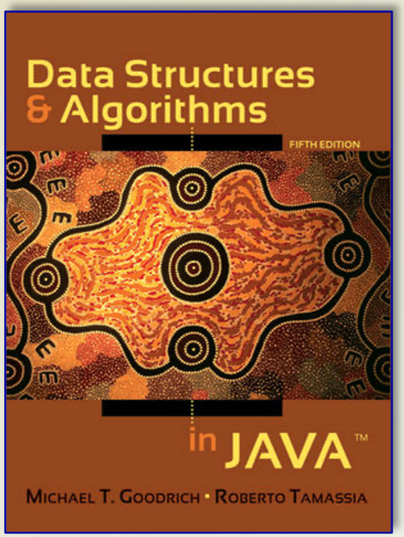
Data Structure & Algorithms in JAVA

5th edition

Michael T. Goodrich
Roberto Tamassia



Chapter 8: Heaps and Priority Queues

CPSC 3200

Algorithm Analysis and Advanced Data Structure

Chapter Topics

- The Priority Queue Abstract Data Type.
- Heaps.
- Adaptable Priority Queue.

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, x)** inserts an entry with key k and value x.
  - **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.

Operation	***Output***	***Priority Queue***
insert$(5, A)$	$e_1[=(5,A)]$	$\{(5,A)\}$
insert$(9, C)$	$e_2[=(9,C)]$	$\{(5,A),(9,C)\}$
insert$(3, B)$	$e_3[=(3,B)]$	$\{(3,B),(5,A),(9,C)\}$
insert$(7, D)$	$e_4[=(7,D)]$	$\{(3,B),(5,A),(7,D),(9,C)\}$
min()	e_3	$\{(3,B),(5,A),(7,D),(9,C)\}$
removeMin()	e_3	$\{(5,A),(7,D),(9,C)\}$
size()	3	$\{(5,A),(7,D),(9,C)\}$
removeMin()	e_1	$\{(7,D),(9,C)\}$
removeMin()	e_4	$\{(9,C)\}$
removeMin()	e_2	$\{\}$

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 $\leq$
  - **Reflexive property:** $x \leq x$
  - **Antisymmetric property:** $x \leq y \wedge y \leq x \Rightarrow x = y$
  - **Transitive property:** $x \leq y \wedge y \leq z \Rightarrow x \leq z$

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>
{
    public  K getKey();
    public  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(x, y):** returns an integer $\boldsymbol{i}$ such that
  - $\boldsymbol{i} < 0$ if $\boldsymbol{a} < \boldsymbol{b}$,
  - $\boldsymbol{i} = 0$ if $\boldsymbol{a} = \boldsymbol{b}$
  - $\boldsymbol{i} > 0$ if $\boldsymbol{a} > \boldsymbol{b}$
  - An error occurs if $\boldsymbol{a}$ and $\boldsymbol{b}$ cannot be compared.

Priority Queue Sorting

- We can use a priority queue to sort a set 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 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.removeFirst*** ()
    P.insert (***e***, ∅)
  while !***P.isEmpty***()
    e ← ***P.removeMin***().***getKey***()
    S.addLast(***e***)

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

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

Selection-Sort Example

	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)	( )

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 $\boldsymbol{n}$ **insert** operations takes time proportional to

     $$1 + 2 + \ldots + \boldsymbol{n}$$

  2. Removing the elements in sorted order from the priority queue with a series of $\boldsymbol{n}$ **removeMin** operations takes $\boldsymbol{O}(\boldsymbol{n})$ time.

- Insertion-sort runs in $\boldsymbol{O}(\boldsymbol{n}^2)$ time

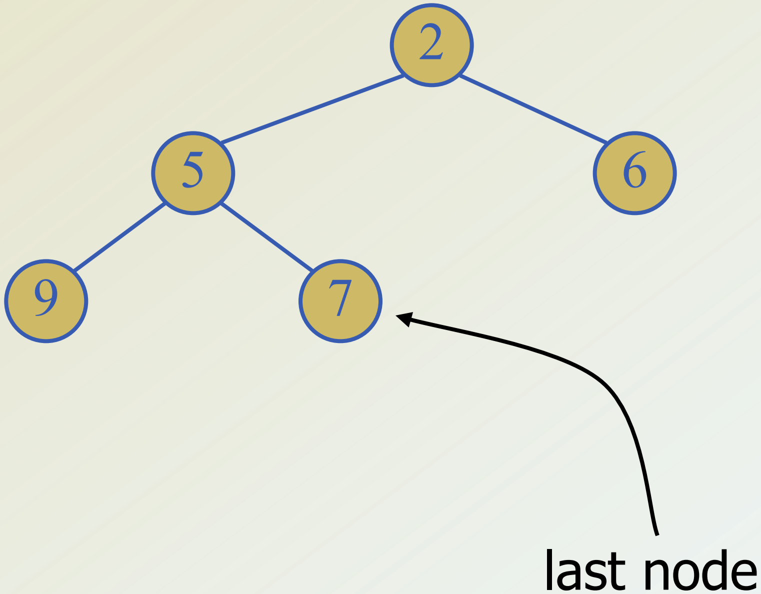
Insertion-Sort Example

	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)	( )

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, $\boldsymbol{key}(\boldsymbol{v}) \geq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$
  - **Complete Binary Tree:** let ***h*** be the height of the heap
    - for ***i*** = 0, … , ***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.

