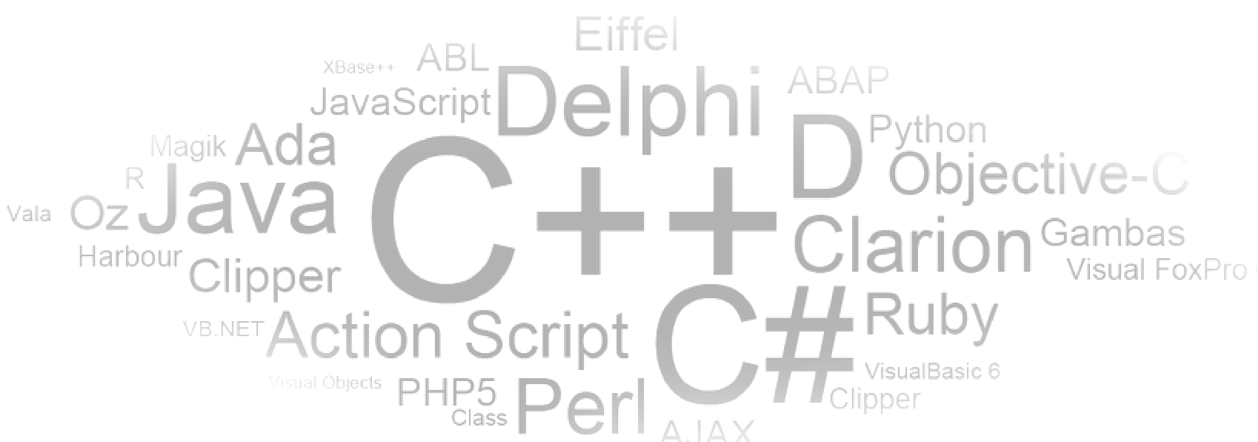
CIS 351-Data Structure-Heap
April 16, 2020

Dr. Farzana Rahman

Syracuse University



What are Heaps Useful for?

- To implement priority queues
- Priority queue = a queue where all elements have a “priority” associated with them
- Remove in a priority queue removes the element with the smallest priority
  - insert
  - removeMin

Heaps

- A Heap is a Binary Tree *H* that stores a collection of keys at its internal nodes and that satisfies two additional properties:

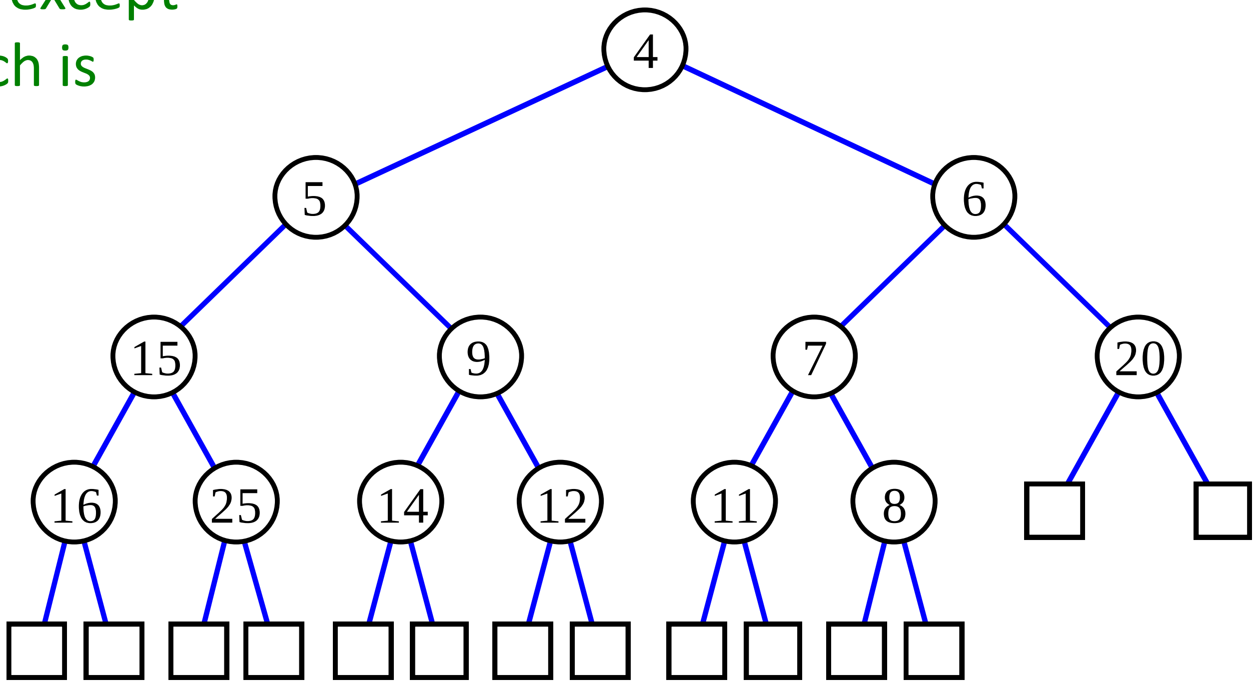
  -**1)Heap Order** Property

  -2)***Structural*** Property

Heaps

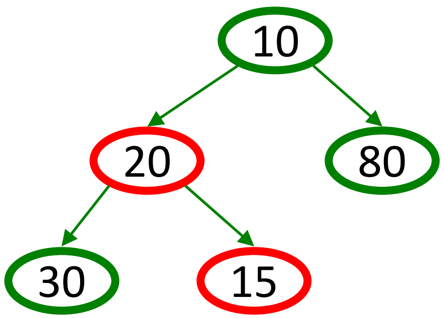
- A *heap* is a binary tree T that stores a key pairs at its internal nodes

- It satisfies two properties:
  - **key(parent) ≤ key(child)**
  - all levels are full, except the last one, which is left-filled

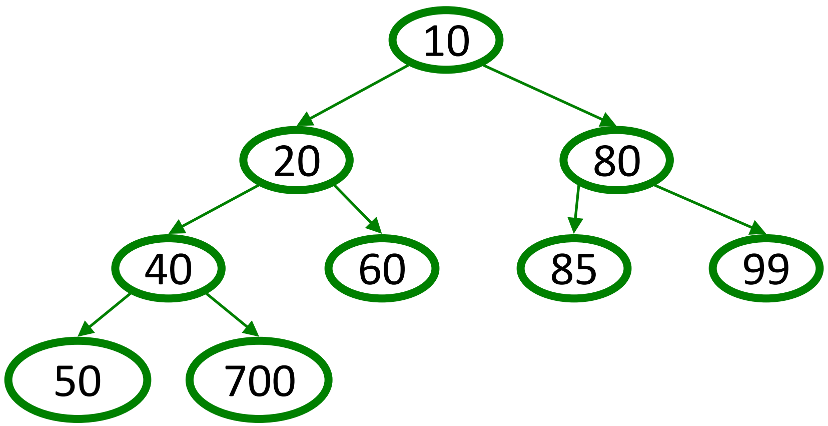


Heap Order Property

Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.



not a heap



Heap-Order Property

- In a heap, for every node X with parent P, the key in P is smaller than or equal to the key in X.
- Thus the minimum element is always at the root.
  - Thus we get the extra operation findMin in constant time.
- A **max heap** supports access of the maximum element instead of the minimum, by changing the heap property slightly.

Heap order - two types

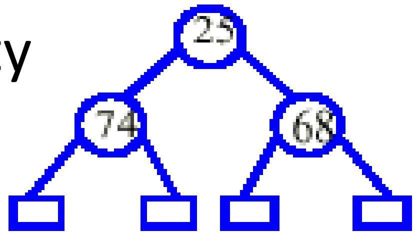
- the ***min-heap property***: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

- the ***max-heap property***: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

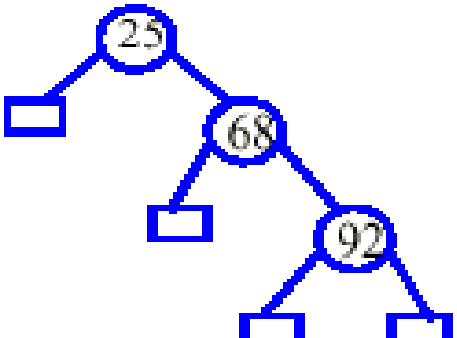
Heap Structure Property

- Complete Binary Tree (Structural): A Binary Tree *T* is complete if each level but the last is full, and, in the last level, all of the internal nodes are to the left of the external nodes.

