Priority Queues (Heaps)



Advanced Data Structures

1

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

2

Priority Queues



* Main operations
  + **insert** (i.e., enqueue)
  + **deleteMin** (i.e., dequeue)
    - Finds the minimum element in the queue, deletes it from the queue, and returns it
* Performance
  + Goal is for operations to be fast
  + Will be able to achieve O(log2N) time insert/deleteMin amortized over multiple operations
  + Will be able to achieve O(1) time inserts amortized over multiple insertions

3

Simple Implementations



* Unordered list
  + O(1) insert
  + O(N) deleteMin
* Ordered list
  + O(N) insert
  + O(1) deleteMin
* Balanced BST
  + O(log2N) insert and deleteMin
* Observation: We don’t need to keep the priority queue completely ordered

4

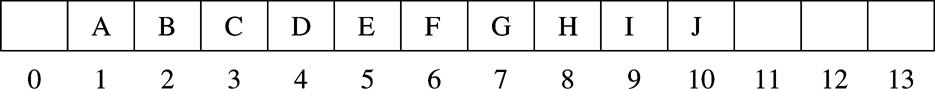
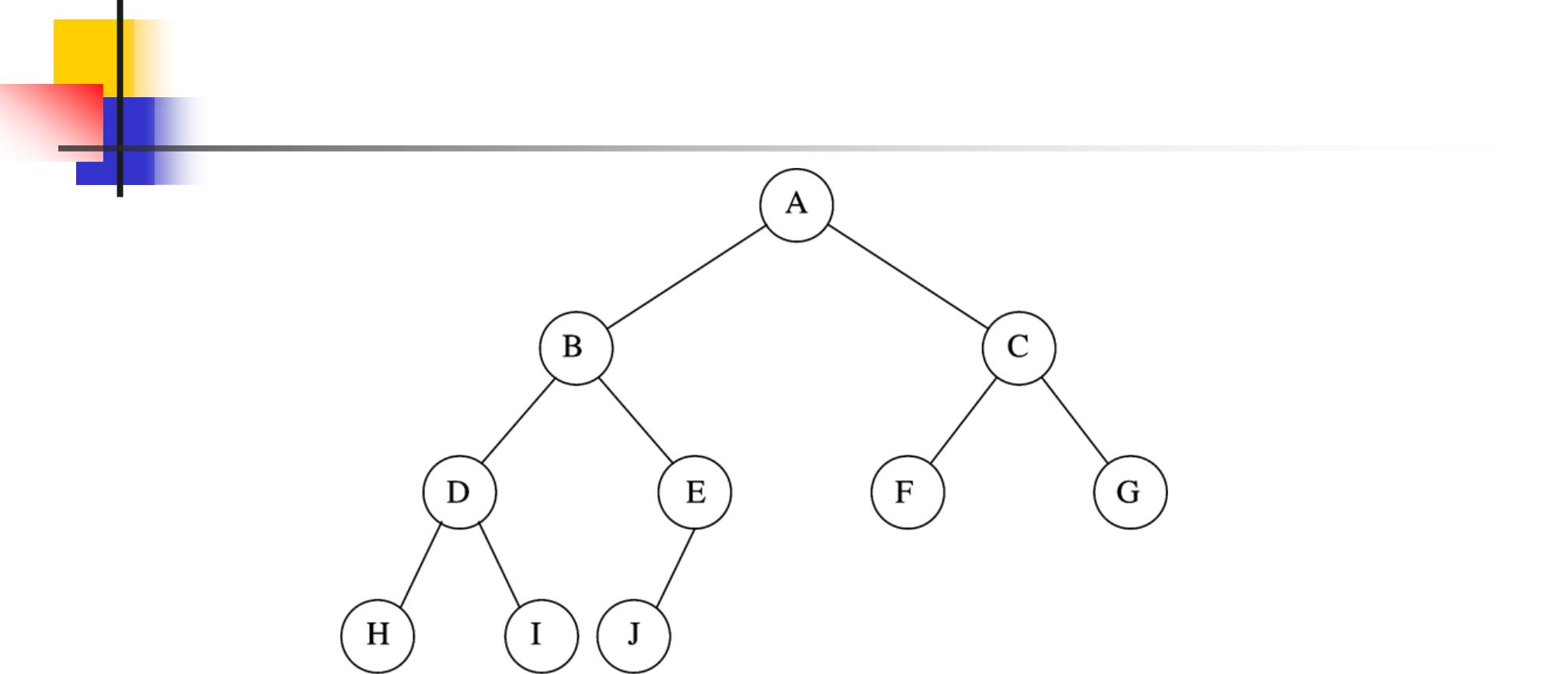
Binary Heap



* A binary heap is a binary tree with two properties
* Structure property
  + A binary heap is a complete binary tree
    - Each level is completely filled
    - Bottom level may be partially filled from left to right
* Height of a complete binary tree with N elements is log2 *N*

5

Binary Heap Example



6

Binary Heap



* Heap-order property
  + For every node X, key(parent(X)) ≤ key(X)
  + Except root node, which has no parent
* Thus, minimum key always at root
  + Or, maximum, if you choose
* Insert and deleteMin must maintain heap-order property

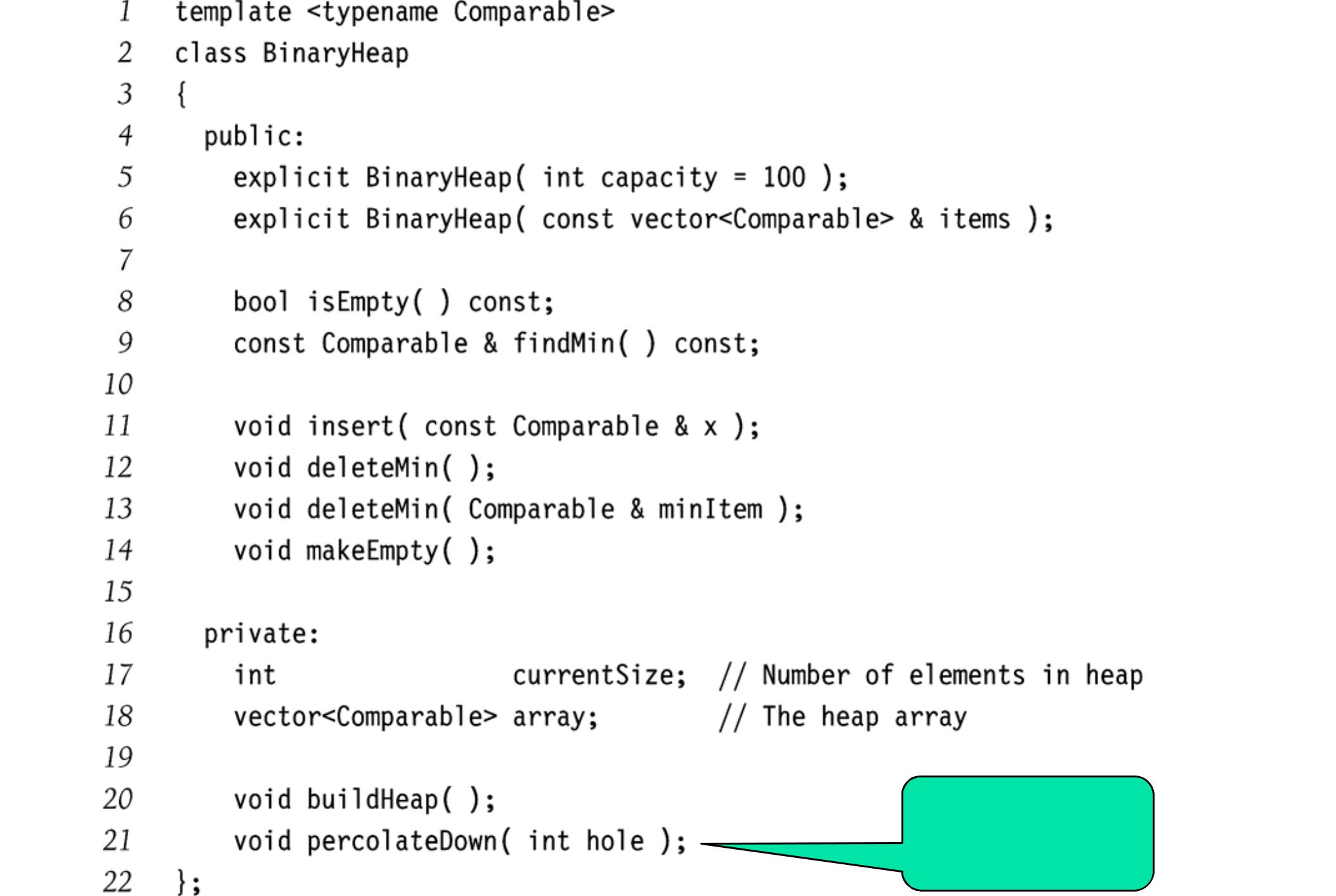
7

Implementing Complete Binary Trees as Arrays



* Given element at position i in the array
  + i’s left child is at position 2i
  + i’s right child is at position 2i+1
  + i’s parent is at position *i* / 2

8



Fix heap after

deleteMin

9

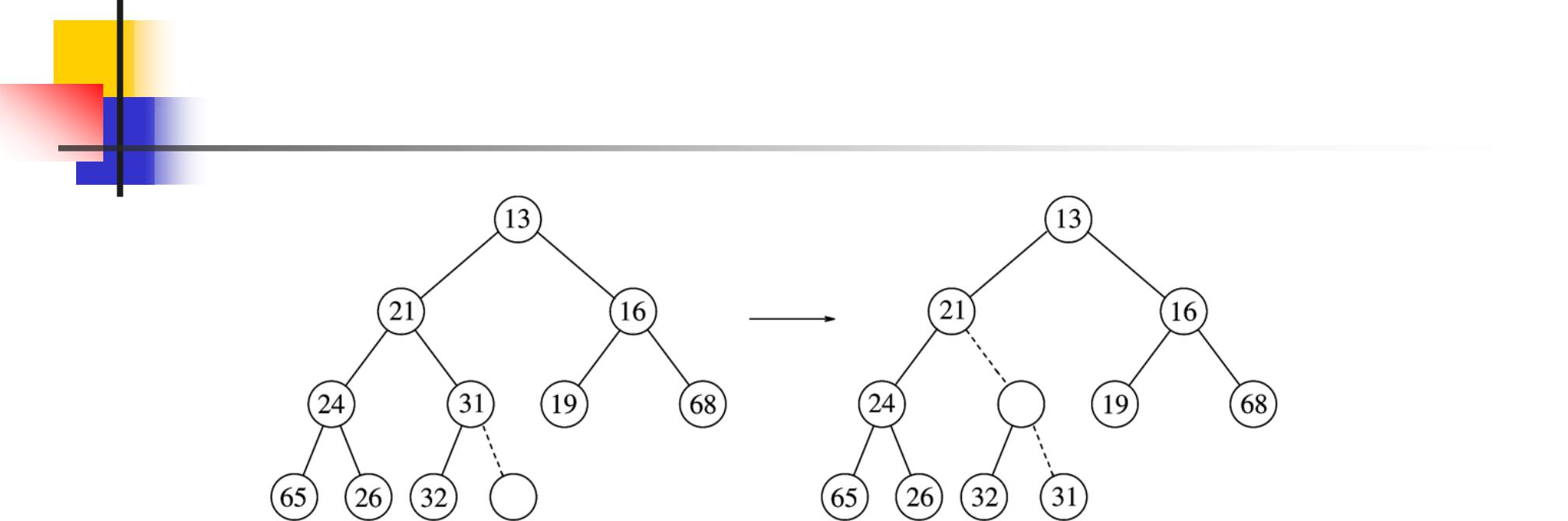
Heap Insert



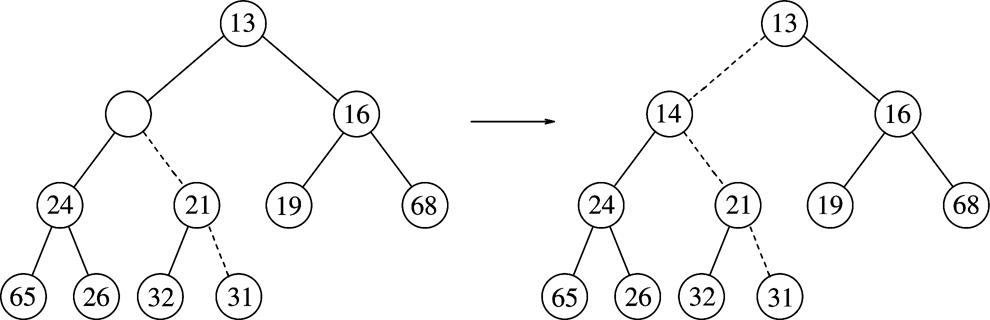
* Insert new element into the heap at the next available slot (“hole”)
  + According to maintaining a complete binary tree
* Then, “percolate” the element up the heap while heap-order property not satisfied

10

Heap Insert: Example

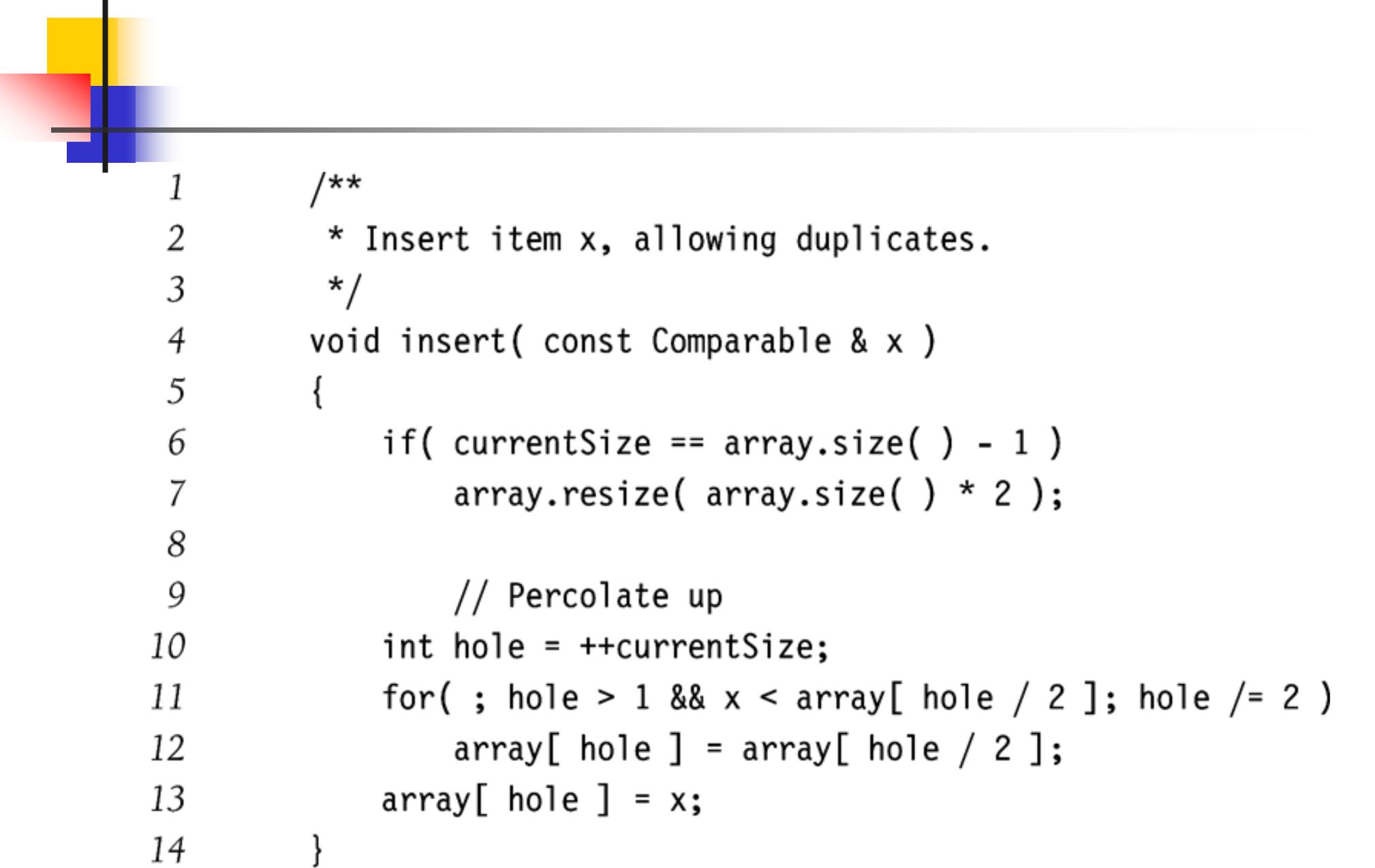


Insert 14:



