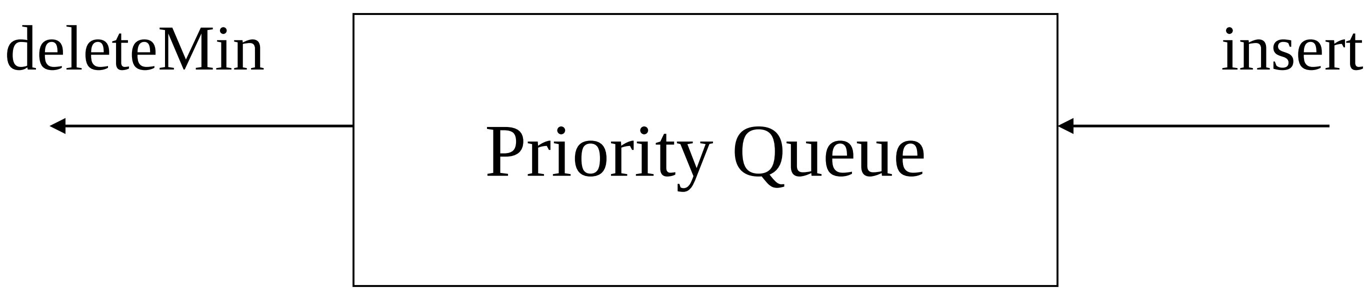
Lecture 16
Priority Queues
(Heaps)

Priority Queues

- Many applications require that we process records with keys in order, but not necessarily in full sorted order.
- Often we collect a set of items and process the one with the current minimum value.
  - e.g. jobs sent to a printer,
  - Operating system job scheduler in a multi-user environment.
  - Simulation environments
- An appropriate data structure is called a *priority queue*.

Definition

- A priority queue is a data structure that supports two basic operations: insert a new item and remove the minimum item.



Simple Implementations

- A simple linked list:
  - Insertion at the front (O(1)); find minimum (O(N)), or
  - Keep list sorted; insertion O(N), findMin O(1)
- A binary search tree:
  - This gives an O(log N) average for both operations.
  - But BST class supports a lot of operations that are not required.
  - Self-balancing BSTs O(log N) worst for both operations.
- An array: Binary Heap
  - Does not require links and will support both operations in O(logN) wost-case time. findMin in O(1) at worst.

