CS 2233-01

Data Structures and Algorithms

Instructor: Dr. Qingguo Wang
College of Computing & Technology
Lipscomb University

Outline

- Priority Queues
- Heap
- Adaptable 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

Priority Queues

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, or null if the the priority queue is empty
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Example

- A sequence of priority queue methods:

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key
- Mathematical concept of total order relation $\leq$
  - Comparability property: either $x \leq y$ or $y \leq x$
  - Antisymmetric property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Entry ADT

- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry

- As a Java interface:

```
/**
 * Interface for a key-value
 * pair entry
 **/
public interface  Entry<K,V>
  {
      K getKey();
      V getValue();
  }
```

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- Primary method of the Comparator ADT
- compare(a, b): returns an integer i such that
  - i < 0 if a < b,
  - i = 0 if a = b
  - i > 0 if a > b
  - An error occurs if a and b cannot be compared.

Example Comparator

- Lexicographic comparison of 2-D points:

```
/** Comparator for 2D points under the
    standard lexicographic order. */
public class Lexicographic implements
    Comparator {
  int xa, ya, xb, yb;
  public int compare(Object a, Object b)
   throws ClassCastException {
    xa = ((Point2D) a).getX();
    ya = ((Point2D) a).getY();
    xb = ((Point2D) b).getX();
    yb = ((Point2D) b).getY();
    if (xa != xb)
        return (xb - xa);
    else
        return (yb - ya);
  }
}
```

- Point objects:

```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D        {
  protected int xc, yc; // coordinates
  public Point2D(int x, int y) {
    xc = x;
    yc = y;
  }
  public int getX() {
        return xc;
  }
  public int getY() {
        return yc;
  }
}
```

Sequence-based Priority Queue

- Implementation with an **unsorted list**

- Performance:
  - insert ?
  - removeMin ?
  - min ?

- Implementation with a **sorted list**

- Performance:

Sequence-based Priority Queue

- Implementation with an **unsorted list**

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a **sorted list**

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

Priority Queue Sorting

- We can use a priority queue to sort a list of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Priority Queue Sorting

- We can use a priority queue to sort a list of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
  Input list *S*, comparator *C* for the elements of *S*
  Output list *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*, ∅)
  while ¬*P.isEmpty*()
    e ← *P.removeMin*().*getKey*()
    S.addLast(*e*)

Example 1: PQ implemented with a sorted sequence

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to
  $$1 + 2 + \ldots + n$$
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Example 2: PQ implemented with an unsorted sequence

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

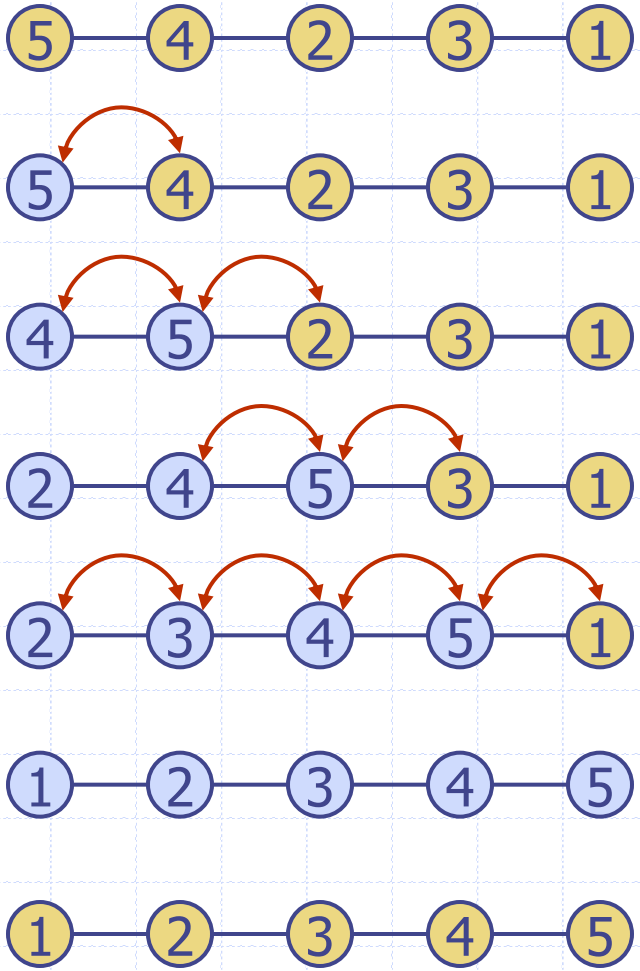
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time

In-place Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



Exercise

- An unsorted list of 5 elements is given in an array: $A[0...5] = \{15, 3, 9, 31, 11\}$. Apply the selection sort algorithm to sort the array A.

Outline

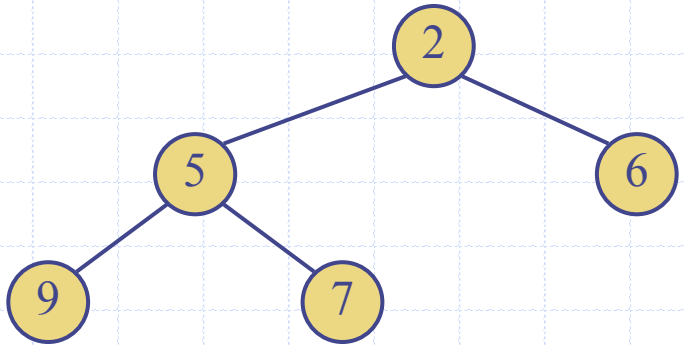
- Priority Queues
- Heap
- Adaptable Priority Queues

Recaps of 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, or null if the the priority queue is empty
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

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

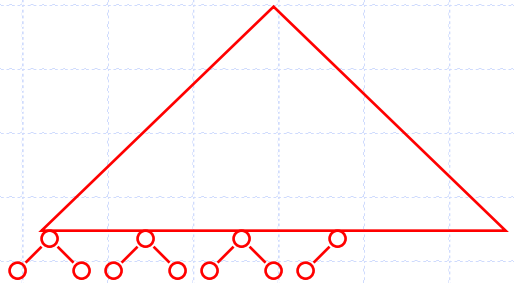
Heaps



Heaps

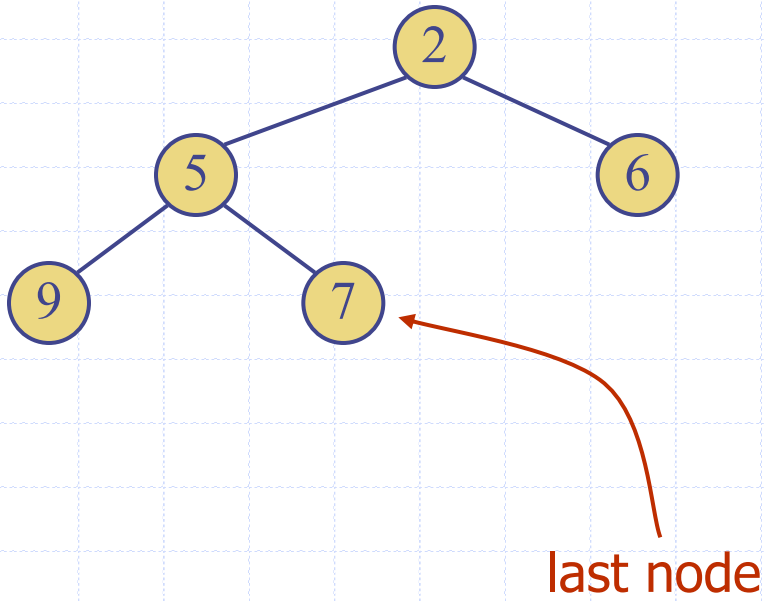
- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
- Heap-Order: for every internal node v other than the root,
  $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i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

- All its levels are full except possibly the last level, where only some rightmost keys may be missing



Heaps (cont.)

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



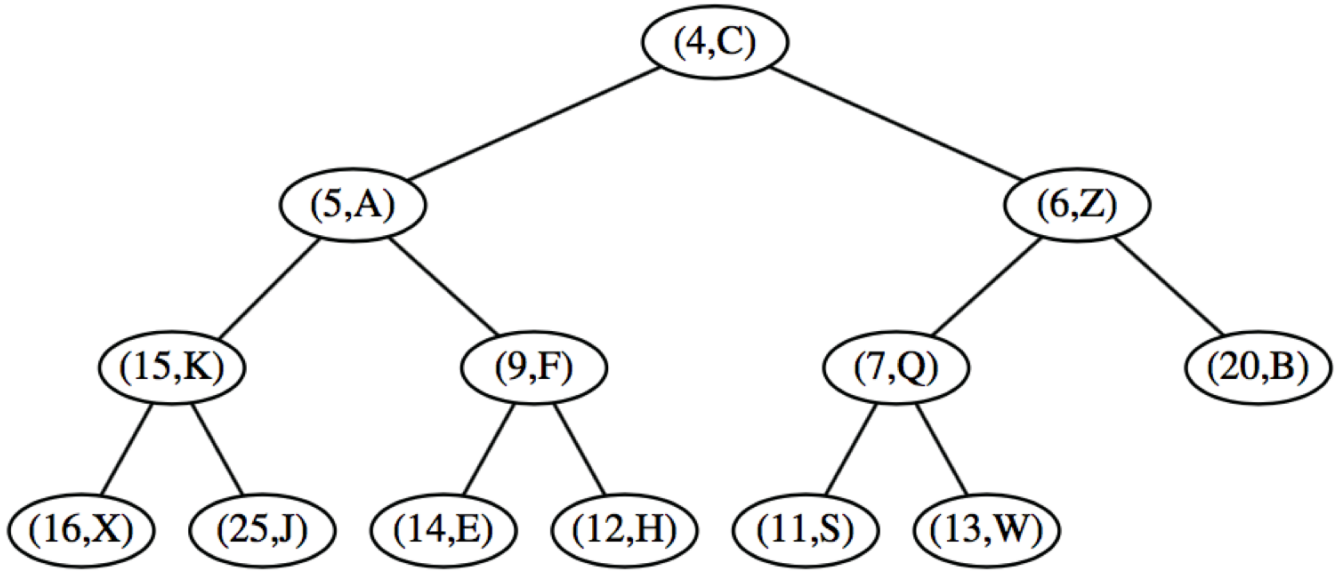
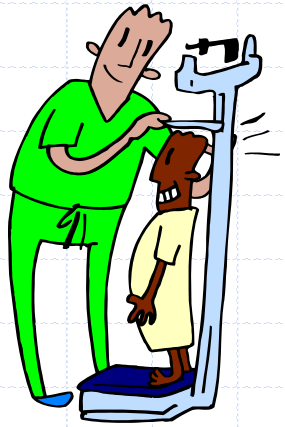