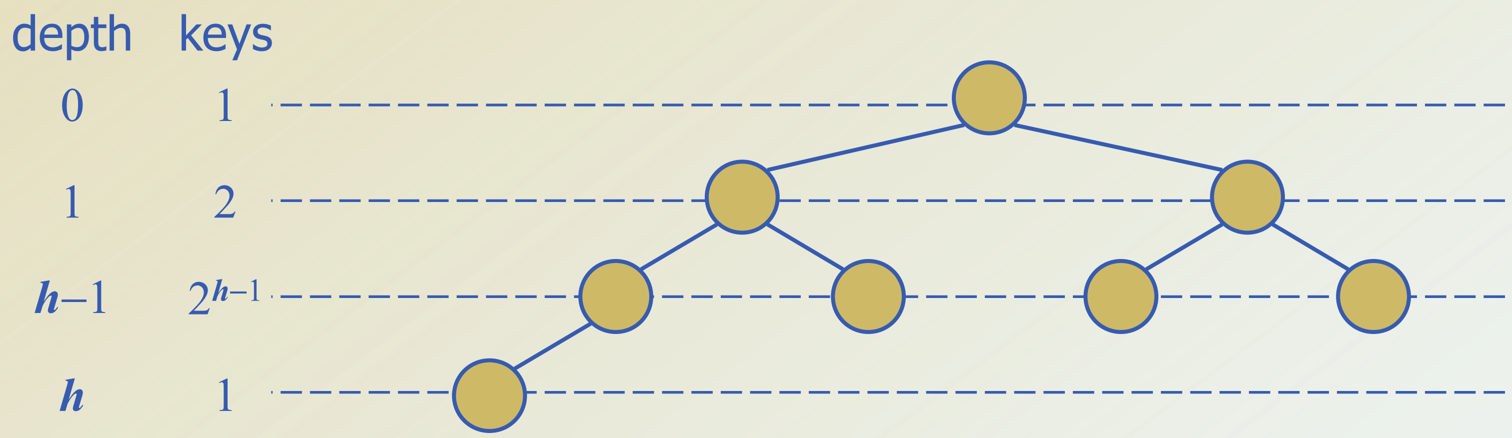
- The last node of a heap is the rightmost node of maximum depth.



Height of a Heap

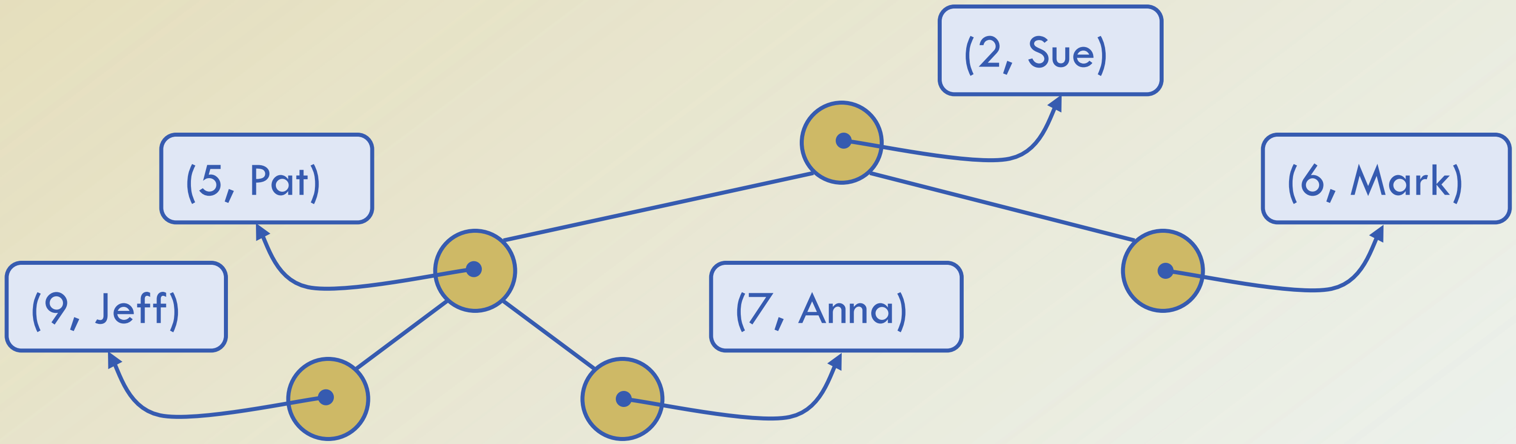
- Theorem: A heap storing n keys has height $O(\log n)$

