Heaps

Readings - Chapter 9

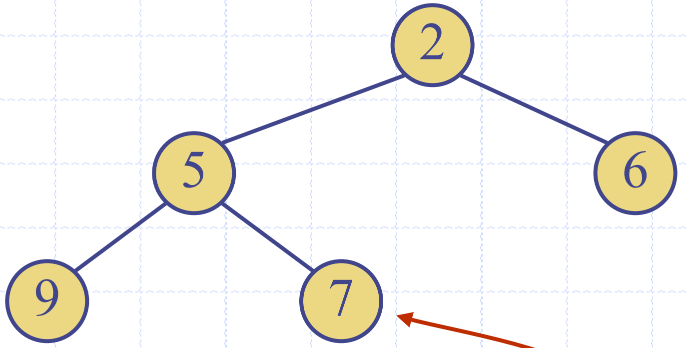
Heaps

- 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,
  $key(v) \geq key(parent(v))$
- 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

heap property

Structural property

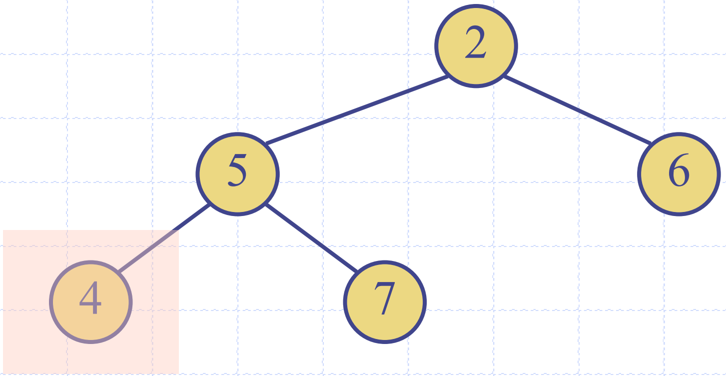
- The last node of a heap is the rightmost node of maximum depth



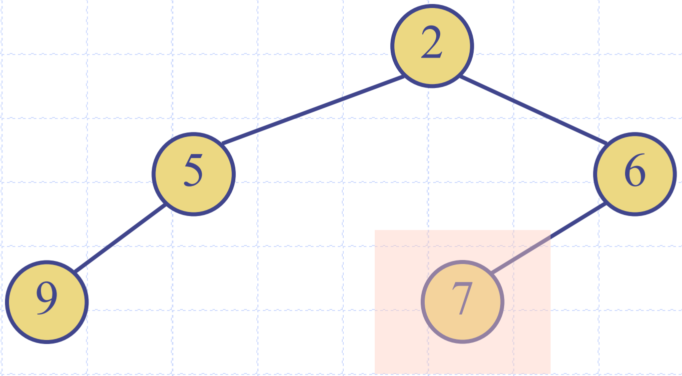
last node

Example of non-Heaps

- Heap property violation



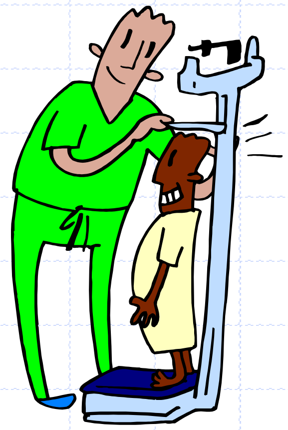
- Structural property violation



Finding the minimum element

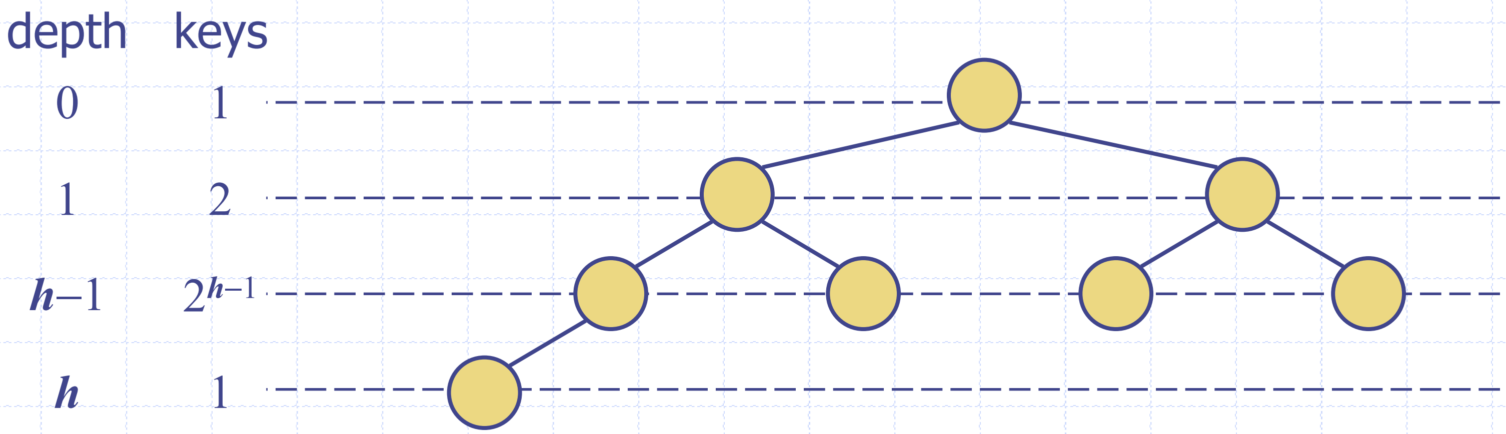
- The element with the smallest key value always sits at the root of the heap.
  - If it is elsewhere, it would have a parent with a larger key, thus violating the heap property.
  - hence finding the minimum can be done in O(1) time

Height of a Heap



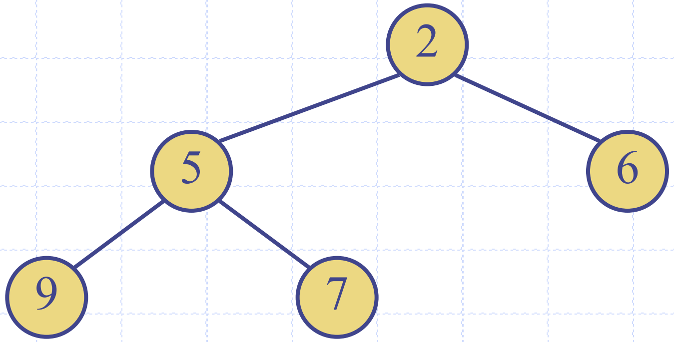
- **Theorem:** A heap storing n keys has height $O(\log n)$

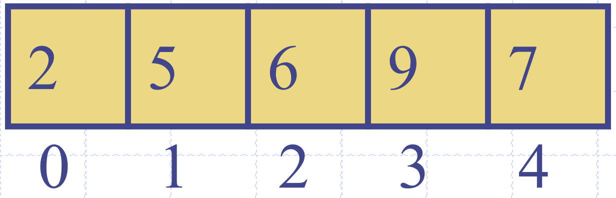
  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$



Array-based Heap Implementation

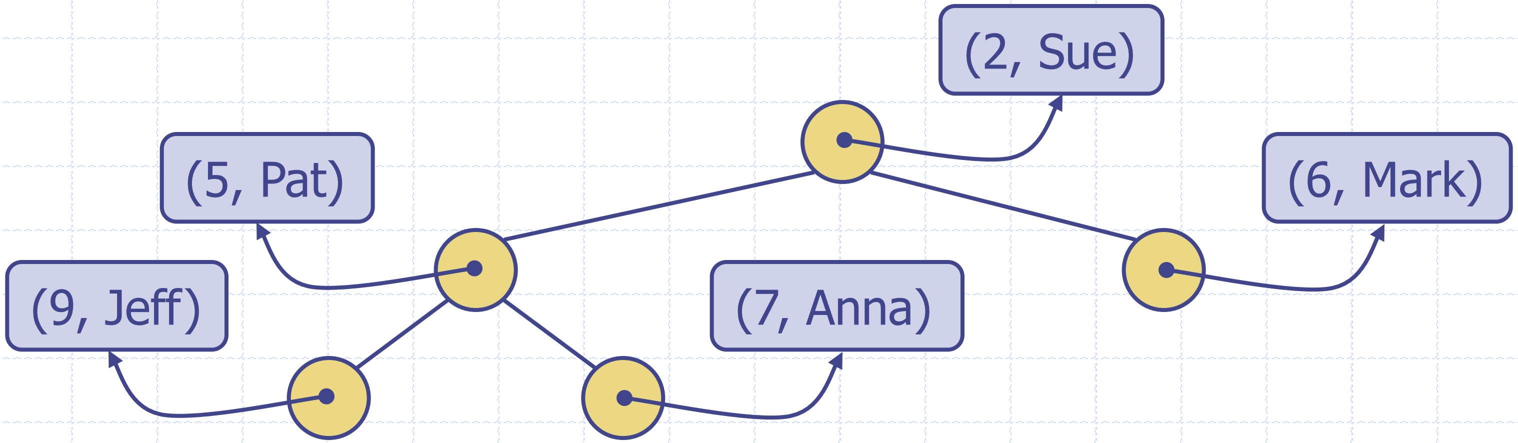
- We can represent a heap with n keys by means of an array of length n
- For the node at index i
  - the left child is at index $2i + 1$
  - the right child is at index $2i + 2$
  - parent(i) is *floor*((i-1)/2)
- Operation add corresponds to inserting at index $n + 1$
- Operation remove_min corresponds to removing at index n





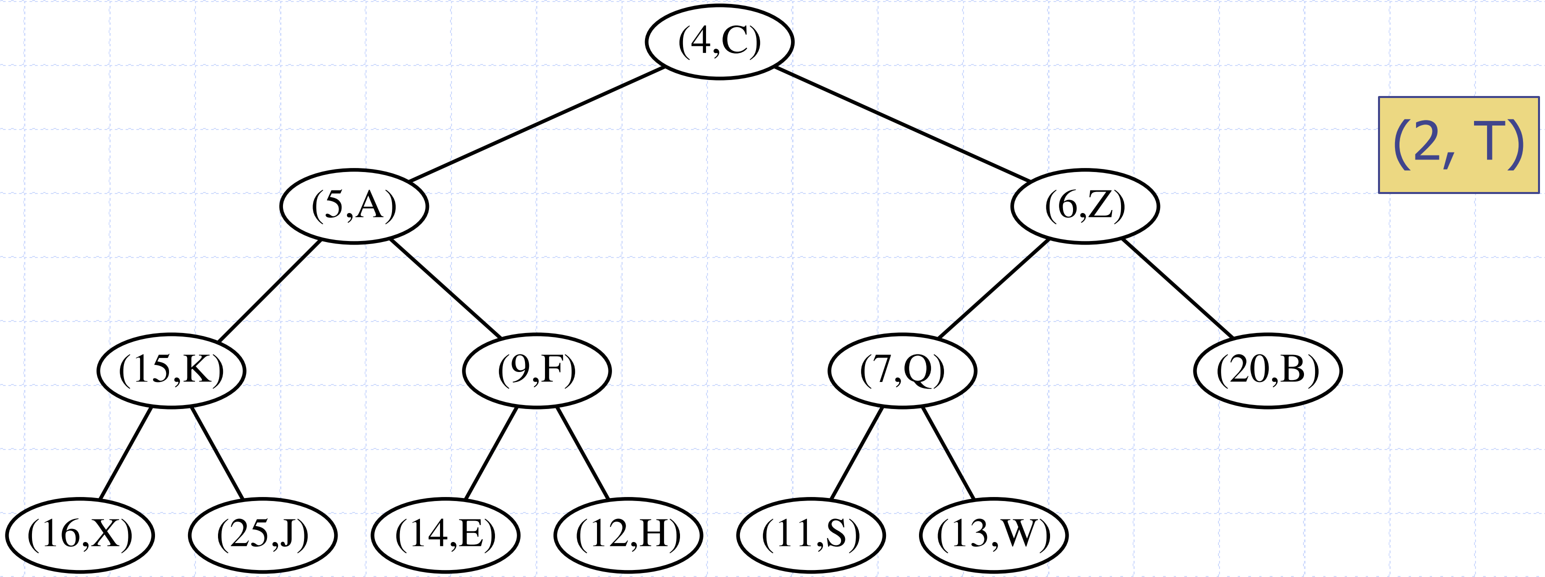
Heaps and Priority Queues

- 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



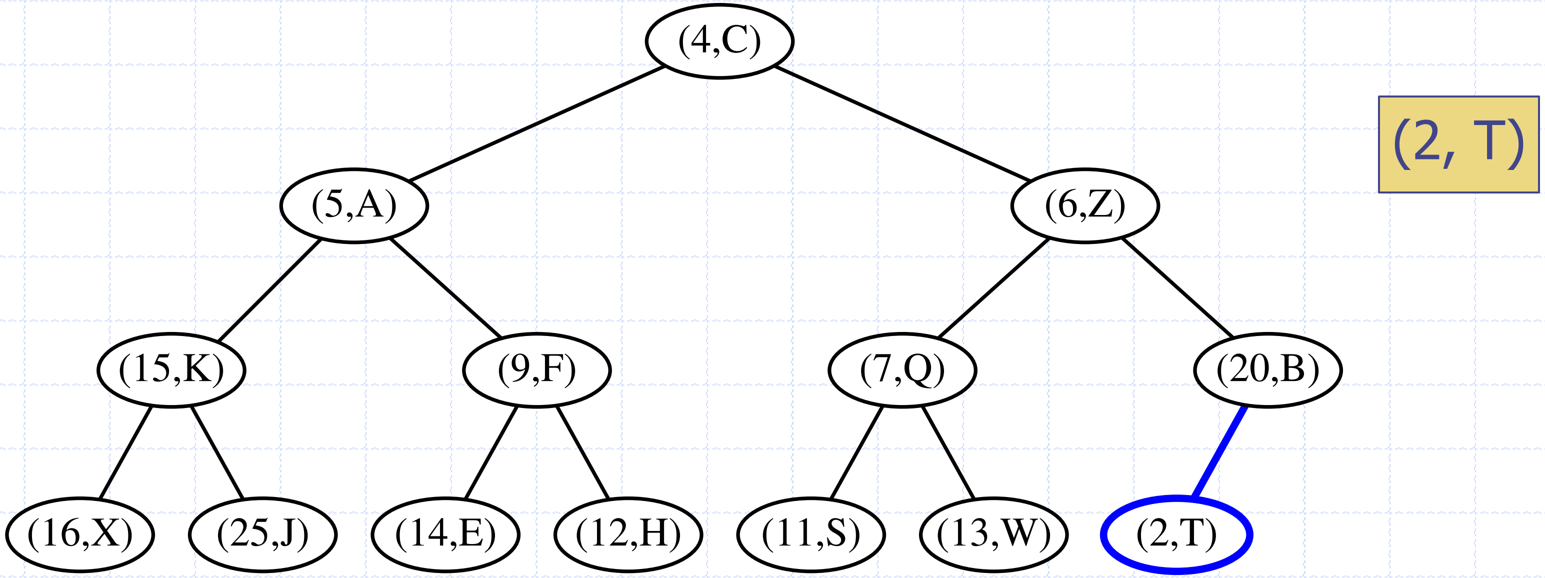
Insertion into a Heap

- Method insertItem 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)



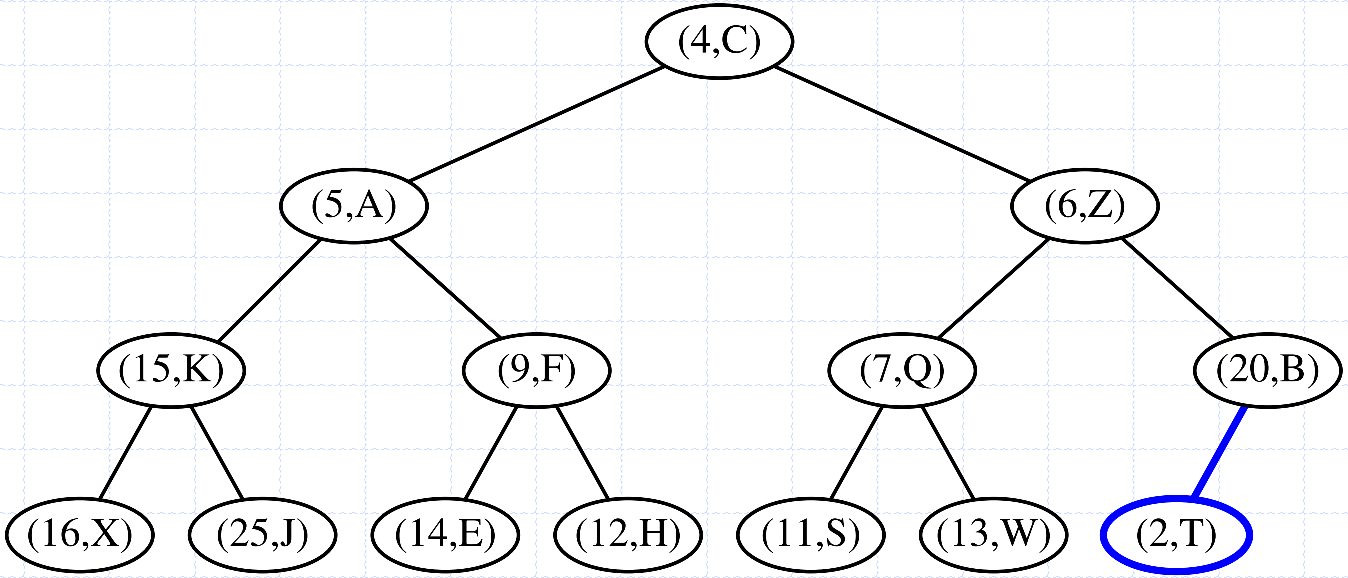
Insertion into a Heap

- Method insertItem 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)



