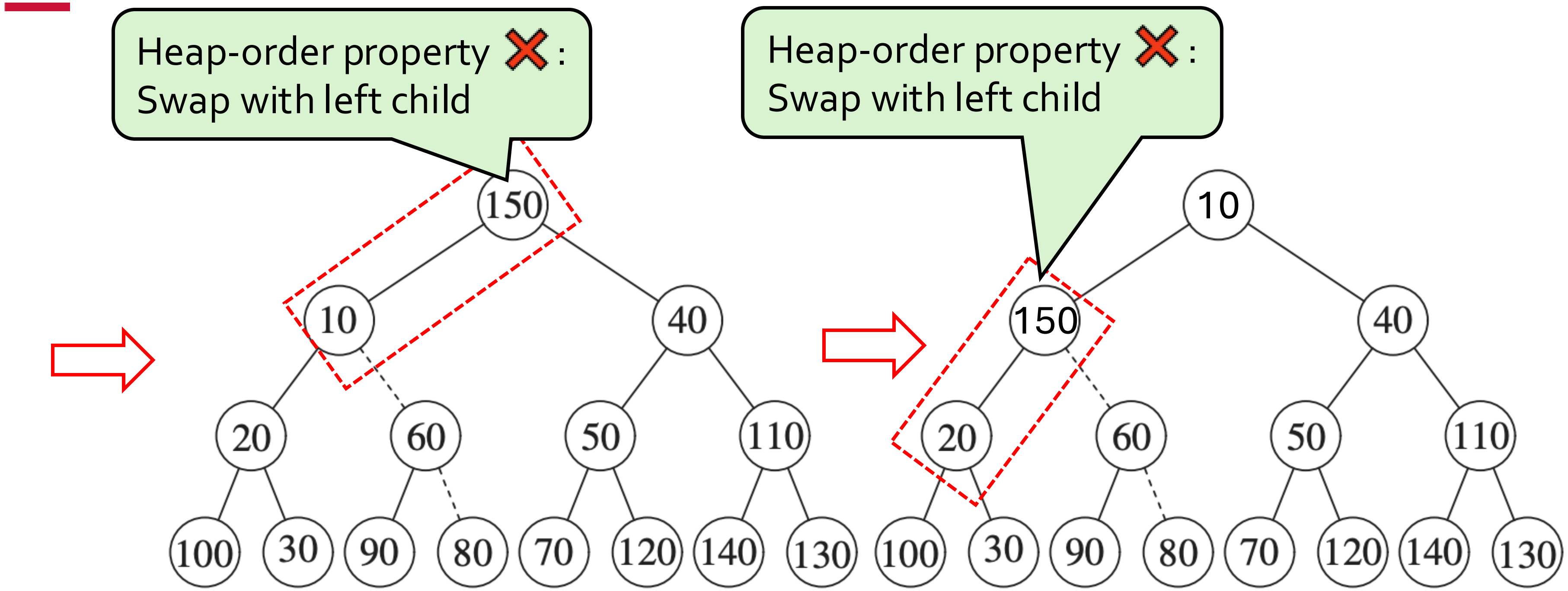
Build a heap

Heap-order property ✗ :
Swap with left child



	150	10	40	20	60	50	110	100	…
0	1 ←	2	3	4	5	6	7	8	

Build a heap



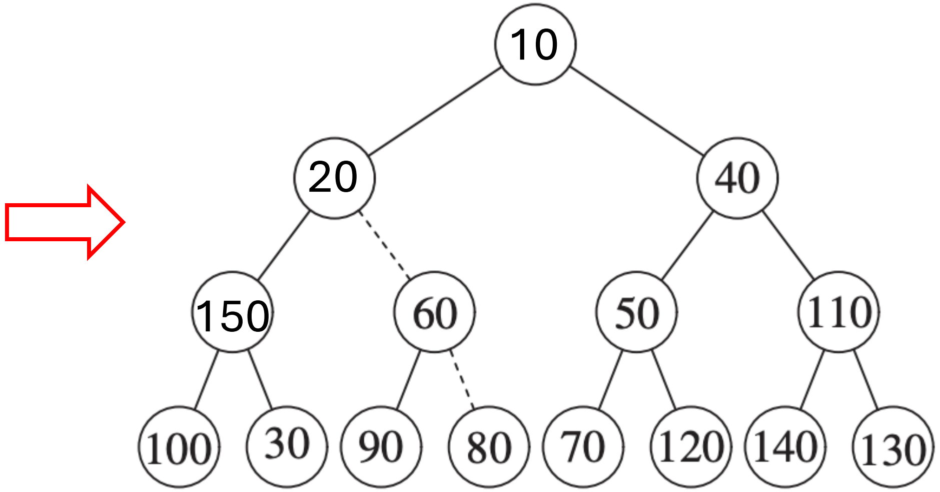
Heap-order property ✗:
Swap with left child