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



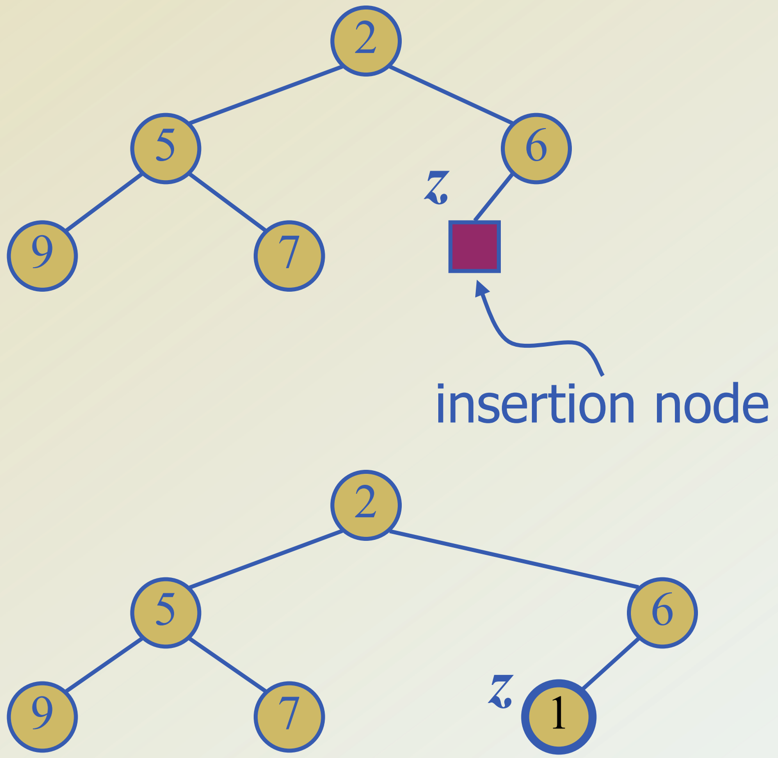
Heaps and Priority Queues

- 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.



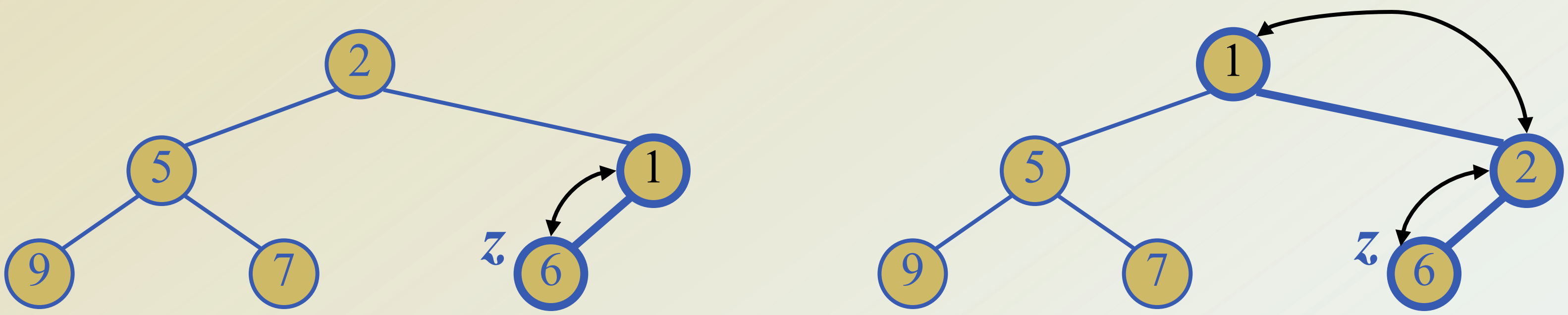
Insertion into a Heap

- Method **insertItem** 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).



