Keys and Total Order Relations

- A Priority Queue ranks its elements by ***key*** with a ***total order*** relation

- Keys:
  - Every element has its own key
  - Keys are not necessarily unique

- Total Order Relation
  - Denoted by $\leq$
  - **Reflexive:** $k \leq k$
  - **Antisymetric:** if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
  - **Transitive:** if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$

- A Priority Queue supports these fundamental methods:
  - insertItem(k, e) `// element e, key k`
  - removeMinElement() `// return and remove the // item with the smallest key`

Sorting with a Priority Queue

- A Priority Queue P can be used for sorting by inserting a set S of n elements and calling removeMinElement() until P is empty:

Algorithm PriorityQueueSort(S, P):
Input: A sequence S storing n elements, on which a total order relation is defined, and a Priority Queue P that compares keys with the same relation
Output: The Sequence S sorted by the total order relation

while !S.isEmpty() **do**
  $e \leftarrow S$.removeFirst()
  P.insertItem(e, e)
while P is not empty **do**
  $e \leftarrow P$.removeMinElement()
  S.insertLast(e)

The Priority Queue ADT

- A prioriy queue *P* must support the following methods:

  - size():
    Return the number of elements in *P*
    Input: None; **Output**: integer

  - isEmpty():
    Test whether *P* is empty
    Input: None; **Output**: boolean

  - insertItem(*k*,*e*):
    Insert a new element *e* with key *k* into *P*
    Input: Objects *k*, *e* **Output**: None

  - minElement():
    Return (but don't remove) an element of *P* with smallest key; an error occurs if *P* is empty.
    Input: None; **Output**: Object *e*

The Priority Queue ADT (contd.)

- minKey():

  Return the smallest key in *P*; an error occurs if *P* is empty

  Input: None; **Output**: Object *k*

- removeMinElement():

  Remove from *P* and return an element with the smallest key; an error condidtion occurs if *P* is empty.

  Input: None; **Output**: Object *e*

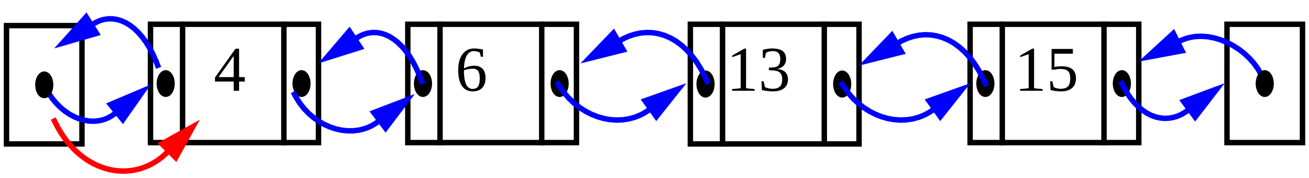
Comparators

- The most general and reusable form of a priority queue makes use of **comparator** objects.
- Comparator objects are external to the keys that are to be compared and compare two objects.
- When the priority queue needs to compare two keys, it uses the comparator it was given to do the comparison.
- Thus a priority queue can be general enough to store any object.
- The comparator ADT includes:
  - isLessThan(*a*, *b*)
  - isLessThanOrEqualTo(*a*,*b*)
  - isEqualTo(*a*, *b*)
  - isGreaterThan(*a*,*b*)
  - isGreaterThanOrEqualTo(*a*,*b*)
  - isComparable(*a*)

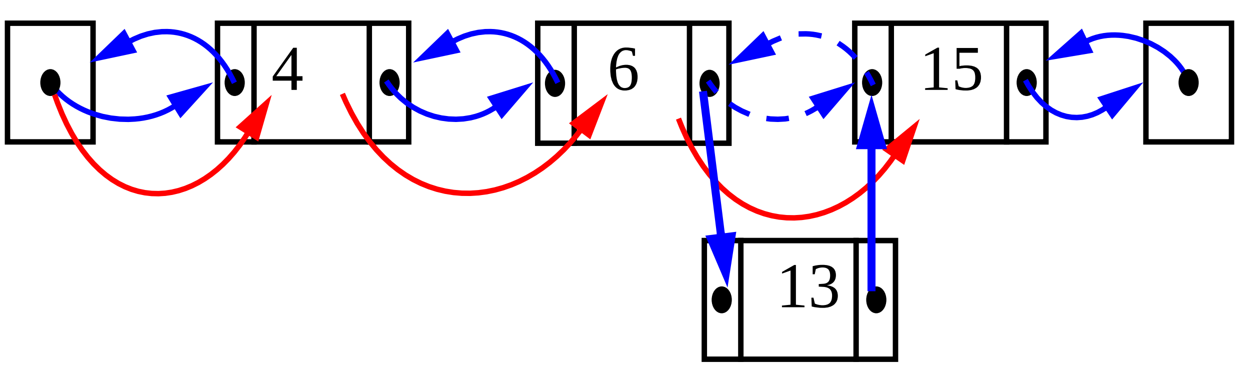
Implementation with a Sorted Sequence

- Another implementation uses a sequence *S*, sorted by keys, such that the first element of *S* has the smallest key.

- We can implement minElement(), minKey(), and removeMinElement() by accessing the first element of *S*. Thus these methods are ***O***(1) (assuming our sequence has an ***O***(1) front-removal)



- However, these advantages comes at a price. To implement insertItem(), we must now scan through the entire sequence. Thus insertItem() is ***O***(n).



Implementation with a Sorted Sequence(contd.)

```
public class SequenceSimplePriorityQueue
implements SimplePriorityQueue {
  //Implementation of a priority queue
  using a sorted sequence
 protected Sequence seq = new NodeSequence();
 protected Comparator comp;
 // auxiliary methods
 protected Object extractKey (Position pos) {
   return ((Item)pos.element()).key();
}
 protected Object extractElem (Position pos) {
   return ((Item)pos.element()).element();
 }
 protected Object extractElem (Object key) {
   return ((Item)key).element();
 }
  // methods of the SimplePriorityQueue ADT
 public SequenceSimplePriorityQueue (Comparator c) {
    this.comp = c; }
 public int size () {return seq.size(); }
```

Implementation with a Sorted Sequence(contd.)

```
public boolean isEmpty () { return seq.isEmpty(); }
public void insertItem (Object k, Object e) throws
InvalidKeyException {
    if (!comp.isComparable(k))
      throw new InvalidKeyException("The key is not
valid");
    else
      if (seq.isEmpty())
        seq.insertFirst(new Item(k,e));
      else
          if (comp.isGreaterThan(k,extractKey(seq.last())))
            seq.insertAfter(seq.last(),new Item(k,e));
        else {
          Position curr = seq.first();
          while (comp.isGreaterThan(k,extractKey(curr)))
            curr = seq.after(curr);
          seq.insertBefore(curr,new Item(k,e));
        }
  }
```

Implementation with a Sorted Sequence(contd.)

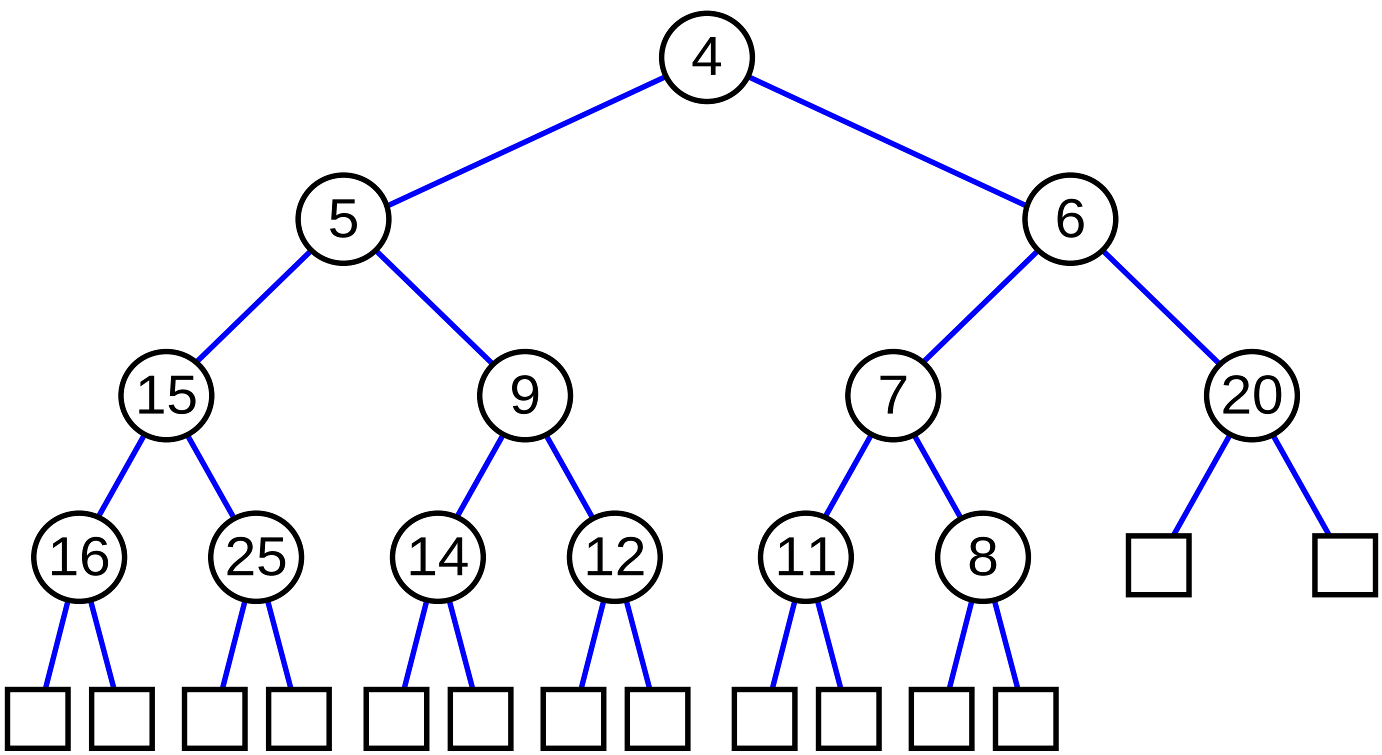
```
public Object minElement () throws
EmptyContainerException {
    if (seq.isEmpty())
      throw new EmptyContainerException("The priority
         queue is empty");
    else
      return extractElem(seq.first());
}
```

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) Complete Binary Tree Property
- Heap-Order Property Property(Relational): In a heap *H*, for every node *v* (except the root), the key stored in v is greater than or equal to the key stored in *v*'s parent.
- Complete Binary Tree Property (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.

Heaps (contd.)

- An Example:



Height of a Heap

- Proposition: A heap H storing n keys has height

$$\boldsymbol{h = \lceil \log(n+1) \rceil}$$

- Justification: Due to H being complete, we know:
  - \# i of internal nodes is at least :

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

  - \# i of internal nodes is at most:

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

  - Therefore:

    $$2^{h-1} \leq n \text{ and } n \leq 2^{h} - 1$$

  - Which implies that:

    $$\log(n + 1) \leq h \leq \log n + 1$$

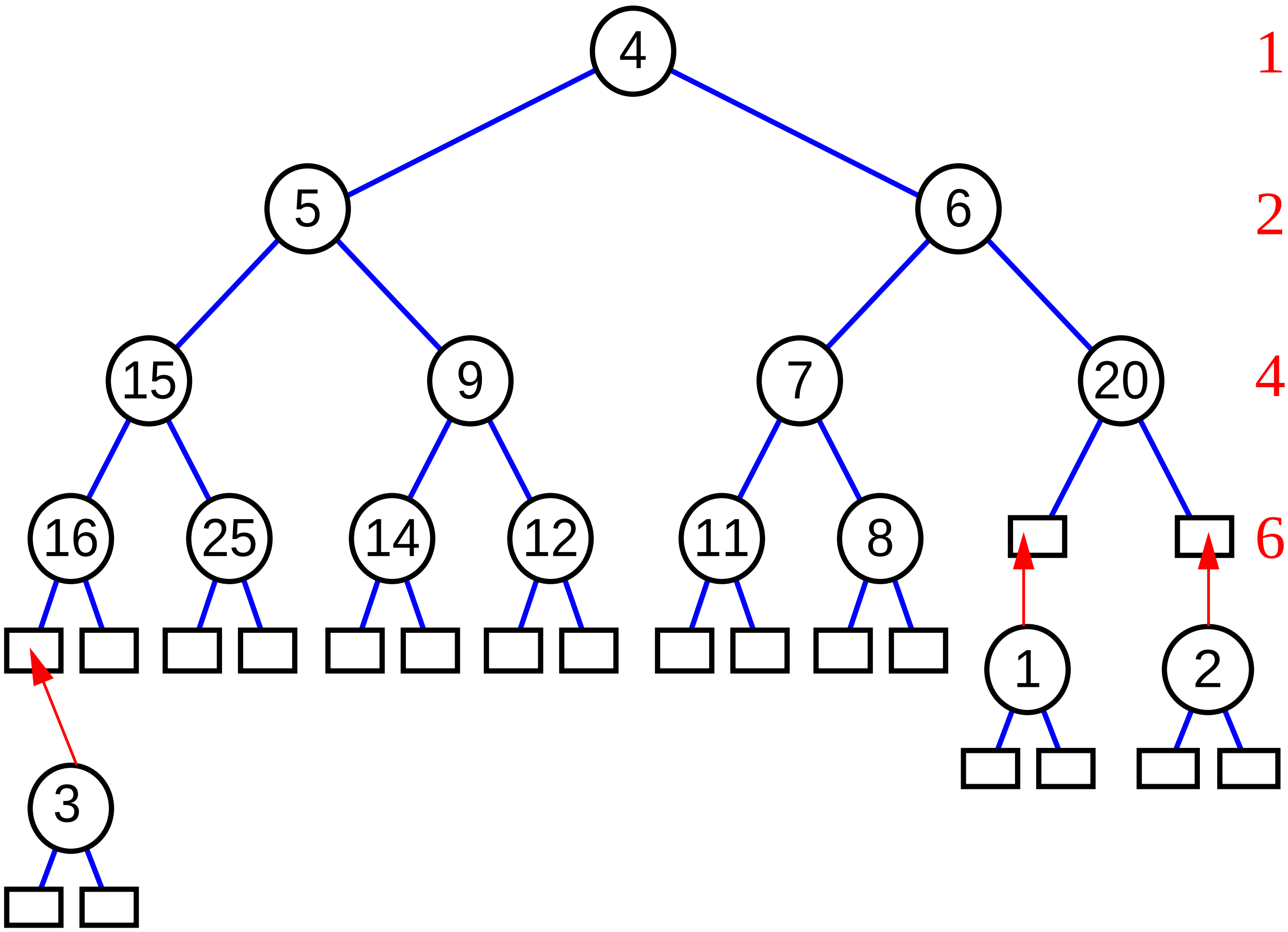
  - Which in turn implies:

    $$\boldsymbol{h = \lceil \log(n+1) \rceil}$$

  - Q.E.D.

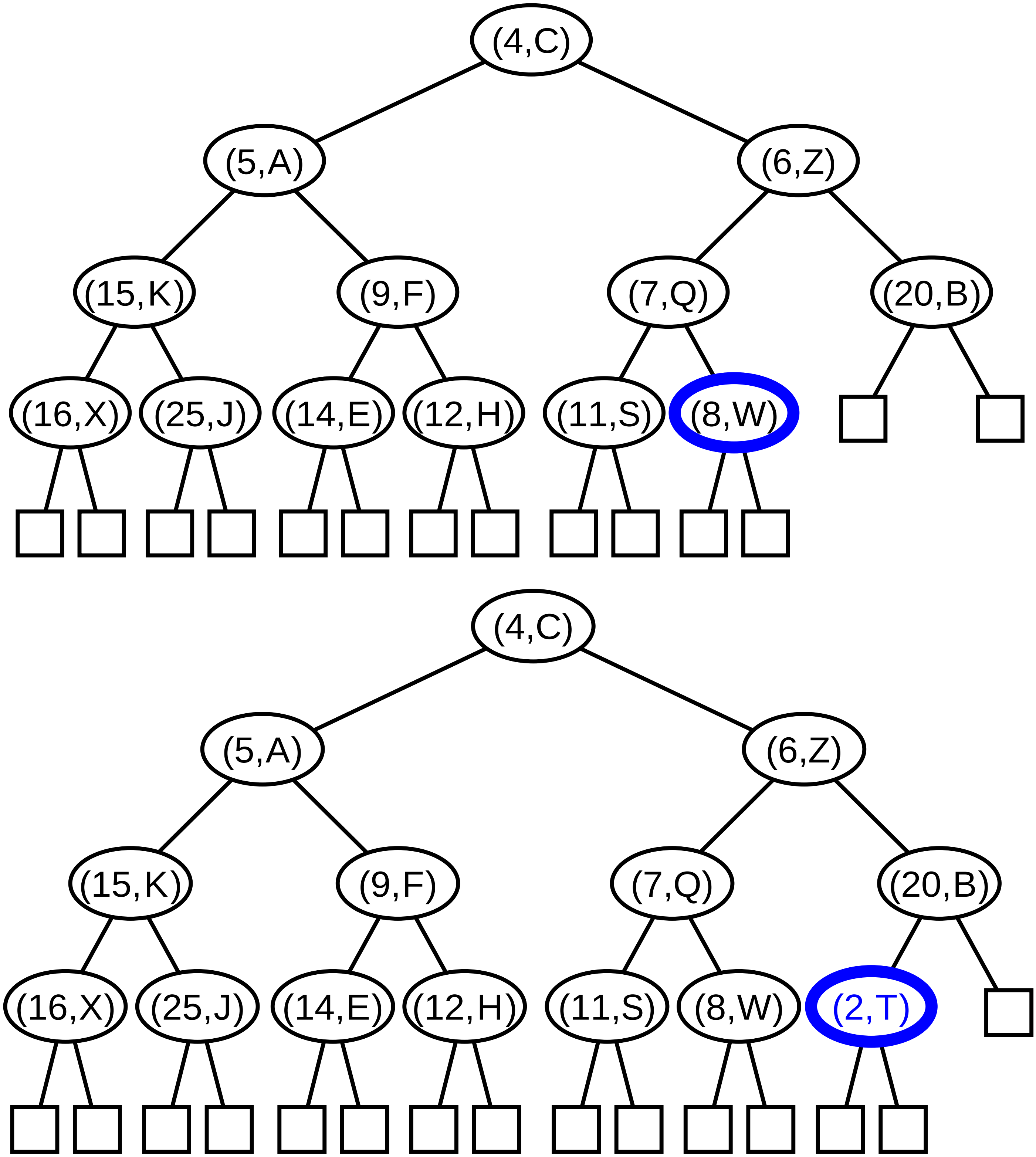
Heigh of a Heap (contd.)

- Let's look at that graphically:

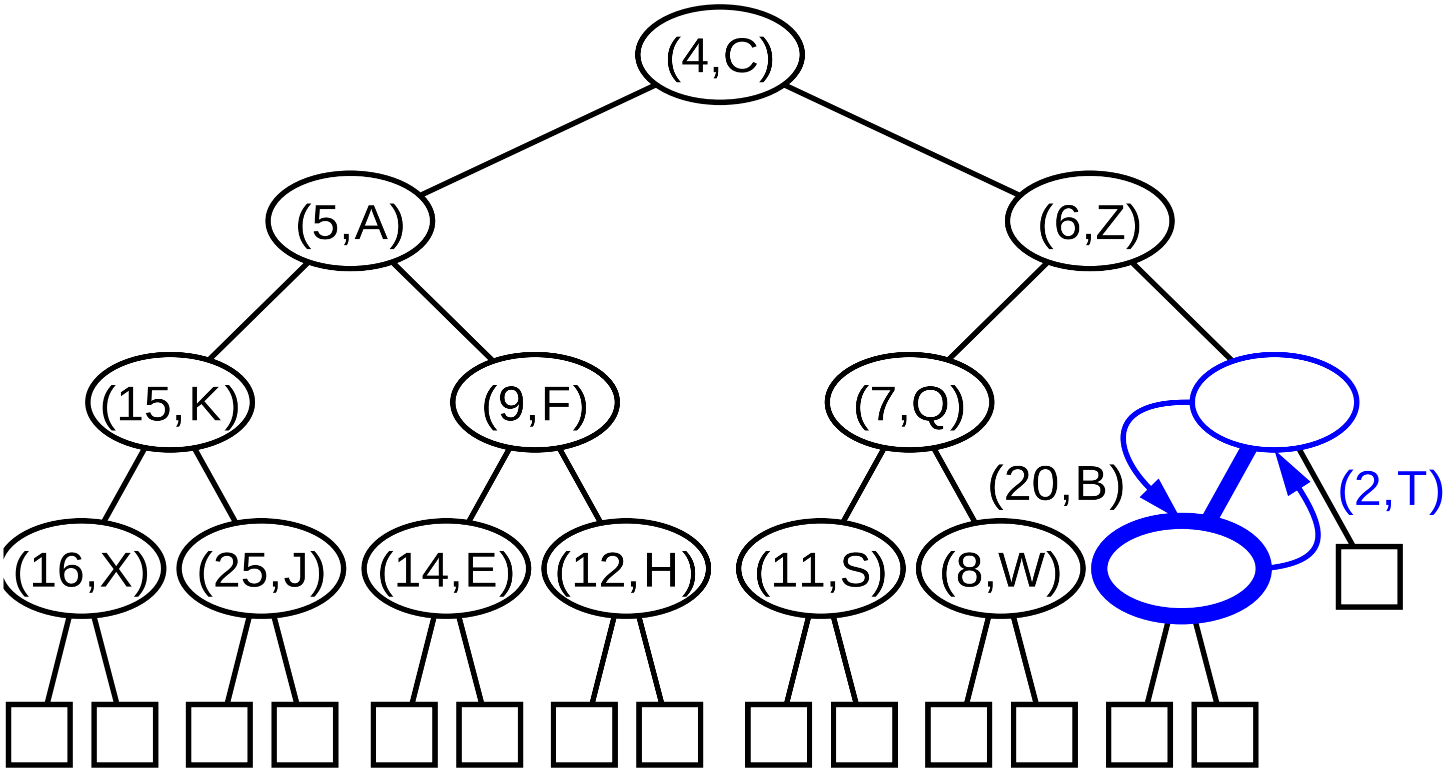


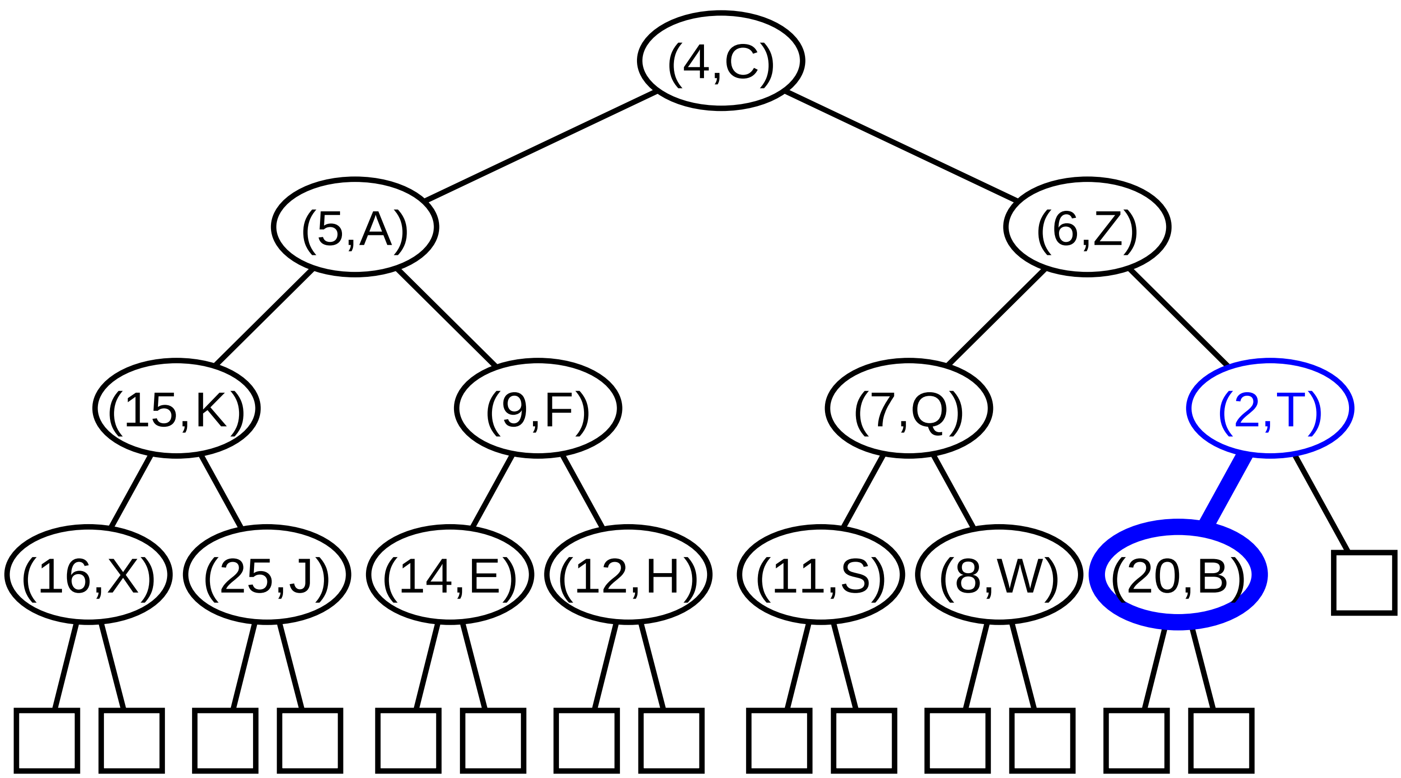
- Consider this heap which has height $h = 4$ and $n = 13$
- Suppose two more nodes are added. To maintain completeness of the tree, the two external nodes in level 4 will become internal nodes: i.e. $n = 15$, $h = 4 = \log(15+1)$
- Add one more: $n = 16$, $h = 5 = \lceil \log(16+1) \rceil$