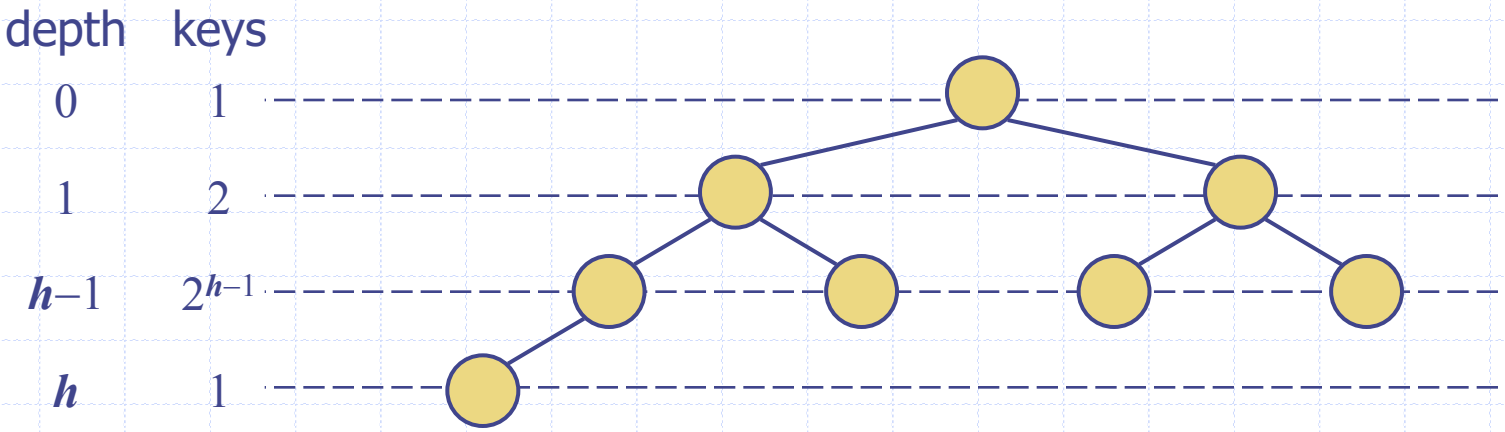
Figure 9.1: Example of a heap storing 13 entries with integer keys. The last position is the one storing entry $(13, W)$.

Height of a Heap



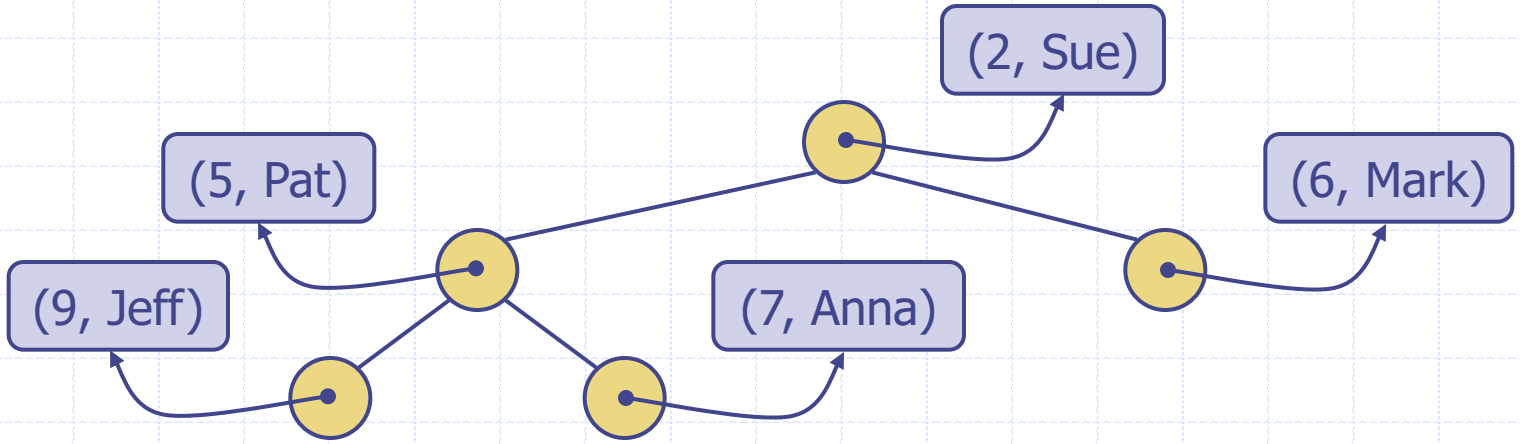
- Theorem: A heap storing n keys has height $O(\log n)$

  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 $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h$
  - Thus, $h \leq \log n$



Heaps and Priority Queues

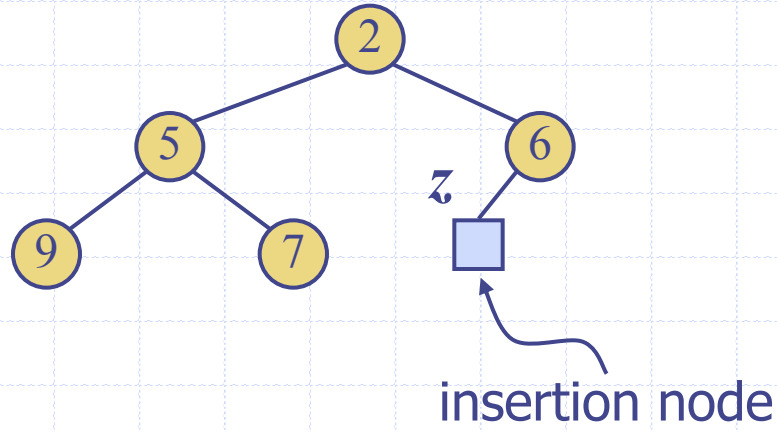
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



Insertion into a Heap

- Method **insert** Item of the priority queue ADT corresponds to the insertion of a key k to the heap

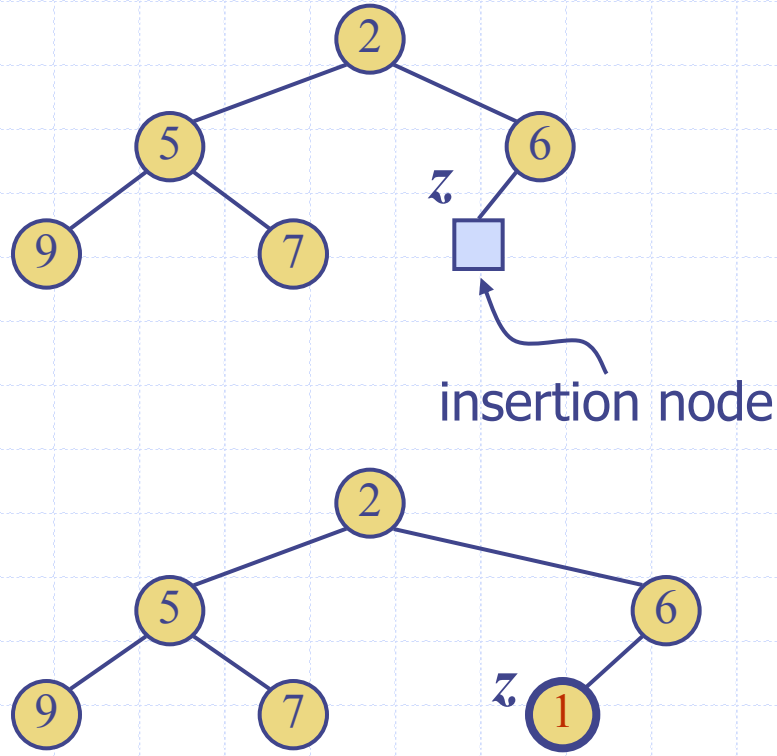
- The insertion algorithm consists of three **steps**
  - Find the insertion node z (the new last node)
  - Store k at z



Insertion into a Heap

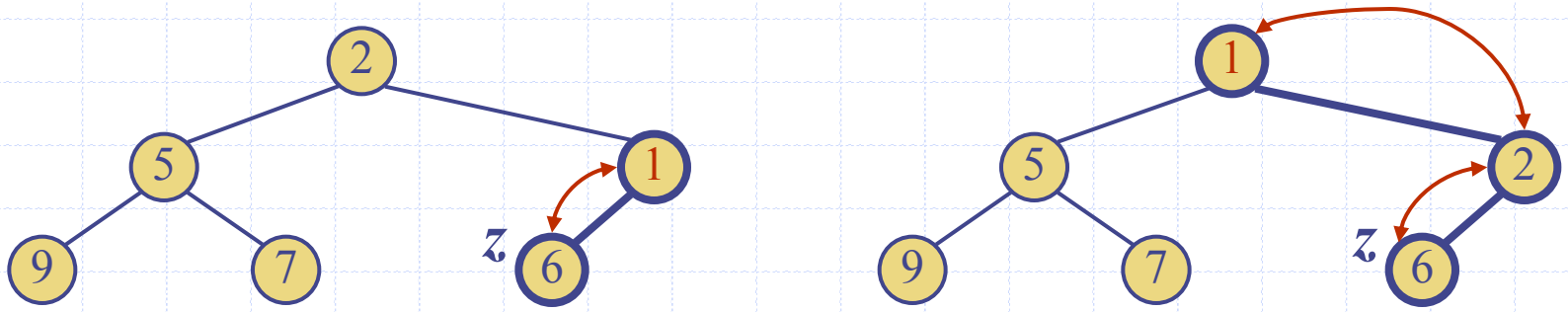
- Method **insert** Item of the priority queue ADT corresponds to the insertion of a key k to the heap

- The insertion algorithm consists of three **steps**
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



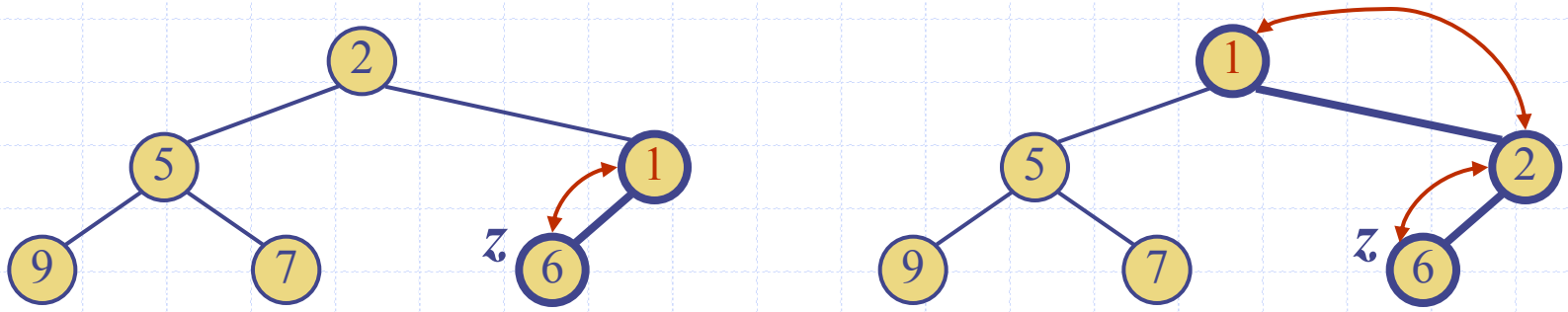
Upheap

- 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
  - terminating when the key k reaches the root or a node whose parent has a key smaller than or equal to k

