FALL 2018 CISC 311

OBJECT & STRUCTURE & ALGORITHM I

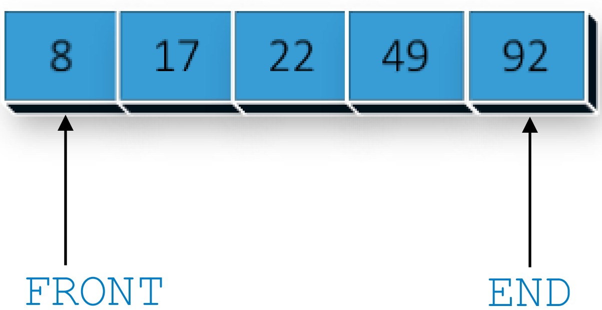
– Chapter 8: Priority Queues and Heaps

Outline

- The priority queue ADT
- The heap ADT
- Implementations
- Heap sort

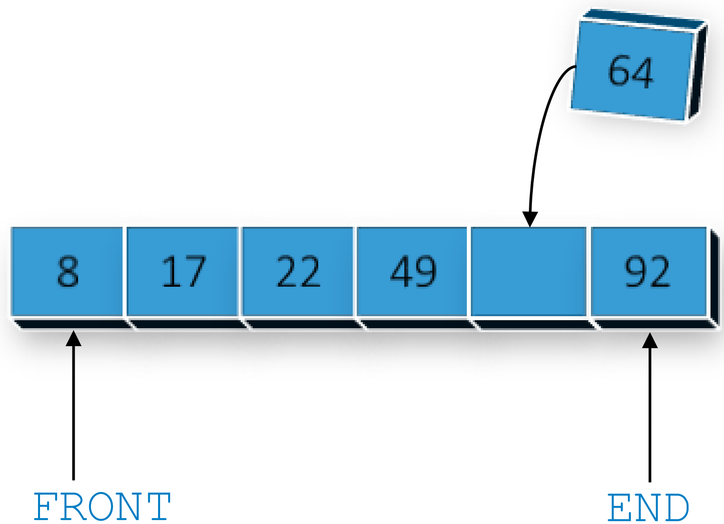
The Priority Queue ADT

- A priority queue is a collection of prioritized elements that allows arbitrary element insertion, and allows the removal of the element that has first priority
  - Insertions are at arbitrary positions
  - Removal are at the front of the priority queue
- The priority queue ADT stores arbitrary objects
  - Example:



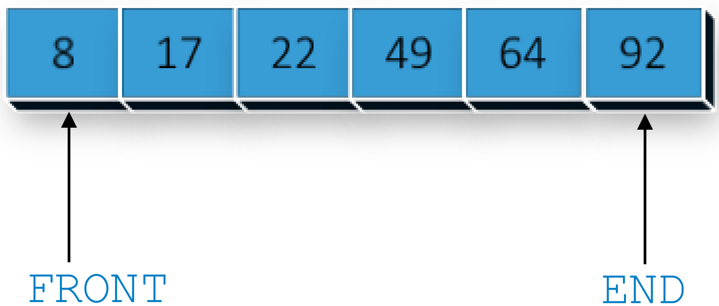
Priority Queue Methods

- `insert(e)`: inserts element e to the back of priority queue
  - Step 1: increment `END` and shift all the elements greater than e forward so it points to the appropriate space



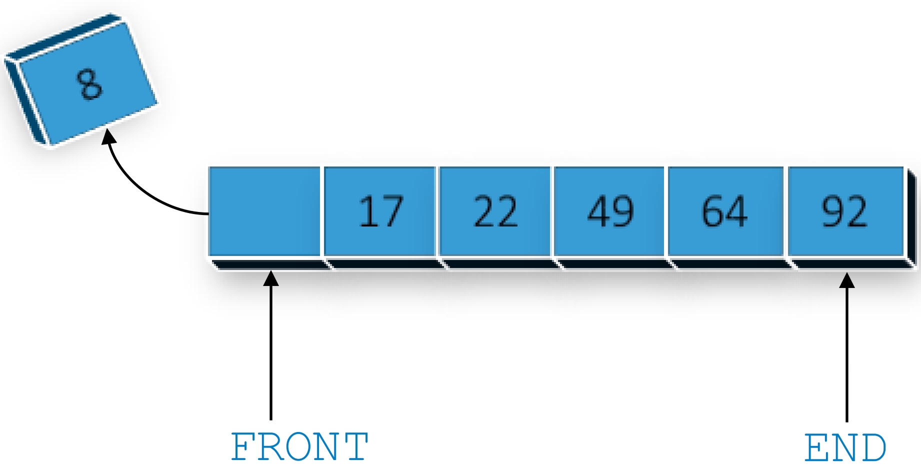
Priority Queue Methods (cont'd.)

- `insert(e)`: inserts element *e* to the back of priority queue
  - Step 2: insert new object



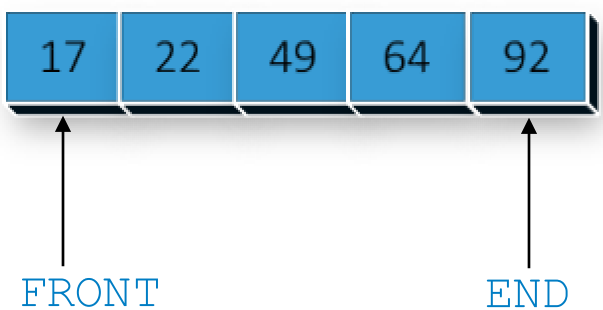
Priority Queue Methods (cont'd.)

- `remove()`: removes and returns the first element (which has first priority) from the priority queue or `null` if the priority queue is empty
  - Step 1: remove the value at front



Priority Queue Methods (cont'd.)

- `remove()`: removes and returns the first element (which has first priority) from the priority queue or `null` if the priority queue is empty
  - Step 2: decrement `FRONT`



Priority Queue Methods (cont'd.)

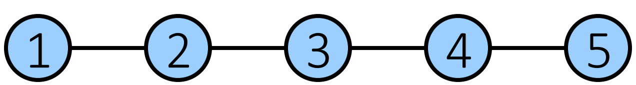
Method	Description
`size()`	Returns the number of elements in the priority queue
`isEmpty()`	Returns a boolean indicating whether the priority queue is empty
`min()`	Returns the first element of the priority queue, without removing it or `null` if the priority queue if empty
`insert(e)`	Adds element *e* to priority queue
`remove()`	Removes and returns the first element from the priority queue or `null` if the priority queue is empty

Priority Queue Implementation

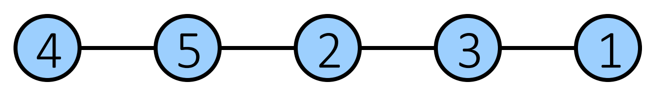
- Array-based implementation (sorted list)
- Doubly linked list-based implementation (unsorted & sorted lists)
- Heap-based implementation

Sequence-based Priority Queue

- Implementation with a sorted list
- Performance:
  - `insert` takes $O(n)$ time since we have to find the place where to insert the item
  - `remove` and `min` take $O(1)$ time, since the smallest key is at the beginning

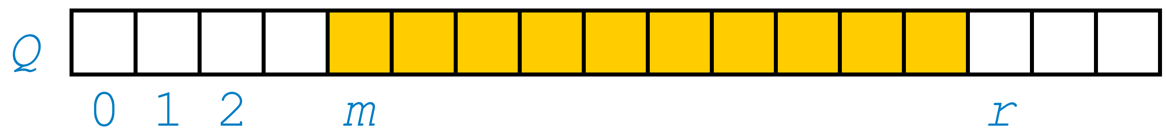


- Implementation with an unsorted list
- Performance:
  - `insert` takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - `remove` and `min` take $O(n)$ time since we have to traverse the entire sequence to find the smallest key



Array-based Implementation

- Use an array of size N
- Two variables keep track of the front and size
  - m : index of the front element
  - sz : number of stored elements

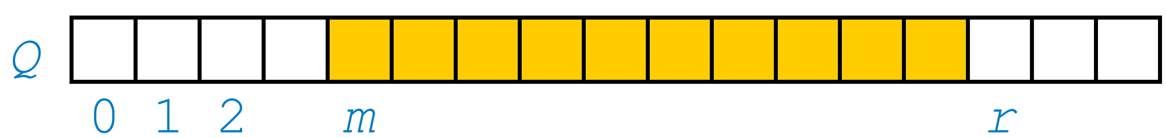


Array-based Implementation (cont'd.)

Algorithm size()
    return sz

Algorithm isEmpty()
    return $sz = 0$

Algorithm min()
    if isEmpty() then
        return null
    return $Q[m]$

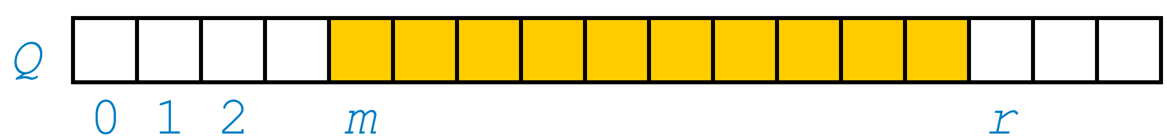


`ArrayPriorityQueue.java`

Array-based Implementation (cont'd.)

- Note that operation `remove` returns `null` if the priority queue is empty

Algorithm remove()
    if isEmpty() then
        return null
    removed $\leftarrow Q[m]$
    $m \leftarrow m + 1$
    $sz \leftarrow sz - 1$
    return *removed*

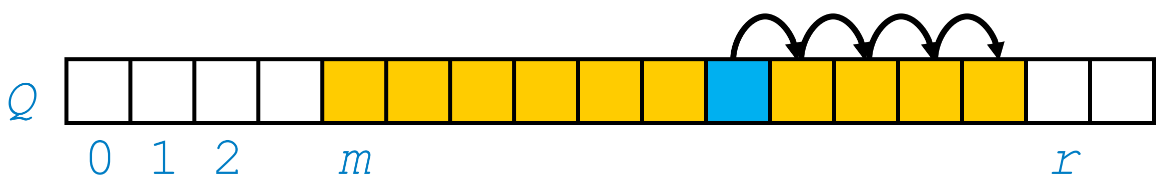


`ArrayPriorityQueue.java`

Array-based Implementation (cont'd.)