Heap-order property ✗:
Swap with left child

	10	150	40	20	60	50	110	100	…
0	1	2	3	4	5	6	7	8	

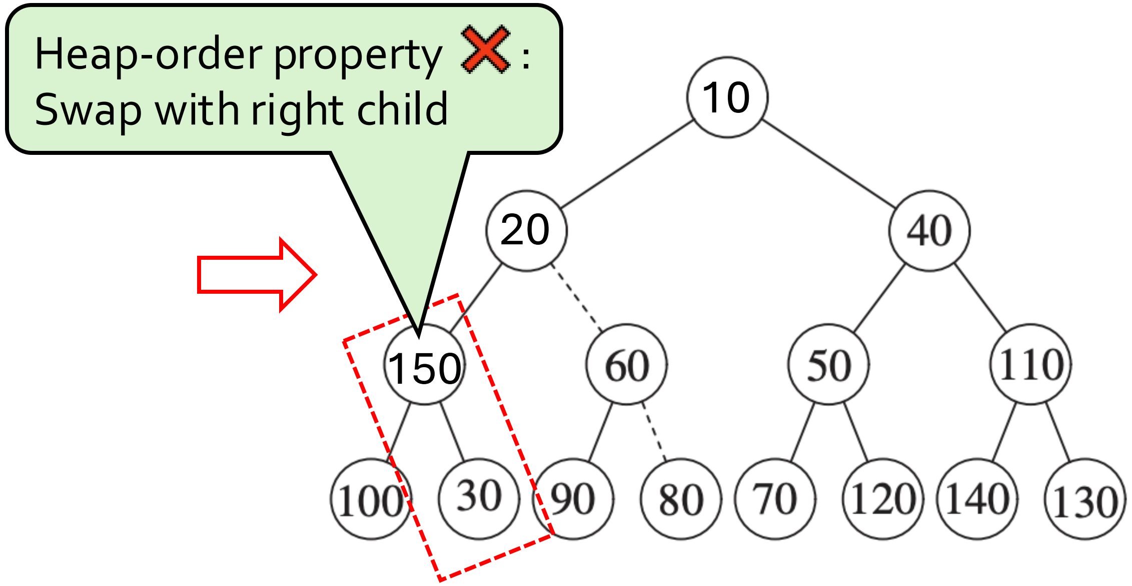
Heaps

Build a heap



	10	20	40	150	60	50	110	100	…
0	1	2	3	4	5	6	7	8	

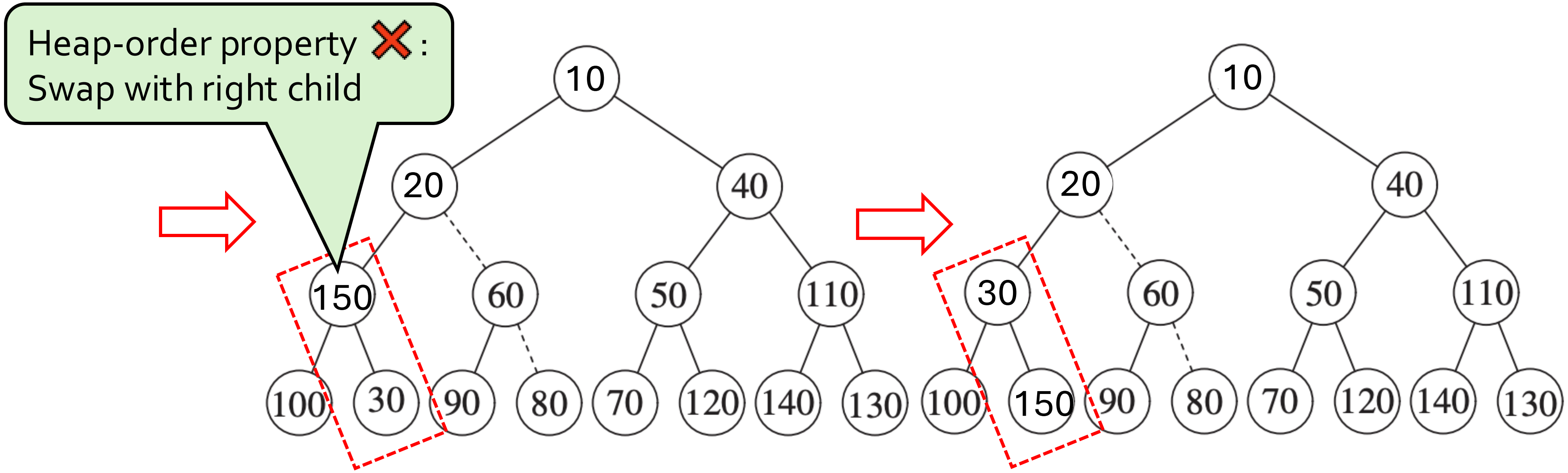
Build a heap



Heap-order property ✗:
Swap with right child

	10	20	40	150	60	50	110	100	…
0	1	2	3	4	5	6	7	8	

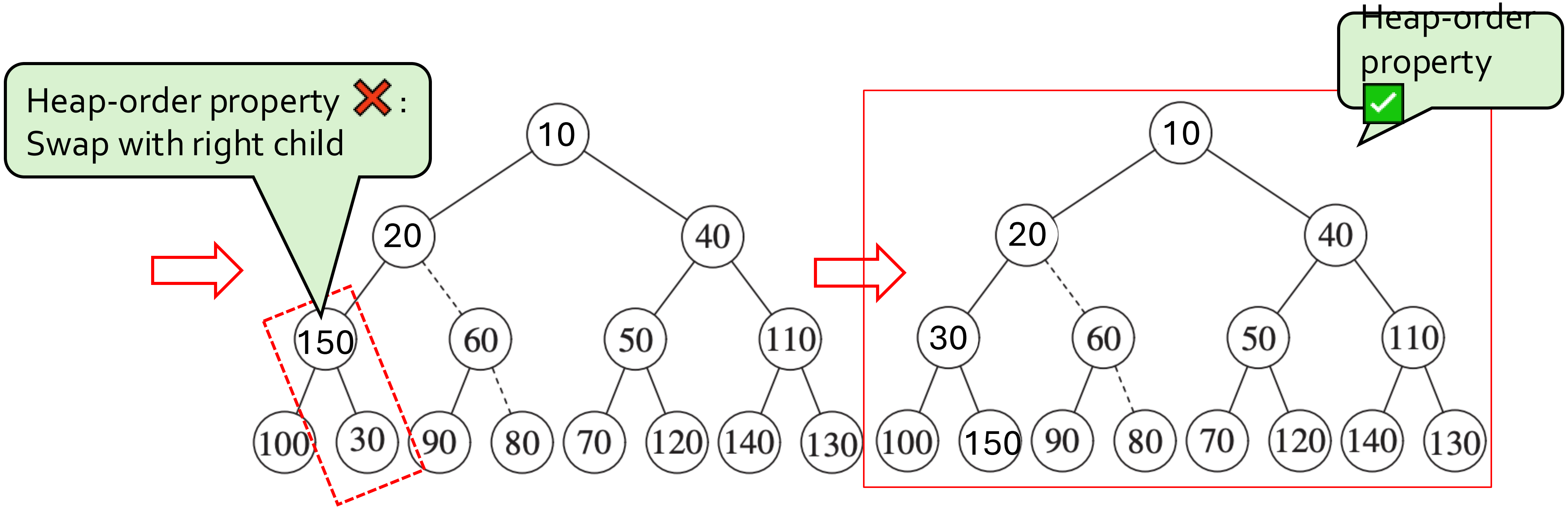
Build a heap



Heap-order property ✗:
Swap with right child

	10	20	40	30	60	50	110	100	...
0	1	2	3	4	5	6	7	8	

Build a heap



Heap-order property ✗: Swap with right child

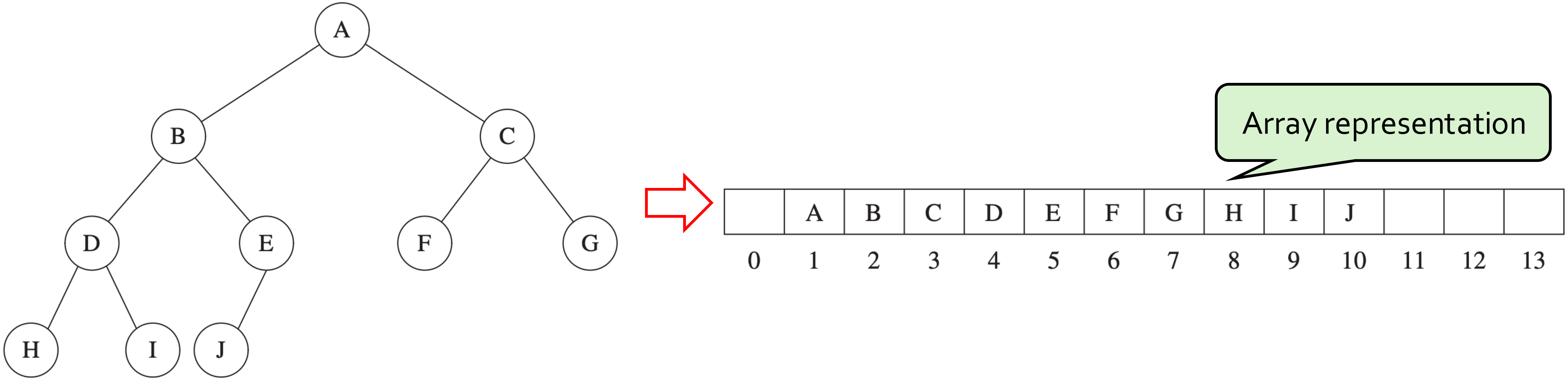
Heap-order property ✓

	10	20	40	30	60	50	110	100	...
0	1	2	3	4	5	6	7	8	

Heaps

buildHeap

- The key point is to find the lowest and right most internal node
  - aka last internal node
- Offset of the last internal node = floor (offset of the last node / 2)
- Why? Parent(i) = at position floor(i/2)

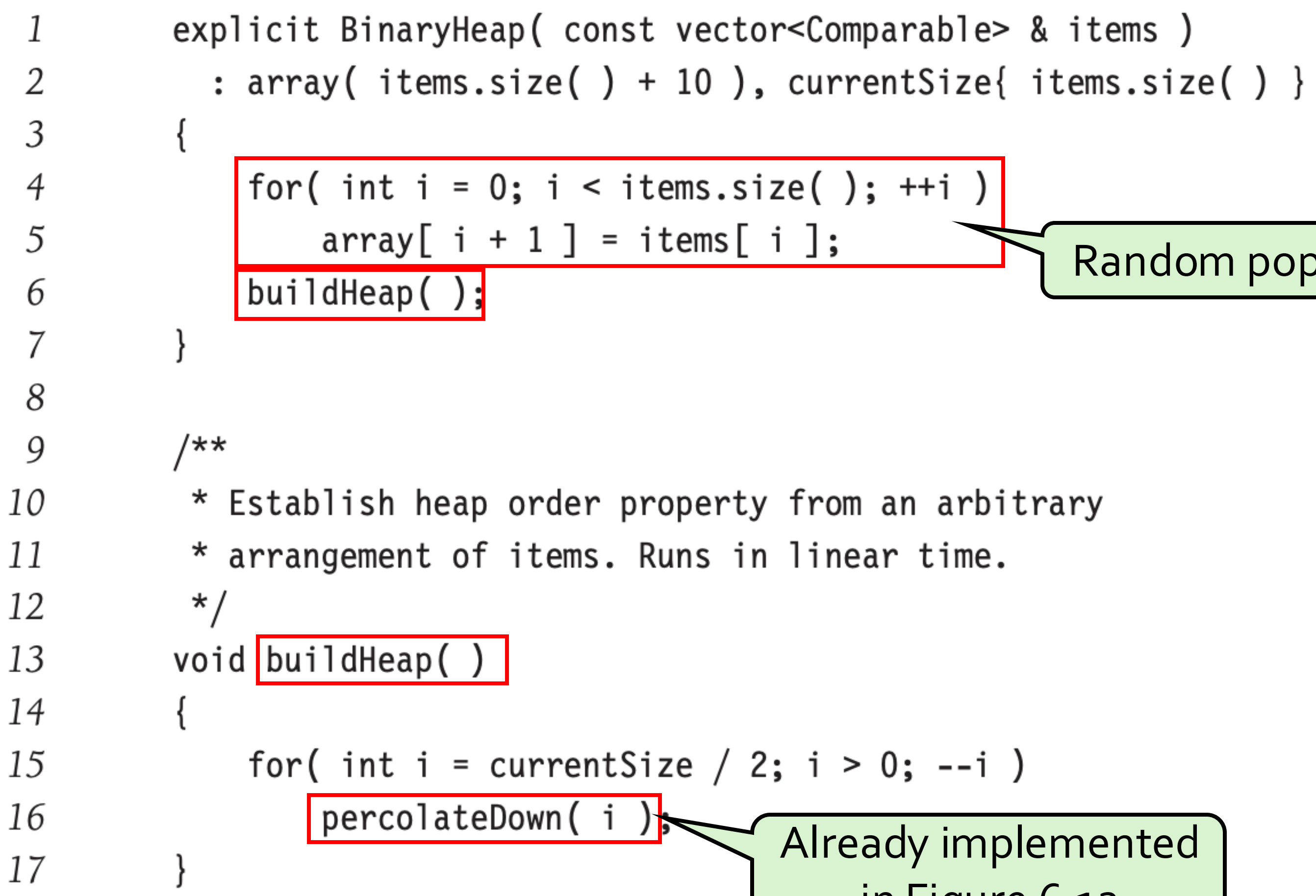


Array representation

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

Figure 6.2 A complete binary tree

buildHeap implementation



```
explicit BinaryHeap( const vector<Comparable> & items )
  : array( items.size( ) + 10 ), currentSize{ items.size( ) }
{
    for( int i = 0; i < items.size( ); ++i )
        array[ i + 1 ] = items[ i ];
    buildHeap( );
}

/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; --i )
        percolateDown( i );
}
```

Random populate

Already implemented in Figure 6.12

Figure 6.14 buildHeap and constructor

buildHeap time complexity

- Run-time = ?
  - O(sum of the heights of all the internal nodes)
- because we may have to percolate all the way down to fix every internal node in the worst-case
- Theorem 6.1: For a perfect binary tree of height h, the sum of heights of all nodes is 2h+1 – 1 – (h + 1)
- Since h=lg(N), then sum of heights is O(N)

Will be slightly better in practice

- Implication:
  - Each insertion costs O(1) amortized time

Binary heap worst-case analysis

- Height: $\lfloor \log(N) \rfloor$
- insert: O(lg(N)) for each insert
- deleteMin: O(lg(N))
- decreaseKey: O(lg(N))
- increaseKey: O(lg(N))
- remove: O(lg(N))

Binary heap v.s. AVL tree

- Binary Heap does not require extra space for pointers
- Binary Heap is easier to implement
- Although operations are of same time complexity, constants in BST are higher

Application: selection problem

- Given a list of n elements, find the kth smallest element
- Algorithm 1:
  - Sort the list: O(n log n)
  - Pick the kth element: O(1)
- A better algorithm:
  - Use a binary heap (minHeap)

Selection using a minHeap

- Input: n elements
- Algorithm:
  1. buildHeap(n) O(n)
  2. Perform k deleteMin() operations O(k log(n))
  3. Report the kth deleteMin output O(1)
- Total run-time = O(n + k log(n) + 1)
- If k = O(n/log n) then the run-time becomes O(n)

Other heaps

- Binomial Heaps
- d-Heaps
- Generalization of binary heaps (ie., 2-Heaps)
- Leftist Heaps
  - Supports merging of two heaps in o(m+n) time (ie., sub- linear)
- Skew Heaps
  - O(log n) amortized run-time
- Fibonacci Heaps

Time complexity per operation

	findMin	insert	deleteMin	merge
Binary heap	O(1)	O(log(n)) worst-case O(1) amortized for buildHeap	O(log(n))	O(n)
Leftist heap	O(1)	O(log(n))	O(log(n))	O(log(n))
Skew heap	O(1)	O(log(n))	O(log(n))	O(log(n))
Binomial heap	O(1)	O(log(n)) worst-case O(1) amortized for sequence of n inserts	O(log(n))	O(log(n))
Fibonacci heap	O(1)	O(1)	O(log(n))	O(1)

Priority queues in STL

- Uses Binary heap
- Default is maxHeap
- Methods
    - Push, top, pop, empty, clear
- For minHeap:
    - declare priority_queue as:
    - priority_queue<int, vector<int>, greater<int>> Q;

