CS206 Data Structures

Priority Queues II

Sung-eui Yoon (윤성의)
Department of Computer Science
KAIST

http://sglab.kaist.ac.kr/~sungeui

Class Objectives (Ch. 9)

- Understand heap sort and bottom-up construction
- Know location-aware entries and their application to adaptable prior queues

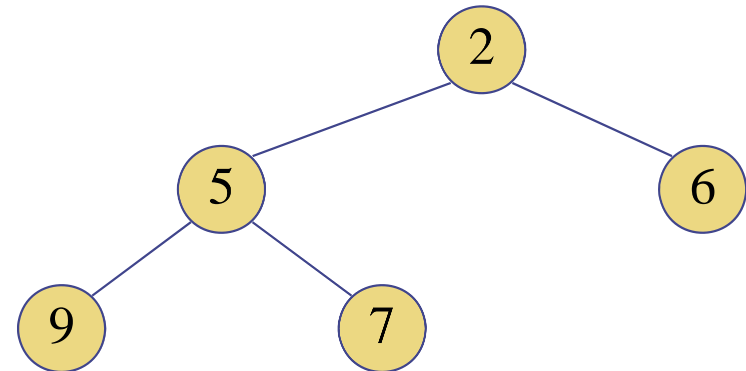
KAIST
Korea Advanced Institute of Science and Technology
한국과학기술원

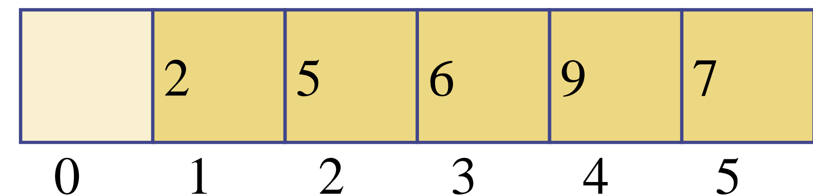
Heap-Sort

- 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

Vector-based Heap Implementation

- We can represent a heap with n keys by means of a vector of length n + 1
- For the node at rank i
  - the left child is at rank 2i
  - the right child is at rank 2i + 1
  - Links between nodes are not explicitly stored
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting at rank n + 1
- Operation removeMin corresponds to removing at rank n
- Yields in-place heap-sort