- Operation `insert` throws an exception if the array is full

```
Algorithm insert(o)
    if sz = Q.length then
        throw IllegalStateException
    r ← m + sz
    for i ← r – 1 to 0 do
        if o < S[i] then
            S[i+1] ← S[i]
        else
            break
    Q[i+1] ← o
    sz ← sz + 1
```



ArrayPriorityQueue.java

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 $y \leq z \Rightarrow x \leq z$
  - Reflexive property: $k \leq k$

Comparator Interface

- 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
- Implement `java.util.Comparator`
- Primary method of Comparator
  - `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

Comparable Interface

- A comparable encapsulates the action of comparing two objects according to the natural ordering
- Implement `java.lang.Comparable`
- It only contains one method
  - `a.compareTo(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
- E.g., the natural ordering of strings is lexicographic, which is a case-sensitive extension of the alphabetic ordering of Unicode

`StringLengthComparator.java`

Performance Comparison

Method	Unsorted linked list	Sorted linked list
`size()`	$O(1)$	$O(1)$
`isEmpty()`	$O(1)$	$O(1)$
`min()`	$O(n)$	$O(1)$
`insert(e)`	$O(1)$	$O(n)$
`remove()`	$O(n)$	$O(1)$

UnsortedPriorityQueue.java

SortedPriorityQueue.java

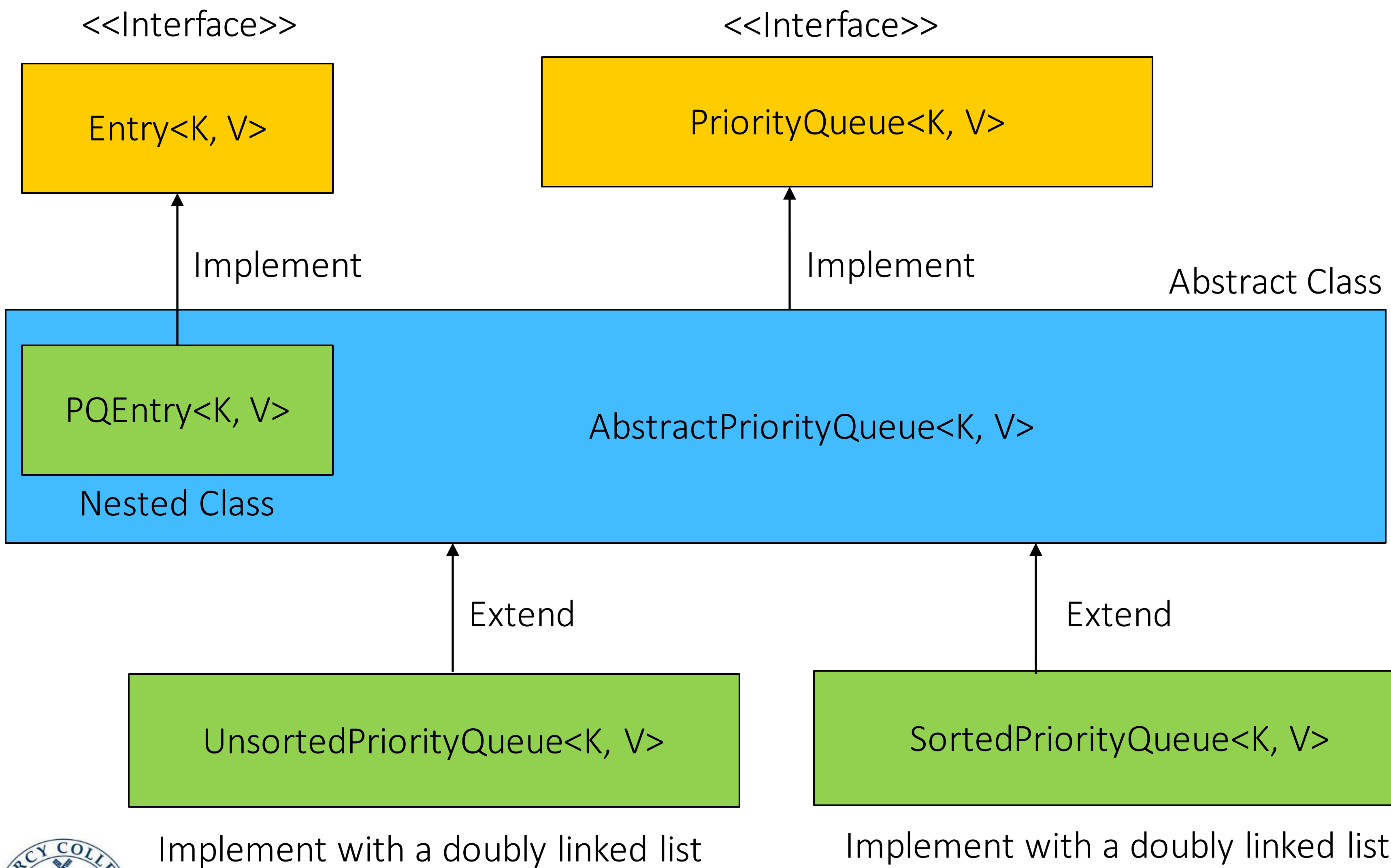
Priority Queue ADT (cont'd.)

- 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
  - `remove()`: 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 priority queue is empty
  - `size()`
  - `isEmpty()`

Priority Queue Methods (cont'd.)

Method	Description
`size()`	Returns the number of elements in the priority queue
`isEmpty()`	Returns a boolean indicating whether the priority queue is empty
`min()`	Returns a priority queue entry (k, v) having the minimal key, without removing it or `null` if the priority queue if empty
`insert(k, v)`	Inserts an entry with key k and value v in the priority queue
`remove()`	Removes and returns an entry (k, v) having minimal key from the priority queue or `null` if the priority queue is empty

Implementation



Entry ADT

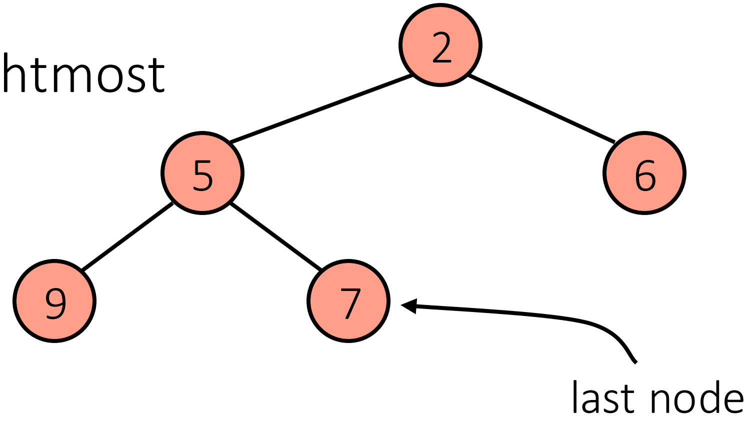
- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - `getKey()`: returns the key for this entry
  - `getValue()`: returns the value associated with this entry

