Applied Algorithms

CSCI-B505 / INFO-I500

Lecture 12.

Priority Queue & Heap Data Structure

M. Oguzhan Kulekci

- Priority Queue
- Heap Data Structure
  - Insert, delete, update operations
  - Heap construction and sorting
- Some examples

Priority Queue

Hospital Emergency Queue

https://www.simplilearn.com/tutorials/data-structure-tutorial/priority-queue-in-data-structure

- Normal queue operations might not be adequate in some situations.
- Each item on the queue has a priority
- Higher priority means getting service earlier

- How to implement such priority queues ? *Arrays, linked-lists, skip-lists ….*

Heap

- How about using a **binary tree** to implement a priority queue ?

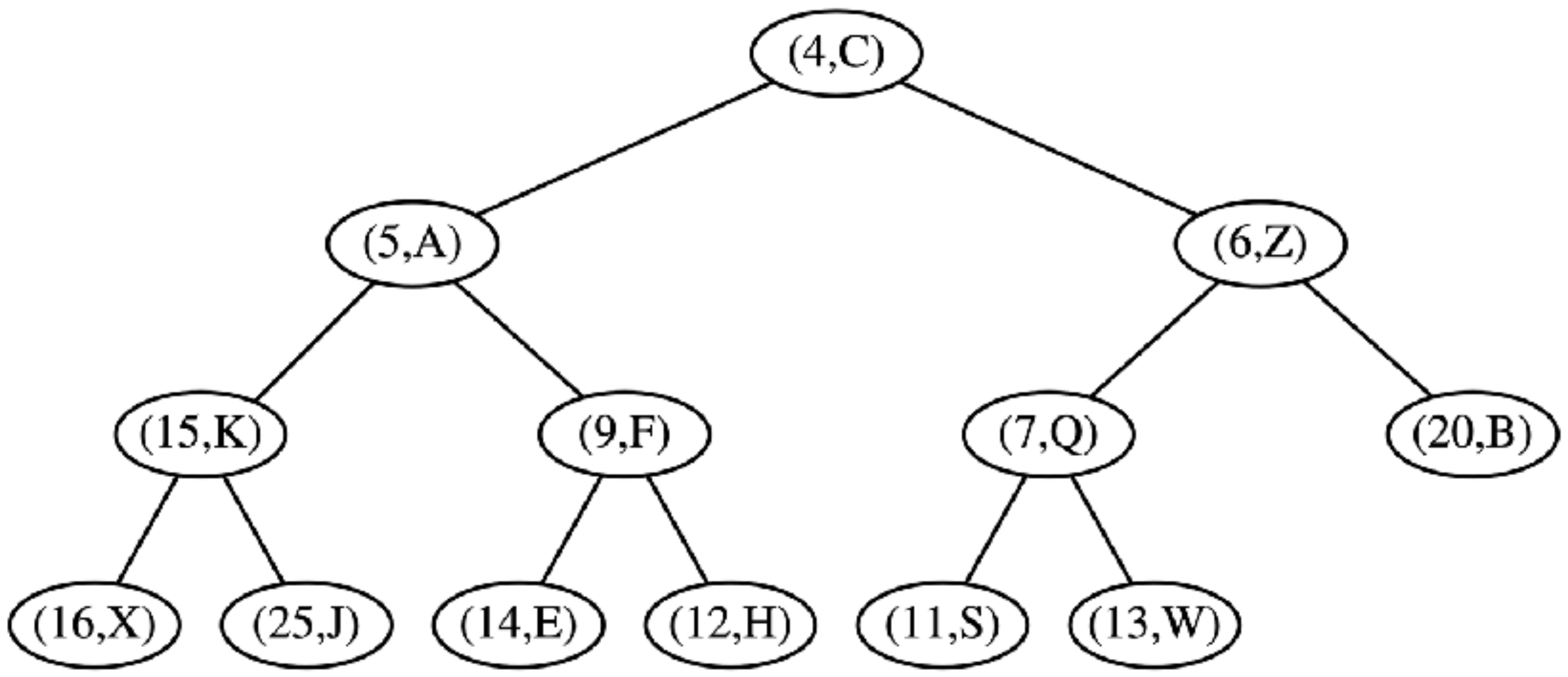


Figure 9.1: Example of a heap storing 13 entries with integer keys. The last position is the one storing entry $(13, W)$.

Min-Heap

Properties (min-heap):

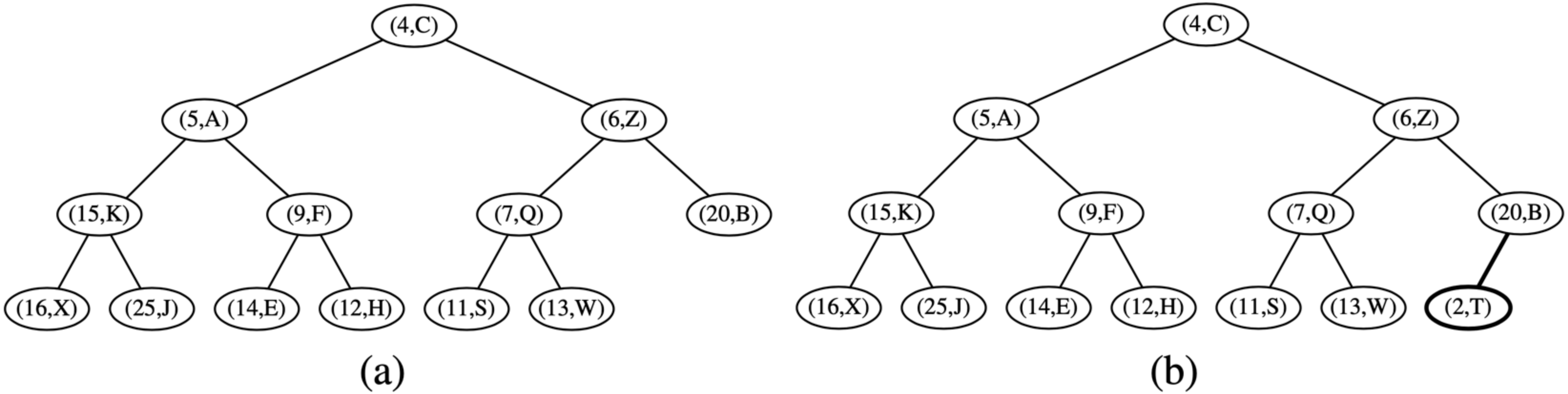
1. Every item in the tree except the root has an associated value that is ***greater*** than or equal to its parent.
2. The tree is *almost* complete, meaning all levels are fully occupied except the lowest level, which is filled from left to right

Max-Heap is smilar…

Up-Heap Bubbling

- How to add an item ?