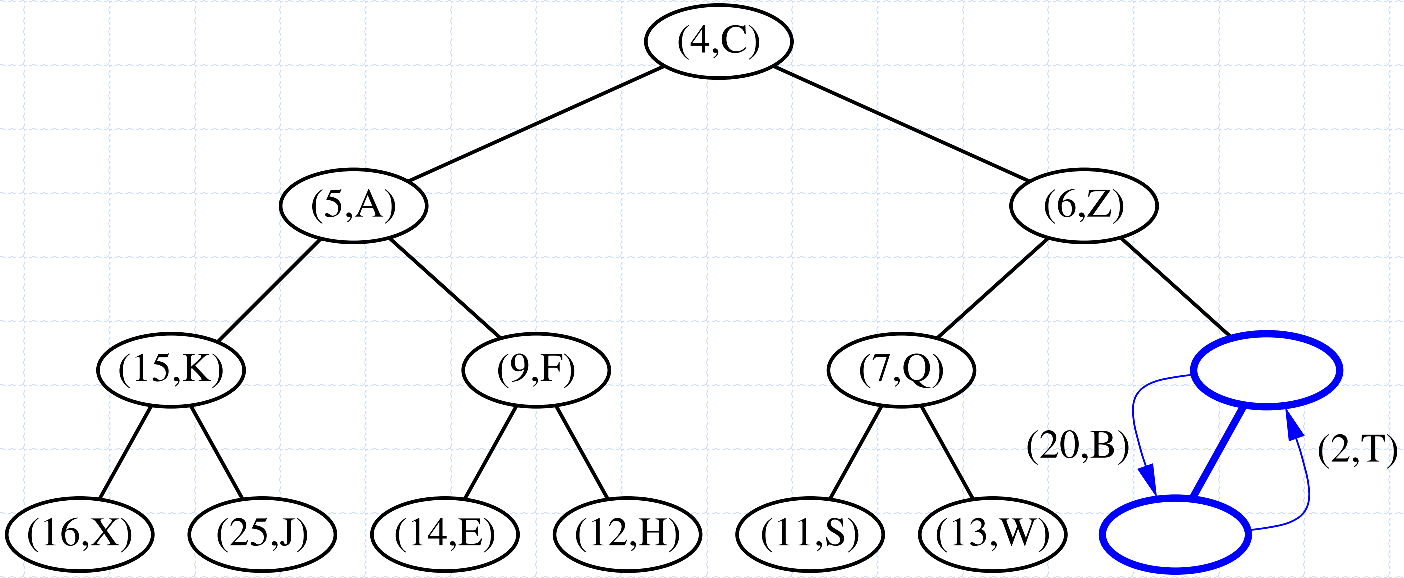
Upheap

- 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 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



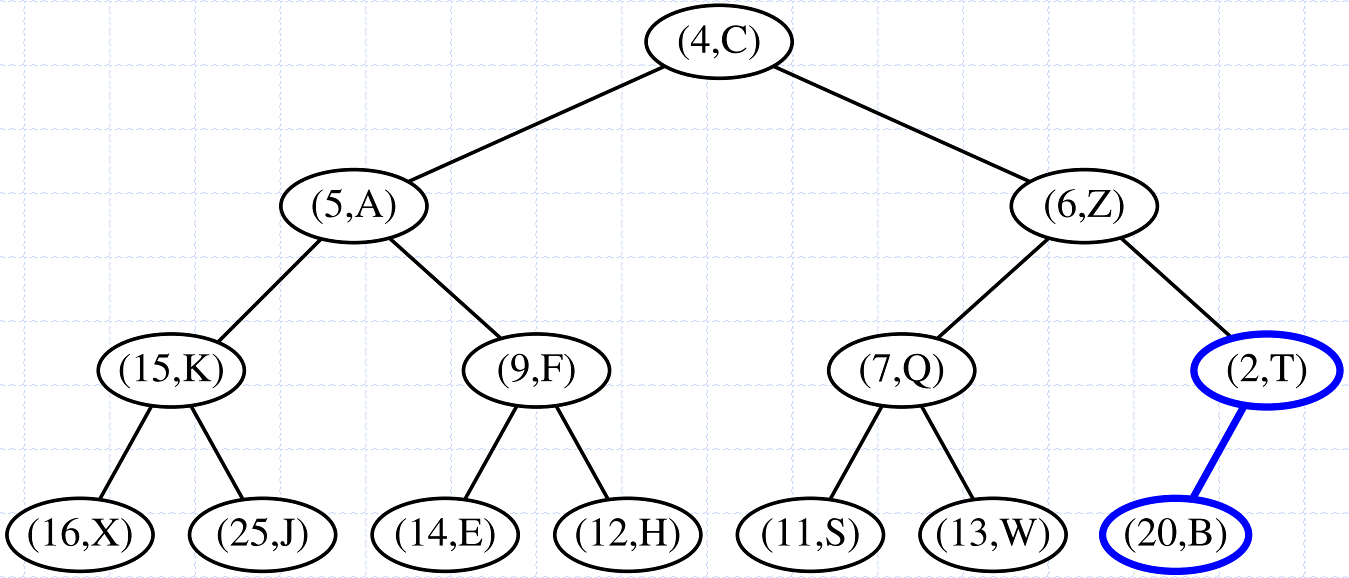
Upheap

- 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 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



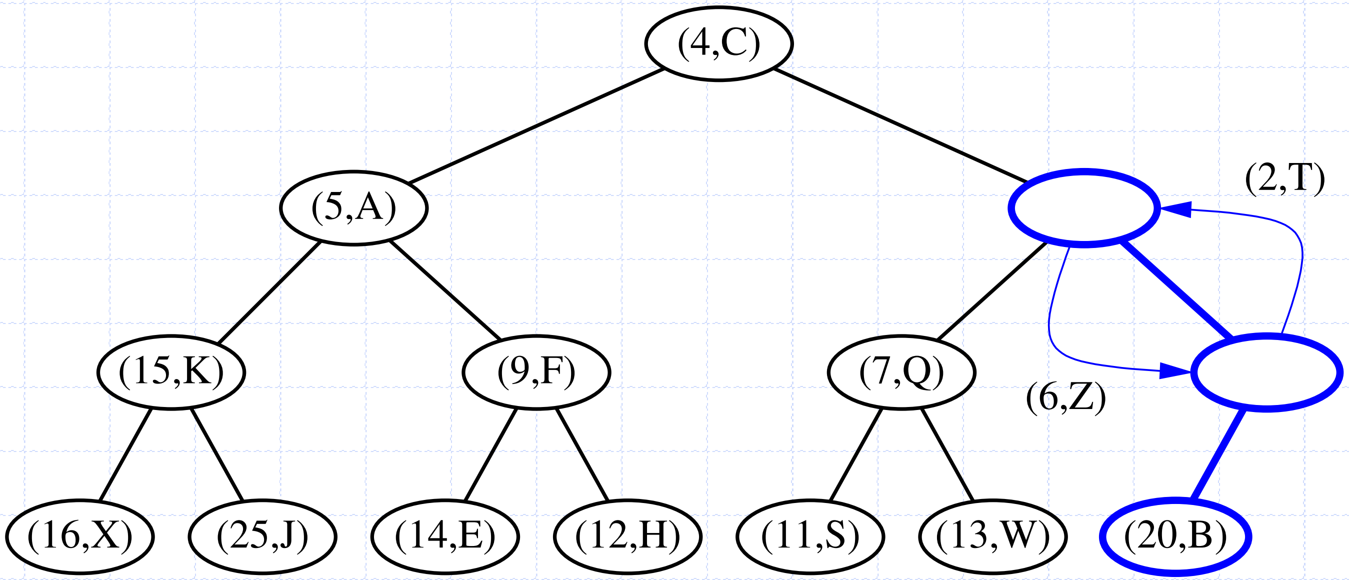
Upheap

- 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 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



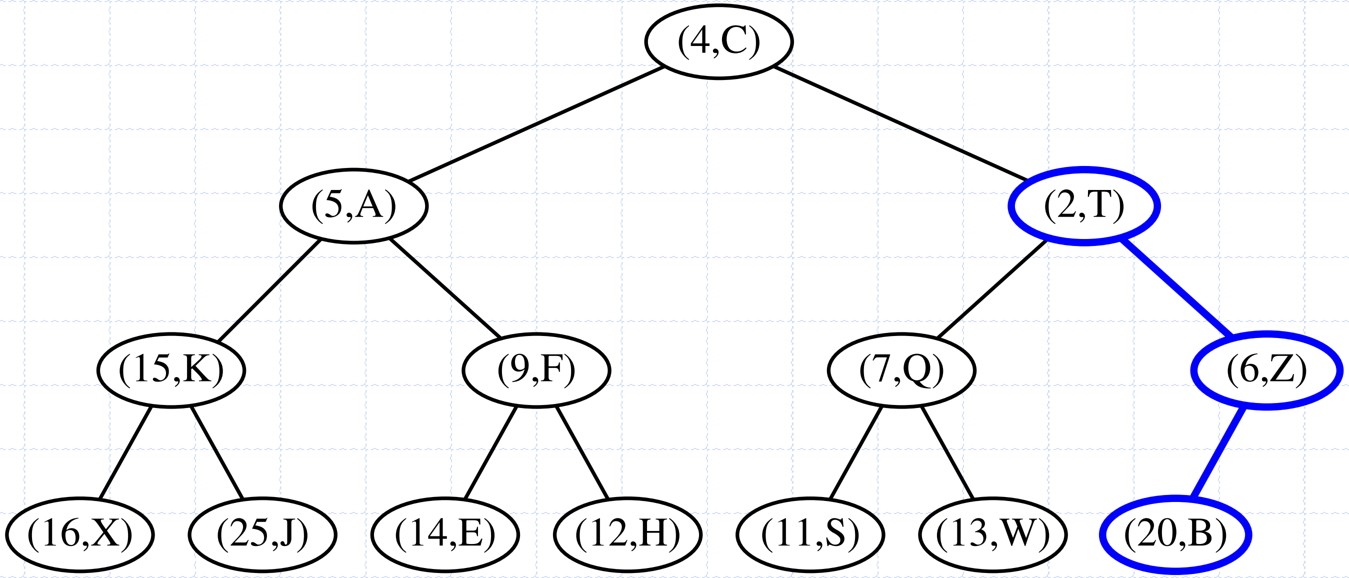
Upheap

- 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 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



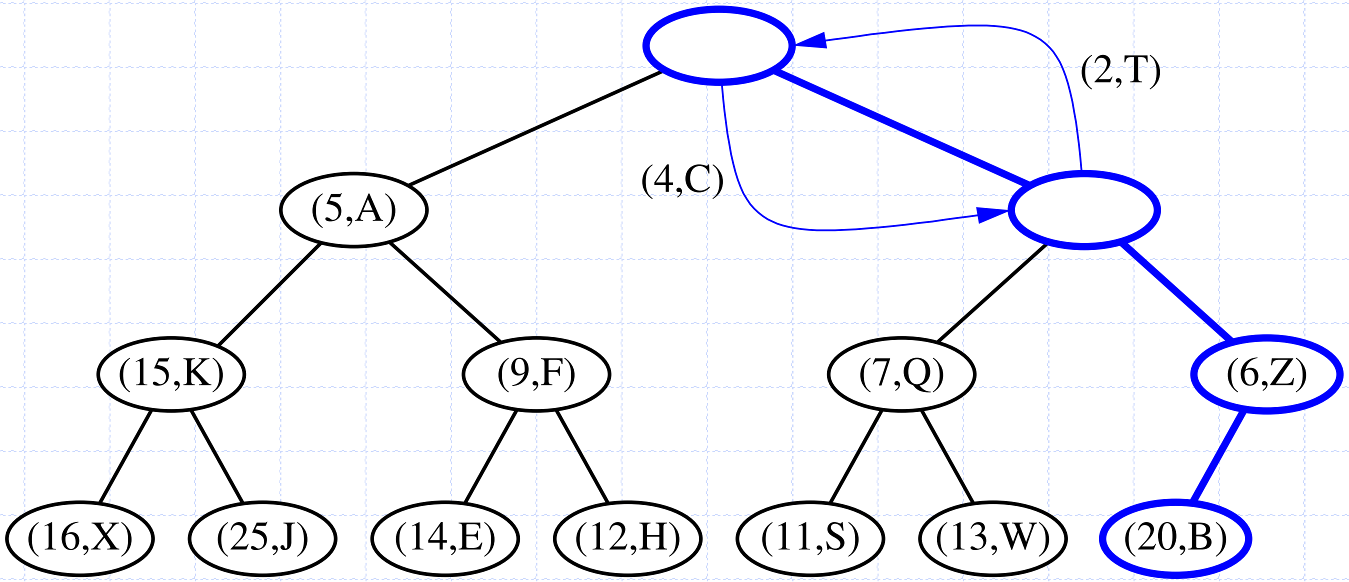
Upheap

- 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 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



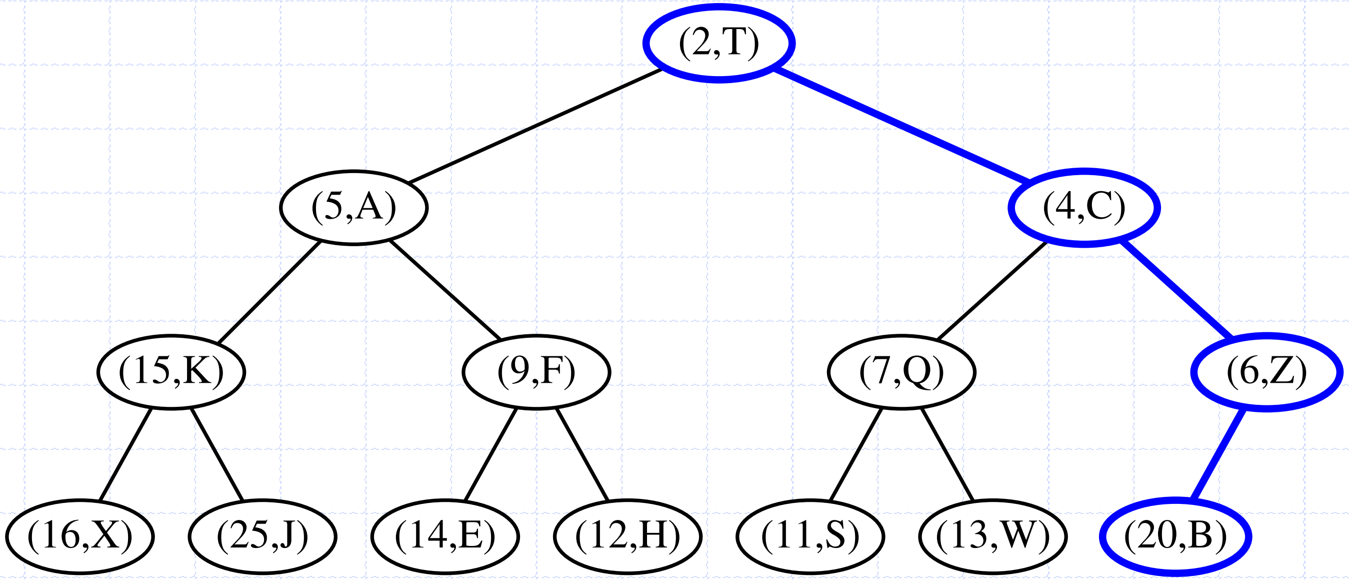
Upheap

- 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 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



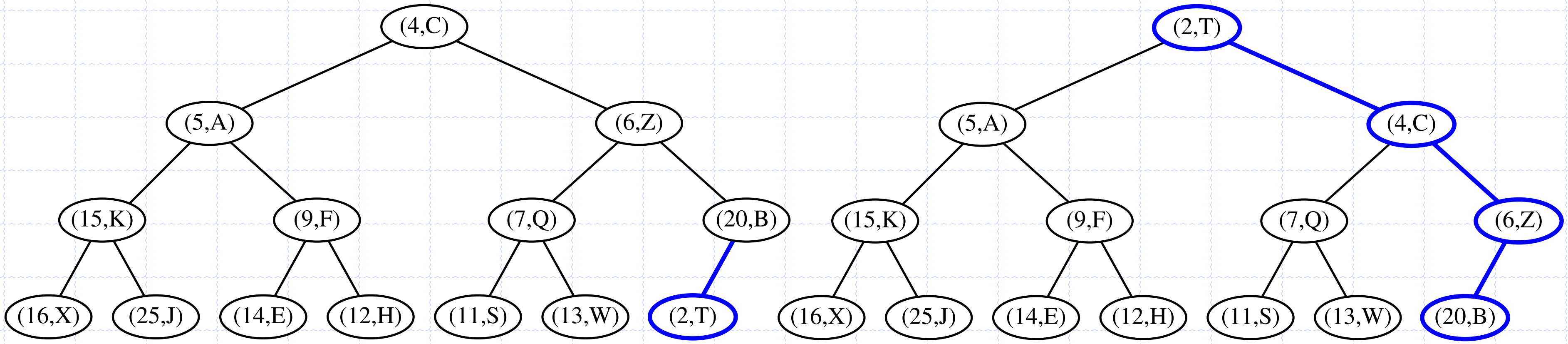
Upheap

- 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 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



