Data Structures and Algorithms
(ES221)

# QUEUES (HEAPS)

Dr. Zubair Ahmad

# Priority Queues

- Jobs sent to a line printer are generally placed on a queue
- In a multiuser environment, the operating system scheduler must decide which of several processes to run
- **The queue is FCFS**
  - dequeue the first job
  - run it until either it finishes or its time limit is up, and
  - enqueue if not done within the time limit
  - FCFS may not always be so efficient
    - May not be fair with the short jobs, waiting too long in the queue
- What is the solution?
- Shortest job first (SJF)
  - finish the short jobs as fast as possible
  - Shorter jobs should have preference over longer jobs running
- Priority Queues
  - Some jobs that are not short are still very important and should also have preference

# Priority Queues

- **FCFS**
  - High Priority jobs may be delayed
  - A long job may force several smaller jobs to wait for unreasonably long period of time

| **Pages** | **100** | **4** | **2** | **2** | **2** | **1** | **111** |
|---|---|---|---|---|---|---|---|
| Wait time | 0 | 100 | 104 | 106 | 108 | 110 | 528 |
| Printing Time | 100 | 104 | 106 | 108 | 110 | 111 | 539 |

Average waiting time = 528 / 6 = 88 units

Average printing time = 539 / 6 = 89.8 units

- **SJF (Shortest Job First)**

| **Pages** | **1** | **2** | **2** | **2** | **4** | **100** | **111** |
|---|---|---|---|---|---|---|---|
| Wait time | 0 | 1 | 3 | 5 | 7 | 11 | 27 |
| Printing Time | 1 | 3 | 5 | 7 | 11 | 111 | 138 |

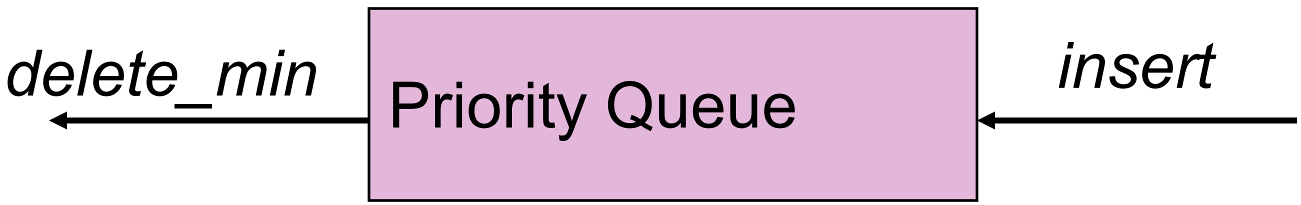
Average waiting time = 27 / 6 = 4.5 units

Average printing time = 138 / 6 = 23 units


3

# Priority Queues Implementation

- Two usual operations
  - ***Insert***: Insert a new job in the queue
  - ***delete_min***: Remove the shortest/highest priority job from the queue



# Priority Queues Implementation

- Simple linked list implementation
  - ***Insert:*** Insert at the front/end of the queue
    - $O(1)$ running time
  - ***Delete_min:*** requires to traverse the whole list
    - $O(n)$ running time

- Sorted linked list implementation
  - ***Insert:*** Insert at its proper location requires traversal
    - $O(n)$ running time
  - ***Delete_min:*** delete the first element which is the min.
    - $O(1)$ running time

# Priority Queues Implementation

- Binary search tree implementation
  - ***Insert:*** Insert at it proper location requires tree traversal
    - $O(\log n)$ running time
  - ***Delete_min:*** delete the deepest child from the leftmost subtree
    - $O(\log n)$ running time

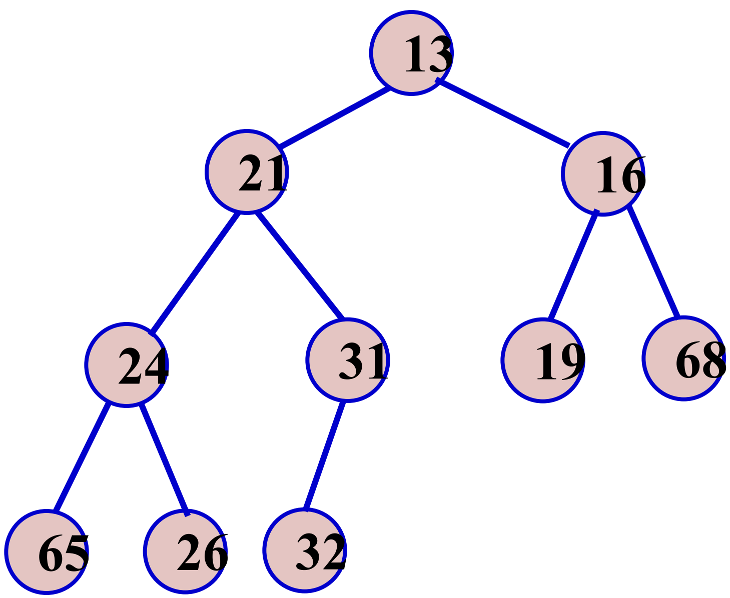
  - Insertions are random
  - deletions are not (why?)
    - Repeated removal of the minimum will hurt the balance of the tree (how?)
    - Worst case scenario
      - The left subtree is depleted (all its nodes deleted)
      - the right subtree would have at most twice as many elements as it should.
      - This adds only a small constant to its expected depth
  - Using Binary Search is over skill
    - We only need two operations

# Heaps

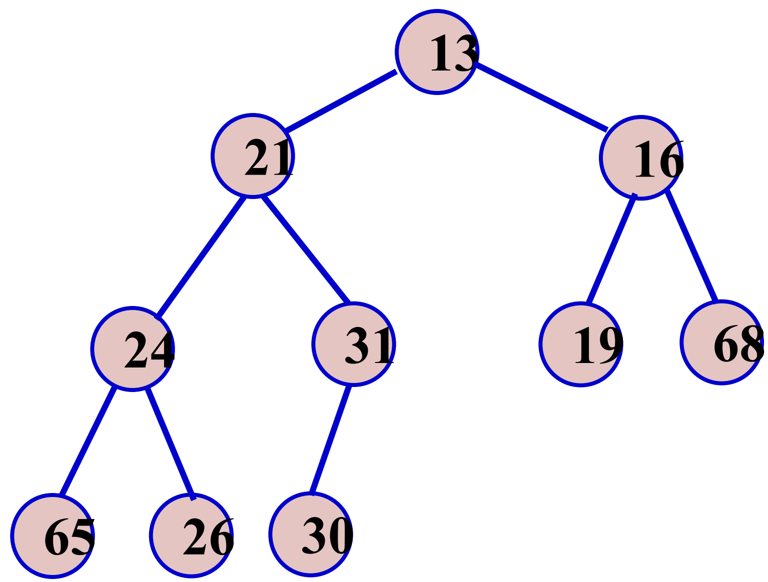
- *binary heap ~ heap*
- *A binary tree which is completely filled*
- *The last level may not be completely filled*
    - It is filled from left to right
- It is easy to show that a complete binary tree of height $h$ has between $2^h$ *and* $2^{h+1} - 1$ *nodes*
- The height of a complete binary tree is $\lfloor \log n \rfloor$ which is clearly $O(\log n)$.

# Heaps

Value of an element at a node is less than or equal to that of its descendants. (Heap property)



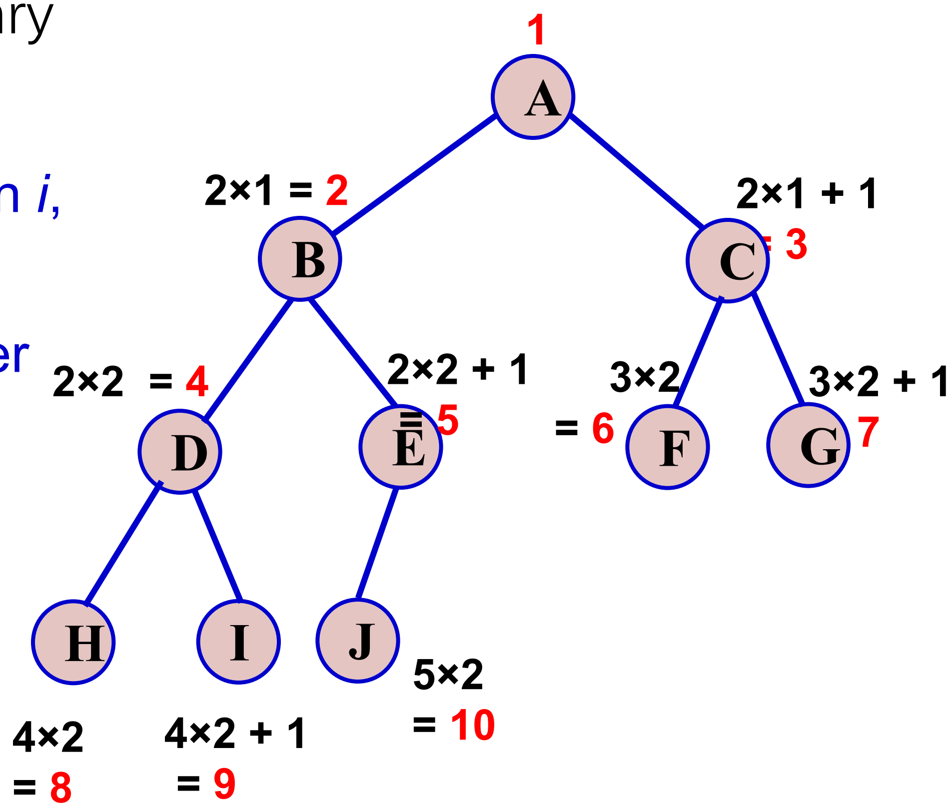

A heap




Not a heap

# Heaps: Array implementation

- An important observation is that because a complete binary tree is so regular, it can be represented in an array and no pointers are necessary




- Start from index 1
- For any element in array position *i*,
  - the left child is in position 2*i*,
  - the right child is in the cell after the left child (2i + 1), and
  - the parent is in position  i/2
- Any problem?
  - an estimate of the maximum heap size is required in advance

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

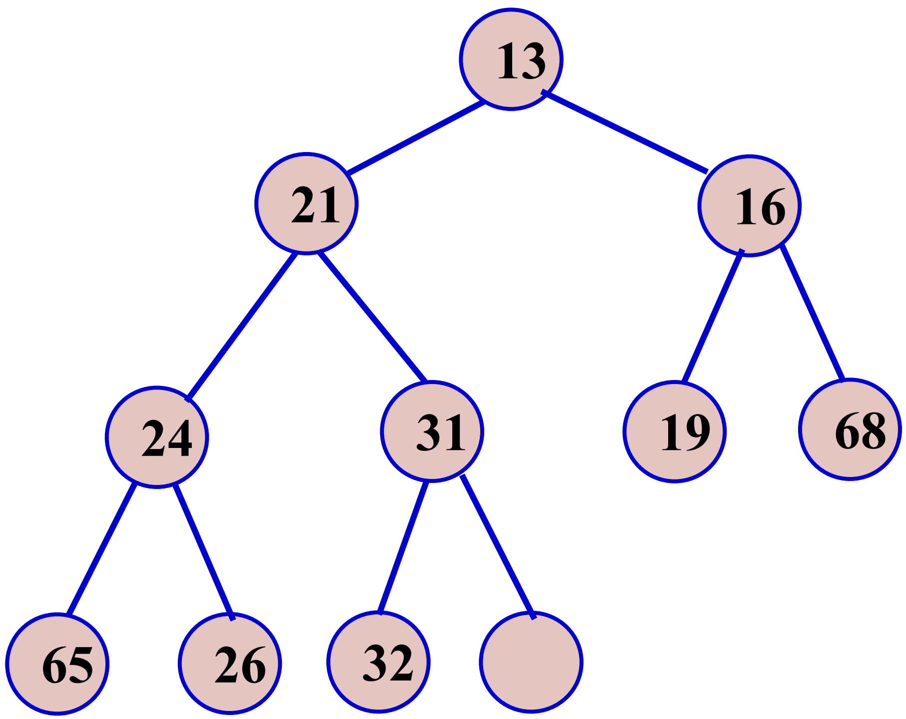
# Heaps - Operations

- Insert
- Delete

# Heaps – insert operation

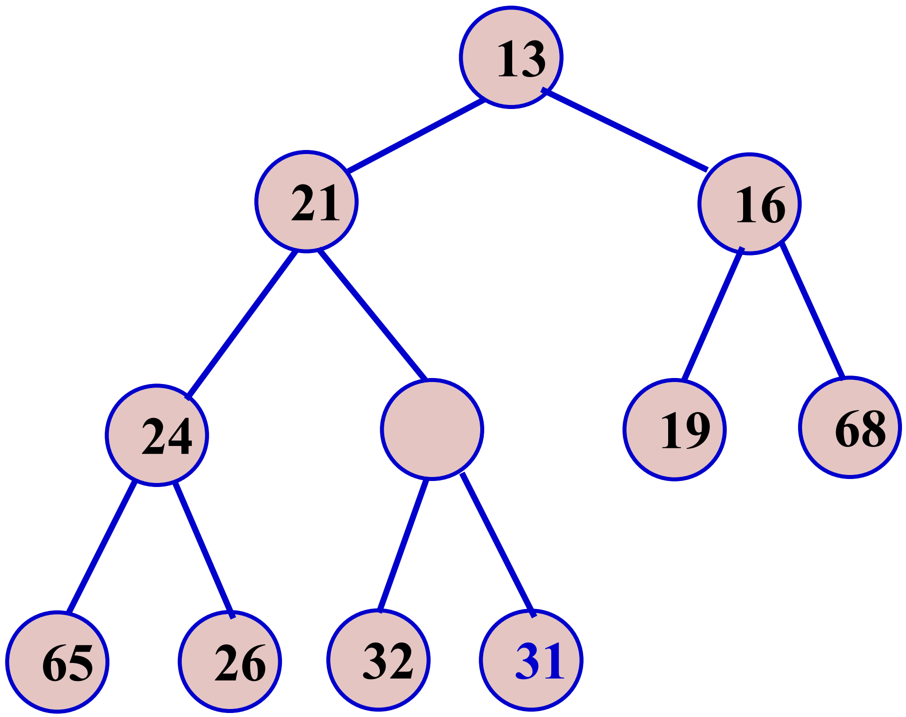
- *Insert x into the heap*
  - *Create a hole in the next available location*
    - *since otherwise the tree will not be complete*
  - *If x* can be placed in the hole without violating heap order, then
    - we do so and are done
  - Else
    - bubble the hole up
      - by sliding the element that is in the hole's parent node into the hole
  - Continue this process until *x* can be placed in the hole
  - This general strategy is known as a *percolate up;*
    - the new element is percolated up the heap until the correct location is found

