EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
HUMANITIES

Department of General and Computational Linguistics

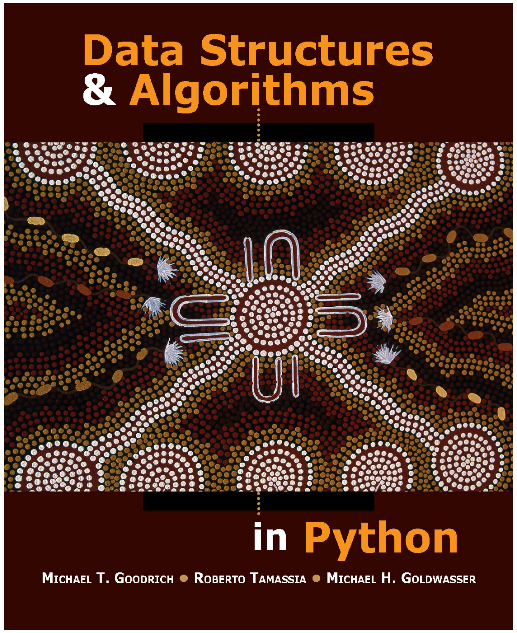
Priority Queues, Binary Heaps & Heapsort

Data Structures and Algorithms for CL III, WS 2019-2020

Corina Dima
corina.dima@uni-tuebingen.de

Data Structures & Algorithms in Python

Michael Goodrich
Roberto Tamassia
Michael Goldwasser


Data Structures
& Algorithms
in Python
Michael T. Goodrich • Roberto Tamassia • Michael H. Goldwasser

9. Priority Queues

- The Priority Queue Abstract Data Type
- Heaps
- Sorting with a Priority Queue

Priority Queue ADT

Priority Queue ADT

- A priority queue stores a collection of items
- Each item is a $(key, value)$ pair
- The $value$ is the element that should be stored
- The key is the priority associated with that particular value
- Similar to a queue, but it is the element with the minimum key that will be next removed from the queue

Main Methods of the Priority Queue ADT

- Methods supported by the priority queue ADT, for a priority queue P:
  - `P.add(k, x)`
    inserts an item with key k and value x
  - `P.min()`
    returns, but does not remove the item with the smallest key
  - `P.remove_min()`
    removes and returns the item with smallest key
  - `P.is_empty()`
    return `True` if priority queue P does not contain any items
  - `len(P)`
    return the number of items in priority queue P

Priority Queue - Example

Operation	Return Value	Priority Queue
P.add(5,A)		
P.add(9,C)		
P.add(3,B)		
P.add(7,D)		
P.min()		
P.remove_min()		
P.remove_min()		
len(P)		
P.remove_min()		
P.remove_min()		
P.is_empty()		
P.remove_min()		

Priority Queue - Example

Operation	Return Value	Priority Queue
`P.add(5,A)`		`{(5,A)}`
`P.add(9,C)`		
`P.add(3,B)`		
`P.add(7,D)`		
`P.min()`		
`P.remove_min()`		
`P.remove_min()`		
`len(P)`		
`P.remove_min()`		
`P.remove_min()`		
`P.is_empty()`		
`P.remove_min()`		

Priority Queue - Example

Operation	Return Value	Priority Queue
`P.add(5,A)`		`{(5,A)}`
`P.add(9,C)`		`{(5,A), (9,C)}`
`P.add(3,B)`		
`P.add(7,D)`		
`P.min()`		
`P.remove_min()`		
`P.remove_min()`		
`len(P)`		
`P.remove_min()`		
`P.remove_min()`		
`P.is_empty()`		
`P.remove_min()`		

Priority Queue - Example

Operation	Return Value	Priority Queue
`P.add(5,A)`		`{(5,A)}`
`P.add(9,C)`		`{(5,A), (9,C)}`
`P.add(3,B)`		`{(3,B),(5,A),(9,C)}`
`P.add(7,D)`		
`P.min()`		
`P.remove_min()`		
`P.remove_min()`		
`len(P)`		
`P.remove_min()`		
`P.remove_min()`		
`P.is_empty()`		
`P.remove_min()`		

Priority Queue - Example

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B),(5,A),(9,C)}
P.add(7,D)		{(3,B),(5,A),(7,D),(9,C)}
P.min()		
P.remove_min()		
P.remove_min()		
len(P)		
P.remove_min()		
P.remove_min()		
P.is_empty()		
P.remove_min()		

Priority Queue - Example

Operation	Return Value	Priority Queue
`P.add(5,A)`		`{(5,A)}`
`P.add(9,C)`		`{(5,A), (9,C)}`
`P.add(3,B)`		`{(3,B),(5,A),(9,C)}`
`P.add(7,D)`		`{(3,B),(5,A),(7,D),(9,C)}`
`P.min()`	`(3,B)`	`{(3,B),(5,A),(7,D),(9,C)}`
`P.remove_min()`		
`P.remove_min()`		
`len(P)`		
`P.remove_min()`		
`P.remove_min()`		
`P.is_empty()`		
`P.remove_min()`		

Priority Queue - Example

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B),(5,A),(9,C)}
P.add(7,D)		{(3,B),(5,A),(7,D),(9,C)}
P.min()	(3,B)	{(3,B),(5,A),(7,D),(9,C)}
P.remove_min()	(3,B)	{(5,A),(7,D),(9,C)}
P.remove_min()		
len(P)		
P.remove_min()		
P.remove_min()		
P.is_empty()		
P.remove_min()		

Priority Queue - Example

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B),(5,A),(9,C)}
P.add(7,D)		{(3,B),(5,A),(7,D),(9,C)}
P.min()	(3,B)	{(3,B),(5,A),(7,D),(9,C)}
P.remove_min()	(3,B)	{(5,A),(7,D),(9,C)}
P.remove_min()	(5,A)	{(7,D),(9,C)}
len(P)		
P.remove_min()		
P.remove_min()		
P.is_empty()		
P.remove_min()		

Priority Queue - Example

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B),(5,A),(9,C)}
P.add(7,D)		{(3,B),(5,A),(7,D),(9,C)}
P.min()	(3,B)	{(3,B),(5,A),(7,D),(9,C)}
P.remove_min()	(3,B)	{(5,A),(7,D),(9,C)}
P.remove_min()	(5,A)	{(7,D),(9,C)}
len(P)	2	{(7,D),(9,C)}
P.remove_min()		
P.remove_min()		
P.is_empty()		
P.remove_min()		

Priority Queue - Example

Operation	Return Value	Priority Queue
`P.add(5,A)`		`{(5,A)}`
`P.add(9,C)`		`{(5,A), (9,C)}`
`P.add(3,B)`		`{(3,B),(5,A),(9,C)}`
`P.add(7,D)`		`{(3,B),(5,A),(7,D),(9,C)}`
`P.min()`	`(3,B)`	`{(3,B),(5,A),(7,D),(9,C)}`
`P.remove_min()`	`(3,B)`	`{(5,A),(7,D),(9,C)}`
`P.remove_min()`	`(5,A)`	`{(7,D),(9,C)}`
`len(P)`	`2`	`{(7,D),(9,C)}`
`P.remove_min()`	`(7,D)`	`{(9,C)}`
`P.remove_min()`		
`P.is_empty()`		
`P.remove_min()`		

Priority Queue - Example

Operation	Return Value	Priority Queue
`P.add(5,A)`		`{(5,A)}`
`P.add(9,C)`		`{(5,A), (9,C)}`
`P.add(3,B)`		`{(3,B),(5,A),(9,C)}`
`P.add(7,D)`		`{(3,B),(5,A),(7,D),(9,C)}`
`P.min()`	`(3,B)`	`{(3,B),(5,A),(7,D),(9,C)}`
`P.remove_min()`	`(3,B)`	`{(5,A),(7,D),(9,C)}`
`P.remove_min()`	`(5,A)`	`{(7,D),(9,C)}`
`len(P)`	`2`	`{(7,D),(9,C)}`
`P.remove_min()`	`(7,D)`	`{(9,C)}`
`P.remove_min()`	`(9,C)`	`{}`
`P.is_empty()`		
`P.remove_min()`		

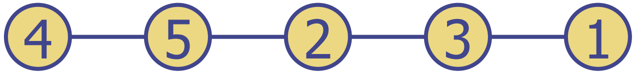
Priority Queue - Example

Operation	Return Value	Priority Queue
`P.add(5,A)`		`{(5,A)}`
`P.add(9,C)`		`{(5,A), (9,C)}`
`P.add(3,B)`		`{(3,B),(5,A),(9,C)}`
`P.add(7,D)`		`{(3,B),(5,A),(7,D),(9,C)}`
`P.min()`	`(3,B)`	`{(3,B),(5,A),(7,D),(9,C)}`
`P.remove_min()`	`(3,B)`	`{(5,A),(7,D),(9,C)}`
`P.remove_min()`	`(5,A)`	`{(7,D),(9,C)}`
`len(P)`	`2`	`{(7,D),(9,C)}`
`P.remove_min()`	`(7,D)`	`{(9,C)}`
`P.remove_min()`	`(9,C)`	`{}`
`P.is_empty()`	`True`	`{}`
`P.remove_min()`		

Priority Queue - Example

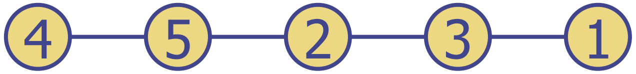
Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B),(5,A),(9,C)}
P.add(7,D)		{(3,B),(5,A),(7,D),(9,C)}
P.min()	(3,B)	{(3,B),(5,A),(7,D),(9,C)}
P.remove_min()	(3,B)	{(5,A),(7,D),(9,C)}
P.remove_min()	(5,A)	{(7,D),(9,C)}
len(P)	2	{(7,D),(9,C)}
P.remove_min()	(7,D)	{(9,C)}
P.remove_min()	(9,C)	{}
P.is_empty()	True	{}
P.remove_min()	“error”	{}

PQ Implementation with an Unsorted List



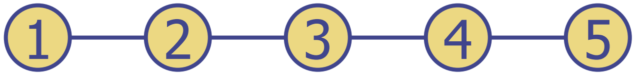
- Performance
  - `P.add(k,v)` takes ? time: the item is added at the end of the list
  - `P.remove_min()` and `P.min()` take ? time, since the list is unsorted, and all items must be inspected to find the one with minimum key

PQ Implementation with an Unsorted List



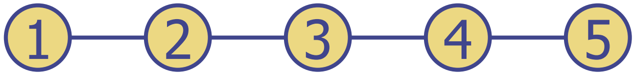
- Performance
  - `P.add(k,v)` takes $O(1)$ time: the item is added at the end of the list
  - `P.remove_min()` and `P.min()` take $O(n)$ time, since the list is unsorted, and all items must be inspected to find the one with minimum key

PQ Implementation with an Sorted List



- Performance
  - `P.add(k,v)` takes ? time, since we have to find the place where to insert the item
  - `P.remove_min()` and `P.min()` take ? time, since the smallest key is at the beginning

PQ Implementation with an Sorted List

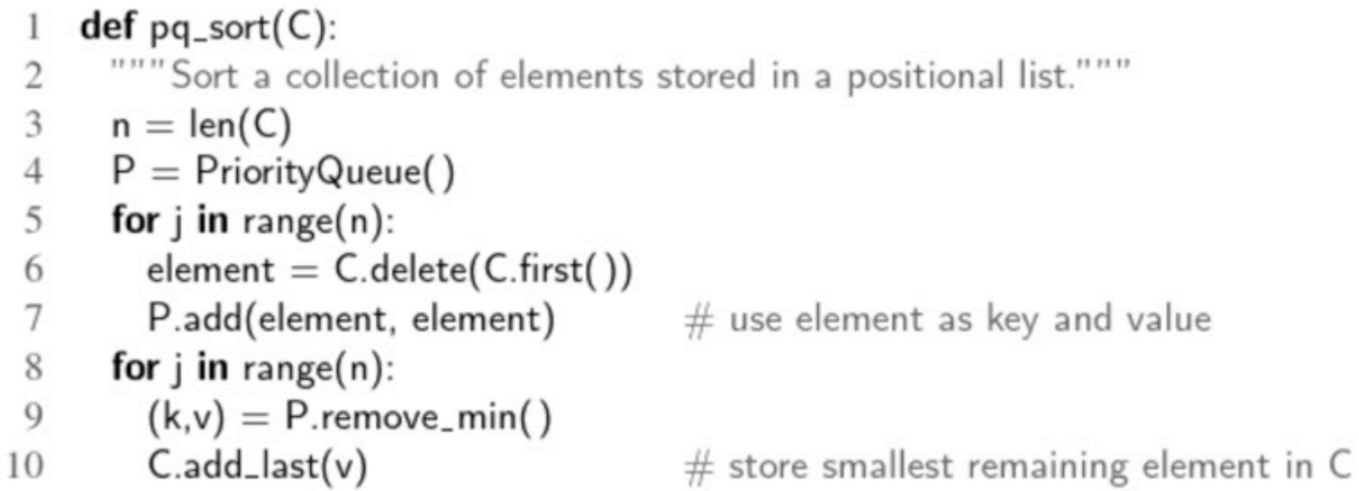


- Performance
  - `P.add(k,v)` takes $O(n)$ time, since we have to find the place where to insert the item
  - `P.remove_min()` and `P.min()` take $O(1)$ time, since the smallest key is at the beginning

Sorting with a Priority Queue

- A priority queue can be used to sort a collection of items with comparable keys

    1. Insert the items one by one using the add() operation
    2. Remove the elements in sorted order by calling remove_min() on the priority queue until all items have been removed



```
def pq_sort(C):
  """Sort a collection of elements stored in a positional list."""
  n = len(C)
  P = PriorityQueue()
  for j in range(n):
    element = C.delete(C.first())
    P.add(element, element)          # use element as key and value
  for j in range(n):
    (k,v) = P.remove_min()
    C.add_last(v)                    # store smallest remaining element in C
```

Insertion Sort Revisited

- Variant of pq_sort() where the priority queue is implemented with a sorted list
- Running time
  - Inserting the elements into the priority queue with n `add()` operations takes time proportional to

$$1 + 2 + 3 + \ ... + n = ?$$

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

Insertion Sort Revisited

- Variant of pq_sort() where the priority queue is implemented with a sorted list
- Running time
  - Inserting the elements into the priority queue with n `add()` operations takes time proportional to

$$1 + 2 + 3 + \ ... + n = \frac{n(n+1)}{2}$$

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

PQ 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)	()

Selection Sort

- Variant of pq_sort() where the priority queue is implemented with an unsorted list
- Running time
  - Inserting the elements into the priority queue with n `add()` operations takes ? time
  - Removing the elements in sorted order from the priority queue with a series of n `remove_min()` operations takes ? time

$$1 + 2 + 3 + \ldots + n = ?$$

  - selection sort runs in ? time

Selection Sort

- Variant of pq_sort() where the priority queue is implemented with an unsorted list
- Running time
  - Inserting the elements into the priority queue with n `add()` operations takes $O(n)$ time
  - Removing the elements in sorted order from the priority queue with a series of n `remove_min()` operations takes $O(n^2)$ time

$$1 + 2 + 3 + \ \ldots + n = \frac{n(n+1)}{2}$$

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

PQ 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)	()

Heaps

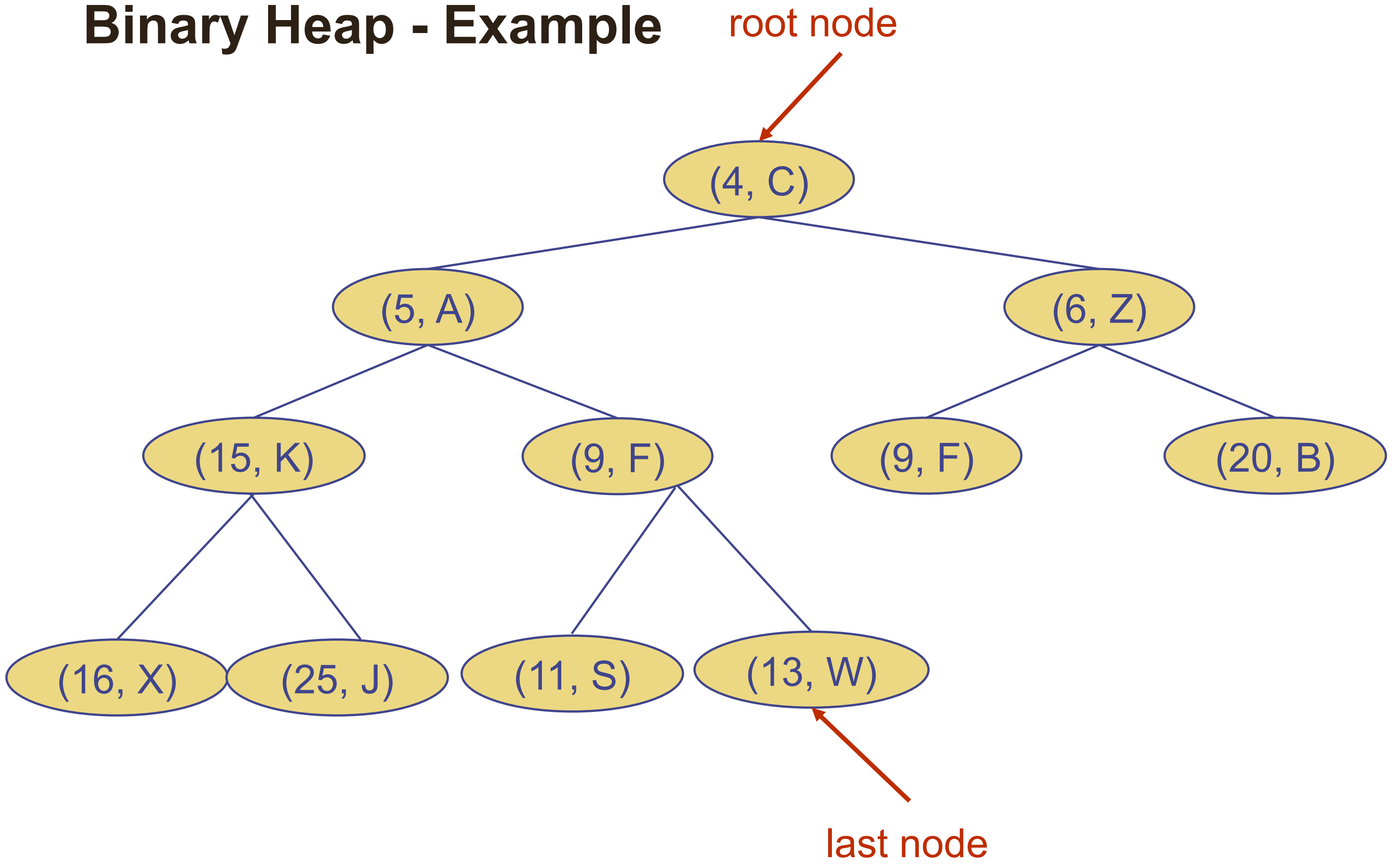
The Binary Heap Data Structure

- A binary heap is a binary tree that stores a collection of items at its nodes and satisfies the following two properties:

