CSI 2103: Data Structures

Priority Queues and Heaps (Ch 9)

Yonsei University

Spring 2022

Seong Jae Hwang

Aims

- More practical version of queue: Priority Queue
- Another data structure based on binary tree: Heap
- How PQ and Heap are related
- How we can sort a sequence of elements using PQ and Heap

Recall Queue

- Queue: FIFO
  - always first in, first out
  - no consideration of the elements' priorities
- Some applications may want to remove based on the priority
  - It's not a matter of "who came first", but "who is the most important"
- Priority Queue (PQ):
  - Each entry is a pair of (key, value)
  - Priority determined by key
    - key can be anything as long as it is ordinal
  - Highest priority entry to be removed is the one with the minimum (or maximum) key in the PQ

Priority Queue ADT

- For a priority queue P (priority based on min):
  - P.add(k, v): insert an item with key k and value v into P
  - P.remove_min(): return a tuple (k, v) with minimum key and remove it
  - P.min(): return a tuple (k, v) with minimum key without removing it
  - P.is_empty()
  - len(P)

Operation	**Return Value**	**Priority Queue**
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min( )	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min( )	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min( )	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min( )	(7,D)	{(9,C)}
P.remove_min( )	(9,C)	{ }
P.is_empty( )	True	{ }
P.remove_min( )	"error"	{ }

Implementation of PQ

- At this point, you may already be thinking if the list to implement PQ should be sorted or not
  - A: Both work with pros and cons trade-off
- Unsorted list

  - add: $O(1)$ time since we can add at the front or end
  - remove_min and min: $O(n)$ time since we have to find the smallest key
- Sorted list

  - add: $O(n)$ time since we have to find the place to add and keep it sortrted
  - remove_min and min: $O(1)$ time since the smallest key is already at the front

Application of PQ

- A simple application of PQ is to directly use PQ to sort a list of comparable elements: PQ sorting
- Give an unsorted input list, simply add all the elements one by one into the PQ
- Then, remove the elements using remove_min()
- The output is, by construction of PQ, sorted (by key)!
- The running time depends on the PQ implementation

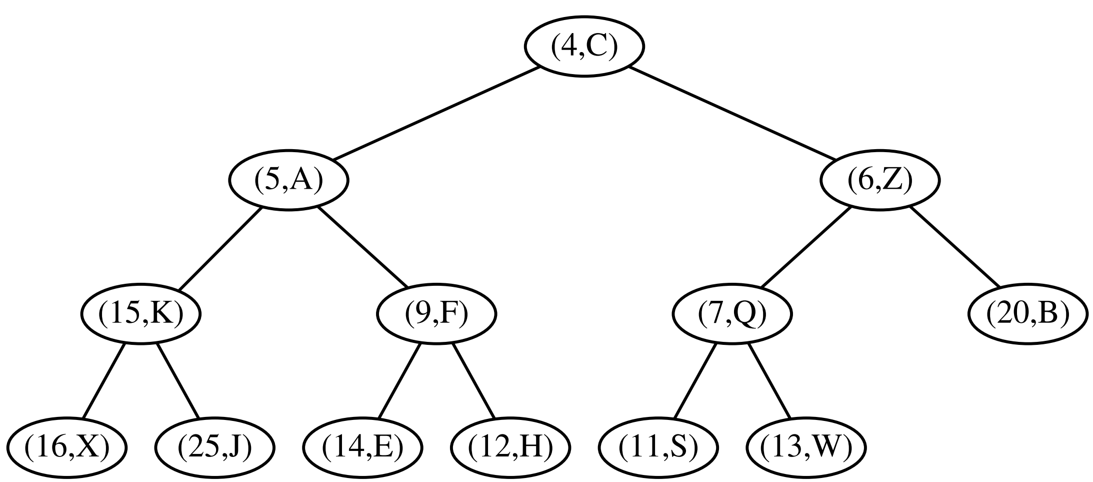
Again, the Trade-off

- Can we use another data structure to balance this trade-off?
  - Instead of an array-based list, use a binary tree

Operation	Unsorted List	Sorted List
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

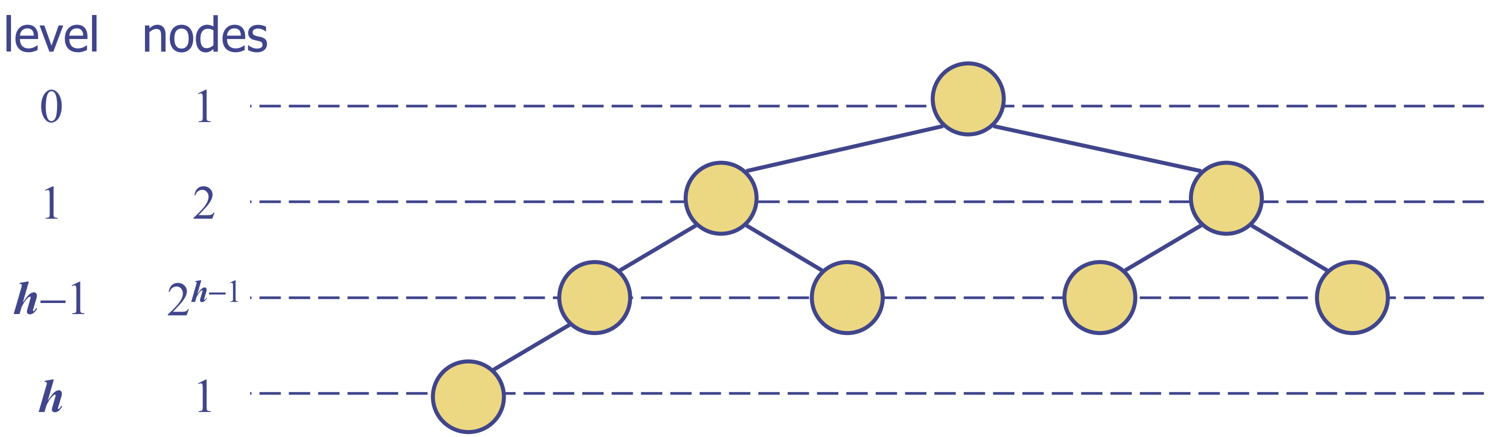
Heaps

- A heap is a binary tree with the following properties:
  - Heap-Order: for every internal node v, key(v) ≥ key(parent(v))
    - If max-heap, then key(v) ≤ key(parent(v))
    - min (or max) of the tree (or subtree) at the top
    - heap means a pile
  - Complete Binary Tree: A heap T with height h is complete if
    - for levels $i = 0, \ldots, h-1$, each level has the maximum number of nodes possible (level i has 2^i nodes)
    - at level h, all leaves are at the left most possible positions
      - another way to say this is that the level-numbering is from 0 to n-1



Height of a Heap

- Why complete tree? Small height!
- Proposition 9.2: A heap T storing n entries has height $O(\log n)$. Precisely, $h = \lfloor \log n \rfloor$.
  - levels 0 through $h-1$: $1+2+4+\cdots+2^{h-1} = 2^h - 1$ nodes
  - level h: at least 1 node to at most 2^h nodes
  - Thus, $2^h \le n \le 2^{h+1} - 1$
  - taking the log (base 2) : $h \le \log n$ and $\log(n+1) - 1 \le h$
  - Since h is an integer, $h = \lfloor \log n \rfloor$



example with h = 3

PQ with Heap

- We saw that unsorted or sorted lists for PQ have trade-offs in time complexity

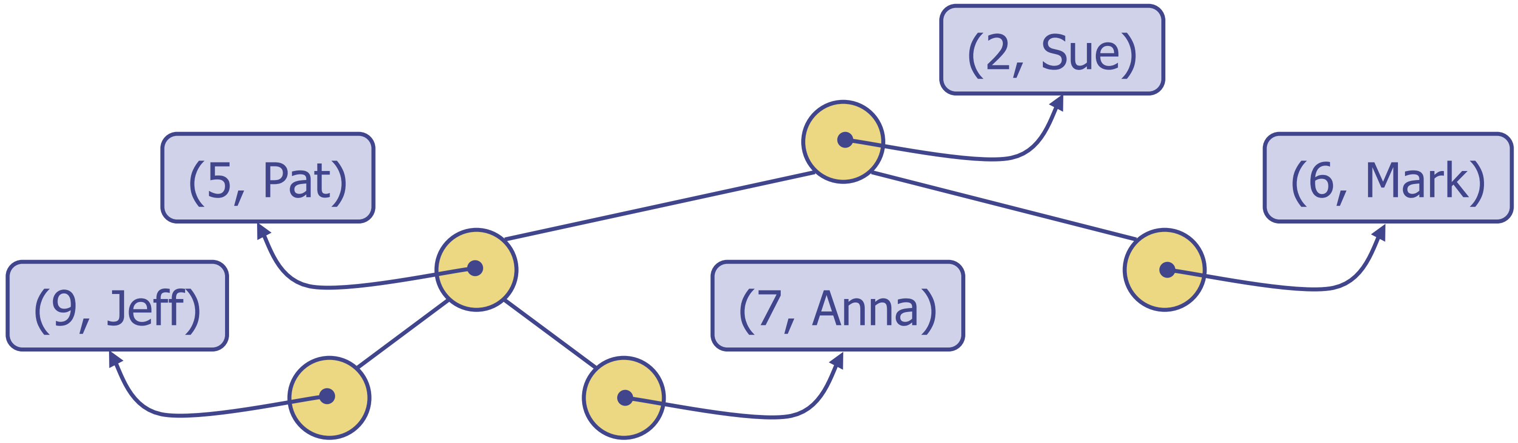
Operation	Unsorted List	Sorted List
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

- Heap is an efficient data structure for keeping track of the min (or max) key
- Use heap to implement a PQ: perform add and remove_min
  - Pro: by construction, keeps track of the min (or max) node
  - Pro: time complexity of operations depend on height h, not n, and a complete binary tree has $h \ll n$!

PQ with Heap

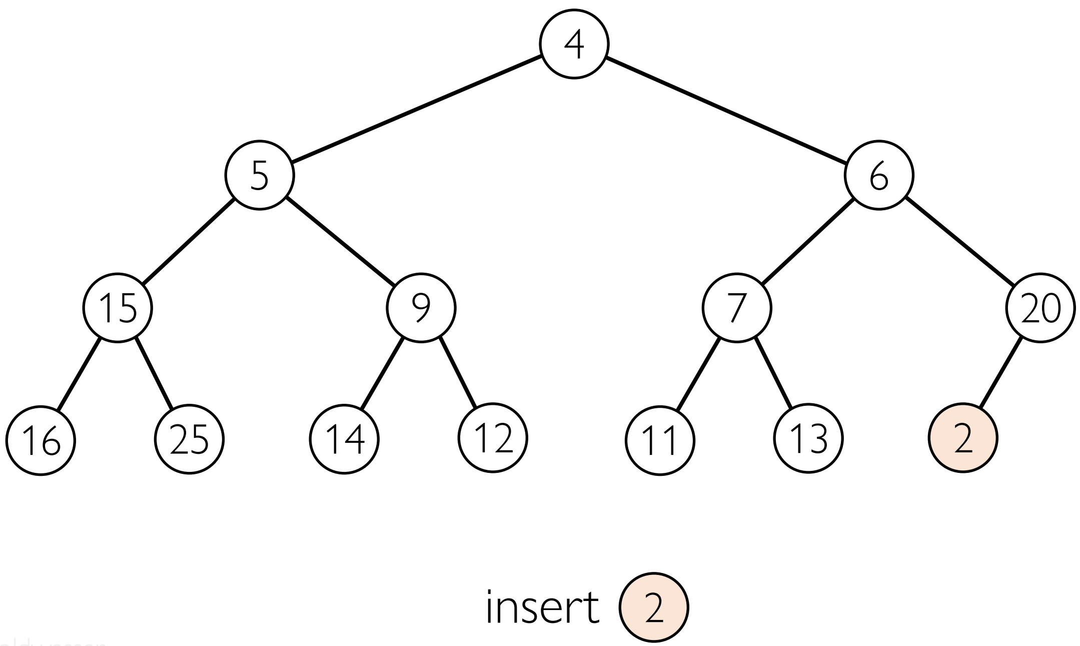
Implement PQ with a Heap

1. Each node has (key, value)
2. Keep track of the "last node"
    1. last level numbering index, which is also
    2. the right-most node of the bottom most level



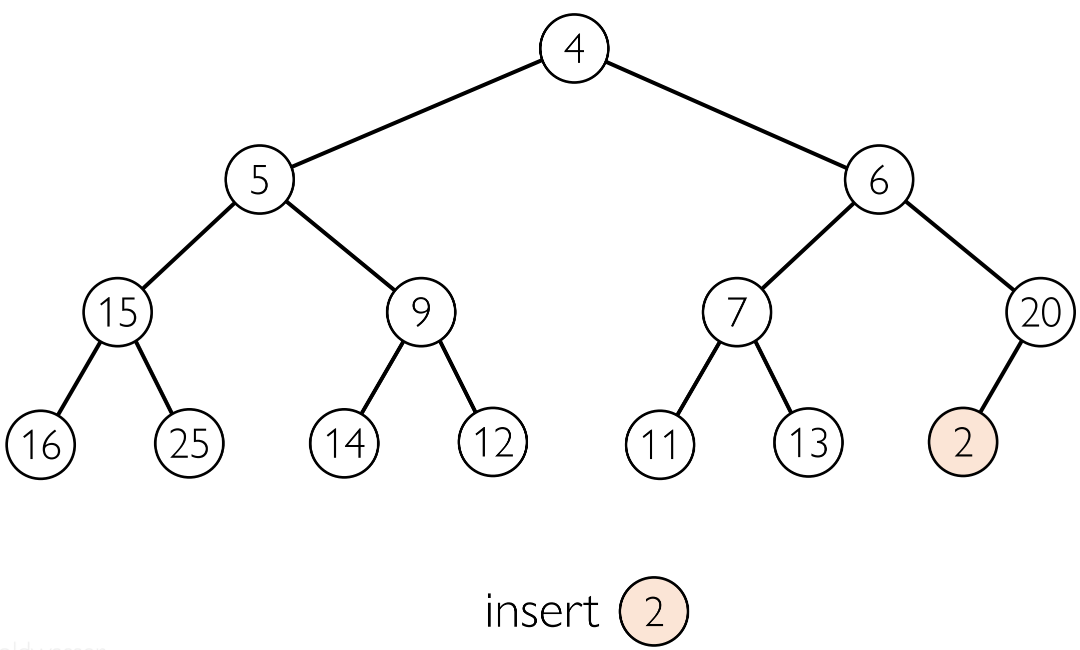
Insertion into a Heap

- add(k, v) in PQ = heap insertion
- Simply add a new node just next to the rightmost node at the bottom level (or leftmost position if the bottom level is full)
- But this may violate the heap-order property!
- Need to organize the tree to restore the heap-order property



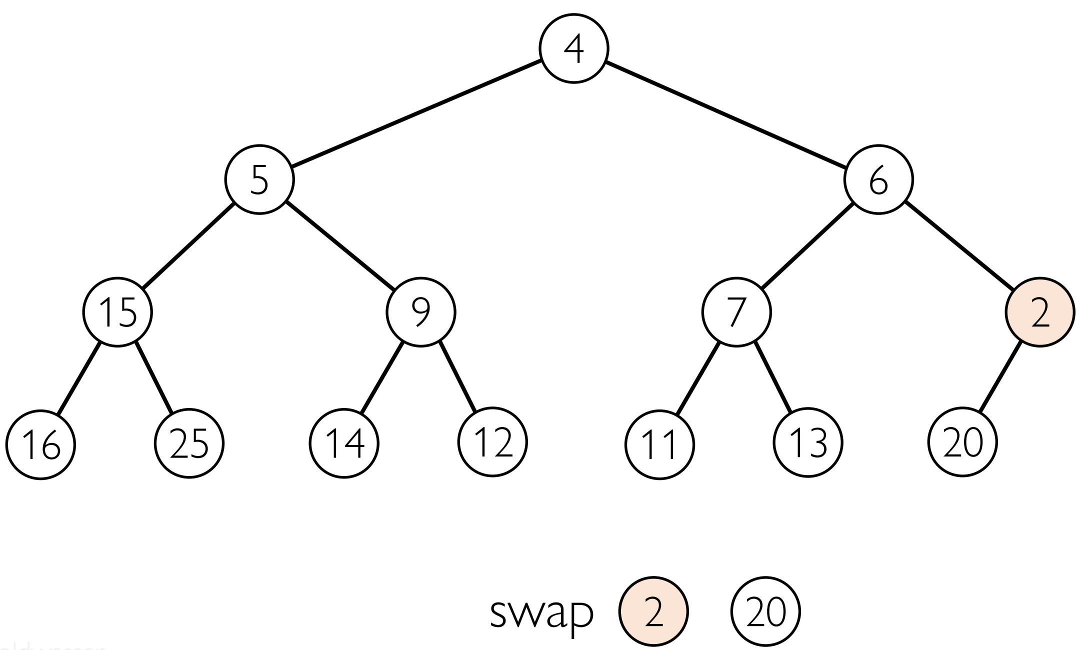
Up-heap bubbling

- Swap the inserted node up the tree until the heap-order property is satisfied
  - in a min-heap, parent key ≤ children keys
- Up-heap bubbling (up-heap for short)



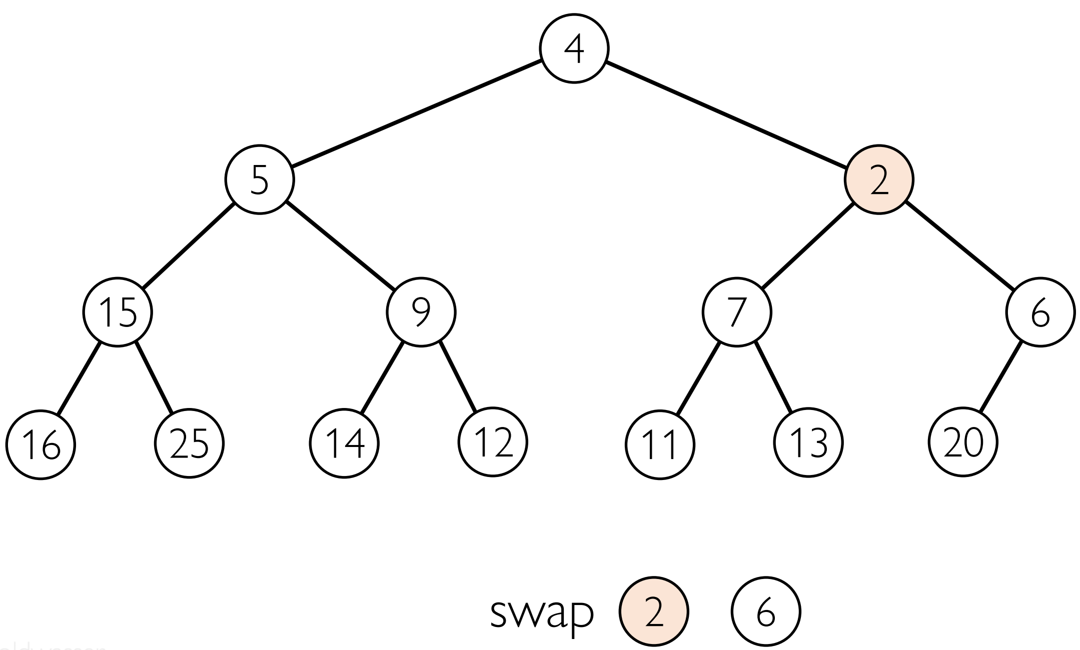
Up-heap bubbling

- Swap the inserted node up the tree until the heap-order property is satisfied
  - in a min-heap, parent key ≤ children keys
- Up-heap bubbling (up-heap for short)