11

Heap Insert: Implementation



12

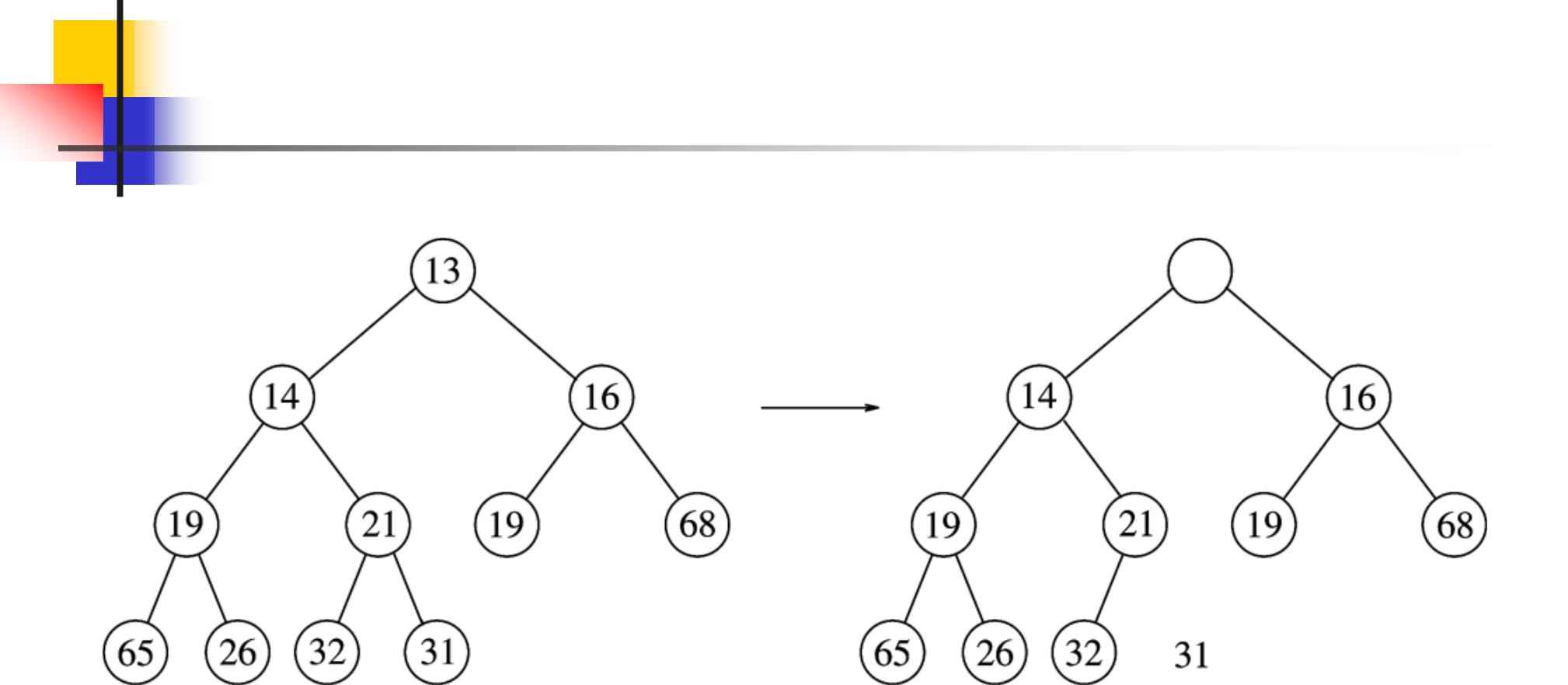
Heap DeleteMin



* 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

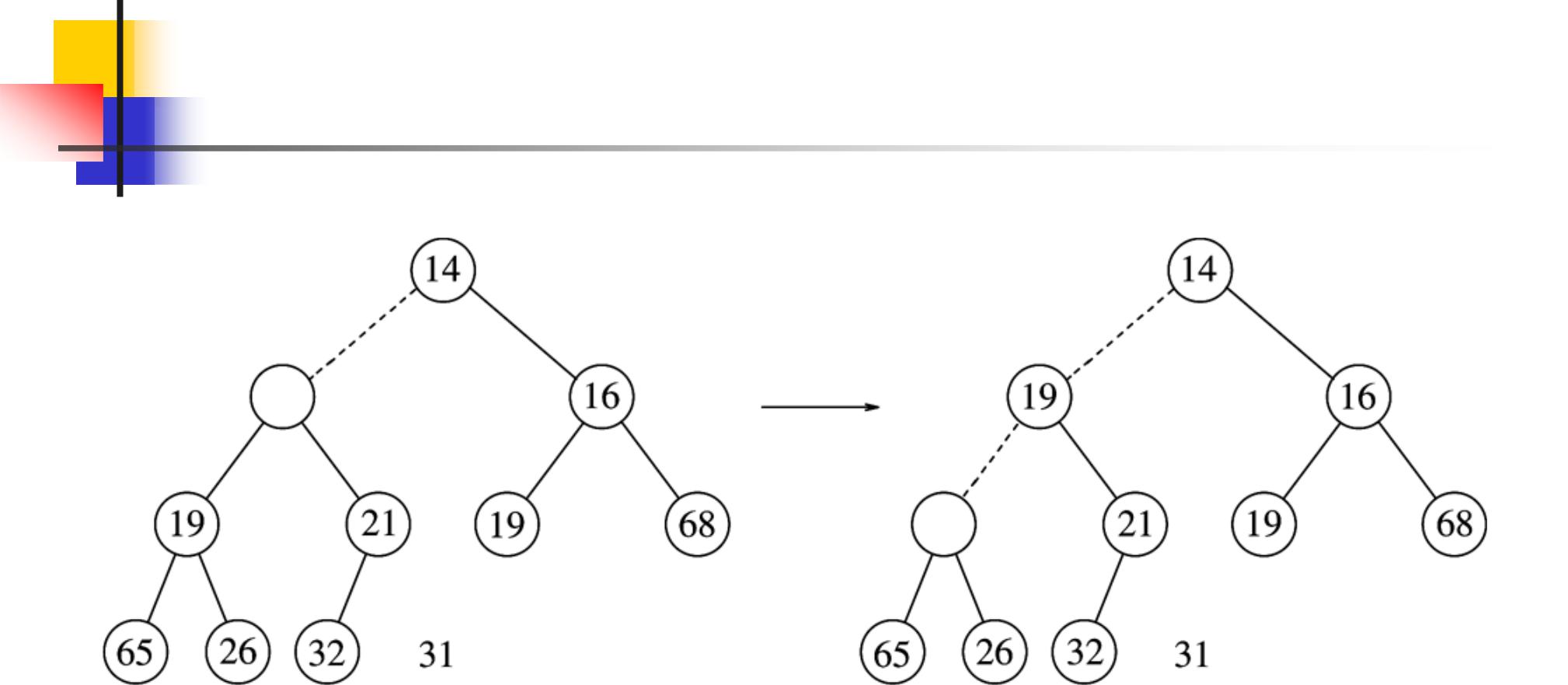
13

Heap DeleteMin: Example



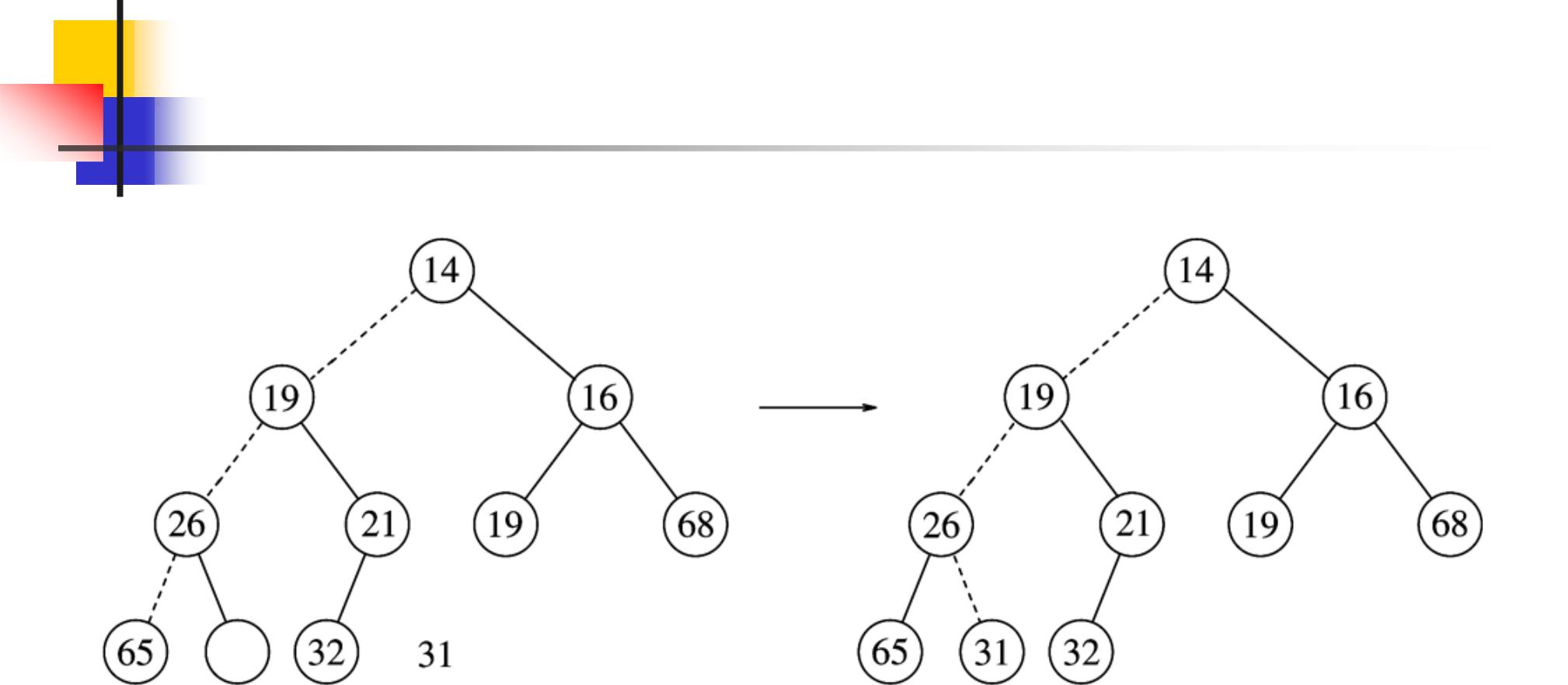
14

Heap DeleteMin: Example



15

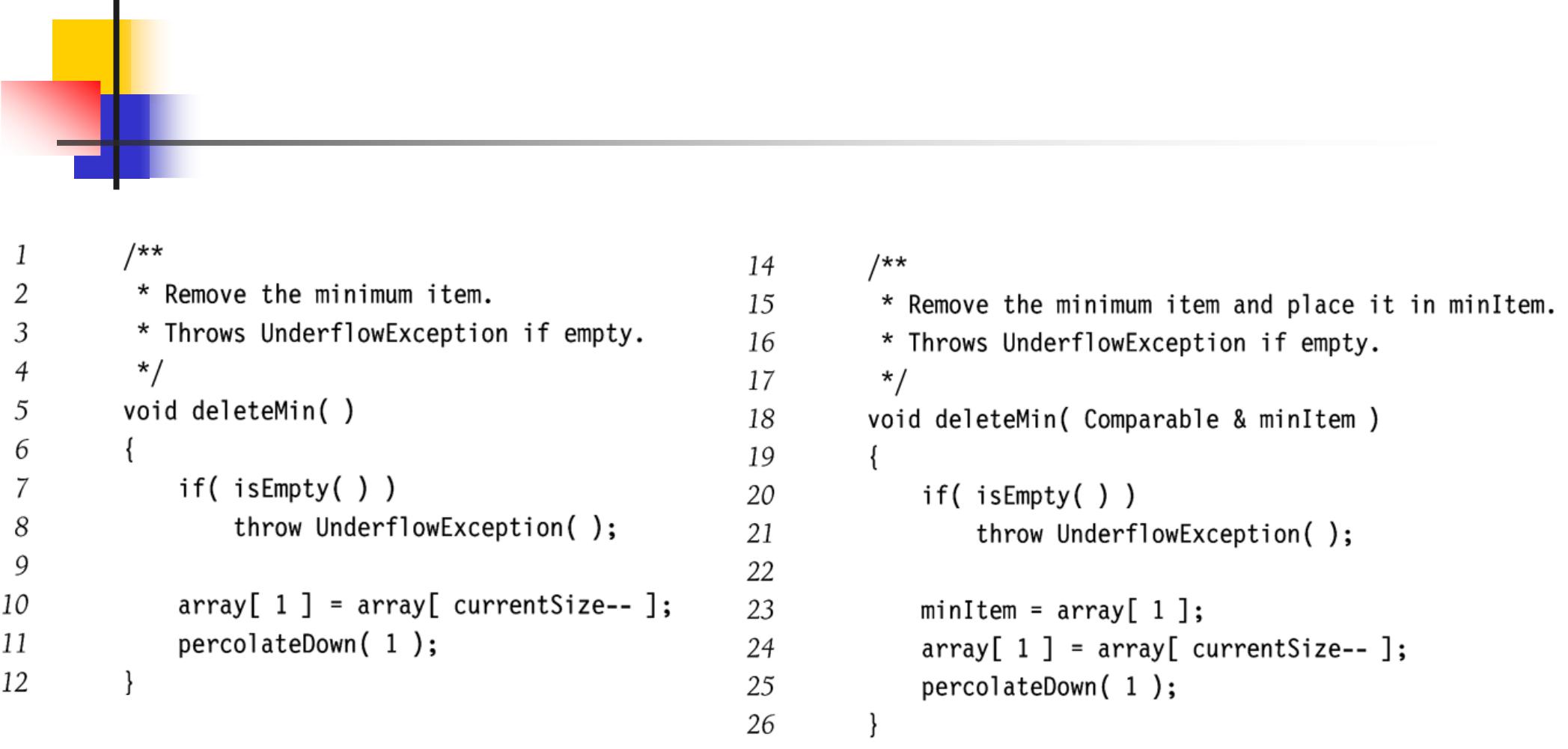
Heap DeleteMin: Example



16

Heap DeleteMin:

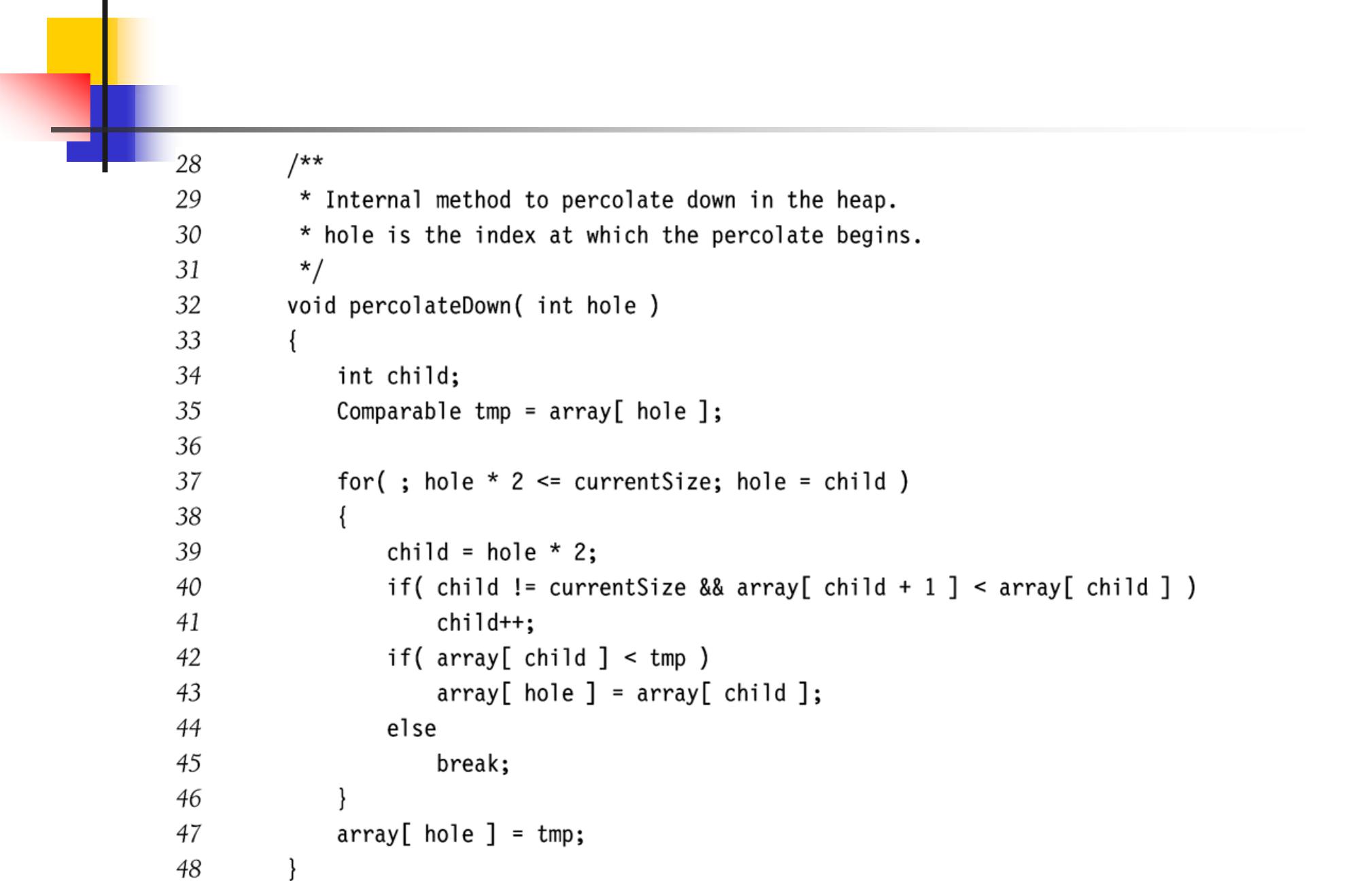
Implementation



17

Heap DeleteMin:

Implementation



18

Other Heap Operations



* decreaseKey(p,v)
  + Lowers value of item p to v
  + Need to percolate up
  + E.g., change job priority
* increaseKey(p,v)
  + Increases value of item p to v
  + Need to percolate down
* remove(p)
  + First, decreaseKey(p,-∞)
  + Then, deleteMin
  + E.g., terminate job

19

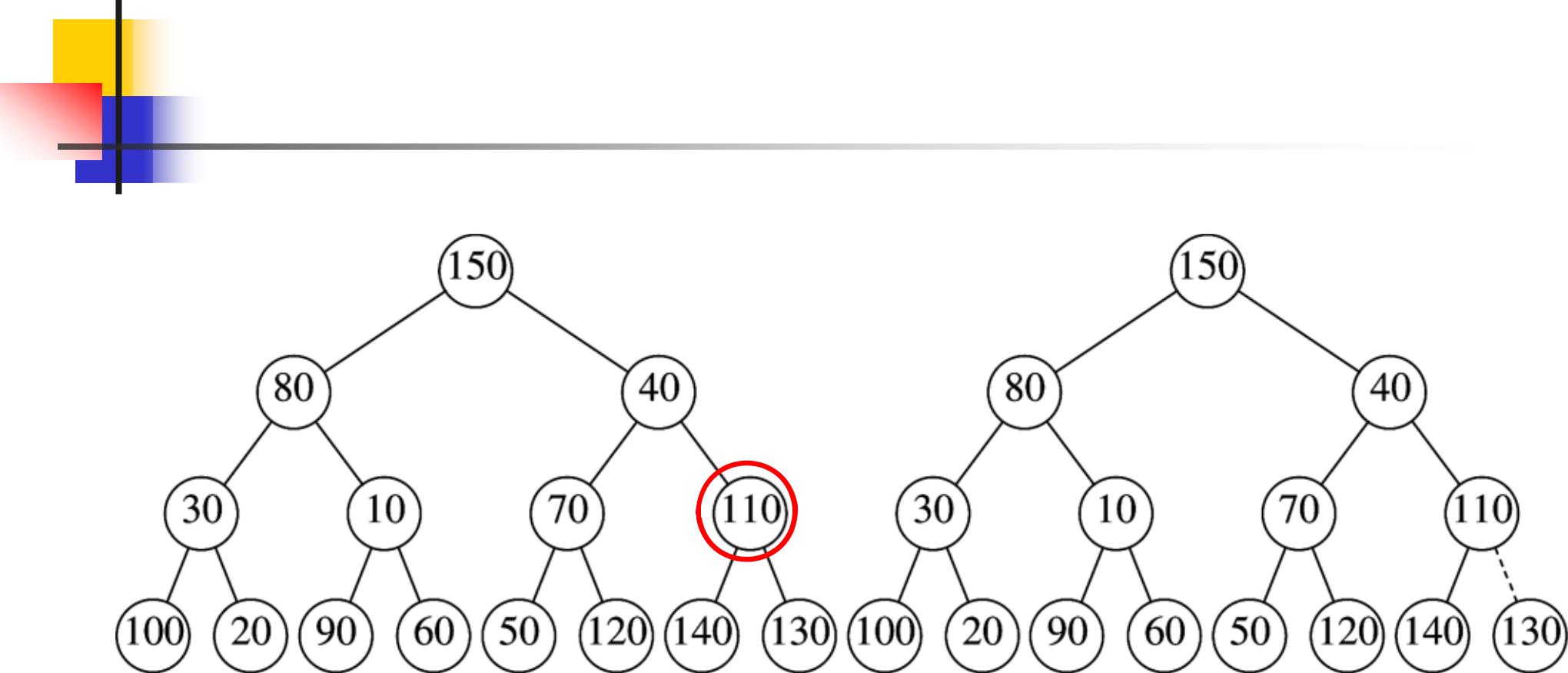
Building a Heap



* Construct heap from initial set of N items
* Solution 1
  + Perform N inserts
  + O(N) average case, but O(N log2 N) worst-case
* Solution 2
  + Assume initial set is a heap
  + Perform a percolate-down from each internal node (H[size/2] to H[1])