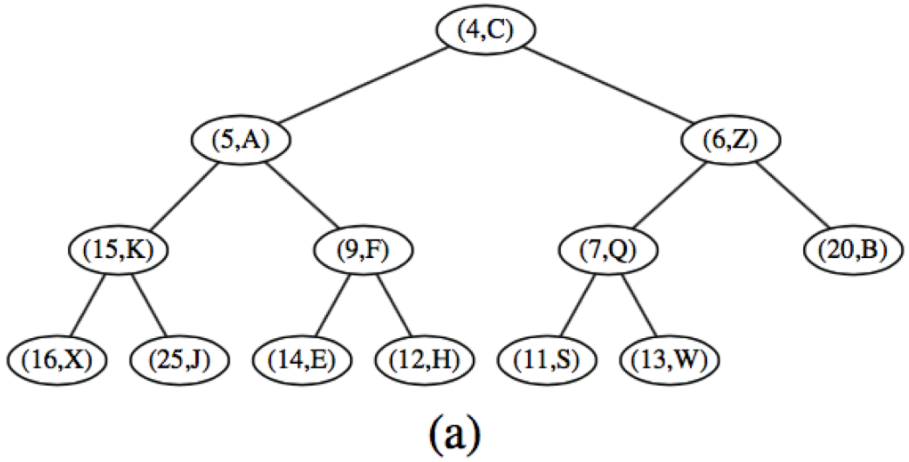


Upheap

- 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
  - terminating 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

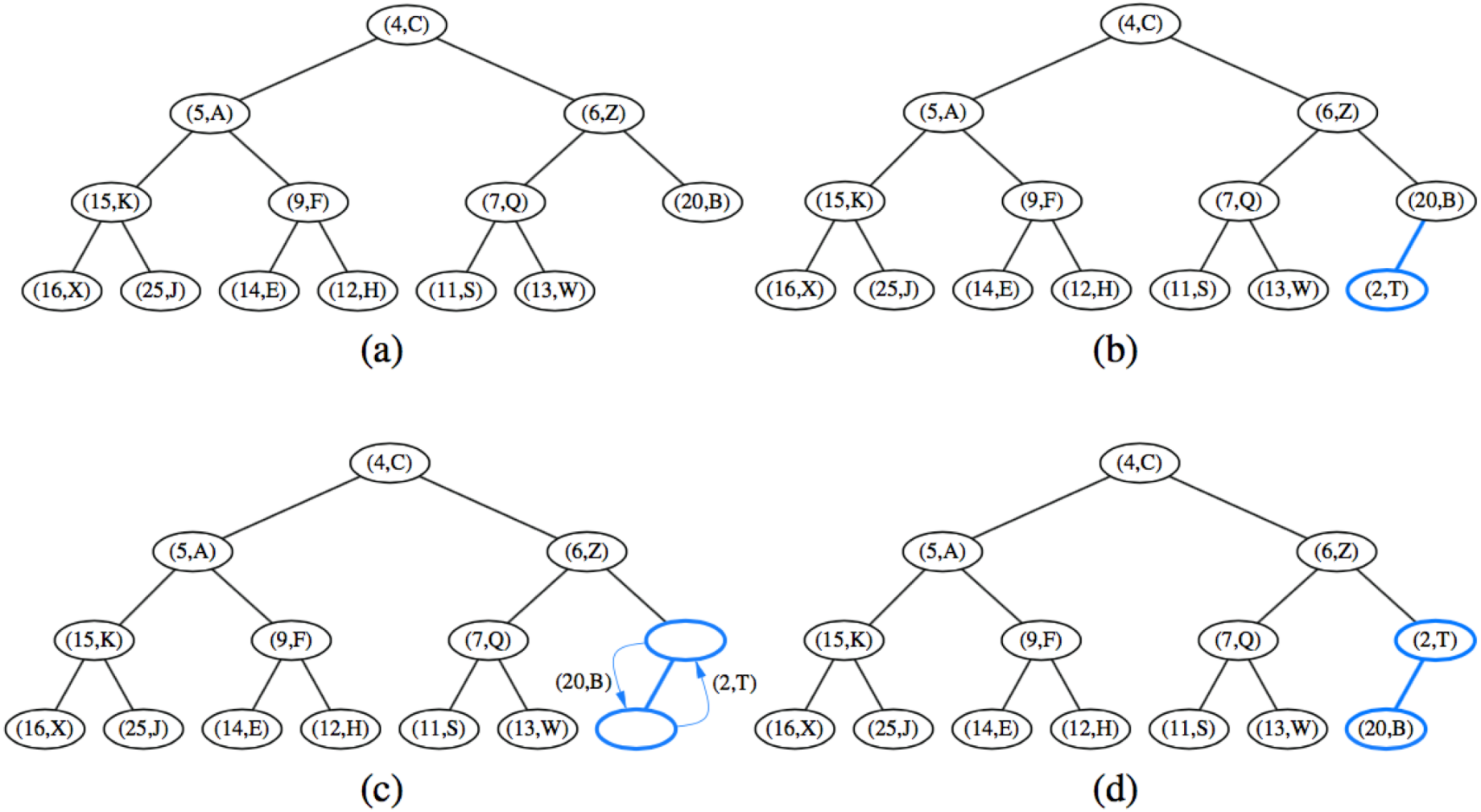


Example 1

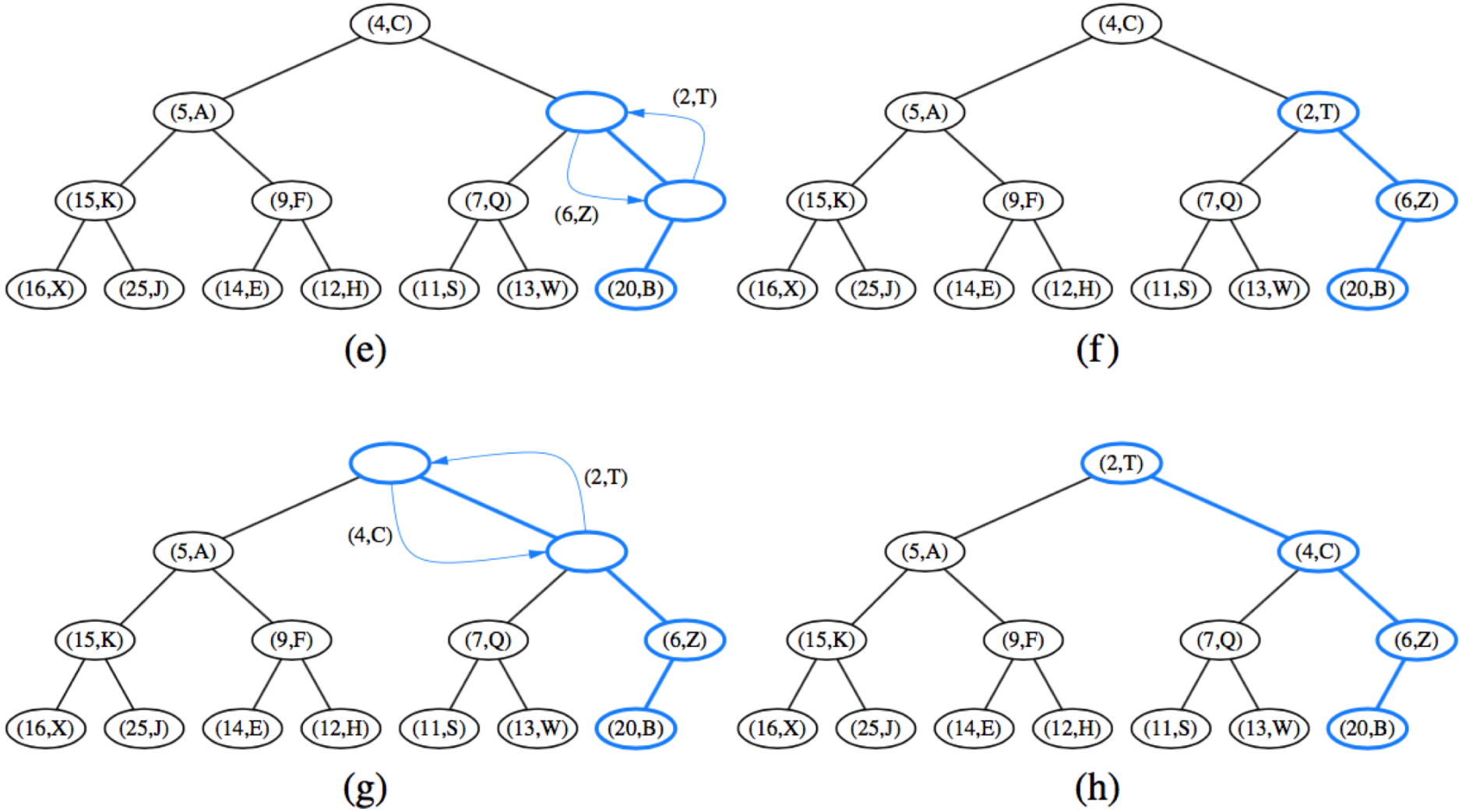


(a)

Example 1

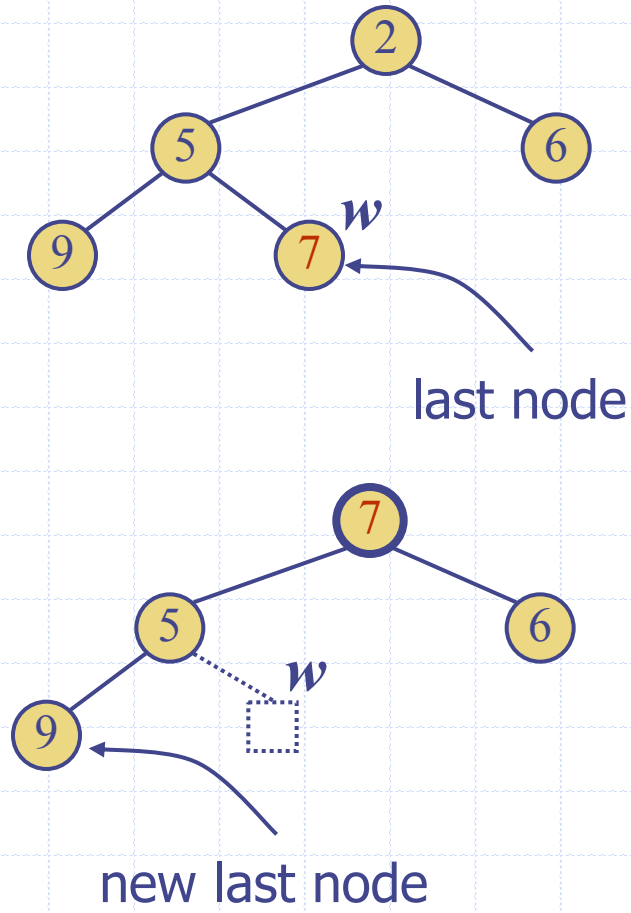

(4,C)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(13,W)
(a)
(4,C)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(13,W)
(2,T)
(b)
(4,C)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(2,T)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(13,W)
(c)
(4,C)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(2,T)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(13,W)
(20,B)
(d)

Example 1 (cont.)