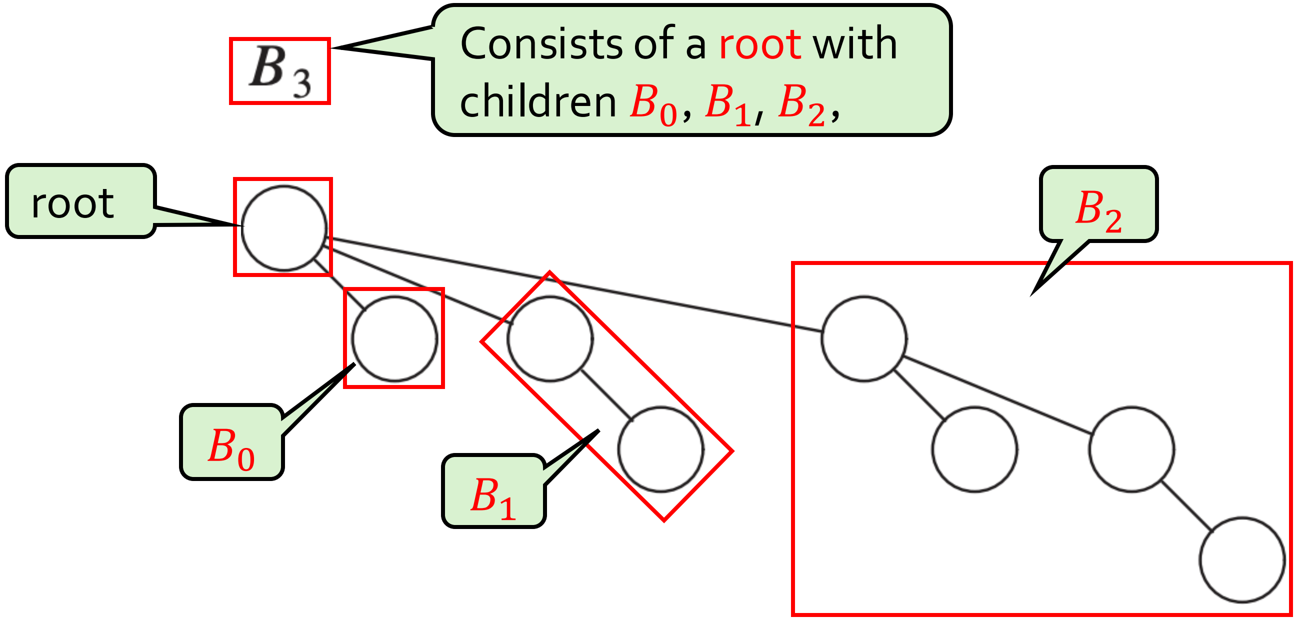
```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    priority_queue<int> Q;
    Q.push(10);
    Q.push(3);
    Q.push(12);
    cout << Q.top() << endl;
    Q.pop();
    cout << Q.top() << endl;
    return 0;
}
```

Binomial heap

- A binomial heap is a forest of heap-ordered binomial trees, satisfying:
  - Structure property
  - Heap-order property
- A binomial heap is different from binary heap in that:
  - Its structure property is totally different from binary heap
  - Its heap-order property (within each binomial tree) is the same as in a binary heap

Definition: binomial tree

- A binomial tree of height k is denoted B_k:
  - consists of a root with children $B_0, B_1, \ldots, B_{k-1}$
  - has 2^k nodes
  - # of nodes at depth d → $\binom{k}{d}$



Definition: binomial tree

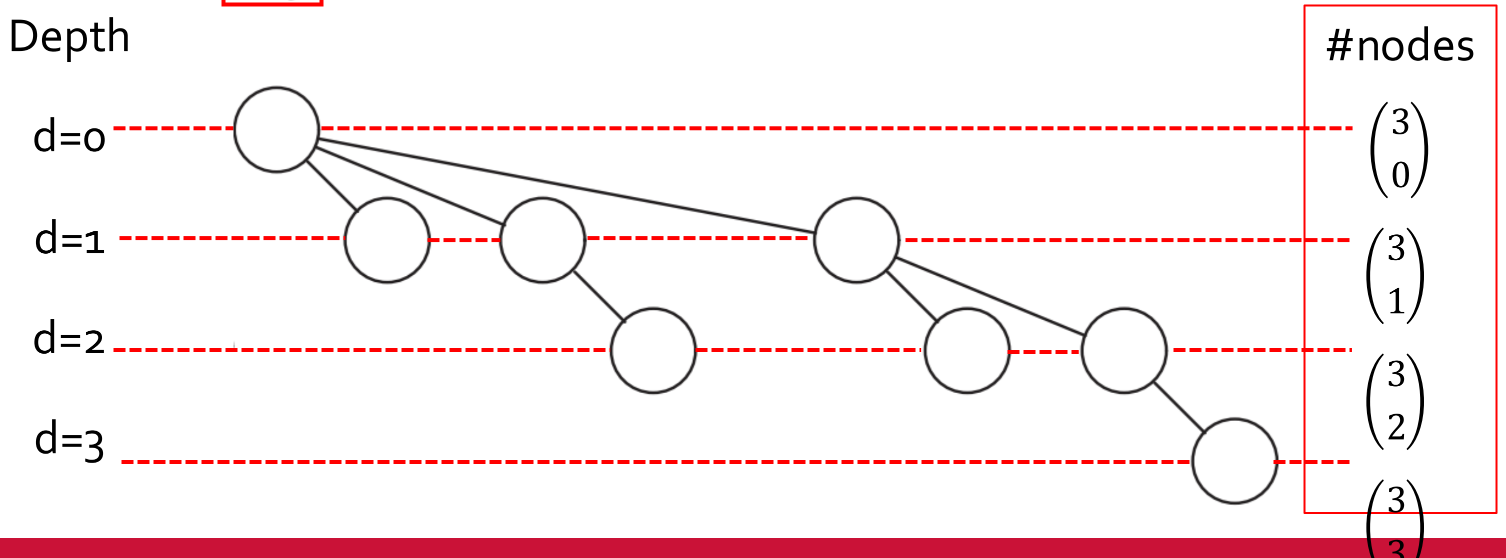
- A binomial tree of height k is denoted B_k:
  - consists of a root with children $B_0, B_1, \ldots, B_{k-1}$
  - has 2^k nodes
  - # of nodes at depth d → $\binom{k}{d}$

