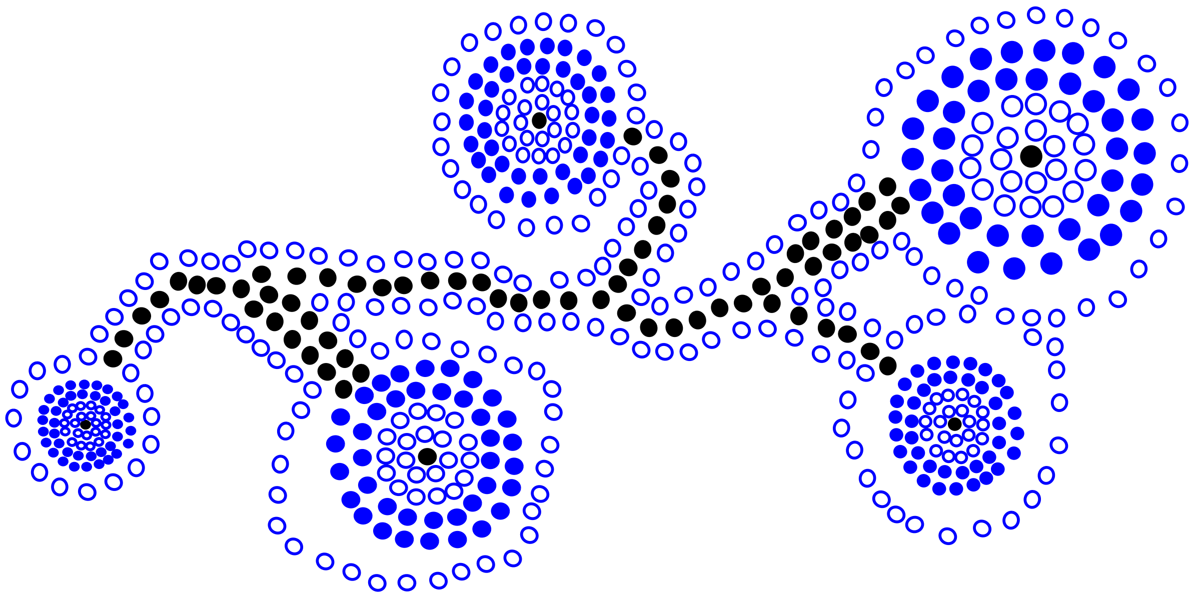
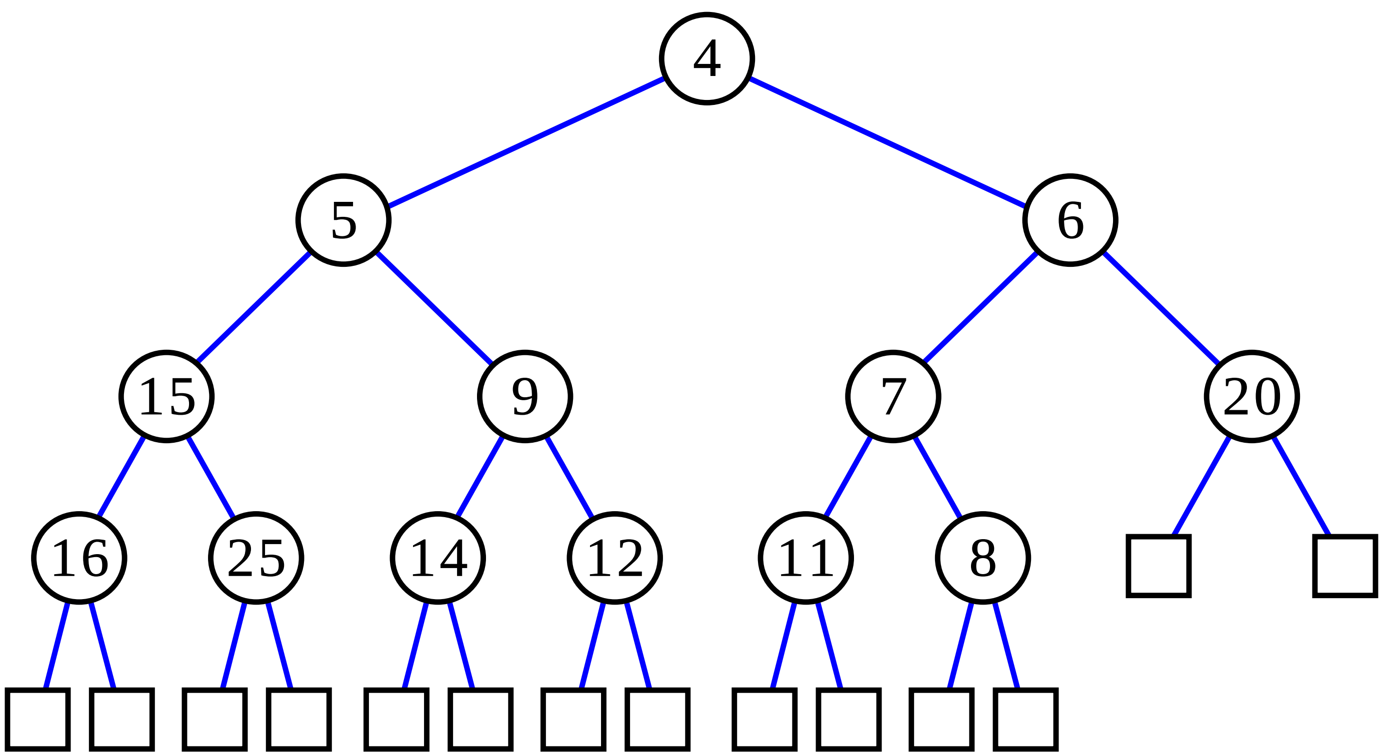
HEAPS

- Heaps
- Properties of Heaps
- HeapSort
- Bottom-Up Heap Construction
- Locators



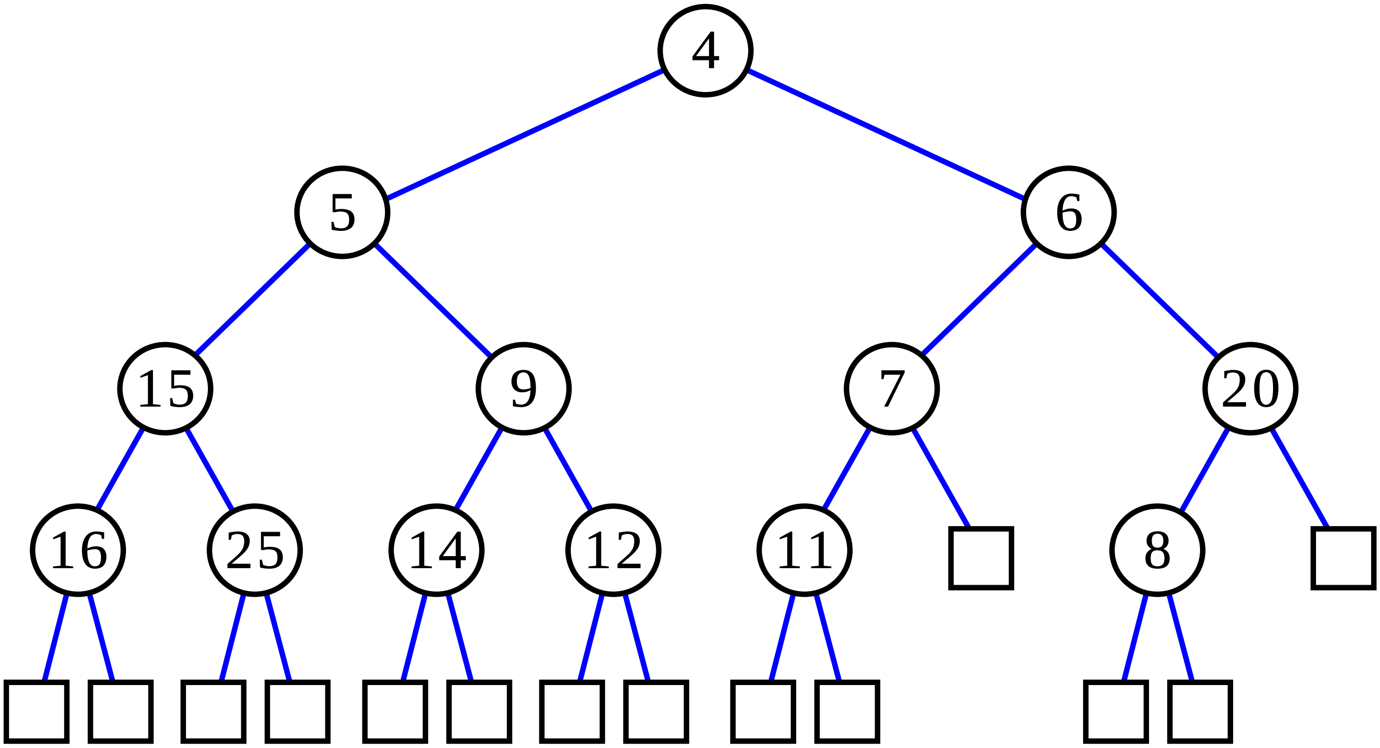
Heaps

- A ***heap*** is a binary tree *T* that stores a collection of keys (or key-element pairs) at its internal nodes and that satisfies two additional properties:
  - ***Order Property:*** key(parent) ≤ key(child)
  - ***Structural Property***: all levels are full, except the last one, which is left-filled (***complete binary tree***)