# Heaps – insert 14



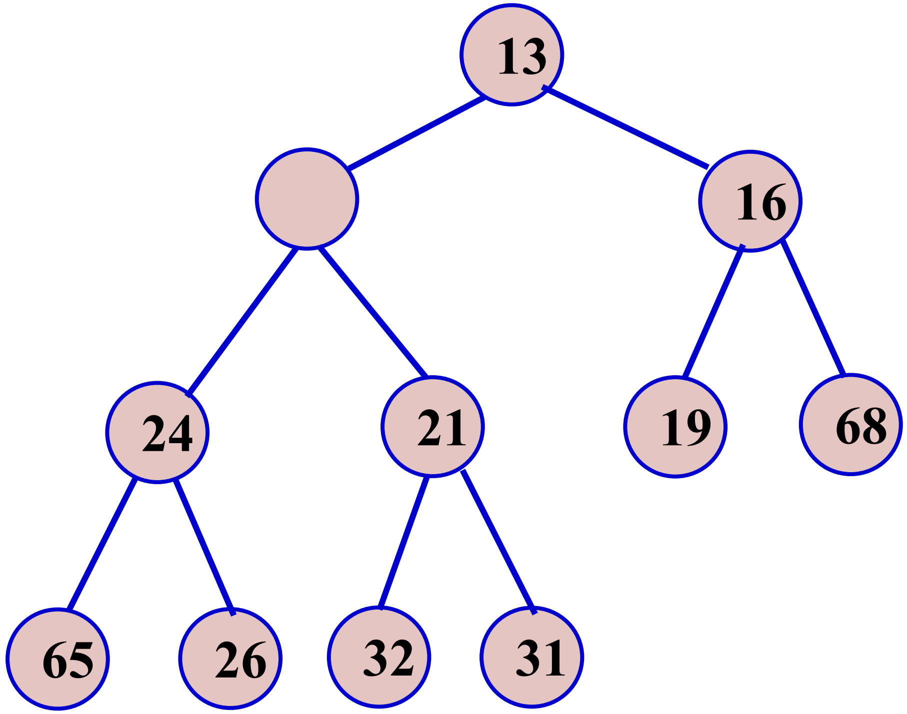

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 13 | 21 | 16 | 24 | 31 | 19 | 68 | 65 | 26 | 32 | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

# Heaps – insert 14



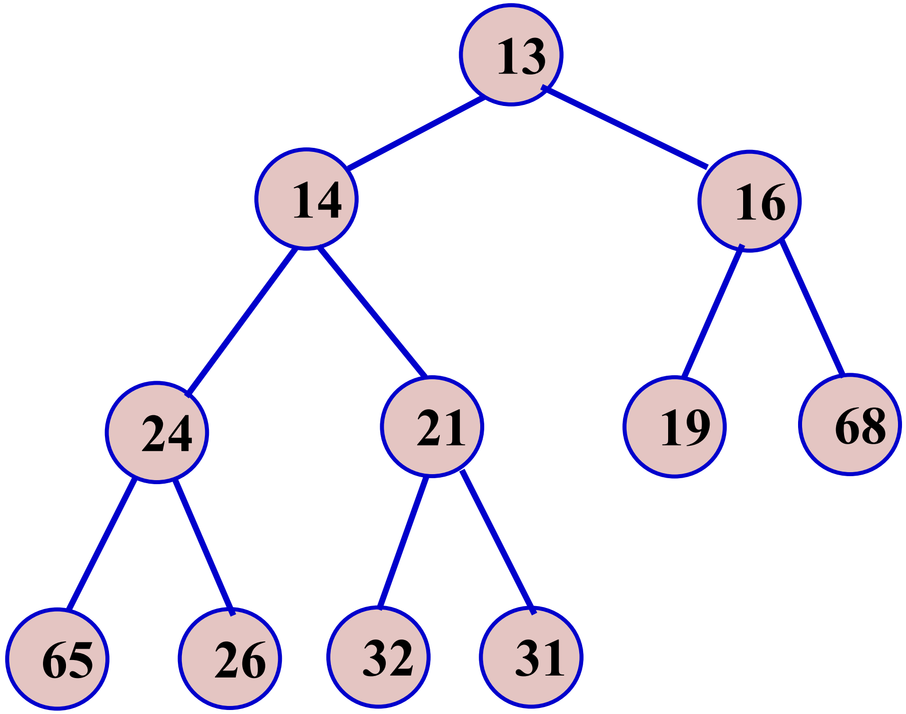

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 13 | 21 | 16 | 24 | | 19 | 68 | 65 | 26 | 32 | 31 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

# Heaps – insert 14




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

# Heaps – insert 14




| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 |

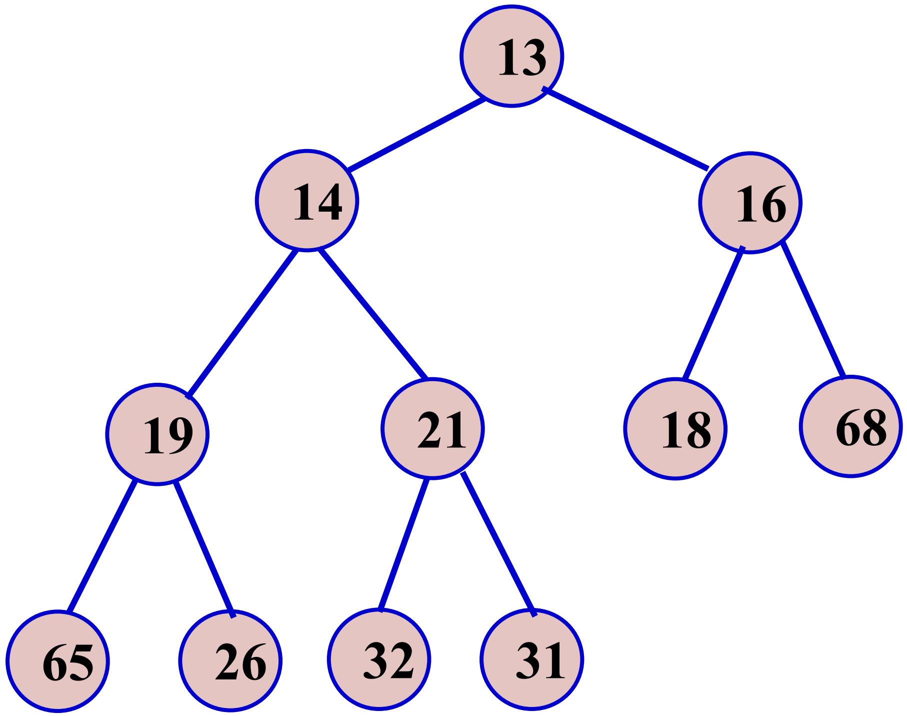
# Heaps – insert Operation

- We could have implemented the percolation in the insert routine by performing repeated swaps until the correct order was established
  - but a swap requires three assignment statements.
- If an element is percolated up ***d*** levels, the number of assignments performed by the swaps would be **3*d***

# Heaps – delete_min Operation

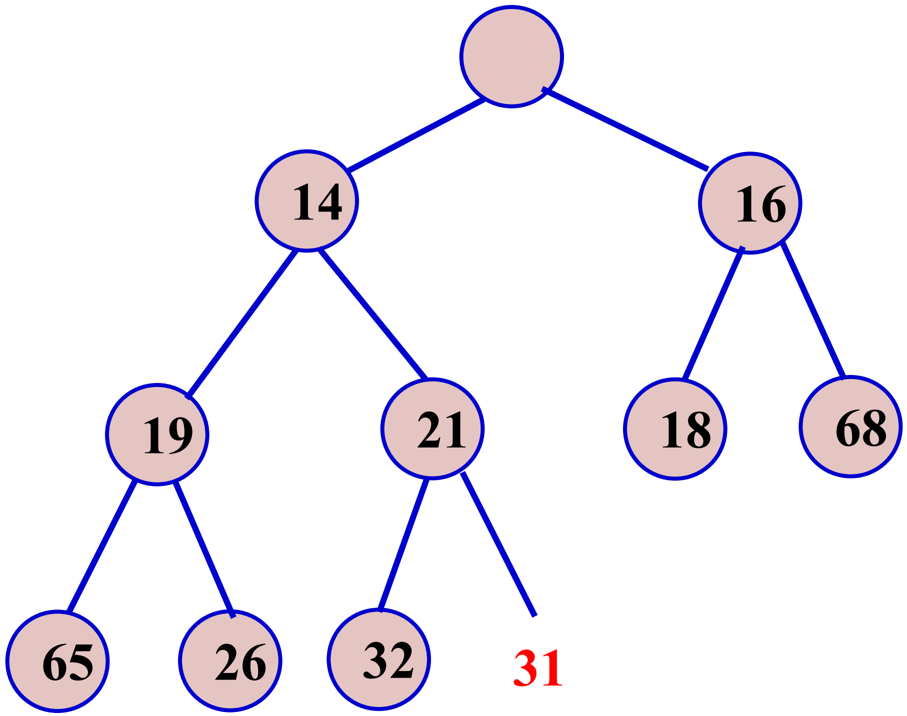
- *Analogous to* insertions.
- When the minimum is removed, a hole is created at the root.
- Since the heap now becomes one smaller, it follows that the last element ***x*** in the heap must move somewhere in the heap
- If ***x*** can be placed in the hole (which is unlikely) then
  - we are done
- Else
  - push the hole down one level
    - by slidimg the smaller of the hole's children into the hole
- Repeat this step until ***x*** can be placed in the hole
- **The result is**: place ***x*** in its correct spot along a path from the root containing *minimum* children
  - This general strategy is known as a *percolate down;*

# Heaps – delete_min : 13



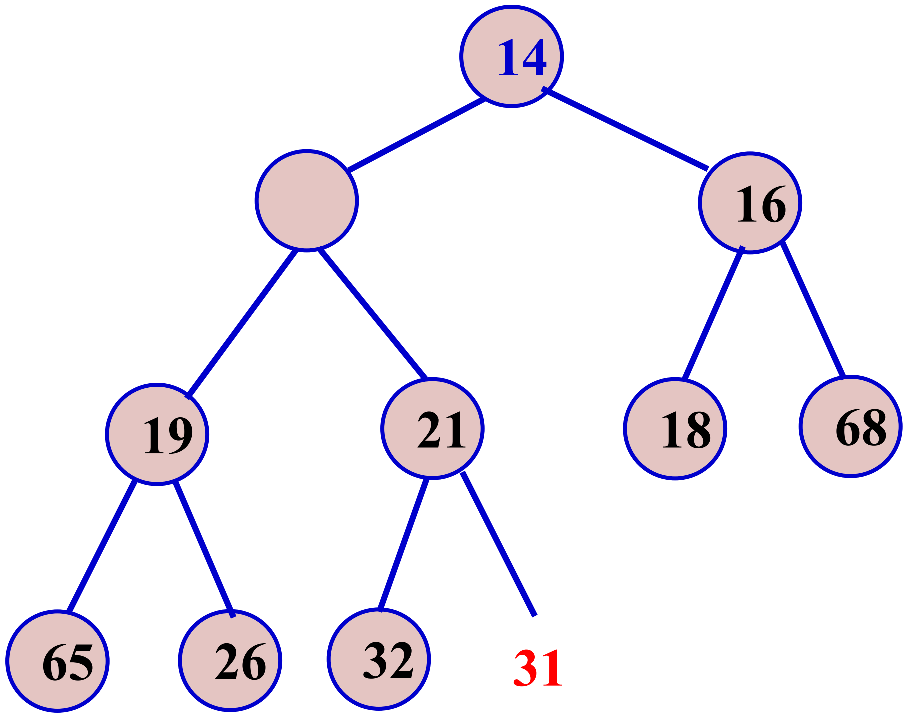

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 13 | 14 | 16 | 19 | 21 | 18 | 68 | 65 | 26 | 32 | 31 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

# Heaps – delete_min : 13



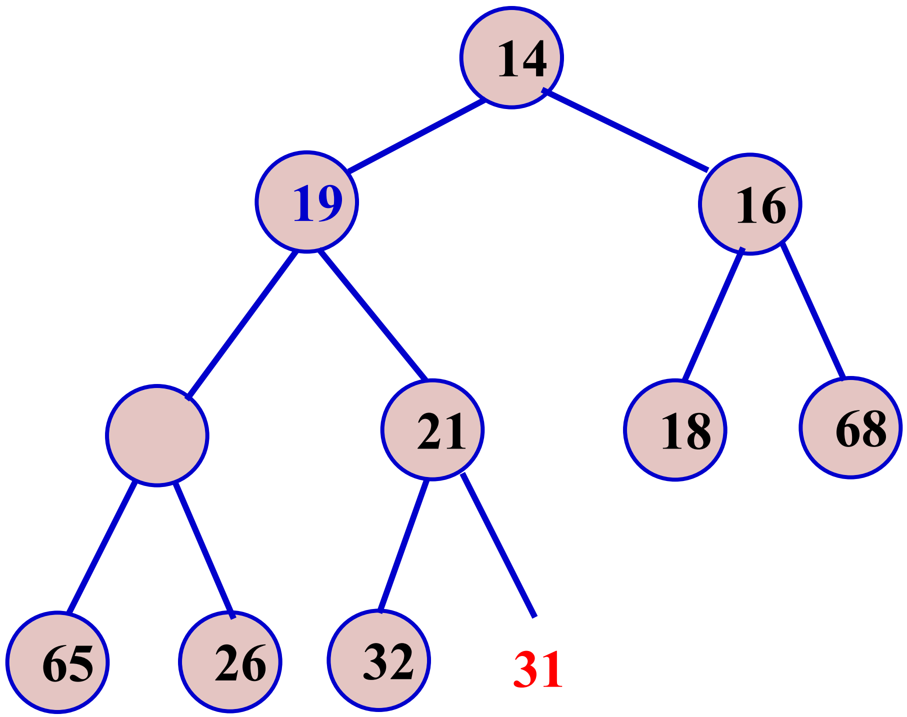

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 14 | 16 | 19 | 21 | 18 | 68 | 65 | 26 | 32 | 31 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

# Heaps – delete_min : 13



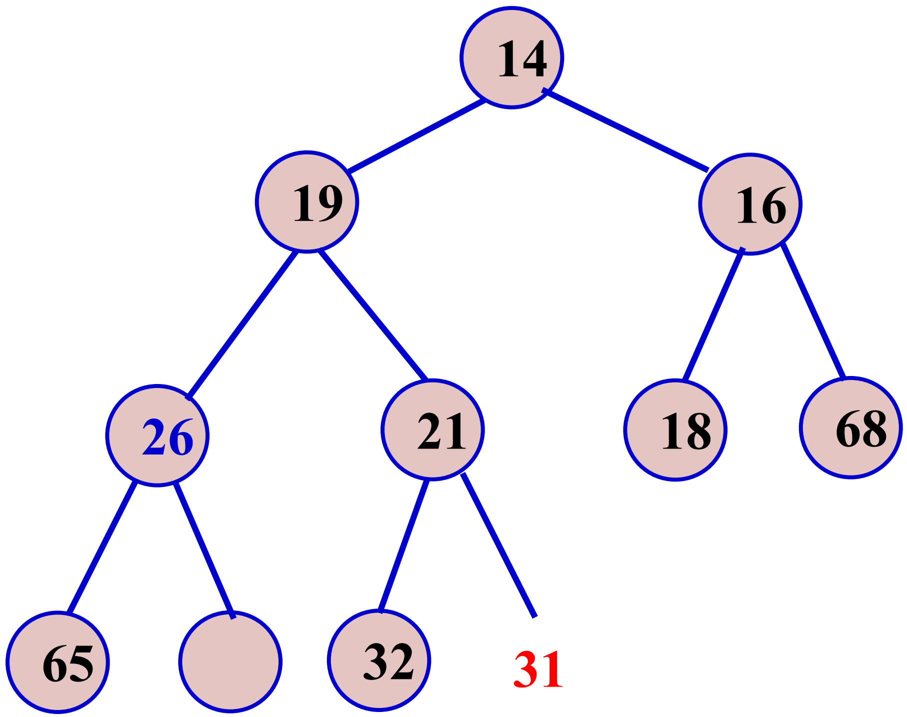

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 14 | | 16 | 19 | 21 | 18 | 68 | 65 | 26 | 32 | 31 | | |

