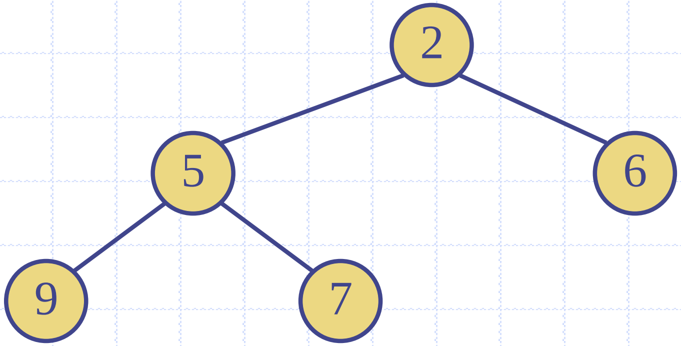
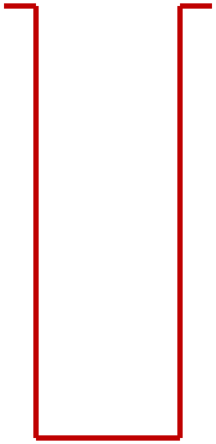
Heaps

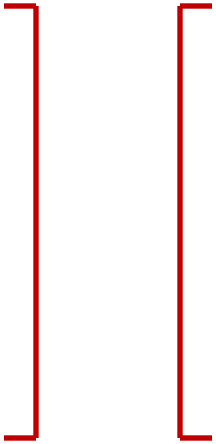


Comparison

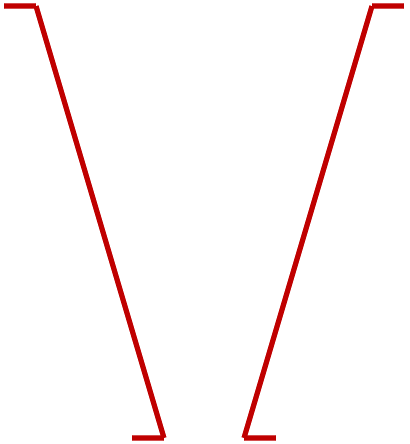
- Stack



- Queue



- Priority queue



Priority Queue ADT

- A priority queue (PQ) stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e) inserts an entry e
  - removeMin()
    removes the entry with smallest key
- Additional methods
  - min(), size(), empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

PQ 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 using selection sort: $O(n^2)$ time
  - Sorted sequence using 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.empty*** ()

        e ← ***S.front***(); ***S.eraseFront***()

        P.insert (***e***, ∅)

    while ¬***P.empty***()

        e ← ***P.removeMin***()

        S.insertBack(***e***)

Two Paradigms of PQ Sorting

- PQ via selection sort
- PQ via insertion sort

		List L	*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)	()

		List L	*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)	()

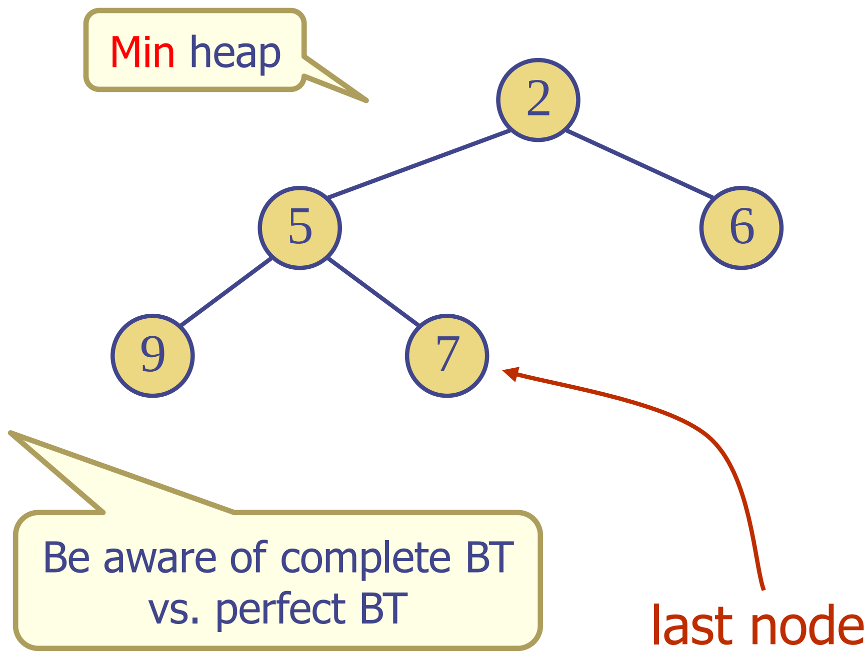
Figure 8.1: Execution of selection-sort on list $L = (7,4,8,2,5,3,9)$.

Time complexity

Operation	*Unsorted List*	*Sorted List*
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Heaps

- 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, $\boldsymbol{key}(\boldsymbol{v}) \geq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$
- Complete binary tree: let $\boldsymbol{h}$ be the height of the heap
  - for $\boldsymbol{i} = 0, \ldots, \boldsymbol{h}-1$, there are $2^{\boldsymbol{i}}$ nodes of depth $\boldsymbol{i}$
  - at depth $\boldsymbol{h}-1$, 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

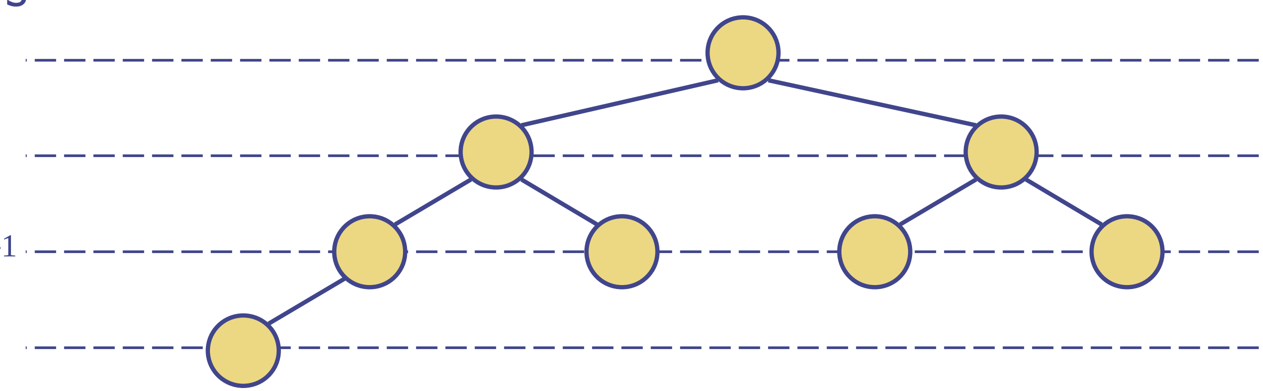
- **Theorem:** A heap storing n keys has height $O(\log n)$