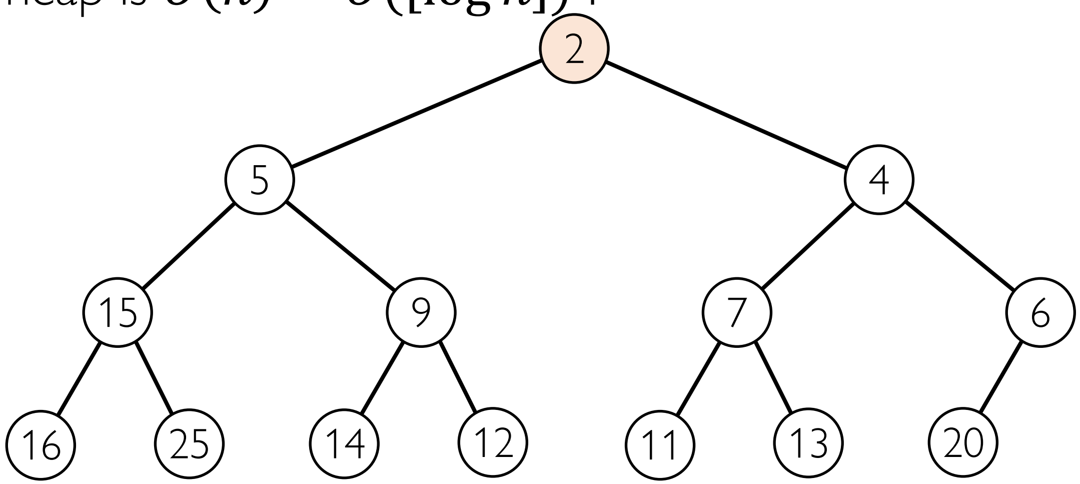
Up-heap bubbling

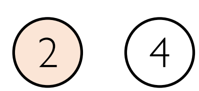
- Swap the inserted node up the tree until the heap-order property is satisfied
  - in a min-heap, parent key ≤ children keys
- Up-heap bubbling (up-heap for short)



Up-heap bubbling

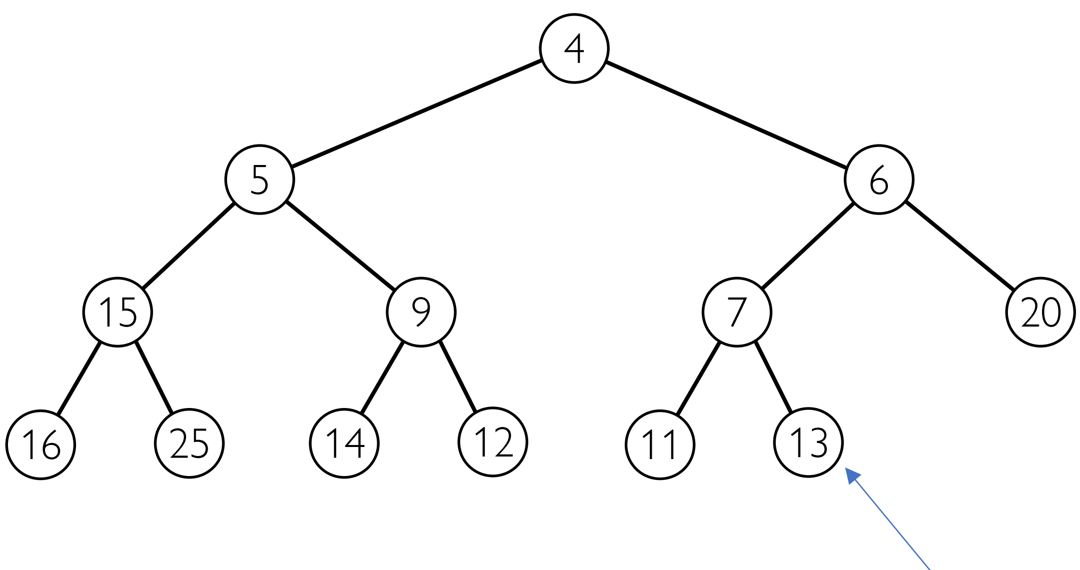
- Swap the inserted node up the tree until the heap-order property is satisfied
  - in a min-heap, parent key ≤ children keys
- Up-heap bubbling (up-heap for short)
- add(k, v) requires insert + up-heap
  - up-heap is $O(h) = O(\lfloor \log n \rfloor)$!



swap 

Removal from a Heap

- remove_min() in PQ = heap removal of root node
- Again, we know the min by how heap is constructed
- But removing the root turns T into a two disconnected subtrees
- Instead, (1) replace root with the "last" node, (2) then remove the "last" node

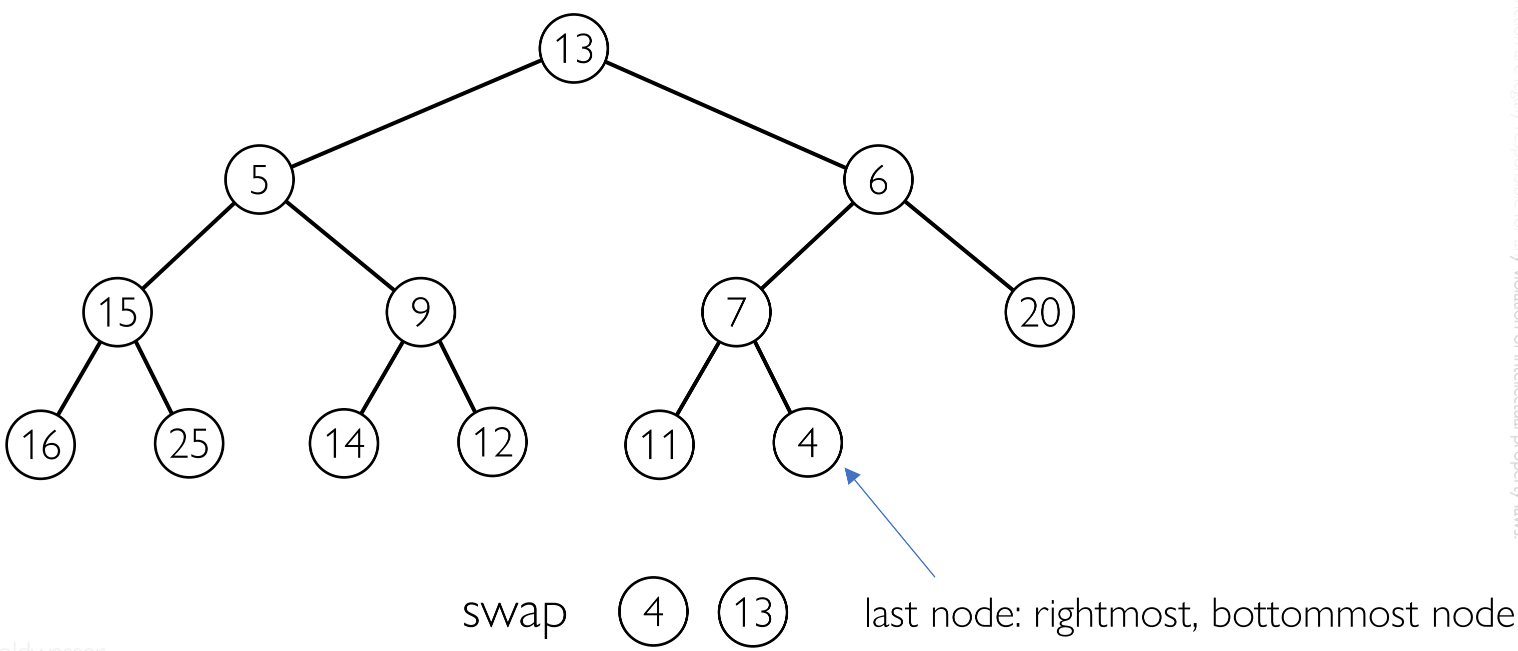


remove 4

last node: rightmost, bottommost node

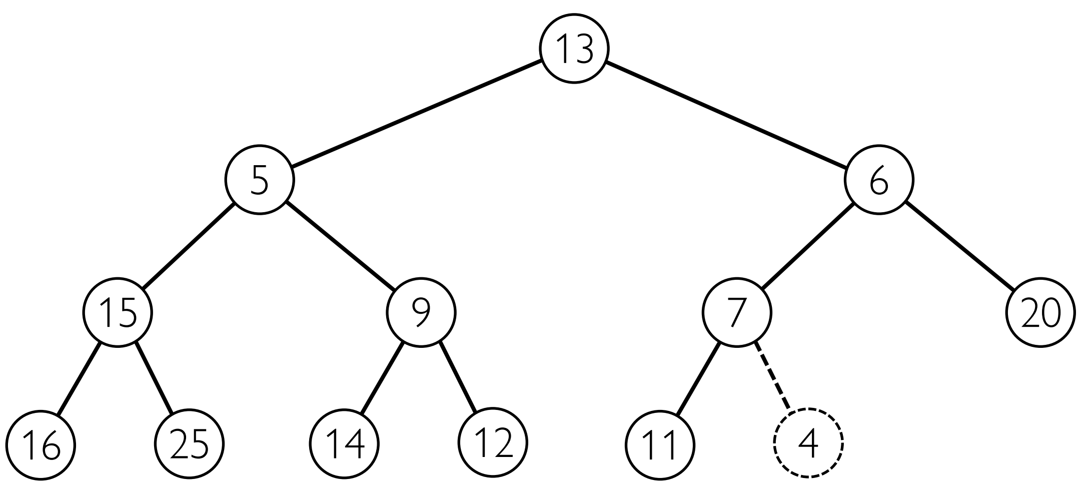
Removal from a Heap

- remove_min() in PQ = heap removal of root node
- Again, we know the min by how heap is constructed
- But removing the root turns T into a two disconnected subtrees
- Instead, (1) replace root with the "last" node, (2) then remove the "last" node



Removal from a Heap

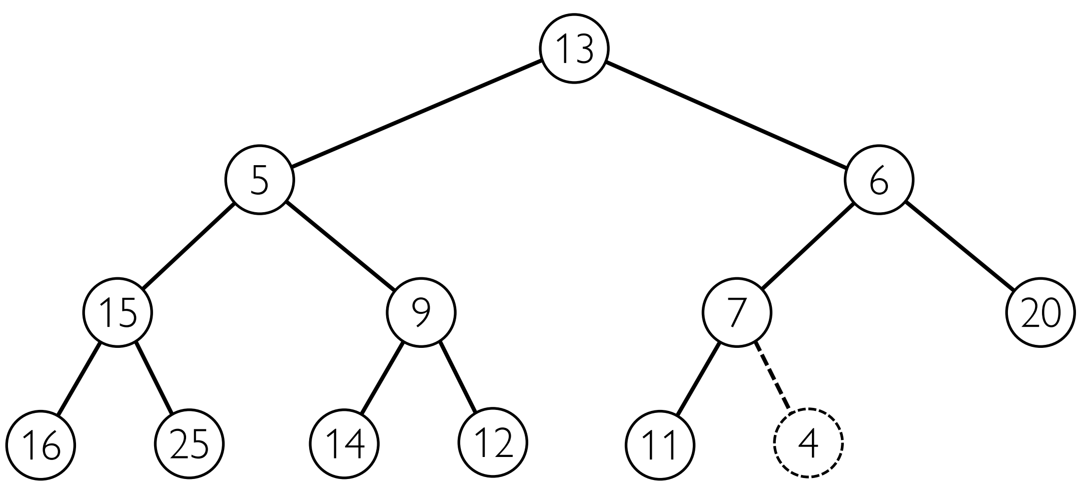
- remove_min() in PQ = heap removal of root node
- Again, we know the min by how heap is constructed
- But removing the root turns T into a two disconnected subtrees
- Instead, (1) replace root with the "last" node, (2) then remove the "last" node
- heap-property violated again



remove 4

Down-heap bubbling

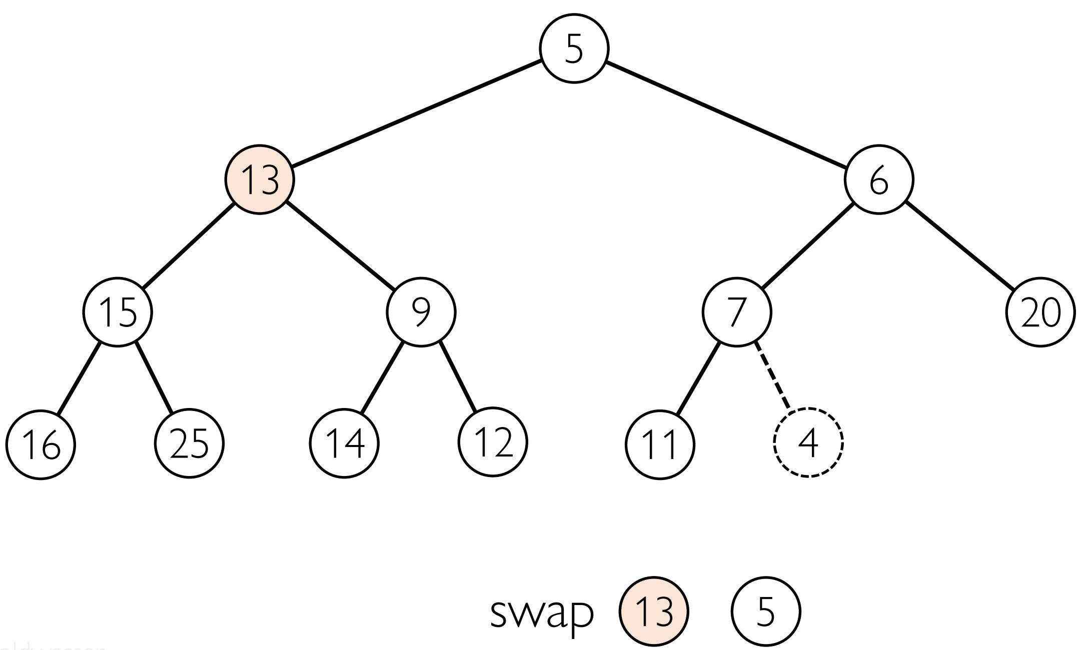
- Swap the inserted node down the tree until the heap-order property is satisfied
  - in a min-heap, parent key ≤ children keys
  - left child or right child?: the one with the smaller key
    - Otherwise, the swapped sibling will violate the heap property again!
- Down-heap bubbling (down-heap for short)



remove 4

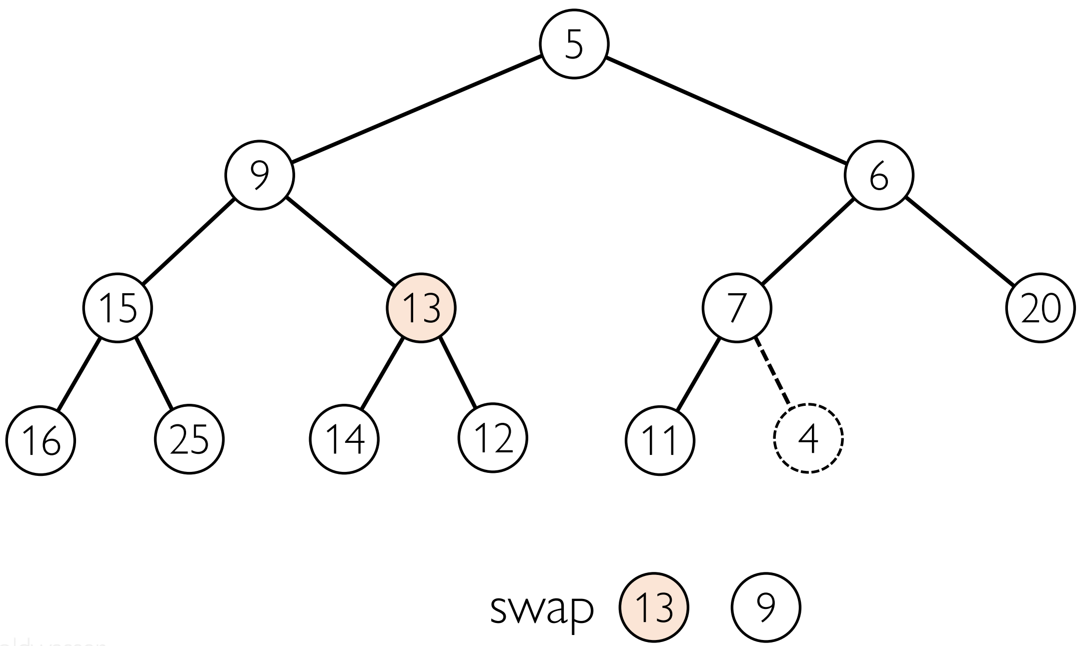
Down-heap bubbling

- Swap the inserted node down the tree until the heap-order property is satisfied
  - in a min-heap, parent key ≤ children keys
  - left child or right child?: the one with the smaller key
    - Otherwise, the swapped sibling will violate the heap property again!
- Down-heap bubbling (down-heap for short)