# Heaps – delete_min : 13



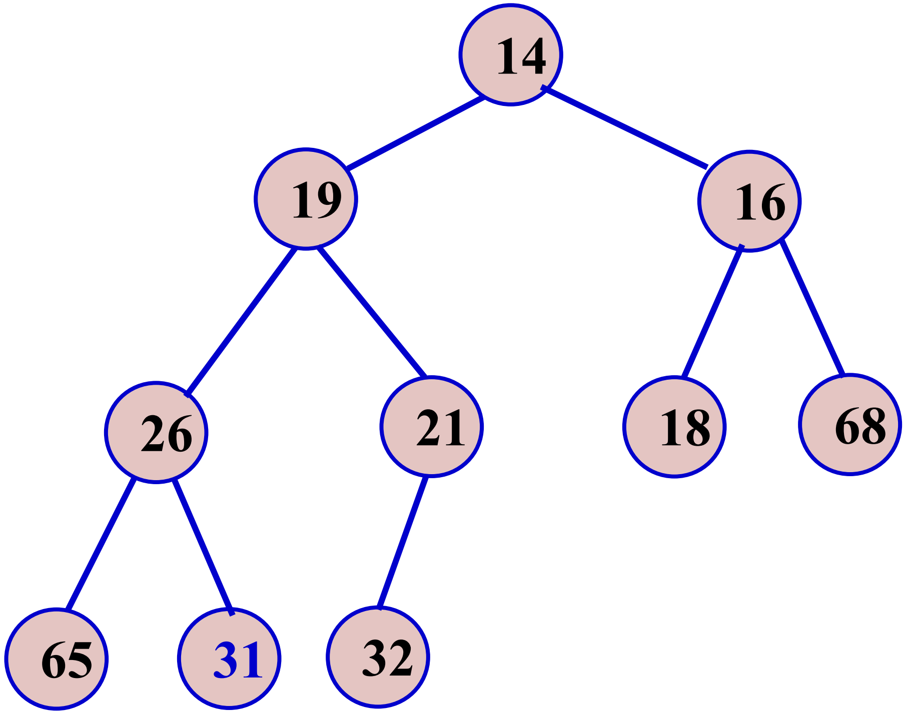

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 14 | 19 | 16 | | 21 | 18 | 68 | 65 | 26 | 32 | 31 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

# Heaps – delete_min : 13




| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 14 | 19 | 16 | 26 | 21 | 18 | 68 | 65 | | 32 | 31 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

# Heaps – delete_min : 13



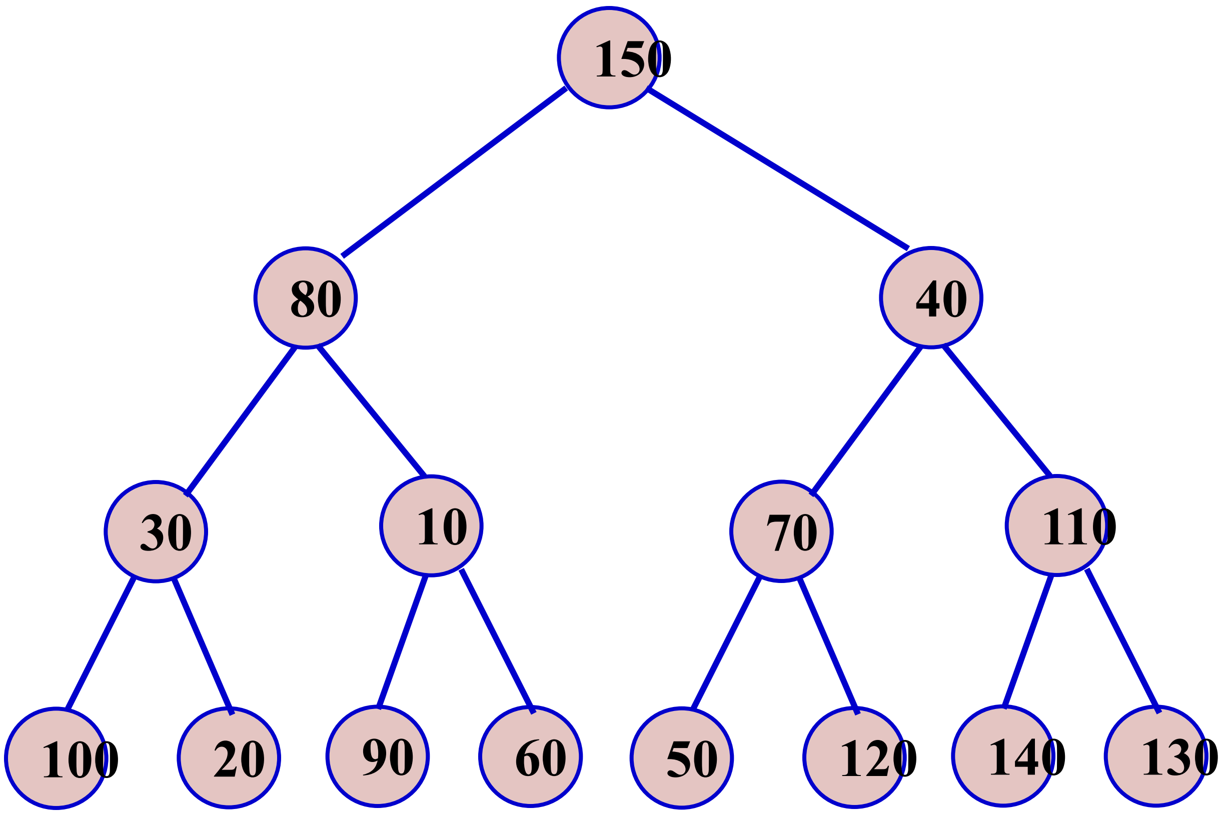

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 14 | 19 | 16 | 26 | 21 | 18 | 68 | 65 | 31 | 32 | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

# Heap Operations: Build Heap

- The general algorithm is to place the *n* keys into the tree in any order, maintaining the structure property.
- Then, if *percolate_down*(*i*) percolates down from node *i*, perform the following algorithm to create a heap-ordered tree

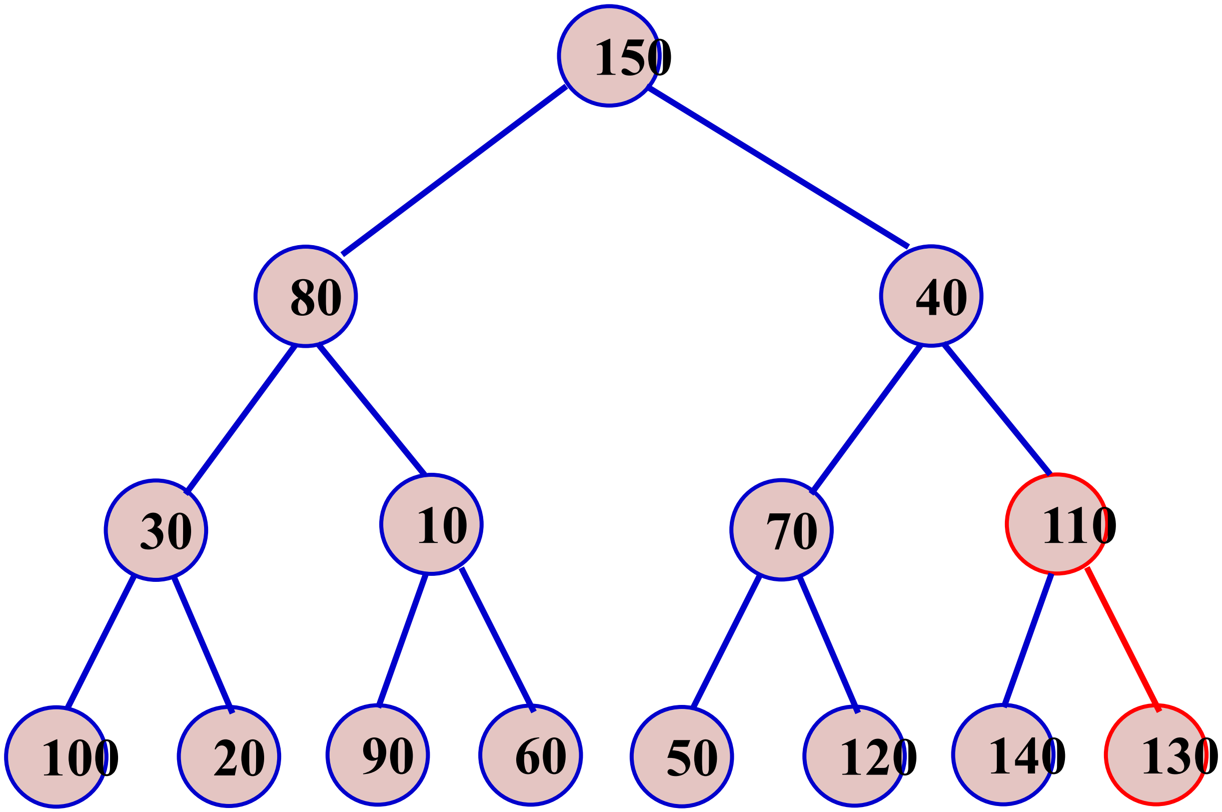
```
for(i = n / 2; i > 0; i-- )
  percolate_down( i );
```

# Build Heap



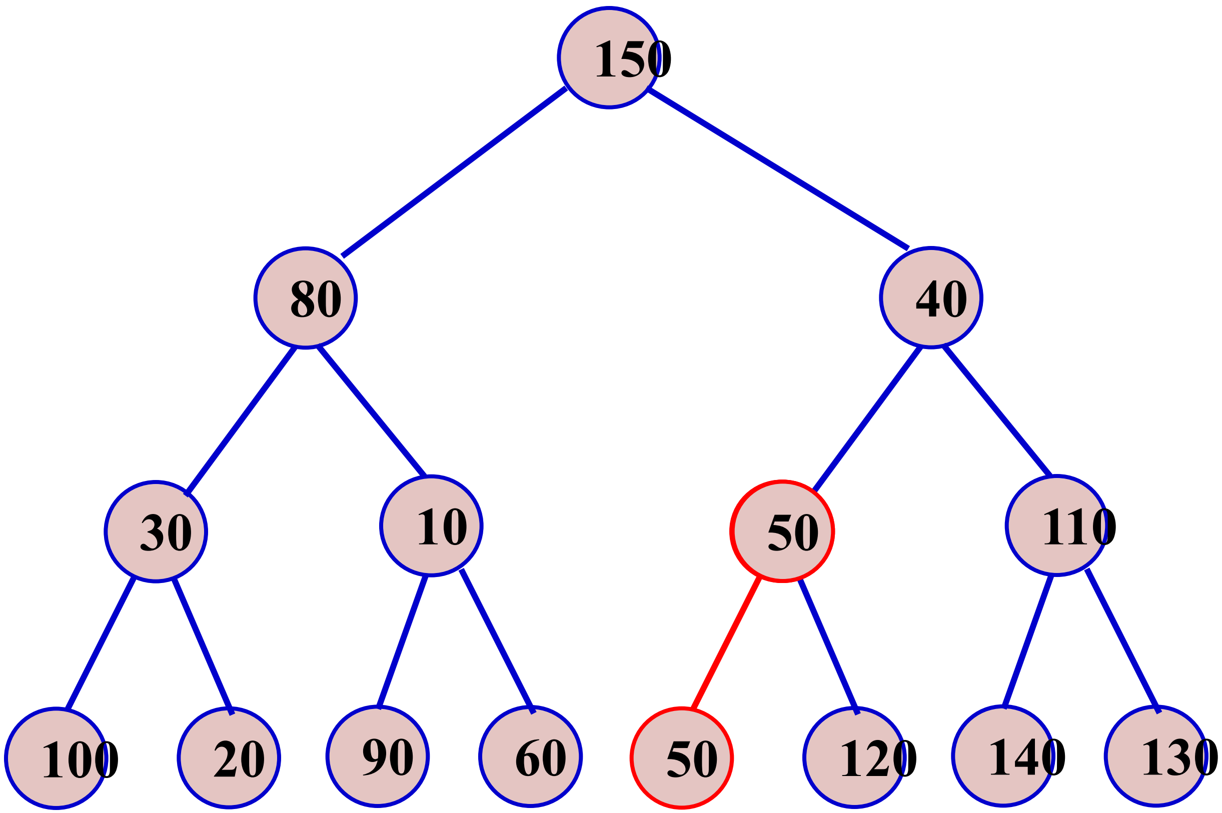

Initial (unordered) Heap

# Build Heap



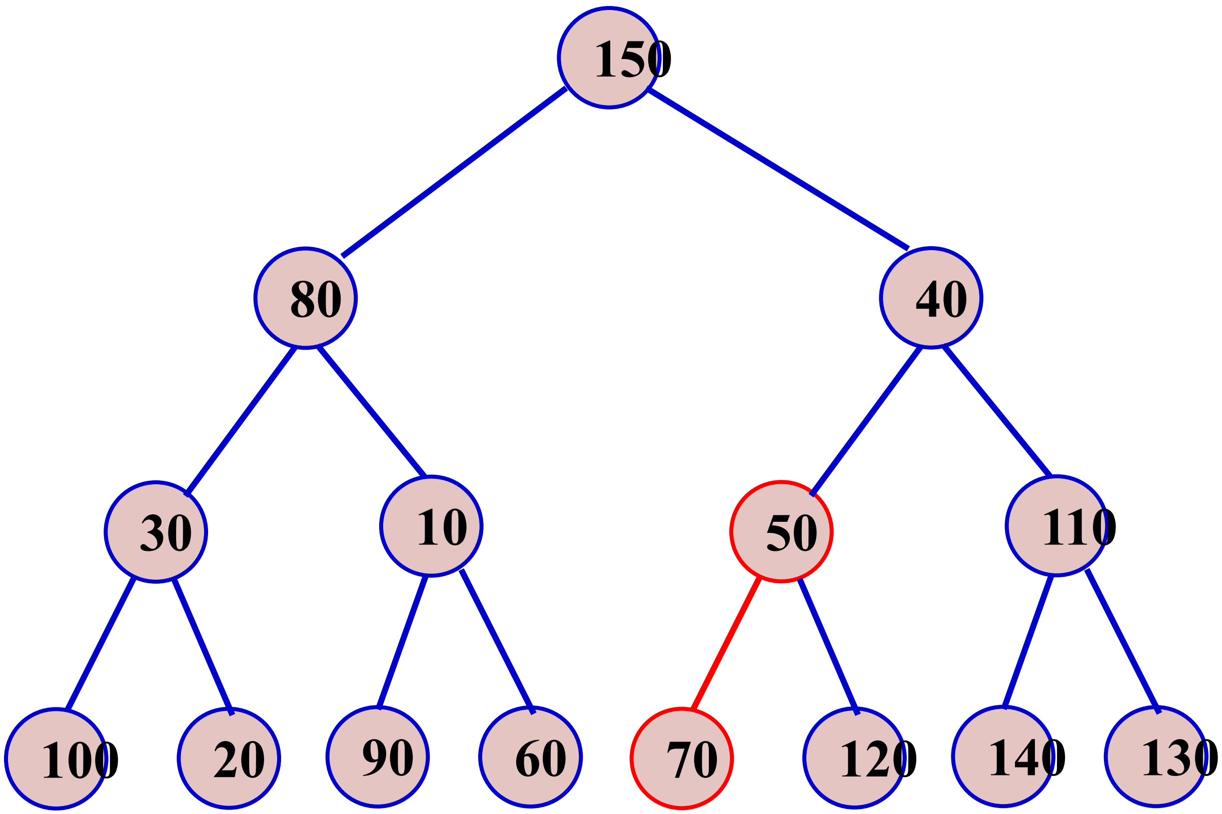

Heap after percolate_down(7)