  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - There are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h ➔ $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h$ ➔ $h \leq \log n$

$$
\begin{aligned}
&2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1 \leq n \leq 2^0 + 2^1 + 2^2 + \cdots + 2^h \\
&\Rightarrow 2^h \leq n \leq 2^{h+1} - 1 \\
&\Rightarrow \log_2(n+1) - 1 \leq h \leq \log_2(n)
\end{aligned}
$$

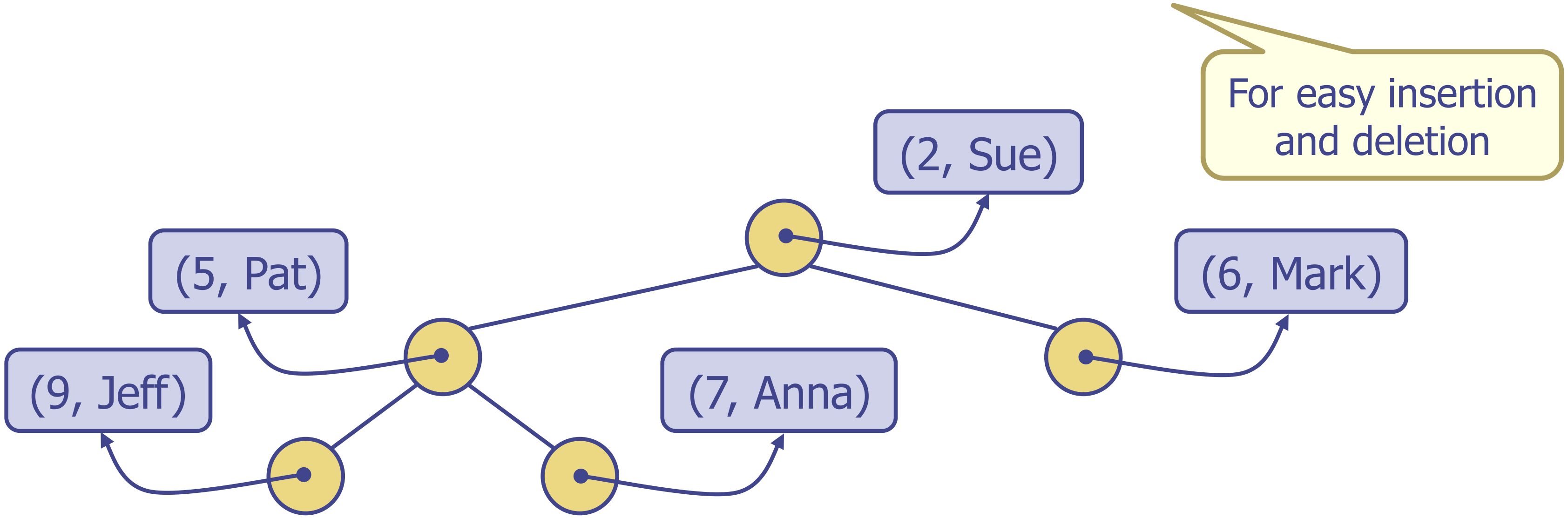
Quiz!

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



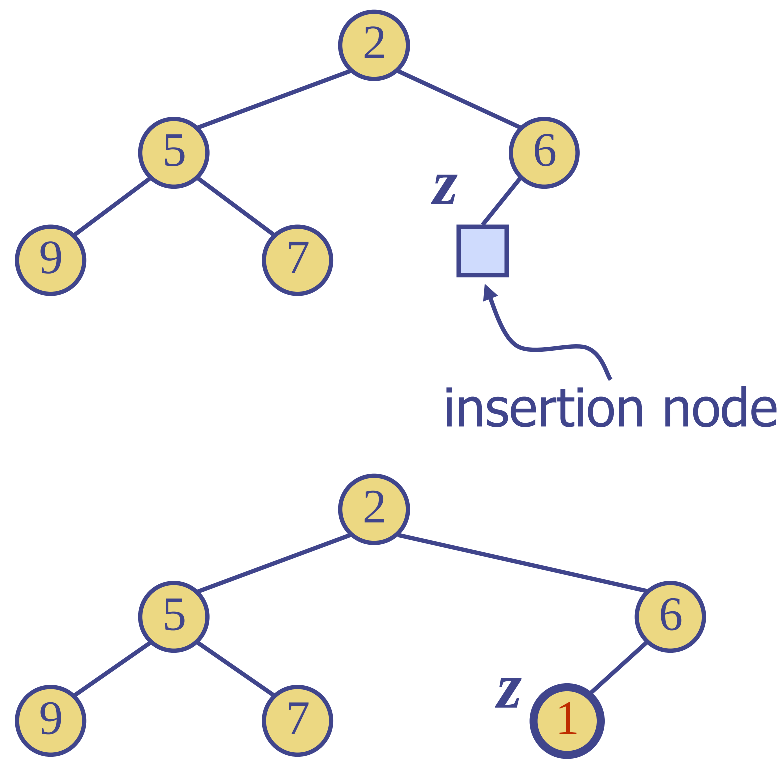
Heaps and Priority Queues

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



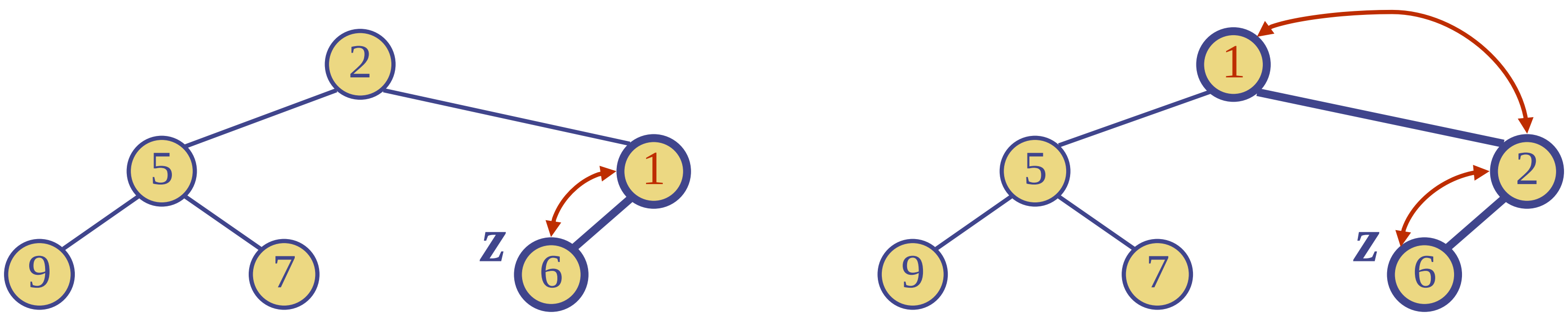
Insertion into a Heap

- Three steps to insert an item of key k to the heap:
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