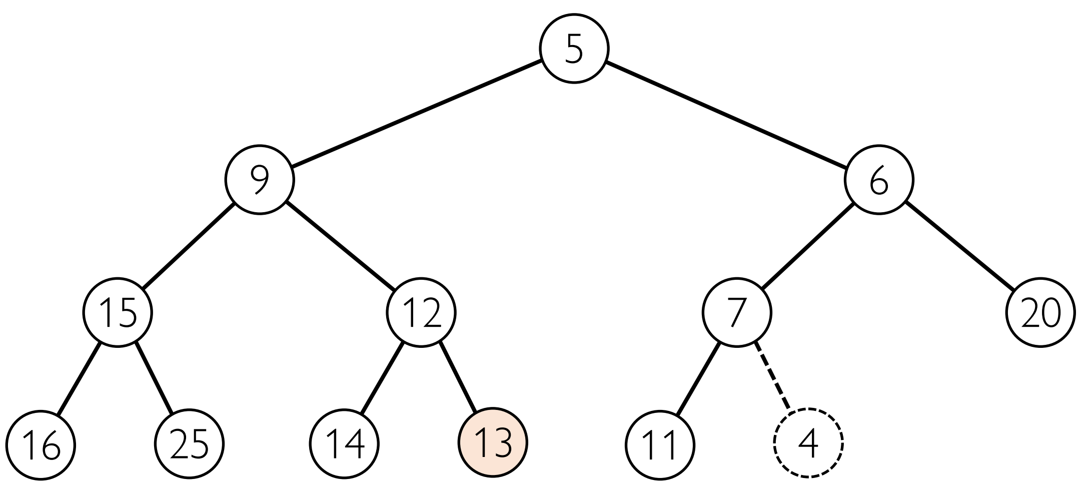
Down-heap bubbling

- Swap the inserted node down the tree until the heap-order property is satisfied
  - in a min-heap, parent key ≤ children keys
  - left child or right child?: the one with the smaller key
    - Otherwise, the swapped sibling will violate the heap property again!
- Down-heap bubbling (down-heap for short)



Down-heap bubbling

- Swap the inserted node down the tree until the heap-order property is satisfied
  - in a min-heap, parent key ≤ children keys
  - left child or right child?: the one with the smaller key
    - Otherwise, the swapped sibling will violate the heap property again!
- Down-heap bubbling (down-heap for short)

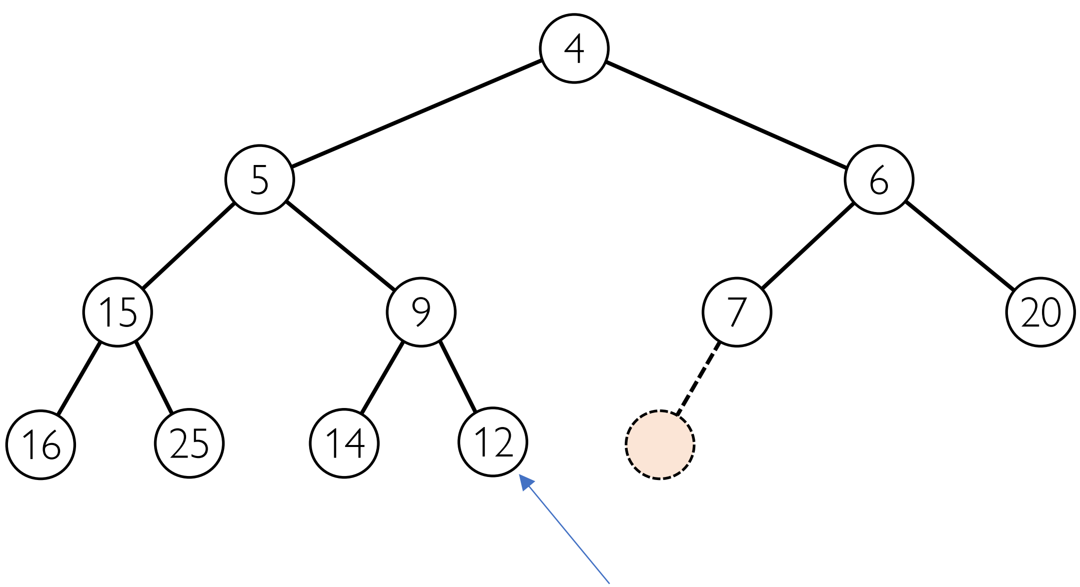


remove_min is also $O(h) = O(\lfloor \log n \rfloor)$!

swap 13 12

How to update the "last" node?

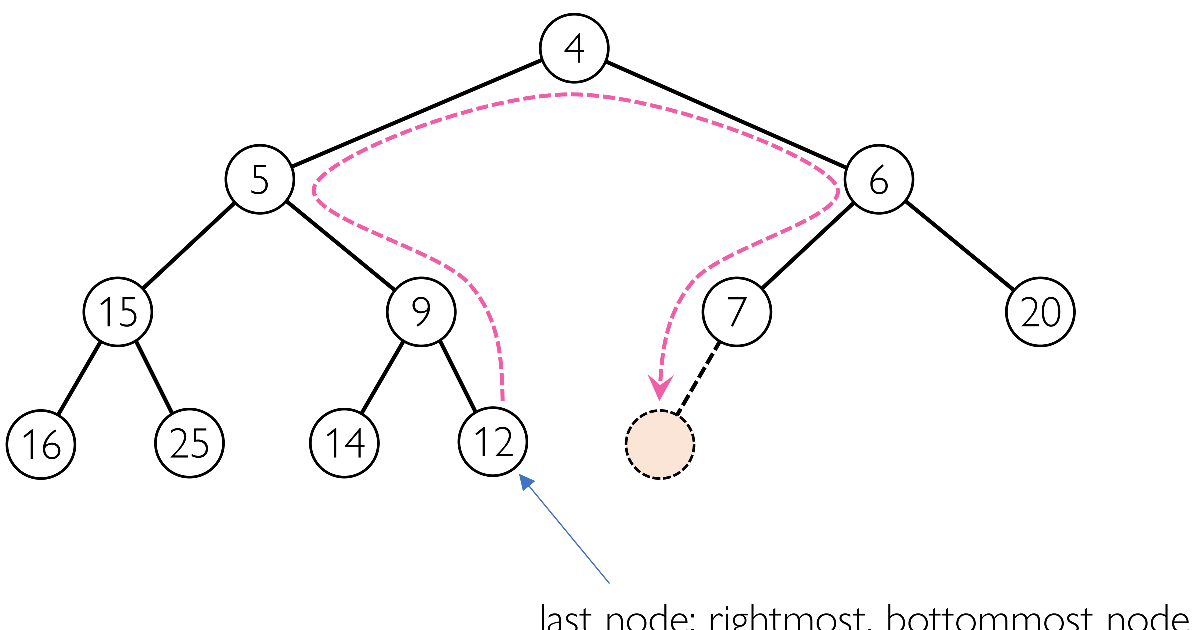
- After an insertion, the new node must be somewhere, and that node now needs to be the new last node
- How do we locate where to add the new node ?
  - Hint: we know the current "last node"



last node: rightmost, bottommost node

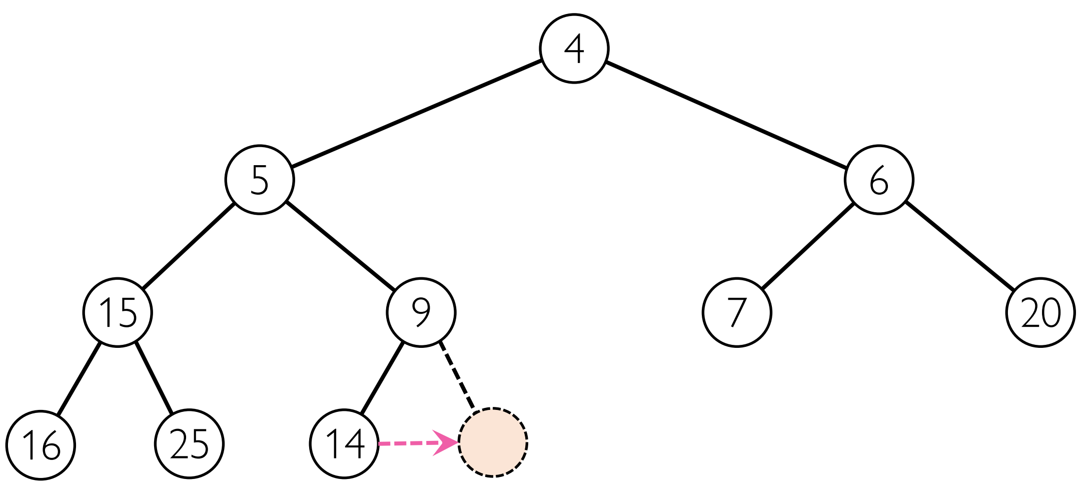
How to update the “last” node?

- Starting from the last node, go up until a left child or root is reached
- If a left child is reached (this includes the last node itself), go to the right child
- Go down left until a leaf is reached



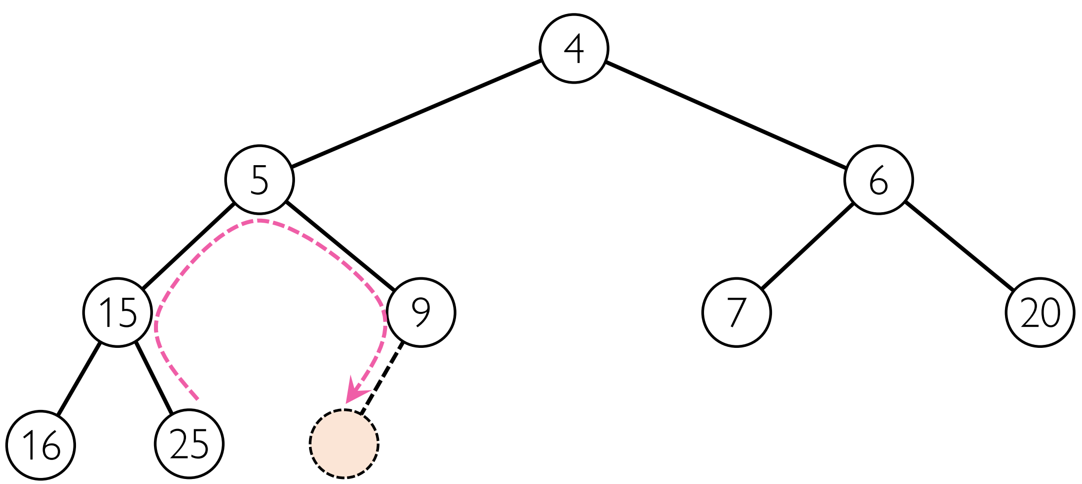
How to update the “last” node?

- Starting from the last node, go up until a left child or root is reached
- If a left child is reached (this includes the last node itself), go to the right child
- Go down left until a leaf is reached



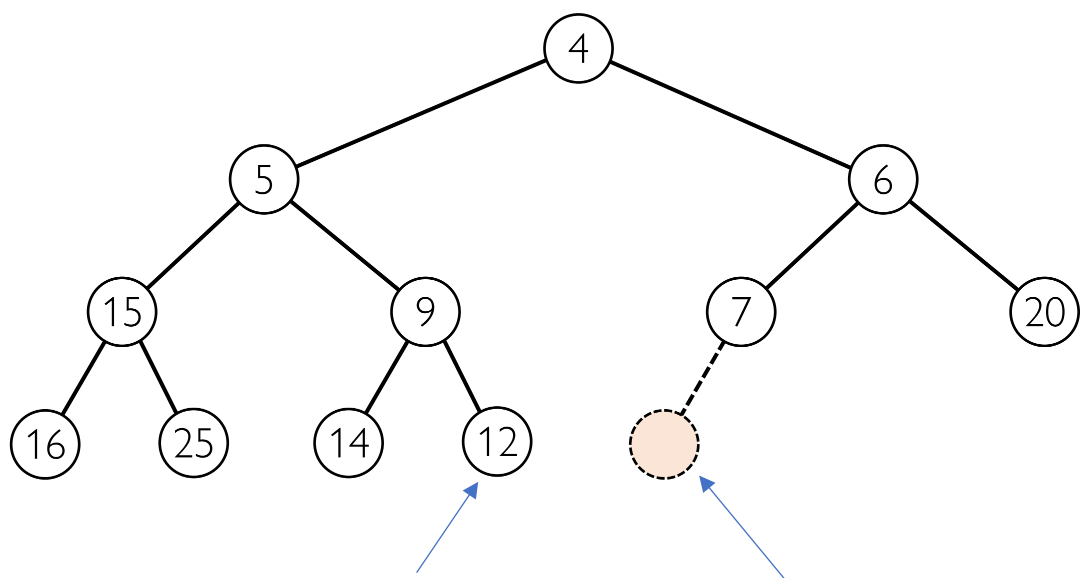
How to update the “last” node?

- Starting from the last node, go up until a left child or root is reached
- If a left child is reached (this includes the last node itself), go to the right child
- Go down left until a leaf is reached



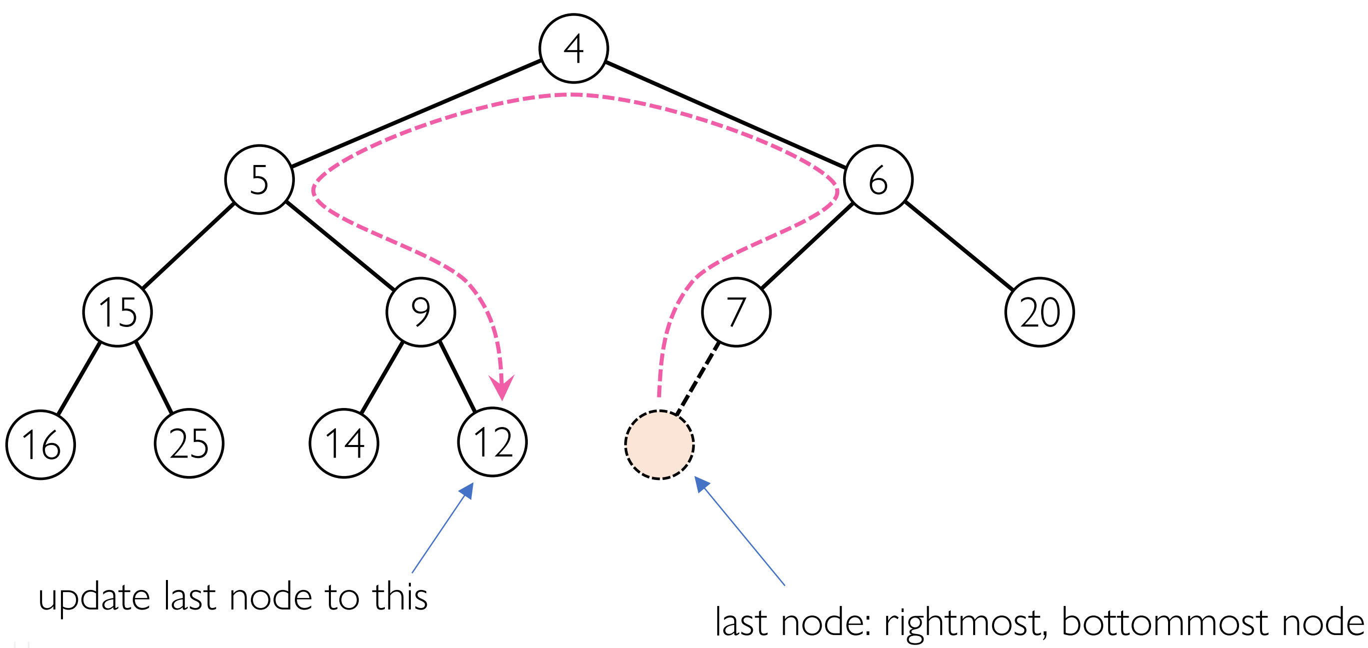
How to update the “last” node?

- What about the update after the last node is removed?
- Hint: very similar to what we did for the insertion update



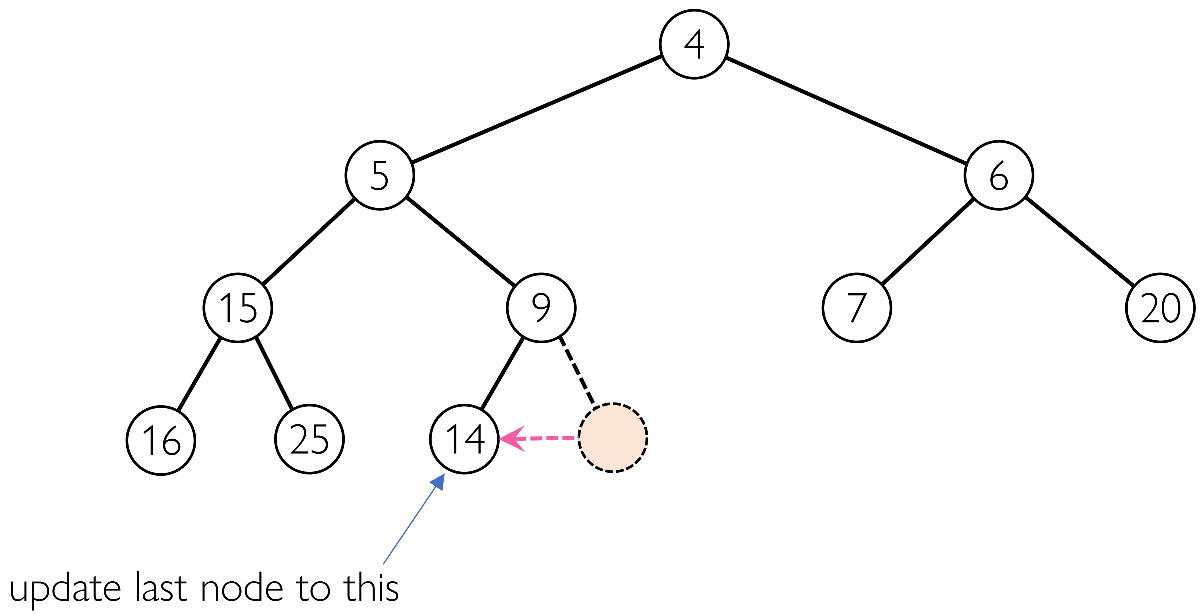
How to update the "last" node?

- Starting from the last node, go up until a right child or root is reached
- If a right child is reached (this includes the last node itself), go to the left child
- Go down right until a leaf is reached



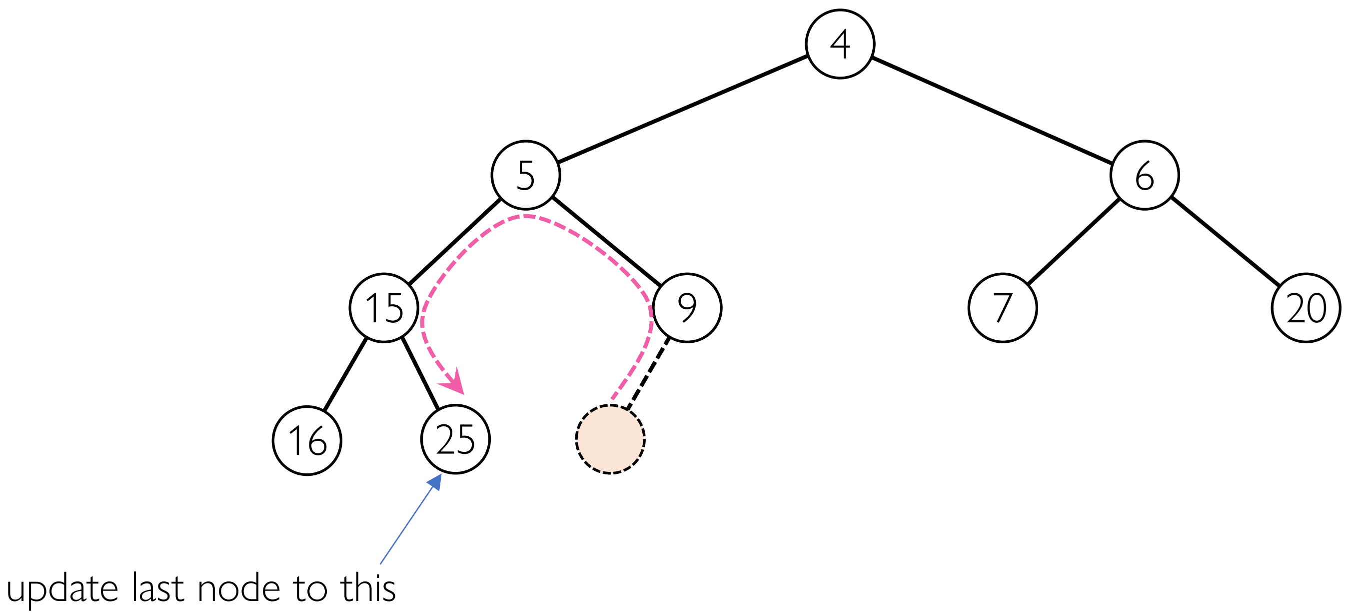
How to update the "last" node?