Insertion into a Heap

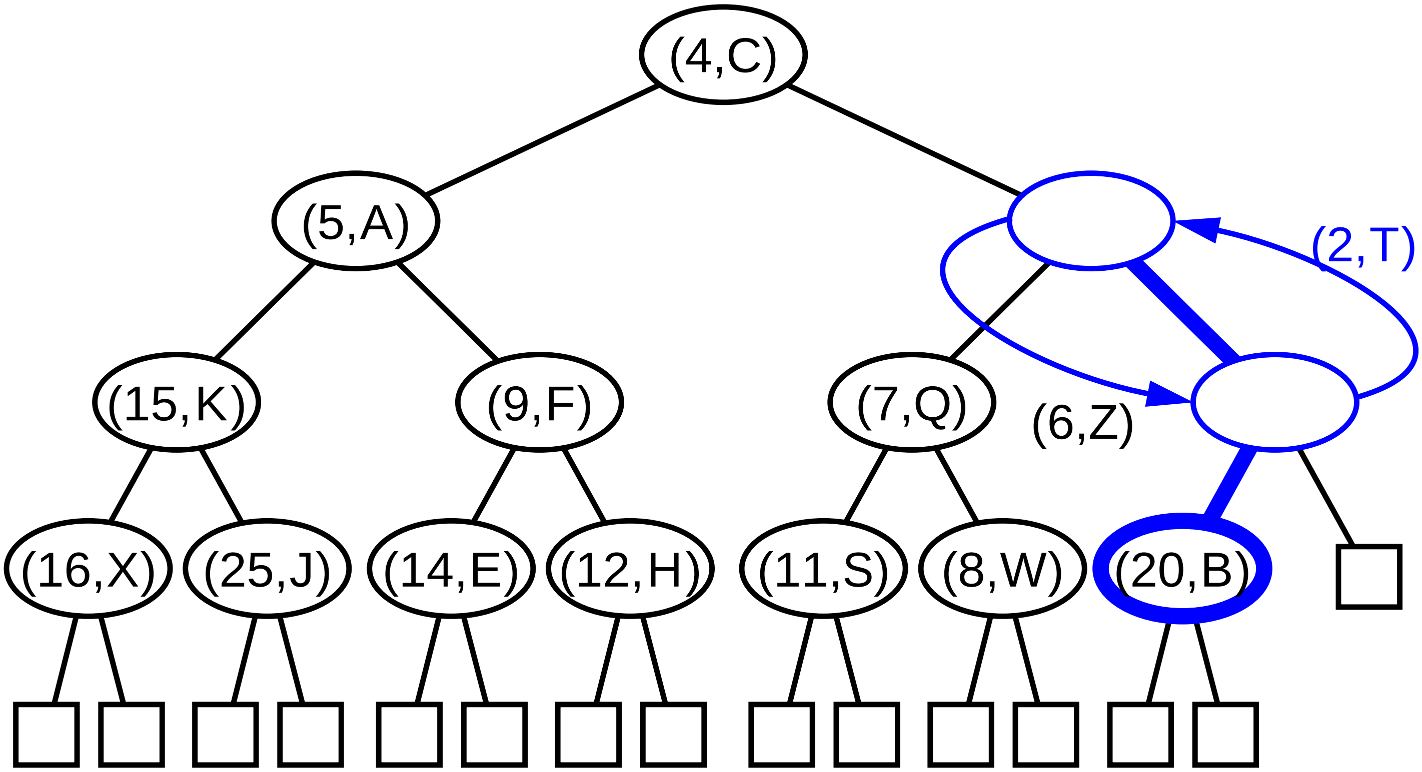


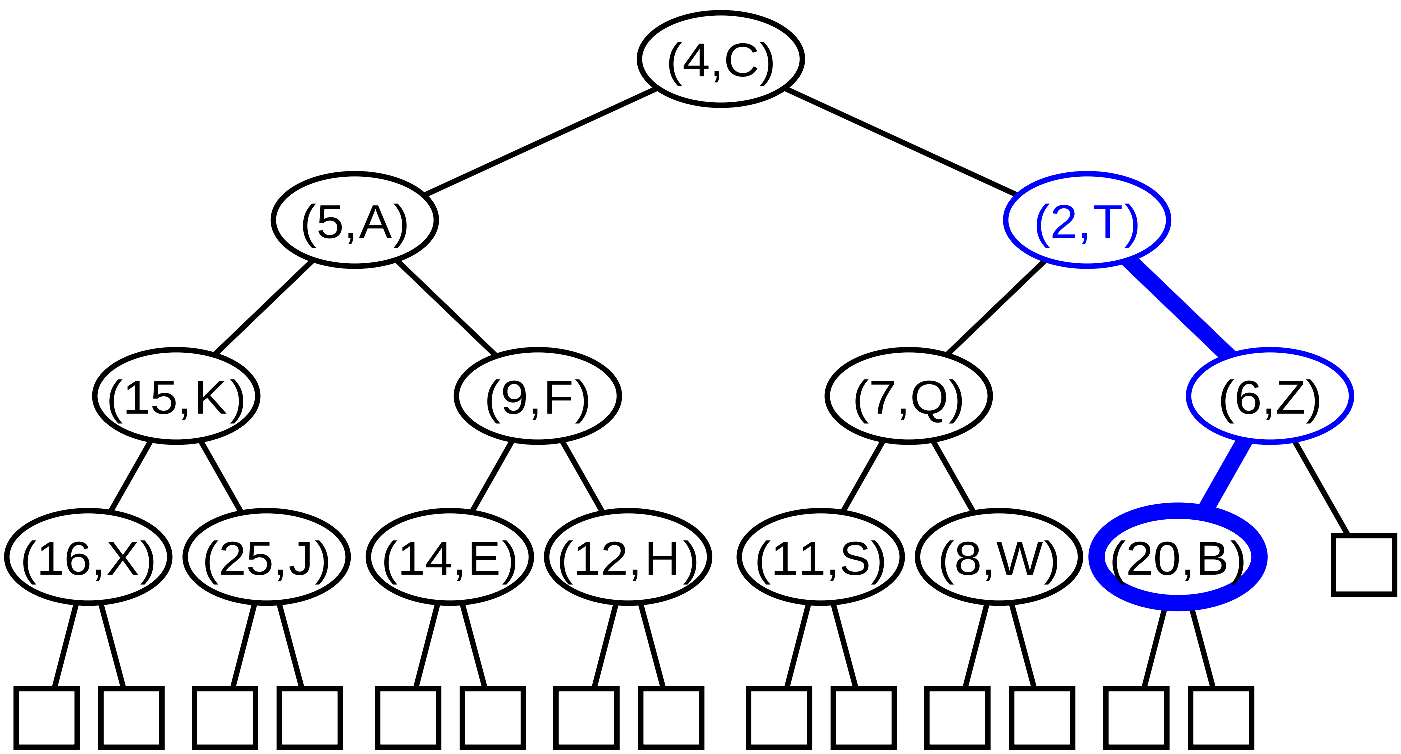
Insertion into a Heap (cont.)



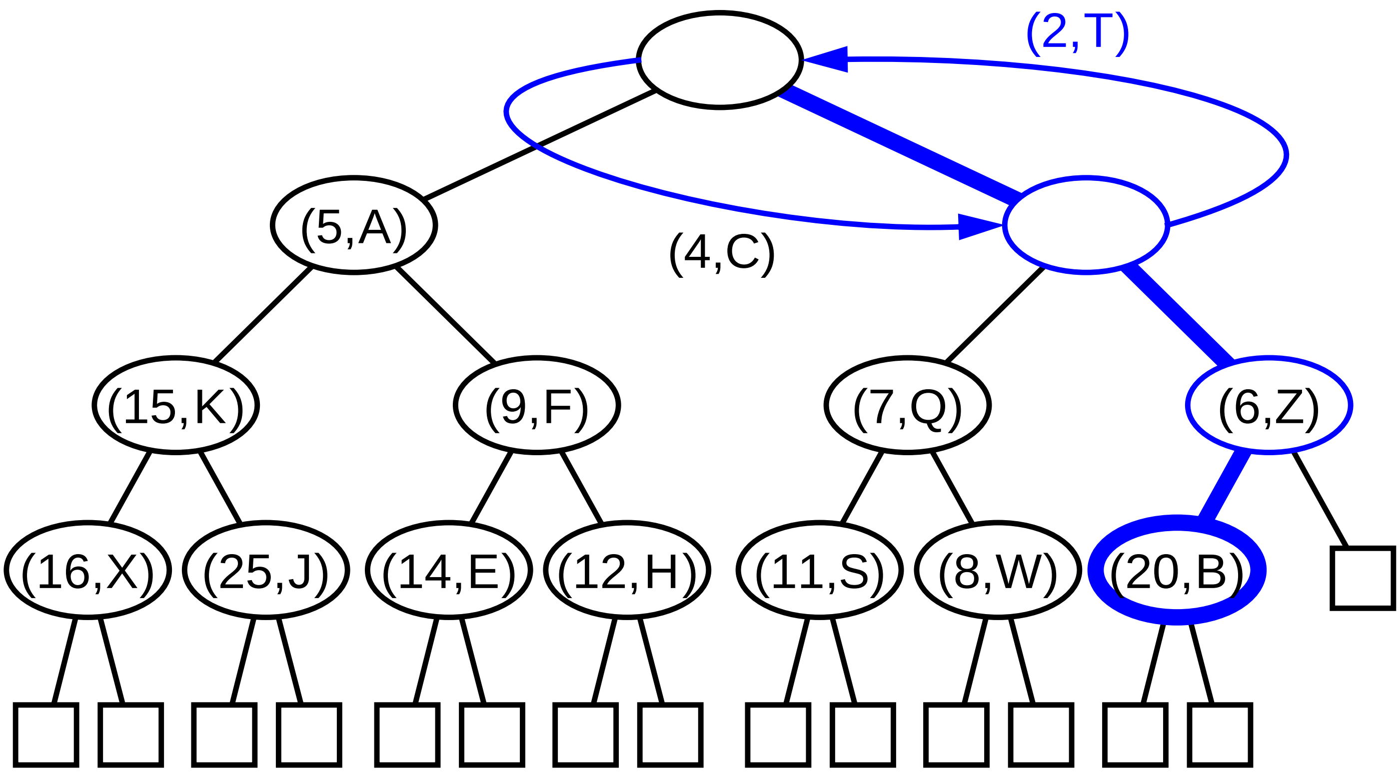


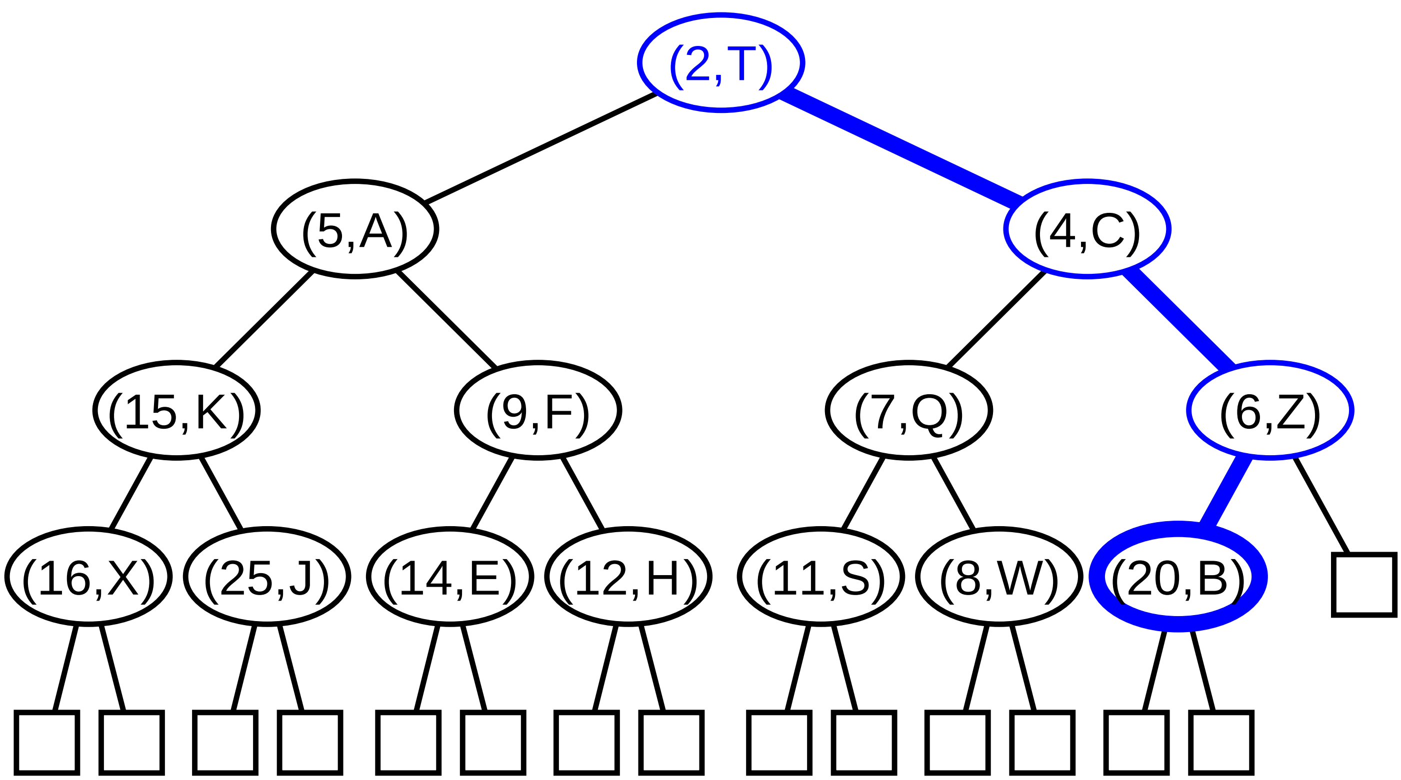
Insertion into a Heap (cont.)