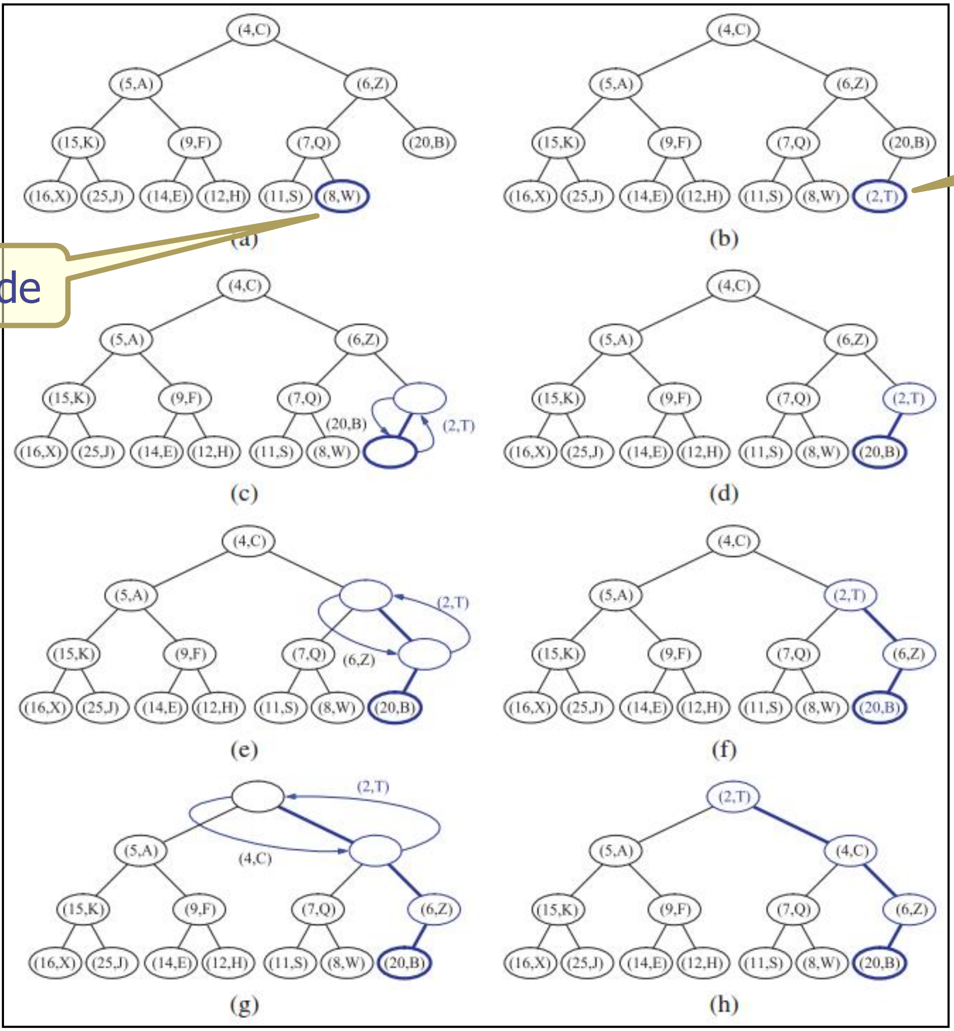
Upheap

- After insertion, 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



Upheap: More Example

Quiz!



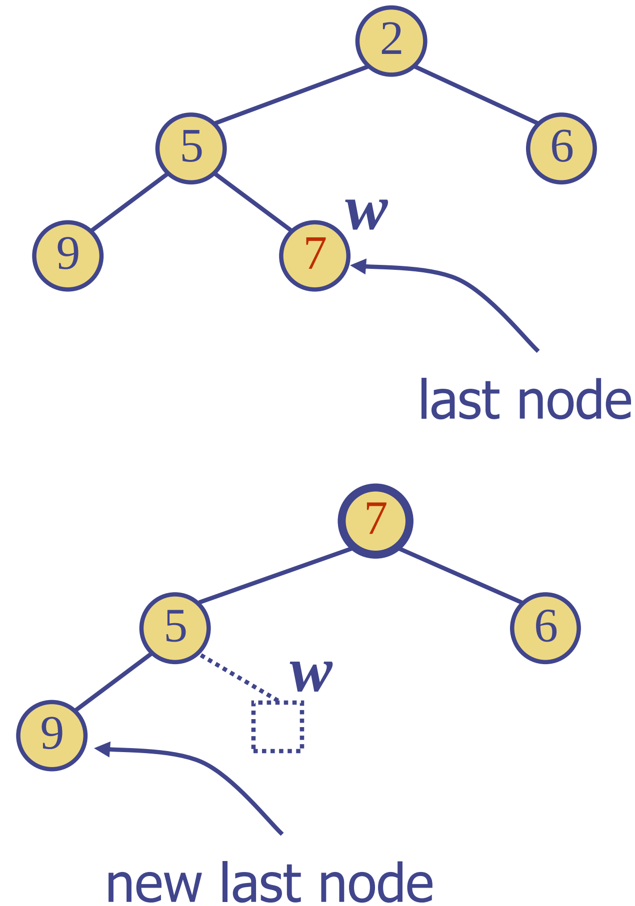
Newly added node

Original last node

Up-heap bubbling

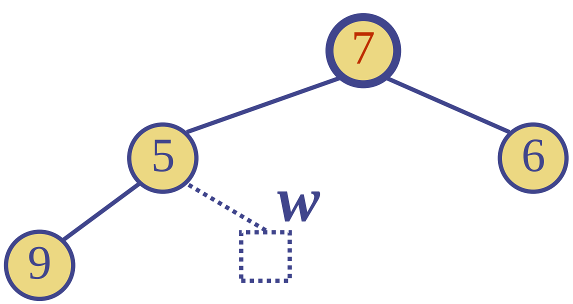
Removal from a Heap (§ 7.3.3)

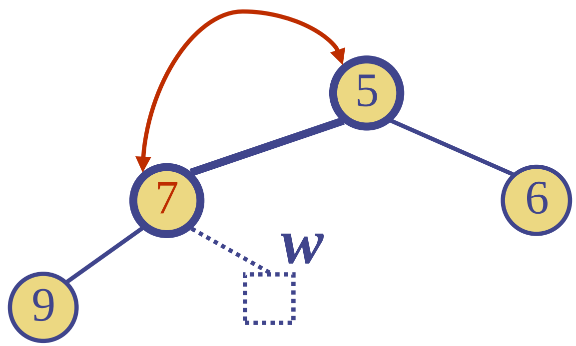
- Three steps to remove the minimum from a heap:
  - Replace the root key with the key of the last node *w*
  - Remove *w*
  - Restore the heap-order property (discussed next)



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
- Downheap 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