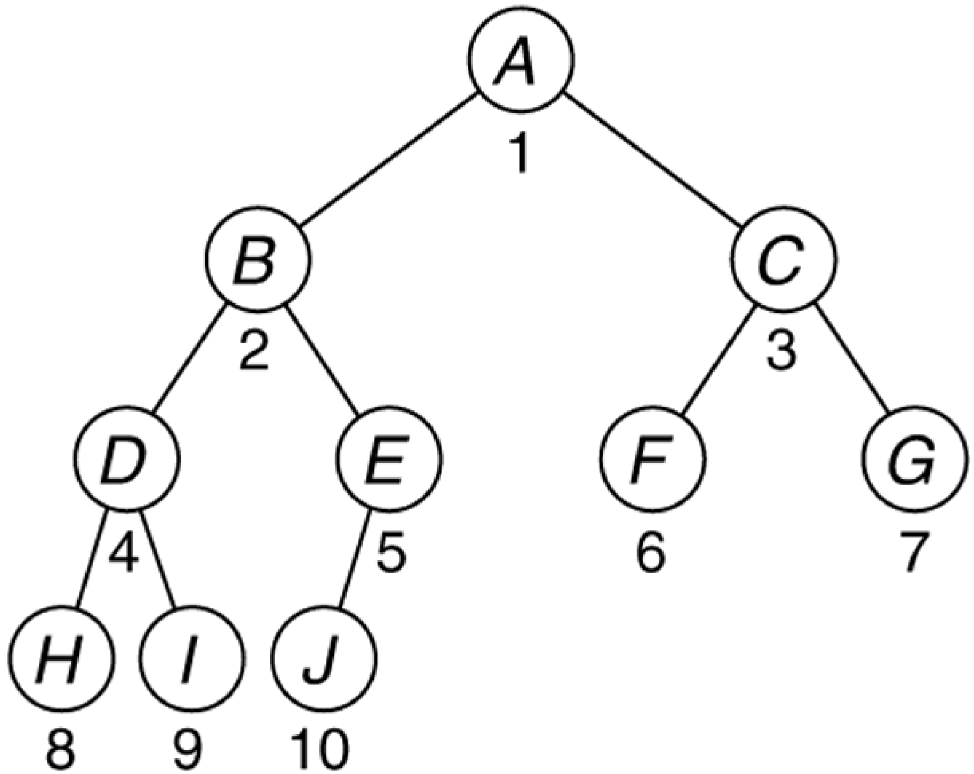
Binary Heap

- The binary heap is the classic method used to implement priority queues.
- We use the term **heap** to refer to the binary heap.
- Heap is different from the term heap used in dynamic memory allocation.
- Heap has two properties:
  - Structure property
  - Ordering property

Structure Property

- A **heap** with N nodes is a *complete binary tree*, represented as an array with indices 1 to N
- A **complete binary tree** is a tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.
- Index of root is 1.
- Parent, left and right children of a node with index i are $\lfloor i/2 \rfloor$. 2i and 2i + 1respectively.

Heap as a complete binary tree

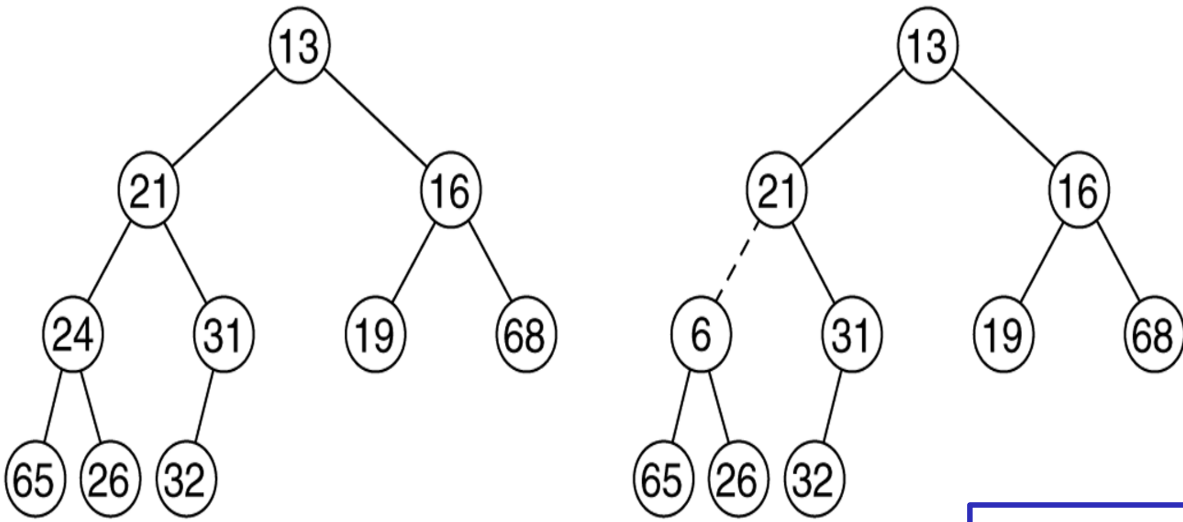


	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap-Order Property

- Min-heap:
- At every node X with parent P, the key at P is smaller than or equal to the key at X.
- Thus the minimum element is always at the root.
  - Thus minimum element can be found in O(1) time.
- Max-heap:
- The data (or key) at every node is larger than the data at both its children
- A **max heap** supports access of the maximum element instead of the minimum.

Min-heap and Max-heap

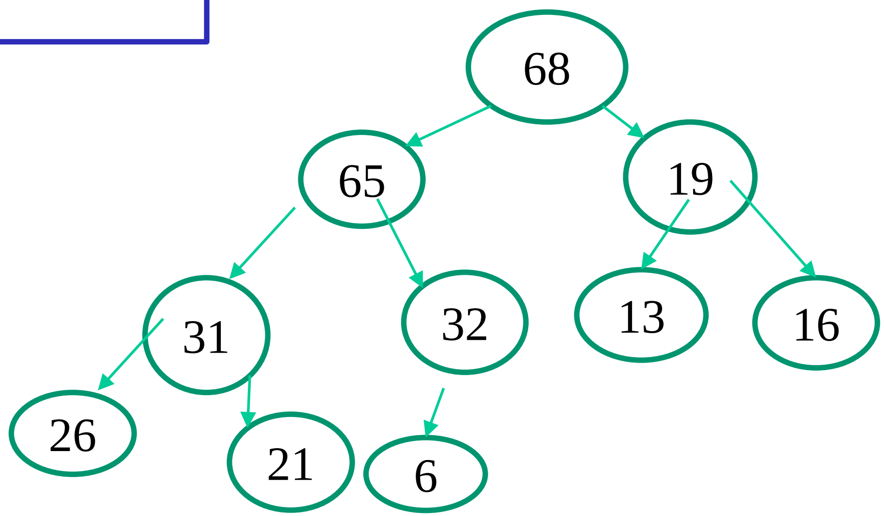


Min-heap

(a)

