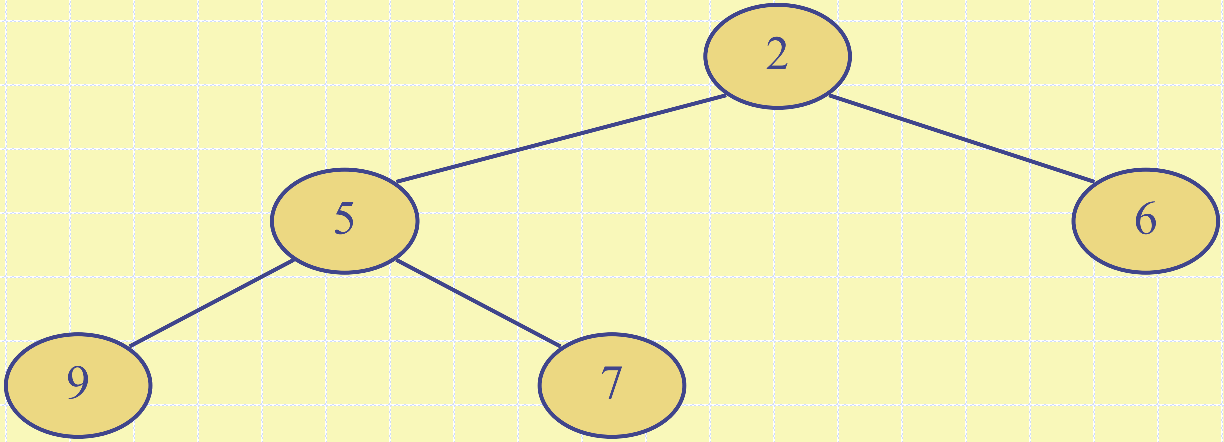
Heaps

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

Coverage & Warning

- Heaps



Warning: As an upfront warning, the “heap” data structure discussed here has nothing to do with the memory heap used in the run-time environment.

Recall 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

Recall P.Q. Sorting

- We use a priority queue
  - Insert the elements with a series of insert operations
  - Remove the elements in sorted order with a series of removeMin operations
- 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 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.remove* (*S. first* ())
  P.insertItem(*e*, ∅)
while ¬*P.isEmpty*()
  e ← *P.removeMin*().*getKey*()
  S.addLast(*e*)

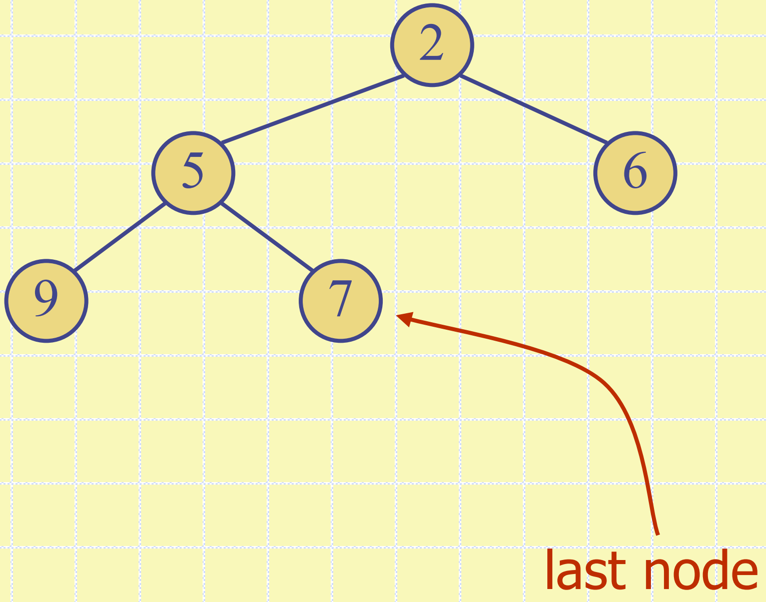
Heaps

- Both insertion-sort and selection-sort of P.Q. achieved running time of $O(n^2)$.
- An efficient realization of a priority queue uses a data structure, called ***heap***.
- This data structure allows us to perform both insertions and removals in logarithmic time, which is significant improvement over the list-based implementation.
- Fundamentally, the heap achieves such improvement by abandoning the idea of storing entries in a list; instead it stores the entries in a binary tree.

Heaps

- In other words, a heap is a binary tree storing entries at its nodes.
- Additionally, a heap satisfies 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
  - there is at most one node with a single child (that is, you cannot find two or more nodes with single child) and this child must be a left child

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



Heap-order Property

- Heap-Order Property: This s a relational property. For every internal node v other than the root, $key(v) \geq key(parent(v))$.
- Consequently, the keys encountered on a path from the root to an external node are in non-decreasing order.
- Additionally, the minimum key (which is the most important one) is hence always stored at the root (or the "top of the heap", hence the name "heap" of this data structure).

Complete Binary Tree Property

- Complete Binary Tree Property: This is a structural property. This property is needed to insure that the height of the heap is as small as possible.

Complete Binary Tree Property

- A heap T with height h is a complete binary tree if each depth $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$ has the maximum possible number of entries, and at least one entry at the last depth (that is depth $\boldsymbol{h}$).

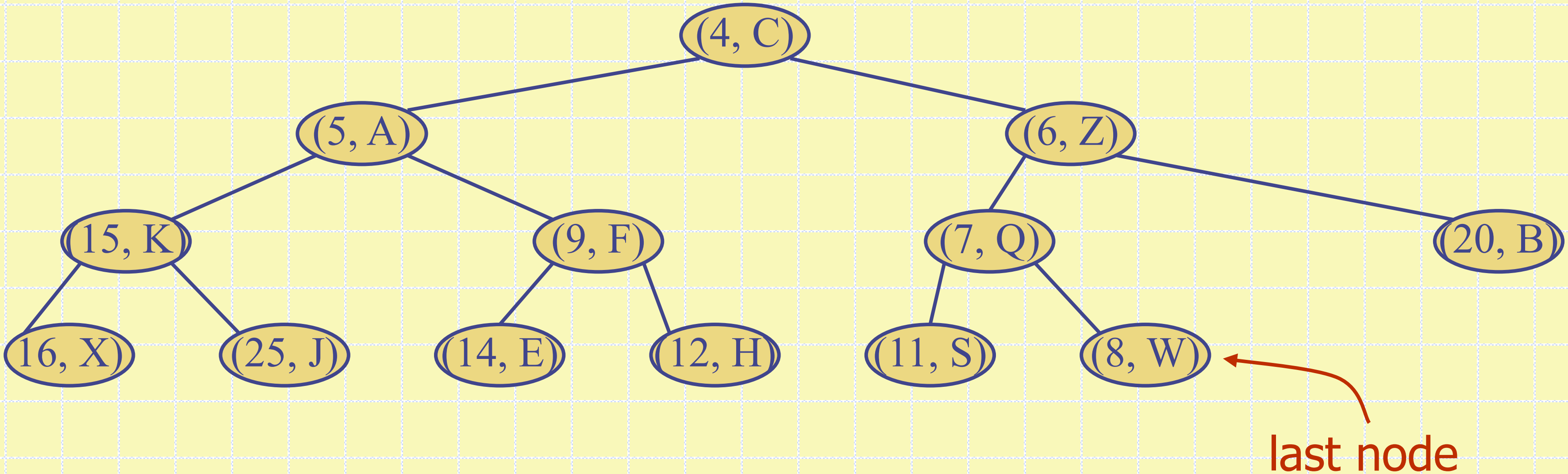
- Formally, each level $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$ must have $2^{\boldsymbol{i}}$ nodes.

- In a complete binary tree, a node v is to the left of node w if v and w are at the same level and that v is encountered before w (from left to right, which is also an inorder traversal).

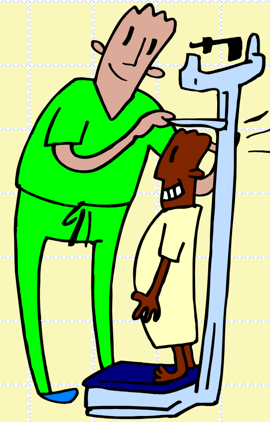
Complete Binary Tree Property

- For instance, node with entry *(15, K)* is to the left of node with entry *(7, Q)*. Similarly, node with entry *(14, E)* is to the left of node with entry *(8, w)*, etc.
- Another important node is the ***last node***. This is the **right-most**, **deepest** **external** node in the tree.

  Note: **right-most** means the one to the right of all other nodes in the level (this may not necessary be a right child).