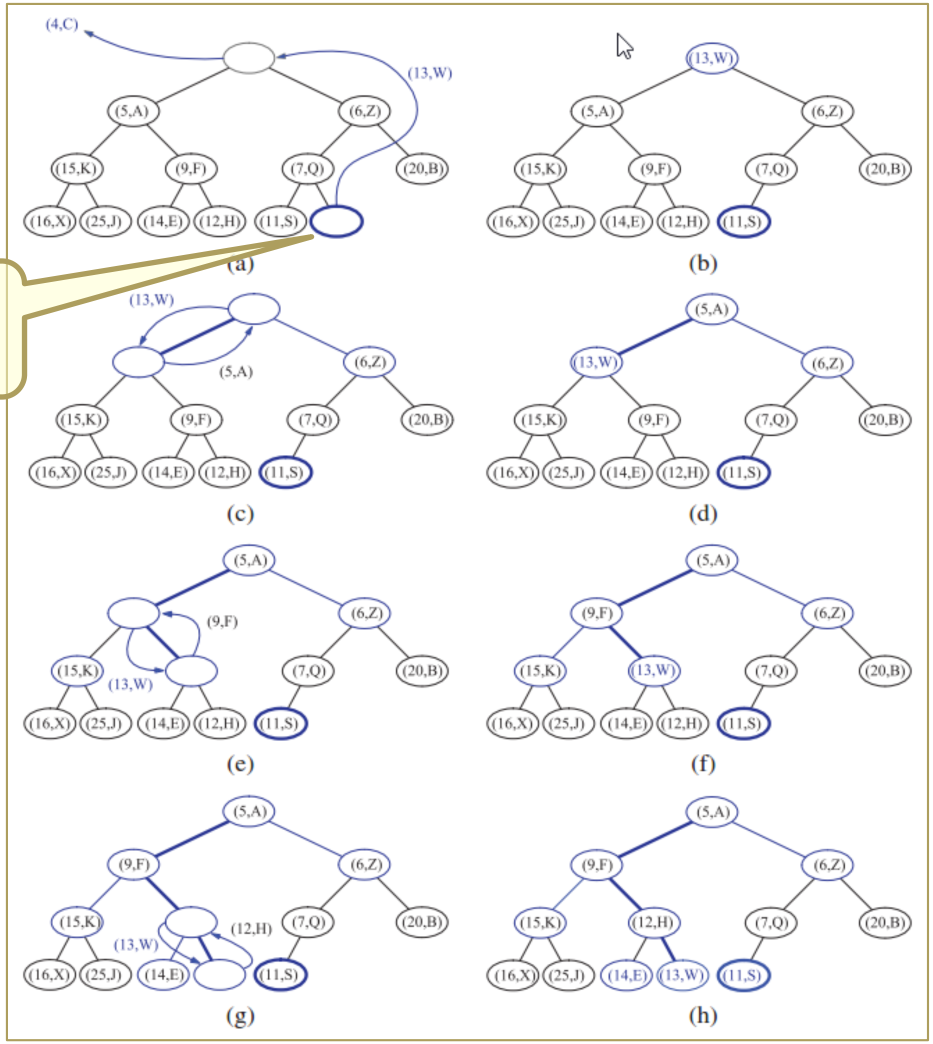


Operation	*Time*
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Quiz!

Downheap: More Example

Quiz!



Move the last to top

Down-heap bubbling

Summary of Basic Principle

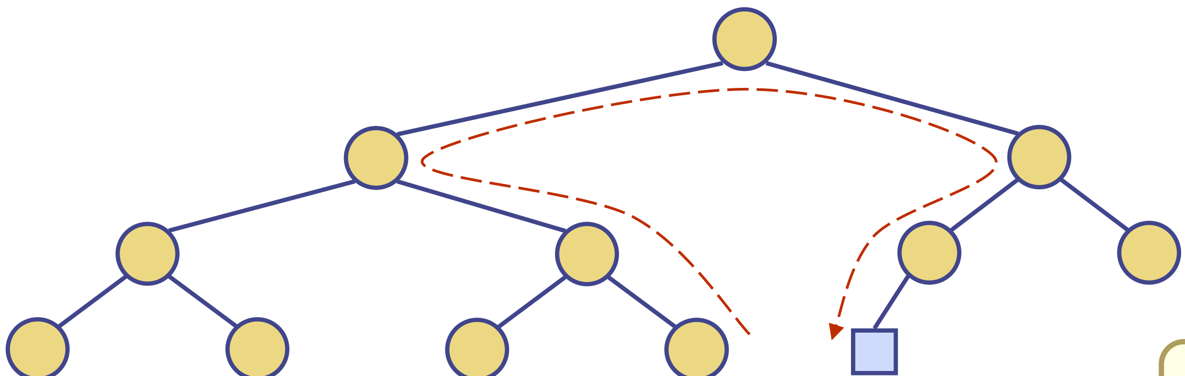
- For both "insertion" and "remove min"
  - Keep heap as complete binary tree
  - Restore heap order

Quiz

- Draw the min-heap tree after each operations:
  1. insert 7
  2. insert 4
  3. insert 1
  4. insert 5
  5. delete-min
  6. insert 9
  7. insert 2
  8. insert 3
  9. delete-min
  10. delete-min

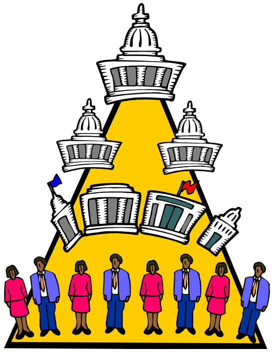
Locating the Node to Insert

- For linked list based implementation of trees, the inserted 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



This is easy for vector-based heaps!

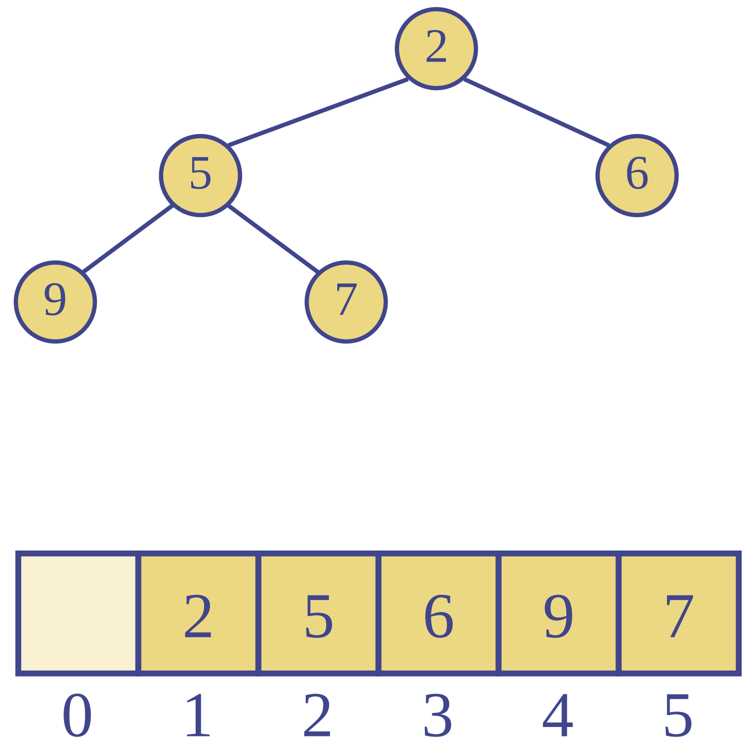
Heap Sort



- Consider a priority queue with n items implemented by a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, empty, and min take time $O(1)$ time
- Heap-sort
  - Sort a sequence of n elements in $O(n \log n)$ time using a heap-based PQ
  - Much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Vector-based Heap Implementation