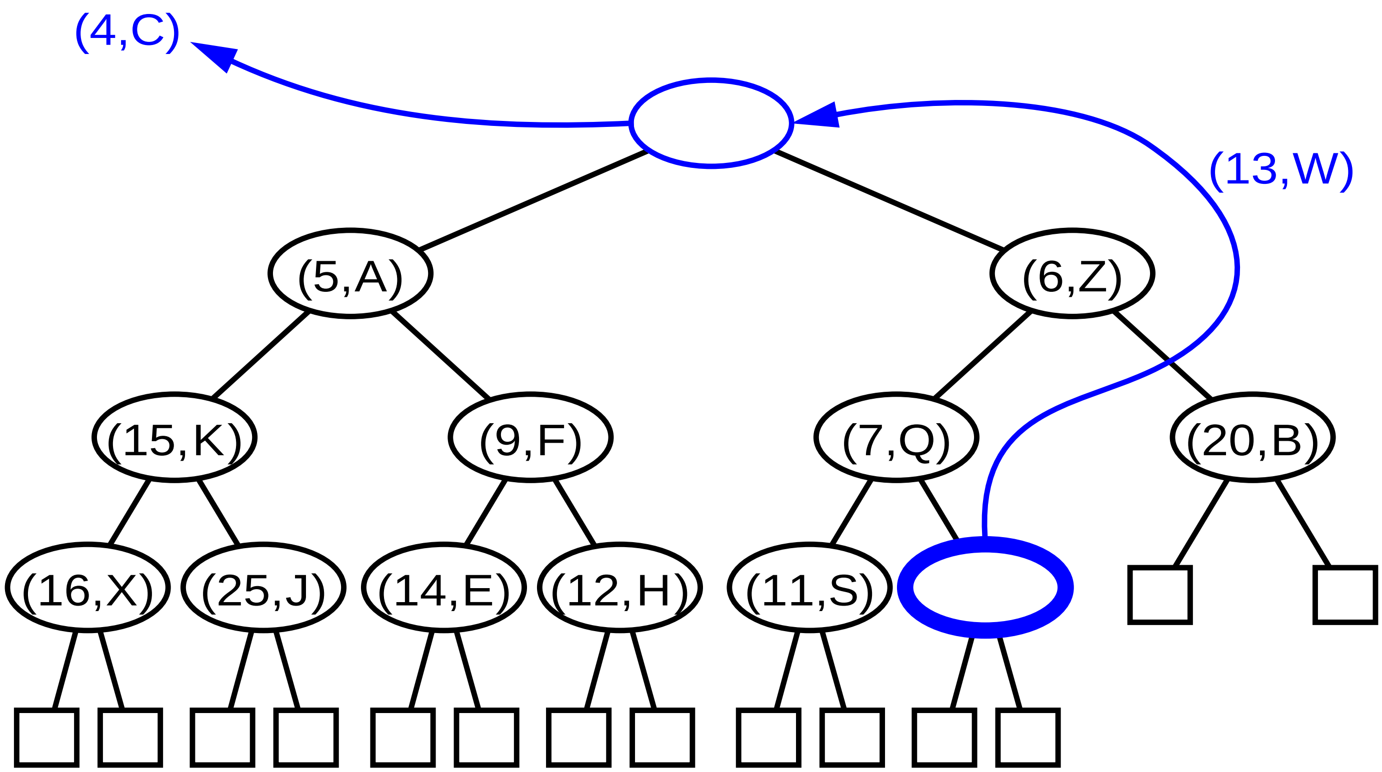


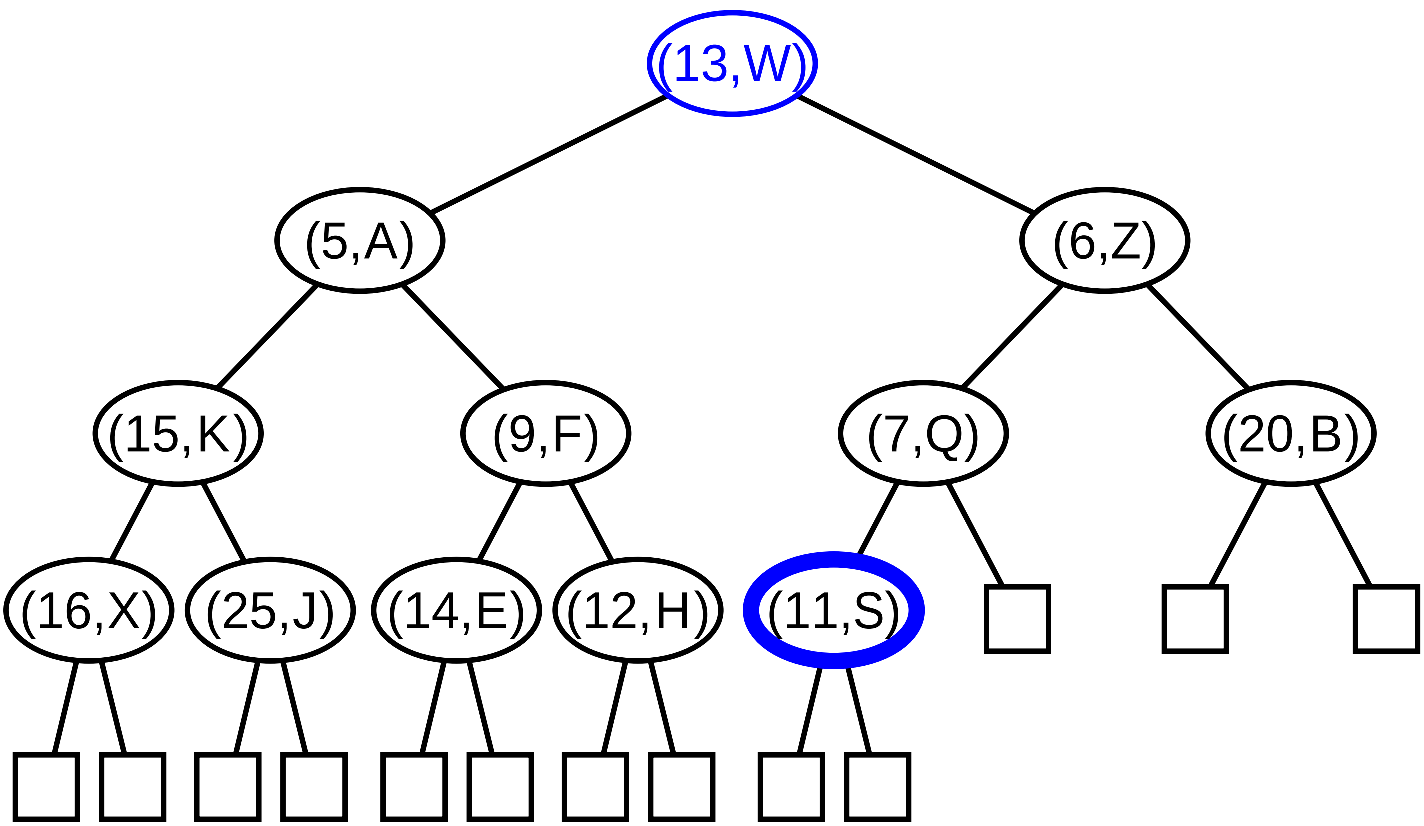
Insertion into a Heap (cont.)