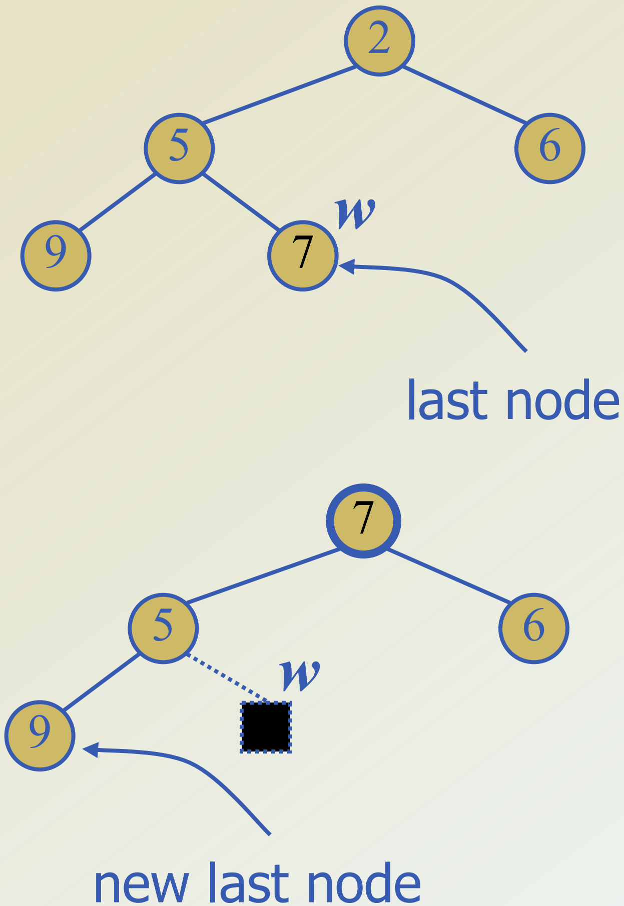
Upheap

- 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.
- **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.



Removal from a Heap (§ 7.3.3)

- 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

Analysis

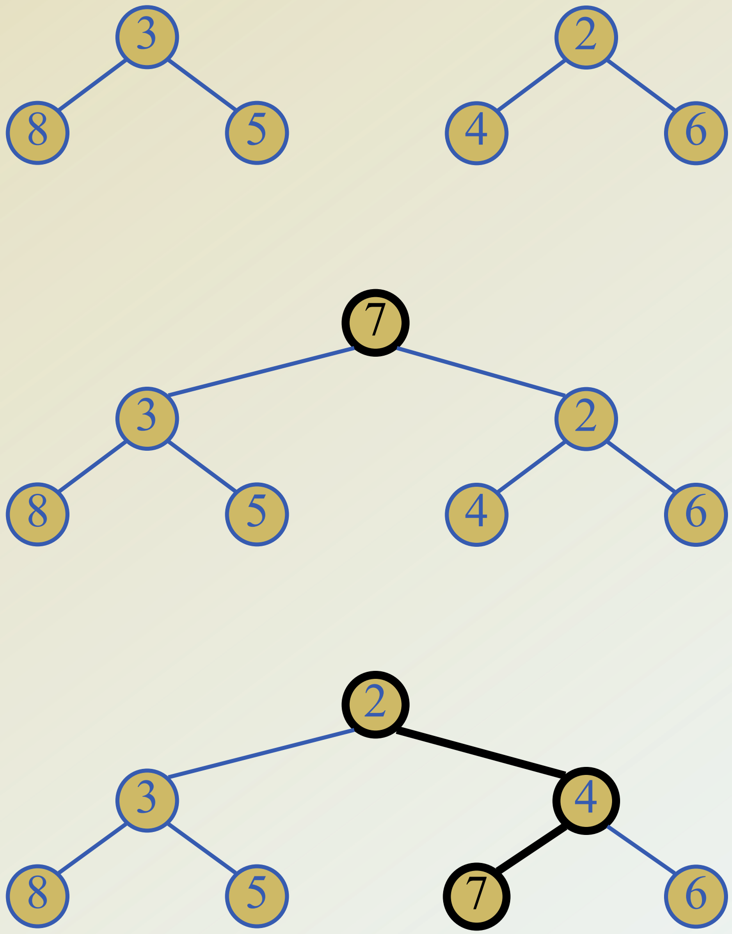
Operation	Time
size, isEmpty	$O(1)$
min,	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

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**.