(b)

Not a heap

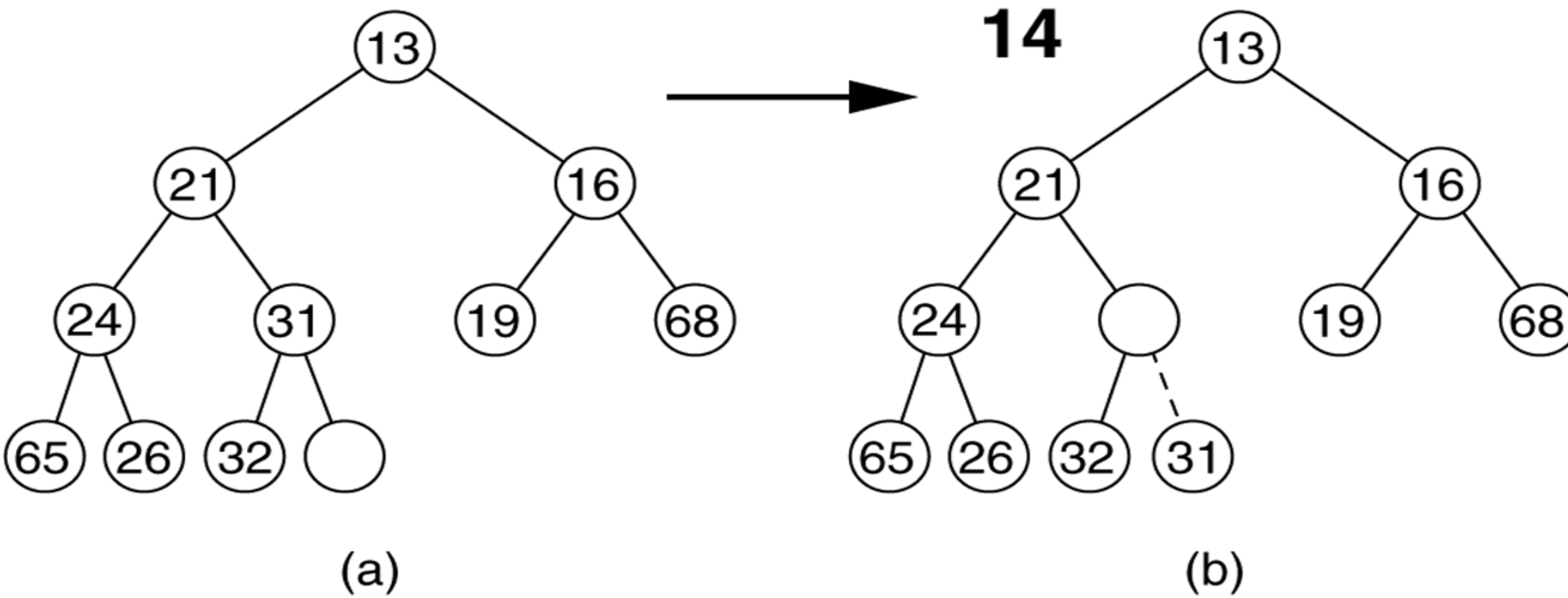


Max-heap

Basic Heap Operations: Insert

- To insert an element X into the heap:
  - We create a hole in the next available location.
  - If X can be placed there without violating the heap property, then we do so and are done.
  - Otherwise
    - we bubble up the hole toward the root by sliding the element in the hole's parent down.
    - We continue this until X can be placed in the hole.
- This general strategy is known as a *percolate up*.