- Starting from the last node, go up until a right child or root is reached
- If a right child is reached (this includes the last node itself), go to the left child
- Go down right until a leaf is reached



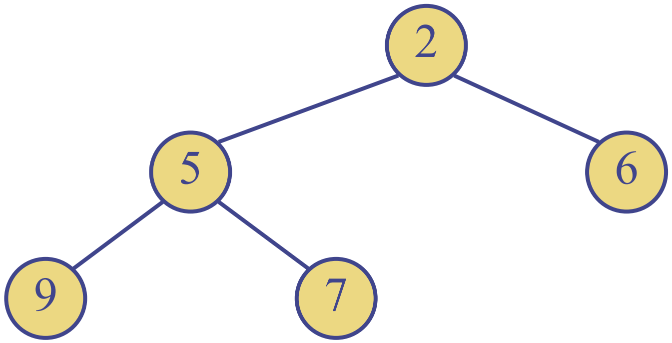
How to update the "last" node?

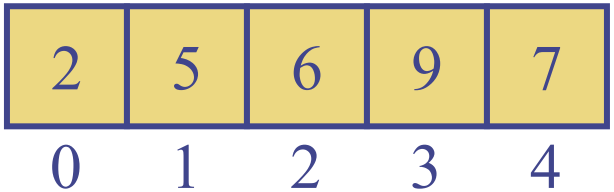
- Starting from the last node, go up until a right child or root is reached
- If a right child is reached (this includes the last node itself), go to the left child
- Go down right until a leaf is reached



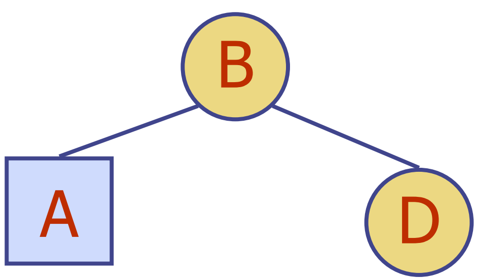
Implementation of Heap

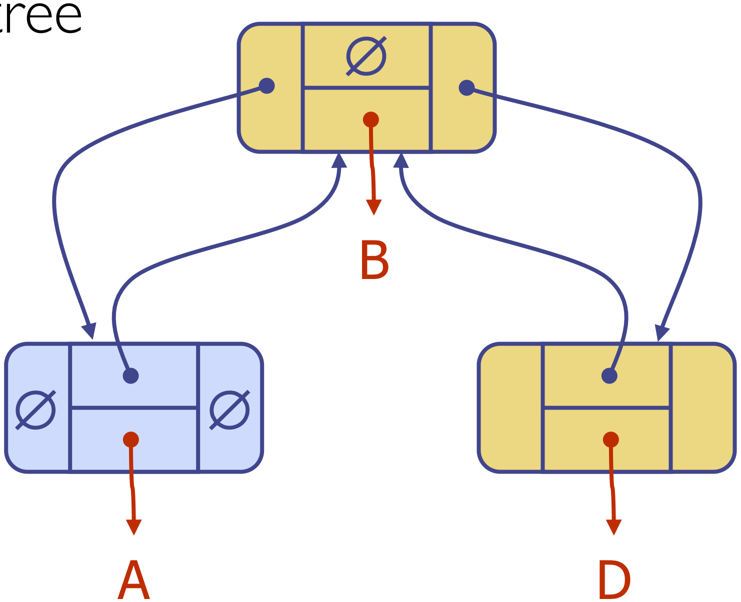
- Array-based just like the binary tree (because it is!)





- Linked-based just like the binary tree





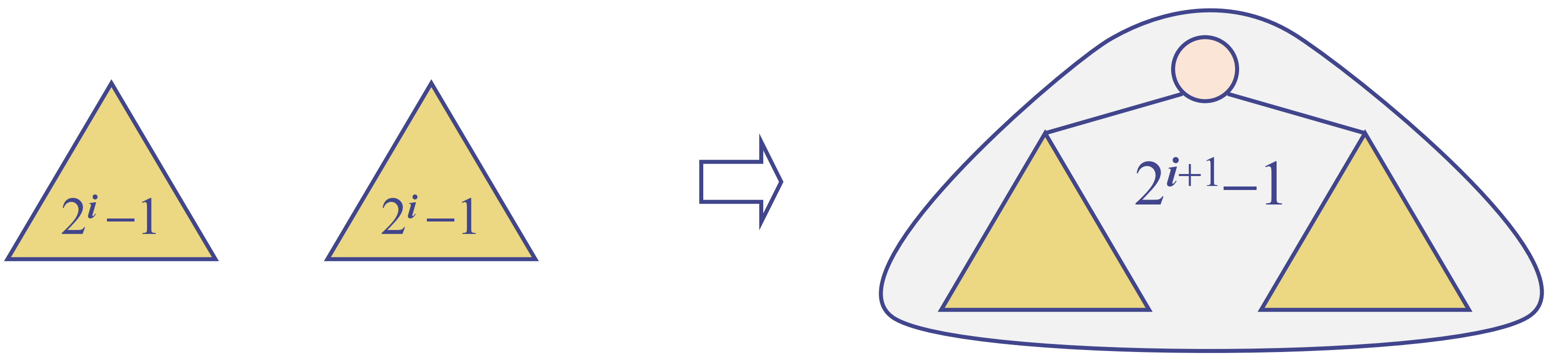
Complexity Analysis

- heap is a very efficient way to implement PQ for both insertion and removal compared to unsorted or sorted list

Operation	Unsorted List	Sorted List	Heap
len	$O(1)$	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$	$O(1)$
min	$O(n)$	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$	$O(\log n)$
remove_min	$O(n)$	$O(1)$	$O(\log n)$

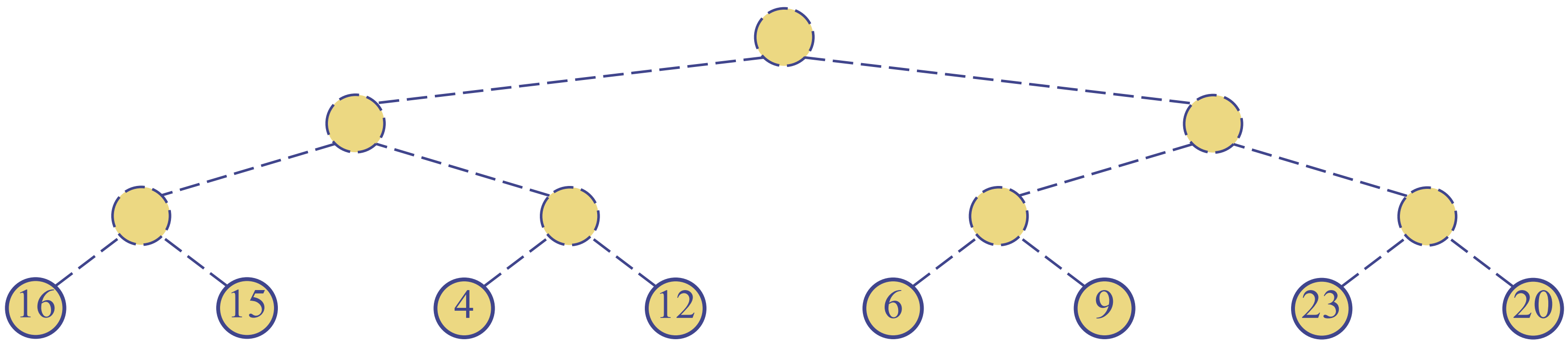
Bottom-Up Heap Construction

- Given a list of numbers, how can we construct a heap?
- 1. Simply perform add n times: $O(n \log n)$
- 2. Bottom-up heap construction
  - When a new node is added, use the node as a new "root" and merge two subtrees
  - Perform this from the bottom most level and recursively move up
  - Down-heap to preserve the heap-property



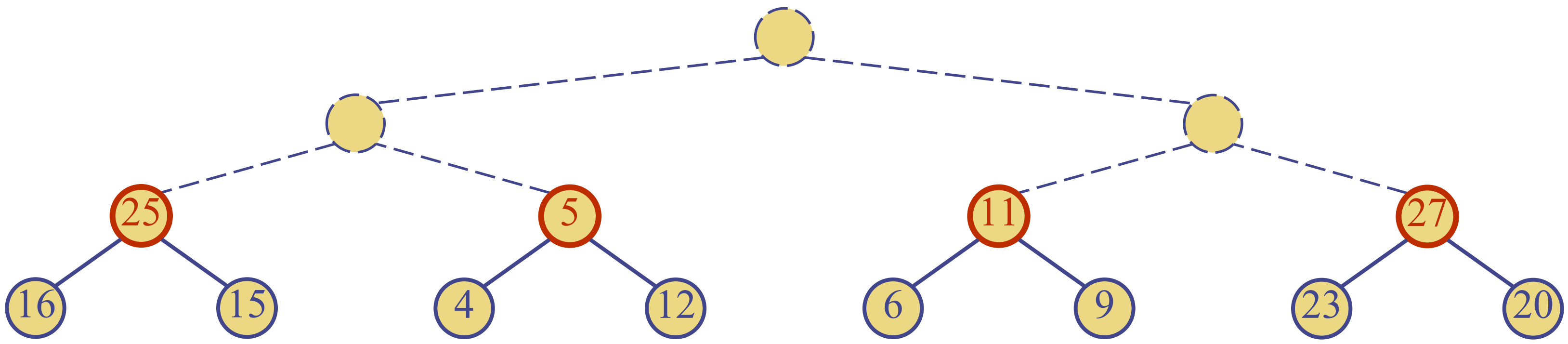
Bottom-Up Heap Construction

- The first n/2 entries of the given list become the "roots" of the bottom most layer



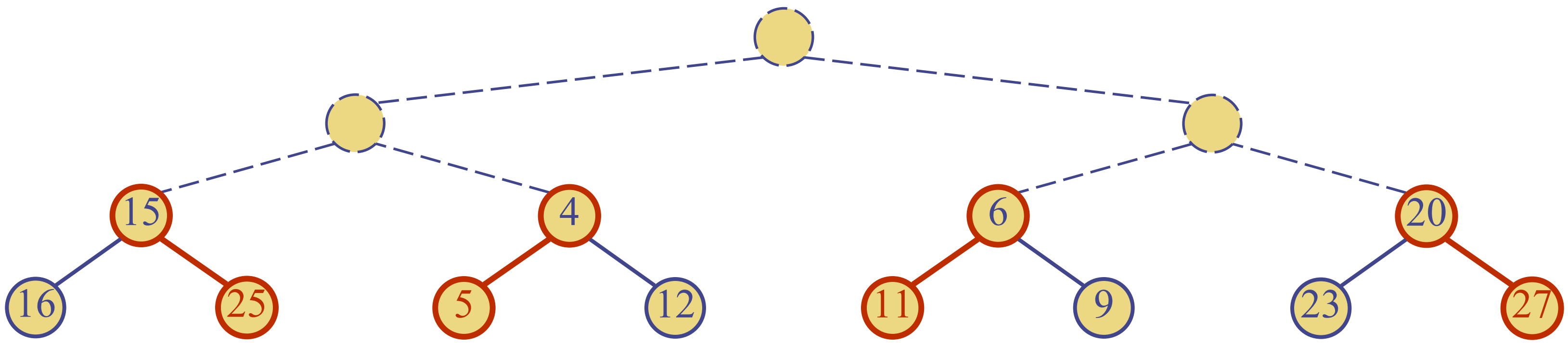
Bottom-Up Heap Construction

- The next n/4 entries of the given list become the “roots” of the subtrees (which are leaves in this case)