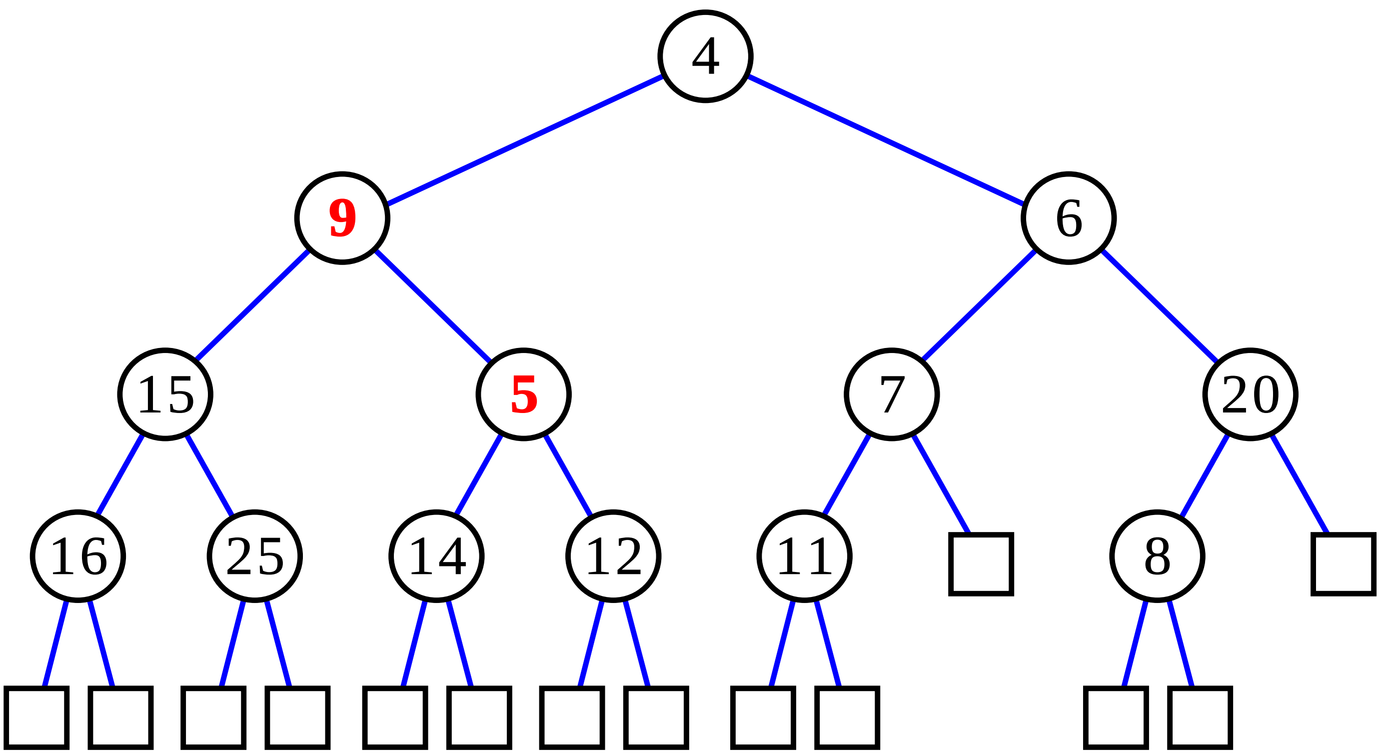


Not Heaps

- bottom level is not left-filled



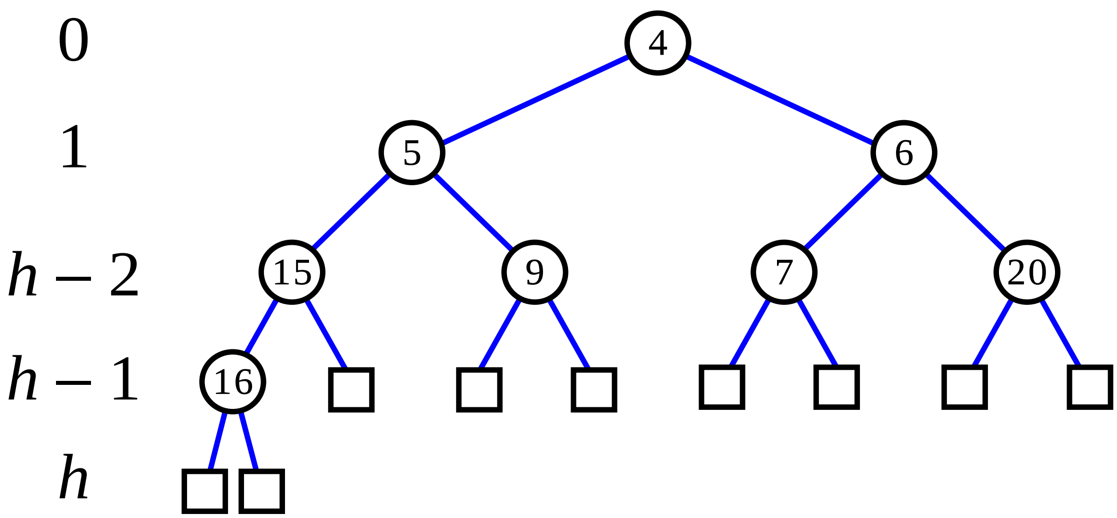
- key(parent)> key(child)



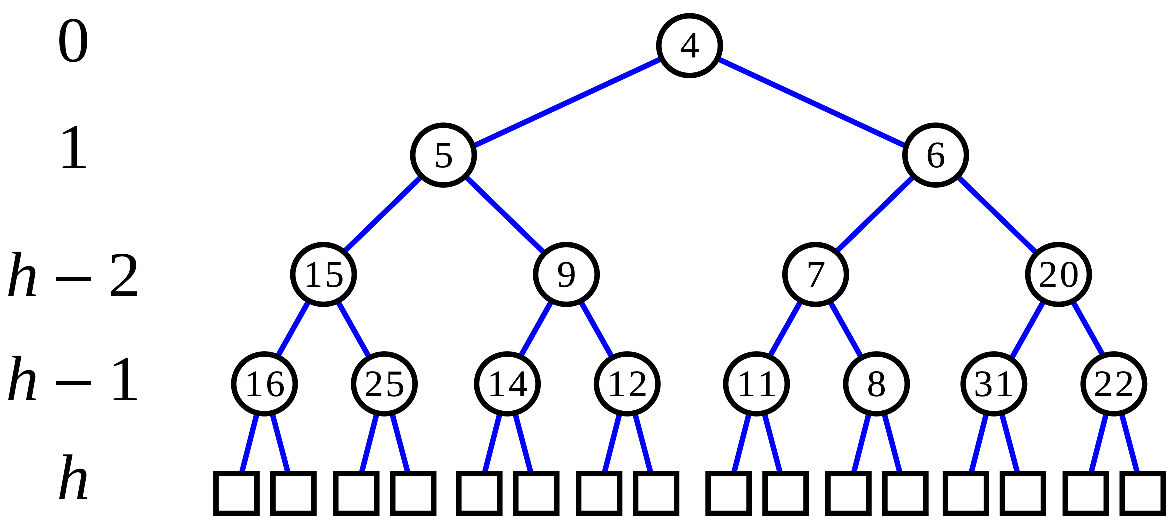
Height of a Heap

A heap T storing n keys has height $h = \lceil \log(n + 1) \rceil$, which is $O(\log n)$

- $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$



- $n \leq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$

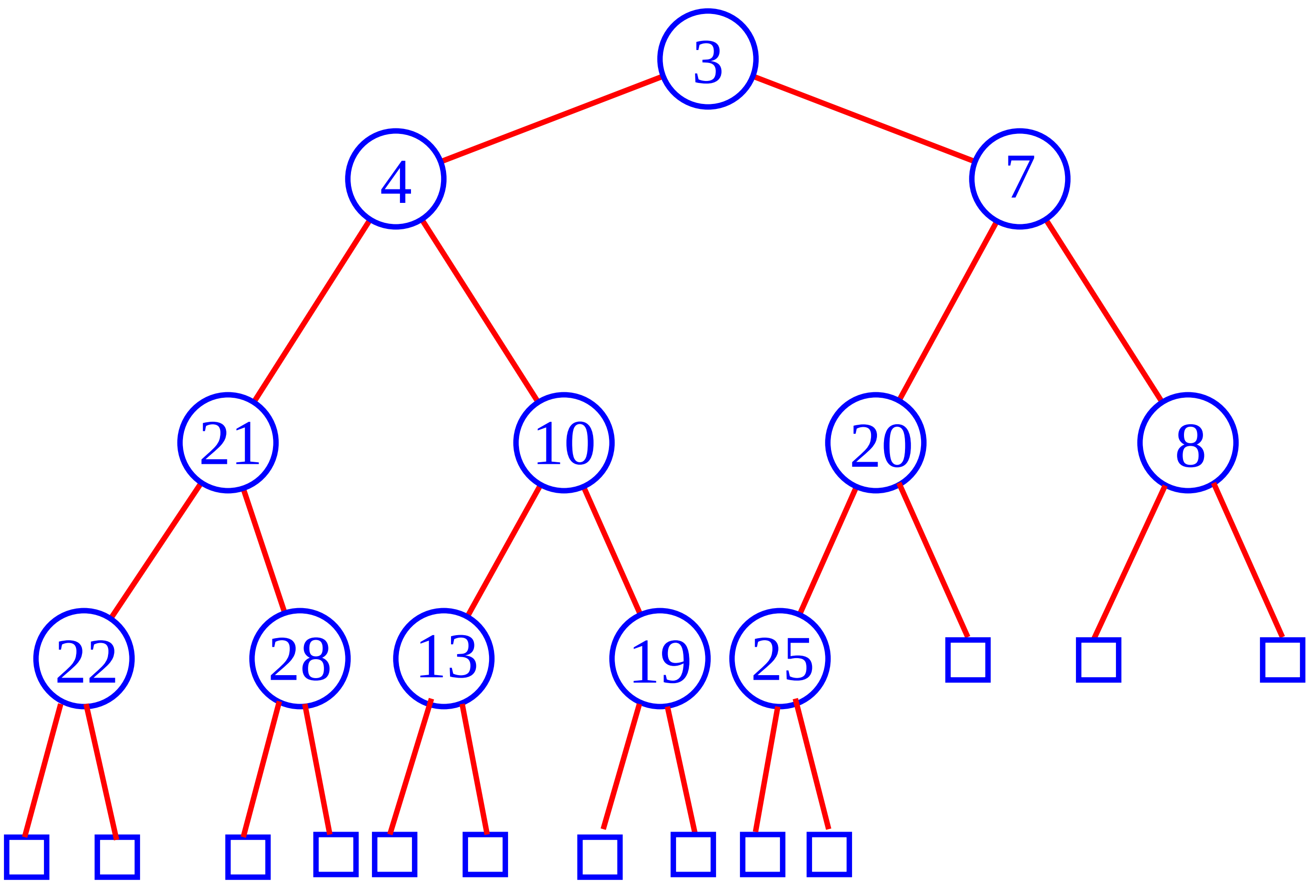


- Therefore $2^{h-1} \leq n \leq 2^h - 1$
- Taking logs, we get $\log(n + 1) \leq h \leq \log n + 1$
- Which implies $\boldsymbol{h = \lceil \log(n+1) \rceil}$

Heap Insertion

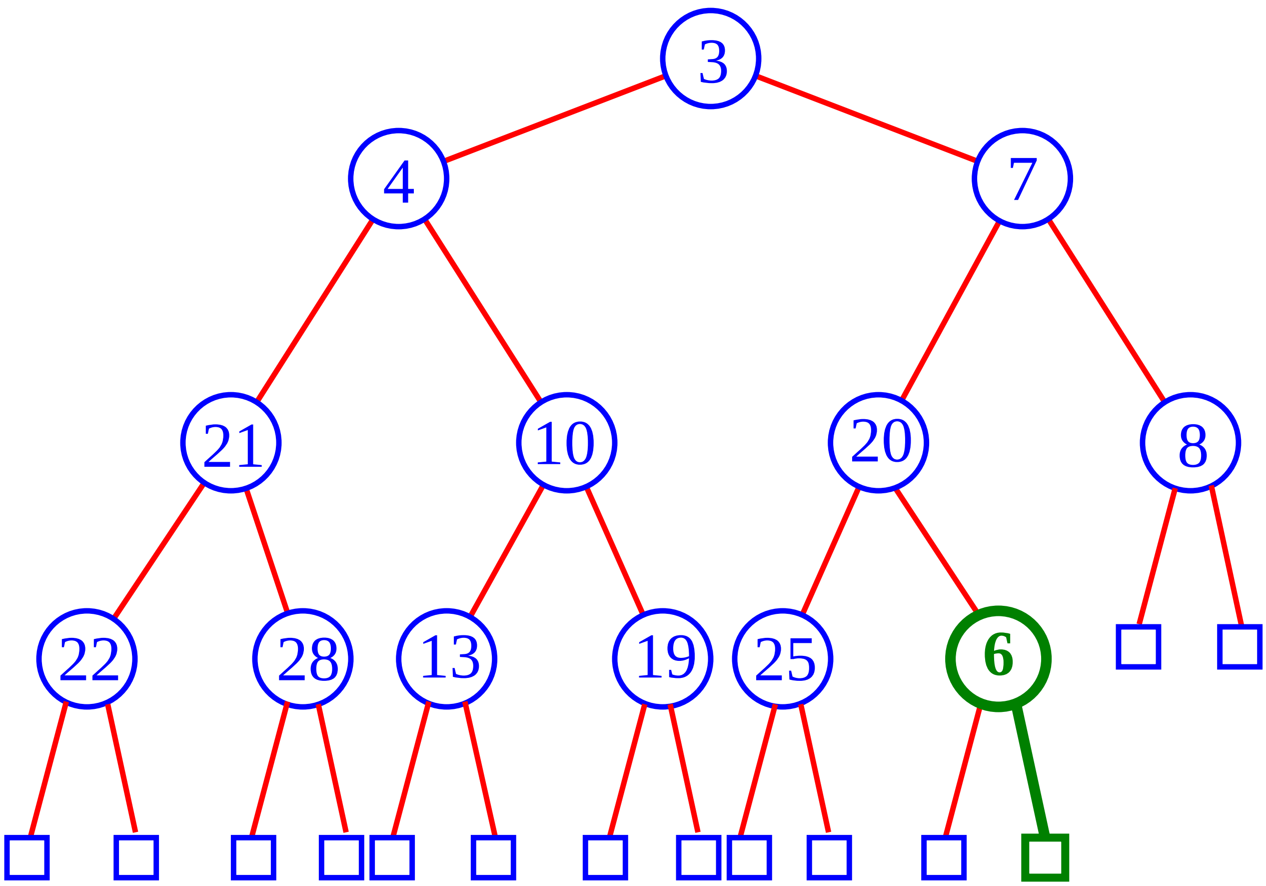
So here we go ...