d-combination of k elements

$$\binom{k}{d} = \frac{k!}{d!\,(k-d)!}$$

is the form of the coefficients in binomial theorem

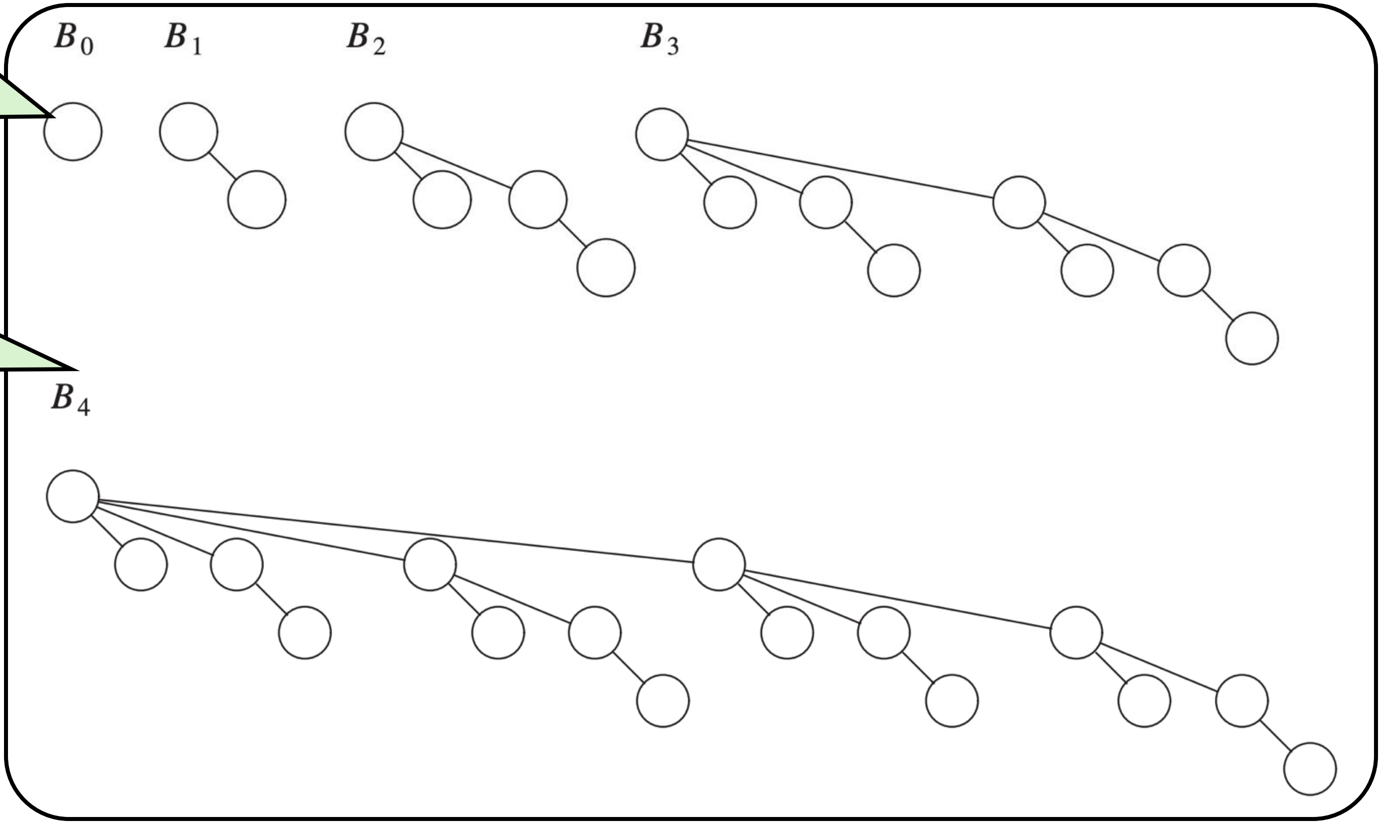
B_3



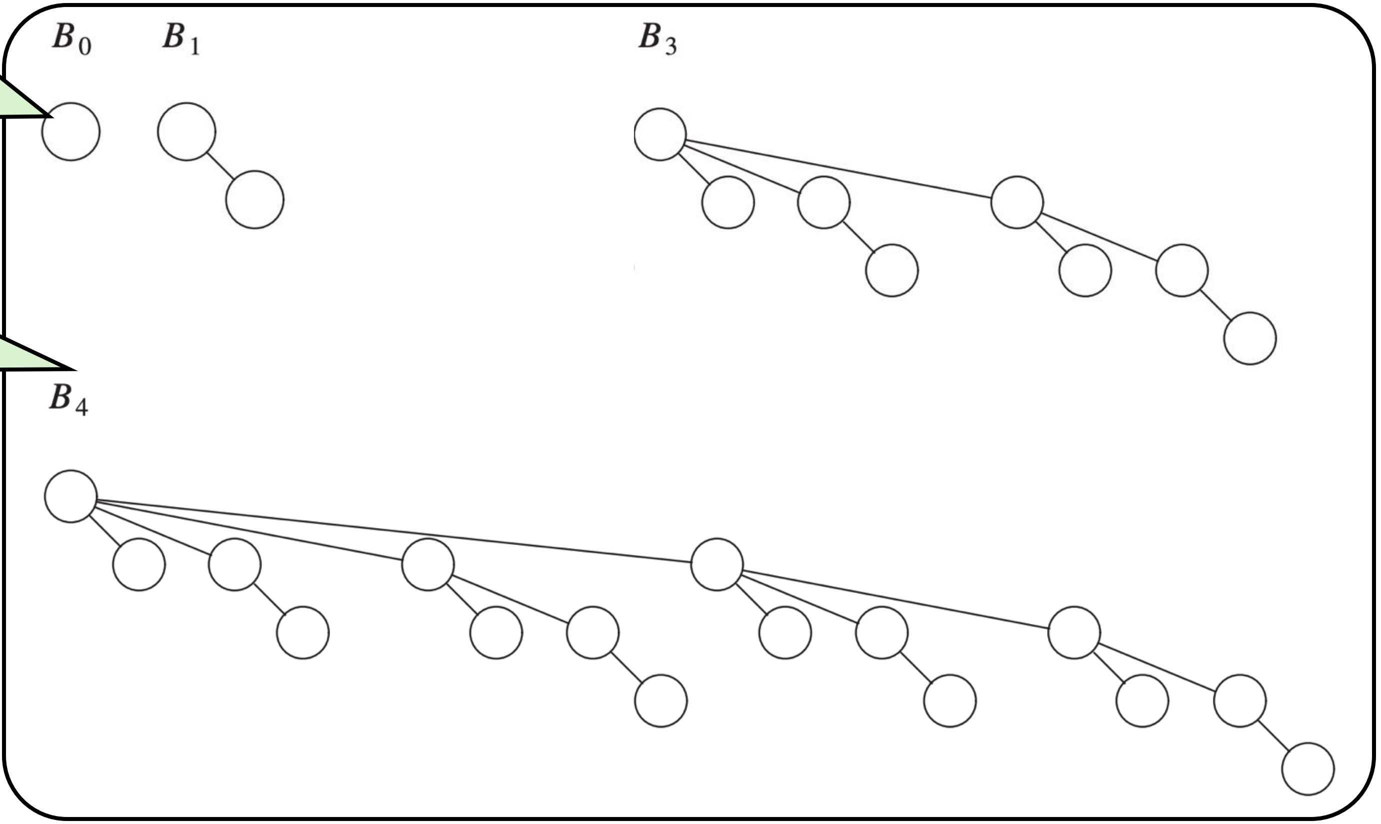
Binomial heaps: example

A binomial heap (a forest of binomial trees) with N = 31

Binary representation of N:
N = 31 = $(1\ 1\ 1\ 1\ 1)_2$



Binomial heaps: example



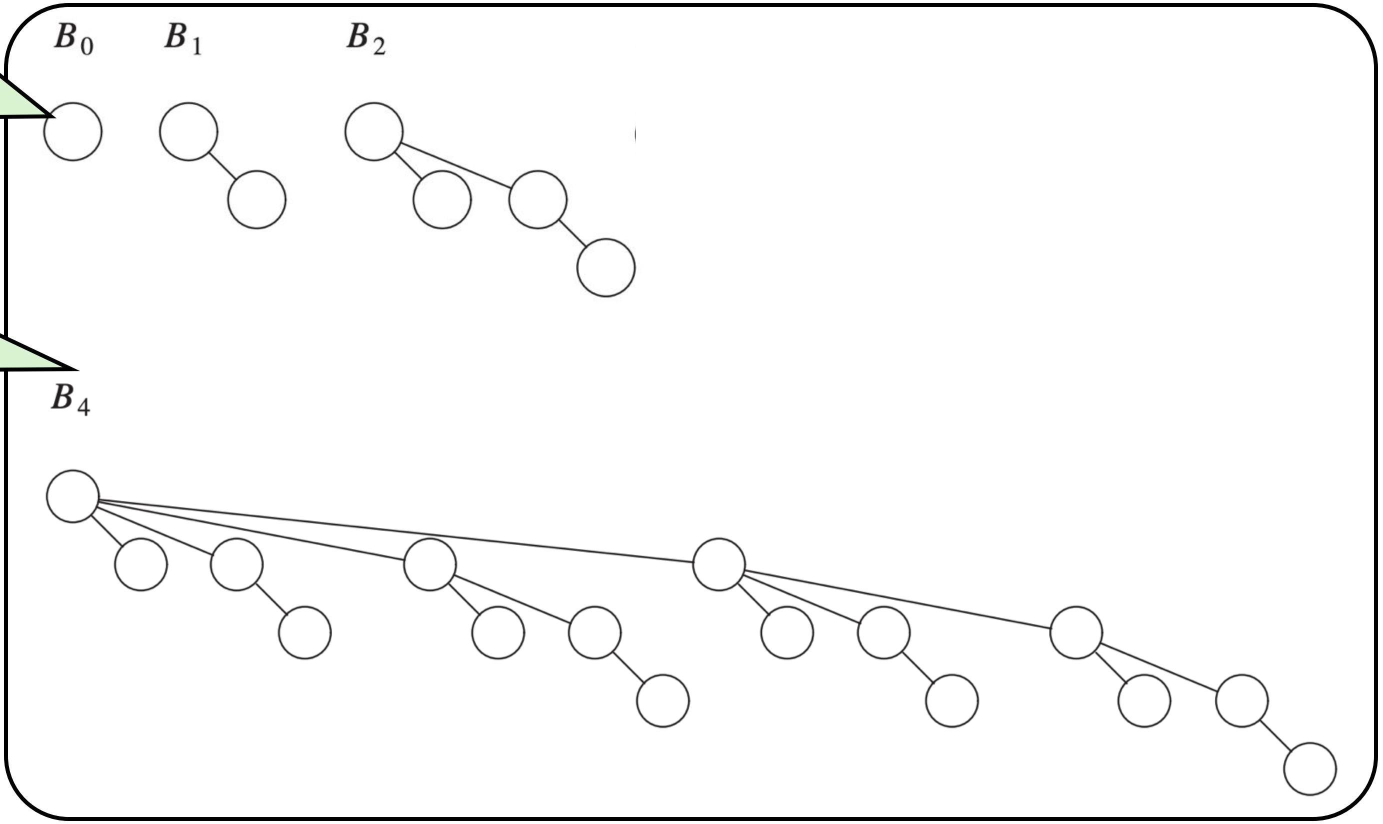
A binomial heap (a forest of binomial trees) with N = 27