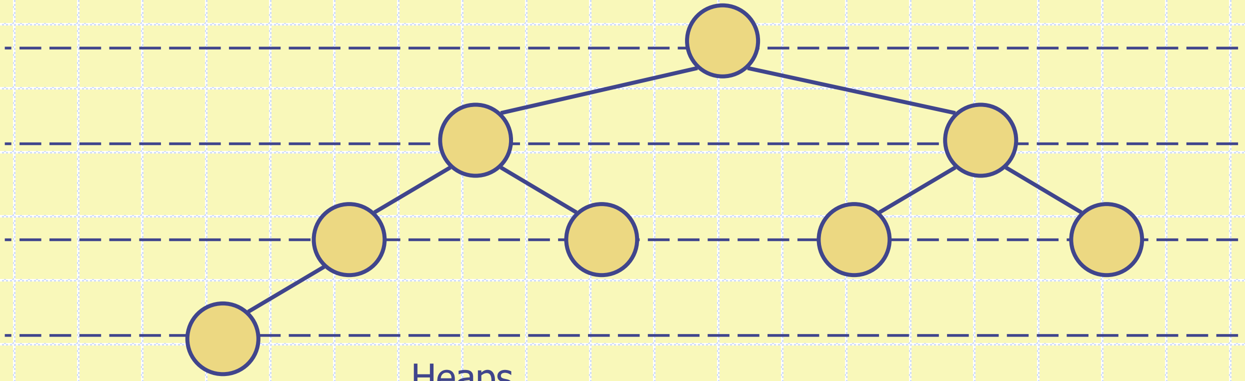
Height of a Heap



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

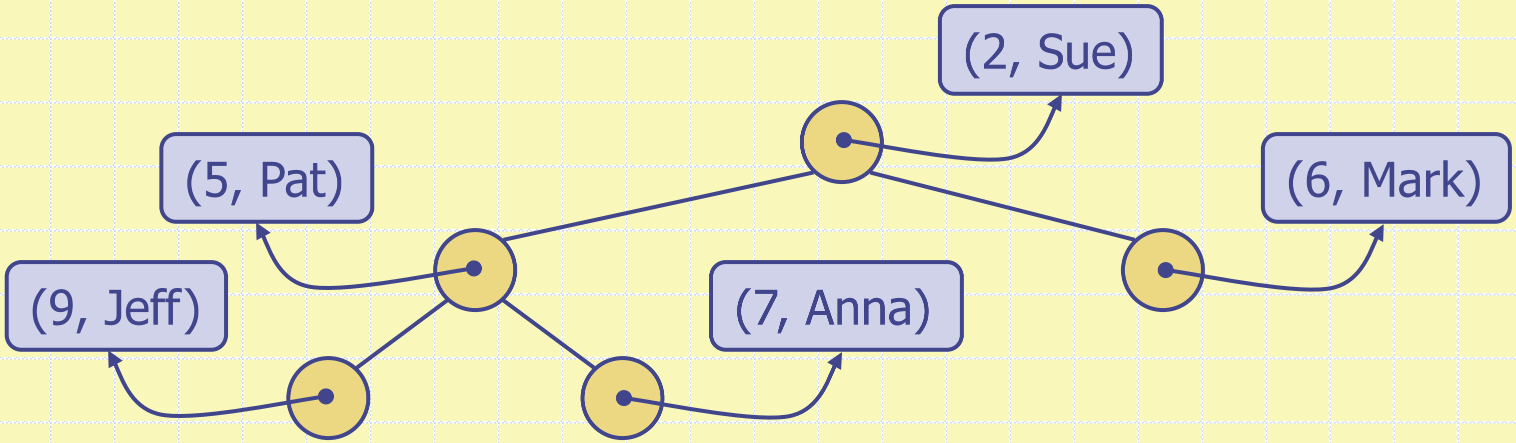
  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$
- This theorem is very important since it implies that if we can perform updates on a heap in a time proportional to its height, then those operations will run in logarithmic time.

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



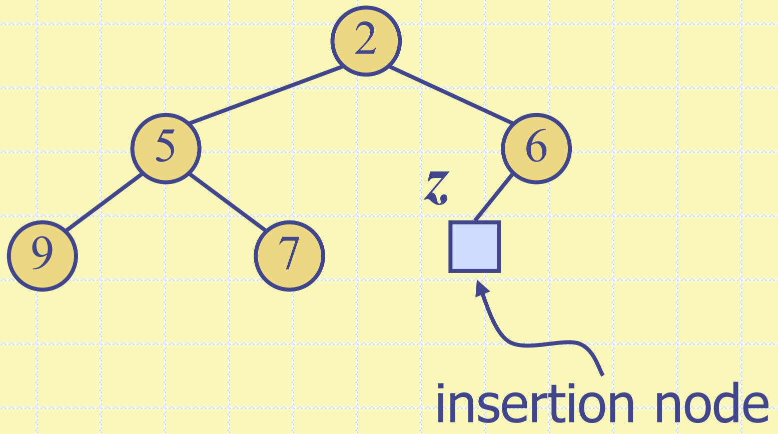
Heaps and Priority Queues

- We can use a heap to implement a priority queue.
- We store a (key, element) item at each node.
- We keep track of the position of the last node.

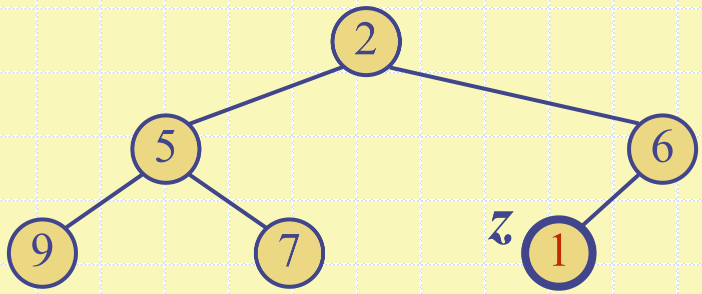


Insertion into a Heap

- Method *insert(k, x)* of the priority queue ADT corresponds to the insertion of a key ***k*** to the heap.

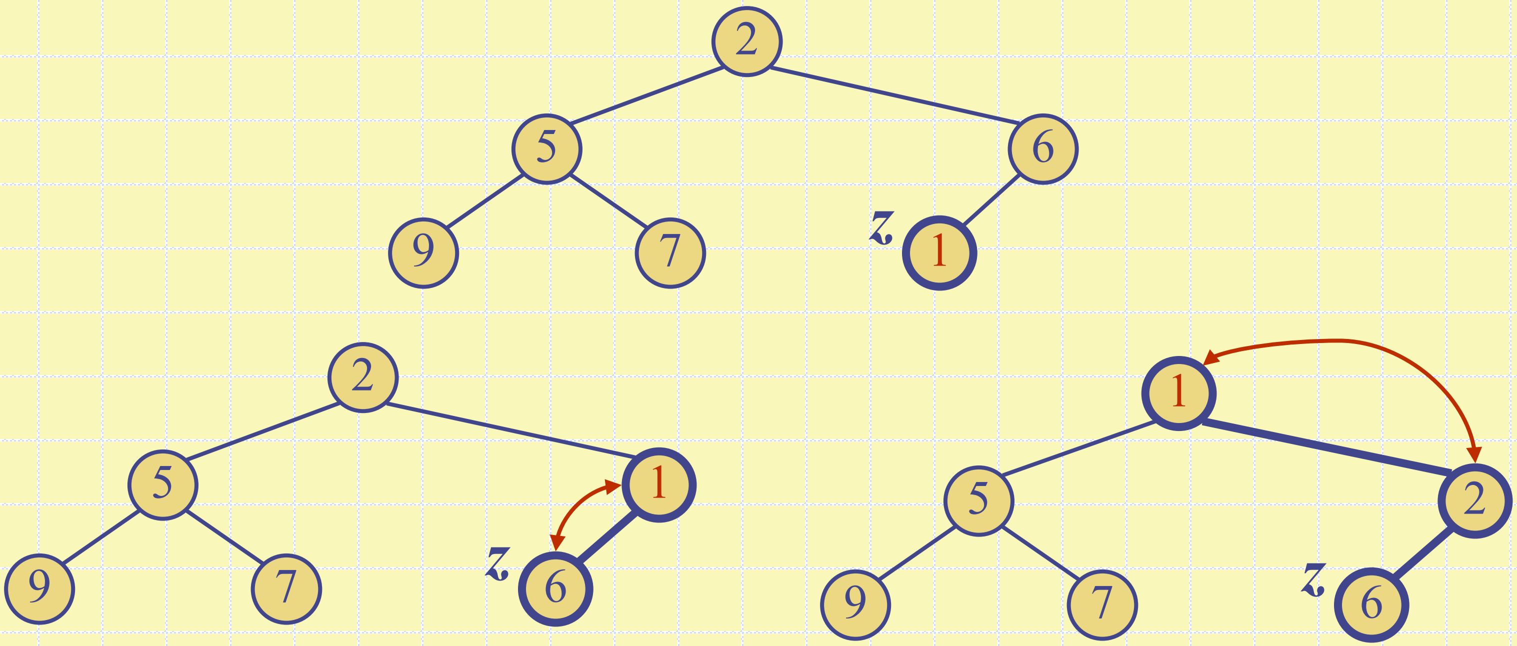


- The insertion algorithm consists of three steps:
  - Find the insertion node ***z*** (the new last node)
  - Store ***k*** at ***z***
  - Restore the heap-order property (discussed next)



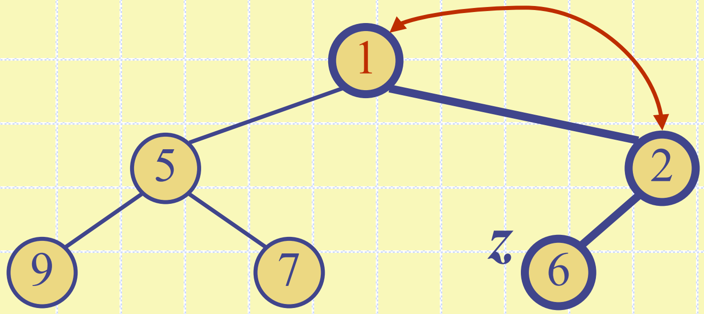
Upheap

- 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.