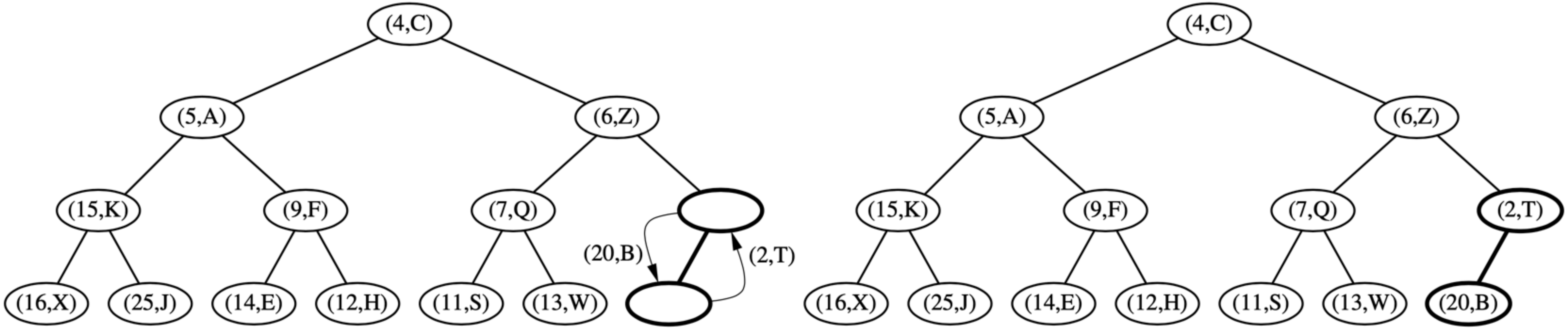
STEP 1. To keep tree **almost complete**, we initially locate the new item to the rightmost positions on the last level heap.



Up-Heap Bubbling

- How to add an item ?

STEP 2. To maintain the heap property, we check the path **up** to the root iteratively and swap the nodes if necessary.

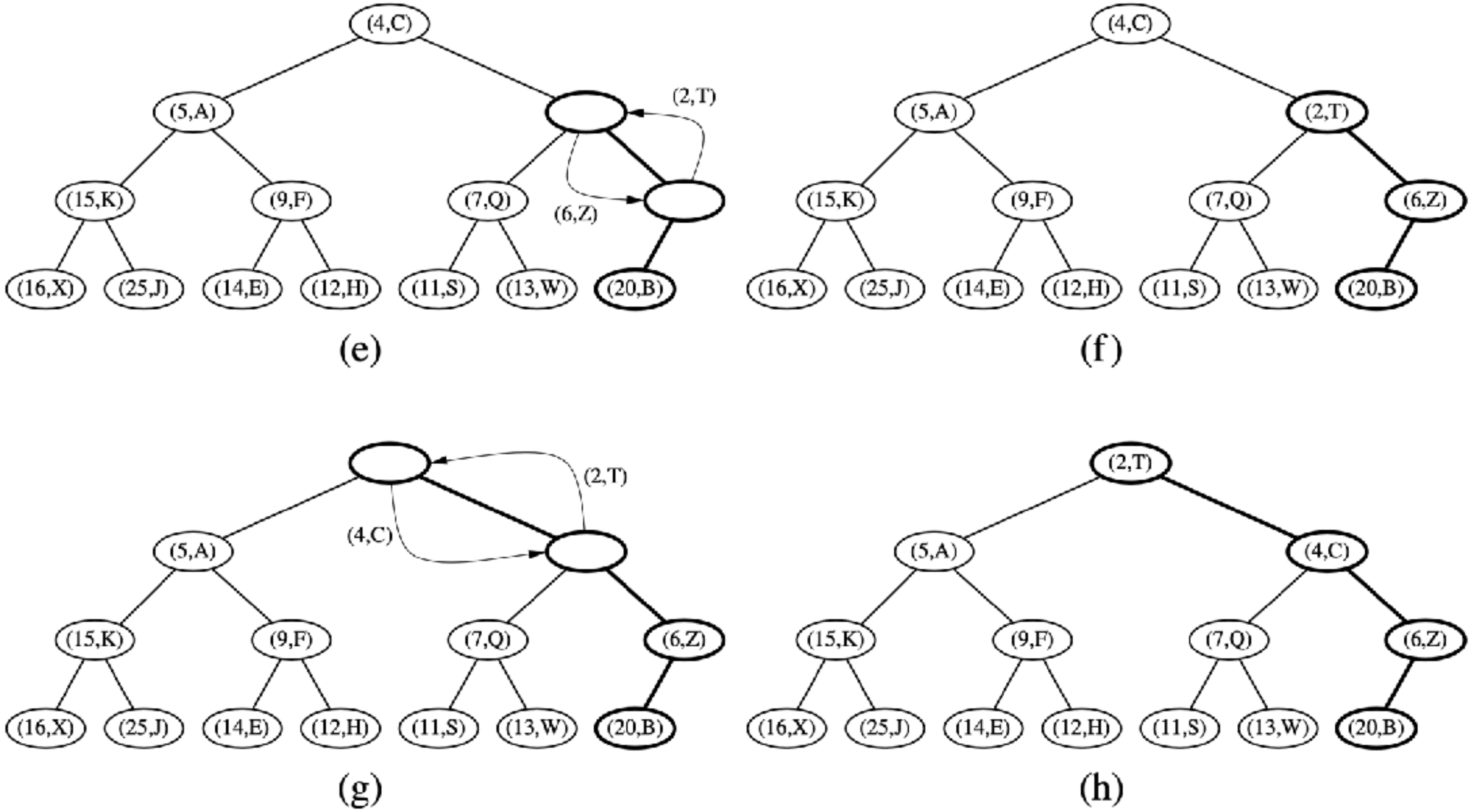


(c) (d)

Up-Heap Bubbling

- How to add an item ?

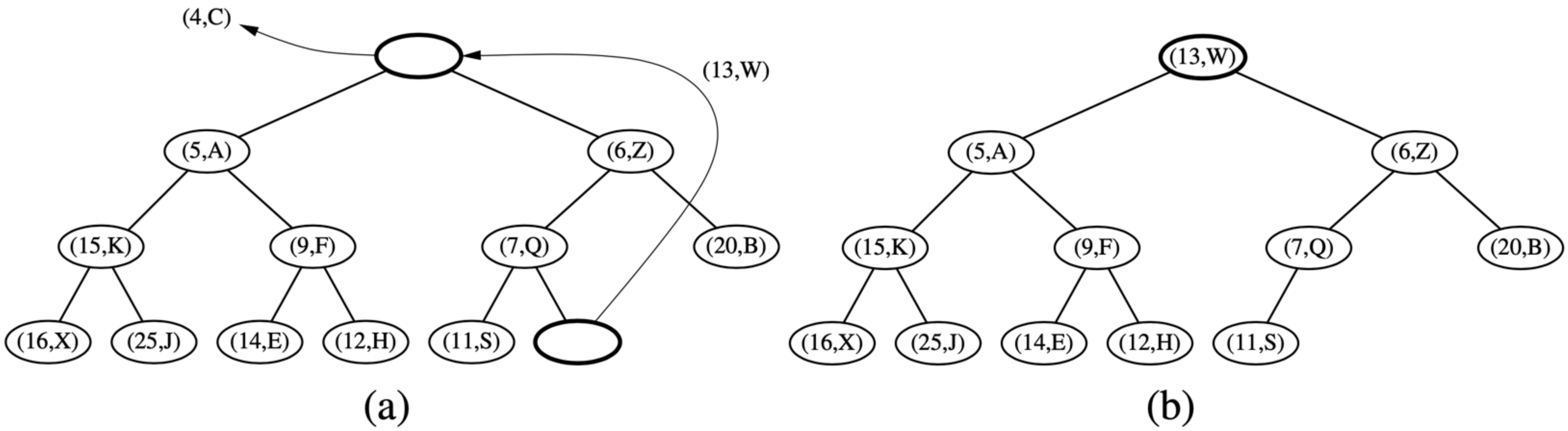
STEP 2. To maintain the heap property, we check the path **up** to the root iteratively and swap the nodes if necessary.



