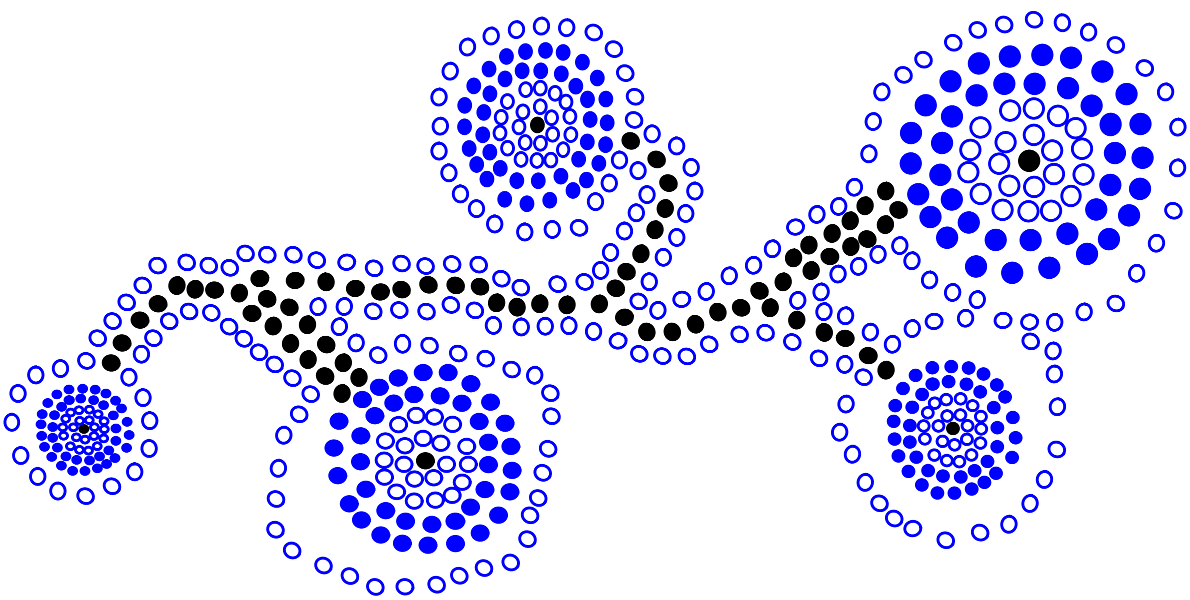
PRIORITY QUEUES

- The Priority Queue Abstract Data Type
- Implementing A Priority Queue With a Sequence



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 $k_1 \leq k_2$
  - **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*

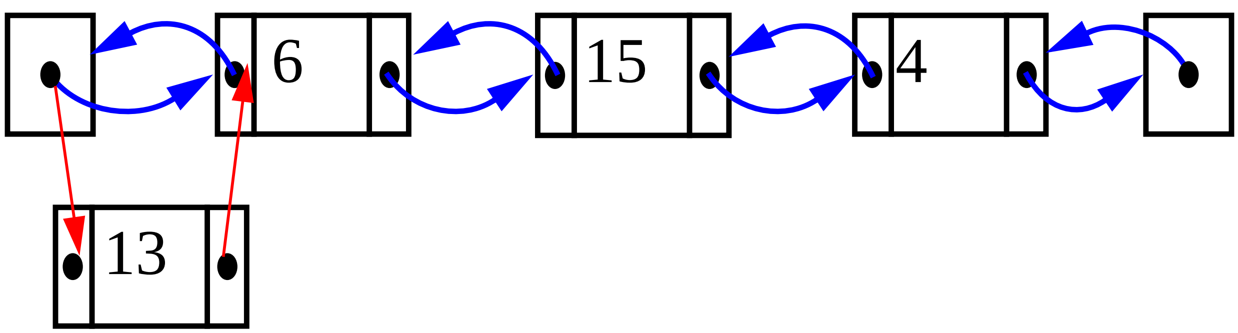
- removeMinElement():
  Remove from *P* and return an element with the smallest key; an error condidtion occurs if *P* is empty.
  Input: None; **Output**: Object *e*