# Build Heap



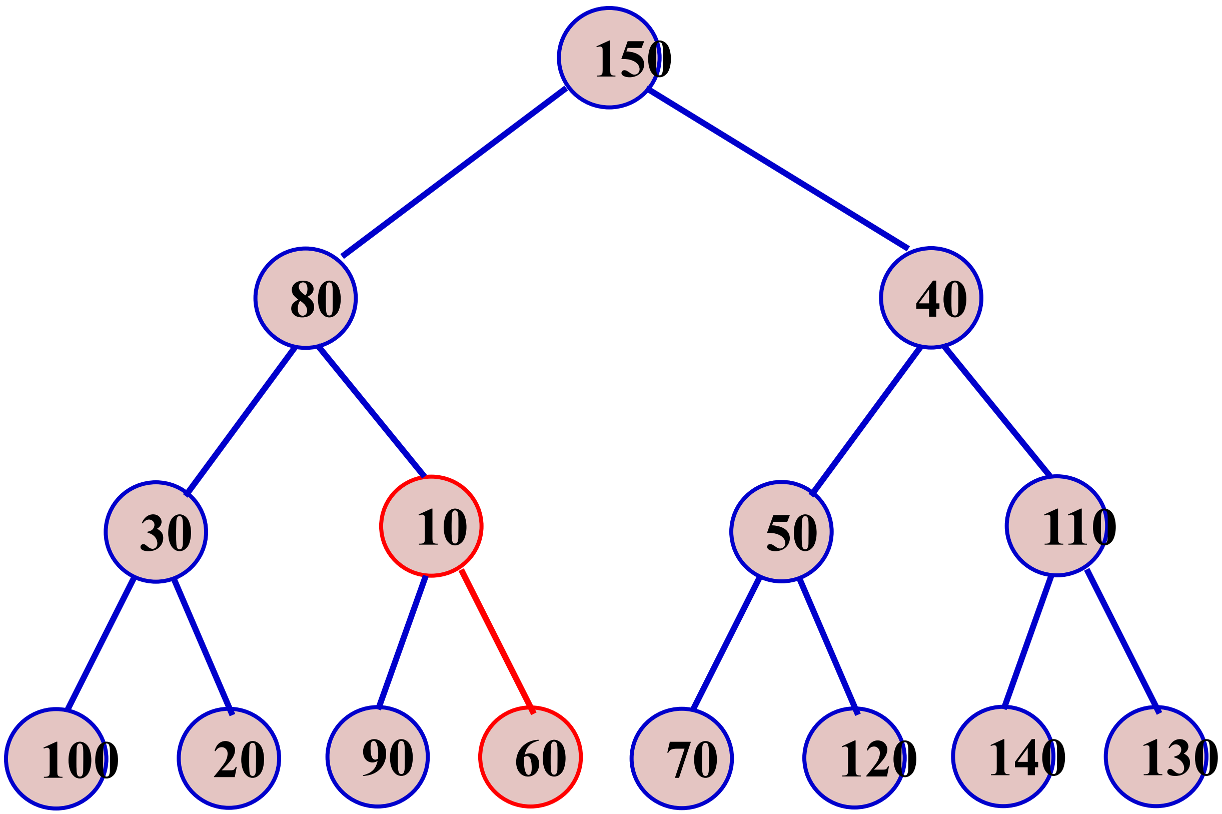

Heap after percolate_down(6)

# Build Heap



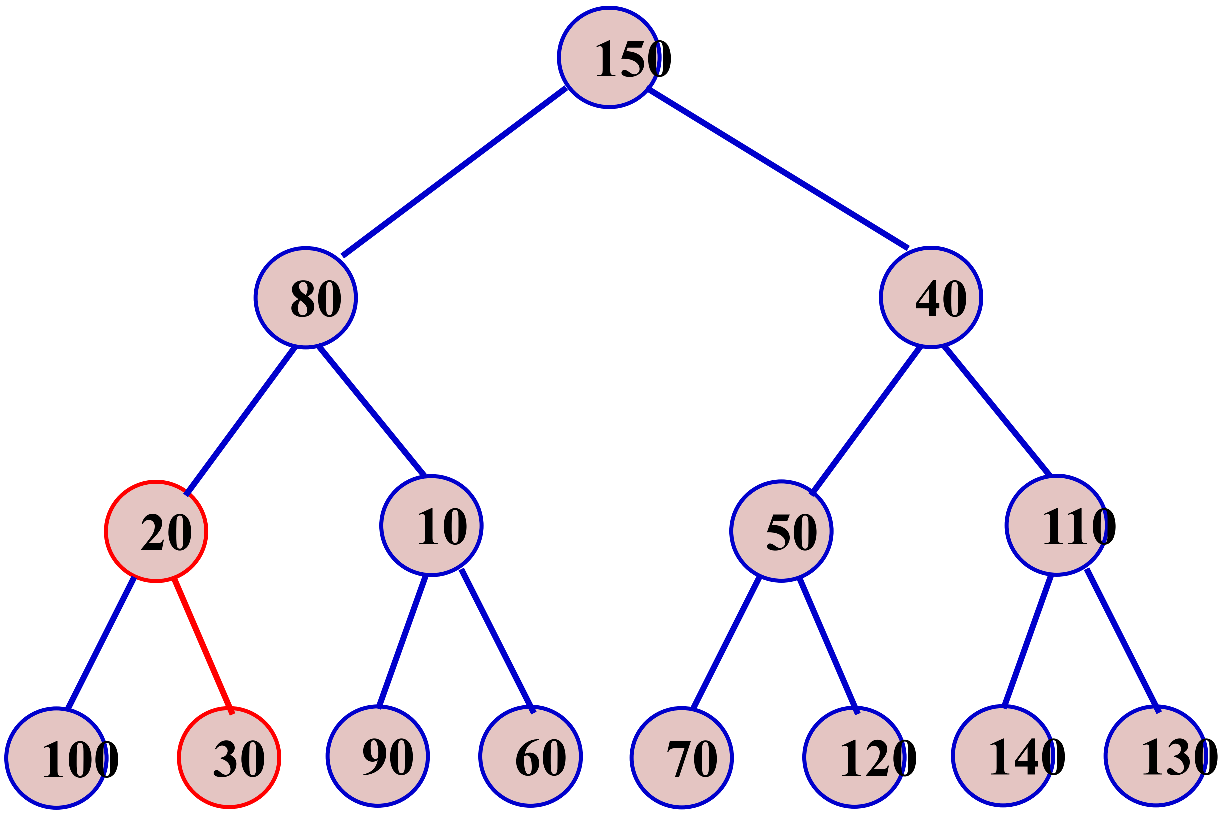

Heap after percolate_down(6)

# Build Heap



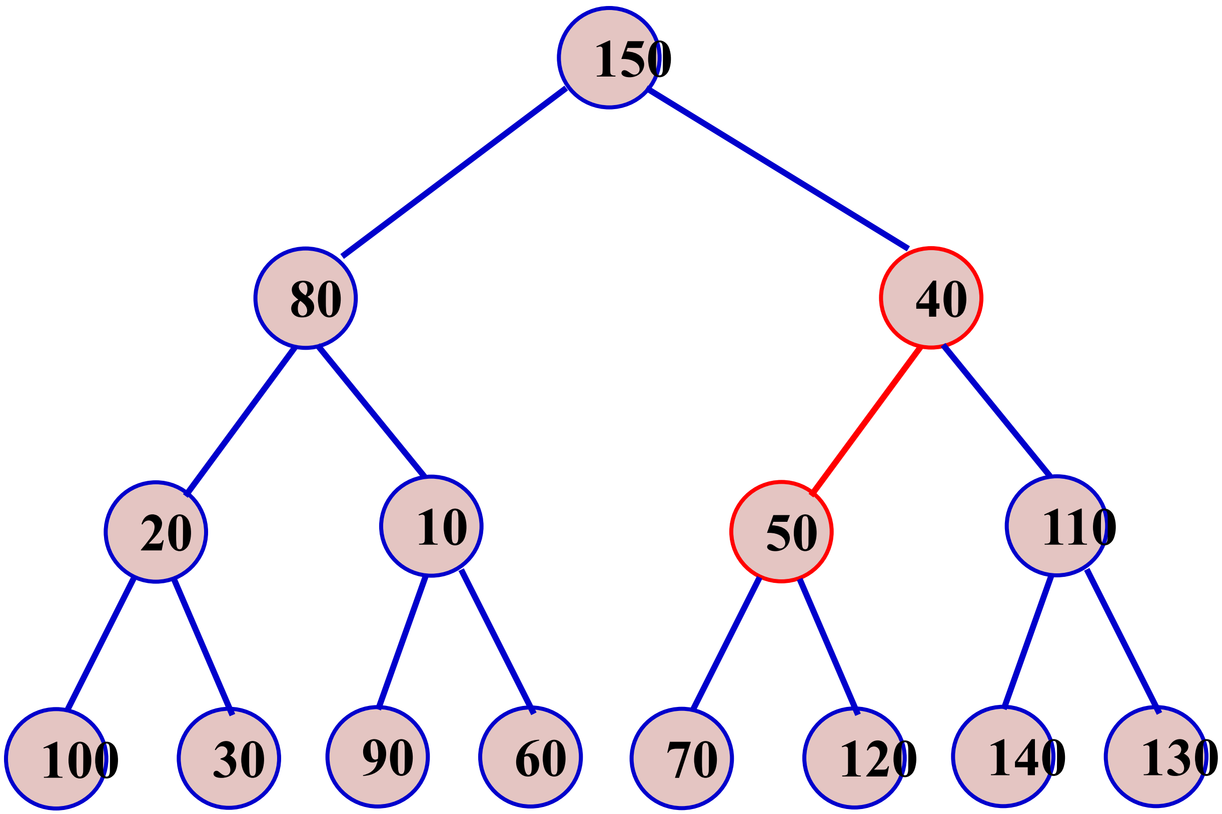

Heap after percolate_down(5)

# Build Heap



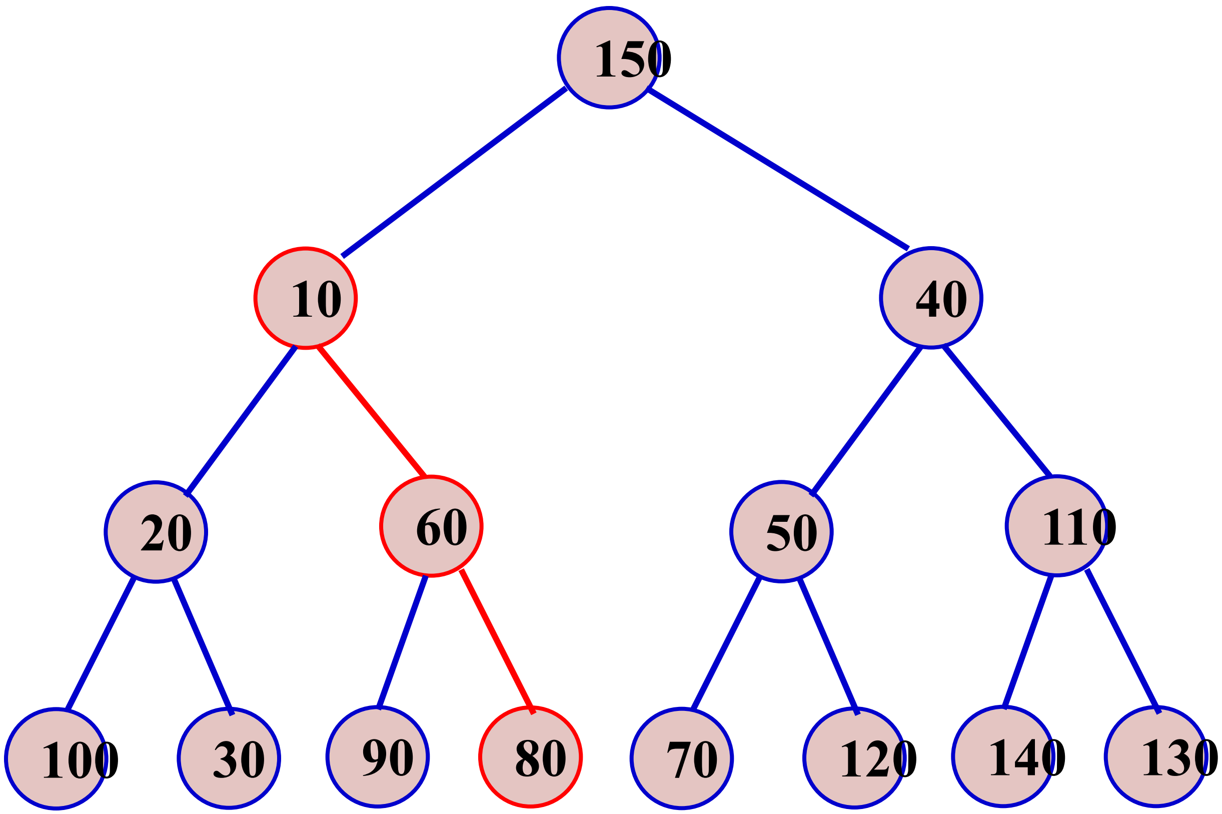

Heap after percolate_down(4)

