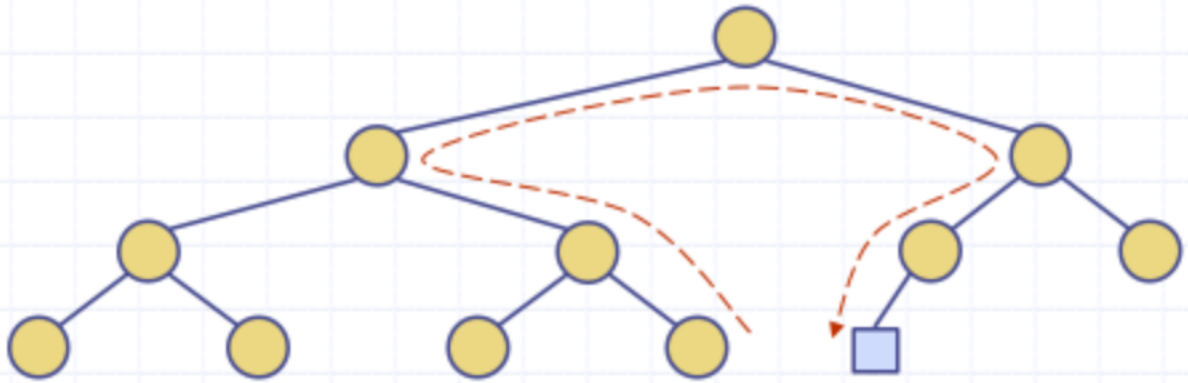
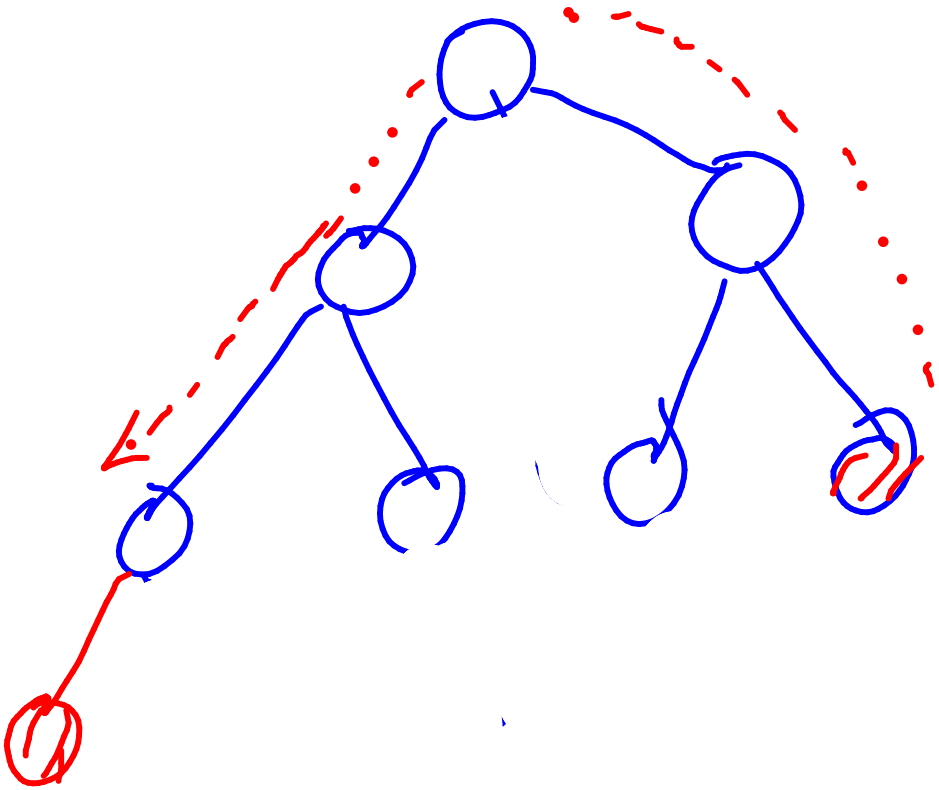
Updating the Last Node [for insertion]

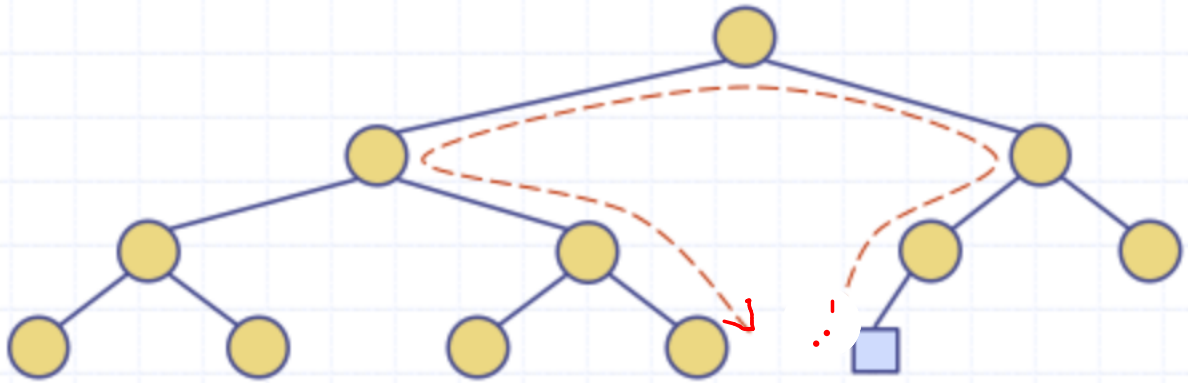
- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right sibling
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal