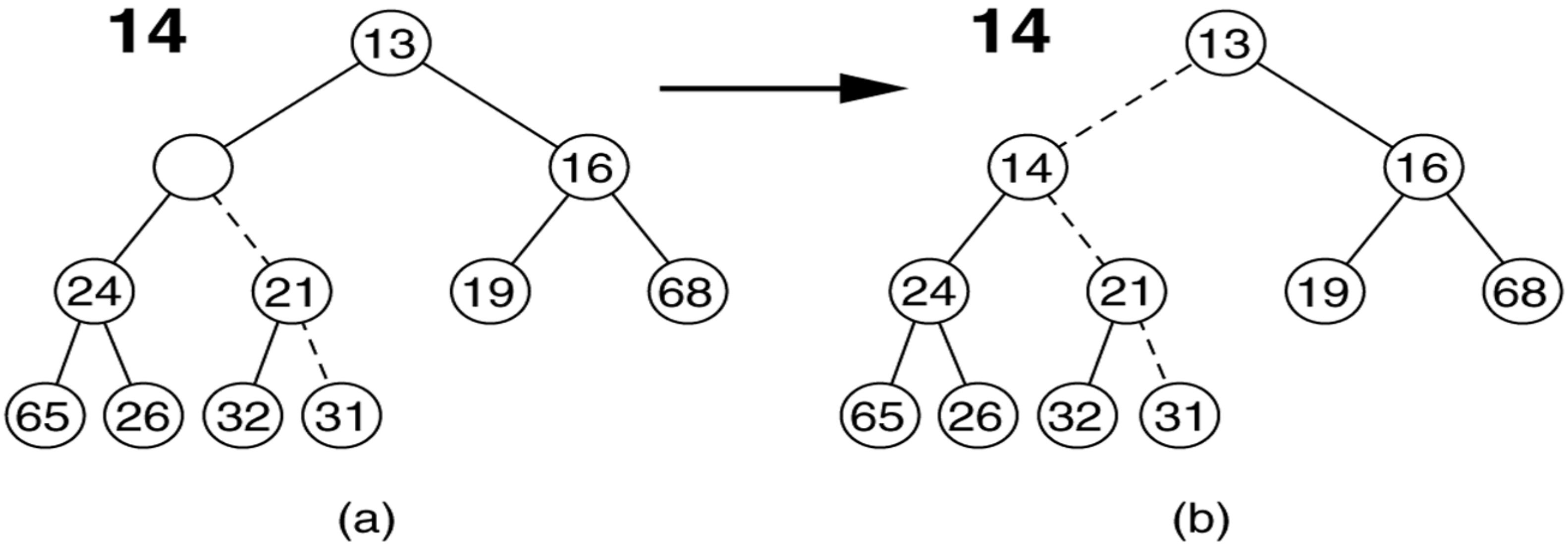
insert 14, creating the hole and bubbling the hole up



0	1	2	3	4	5	6	7	8	9	10	11	12
	13	21	16	24	31	19	68	65	26	32	14	

0	1	2	3	4	5	6	7	8	9	10	11	12
	13	21	16	24	14	19	68	65	26	32	31	

insert 14, creating the hole and bubbling the hole up (contd.)