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



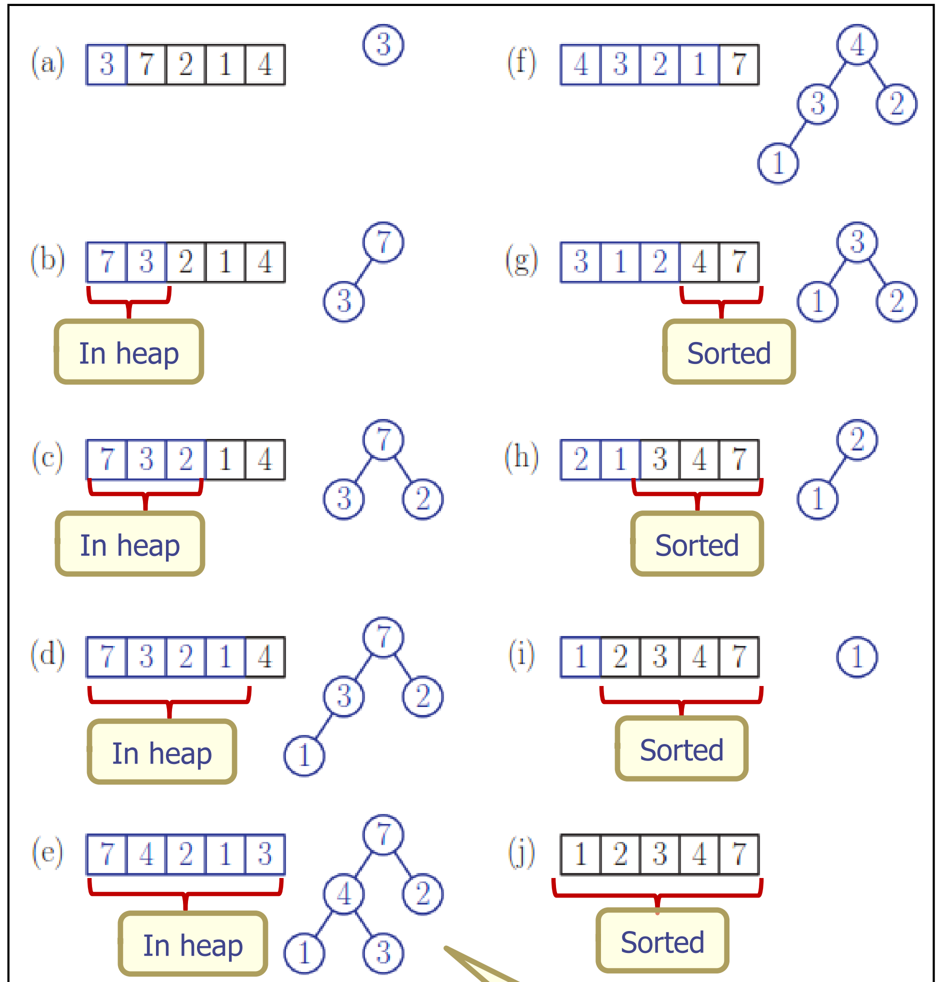
In-place Heap Sort

Quiz!

Stage 1: Heap construction

Stage 2: output

Trick: It's easier to draw the tree first!



Max heap

Quiz

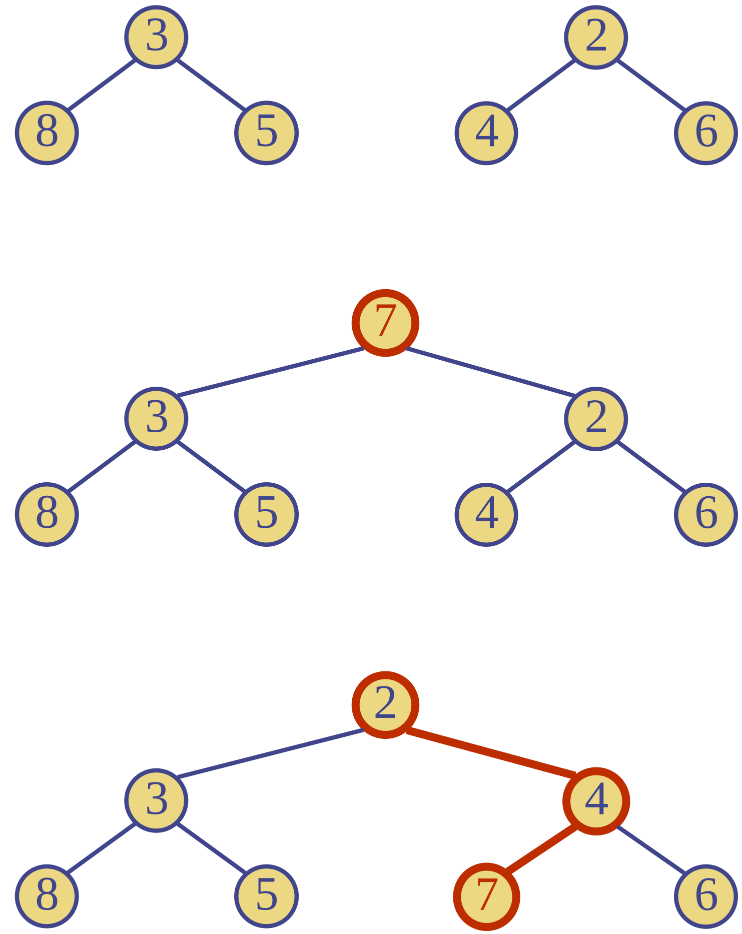
- How to perform in-place heap sort for a given vector x=[3 8 5 1 9]

Heap Sort Demo

- Interactive demo of heap sort
  - http://algoviz.org/OpenDSA/Books/OpenDSA/html/Heapsort.html#

