CSCI2100 Data Structures
Heaps

Irwin King

king@cse.cuhk.edu.hk
http://www.cse.cuhk.edu.hk/~king

Department of Computer Science & Engineering
The Chinese University of Hong Kong

Introduction

- In some applications, a simple queue may not be the best strategy to complete jobs.
    - Printer queue
    - Multiprocessing queue
- Problems
    - Sometimes it seems that small jobs take longer
    - Important jobs can't be done first

Priority Queues (Heaps)

- Different from a simple queue where one adds an entry at the end and takes an entry at the front,
- A priority queue takes an entry that satisfies some special properties among all the entries and place it at the front so to be taken out first.

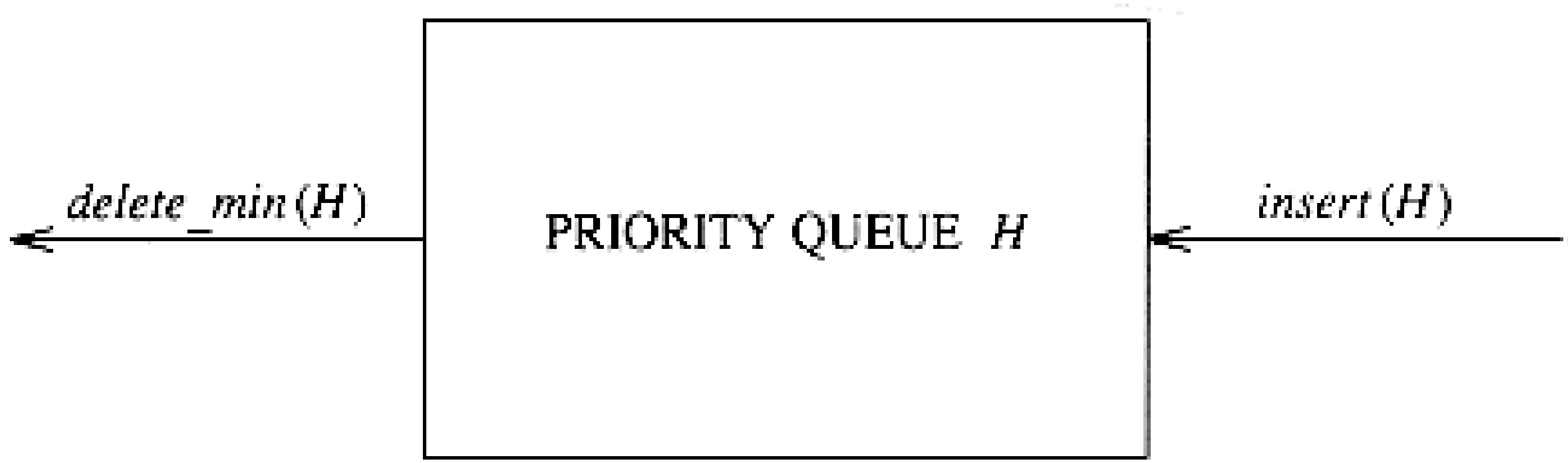
Example

- In a job queue, there are many algorithms that can be implemented to accomplish tasks.
  - first-come-first-serve
  - shortest-job-first
  - longest-job-first
  - priority-first
  - combination of the above

Priority Queue

- A priority queue consists of entries, each of which contains a key called the priority of the entry.
- A priority queue has only two operations other than the usual creation, size, full, and empty operations:
  - Insert--inserts an entry.
  - Delete_Min--finds, passes back, and removes the entry having the highest priority.
- If entries have equal priorities, then the first entry inserted is removed first.

Model of a Priority Queue



Implementation of a Priority Queue

- Several possible implementations are possible.
  - Simple linked list
  - A sorted contiguous list
  - An unsorted list
  - Binary search tree

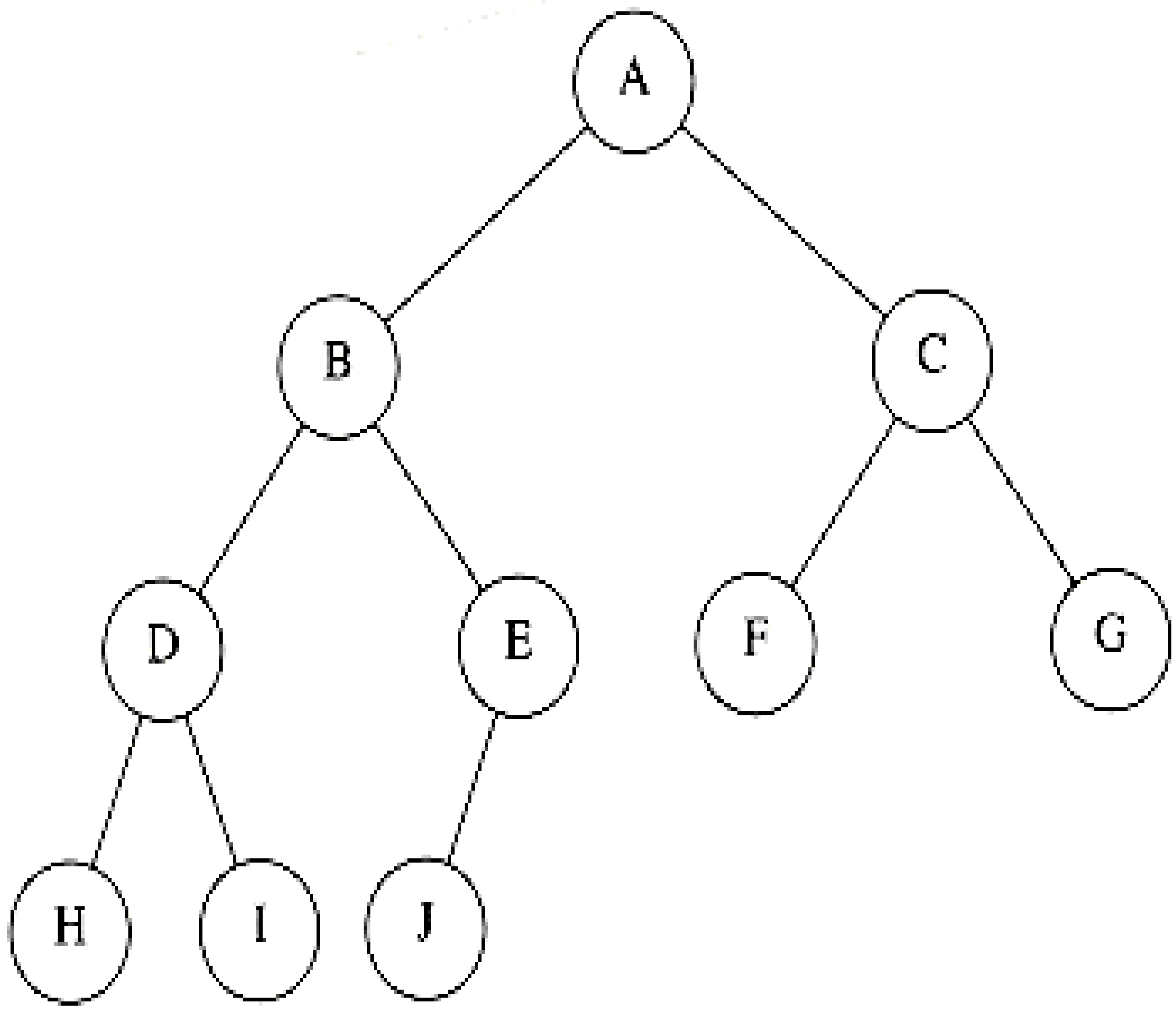
Binary Heap (or just Heap)

