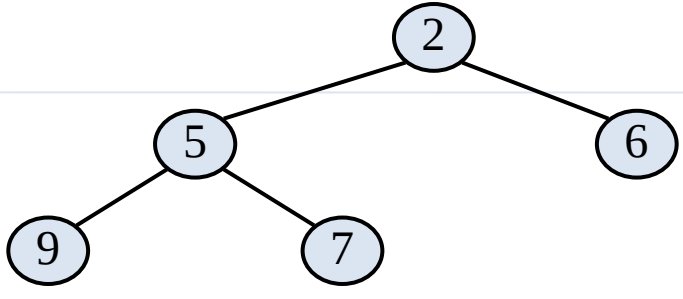
HEAPS:
IMPLEMENTING EFFICIENT PRIORITY QUEUES



Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

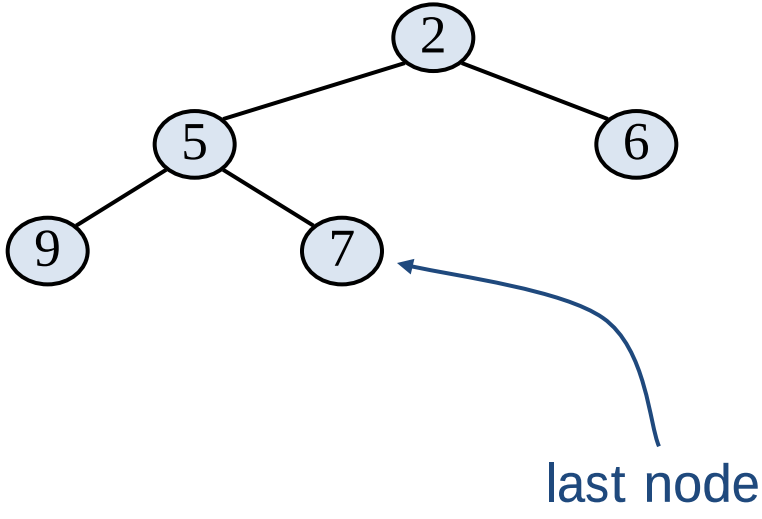
RECALL PRIORITY QUEUE ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, v): inserts an entry with key k and value v
  - removeMin(): removes and returns the entry with smallest key
- Additional methods
  - min(): returns, but does not remove, an entry with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

HEAPS

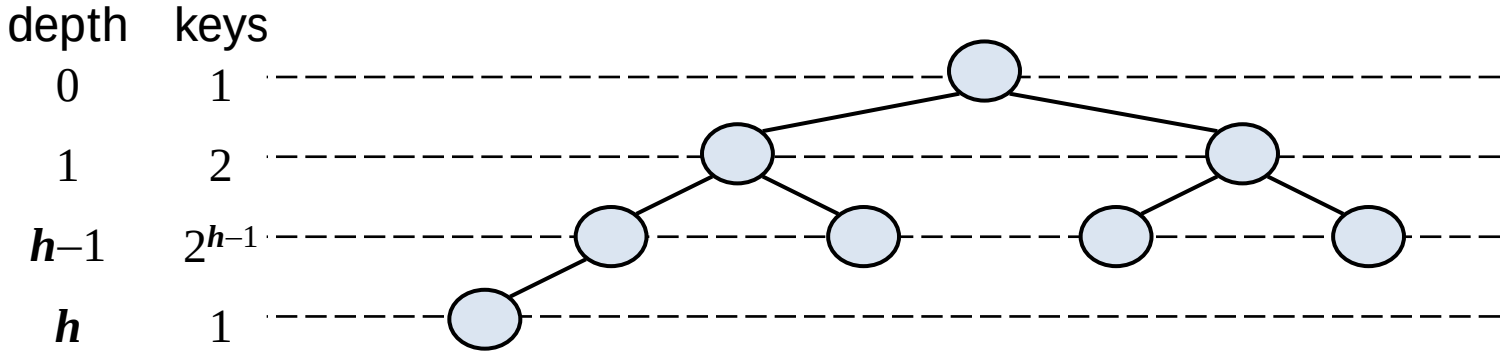
- A binary heap is a binary tree storing keys at its nodes and satisfying the following properties:
- (min) Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h-1$, there are 2^i nodes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

- The last node of a heap is the rightmost node of maximum depth



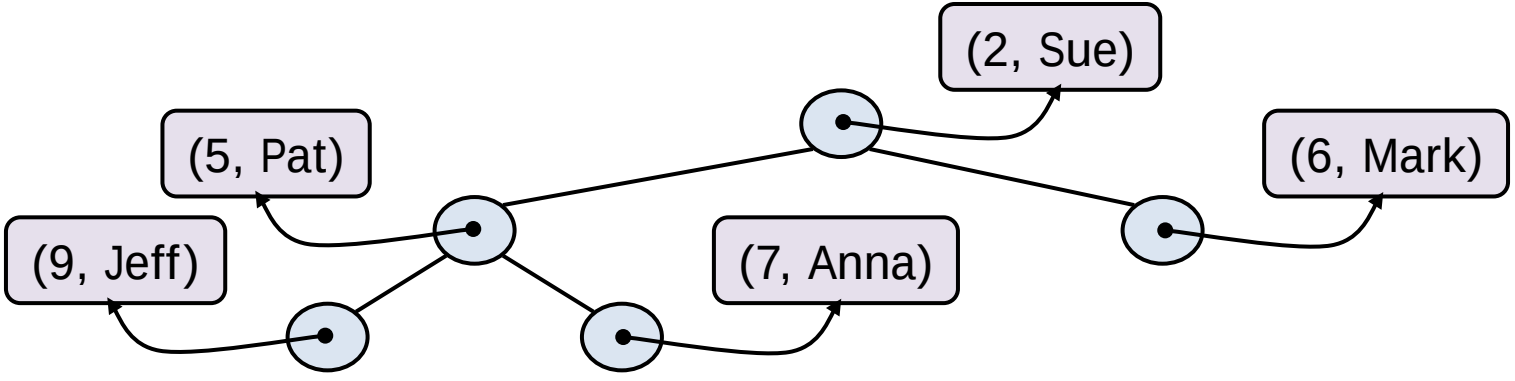
HEIGHT OF A HEAP

- Theorem: A heap T storing n entries has height $h = \lfloor \log n \rfloor$.
- 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., $h \leq \log n$