- A structure fulfilling the Structural Property



- A structure which fails to fulfill the Structural Property

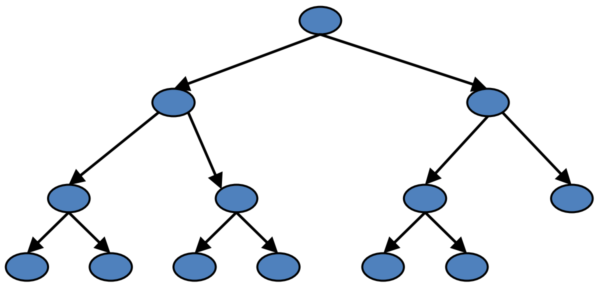


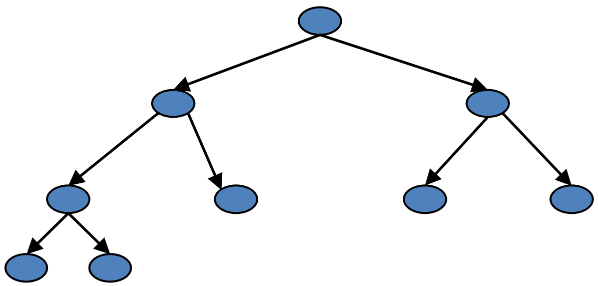
Heap Structure Property

- A binary heap is a ***complete*** binary tree.

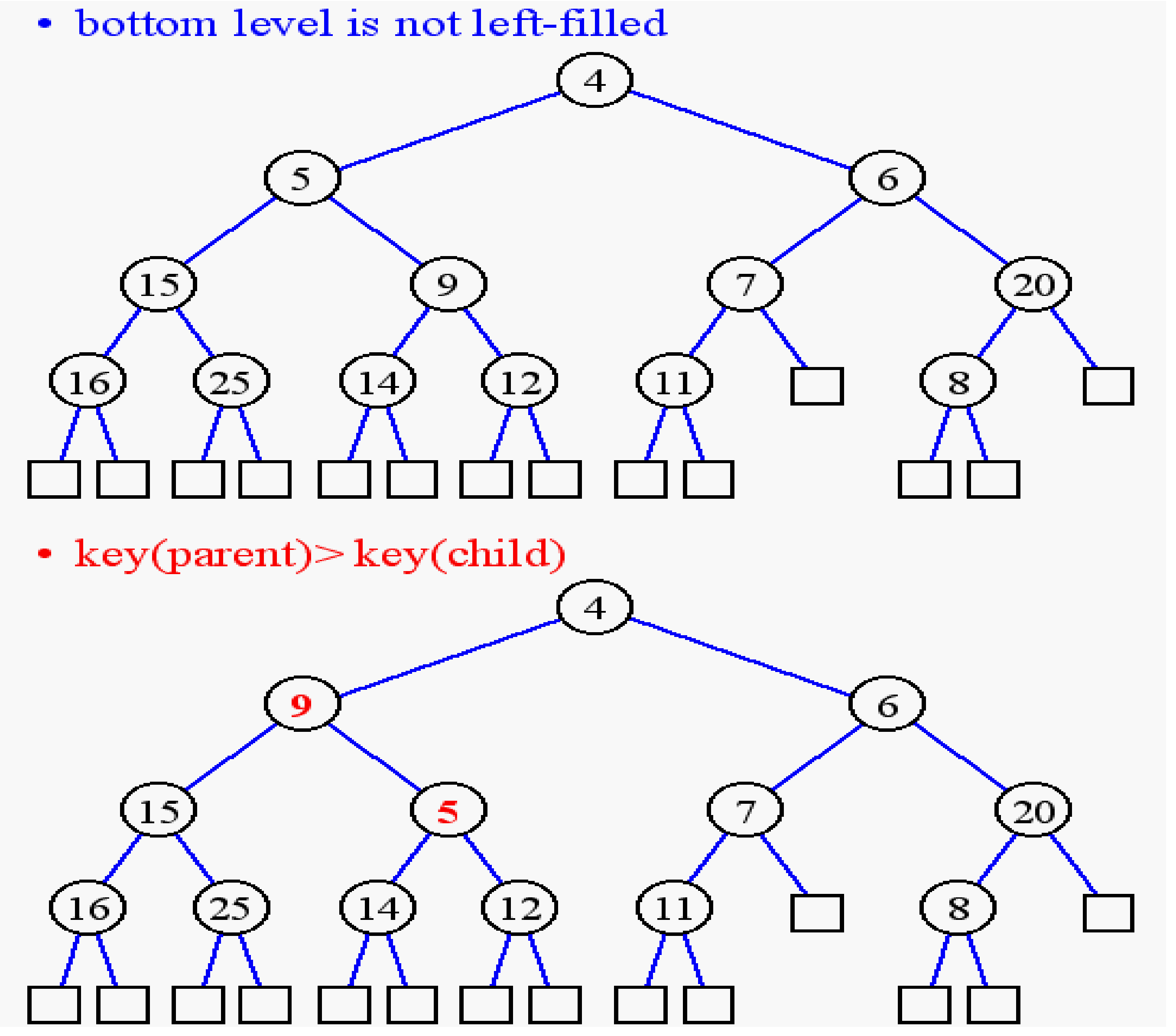
Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:

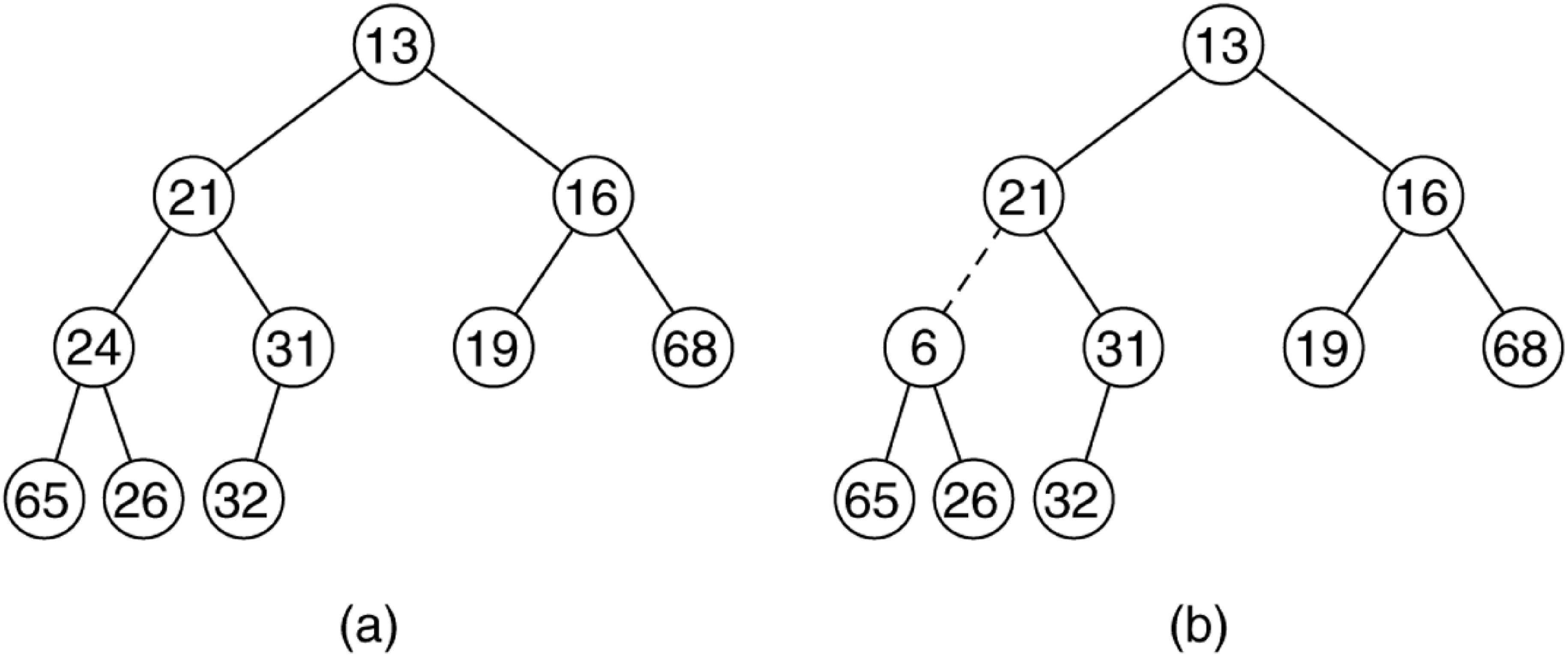




Heap or Not a Heap?