# Build Heap



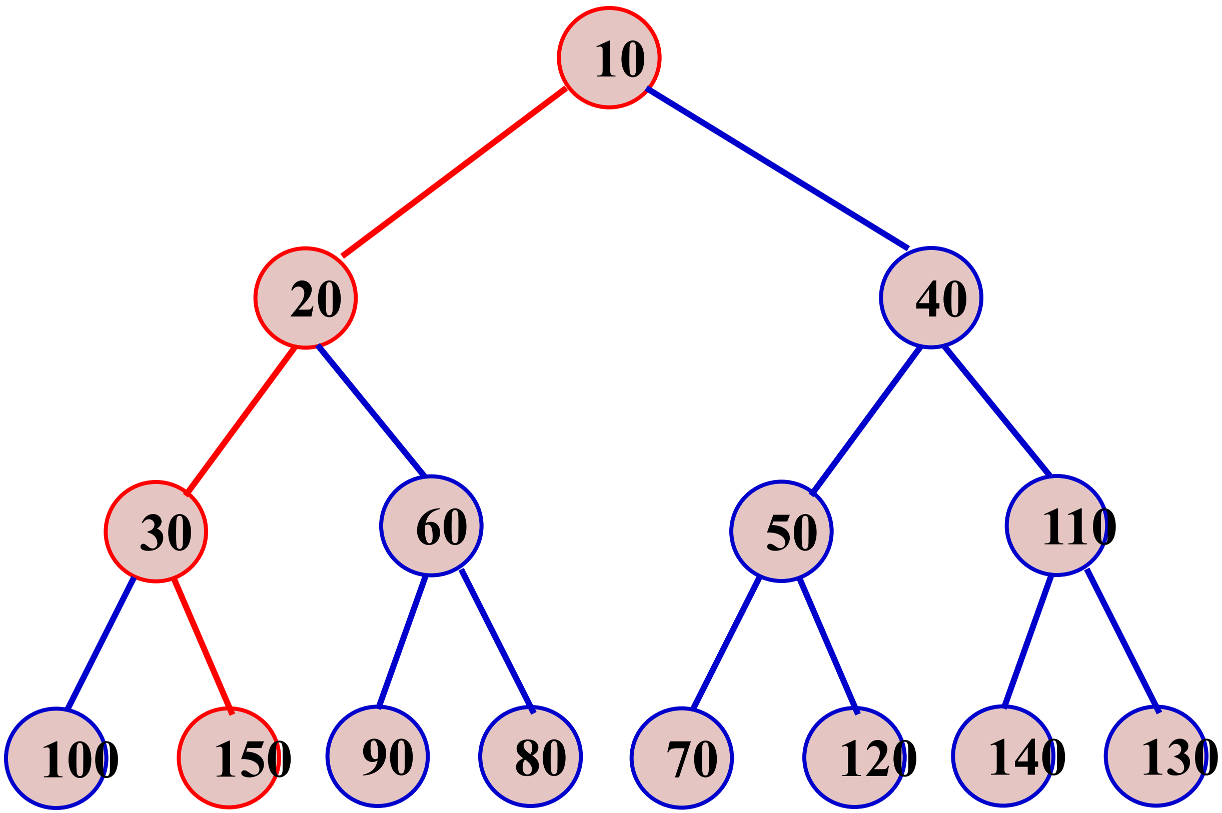

Heap after percolate_down(3)

# Build Heap



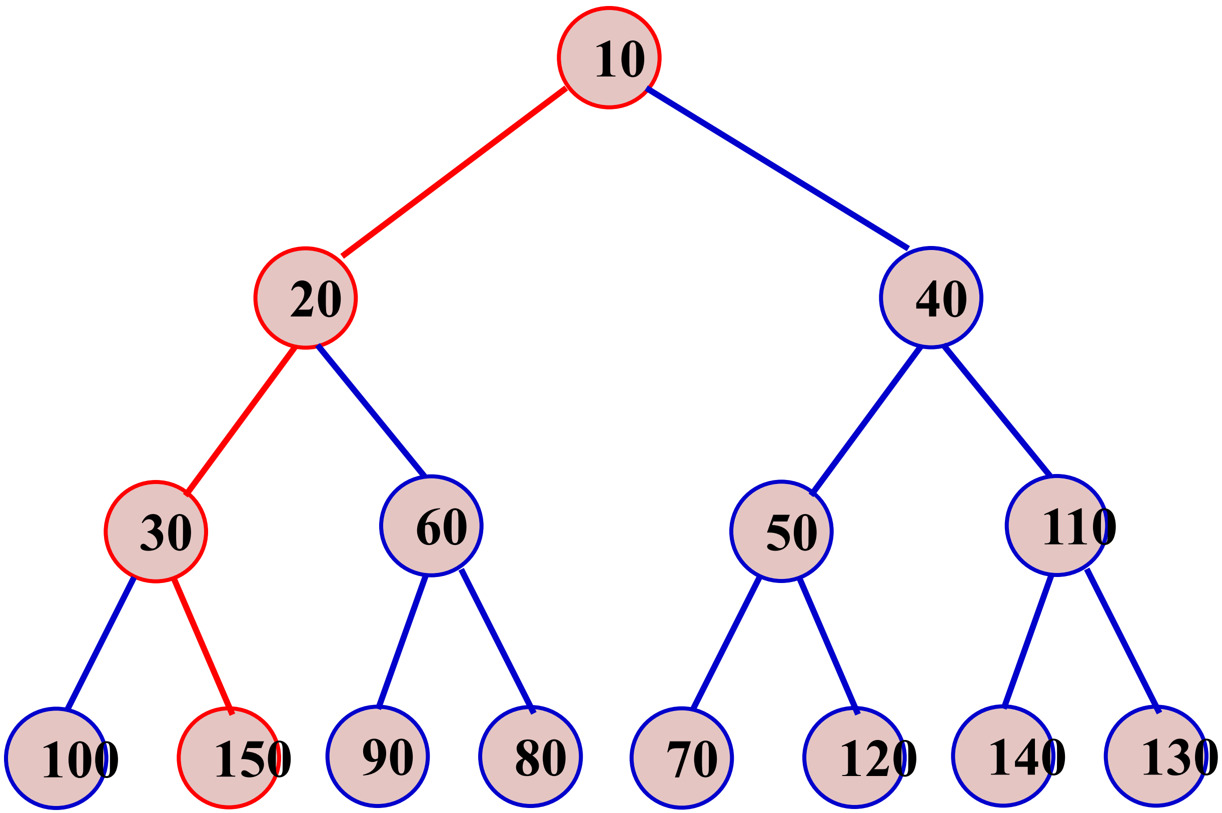

Heap after percolate_down(2)

# Build Heap




Heap after percolate_down(1)

# Build Heap




Heap after percolate_down(1)

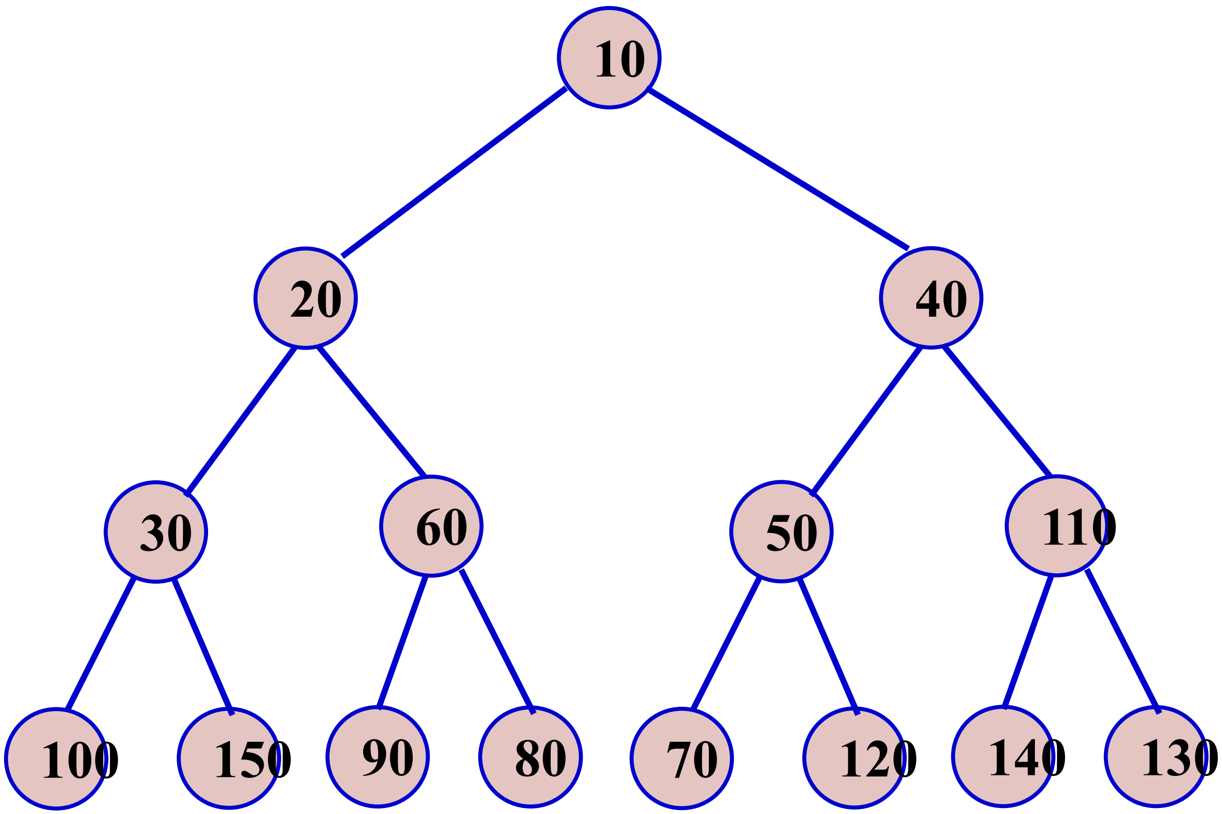
# Heap Operations: Decrease Key

- The ***decrease_key(x, del, H)*** operation lowers the value of the key at position ***x*** by a positive amount ***del***.
- Since this might violate the heap order, it must be fixed by a ***percolate up***.
- This operation could be useful to system administrators:
  - they can make their programs run with highest priority

# Heap Operations: Increase Key

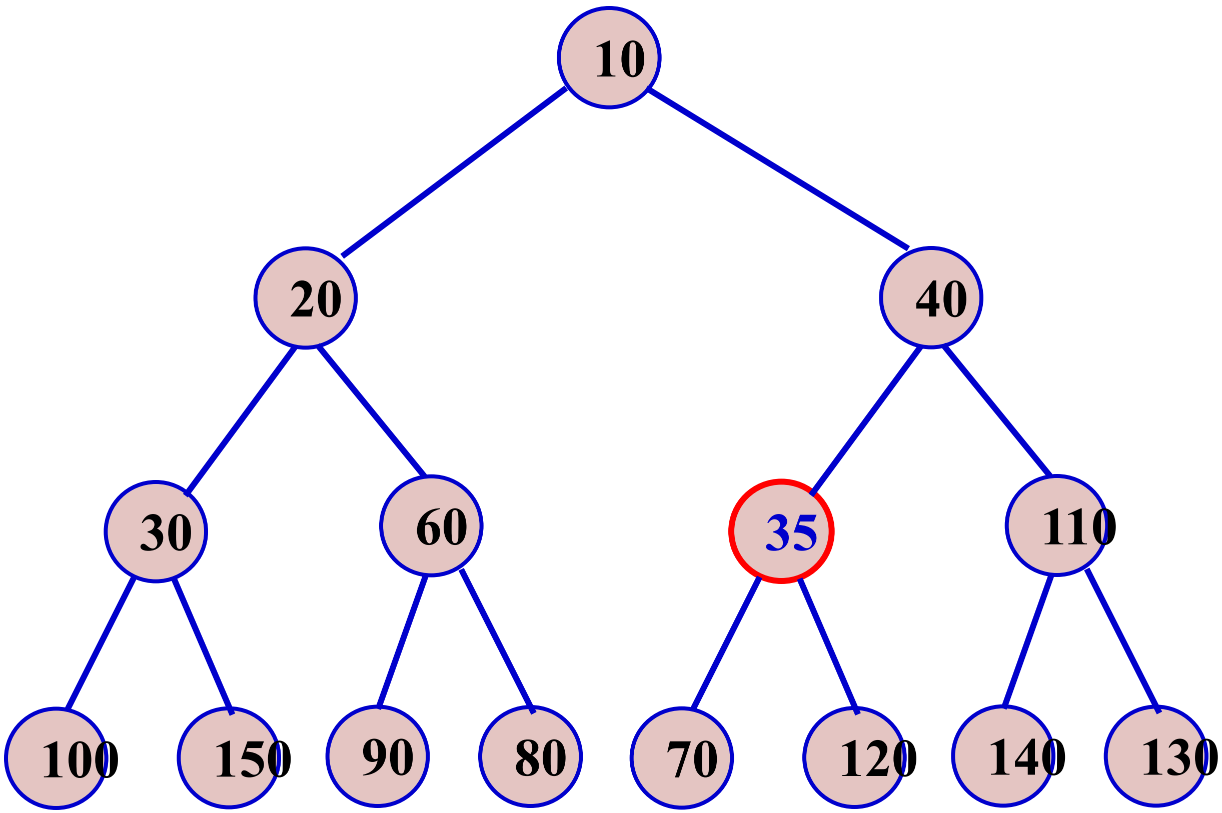
- The ***increase_key(x*, del , *H)*** *operation increases the value of the key at position* ***x*** *by a positive amount* ***del****.*
- *This is done with a* ***percolate down****.*
- *Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.*