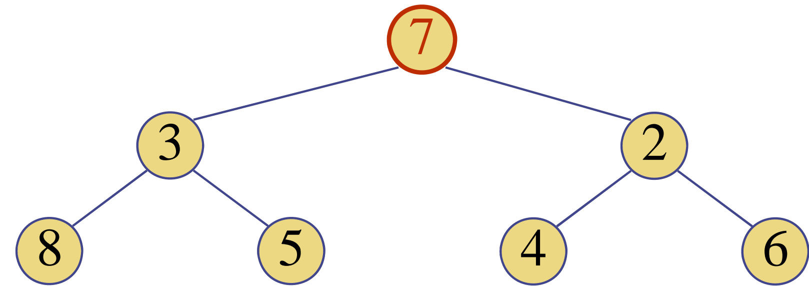


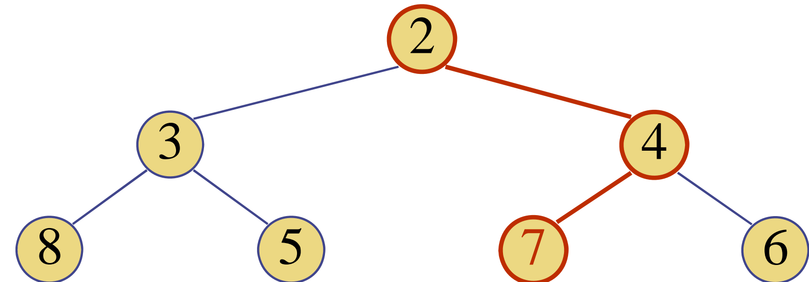


KAIST
Korea Advanced Institute of Science and Technology
한국과학기술원

Merging Two Heaps

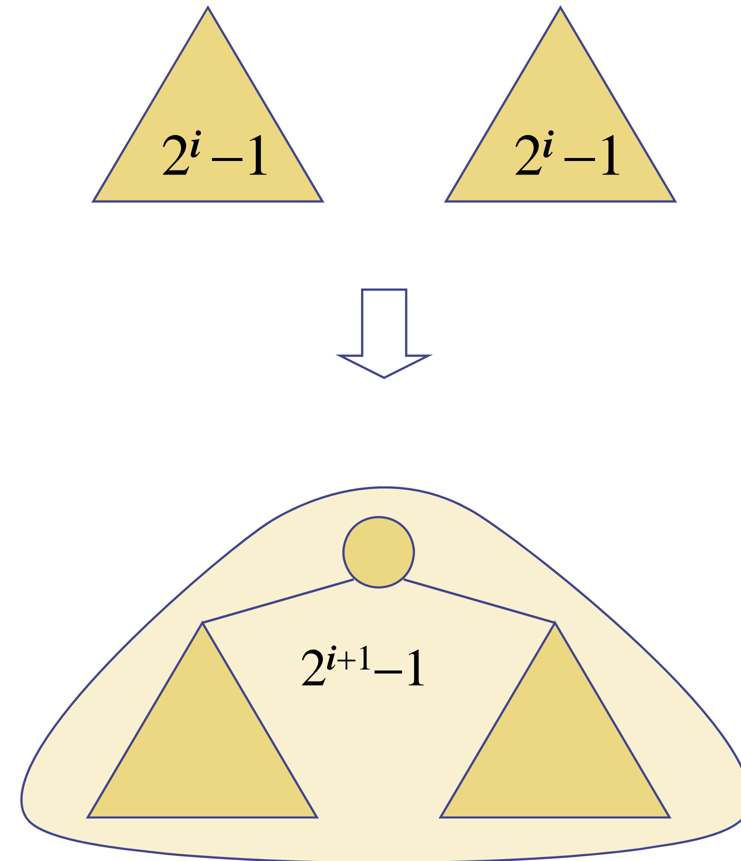
- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property