`Entry.java`

`PQEntry.java`

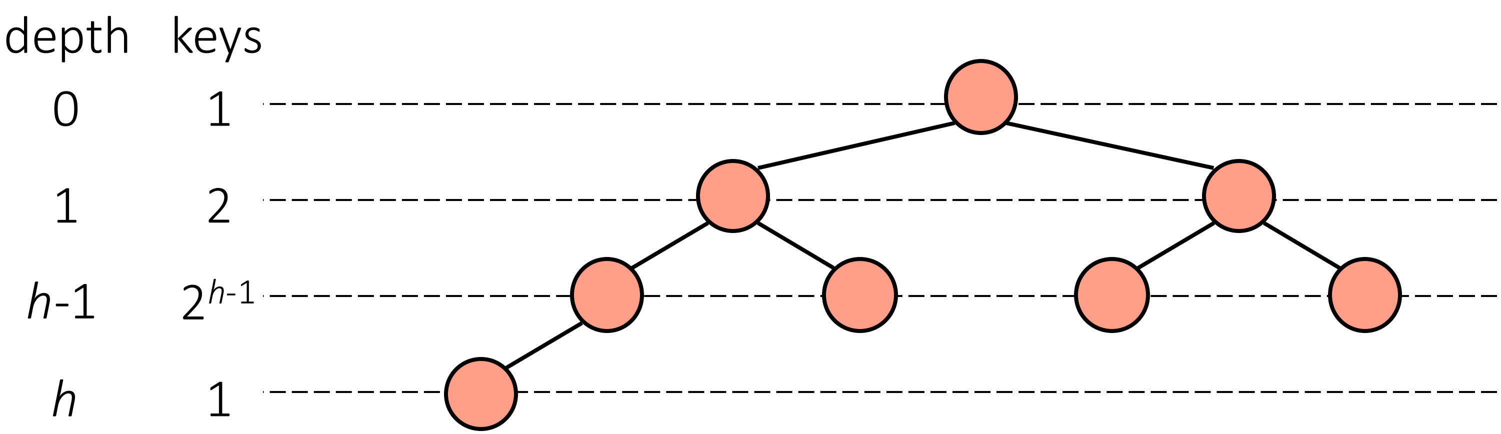
The Heap ADT

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



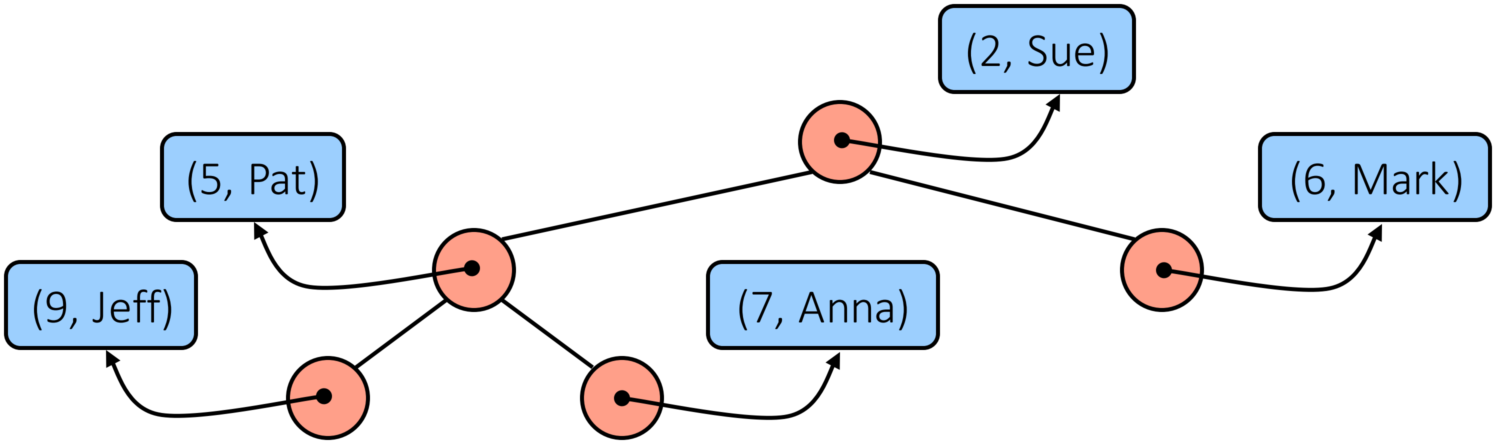
Height of a Heap

- 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., $h \leq \log n$



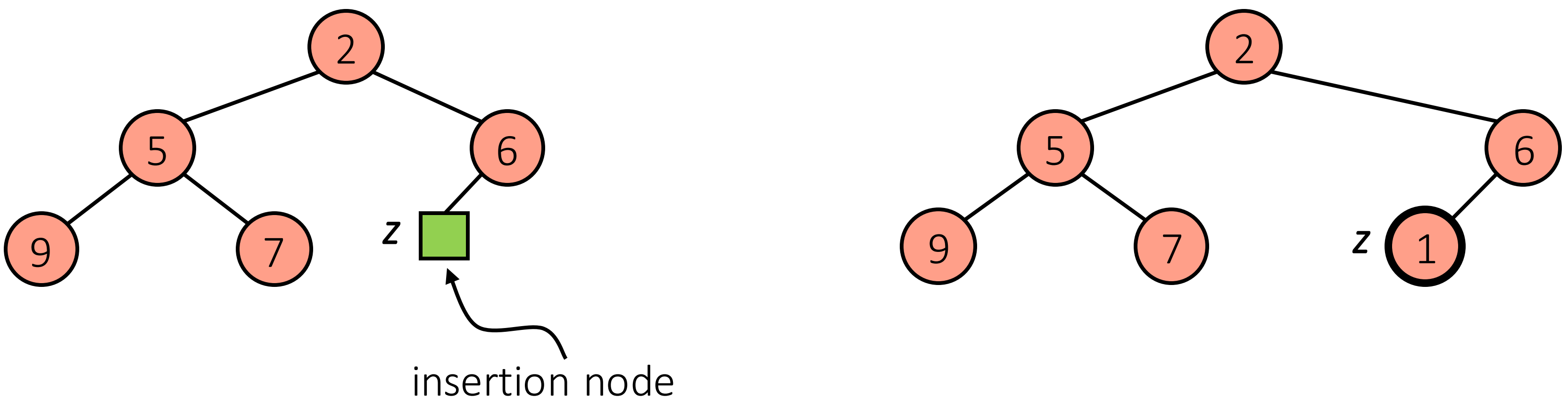
Heaps and Priority Queues

- 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



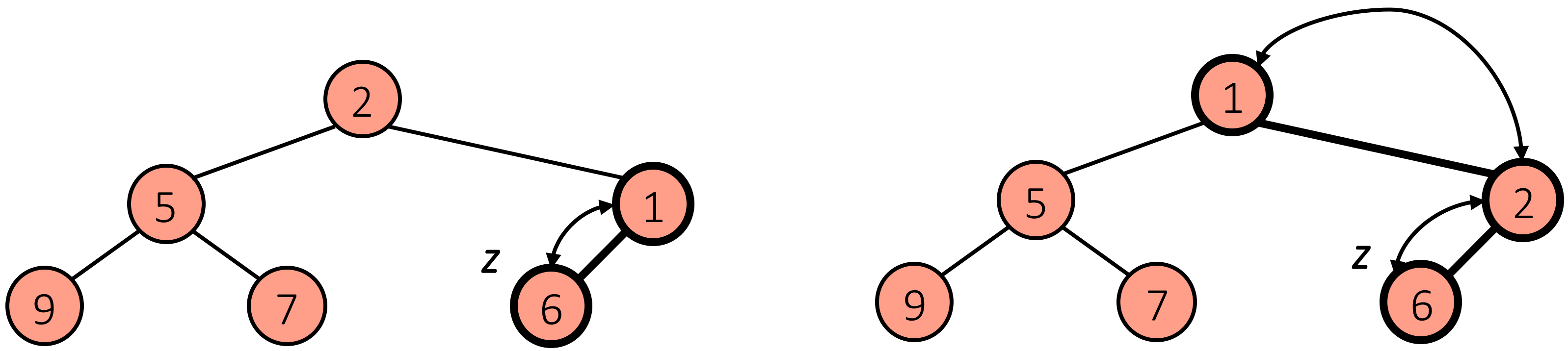
Insertion into a Heap

- Method `insert` 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



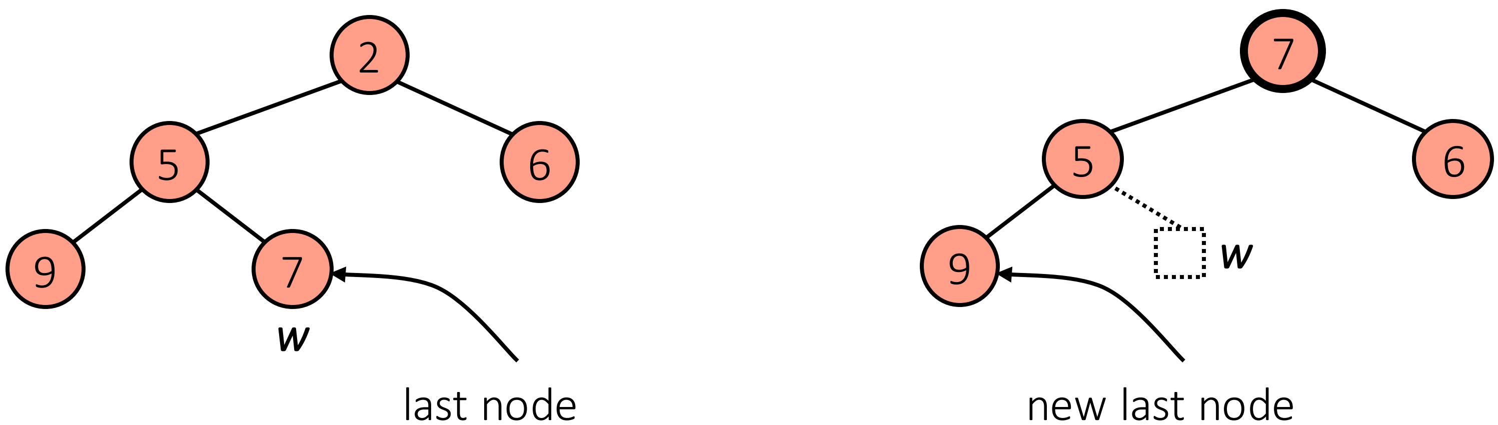
Upheap

- 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



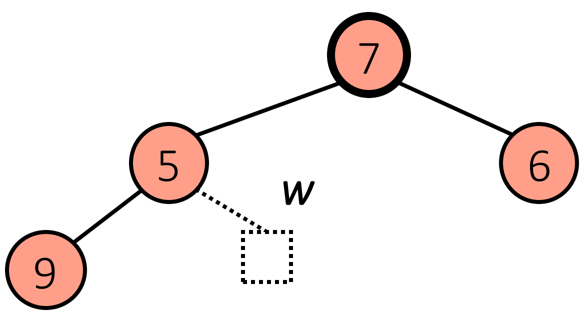
Removal from a Heap

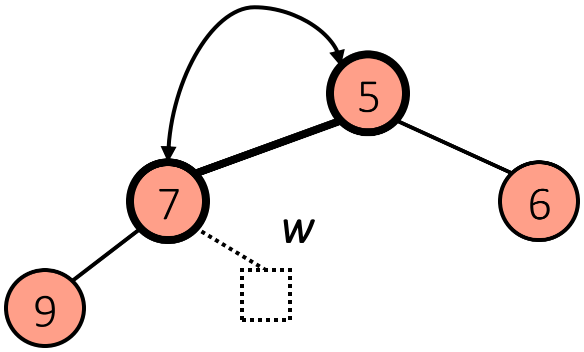
- Method `remove` of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property



Downheap

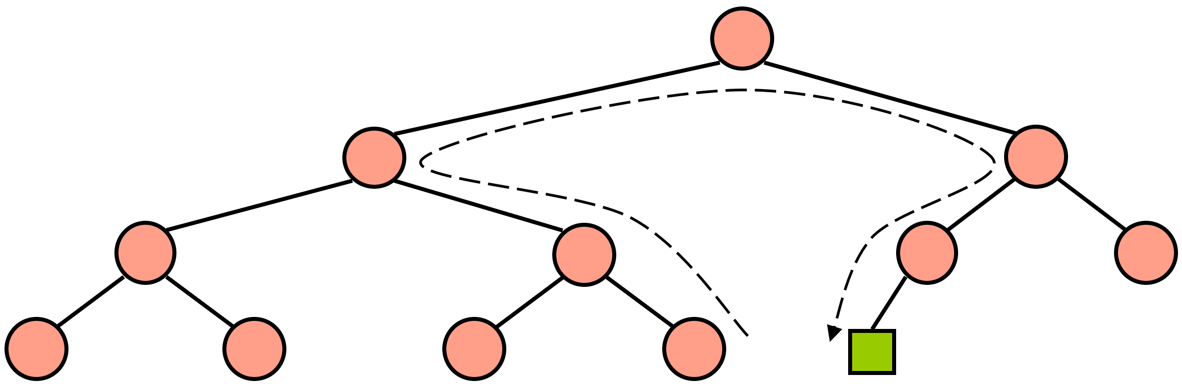
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Upheap terminates when key k reaches a leaf or a node whose children have keys greater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