# Heap Operations: Decrease Key




***decrease_key(6, 15, H)***

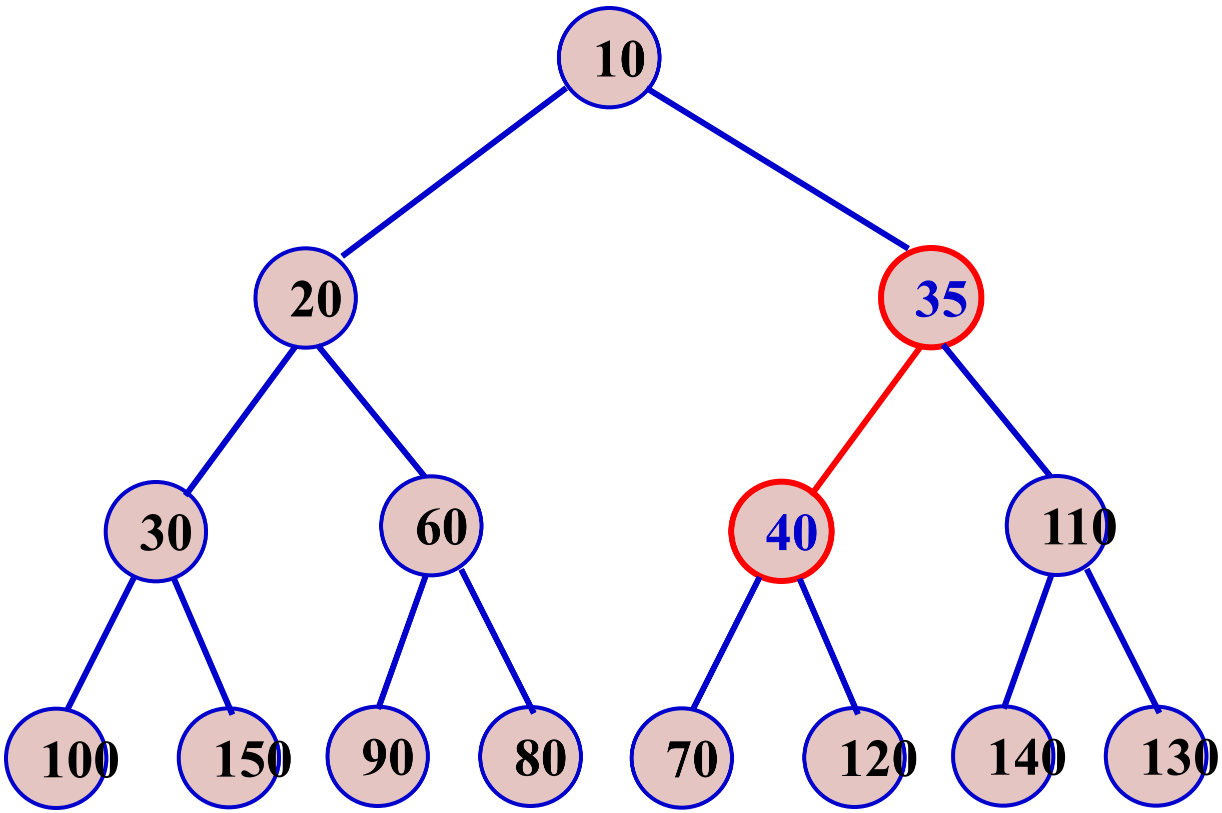
# Heap Operations: Decrease Key




**Heap decrease_key(6, 15)**

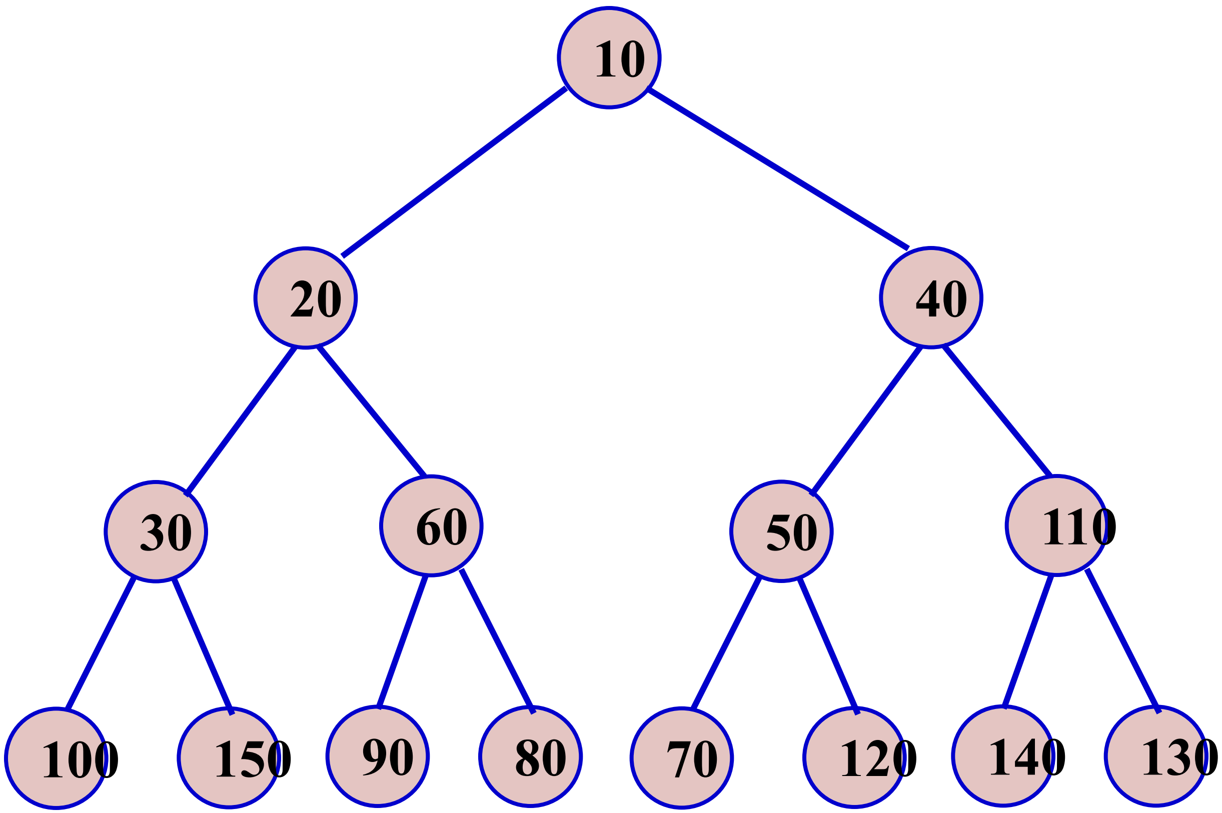
**percolate_up(6)**

# Heap Operations: Decrease Key



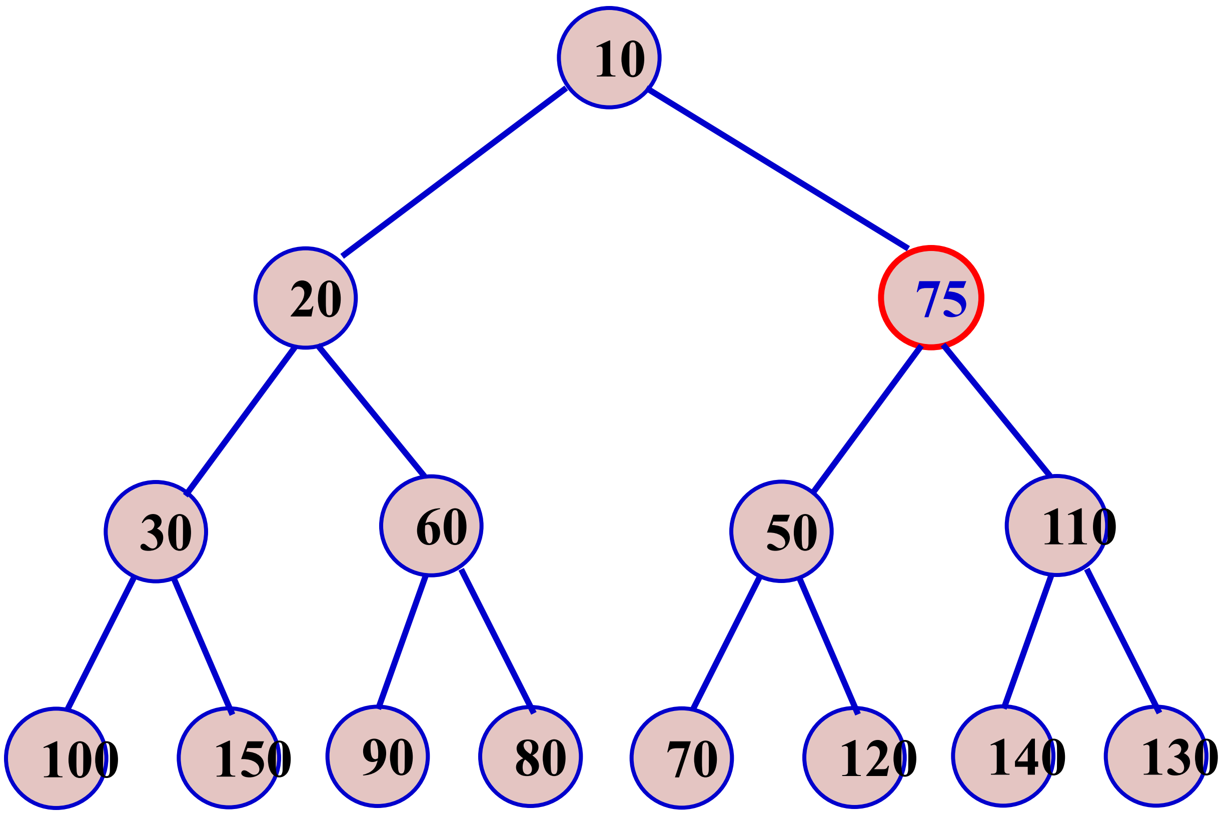

**Heap after percolate_up(6)**

# Heap Operations: Increase Key




***Increase_key(3, 35, H)***

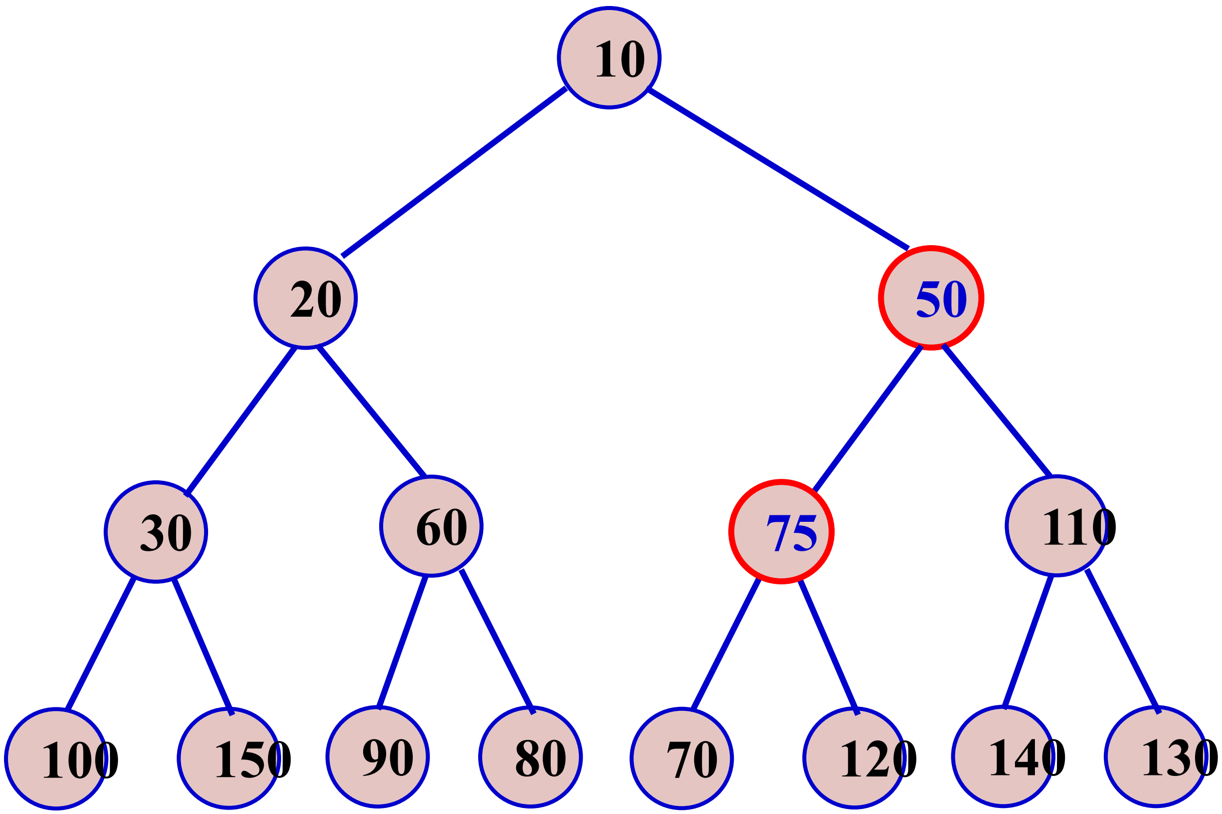
# Heap Operations: Increase Key




**Heap before percolate_down(3)**

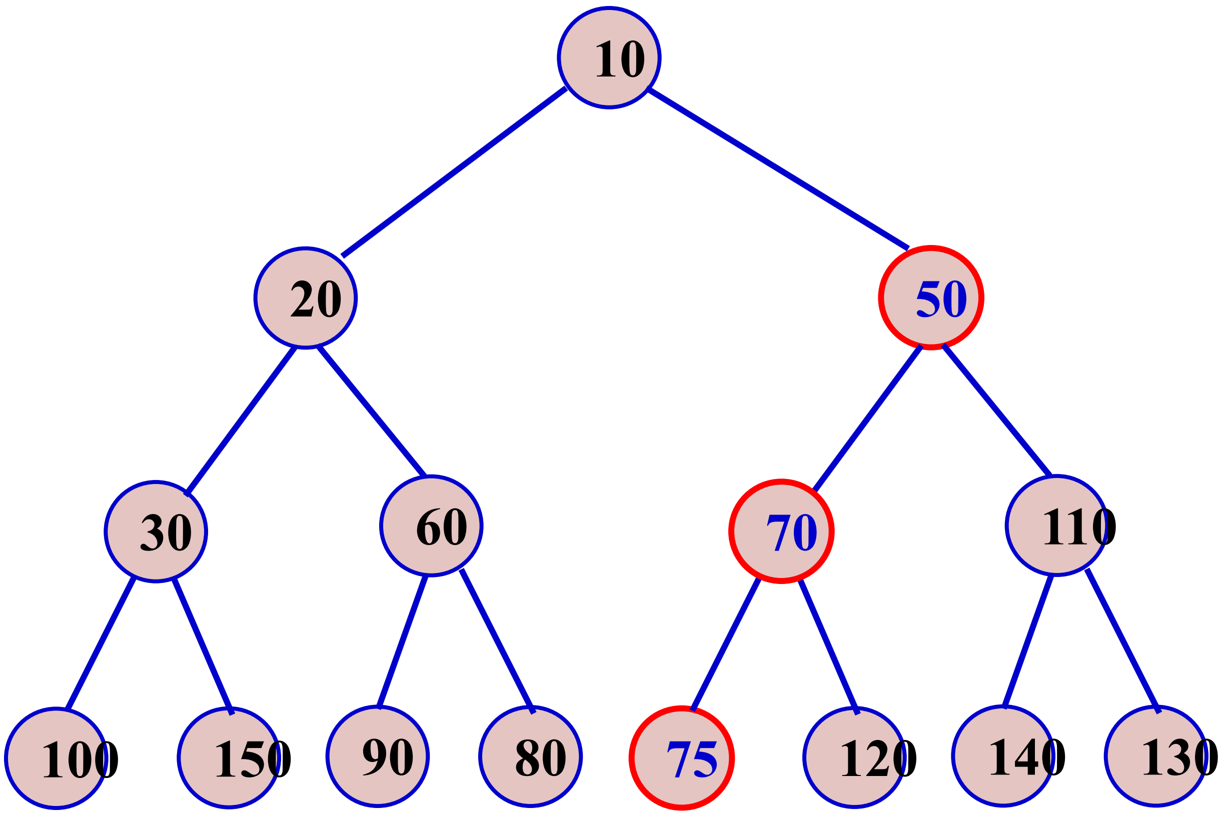
***Increase_key(3, 35, H)***

# Heap Operations: Increase Key




**percolate_down(3)** in action

# Heap Operations: Increase Key

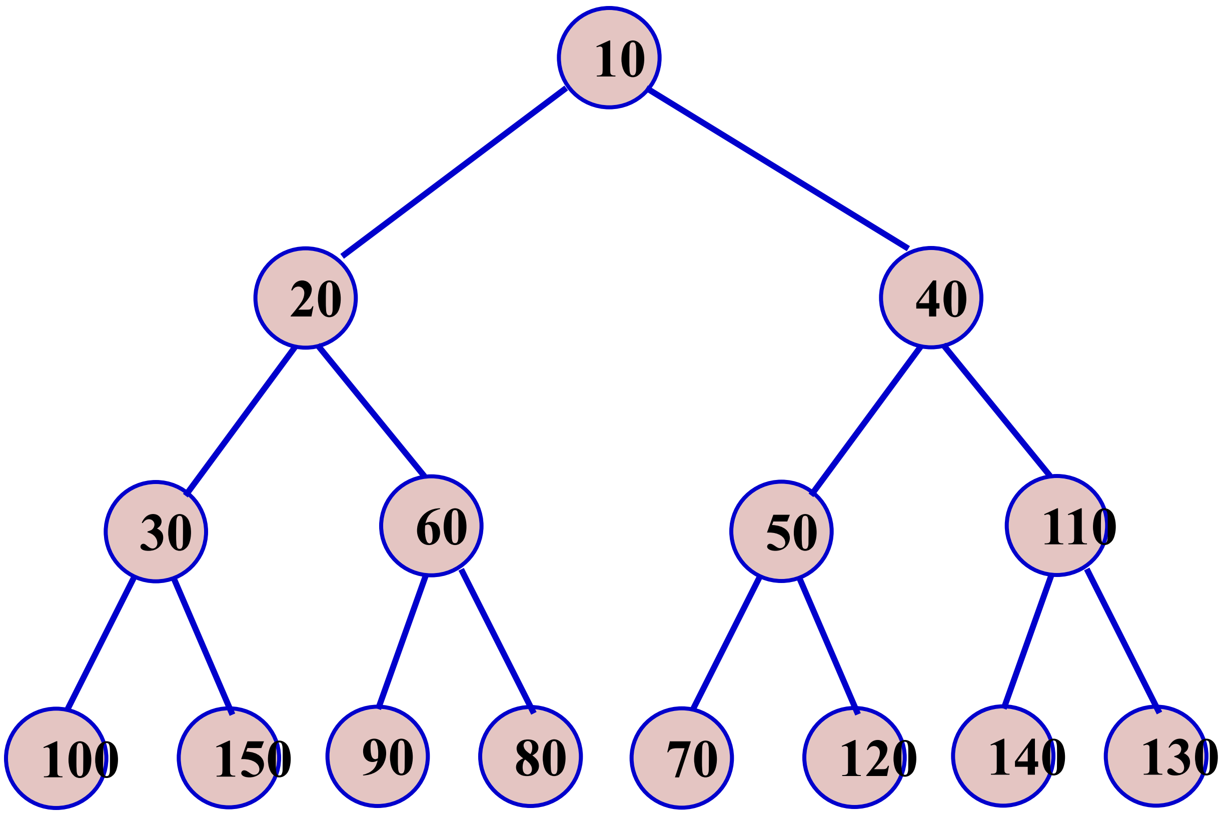



Heap after percolate_down(3)

# Heap Operations: Delete

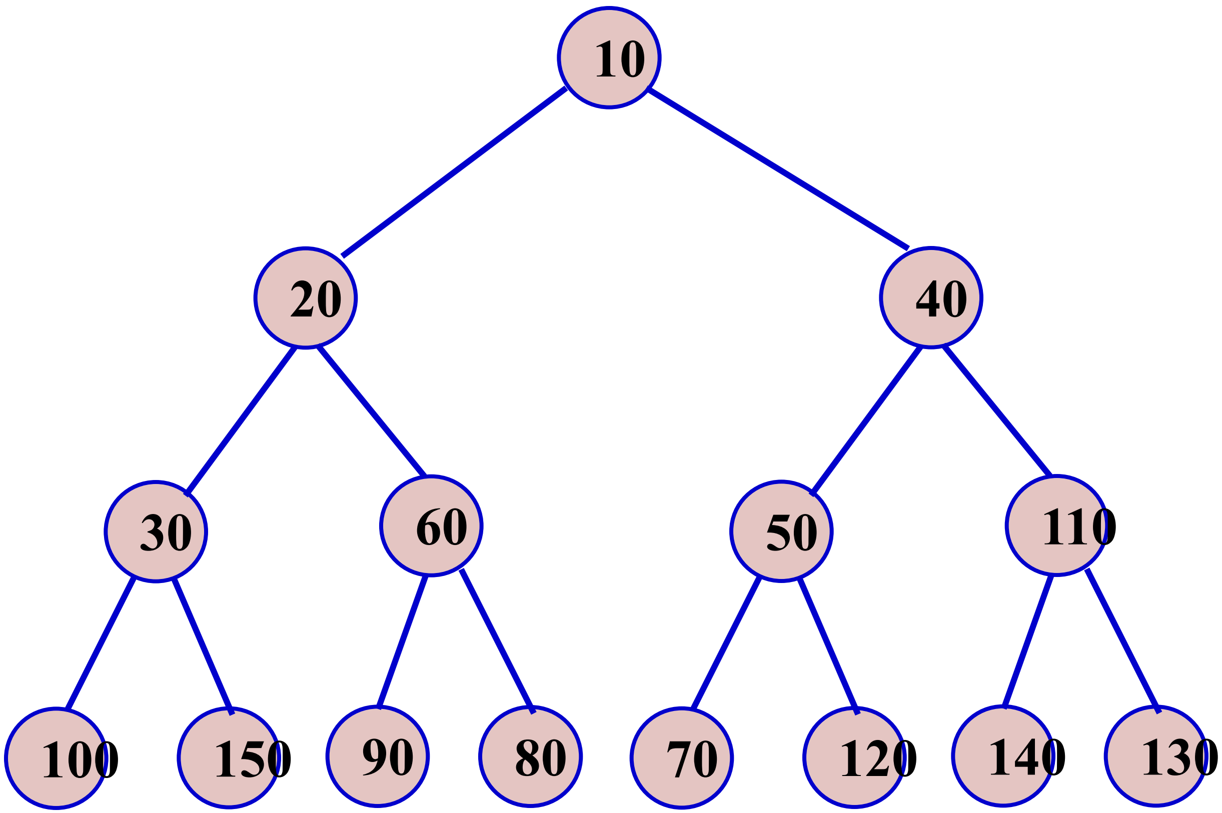
- *The* ***delete(x, H)*** *operation removes the node at position* ***x*** *from the heap.*
- *This is done by first performing* ***decrease_key(x, ∞ , H)***
- *and then performing* ***delete_min (H)****.*
- *When a process is terminated by a user (instead of finishing normally), it must be removed from the priority queue.*