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 an Unsorted Sequence

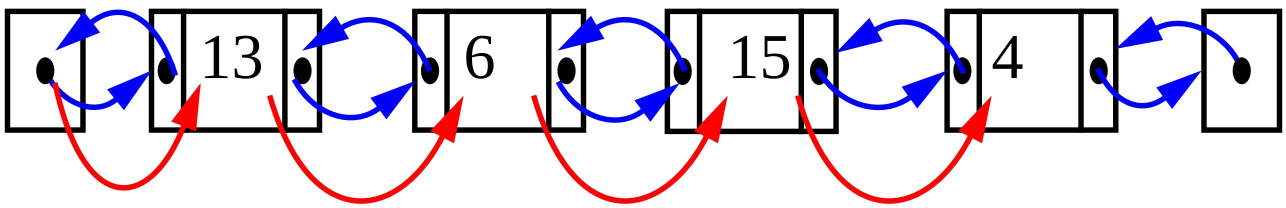
- Let's try to implement a priority queue with an unsorted sequence *S*.
- The elements of *S* are a composition of two elements, *k*, the key, and *e*, the element.
- We can implement insertItem() by using insertFirst() of the sequence. This would take ***O***(1) time.



- However, because we always insertFirst(), despite the key value, our sequence is not ordered.

Implementation with an Unsorted Sequence (contd.)

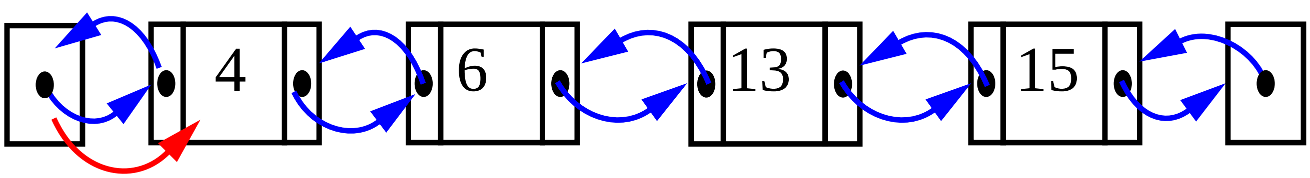
- Thus, for methods such as minElement(), minKey(), and removeMinElement(), we need to look at all elements of *S*. The worst case time complexity for these methods is $\boldsymbol{O}(n)$.