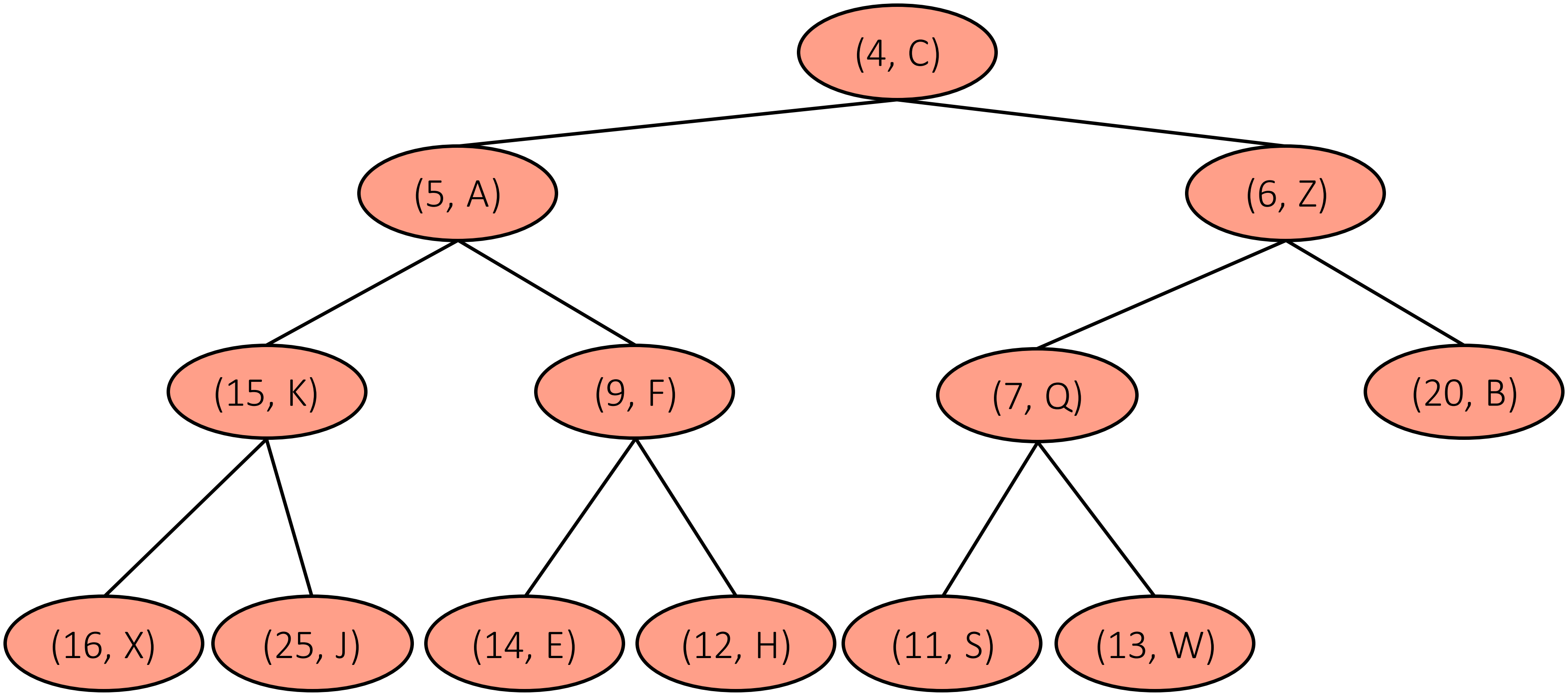


Updating the Last Node

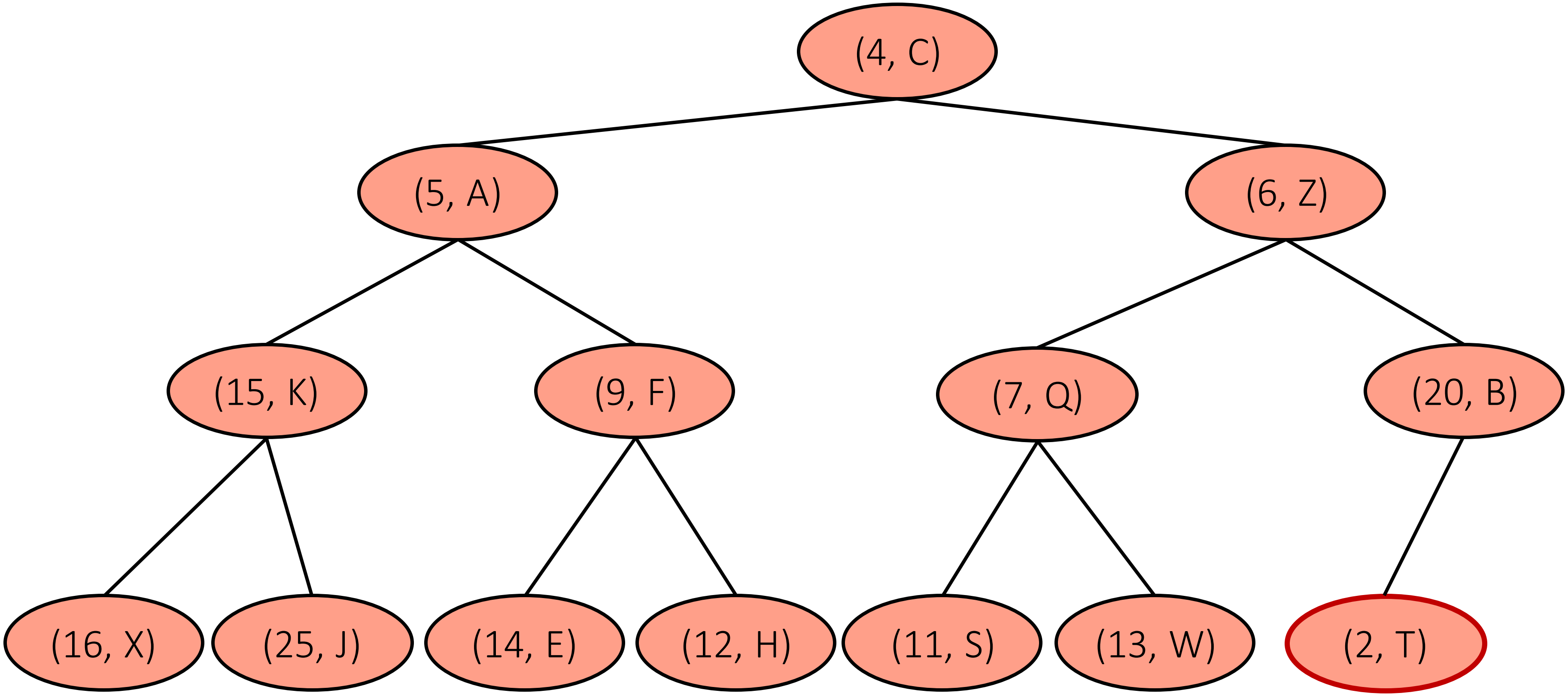
- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