Down-Heap Bubbling

- How to remove the root item, which is the highest priority?

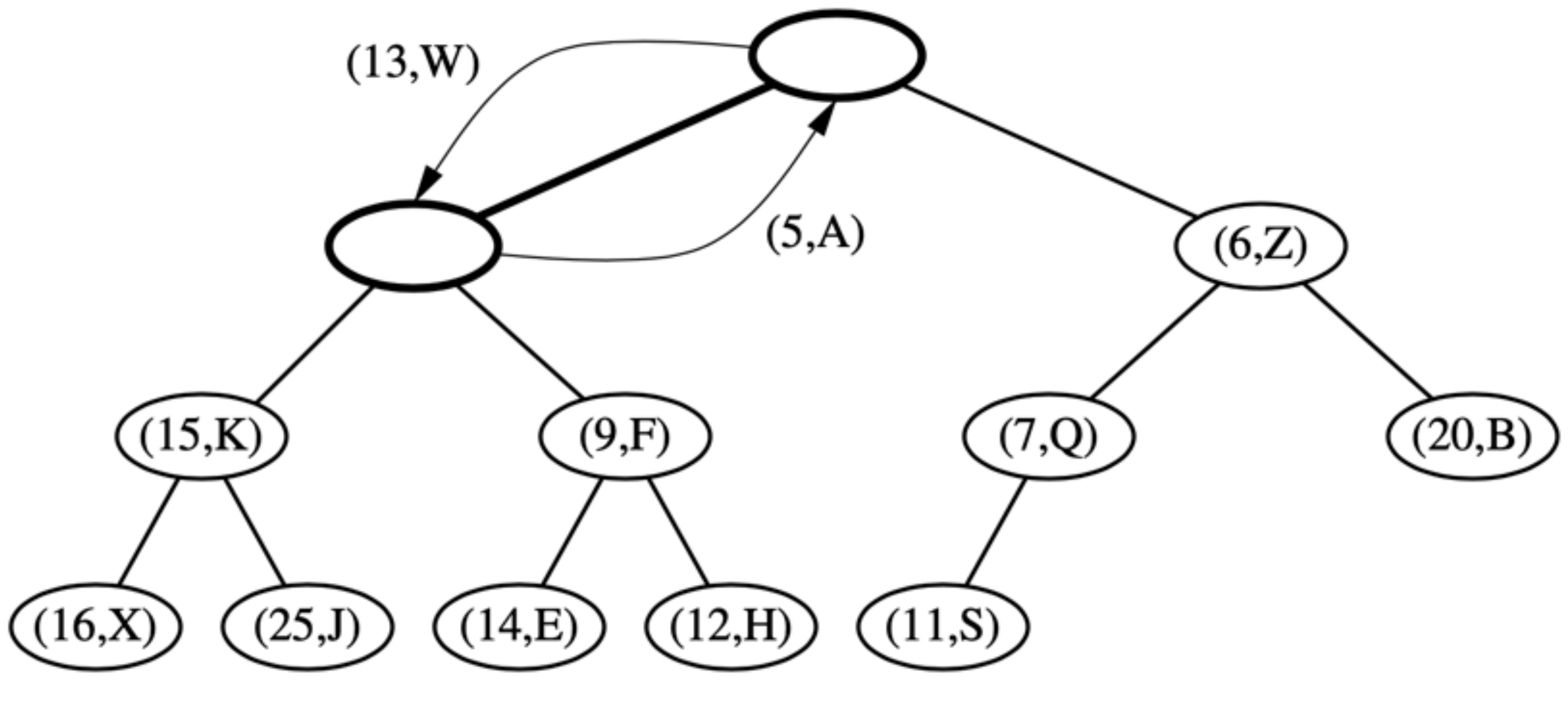
STEP 1. To keep tree **almost complete**, remove the root and move the right-most item on the last level to the root position.