HEAPS AND PRIORITY QUEUES

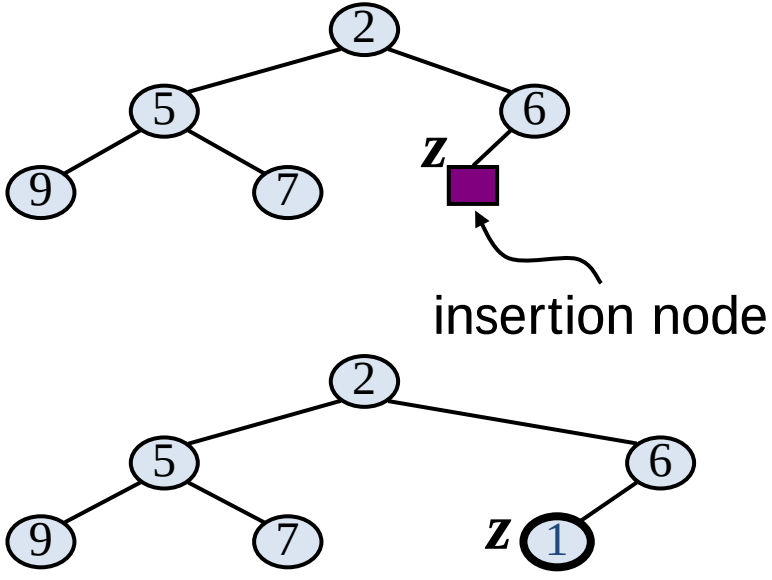
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



- Size() and isEmpty() methods can be implemented based on examination of the tree
- Min() operation is equally trivial
- Algorithms for implementing the insert and removeMin methods will be discussed.

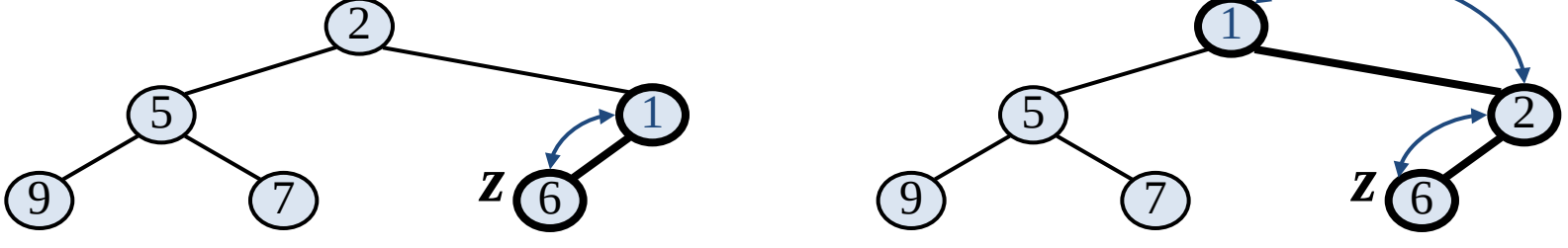
INSERTION INTO A HEAP

- Method *insert(k, v)* of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps to maintain the ***complete binary tree property***,
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property



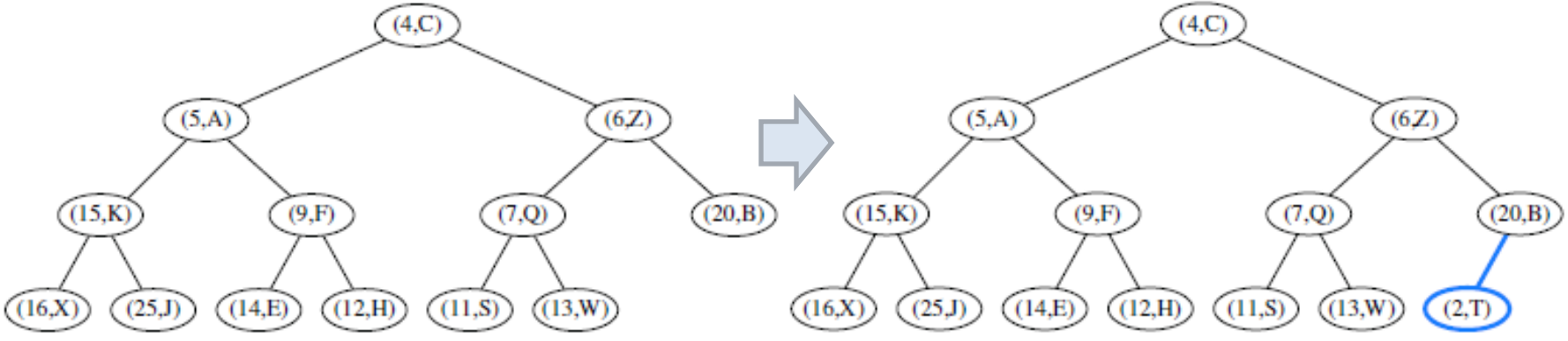
UPHEAP

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



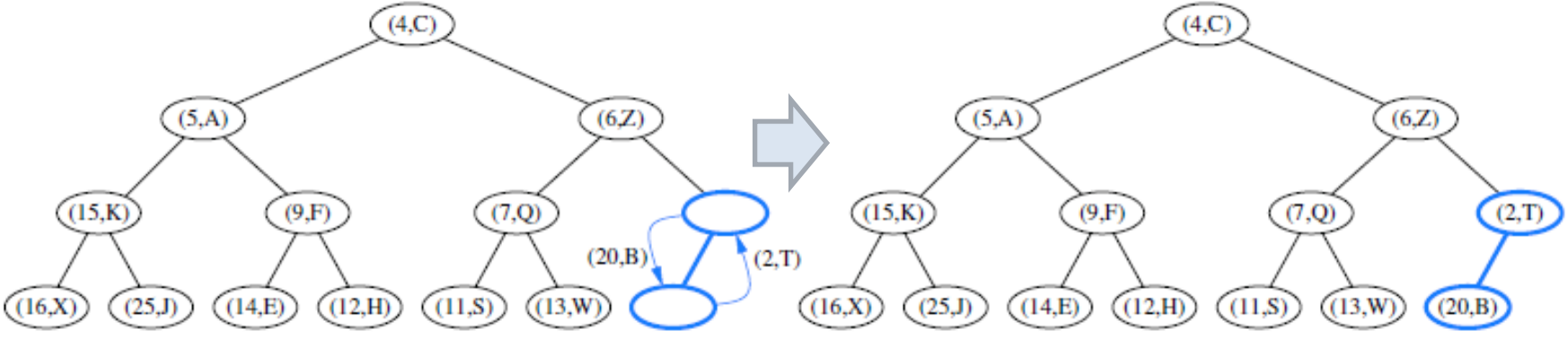
EXAMPLE OF UP-HEAP

Inserting (2,T)