The key to insert is **6**



Heap Insertion

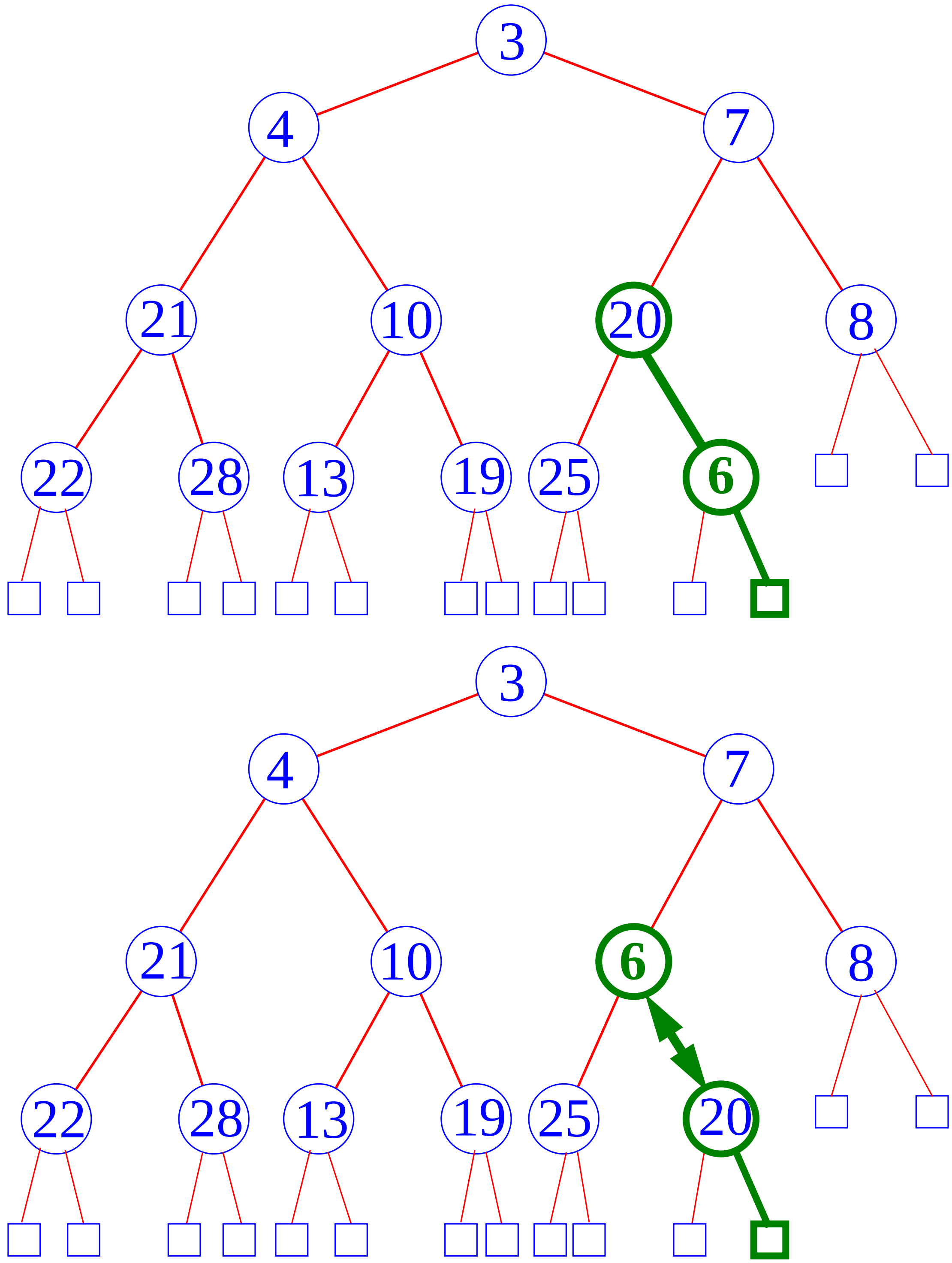
Add the key in the ***next available position*** in the heap.



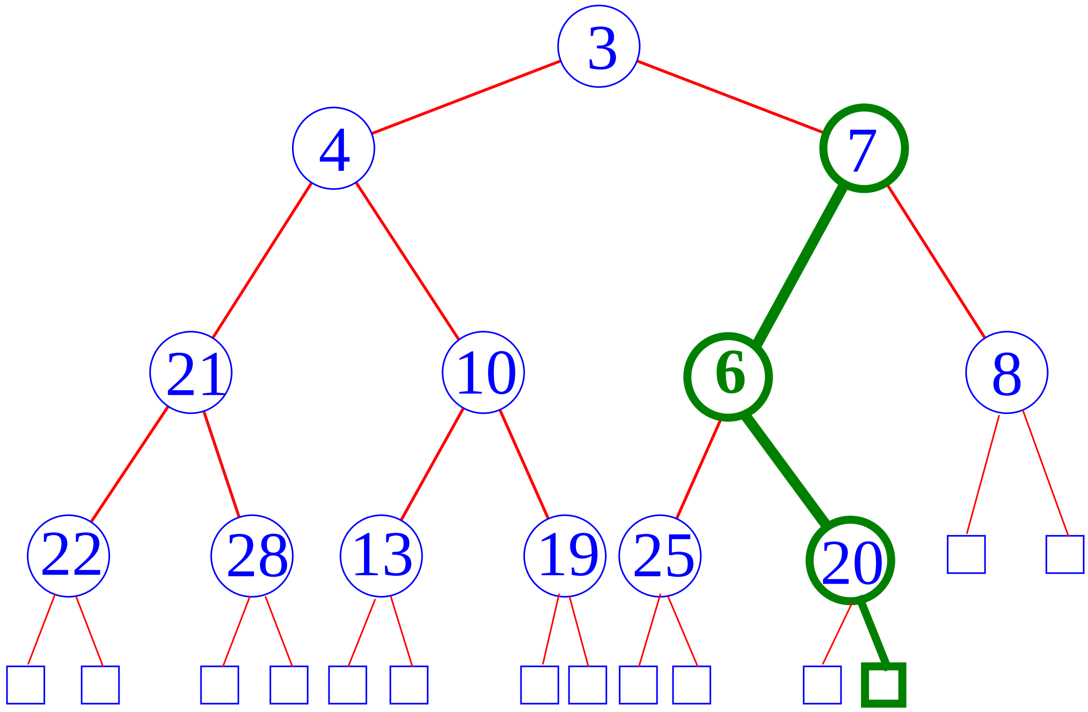
Now begin ***Upheap***.