Two complete trees: (a) a heap (b) not a heap

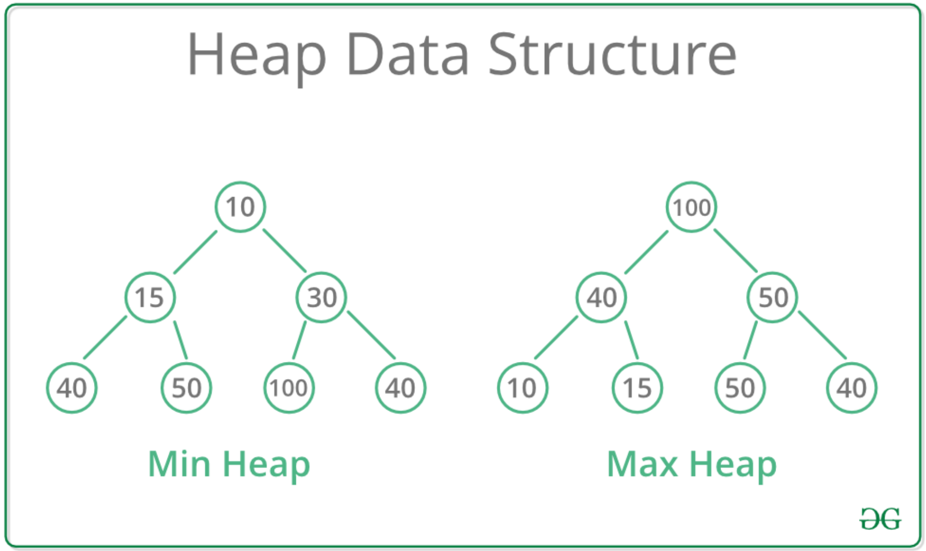


(a) (b)

Heap types

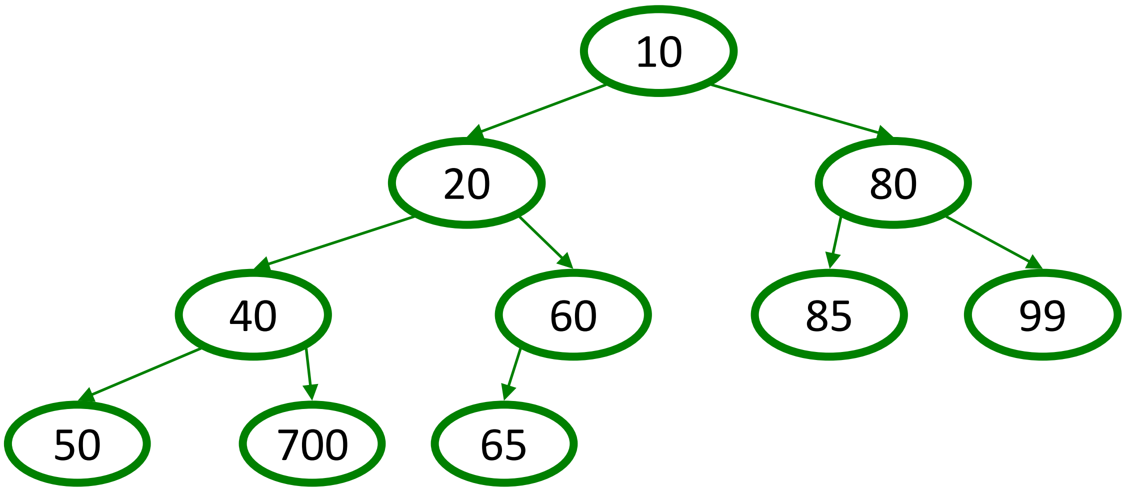
Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



Heap Operations

- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.



Heap – Insert(val)

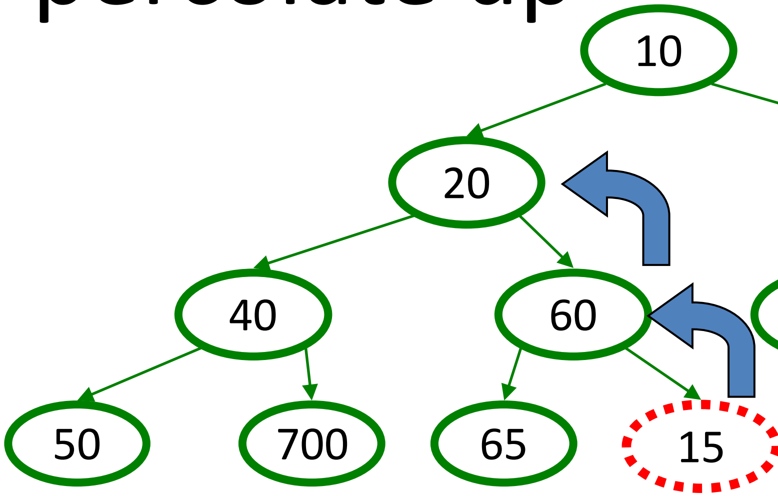
Basic Idea:

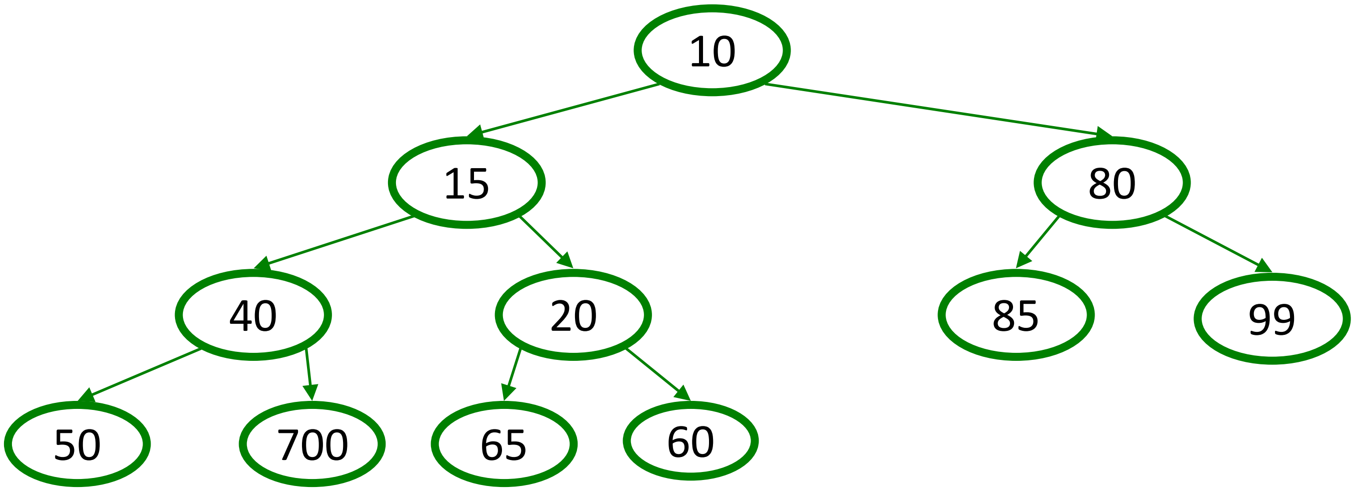
1. Put val at “next” leaf position
2. Percolate up by repeatedly exchanging node until no longer needed

Inserting in a heap: Percolate?