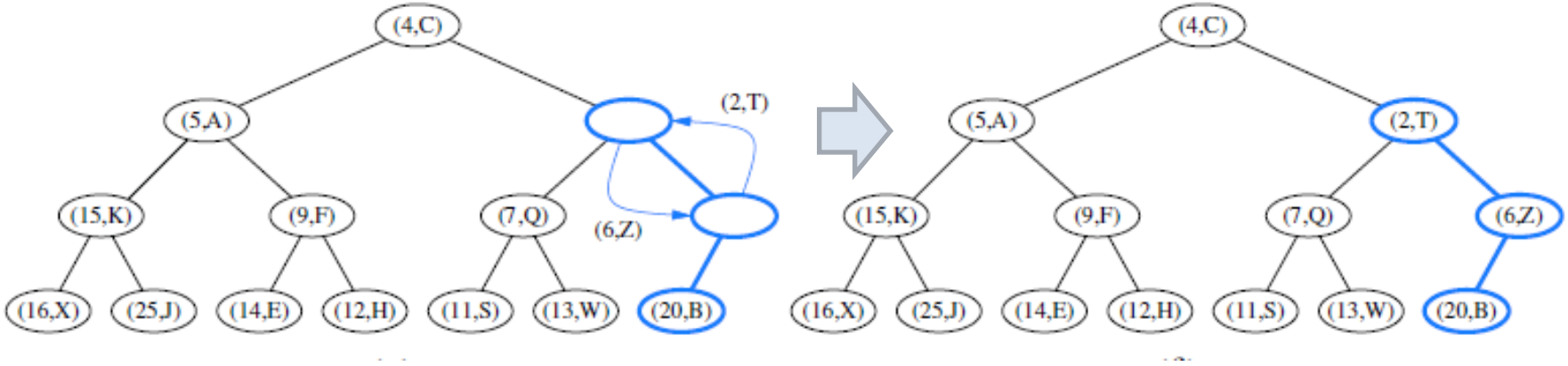
EXAMPLE OF UP-HEAP CONT.

Swap (20,B) with (2,T)



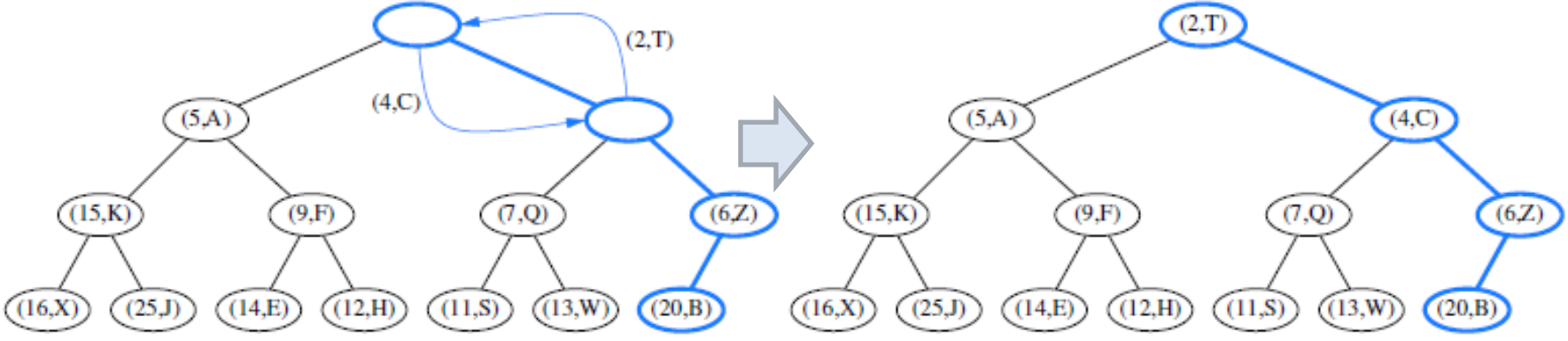
EXAMPLE OF UP-HEAP CONT.

Swap (6,Z) with (2,T)



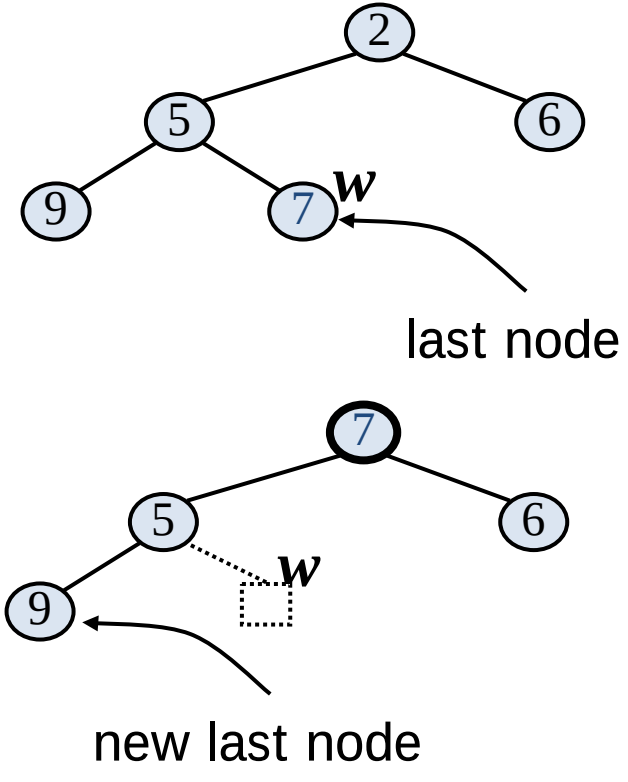
EXAMPLE OF UP-HEAP CONT.