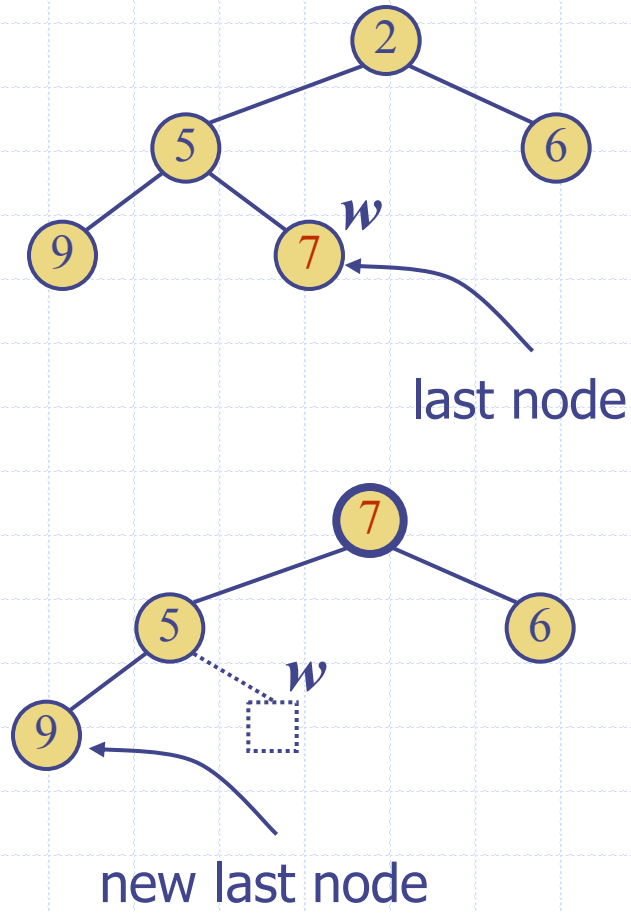
Removal from a Heap

- 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



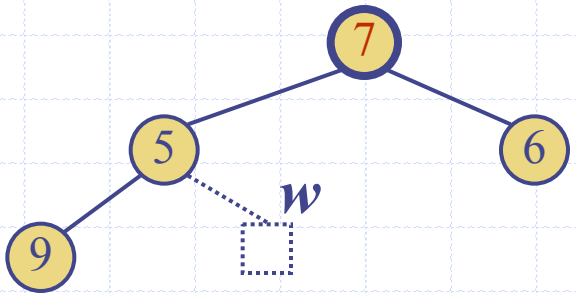
Removal from a Heap

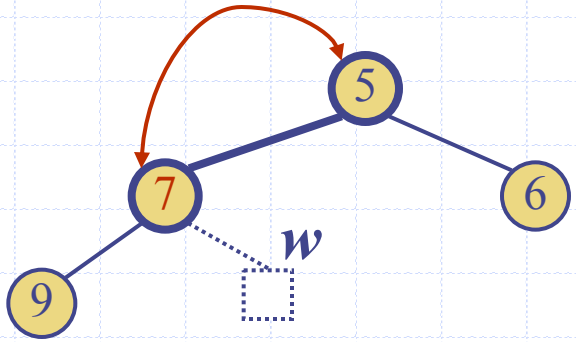
- Method **removeMin** of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three **steps**
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)



Downheap

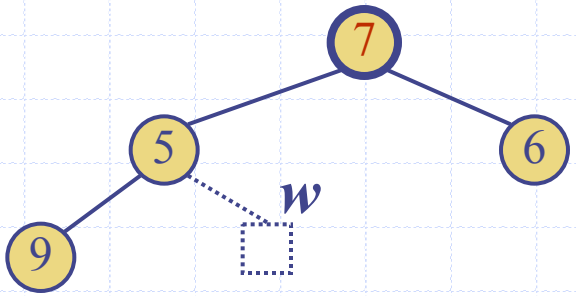
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm **downheap** restores the heap-order property by
  - swapping key k along a downward path from the root
  - terminating when key k reaches a leaf or a node whose children have keys greater than or equal to k

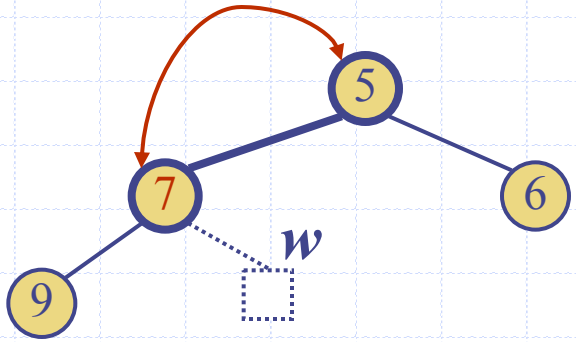




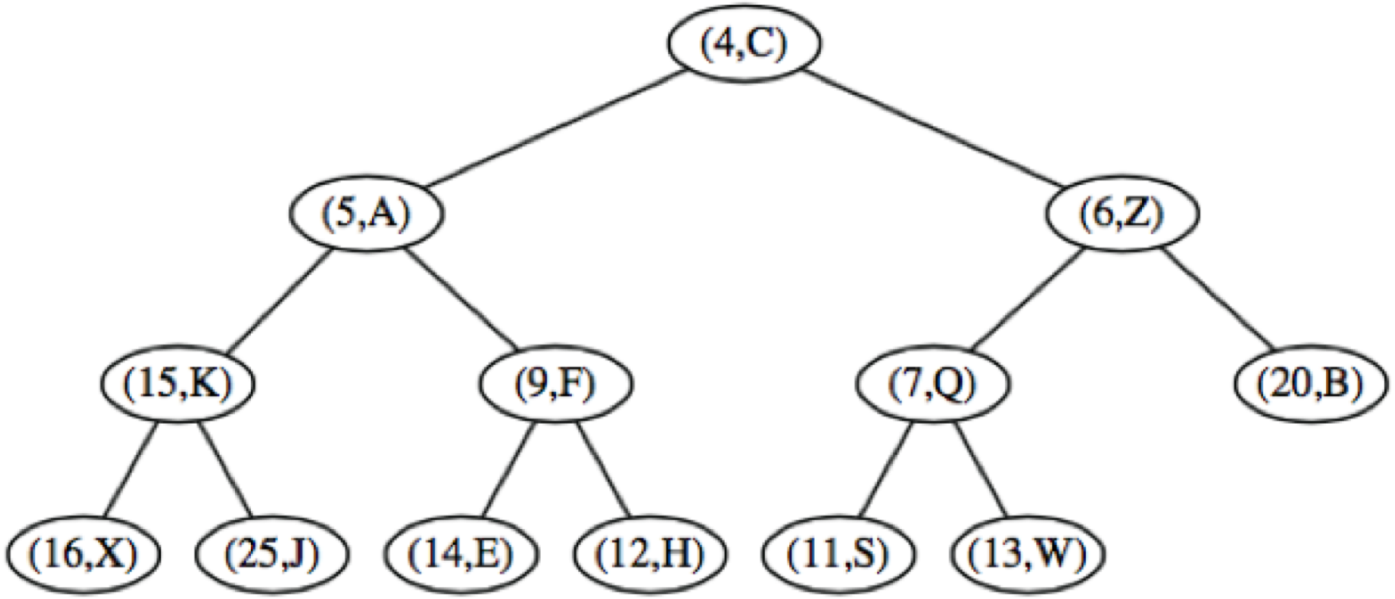
Downheap

- 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
  - terminating 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