20

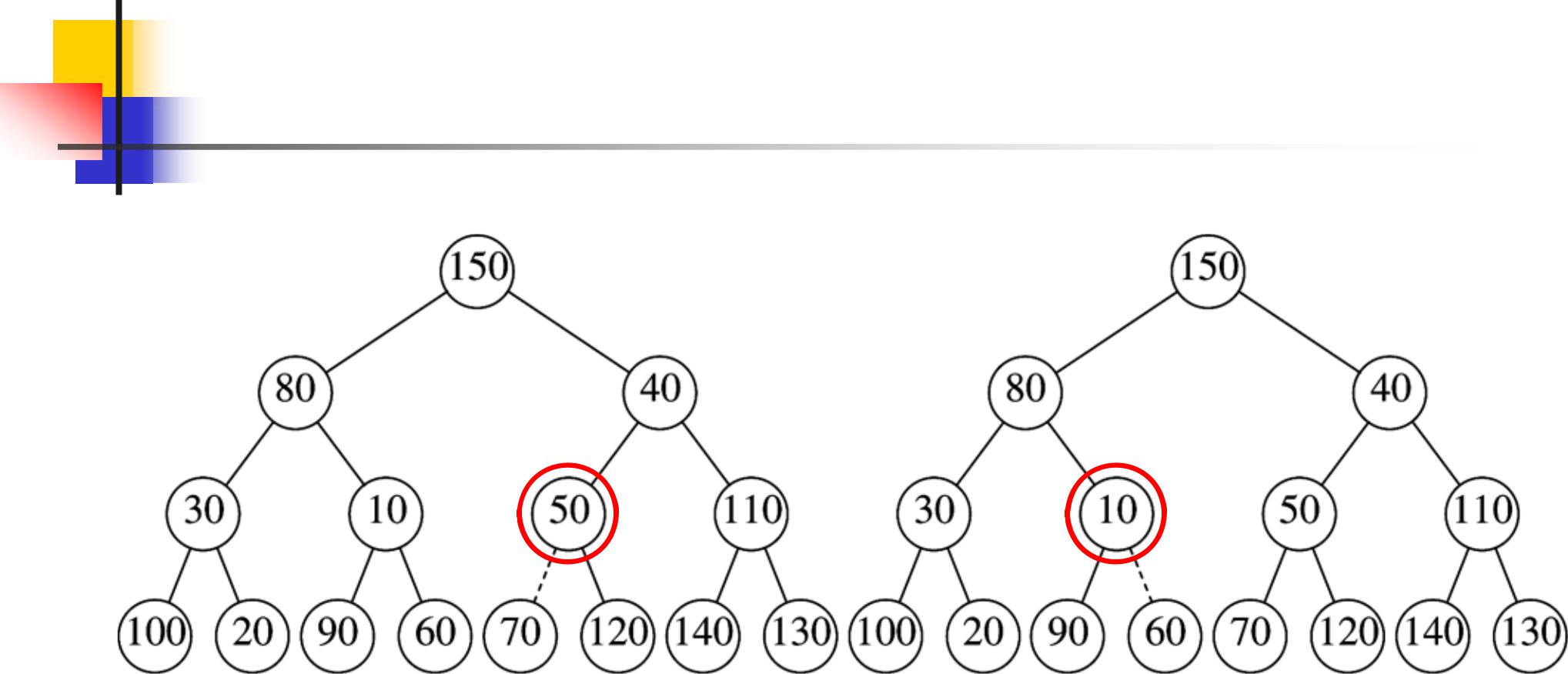
BuildHeap Example



Leaves are all valid heaps

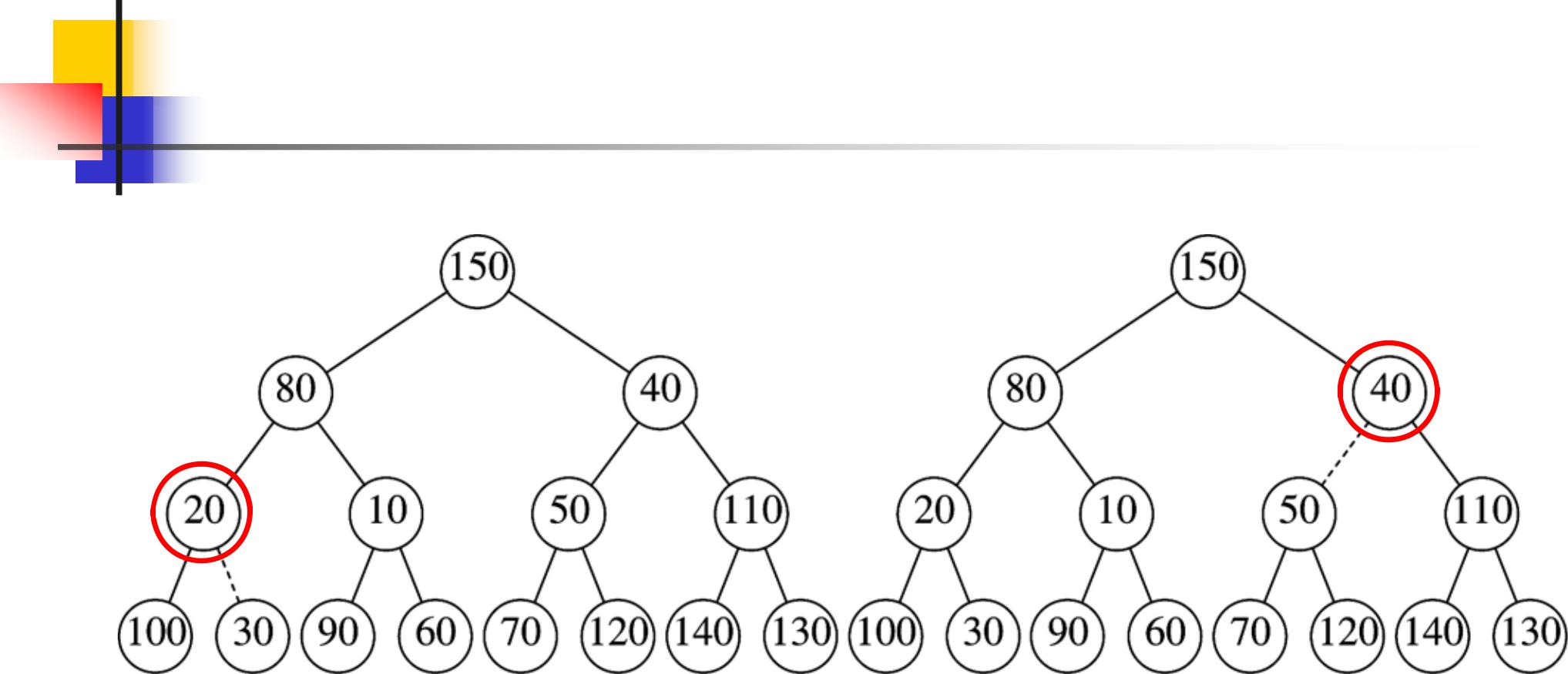
21

BuildHeap Example



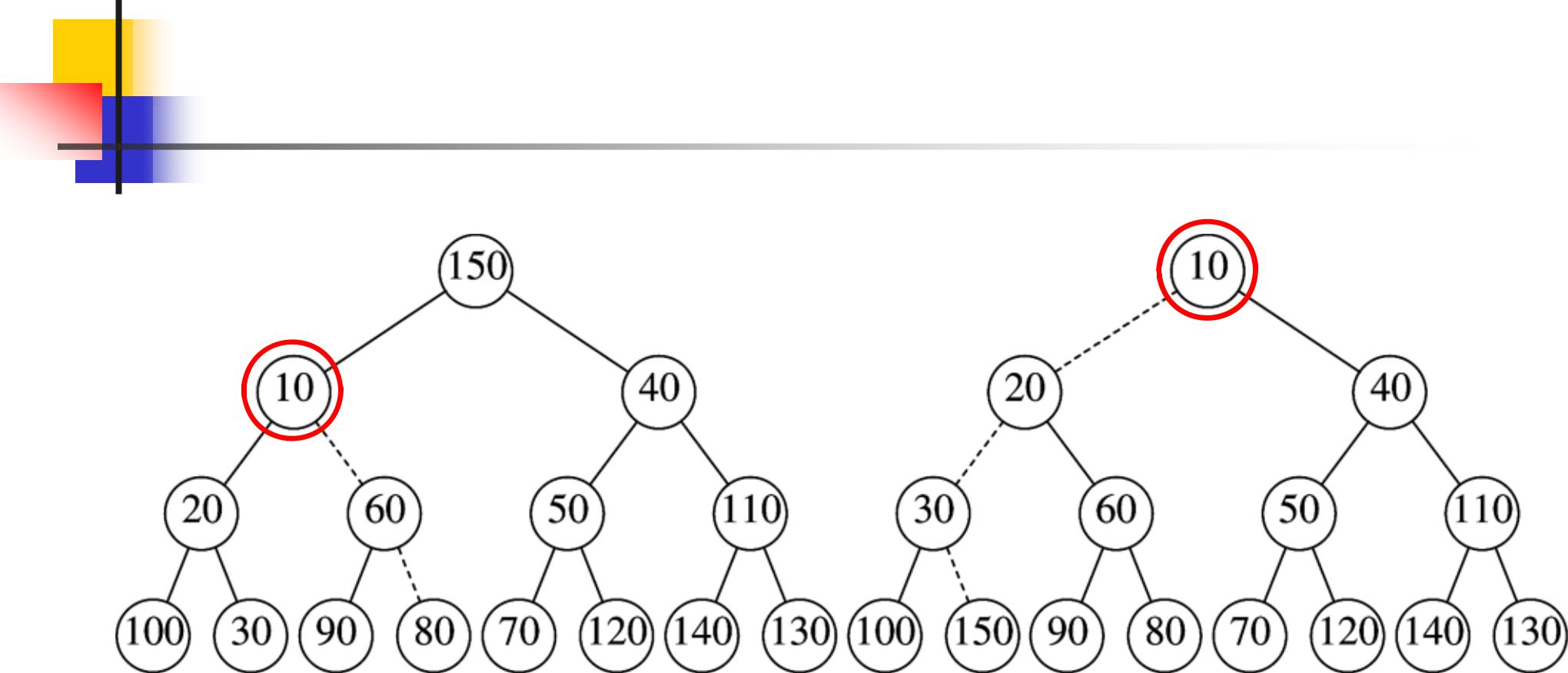
22

BuildHeap Example



23

BuildHeap Example



24

BuildHeap Implementation



25

BuildHeap Analysis



* Running time of buildHeap proportional to sum of the heights of the nodes
* Theorem 6.1
  + For the perfect binary tree of height h containing 2h+1 – 1 nodes, the sum of heights of the nodes is 2h+1 – 1 – (h + 1)
* Since N = 2h+1 – 1, then sum of heights is O(N)
* Slightly better for complete binary tree

26

Binary Heap Operations Worst-case Analysis



* Height of heap is log2 *N*
* insert: O(log2N)
  + 2.607 comparisons on average, i.e., O(1)
* deleteMin: O(log2N)
* decreaseKey: O(log2N)
* increaseKey: O(log2N)
* remove: O(log2N)
* buildHeap: O(N)

27

Applications



* Operating system scheduling
  + Process jobs by priority
* Graph algorithms
  + Find the least-cost, neighboring vertex
* Event simulation
  + Instead of checking for events at each time click, look up next event to happen

28

Priority Queues: Alternatives to Binary Heaps



* d-Heap
  + Each node has d children
  + insert in O(logd N) time
  + deleteMin in O(d logd N) time
* Binary heaps are 2-Heaps

29

Mergeable Heaps



* Heap merge operation
  + Useful for many applications
  + Merge two (or more) heaps into one
  + Identify new minimum element
  + Maintain heap-order property
  + Merge in O(log N) time
  + Still support insert and deleteMin in O(log N) time
    - Insert = merge existing heap with one-element heap
* d-Heaps require O(N) time to merge

30

Leftist Heaps



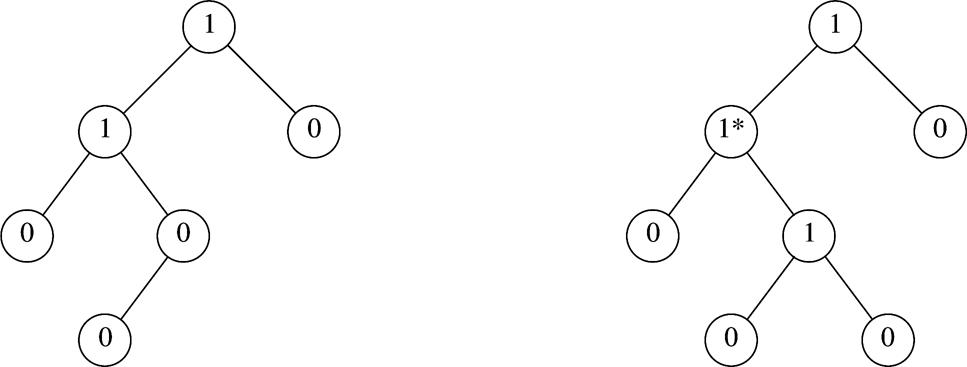
* Null path length npl(X) of node X
  + Length of the shortest path from X to a node without two children
* Leftist heap property
  + For every node X in heap, npl(leftChild(X)) ≥ npl(rightChild(X))
* Leftist heaps have deep left subtrees and shallow right subtrees
  + Thus if operations reside in right subtree, they will be faster

31

Leftist Heaps



npl(X) shown in nodes



Leftist heap Not a leftist heap

32

Leftist Heaps



* Theorem 6.2
  + A leftist tree with r nodes on the right path must have at least 2r – 1 nodes.
* Thus, a leftist tree with N nodes has a right path with at most log(*N* +1) nodes

33

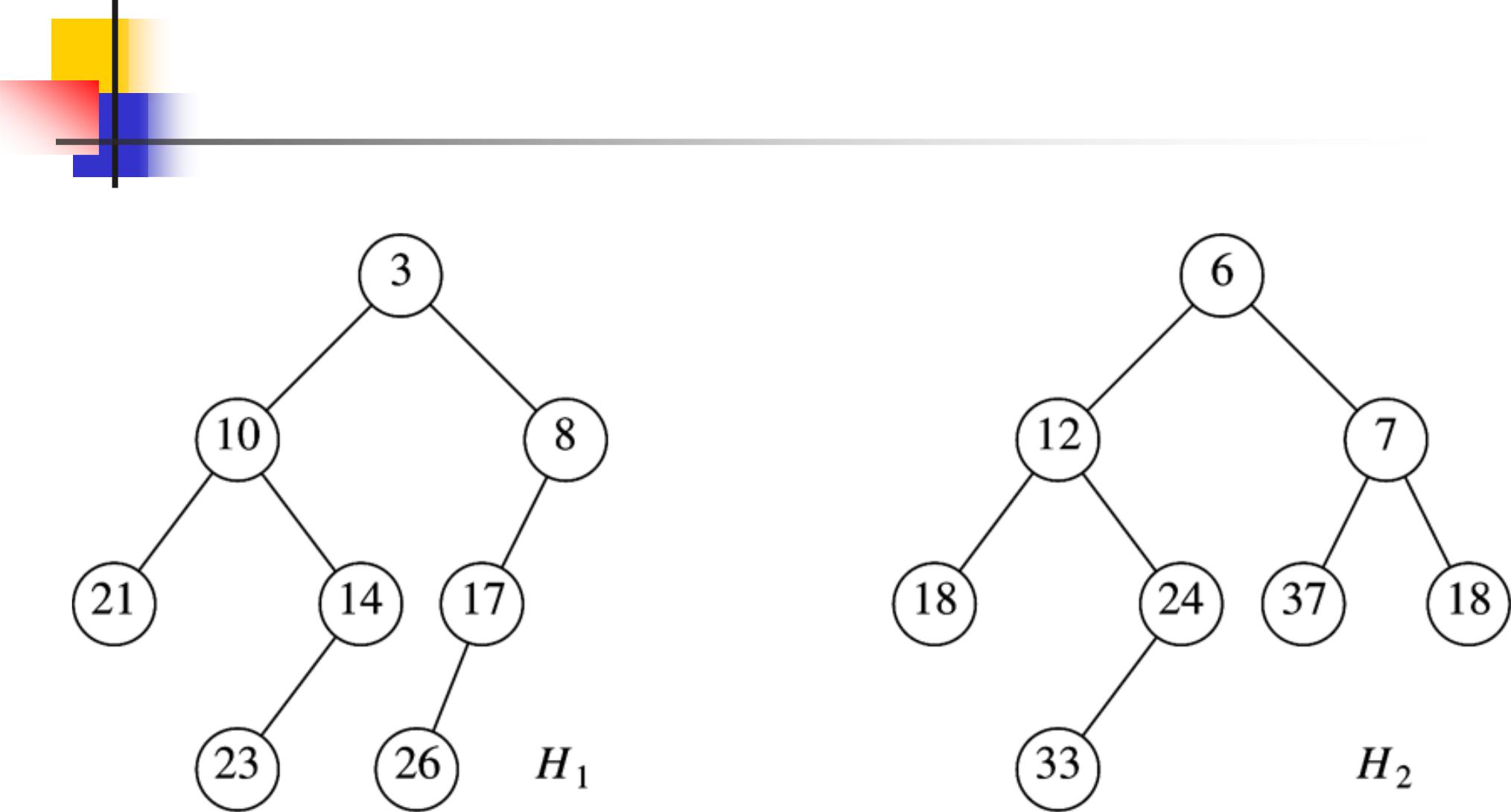
Leftist Heaps



* Merge heaps H1 and H2
  + Assume root(H1) > root(H2)
  + Recursively merge H1 with right subheap of H2
  + If result is not leftist, then swap the left and right subheaps
  + Running time O(log N)
* DeleteMin
  + Delete root and merge children

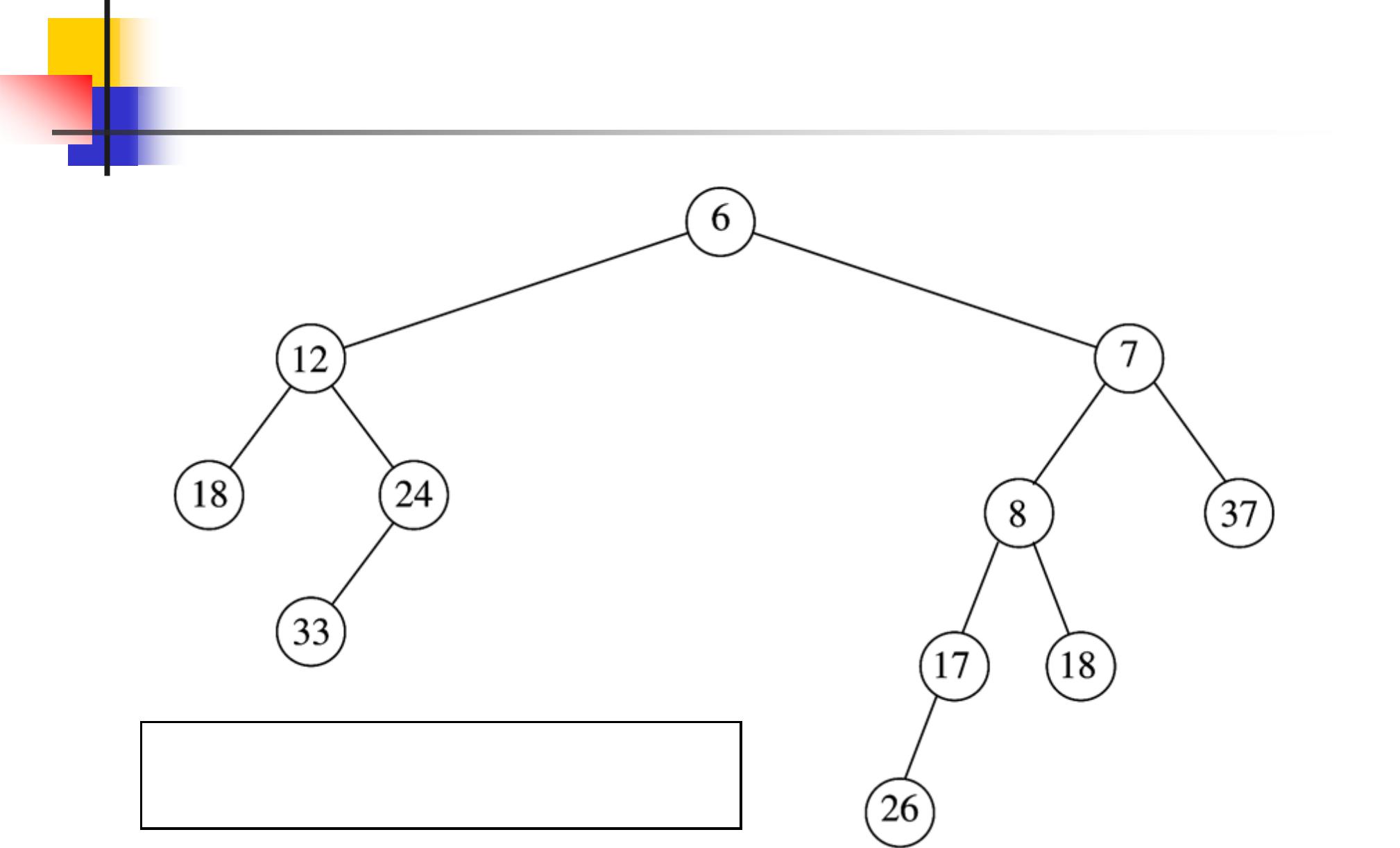
34

Leftist Heaps: Example



35

Leftist Heaps: Example

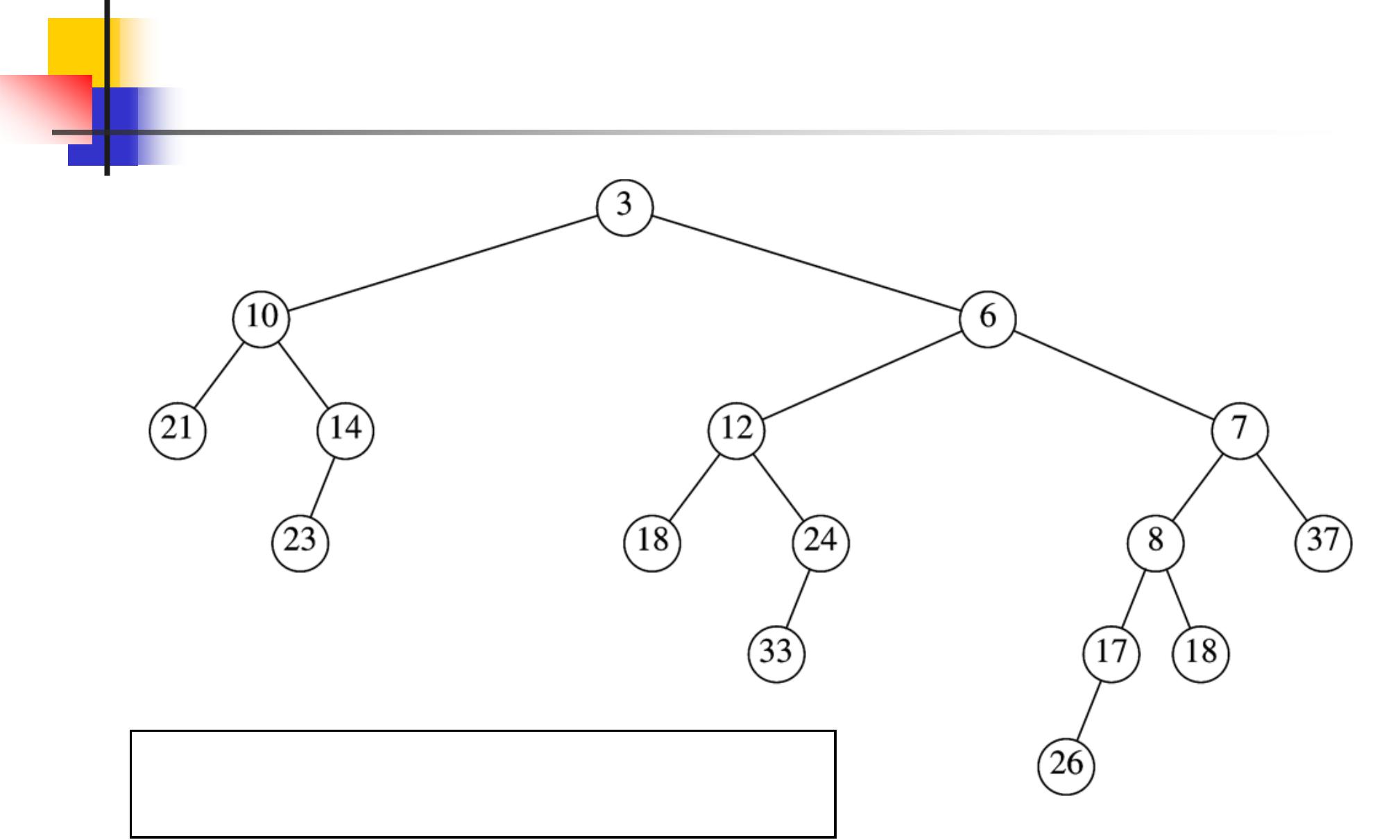


Merge H2 (larger root) with right

sub-heap of H1 (smaller root).