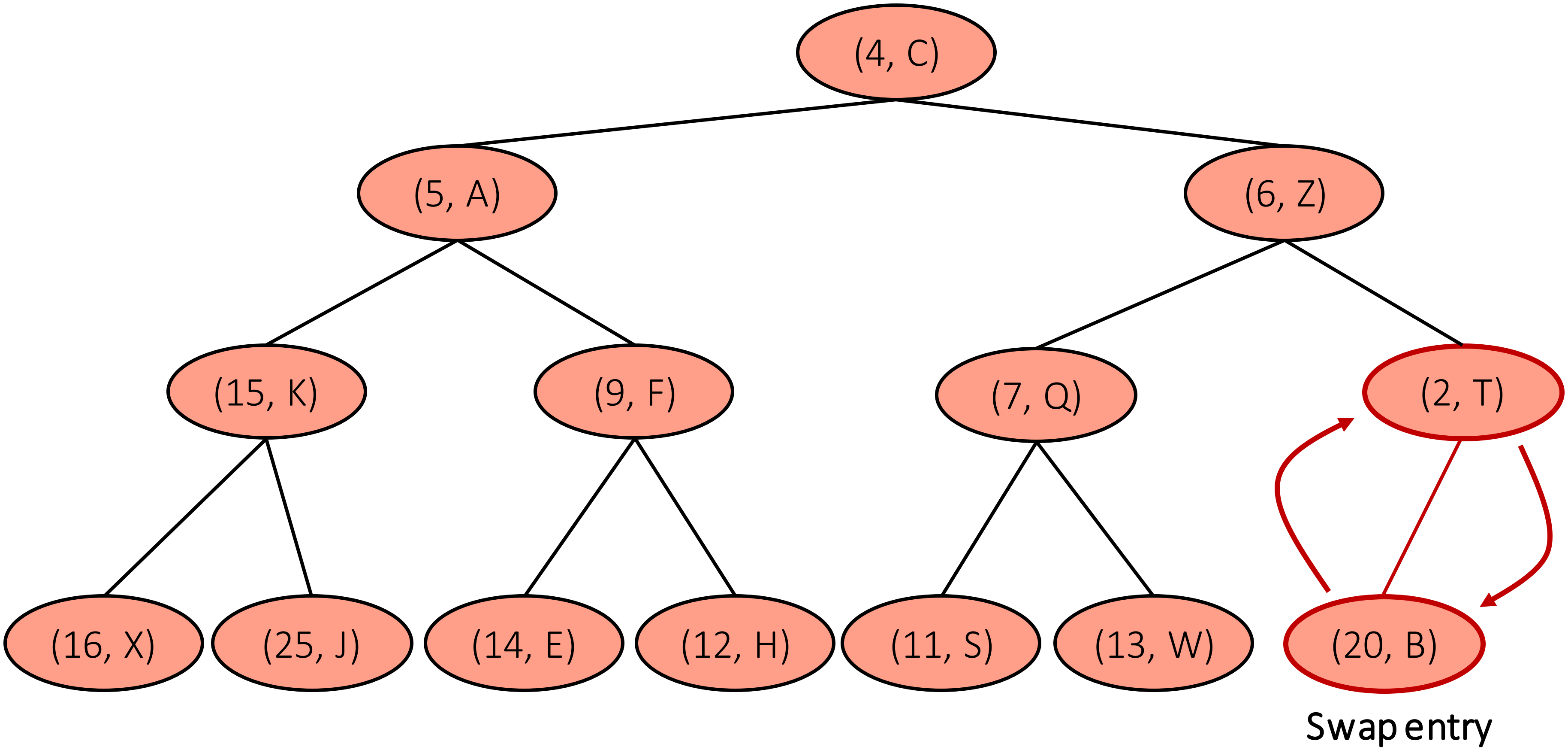
Heap-based Implementation



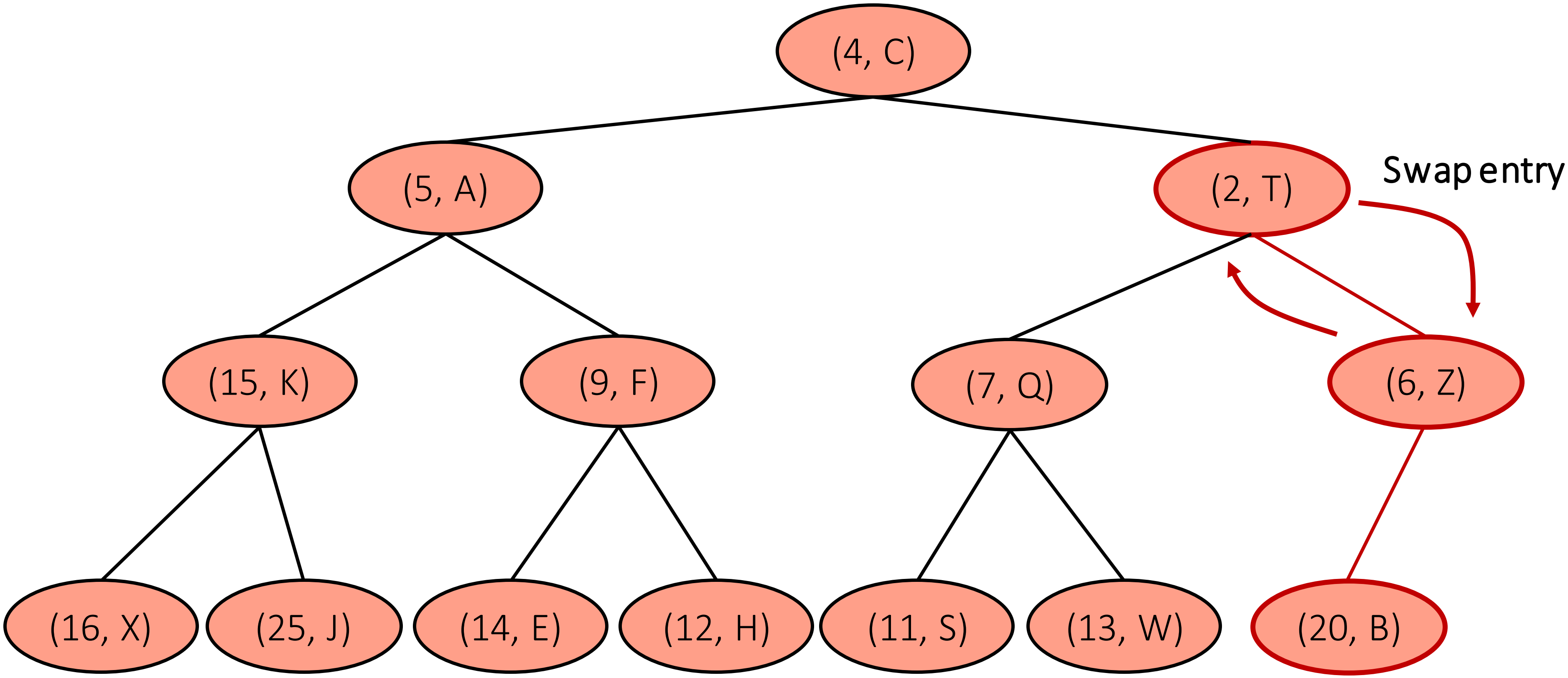
Upheap



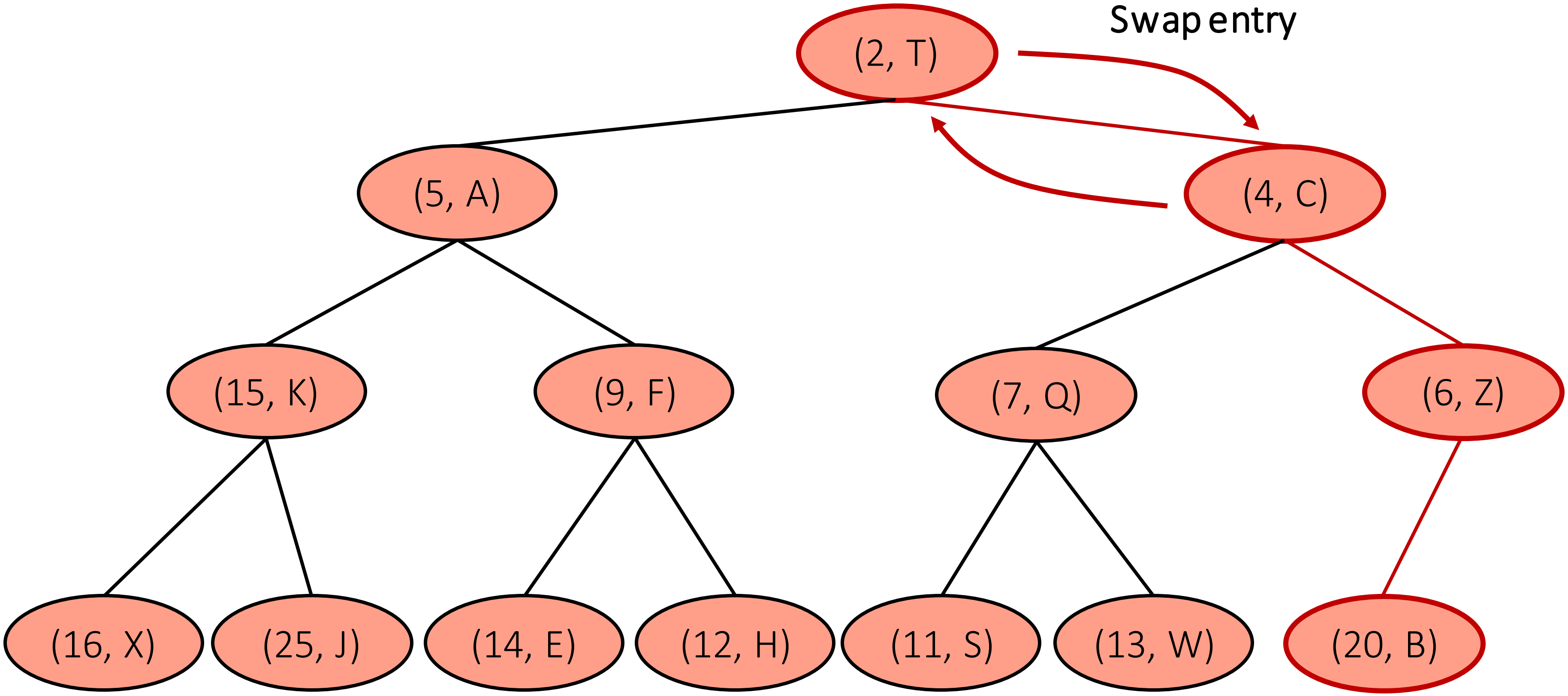
Upheap (cont'd.)



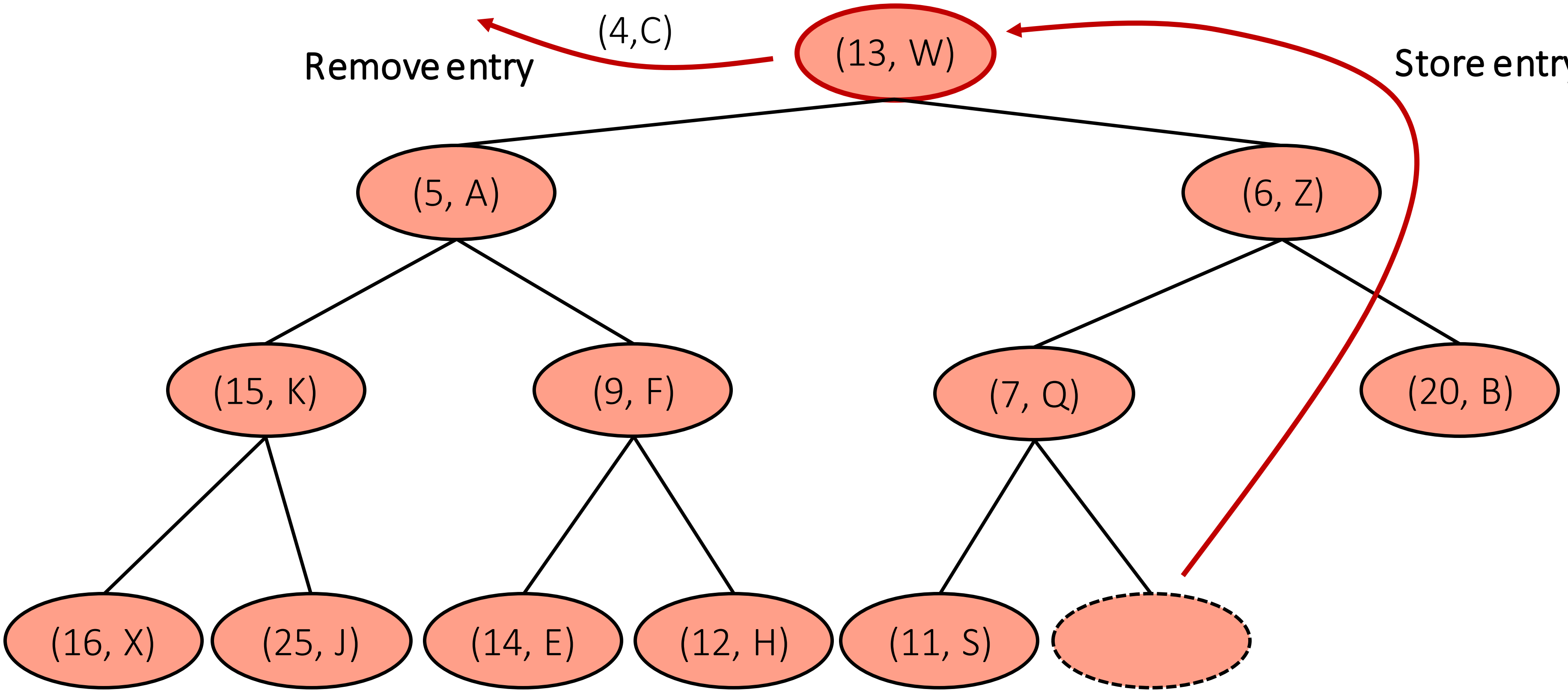
Upheap (cont'd.)