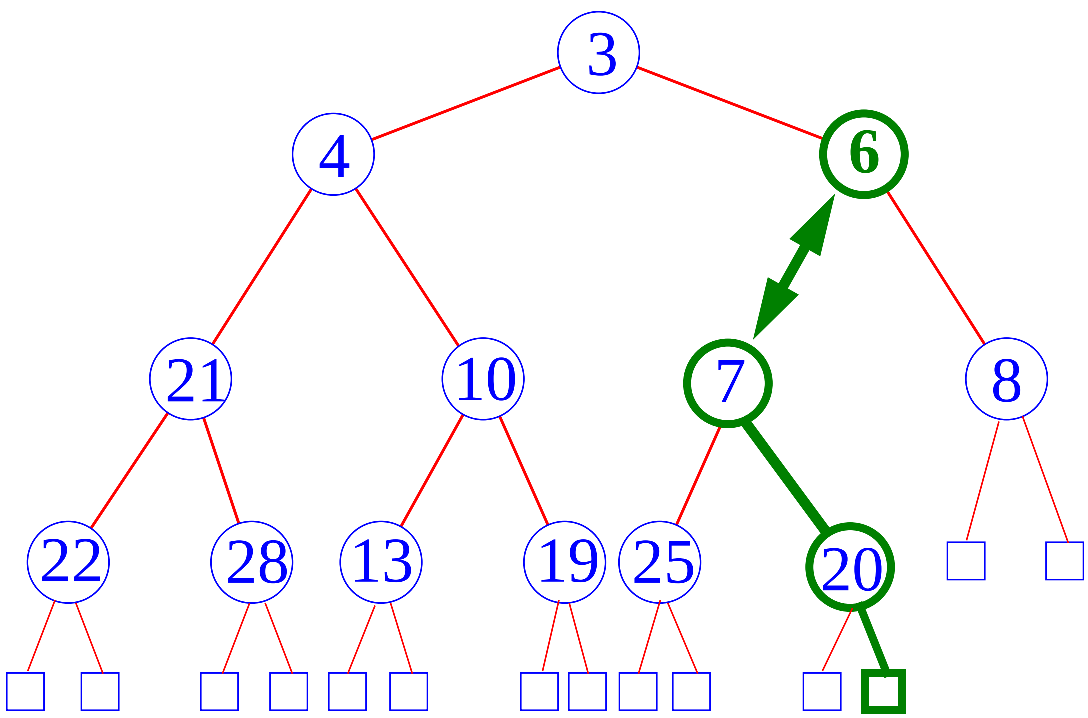
Upheap

- ***Swap parent-child keys out of order***

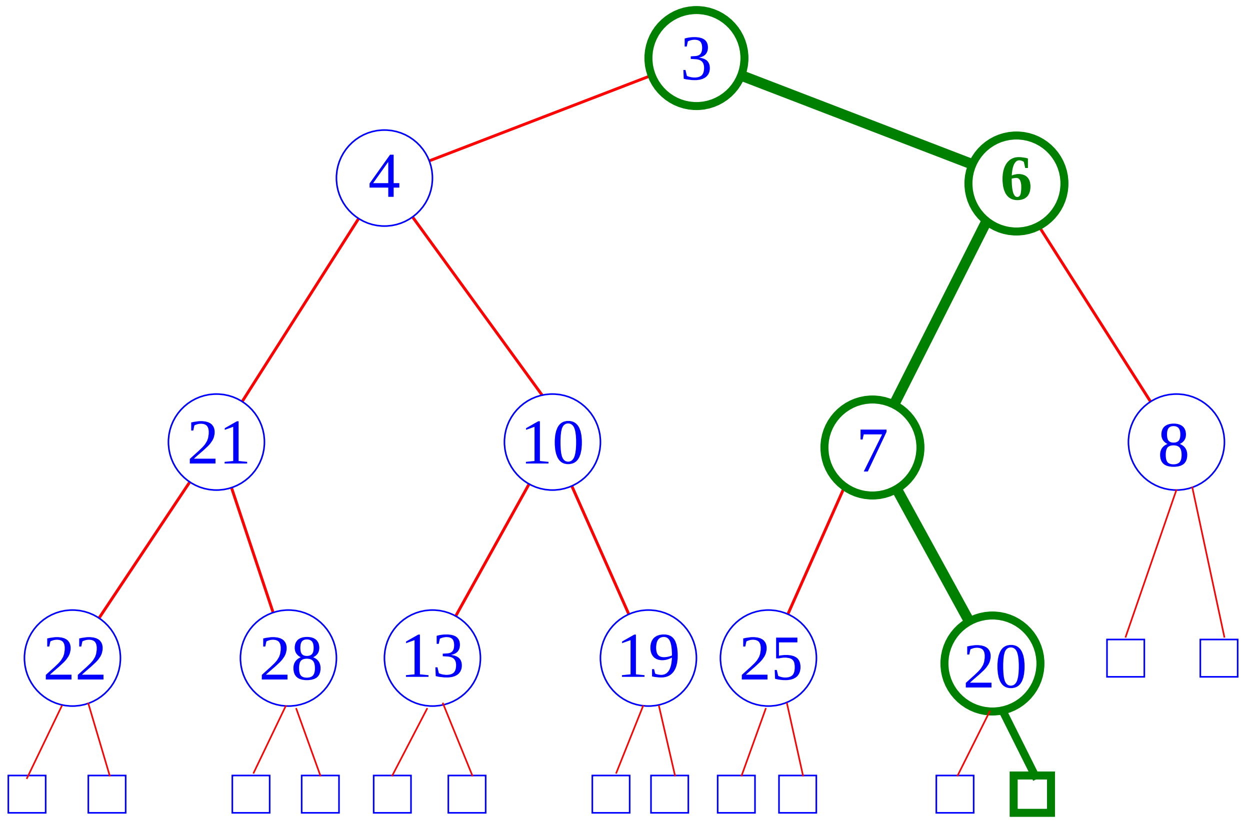


Upheap Continues





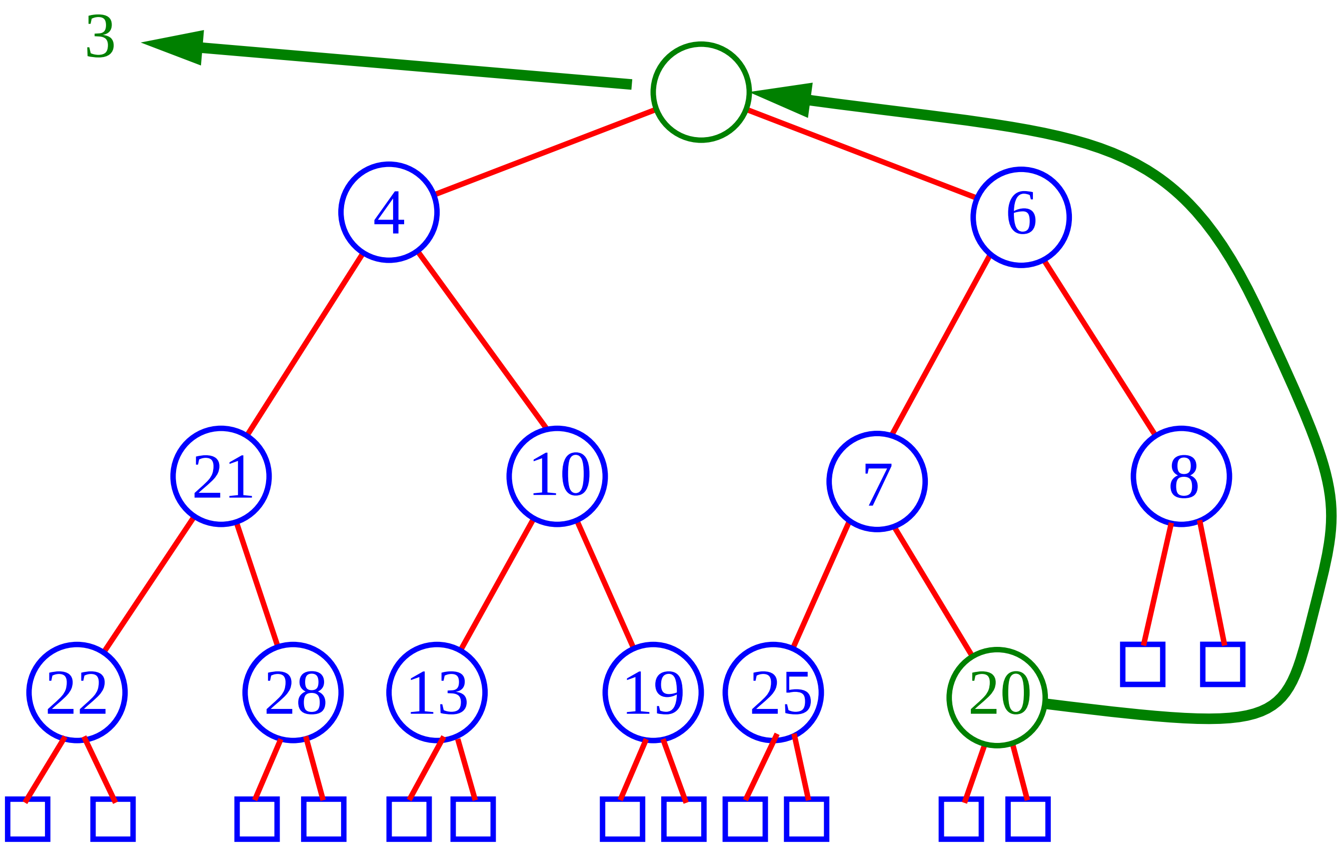
End of Upheap



- ***Upheap*** terminates when new key is greater than the key of its parent **or** the top of the heap is reached
- (total #swaps) $\leq$ ($h - 1$), which is O(log n)

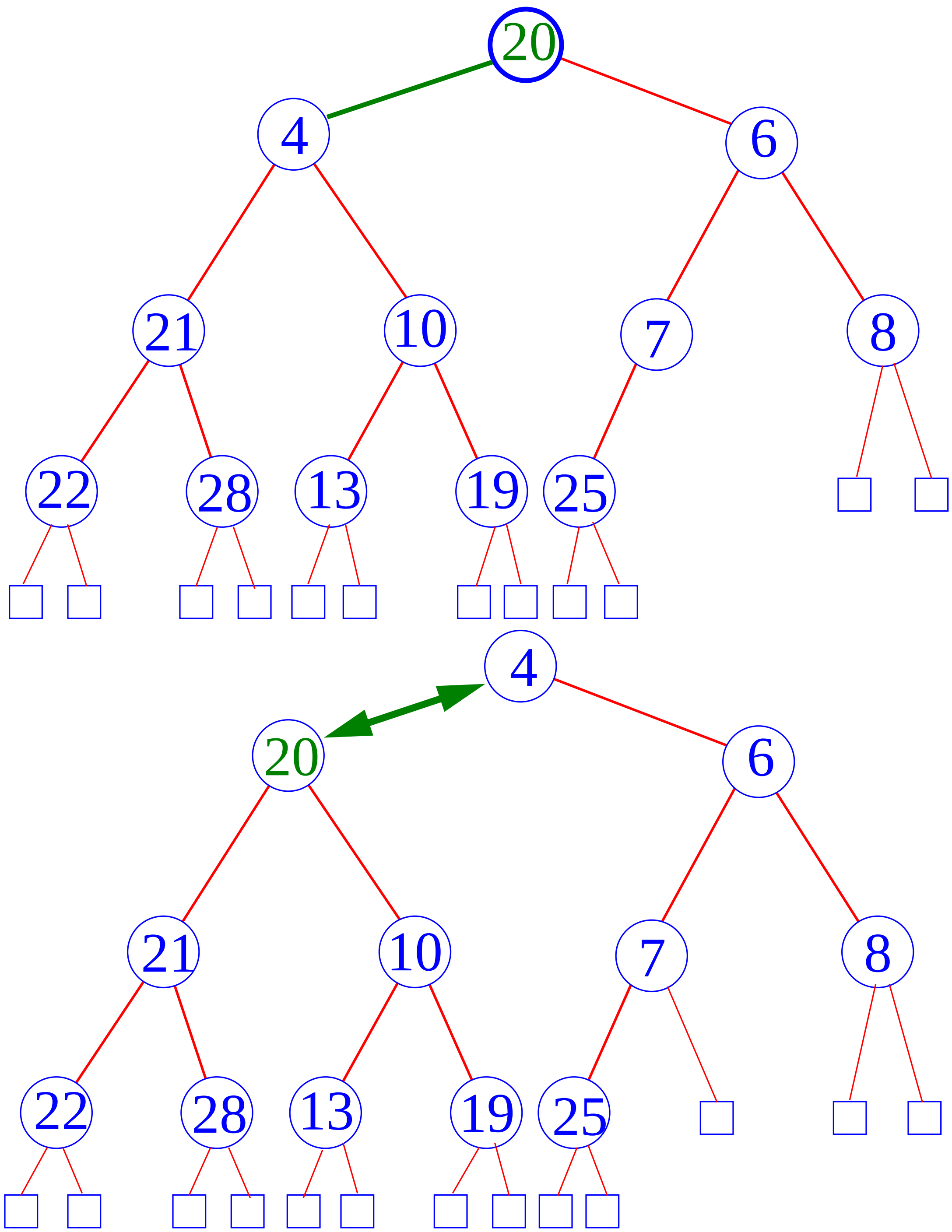
Removal From a Heap

RemoveMin()



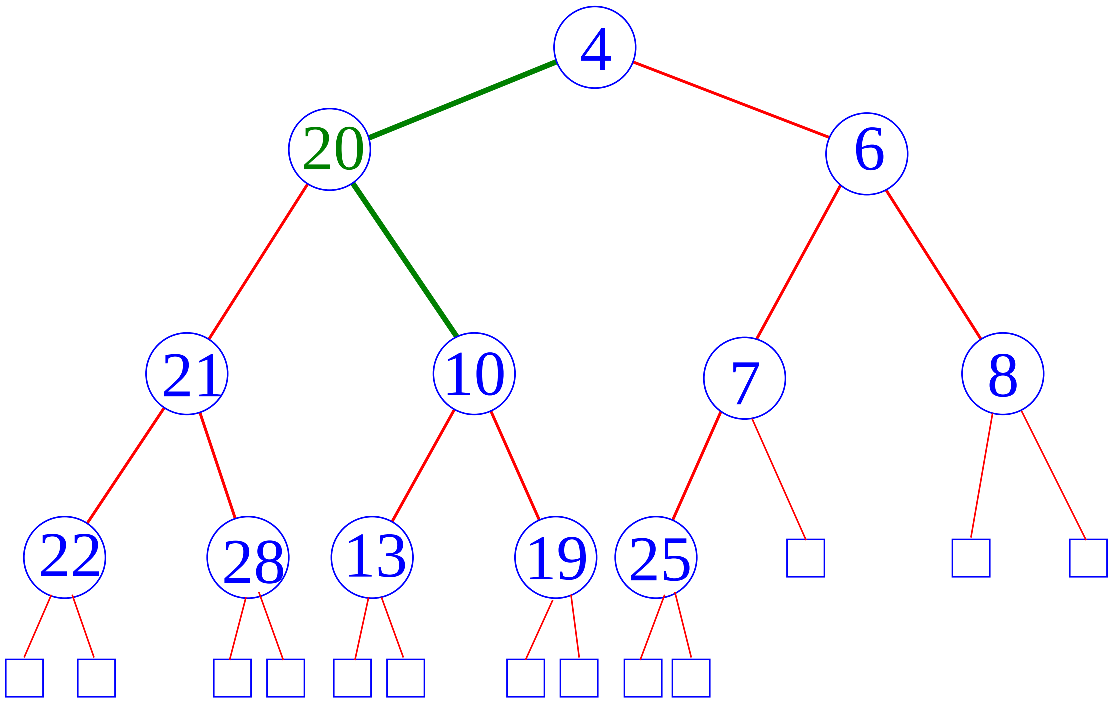
- The removal of the top key leaves a hole
- We need to fix the heap
- First, replace the hole with the last key in the heap
- Then, begin ***Downheap***