    1. Heap-Order Property: in a heap T, for every position p other than the root, the key stored in p is greater than or equal to the key stored at p's parent

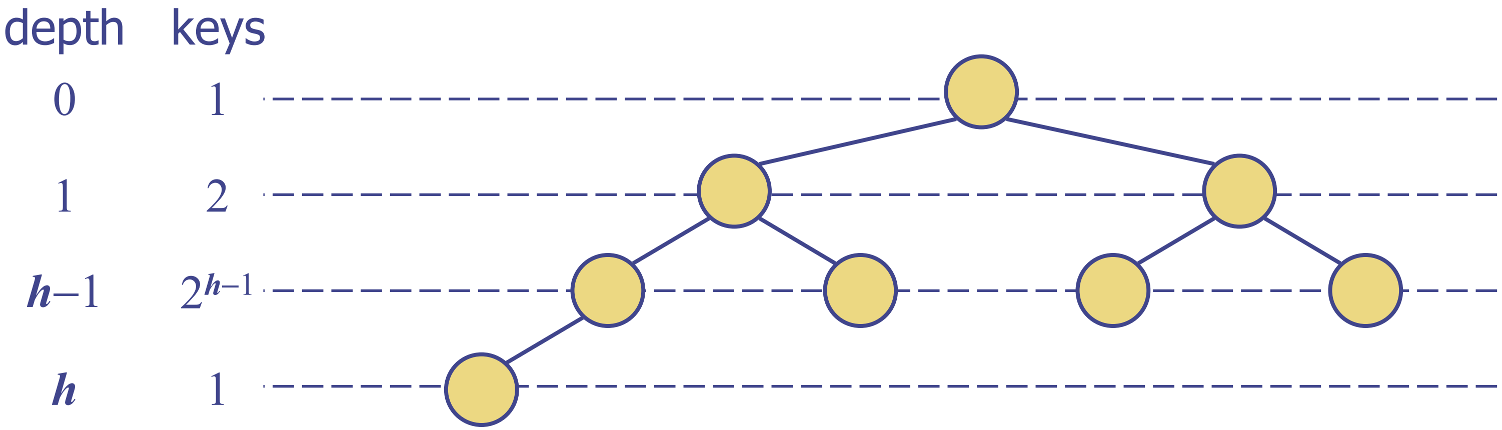
    2. Complete Binary Tree Property: a heap T with height h is a complete binary tree if levels $0, 1, \ldots, h-1$ of T have the maximum number of nodes possible, 2^i for $i = 0, \ldots, h-1$, and the remaining nodes at level h reside in the leftmost possible position at that level

Binary Heap - Example



Height of a Binary Heap

- Theorem: A heap T storing n keys has height $h = \lfloor \log_2 n \rfloor$.



depth	keys
0	1
1	2
$h-1$	2^{h-1}
h	1

Height of a Binary Heap (cont'd)

- Proof (using the complete binary tree property)
  - Let h be the height of heap T storing n keys
  - T is complete, therefore on the levels 0 through $h-1$ there are exactly $2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1$ nodes.
  - On level h, T has at least 1 and at most 2^h nodes
  - Therefore $n \geq 2^h - 1 + 1$ and $n \leq 2^h - 1 + 2^h$
  - Simplifying, $2^h \leq n \leq 2^{h+1} - 1$
  - Taking the $\log_2$ of both sides of $2^h \leq n$: $\log_2 2^h \leq \log_2 n$
  - Simplifying, $h \leq \log_2 n$
  - $n + 1 \leq 2^{h+1} \Rightarrow \log_2(n+1) \leq h + 1 \Rightarrow \log_2(n+1) - 1 \leq h$
  - $\log_2(n+1) - 1 \leq h \leq \log_2(n) \Rightarrow h = \lfloor \log_2 n \rfloor$, since h is integer

Heaps and Priority Queues

- The theorem regarding the height of a binary heap, $h = \lfloor \log_2 n \rfloor$, implies that we can perform update operations on a heap in time proportional to its height – that is – logarithmic time, $O(\log n)$
- Make the priority queue operations more efficient by implementing priority queues using binary heaps

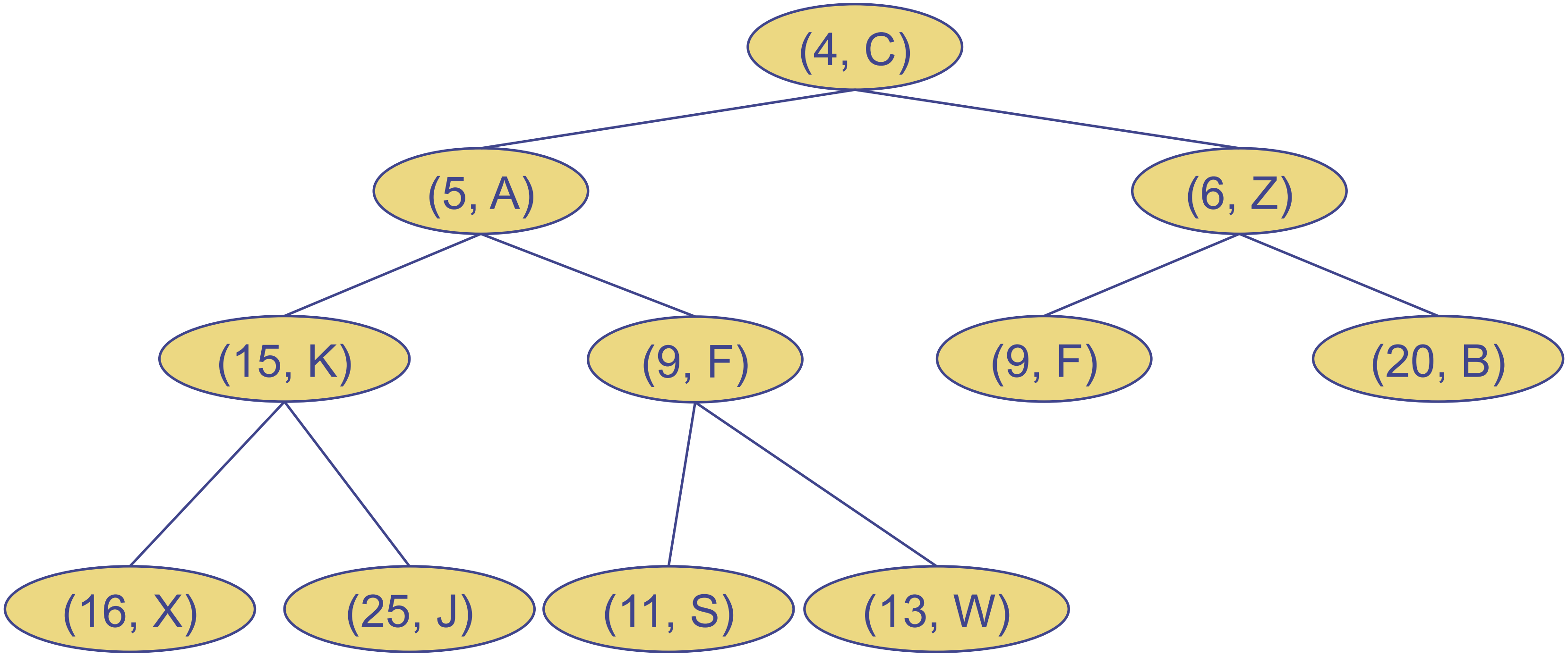
Adding an Item to a Binary Heap

- `P.add(k,v)`, implemented with a binary heap T
- The pair (k, v) is stored as an item at a new node in the tree
- To maintain the complete binary tree property of the heap, the new node should be placed at position p:
  - Just beyond the rightmost node at the bottom level of the tree
  - Or at the leftmost position of a new level, if the bottom level is already full or the heap is empty

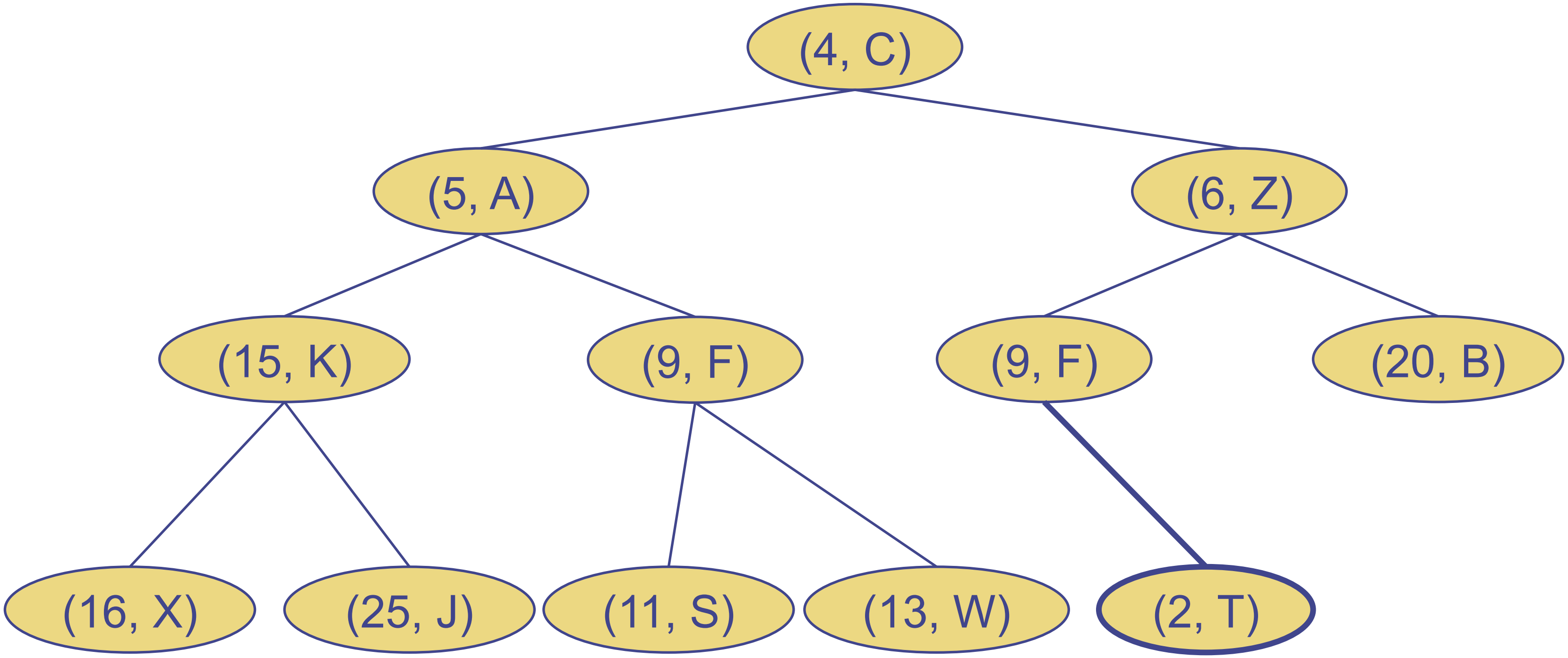
Up-Heap Bubbling After an Insertion

- After a new pair (k, v) has been inserted, the tree T is complete, but it may violate the heap-order property
- Up-heap bubbling is ran to ensure that the new entry is placed at it's proper place, by using swaps
- The key at position p is compared to that of p's parent, q
  - If $k_p \geq k_q$, the heap order property is satisfied, stop
  - If $k_p < k_q$, the heap order property has to be restored locally, by swapping the items on positions p and q; the new item moves up one level; the heap order property might still need to be restored, so the swapping process continues until the heap order property is again satisfied

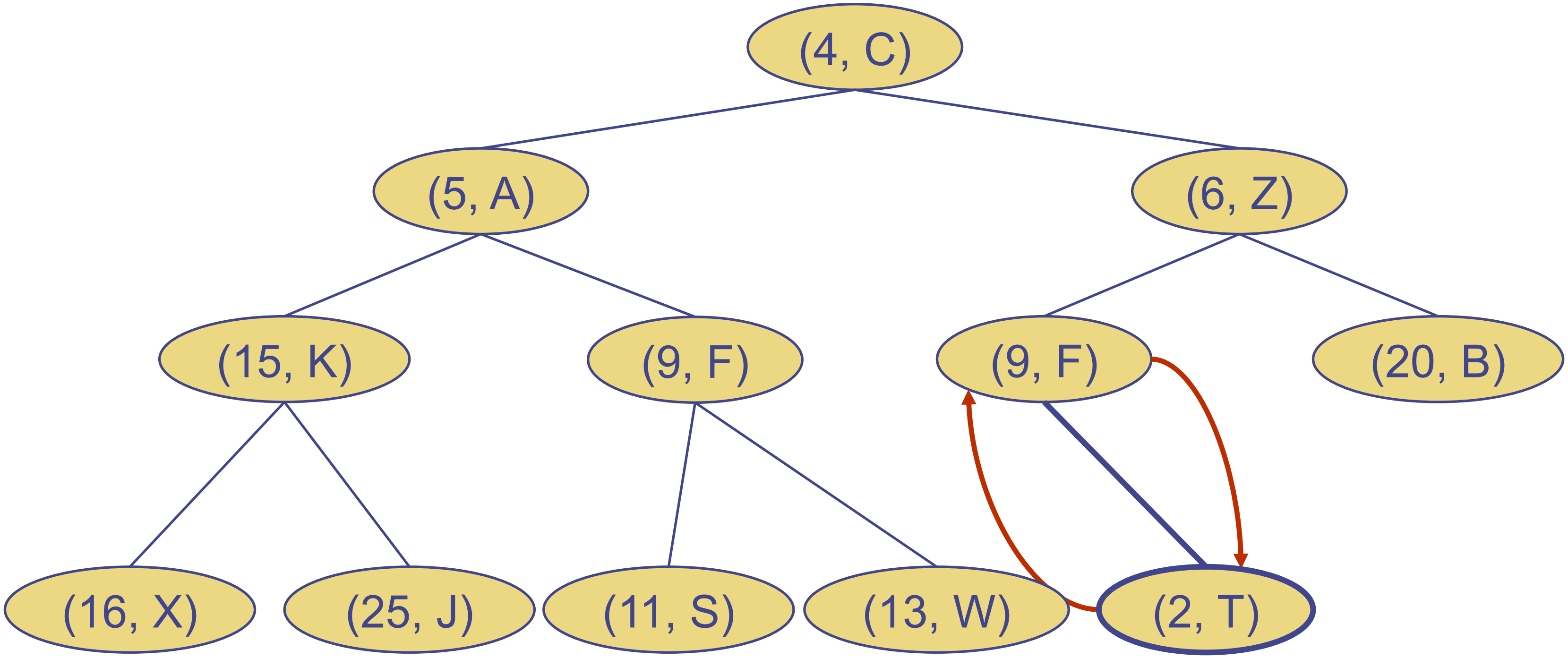
Up-Heap Bubbling - Example



Up-Heap Bubbling – Example (cont'd)