Updating the Last Node [for deletion]

- 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
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



Recap: (PQ using Heap)

insertion = $O(\log n)$ [upheap]

remove min = Deletion = $O(\log n)$ [down-heap]

get min = $O(1)$ [get the root]

Naive (not-inplace) implementation of sorting using Heap PQ:

$\log 1 + \log 2 + \cdots \log n$ } insertion

$+ \log n + \log(n-1) + \cdots \log(1)$ } remove min

$= 2 \log \lfloor n \rfloor = O(n \log n)$

Can be shown that

$\log \lfloor n = O(n \log n)$

[We have assumed that update last is $O(1)$ when heap is implemented using an array. However, even otherwise, you will need to visit only $O(\log n)$ nodes if you follow procedure for updating last pointer described earlier]

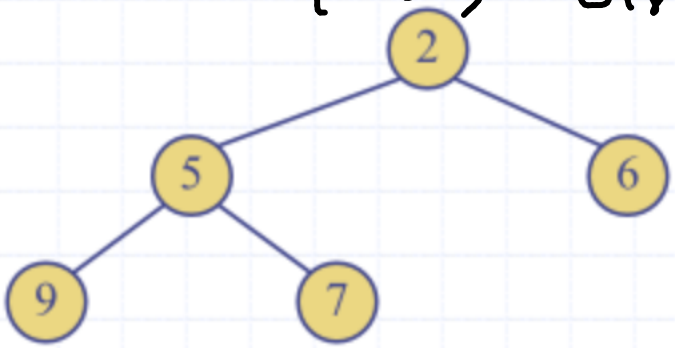
Heap-Sort (§2.4.4)

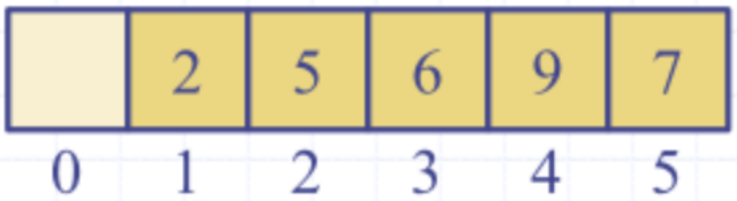
- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Vector-based Heap Implementation (§2.4.3)

Growable array using some doubling (etc) strategy

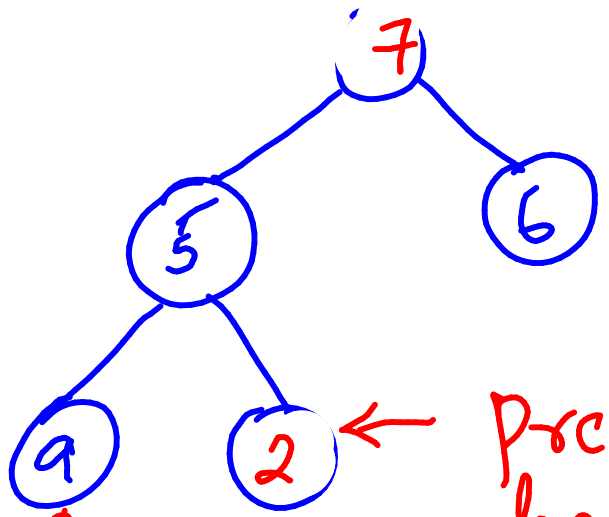
- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting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





In-place heap sort (swap ② & ⑦)

_ 2 5 6 9 7



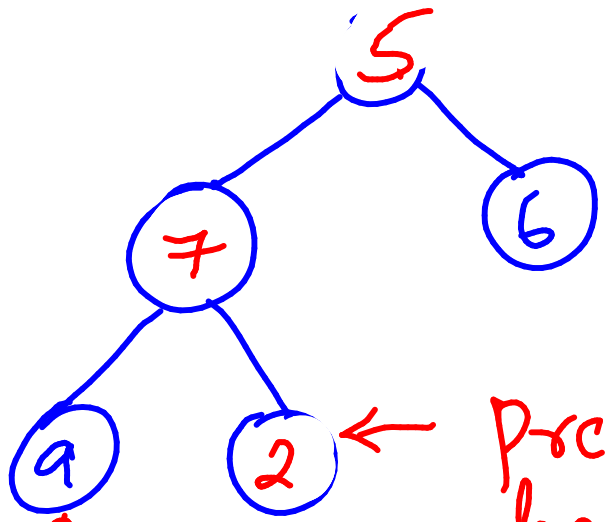
Assume an initial heap on the keys

← pretend that the heap does not contain ②

update last pointer

Down heap on 7

_ 2 5 6 9 7

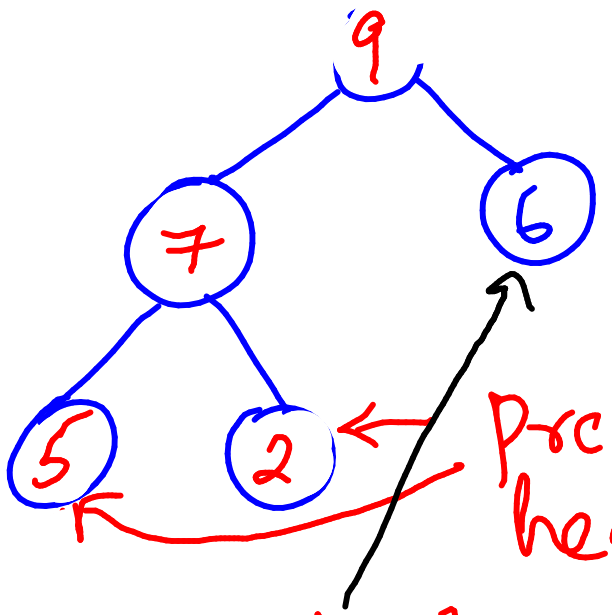


← pretend that the heap does not contain (2)