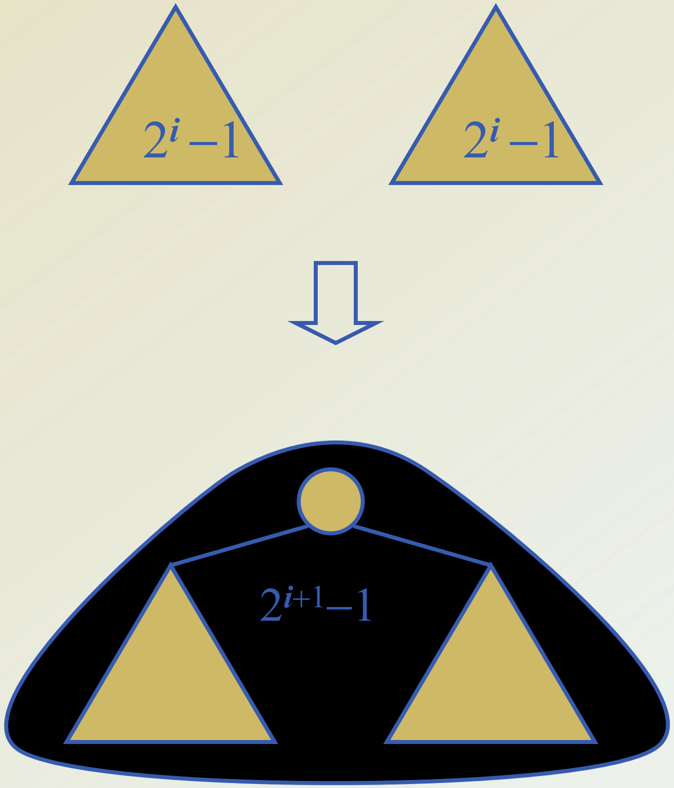
Merging Two Heaps

- We are given two 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