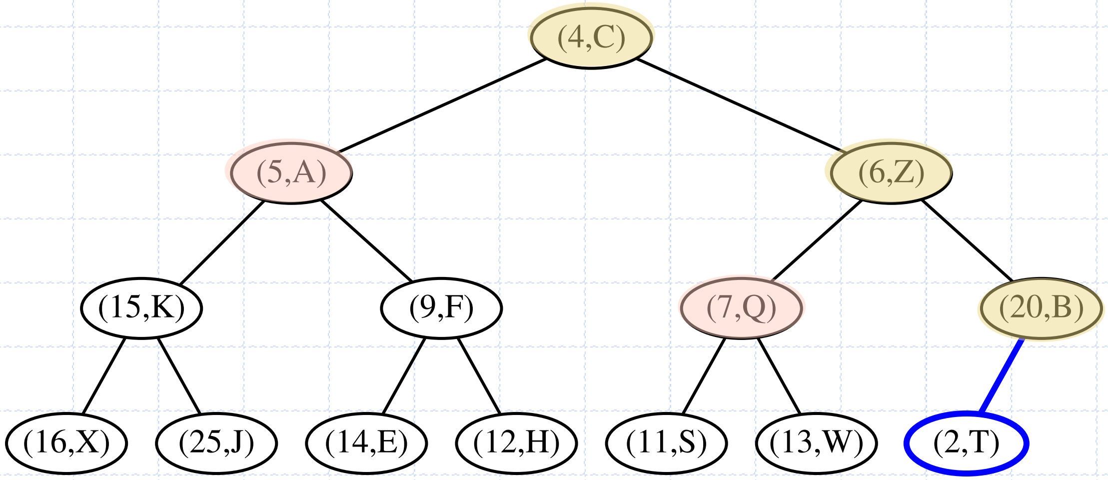
Another View of Insertion

- Enlarge heap
- Consider path from root to inserted node
- Find topmost element on this path with higher priority that of inserted element
- Insert new element at this location by shifting down the other elements on the path



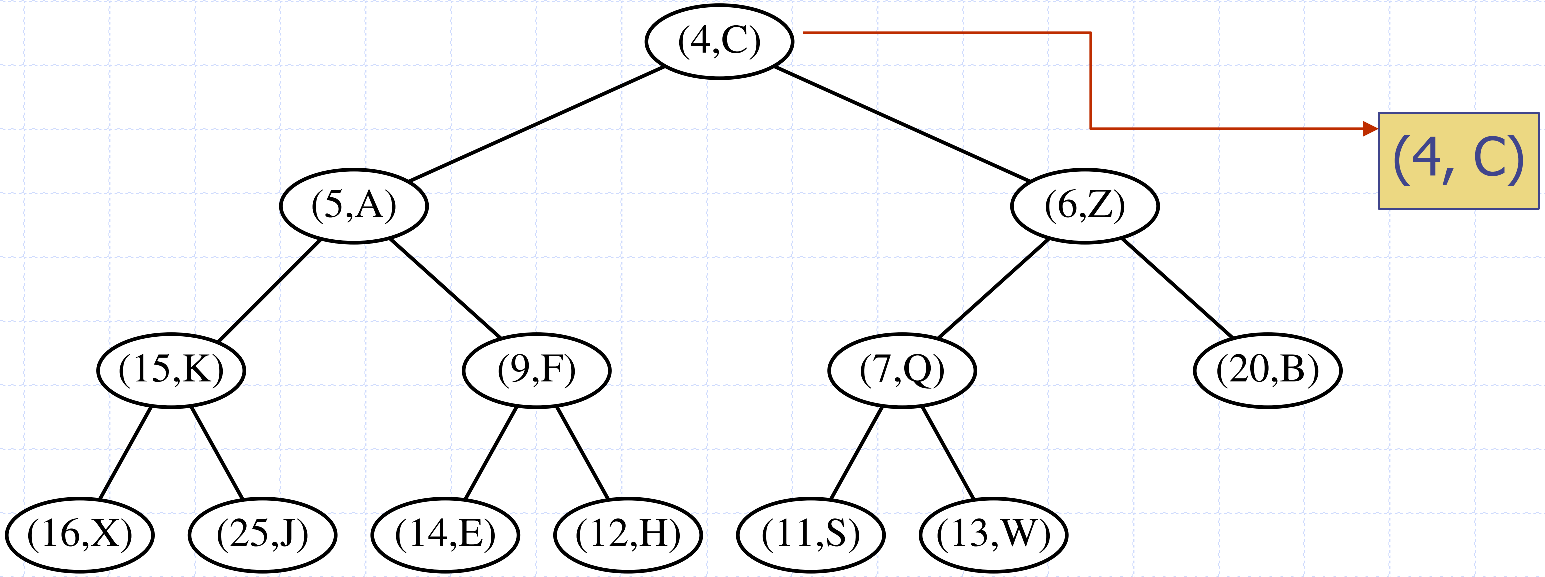
Correctness of Upheap

- The only nodes whose contents change are the ones on the path
- Heap property may be violated only for children on these nodes
- But new contents of these nodes only have lower priority
- So heap property is not violated



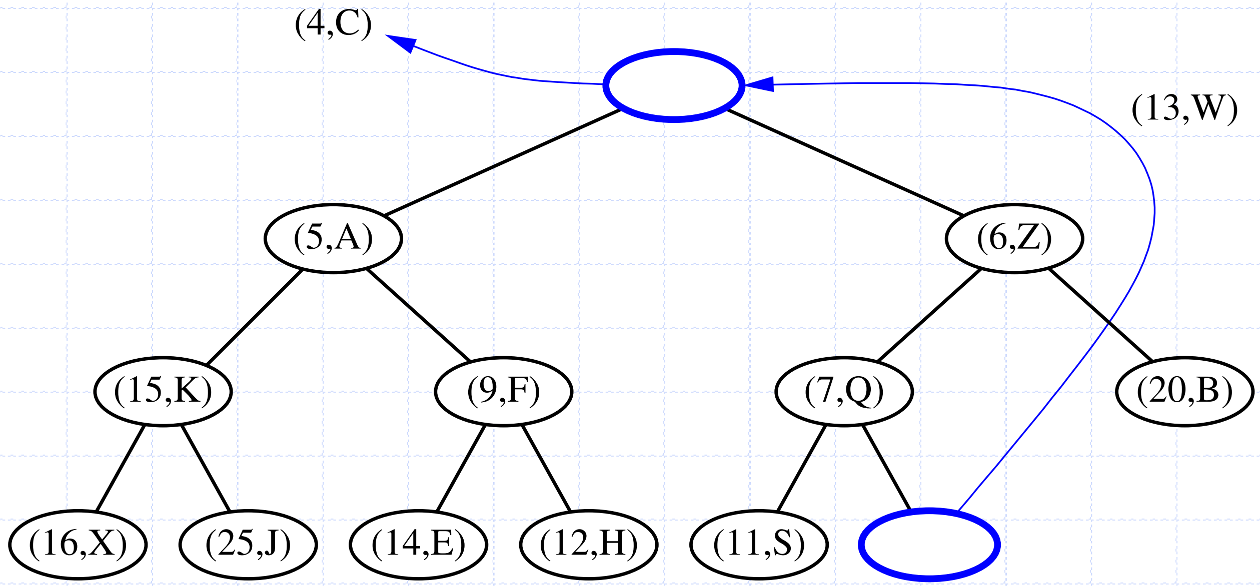
Removal from a Heap

- Method removeMin of the priority queue ADT corresponds removing the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)



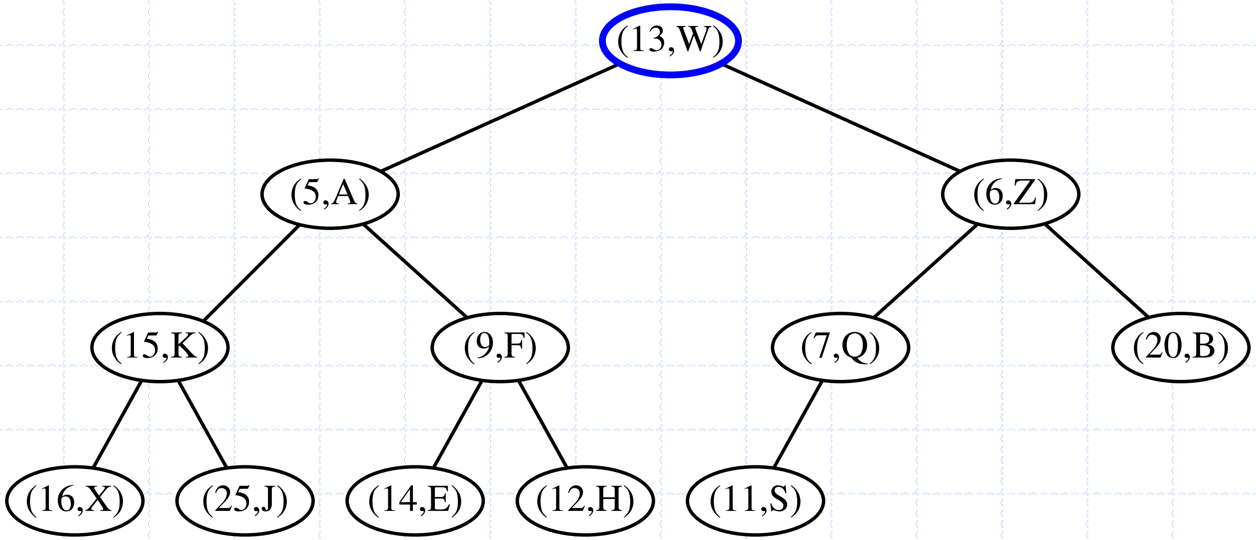
Removal from a Heap

- Method removeMin of the priority queue ADT corresponds removing the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)