Swap (4,C) with (2,T)



REMOVAL FROM A HEAP

- Method removeMin 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
  - Remove w
  - Restore the heap-order property (discussed next)

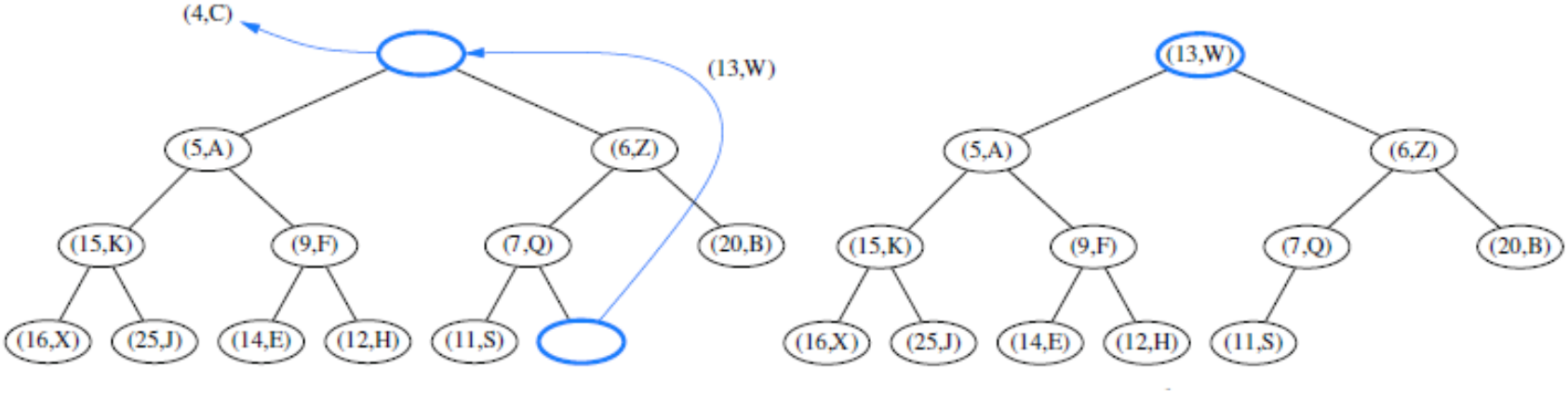


DOWNHEAP

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Upheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

EXAMPLE DOWN-HEAP

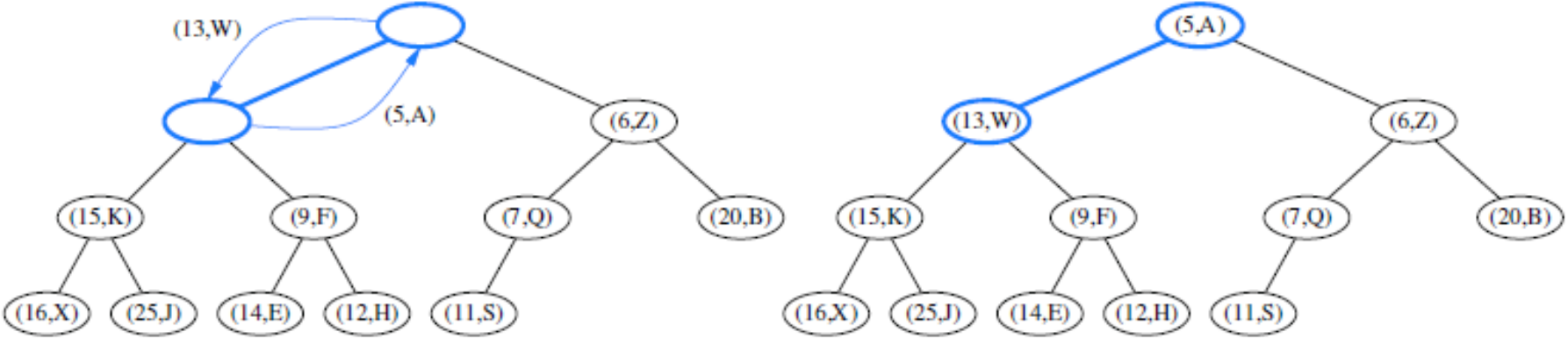
Swap last node with root



delete previous root (the last node)

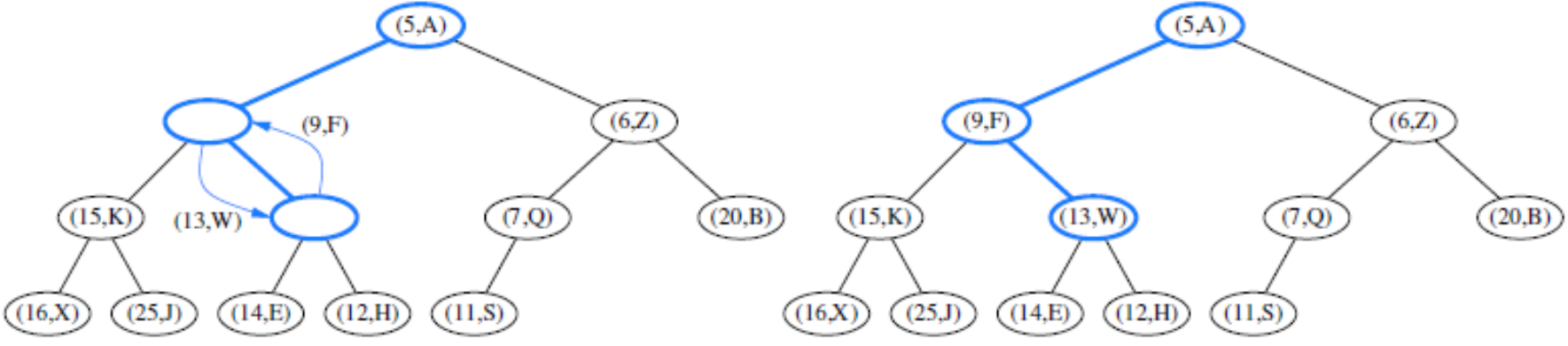
EXAMPLE DOWN-HEAP CONT.

Swap (13,W) with (5,A)



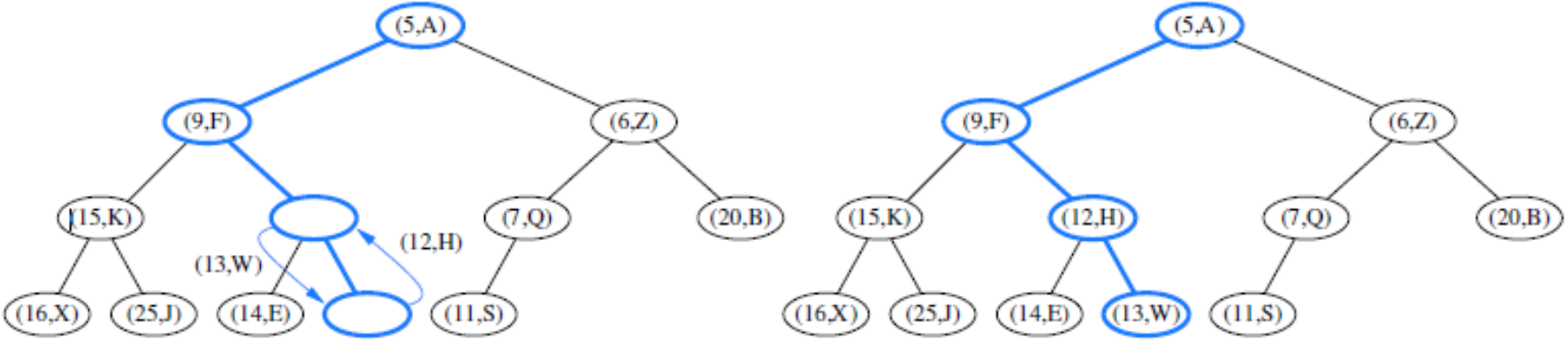
EXAMPLE DOWN-HEAP CONT.