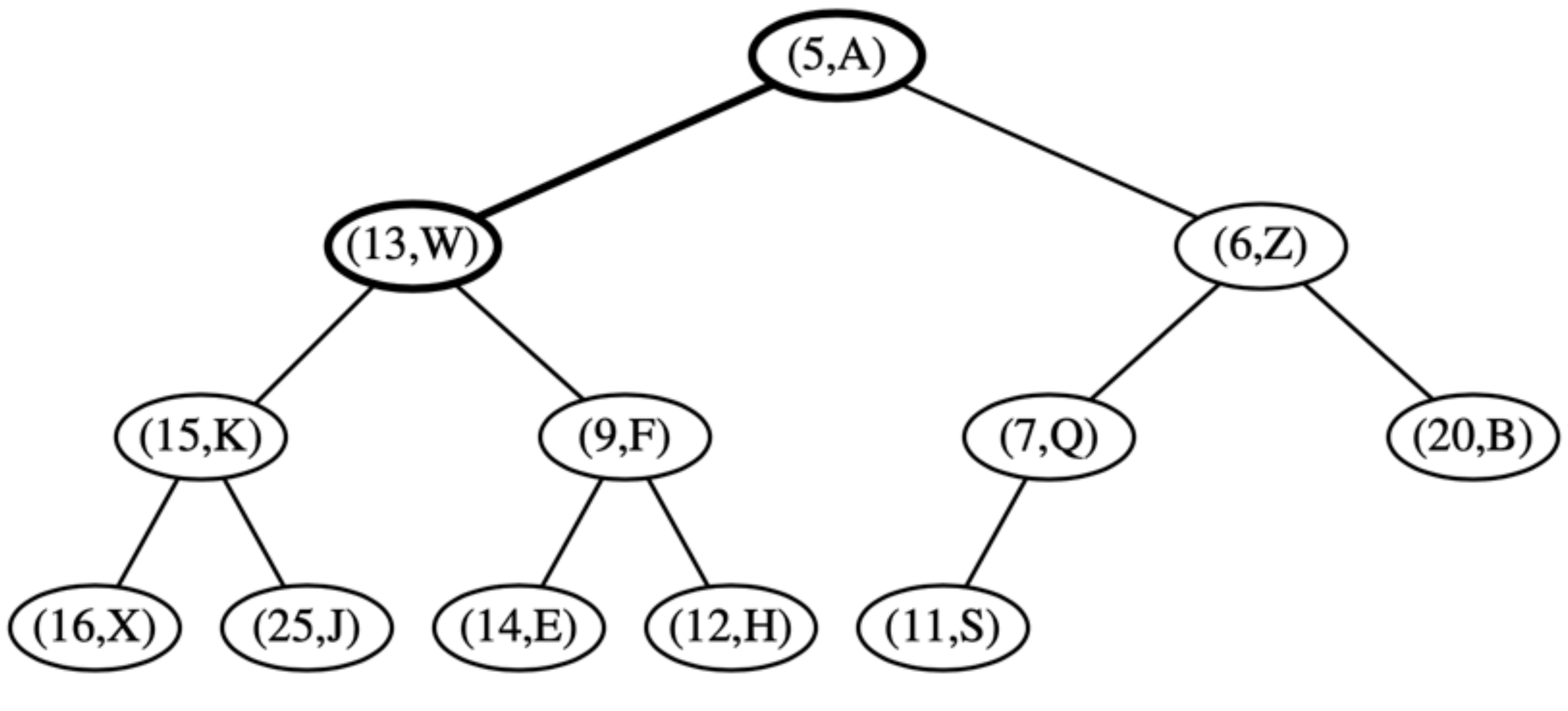
Down-Heap Bubbling

- How to remove the root item, which is the highest priority?

STEP 2. To maintain the heap property, we check the path **down** to the last level iteratively and swap the nodes if necessary.



(c)

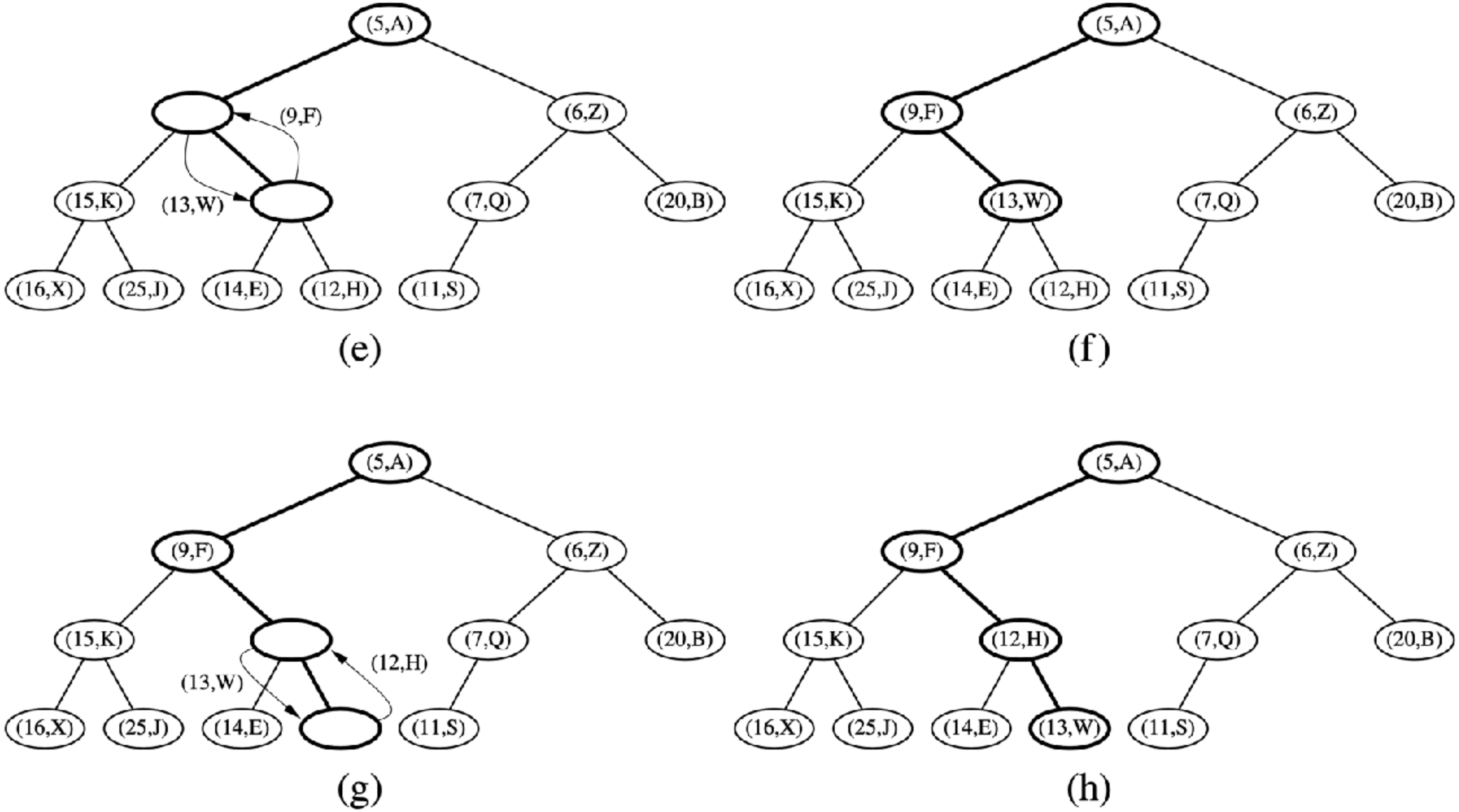


(d)

Down-Heap Bubbling

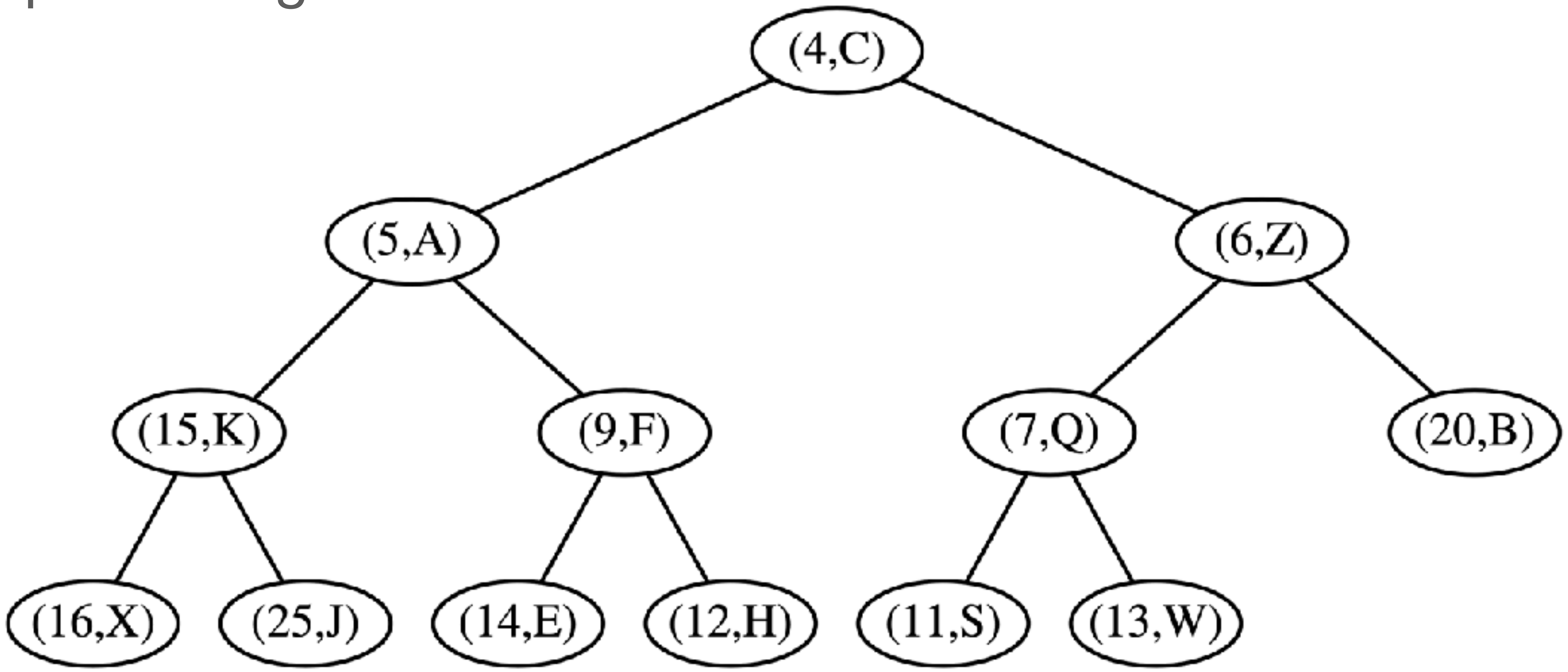
- How to remove the root item, which is the highest priority?