Binary representation of N:
N = 27 = $(1\ 1\ 0\ 1\ 1)_2$

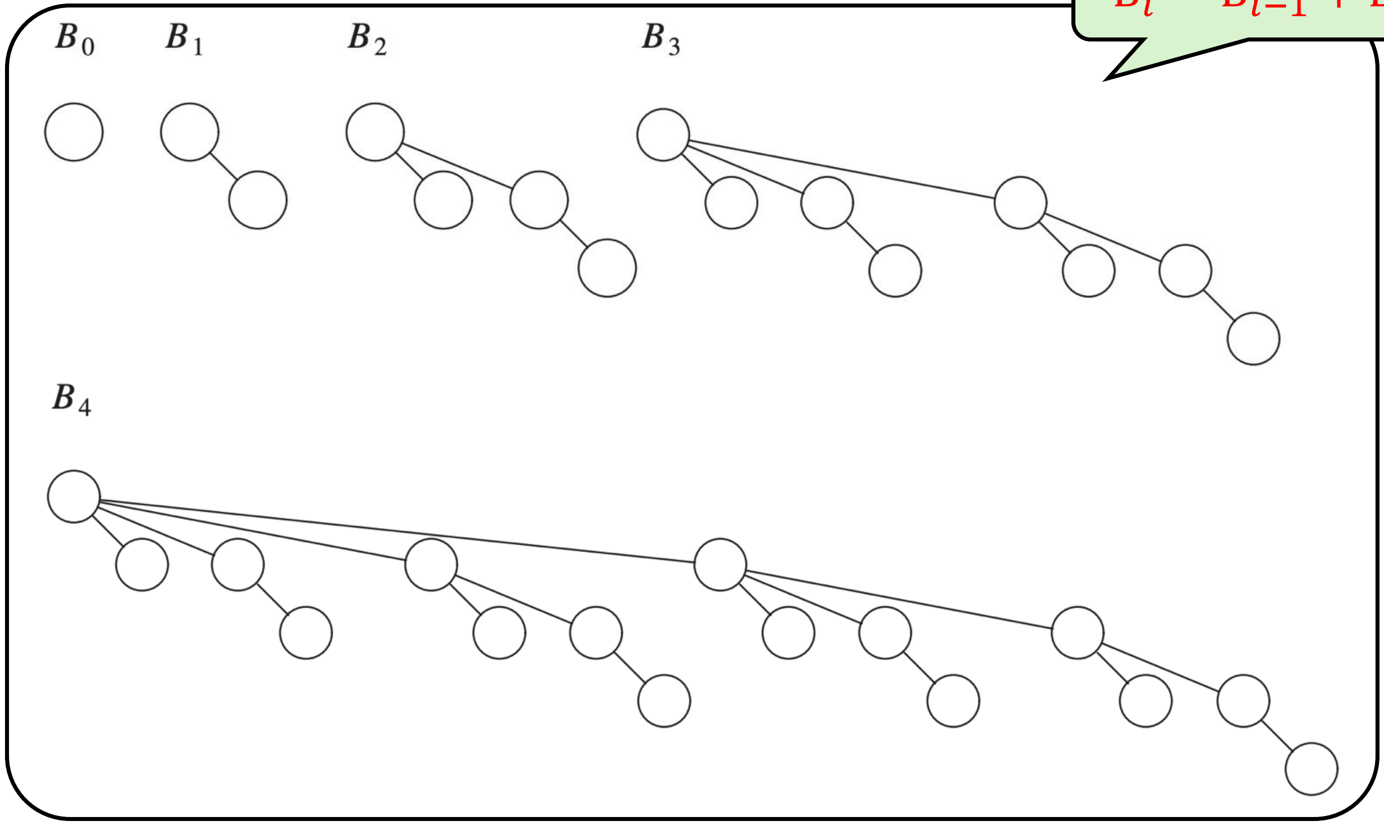
Binomial heaps: example

A binomial heap (a forest of binomial trees) with N = 23

Binary representation of N: N = 23 = $(1\ 0\ 1\ 1\ 1)_2$

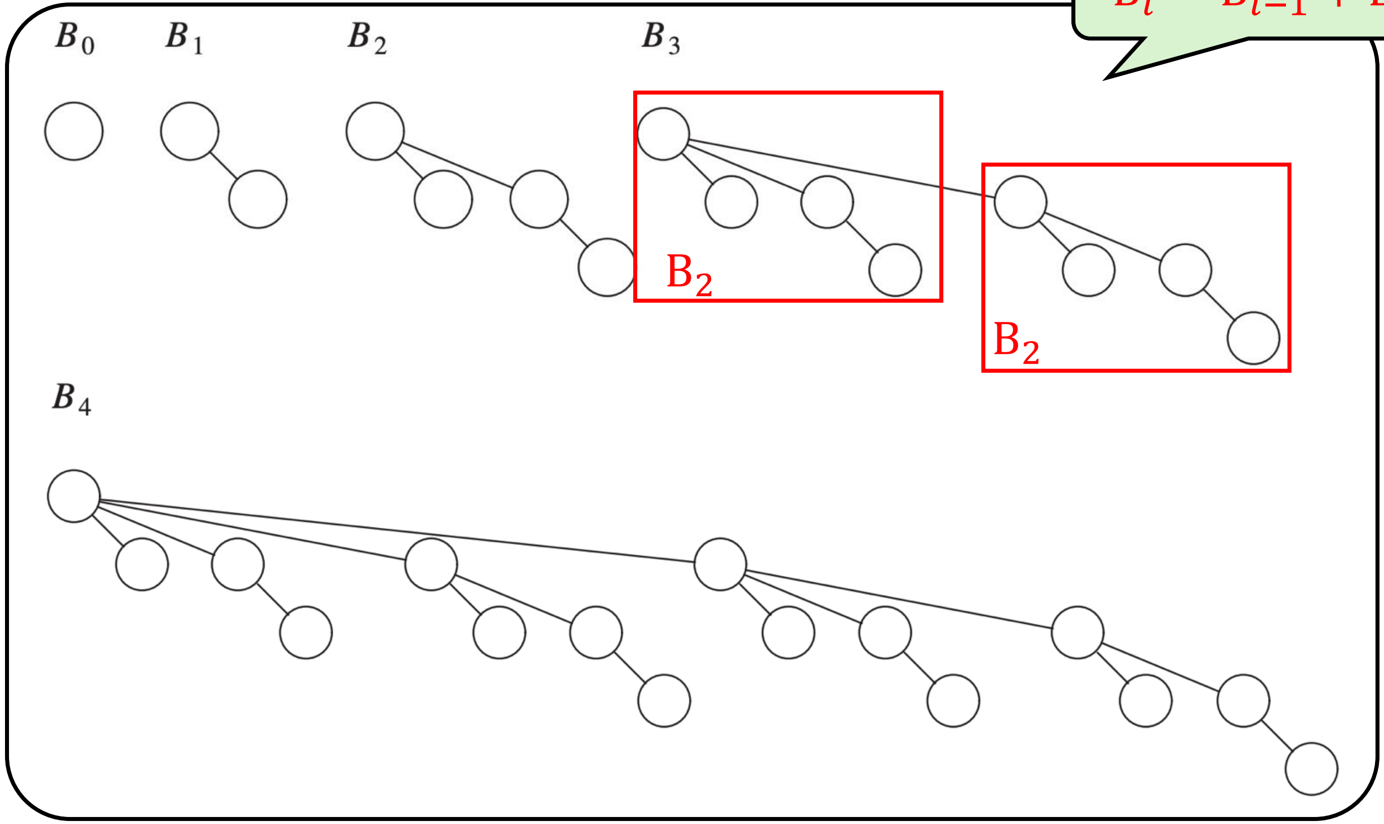


Binomial heaps: example



$\mathrm{B}_i = \mathrm{B}_{i-1} + \mathrm{B}_{i-1}$

Binomial heaps: example



$B_i = B_{i-1} + B_{i-1}$

Binomial heap property

- Lemma: There exists a binomial heap for every positive value of n
- Proof:
  - All values of n can be represented in binary representation
    - Have one binomial tree for each power of two with co-efficient of 1
    - Eg., 10 → $(1010)_2$ → forest contains {B_3, B_1}

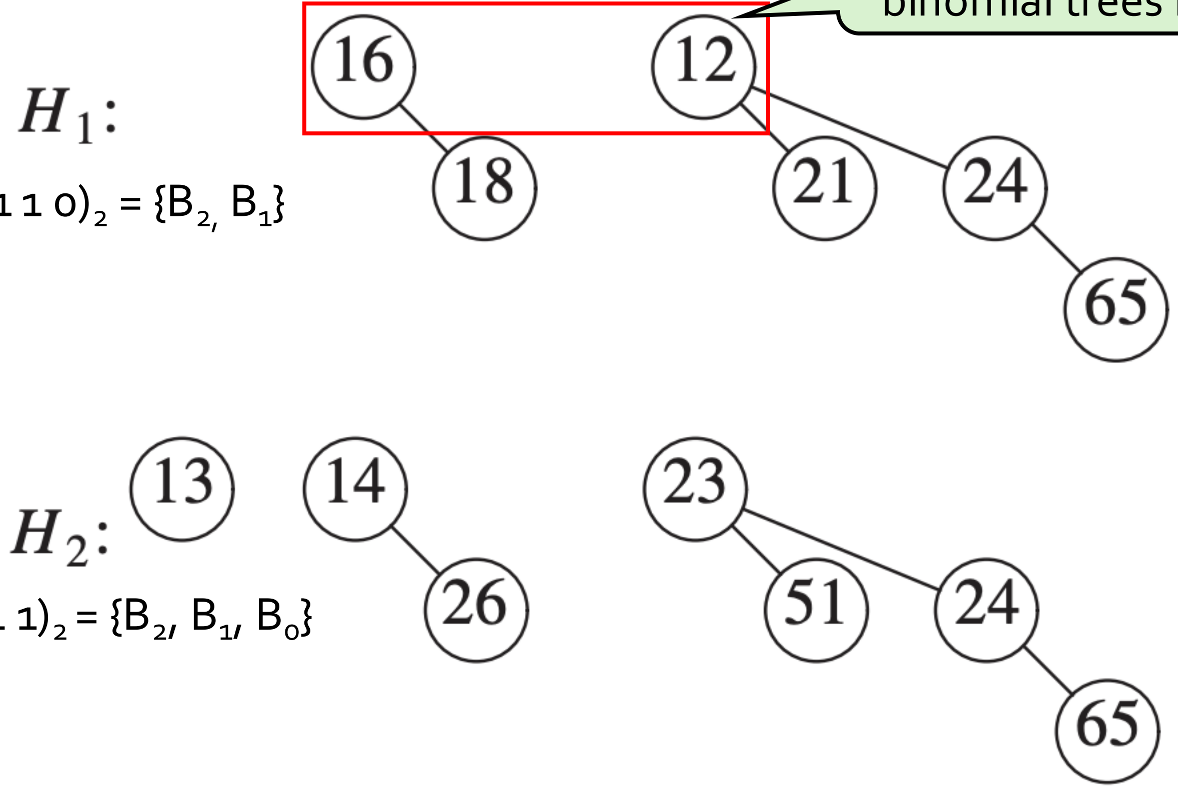
Binomial heaps: heap-order

- Each binomial tree should contain the minimum element at the root of every subtree
    - Just like binary heap, except that the tree here is a binomial tree structure (and not a complete binary tree)
- The order of elements across binomial trees is irrelevant

Definition: binomial heaps

- A binomial heap of n nodes is:
- (Structure property) A forest of binomial trees as described by the binary representation of n
- (Heap-Order Property) Each binomial tree is a min-heap or a max-heap

Binomial heaps: examples



The order of elements across binomial trees is irrelevant

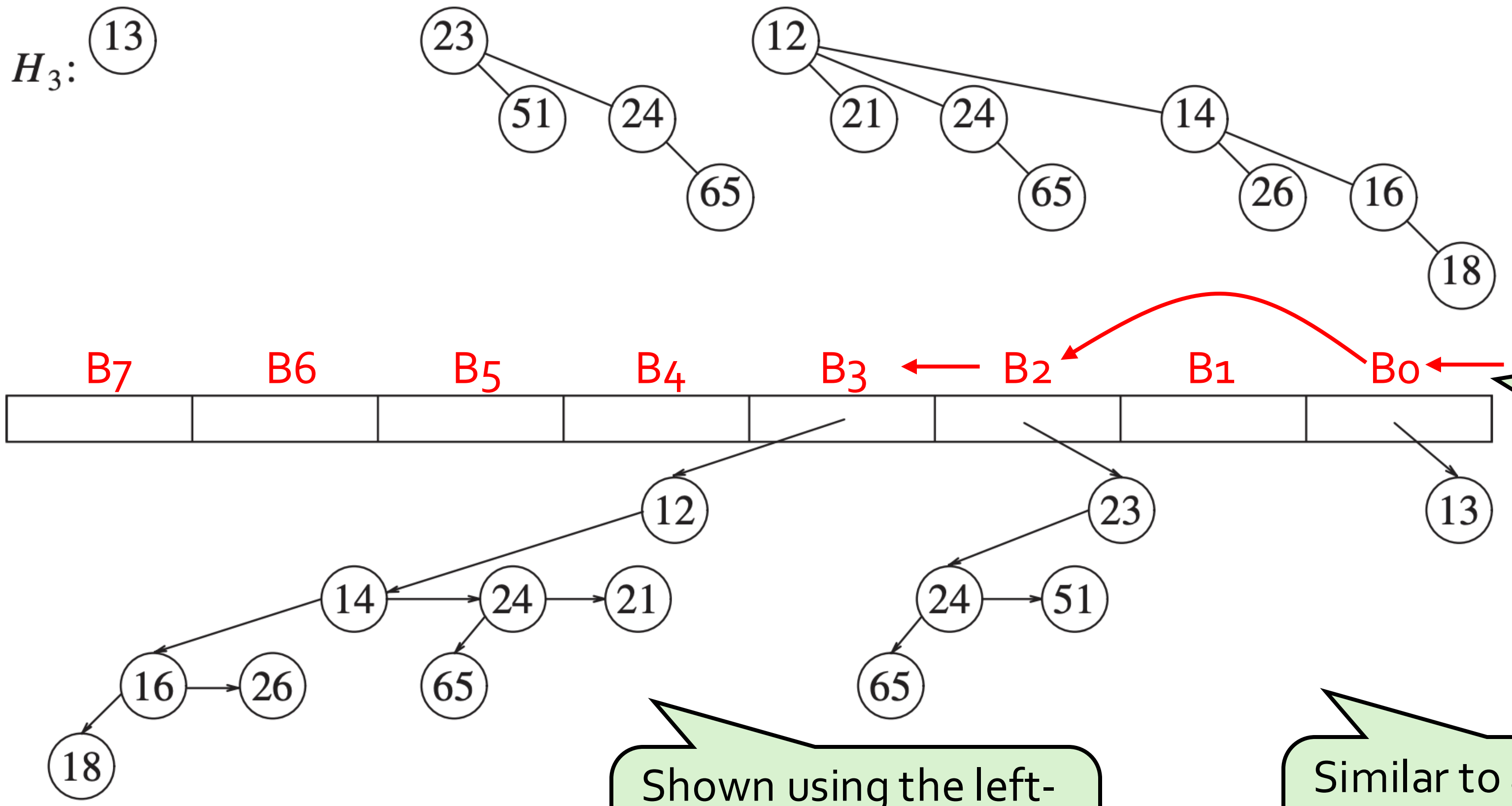
N = 6 = $(1\ 1\ 0)_2$ = {B_2, B_1}

N = 7 = $(1\ 1\ 1)_2$ = {B_2, B_1, B_0}

Key properties

- Could there be multiple trees of the same height in a binomial tree?
  - no
- What is the upper bound of i on B_i in a binomial heap of n nodes?
  - floor(log_2(n))
- Given n, can we tell (for sure) if B_k exists?
  - B_k exists if and only if:
  - the kth least significant bit is 1
  - in the binary representation of n
  - e.g., (1010)2

Binomial heaps: implementation



Maintain a linked list of tree pointers (for the forest)

Figure 6.51 Representation of bin

Shown using the left-child, right-sibling pointer method

Similar to a bit-based representation of a binary number n

Binomial heaps: operations

- deleteMin()
- insert(x)
- merge(H1, H2)

H1, H2: two binomial heaps

Binomial heaps: deleteMin()

- Goal:
  - Given a binomial heap, H, find the minimum and delete it
- Observation:
  - The root of each binomial tree in H contains its minimum element
- Approach: Therefore, return the minimum of all the roots (minimums)
- Complexity: O(log n) comparisons (why?)