Swap (13,W) with (9,F)



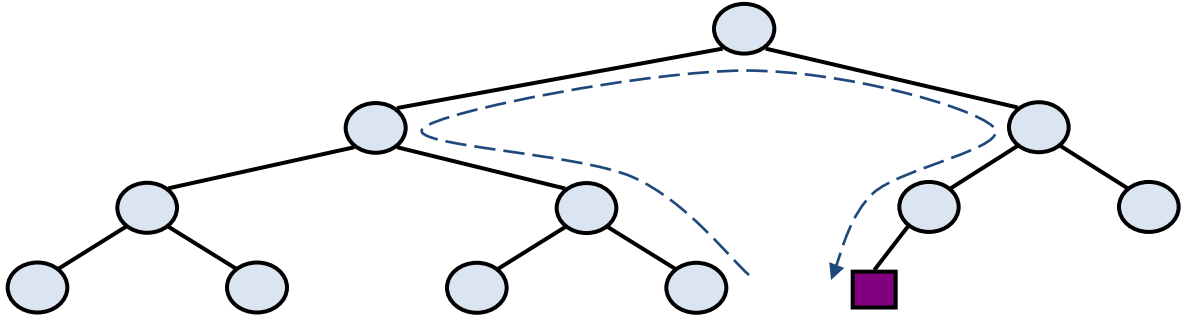
EXAMPLE DOWN-HEAP CONT.

Swap (13,W) with (12,H)



UPDATING THE LAST NODE

- The last node is the rightmost node at the bottom level of the tree, or as the leftmost position of a new level
- The last node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal

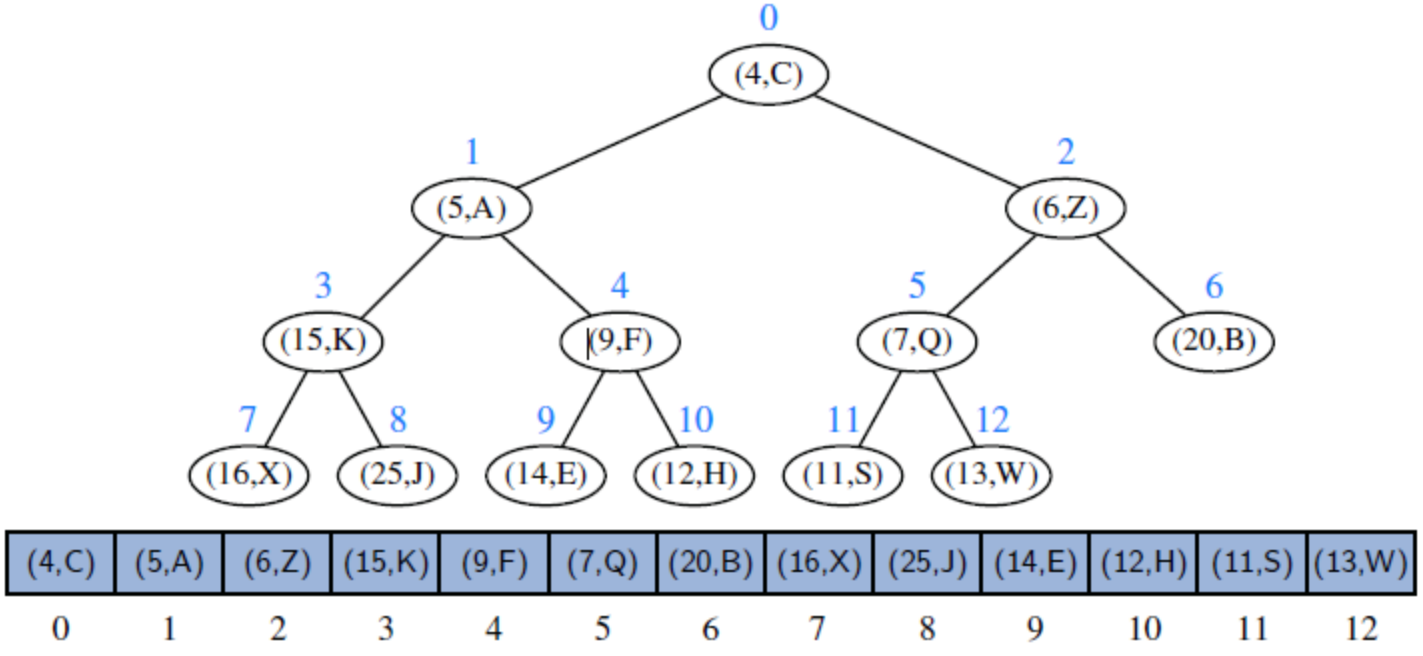


HEAP-SORT

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

ARRAY-BASED HEAP IMPLEMENTATION

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Methods insert and removeMin depend on locating the last position of a heap (in heap of size n, the last position at index $n-1$.)
  - **insert** corresponds to inserting at rank $n + 1$
  - **removeMin** corresponds to removing at rank n
- Space usage of an array-based representation of a complete binary tree with n nodes is O(n),
- Time bounds of methods for adding or removing elements become **amortized.** (occasional resizing of array needed)
- Yields in-place heap-sort



JAVA IMPLEMENTATION 1

Although we think of our heap as a binary tree, we do not formally use the binary tree ADT but use the more efficient array-based representation of a tree.

```
1  /** An implementation of a priority queue using an array-based heap. */
2  public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3    /** primary collection of priority queue entries */
4    protected ArrayList<Entry<K,V>> heap = new ArrayList<>();
5    /** Creates an empty priority queue based on the natural ordering of its keys. */
6    public HeapPriorityQueue() { super(); }
7    /** Creates an empty priority queue using the given comparator to order keys. */
8    public HeapPriorityQueue(Comparator<K> comp) { super(comp); }
9    // protected utilities
10   protected int parent(int j) { return (j-1) / 2; }          // truncating division
11   protected int left(int j) { return 2*j + 1; }
12   protected int right(int j) { return 2*j + 2; }
13   protected boolean hasLeft(int j) { return left(j) < heap.size(); }
14   protected boolean hasRight(int j) { return right(j) < heap.size(); }
```

compute the position of parent or child

JAVA IMPLEMENTATION 2

```
48    /** Returns the number of items in the priority queue. */
49    public int size() { return heap.size(); }
```

```
15    /** Exchanges the entries at indices i and j of the array list. */
16    protected void swap(int i, int j) {
17      Entry<K,V> temp = heap.get(i);
18      heap.set(i, heap.get(j));
19      heap.set(j, temp);
20    }
```