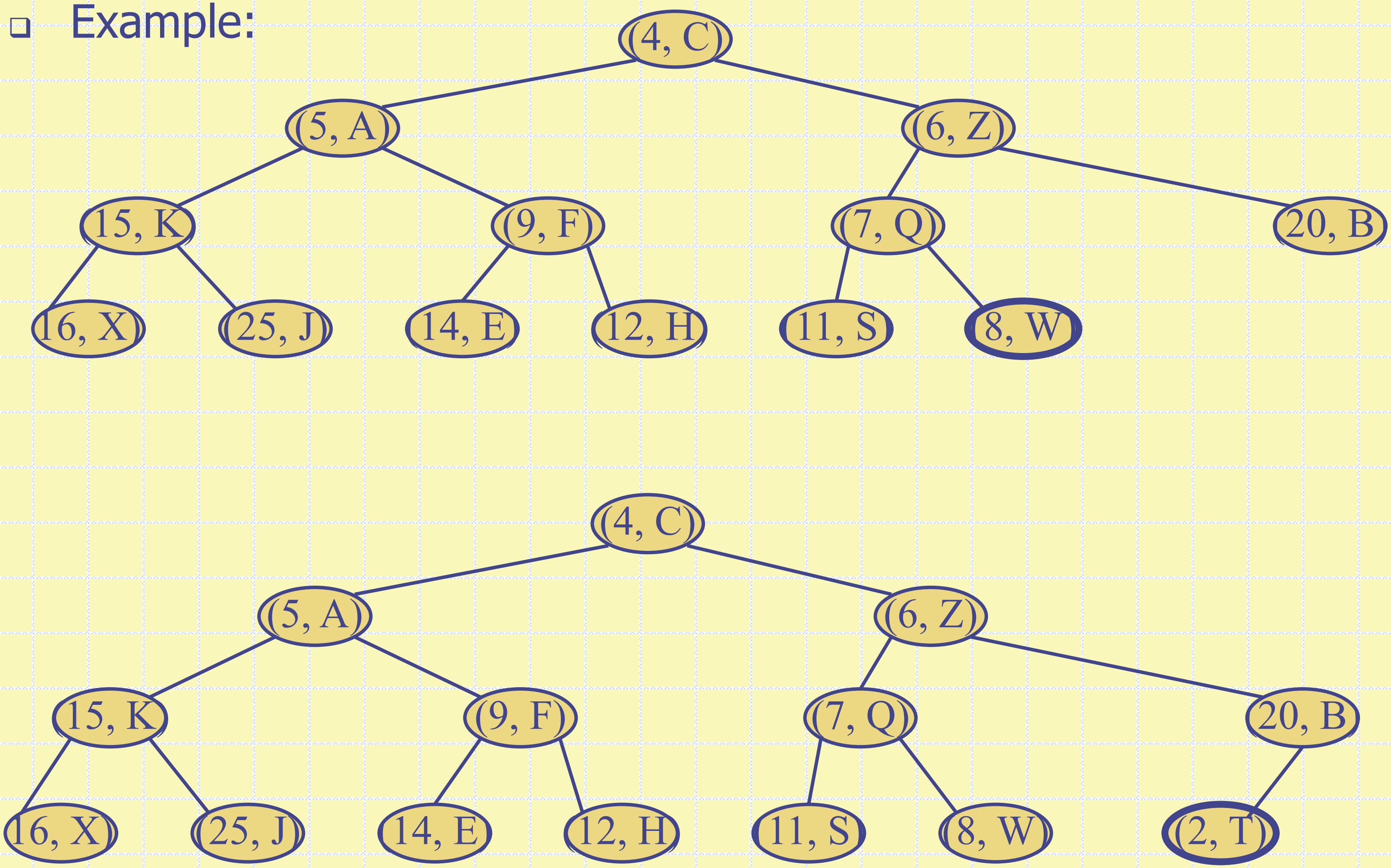
Upheap

- 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.



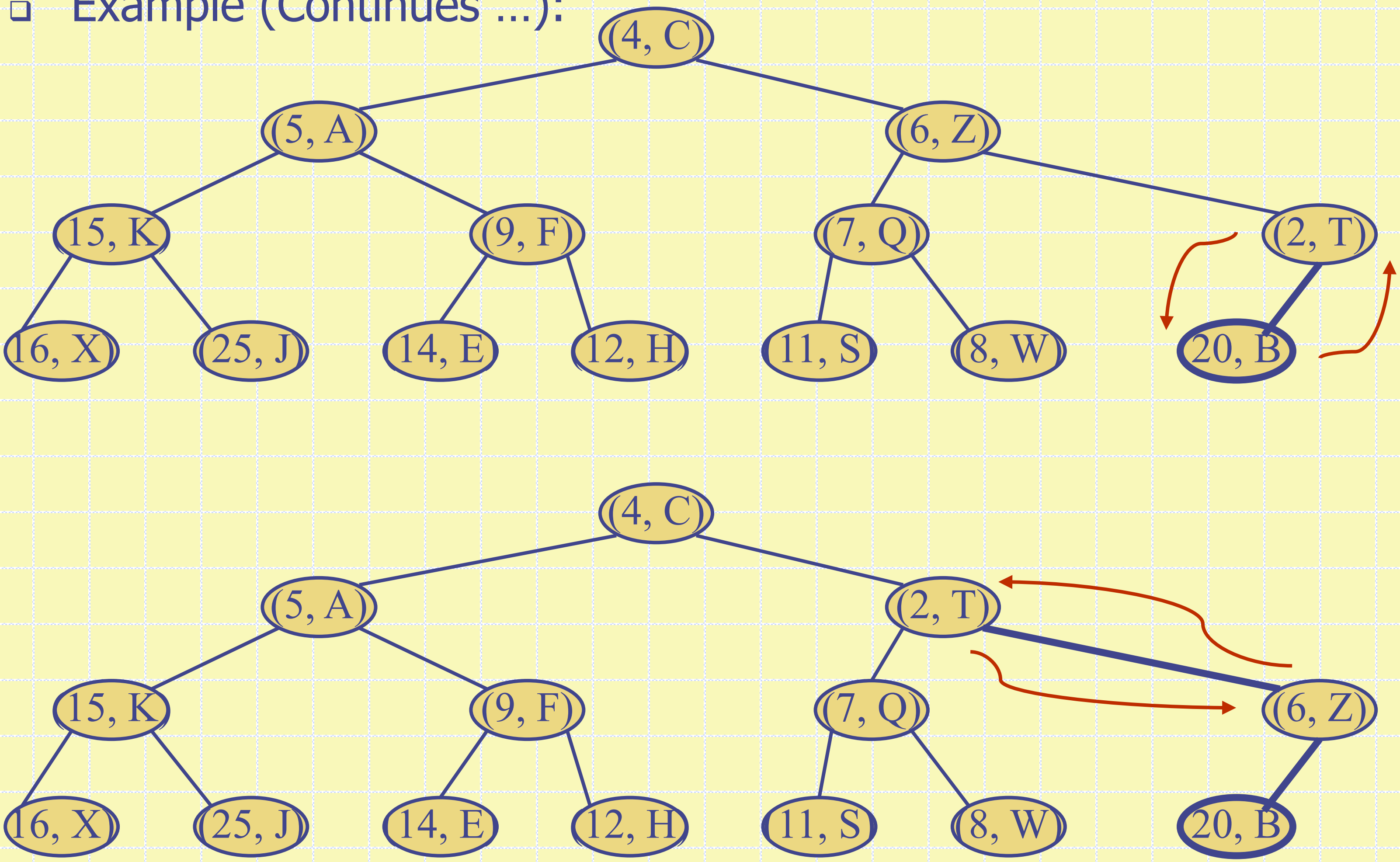
Upheap

- Example:



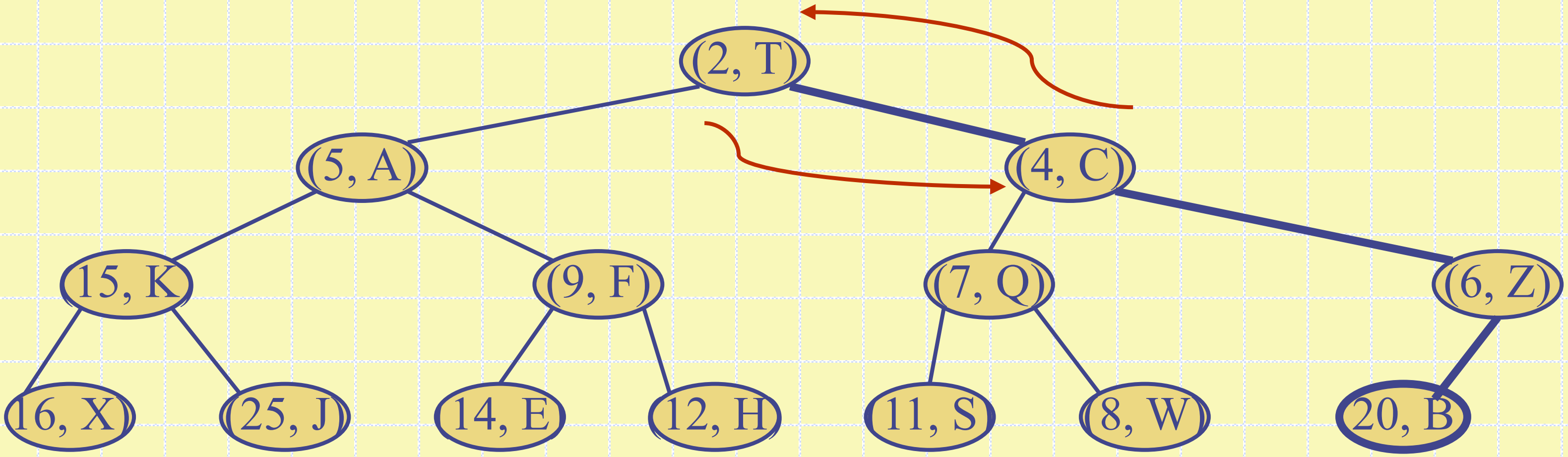
Upheap

- Example (Continues …):



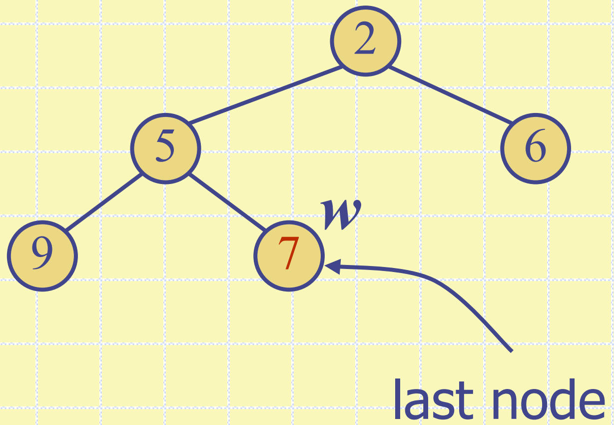
Upheap

- Example (Continues …):

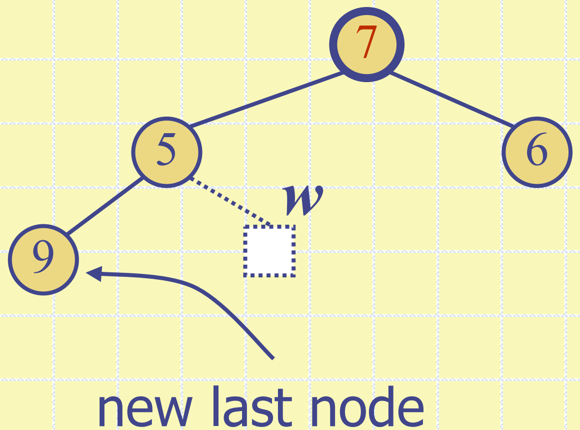


Removal from a Heap (§ 7.3.3)

- Method *removeMin()* of the priority queue ADT corresponds to the removal of the root key from the heap.