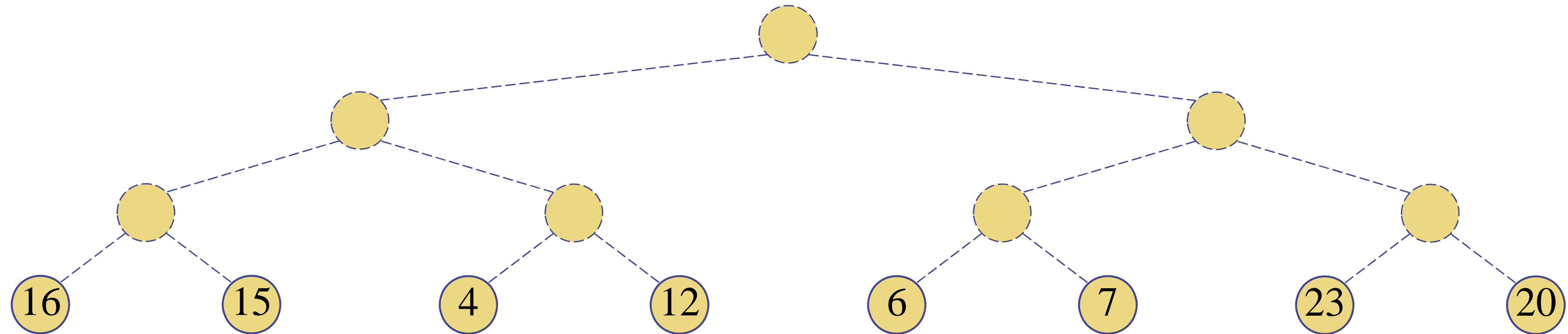
Bottom-up Heap Construction

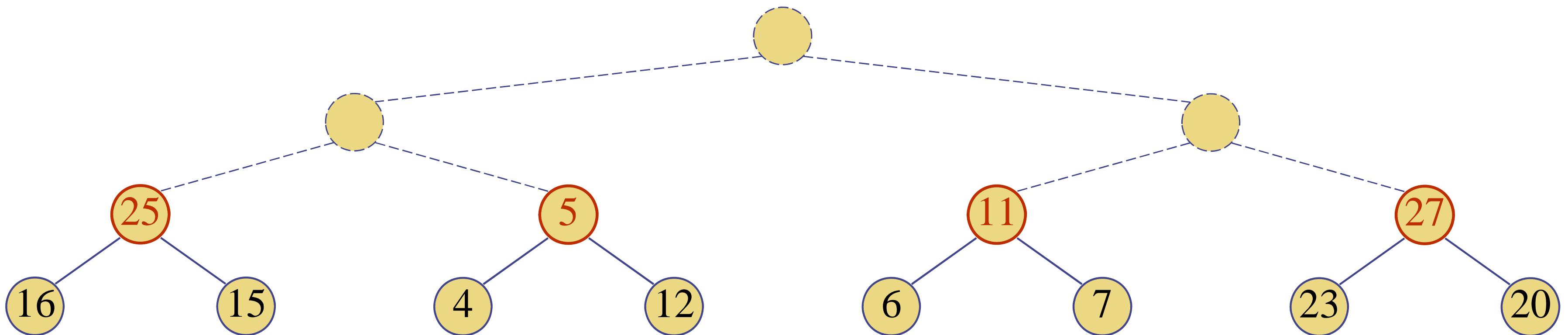
- We can construct a heap storing n given keys in using a bottom-up construction with log n phases
- In phase i, pairs of heaps with 2^i -1 keys are merged into heaps with 2^{i+1}-1 keys



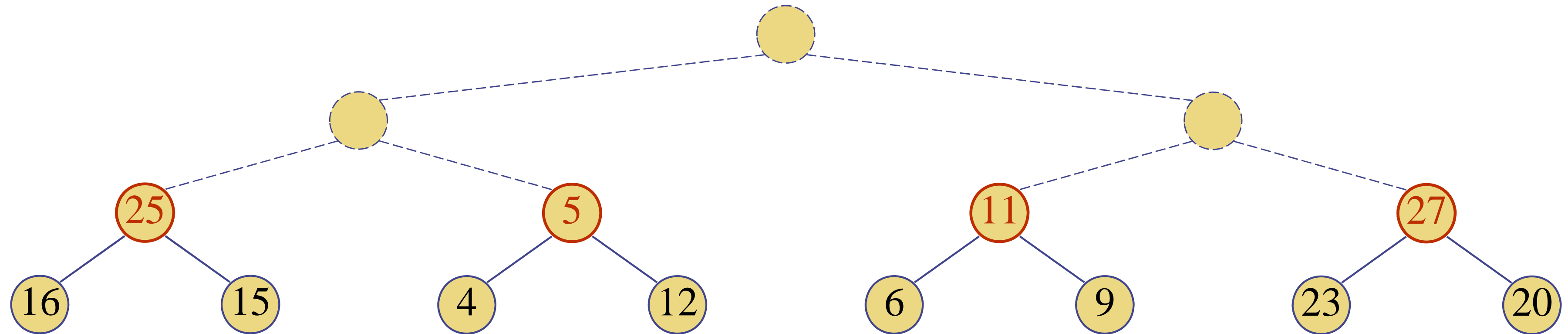
KAIST
Korea Advanced Institute of Science and Technology
한국과학기술원