Binomial heaps: deleteMin()

- Goal:
  - Given a binomial heap, H, find the minimum and delete it
- Observation:
  - The root of each binomial tree in H contains its minimum element
- Approach: Therefore, return the minimum of all the roots (minimums)
- Complexity: O(log(n)) comparisons (why?)
- because there are at most O(log n) trees

deleteMin() example

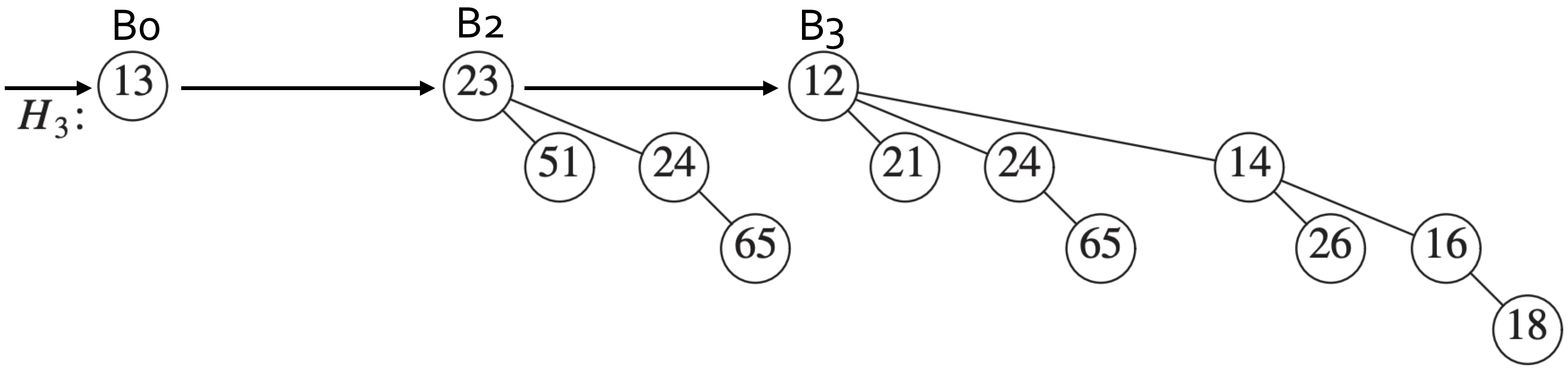


Figure 6.38 Binomial queue H_3: the result of merging H_1 and H_2

Heaps

deleteMin() example

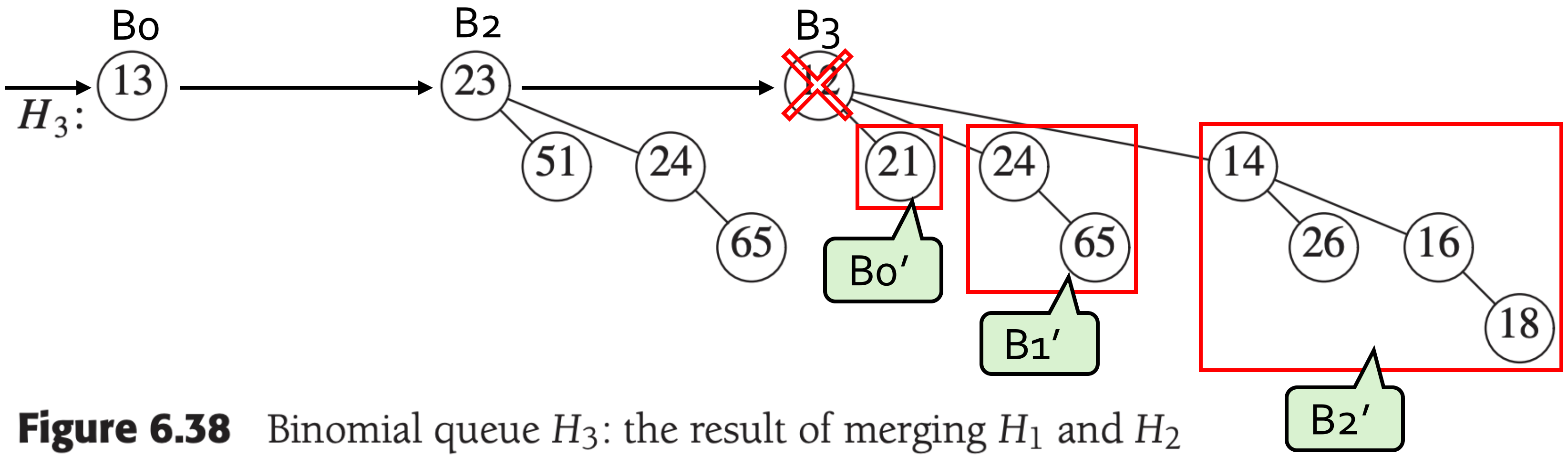


Figure 6.38 Binomial queue H_3: the result of merging H_1 and H_2

New heap : merge { Bo, B2 } & { Bo', B1', B2' }

Binomial heaps: insert(x)

- Goal:
  - To insert a new element x into a binomial heap H
- Observation:
  - Element x can be viewed as a single element binomial heap
- insert (H,x) ←→ merge(H, {x})
- If we decide how to do merge, we will automatically figure out how to implement both insert() and deleteMin()

Binomial heaps: merge(H_1, H_2)

- Let n_1 be the number of nodes in H_1
- Let n_2 be the number of nodes in H_2
- Therefore, the new heap is going to have $n_1 + n_2$ nodes
- Assume $n = n_1 + n_2$
- Logic:
- Merge trees of same height, starting from lowest height trees
- If only one tree of a given height, then just copy that
- Otherwise, need to do carryover (just like adding two binary numbers)

Binomial heaps: merge(H1, H2)

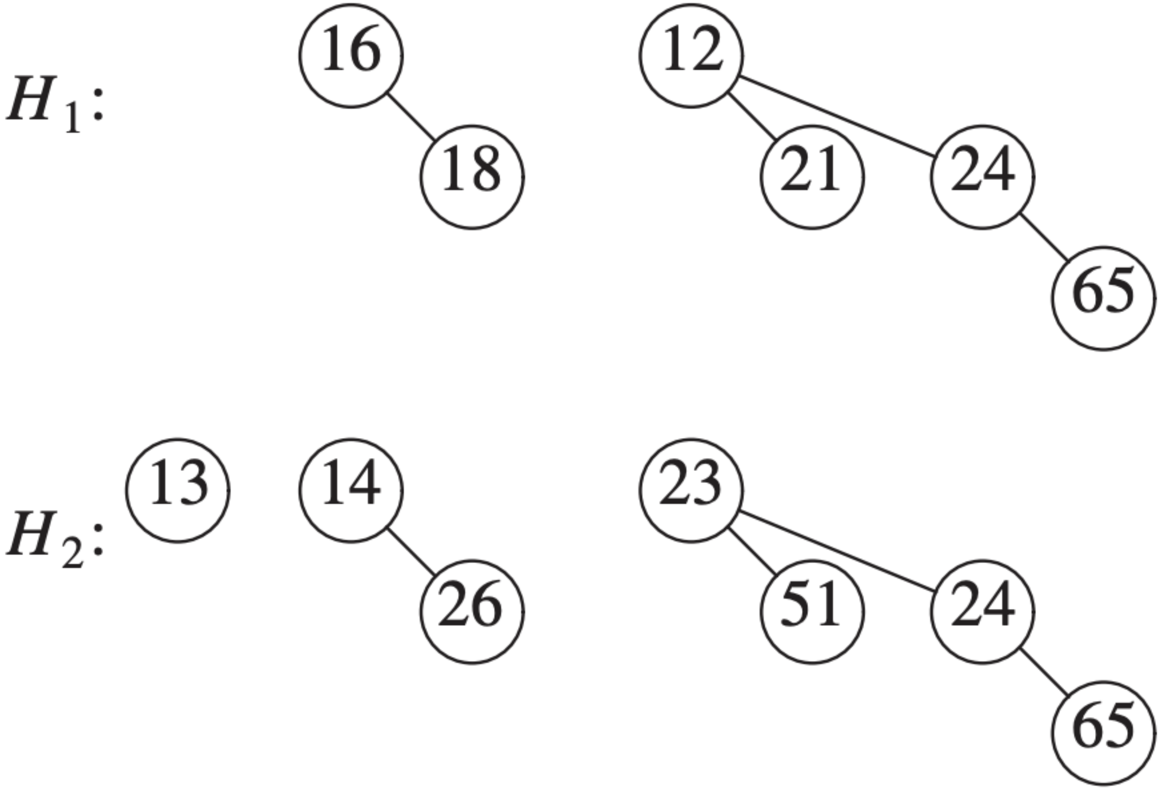


Figure 6.36 Two binomial queues H_1 and H_2

Binomial heaps: merge(H1, H2)