- 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: percolate up





- ***Upheap*** checks if the new node is smaller than its parent. If so, it switches the two
- ***Upheap*** continues up the tree

DeleteMin in Min-heaps

- The minimum value in a min-heap is at the root!
- To delete the min, you can't just remove the data value of the root, because every node must hold a key
- Instead, take the last node from the heap, move its key to the root, and delete that last node
- But now, the tree is no longer a heap (still almost complete, but the root key value may no longer be ≤ the keys of its children

DeleteMin in Min-heaps

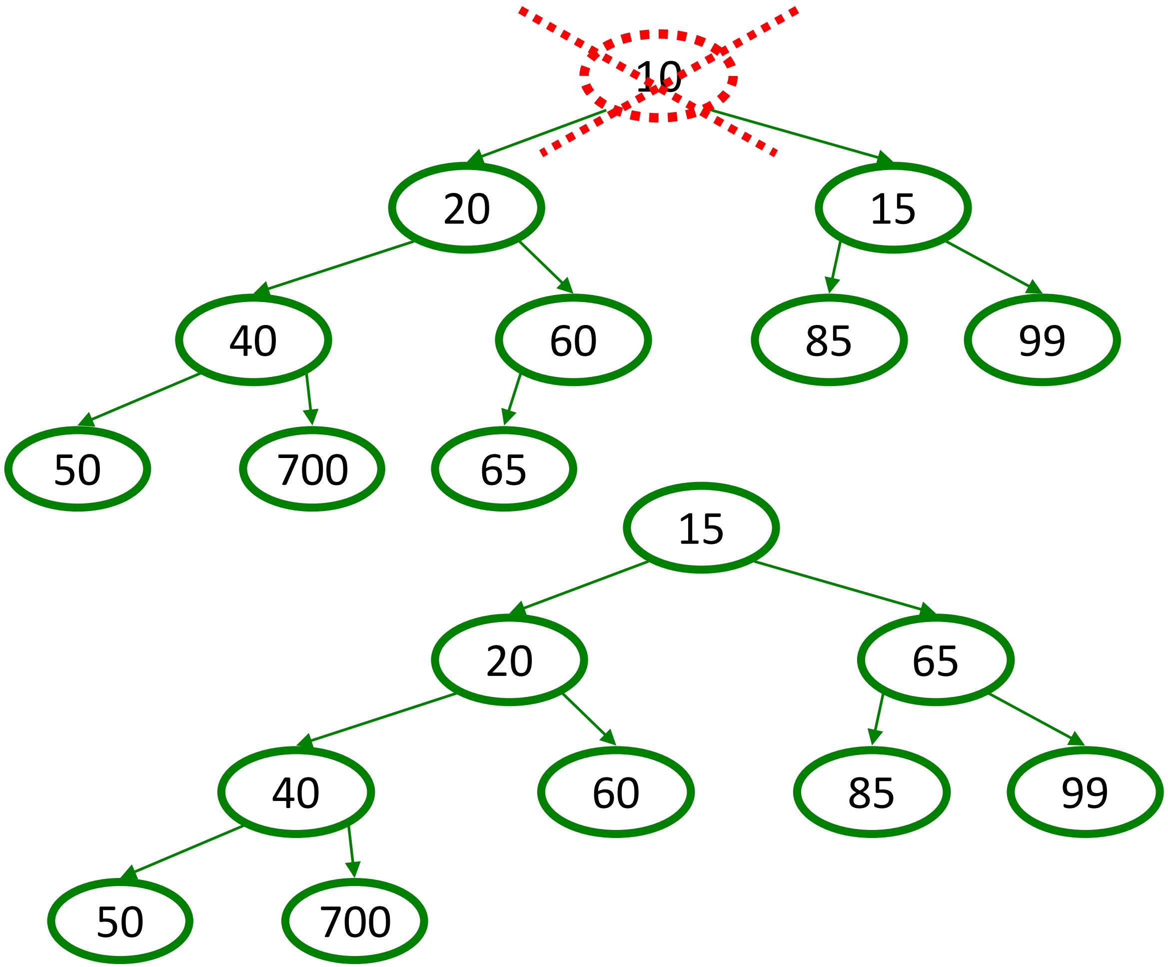
- Remove the minimum; a hole is created at the root.
- The last element X must move somewhere in the heap.
  - If X can be placed in the hole then we are done.
  - Otherwise,
    - We slide the smaller of the hole's children into the hole, thus pushing the hole one level down.
    - We repeat this until X can be placed in the hole.

Heap – Deletemin

Basic Idea:

1. Remove root (that is always the min!)
2. Put “last” leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

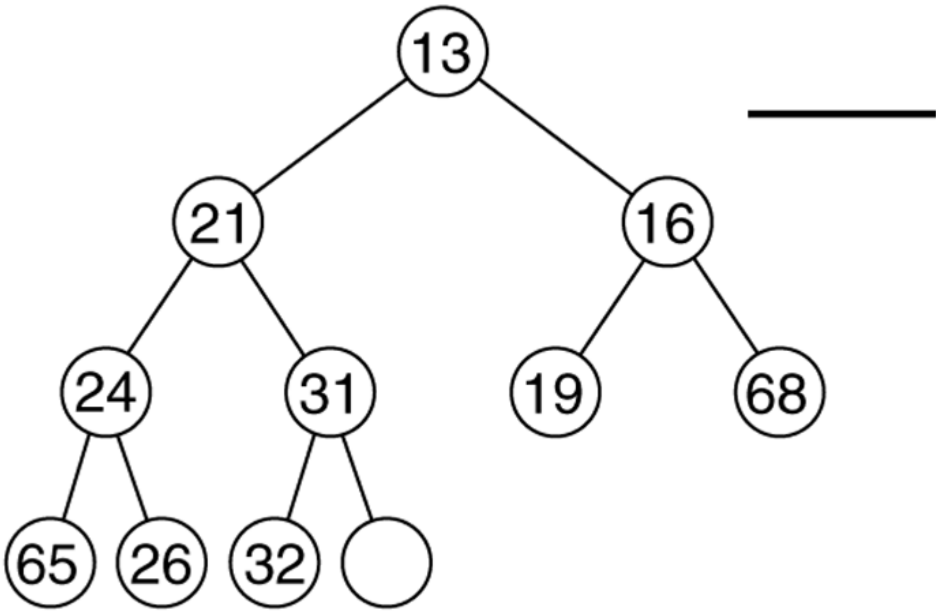
DeleteMin: percolate down



Downheap compares the parent with the smallest child. If the child is smaller, it switches the two.