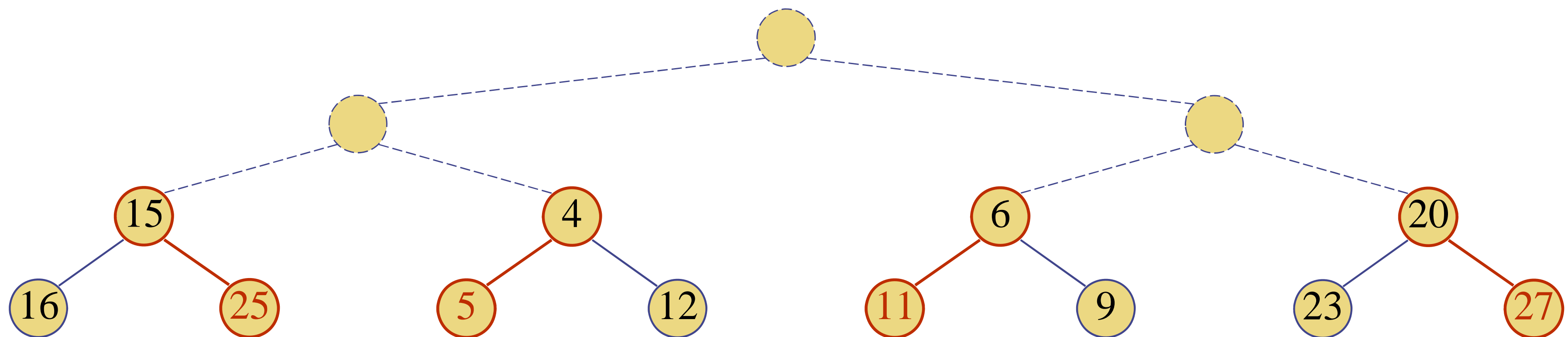
Example





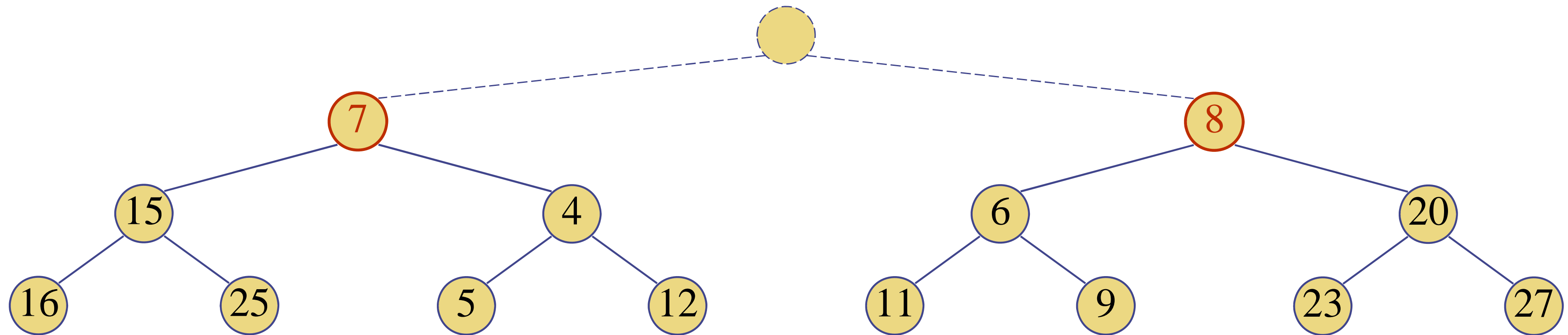
Example (contd.)