- Heaps have two properties
  - Structure property
  - Heap order property
- As with AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order.

Structure Property

- A heap is a binary tree that is completely filled, with the possible **<u>exception</u>** of the bottom level, which is filled from left to right.
- Such a tree is known as a complete binary tree.

Example



Observation

- A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes.
- This implies that the height of a complete binary tree is $\lfloor \log n \rfloor$, which is clearly $O(\log n)$.
- Because a complete binary tree is so regular, it can be represented in an array and no pointers are necessary.

Example of an Implementation

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

- For any element in array position i, the left child is in position 2i, the right child is in the cell after the left child ($2i + 1$), and the parent is in position $\lfloor i/2 \rfloor$.

- Thus not only are pointers not required, but the operations required to traverse the tree are extremely simple.

- Problem is the estimation of the maximum heap size is required in advance.

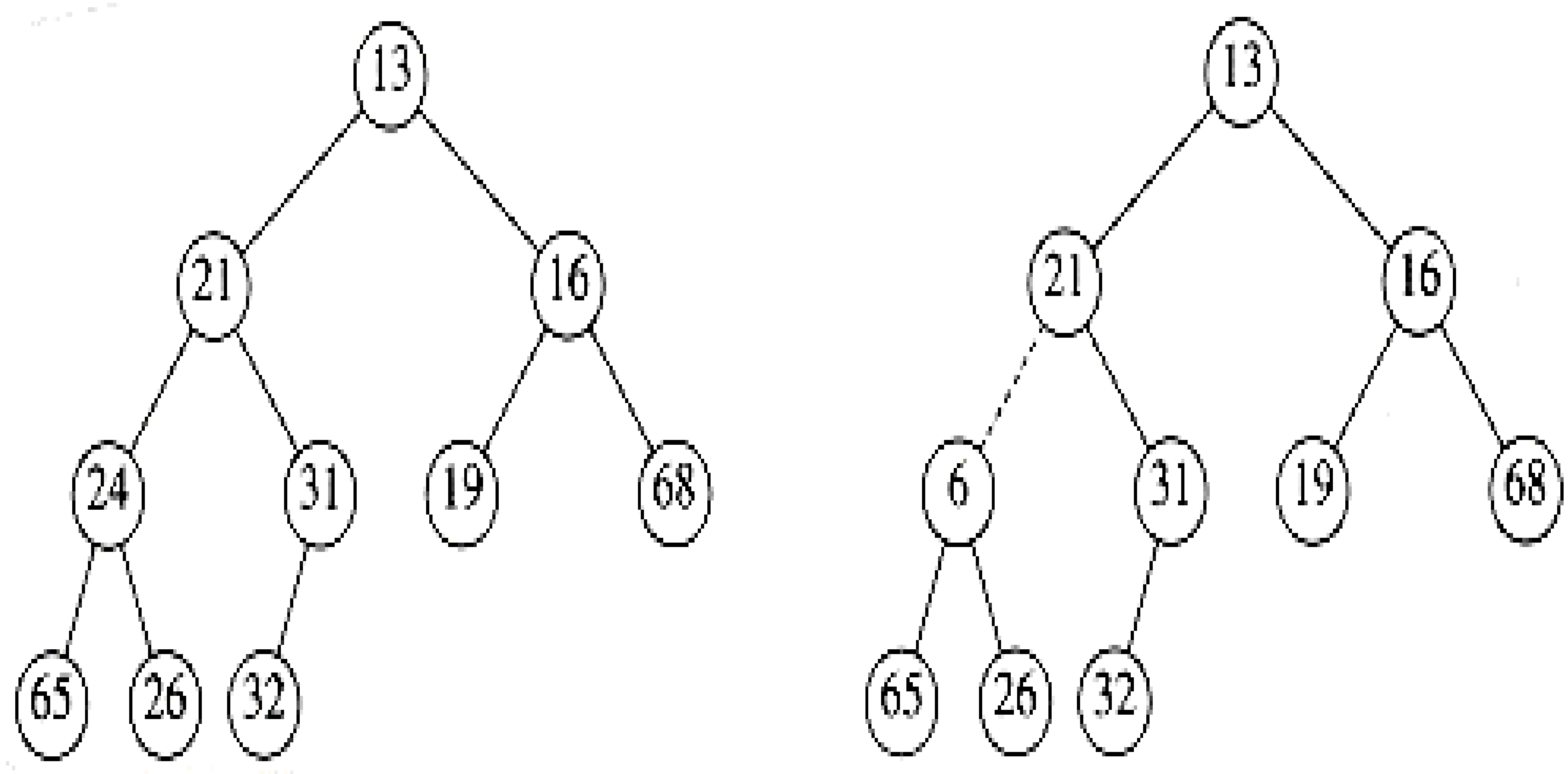
Heap Order Property

- The property that allows operations to be performed quickly is the heap order property.
- For a heap, the smallest element should be at the root so that the operation to remove will be quick.
- By the heap order property, the minimum element can always be found at the root.
- Thus, we get the extra operation, find_min, in constant time, O(1).

Heap Order Property

- Since we want to be able to find the minimum quickly, it makes sense that the smallest element should be at the root.
- If we consider that any subtree should also be a heap, then any node should be smaller than all of its descendants.
- Applying this logic, we arrive at the heap order property.
- In a heap, for every node X, <u>the key in the parent of X is smaller than (or equal to) the key in X</u>, with the obvious exception of the root (which has no parent).

Example

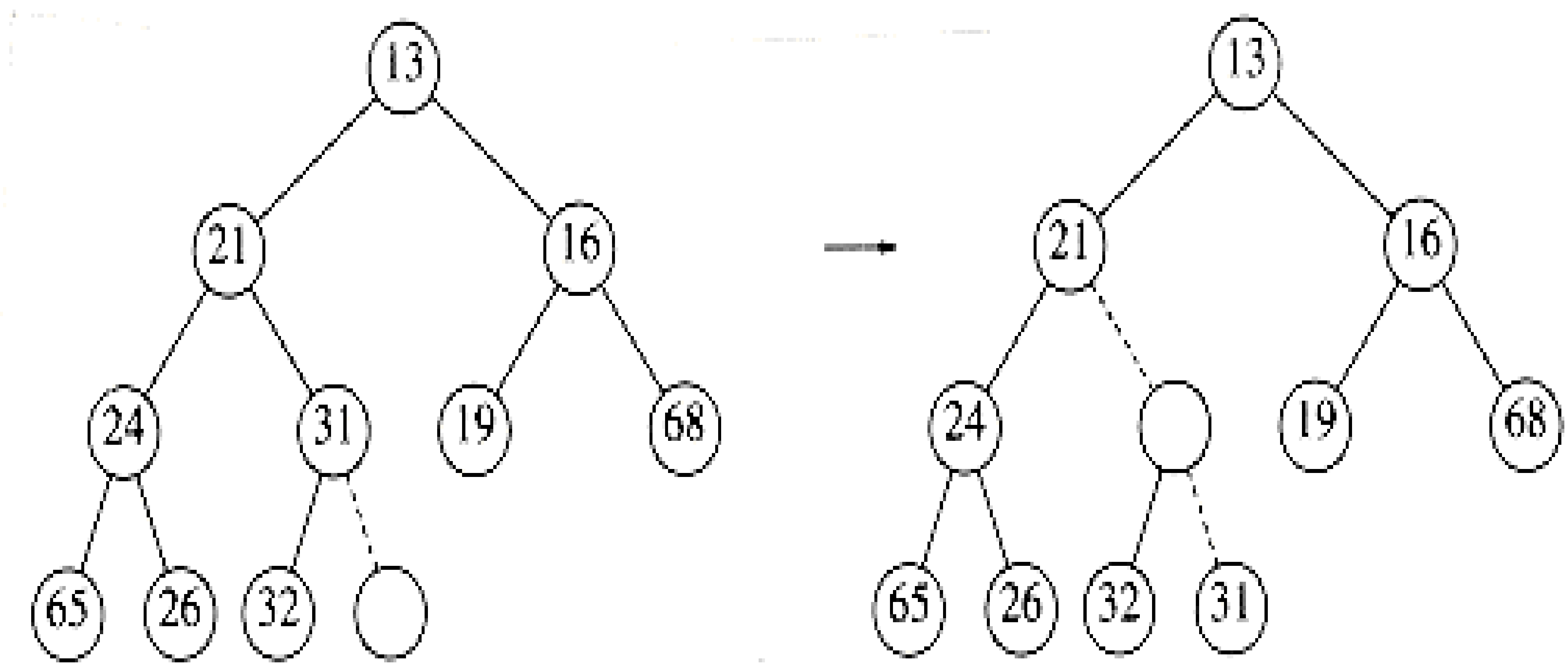


- Two complete trees (only the left tree is a heap).