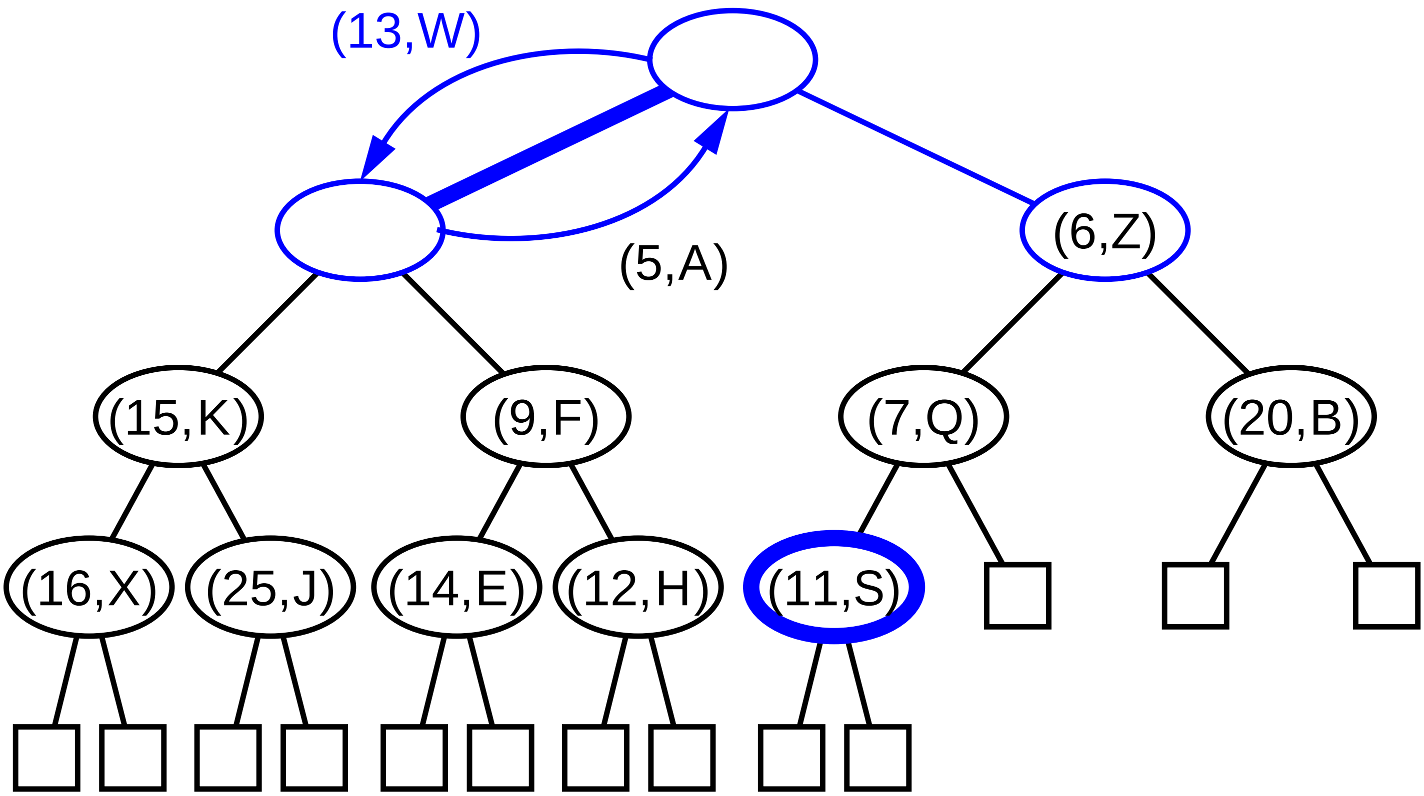


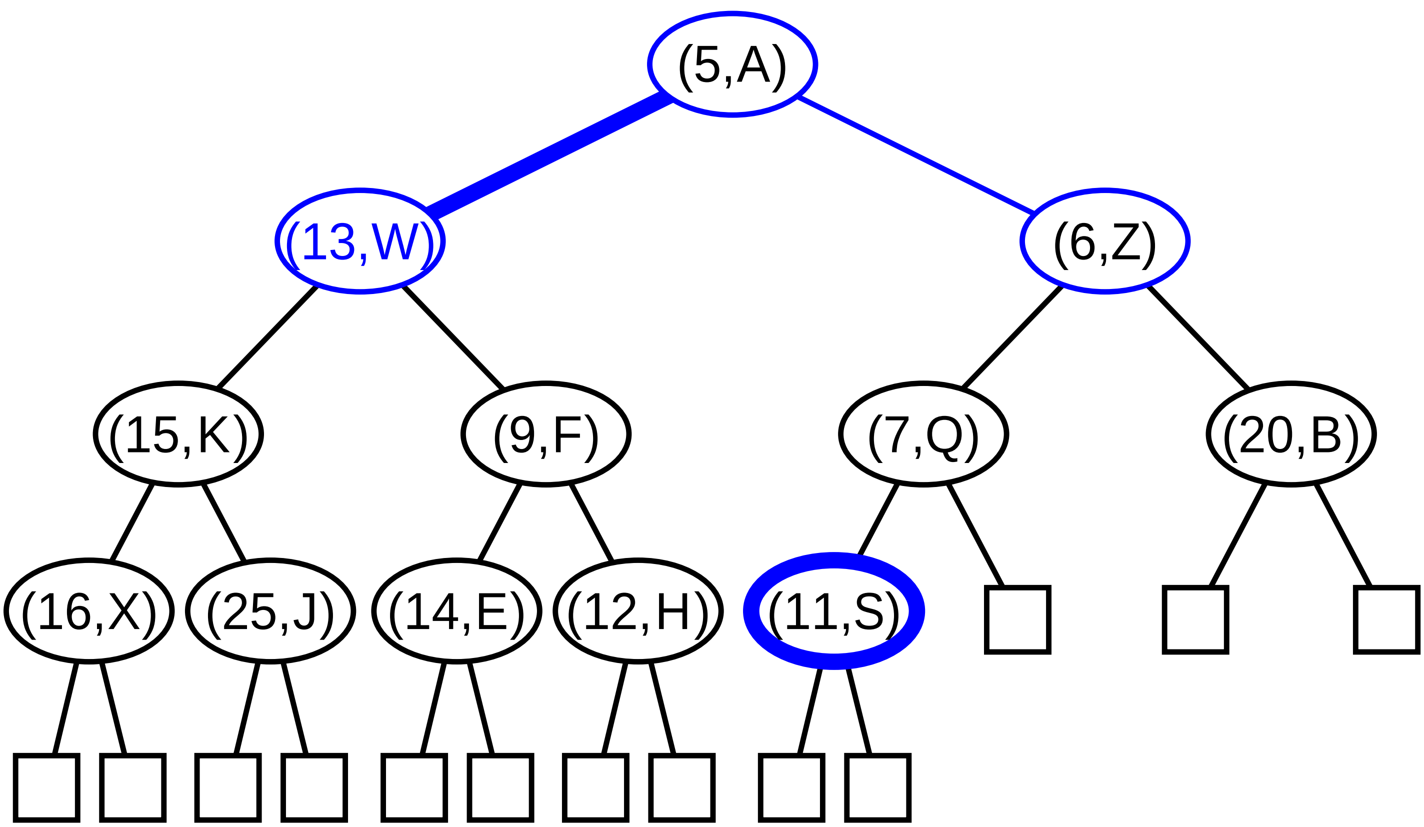
Removal from a Heap



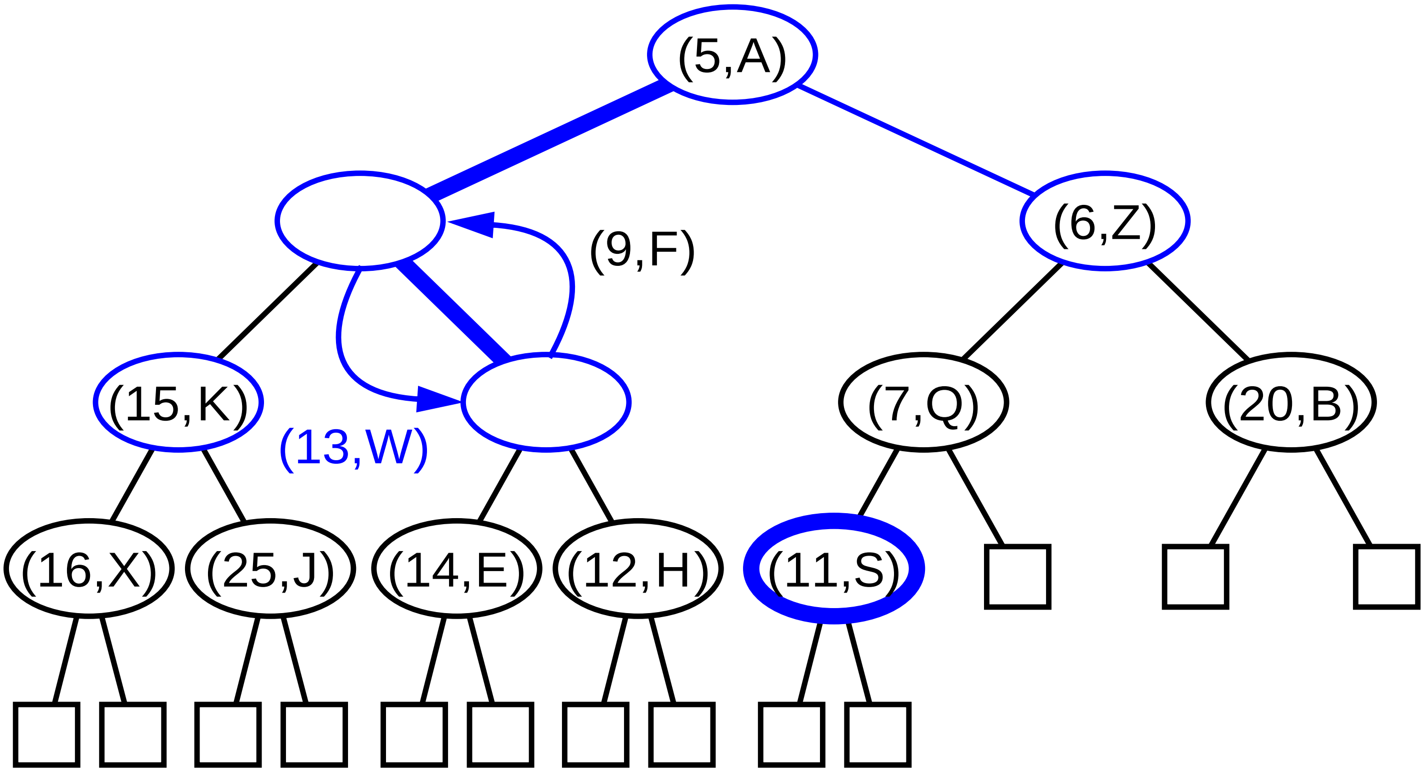


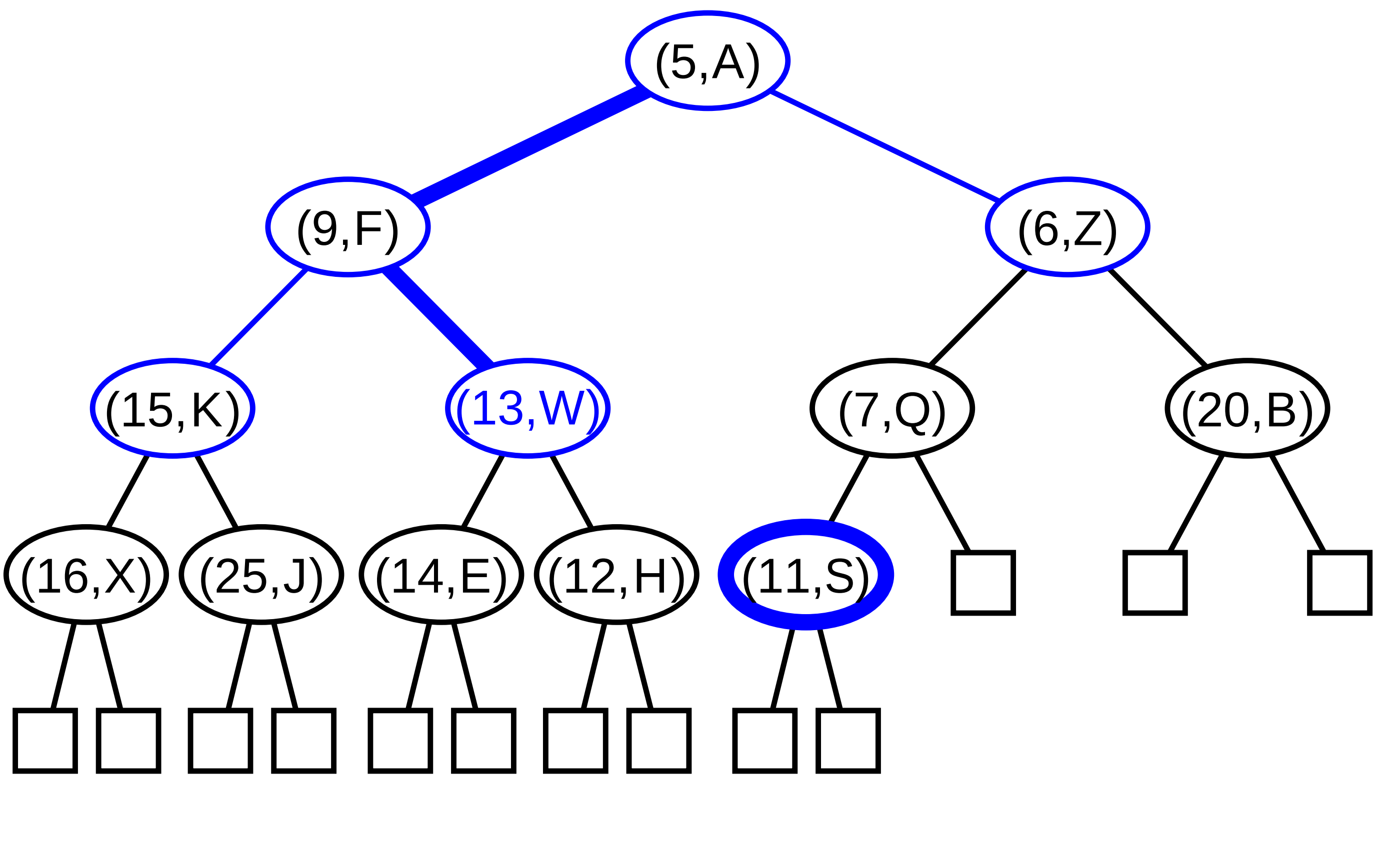
Removal from a Heap (cont.)