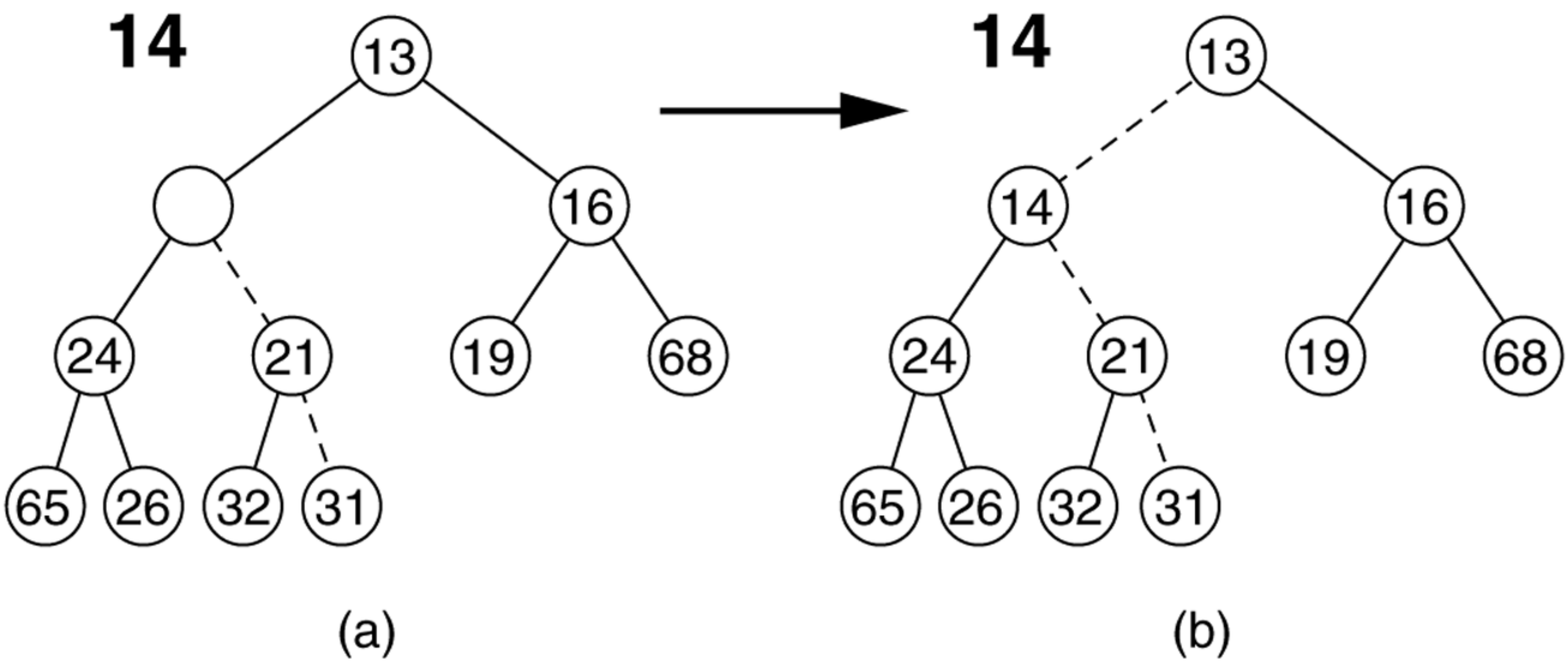
Examples

Attempt to insert 14, creating the hole and bubbling the hole up

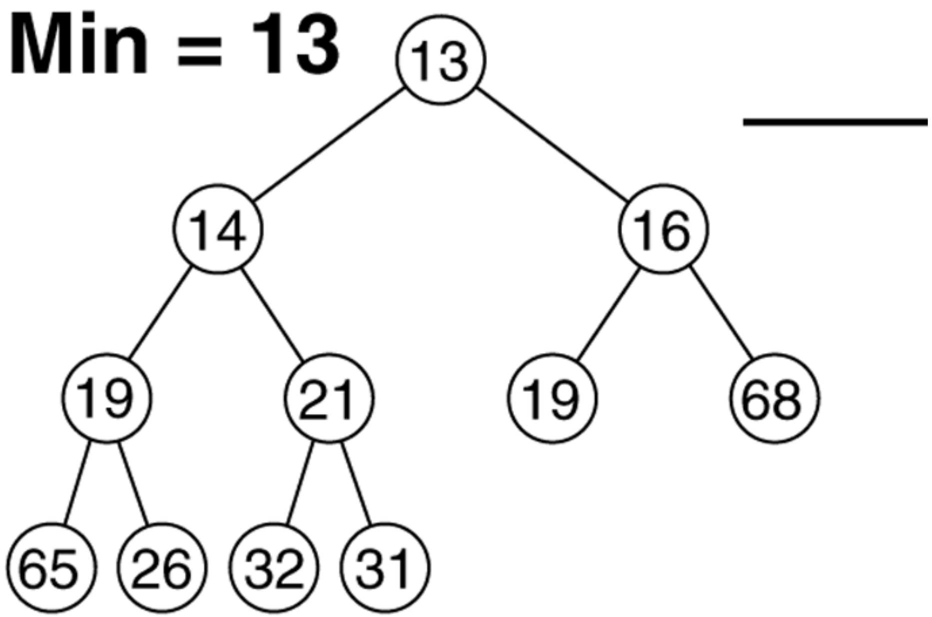


(a)

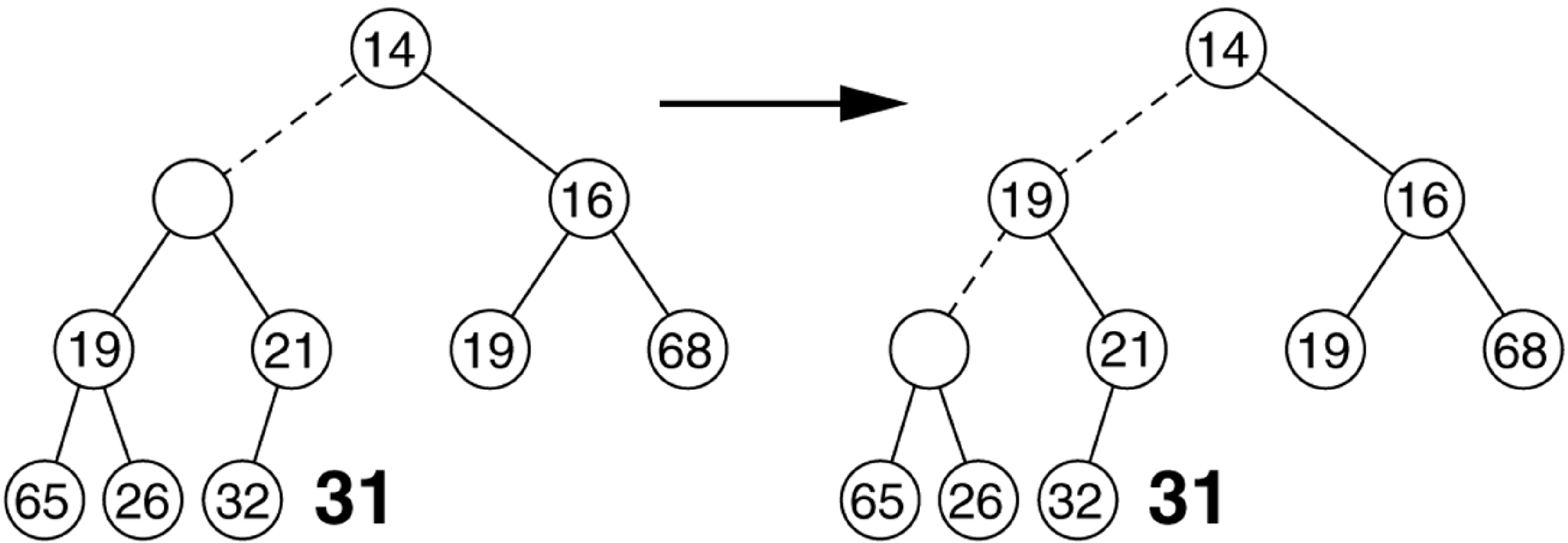
The remaining two steps required to insert 14 in the original heap shown