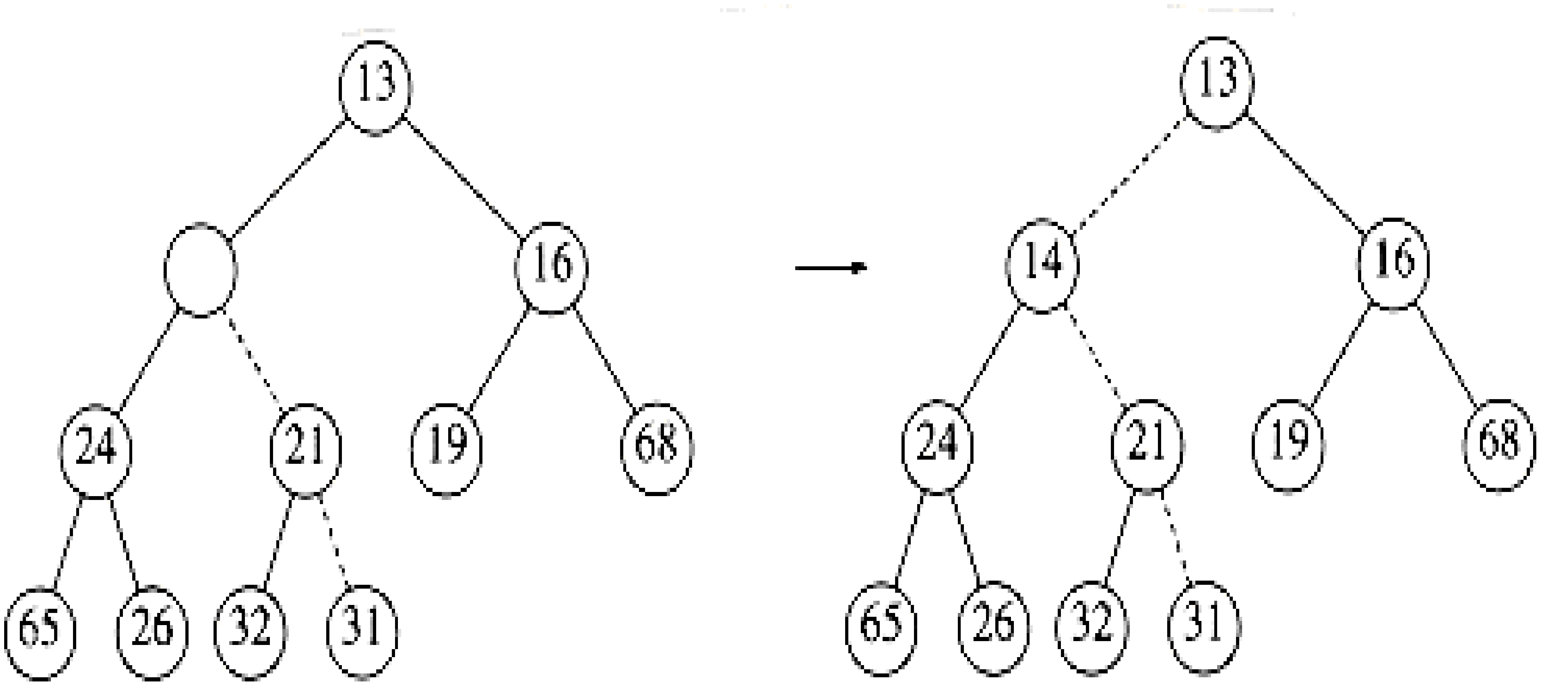
Heap Operations - Insert

- We create a hole in the next available location.
  - If x can be placed in the hole without violating the heap order, then we do so and are done.
- Otherwise we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root.
- We continue this process until x can be placed in the hole.
- This strategy is known as a percolate up.

Example-Insert 14



Example



Observation

- The time to do the insertion could be as much as O(log *n*) if the element to be inserted is the new minimum and is percolated all the way to the root.
- It has been shown that 2.607 comparisons are required on average to perform an insert.
- The average insert moves an element up 1.607 levels.