JAVA IMPLEMENTATION 3

```
55   /** Inserts a key-value pair and returns the entry created. */
56   public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
57     checkKey(key);        // auxiliary key-checking method (could throw exception)
58     Entry<K,V> newest = new PQEntry<>(key, value);
59     heap.add(newest);                          // add to the end of the list
60     upheap(heap.size( ) − 1);                  // upheap newly added entry
61     return newest;
62   }

21   /** Moves the entry at index j higher, if necessary, to restore the heap property. */
22   protected void upheap(int j) {
23     while (j > 0) {                 // continue until reaching root (or break statement)
24       int p = parent(j);
25       if (compare(heap.get(j), heap.get(p)) >= 0) break;   // heap property verified
26       swap(j, p);
27       j = p;                                   // continue from the parent's location
28     }
29   }
```

A new entry is **added** the end of the array-list, and then repositioned as needed with **upheap**.

JAVA IMPLEMENTATION 4

```
63    /** Removes and returns an entry with minimal key (if any). */
64    public Entry<K,V> removeMin( ) {
65      if (heap.isEmpty( )) return null;
66      Entry<K,V> answer = heap.get(0);
67      swap(0, heap.size( ) − 1);                  // put minimum item at the end
68      heap.remove(heap.size( ) − 1);              // and remove it from the list;
69      downheap(0);                                // then fix new root
70      return answer;
71    }
72  }
```

```
30    /** Moves the entry at index j lower, if necessary, to restore the heap property. */
31    protected void downheap(int j) {
32      while (hasLeft(j)) {                    // continue to bottom (or break statement)
33        int leftIndex = left(j);
34        int smallChildIndex = leftIndex;          // although right may be smaller
35        if (hasRight(j)) {
36            int rightIndex = right(j);
37            if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
38              smallChildIndex = rightIndex;       // right child is smaller
39        }
40        if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
41          break;                                  // heap property has been restored
42        swap(j, smallChildIndex);
43        j = smallChildIndex;                      // continue at position of the child
44      }
45    }
46
```

To **remove** the entry with minimal key (which resides at index 0), we move the last entry of the array-list from index $n-1$ to index 0, and then invoke **downheap** to reposition it.

ANALYSIS OF A HEAP-BASED PRIORITY QUEUE

Assuming that two keys can be compared in $O(1)$ time and that the heap T is implemented with an array-based or linked-based tree representation.

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

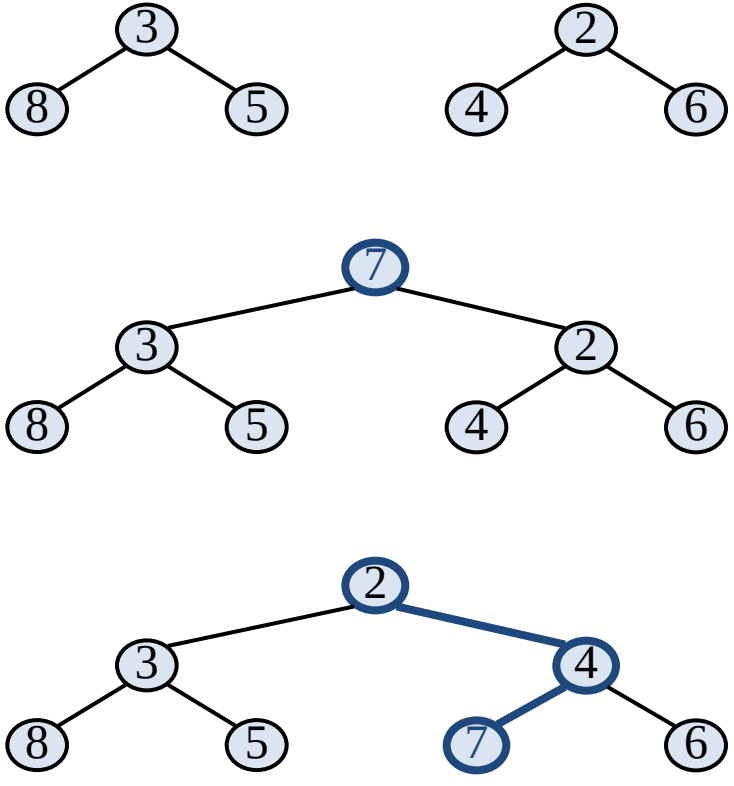
*amortized, if using dynamic array

BOTTOM-UP HEAP CONSTRUCTION

- If we start with an initially empty heap, n successive calls to the insert operation will run in $O(n \log n)$ time in the worst case.
- However, if all n key-value pairs to be stored in the heap are given in advance, such as during the first phase of the heap-sort algorithm, there is an alternative ***bottom-up*** construction method that runs in $O(n)$ time.
- we describe this bottom-up heap construction assuming the number of keys, n, is an integer such that $n = 2^{h+1} - 1$.
  - That is, the heap is a complete binary tree with every level being full, so the heap has height $h = \log(n+1)-1$.