0	1	2	3	4	5	6	7	8	9	10	11	12
	13	21	16	24	14	19	68	65	26	32	31	

0	1	2	3	4	5	6	7	8	9	10	11	12
	13	14	16	24	21	19	68	65	26	32	31	

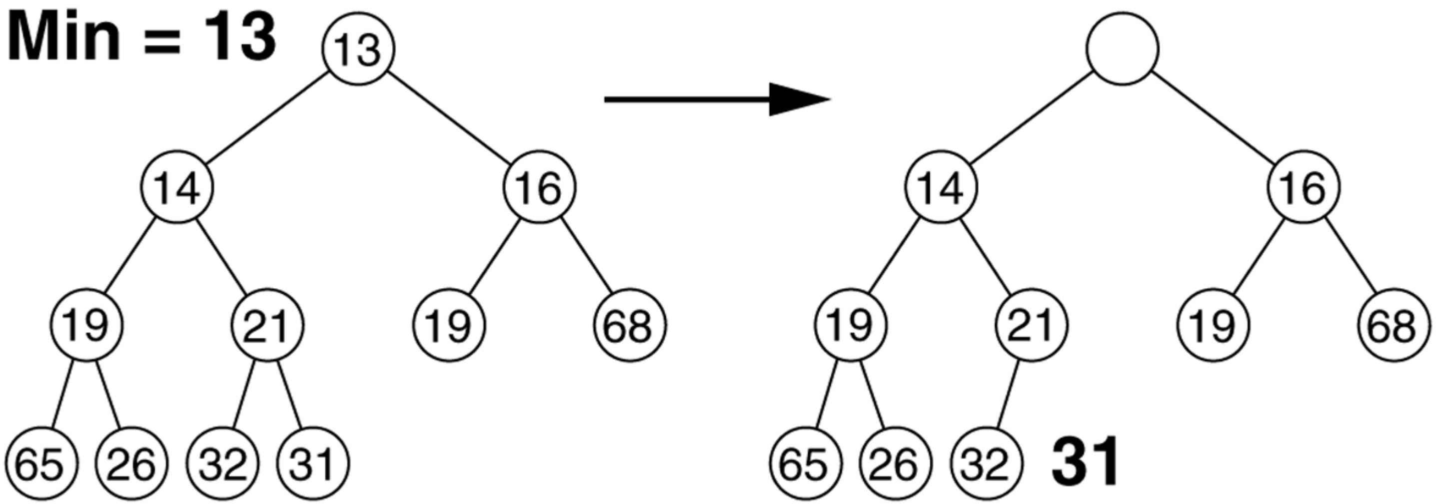
0	1	2	3	4	5	6	7	8	9	10	11	12
	13	14	16	24	21	19	68	65	26	32	31	

Delete Minimum

- **deleteMin** is handled in a similar manner as insertion:
- Remove the minimum; a hole is created at the root.
- The last element X must move somewhere in the heap.
  - If X can be placed in the hole then we are done.
  - Otherwise,
    - We slide the smaller of the hole's children into the hole, thus pushing the hole one level down.
    - We repeat this until X can be placed in the hole.
- deleteMin is logarithmic in both the worst and average cases.