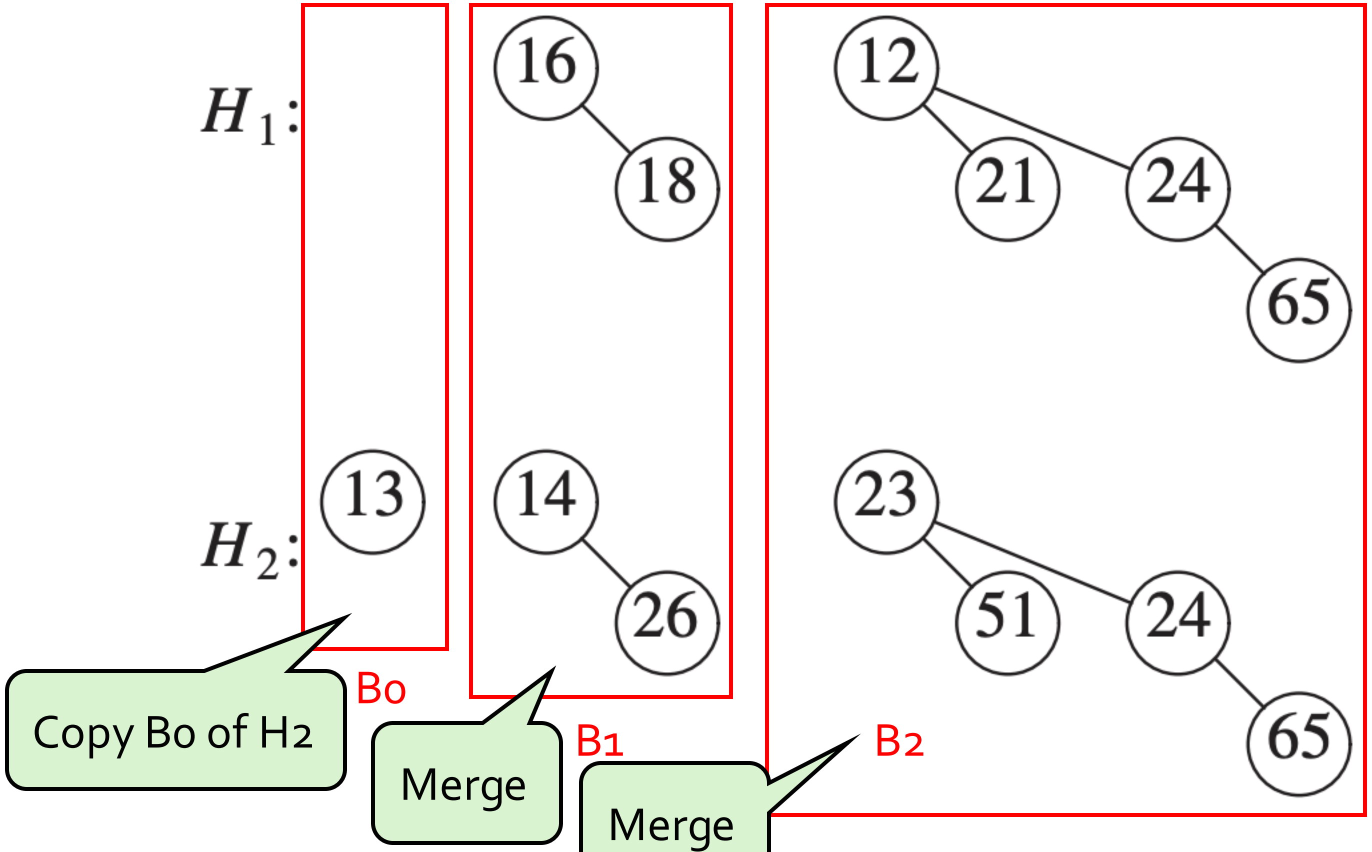
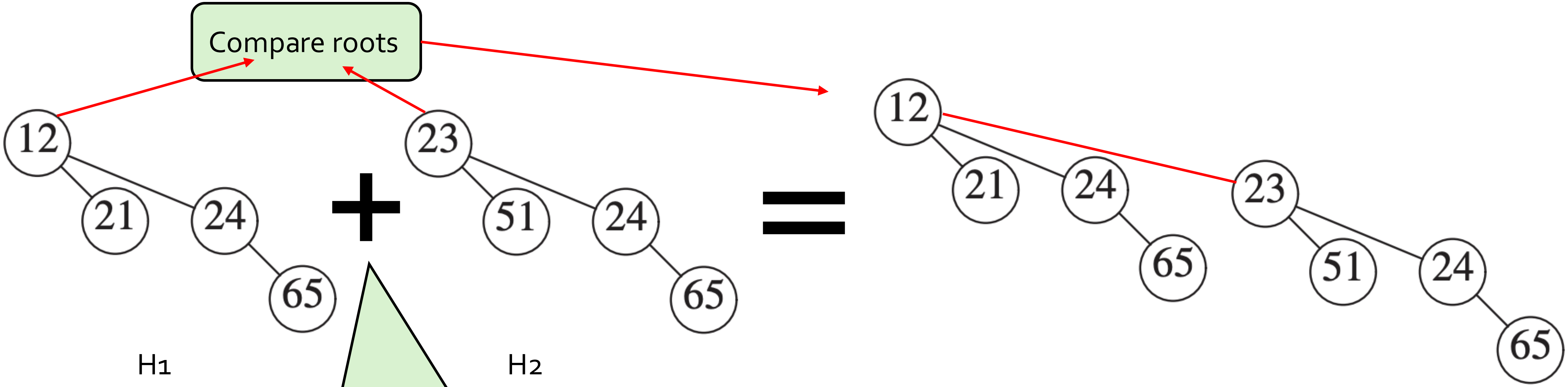


Figure 6.36 Two binomial queues H_1 and H_2

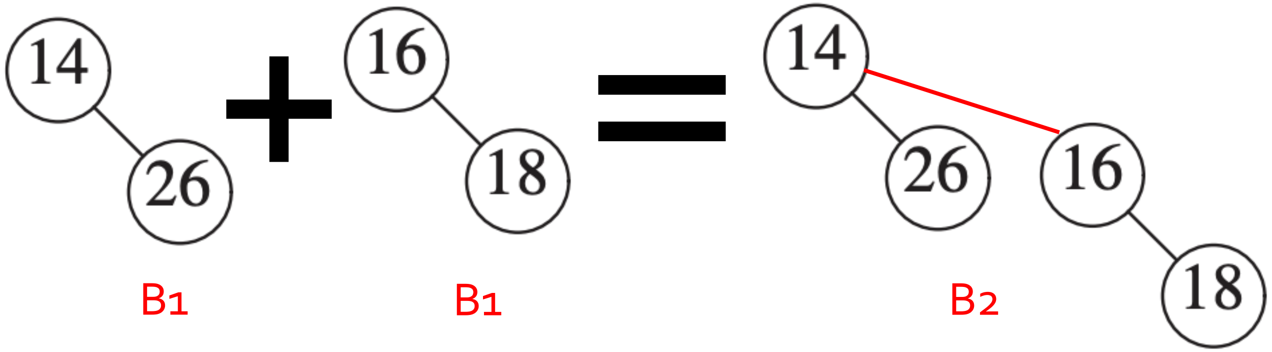
Binomial heaps: merge(H1, H2)



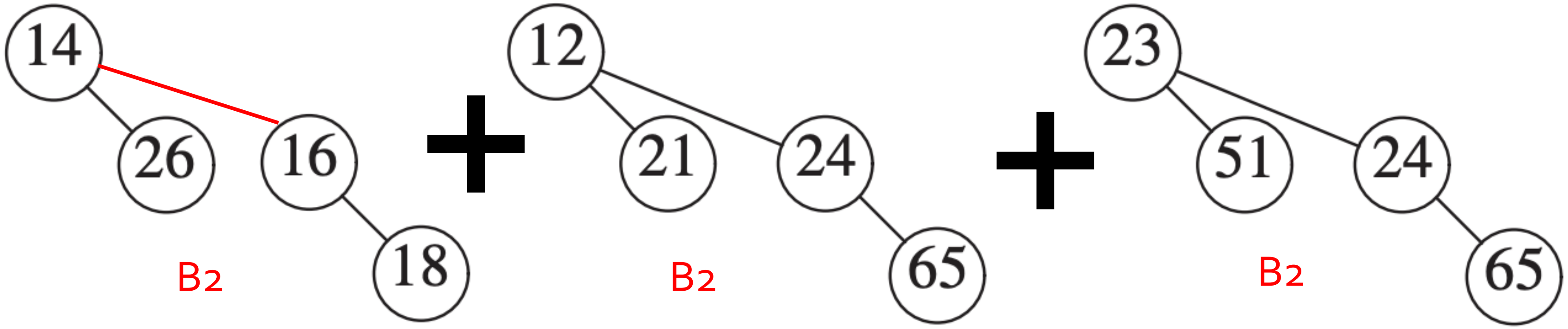
Compare roots

Note:
Merge is defined for only binomial trees with the same height

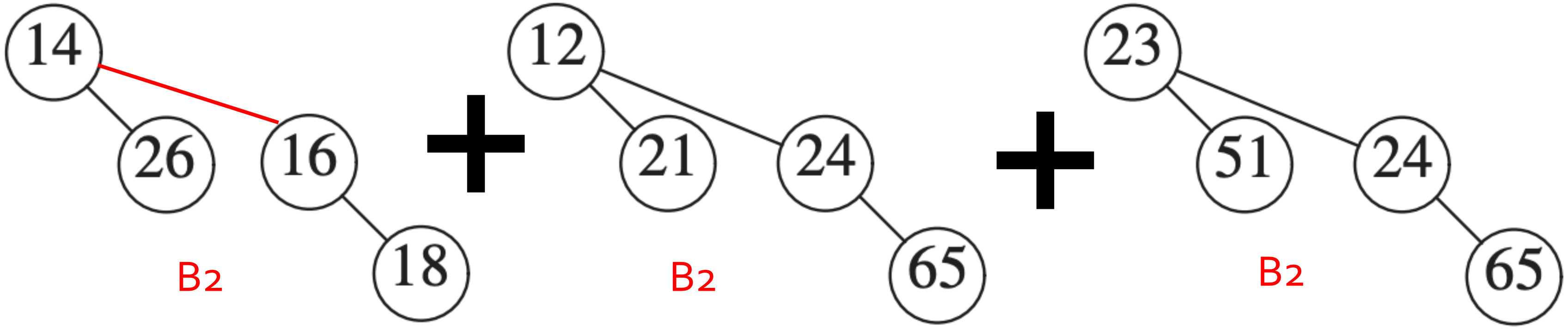
Binomial heaps: merge(H1, H2)



Binomial heaps: merge(H1, H2)



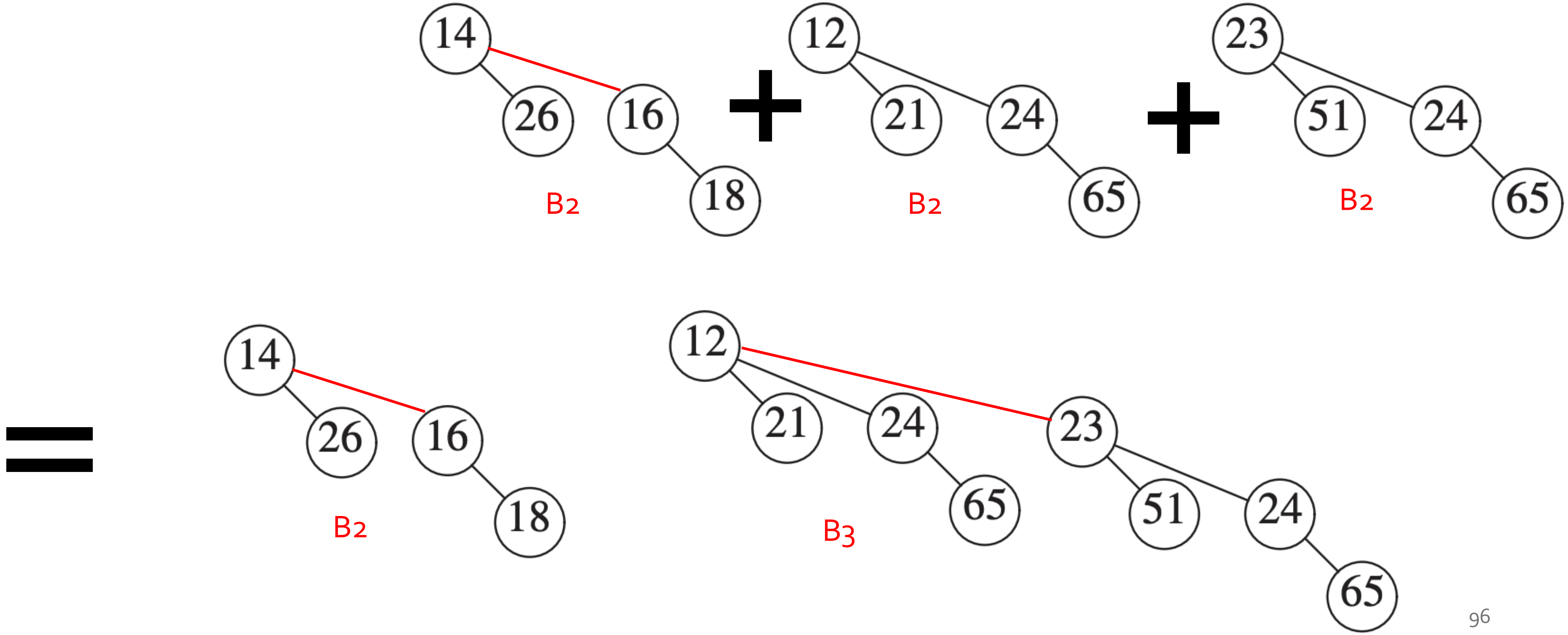
Binomial heaps: merge(H1, H2)