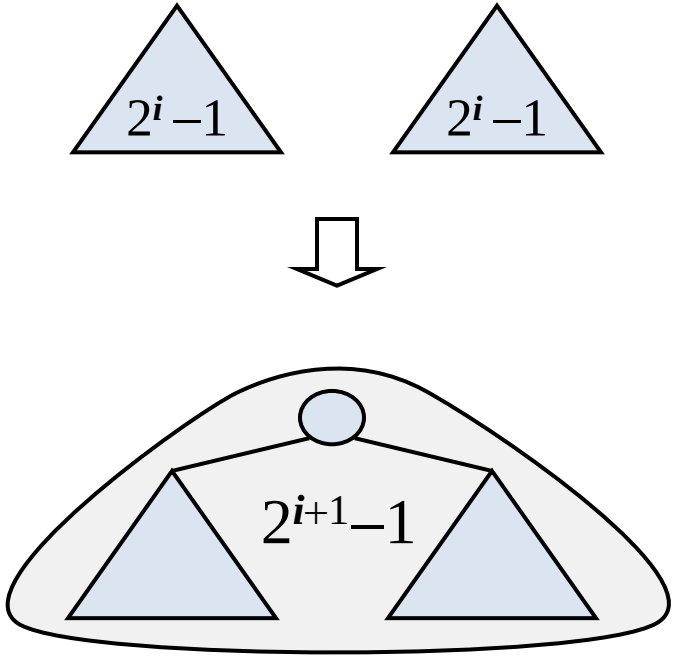
MERGING TWO HEAPS

- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



BOTTOM-UP HEAP CONSTRUCTION

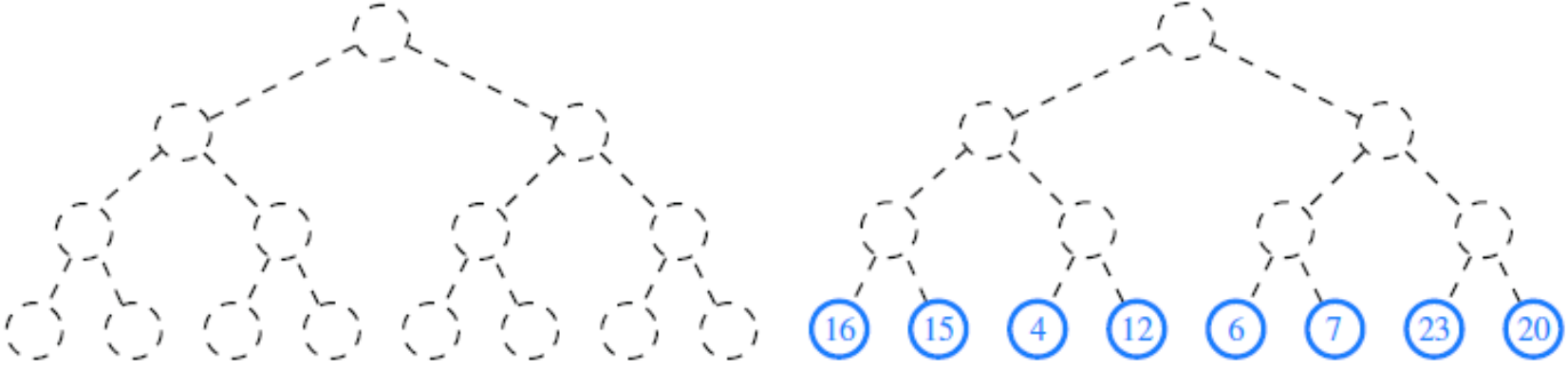
- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys



EXAMPLE OF **BOTTOM-UP HEAP CONSTRUCTION**

Consists of the following $h+1 = \log(n+1)$ steps:

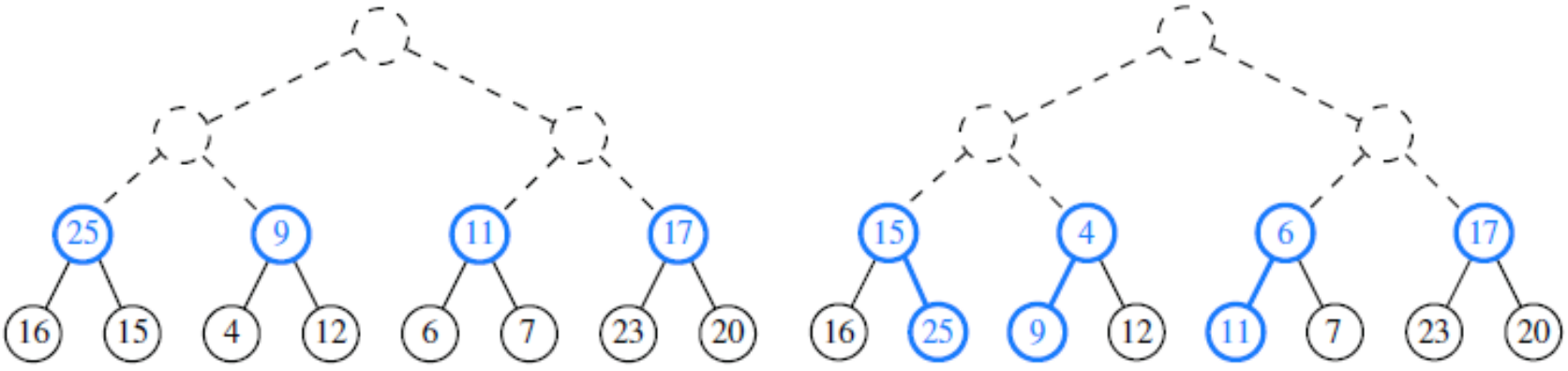
1 > Construct $(n+1)/2$ elementary heaps storing one entry each.



EXAMPLE

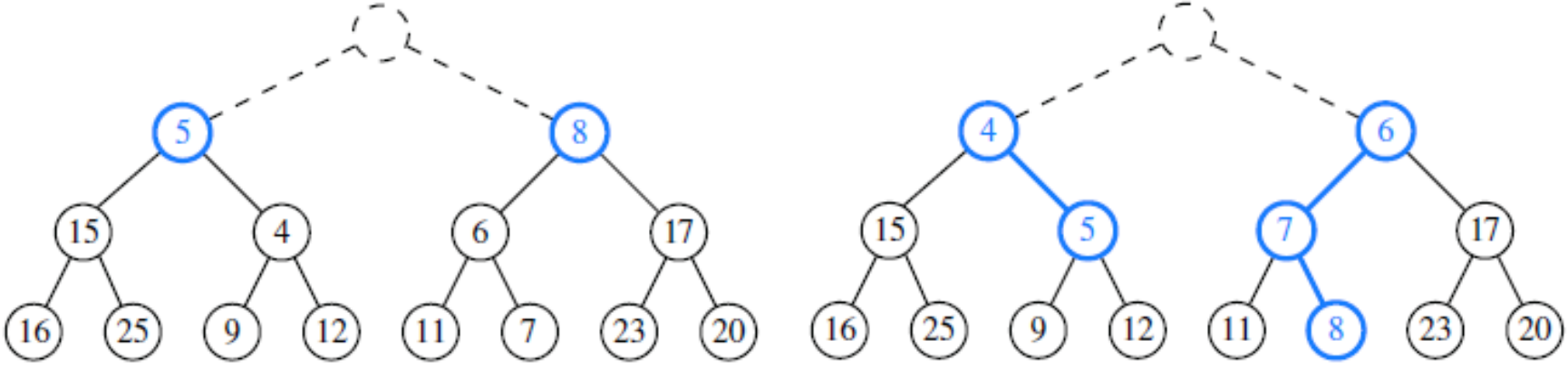
2 > Form $(n+1)/4$ heaps, each storing three entries, by joining pairs of elementary heaps and adding a new entry.