STEP 2. To maintain the heap property, we check the path **down** to the last level iteratively and swap the nodes if necessary.



Adaptable Priority Queue

- How to handle removing an item in the heap or changing its priority?
    - Assume the location of the item is provided !
    - **Update:** If decreasing the value, then check up-heap bubbling.
      If increasing the value, then check down-heap bubbling.
    - **Remove:** Move the rightmost item of the last row to the desired location, check up-heap or down-heap bubbling



Implementing the Heap Data Structure

- Remember the array implementation of a binary tree
- A heap is nothing other than an array, and no worries about vacancies in ordinary binary-tree-implementing arrays due to **"almost complete"** property.
- We need traversal up and down, who is the parent, right-child, left-child ?
- Easy arithmetic operations depending on 0- or 1-based array implementation.

1	2	3	4	5	6	7	8	9	10	11	12	13
4	5	6	15	9	7	20	16	25	14	12	11	13

parent(i): $\lfloor i/2 \rfloor$

left-child(i): $i \cdot 2$

reft-child(i): $i \cdot 2 + 1$

0	1	2	3	4	5	6	7	8	9	10	11	12
4	5	6	15	9	7	20	16	25	14	12	11	13

parent(i): $\lfloor (i-1)/2 \rfloor$

left-child(i): $i \cdot 2 + 1$

reft-child(i): $i \cdot 2 + 2$

Heap Construction

- Given an array, how to make it a heap?
- Can be done in $O(n)$-time !
- Bottom-up heap construction: All the items after a location is already a heap!

Assume, w.l.g, $n = 2^{h+1} - 1$, for some h, e.g., $1,3,7,15,31,....$ A complete binary tree

$a[1]$ $a[(n+1)/2 - 1]$ | $a[(n+1)/2]$ $a[n]$

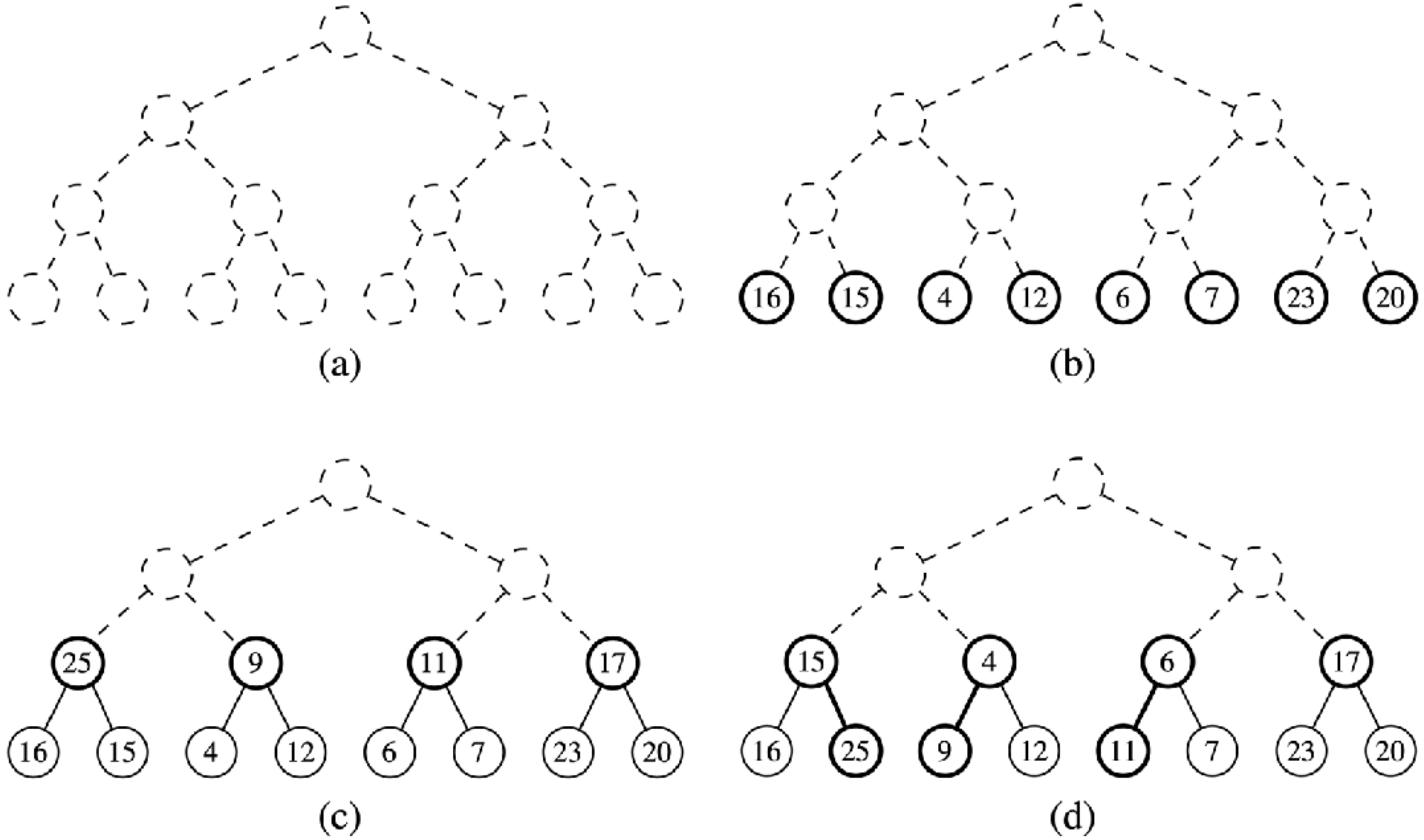
Each item in the first half, **from the right-most towards the left-most** will be subject to a down-heap bubbling.