update last pointer

Swap 5 & 9

_ 2 5 6 9 7

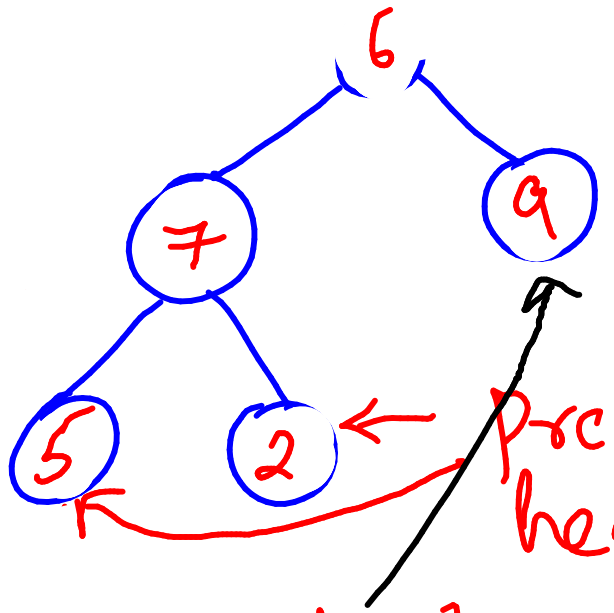


pretend that the heap does not contain (2) & (5)

update last pointer

Downheap on 9

_ 2 5 6 9 7

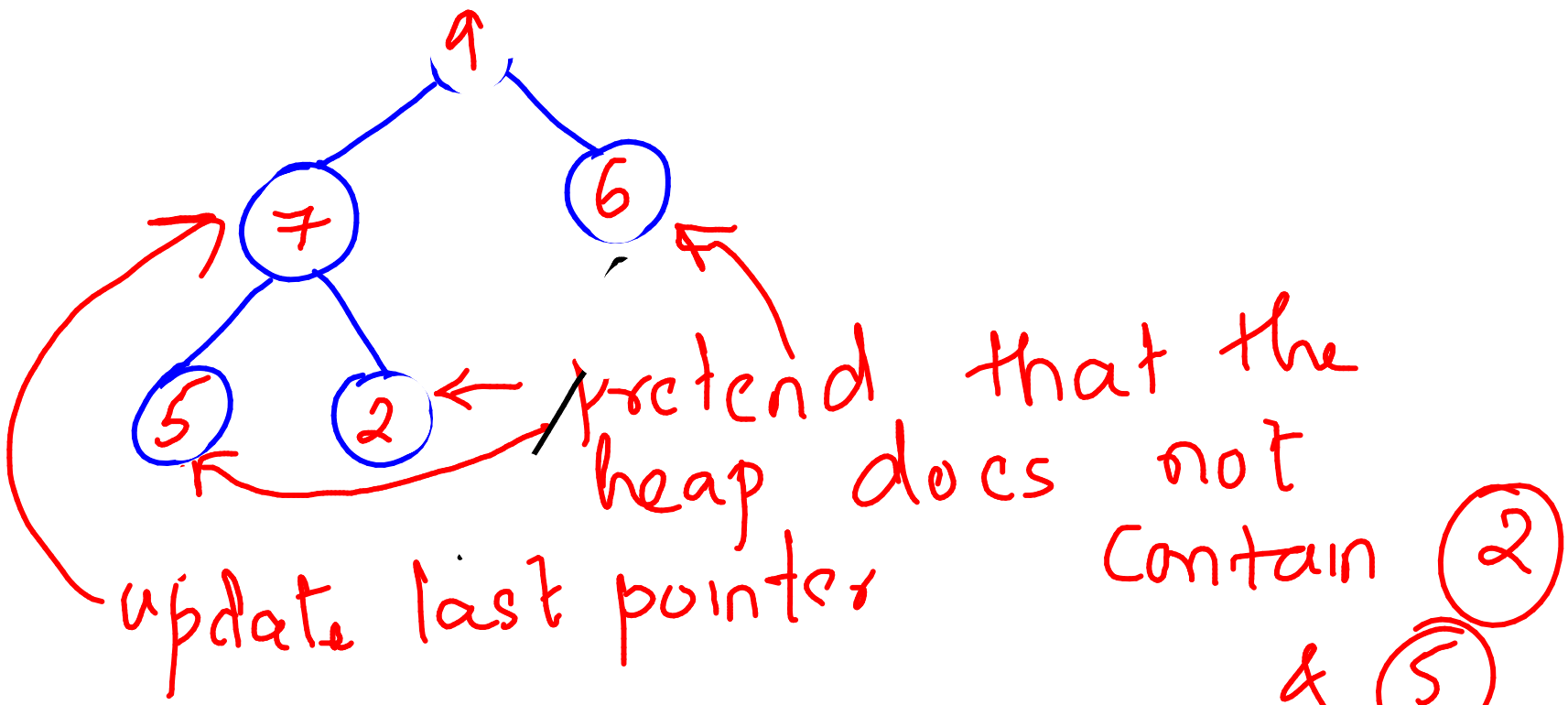


pretend that the heap does not contain (2) & (5)