# Heap Operations: Delete



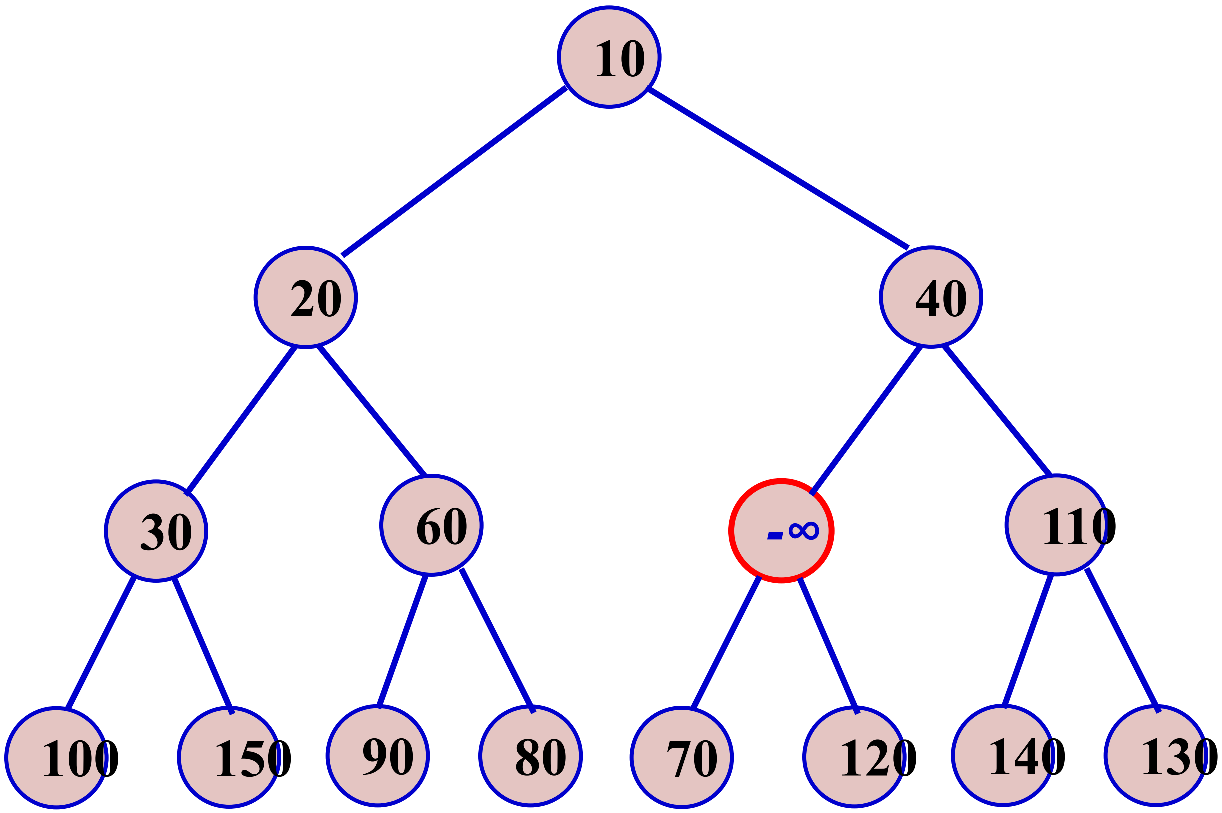

***delete(6, H)***

# Heap Operations: Delete




***delete(6, H)*** *in action:* ***decrease_key(6, ∞ , H)***

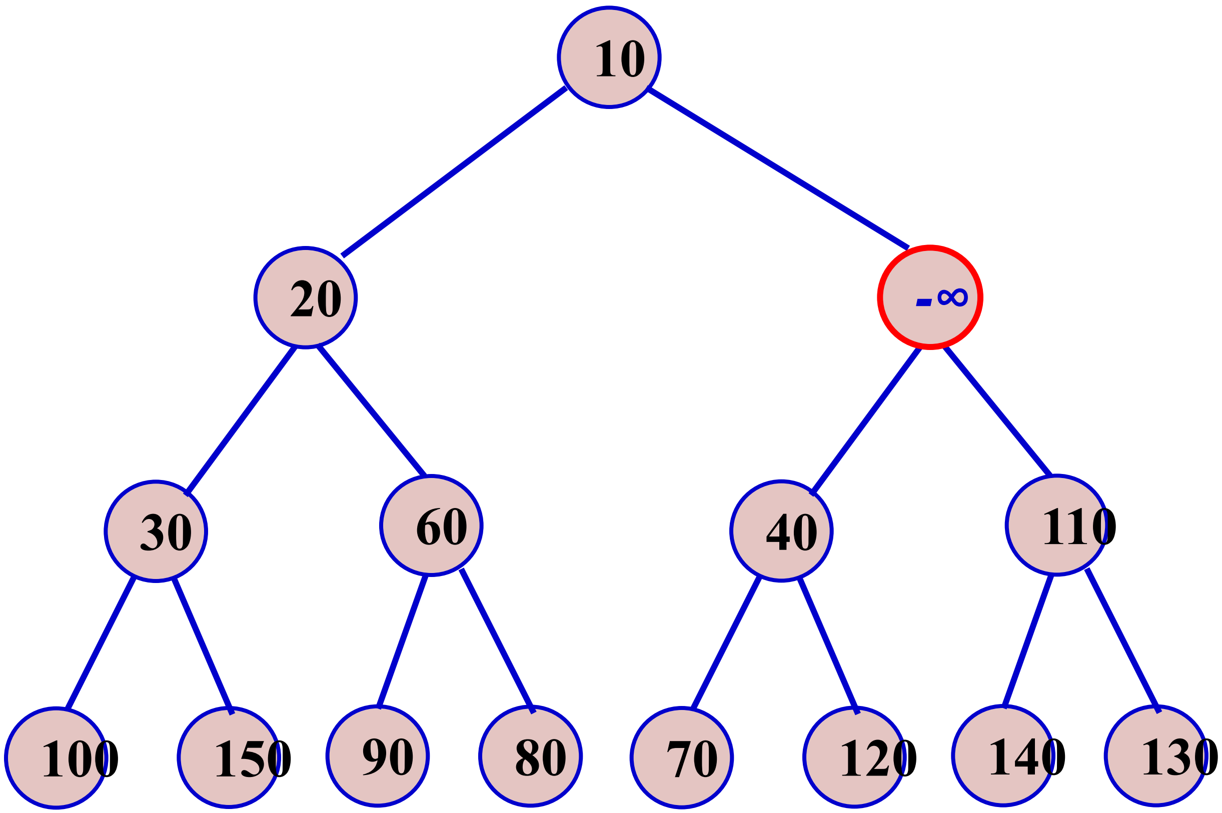
# Heap Operations: Delete




***delete(6, H)*** *in action:* ***decrease_key(6, ∞ , H)***

➔ **percolate_up(6)**

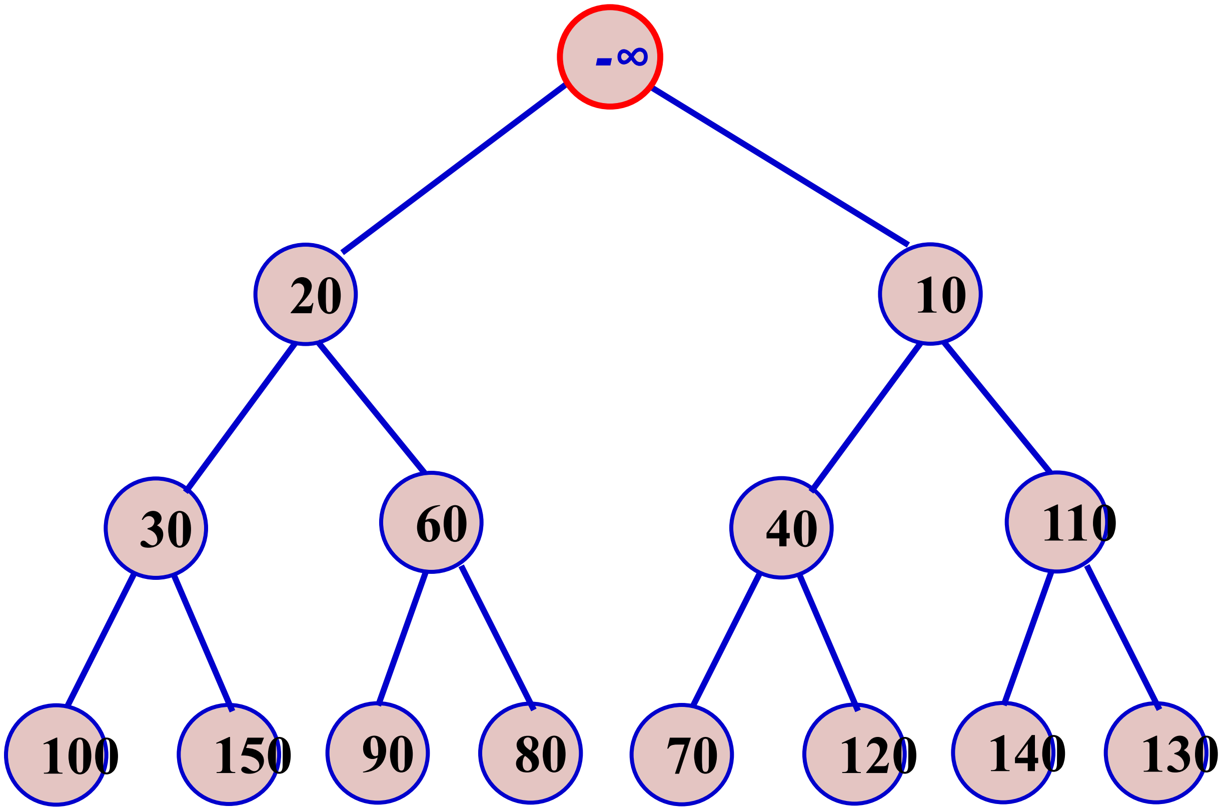
# Heap Operations: Delete




***delete(6, H)*** *in action:* ***decrease_key(6, ∞ , H)***

➔ **percolate_up(6)**

# Heap Operations: Delete

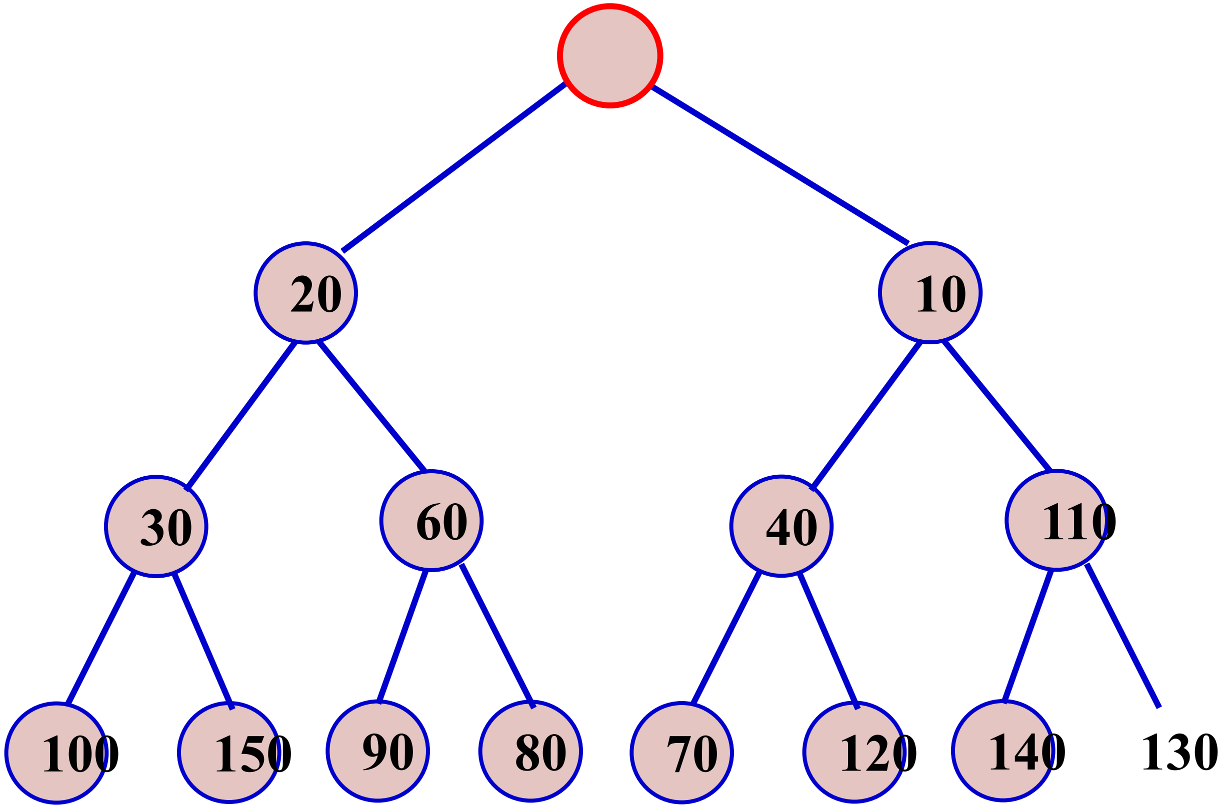



***delete(6, H)*** *in action:* ***decrease_key(6, ∞ , H)***

➔ **percolate_up(6)**