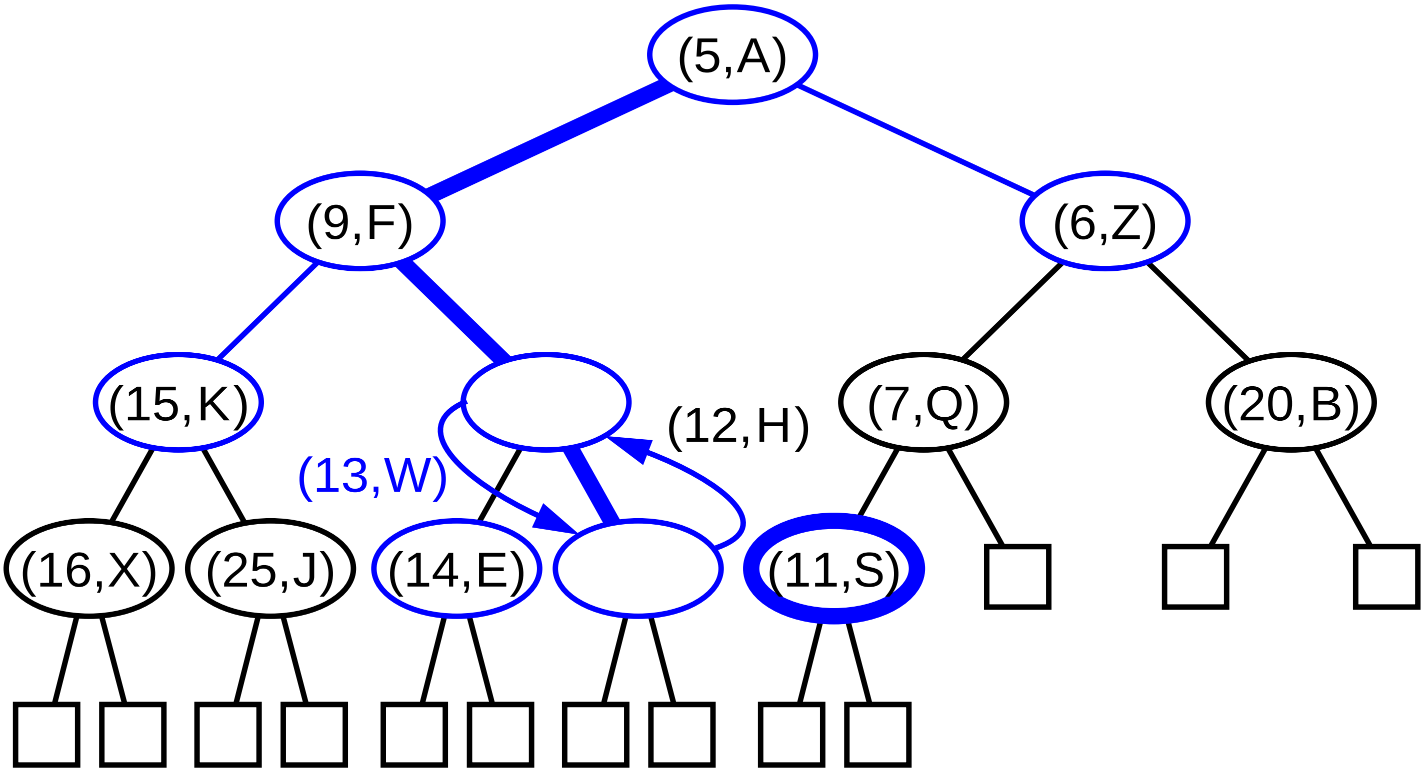


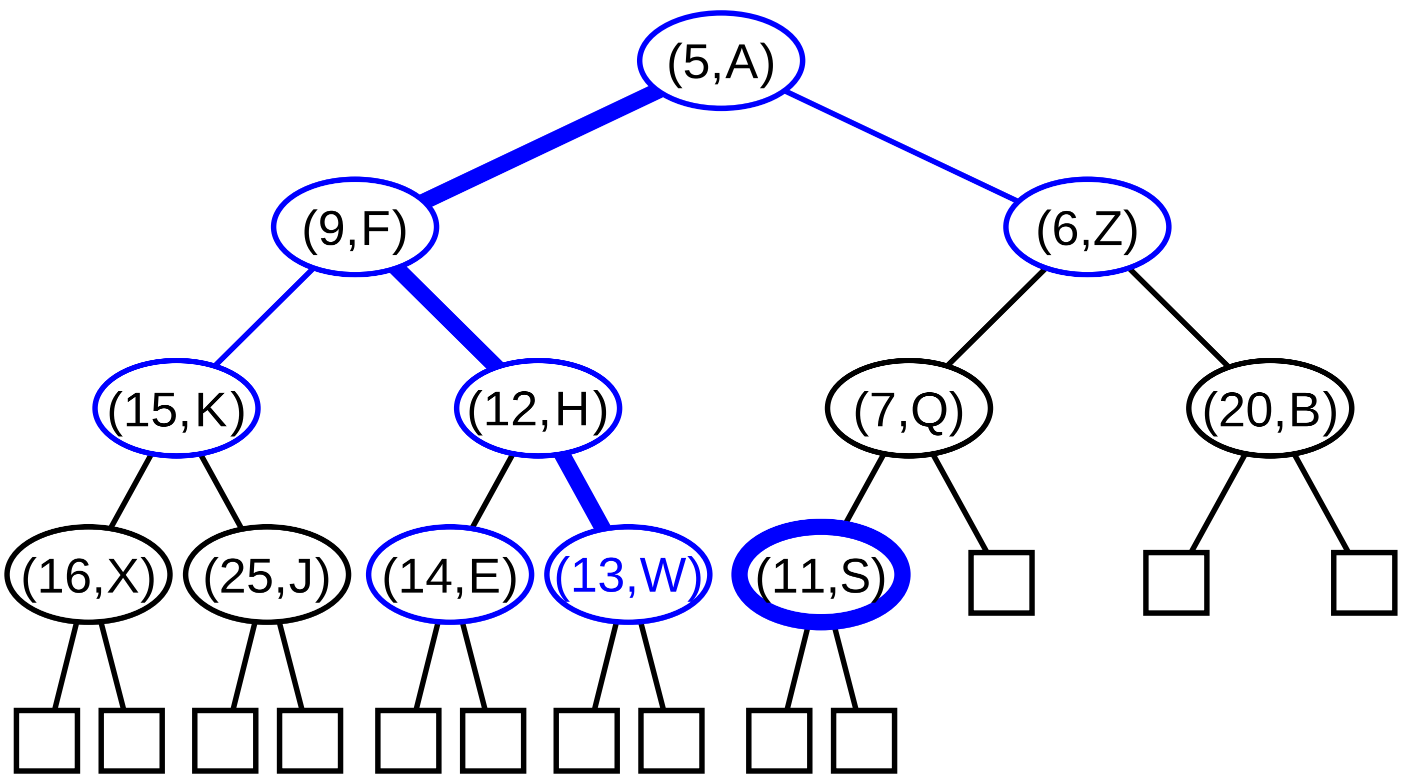
Removal from a Heap (cont.)





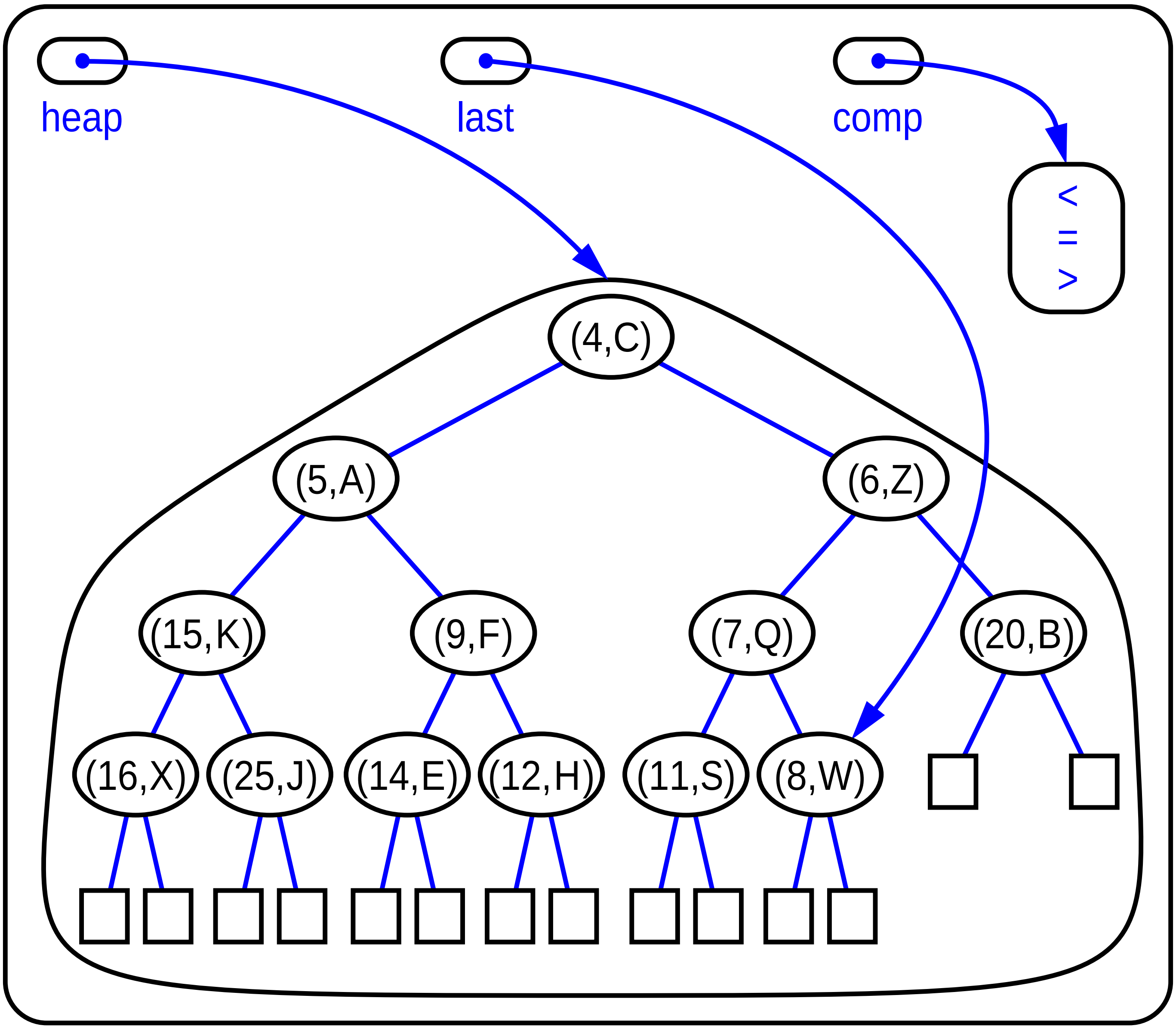
Removal from a Heap(cont.)





Implementation of a Heap

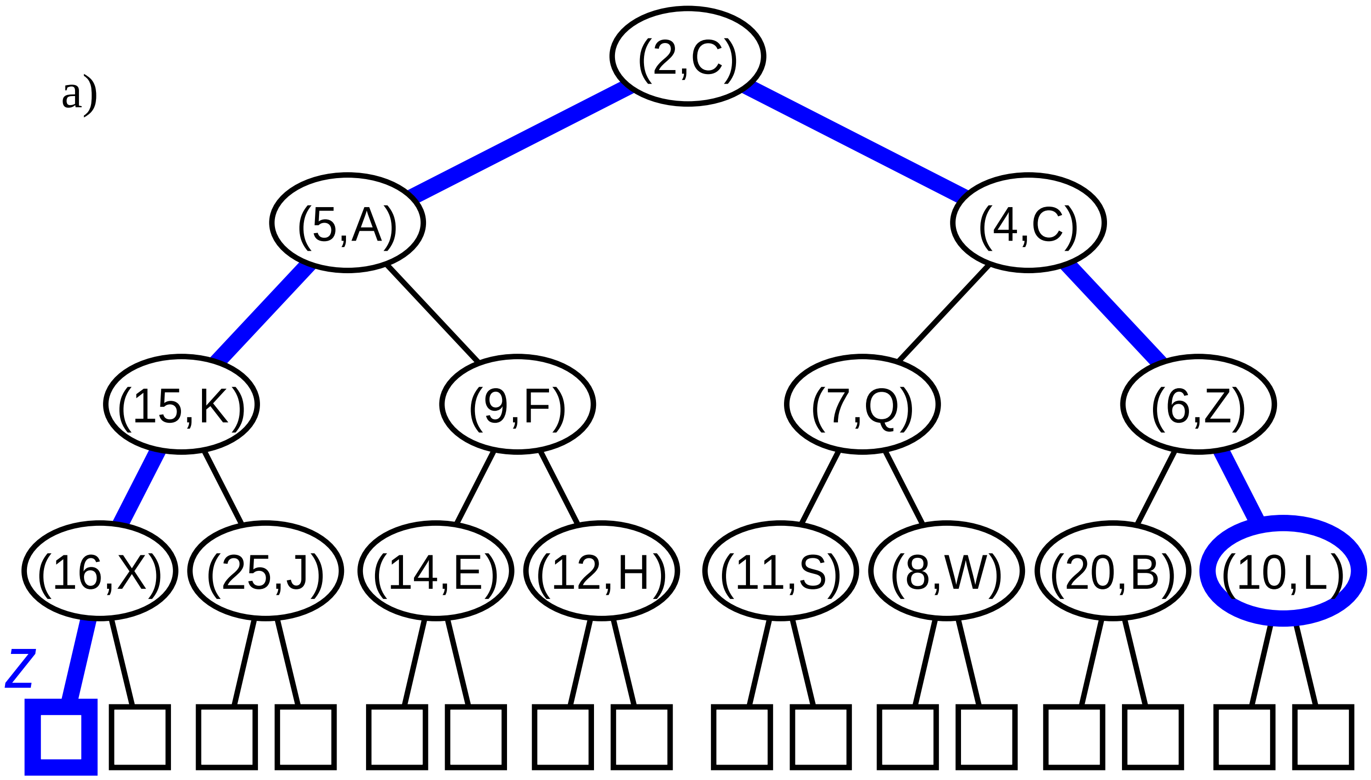
```
public class HeapSimplePriorityQueue implements
   SimplePriorityQueue {
   BinaryTree T;
   Position last;
   Comparator comparator;
   ...
}
```