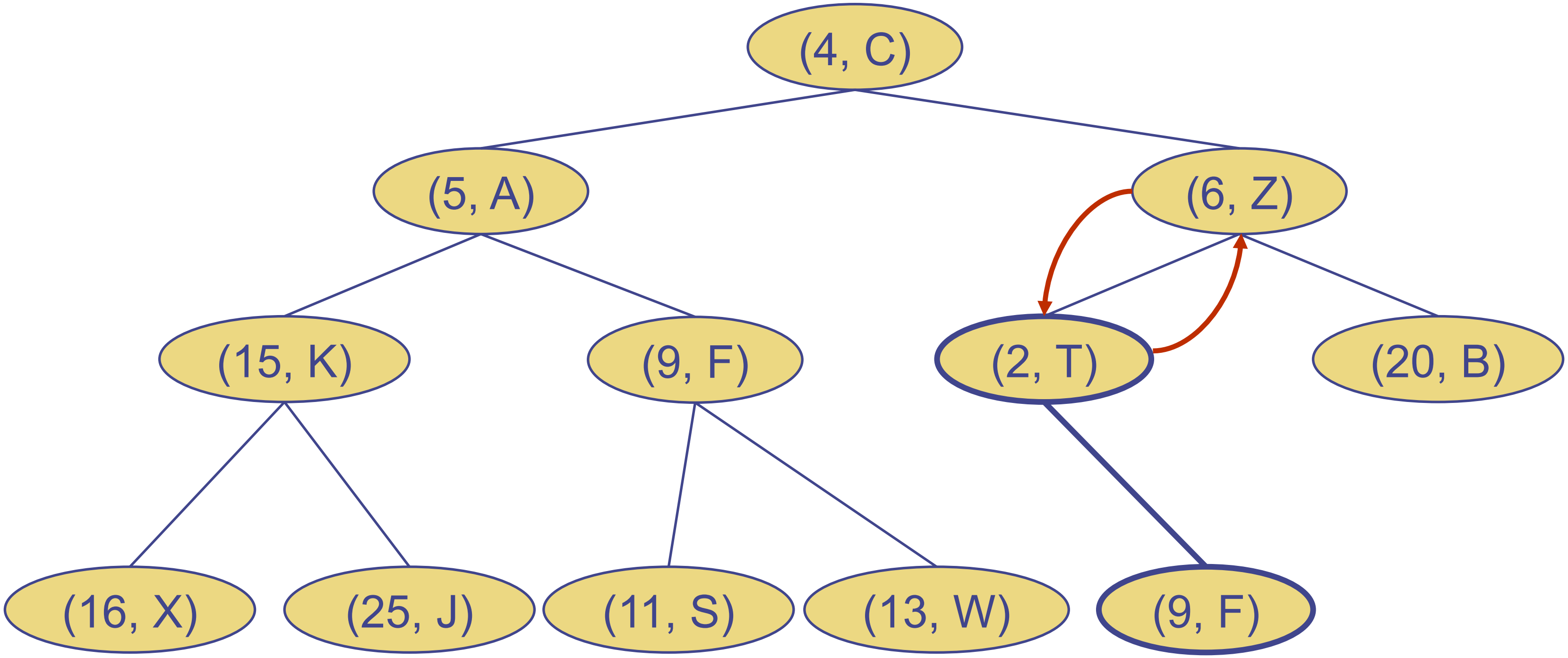


Up-Heap Bubbling – Example (cont'd)



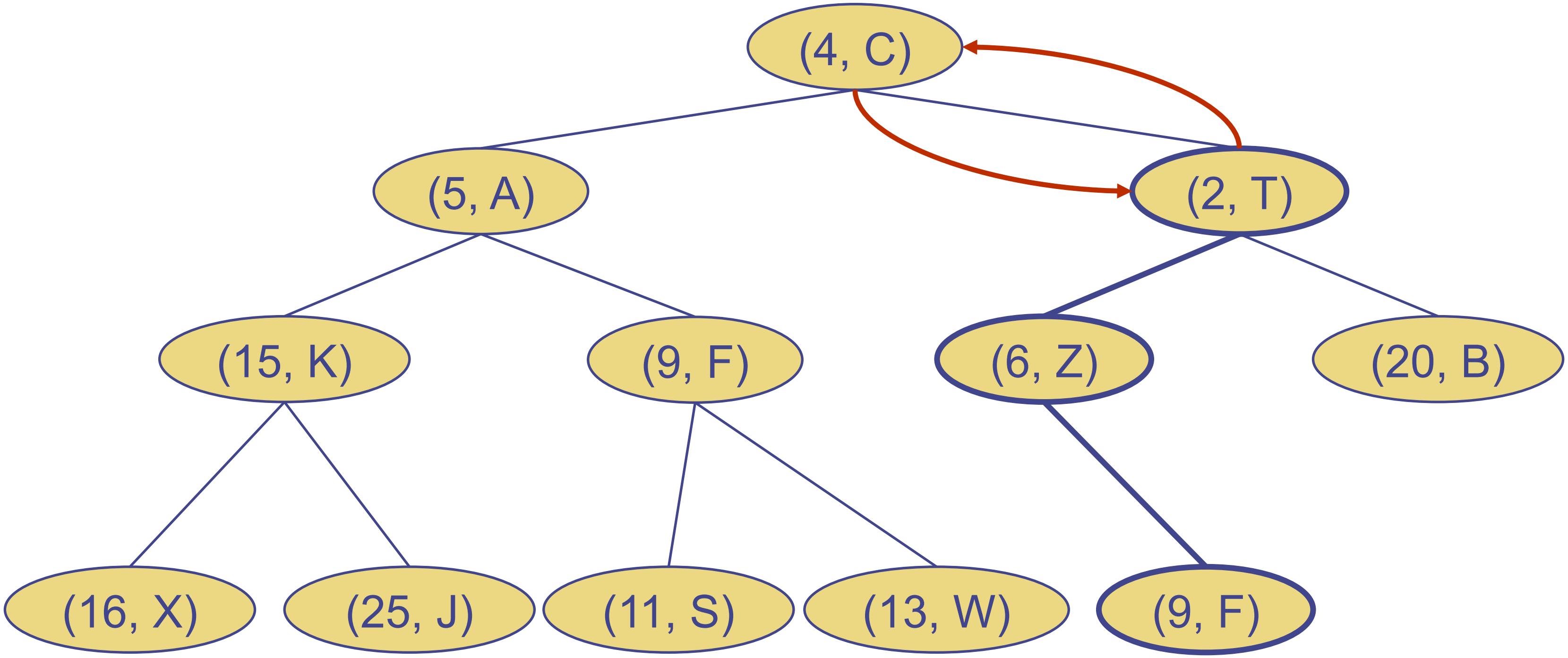
- Heap-order property not satisfied, swap item with key 2 with the parent item with key 9

Up-Heap Bubbling – Example (cont’d)



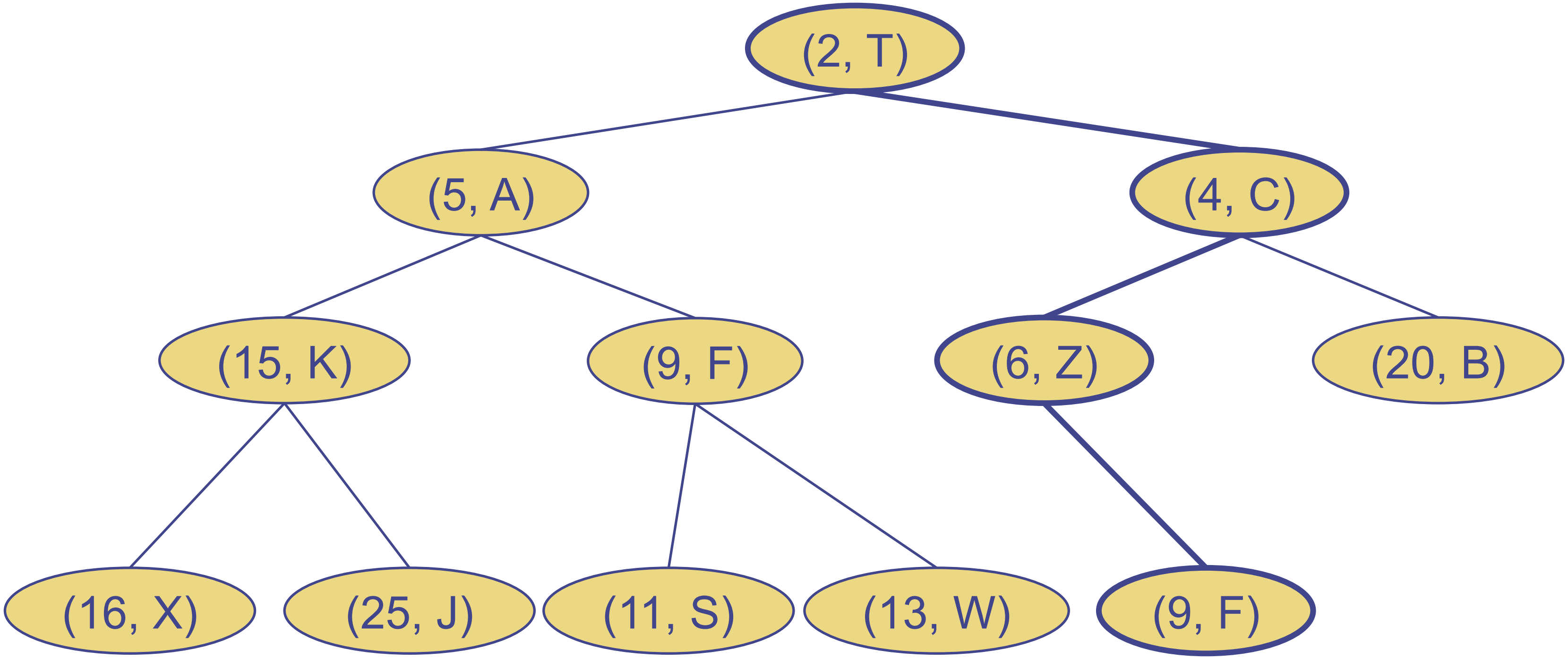
- Heap-order property not satisfied, swap item with key 2 with the parent item with key 6

Up-Heap Bubbling – Example (cont'd)



- Heap-order property not satisfied, swap item with key 2 with root item, with key 4

Up-Heap Bubbling – Example (cont'd)



- Heap-order property is satisfied, stop

Up-Heap Bubbling – Wrap-up

- The up-heap bubbling process terminates when the new item with key k reaches the root, or a node whose parent has a key smaller than or equal to k
- Since a binary heap has height $h = \lfloor \log n \rfloor$, the up-heap bubbling process runs in $O(\log n)$ time

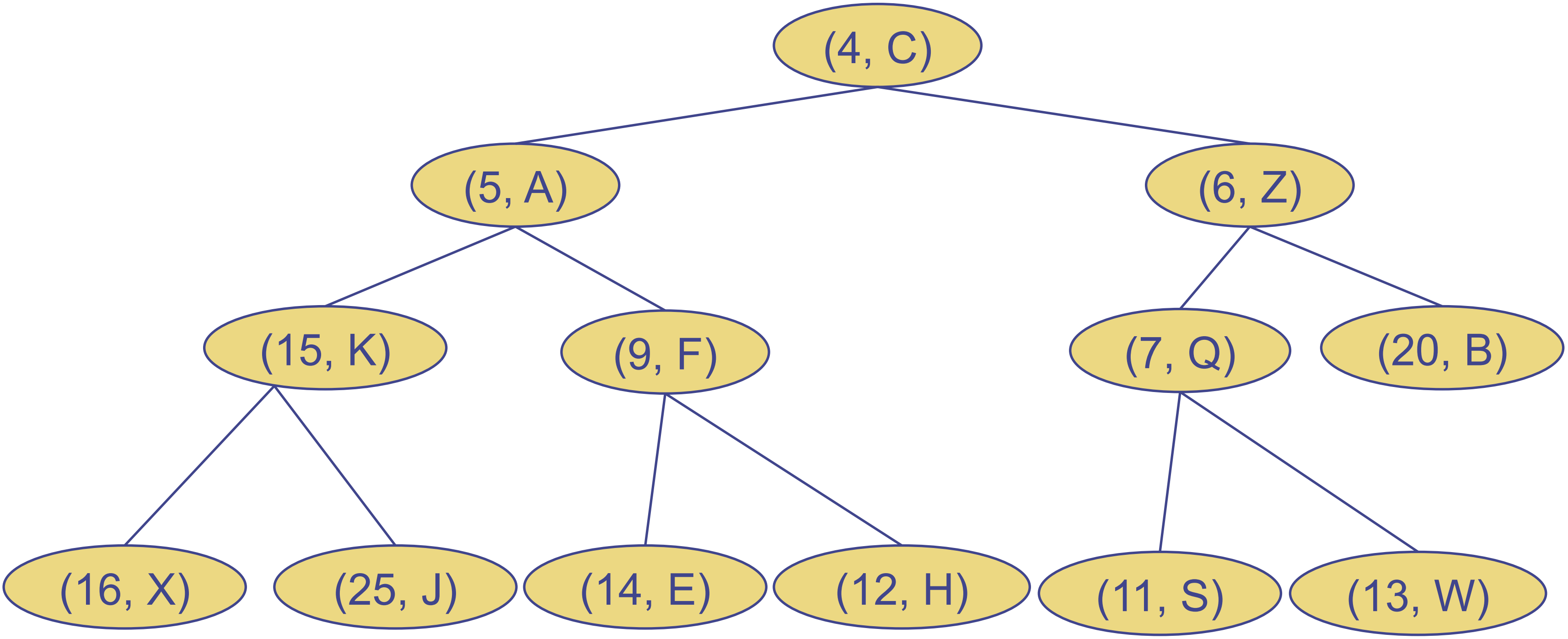
Removing the Item with Minimum Key from a Binary Heap

- The item with the smallest key is stored at the root of the heap T
- The root item cannot simply be removed – this would lead to two disconnected trees
- Instead, the leaf at the last position p of T is removed (the rightmost leaf on the lowest level of T), thus ensuring that the heap keeps respecting the complete binary tree property
- The last item is preserved by copying it into the root element r, in place of the minimum element
- The heap-order property is then restored via the process called down-heap bubbling

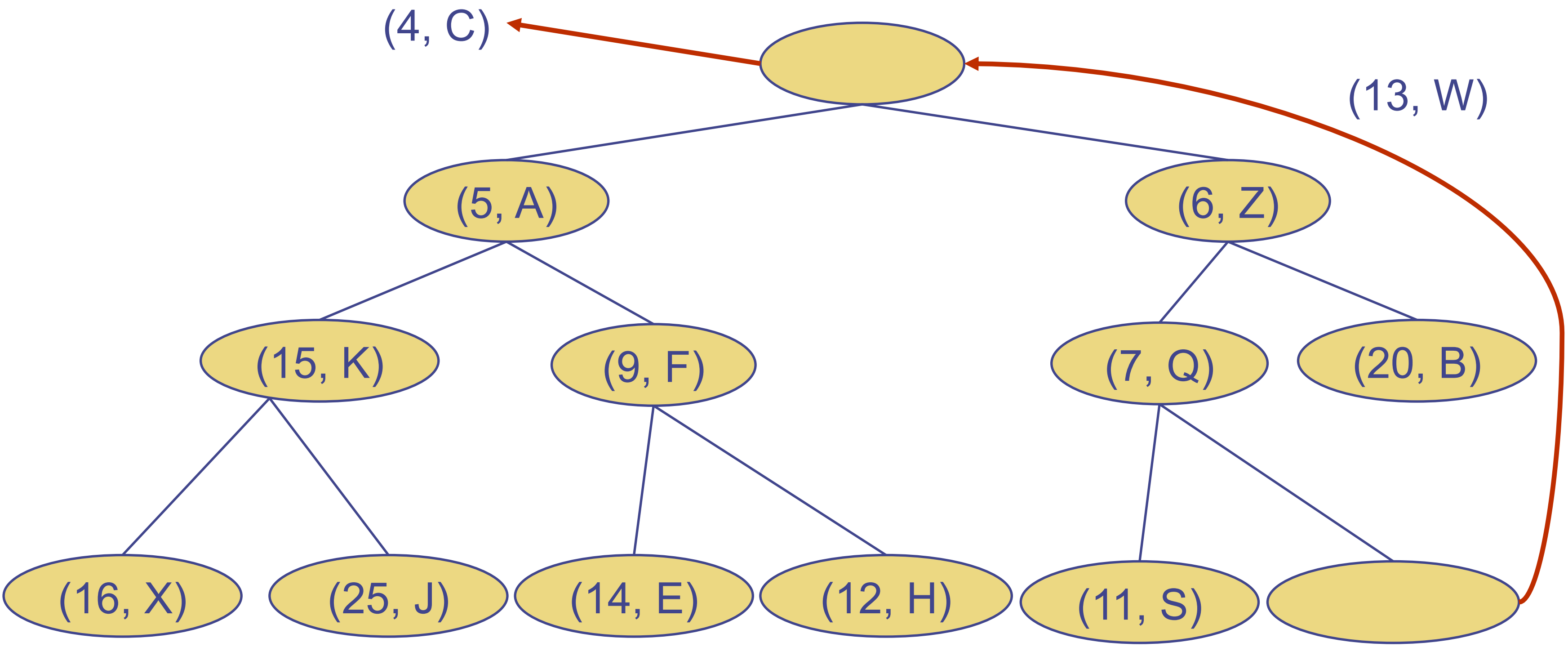
Down-Heap Bubbling

- After replacing the root item with the item with key k from the last node, the heap-order property might be violated
- Down-heap bubbling restores the heap-order property by swapping the item with key k along a downward path from the root
- If p initially denotes the root of T, two cases can be distinguished in the process:
  - If p has no right child, then c is the left child of p
  - If p has both a left and a right child, then c is the child of p with minimal key
- If $k_c \leq k_p$, the heap-order property is satisfied, stop
- If $k_c > k_p$, the heap-order property has to be restored, by swapping the items on positions p and c; process continues until the heap-order property is restored