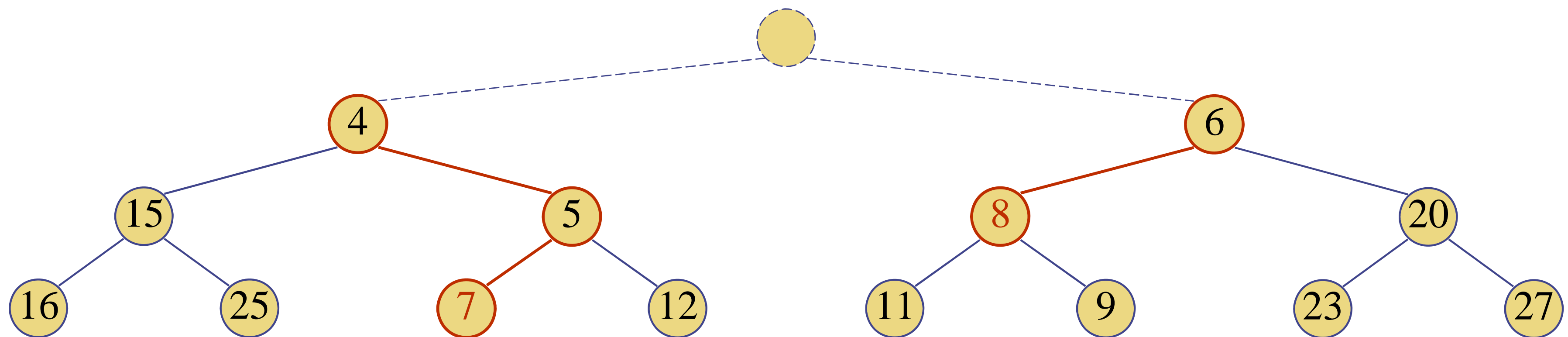




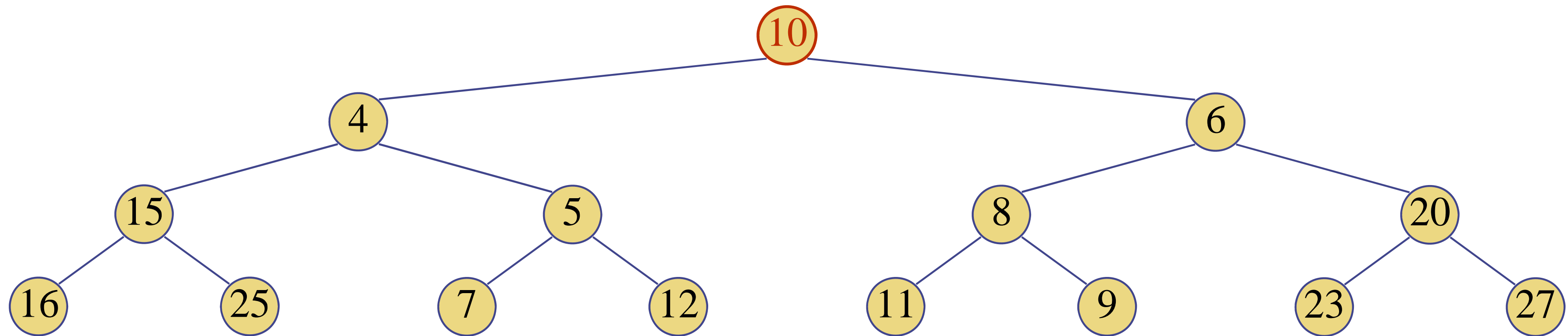
KAIST
Korea Advanced Institute of Science and Technology
한국과학기술원

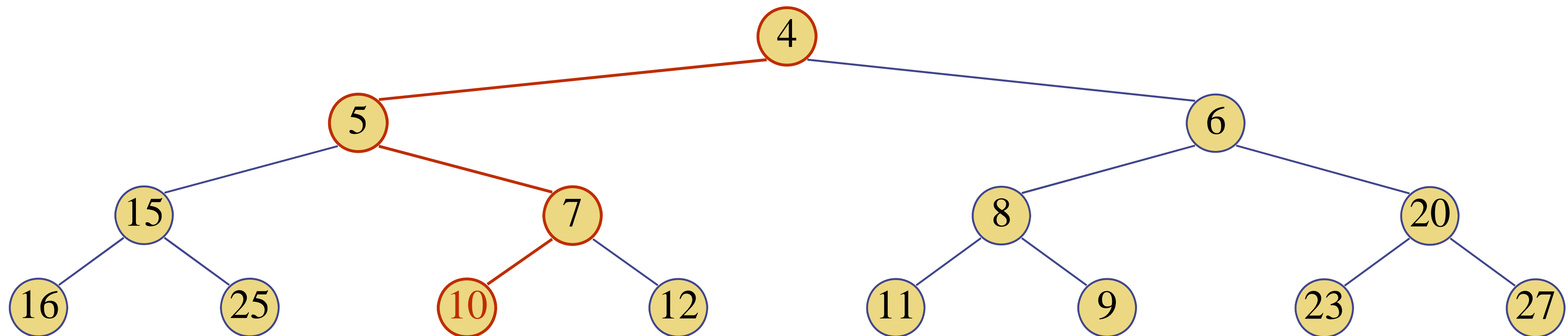
Example (contd.)





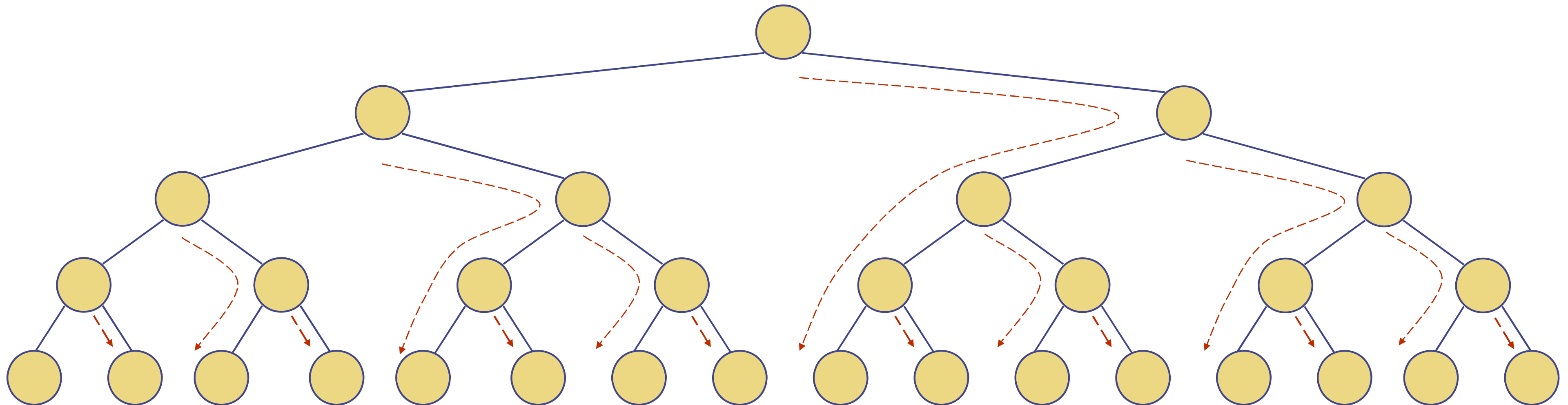
Example (end)