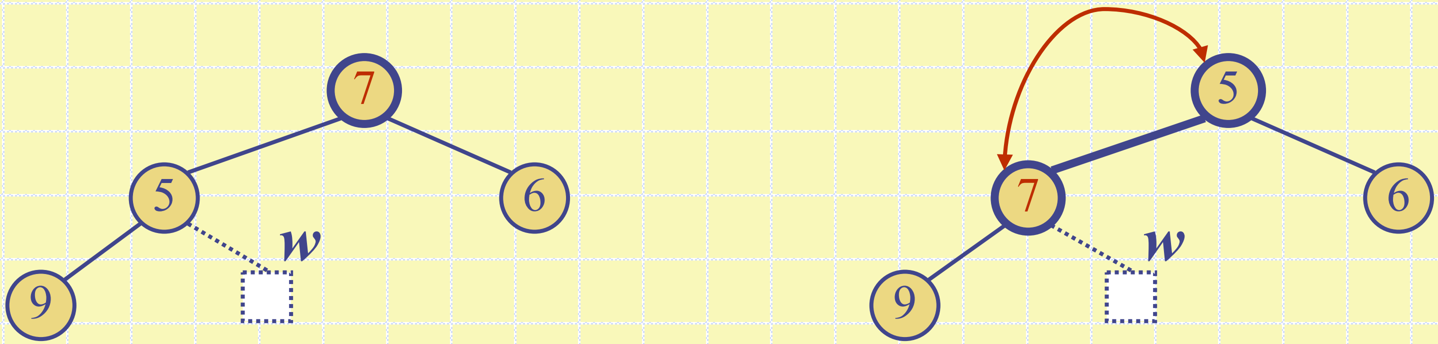


- The removal algorithm consists of four steps:
  - Return the root entry
  - Replace the root key (entry in fact) with the key of the last node ***w***
  - Remove ***w***
  - Restore the heap-order property (discussed next)



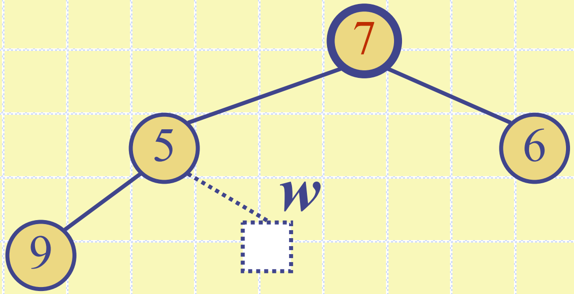
Downheap

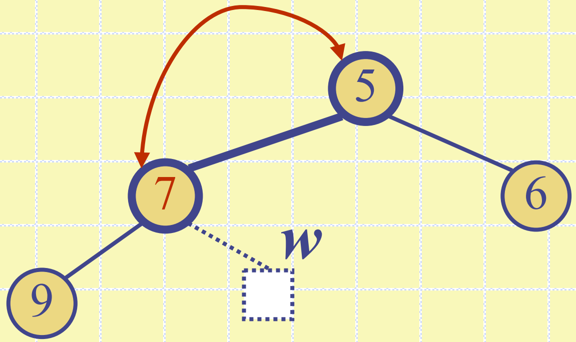
- After replacing the root key with the key k of the last node, the heap-order property may be violated.
- Algorithm *downheap* restores the heap-order property by swapping key k along a downward path from the root.
  - If T has no right child, then the swapping (if needed) starts at the left child (which is the only one actually)
  - If both left and right children are there, then the swapping occurs with the one that has the smaller key; otherwise only one side of the tree may be corrected, which may force further swap operations.



Downheap

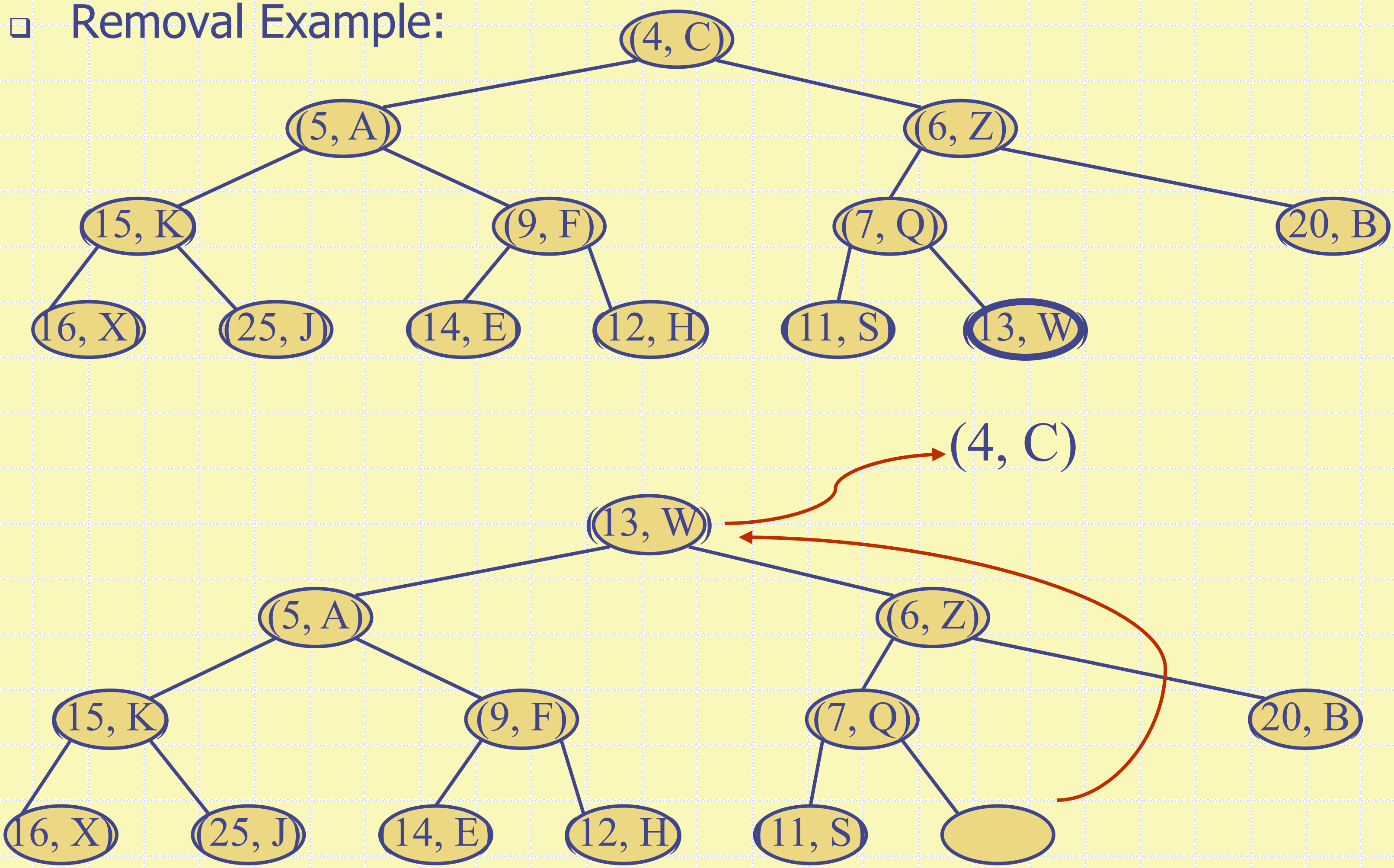
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k.
- The downward swapping process is called ***down-heap bubbling***.
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time.