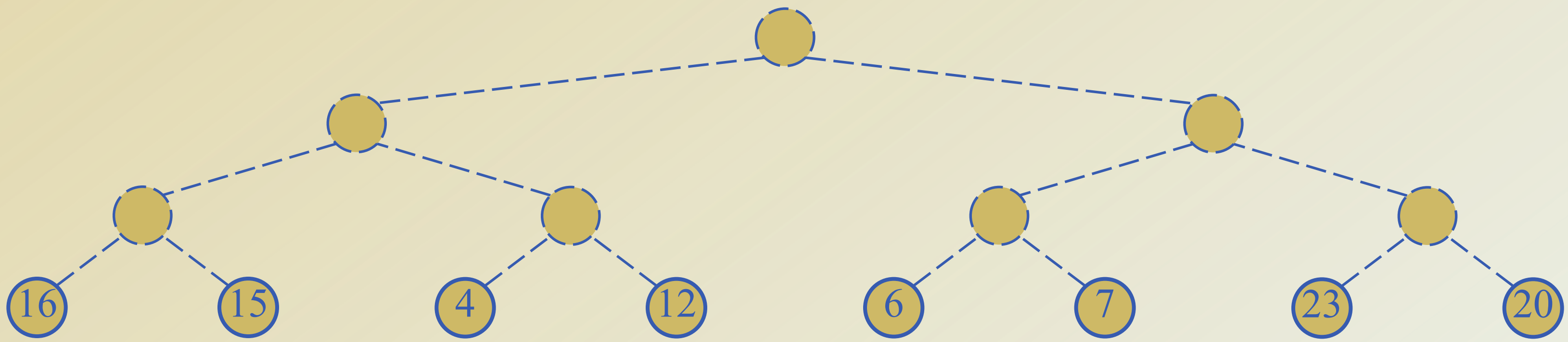


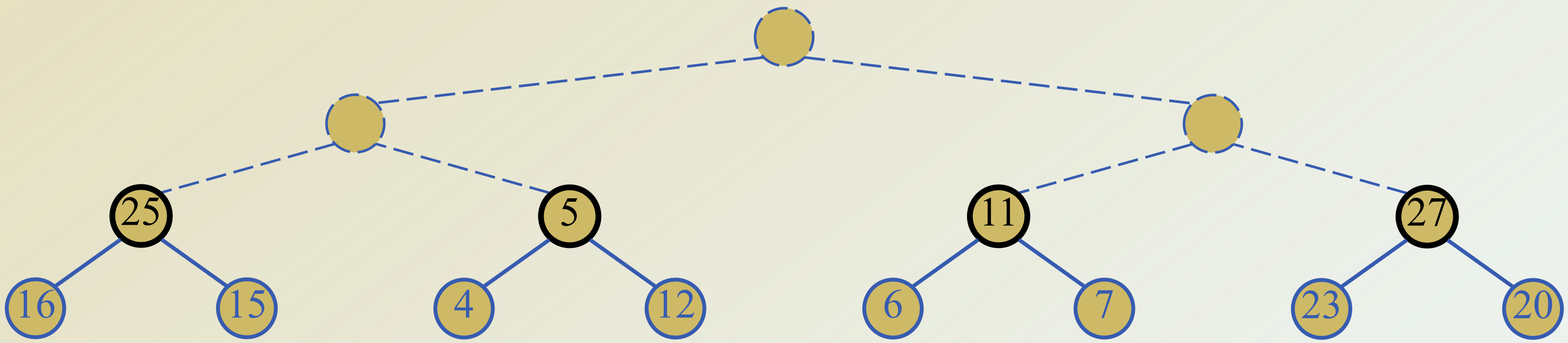
Bottom-up Heap Construction

- 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