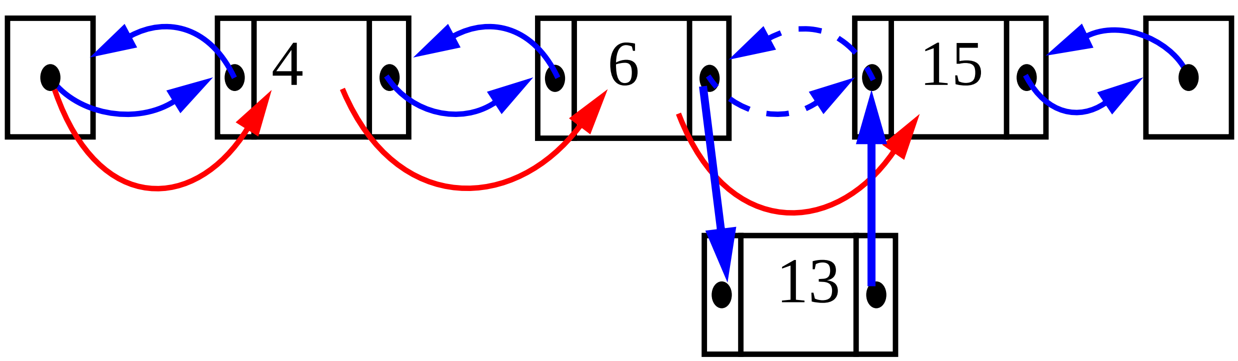
Implementation with a Sorted Sequence

- 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());
}
```

Selection Sort

- Selection Sort is a variation of PriorityQueueSort that uses an unsorted sequence to implement the priority queue P.
- **Phase 1**, the insertion of an item into P takes $\boldsymbol{O}(1)$ time.
- **Phase 2**, removing an item from P takes time proportional to the number of elements in P

	Sequence *S*	Priority Queue *P*
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(7, 4)
...	...	...
(g)	()	(7, 4, 8, 2, 5, 3, ,9)
Phase 2:		
(a)	(2)	(7, 4, 8, 5, 3, 9)
(b)	(2, 3)	(7, 4, 8, 5, 9)
(c)	(2, 3, 4)	(7, 8, 5, 9)
(d)	(2, 3, 4, 5)	(7, 8, 9)
(e)	(2, 3, 4, 5, 7)	(8, 9)
(f)	(2, 3, 4, 5, 7, 8)	(9)
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Selection Sort (cont.)

- As you can tell, a bottleneck occurs in Phase 2. The first removeMinElement operation take $\boldsymbol{O}(n)$, the second $\boldsymbol{O}(n\text{-}1)$, etc. until the last removal takes only $\boldsymbol{O}(1)$ time.
- The total time needed for phase 2 is:

$$O(n + (n - 1) + \ldots + 2 + 1) \equiv O\left(\sum_{i=1}^{n} i\right)$$

- By a common proposition:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- The total time complexity of phase 2 is then $\boldsymbol{O}(n^2)$. Thus, the time complexity of the algorithm is $\boldsymbol{O}(n^2)$.

Insertion Sort