DeleteMin = Delete 13

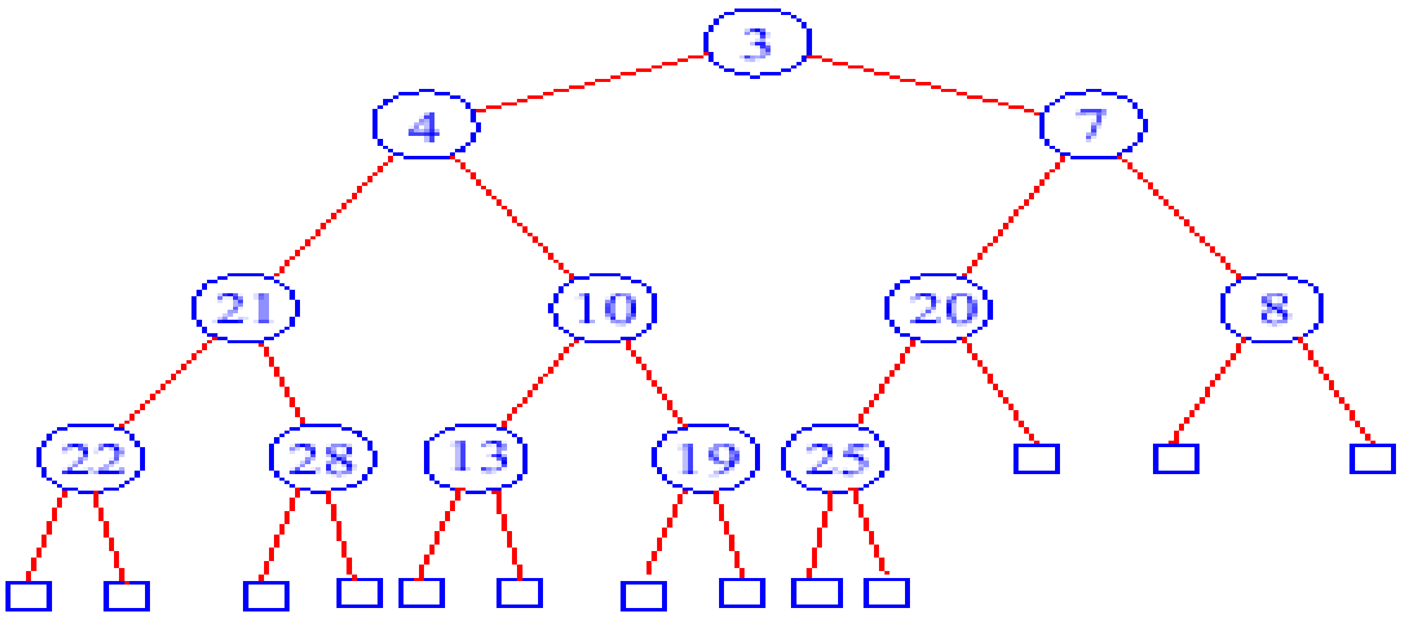


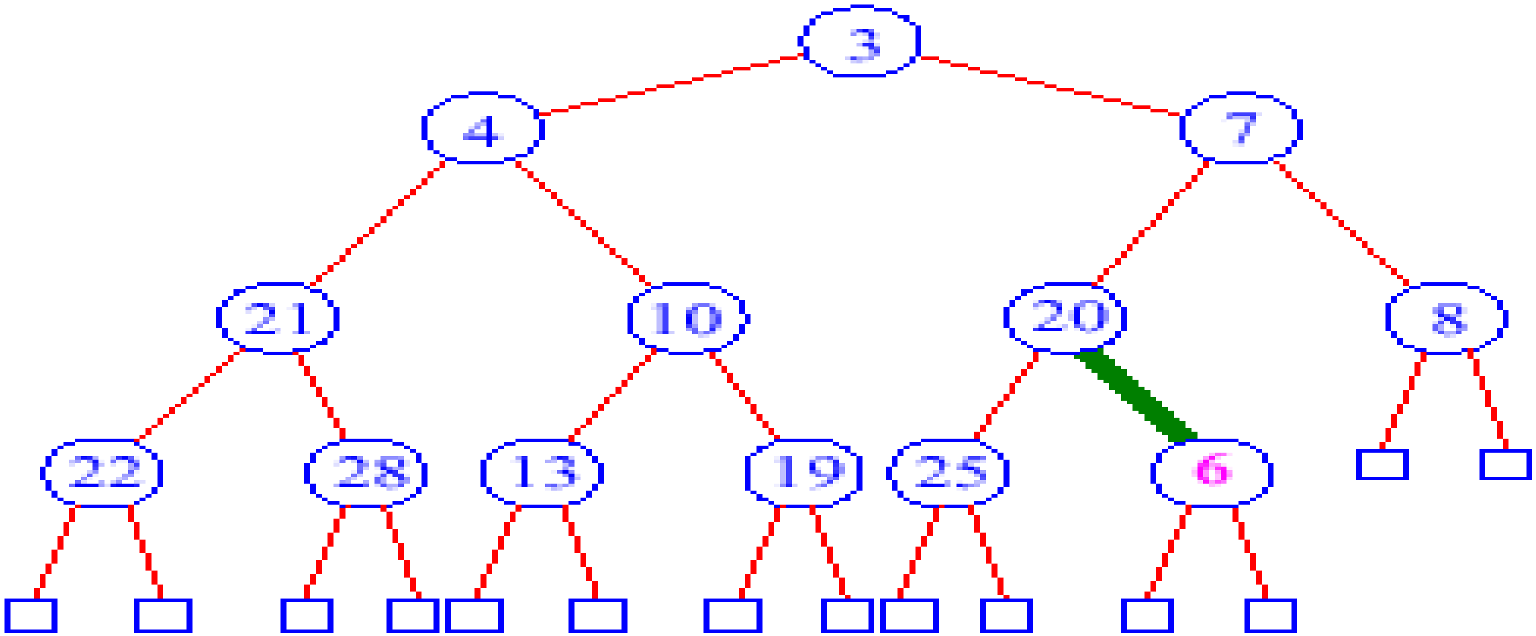
The next two steps in the deleteMin operation



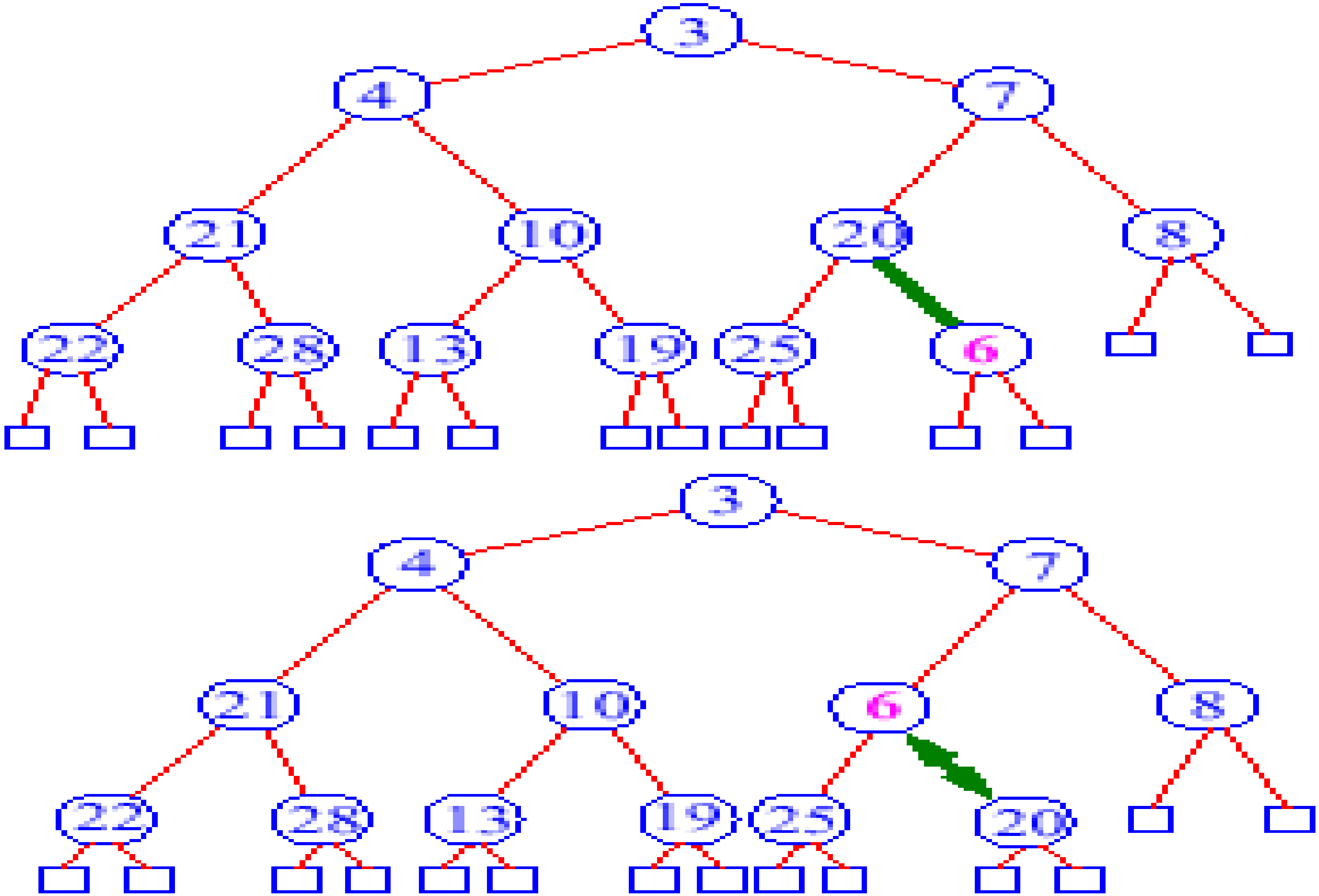
Heap Insertion

The key to insert is “6”