Delete-Min

Creation of the hole at the root

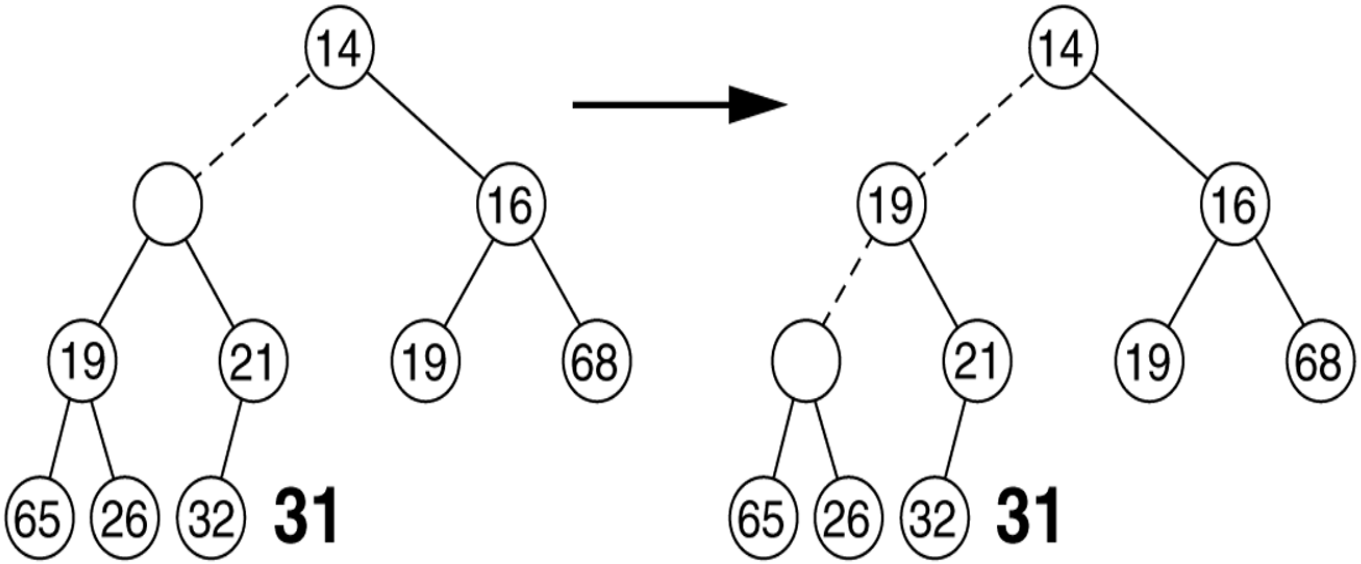


0	1	2	3	4	5	6	7	8	9	10	11	12
	13	14	16	19	21	19	68	65	26	32	31	

0	1	2	3	4	5	6	7	8	9	10	11	12
	31	14	16	19	21	19	68	65	26	32		

Delete-Min

Next two steps

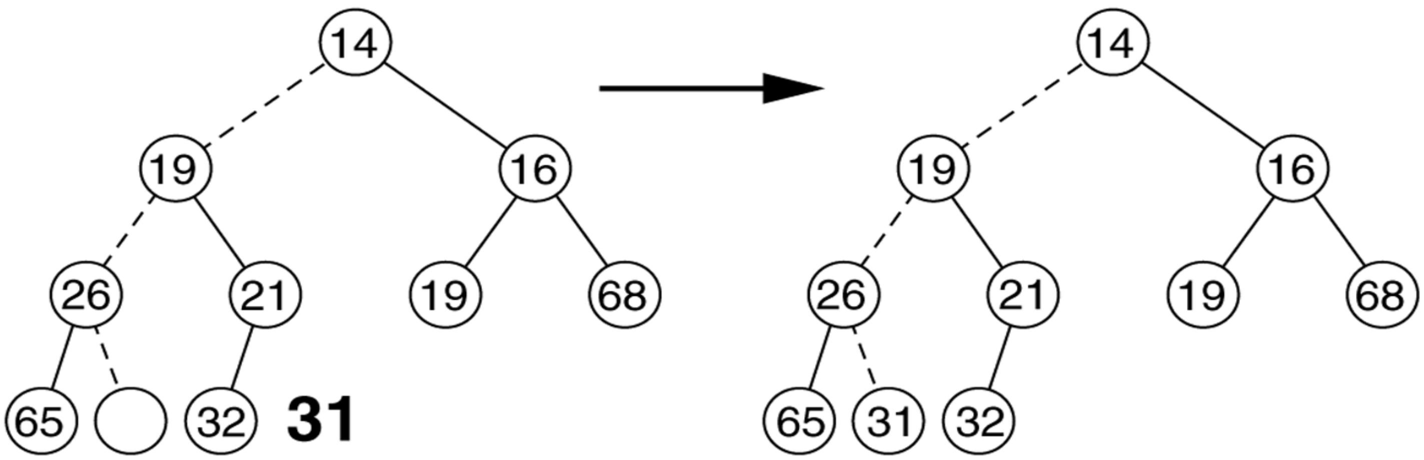


0	1	2	3	4	5	6	7	8	9	10	11	12
	14	31	16	24	21	19	68	65	26	32		

0	1	2	3	4	5	6	7	8	9	10	11	12
	14	19	16	31	21	19	68	65	26	32		

Delete-Min

Last two steps



0	1	2	3	4	5	6	7	8	9	10	11	12
	14	19	16	31	21	19	68	65	26	32		

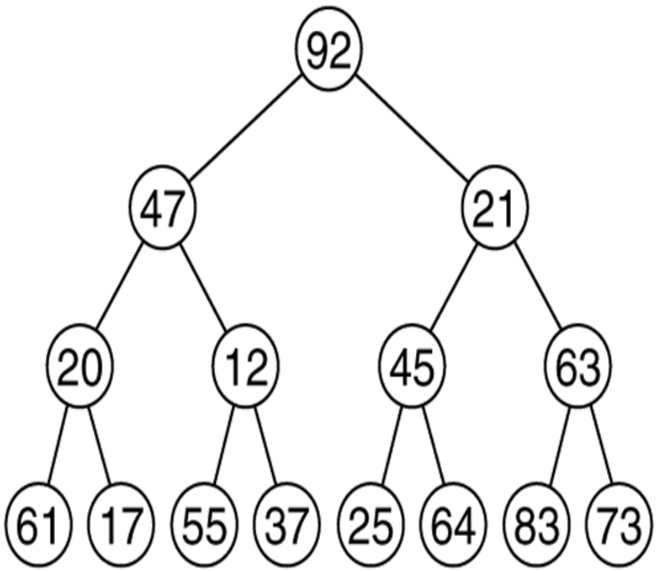
0	1	2	3	4	5	6	7	8	9	10	11	12
	14	19	16	26	21	19	68	65	31	32		