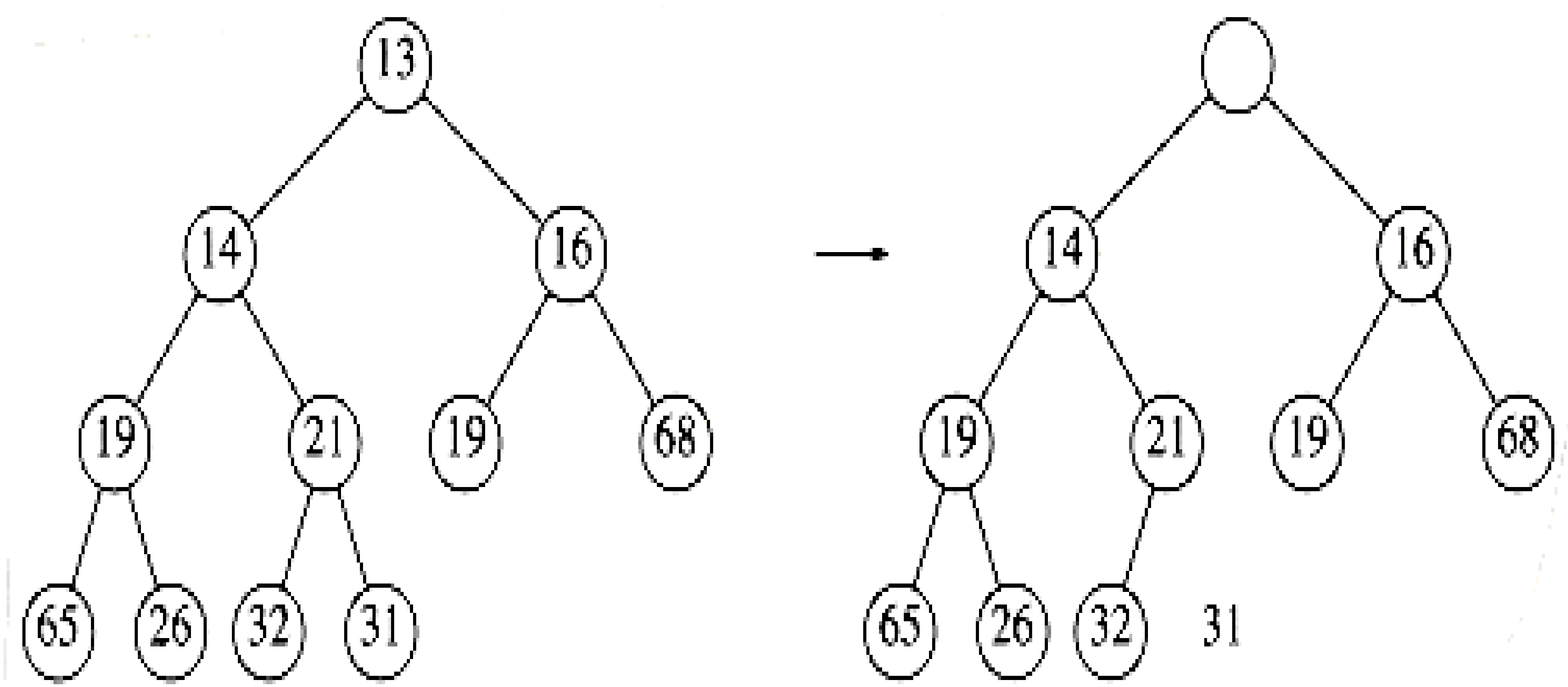
Heap Operations - Delete

- Deletions are handled in a similar manner as insertions.
- Finding the minimum is easy; the hard part is removing it.
- When the root is removed, a hole is created.
- We then need to slide the smaller of the hole's children into the hole, thus pushing the hole down one level.

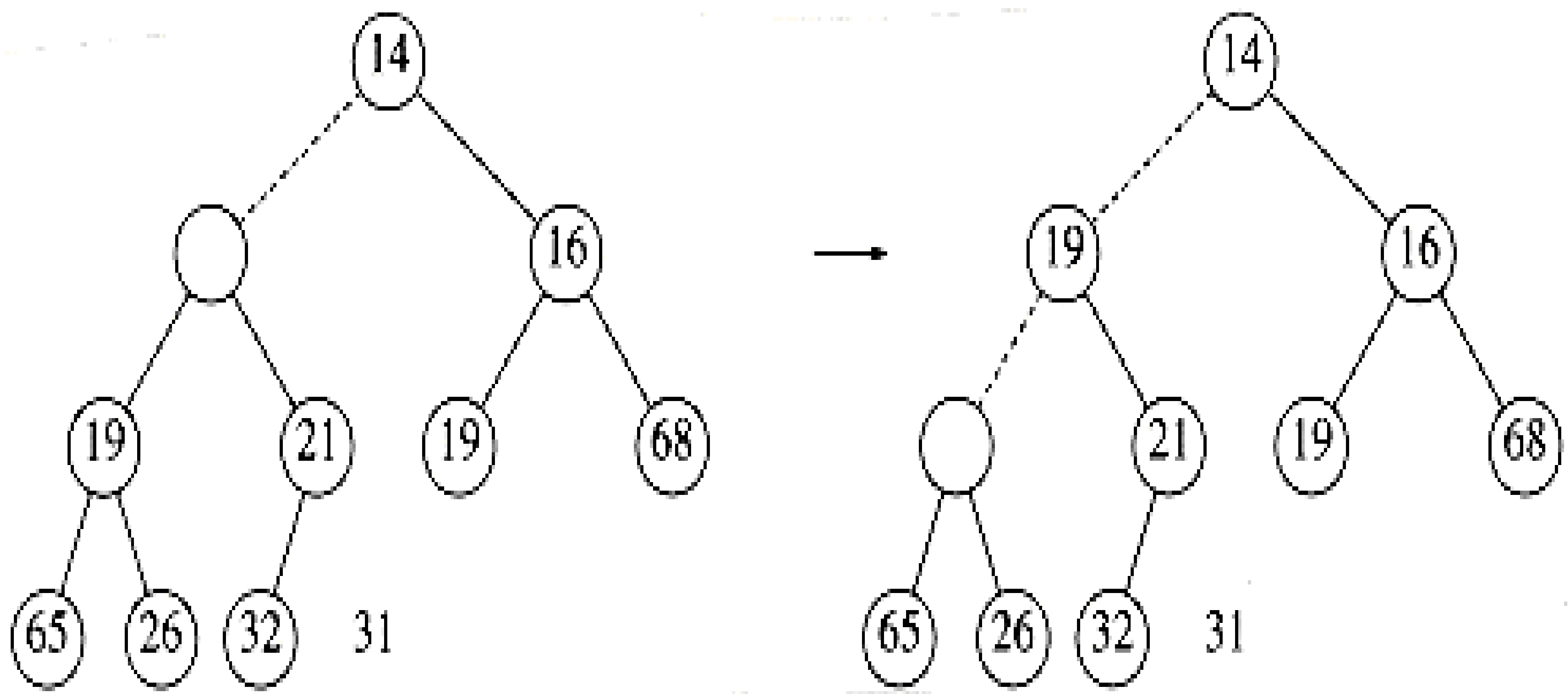
Deletion

- We repeat this step until *x* can be placed in the hole.
- Thus, our action is to place *x* in its correct spot along a path from the root containing minimum children.
- The rearranging will typically take less than O(log *n*).