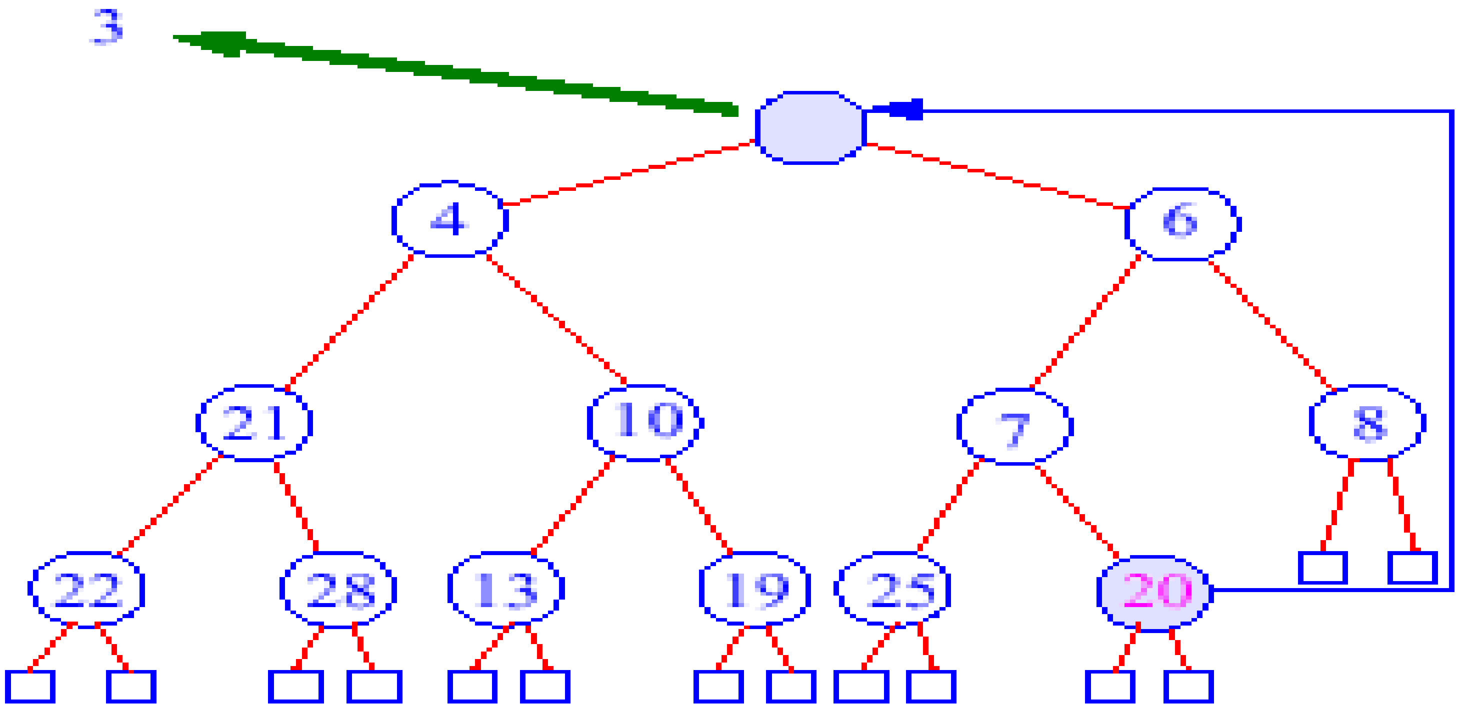



3
4
7
21
10
20
8
22
28
13
19
25
6

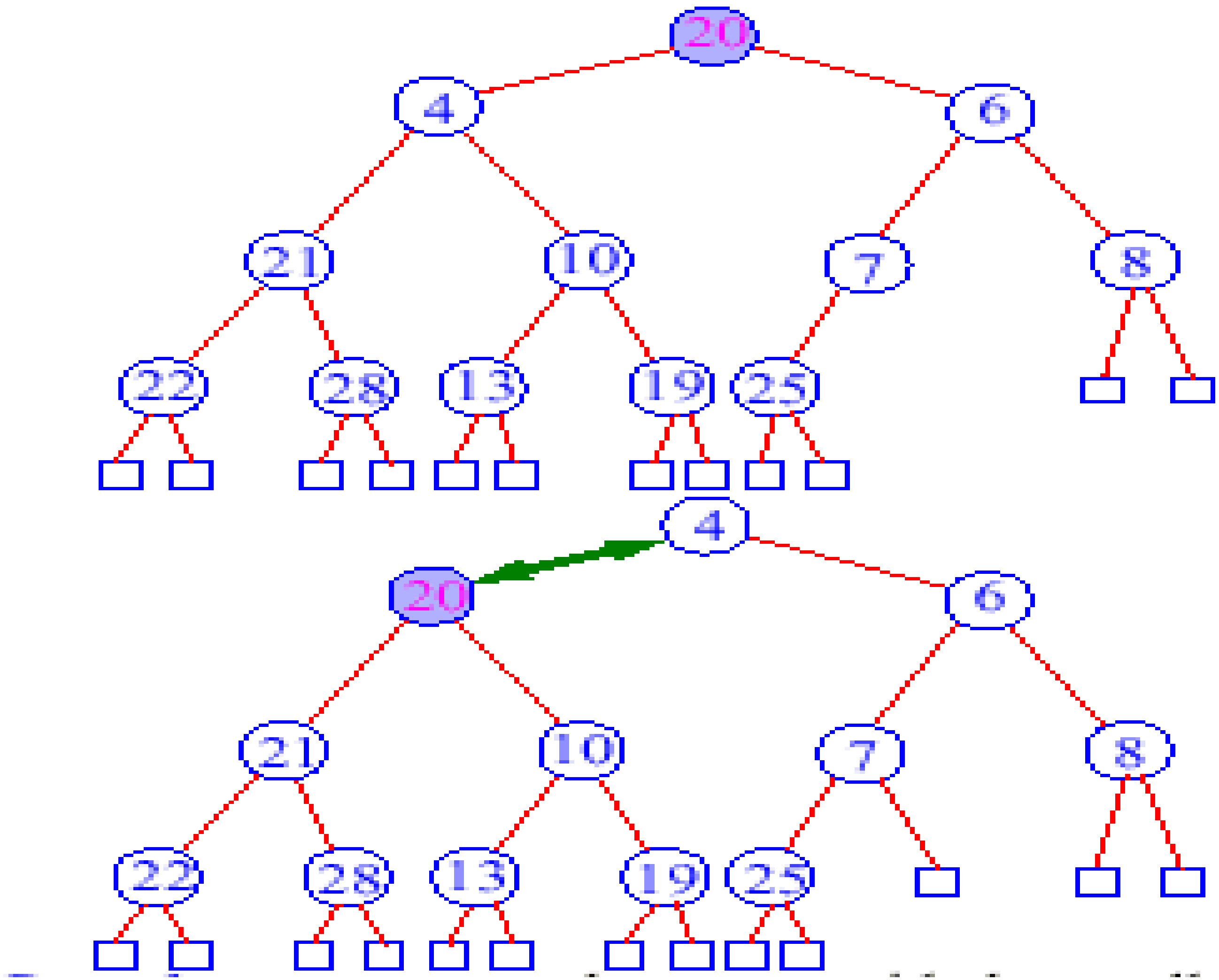
Upheap



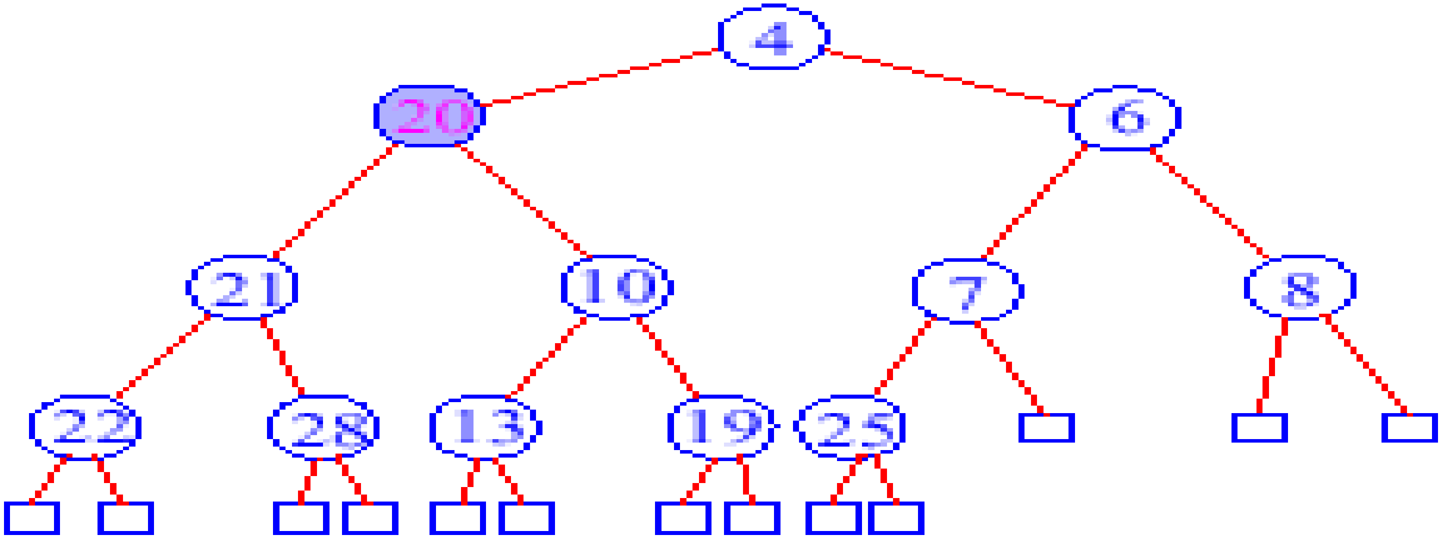
Removal From a Heap: RemoveMinElement()