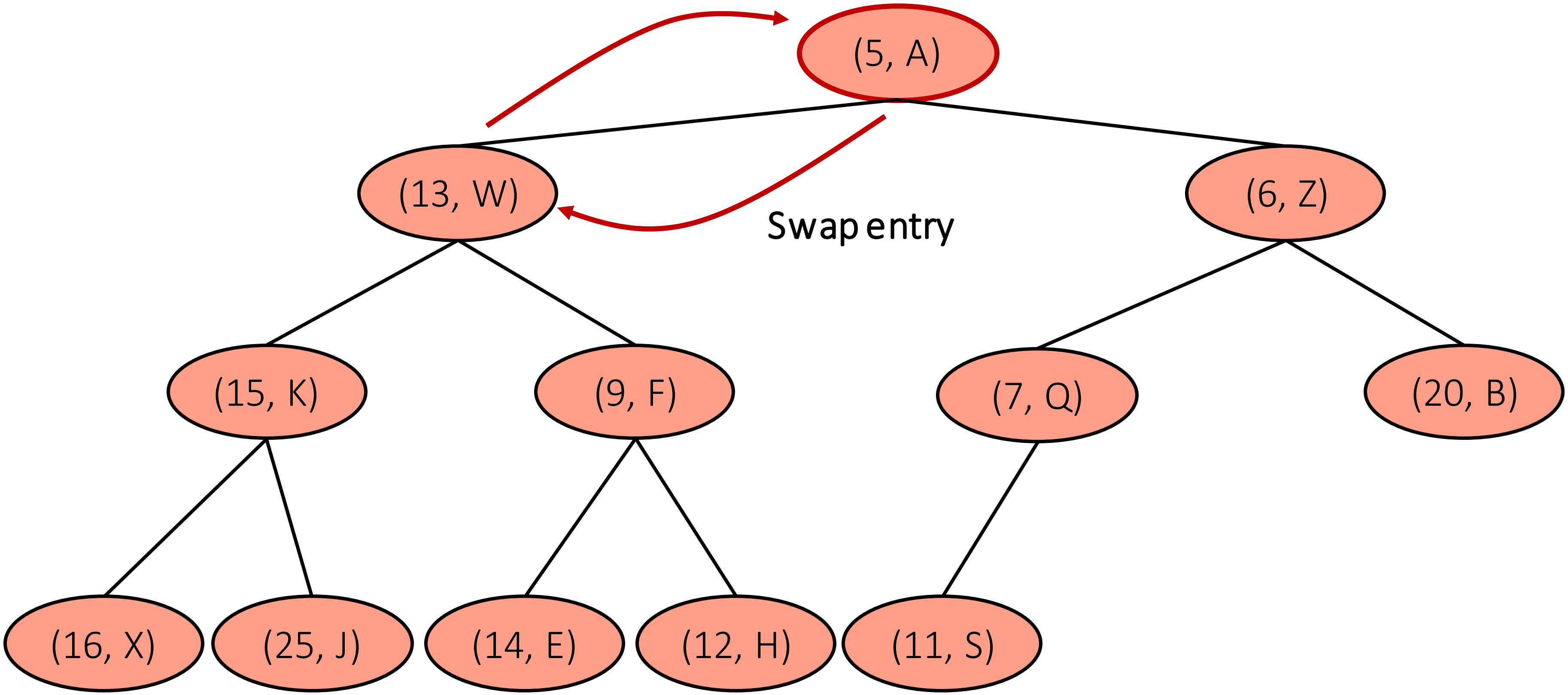
Upheap (cont'd.)



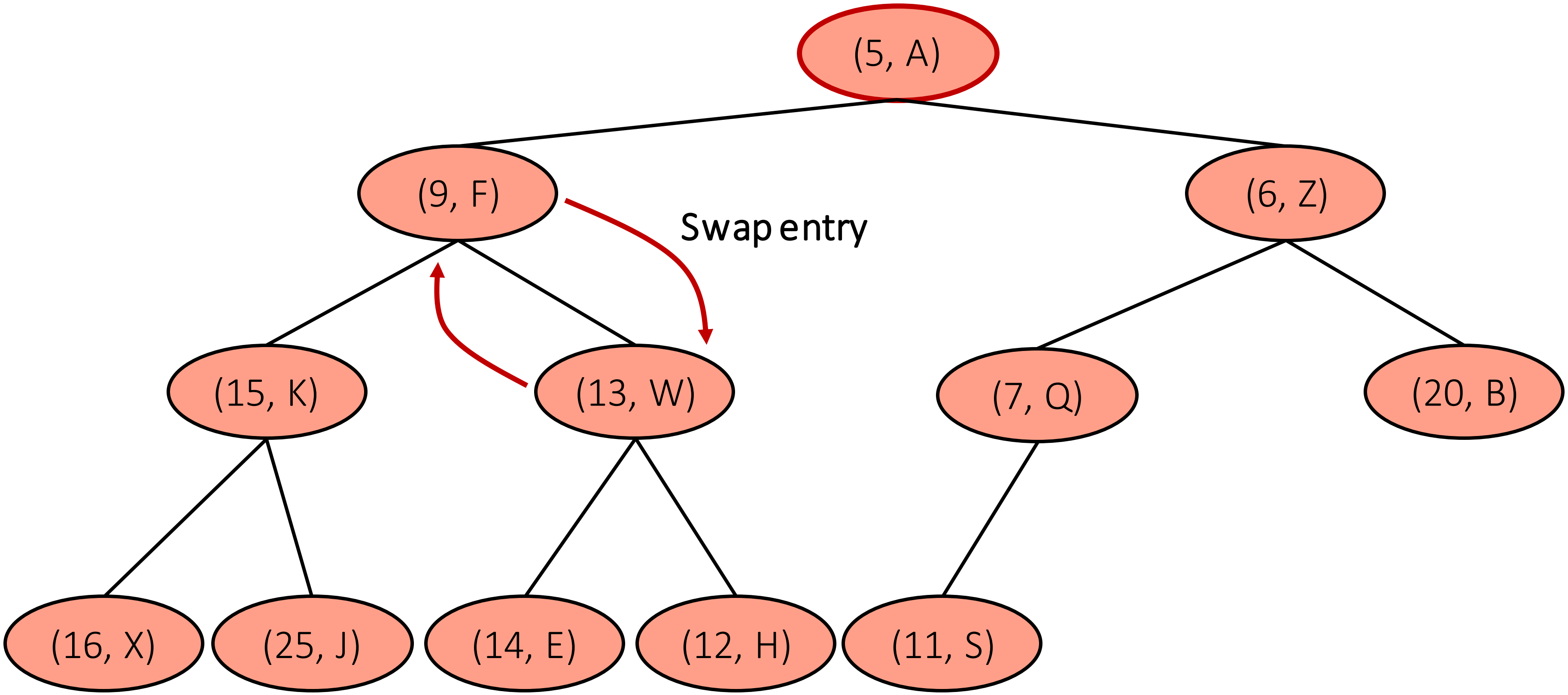
Downheap


(4,C)
Remove entry
(13, W)
Store entry
(5, A)
(6, Z)
(15, K)
(9, F)
(7, Q)
(20, B)
(16, X)
(25, J)
(14, E)
(12, H)
(11, S)

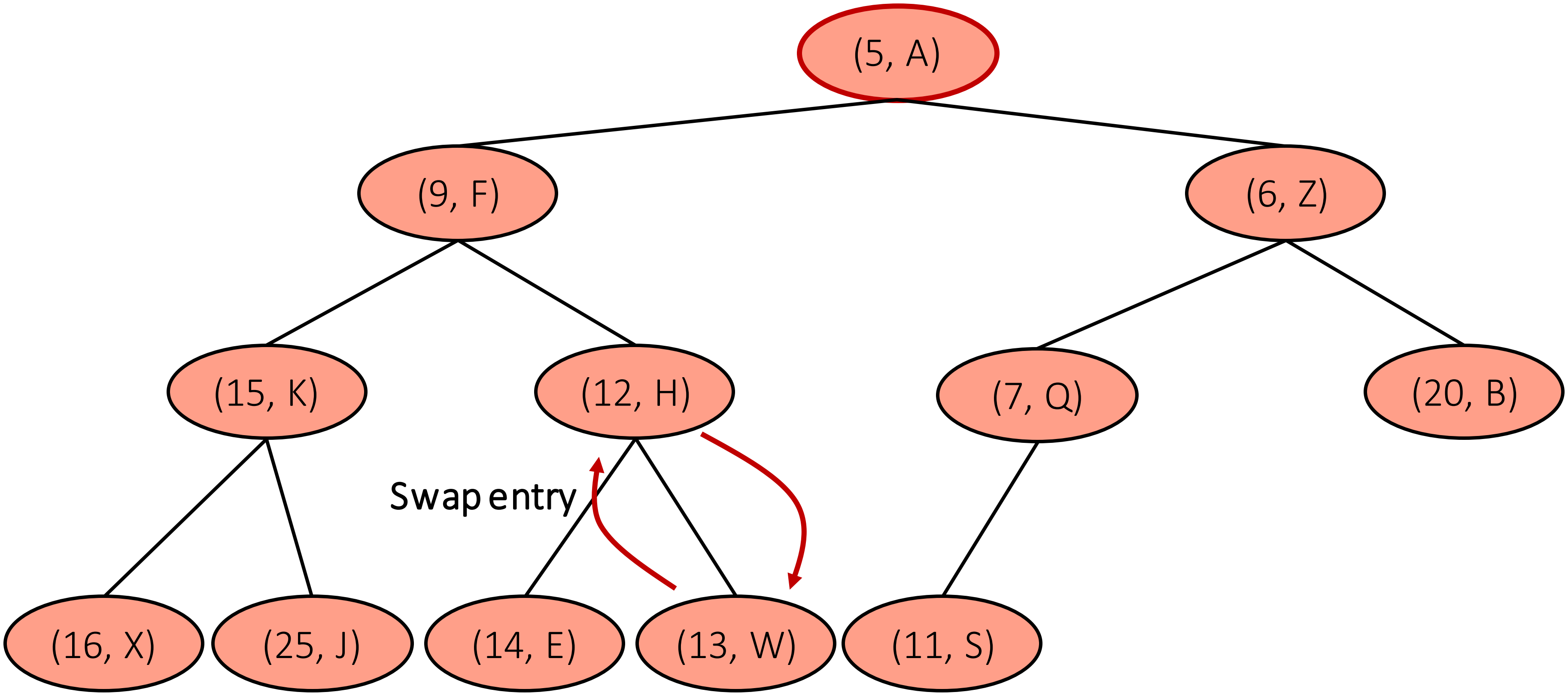
Downheap (cont'd.)