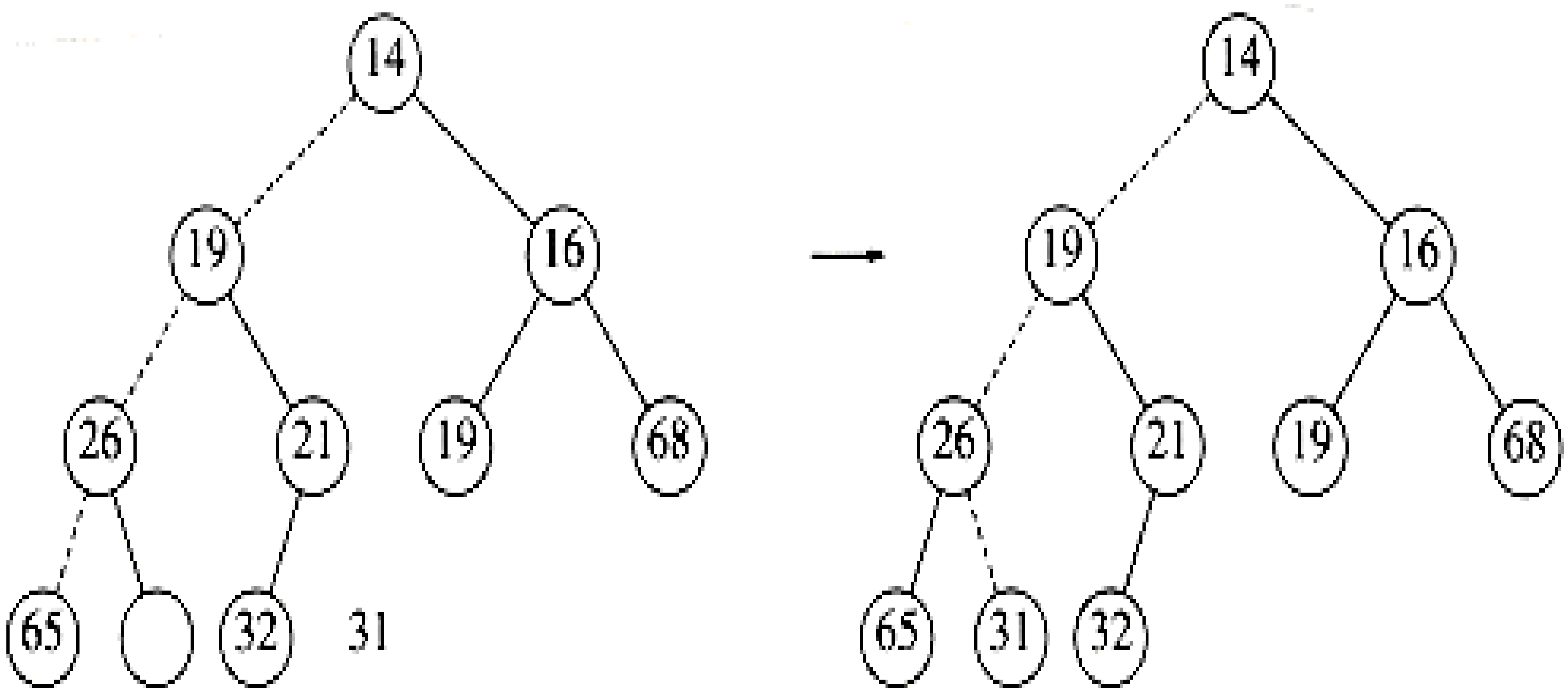
Example



Example



Example



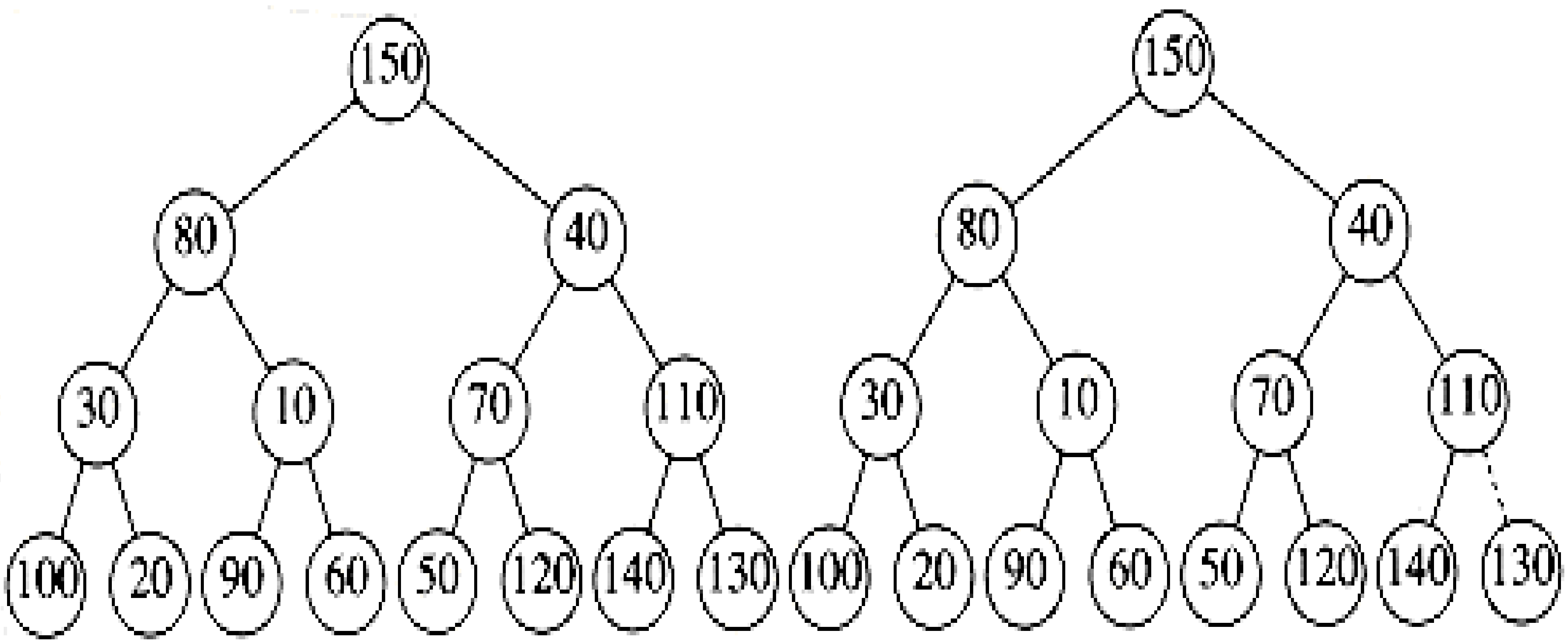
Other Heap Operations

- Finding the minimum can be performed in constant time.
- No help in finding the maximum.
- There is no ordering information.
- Decrease_Key (P,Δ)
- Increase_Key(P,Δ)
- Remove(I)
- Build_Heap