No need to do anything for the second half, $(n+1)/2$ elements since these are the leaves.

Heap Construction

Input Array:

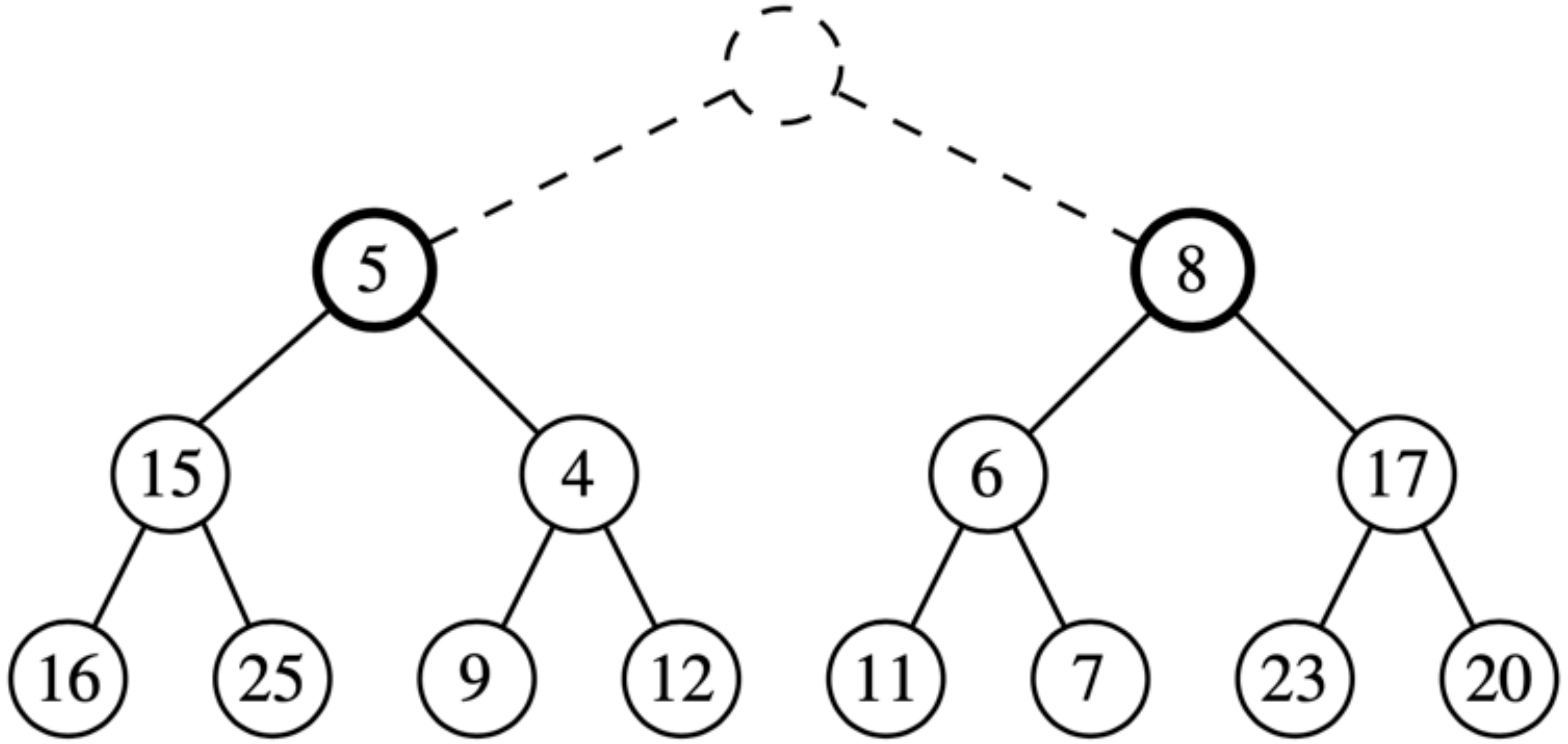
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	5	8	**25**	**9**	**11**	**17**	**16**	**15**	**4**	**12**	**6**	**7**	**23**	**20**



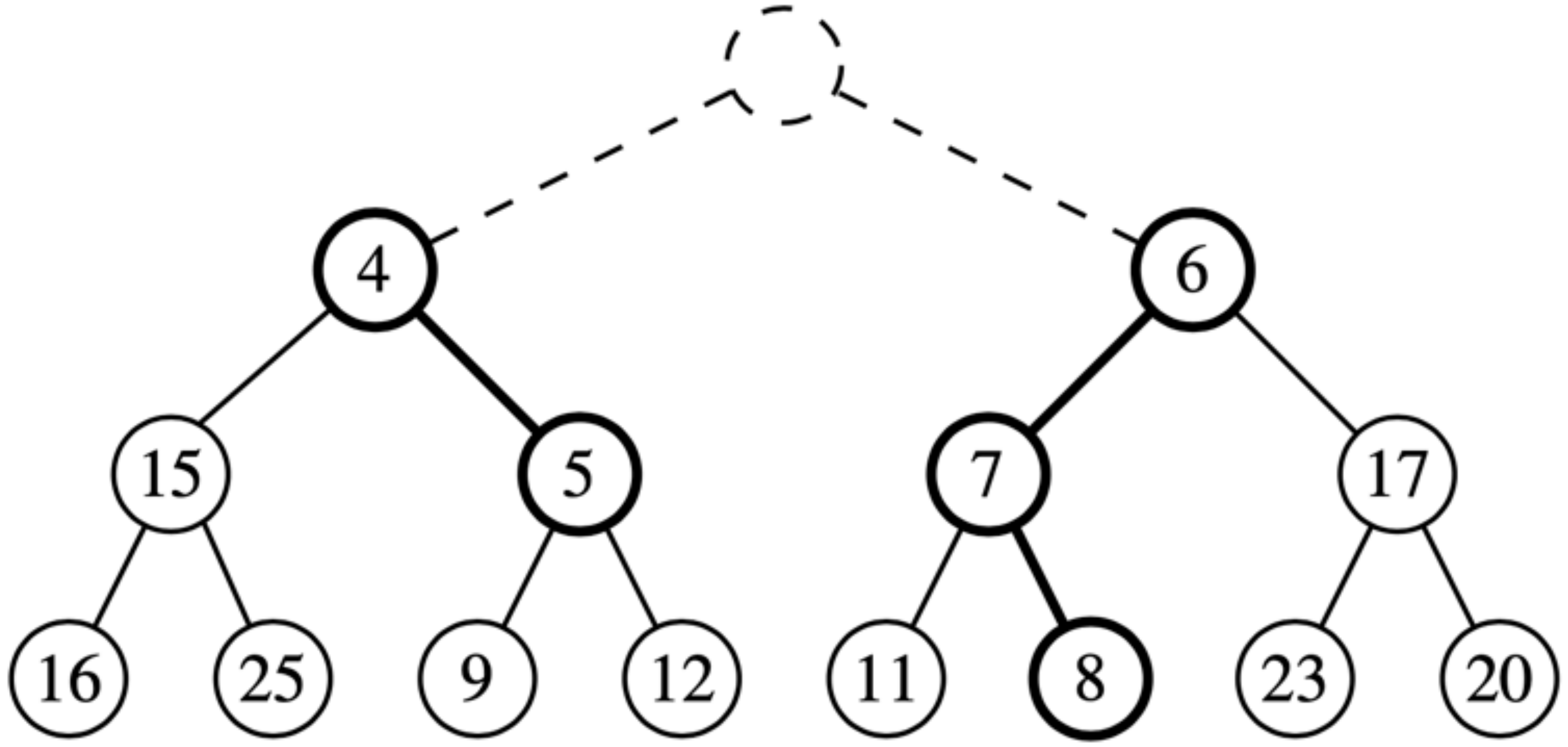
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	5	8	**15**	**4**	**6**	**17**	**16**	**25**	**9**	**12**	**11**	**7**	**23**	**20**