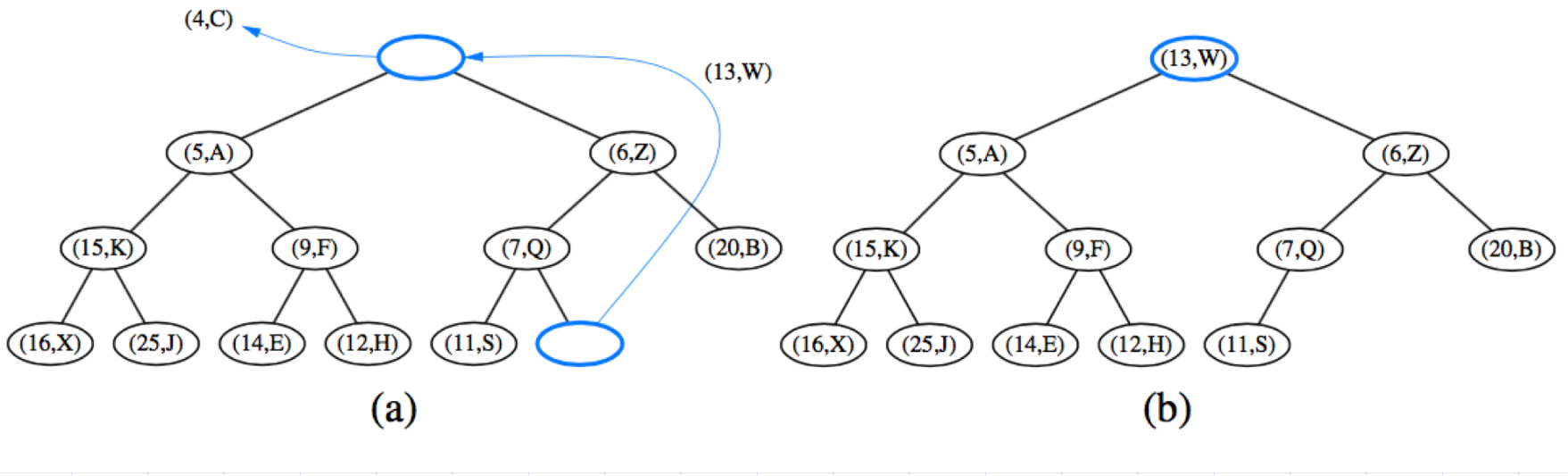




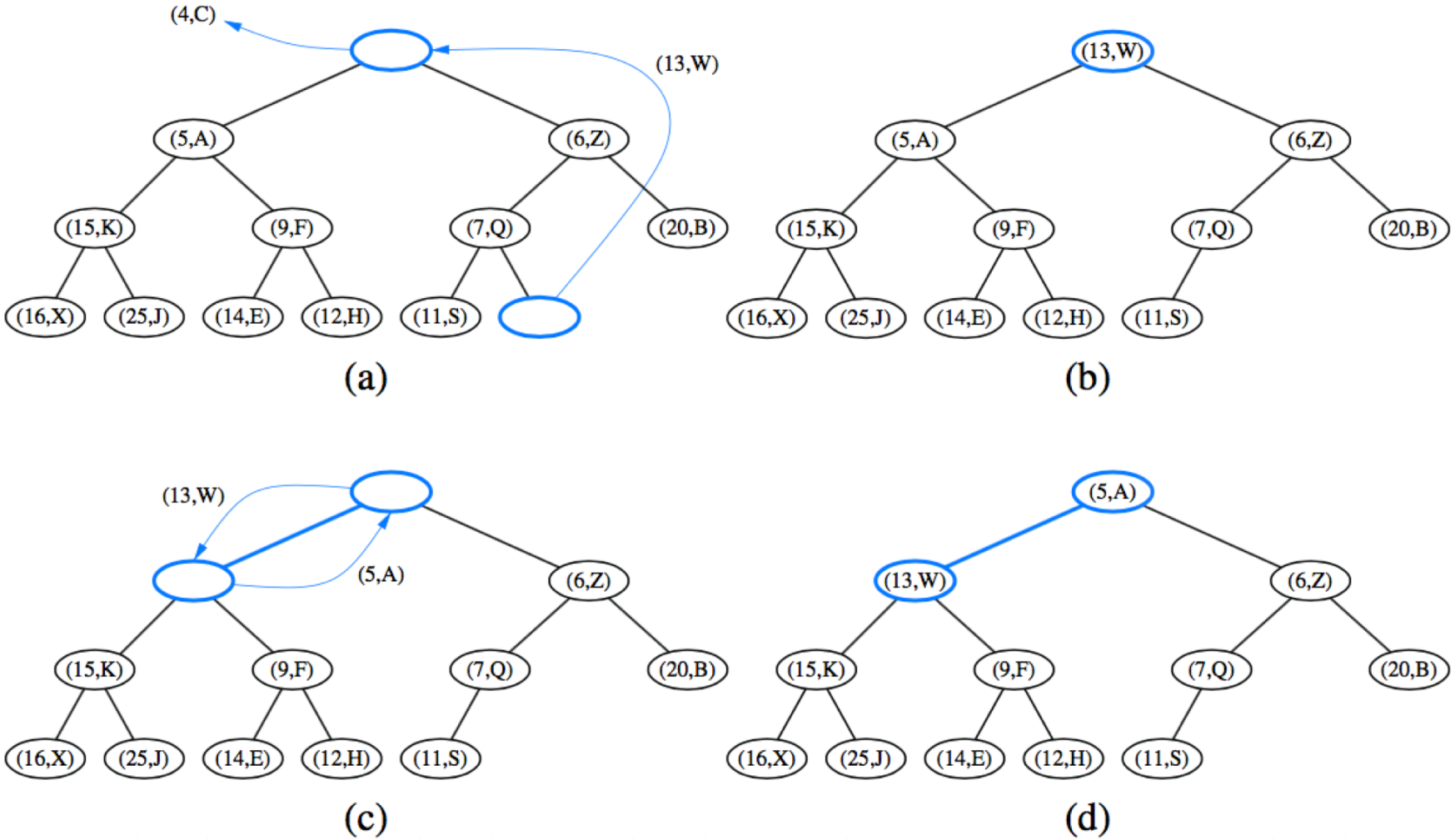
Example 2



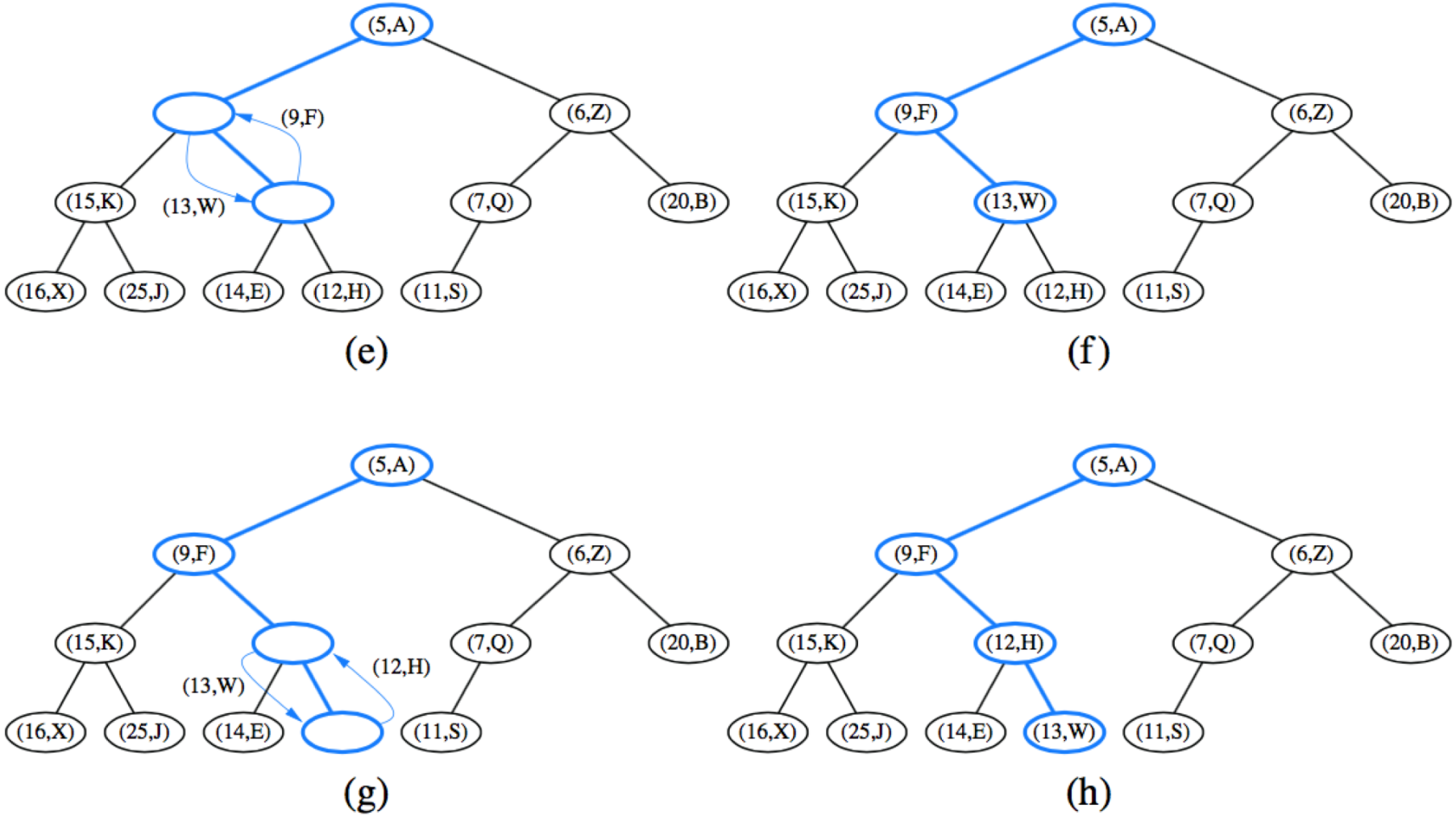
Example 2 (cont.)



Example 2 (cont.)



Example 2 (cont.)



Heap-Sort

- Using a heap-based priority queue, we can sort a sequence of n elements in **O(??)** time/space

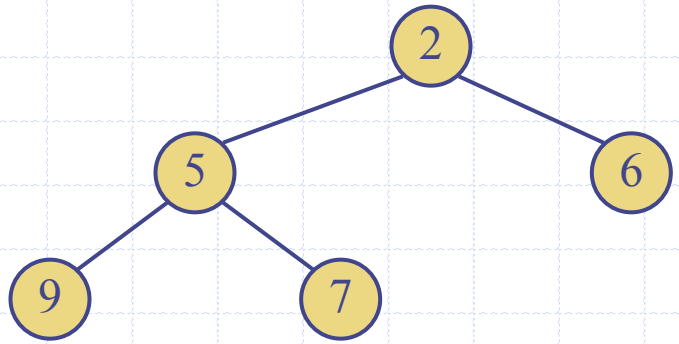
- The resulting algorithm is called heap-sort

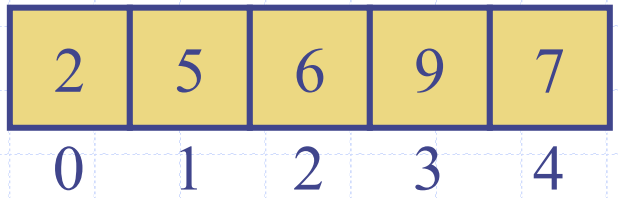
Recaps of Heap-Sort

- 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

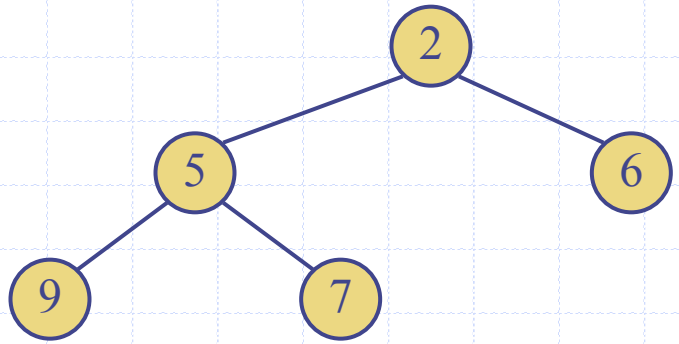
- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- **Links** between nodes are not explicitly stored

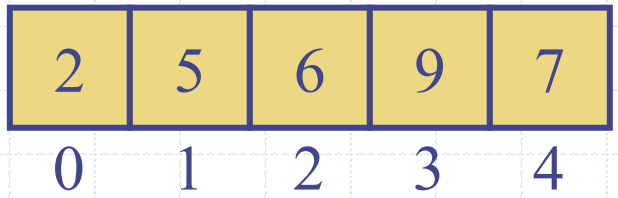




Array-based Heap Implementation

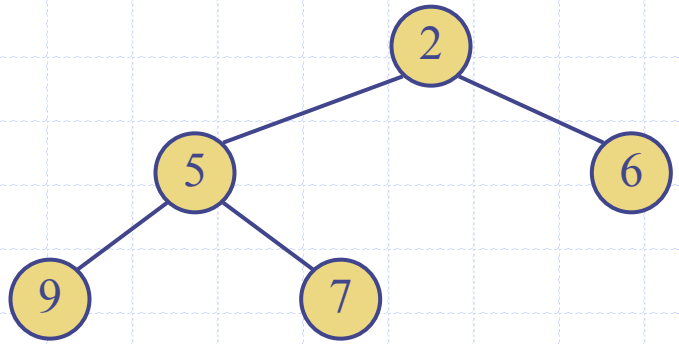
- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- **Links** between nodes are not explicitly stored
- Operation **add** corresponds to inserting at rank $n + 1$
- Operation **remove_min** corresponds to removing at rank 0

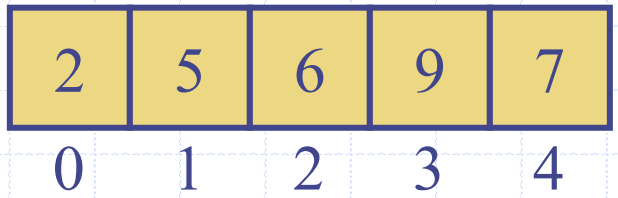




Array-based Heap Implementation

- 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
- Operation **add** corresponds to inserting at rank $n + 1$
- Operation **remove_min** corresponds to removing at rank 0
- Yields in-place heap-sort