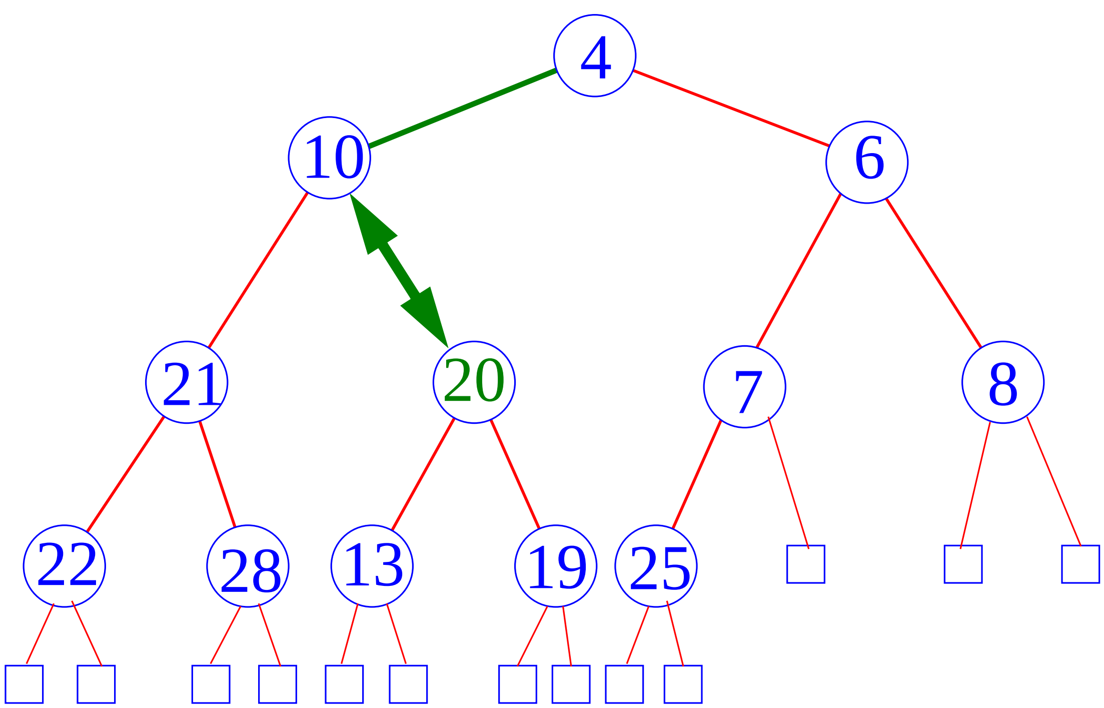
Downheap



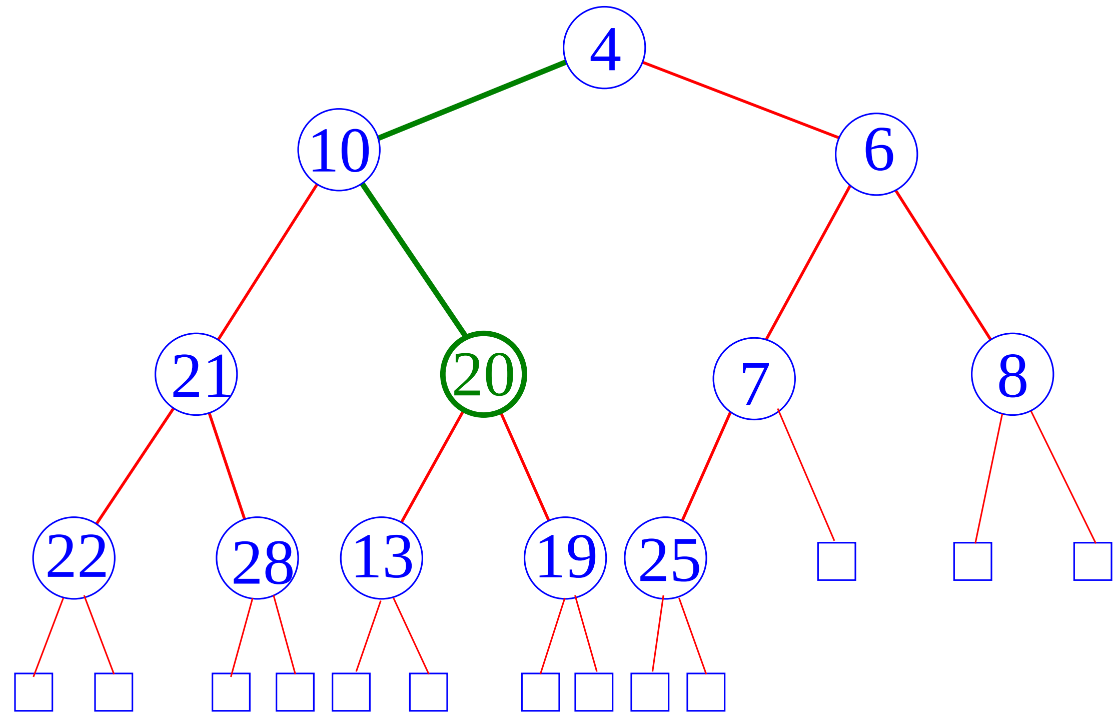
Downheap compares the parent with the smallest child. If the child is smaller, it switches the two.