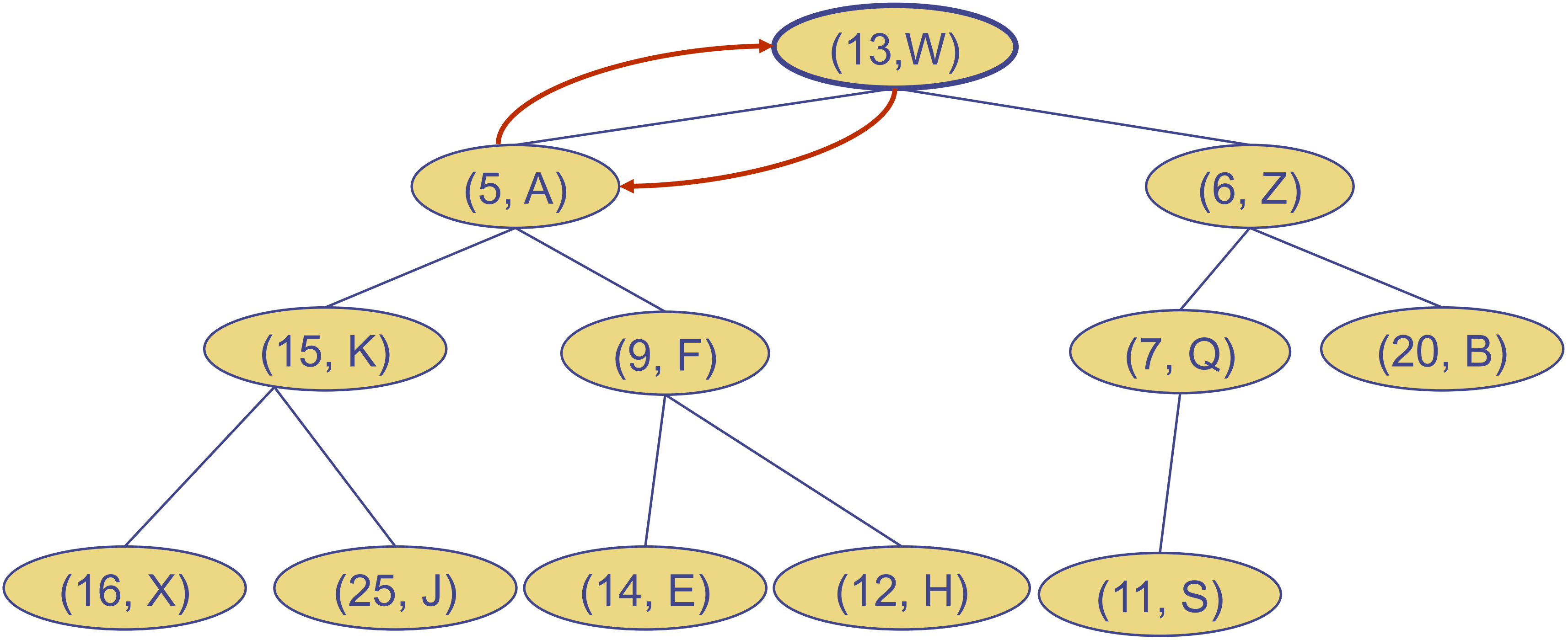
Down-Heap Bubbling - Example



Down-Heap Bubbling – Example (cont'd)

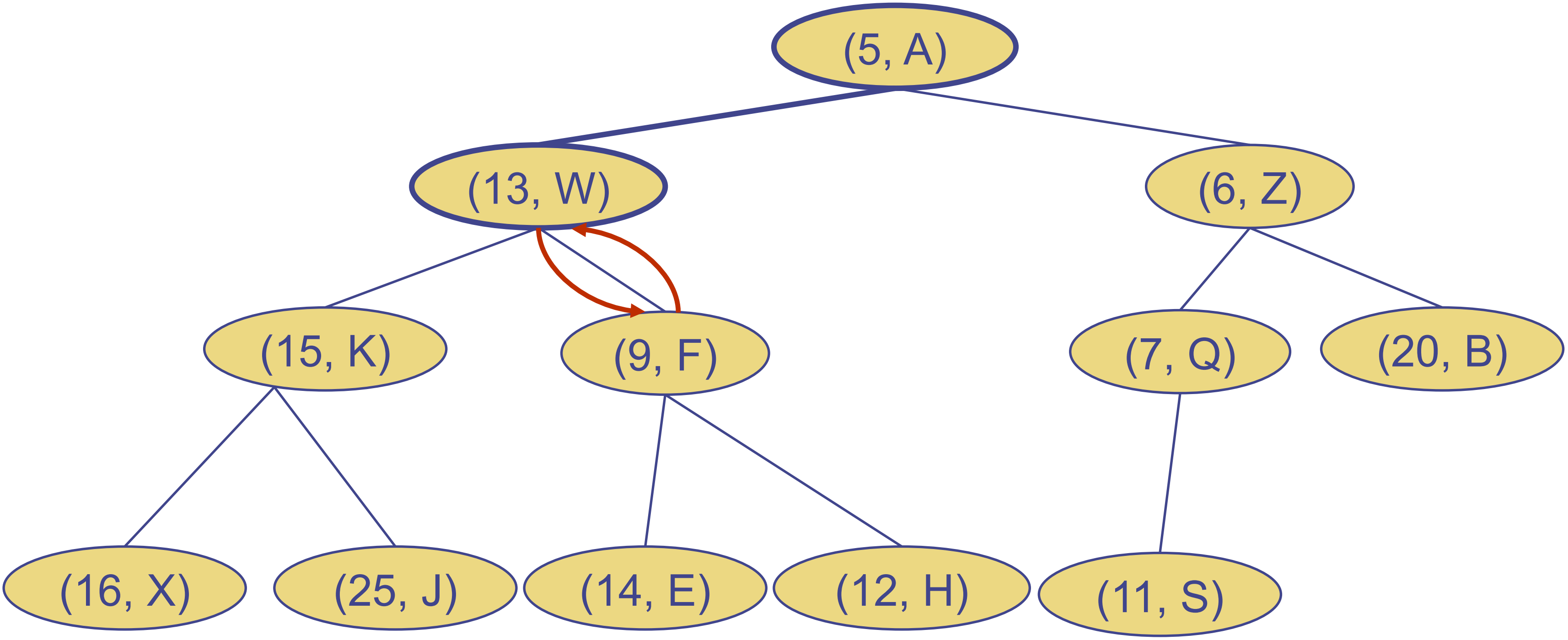


Down-Heap Bubbling – Example (cont'd)



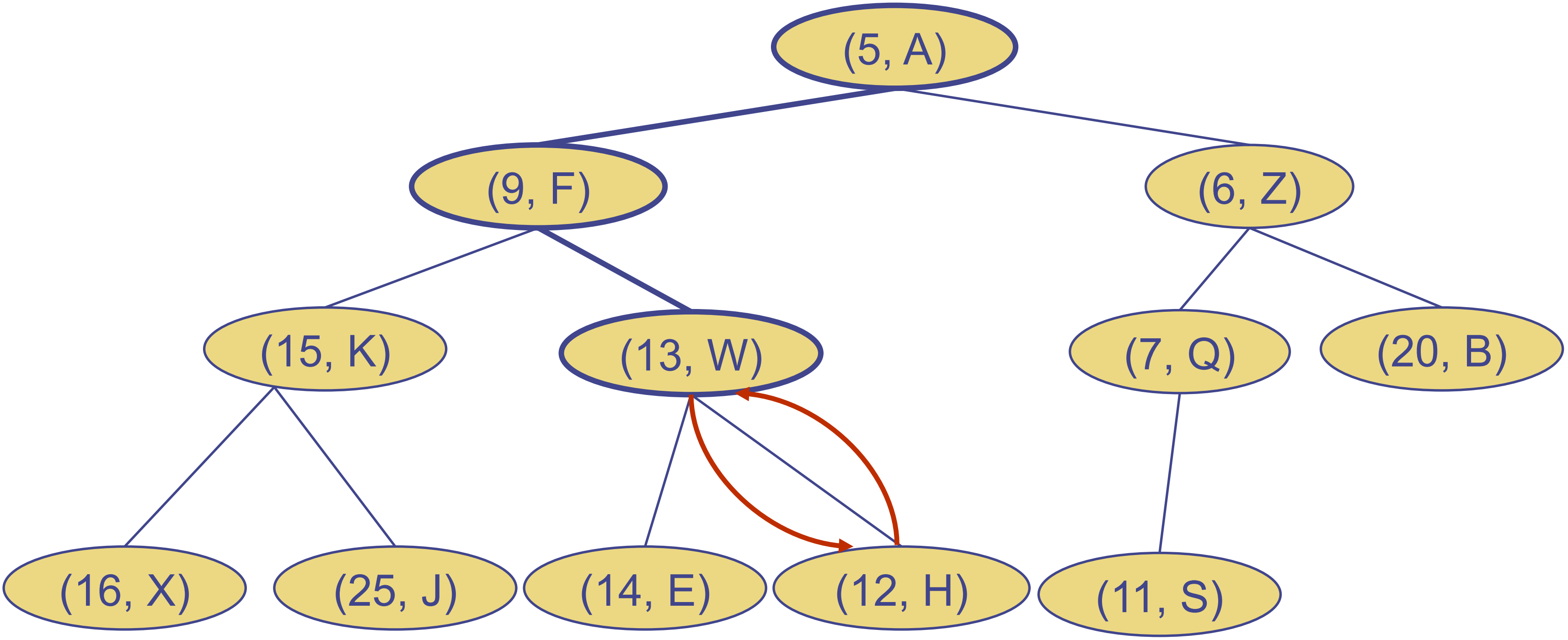
- Heap-order property not satisfied, swap root item, (13,W) with the child with minimum key, (5,A)

Down-Heap Bubbling – Example (cont'd)



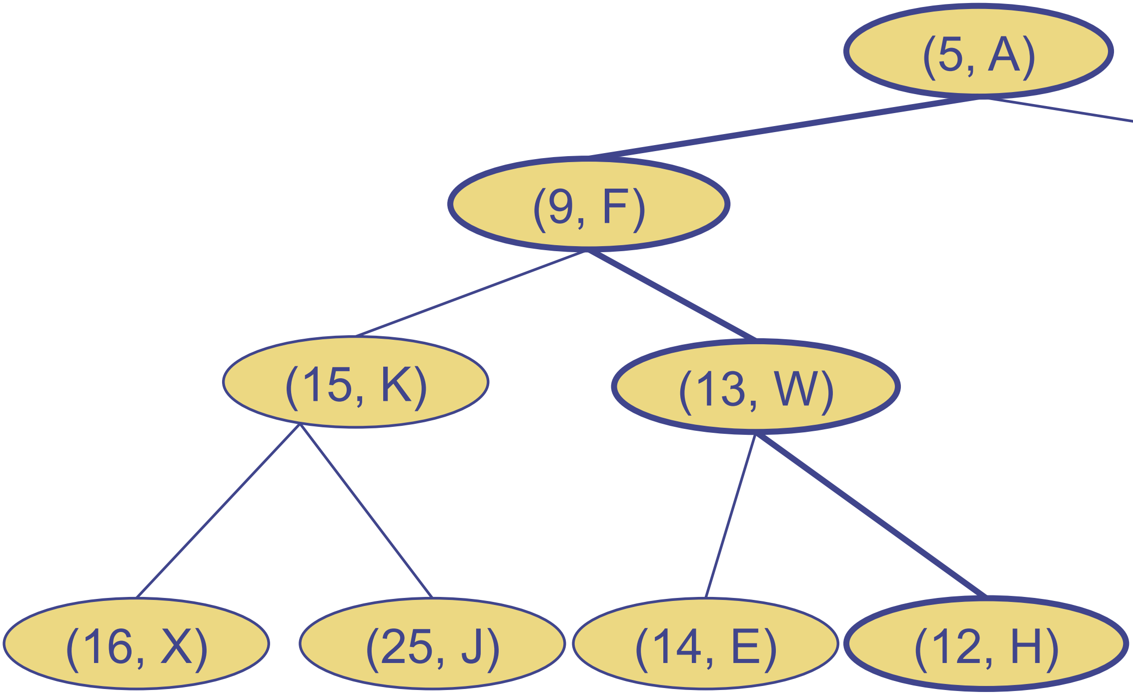
- Heap-order property not satisfied, swap the item (13,W) with the child with minimum key, (9,F)

Down-Heap Bubbling – Example (cont’d)



- Heap-order property not satisfied, swap the item (13,W) with the child with minimum key, (12,H)

Down-Heap Bubbling – Example (cont’d)



- Heap-order property is satisfied, stop

Down-Heap Bubbling – Wrap-up

- Down-heap bubbling terminates when the new item with key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a binary heap has height $h = \lfloor \log n \rfloor$, the down-heap bubbling process runs in $O(\log n)$ time

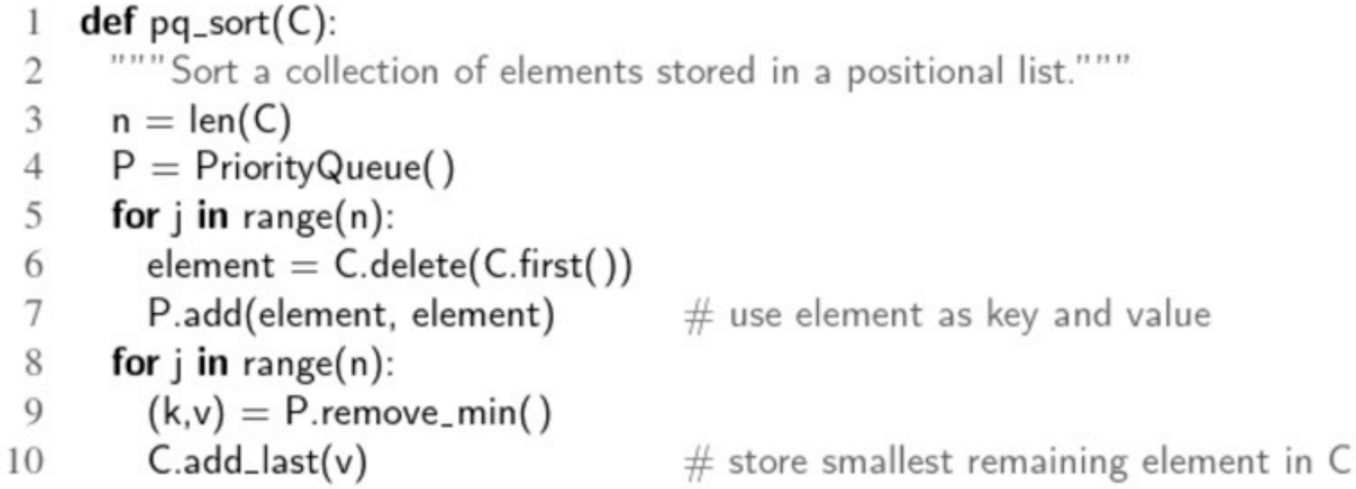
PQ implementation using a Binary Heap

- Performance
  - `P.add(k,v)` takes $O(\log n)$ time, since we have to do up-heap bubbling on the full height of the tree in the worst case
  - `P.min()` takes $O(1)$ time, since the smallest key is at the top of the heap
  - `P.remove_min()` takes $O(\log n)$ time, since we have to do down-heap bubbling on the full height of the tree in the worst case

Sorting with a Priority Queue

- A priority queue can be used to sort a collection of items with comparable keys

1. Insert the items one by one using the add() operation
2. Remove the elements in sorted order by calling remove_min() on the priority queue until all items have been removed



```
def pq_sort(C):
  """Sort a collection of elements stored in a positional list."""
  n = len(C)
  P = PriorityQueue()
  for j in range(n):
    element = C.delete(C.first())
    P.add(element, element)          # use element as key and value
  for j in range(n):
    (k,v) = P.remove_min()
    C.add_last(v)                    # store smallest remaining element in C
```

Heap Sort

- Variant of pq_sort() where the priority queue is implemented with a heap
- Running time
  - Inserting n elements into the priority queue with n `add()` operations takes ? time
  - Removing n elements from the constructed priority queue using n `remove_min()` operations takes ? time
  - The heap sort algorithm sorts a collection C of n elements in ? time, assuming two elements of C can be compared in $O(1)$ time
  - Space usage of heap sort is $O(n)$

Heap Sort

- Variant of pq_sort() where the priority queue is implemented with a heap
- Running time
  - Inserting n elements into the priority queue with n `add()` operations takes $O(n \log n)$ time - but can be improved to $O(n)$
  - Removing n elements from the constructed priority queue using n `remove_min()` operations takes $O(n \log n)$ time
  - The heap sort algorithm sorts a collection C of n elements in $O(n \log n)$ time, assuming two elements of C can be compared in $O(1)$ time
  - Space usage of heap sort is $O(n)$