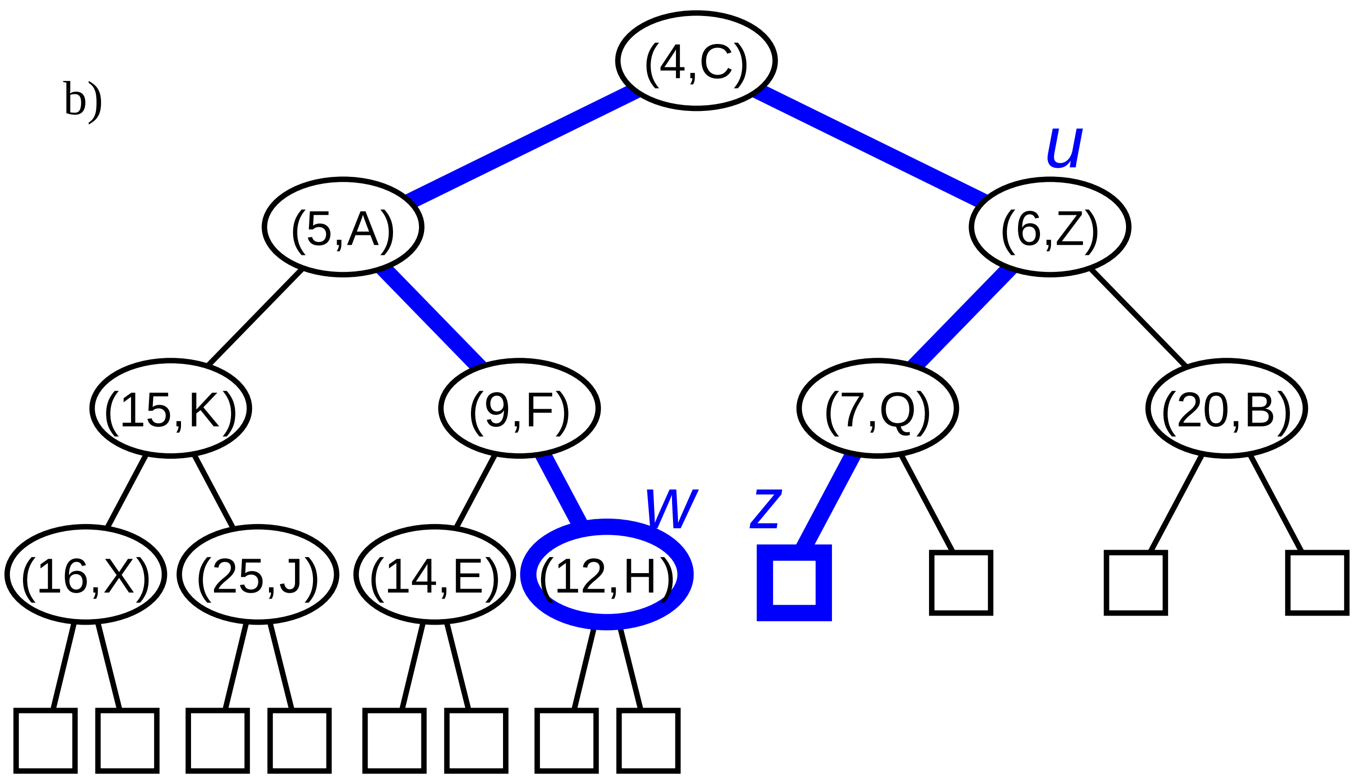
Implementation of a Heap(cont.)

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

a)



b)



Heap Sort

- All heap methods run in logarithmic time or better
- If we implement PriorityQueueSort using a heap for our priority queue, insertItem and removeMinElement each take O(log*k*), *k* being the number of elements in the heap at a given time.
- We always have n or less 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.

Bottom-Up Heap Construction

- If all the keys to be stored are given in advance we can build a heap bottom-up in O(n) time.
- Note: for simplicity, we describe bottom-up heap construction for the case for n keys where:

$$n = 2^h - 1$$

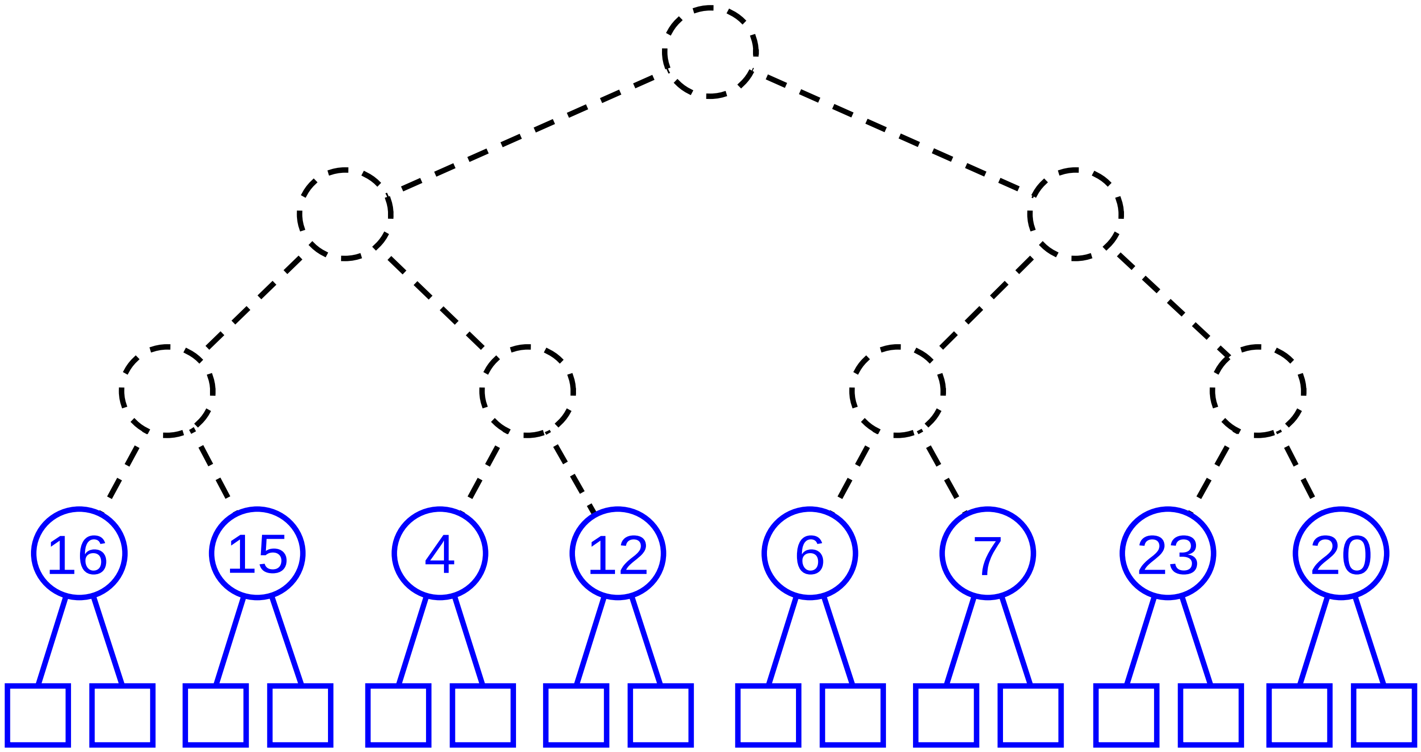
  h being the height.

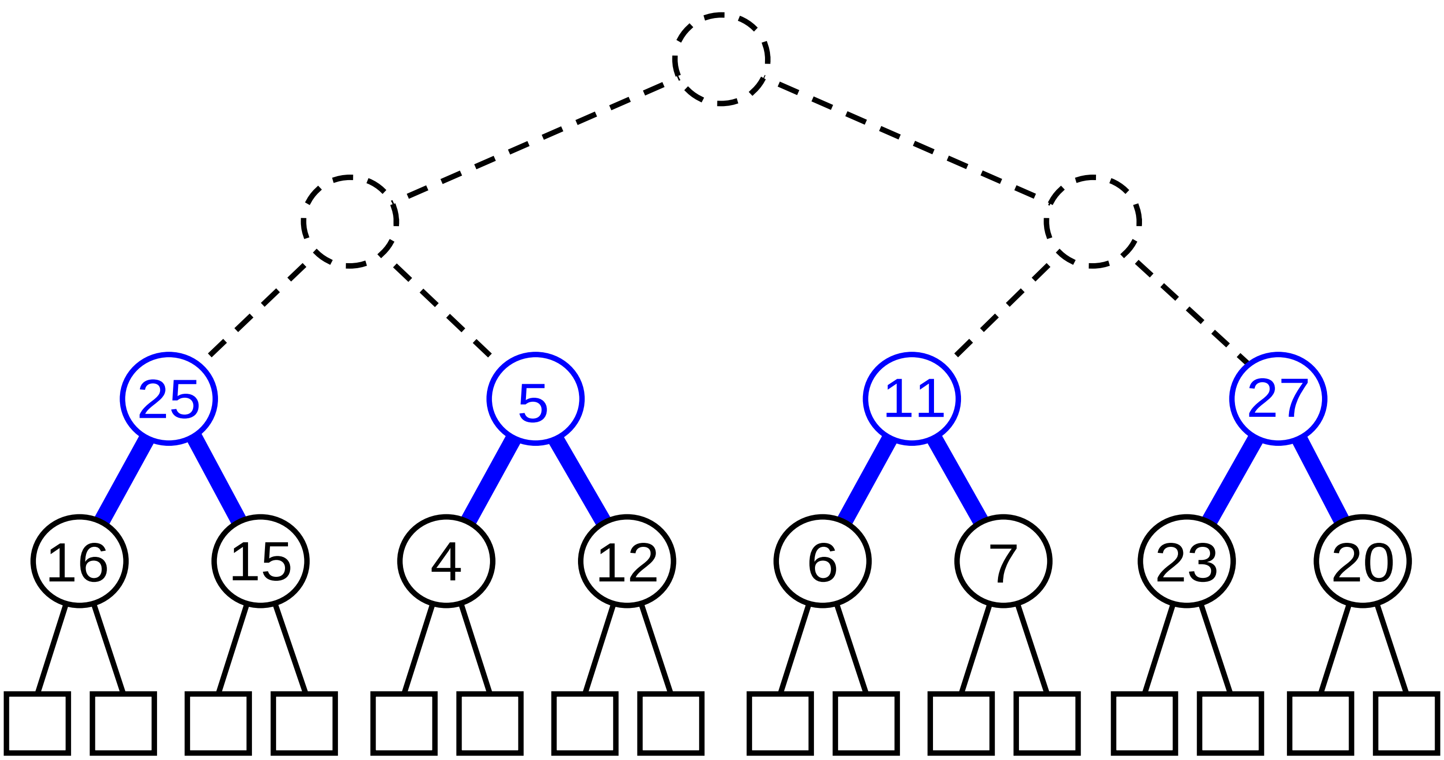
- Steps:
  1) Construct $(n+1)/2$ elementary heaps with one key each.
  2) Construct $(n+1)/4$ heaps, each with 3 keys, by joining pairs of elementary heaps and adding a new key as the root. The new key may be swapped with a child in order to perserve heap-order property.
  3) Construct $(n+1)/8$ heaps, each with 7 keys, by joining pairs of 3-key heaps and adding a new key. Again swaps may occur.

  ...

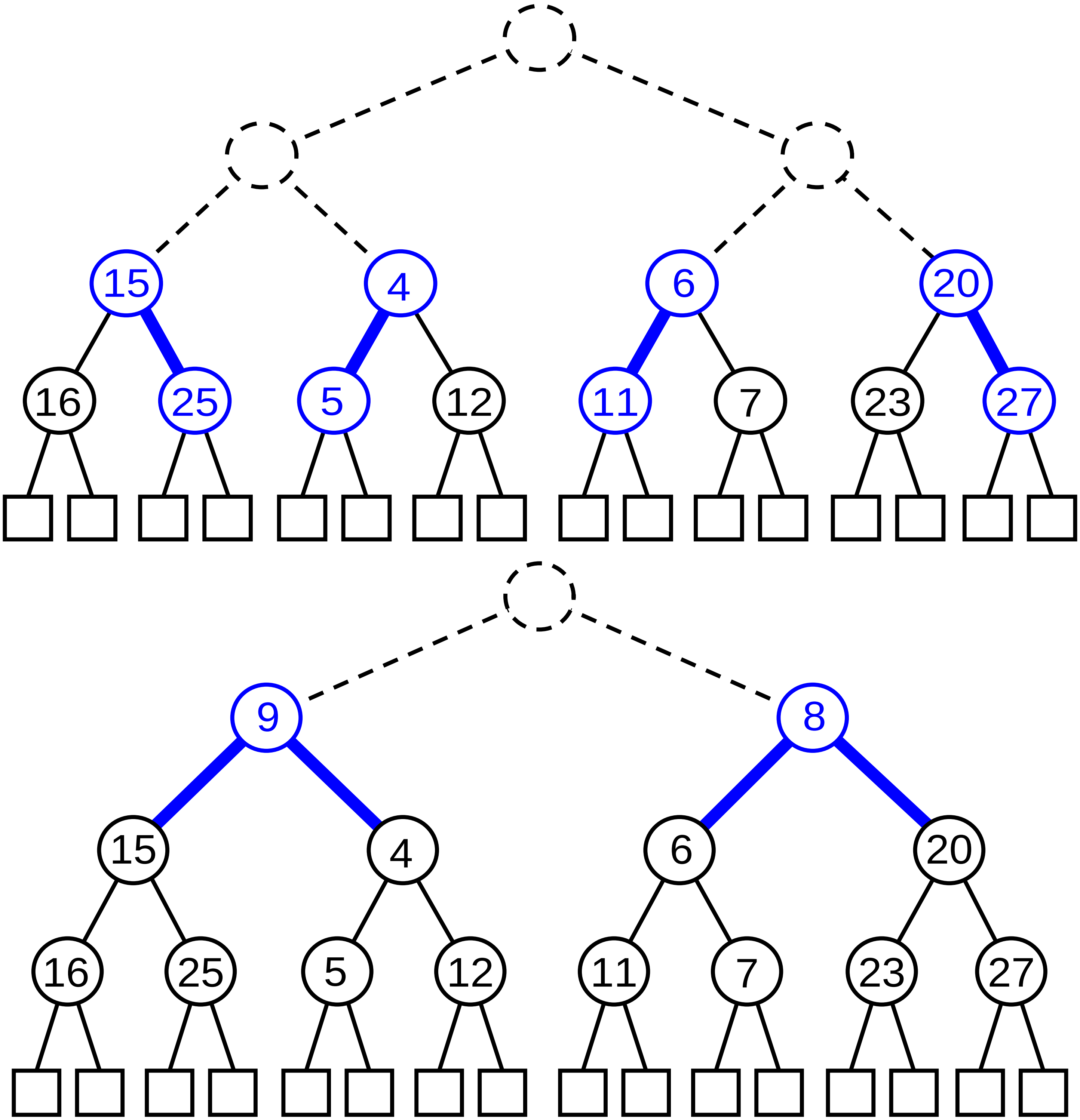
  4) In the ith step, $2 \leq i \leq h$, we form $(n+1)/2^i$ heaps, each storing 2^i-1 keys, by joining pairs of heaps storing $(2^{i-1} - 1)$ keys. Swaps may occur.