36

Leftist Heaps: Example

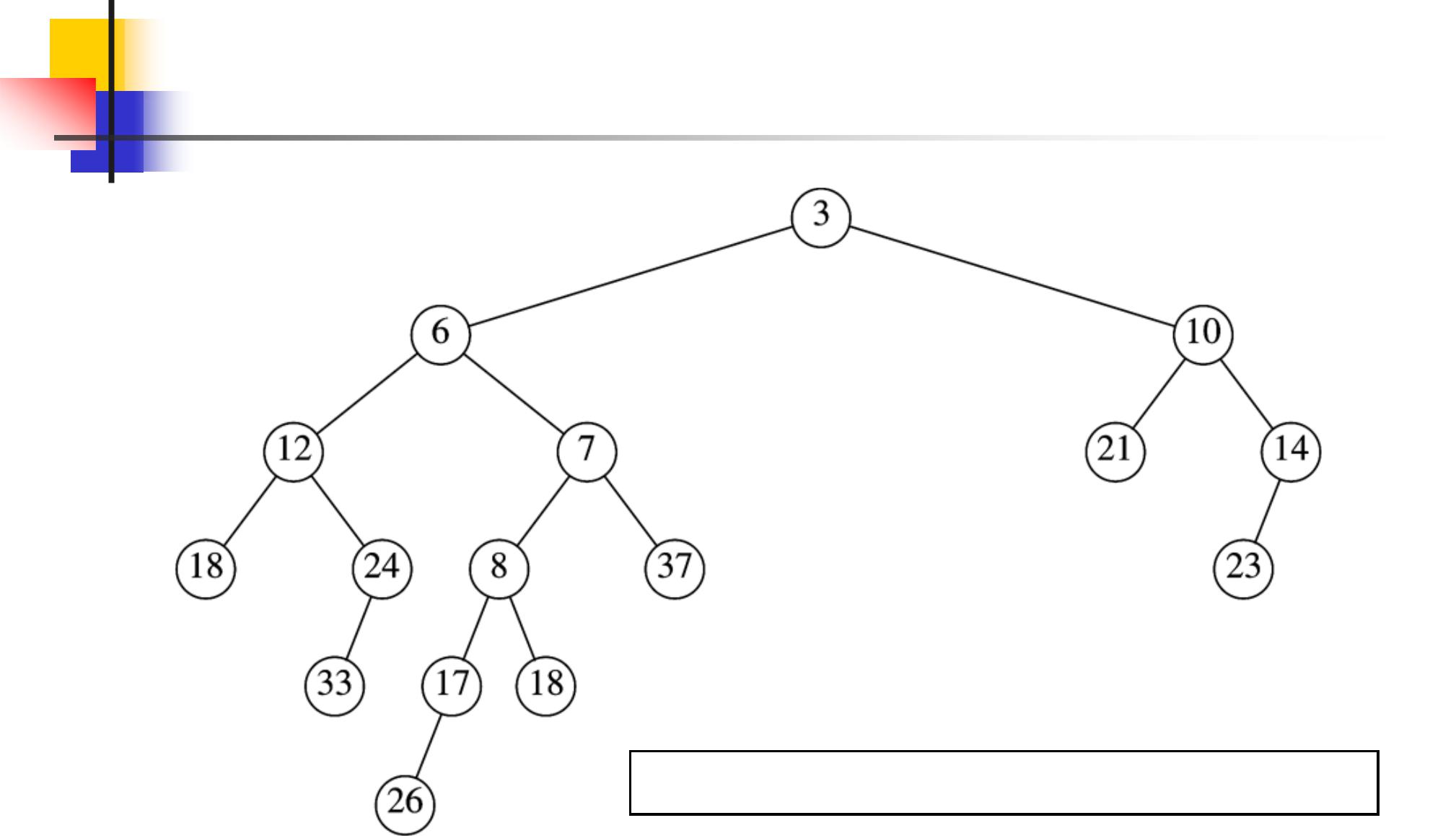


Attach previous heap as H1’s right child.

Leftist heap?

37

Leftist Heaps: Example



Swap root’s children to make leftist heap.

38

Skew Heaps



* Self-adjusting version of leftist heap
* Skew heaps are to leftist heaps as splay trees are to AVL trees
* Skew merge same as leftist merge, except we always swap left and right subheaps
* No need to maintain or test NPL of nodes
* Worst case is O(N)
* Amortized cost of M operations is O(M log N)

39

Binomial Queues



* Support all three operations in O(log N) worst-case time per operation
* Insertions take O(1) average-case time
* Key idea
  + Keep a collection of heap-ordered trees to postpone merging

40

Binomial Queues



* A binomial queue is a forest of binomial trees

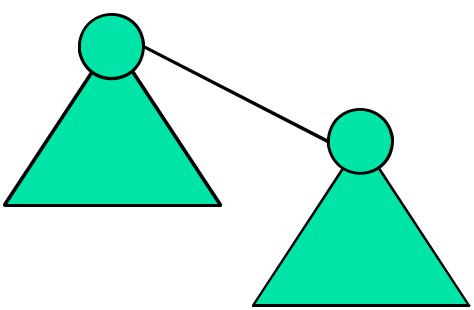




Each in heap order

Each of a different height

* A binomial tree Bk of height k consists of two Bk-1 binomial trees
  + The root of one Bk-1 tree is the child of the root of the other Bk-1 tree



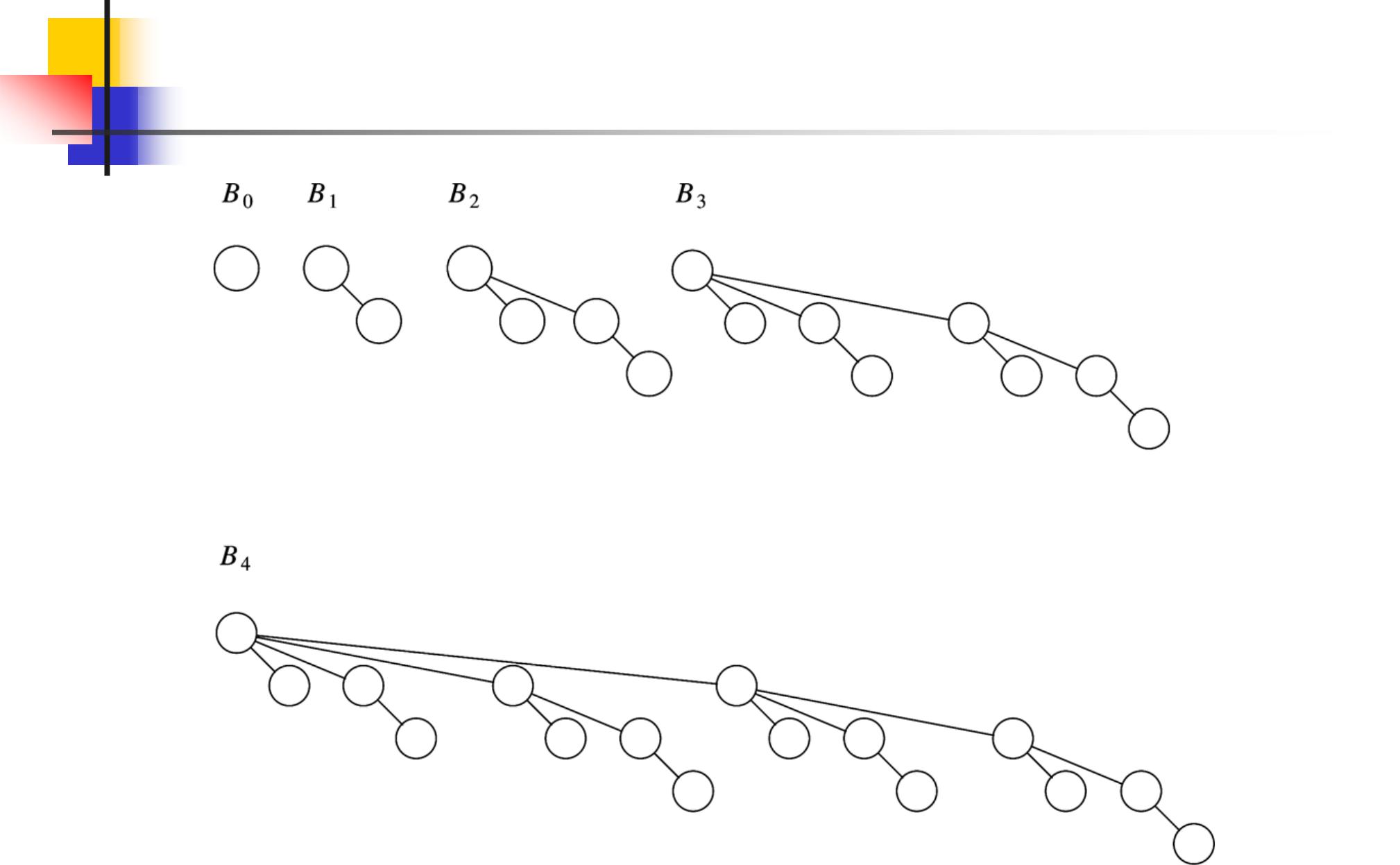
Bk =

Bk-1

Bk-1

41

Binomial Trees



42

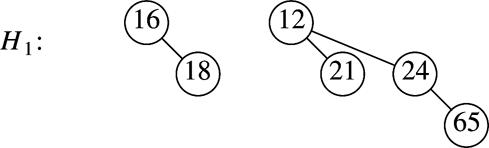
Binomial Trees



* Binomial trees of height k have exactly 2k nodes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | *k* | , the |  |
|  Number of nodes at depth d is | |  |  |
| binomial coefficient | *d* | |  |  |
|  |  |  |  |

* A priority queue of any size can be represented by a binomial queue
  + Binary representation of Bk



43

Binomial Queue Operations



* Minimum element found by checking roots of all trees
  + At most (log2 N) of them, thus O(log N)
  + Or, O(1) by maintaining pointer to minimum element

44

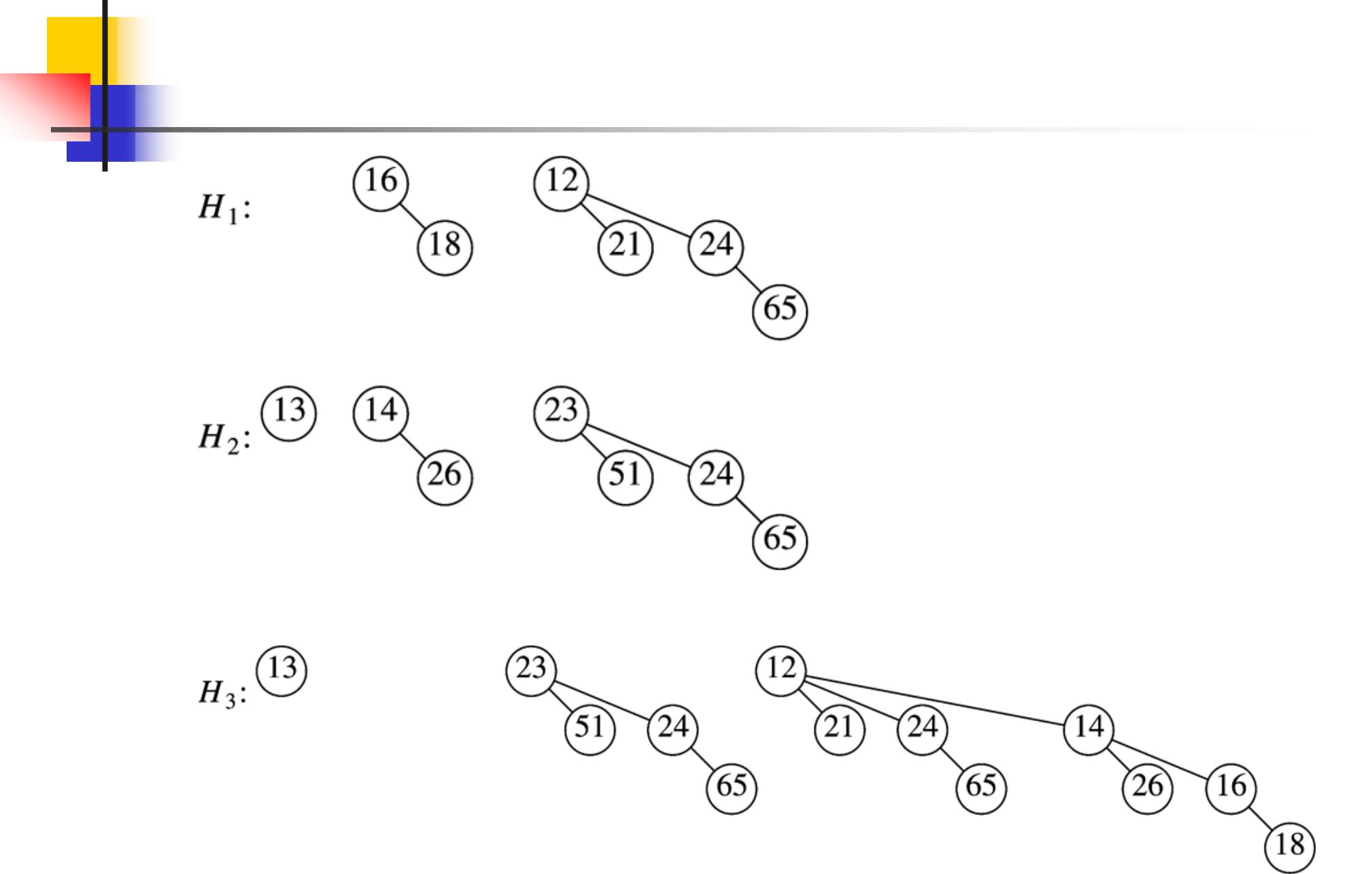
Binomial Queue Operations



* Merge (H1,H2)  H3
  + Add trees of H1 and H2 into H3 in increasing order by depth
  + Traverse H3
    - If find two consecutive Bk trees, then create a Bk+1 tree
    - If three consecutive Bk trees, then leave first, combine last two
    - Never more than three consecutive Bk trees
* Keep binomial trees ordered by height
* min(H3) = min(min(H1),min(H2))
* Running time O(log N)