Types of Heaps

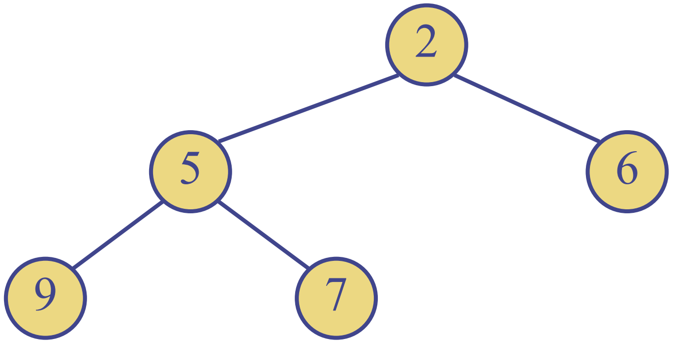
- Min-heap, presented – the minimum element is at the top
- Max-heap
  - the maximum element is at the top
  - the key at each position is at least as large as its children
  - `add(k,v)` – inserts the item (k, v) into the heap
  - `remove_max()` – removes and retrieves the maximum element of the heap
  - `max()` – retrieve, but do not remove the maximum element of the heap

In-place Heap Sort

- When the collection to be sorted is implemented as an array-based sequence (i.e. Python list), we can reduce the space requirement
- We can use a portion of the list to store the heap, and avoid the auxiliary heap data structure; use a max-heap
- At any time during execution:
  - use the leftmost portion of the array, up to the index $i - 1$, to store the items of the heap
  - use the right portion of the array, from i to $n - 1$, to store the elements of the sequence
- The first i elements of the array (indices $0, \ldots, i - 1$) provide the array-list representation of the heap

Array-Based Heap Implementation

- A binary heap with n keys can be represented by means of an array of length n
- For the node at position i
  - The left child is at $2i + 1$
  - The right child is at $2i + 2$
- Links between nodes are not explicitly stored
- The `add()` operation corresponds to inserting at position $n + 1$
- The `remove_min()` operation corresponds to removing at position n



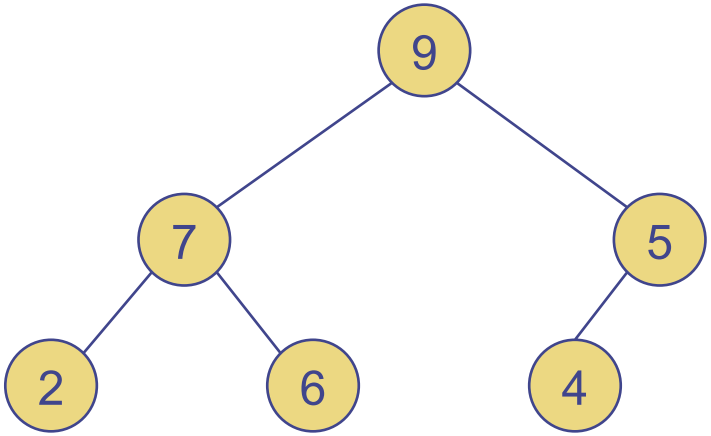
2	5	6	9	7
0	1	2	3	4

In-place Heap Sort (cont'd)

- In the first phase of the algorithm, start with an empty heap, and move the boundary between the heap and the sequence from left to right, one step at a time
- In step i, for $i = 1, \ldots, n$, we expand the heap by adding the element at index $i - 1$
- In the second phase of the algorithm, we start with an empty sequence and move the boundary between the heap and the sequence from right to left, one step at a time
- At step i, for $i = 1, \ldots, n$, we remove the maximum element from the heap and store it at index $n - i$

In-place Heap Sort – example, phase 2

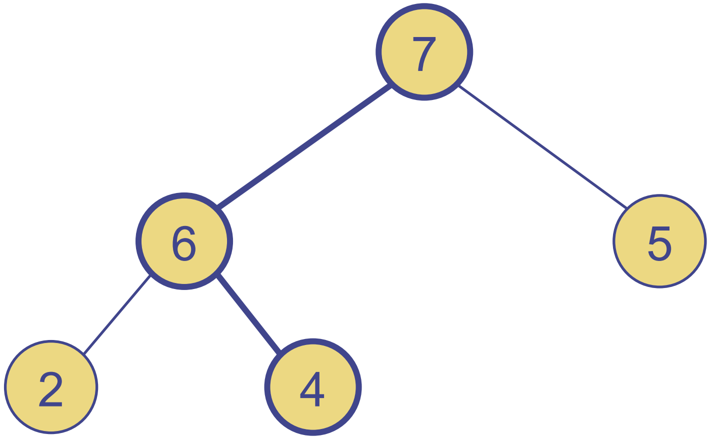
9	7	5	2	6	4



- Sequence (yellow) empty
- Heap (blue) has six elements

In-place Heap Sort – example, phase 2 (cont'd)

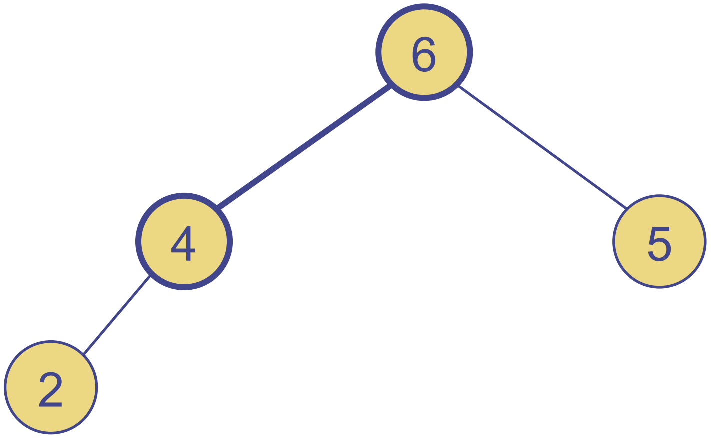
7	6	5	2	4	9



- Remove the largest element in the heap, 9; 4 moves to root
- Move the boundary one step from right to left
- Sequence has one element [9]
- Heap has five elements; down-heap bubbling to move 4 to its place

In-place Heap Sort - example, phase 2 (cont’d)

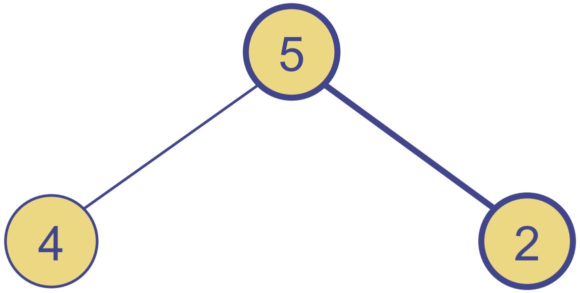
6	4	5	2	7	9



- Removed the largest element in the heap, 7; 4 moves to root
- Move the boundary one step from right to left
- Sequence has two elements [7, 9]
- Heap has four elements; down-heap bubbling to move 4 to its position

In-place Heap Sort - example, phase 2 (cont’d)

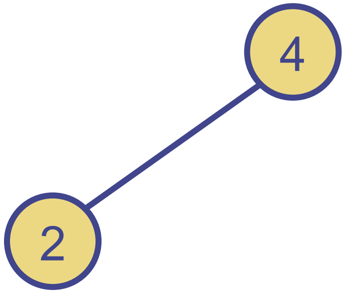
5	4	2	6	7	9



- Remove the largest element 6; move 2 to the root
- Move the boundary one step from right to left
- Sequence has three elements [6,7,9]
- Heap has three elements; down-heap bubbling to move 2 to its position

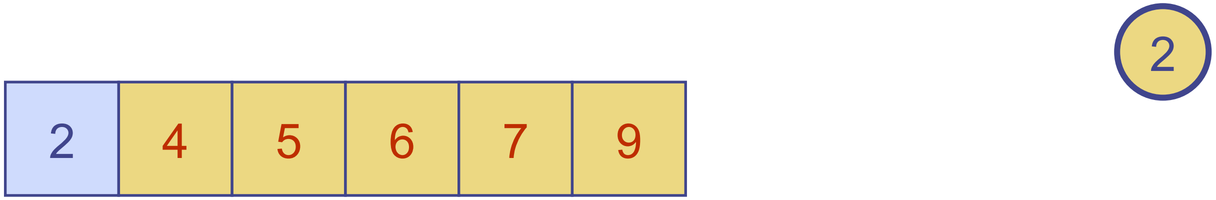
In-place Heap Sort - example, phase 2 (cont’d)

4	2	5	6	7	9



- Remove the largest element, 5; 2 goes to the root
- Move the boundary one step from right to left
- Sequence has four elements [5, 6,7,9]
- Heap has two elements; down-heap bubbling to move 2 to its position

In-place Heap Sort - example, phase 2 (cont’d)



2	4	5	6	7	9

- Remove the largest element 4; 2 moves to the root
- Move the boundary one step from right to left
- Sequence has five elements [4,5,6,7,9]
- Heap has one element

In-place Heap Sort - example, phase 2 (cont’d)

2	4	5	6	7	9

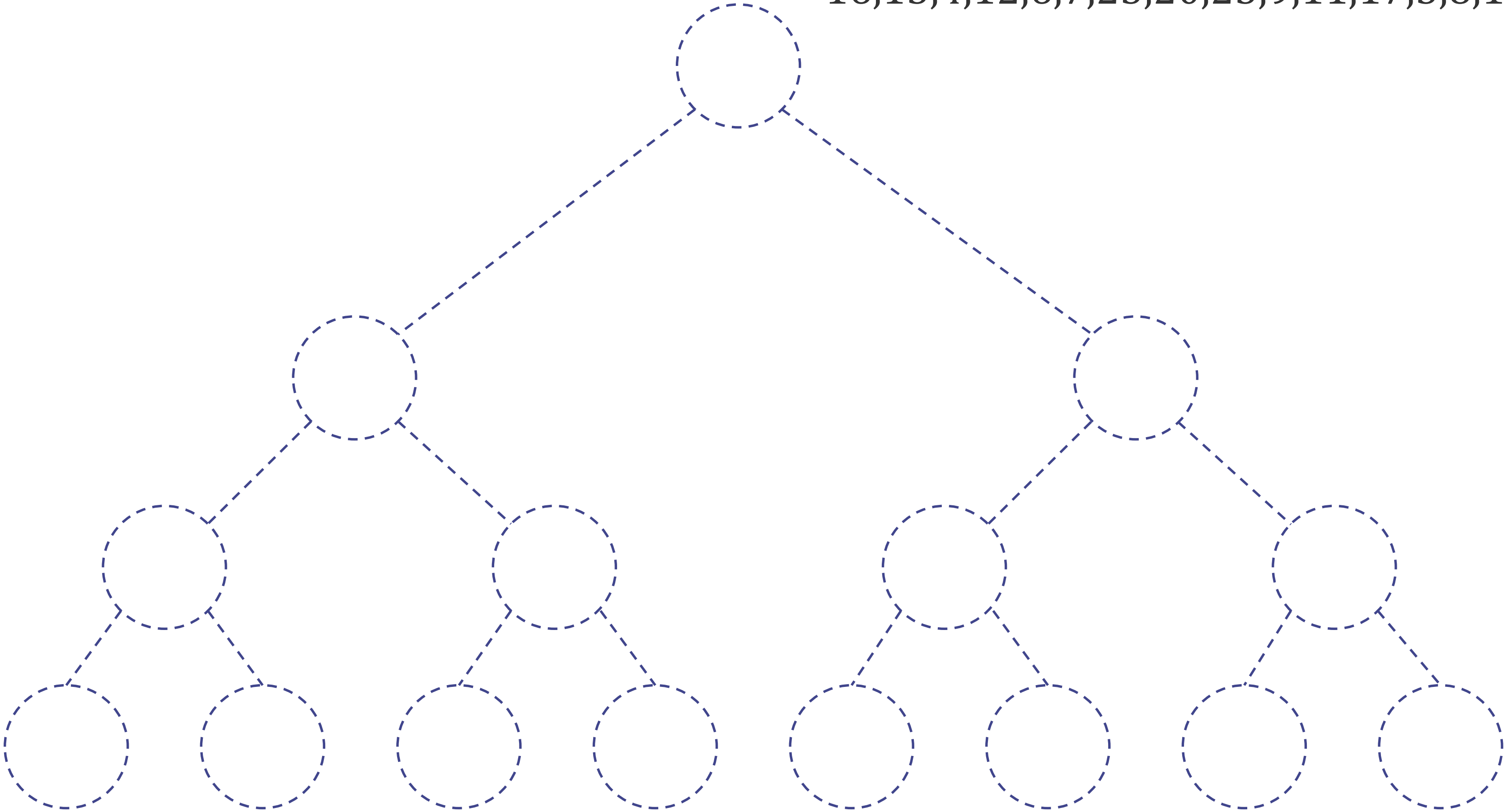
- Remove last element from the heap, 2
- Move the boundary one step from right to left
- Sequence has six elements [2,4,5,6,7,9]
- Heap is empty

Bottom-Up Heap Construction

- Starting with an initially empty heap and having n successive calls to the add() operation leads to a $O(n \log n)$ running time in the worst case
- However, if all the values are known in advance, there is an alternative bottom-up construction that runs in $O(n)$ time
- For simplicity, assume that we are constructing a heap with n items, such that $n = 2^{h+1} - 1$ – that is, the heap is a complete binary tree with every level being full
- Typically the function is called `heapify()`

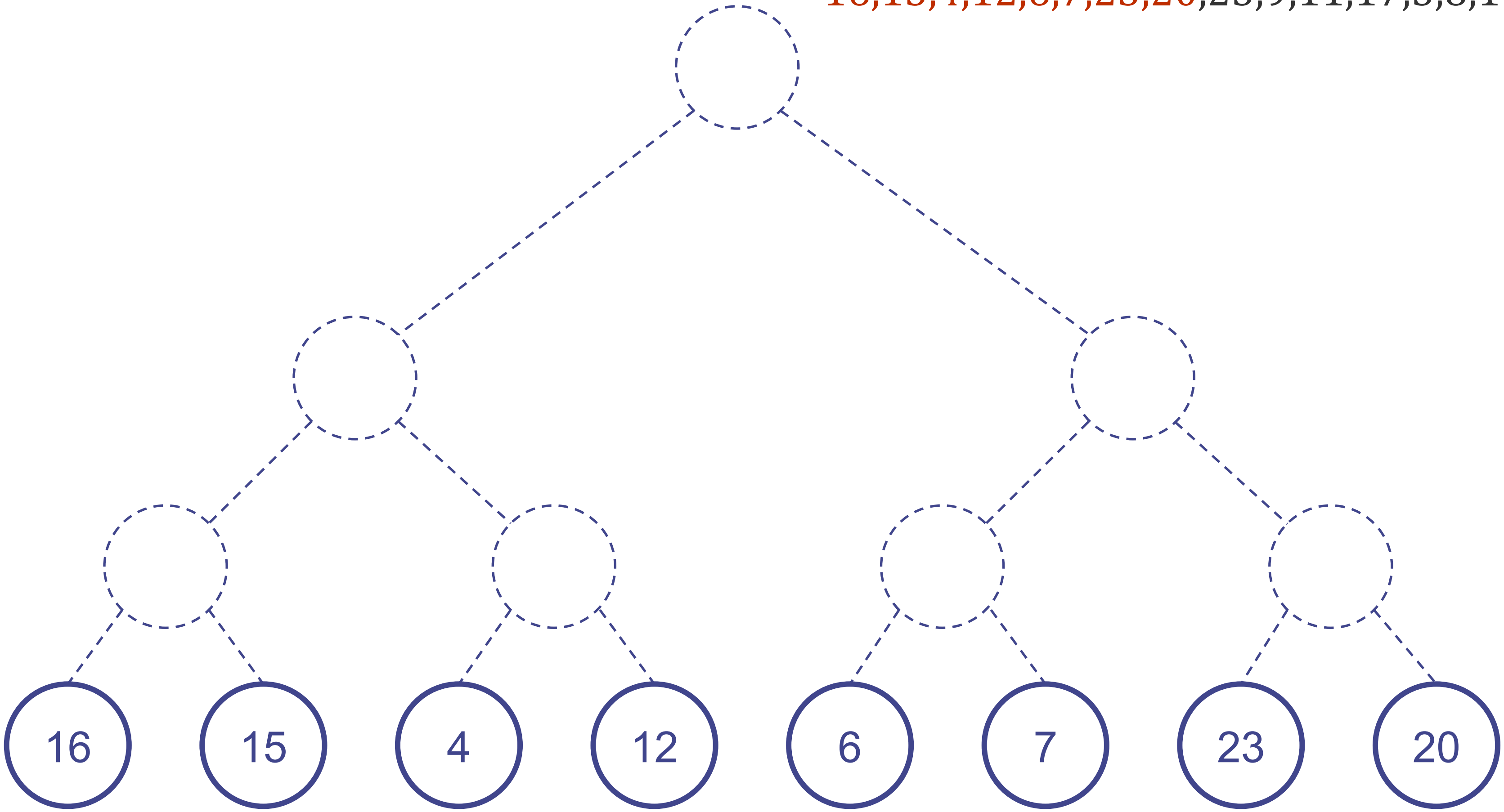
Bottom-Up Heap Construction - example

16,15,4,12,6,7,23,20,25,9,11,17,5,8,14



Bottom-Up Heap Construction – example (cont’d)