update last pointer

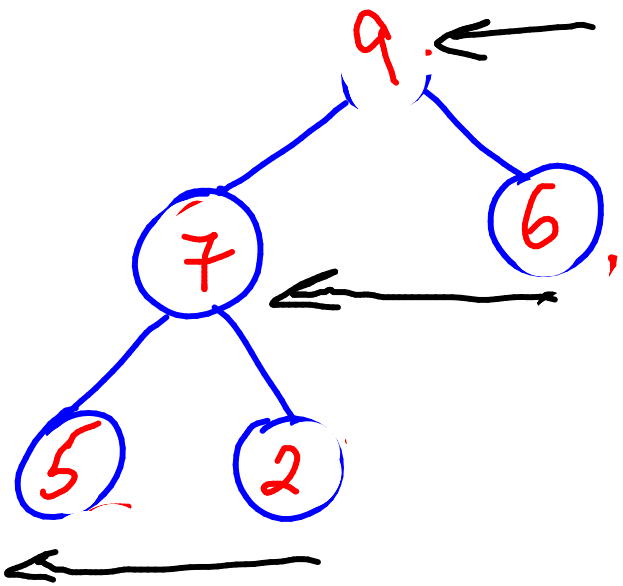
swap ⑥ & ⑨

2 5 6 9 7



....Finally!

2 5 6 9 7

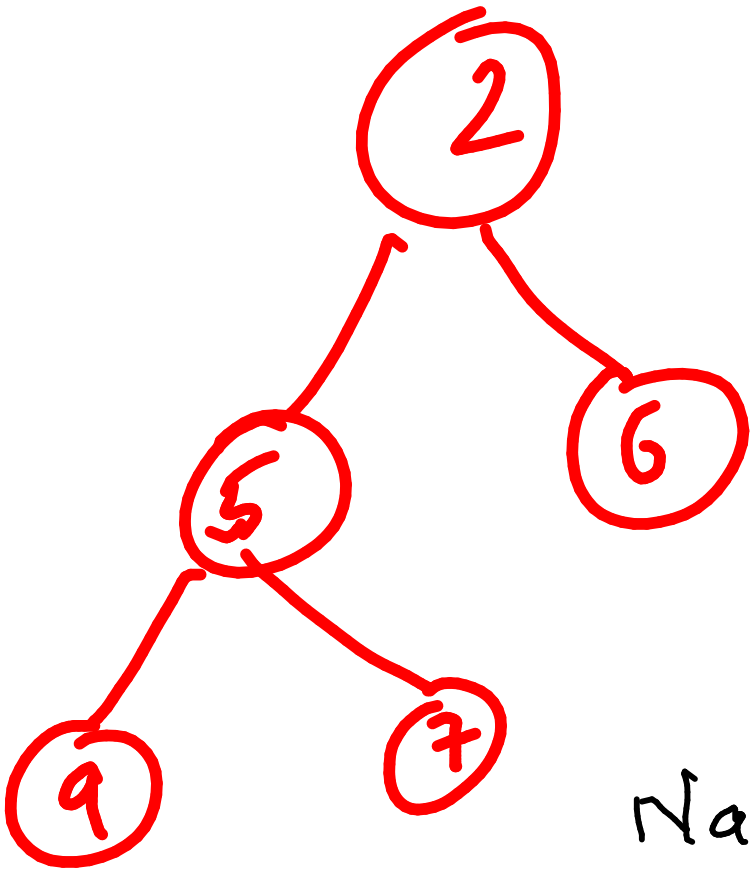


9 7 6 5 2

(in the array!)