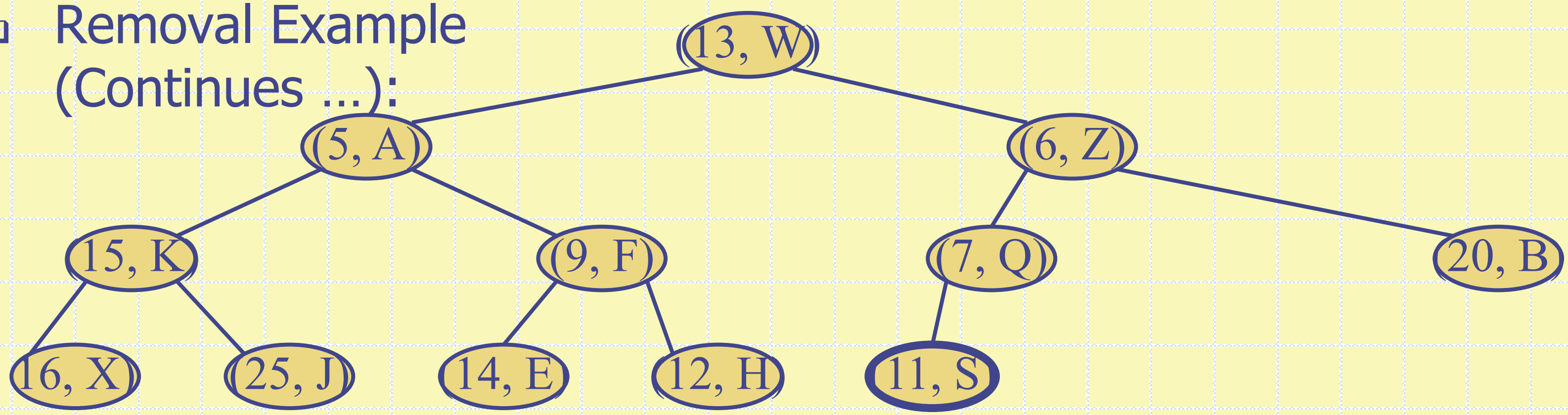
Downheap

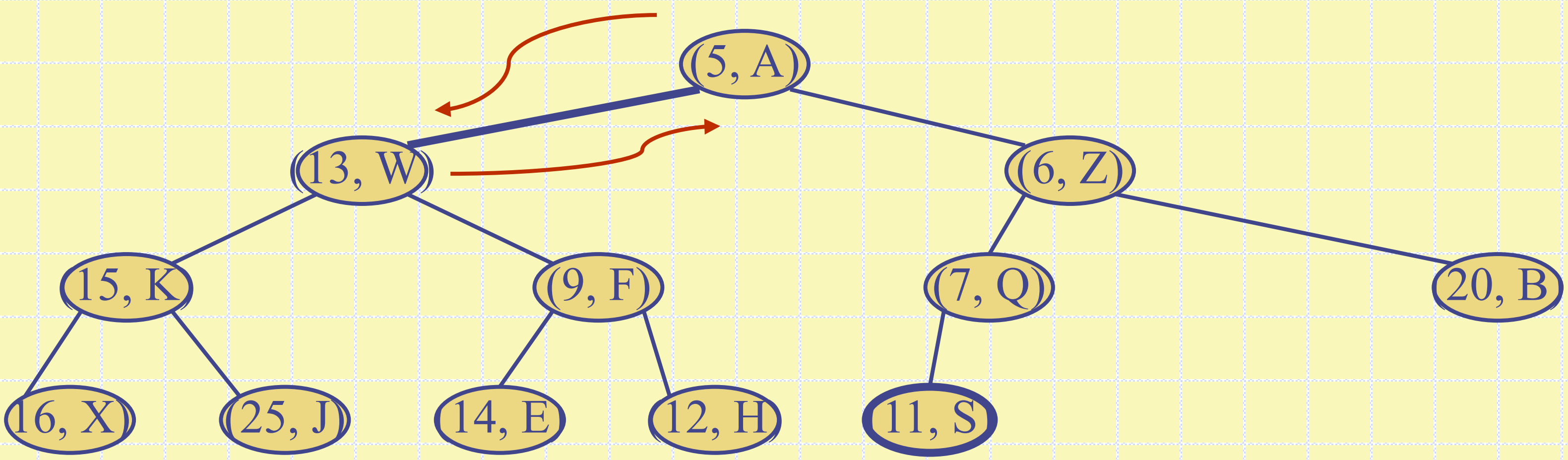
- Removal Example:



Downheap

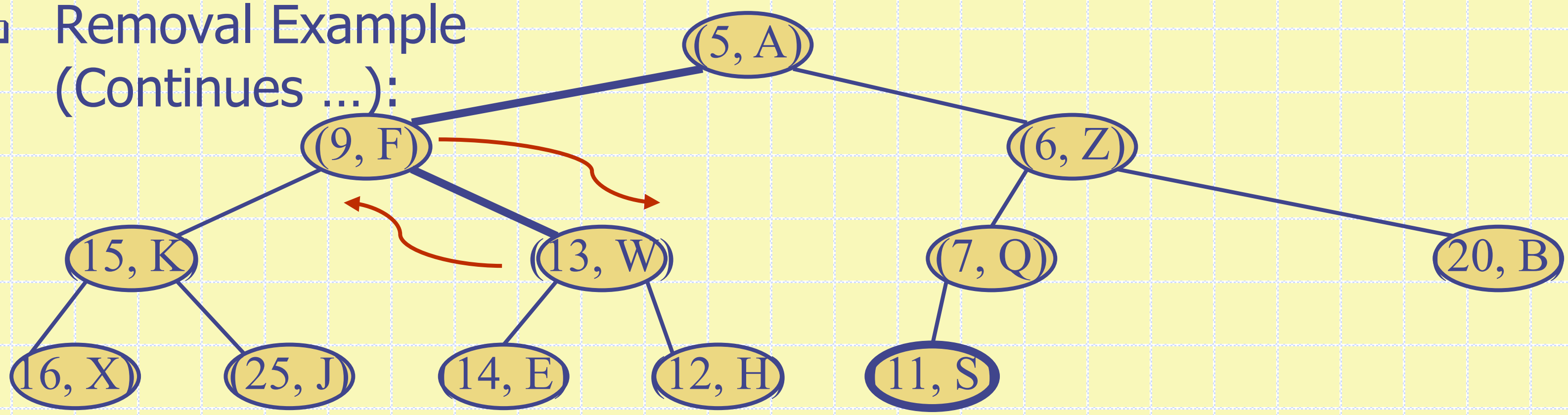
- Removal Example (Continues …):

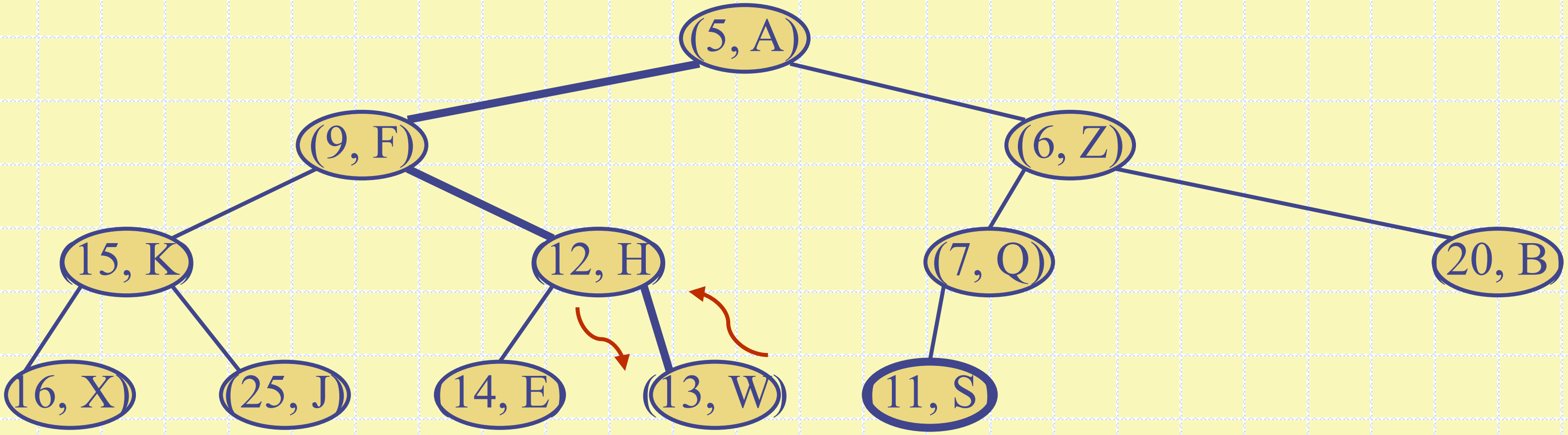




Downheap

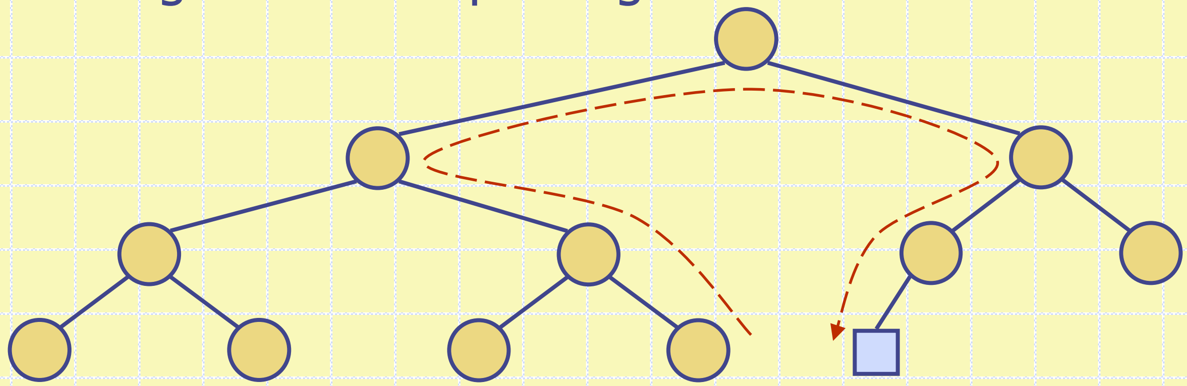
- Removal Example (Continues …):





Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
- For instance, if the current last node is a left child then:
  - Go up to the father and down right to the new last node
- If the current last node is a right child then:
  - 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

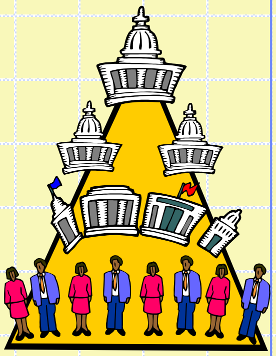


Heap Performance

- The performance of a P.Q. realized by means of a heap is as follows:

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

Heap-Sort



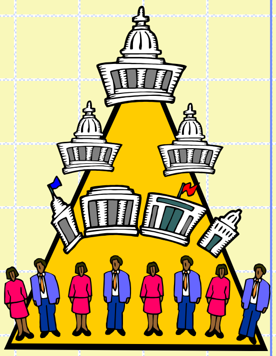
- Let us recall the PQ-Sort algorithm

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***

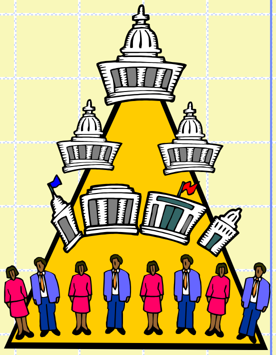
  while ¬*S.isEmpty* ()
    e ← *S.removeFirst* ()
    P.insert (*e*, ∅)
  while ¬*P.isEmpty*()
    e ← *P.removeMin*().*getKey*()
    S.addLast(*e*)

Heap-Sort



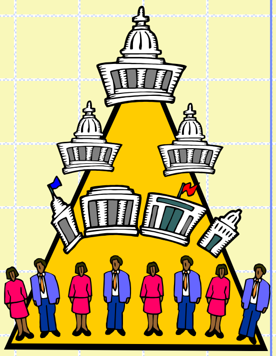
- Using a heap-based priority queue, the following is true:
  - Each insertion takes $O(\log n)$ time
    - In fact the insertion of any entry i takes exactly $\boldsymbol{O(1 + \log i)}$ since this is actually the i^{th} insertion where $1 <= i <= n$.
  - Each removal takes $O(\log n)$ time
    - In fact the removal of any entry i takes exactly $\boldsymbol{O(1 + \log(n - i + 1))}$ since this is actually the i^{th} removal (that is, some elements may have already been removed) where $1 <= i <= n$.
- Consequently, the entire algorithm consumes $O(n \log n)$ time to have n insertions in phase I (first loop), and $O(n \log n)$ time to have n removals in phase II (second loop) which leads to a final complexity of $\boldsymbol{O(n \log n)}$.