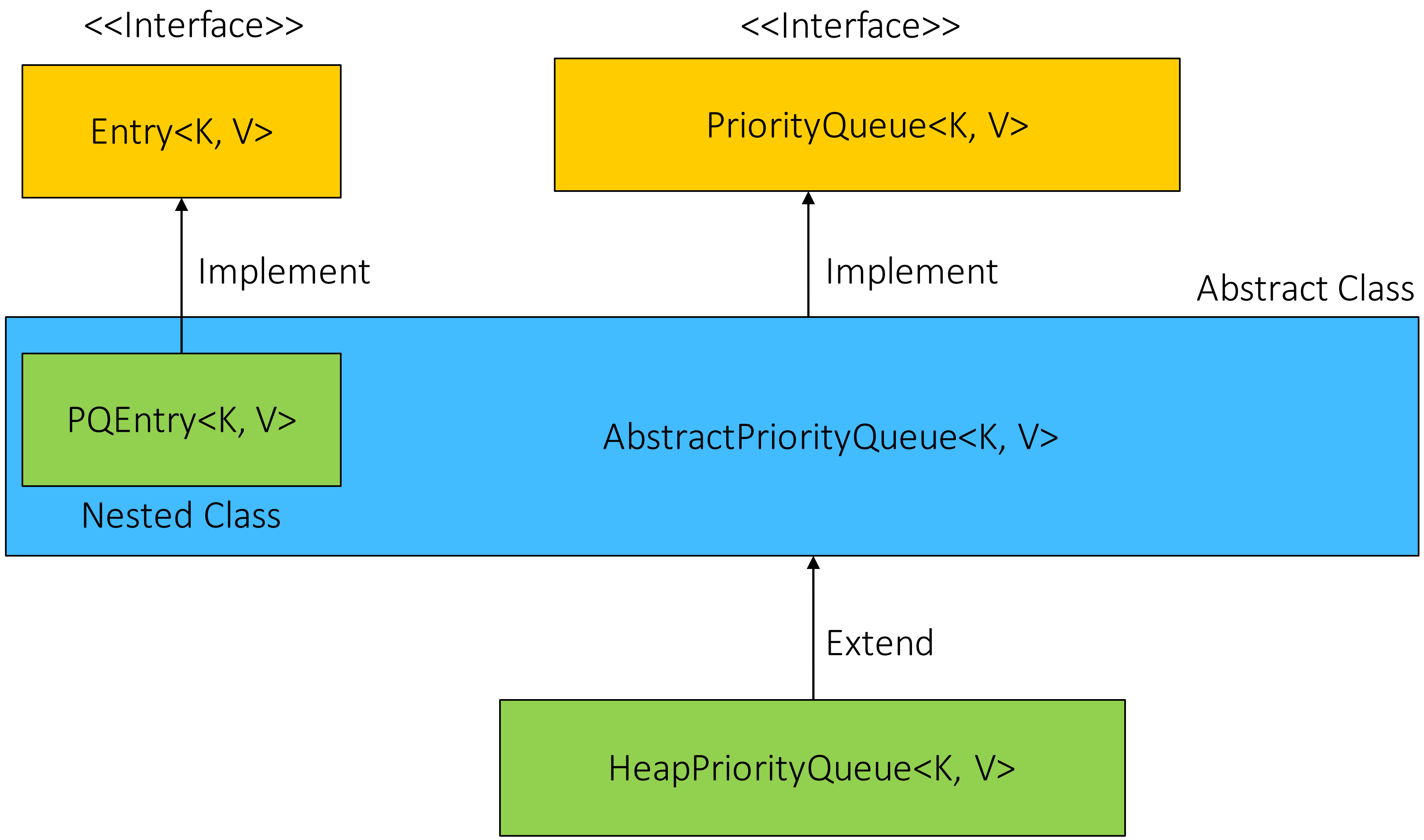
Downheap (cont'd.)



Downheap (cont'd.)



Implementation



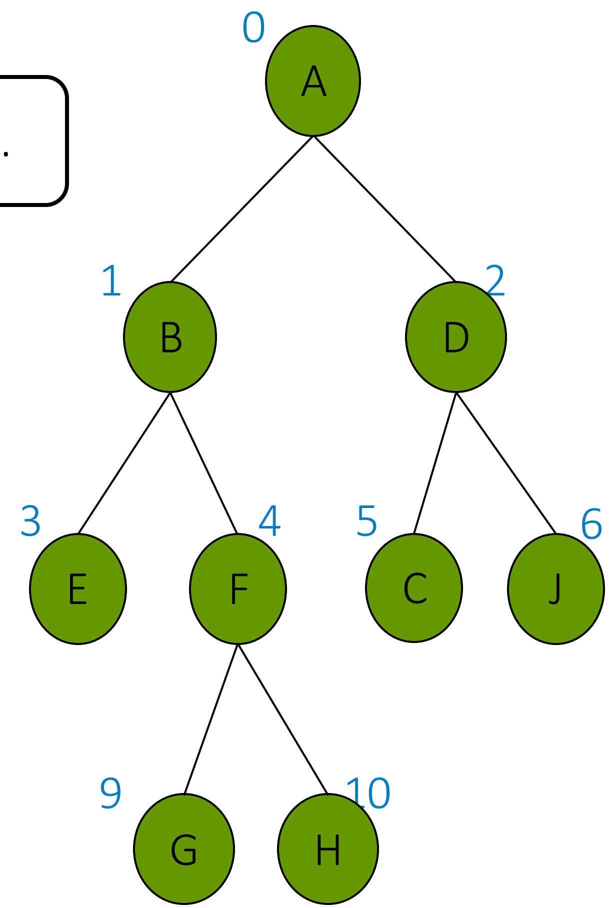
Recall Array-Based Binary Trees

- Based on a way of numbering the node of T
  - If p is the root of T, then $f(p) = 0$
  - If p is the left child of node q, then $f(p) = 2f(q) + 1$
  - If p is the right child of node q, then $f(p) = 2f(q) + 2$
  - If p is the parent of node q, then $f(p) = (f(q) – 1)/2$
  - $f()$: Level numbering of the nodes in a binary tree T

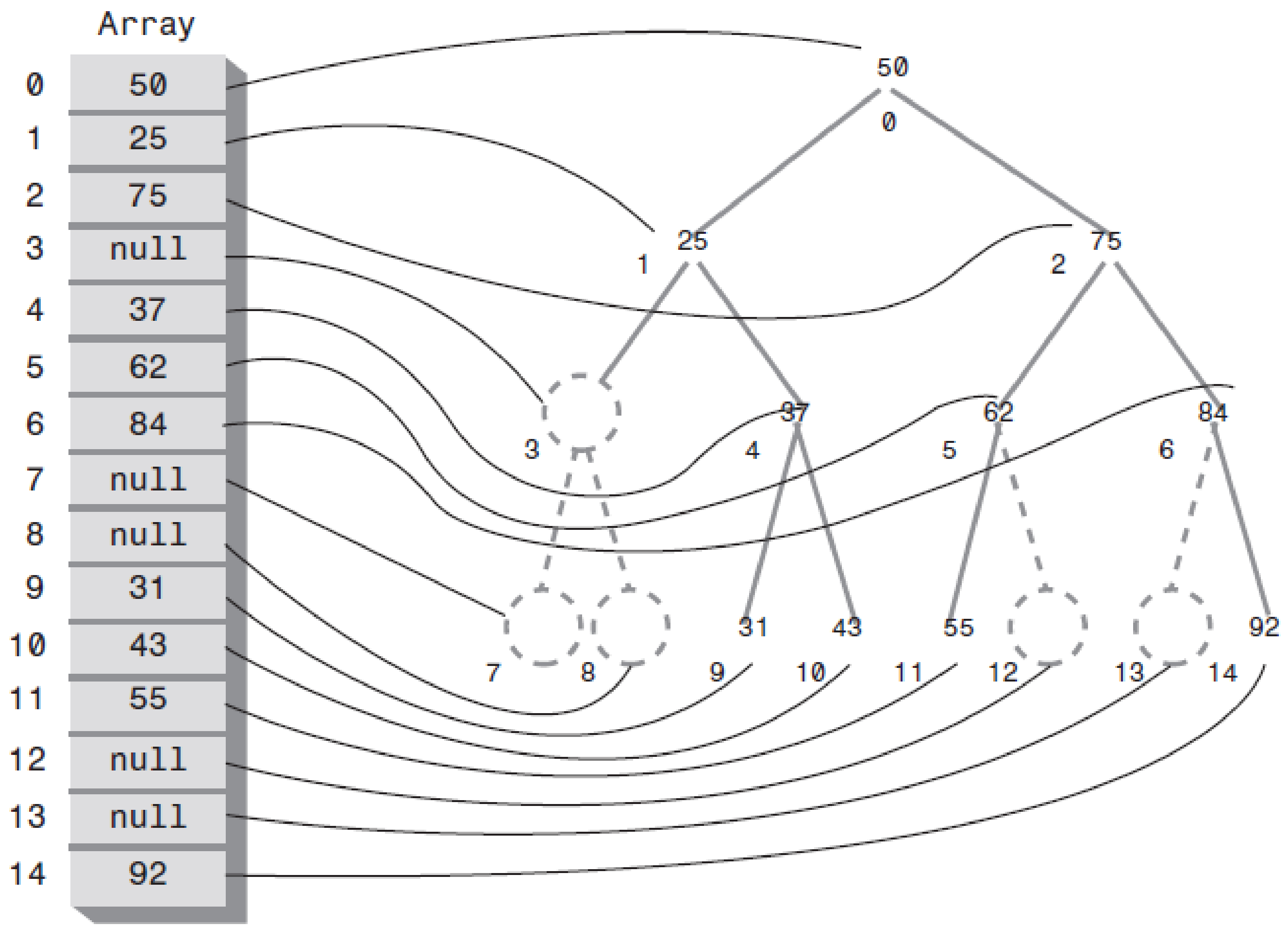
`ArrayBinaryTree.java`

Recall Array-Based Binary Trees (cont'd.)