ie n-place Heap sort

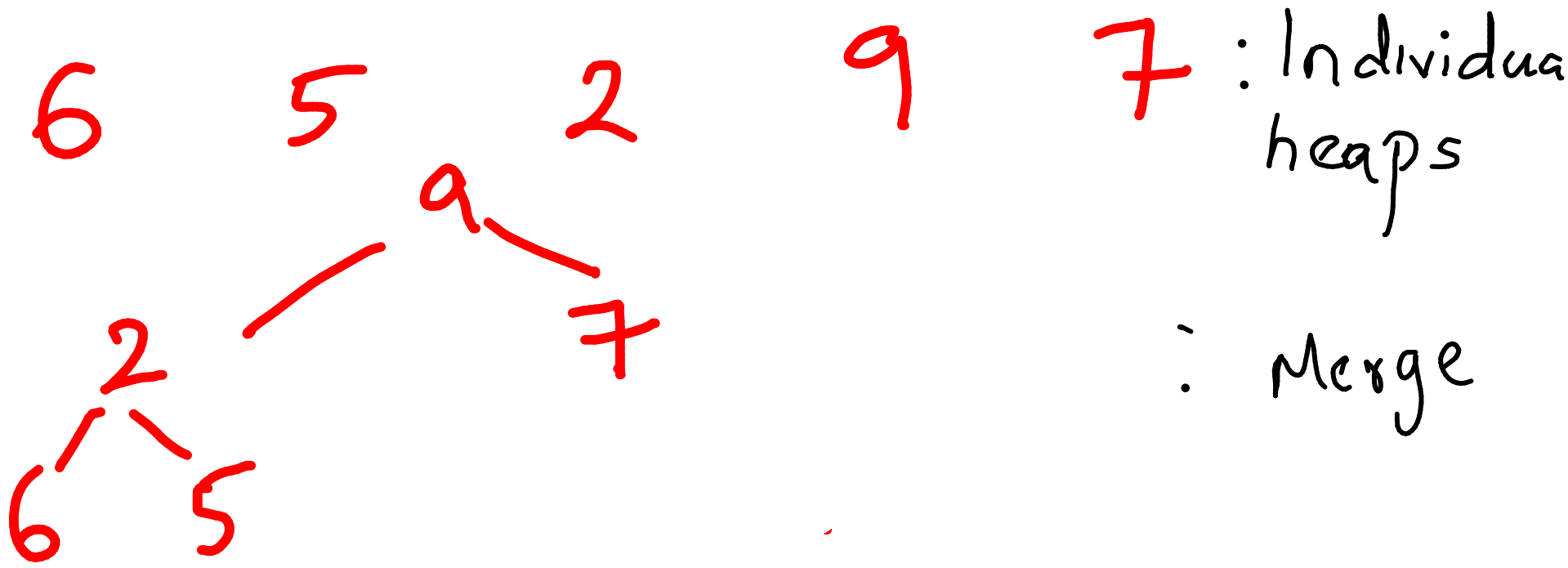
Question: How to generate initial heap



from

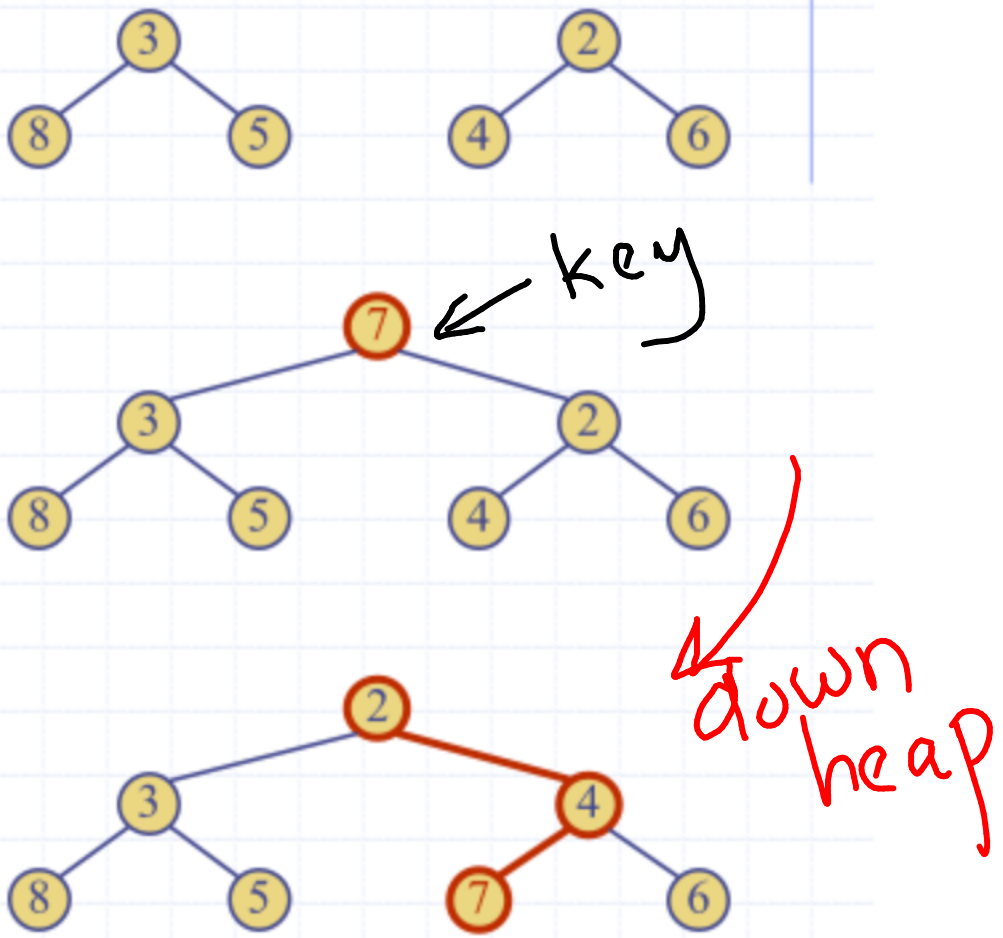
6 5 2 7 9

Naive: Insert 6 then 5 ... then 9


6
5
2
9
7
: Individua
heaps
9
2
7
6
5
: Merge

Merging Two Heaps

- We are given two 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



∴ the structural prop not violated

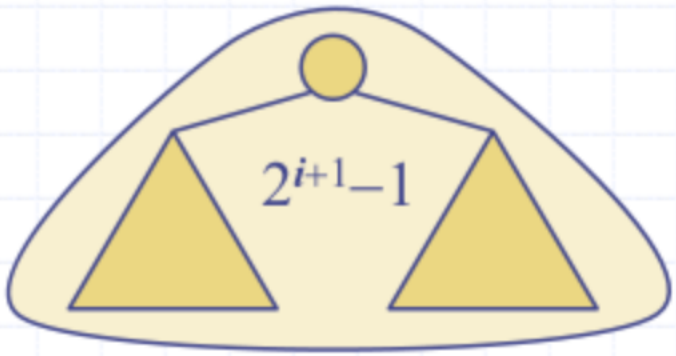
Assume: Left heap is "complete" & right heap is left compressed

Bottom-up Heap Construction (§2.4.3)

(collectively without inserting one at a time)