Bottom-Up Heap Construction (cont.)

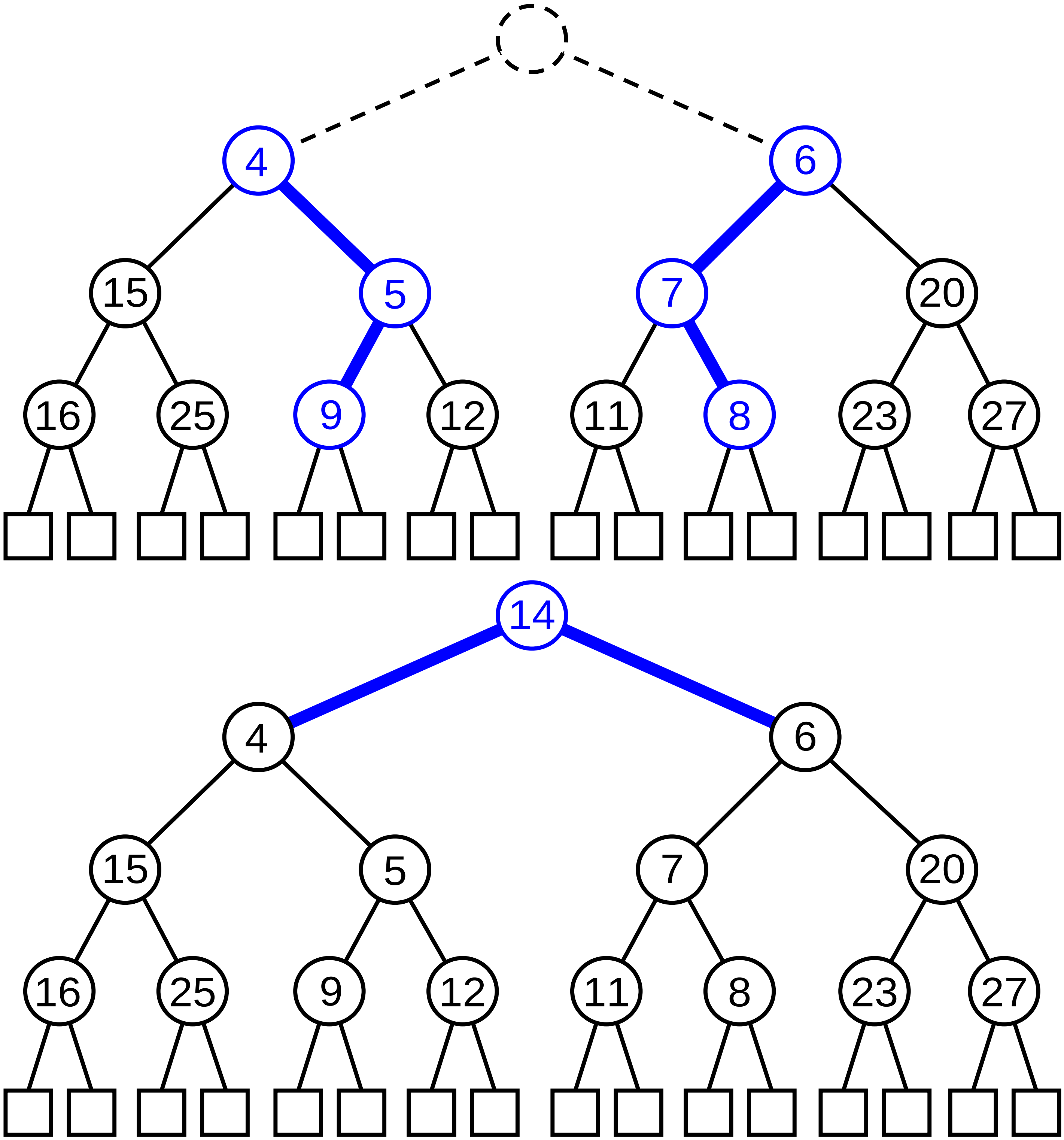




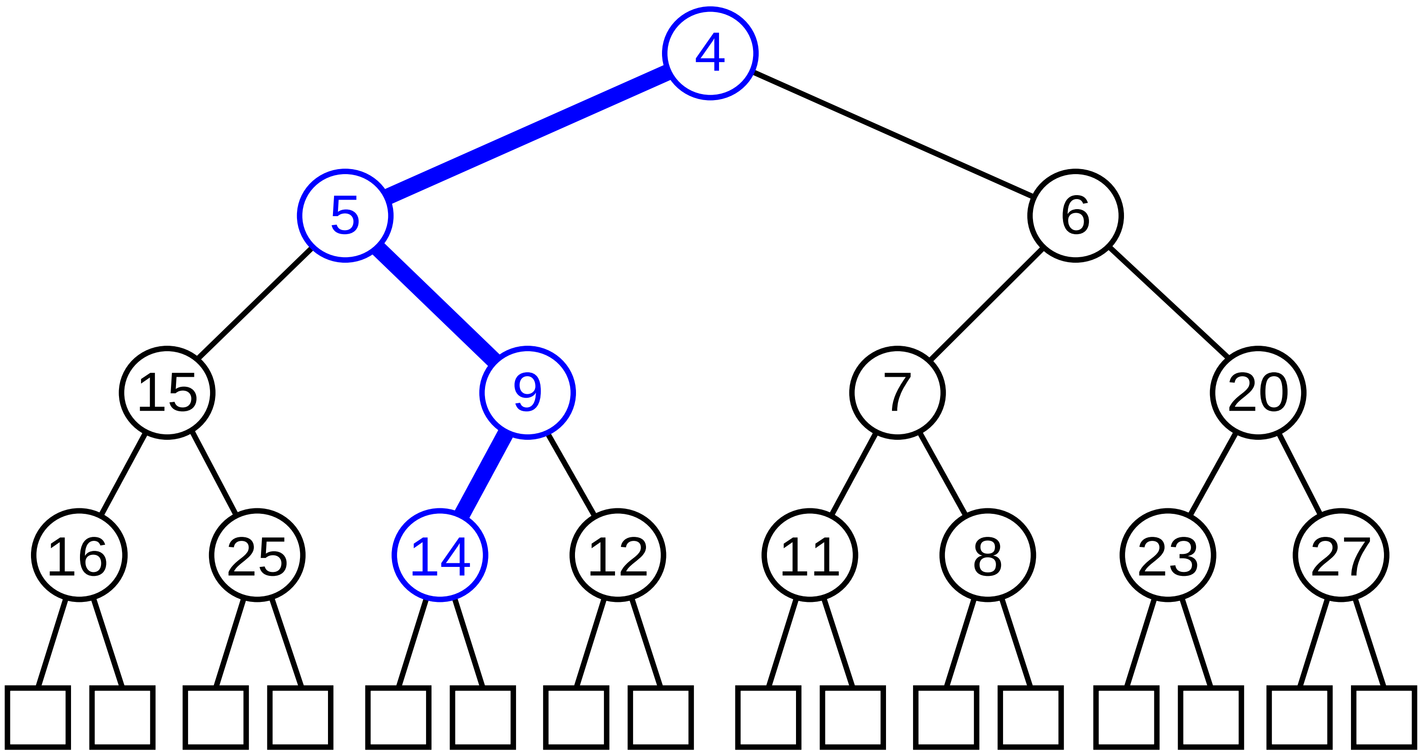
Bottom-Up Heap Construction (cont.)



Bottom-Up Heap Construction (cont.)



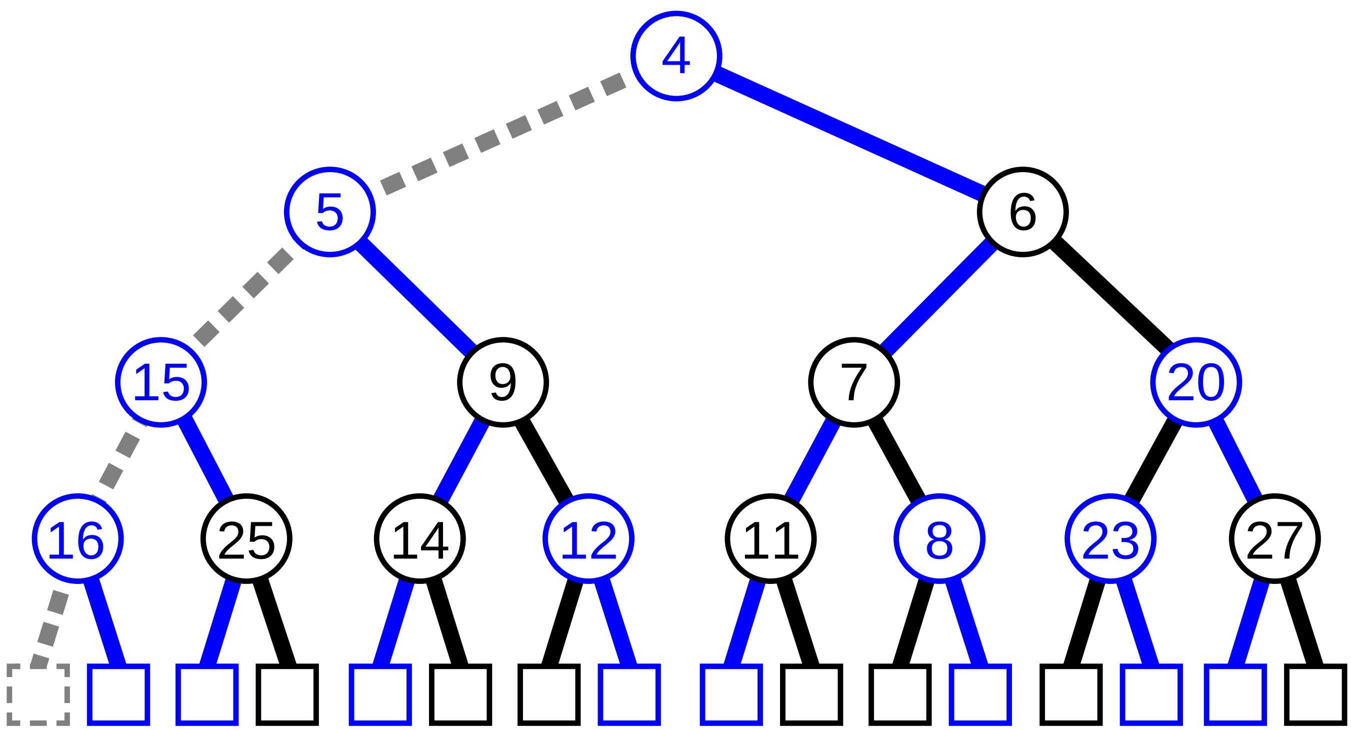
Bottom-Up Heap Construction (cont.)



The End

Analysis of Bottom-Up Heap Construction

- Proposition: Bottom-up heap construction with n keys takes $\boldsymbol{O}(n)$ time.
  - Insert $(n + 1)/2$ nodes
  - Insert $(n + 1)/4$ nodes
  - Upheap at most $(n + 1)/4$ nodes 1 level.
  - Insert $(n + 1)/8$ nodes
  - ...
  - Insert 1 node.
  - Upheap at most 1 node 1 level.



- n inserts, $n/2$ upheaps of 1 level = $\boldsymbol{O}(n)$