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



Adaptable Priority Queues

KAIST
Korea Advanced Institute of Science and Technology
한국과학기술원

Recall Entry and Priority Queue ADTs

- An entry stores a (key, value) pair within a data structure
- Methods of the entry ADT:
  - key(): returns the key associated with this entry
  - value(): 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()

KAIST
Korea Advanced Institute of Science and Technology
한국과학기술원

Motivating Example

- Suppose we have an online trading system where orders to purchase and sell a given 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 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 becompared 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	$(7, D)$

KAIST
Korea Advanced Institute of Science and Technology
한국과학기술원

Locating Entries

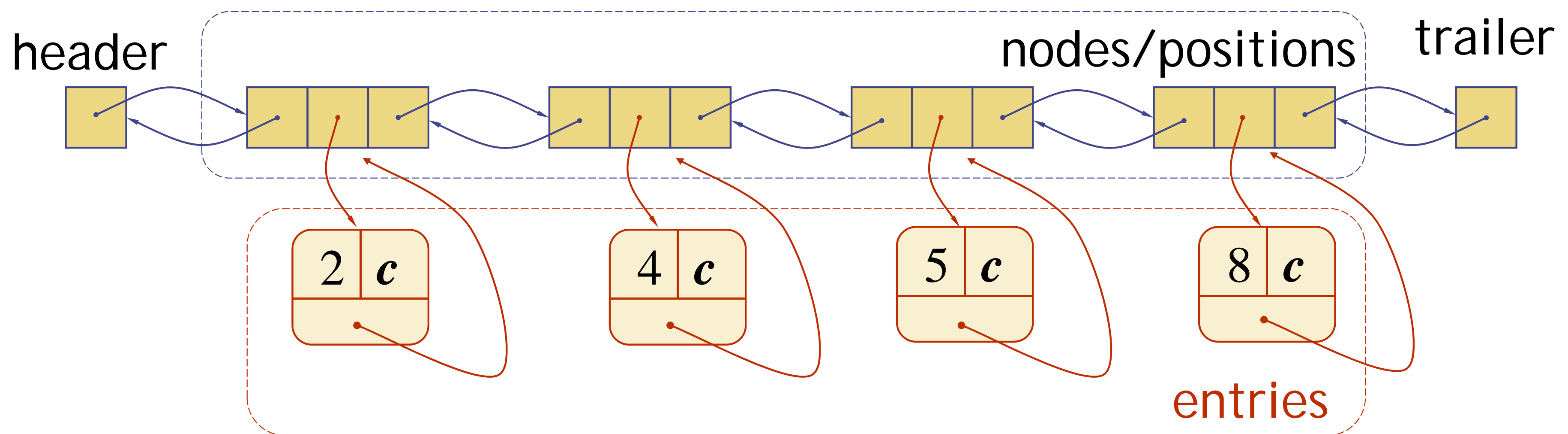
- In order to implement the operations remove(k), replaceKey(e), and replaceValue(k), we need fast ways of locating an entry e in a priority queue.
- We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.