Java Implementation

```
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( ); }
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   }
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   }
```

Java Implementation, 2

```
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
47   // public methods
48   /** Returns the number of items in the priority queue. */
49   public int size( ) { return heap.size( ); }
50   /** Returns (but does not remove) an entry with minimal key (if any). */
51   public Entry<K,V> min( ) {
52     if (heap.isEmpty( )) return null;
53     return heap.get(0);
54   }
```

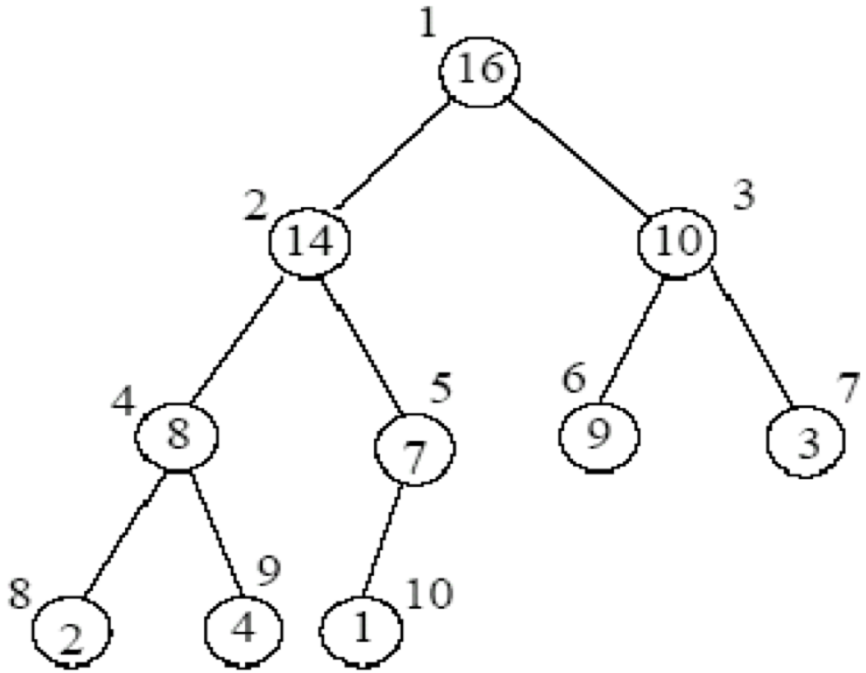
Java Implementation, 3

```
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    }
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  }
```

© 2014 Goodr

Exercise 2

- What is the array representation of the following heap?

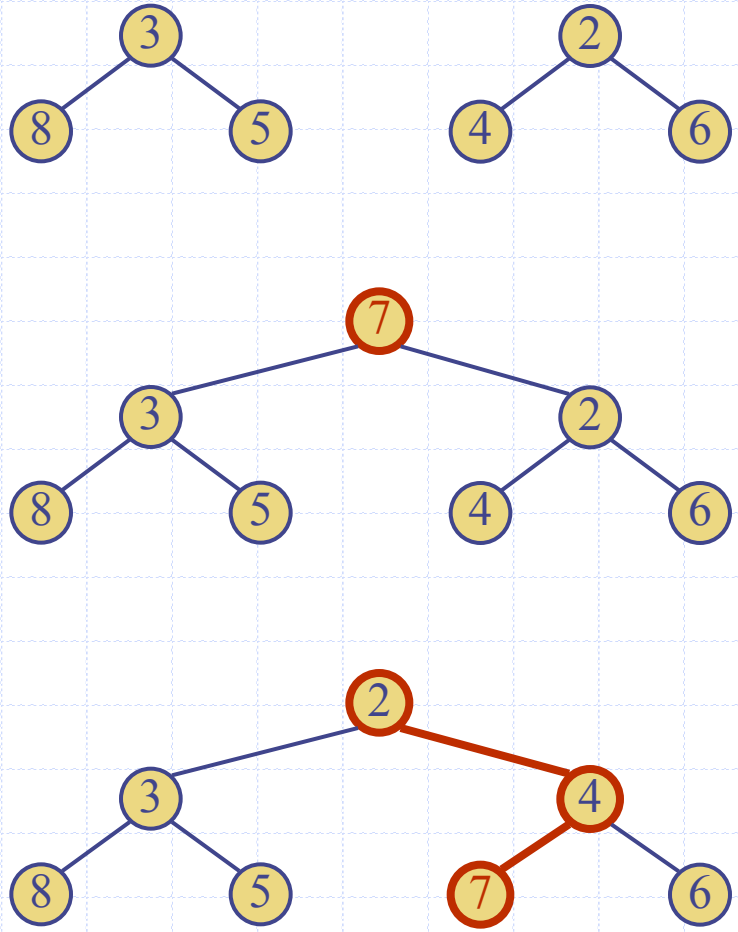


Exercise 3

- Verify an array [6 7 12 10 15 17] is a representation of a min-heap.