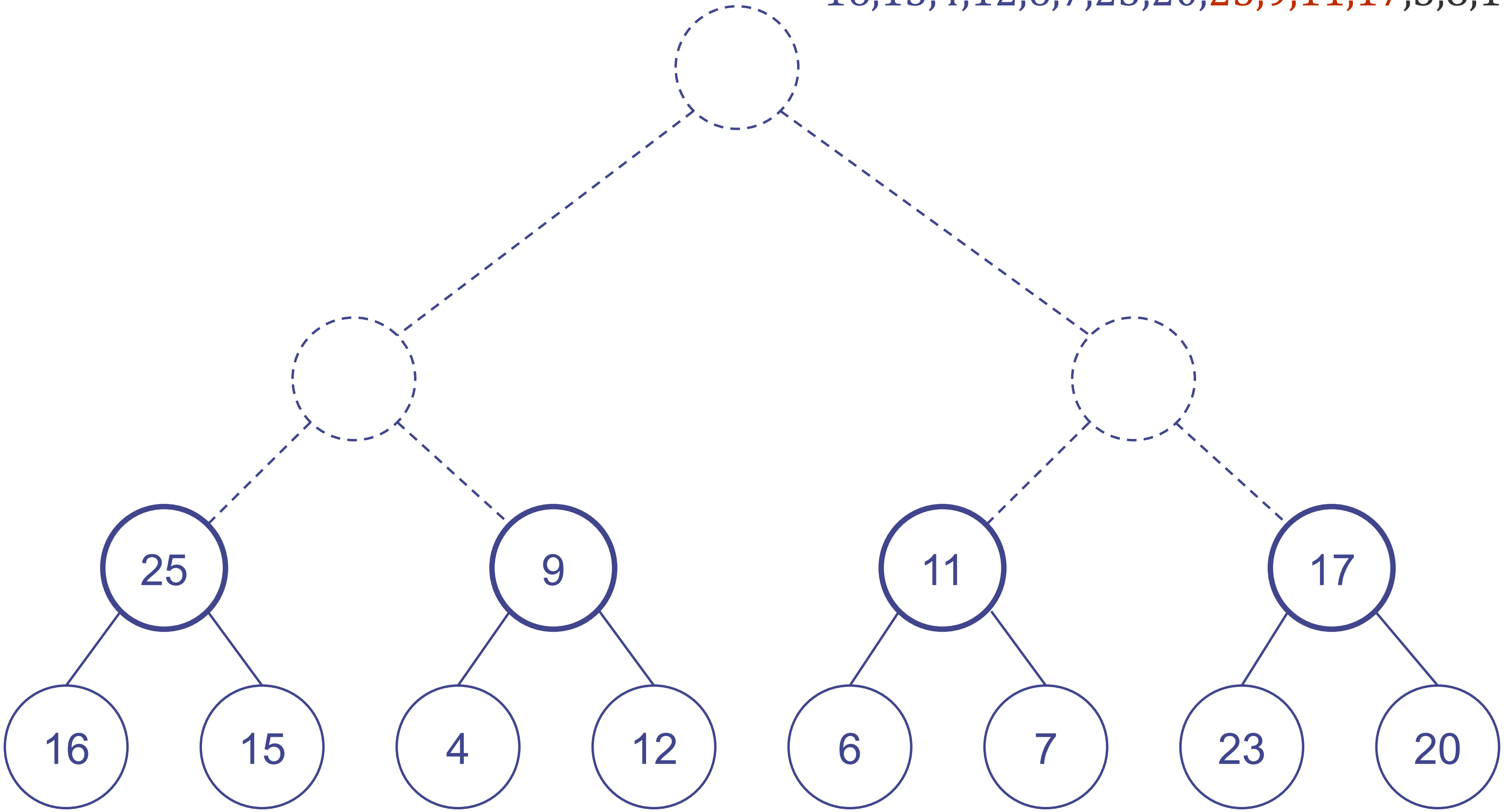
16,15,4,12,6,7,23,20,25,9,11,17,5,8,14



- Step 1: construct $(n+1)/2$ elementary heaps storing one element each

Bottom-Up Heap Construction – example (cont'd)

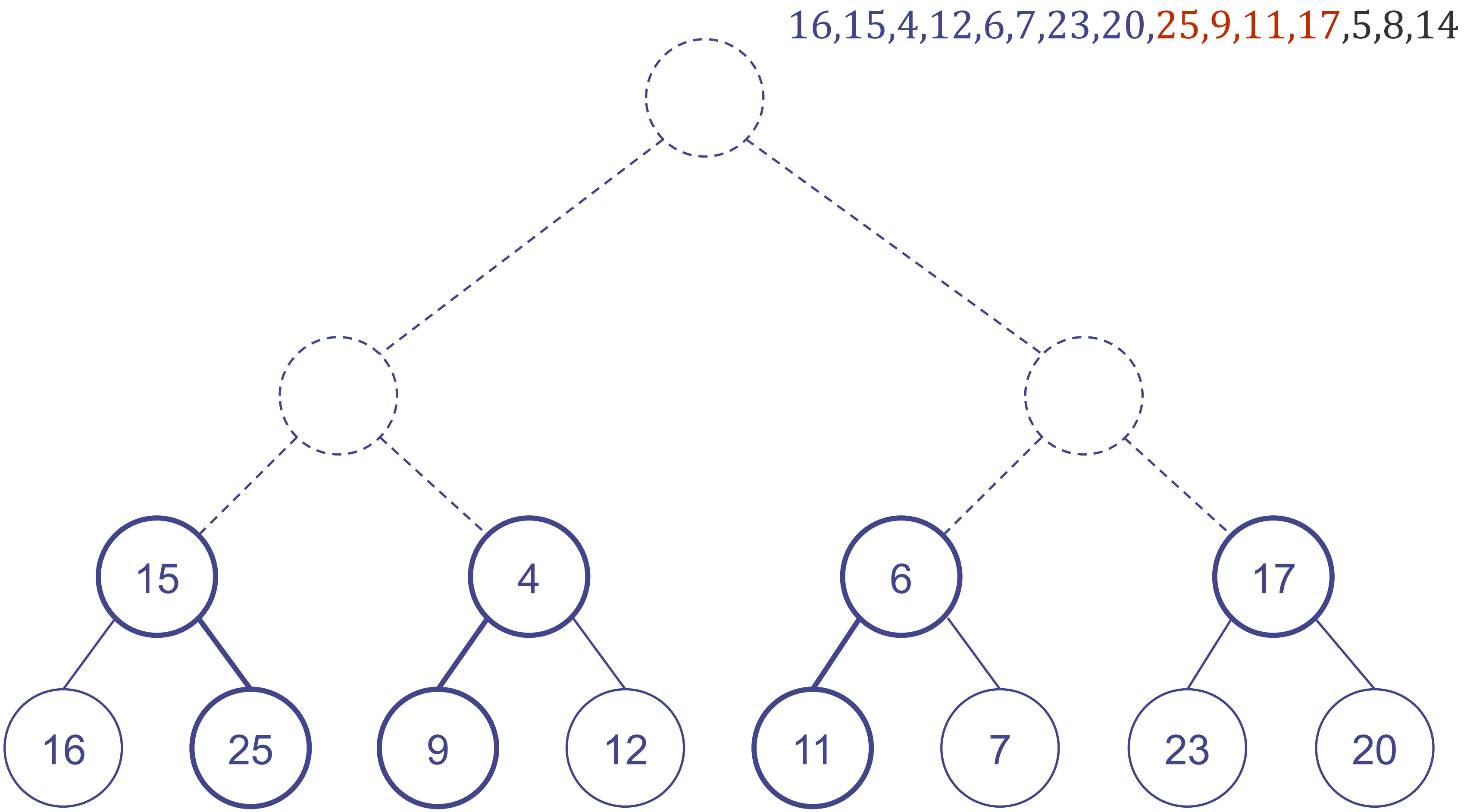
16,15,4,12,6,7,23,20,25,9,11,17,5,8,14



- Step 2: construct $(n+1)/4$ heaps, each storing three elements

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

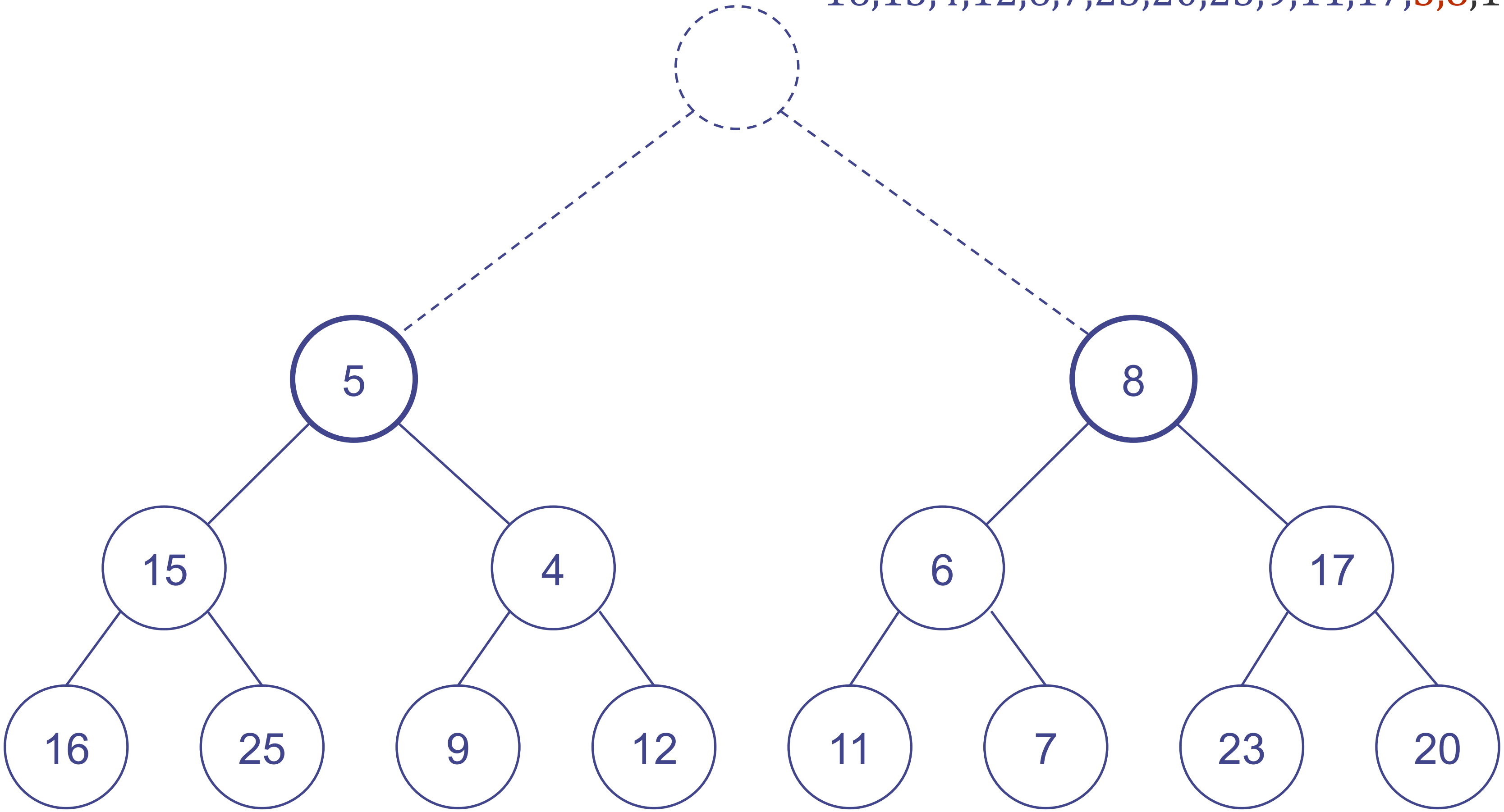
Bottom-Up Heap Construction – example (cont'd)



- Step 2: the new entries might have to be swapped with their children to preserve the heap-order property

Bottom-Up Heap Construction – example (cont'd)

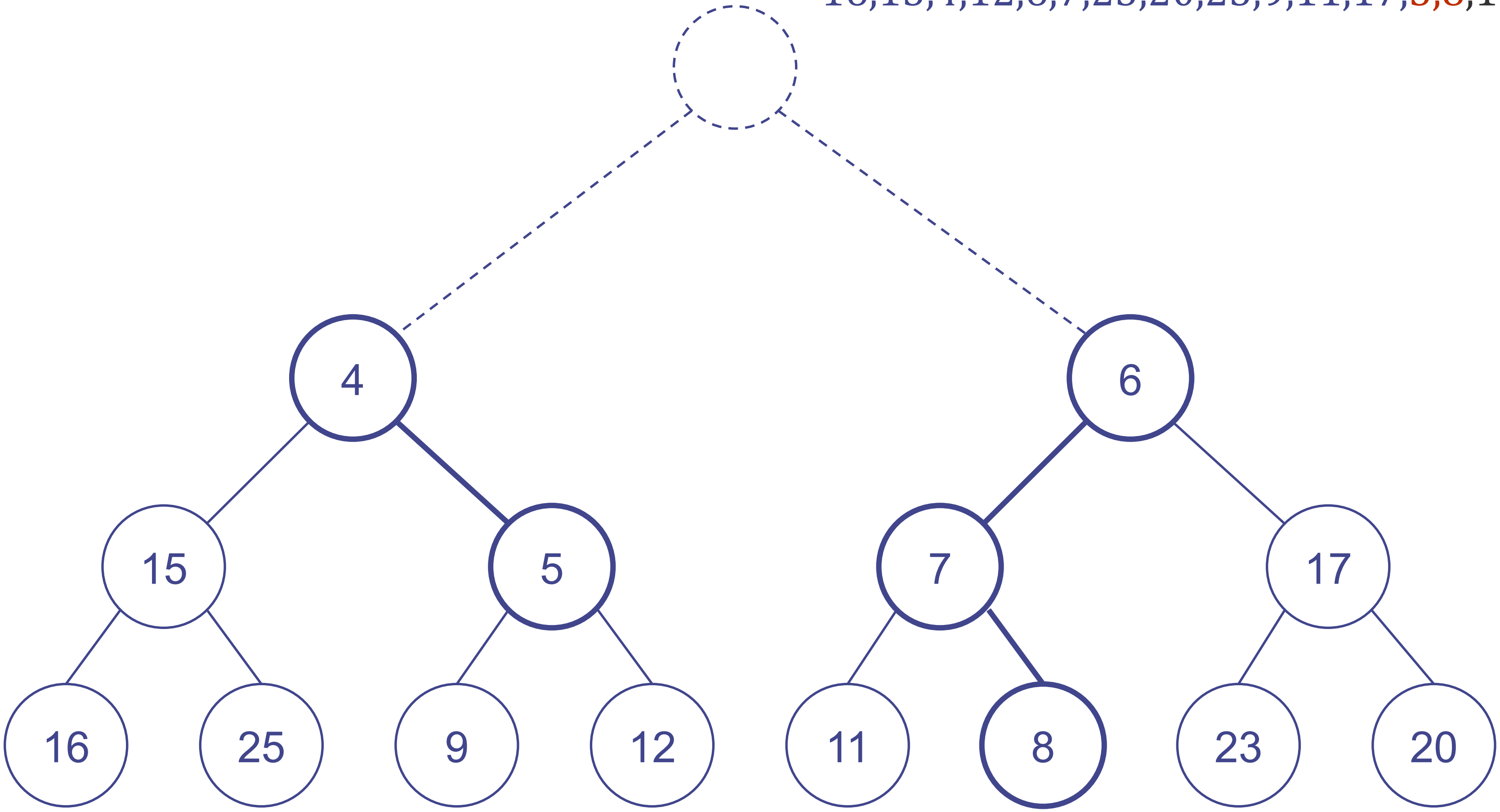
16,15,4,12,6,7,23,20,25,9,11,17,5,8,14



- Step 3: construct $(n+1)/8$ heaps, each containing 7 elements

Bottom-Up Heap Construction – example (cont'd)