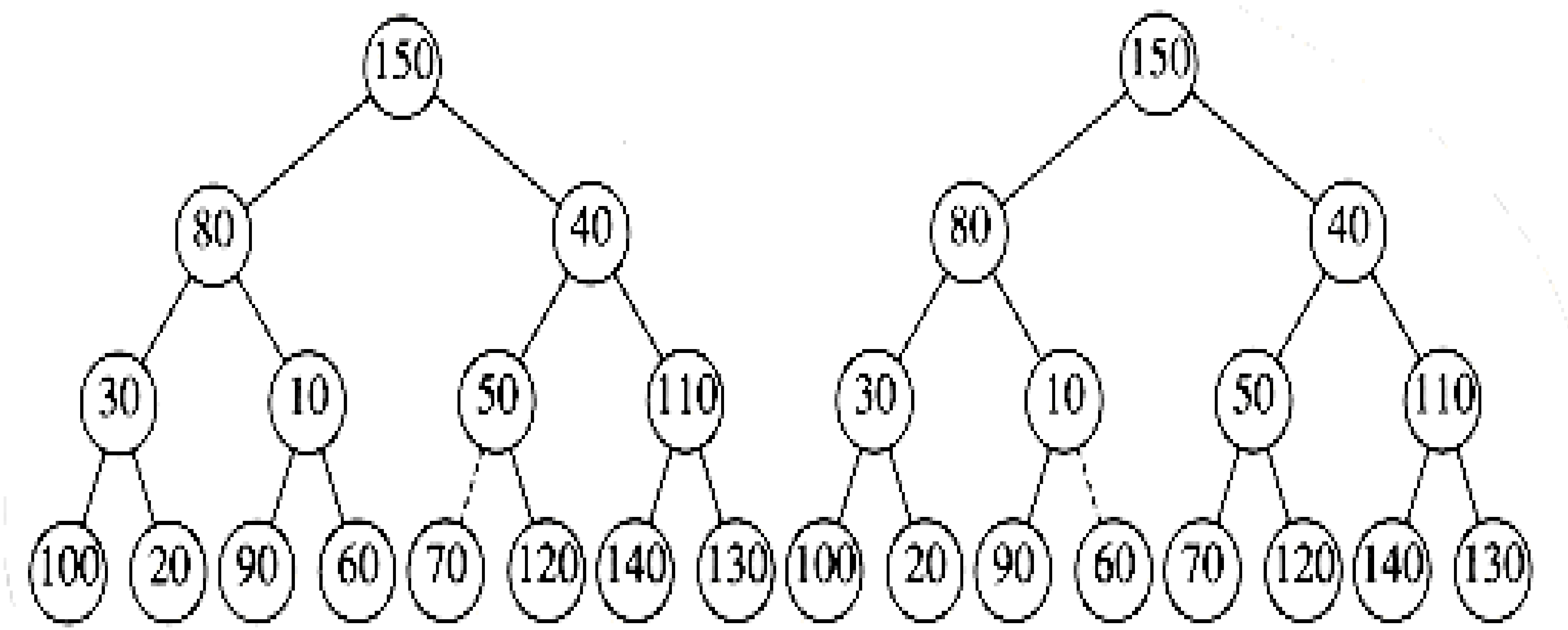
Observation on Build_Heap

- Takes *n* keys and places them into an empty heap.
- We could perform *n* successive Inserts.
- This will take O(*n*) average but O(*n* log *n*) worst-case.
- One other way is to place the n keys into the tree in any order.
    - Then perform Percolate_Down on half of the nodes.

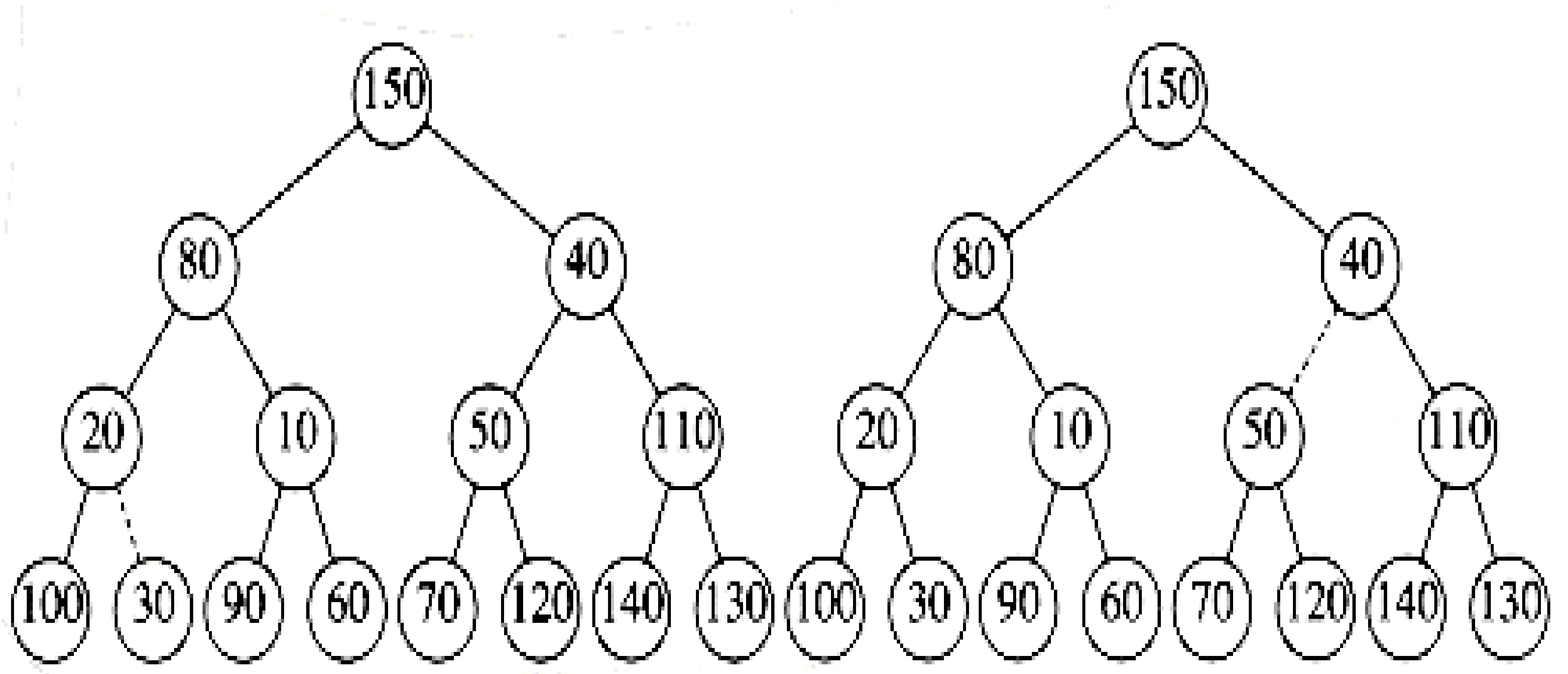
Example-Initial, Percolate Down(7)