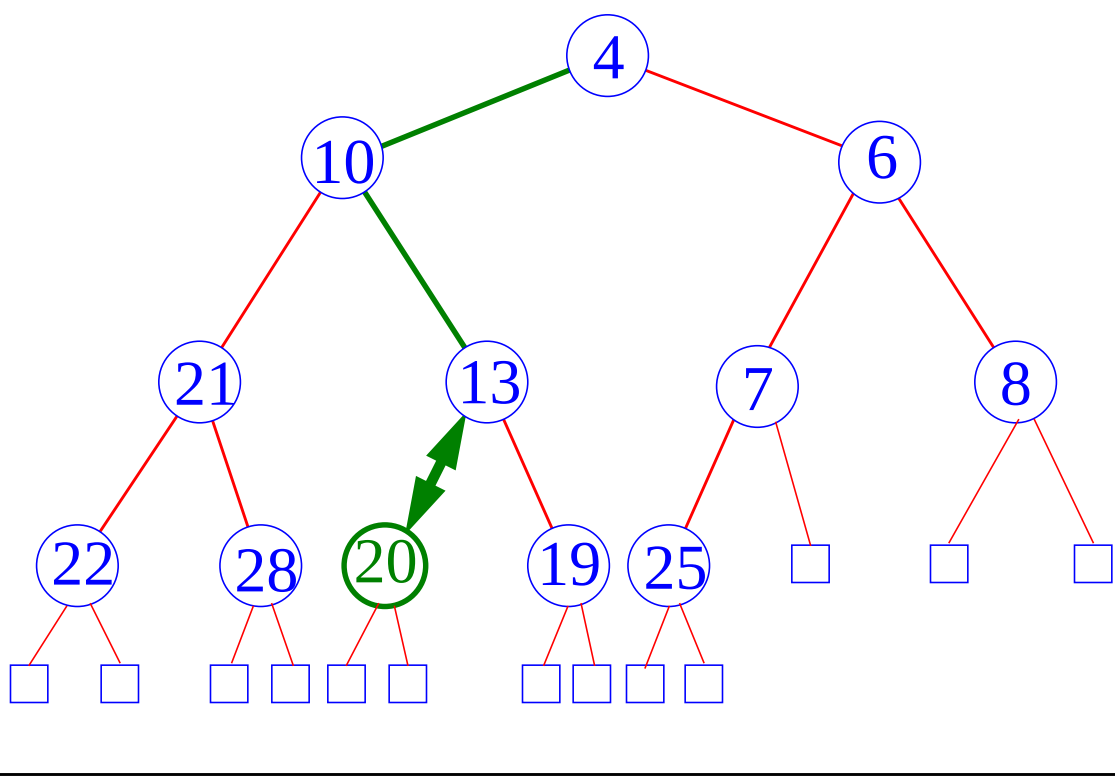
Downheap Continues



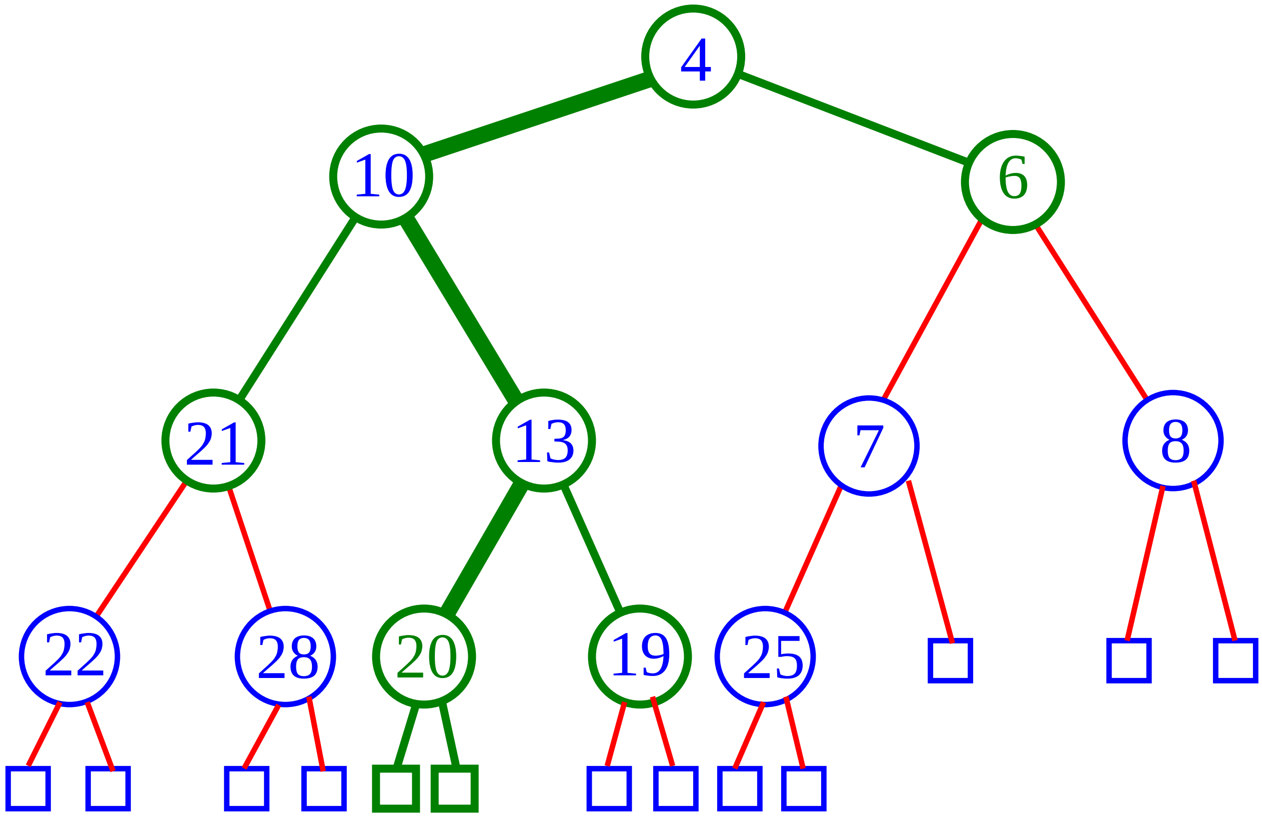


Downheap Continues





End of Downheap

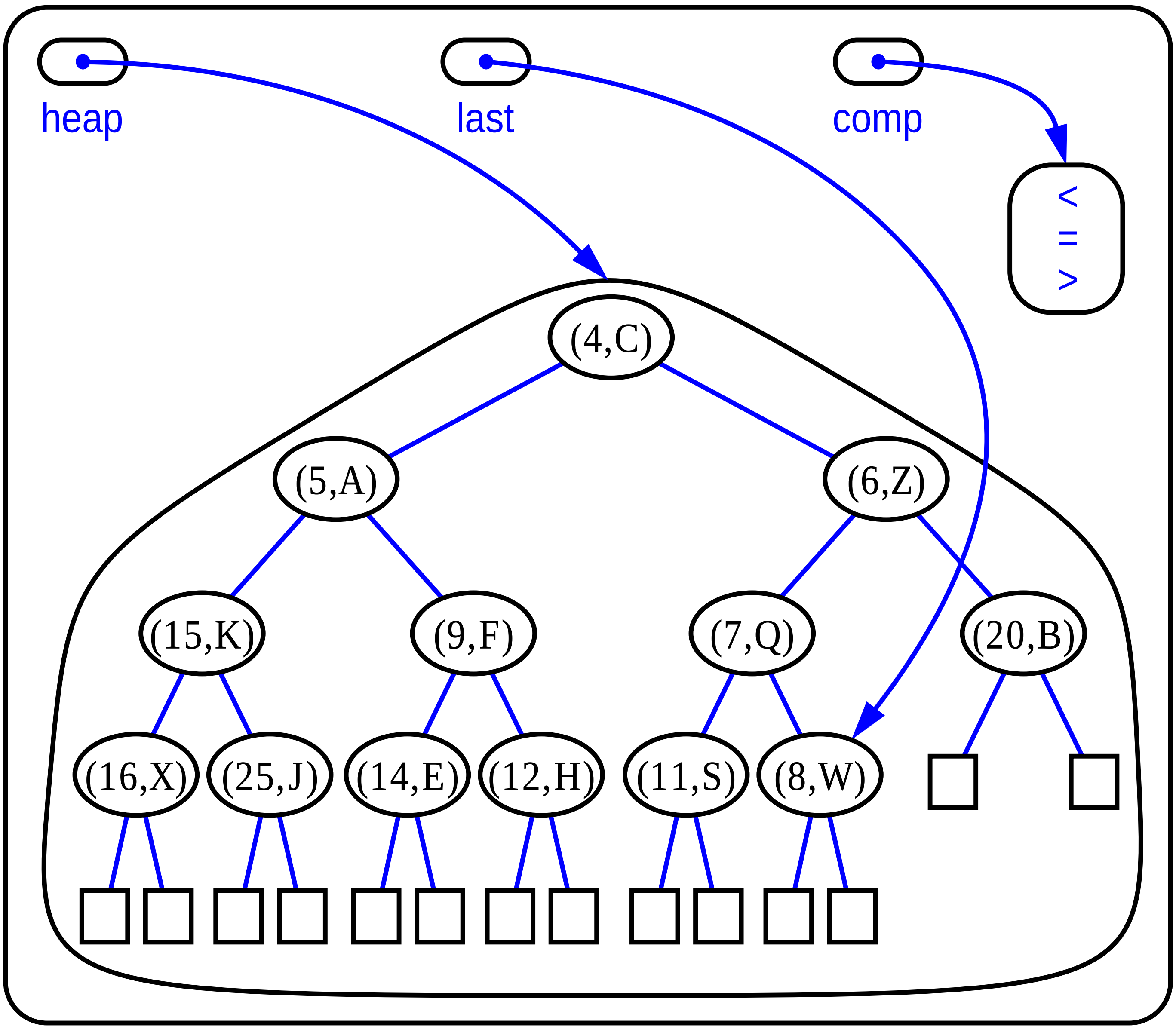


- ***Downheap*** terminates when the key is greater than the keys of both its children **or** the bottom of the heap is reached.

- (total #swaps) $\leq$ $(h - 1)$, which is $O(\log n)$

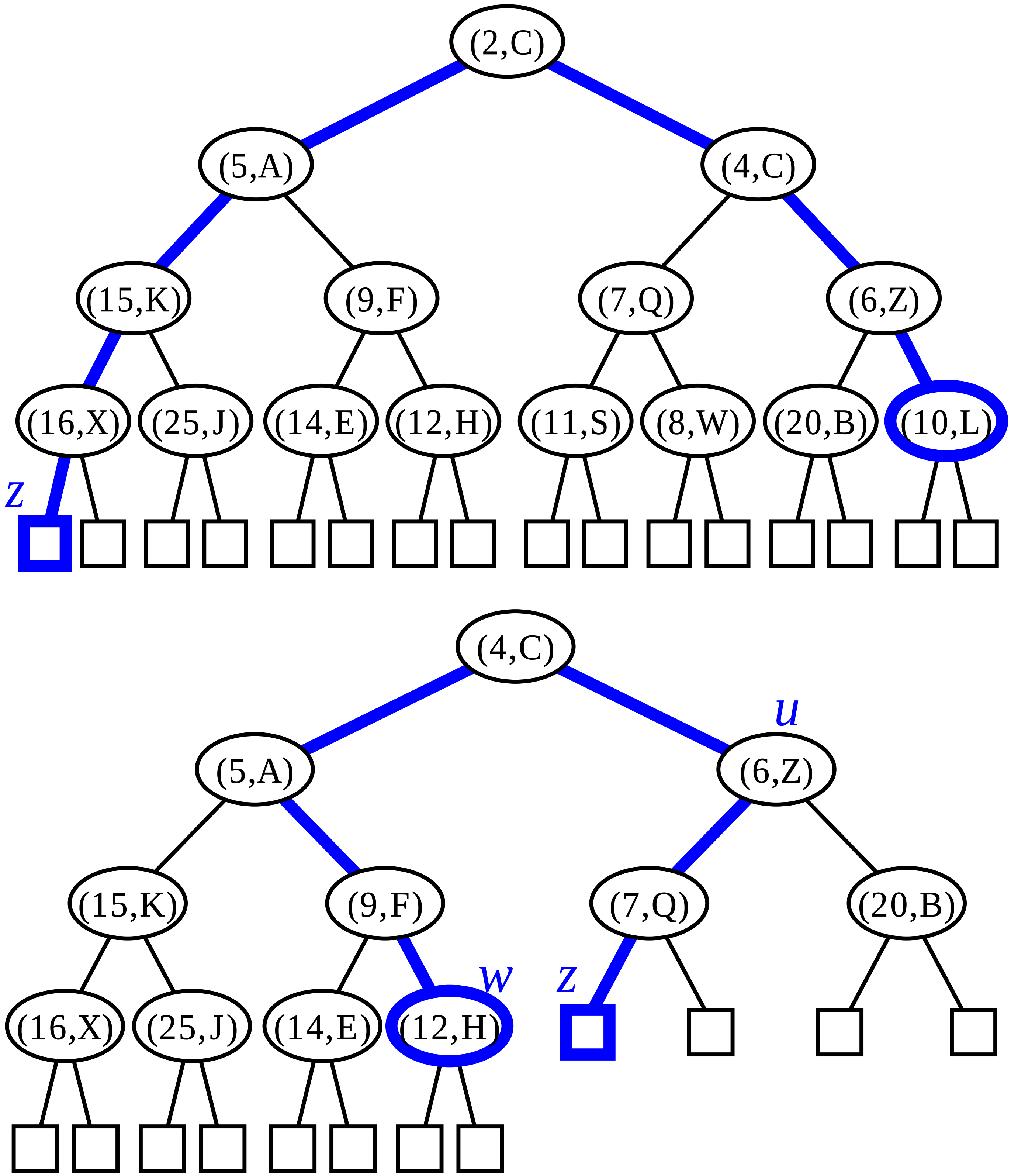
Implementation of a Heap

```
public class HeapPriorityQueue implements PriorityQueue
{
   BinaryTree T;
   Position last;
   Comparator comparator;
   ...
}
```



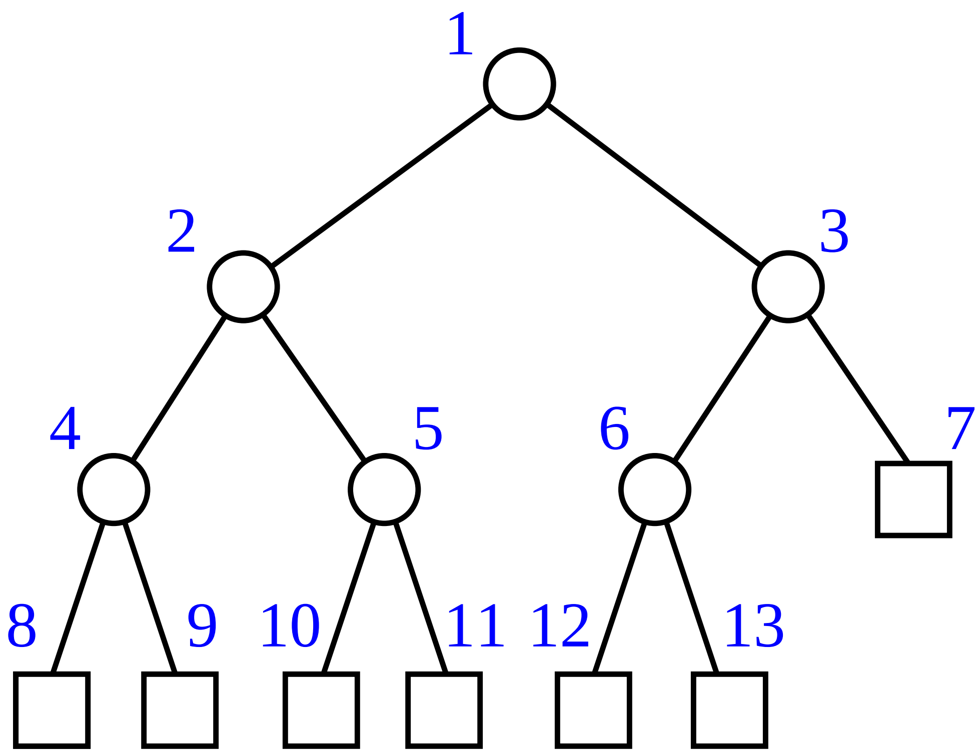
Implementation of a Heap(cont.)

- Two ways to find the insertion position z in a heap:



Vector Based Implementation

- Updates in the underlying tree occur only at the "last element"
- A heap can be represented by a vector, where the node at rank i has
  - left child at rank $2i$ and
  - right child at rank $2i + 1$



- The leaves do no need to be explicitly stored
- Insertion and removals into/from the heap correspond to insertLast and removeLast on the vector, respectively

Heap Sort

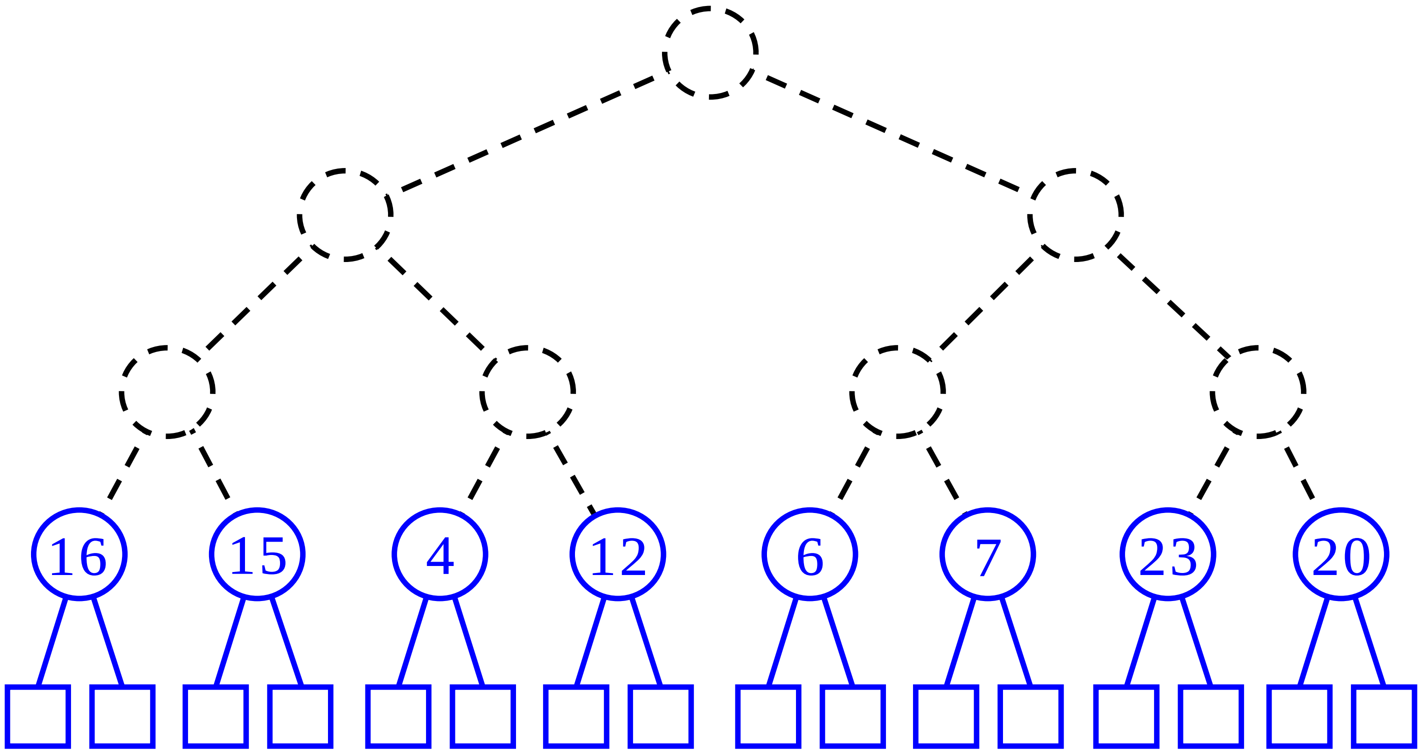
- All heap methods run in logarithmic time or better
- If we implement PriorityQueueSort using a heap for our priority queue, insertItem and removeMin each take O(log k), k being the number of elements in the heap at a given time.
- We always have at most n elements in the heap, so the worst case time complexity of these methods is O(log n).
- Thus each phase takes O(n log n) time, so the algorithm runs in O(n log n) time also.
- This sort is known as ***heap-sort***.
- The O(n log n) run time of heap-sort is much better than the O(n^2) run time of selection and insertion sort.