Location-Aware Entries

- A locator-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

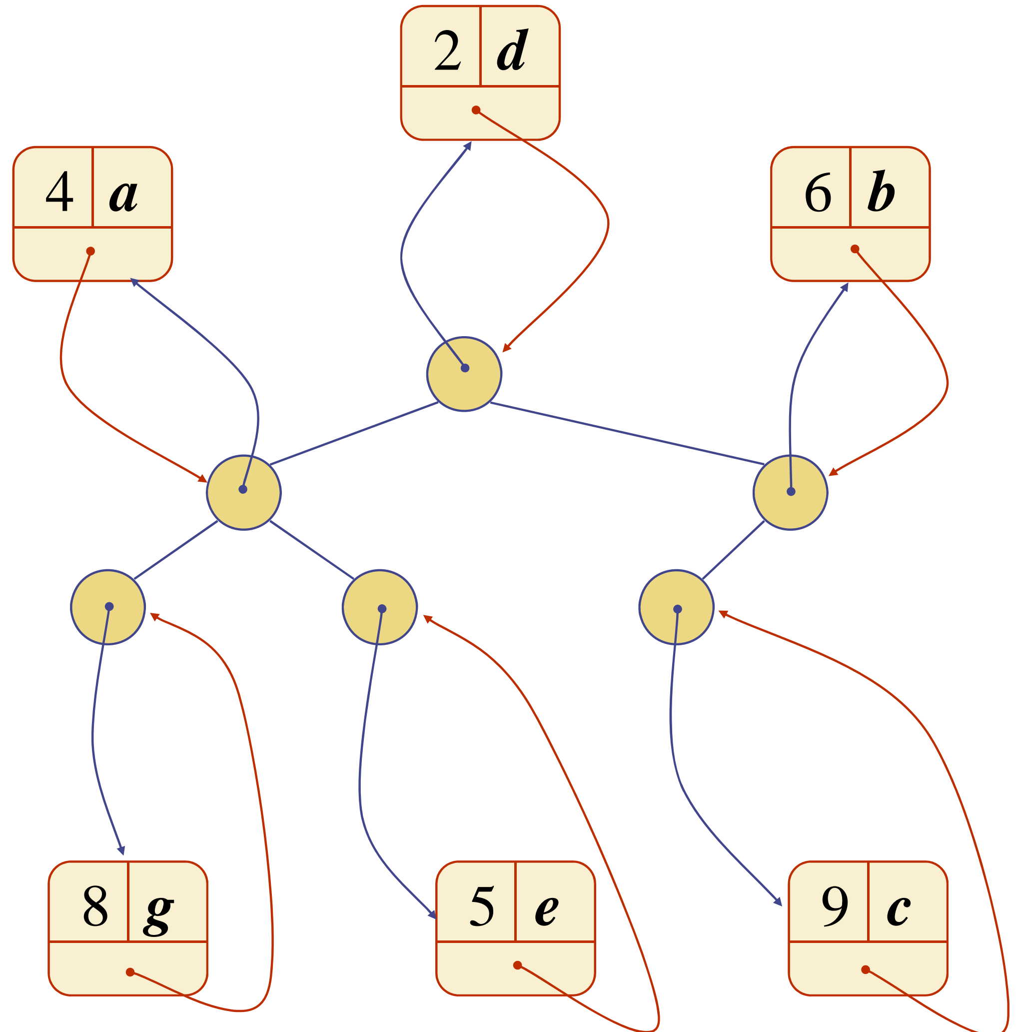
- ☐ A location-aware list entry is an object storing
  - key
  - value
  - position (or rank) of the item in the list
- ☐ In turn, the position (or array cell) stores the entry
- ☐ Back pointers (or ranks) are updated during swaps



KAIST
Korea Advanced Institute of Science and Technology
한국과학기술원

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

- Using location-aware entries we can achieve the following running times (times better than those achievable without 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	**$O(1)$**	**$O(1)$**	**$O(\log n)$**
replaceKey	**$O(1)$**	$O(n)$	**$O(\log n)$**
replaceValue	**$O(1)$**	**$O(1)$**	**$O(1)$**

Class Objectives (Ch. 9) were:

- Understand heap sort and bottom-up construction
- Know location-aware entries and their application to adaptable prior queues

PA 4

☐ Implement the priority queue using heap

Next Time

- Maps and hashes

- Questions:
  - Come up with one question on what we have discussed in the class and submit at the end of the class
  - 1 for typical questions and 2 for questions with thoughts or that surprised me
  - Write questions at least 4 times; you can type at KLMS

- HW:
  - Go over the next lecture slides before the class
  - Just 10 min ~ 20 min should be okay