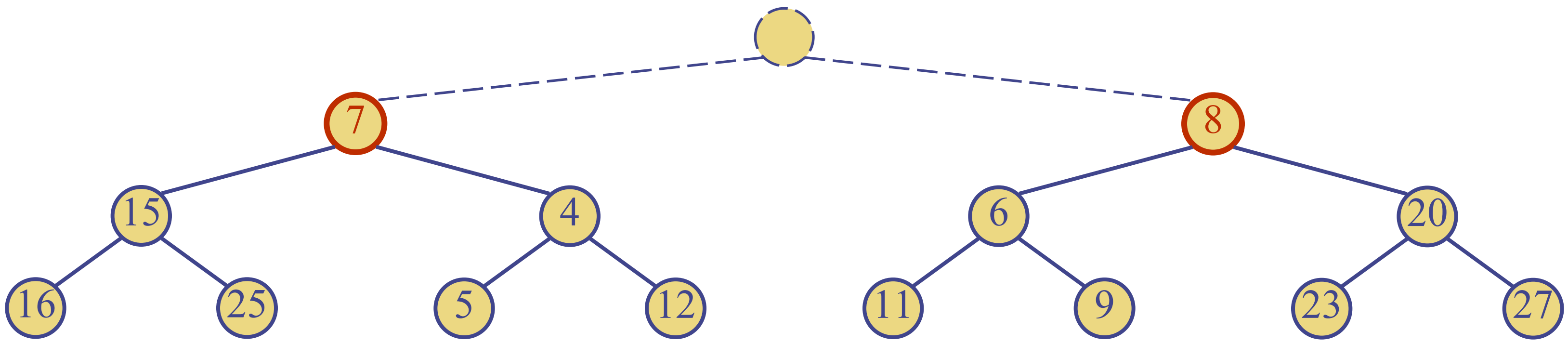
Bottom-Up Heap Construction

- Down-heap on each subtree



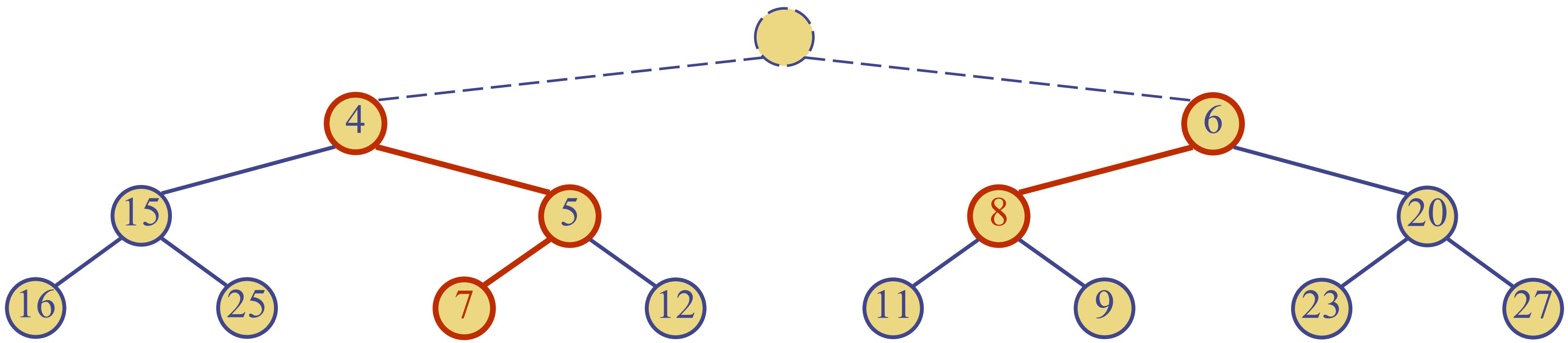
Bottom-Up Heap Construction

- The next n/8 nodes of the given list become the “roots” of the subtrees



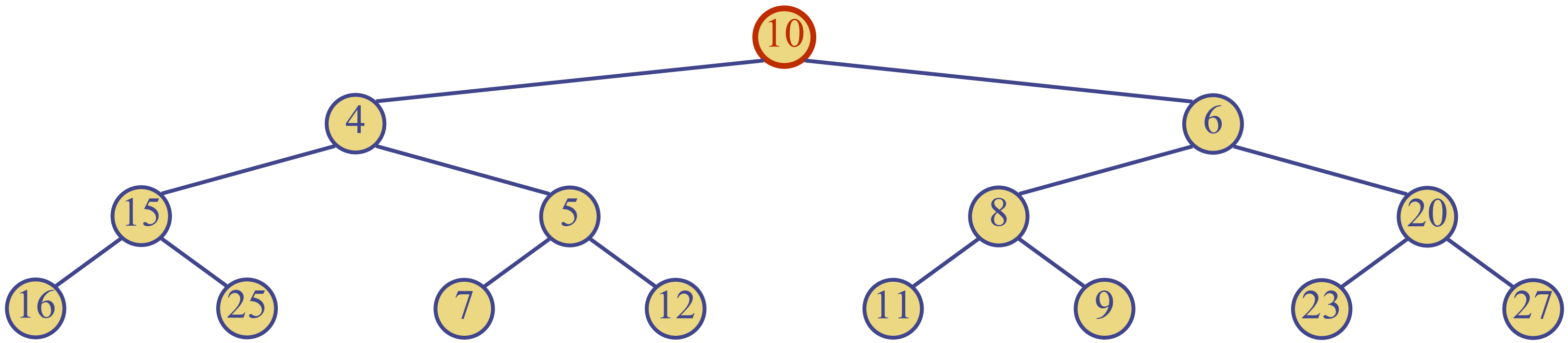
Bottom-Up Heap Construction

- Down-heap on each subtree



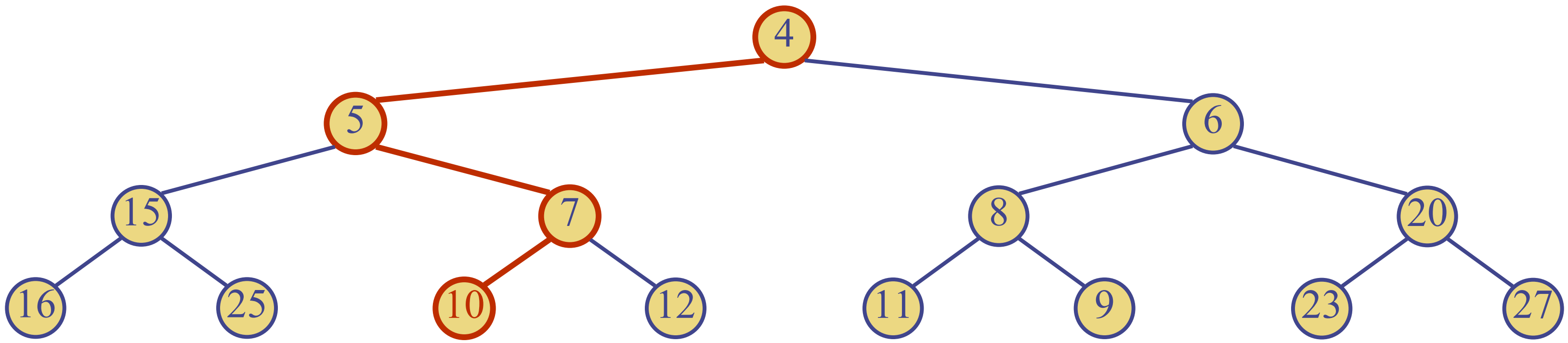
Bottom-Up Heap Construction

- The next n/16 entry of the given list becomes the "root" of the subtrees



Bottom-Up Heap Construction

- Down-heap on each subtree

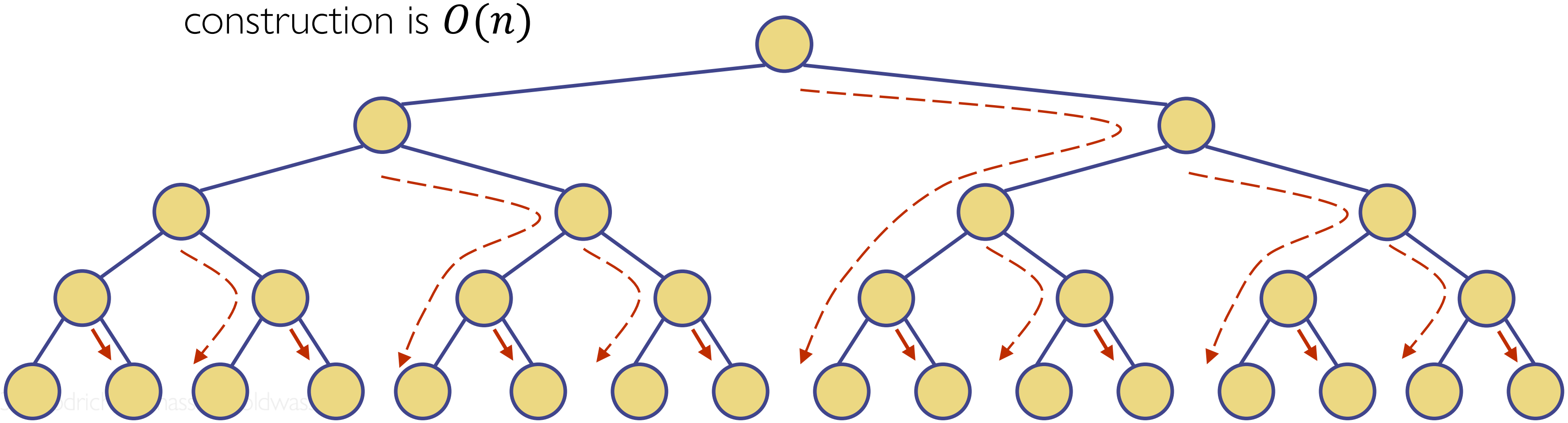


Analysis

- Seems there are lots of down-heap operations which is $O(\log n)$ each…is this really efficient?

this path is one of possible paths that reaches a leaf

- Simple worst-case analysis:
  - Suppose each new node always performs the down-heap which reaches the "inorder successor": right child -> left children until left leaf is reached (shown in red dashed lines)
  - Based on those down-heap paths, each node is involved in those paths at most two times (this because we construct from bottom-up)
  - Implies that the worst-case down-heap operations is $O(n)$
  - Other operations (merge and add) are $O(1)$, so the bottom-up heap construction is $O(n)$



Recall: Sorting with PQ

- We can use PQ straight-up to sort a list of keys
  - insert all entries and remove all entries
- But the complexity depends on the implementation of PQ

Selection-sort

PQ with an unsorted list:
search for min when removing $O(n^2)$

		Collection C	*Priority Queue P*
Input		(7,4,8,2,5,3)	()
Phase 1	(a)	(4,8,2,5,3)	(7)
	(b)	(8,2,5,3)	(7,4)
	⋮	⋮	⋮
	(f)	()	(7,4,8,2,5,3)
Phase 2	(a)	(2)	(7,4,8,5,3)
	(b)	(2,3)	(7,4,8,5)
	(c)	(2,3,4)	(7,8,5)
	(d)	(2,3,4,5)	(7,8)
	(e)	(2,3,4,5,7)	(8)
	(f)	(2,3,4,5,7,8)	()

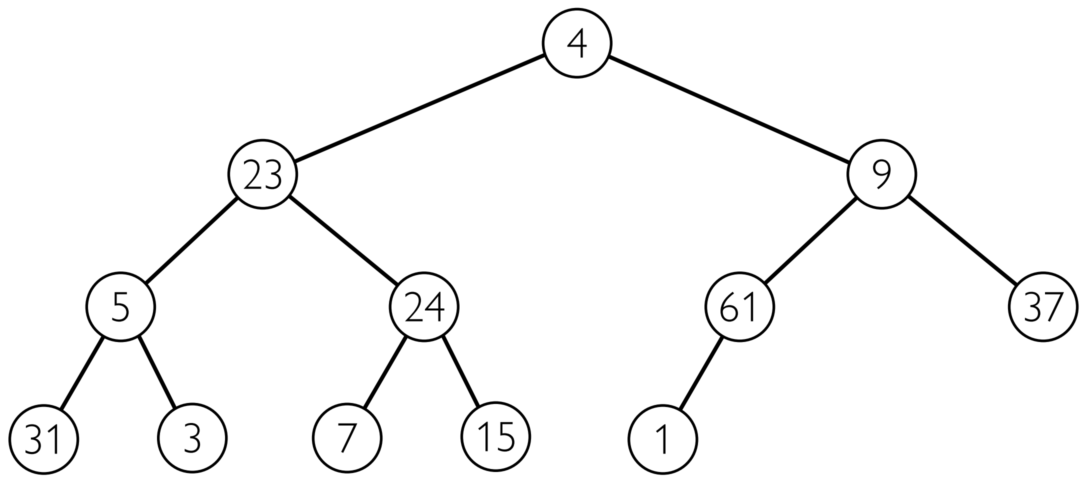
Insertion-sort

PQ with a sorted list:
search for min when inserting $O(n^2)$

		Collection C	*Priority Queue P*
Input		(7,4,8,2,5,3)	()
Phase 1	(a)	(4,8,2,5,3)	(7)
	(b)	(8,2,5,3)	(4,7)
	(c)	(2,5,3)	(4,7,8)
	(d)	(5,3)	(2,4,7,8)
	(e)	(3)	(2,4,5,7,8)
	(f)	()	(2,3,4,5,7,8)
Phase 2	(a)	(2)	(3,4,5,7,8)
	(b)	(2,3)	(4,5,7,8)
	⋮	⋮	⋮
	(f)	(2,3,4,5,7,8)	()

Heap-Sort

- PQ with Heap for sorting: Heap-Sort
  - sorting a sequence of n elements in $O(n \log n)$



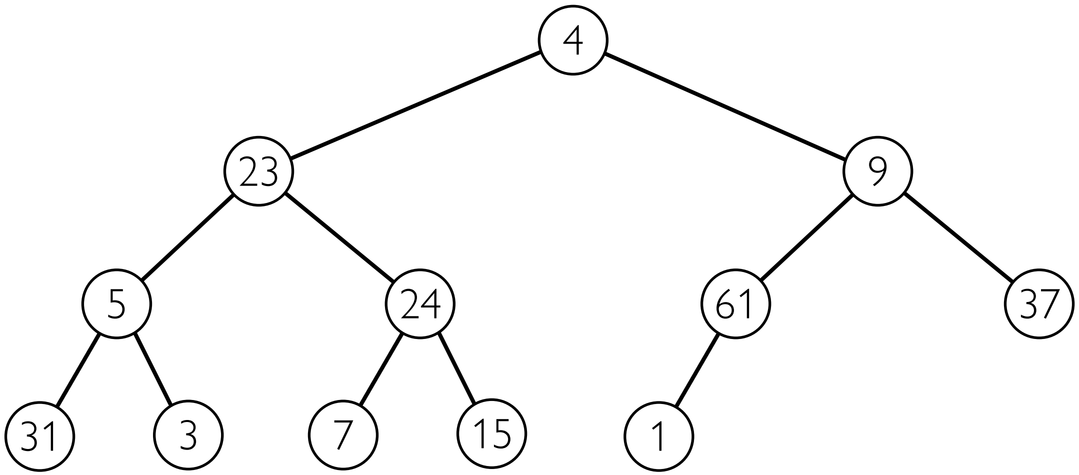
	4	23	9	5	24	61	37	31	3	7	15	1		

*currently not a heap

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

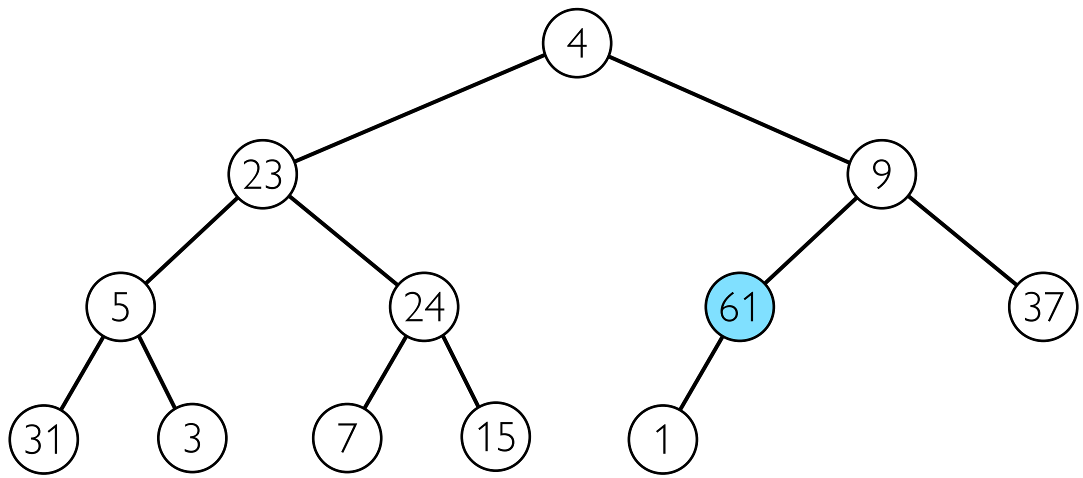
- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"



Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

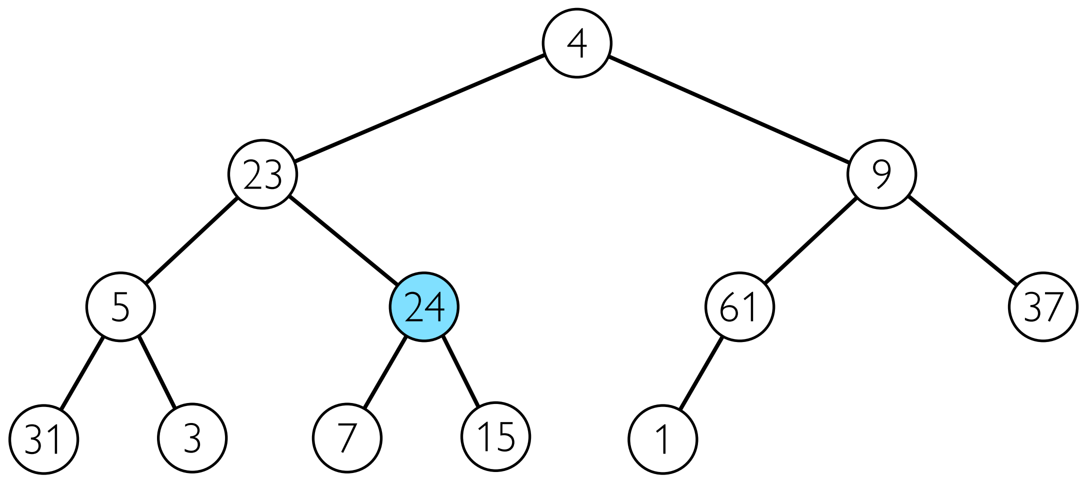


down-heap 61

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

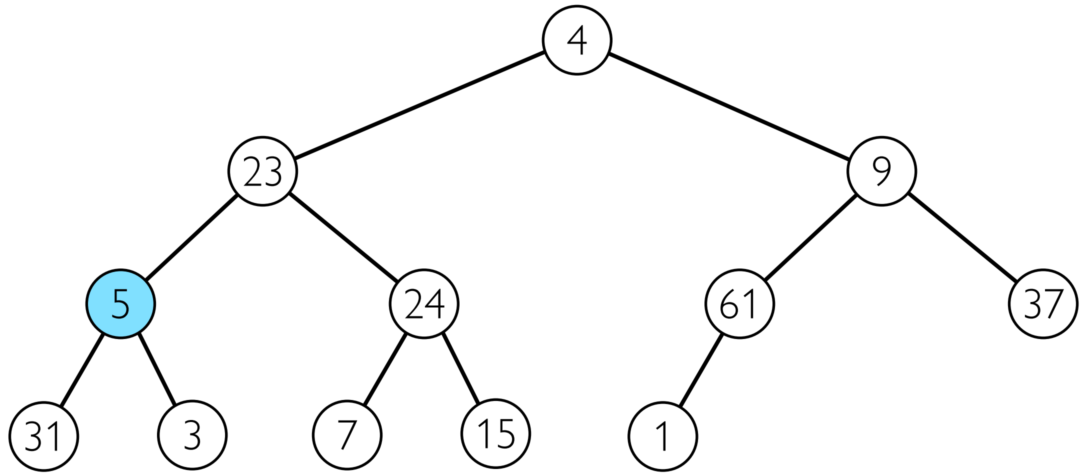


down-heap 24

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called “heapify”