45

Merge Example



46

Binomial Queue Operations



* Insert (x, H1)
  + Create single-element queue H2
  + Merge (H1,H2)
* Running time proportional to minimum k such that Bk not in heap
* O(log N) worst case
* Probability Bk not present is 0.5
  + Thus, likely to find empty Bk after two tries on average
  + O(1) average case

47

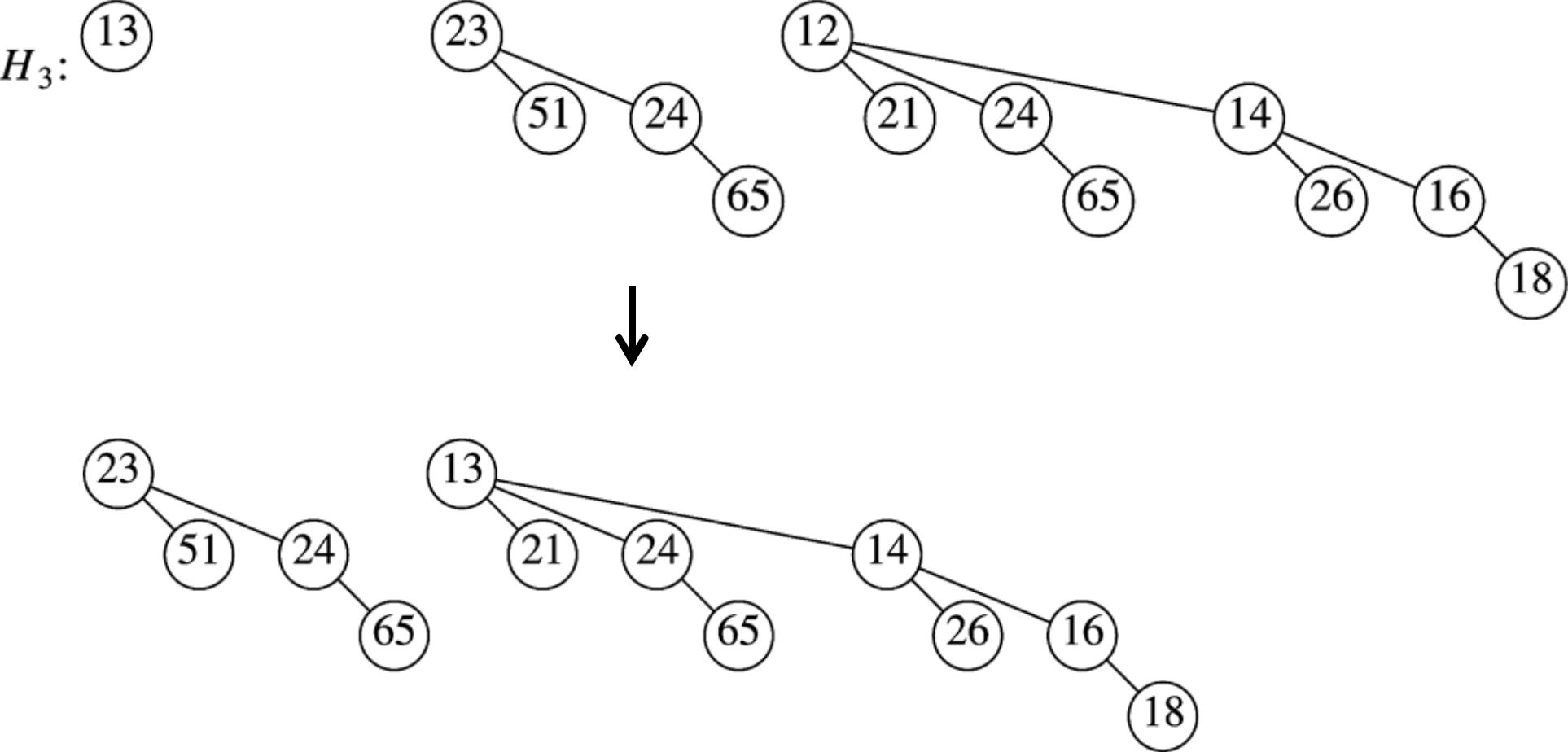
Binomial Queue Operations



* deleteMin (H1)
  + Remove min(H1) tree from H1
  + Create heap H2 from the children of min(H)
  + Merge (H1,H2)
* Running time O(log N)

48

deleteMin Example

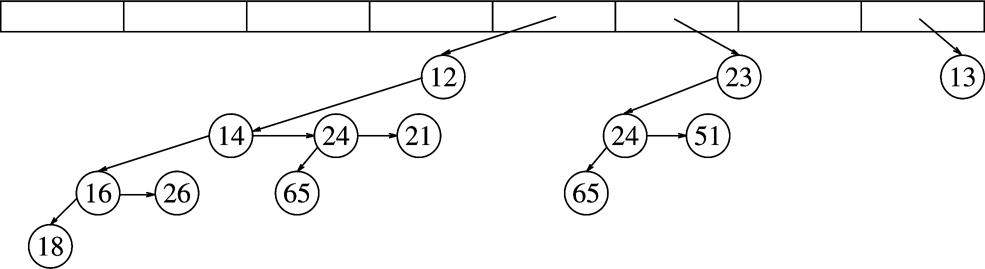


49

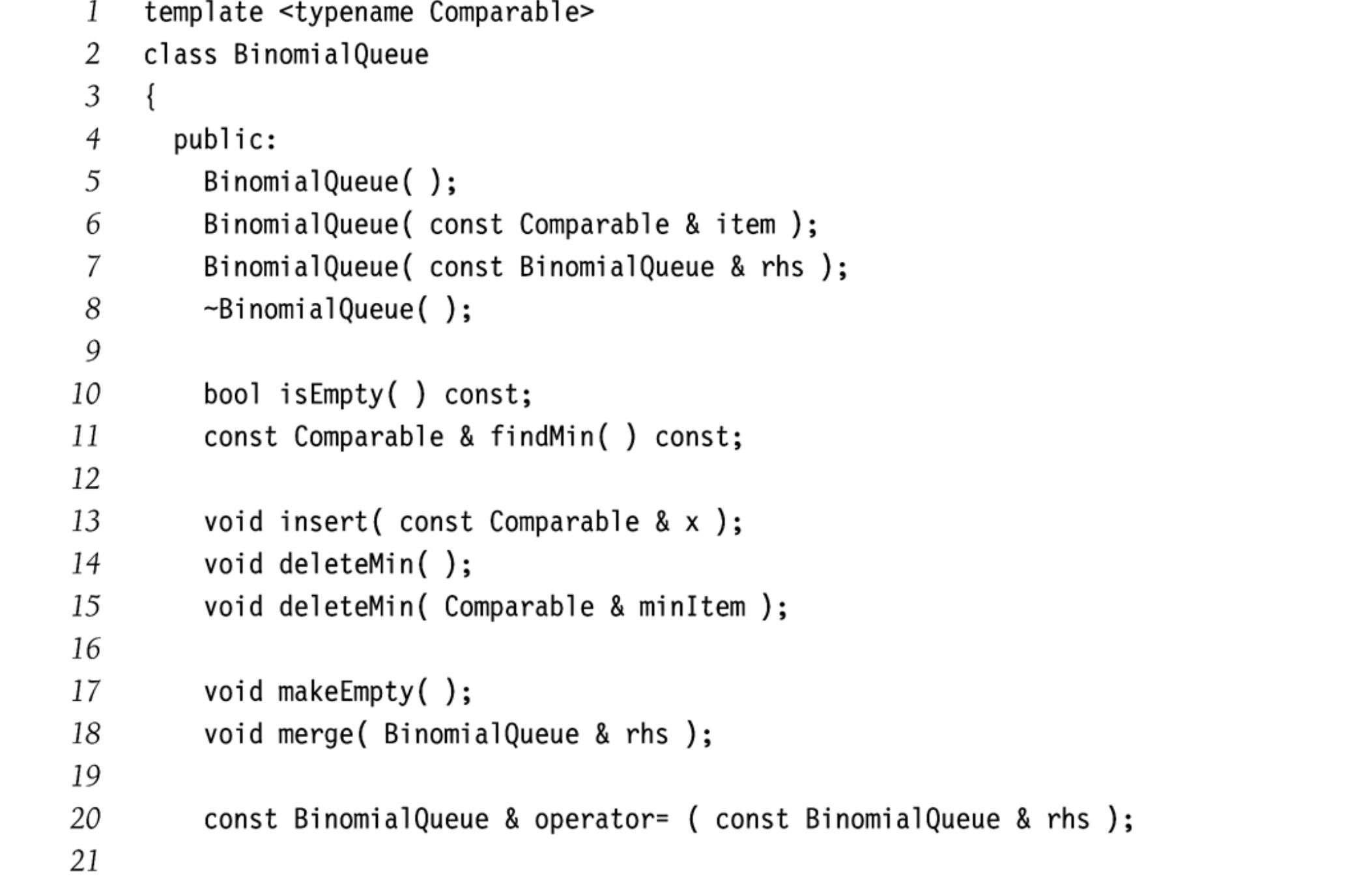
Binomial Queue Implementation

* Array of binomial trees
* Trees use first-child, right-sibling representation

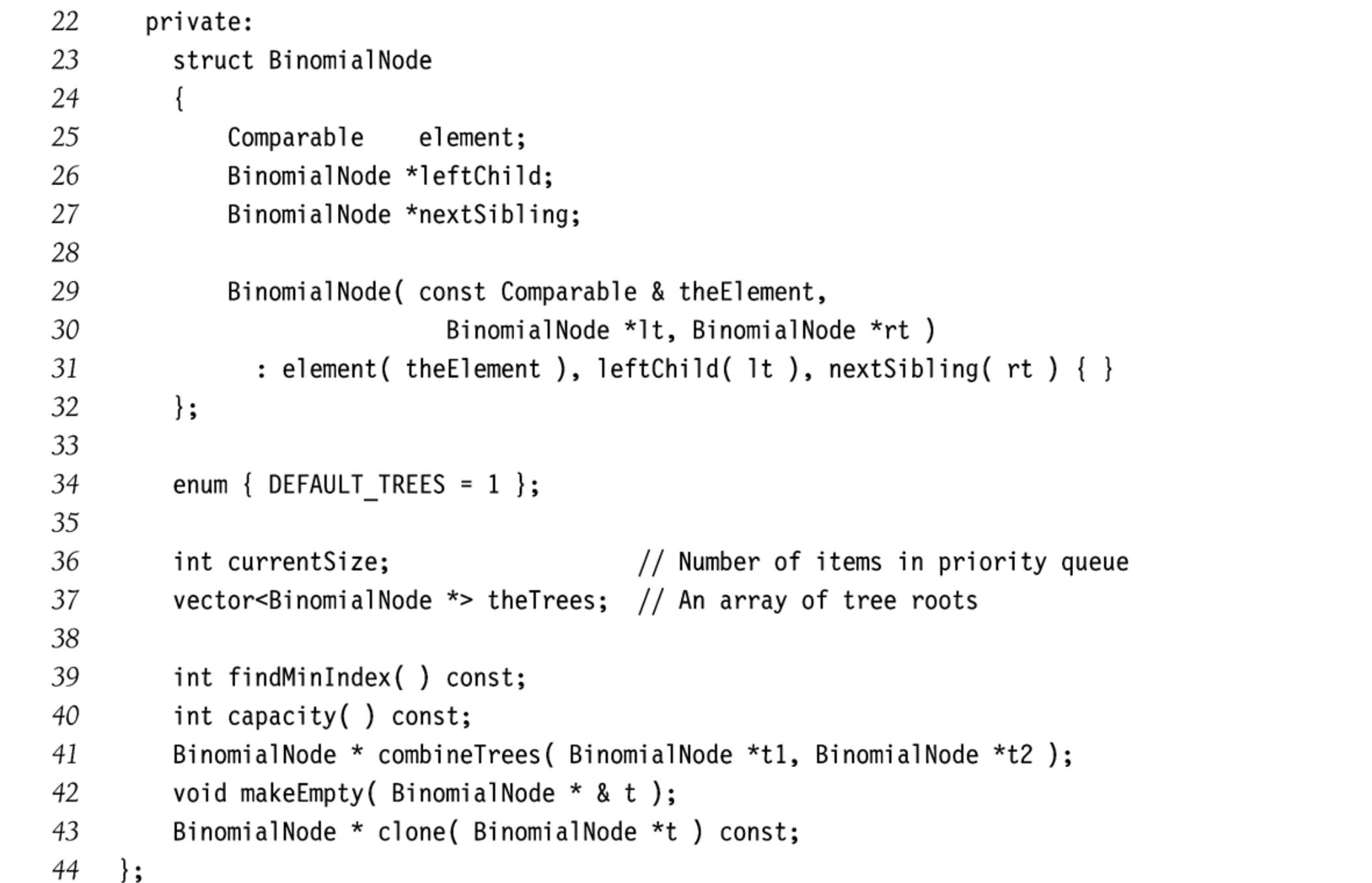
H3:



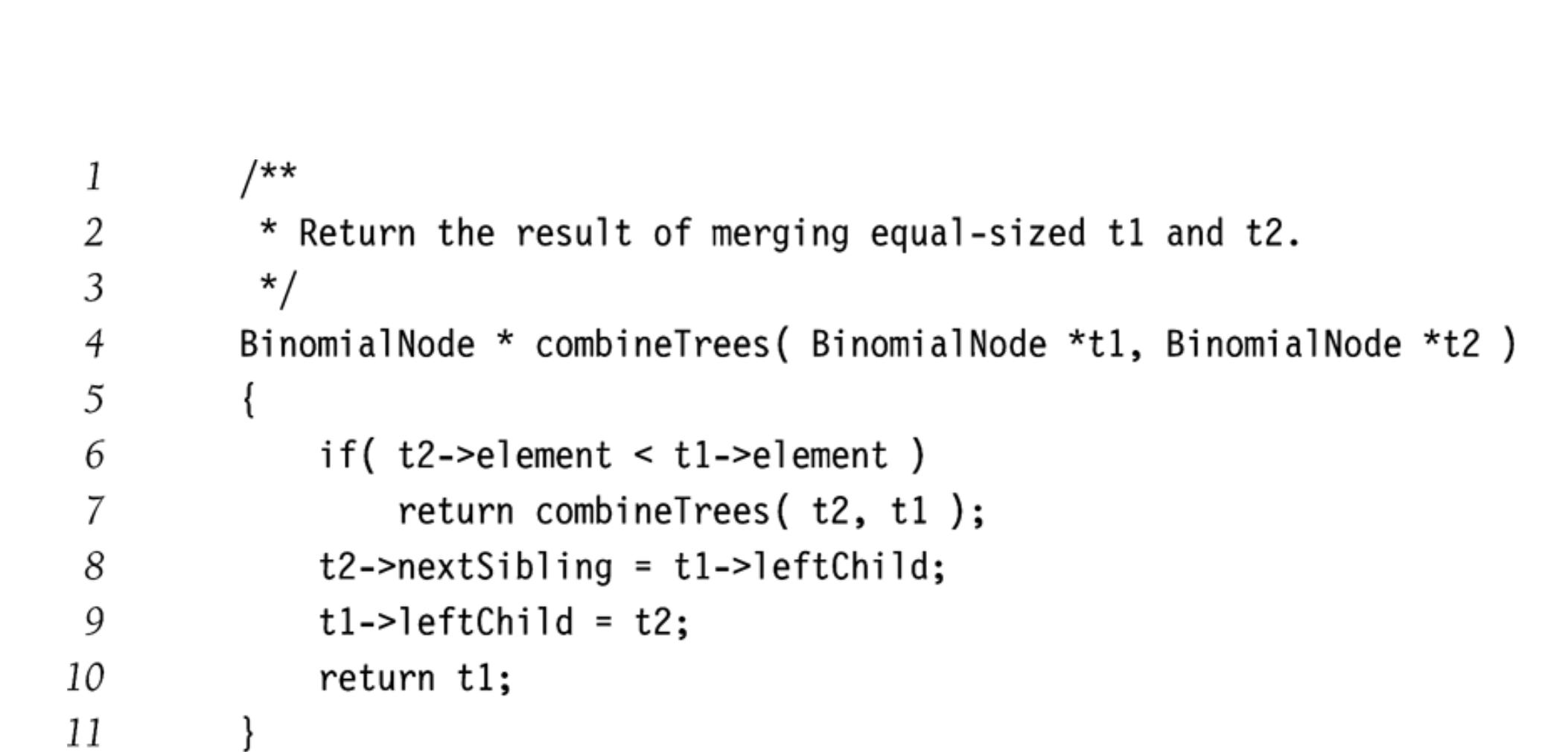
50



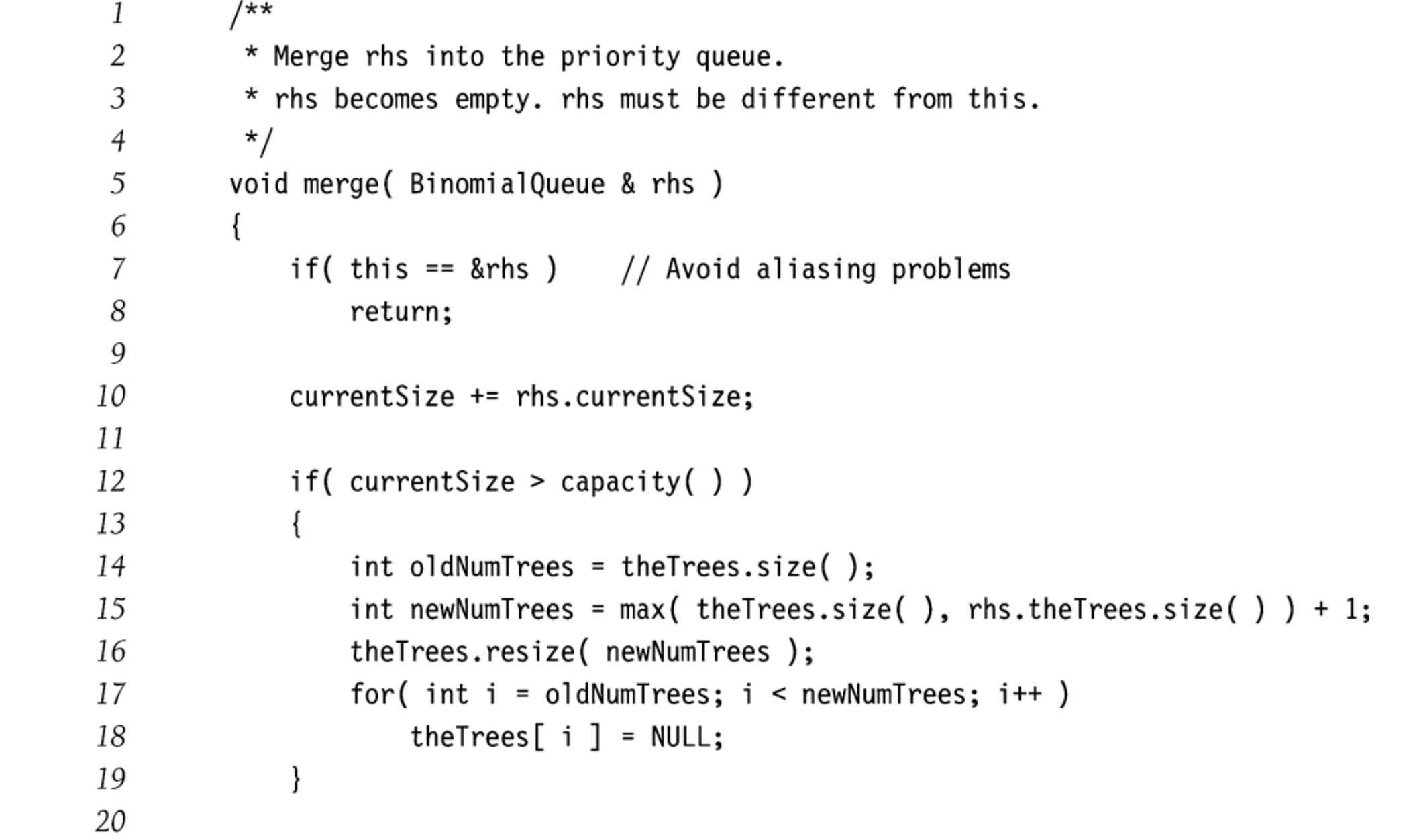
51



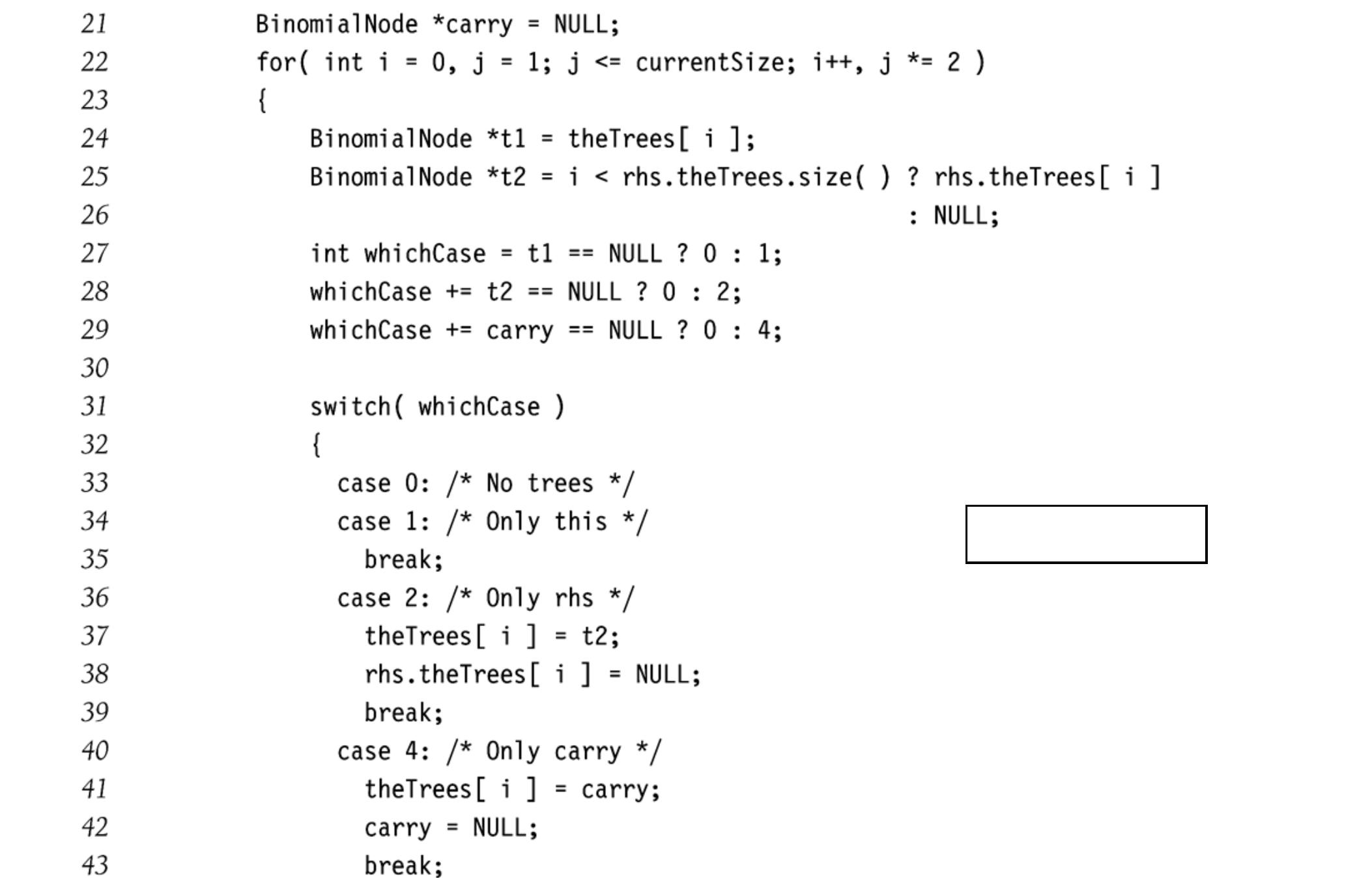
52



53

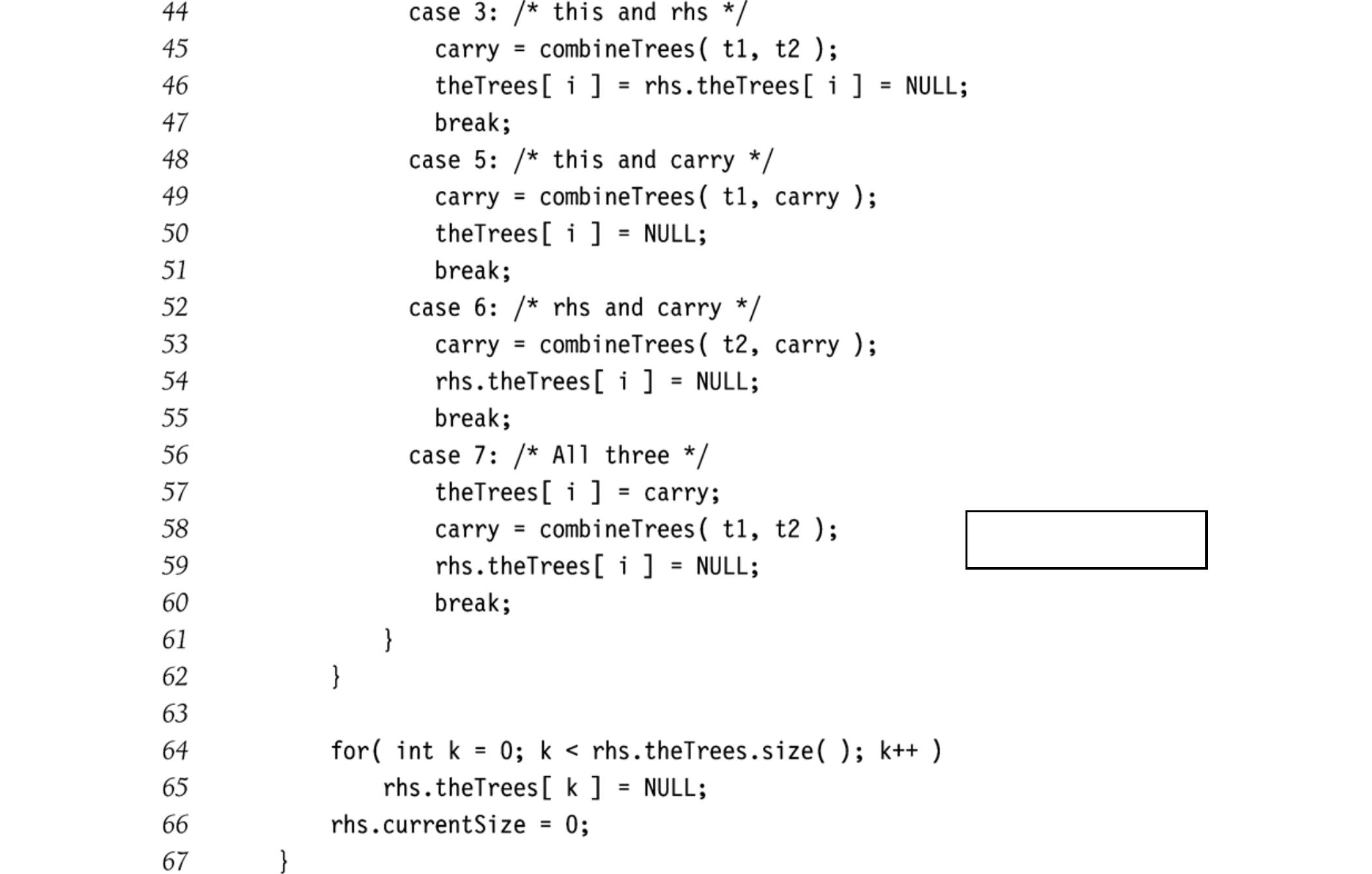


54



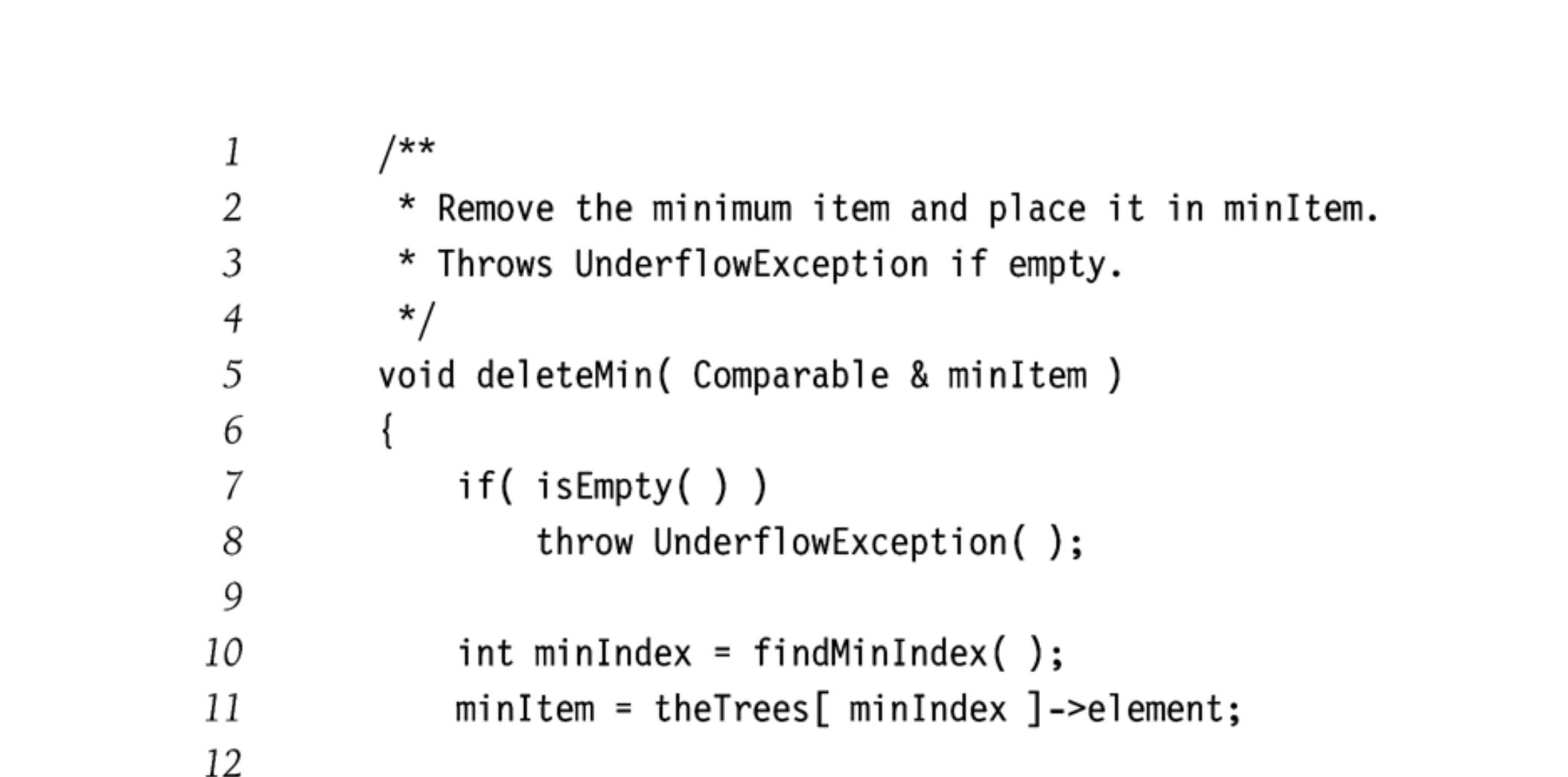
merge (cont.)