**delete_min(H)**

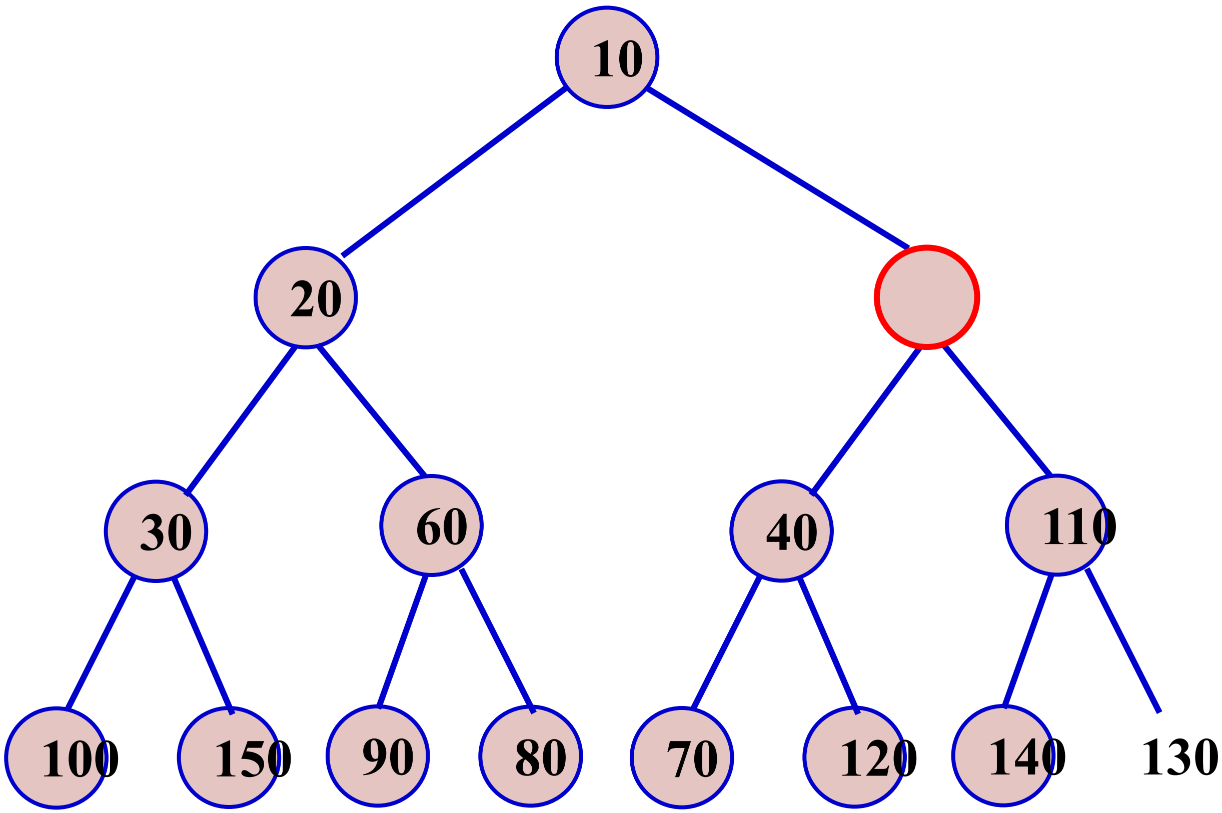
# Heap Operations: Delete




***delete(6, H)*** *in action:* ***decrease_key(6, ∞ , H)***

➔ ***delete_min (H)***

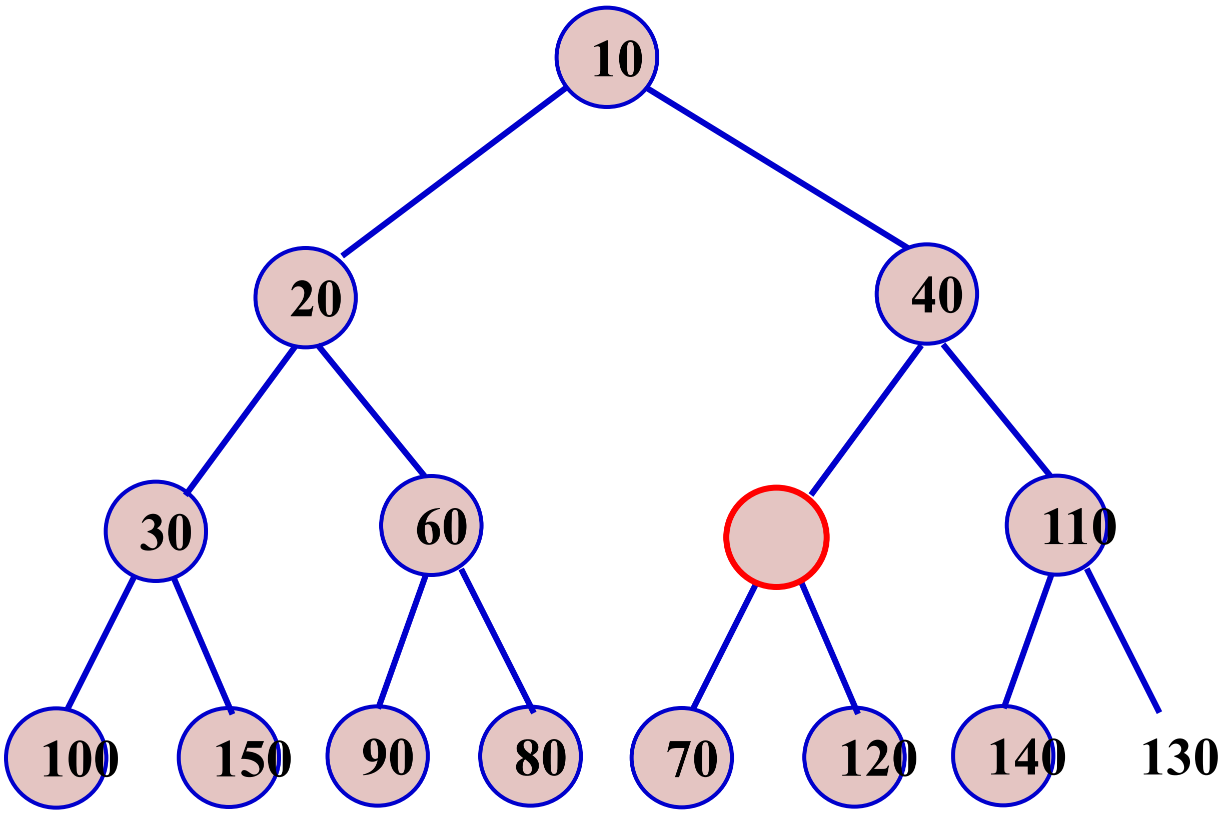
# Heap Operations: Delete




***delete(6, H)*** *in action:* ***decrease_key(6, ∞ , H)***

➔ ***delete_min (H)***

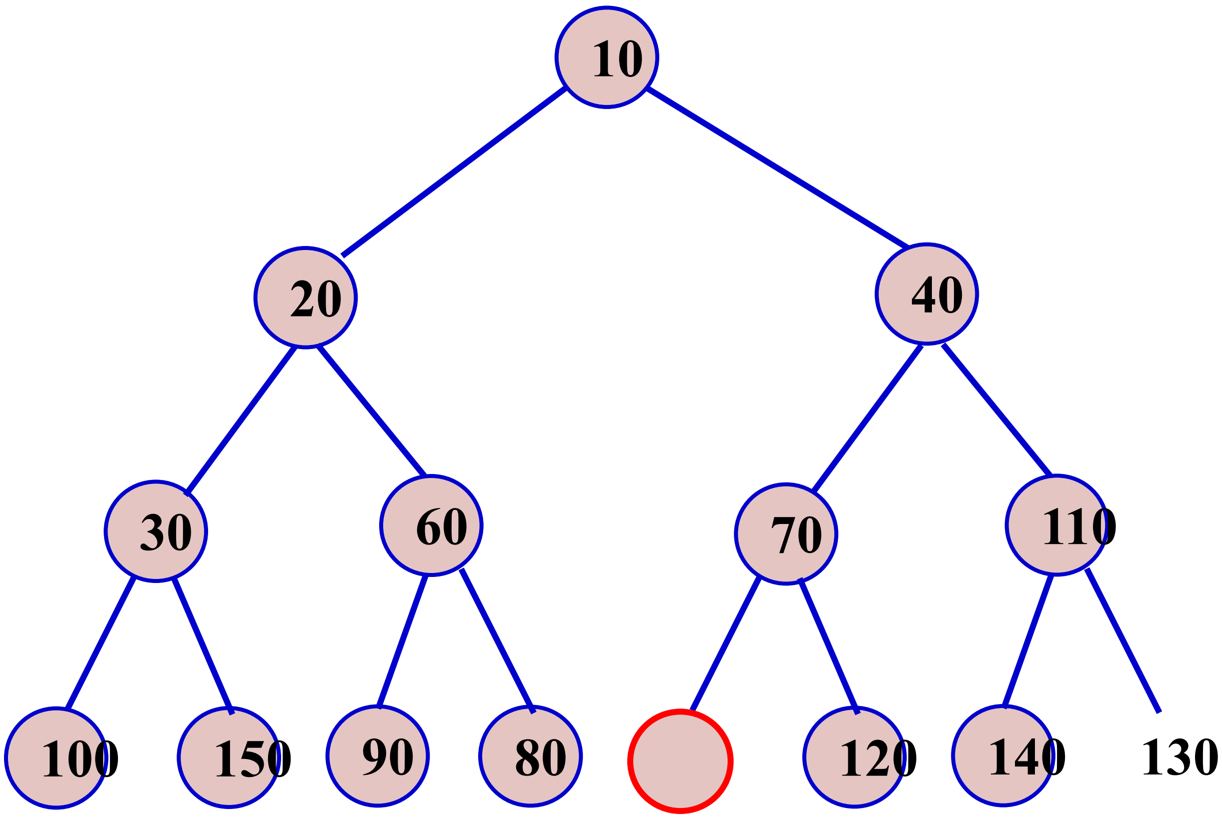
# Heap Operations: Delete




***delete(6, H)*** *in action:* ***decrease_key(6, ∞ , H)***

➔ ***delete_min (H)***

# Heap Operations: Delete




***delete(6, H)*** *in action:* ***decrease_key(6, ∞ , H)***

➔ ***delete_min (H)***

# Questions?

zahmaad.github.io