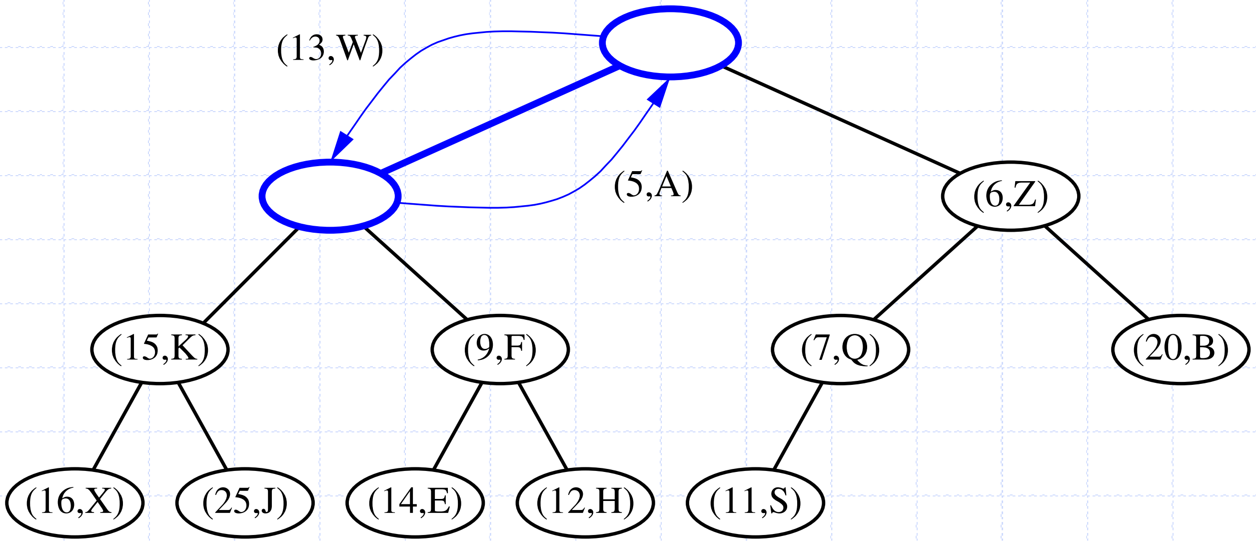
Downheap

- 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



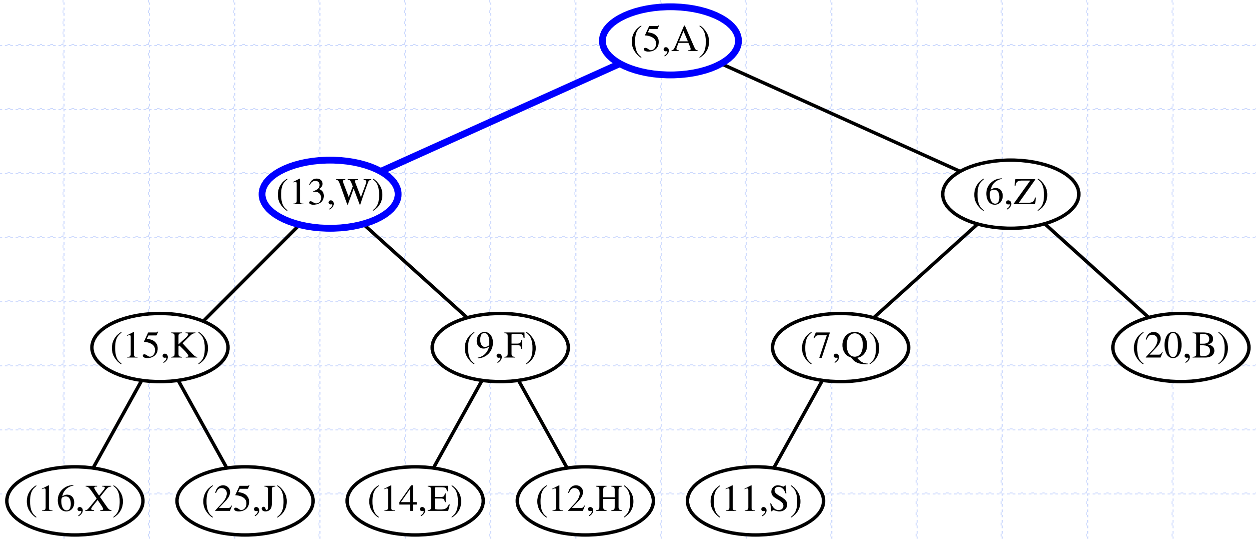
Downheap

- 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



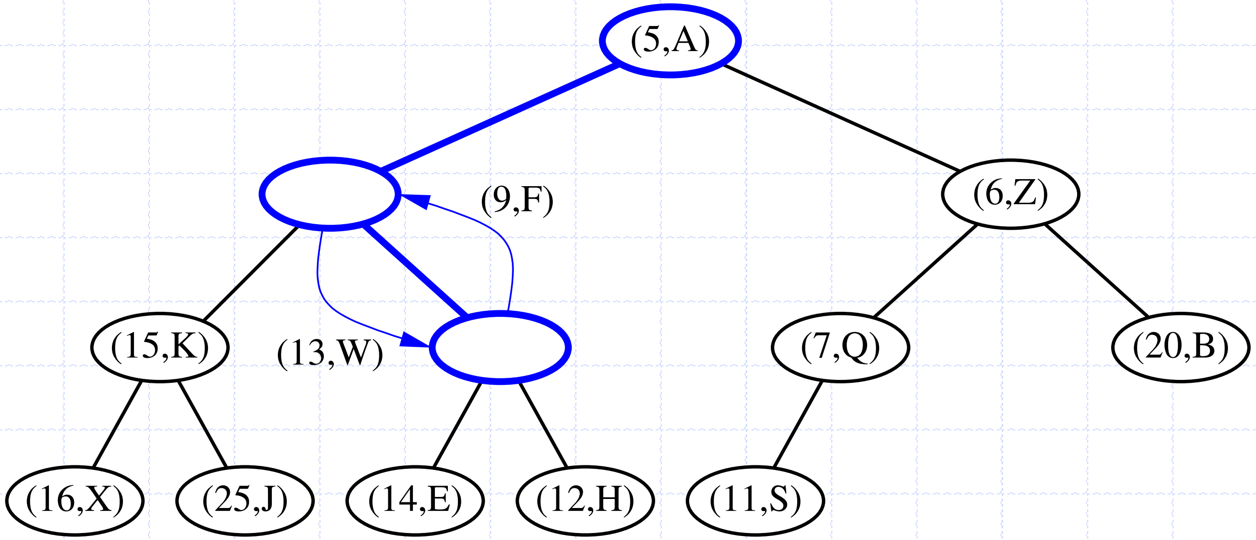
Downheap

- 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



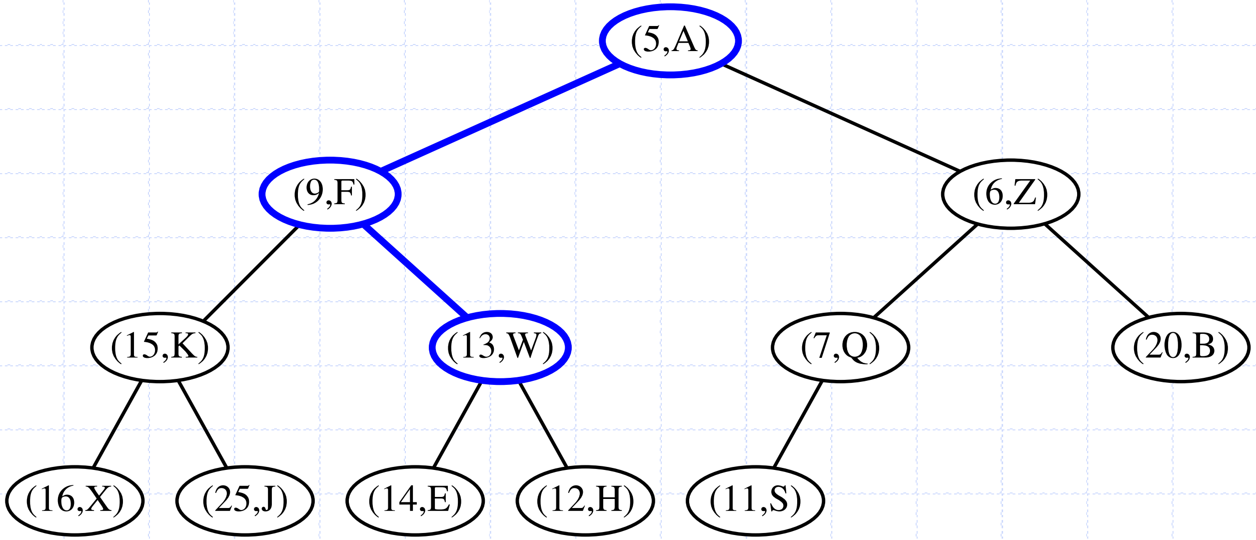
Downheap

- 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



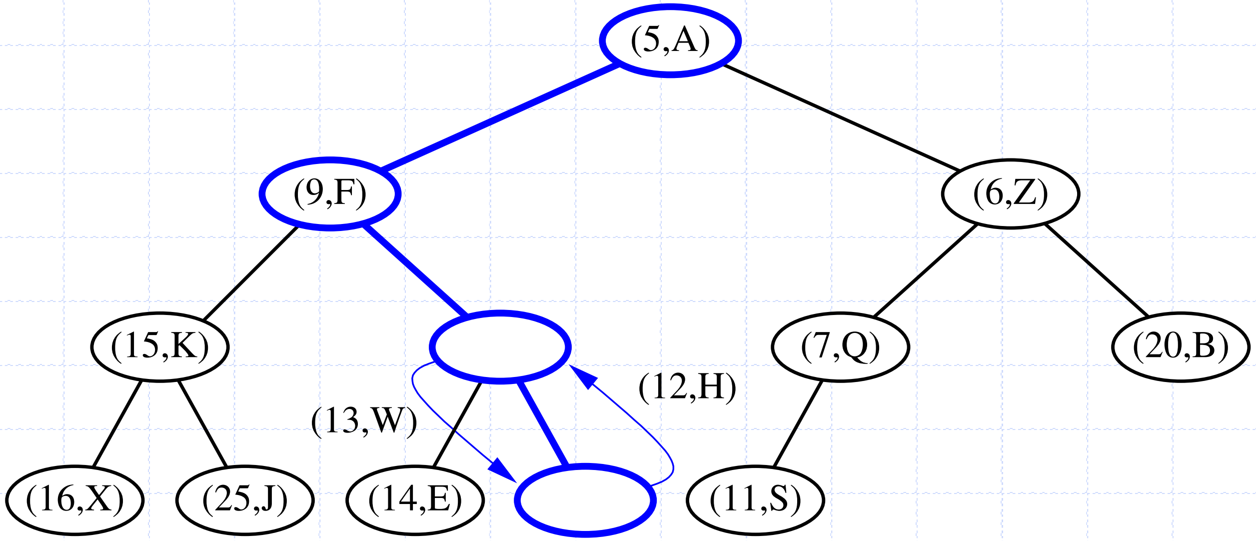
Downheap

- 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



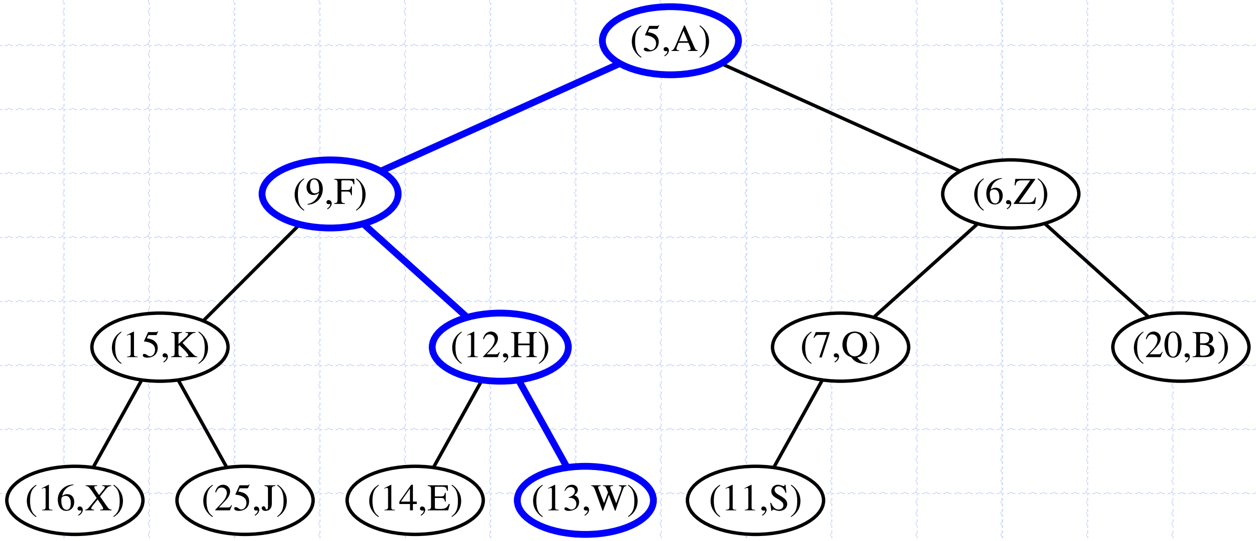
Downheap

- 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



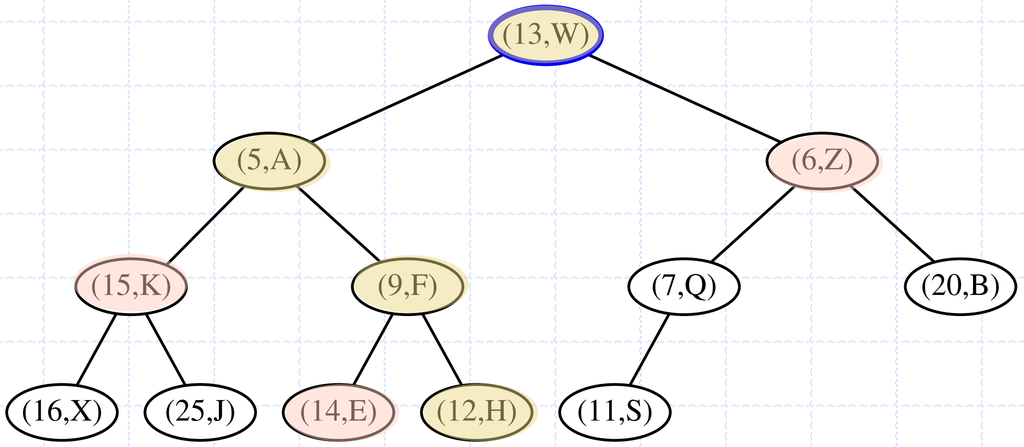
Downheap

- 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



Correctness of Downheap

- Downheap traces a path down the tree
- For a node on the path (say j) - both *key(left(j))* and *key(right(j))* > *key(j)*
- All elements on path have lower key values than their siblings
- All elements on this path are moved up.
- Hence the heap property is not violated.



Run Time Analysis

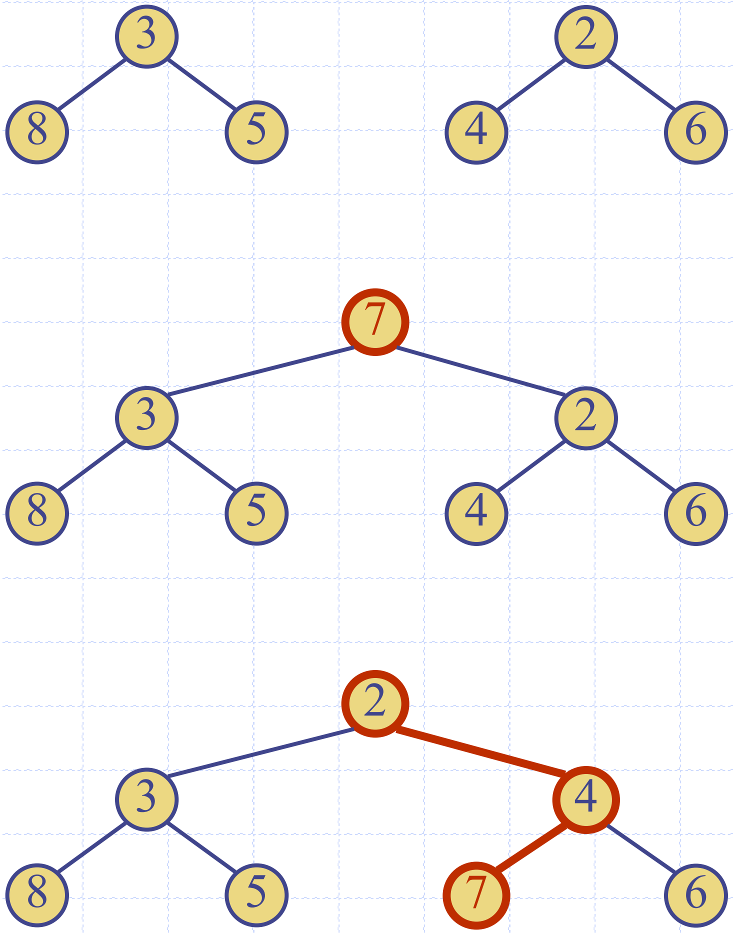
- heap of n nodes has height O(log n)
- insertion - Upheap – move the element all the way to the top
  - O(log n) steps in worst case
- removal – Downheap – move the root element all the way to a leaf
  - O(log n) steps in worst case

Building a heap

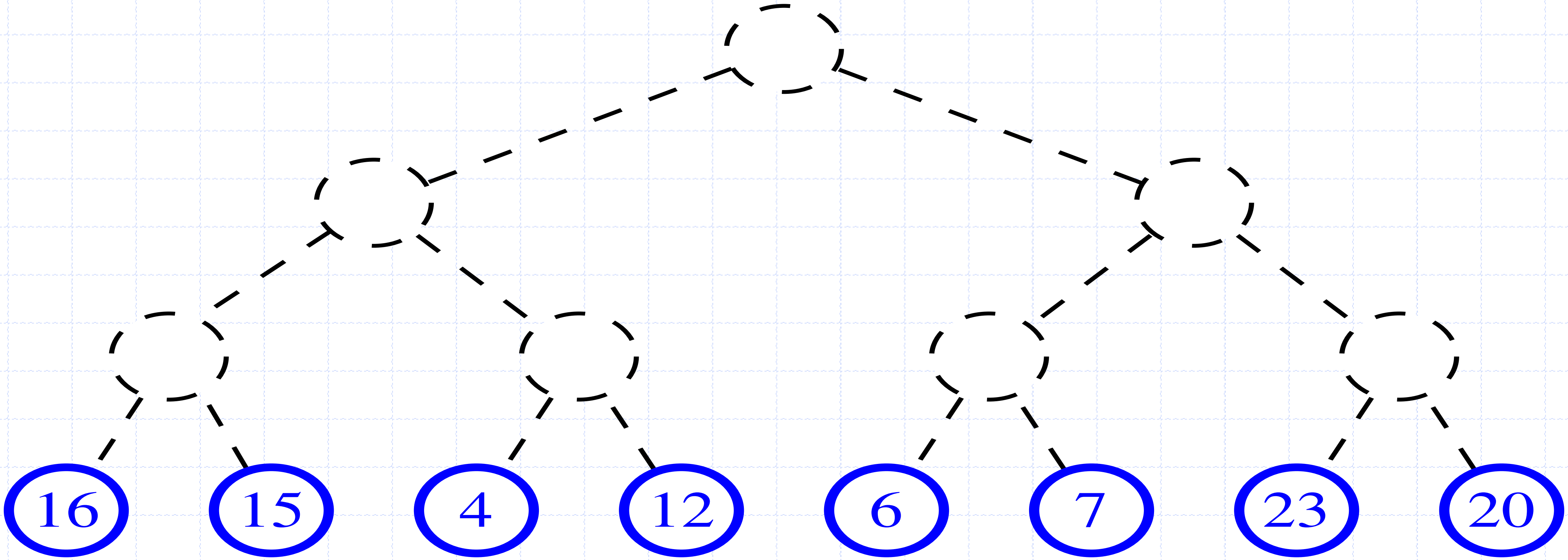
- Call Upheap procedure of the heap n times.
  - Upheap is O(log n)
  - inserting first element – O(log 1)
  - inserting second element – O(log 2)
  - ..
  - inserting the nth element – O(log n)
  - so for the n insertions – log 1+ log 2+…+log n = log n! = O(nlog n)

Merging Two Heaps

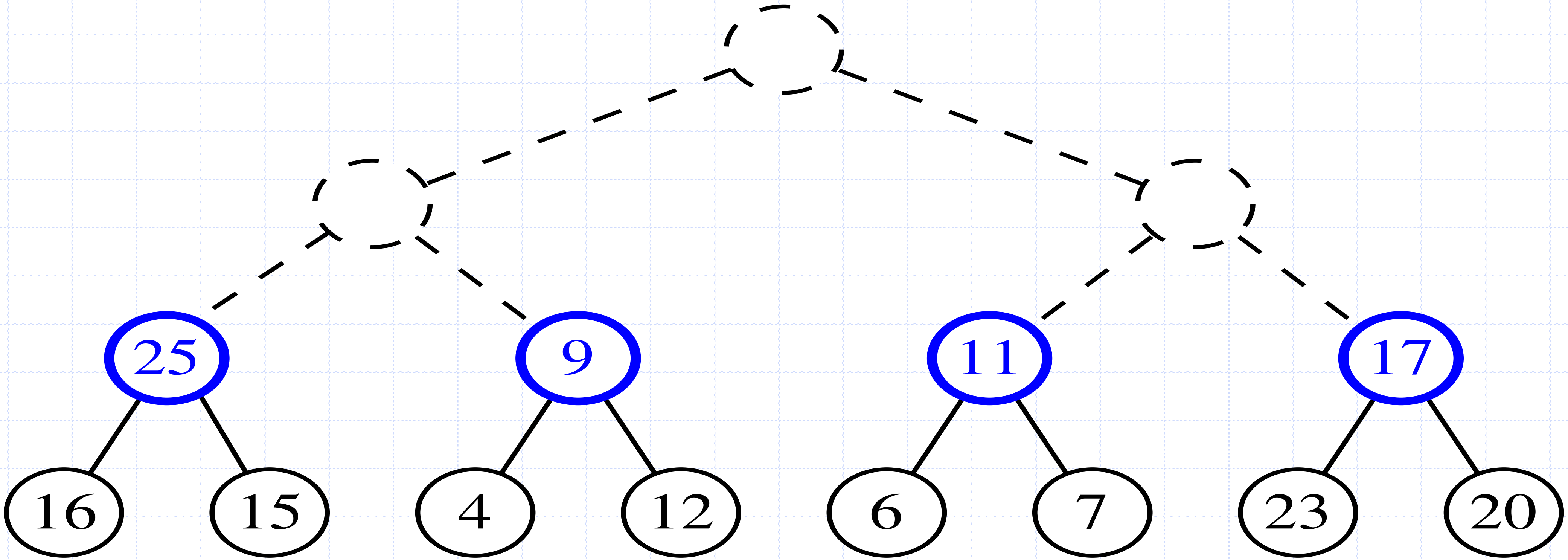
- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform Downheap to restore the heap-order property