- The new entry is placed at the root and may have to be swapped with the entry stored at a child to preserve the heap-order property.



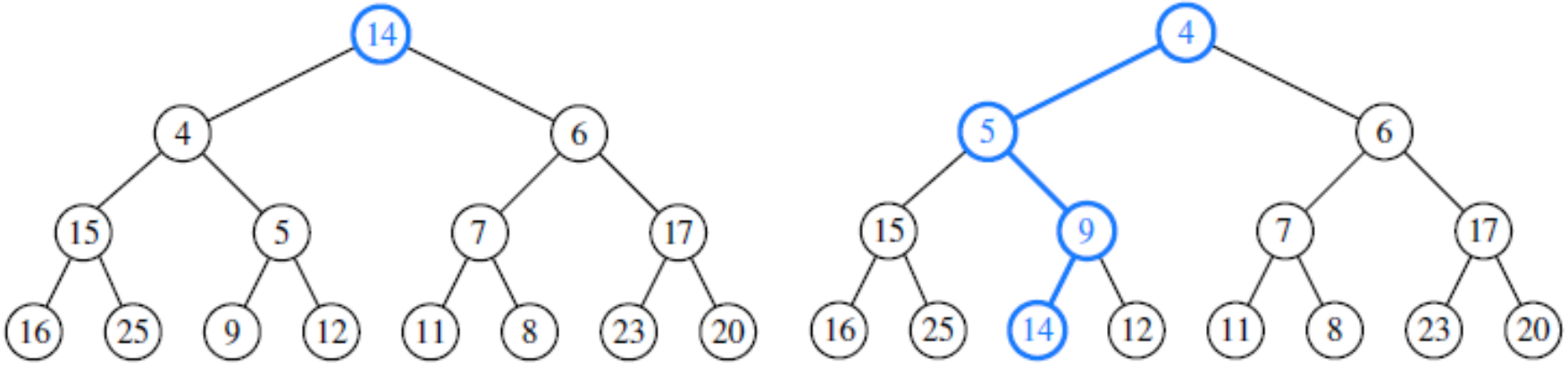
EXAMPLE (CONTD.)

i> In the generic i^{th} step, $2 \le i \le h$, we form $(n+1)/2^i$ heaps, each storing $2^i - 1$ entries, by joining pairs of heaps storing $(2^{i-1} - 1)$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property



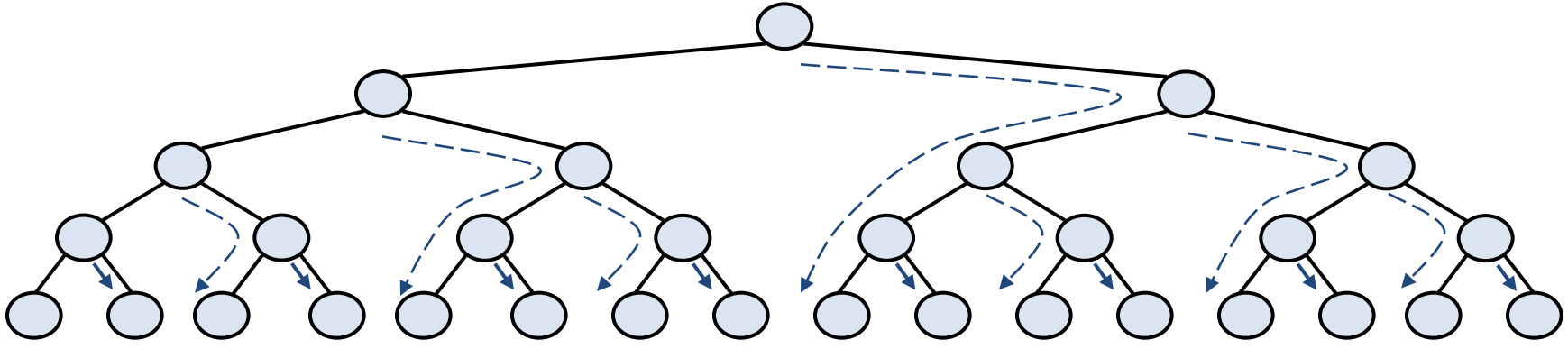
EXAMPLE (CONTD.)

h+1> In the last step, storing all the *n* entries, by joining two heaps storing (*n*–1)/2 entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.



ANALYSIS

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort



USING THE JAVA.UTIL.PRIORITYQUEUE CLASS

- Difference:
- managing keys and values:
  - Our public interface distinguishes between keys and values,
  - the java.util.PriorityQueue class relies on a single element type that is treated as a key.
    - If a user wishes to insert distinct keys and values, the burden is on the user to define and insert appropriate composite objects, and to ensure that those objects can be compared based on their keys.

Our Priority Queue ADT	java.util.PriorityQueue Class
insert(k, v)	add(**new** SimpleEntry(k, v))
min( )	peek( )
removeMin( )	remove( )
size( )	size( )
isEmpty( )	isEmpty( )

RECALL PQ SORTING

- We use a priority queue
  - Insert the elements with a series of insert operations
  - Remove the elements in sorted order with a series of removeMin operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: $O(n^2)$ time
  - Sorted sequence gives insertion-sort: $O(n^2)$ time
- Can we do better?

Algorithm *PQ-Sort*(*S*, *C*)
Input sequence *S*, comparator *C* for the elements of *S*
Output sequence *S* sorted in increasing order according to *C*
P ← priority queue with comparator *C*
while ¬*S.isEmpty* ()
e ← *S.remove* (*S. first* ())
P.insert (*e*, *e*)
while ¬*P.isEmpty*()
e ← *P.removeMin*().*getKey*()
S.addLast(*e*)

HEAP SORT

- Consider the pqSort scheme, this time using a heap-based implementation of the priority queue
- Phase 1: insert all data into heap:
  - takes $O(n \log n)$ time. (Could be improved to $O(n)$ with bottom-up construction)
- Phase 2: removeMin all data in the heap
  - j th removeMin operation runs in $O(\log(n - j + 1))$, since the heap has $n - j + 1$ entries at the time the operation
  - Summing over all j, this phase takes $O(n \log n)$ time
- Overall: The heap-sort algorithm sorts a sequence S of n elements in $O(n \log n)$ time, assuming two elements of S can be compared in $O(1)$ time.