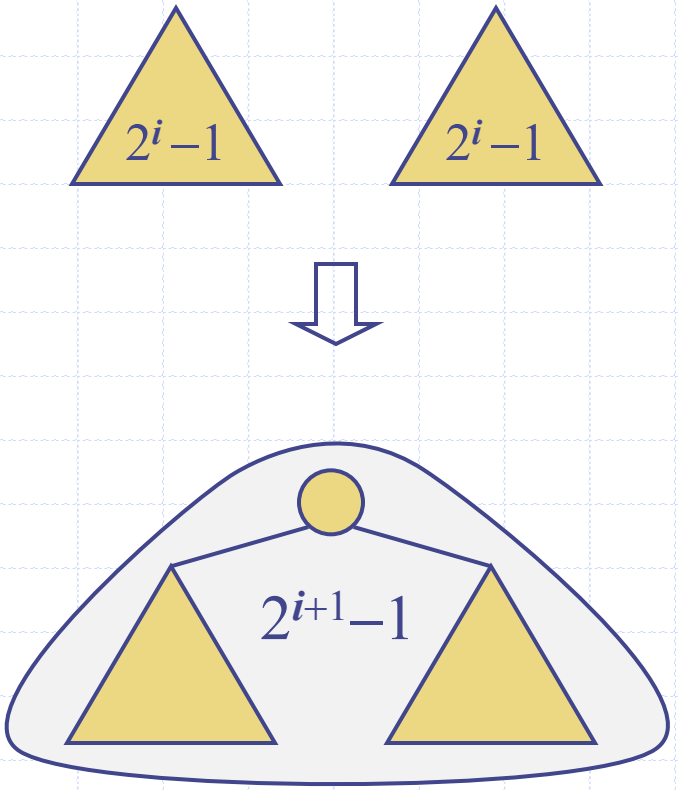
Merging Two Heaps

- 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

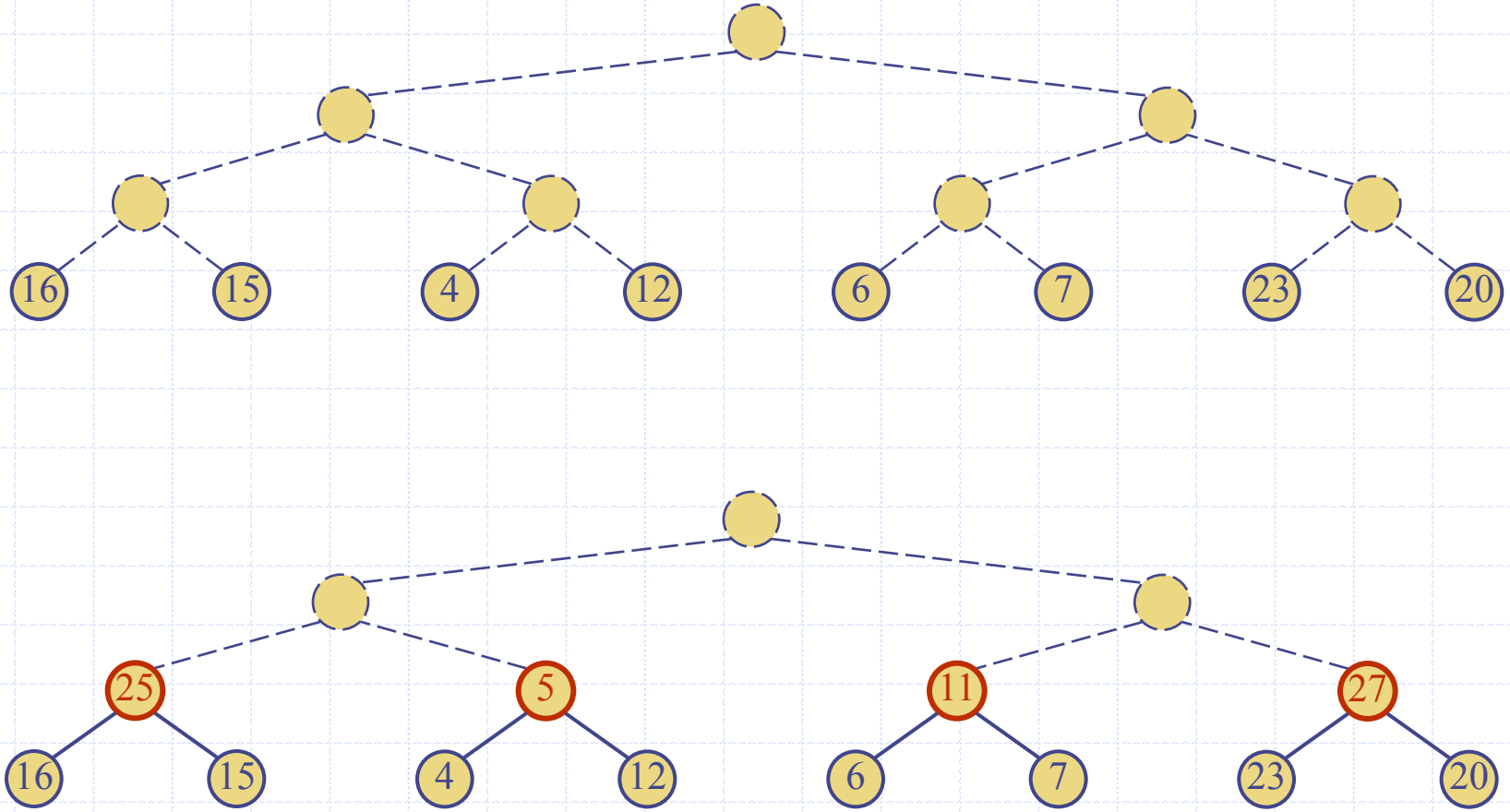


Bottom-up Heap Construction

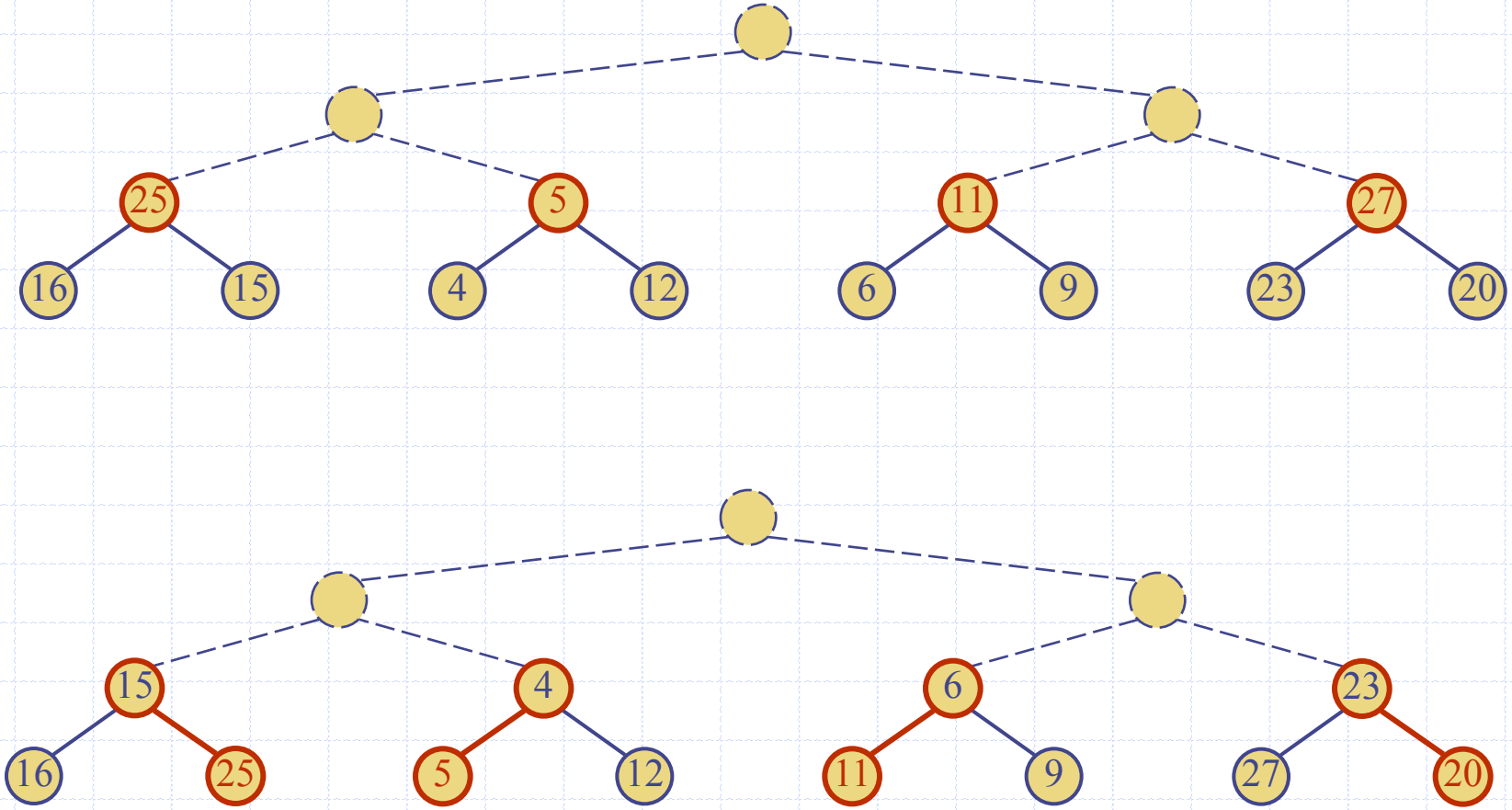
- We can construct a heap storing n given keys using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys



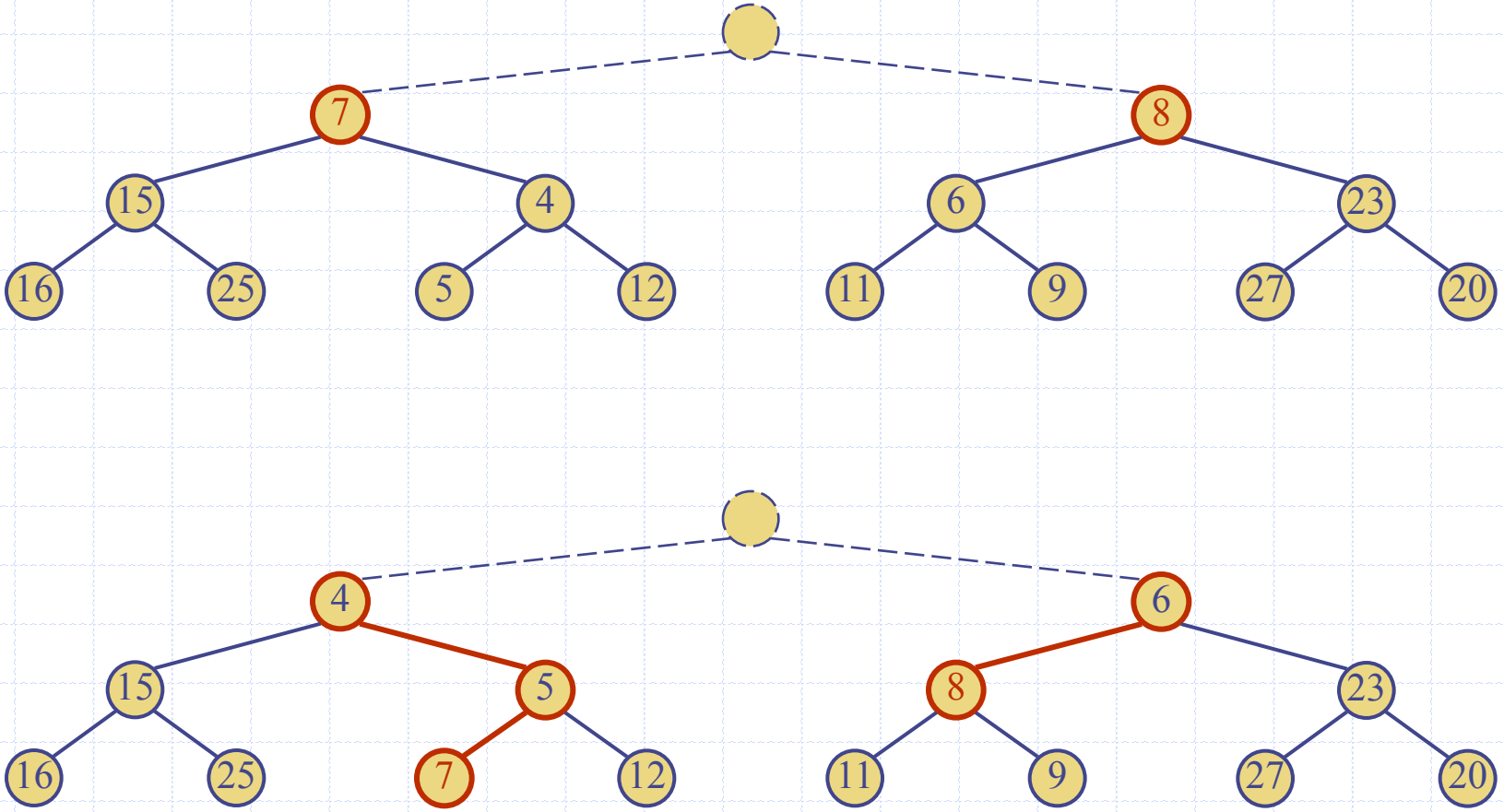
Example



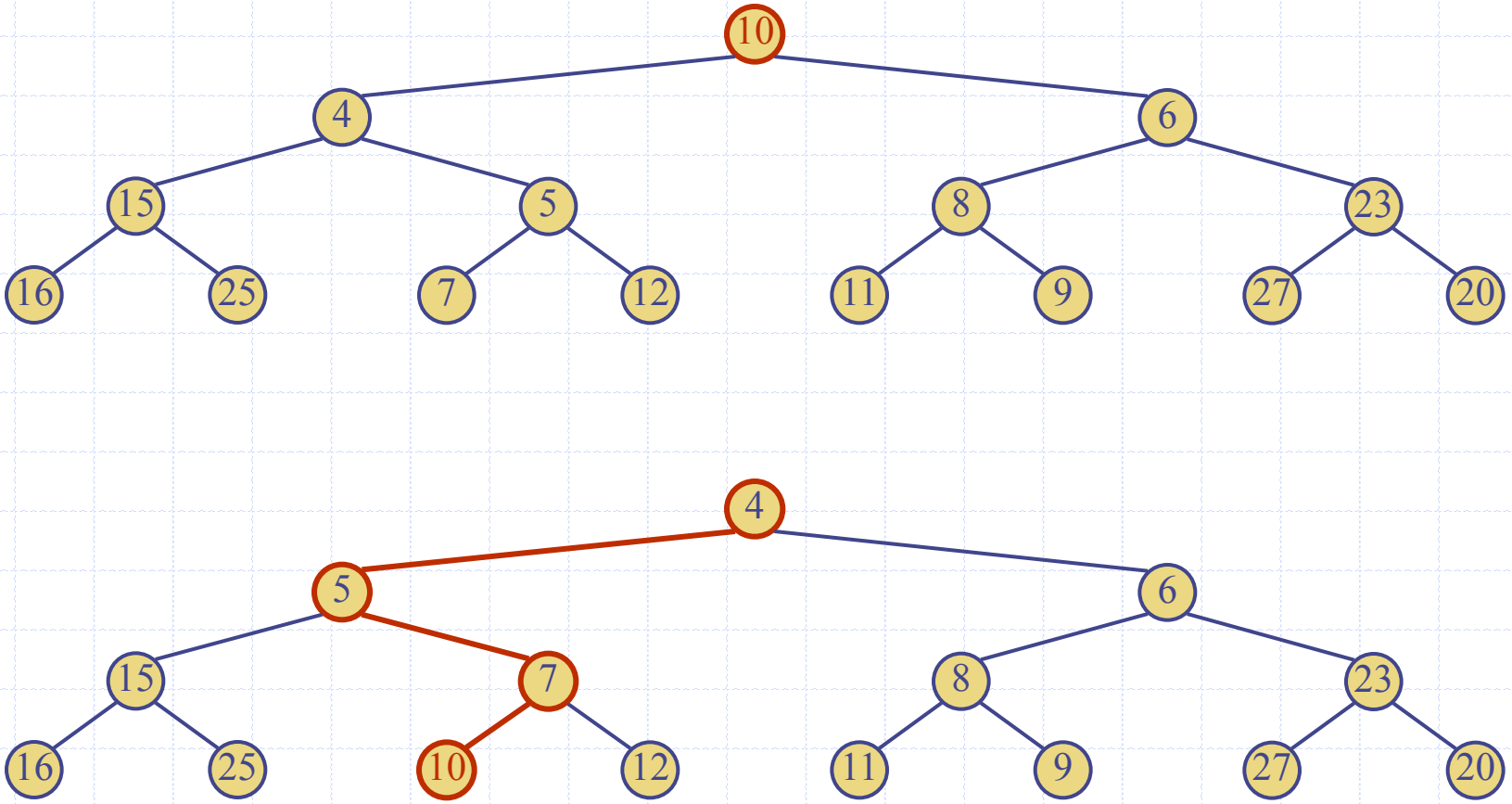
Example (contd.)