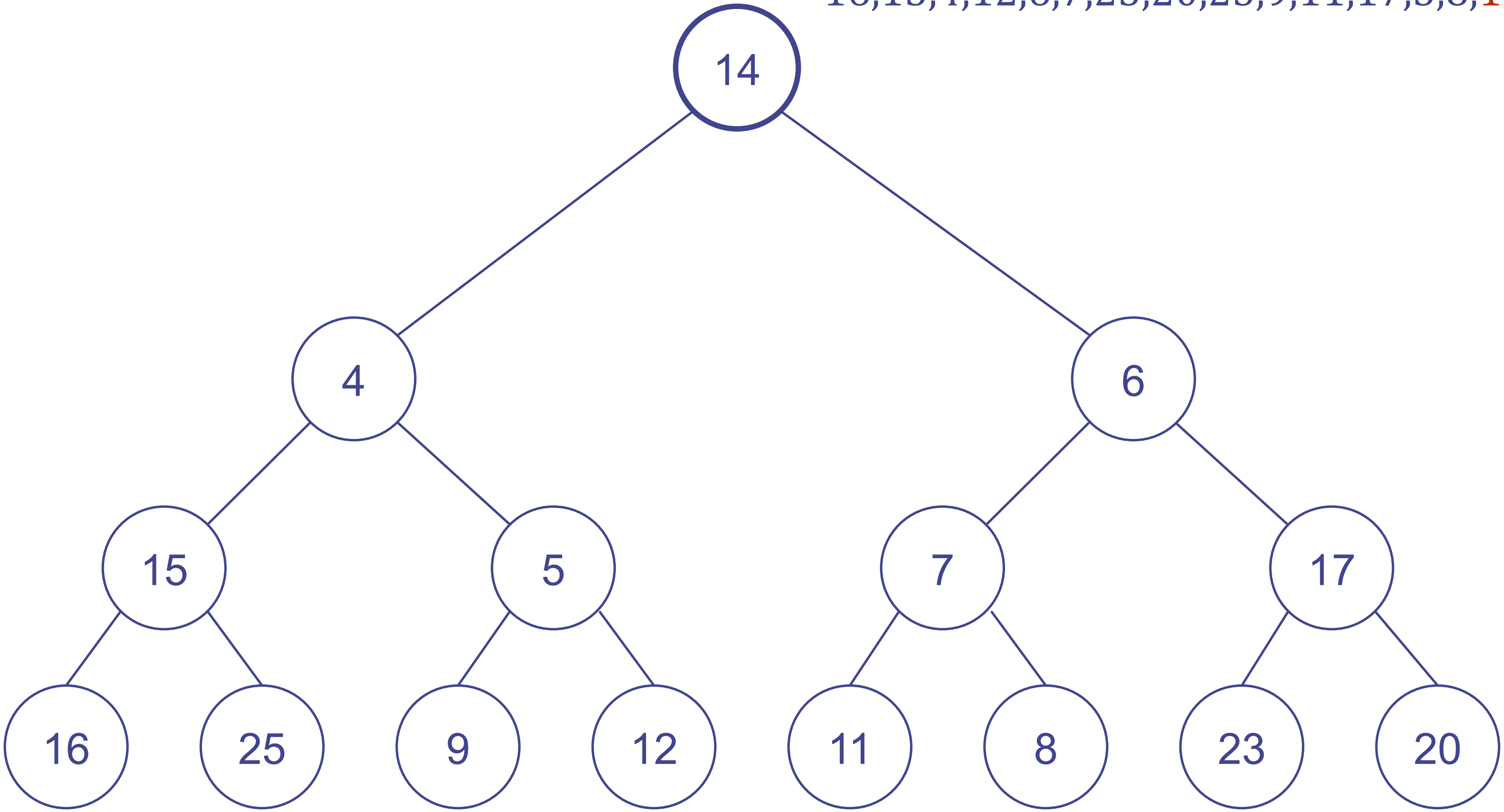
16,15,4,12,6,7,23,20,25,9,11,17,5,8,14



- Step 3: down-heap bubbling of the new elements

Bottom-Up Heap Construction – example (cont'd)

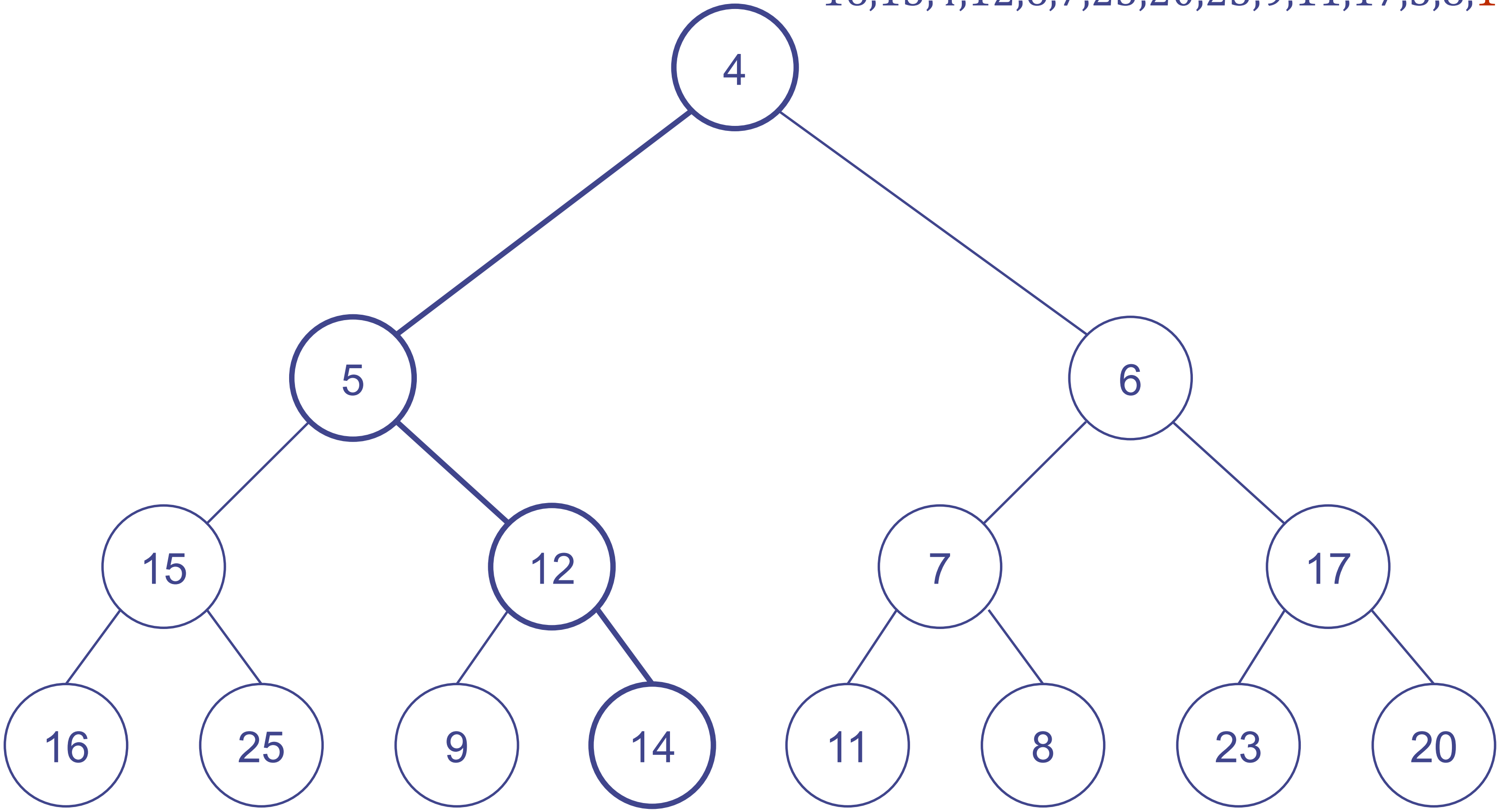
16,15,4,12,6,7,23,20,25,9,11,17,5,8,14



- Step 4: form the final heap, by adding the last entry

Bottom-Up Heap Construction – example (cont'd)

16,15,4,12,6,7,23,20,25,9,11,17,5,8,14



- Step 4: down-heap bubbling of the root element

Python’s heapq Module

- Python’s standard distribution includes the heapq module, which provides functions that allow a standard Python list to be managed as a min-heap
- n elements are stored in list cells from $L[0]$ through $L[n-1]$
- The smallest element is at the root, $L[0]$
- Operations
  - `heappush(L,e)`: push element e onto list L and restore the heap-order property
  - `heappop(L)`: pop and return the element with the smallest value from the list L, and re-establish the heap-order property
  - `heappushpop(L,e)`: push element e on the list L and then pop and return the smallest element
  - `heapify(L)`: transform an unordered list to satisfy the heap-order property in $O(n)$ time using the bottom-up construction algorithm

Sorting – Wrap-up

Algorithm	Time	Notes
insertion sort	$O(n^2)$	- in-place - slow, but good for small inputs
quick sort	$O(n \log n)$	- in-place, randomized - fastest, good for large inputs
heap sort	$O(n \log n)$	- in-place - fast, good for large inputs

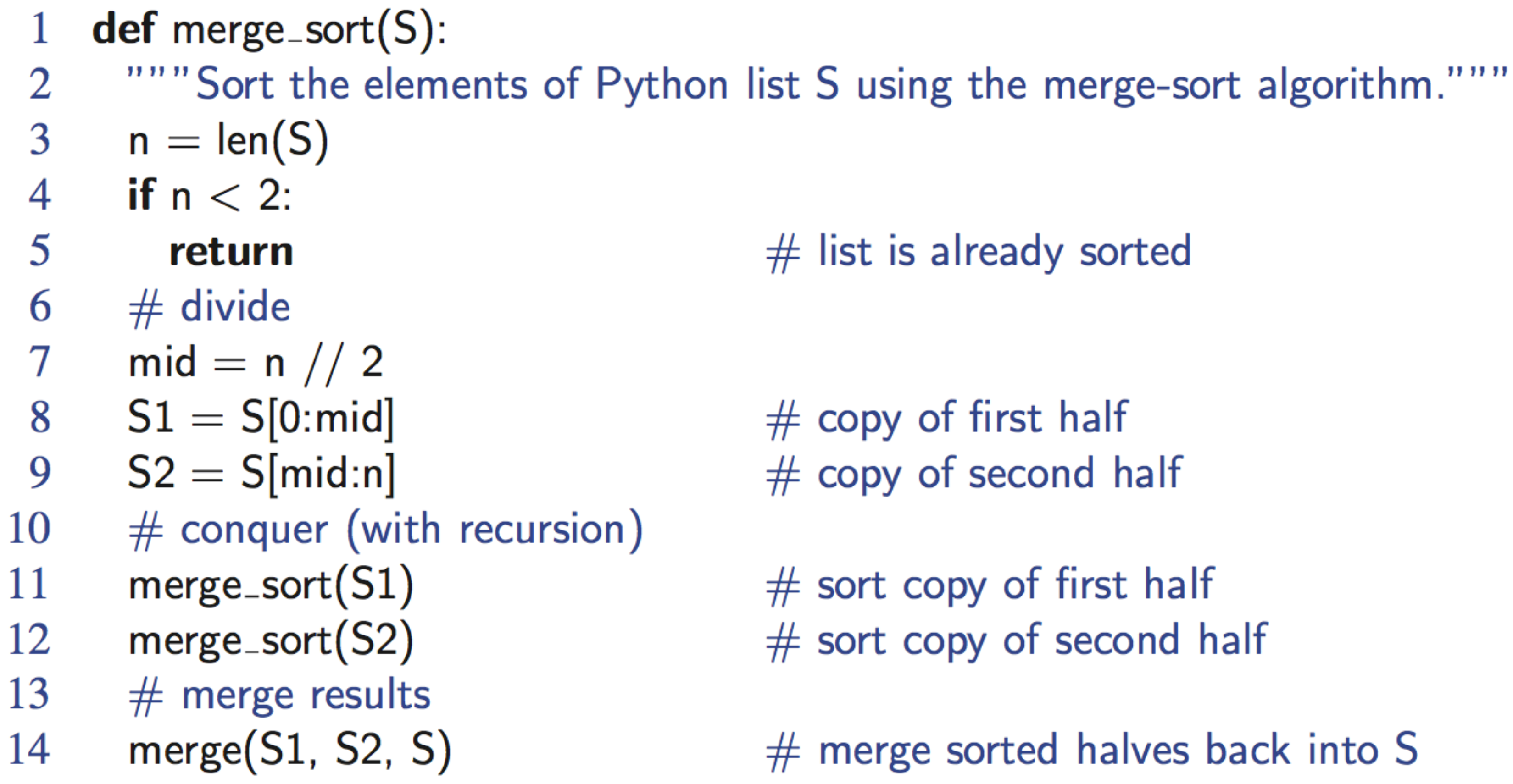
Remember Merge Sort

- Another instance of sorting algorithm based on the divide-and-conquer paradigm, just like quick sort

- To sort a sequence S with n items using merge sort:

  1. Divide: If S has zero or one element, return S immediately; it is already sorted; otherwise, if S has at least two elements, remove all the elements from S and put them in two sequences, S_1 (containing the first $\lfloor n/2 \rfloor$ elements) and S_2 (containing the remaining $\lceil n/2 \rceil$ elements)
  2. Conquer: Recursively sort sequences S_1 and S_2
  3. Combine: Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence

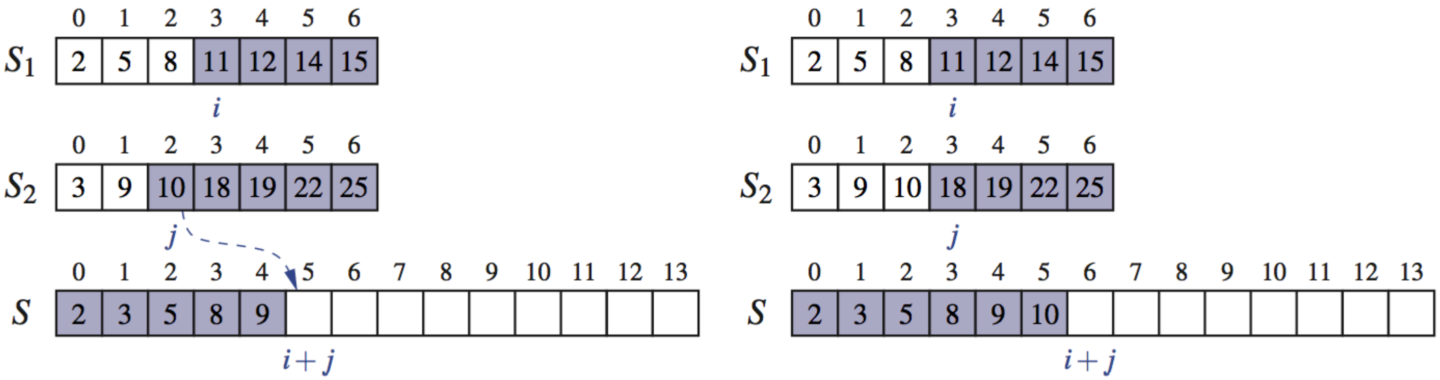
Merge Sort - Algorithm



```
def merge_sort(S):
  """Sort the elements of Python list S using the merge-sort algorithm."""
  n = len(S)
  if n < 2:
    return                              # list is already sorted
  # divide
  mid = n // 2
  S1 = S[0:mid]                         # copy of first half
  S2 = S[mid:n]                         # copy of second half
  # conquer (with recursion)
  merge_sort(S1)                        # sort copy of first half
  merge_sort(S2)                        # sort copy of second half
  # merge results
  merge(S1, S2, S)                      # merge sorted halves back into S
```

Merge Sort – Algorithm (cont’d)

```
def merge(S1, S2, S):
  """Merge two sorted Python lists S1 and S2 into properly sized list S."""
  i = j = 0
  while i + j < len(S):
    if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
      S[i+j] = S1[i]              # copy ith element of S1 as next item of S
      i += 1
    else:
      S[i+j] = S2[j]              # copy jth element of S2 as next item of S
      j += 1
```



Sorting – Wrap-up

Algorithm	Time	Notes
insertion sort	$O(n^2)$	- in-place - slow, but good for small inputs
quick sort	$O(n \log n)$	- in-place, randomized - fastest, good for large inputs
heap sort	$O(n \log n)$	- in-place - fast, good for large inputs
merge sort	$O(n \log n)$	- fast, sequential data access, good for very large datasets

Stable Sorting

- When sorting key-value pairs, an important issue is how are equal keys handled

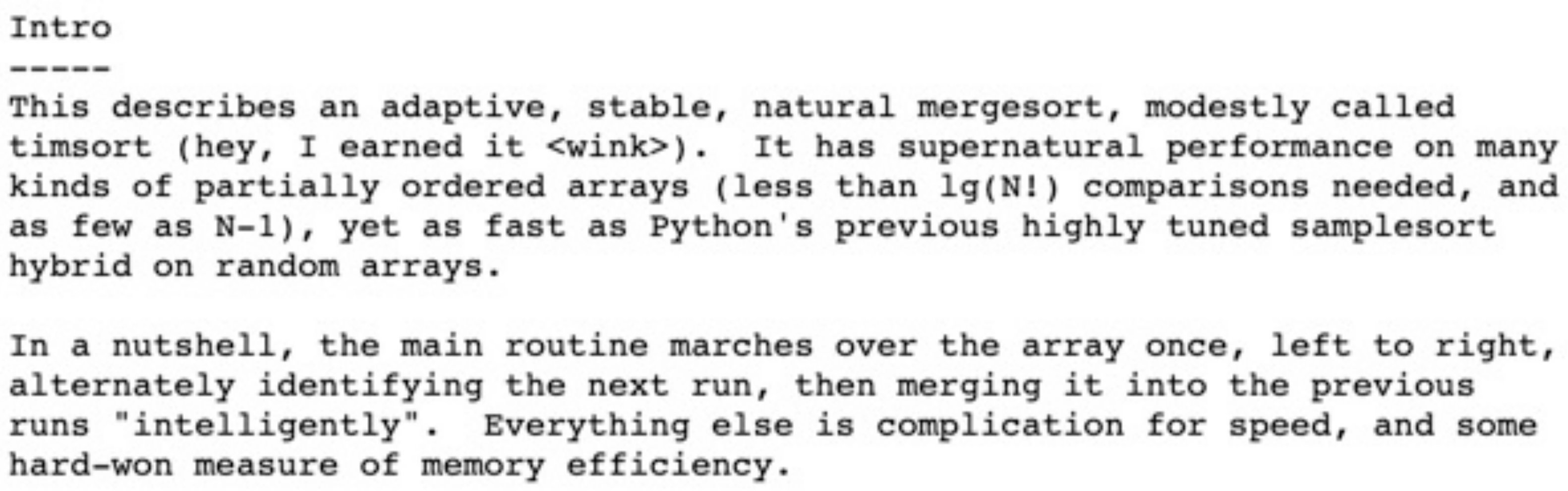
- Given the sequence $S = ((k_0, v_0), \ldots, (k_{n-1}, v_{n-1}))$, we say that a sorting algorithm is stable if, for any two entries (k_i, v_i) and (k_j, v_j) such that $k_i = k_j$ and (k_i, v_i) precedes (k_j, v_j) in S before sorting, the entry (k_i, v_i) will also precede (k_j, v_j) after sorting