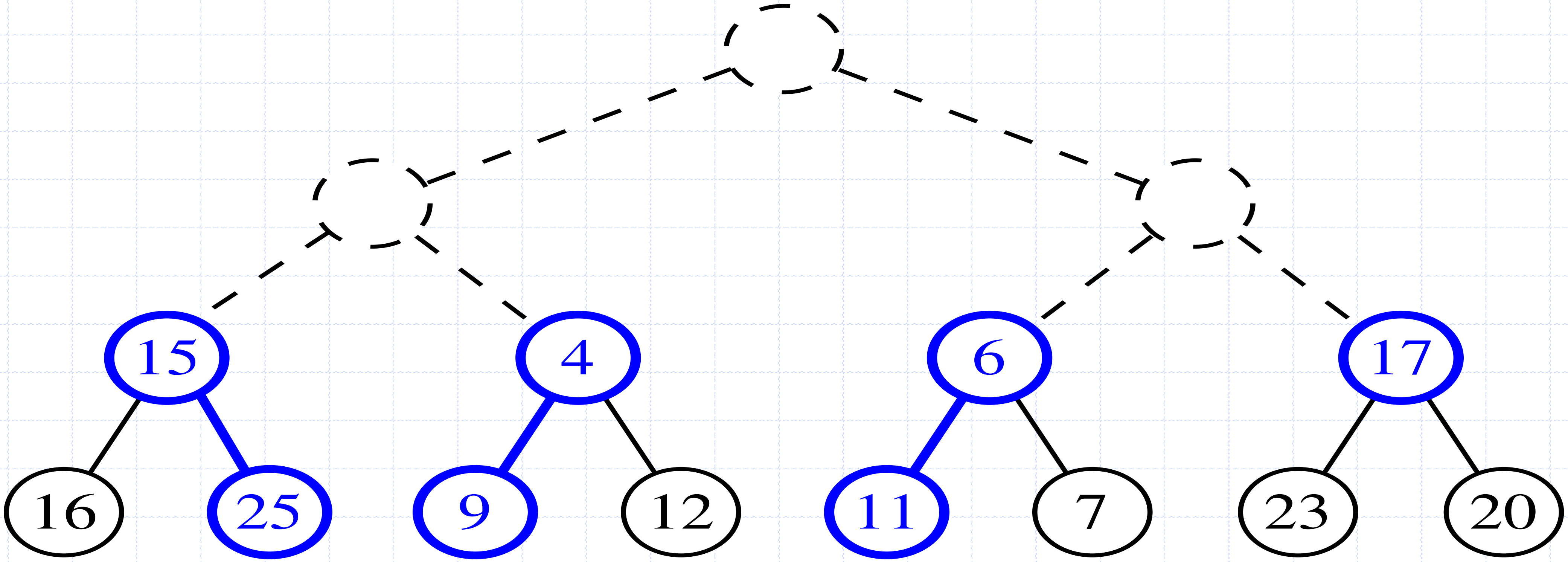
Building a Heap – Bottomup



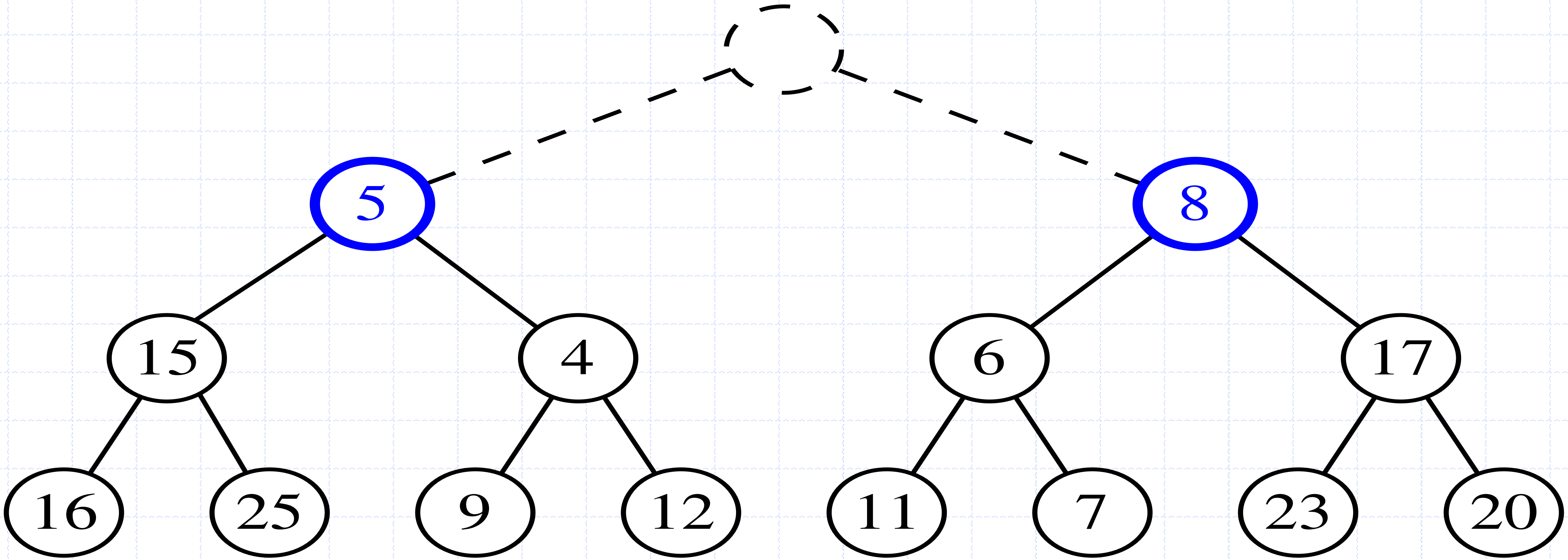
Building a Heap – Bottomup



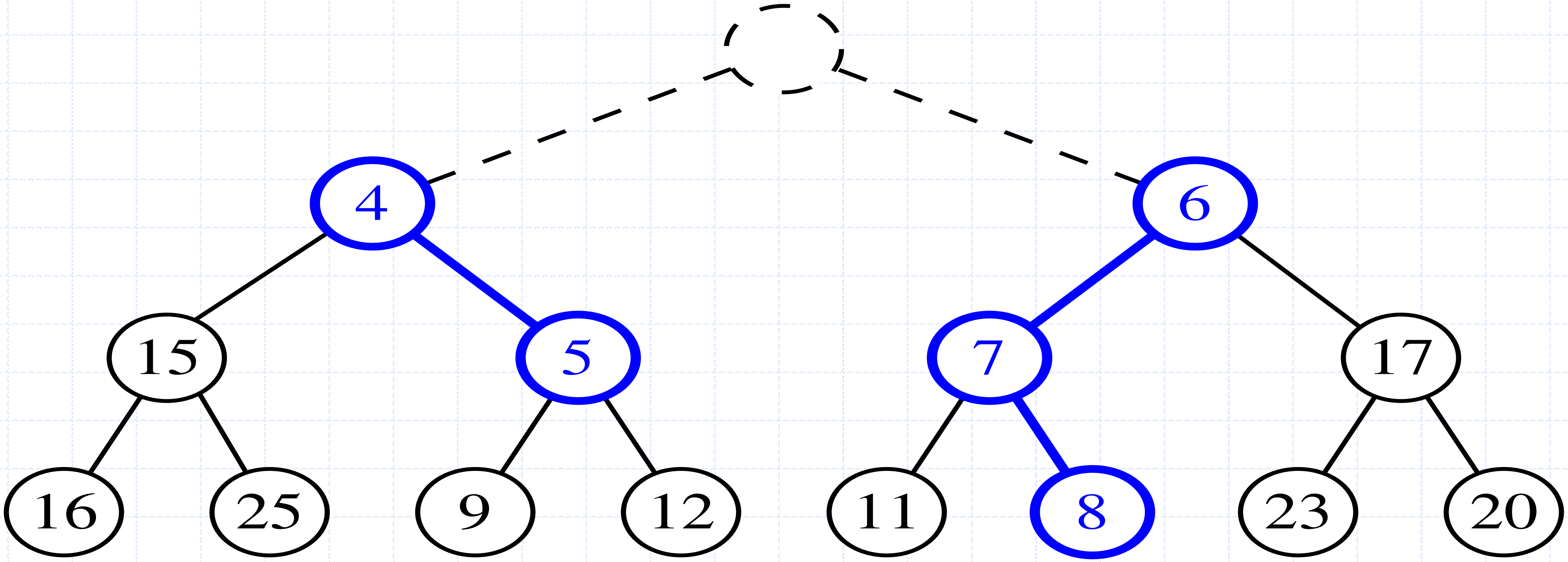
Building a Heap – Bottomup



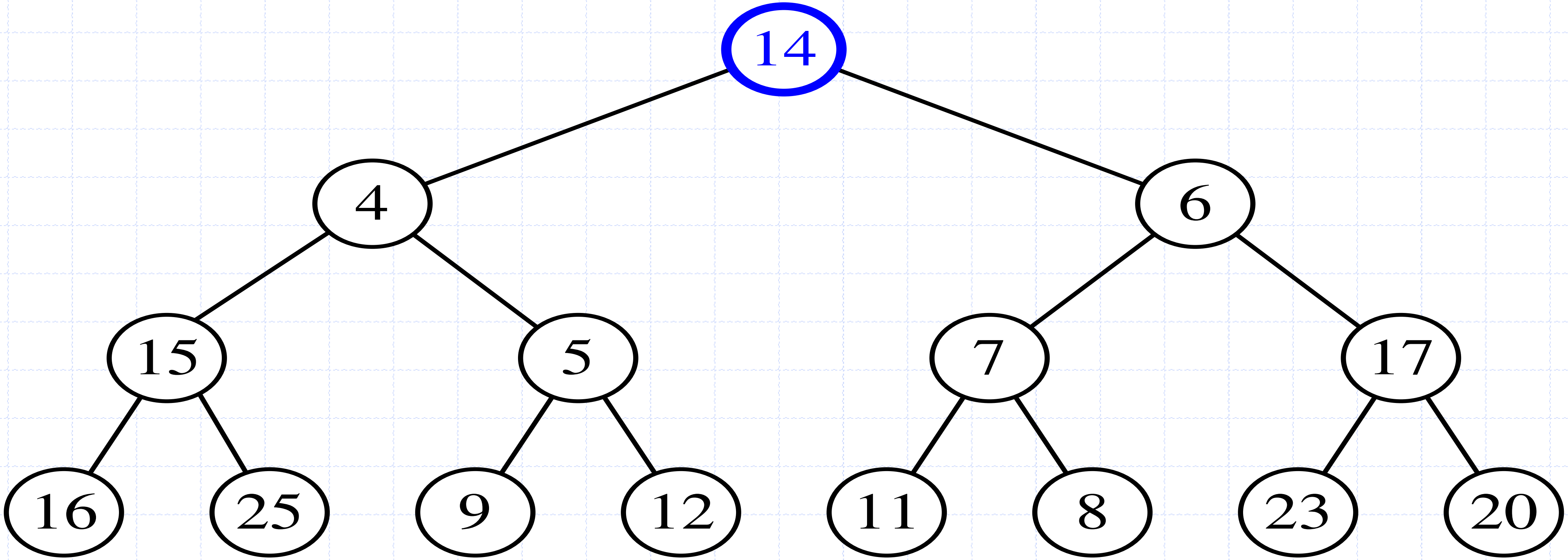
Building a Heap – Bottomup



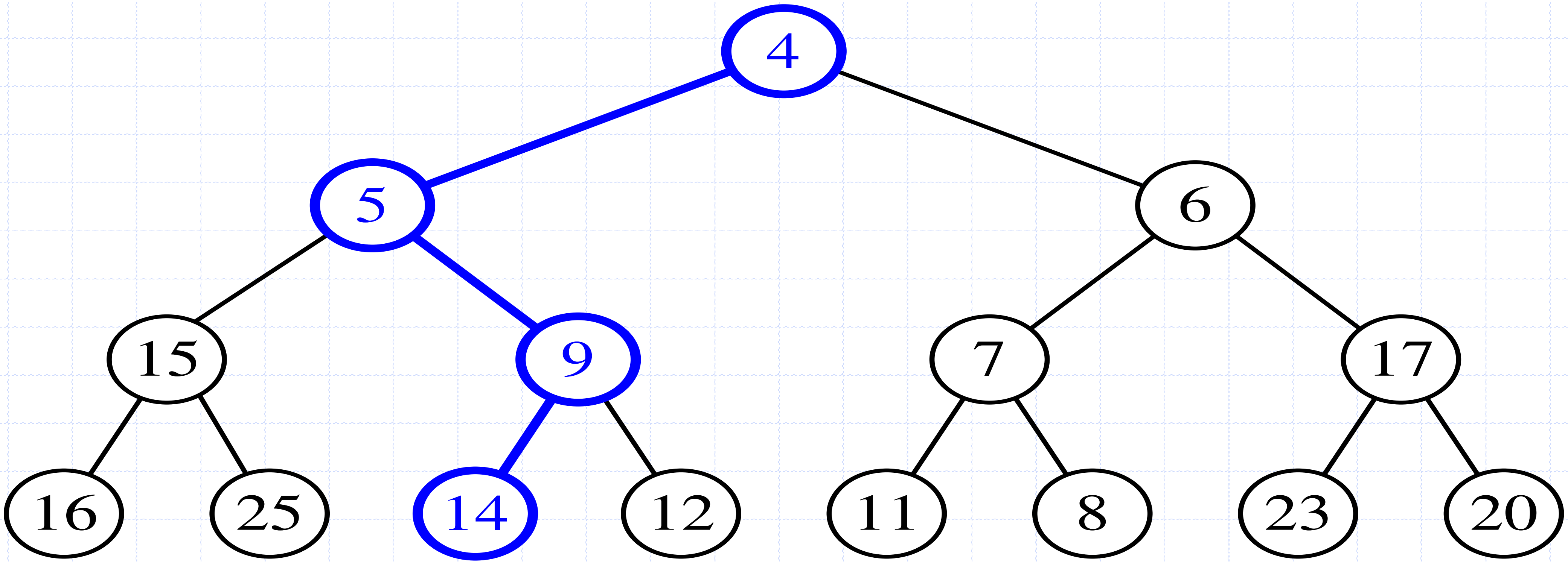
Building a Heap – Bottomup



Building a Heap – Bottomup

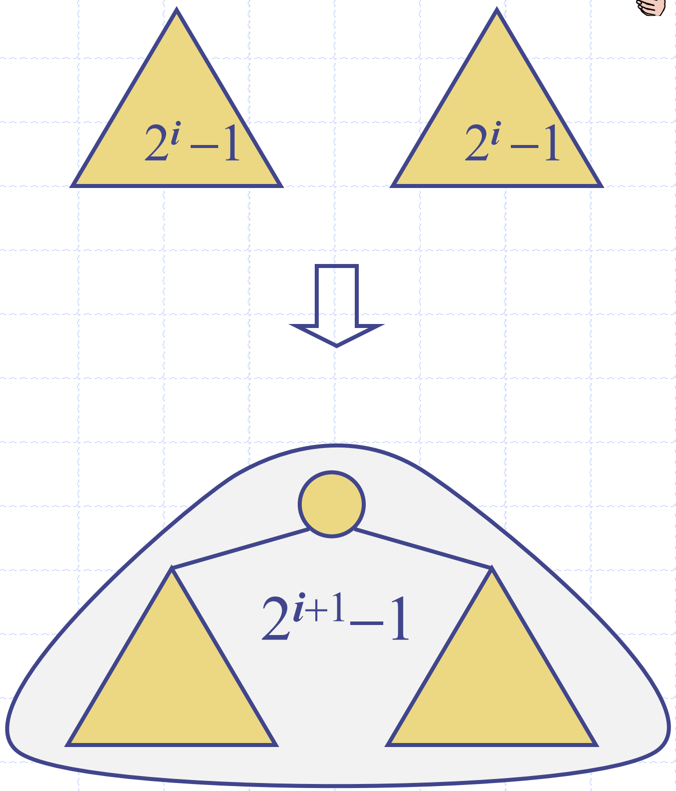


Building a Heap – Bottomup



Bottom-up Heap Construction

- 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

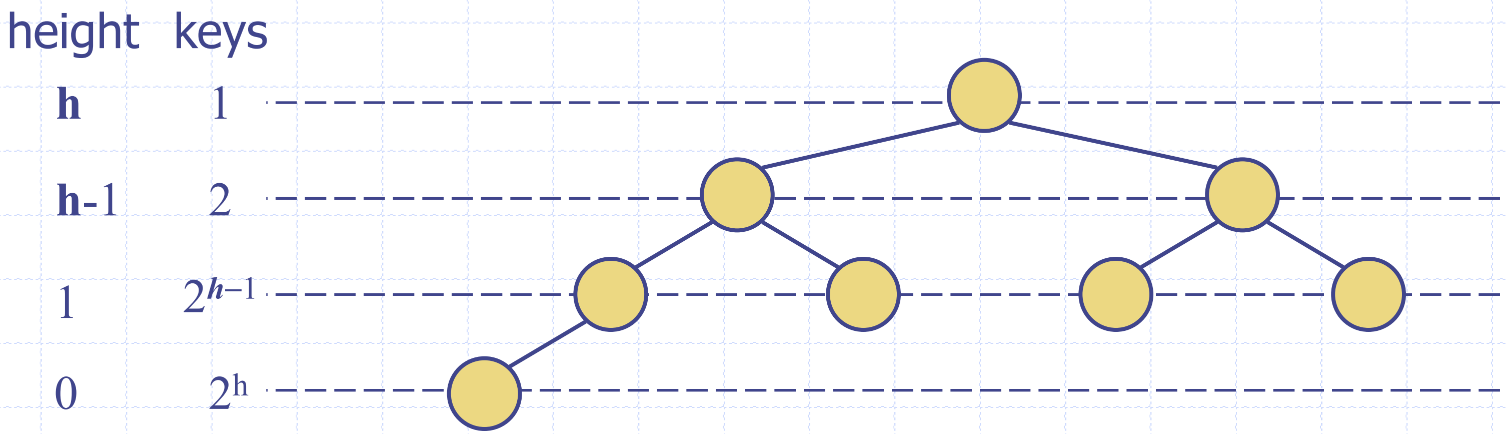


Building a Heap – Bottomup Analysis

- Correctness: induction on *i*, all trees rooted at $m > i$ are heaps
- Running time: n calls to Downheap
  - Downheap – $O(\log n)$
  - so total running time $O(n \log n)$
- We can provide a better bound – $O(n)$
  - Idea – for most of the time, Downheap works on smaller than n element heaps