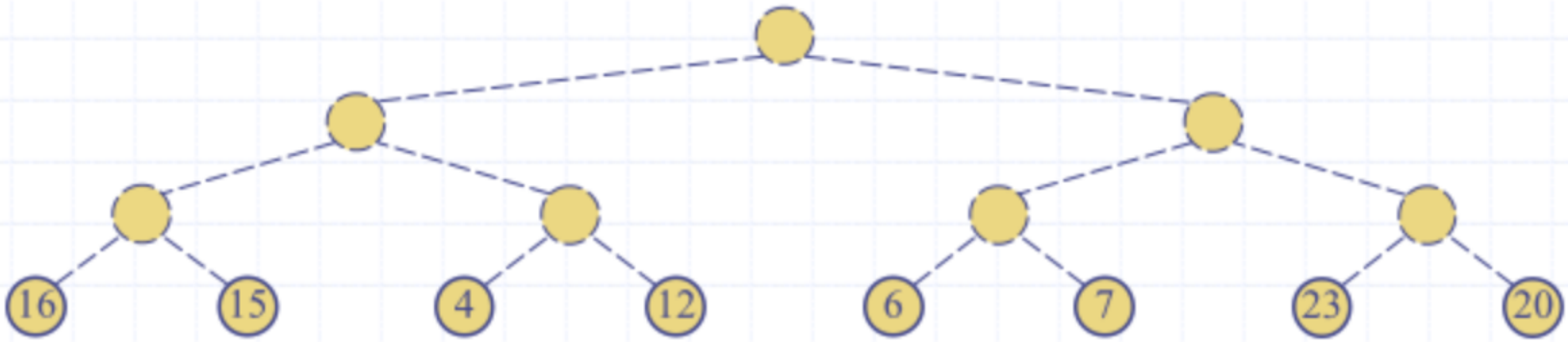
- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

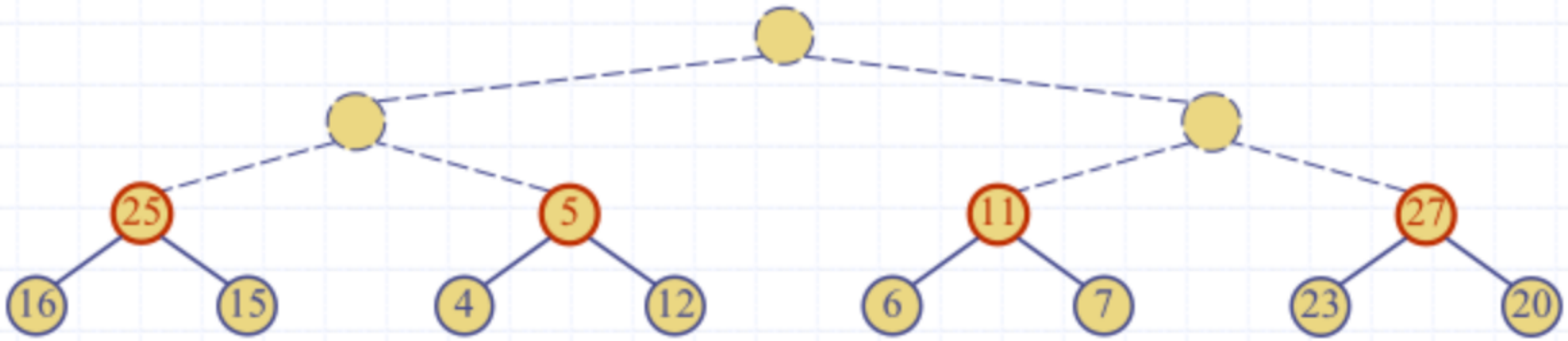
⇩ (downheap)