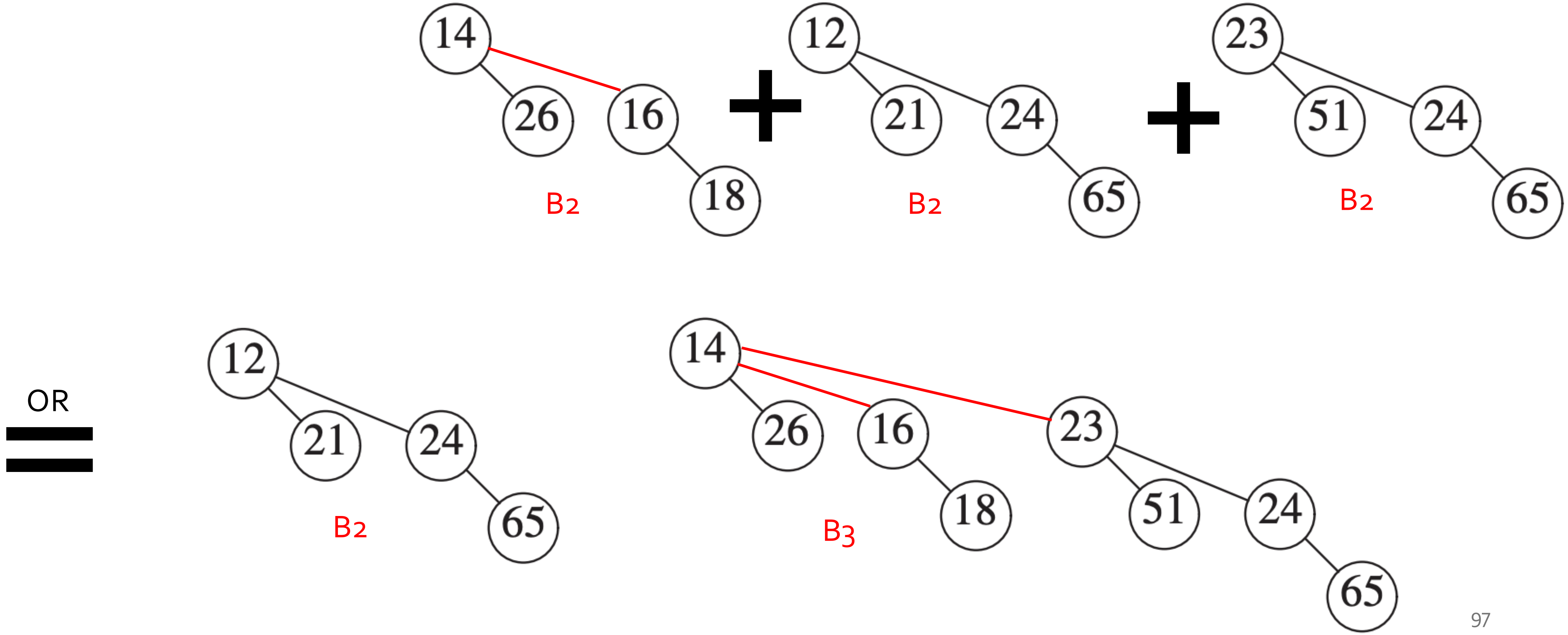
Merge any two and leave the third as carry-over

Heaps

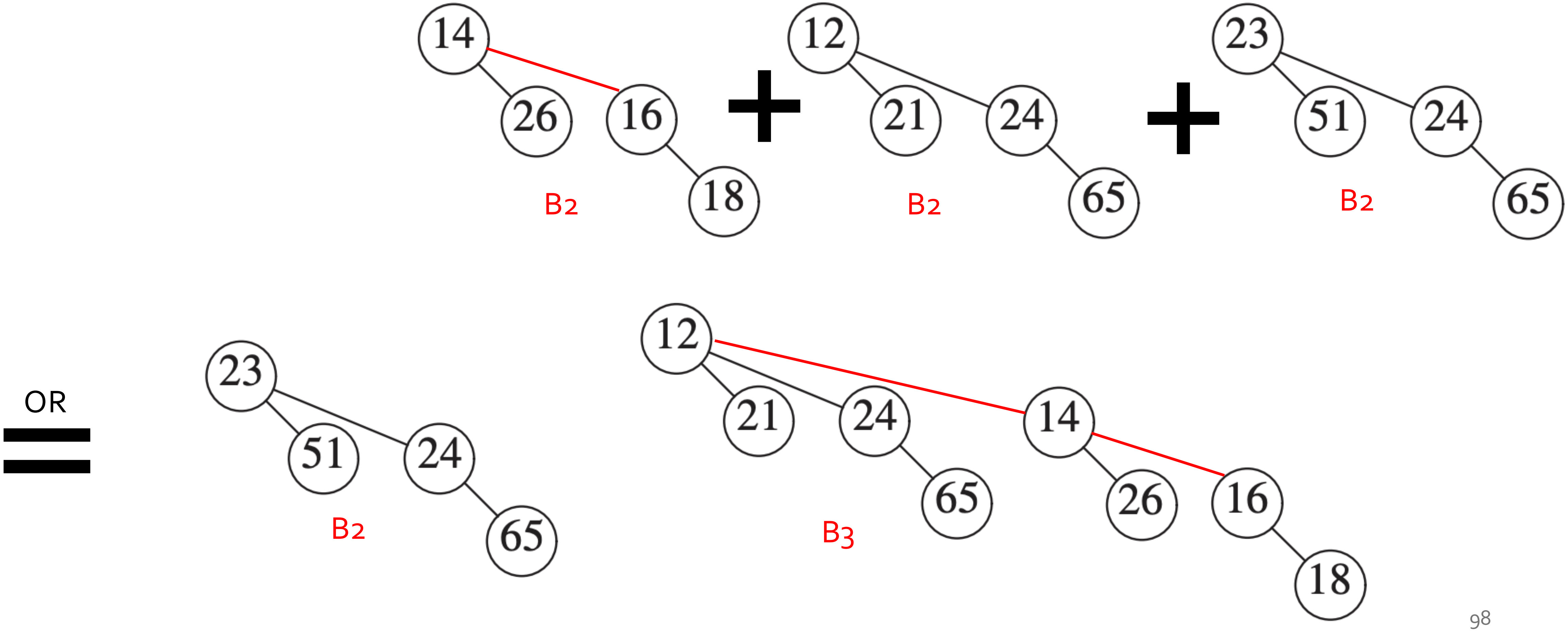
Binomial heaps: merge(H1, H2)



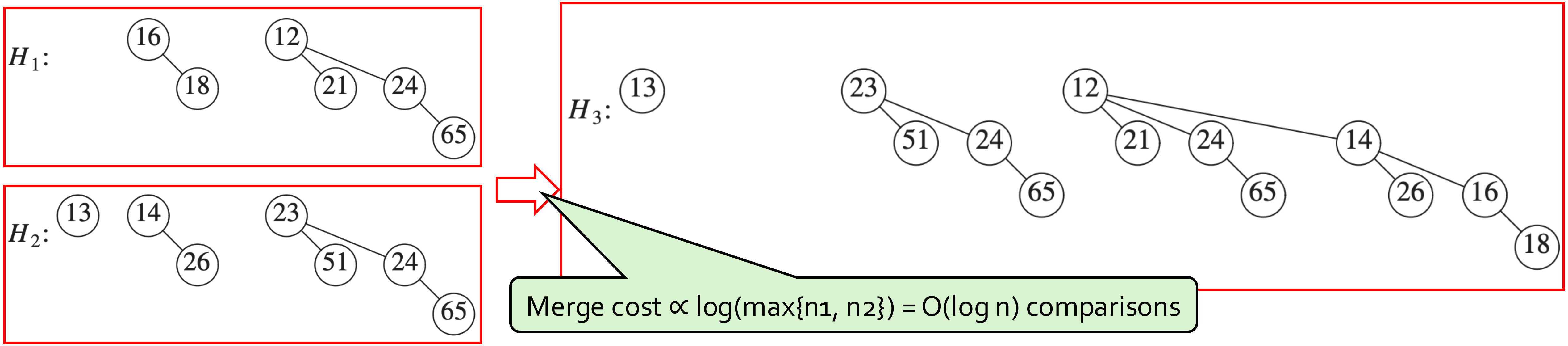
Binomial heaps: merge(H1, H2)



Binomial heaps: merge(H1, H2)



Binomial heaps: merge(H1, H2)



Merge cost ∝ log(max{n1, n2}) = O(log n) comparisons

Merge: time complexity

- Merge takes O(log n) comparisons
- Therefore:
  - insert and deleteMin also take O(log n)
- It can be further proved that an uninterrupted sequence of m insert operations takes only O(m) time per operation, implying O(1) amortize time per insert

Binomial heaps: time complexity

- insert
  - O(lg(n)) worst-case
  - O(1) amortized time if insertion is done in an uninterrupted sequence (i.e., without being intervened by deleteMins)
- deleteMin, findMin
  - O(lg(n)) worst-case
- merge
  - O(lg(n)) worst-case

Binomial heaps: summary

- Binomial heap-based queues maintain the minimum or maximum element of a set
- Support O(log N) operations worst-case
  - Especially merge
- Many applications
- Merge jobs from multiple workers

Time complexity per operation

	findMin	insert	deleteMin	merge
Binary heap	O(1)	O(log(n)) worst-case O(1) amortized for buildHeap	O(log(n))	O(n)
Leftist heap	O(1)	O(log(n))	O(log(n))	O(log(n))
Skew heap	O(1)	O(log(n))	O(log(n))	O(log(n))
Binomial heap	O(1)	O(log(n)) worst-case O(1) amortized for sequence of n inserts	O(log(n))	O(log(n))
Fibonacci heap	O(1)	O(1)	O(log(n))	O(1)