Building a Heap – Bottomup Analysis (2)

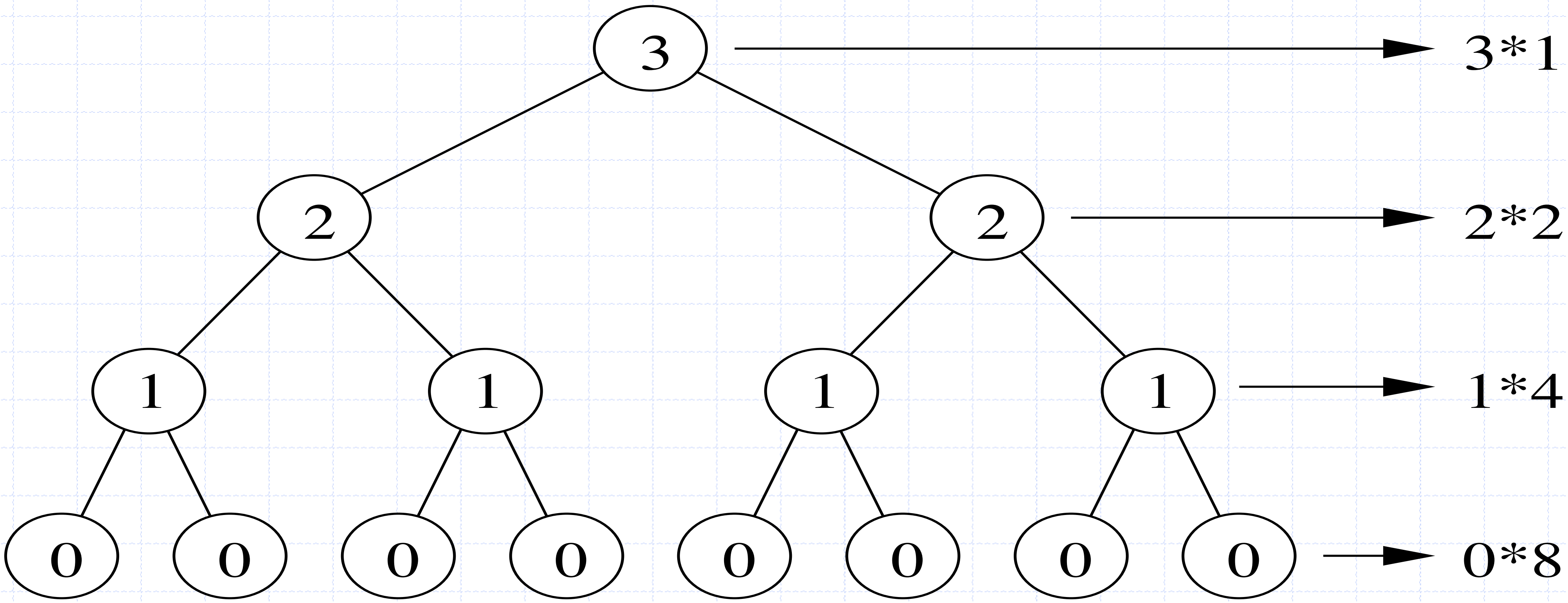
- height of node: length of longest path from node to leaf
- height of tree: height of root



Building a Heap – Bottomup Analysis (3)

- height of node: length of longest path from node to leaf
- height of tree: height of root
- time for Downheap(i) = O(height of subtree rooted at i)
- Let us assume a complete binary tree
  - $n = 2^{h+1} - 1$

Building a Heap – Bottomup Analysis (4)



Building a Heap – Bottomup Analysis (5)

- 0^{th} level - 2^h nodes, no need to call Downheap
- 1^{st} level - 2^{h-1} nodes, Downheap - 1 swap
- 2^{nd} level - 2^{h-2} nodes, Downheap - 2 swaps
- j^{th} level - 2^{h-j} nodes, Downheap - j swaps
- h^{th} level – 1 node, Downheap – h swaps
- So total number of swaps is

$$\sum_{j=0}^{h} j 2^{h-j} = \sum_{j=0}^{h} j \frac{2^h}{2^j} = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

  - It can be shown that, this is bound by $2^{(h+1)}$
  - $2^{(h+1)} = n-1$
- Therefore O(n)

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$n + n-1 + \ldots + 2 + 1$$

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

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)	()

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with n insert operations takes time proportional to

$$1 + 2 + \ldots + \boldsymbol{n}$$

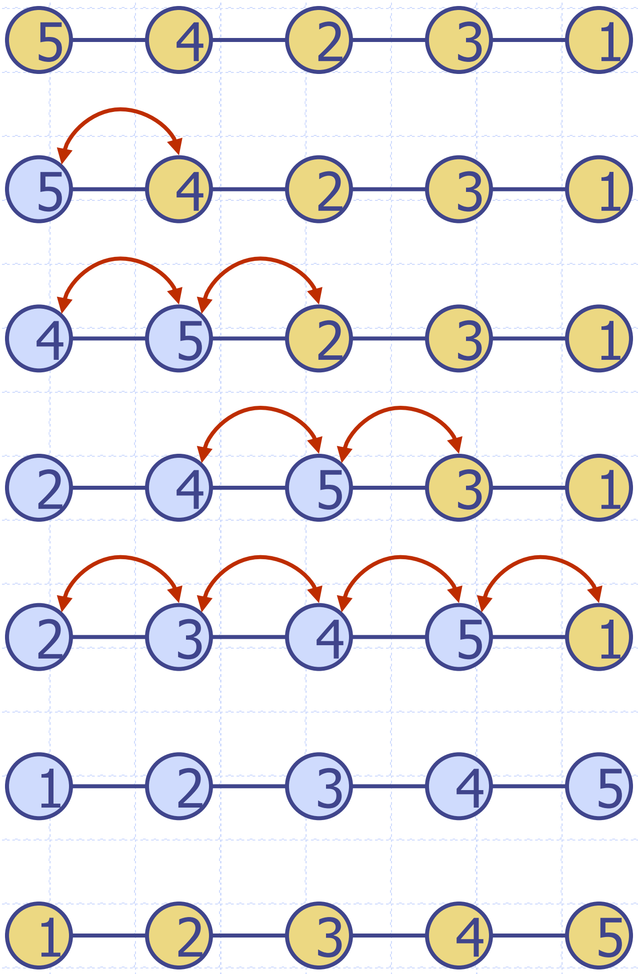
  - Removing the elements in sorted order from the priority queue with a series of $\boldsymbol{n}$ removeMin operations takes $O(\boldsymbol{n})$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

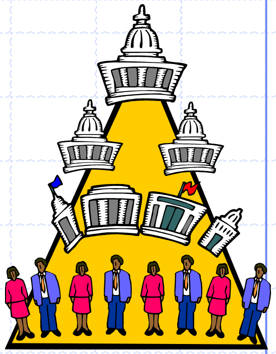
	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

In-place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



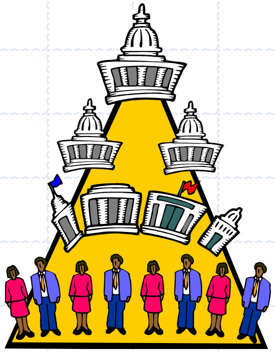
Heap-Sort



- 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

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

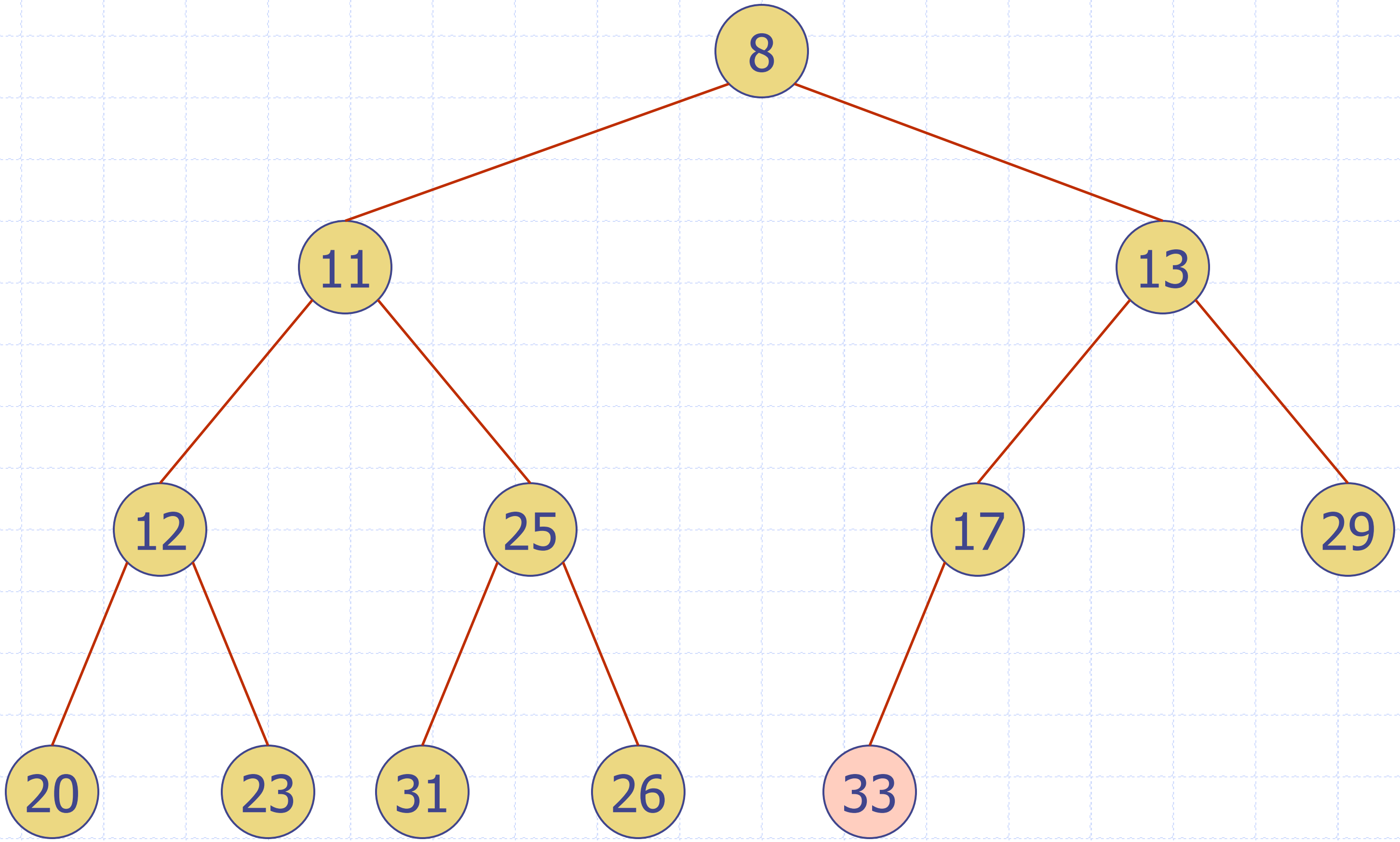


- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Heap Sort

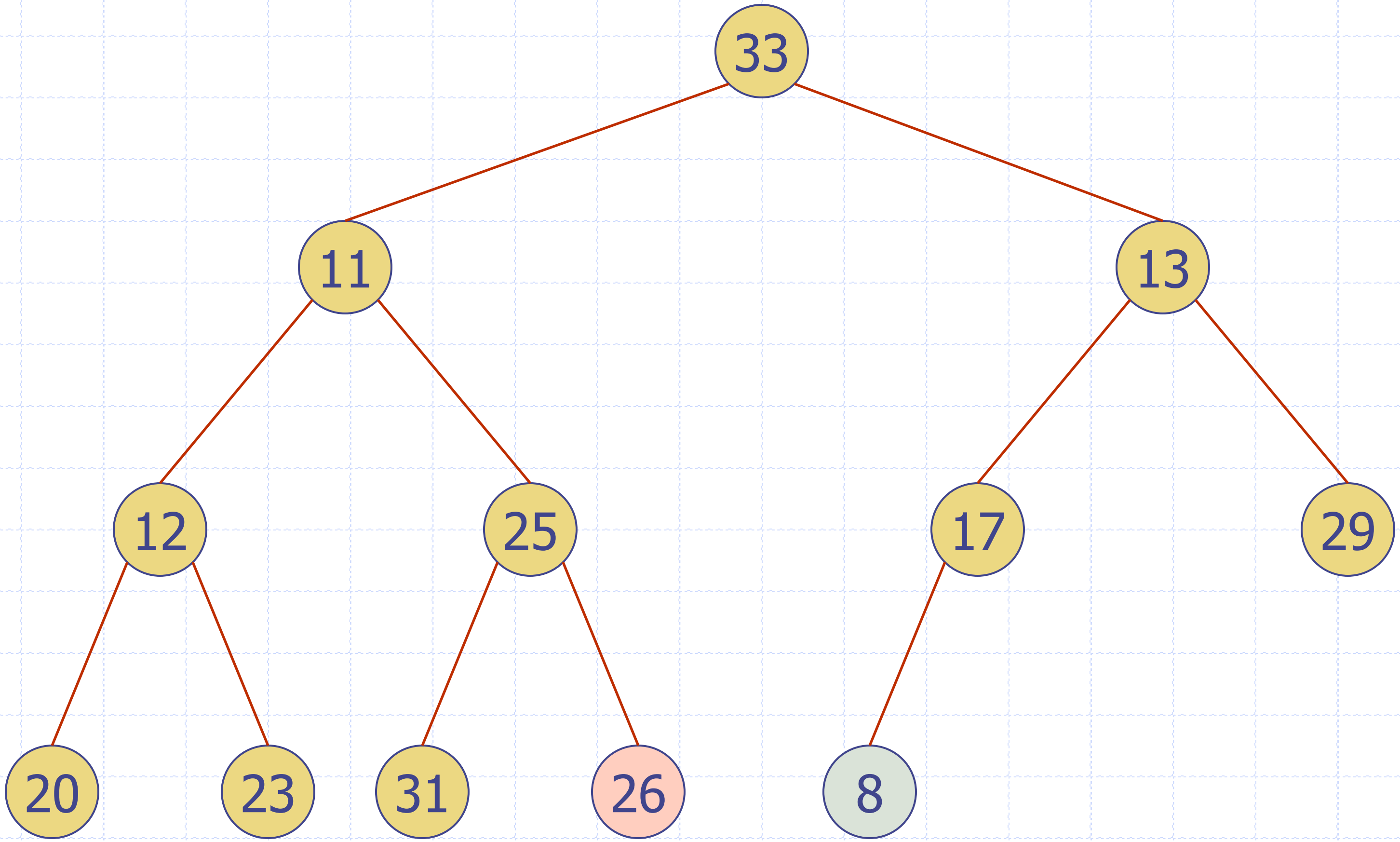
- Create a heap
- Do removeMin repeatedly till head becomes empty
- To do an in place sort, we move deleted element to end of heap