0	1	2	3	4	5	6	7	8	9	10	11	12
	14	19	16	26	21	19	68	65	31	32		

Building a Heap

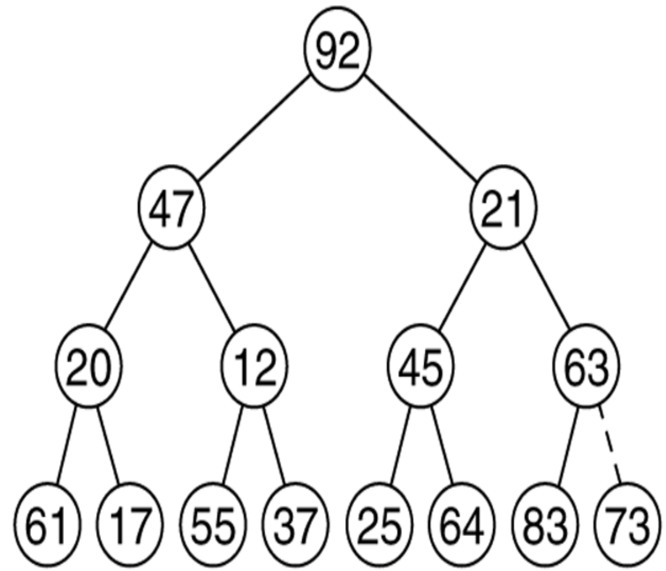
- Take as input N items and place them into an empty heap.
- Obviously this can be done with N successive inserts: $O(N \log N)$ worst case.
- However `buildHeap` operation can be done in linear time ($O(N)$) by applying a percolate down routine to nodes in reverse level order.

Implementation of the linear-time build-Heap method



(a)

Initial complete tree



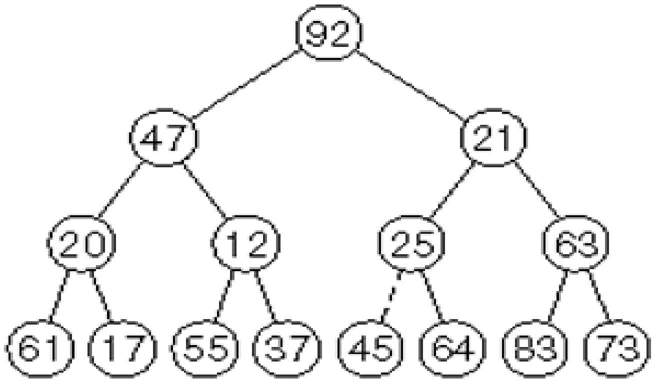
(b)

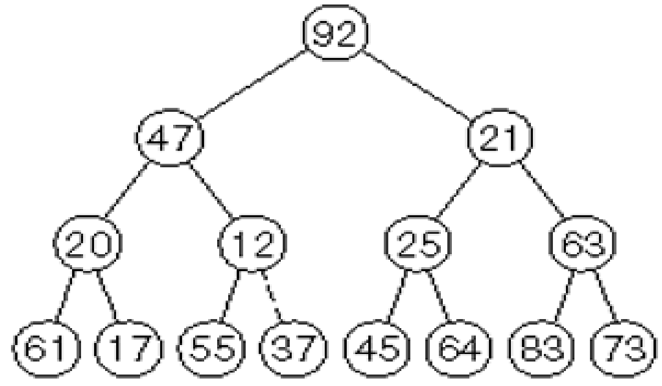
After percolatedown (at node 7)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	45	63	61	17	55	37	25	64	83	73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	45	63	61	17	55	37	25	64	83	73