Heap-Sort



- The resulting algorithm using heap to realize the P.Q. is hence called ***heap-sort***.
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort.

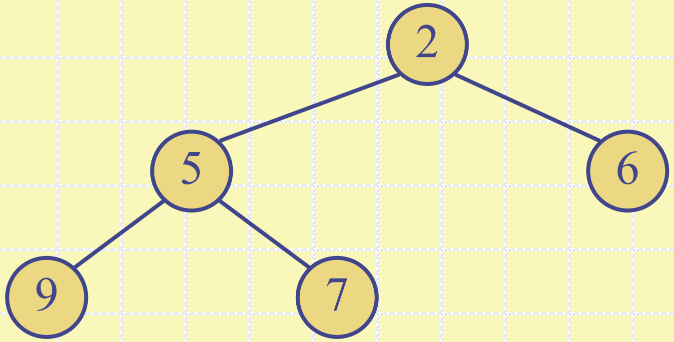
Heap-Sort



- 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

Array-based Heap Implementation

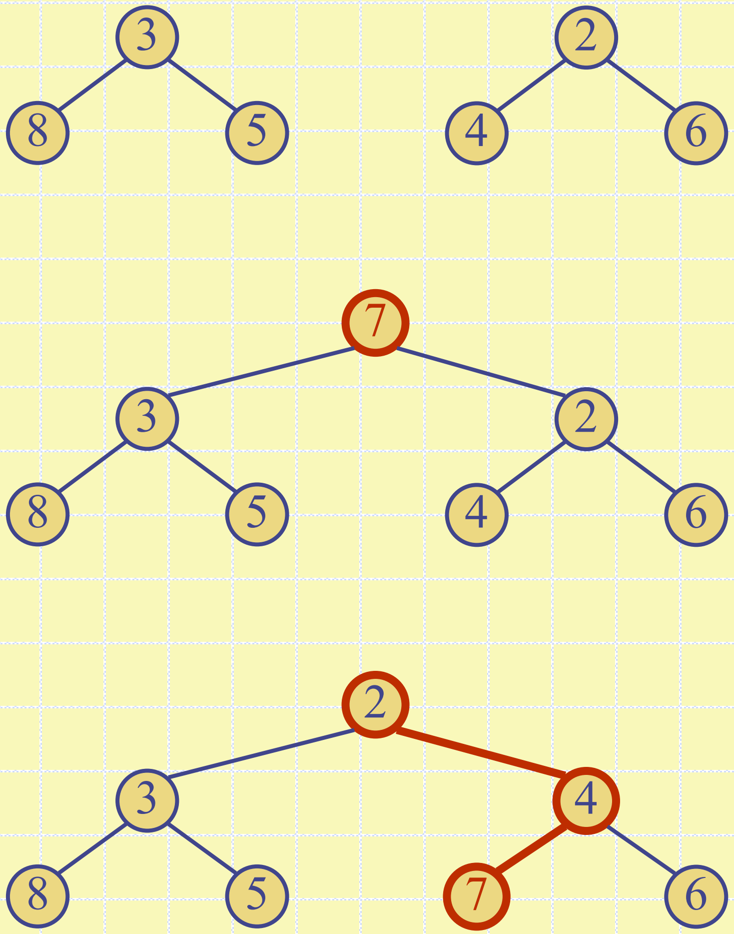
- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank(index) $2i + 1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Operation insert corresponds to inserting at rank n
- Operation removeMin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



2	5	6	9	7
0	1	2	3	4

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

- It is possible to construct a heap in o(n log n) time.

- Simply, perform n successive insertions.
  - Since worst case for any of these insertions is $log\ n$, the total time consumption is $O(n\ log\ n)$.

- However, if all the entries (that is all the key-value pairs) are known and given in advance, we can have an alternative approach that results only on a complexity of $O(n)$.

- Such approach is referred to as the ***bottom-up construction*** approach.

Bottom-up Heap Construction

- To simplify the description of the algorithm, let us assume that the heap is a complete binary tree; that is all levels are full.
- At each height *i*, we have 2^i nodes
- Total number of nodes for a tree of height *h* is:

  $n = 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^h$

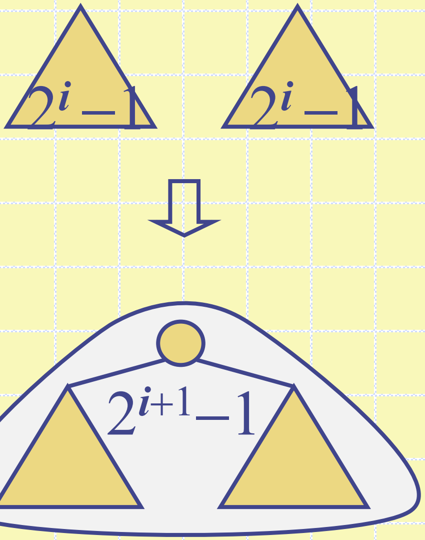
  which is equivalent to $2^{h+1} - 1$

  for example, if h=4 then we have 31 nodes, which is $2^5 - 1$

  - Consequently,
    - $n = 2^{h+1} - 1$
    - $h = \log(n + 1) - 1$
- The construction of the heap consists of $h + 1$ steps (which is also $\log(n + 1)$ steps).

Bottom-up Heap Construction

- Generally, in each step i, pairs of heaps with $2^i - 1$ keys are merged to construct larger heaps with $2^{i+1} - 1$ keys.

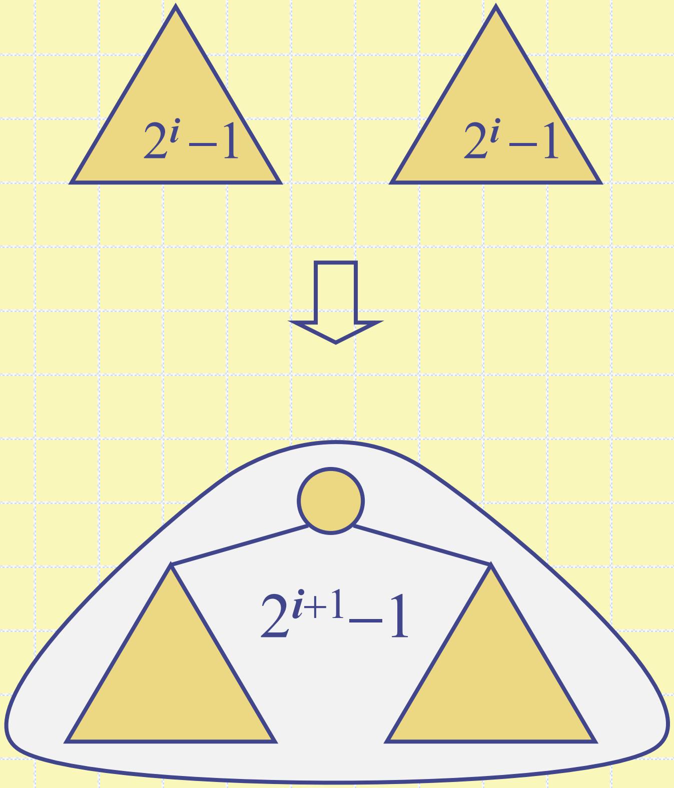


- Example: Assume n = 31, h = 4

Step # i	**# of heaps** $n+1/2^i$	**# of elements in each heap** $2^i - 1$ **(this leads to a larger heap of** $2^{i+1} - 1$ **at next step** $i+1$**)**
1	16	1
2	8	3
3	4	7
4	2	15
5	1	31

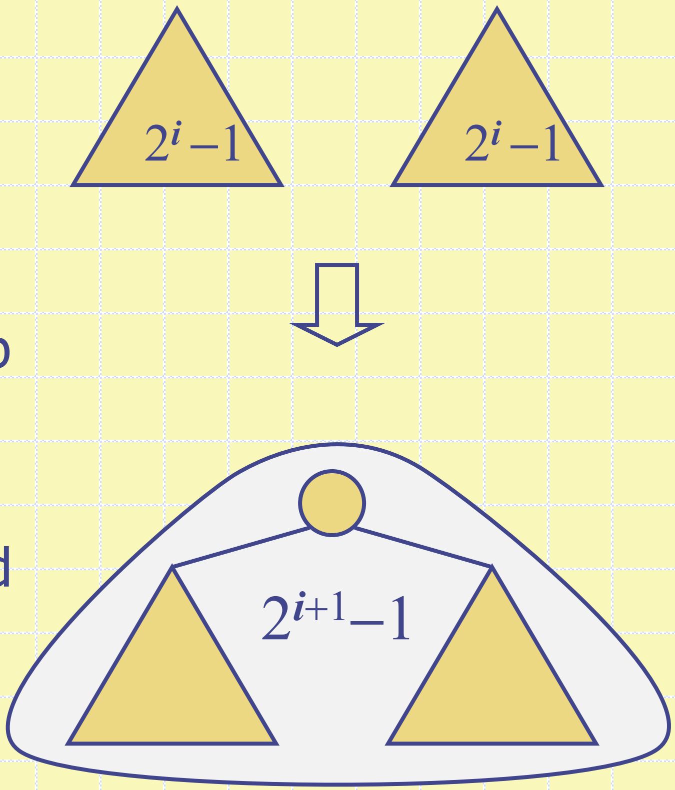
Bottom-up Heap Construction

- Generally, in each step $\boldsymbol{i}$, pairs of heaps with $2^{i}-1$ keys are merged to construct larger heaps with $2^{i+1}-1$ keys.
- Step 1: construct $(n+1)/2^{1}$ elementary heaps with one entry each.
- Step 2: construct $(n+1)/2^{2}$ heaps, each storing 3 entries by joining pairs of elementary heaps and adding a new entry. New entry is added at the root, and swapping is performed if needed.