Heap Sort



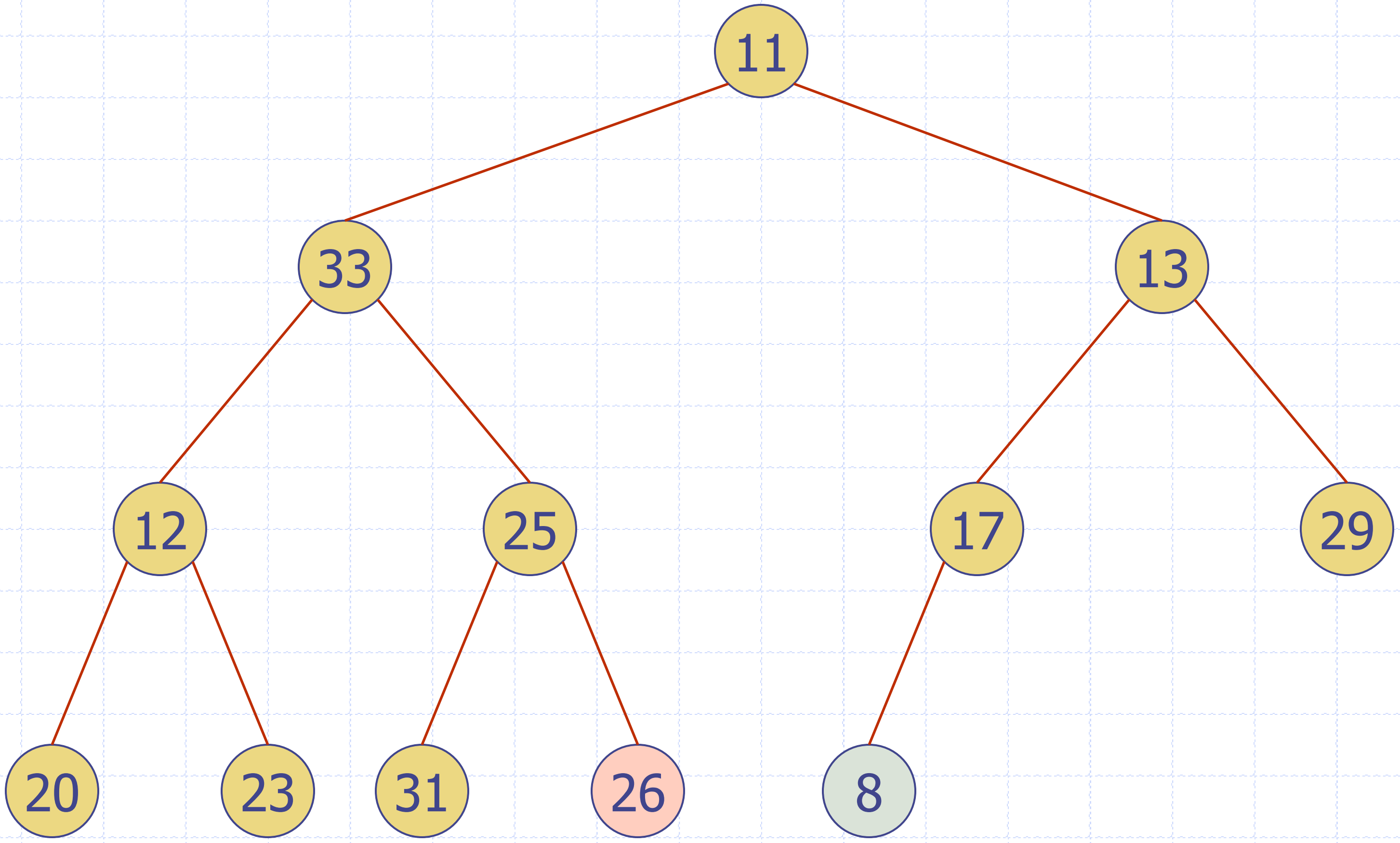
8	**11**	**13**	**12**	**25**	**17**	**29**	**20**	**23**	**31**	**26**	**33**

Heap Sort



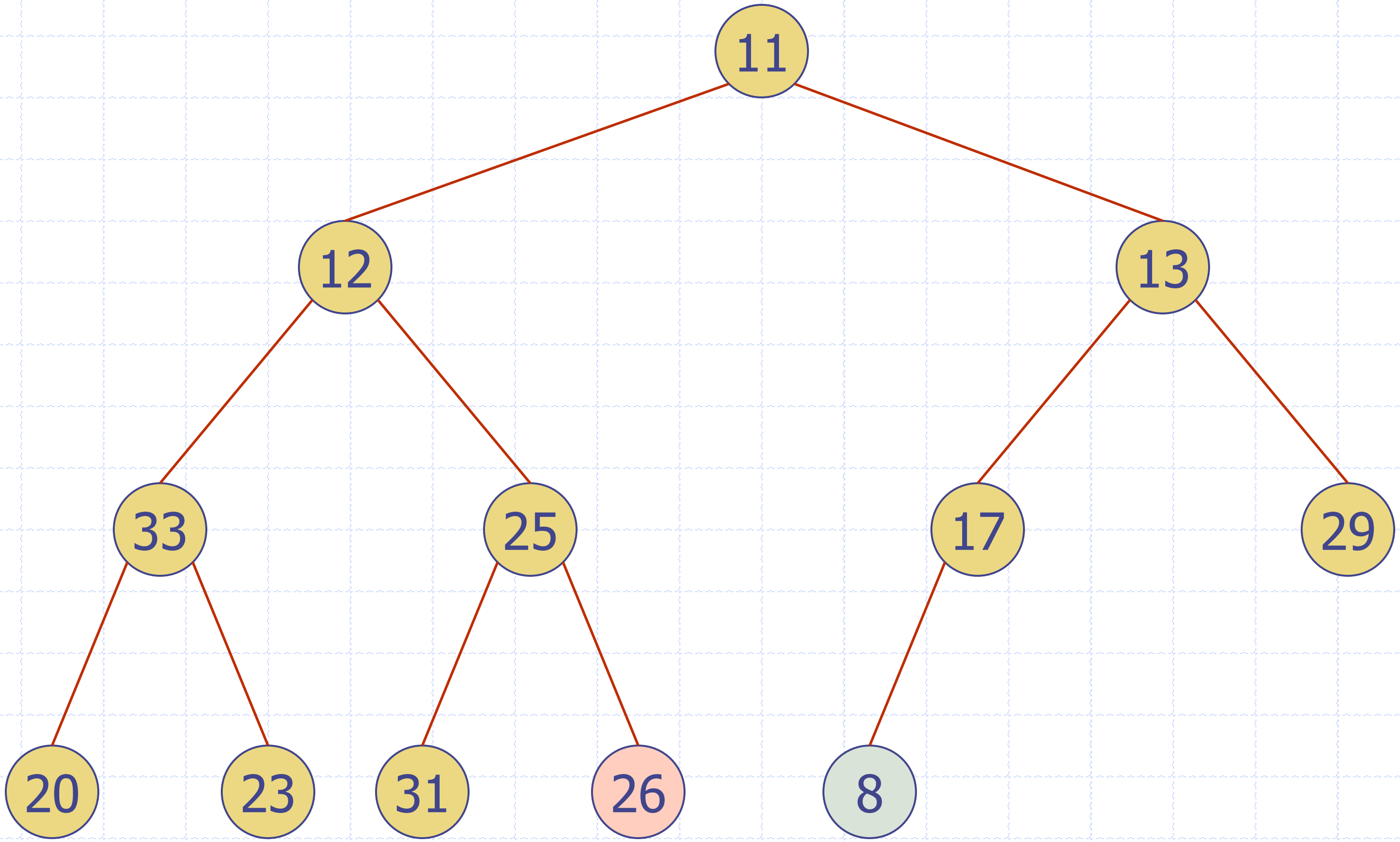
33	**11**	**13**	**12**	**25**	**17**	**29**	**20**	**23**	**31**	**26**	**8**

Heap Sort



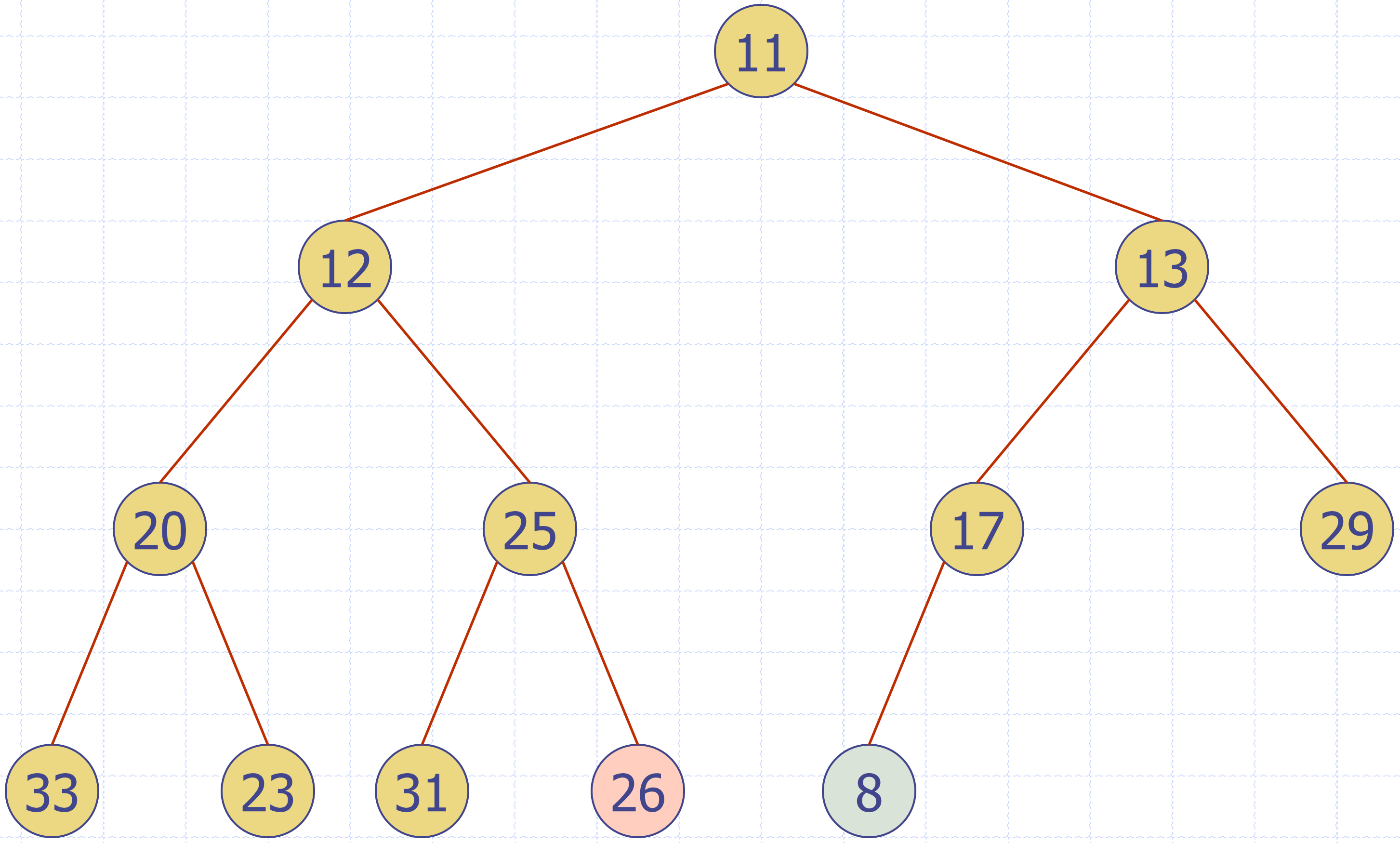
11	**33**	**13**	**12**	**25**	**17**	**29**	**20**	**23**	**31**	**26**	**8**

Heap Sort



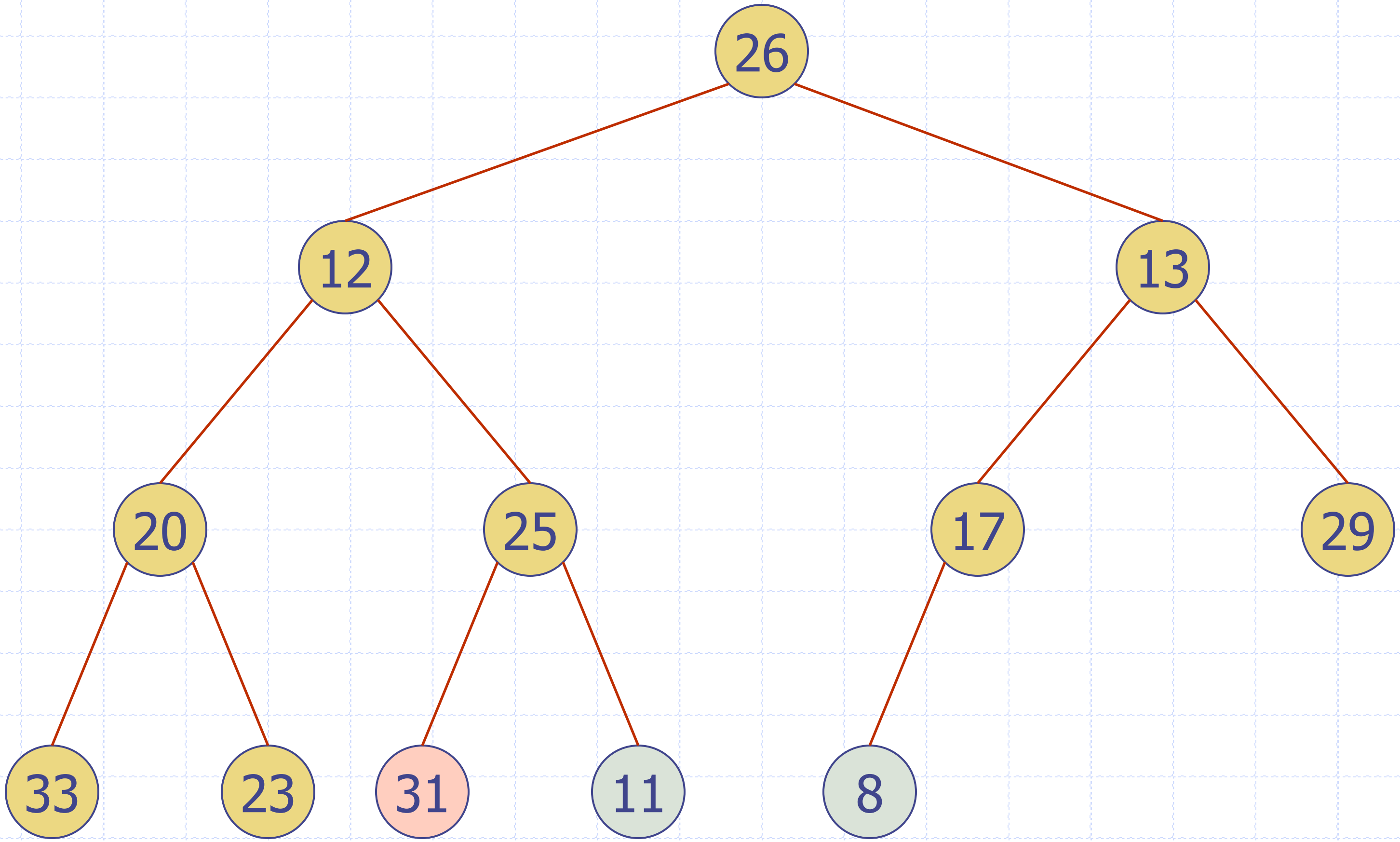
11	**12**	**13**	**33**	**25**	**17**	**29**	**20**	**23**	**31**	**26**	**8**

Heap Sort



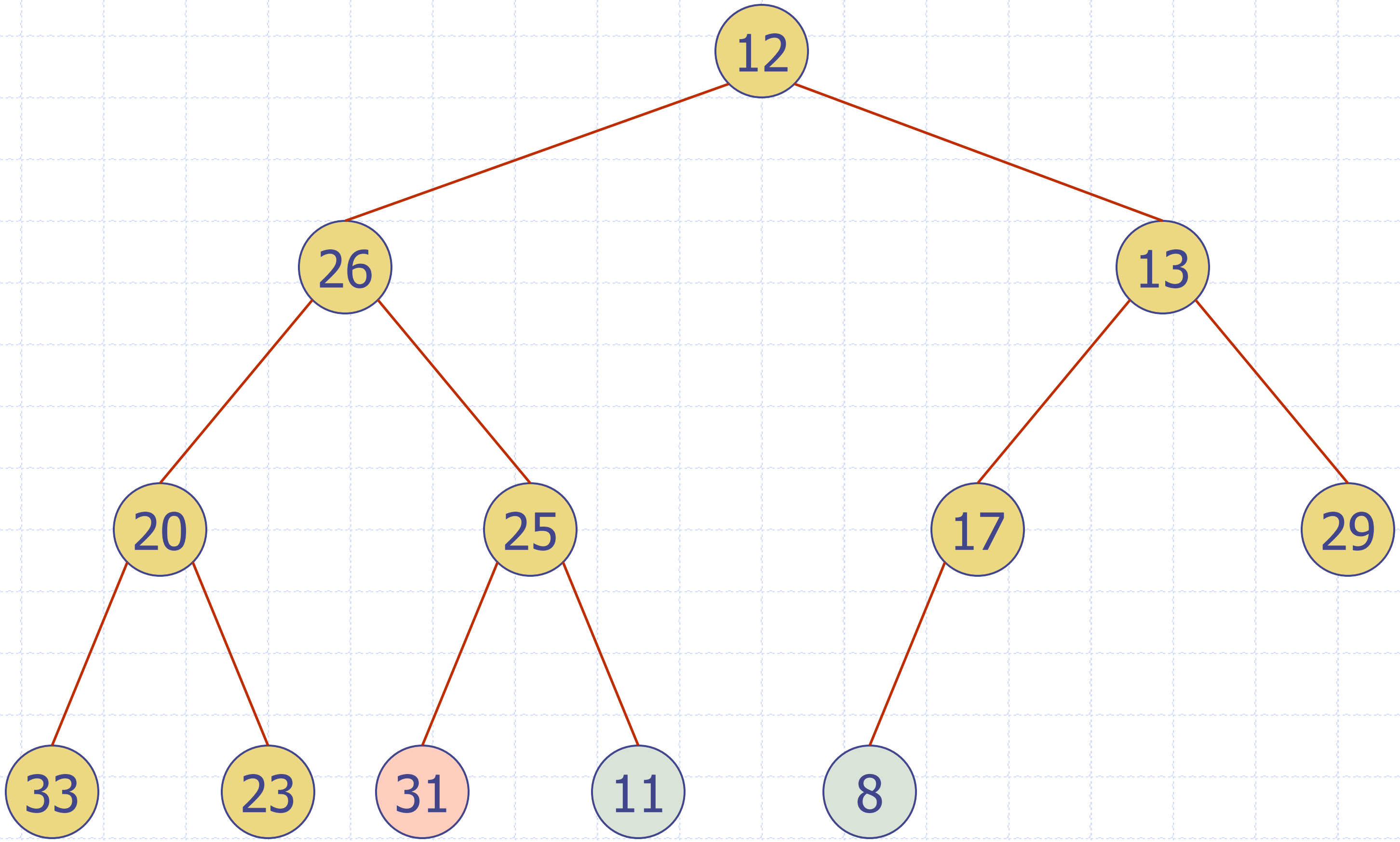
11	**12**	**13**	**20**	**25**	**17**	**29**	**33**	**23**	**31**	**26**	**8**