In-Place Heap-Sort

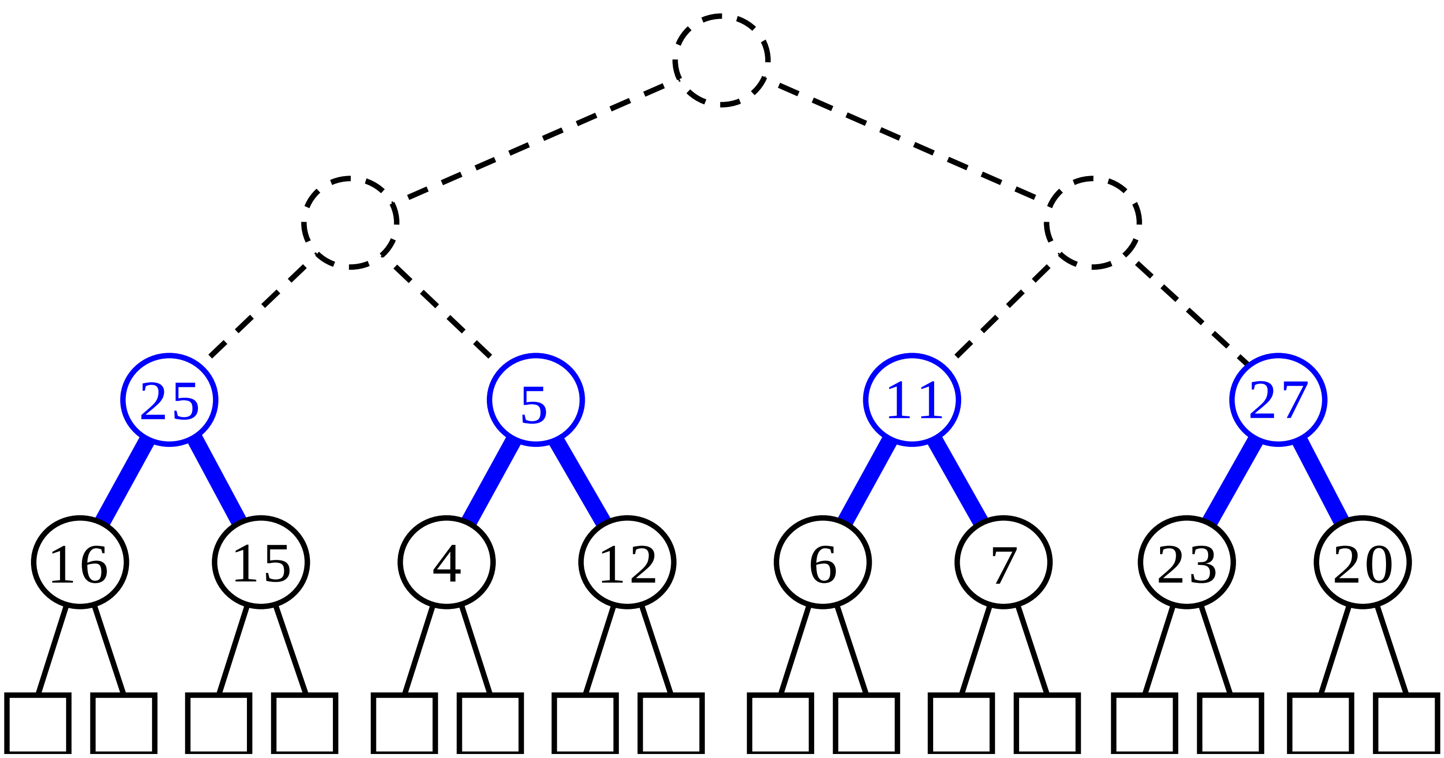
- Do not use an external heap
- Embed the heap into the sequence, using the vector representation