Bottom-up Heap Construction

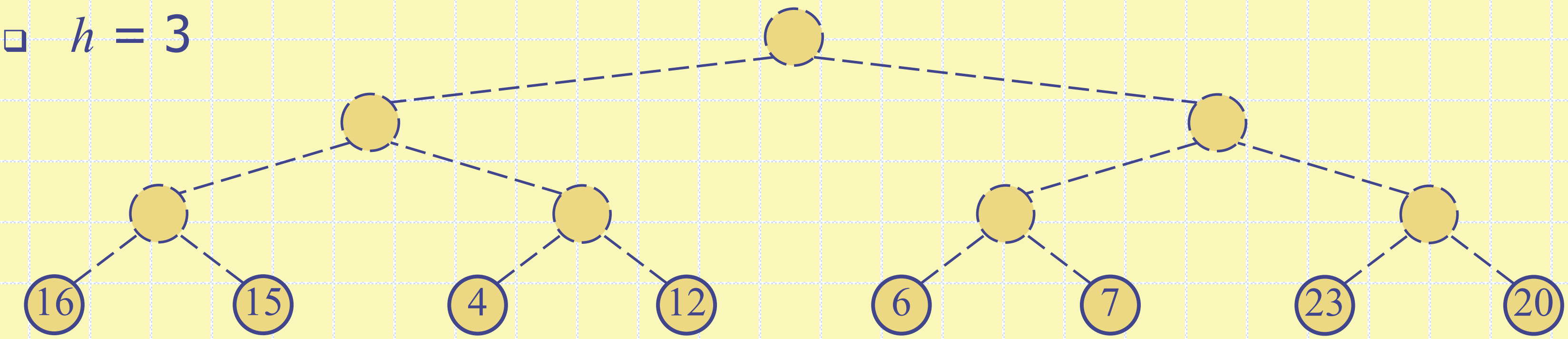
- :
- Step $h + 1$: In this last step, construct $(n + 1) / 2^{h+1}$ heap. This is actually the final heap containing all n elements by joining the last two heaps (each storing $(n - 1)/2$ entries) from Step h.



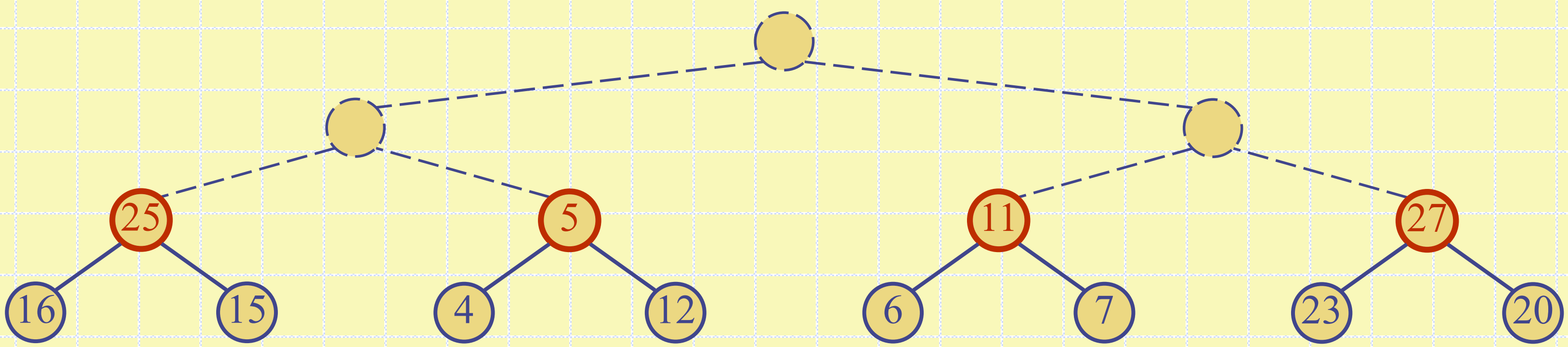
- As before, the new entry is added at the root, and swapping is performed if needed.
  - Note that all swap operations in each of the steps use down-heap bubbling to preserve the heap-order property.

Bottom-up Heap Construction Example

- $h = 3$

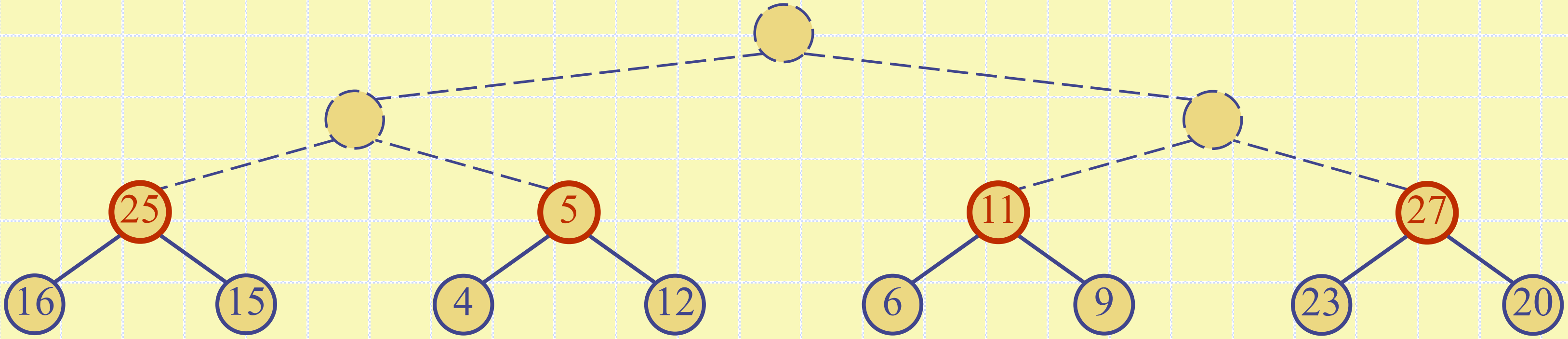


Step 1

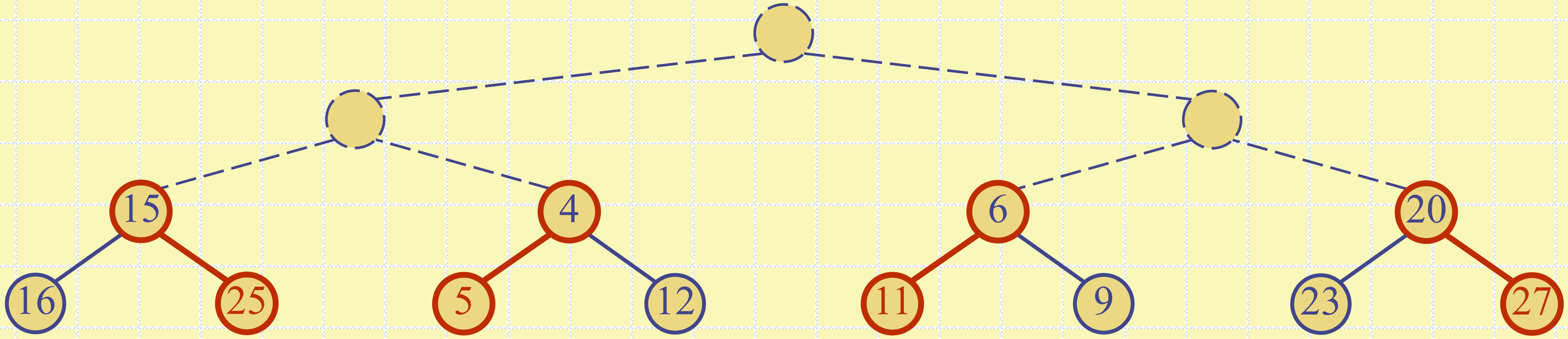


Step 2

Example (contd.)

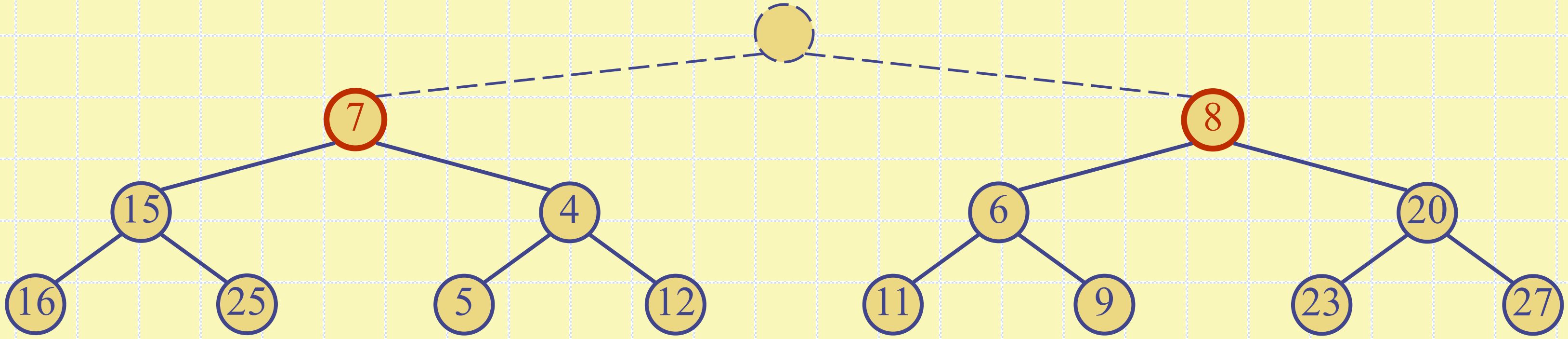


Heap-order property is violated; we need to apply down-heap bubbling

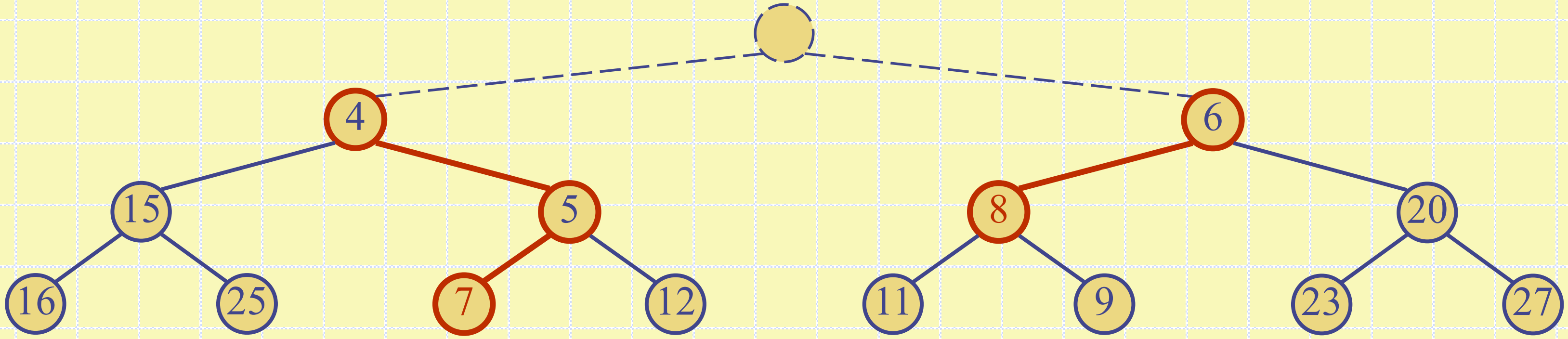


Step 2 is concluded

Example (contd.)