Sorting – Wrap-up

Algorithm	Time	Notes
insertion sort	$O(n^2)$	- in-place - slow, but good for small inputs - stable
quick sort	$O(n \log n)$	- in-place, randomized - fastest, good for large inputs - not stable
heap sort	$O(n \log n)$	- in-place - fast, good for large inputs - not stable
merge sort	$O(n \log n)$	- not in-place - fast, sequential data access, good for very large datasets - stable

Timsort – a hybrid sorting algorithm

- Developed by Tim Peters for Python, in 2001
- Currently the default sorting algorithm in Python & Java
- Takes advantage of consecutive ordered elements – natural runs
- Collects elements into runs, then simultaneously merges the runs
- hybrid between binary insertion sort and merge sort



```
Intro
-----
This describes an adaptive, stable, natural mergesort, modestly called
timsort (hey, I earned it <wink>).  It has supernatural performance on many
kinds of partially ordered arrays (less than lg(N!) comparisons needed, and
as few as N-1), yet as fast as Python's previous highly tuned samplesort
hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right,
alternately identifying the next run, then merging it into the previous
runs "intelligently".  Everything else is complication for speed, and some
hard-won measure of memory efficiency.
```

- From https://bugs.python.org/file4451/timsort.txt

Sorting – Wrap-up

Algorithm	Time	Notes
insertion sort	$O(n^2)$	- in-place - slow, but good for small inputs - stable
quick sort	$O(n \log n)$	- in-place, randomized - fastest, good for large inputs - not stable
heap sort	$O(n \log n)$	- in-place - fast, good for large inputs - not stable
merge sort	$O(n \log n)$	- usually, not in-place - fast, sequential data access, good for very large datasets - stable
timsort	$O(n \log n)$	- in-place - stable - fast, good for large data

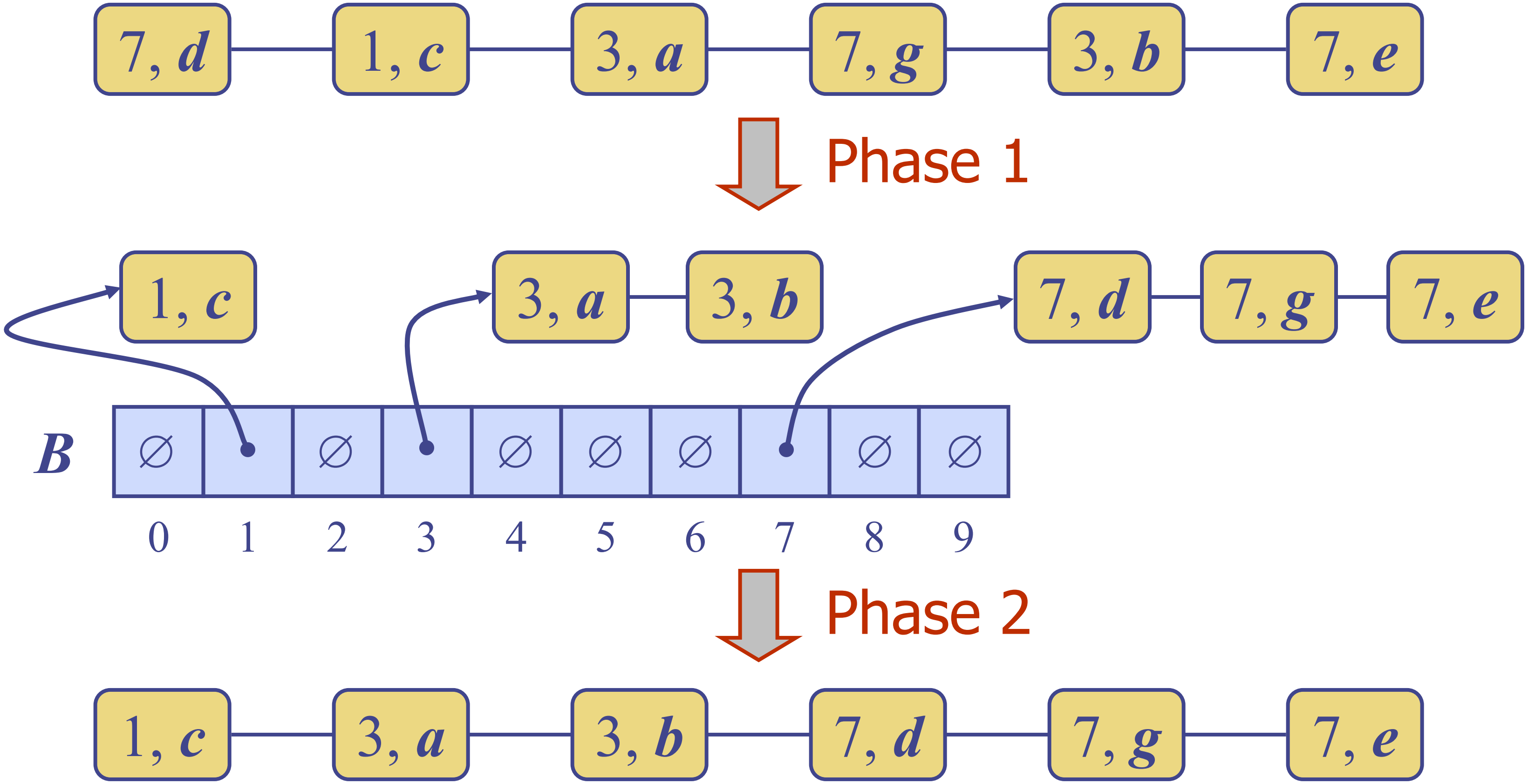
Bonus round
Bucket Sort & Radix Sort

Bucket Sort

- consider a sequence S containing n (key, value) entries
  - the keys are integers in the range $[0, N-1]$, $N \geq 2$
  - sorting S according to the keys is possible in $O(n+N)$ time
  - not using comparisons
- Phase 1
  - use keys as indices into a bucket array B, indexed from 0 to N
  - an entry with key k is placed into the bucket $B[k]$ – also an array
  - add all entries of S to B
- Phase 2
  - add the sorted entries back to S by reading the contents of each bucket of B, in order

Bucket Sort - Example

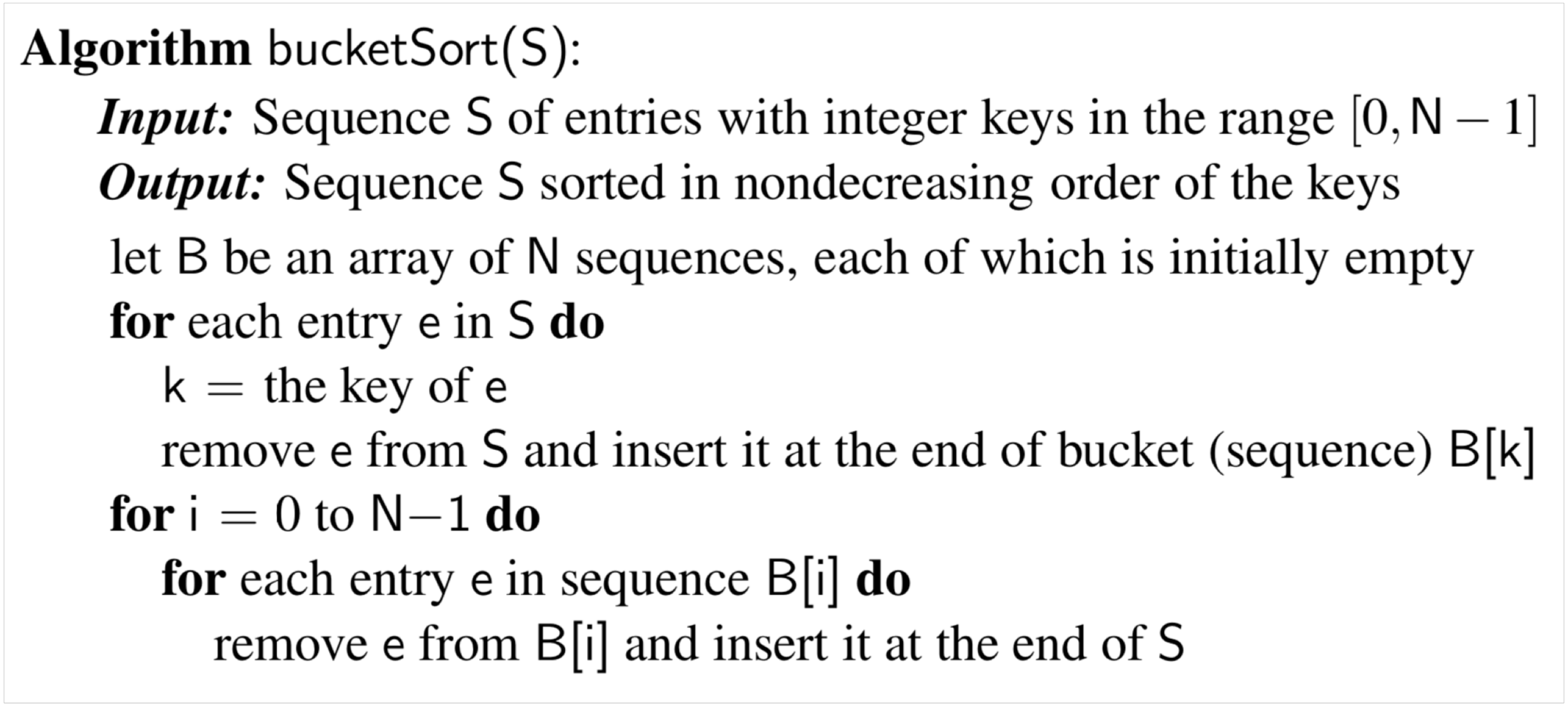
- Key range $[0, 9]$



Phase 1

Phase 2

Bucket Sort - Algorithm



```
Algorithm bucketSort(S):
    Input: Sequence S of entries with integer keys in the range $[0, N-1]$
    Output: Sequence S sorted in nondecreasing order of the keys
    let B be an array of N sequences, each of which is initially empty
    for each entry e in S do
        $k =$ the key of e
        remove e from S and insert it at the end of bucket (sequence) B[k]
    for $i = 0$ to $N-1$ do
        for each entry e in sequence B[i] do
            remove e from B[i] and insert it at the end of S
```

Bucket Sort – Running Time

- Phase 1
  - adding sequence elements to bucket array
  - $O(n)$ time
- Phase 2
  - puting back the sorted entries into S
  - $O(n + N)$ time
- Bucket sort runs in $O(n + N)$ time
- stable sort
- efficient when N is small compared to n, e.g. $N = O(n)$, $N = O(n \log n)$
- performance gets worse as N grows compared to n

Radix Sort

- suppose we want to sort entries where the keys are pairs (k, l)
- k, l integers in $[0, N-1]$
- $(k_1, l_1) < (k_2, l_2)$ if $k_1 < k_2$ or if $k_1 = k_2$ and $l_1 < l_2$
- radix sort sorts a sequence with keys that are pairs by applying the stable bucket sort algorithm on the sequence twice
  - first using the second component, l
  - then using the first component, k

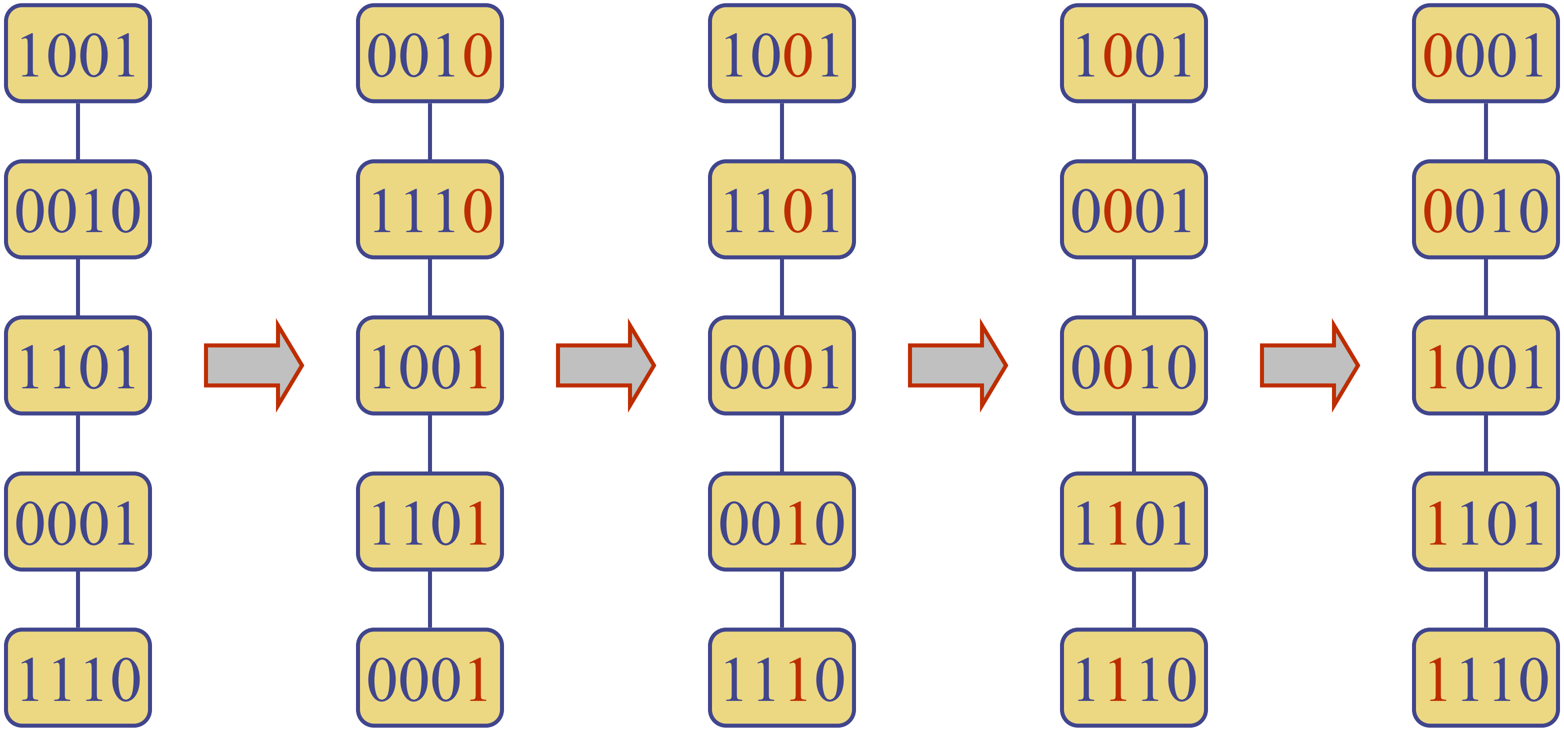
Radix Sort - Algorithm

```
Algorithm radix_sort(S, N)
    Input sequence S of d-tuples such
        that $(0, \dots, 0) \le (x_1, \dots, x_d)$ and
        $(x_1, \dots, x_d) \le (N - 1, \dots, N - 1)$
        for each tuple $(x_1, \dots, x_d)$ in S
    Output sequence S sorted in
        lexicographic order
    for $i \leftarrow d$ downto 1
        bucket_sort(S, N)
```

- radix sort runs in $O(d(n + N))$ time

Radix Sort - Example

- Sorting a sequence of 4-bit integers



1001	0010	1001	1001	0001
0010	1110	1101	0001	0010
1101	1001	0001	0010	1001
0001	1101	0010	1101	1101
1110	0001	1110	1110	1110

Thank you.

Notation

- $\lfloor x \rfloor$ indicates the floor of x, that is, the largest integer k such that $k \leq x$
- $\lceil x \rceil$ indicates the ceiling of x, that is, the smallest integer m such that $x \leq m$