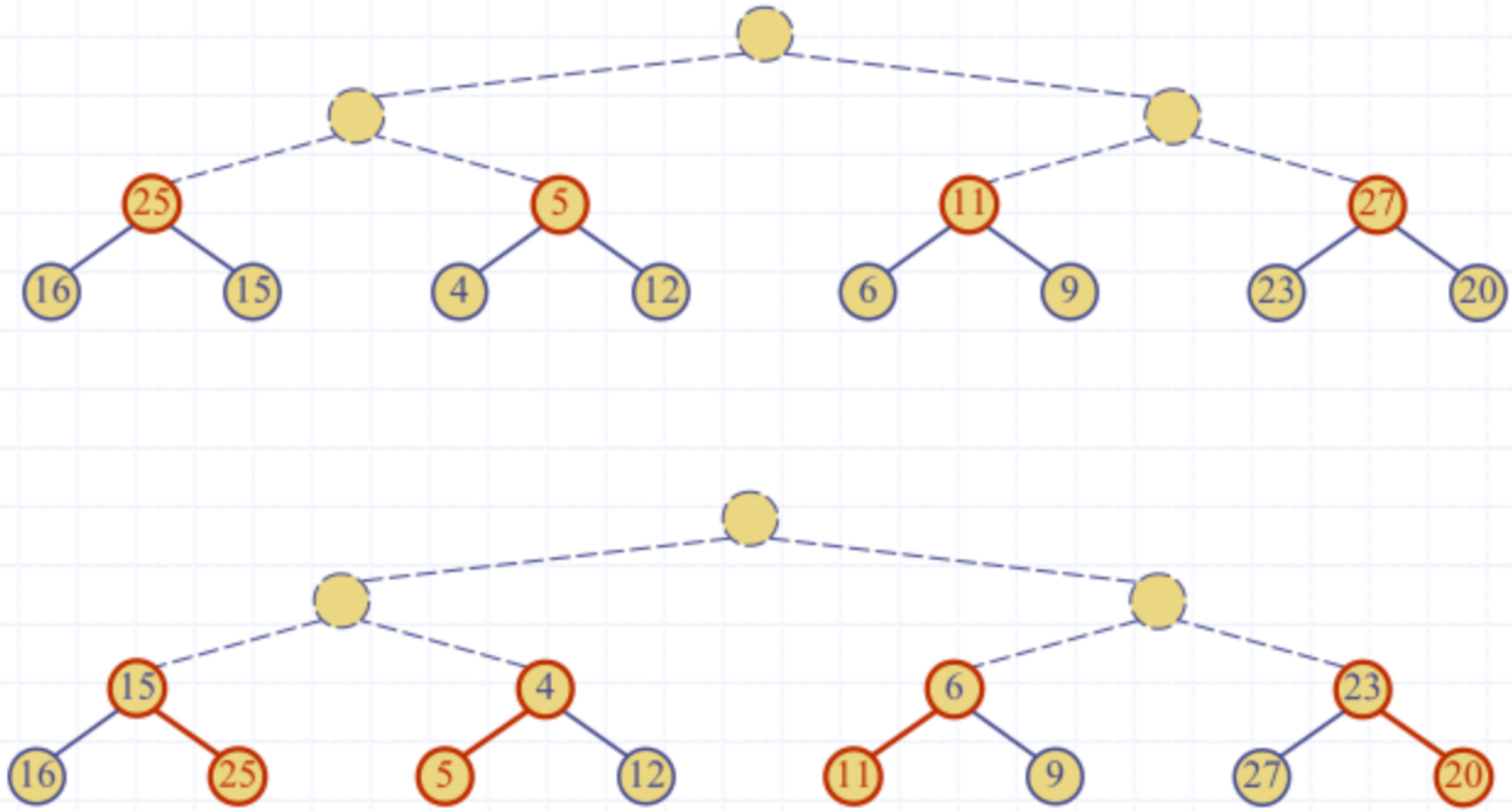
$= O(i)$

Example

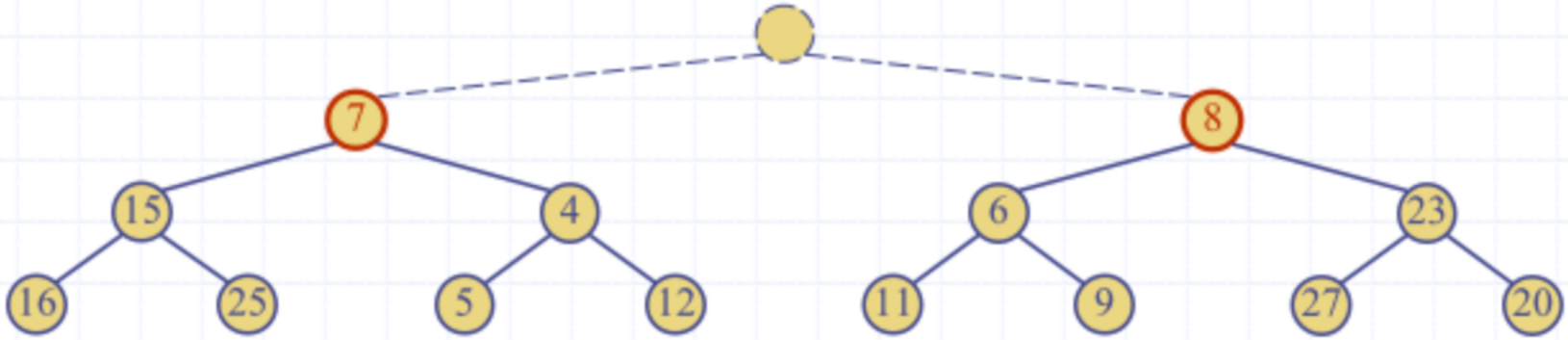


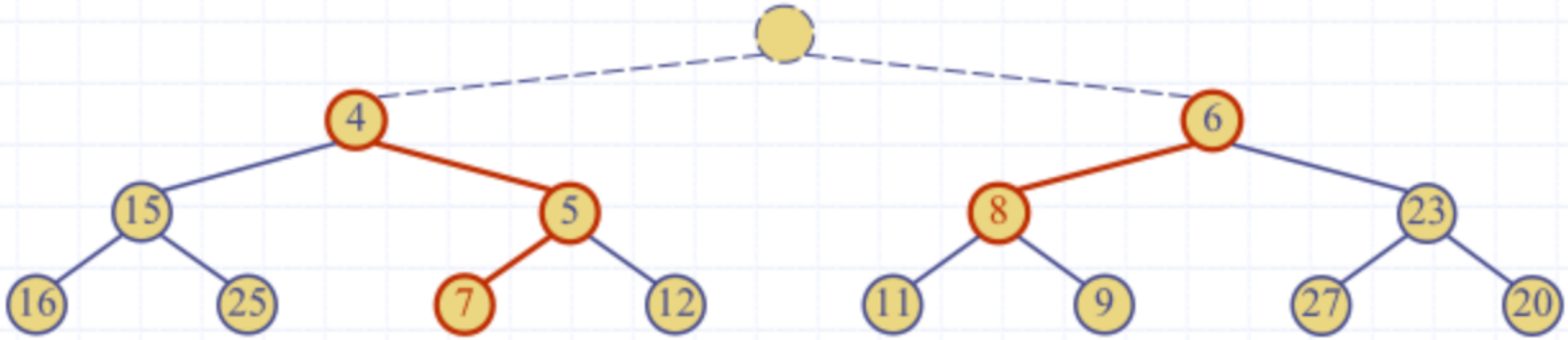


Example (contd.)