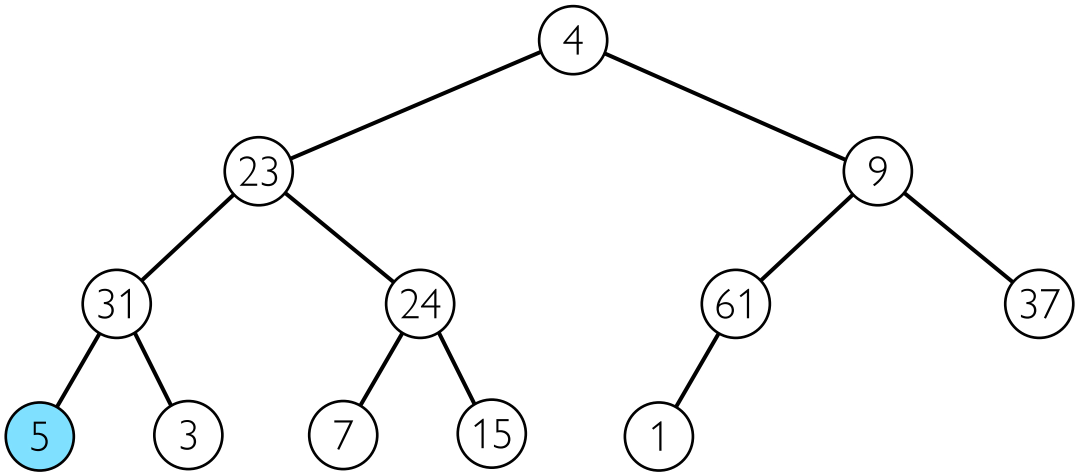


down-heap 5

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

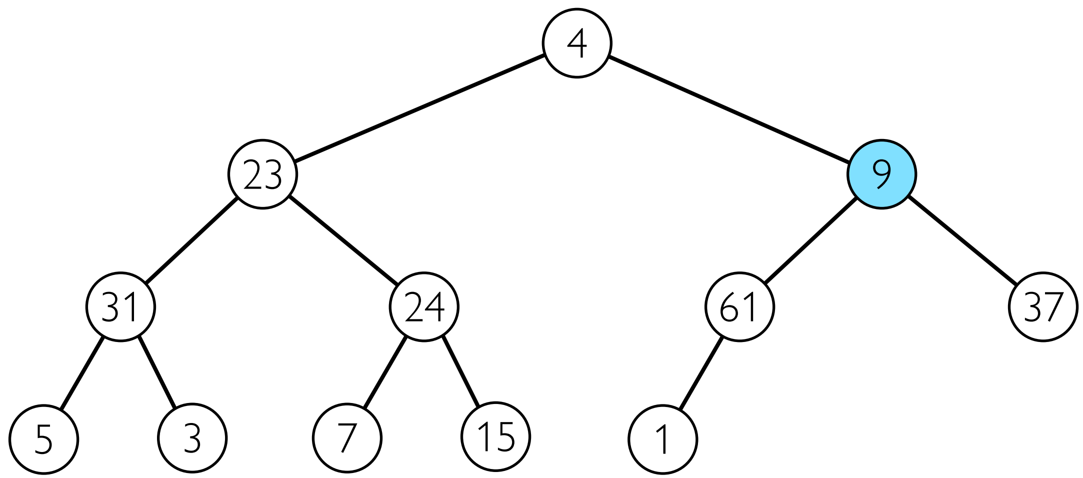


down-heap 5

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

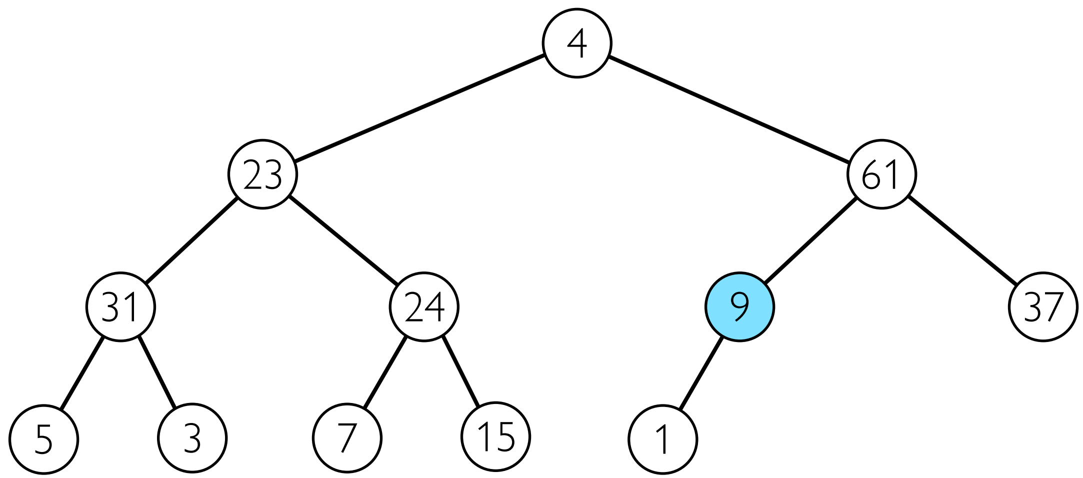


down-heap 9

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

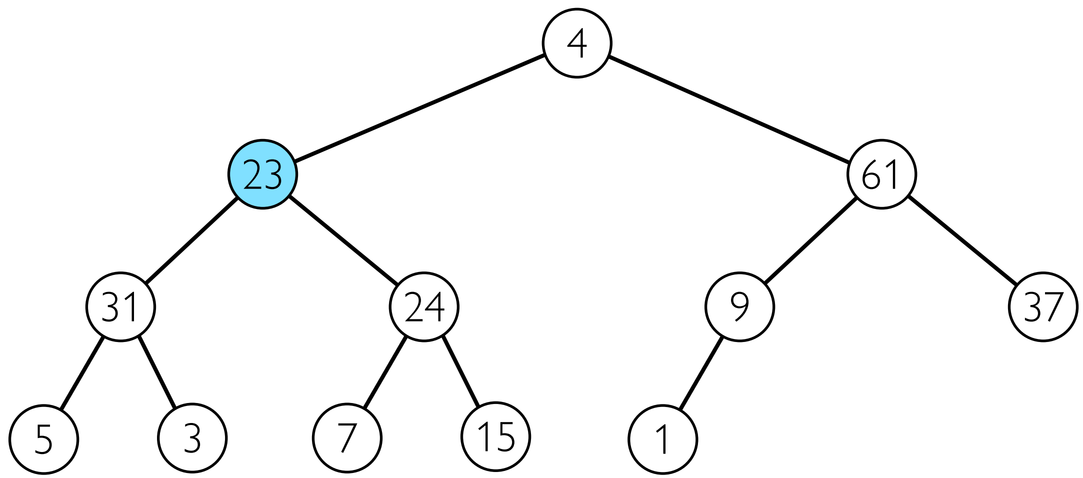


down-heap 9

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

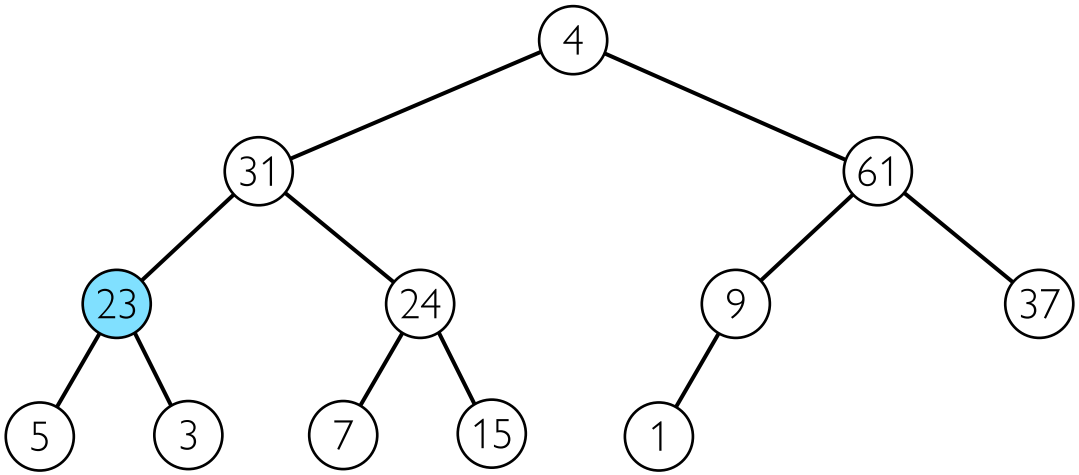


down-heap 23

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

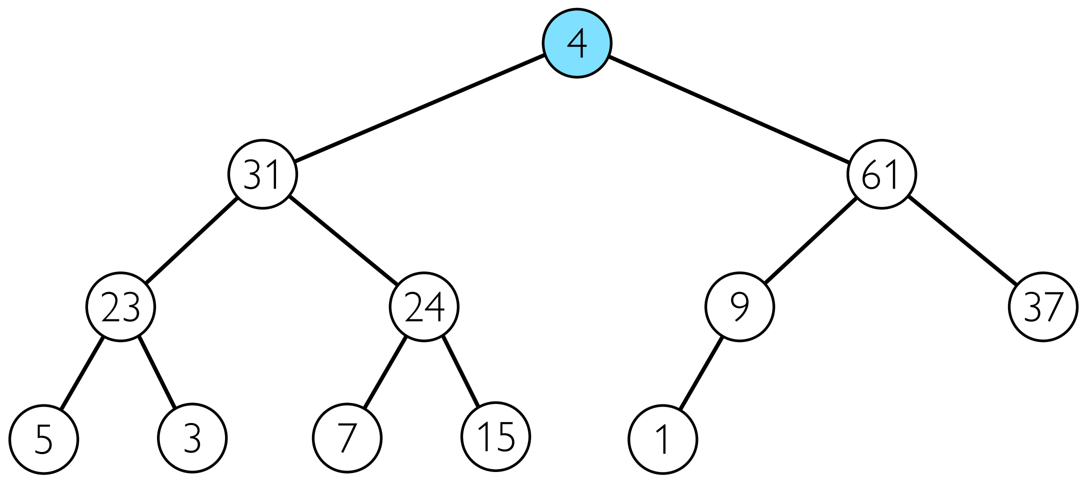


down-heap 23

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

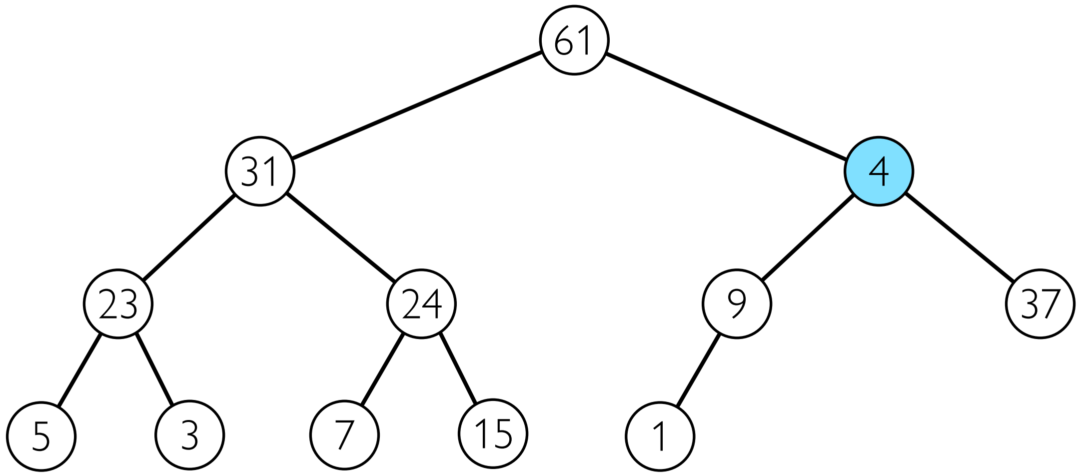


down-heap 4

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

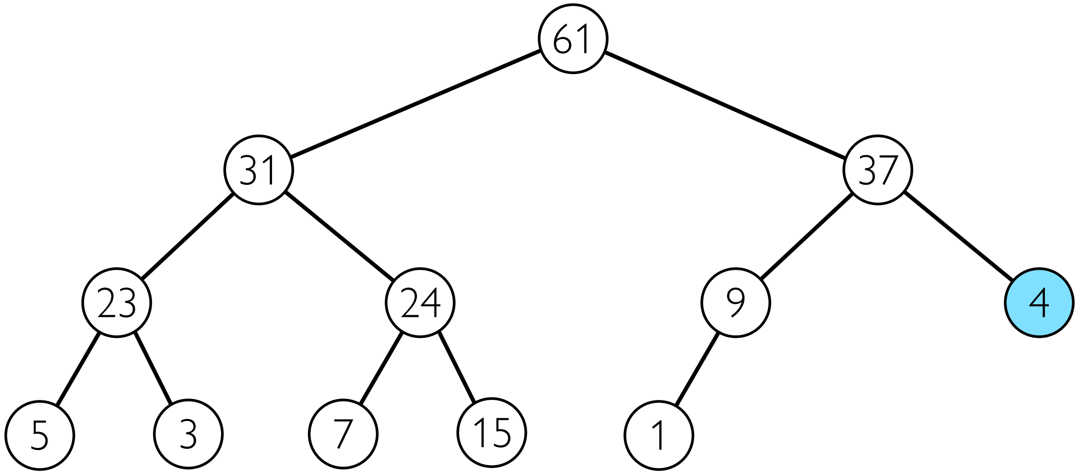


down-heap 4

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Down-heap from bottom (recall bottom-up construction)
  - Also called "heapify"