Implementation of the linear-time build-Heap method





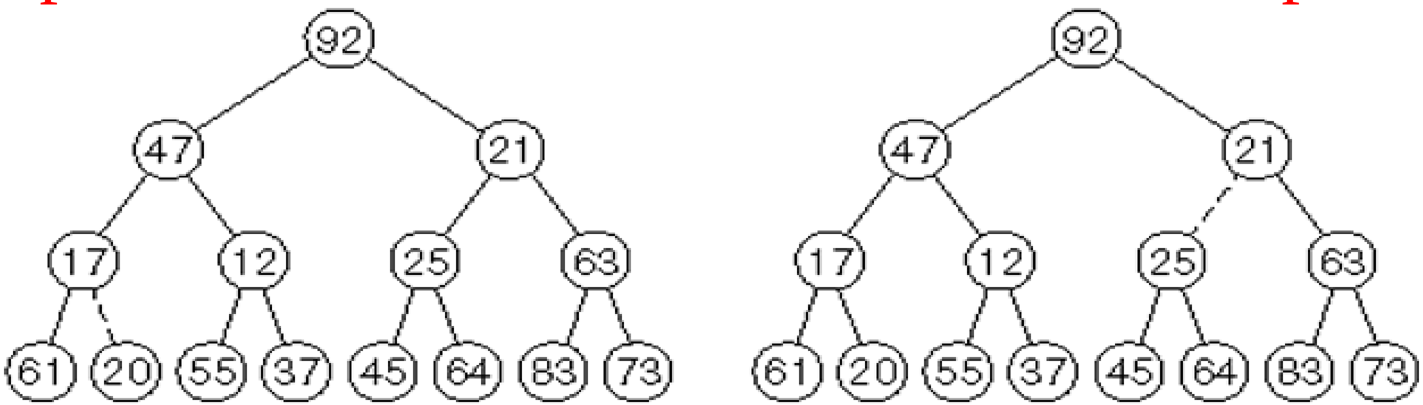
(a) After percolateDown (at node 6)

(b) after percolateDown (at node 5)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	25	63	61	17	55	37	45	64	83	73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	20	12	25	63	61	17	55	37	45	64	83	73

Implementation of the linear-time build-Heap method



(a) (b)

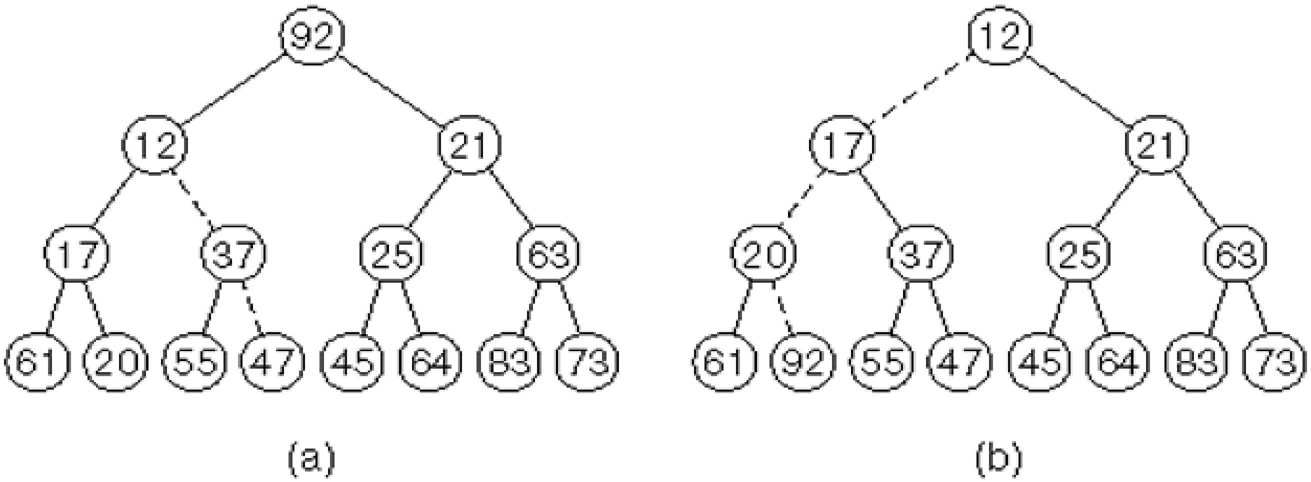
(a) After percolateDown(4)

(b) after percolateDown(3)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	17	12	25	63	61	20	55	37	45	64	83	73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	47	21	17	12	25	63	61	20	55	37	45	64	83	73

Implementation of the linear-time build-Heap method



(a) After percolateDown(2) ((b) after percolateDown(1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	92	12	21	17	37	25	63	61	20	55	47	45	64	83	73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	12	17	21	20	37	25	63	61	92	55	47	45	64	83	73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	12	17	21	20	37	25	63	61	92	55	47	45	64	83	73

Final Heap

Analysis of `buildHeap`

- The linear time bound of `buildHeap`, can be shown by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines.
- For the perfect binary tree of height h containing $N = 2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $N - h - 1$.
- Thus it is $O(N)$.