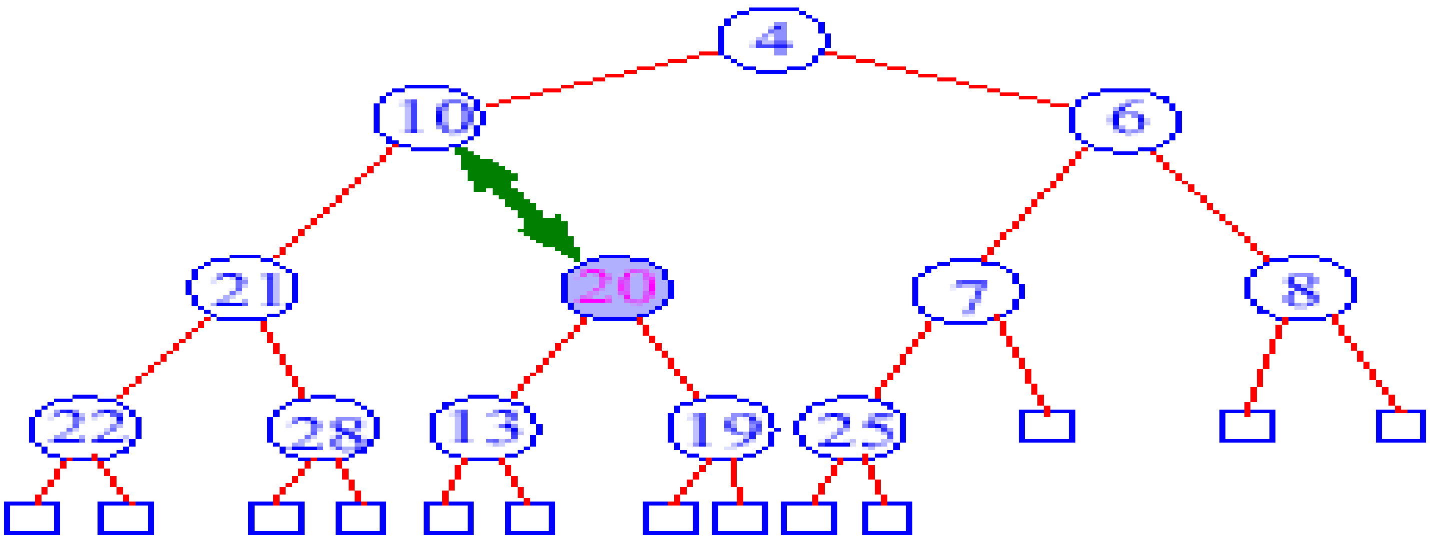


Downheap

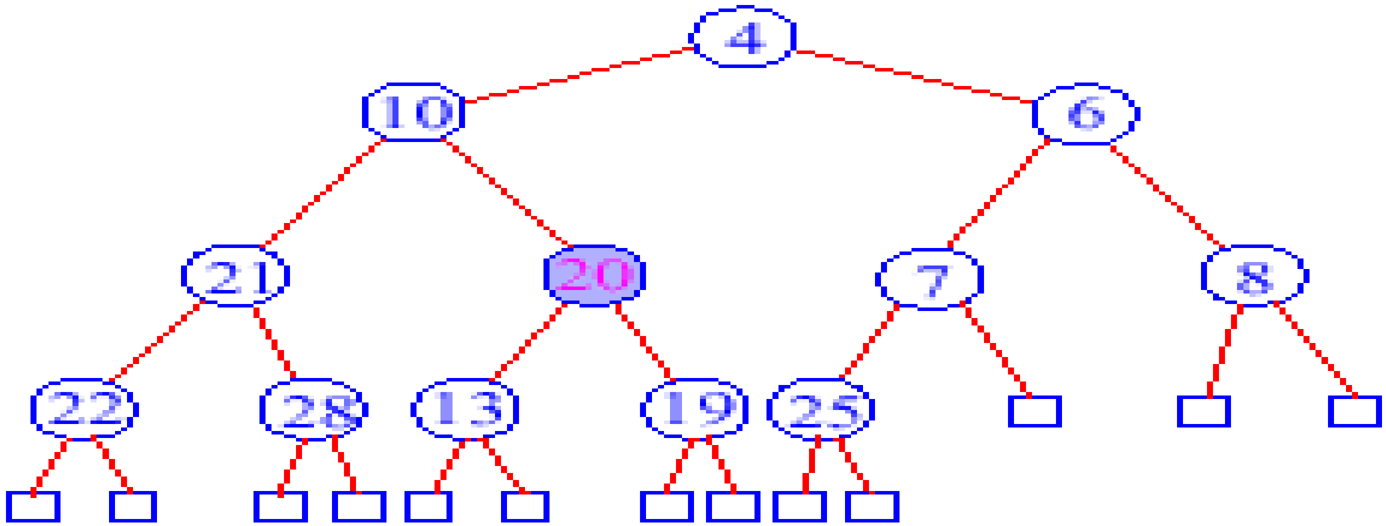


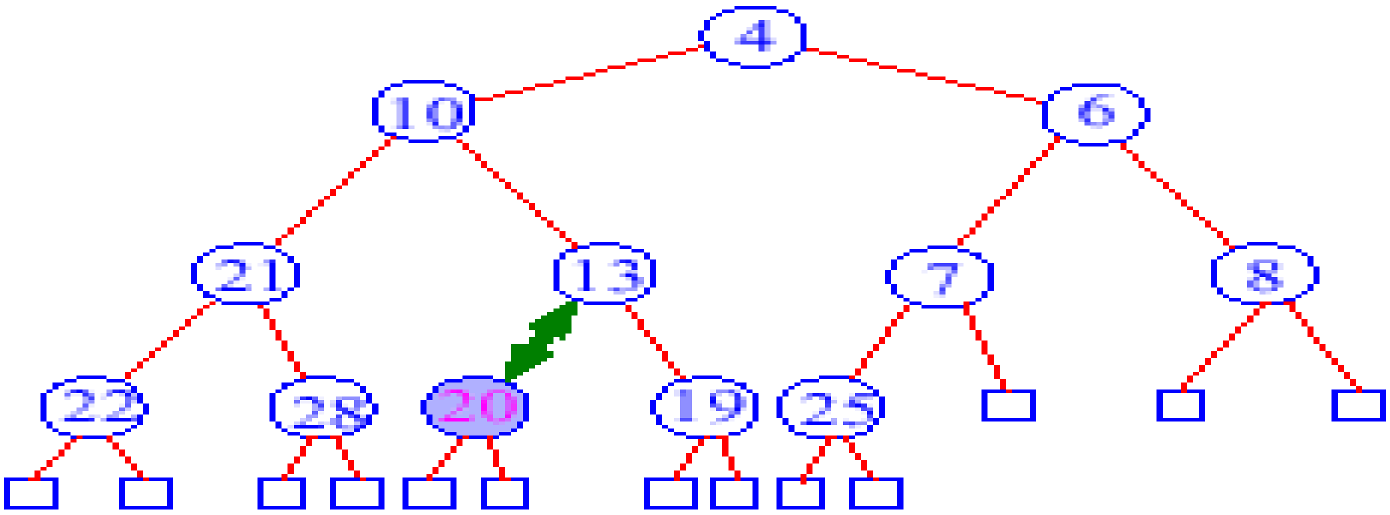
Downheap Continues





Downheap Continues





End of Downheap