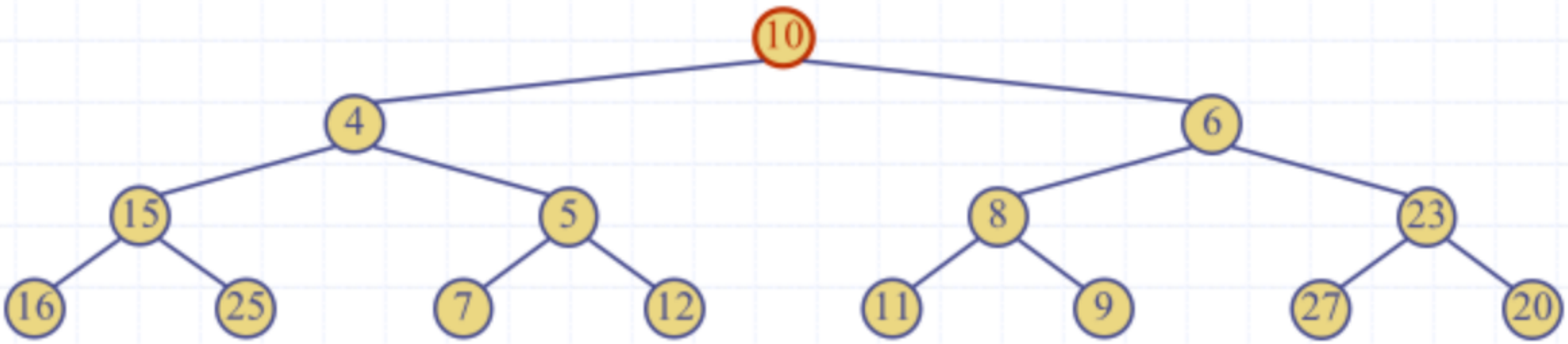


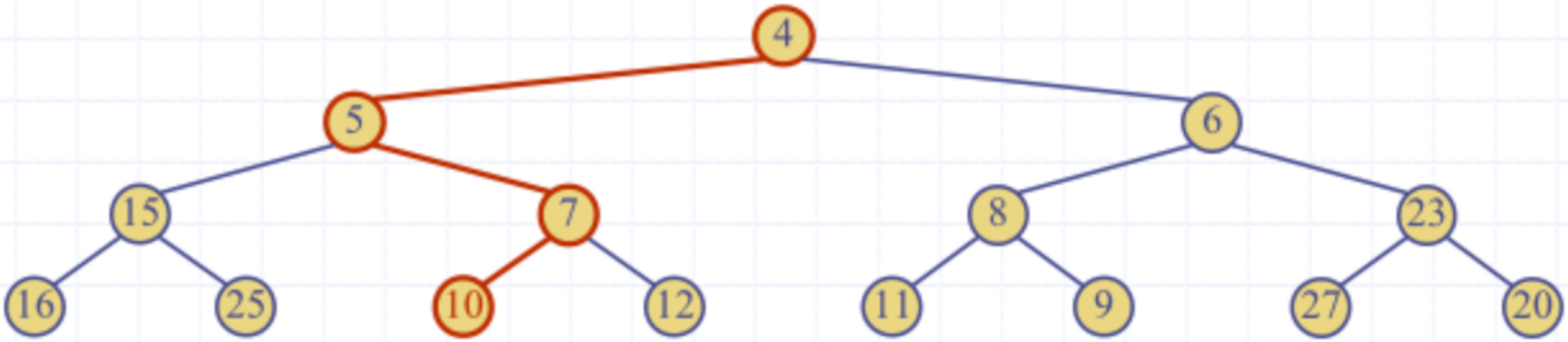
Example (contd.)





Example (end)



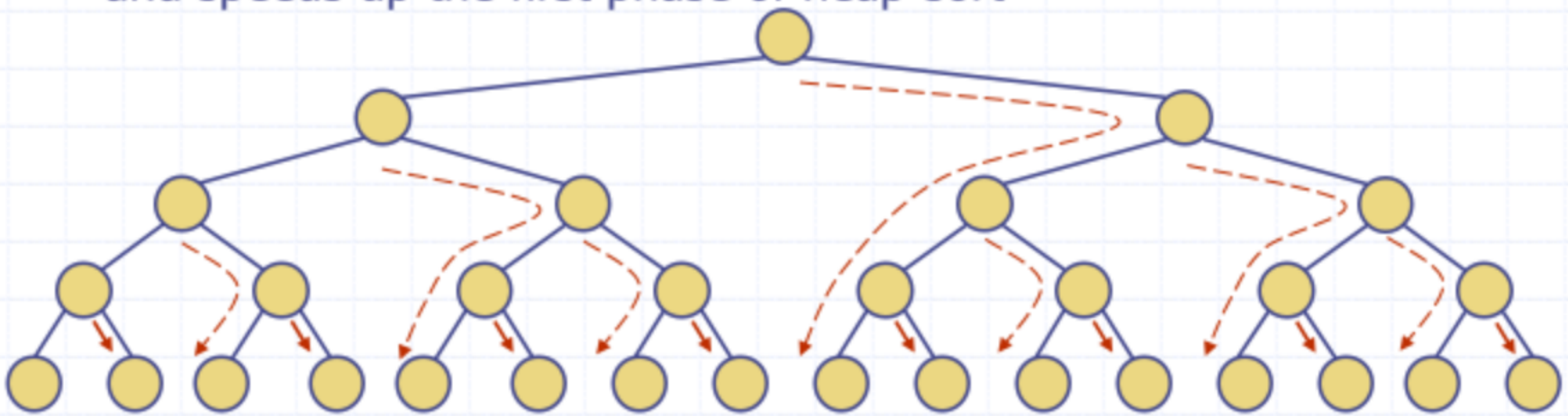


Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort

H/w

→ downheap paths



Ans: Constant

H/w: In bottom-up heap construction, what is max # of accesses to a key?