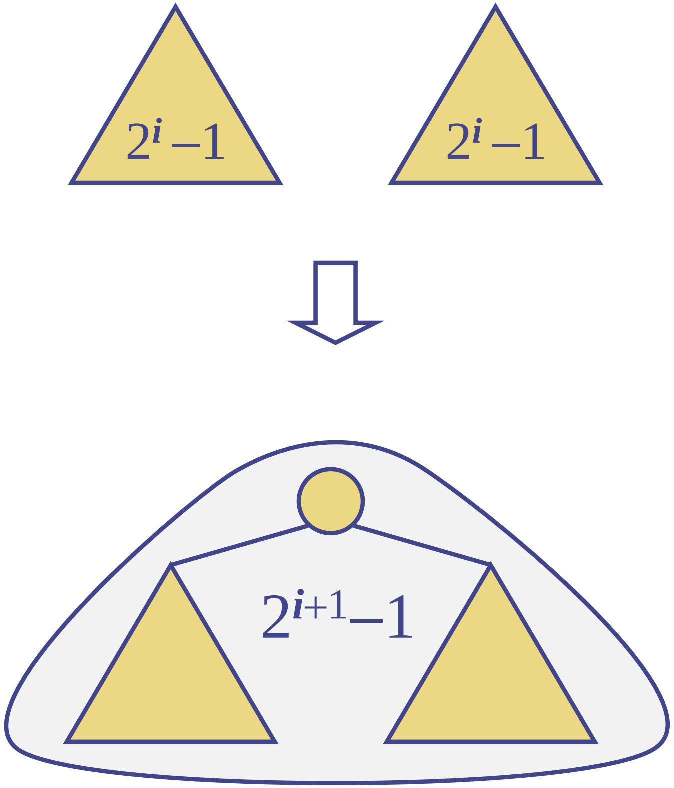
How to Merge Two Heaps

- Given two heaps and a key k
- Create a new heap with the root node storing k and the two heaps as subtrees
- Perform downheap to restore the heap-order property

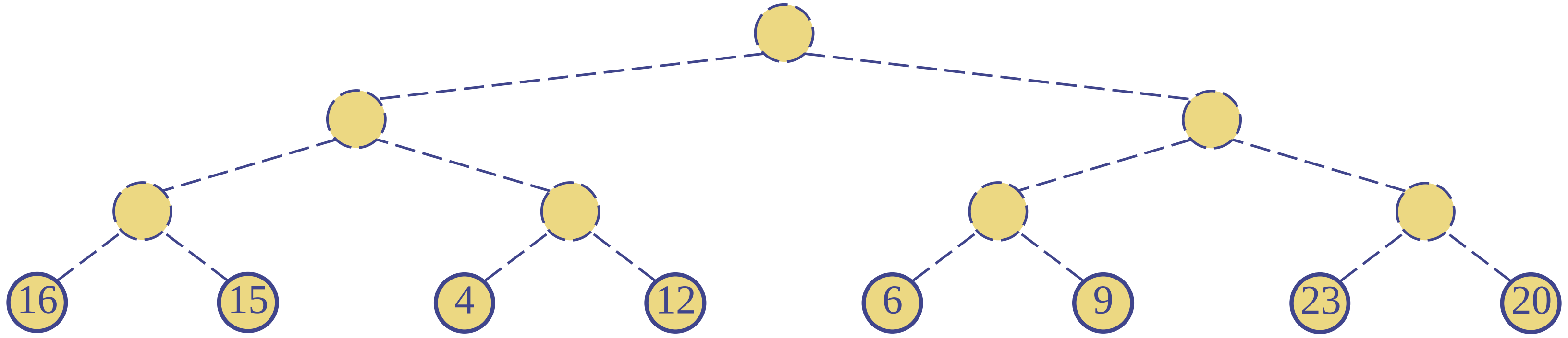


Bottom-up Heap Construction

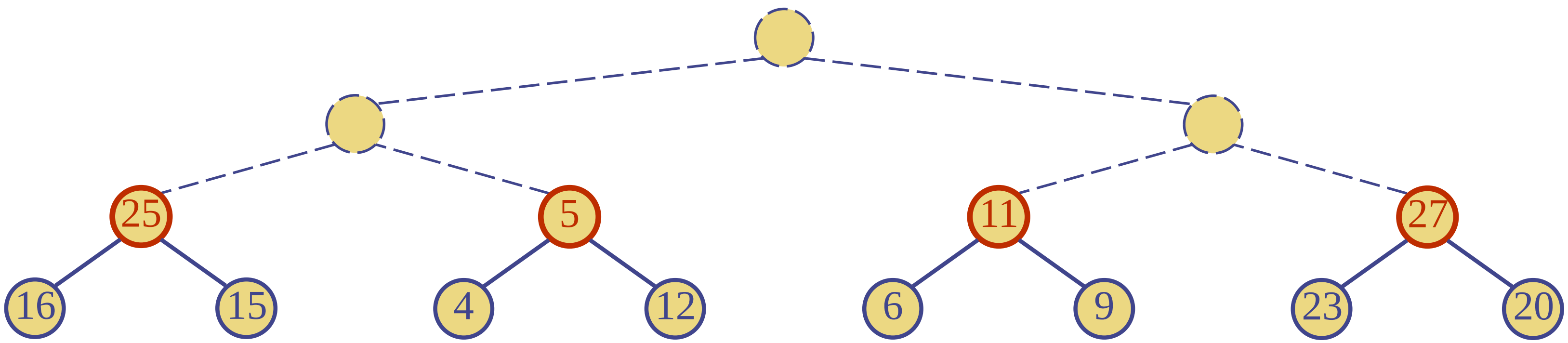
- Goal: Construct a heap of n keys using a bottom-up method with $O(n)$ complexity
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