- Nodes are stored in an array A

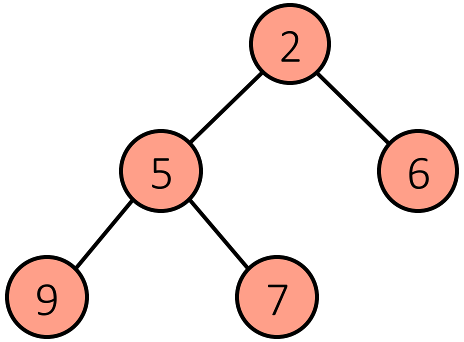


- Node v is stored at $A[rank(v)]$
  - $rank(root) = 0$
  - If node is the left child of $parent(node)$, $rank(node) = 2 \cdot rank(parent(node)) + 1$
  - If node is the right child of $parent(node)$, $rank(node) = 2 \cdot rank(parent(node)) + 2$


Array
0 50
1 25
2 75
3 null
4 37
5 62
6 84
7 null
8 null
9 31
10 43
11 55
12 null
13 null
14 92
50
0
25
1
75
2
3
37
4
62
5
84
6
31
43
55
92
7
8
9
10
11
12
13
14

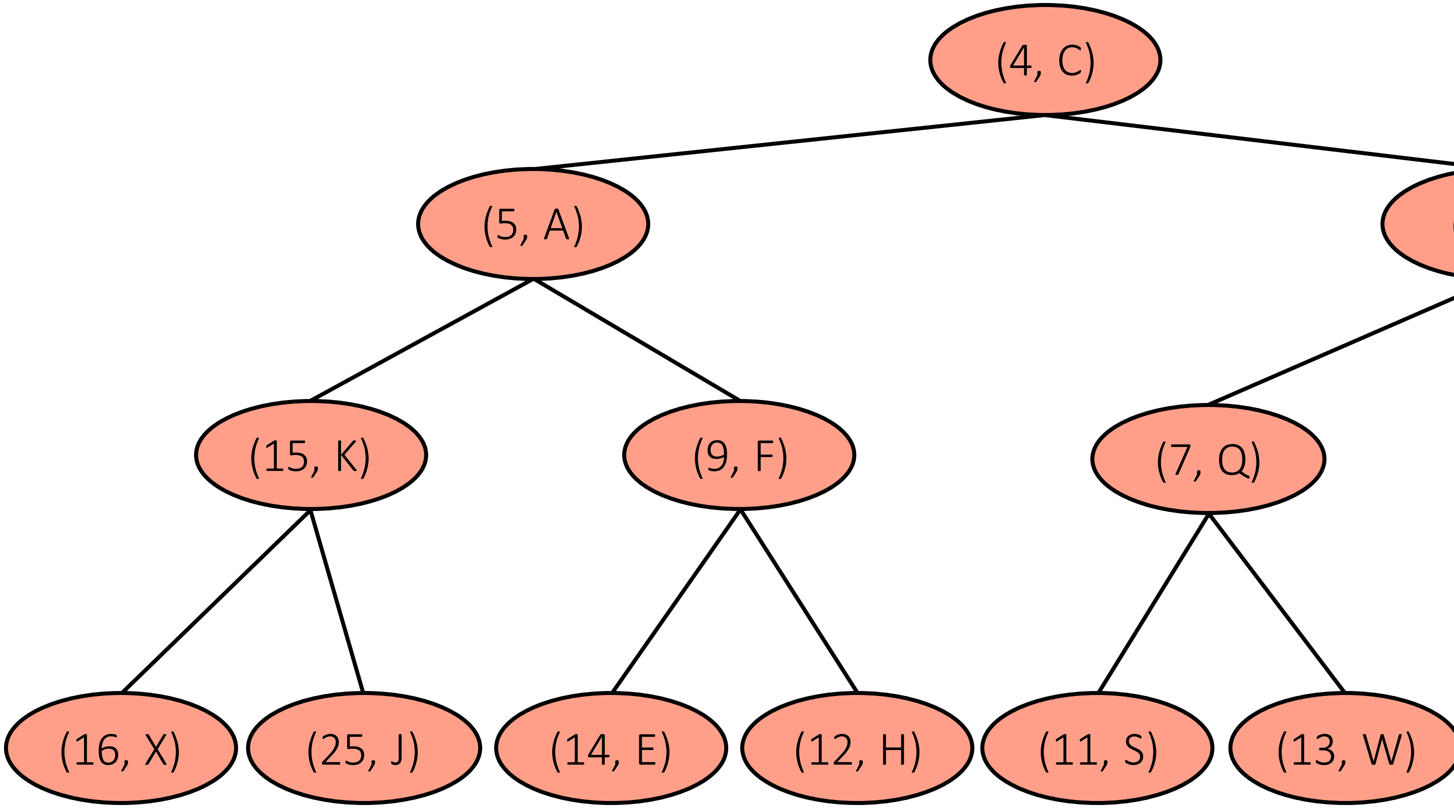
Array-based Heap Implementation

- 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 corresponds to removing at rank n
- Yields in-place heap-sort



2	5	6	9	7
0	1	2	3	4

HeapPriorityQueue.java



(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)
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance Comparison of Priority Queue

Method	Sorted array	Unsorted linked list	Sorted linked list	Heap
`size()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`isEmpty()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`min()`	$O(1)$	$O(n)$	$O(1)$	$O(1)$
`insert(e)`	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
`remove()`	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$

Priority Queue Sorting

- We can use a priority queue to sort a list of comparable elements
  - Insert the elements one by one with a series of insert operations
  - Remove the elements in sorted order with a series of remove operations

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
  $P \leftarrow$ priority queue with comparator C
  while $\neg S$.isEmpty() **do**
    $e \leftarrow S$.remove(S.first())
    P.insert(e, $\emptyset$)
  while $\neg P$.isEmpty() **do**
    $e \leftarrow P$.remove().getKey()
    S.addLast(e)

Priority Queue Sorting

- 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 list S, comparator C for the elements of S

  Output list S sorted in increasing order according to C

  $P \leftarrow$ priority queue with comparator C

  while $\neg S$.isEmpty() **do**

    $e \leftarrow S$.remove(S.first())

    P.insert(e, $\emptyset$)

  while $\neg P$.isEmpty() **do**

    $e \leftarrow P$.remove().getKey()

    S.addLast(e)

Heap-Sort

- Consider a priority queue with n items implemented by means of a heap
  - The space used is $O(n)$
  - Methods insert and remove 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

`HeapSort.java`

Recall 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:
  - Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  - Removing the elements in sorted order from the priority queue with n remove 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)	()

Recall 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:
  - Inserting the elements into the priority queue with n insert operations takes time proportional to

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

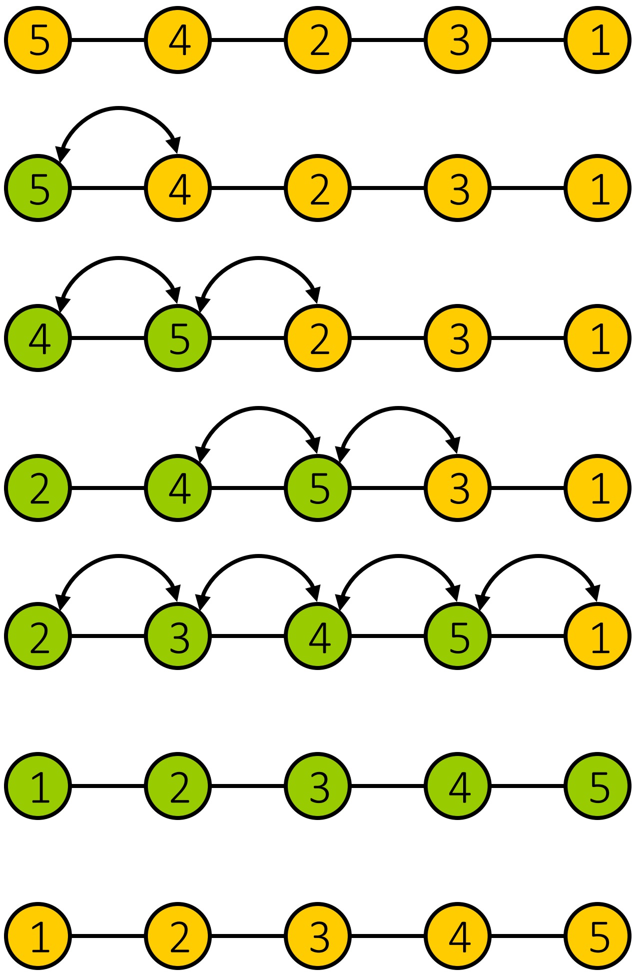
  - Removing the elements in sorted order from the priority queue with a series of n remove operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

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)	()

In-place Insertion-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



Algorithm	Time	Notes
Selection sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs) ▪ stable
Insertion sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs) ▪ stable
Heap sort	$O(n \log n)$	▪ in-place ▪ not stable
Quick sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs) ▪ not stable
Merge sort	$O(n \log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M) ▪ stable

Applications of Priority Queue

- Standby flyers
- Auctions
- Stock market

Summary

- Priority Queue ADT
  - Implementation:
    - Sorted array
    - Sorted doubly linked list
    - Unsorted doubly linked list
    - Heap
- Heap ADT: complete binary tree
  - Upheap, downheap
  - Implementation:
    - Array-list-based binary tree
- PQ sort
  - Heap sort