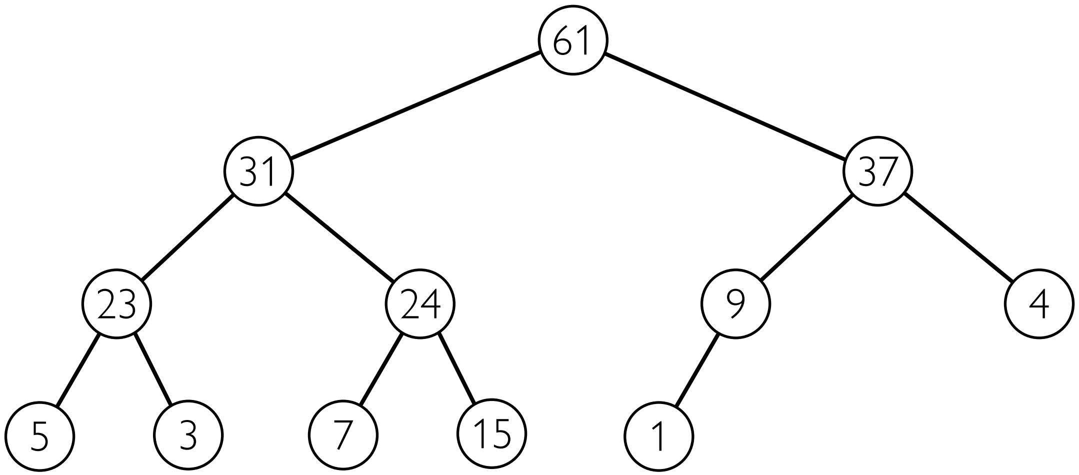


down-heap 4

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

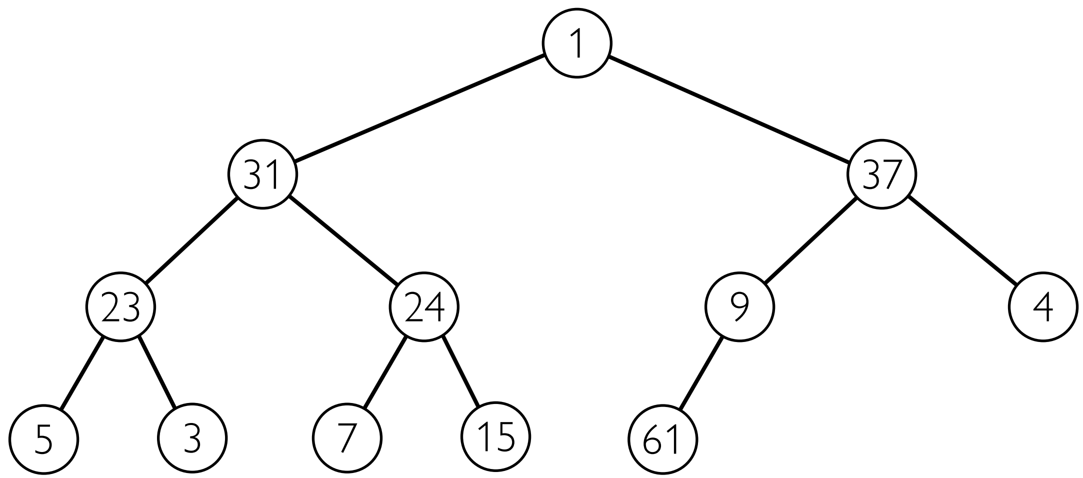


"delete" root by swapping with last node

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

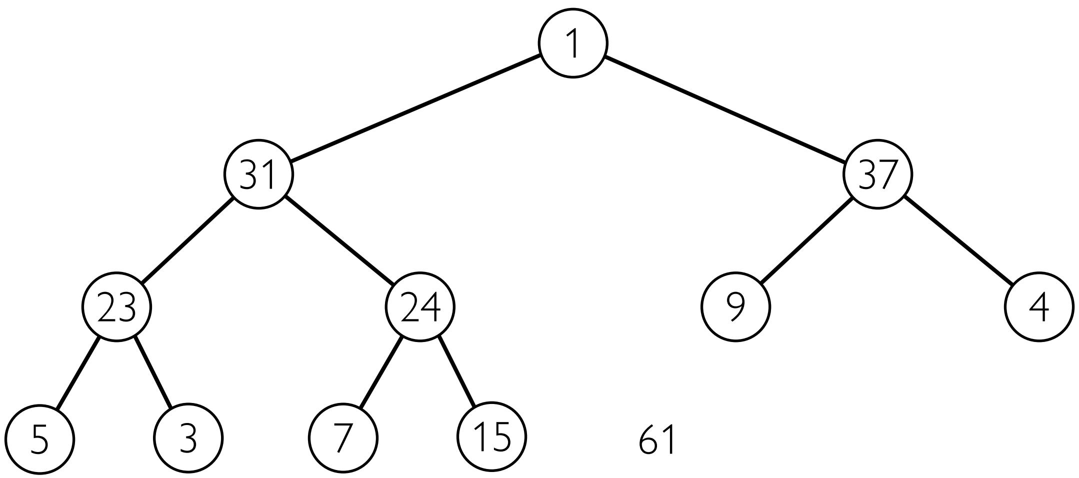


"delete" root by swapping with last node

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

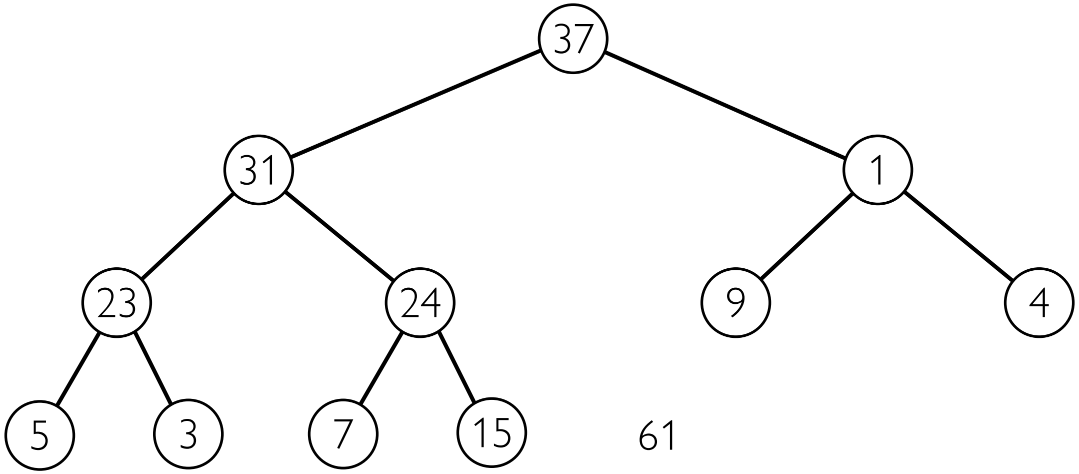


remove last node and down-heap 1

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

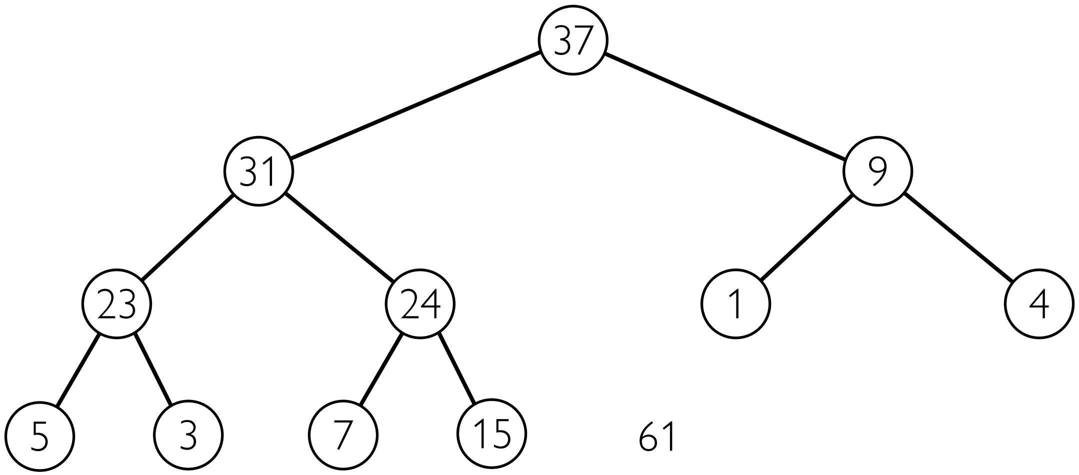


remove last node and down-heap 1

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

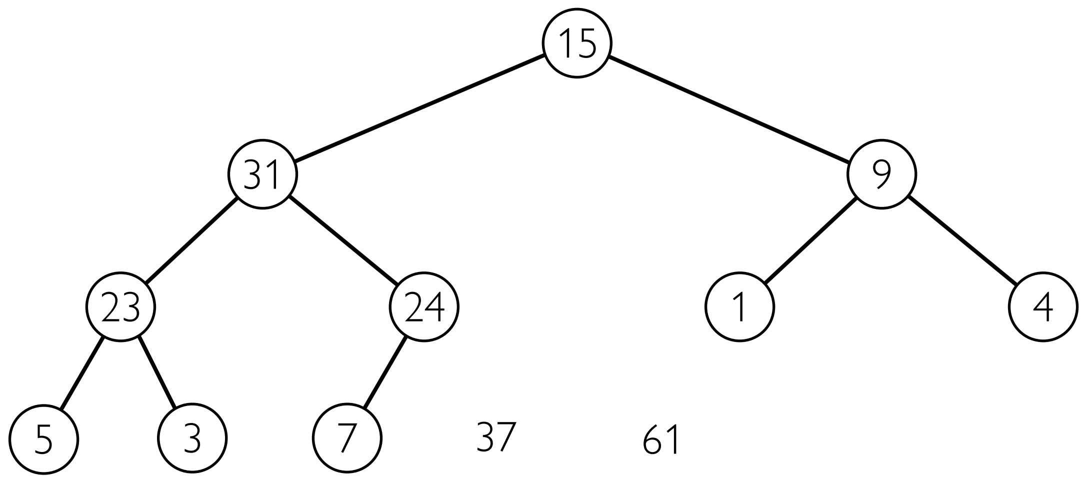


remove last node and down-heap 1

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

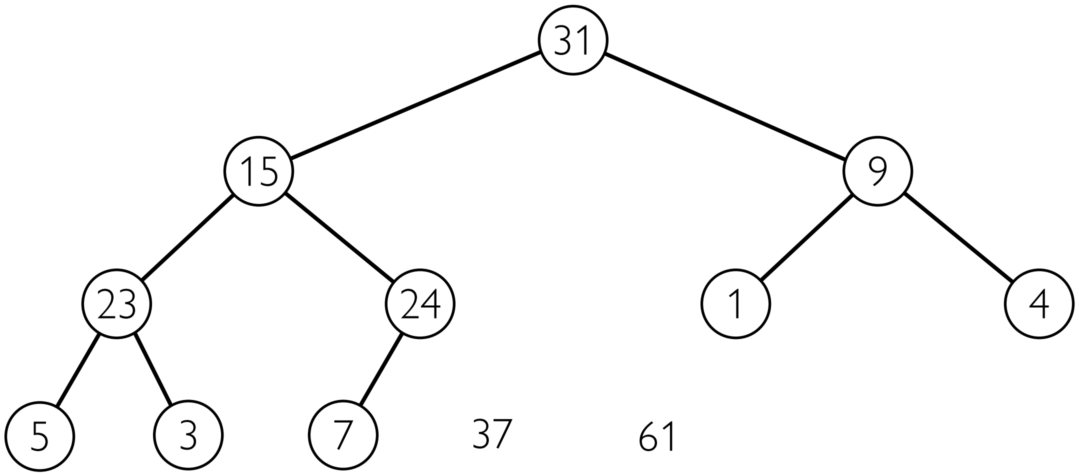


swap 37 and 15
delete last node

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

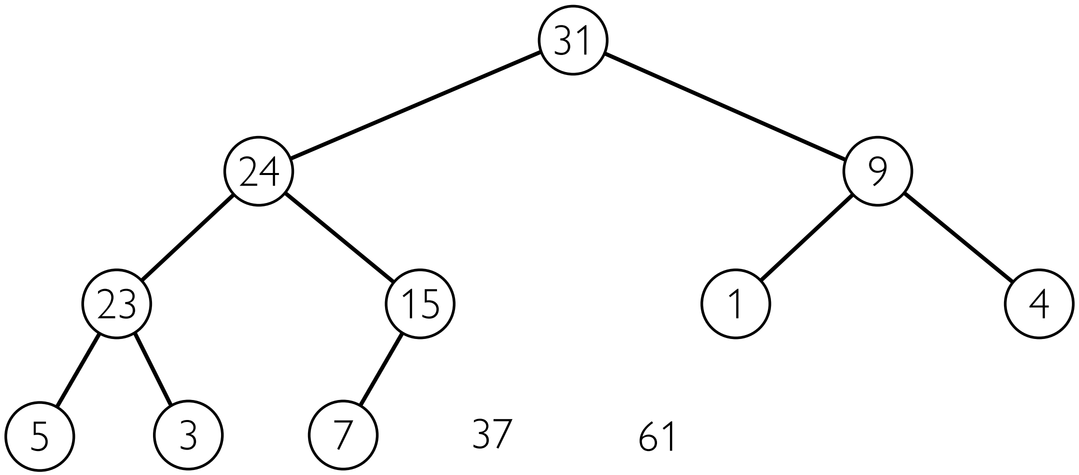
- Delete the maximum and down-heap



swap 37 and 15
remove last node
down-heap 15
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

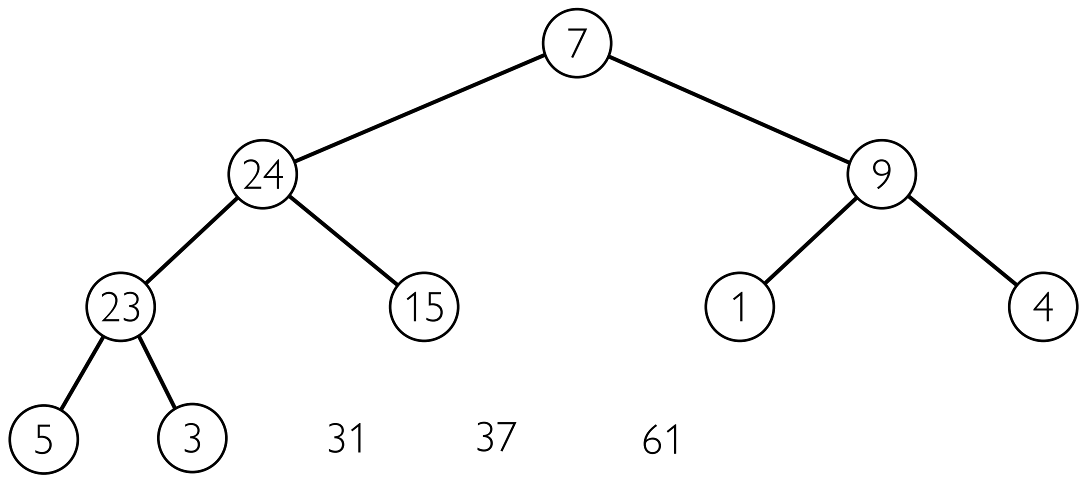
- Delete the maximum and down-heap



swap 37 and 15
remove last node
down-heap 15
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

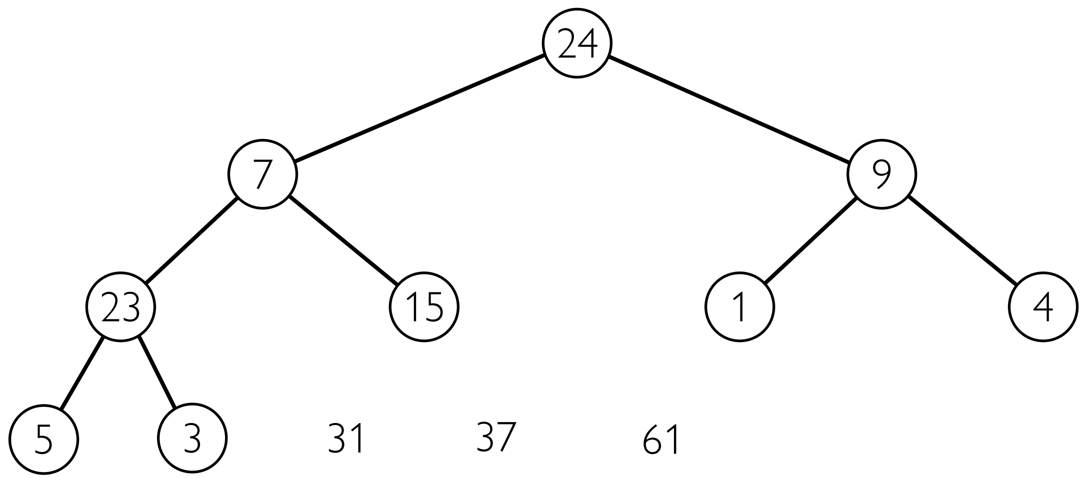
- Delete the maximum and down-heap



swap 31 and 7
remove last node
down-heap 7
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

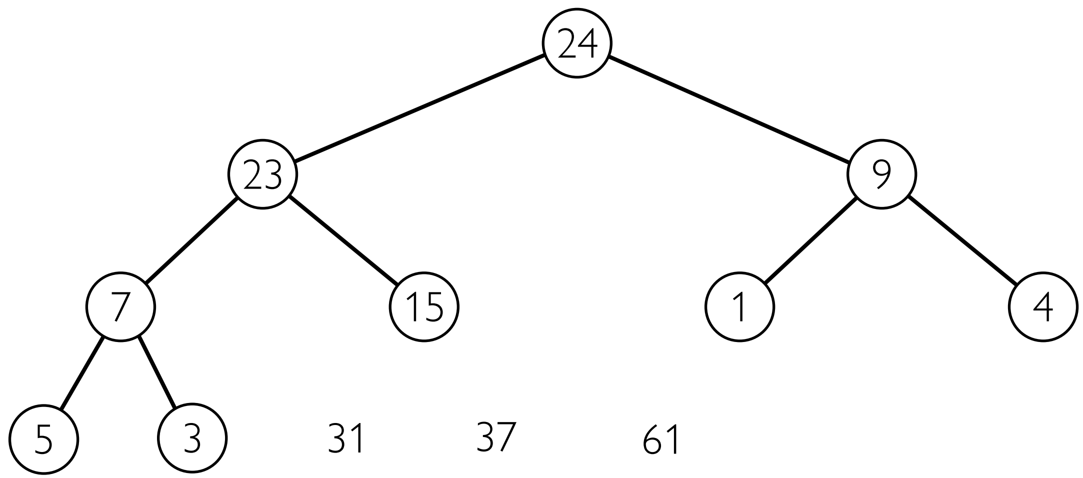
- Delete the maximum and down-heap



swap 31 and 7
remove last node
down-heap 7
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

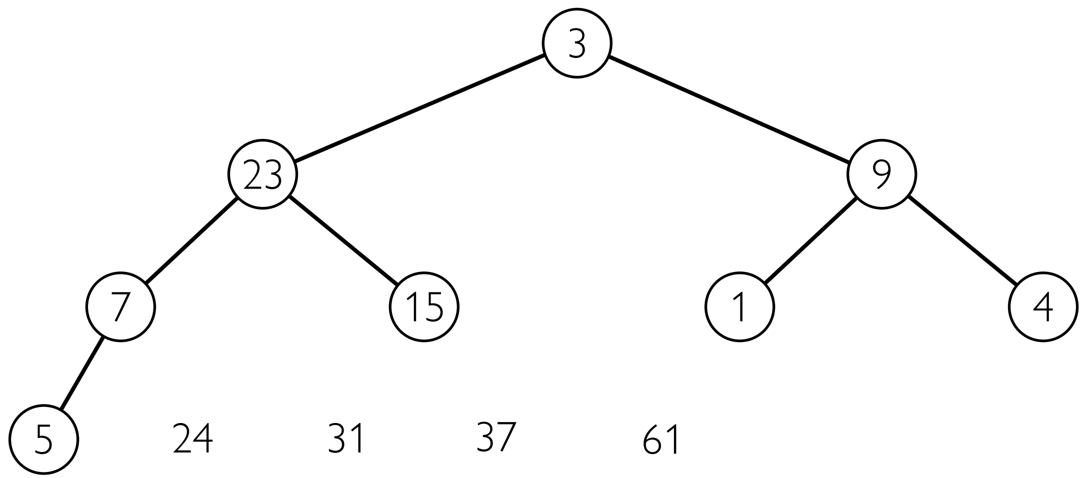
- Delete the maximum and down-heap



swap 31 and 7
remove last node
down-heap 7
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

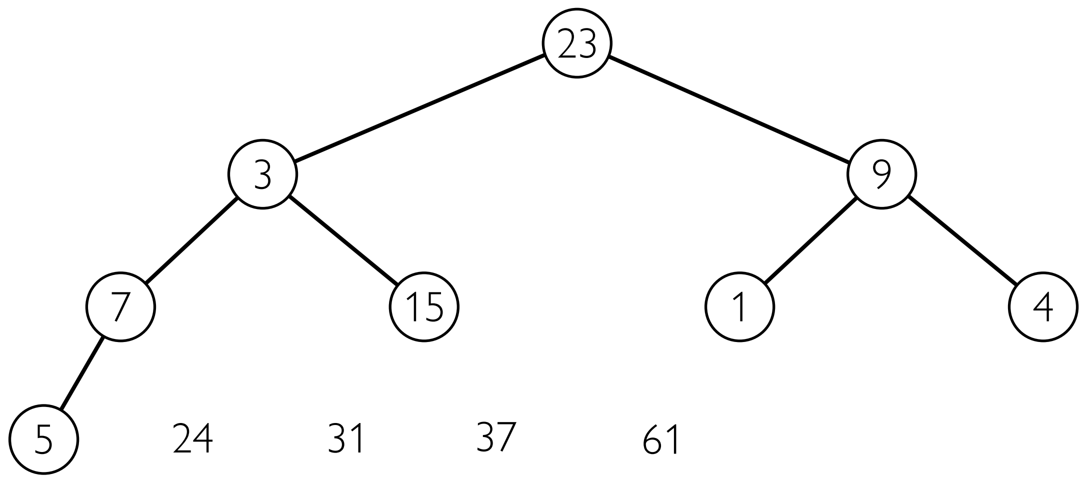
- Delete the maximum and down-heap



swap 24 and 3
remove last node
down-heap 3
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

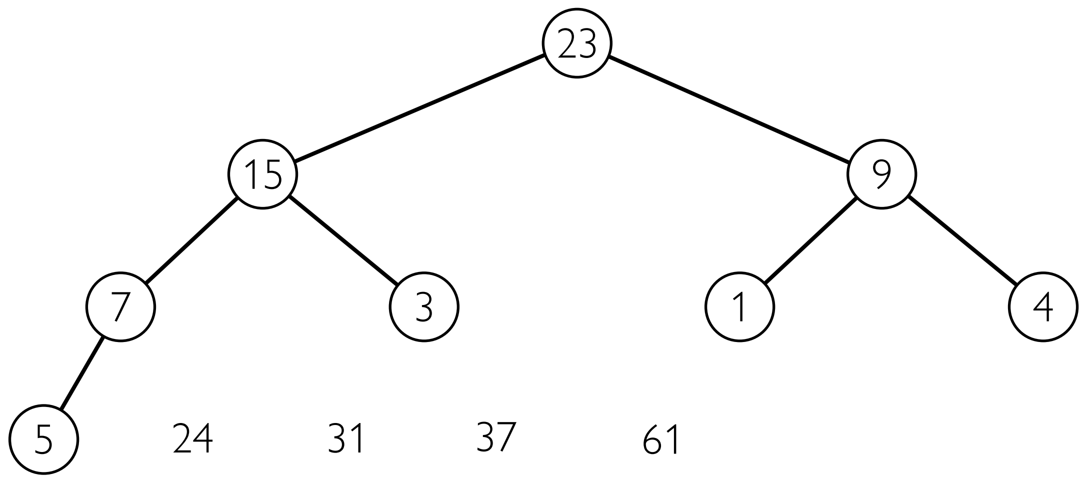
- Delete the maximum and down-heap



swap 24 and 3
remove last node
down-heap 3
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

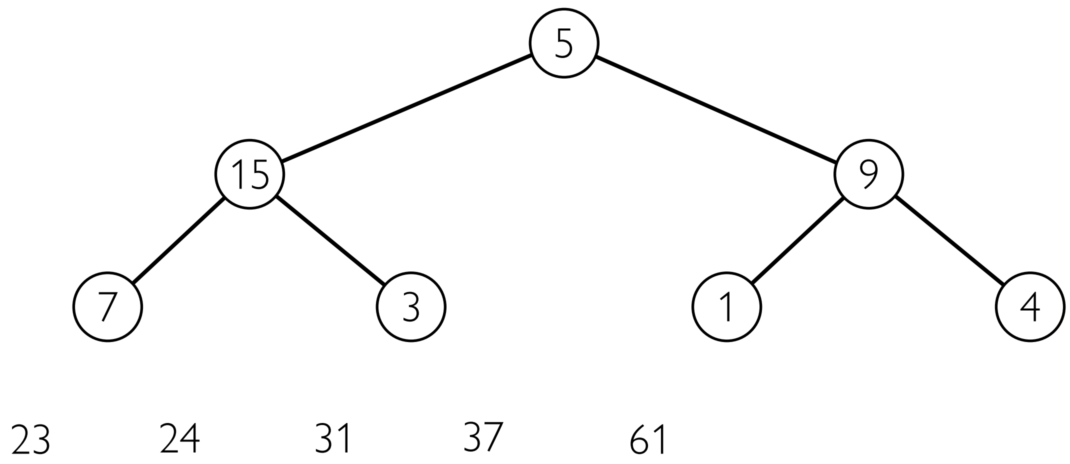
- Delete the maximum and down-heap



swap 24 and 3
remove last node
down-heap 3
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

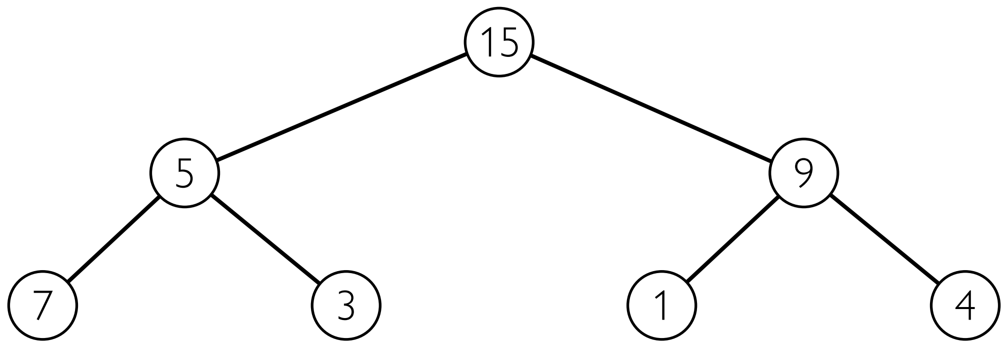


swap 23 and 5
remove last node
down-heap 5

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap



23 24 31 37 61

swap 23 and 5

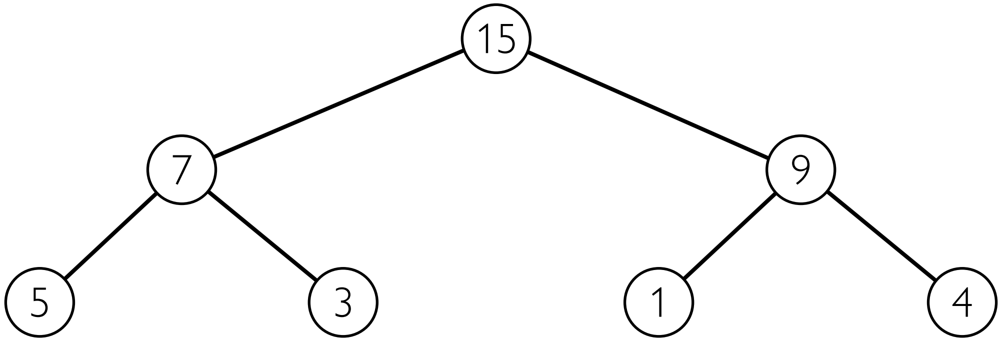
remove last node

down-heap 5

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

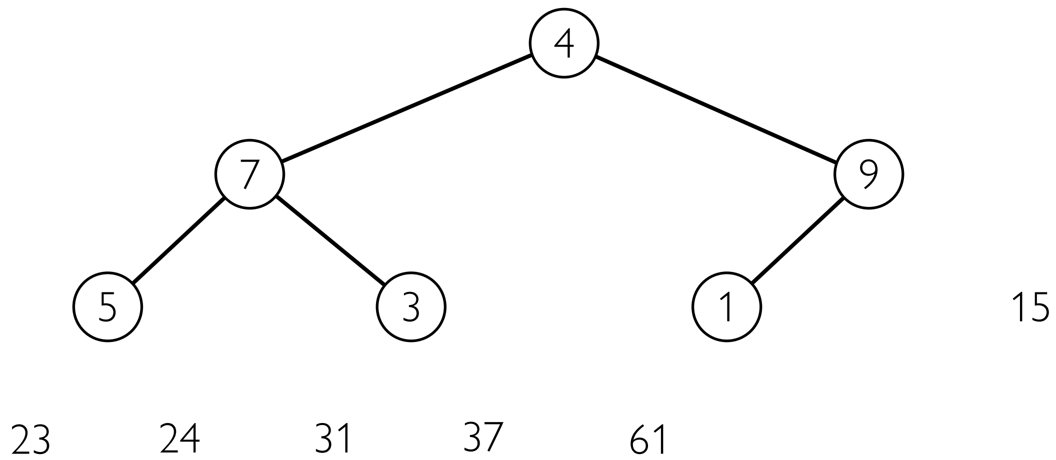


23 24 31 37 61

swap 23 and 5
remove last node
down-heap 5
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