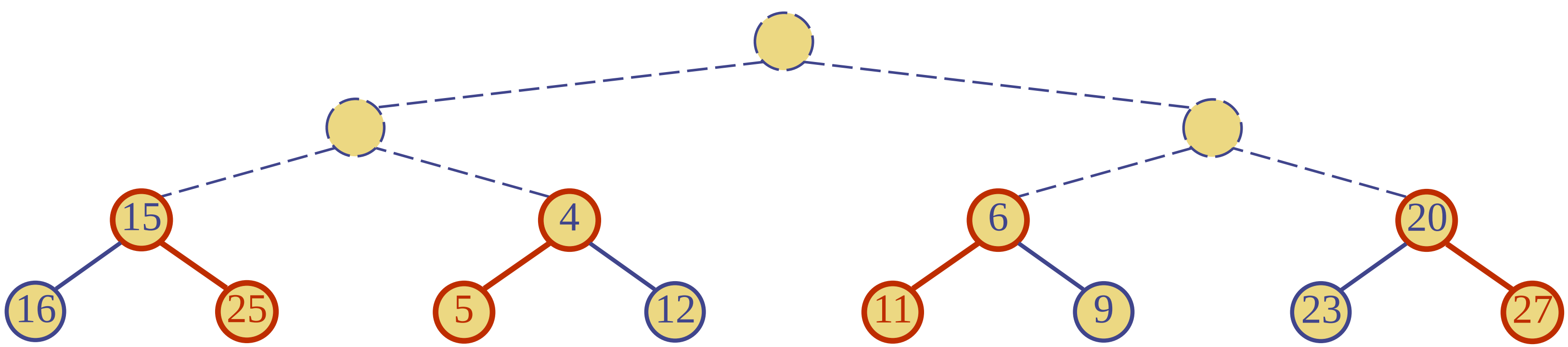


Example

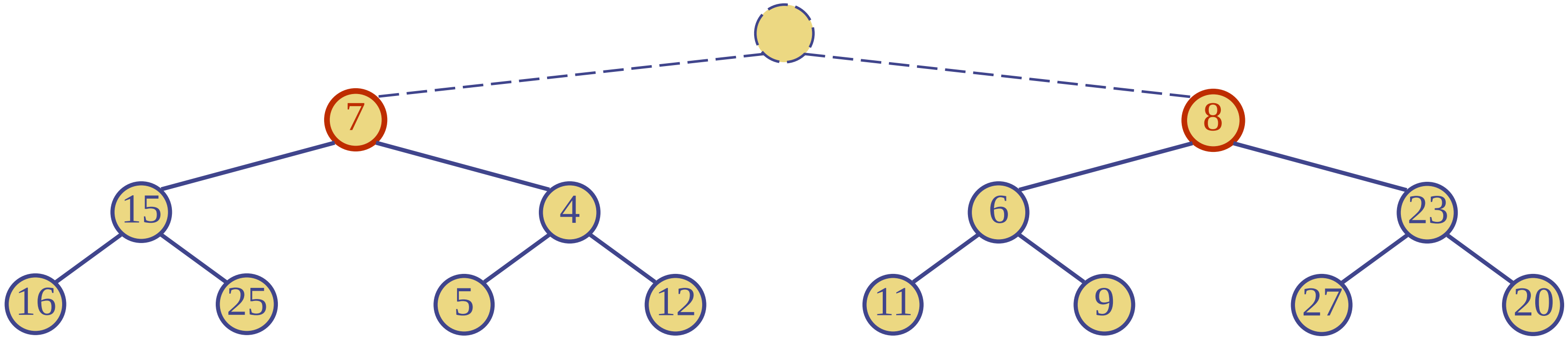


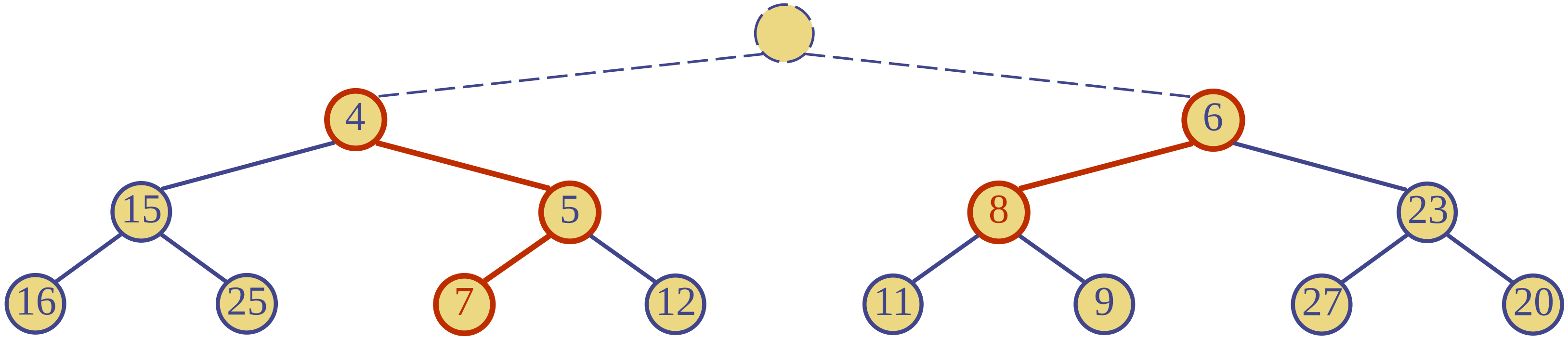
Example (contd.)



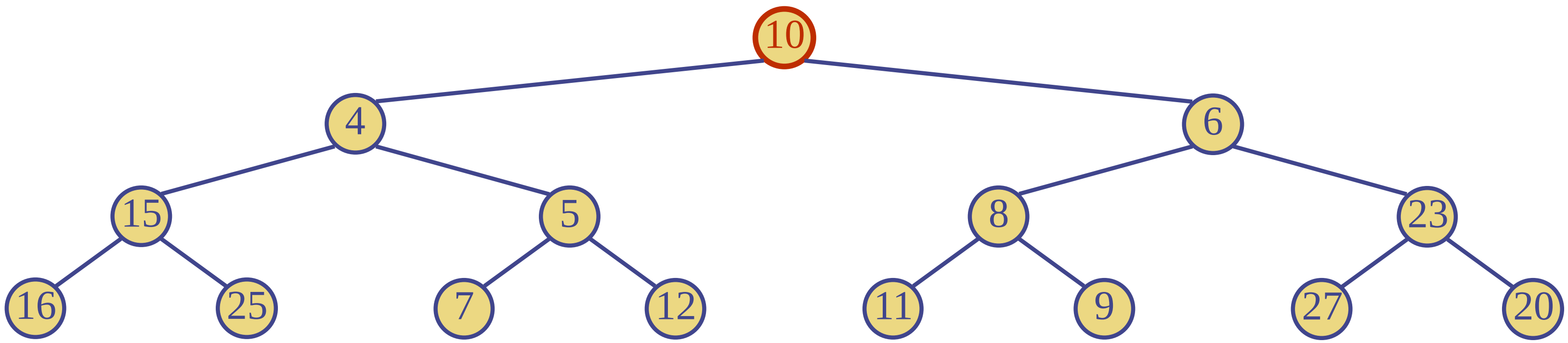


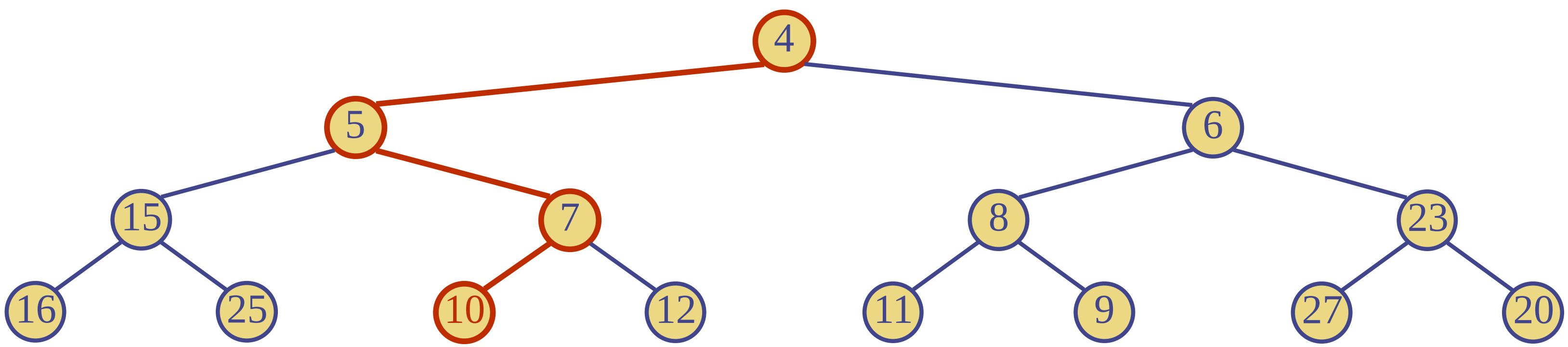
Example (contd.)