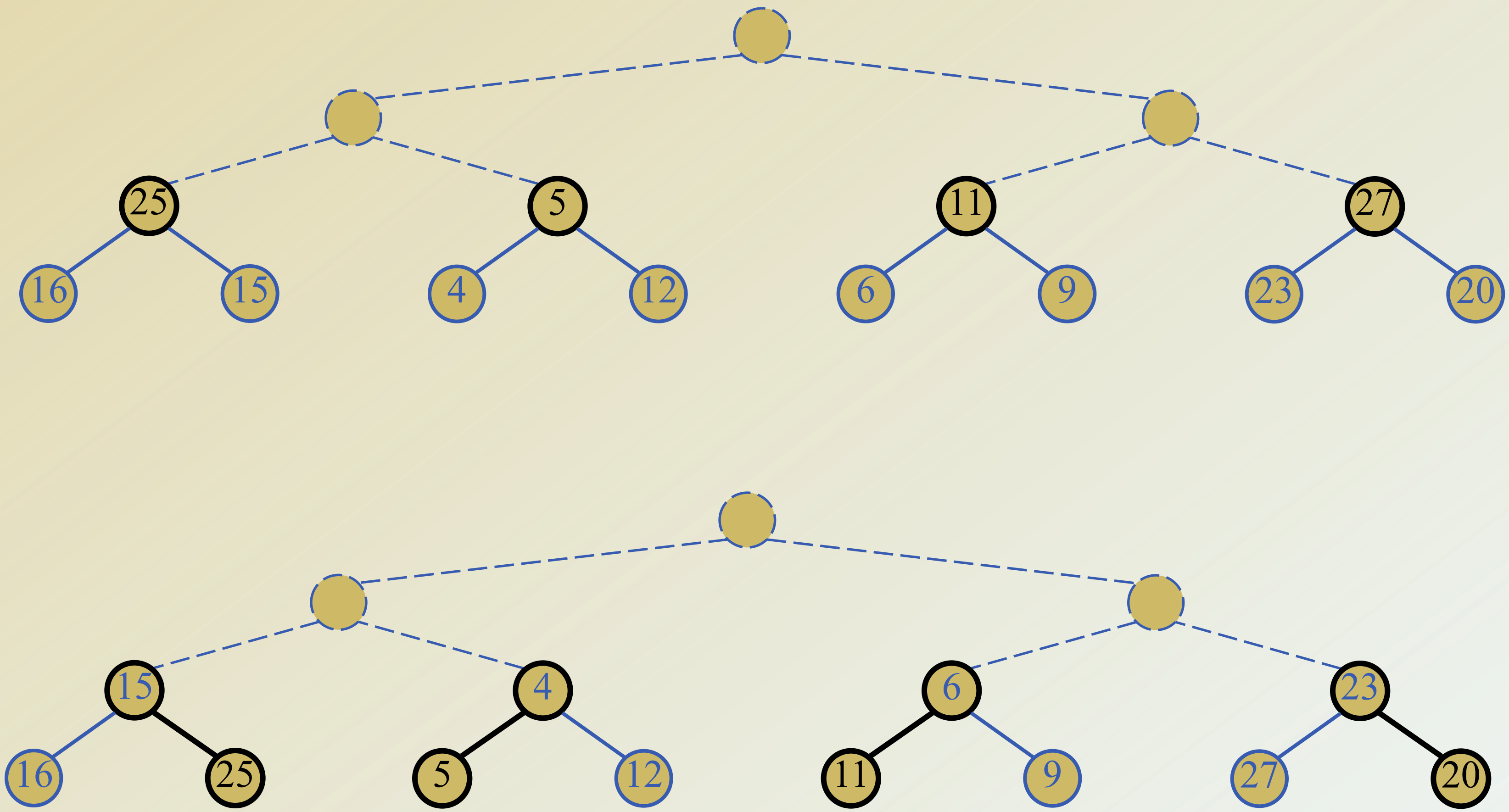


Example

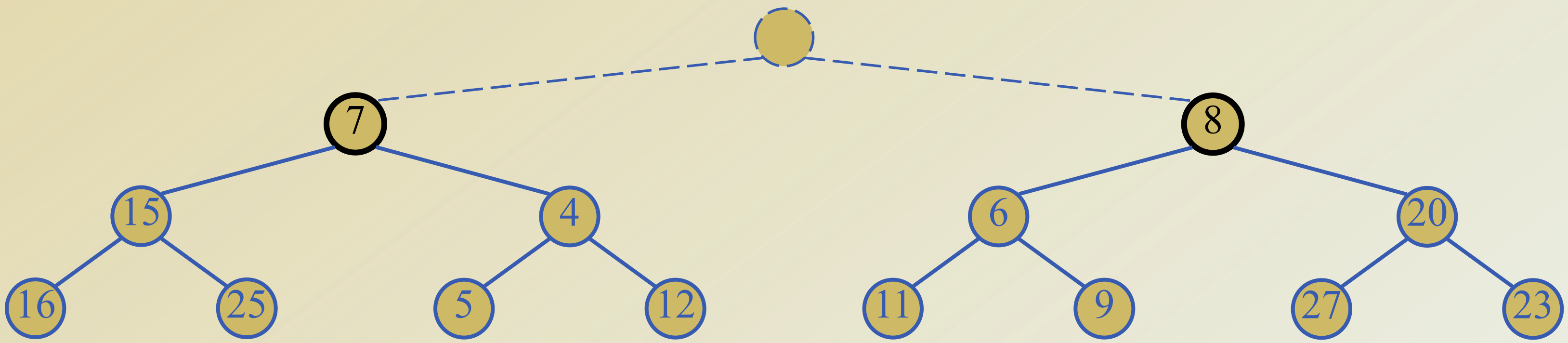


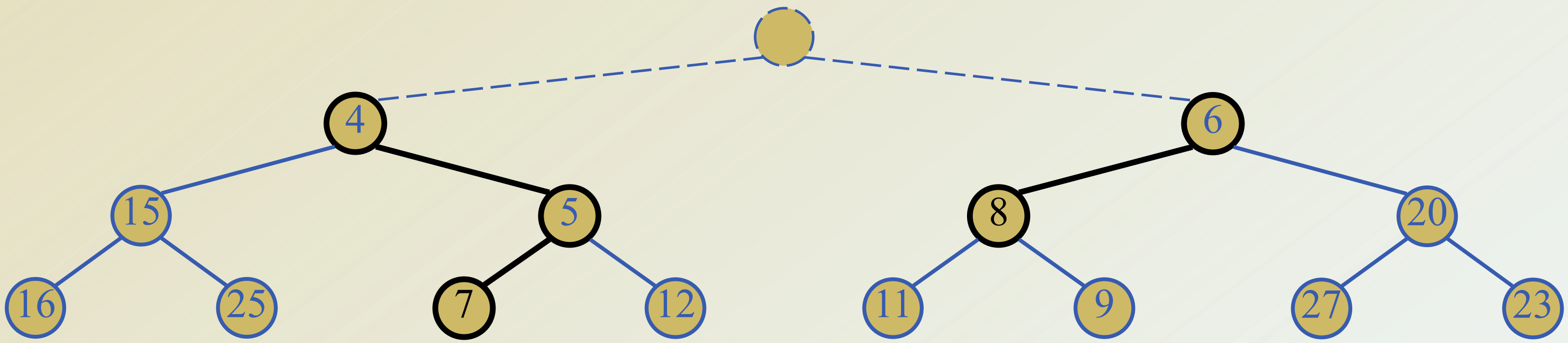


Example (contd.)