Example (contd.)



Example (end)



Exercise 4

- Construct a heap from the following input sequence: (2, 5, 16, 4, 10, 23, 39, 18, 26, 15).

Analysis of Bottom-Up Heap Construction

- Assume a full tree: $n = 2^{h+1} - 1$;
- Let h be height of tree: $h = \lfloor \log_2 n \rfloor$, (or $\lceil \log_2 (n+1) \rceil - 1 = h$)
- Each key level i will travel to the leaf (level h) in worst case
- Moving to next level down requires 2 comparisons (*find larger child* + *determine whether exchange is required*)
  - Total comparisons involving key at level i: $2(h-i)$
- Total for heap construction phase:

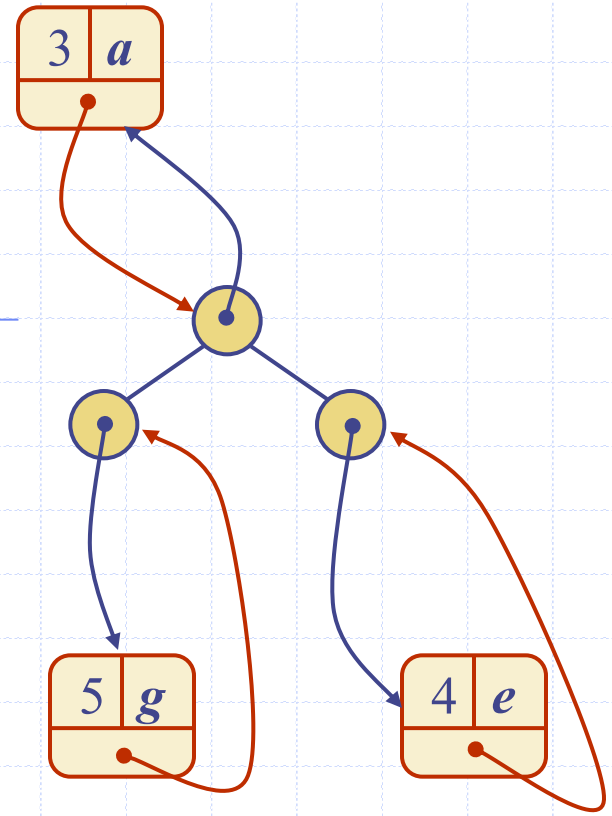
$$\sum_{i=0}^{h-1} \sum_{level\ i\ keys} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)) = \Theta(n)$$

Outline

- Priority Queues
- Heap
- Adaptable 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

Adaptable Priority Queues



Entry and Priority Queue ADTs

- An entry stores a (key, value) pair
- Entry ADT methods:
  - getKey(): returns the key associated with this entry
  - getValue(): returns the value paired with the key associated with this entry
- Priority Queue ADT:
  - insert(k, x)
    inserts an entry with key k and value x
  - removeMin()
    removes and returns the entry with smallest key
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()

Example

- Online trading system where orders to purchase and sell a stock are stored in two priority queues (one for sell orders and one for buy orders) as (p,s) entries:
  - The key, p, of an order is the price
  - The value, s, for an entry is the number of shares
  - A buy order (p,s) is executed when a sell order (p’,s’) with price p’ $\leq$ p is added (the execution is complete if s’ $\geq$ s)
  - A sell order (p,s) is executed when a buy order (p’,s’) with price p’ $\geq$ p is added (the execution is complete if s’ $\geq$ s)
- What if someone wishes to cancel their order before it executes?
- What if someone wishes to update the price or number of shares for their order?

Methods of the Adaptable Priority Queue ADT

- remove(e): Remove from P and return entry e.
- replaceKey(e,k): Replace with k and return the key of entry e of P; an error condition occurs if k is invalid (that is, k cannot be compared with other keys).
- replaceValue(e,v): Replace with v and return the value of entry e of P.

Example

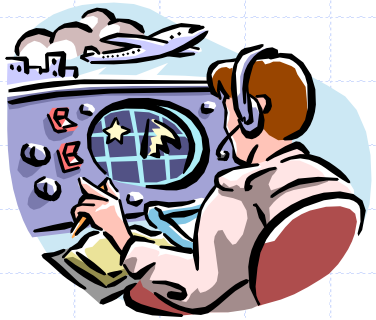
Operation	*Output*	*P*
insert(5,A)	e_1	(5,A)
insert(3,B)	e_2	(3,B),(5,A)
insert(7,C)	e_3	(3,B),(5,A),(7,C)
min()	e_2	(3,B),(5,A),(7,C)
key(e_2)	3	(3,B),(5,A),(7,C)
remove(e_1)	e_1	(3,B),(7,C)
replaceKey(e_2,9)	3	(7,C),(9,B)
replaceValue(e_3,D)	C	(7,D),(9,B)
remove(e_2)	e_2	(7,D)

Locating Entries

- In order to implement the operations remove(e), replaceKey(e,k), and replaceValue(e,v), we need fast ways of locating an entry e in a priority queue.

- We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.

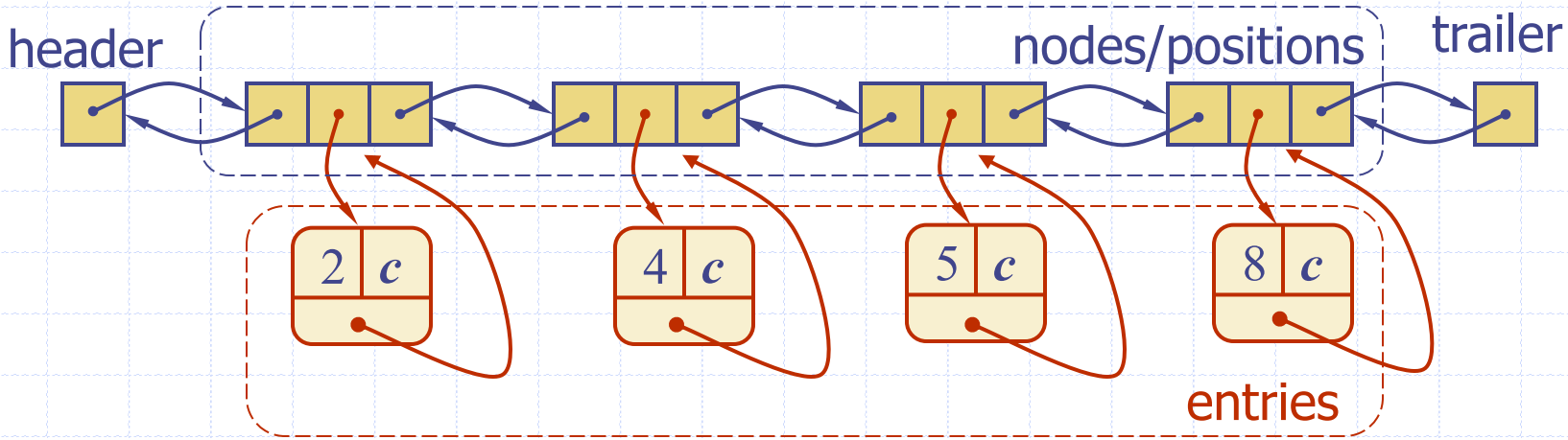
Location-Aware Entries



- A location-aware entry identifies and tracks the location of its (key, value) object within a data structure
- Intuitive notion:
  - Coat claim check
  - Valet claim ticket
  - Reservation number
- Main idea:
  - Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier

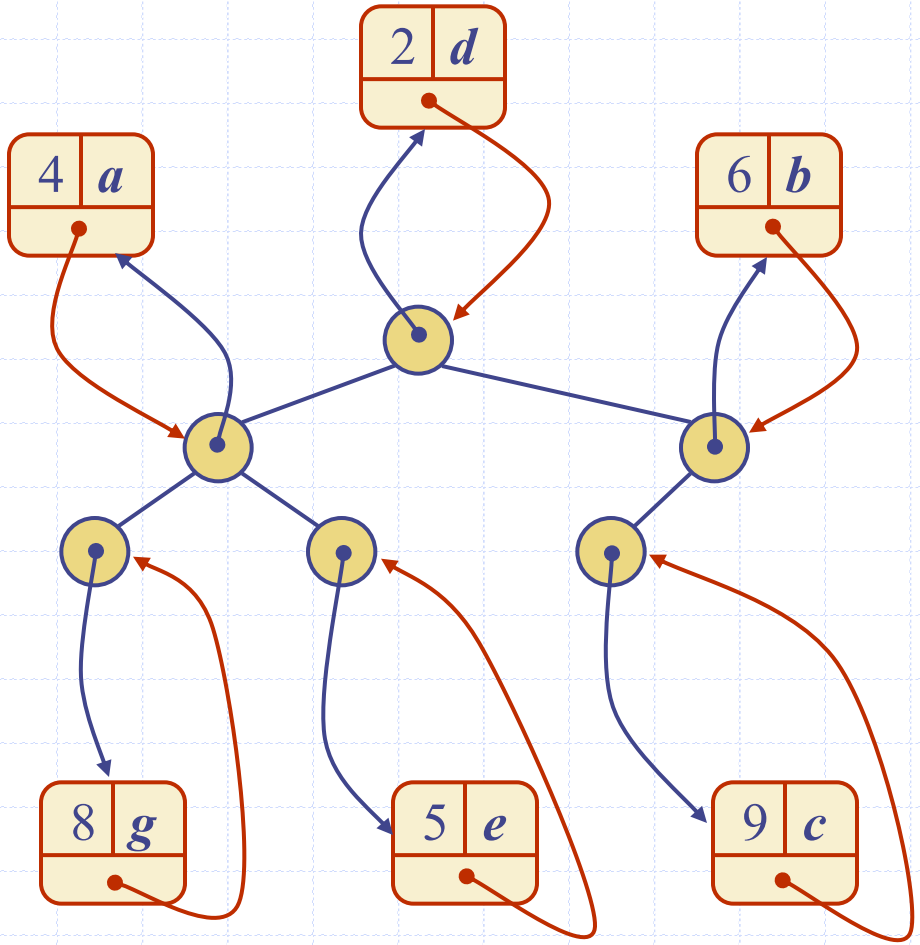
List Implementation

- A location-aware list entry is an object storing
  - key
  - value
  - position (or rank) of the item in the list
- In turn, the position (or array cell) stores the entry
- Back pointers (or ranks) are updated during swaps



Heap Implementation

- A location-aware heap entry is an object storing
  - key
  - value
  - position of the entry in the underlying heap
- In turn, each heap position stores an entry
- Back pointers are updated during entry swaps



Performance

- Improved times thanks to location-aware entries are highlighted in red

Method	Unsorted List	Sorted List	Heap
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(\log n)}$
replaceKey	$\boldsymbol{O(1)}$	$O(n)$	$\boldsymbol{O(\log n)}$
replaceValue	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$