55

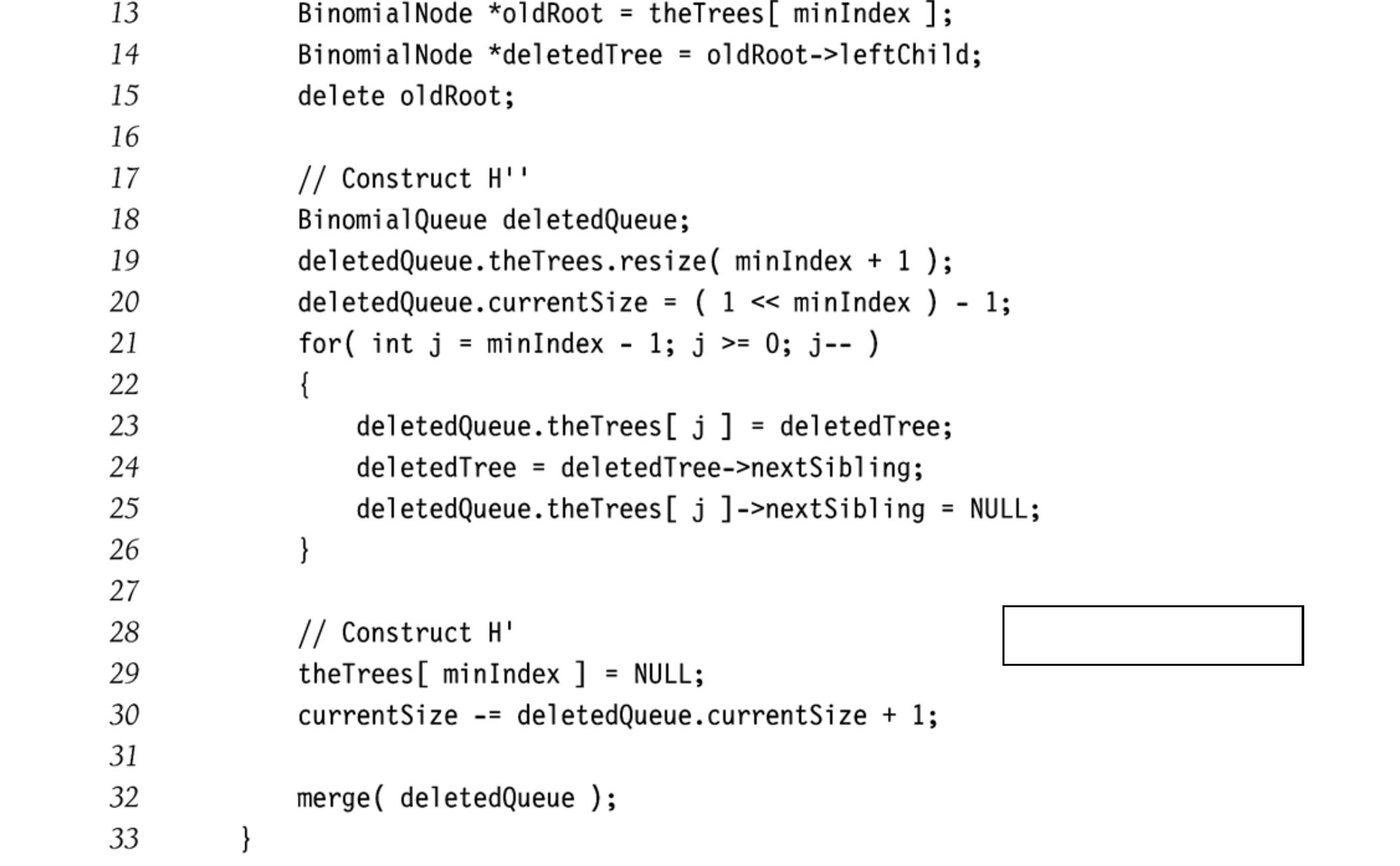


merge (cont.)

56

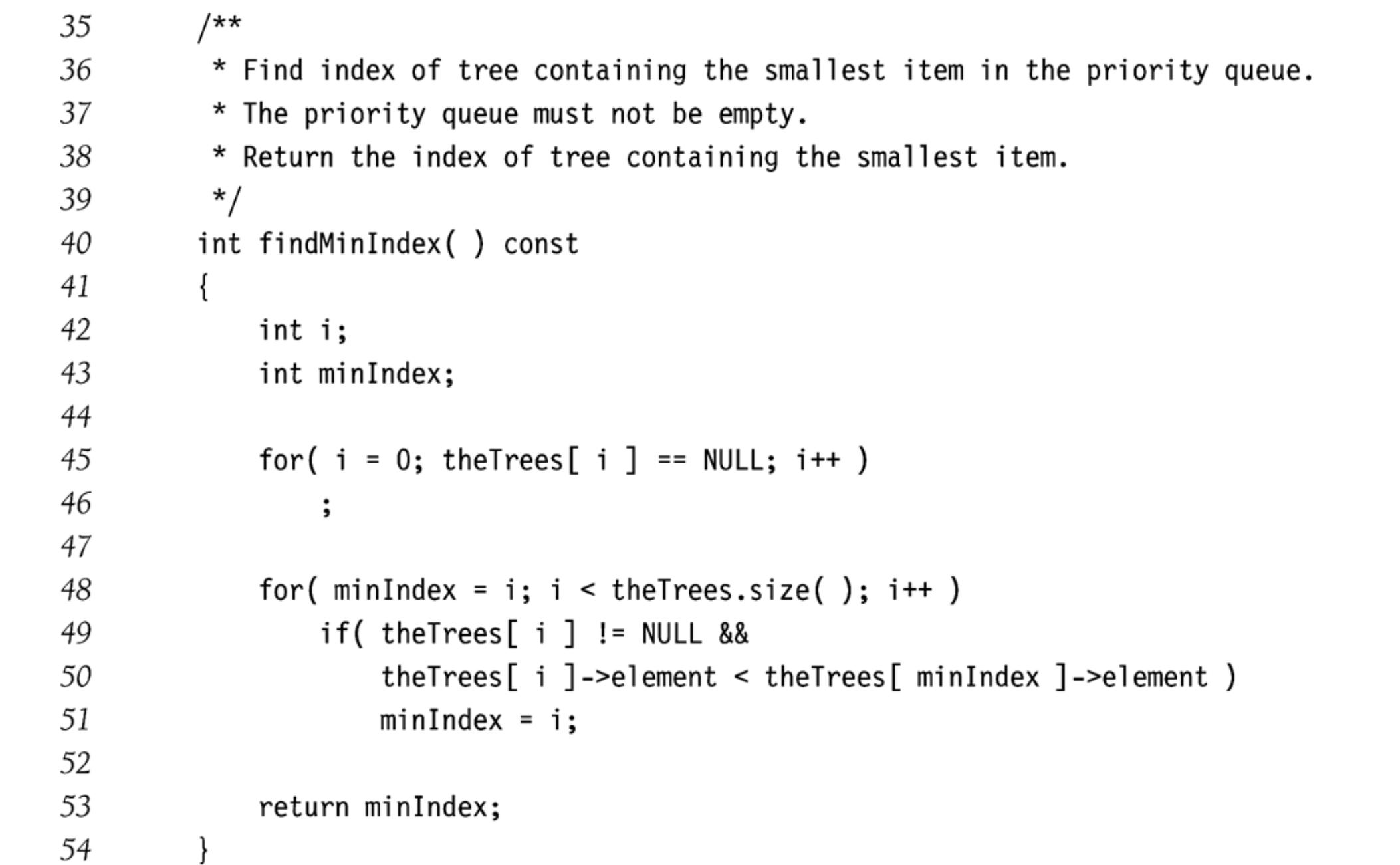


57



deleteMin (cont.)

58



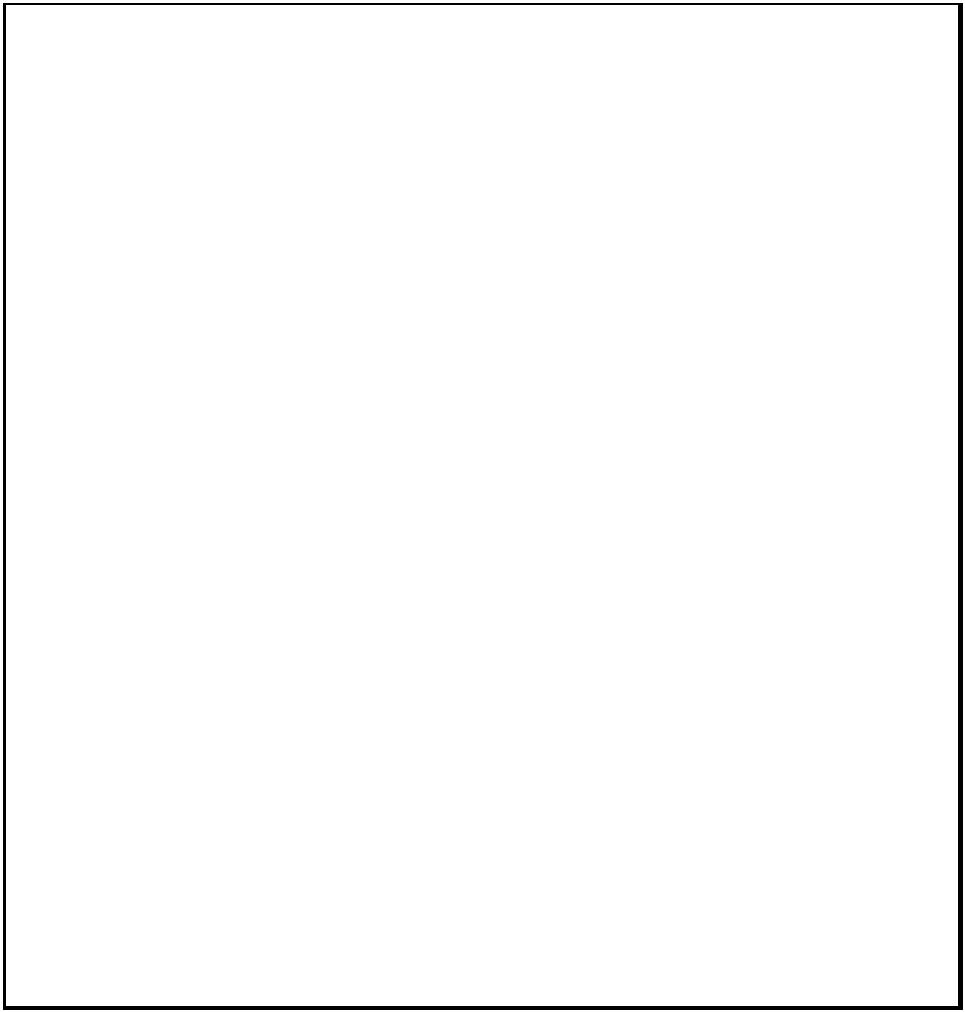
59

Priority Queues in STL



* Binary heap
* Maintains maximum element
* Methods
  + Push, top, pop, empty, clear

**#include <iostream> #include <queue> using namespace std;**



**int main ()**

**{**

**priority\_queue<int> Q; for (int i=0; i<100; i++)**

**Q.push(i);**

**while (! Q.empty())**

**{**

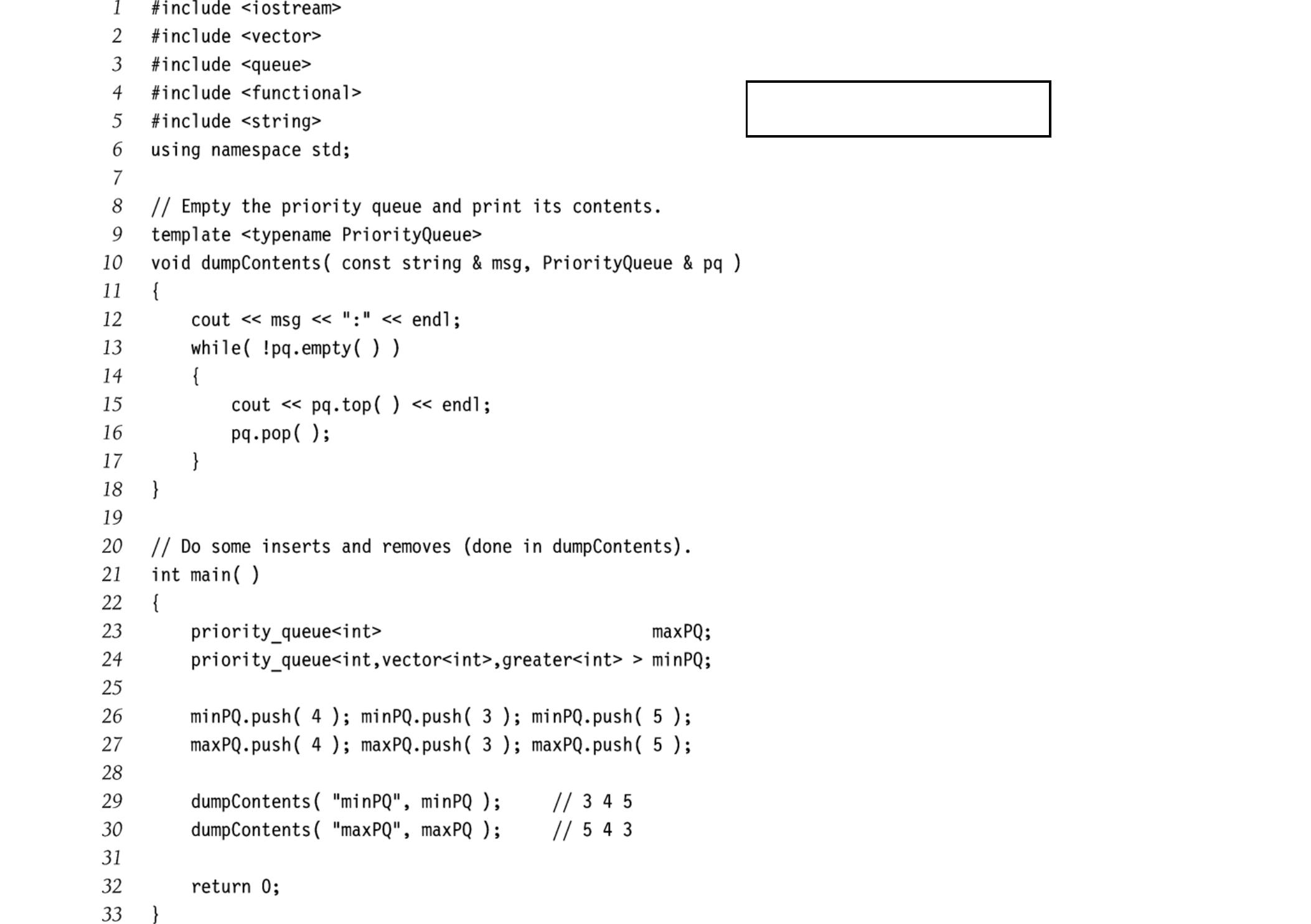
**cout << Q.top() << endl;**

**Q.pop();**

**}**

**}**

60

STL priority queue

61

Summary



* Priority queues maintain the minimum or maximum element of a set
* Support O(log N) operations worst-case
  + insert, deleteMin, merge
* Support O(1) insertions average case
* Many applications in support of other algorithms

62