Bottom-Up Heap Construction

- build $(n + 1)/2$ trivial one-element heaps

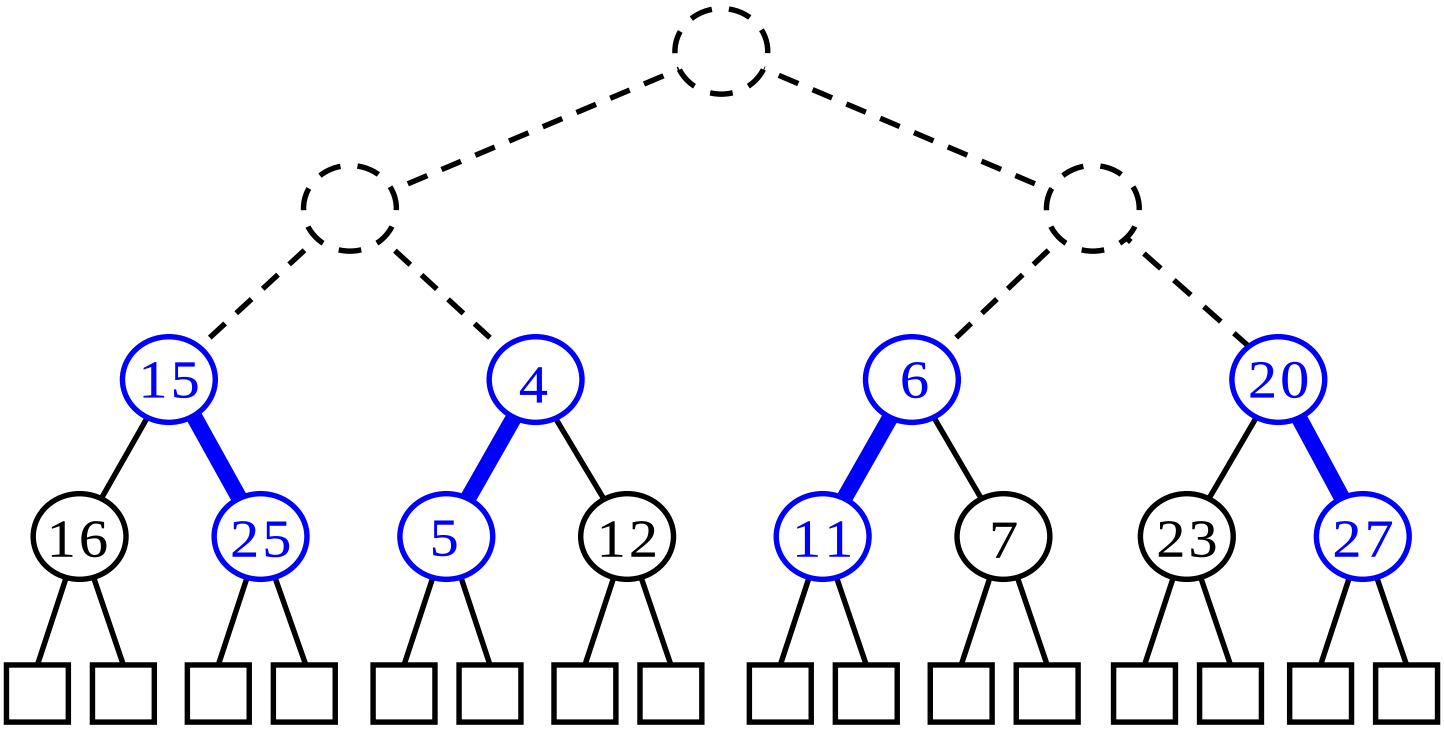


- now build three-element heaps on top of them

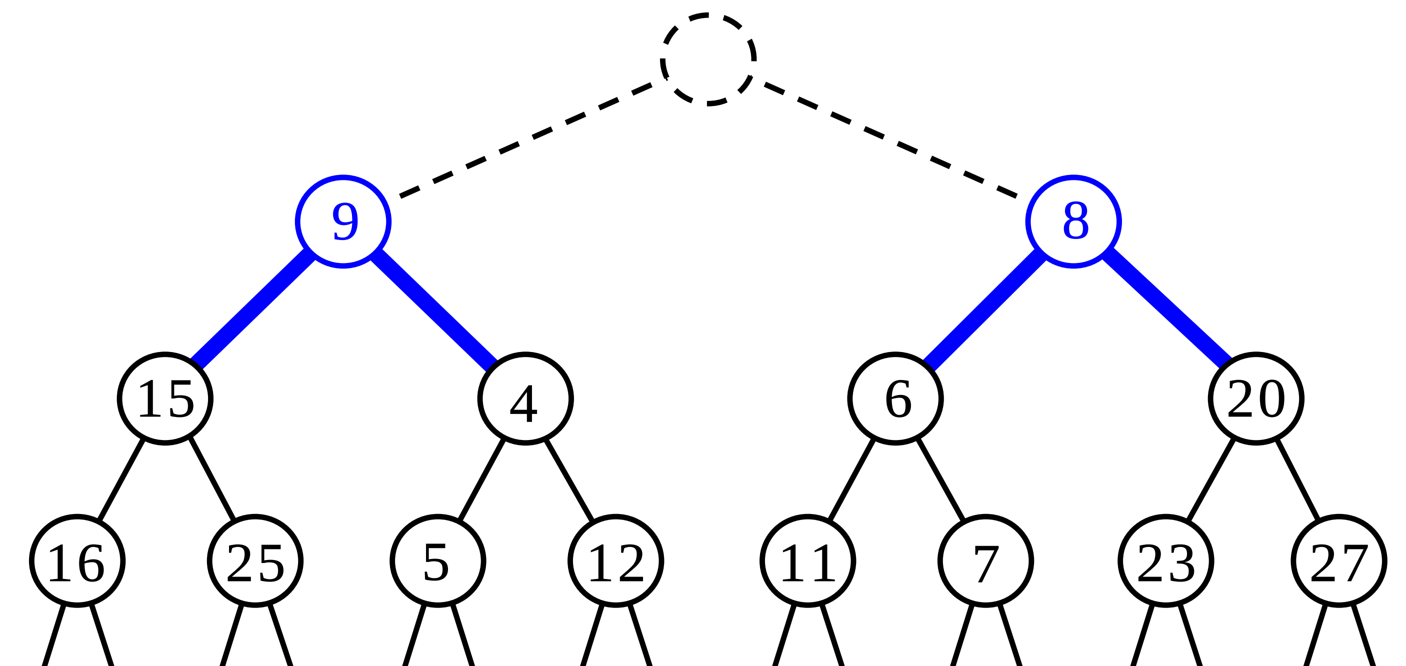


Bottom-Up Heap Construction

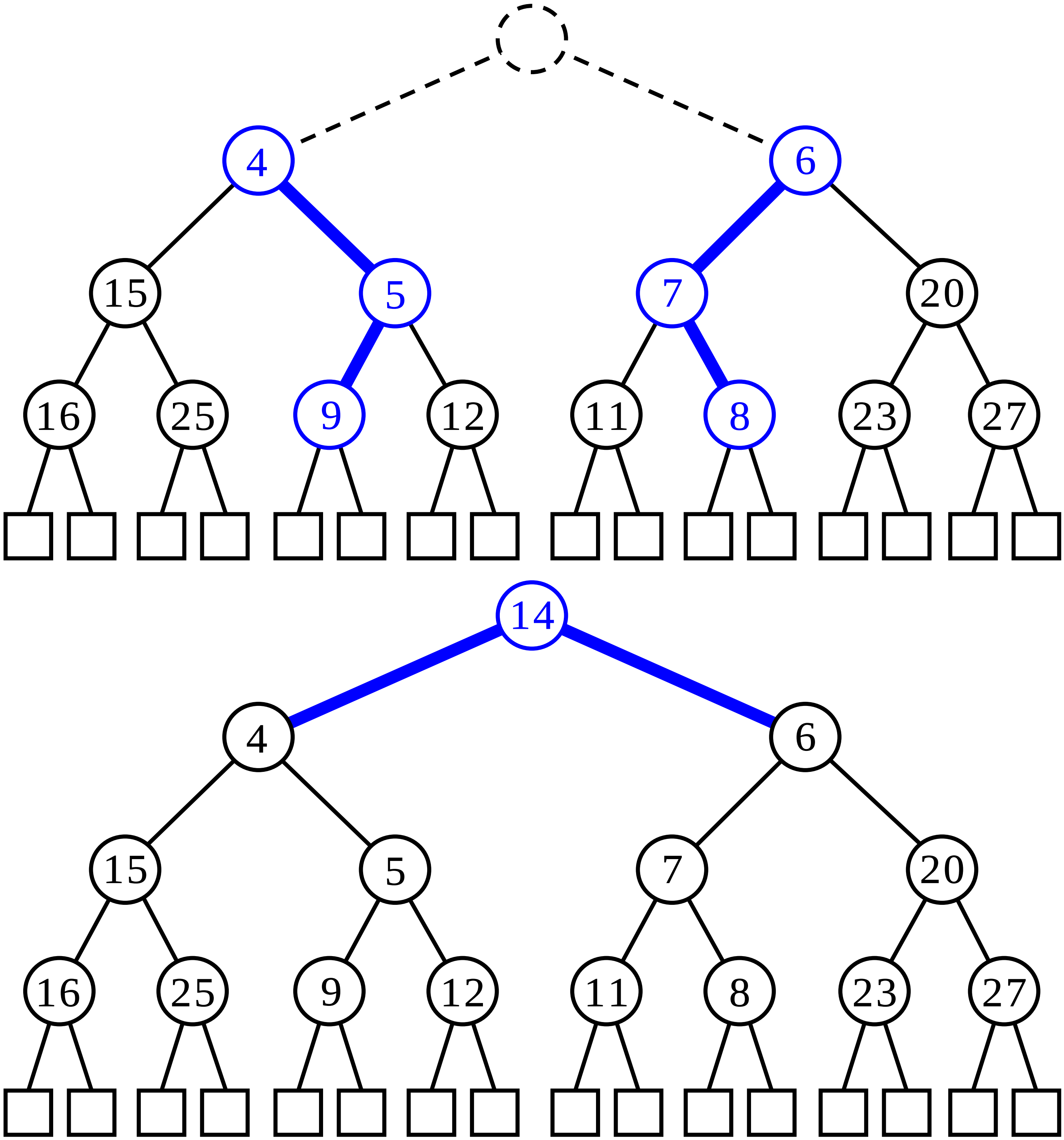
- ***downheap*** to preserve the order property



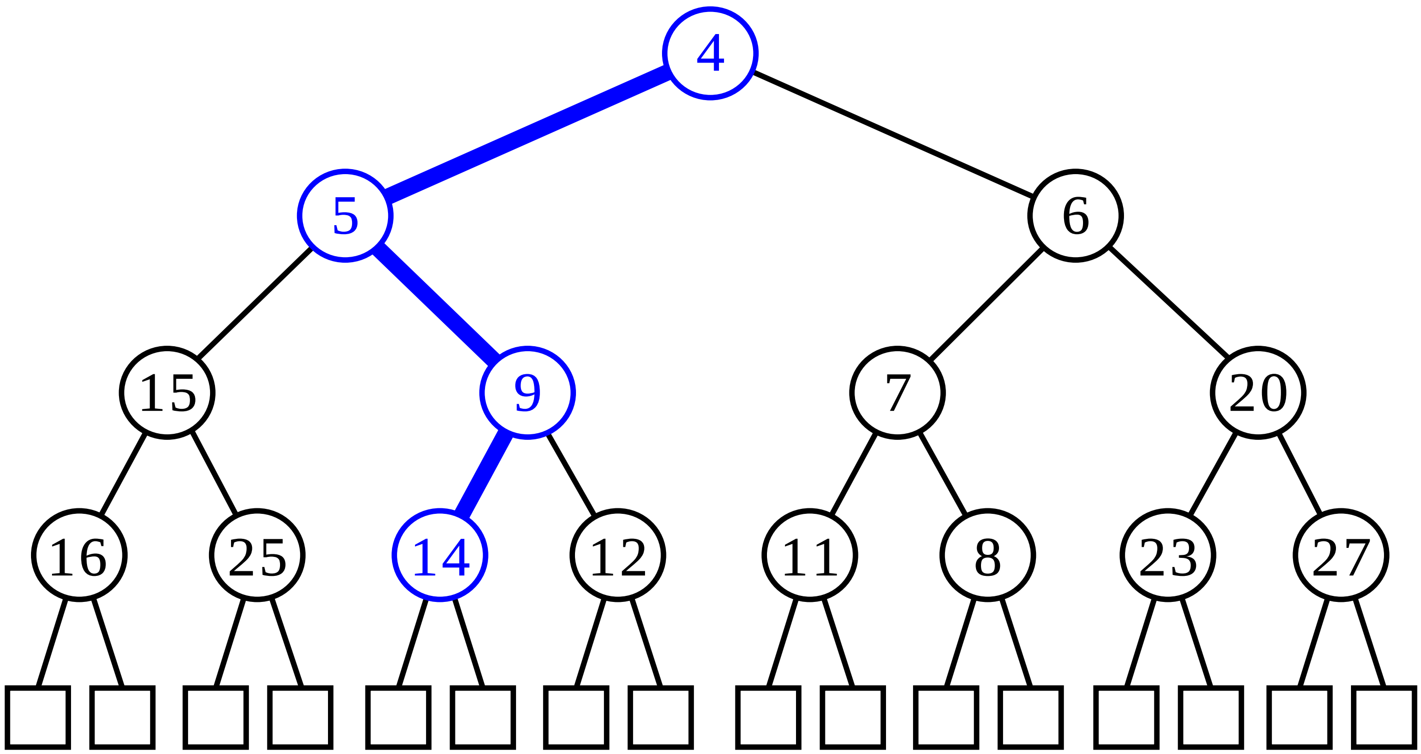
- now form seven-element heaps



Bottom-Up Heap Construction (cont.)



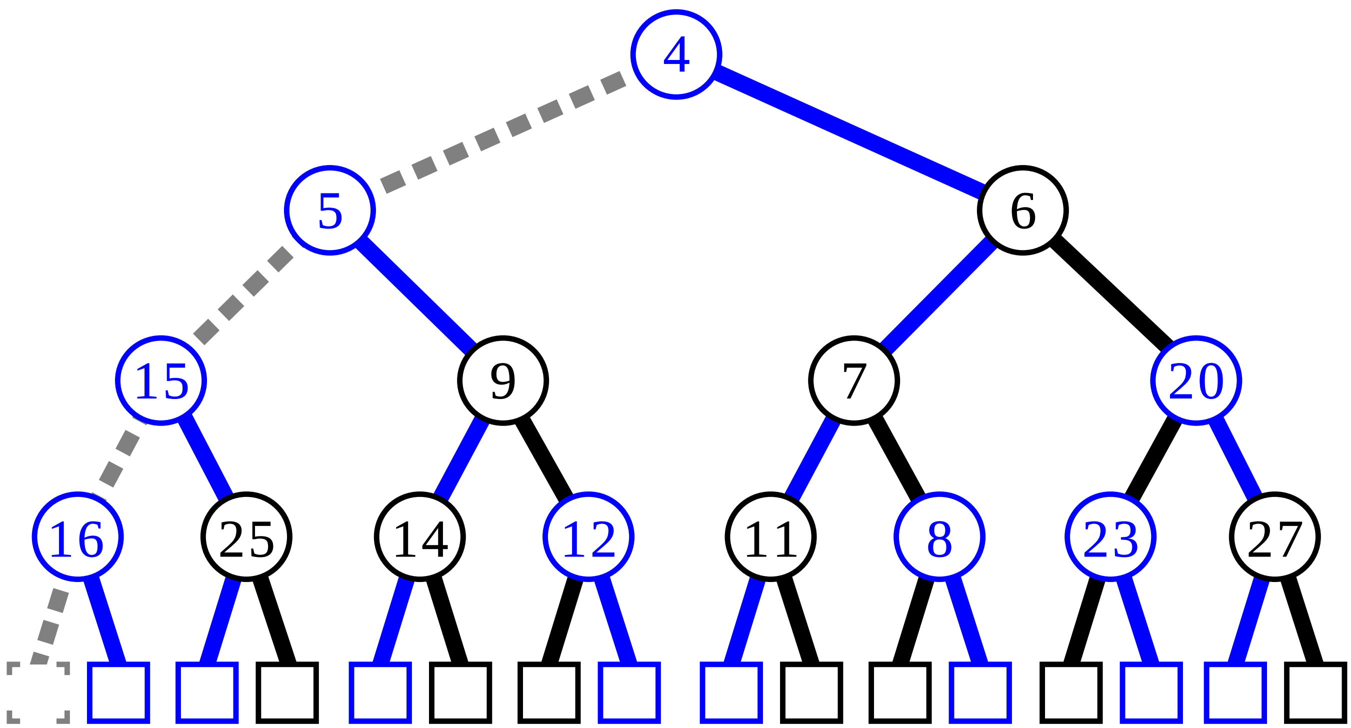
Bottom-Up Heap Construction (cont.)



The End

Analysis of Bottom-Up Heap Construction

- Proposition: Bottom-up heap construction with n keys takes $O(n)$ time.
  - Insert $(n + 1)/2$ nodes
  - Insert $(n + 1)/4$ nodes and downheap them
  - Insert $(n + 1)/8$ nodes and downheap them
  - ...
  - visual analysis:



- n inserts, $n/2$ upheaps with total $O(n)$ running time

Locators

- Locators can be used to keep track of elements as they are moved around inside a container.
- A ***locator*** sticks with a specific element, even if that element changes positions in the container.
- The locator ADT supports the following fundamental methods:
  - element(): return the element of the item associated with the locator.
  - key(): return the key of the item assocated with the locator.
- Using locators, we define additional methods for the priority queue ADT
  - insert(k,e): insert (k,e) into P and return its locator
  - min(): return the locator of an element witih smallest key
  - remove(l): remove the element with locator l
- In the stock trading application, we return a locator when an order is placed. The locator allows to specify unambiguously an order when a cancellation is requested

Positions and Locators

- At this point, you may be wondering what the difference is between locators and positions, and why we need to distinguish between them.
- It’s true that they have very similar methods
- The difference is in their primary usage
- Positions abstract the specific implementation of accessors to elements (indices vs. nodes).
- Positions are defined relatively to each other (e.g., previous-next, parent-child)
- Locators keep track of where elements are stored. In the implementation of an ADT withy locators, a locator typically holds the current position of the element.
- Locators associate elements with their keys

Locators and Positions at Work

- For example, consider the CS16 Valet Parking Service (started by the TA staff because they had too much free time on their hands).
- When they began their business, Andy and Devin decided to create a data structure to keep track of where exactly the cars were.
- Andy suggested having a ***position*** represent what ***parking space*** the car was in.
- However, Devin knew that the TAs were driving the customers' cars around campus and would not always park them back into the same spot.
- So they decided to install a ***locator*** (a ***wireless tracking device***) in each car. Each locator had a unique code, which was written on the claim check.
- When a customer demanded her car, the HTAs activated the locator. The horn of the car would honk and the lights would flash.
- If the car was parked, Andy and Devin would know where to retrieve it in the lot.
- Otherwise, the TA driving the car knew it was time to bring it back.