Heap Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	5	8	15	4	6	17	16	25	9	12	11	7	23	20



(e)

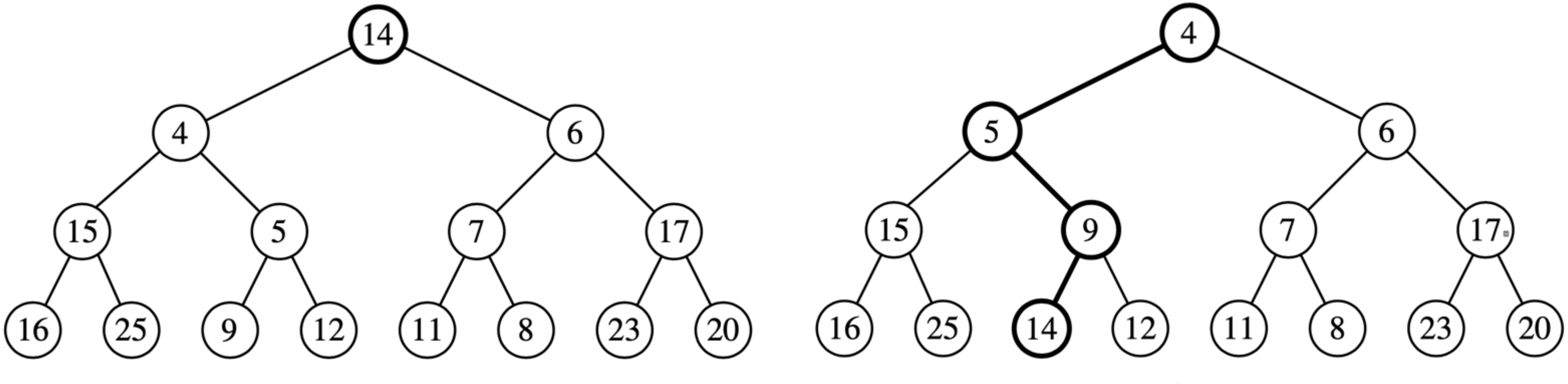


(f)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	4	6	15	5	7	17	16	25	9	12	11	8	23	20

Heap Construction

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	4	6	15	5	7	17	16	25	9	12	11	8	23	20



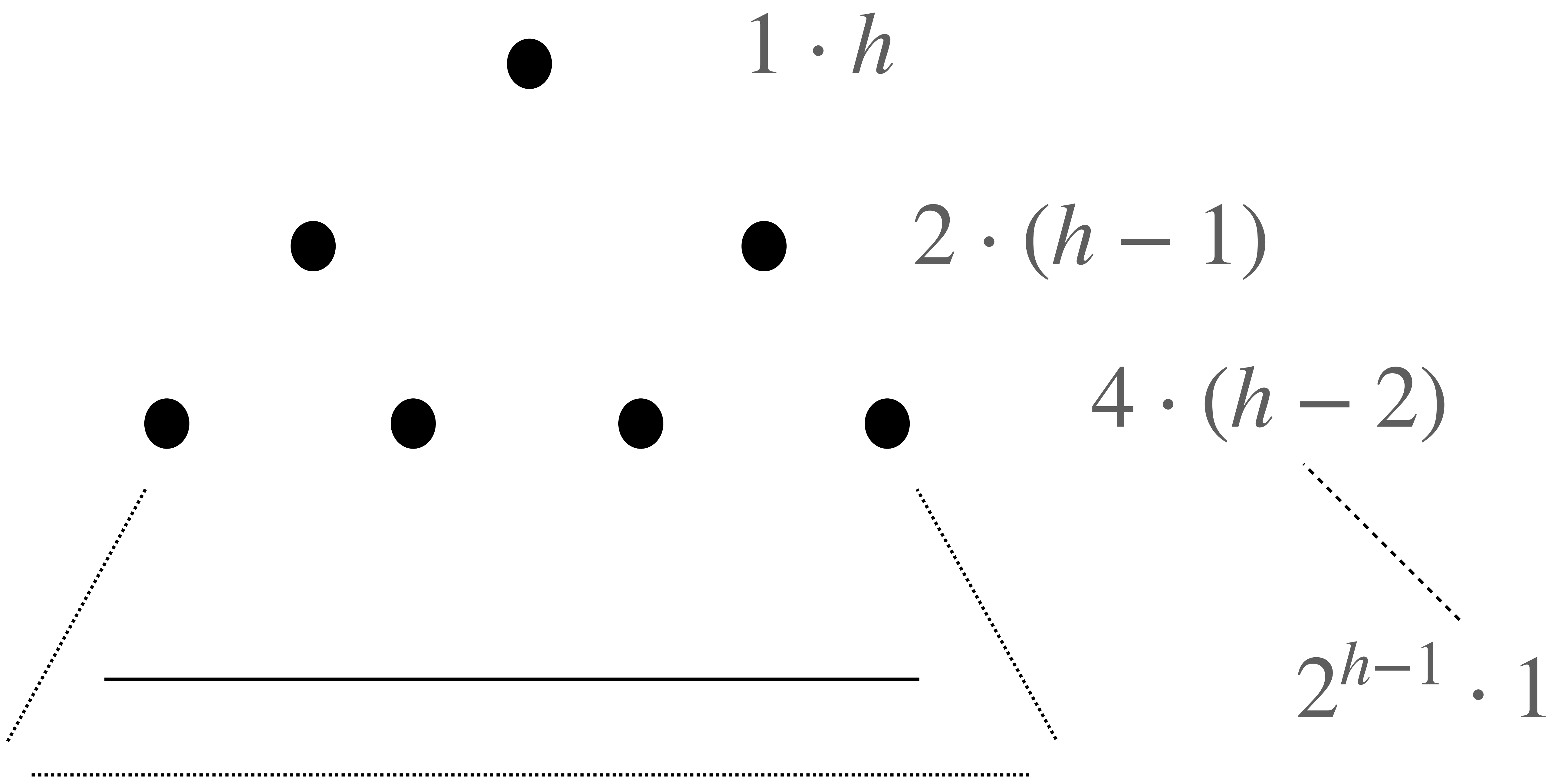
(g) (h)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	5	6	15	9	7	17	16	25	14	12	11	8	23	20