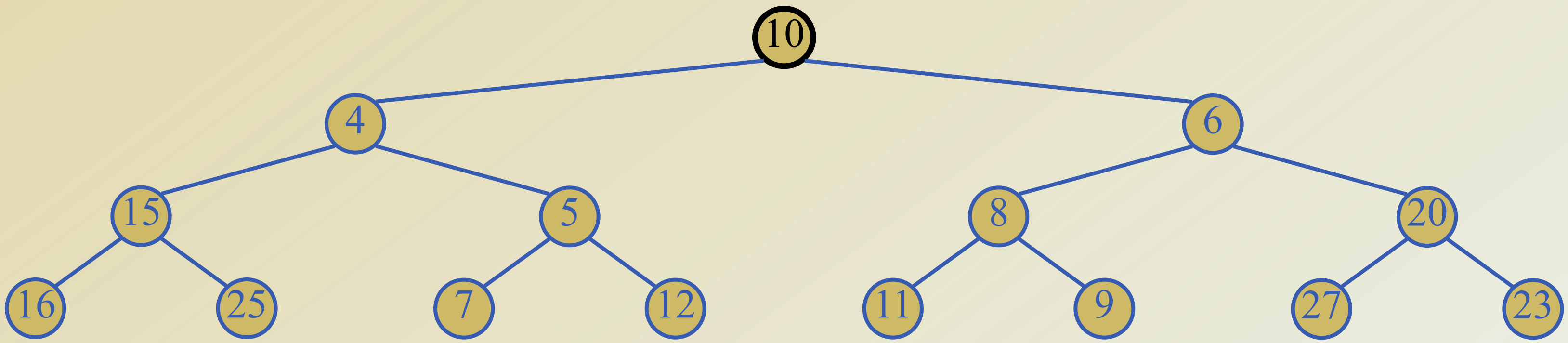


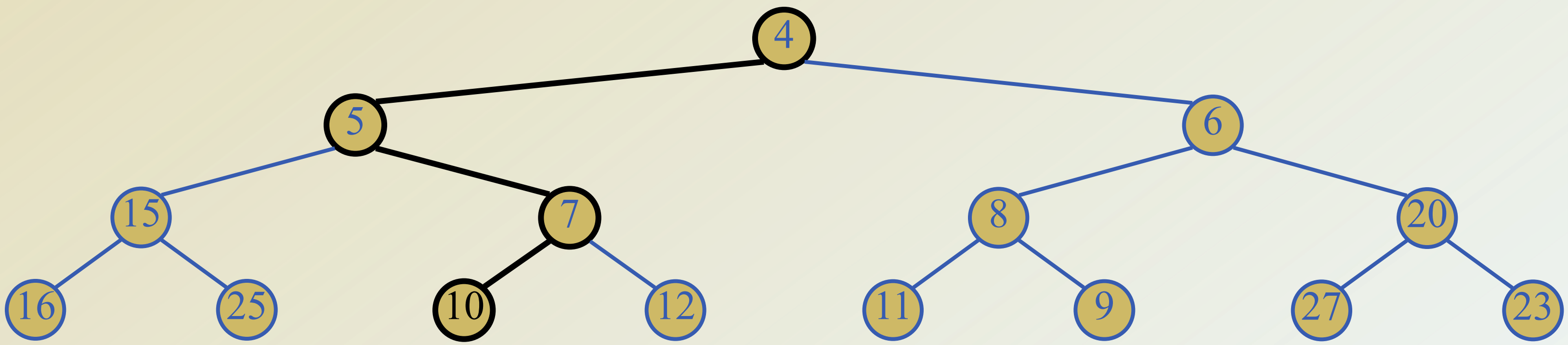
Example (contd.)





Example (end)





Recursive Bottom-Up Heap Construction

Algorithm BottomUpHeap(S):

Input: A list L storing n = 2h+1−1 entries

Output: A heap T storing the entries in L.

if S.isEmpty() **then**

  return an empty heap

e ← L.remove(L.first())

Split L into two lists, L1 and L2, each of size (n−1)/2

T1 ← BottomUpHeap(L1)

T2 ← BottomUpHeap(L2)

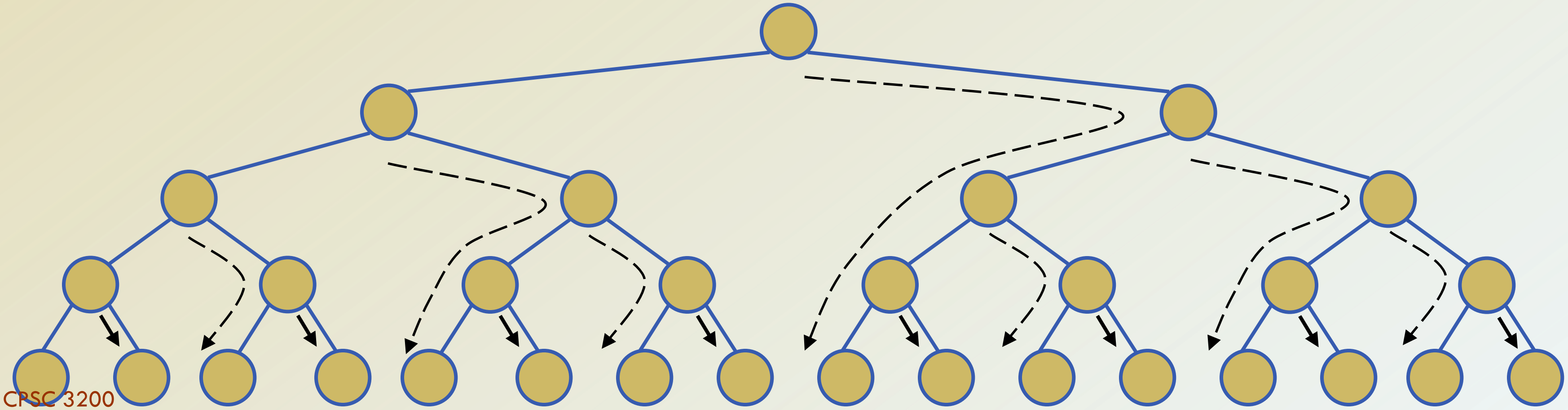
Create binary tree T with root r storing e, left subtree T1, and right subtree T2

Perform a down-heap bubbling from the root r of T, if necessary

return T

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 $\boldsymbol{O}(\boldsymbol{n})$
- Thus, bottom-up heap construction runs in $\boldsymbol{O}(\boldsymbol{n})$ time
- Bottom-up heap construction is faster than $\boldsymbol{n}$ successive insertions and speeds up the first phase of heap-sort.



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( )**

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,x):** Replace with ***x*** 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)

Analysis

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	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$
replaceValue	$O(1)$	$O(1)$	$O(1)$

Running times of the methods of an adaptable priority queue of size n, realized by means of an unsorted list, sorted list, and heap, respectively.
The space requirement is **O(n)**

End of Chapter 8