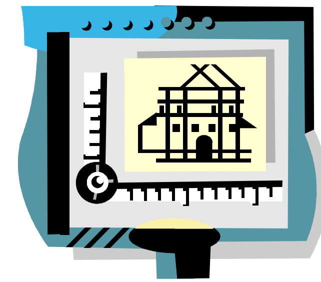


Example (end)





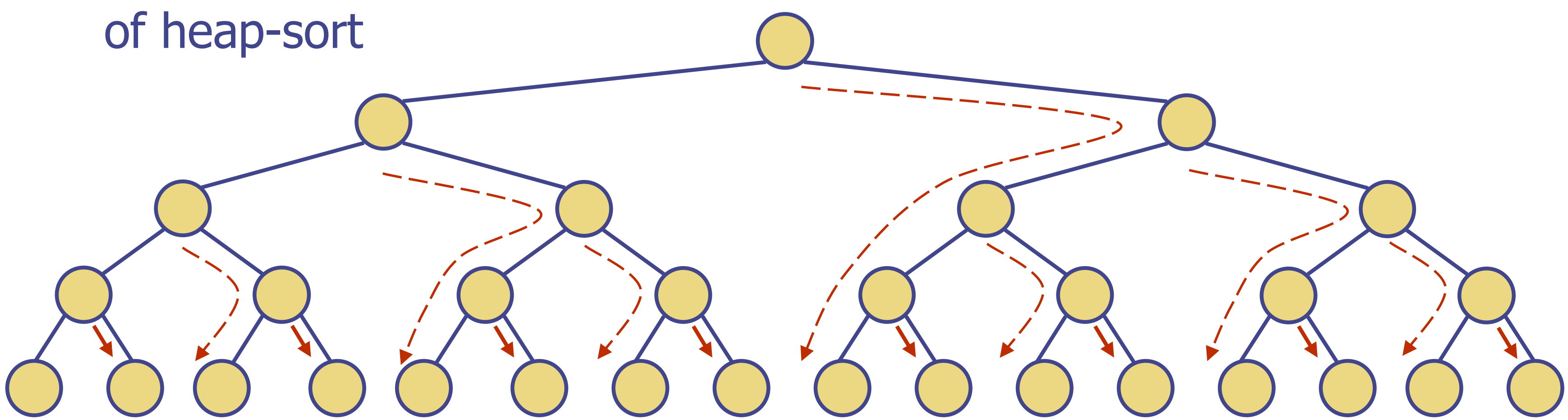
Analysis via Visualization

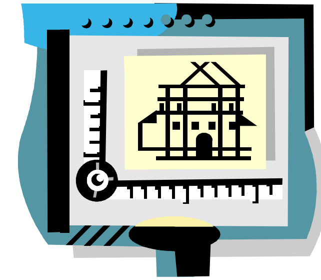


- Cost of combining two heaps = Length of the path from the new root to its inorder successor (which goes first right and then repeatedly goes left until the bottom of the heap)
- The path to inorder successor may differ from the downheap path.
- Each node is traversed by at most two such paths ➔ Total number of nodes of the paths is $2(2^h-1)-h = O(n)$ ➔ Bottom-up heap construction runs in $O(n)$ time

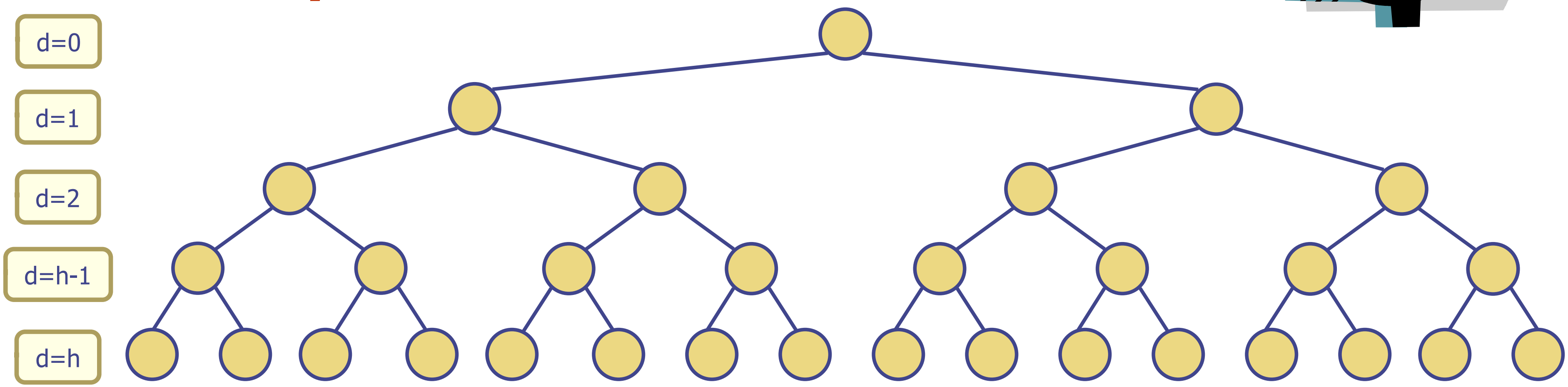
No. of internal nodes = 2^h-1

- Faster than n successive insertions and speeds up the first phase of heap-sort





Analysis via Math



$$T = 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + 2^{h-3} \cdot 3 + \cdots + 2^{0} \cdot h$$

$$2T = 2^{h} \cdot 1 + 2^{h-1} \cdot 2 + 2^{h-2} \cdot 3 + \cdots + 2^{1} \cdot h$$

$$2T - T = 2^{h} + 2^{h-1} + 2^{h-2} + \cdots + 2^{1} - h$$

$$T = 2\left(2^{h} - 1\right) - h = 2^{h+1} - 2 - h = n - \log_2(n+1)$$

$$T \Rightarrow O(n)$$

$n=2^{h+1}-1$
$h=\log_2(n+1)-1$

Adaptable Priority Queues

- New functions for
  - Delete an arbitrary node
    - Replace with the last one in heap
    - Bubble down
  - Update the key of an arbitrary node
    - Bubble down or up depending on
      - Difference in keys
      - Min or max heaps

Practical App of Priority Queues

- Transactions in stock market
- Event-driven simulation
  - Molecular dynamics simulation

Exercises

- How to build a heap in O(n)?
  - 1 3 5 7 9 11 13 15 2 4 6 8 10 12 14
- How many different possible insertion sequences can be use to generate the following min heap?