Heap Construction

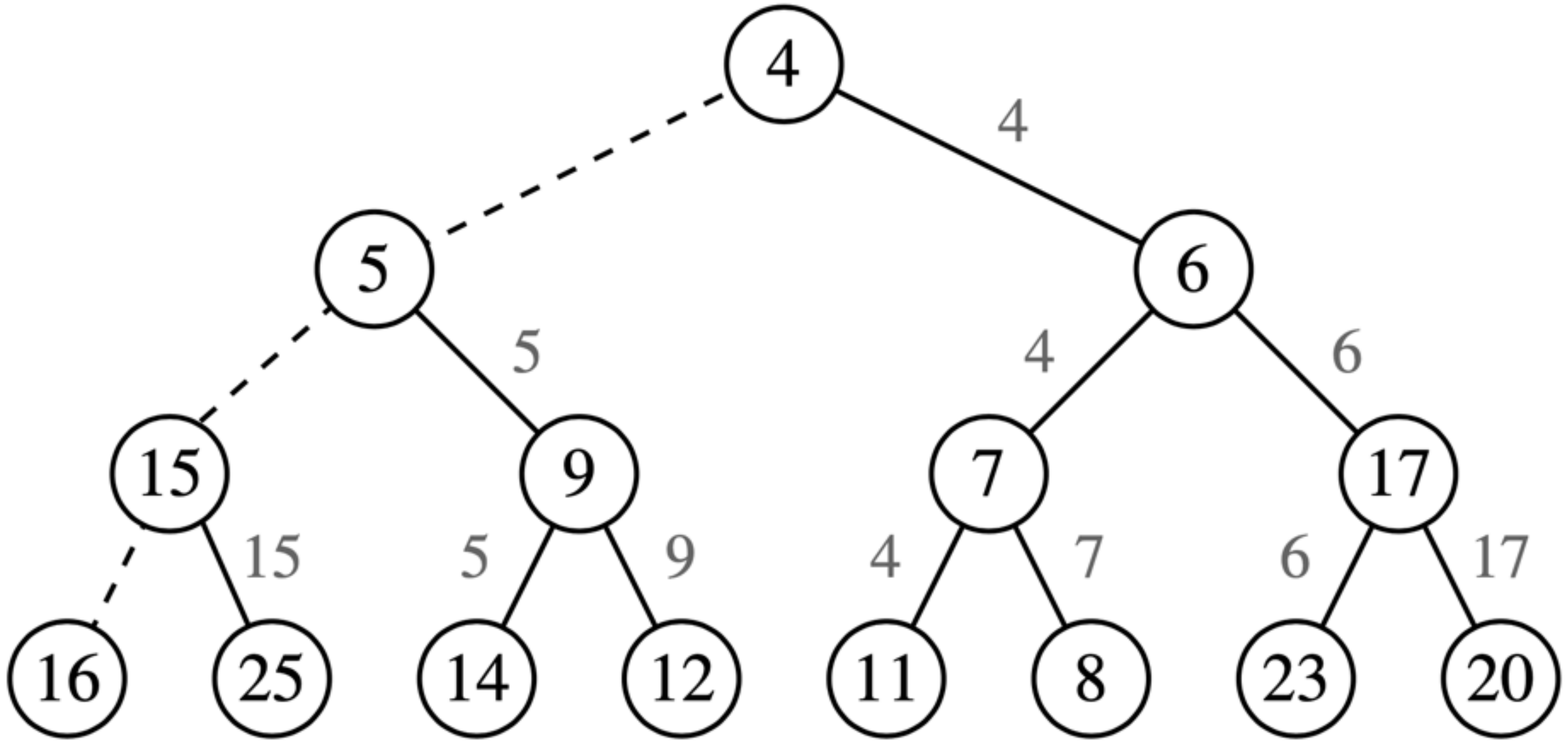
- How many *bubbling* steps are used per item ?
- In the worst case, each item will down-bubble to the lowest level, which depends on the level of the item in the tree.

$$1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \ldots + 2^r \cdot (h-r) + \ldots 2^{h-1} \cdot 1 = ?$$



Heap Construction

- A clever way to get this count
  - Associate each edge with an update operation
  - Observe that no edge is shared with multiple nodes
  - The number of edges in the tree, which is less than n, is upper-bound for the construction steps.
  - Hence, the construction is $O(n)$-time



- The path for each node is go right child, then always follow left-child until a leaf node.
- Successor of each node according to in-order traversal
- Each node have its path to a leaf and no edge is shared between any nodes !
- Maybe the exact path during the construction is different, but the path length does not change and we are trying to count the steps.

Heap Sort

- So, we can modify an input array to become a heap in O(n)-time
- We can use this heap for sorting
  - Extract the root for n times
- Building the heap plus n times root removal is $O(n) + O(n \log n) \rightarrow O(n \log n)$
- This is the heap-sort algorithm!
  Notice that it is an in-place sorting algorithm, no need for auxiliary space

Top-k Queries

- Return the k largest elements of a given sequence.
- In static case, just sort everything in $O(n \log n)$-time and return the largest k elements.
- In streaming case, it needs different handling.
    - Maintain the set of k largest elements observed so far, and then compare each newcomer with this set to replace the minimum element when the newcomer is larger than that.
    - Different ways can be modeled, but naively it yields $O(n \cdot k)$-time.

Top-k Queries

- A better solution is with the min-heap data structure:
  - Construct a min-heap with the first k elements in $O(k)$ time.
  - For every position after k, compare the candidate's value with the root of min-heap, and perform root-removal with new item insertion, if candidate is larger than the root. This takes $O(\log k)$ time.
- Thus, top-k queries can be answered in $O(n \log k)$ time with the heap.

Another example

- Given a string (or a stream !), find the first non-repeating k symbols.

- For instance, if S= **`abracadabraXYXZ`** and $k = 3$, then the output is **`c,d,Y`**

- Maintain a table holding the individual symbols that appear only once and also the first position of their appearance, which is $O(n)$-time.

- Create a min-heap from all those unique ones (which can be as many as n) and then extract k times, $O(n + k \log n)$-time with $O(n)$ heap size.

- Maintain a max-heap of size k elements. Whenever a unique element appears with a position less than the maximum of the k symbols in the heap, remove the root and insert the new item. This update may be required as many as n times and thus, $O(k + n \log k)$-time with $O(k)$ heap size.

Yet another one…

- Given n ropes, we want to connect them to make a single one. The cost of connecting two ropes is the sum of their lengths. What should be a good solution for that.

Example: If rope lengths are {3,5,1,8}, then

1 + 3 = 4	{4,5,8}
4 + 5 = 9	{8,9}
8 + 9 = 17	

Total cost: 4 + 9 + 17 = 30

Reading assignment

- Read the chapter 9 from Goodrich, chapter 6 from Cormen.