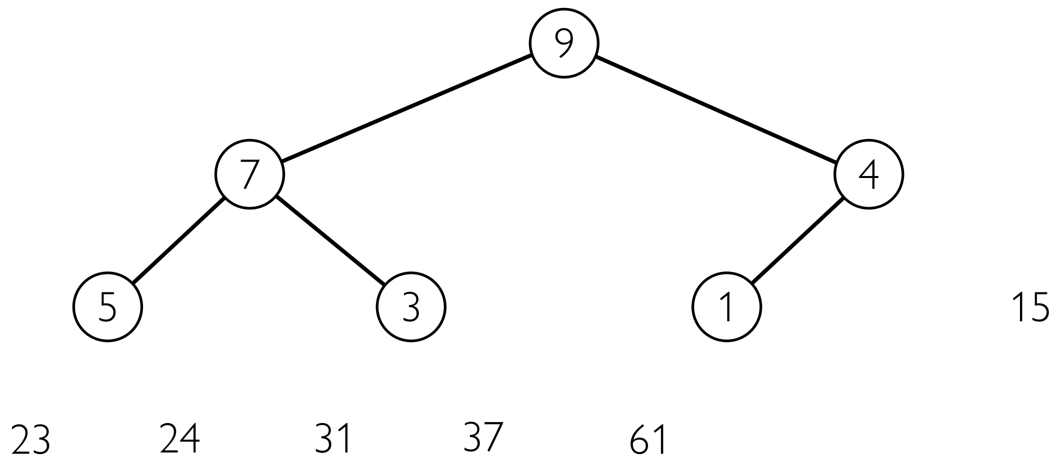
- Delete the maximum and down-heap



swap 15 and 4
remove last node
down-heap 4
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

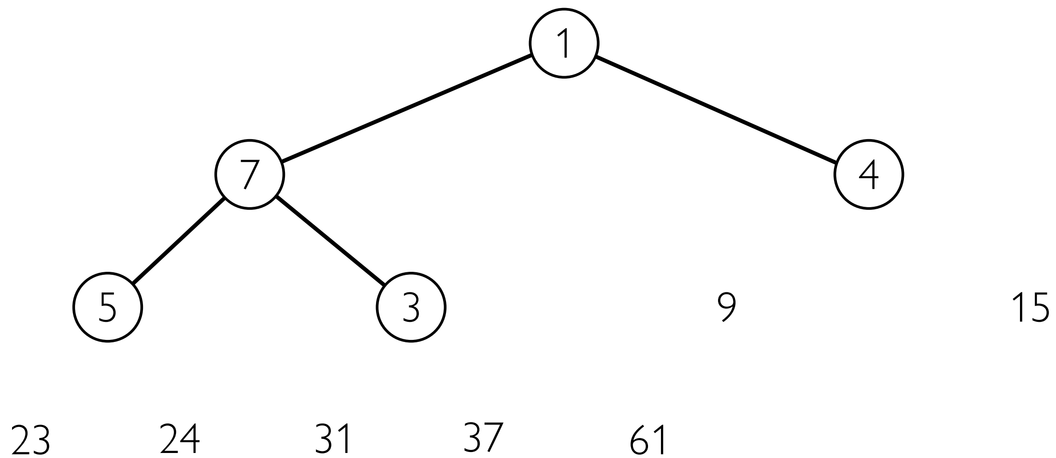
- Delete the maximum and down-heap



swap 15 and 4
remove last node
down-heap 4
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

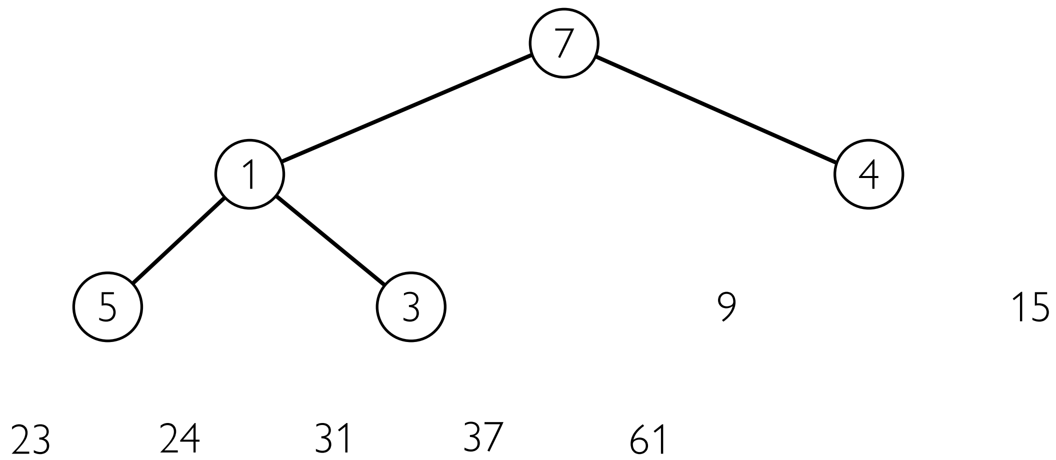
- Delete the maximum and down-heap



swap 9 and 1
remove last node
down-heap 1
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap



swap 9 and 1

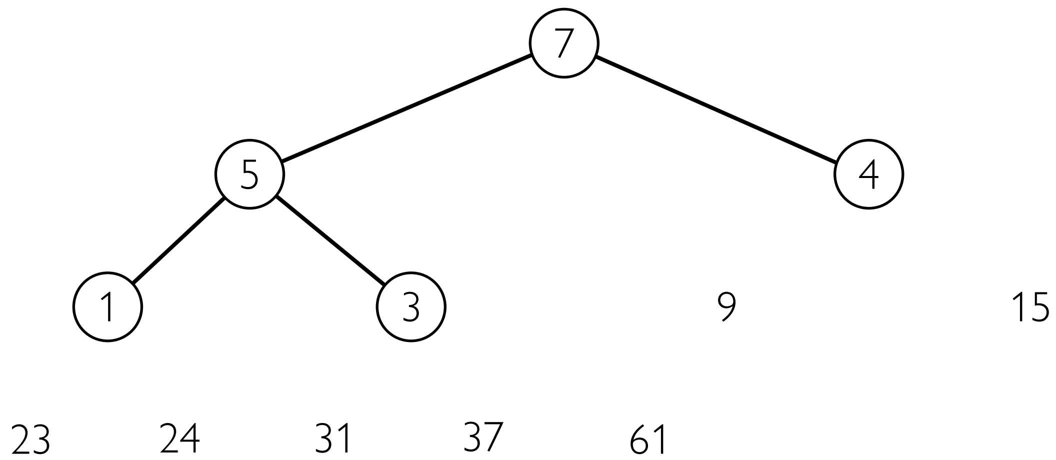
remove last node

down-heap 1

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

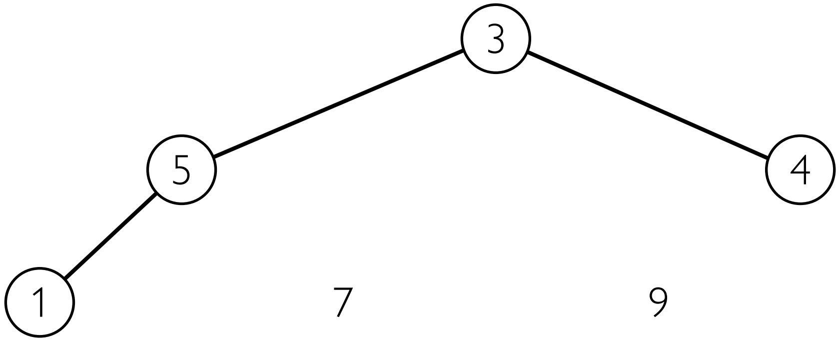
- Delete the maximum and down-heap



swap 9 and 1
remove last node
down-heap 1
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap



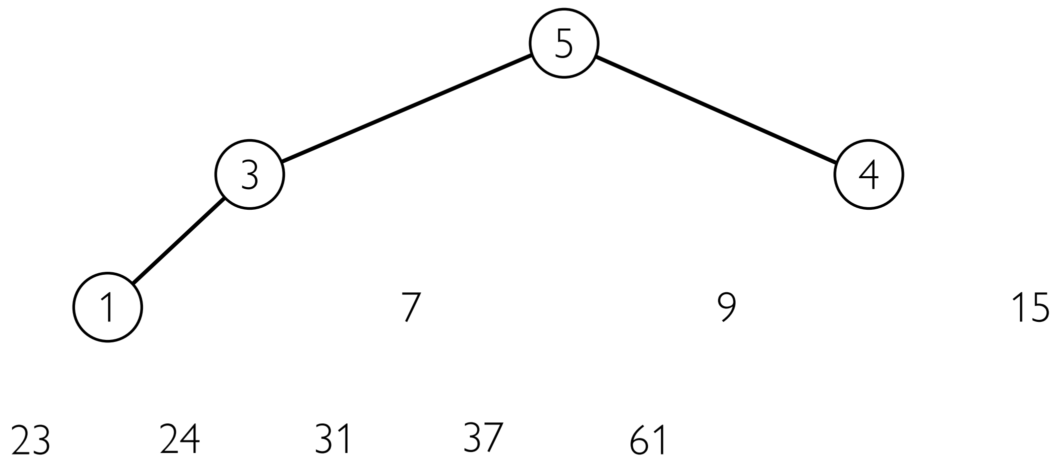
7 9 15

23 24 31 37 61

swap 7 and 3
remove last node
down-heap 3
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap



swap 7 and 3

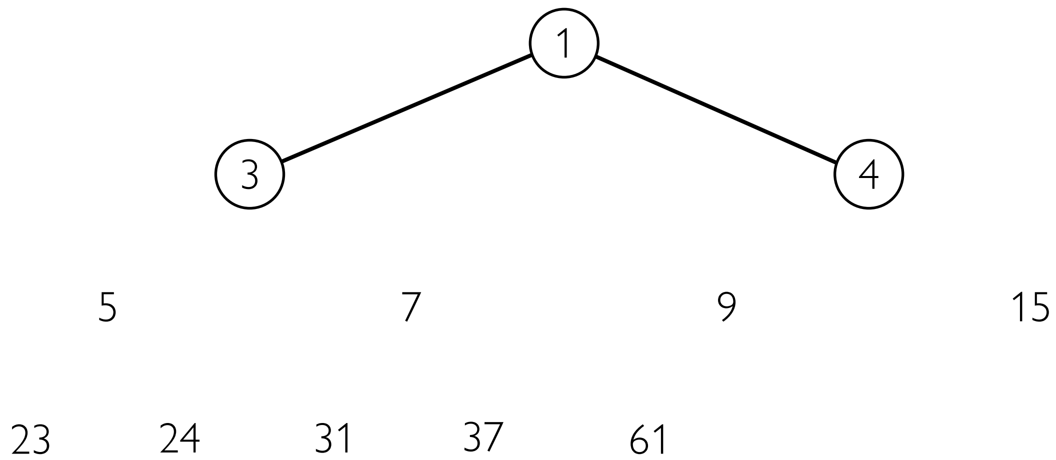
remove last node

down-heap 3

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

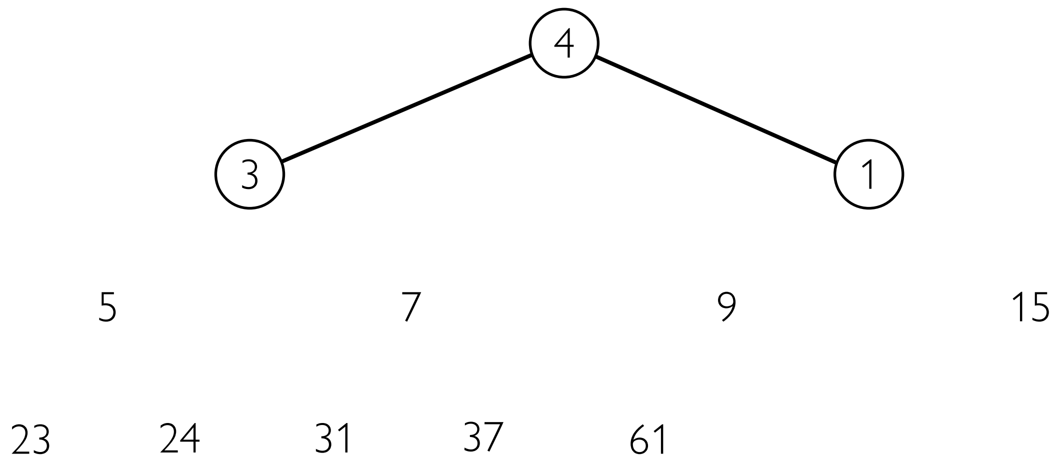
- Delete the maximum and down-heap



swap 5 and 1
remove last node
down-heap 1
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

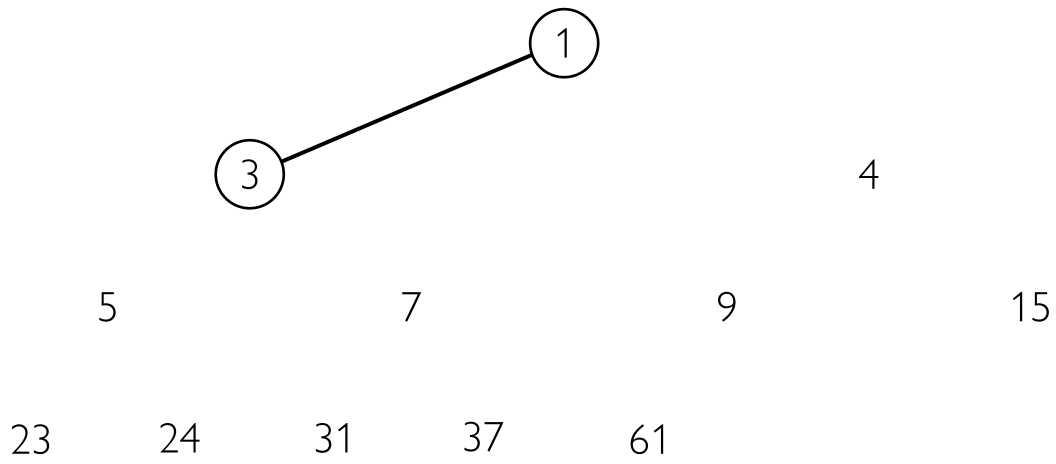
- Delete the maximum and down-heap



swap 5 and 1
remove last node
down-heap 1
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

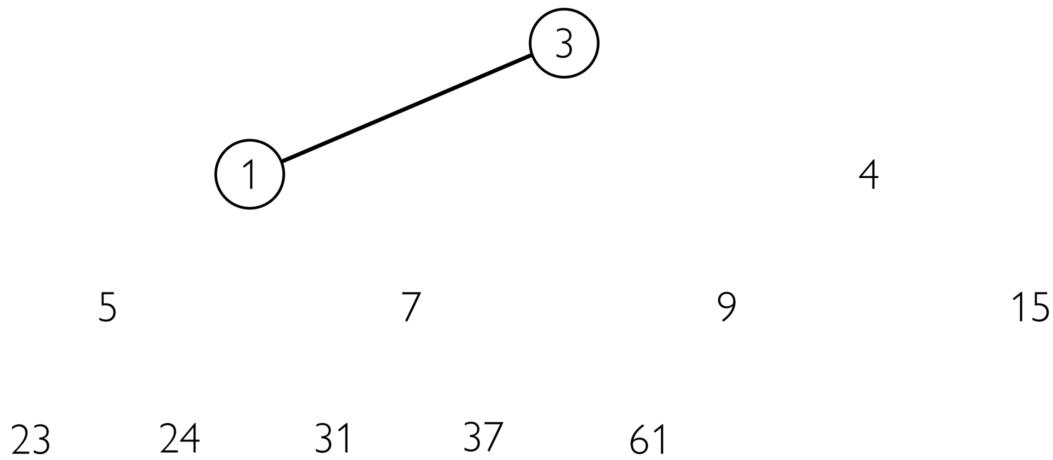
- Delete the maximum and down-heap



swap 4 and 1
remove last node
down-heap 1
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap



swap 4 and 1
remove last node
down-heap 1
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

1

3 4

5 7 9 15

23 24 31 37 61

swap 3 and 1
remove last node
down-heap 1
Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Delete the maximum and down-heap

1

3 4

5 7 9 15

23 24 31 37 61

remove 1

Max-heap in this example, so parent key ≥ children keys

Heap-Sort

- Much faster than selection-sort and insertion-sort

	1	3	4	5	7	9	15	23	24	31	37	61		

Max-heap in this example, so parent key ≥ children keys

Summary

- Priority Queue
  - More practical than the queue
  - Also for sorting a sequence of elements
- Heap
  - Special binary tree
  - Very useful
  - Seemingly complicated operations, but very worth it
  - Fast construction
- PQ with Heap: Heap-Sort
  - Faster than other PQ-based sorting methods (selection-sort and insertion-sort)
- Next: An even more generic data structure to store (key, value) pairs