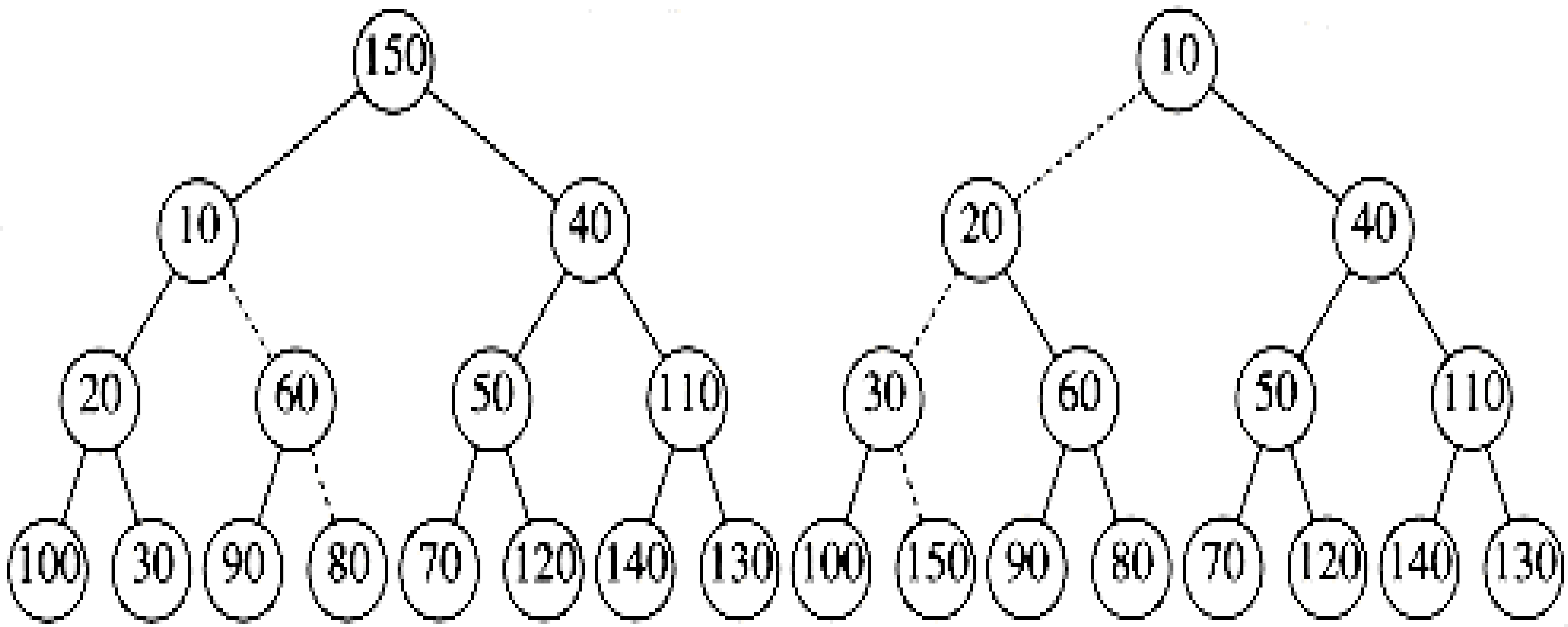
Example-Percolate_Down(6), Percolate_Down(5)



Example-Percolate_Down(4), Percolate_Down(3)



Example-Percolate_Down(2), Percolate_Down(1)



Back to the k Selection Problem

First Algorithm

- We now could use what we just learned and apply it to find out the *k*-th smallest or largest element in a set.
- To build a heap, it takes $O(n)$ average and $O(n \log n)$ for worst case scenario.
- To delete a heap, it take $O(\log n)$.
- Hence, the total running time is $O(n + k \log n)$.

More

- For small k then the running time dominated by the heap building operation and is O(*n*).
- For larger values of *k*, the running time is O(*k* log *n*) time.

Second Algorithm

- We could also build a smaller heap tree of *k* elements.
- It then compares the remaining entries against the heap. If the new element is larger, then it replaces the root or else it is being discarded.
- To build a *k* element heap, the time will be O(*k*).

More

- The time to process each of the remaining elements is O(1), to test if the element goes into the heap, plus O(log k), to delete the root and insert the new element if this is necessary.
- Thus, the total time is O(k + (n-k) log k) = O(n log k).
- This algorithm also gives a bound of n log n for finding the median.