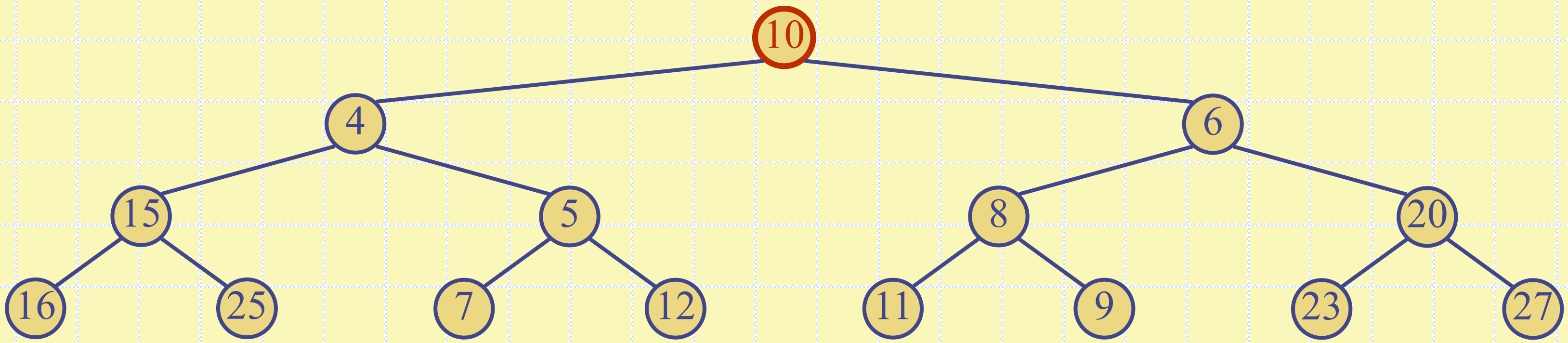


Step 3

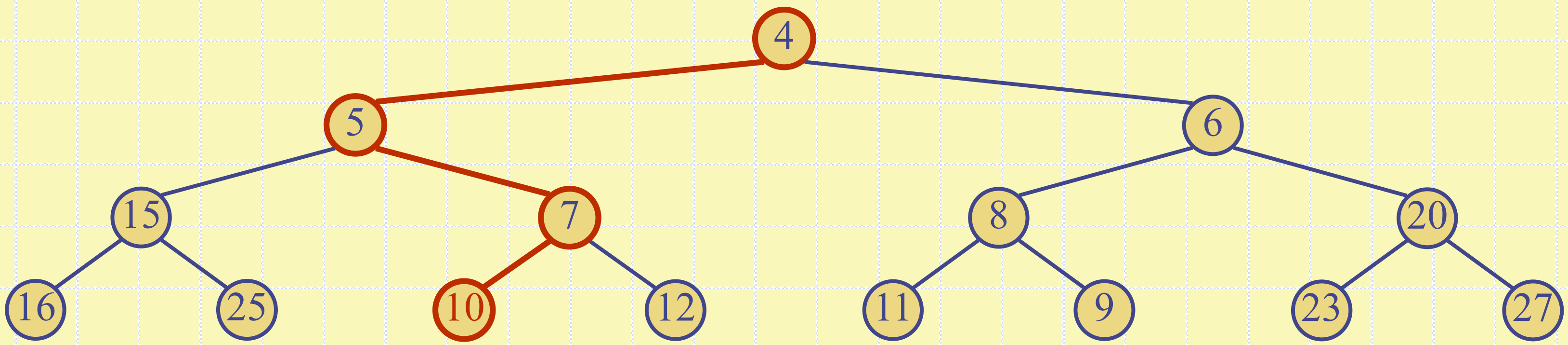


Heap-order property was violated; Heap after applying down-heap bubbling

Example (end)

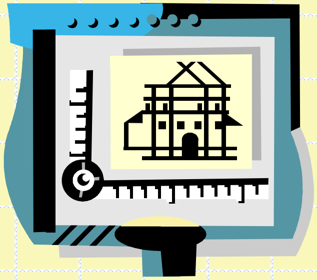


Step 4



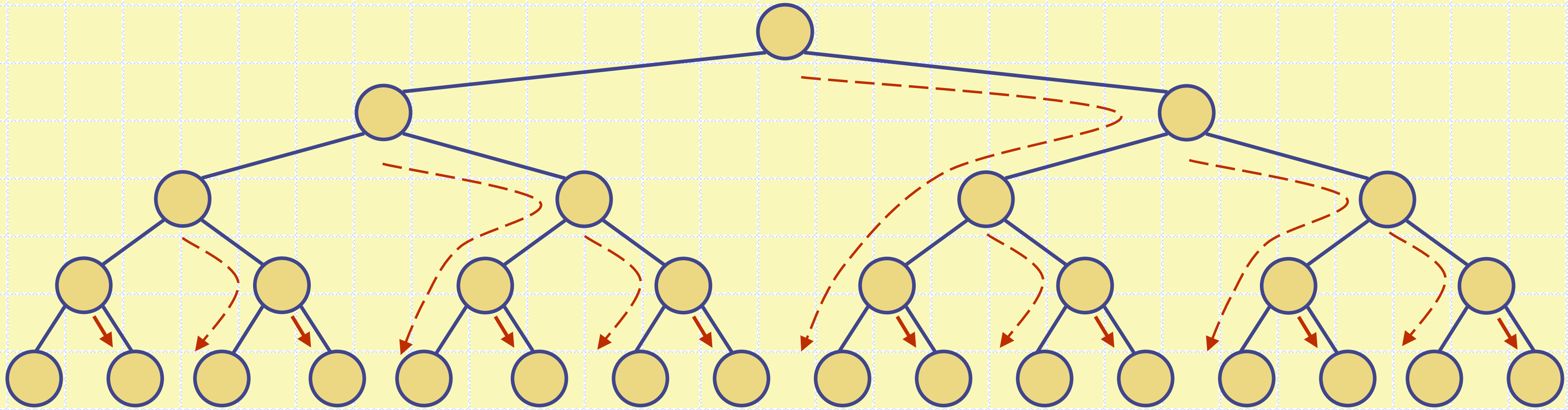
Final heap after applying down-heap bubbling

Analysis



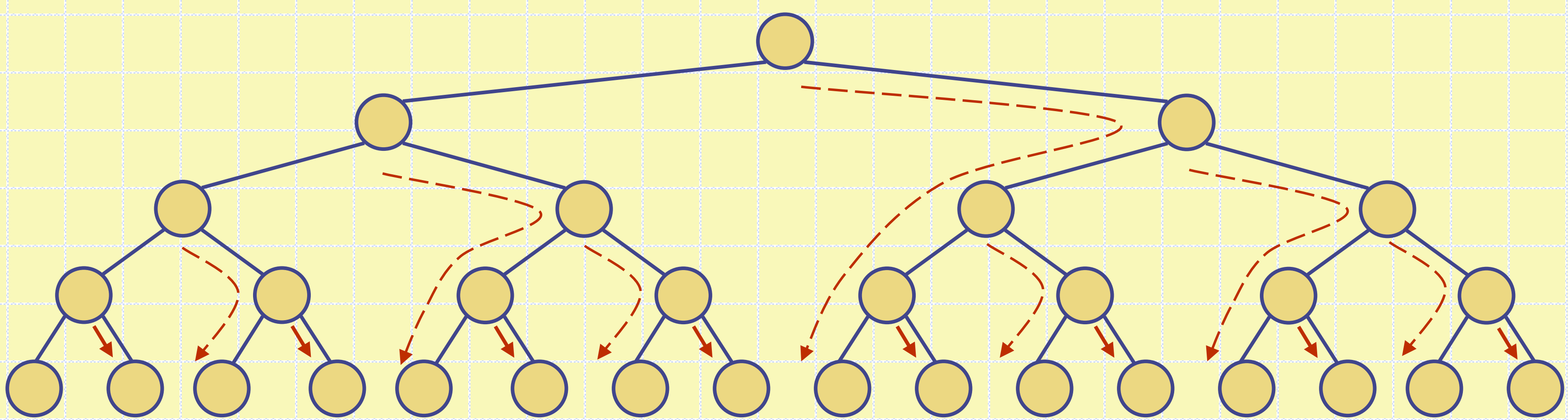
- Let us consider any internal node v. The construction of the tree rooted at v (subtree actually, except when v is the root) is proportional to the size of the paths of this tree.

- Which paths are associated with a node?
- An ***associated*** *path, $p(v)$*,with a node v is the one that goes to the right child of v then goes left downward until it reaches an external node (note that this path may differ from the actual downheap path).

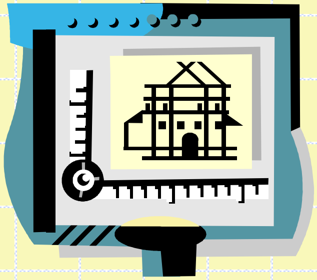


Analysis

- Clearly the size (number of nodes) of $p(v)$ is equal to the: height of the tree rooted at $v+1$.
- Each node in the tree belongs to at most 2 associated paths: the one rooted at v and the one coming down from its parent (note that the root and all the nodes on the left-most root-to-leaf path belong only to one associated path).



Analysis



- Therefore, the total sizes of the paths associated with the internal nodes of the tree is at most $2n - 1$.

- Consequently, the construction of the tree (all its paths) using bottom-up heap construction runs in $O(n)$ time.

➔ Bottom-up heap construction (while does not change the general complexity of heap-sort), it speeds up the first phase of the algorithm.