Heap Sort



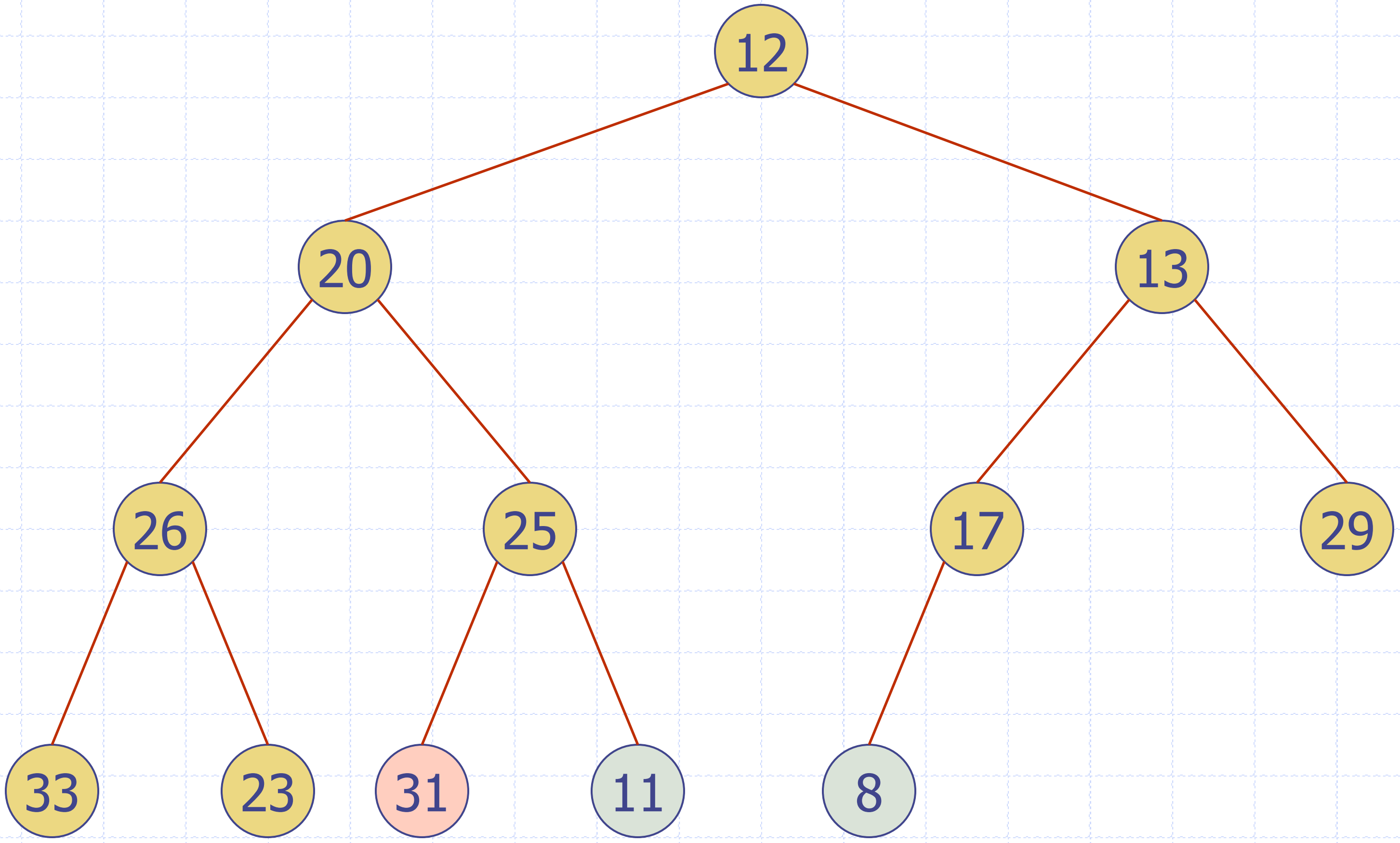
26	**12**	**13**	**20**	**25**	**17**	**29**	**33**	**23**	**31**	**11**	**8**

Heap Sort



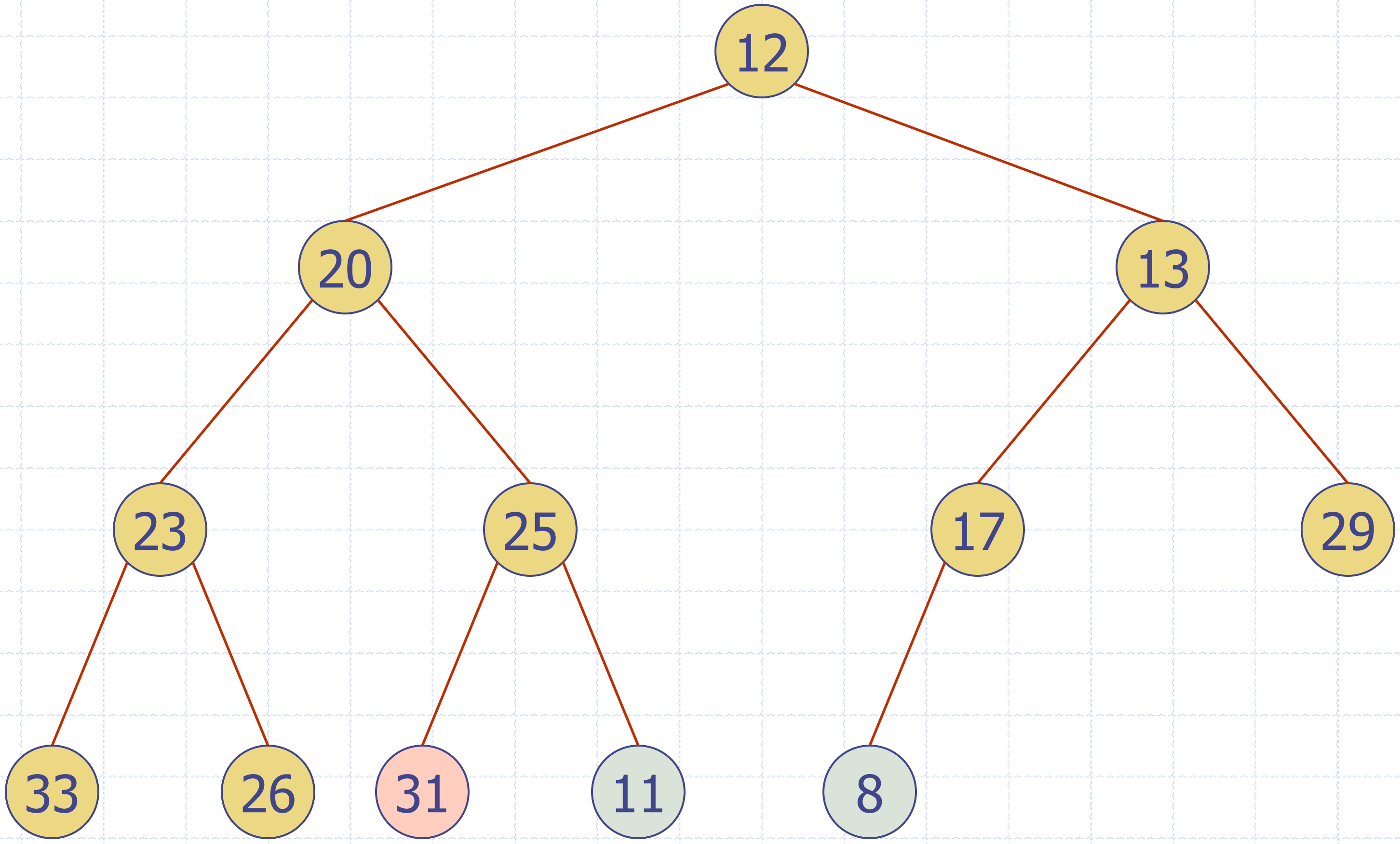
12	26	13	20	25	17	29	33	23	31	11	8

Heap Sort



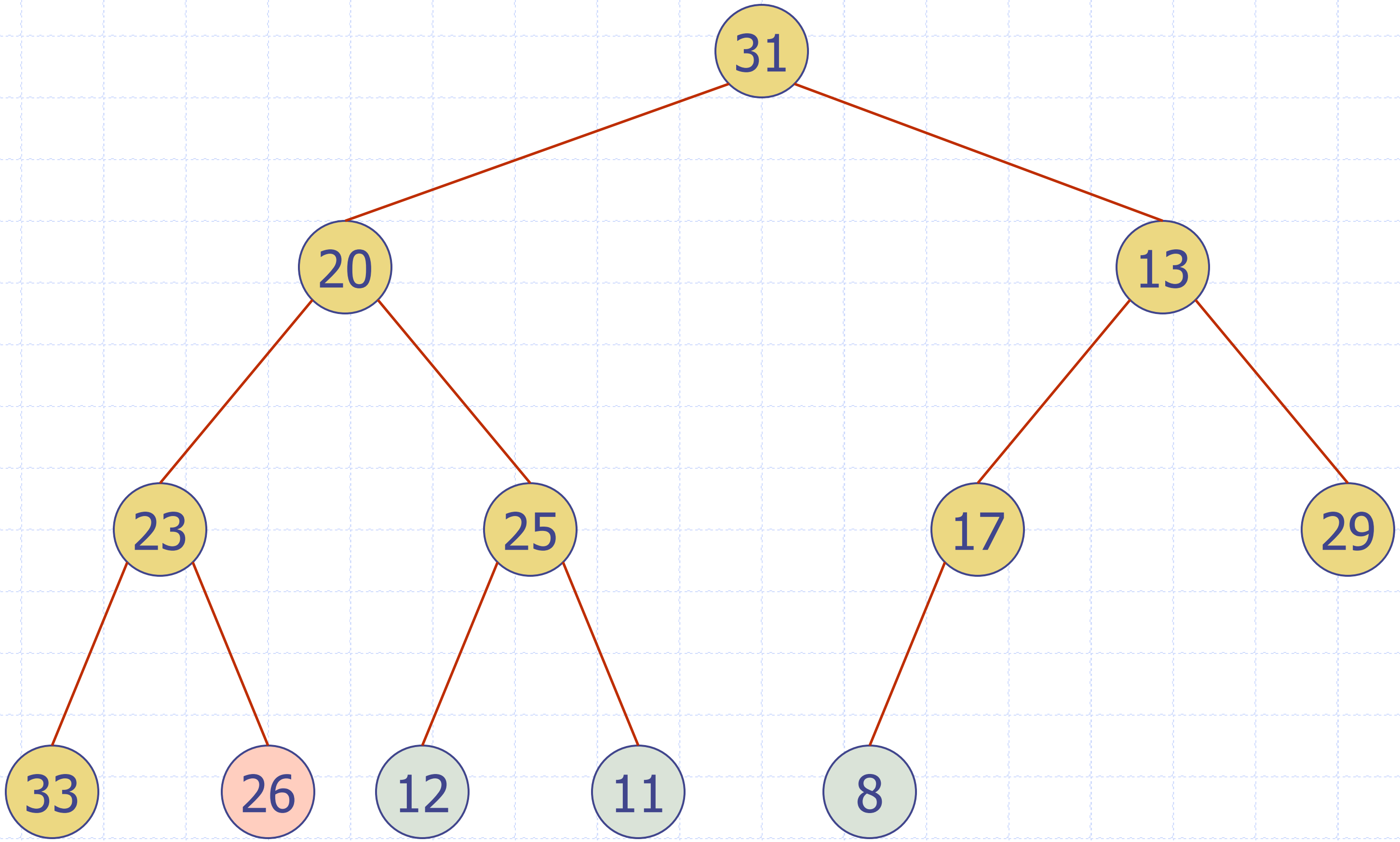
12	**20**	**13**	**26**	**25**	**17**	**29**	**33**	**23**	**31**	**11**	**8**

Heap Sort



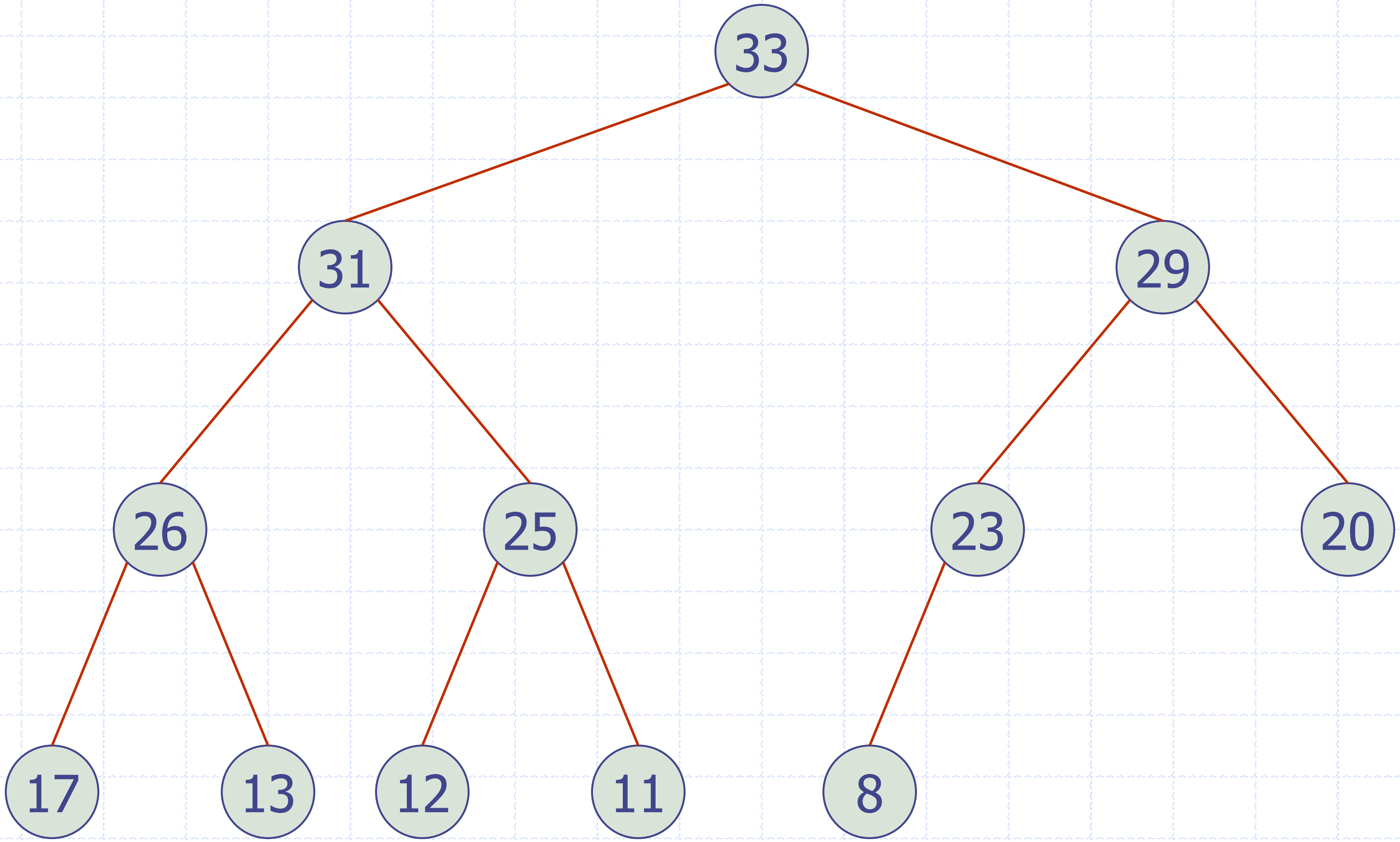
12	20	13	23	25	17	29	33	26	31	11	8

Heap Sort



31	**20**	**13**	**23**	**25**	**17**	**29**	**33**	**26**	**12**	**11**	**8**

Heap Sort



33	**31**	**29**	**26**	**25**	**23**	**20**	**17**	**13**	**12**	**11**	**8**

Heapsort - Analysis

- Create a heap - O(n)
- Do removeMin repeatedly till head becomes empty
  - O(log n) + O(log n-1) + O(log n-2) + … O(1) = O(n log n)
- To do an in place sort, we move deleted element to end of heap