- Insertion sort is the sort that results when we perform a PriorityQueueSort implementing the priority queue with a sorted sequence.
- We improve phase 2 to $\boldsymbol{O}(n)$.
- However, phase 1 now becomes the bottleneck for the running time. The first insertItem takes $\boldsymbol{O}(1)$, the second $\boldsymbol{O}(2)$, until the last opertation taked $\boldsymbol{O}(n)$.
- The run time of phase 1 is $\boldsymbol{O}(n^2)$ thus the run time of the algorithm is $\boldsymbol{O}(n^2)$.

Insertion Sort(cont.)

	Sequence *S*	Priority Queue *P*
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(4, 7)
(c)	(2, 5, 3, 9)	(4, 7, 8)
(d)	(5, 3, 9)	(2, 4, 7, 8)
(e)	(3, 9)	(2, 4, 5, 7, 8)
(f)	(9)	(2, 3, 4, 5, 7, 8)
(g)	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2:		
(a)	(2)	(3, 4, 5, 7, 8, 9)
(b)	(2, 3)	(4, 5, 7, 8, 9)
...	...	...
(g)	(2, 3, 4, 5, 7, 8, 9)	()

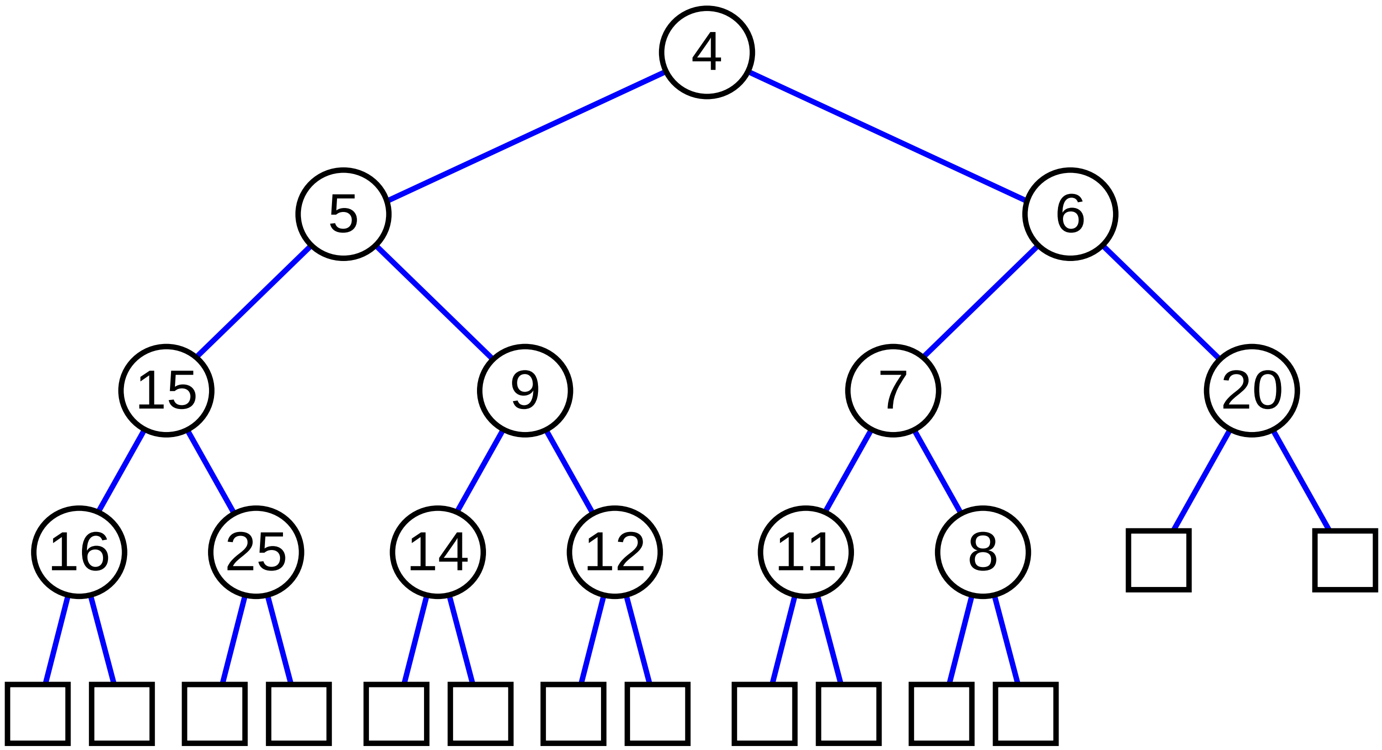
- Selection and insertion sort both take $\boldsymbol{O}(n^2)$.
- Selection sort will always take $\Omega(n^2)$ time, no matter the input sequence.
- The run of insertion sort varies depends on the input sequence.
- We have yet to see the ultimate priority queue....

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:

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

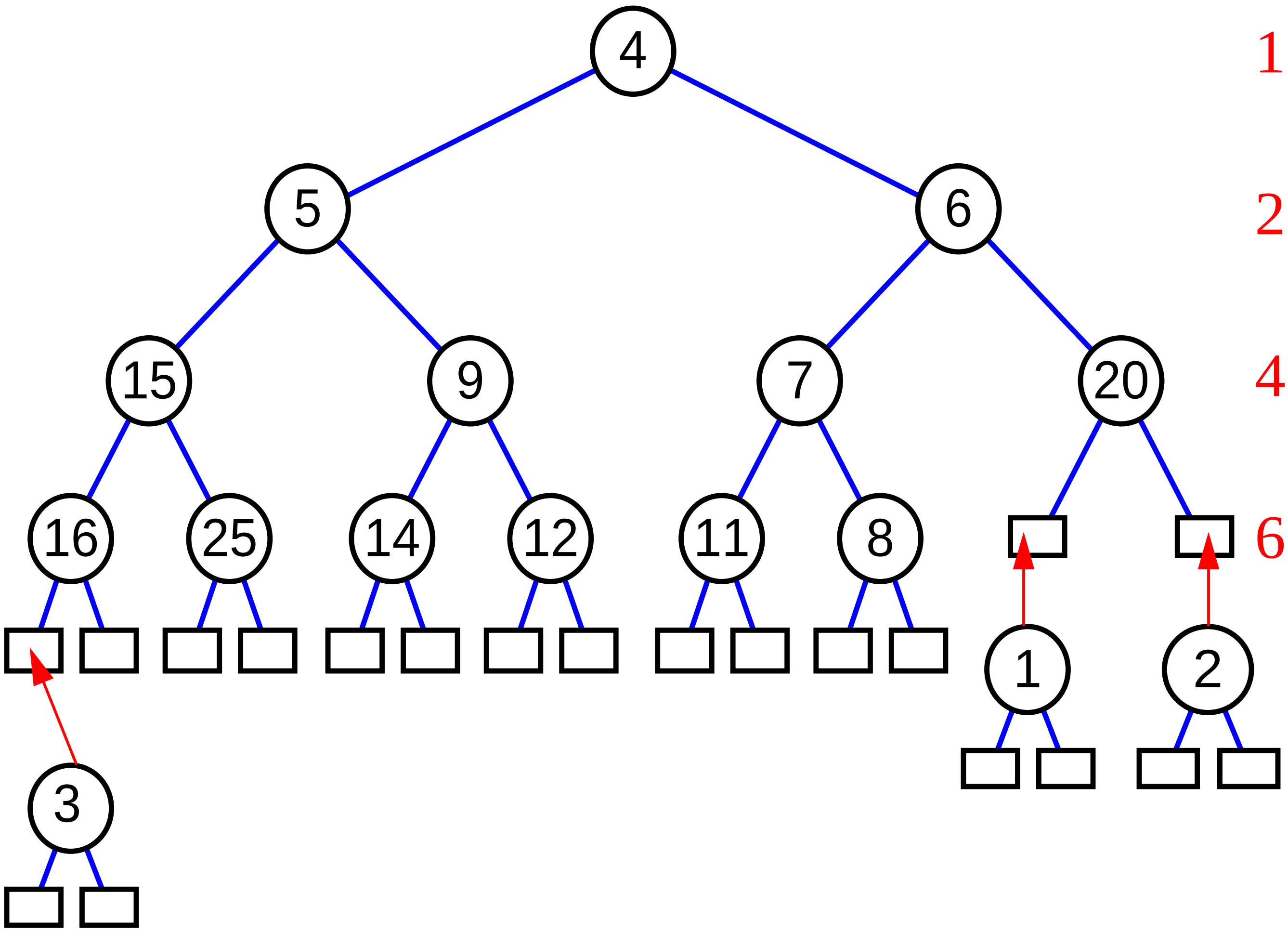
  - Which in turn implies:

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

  - Q.E.D.

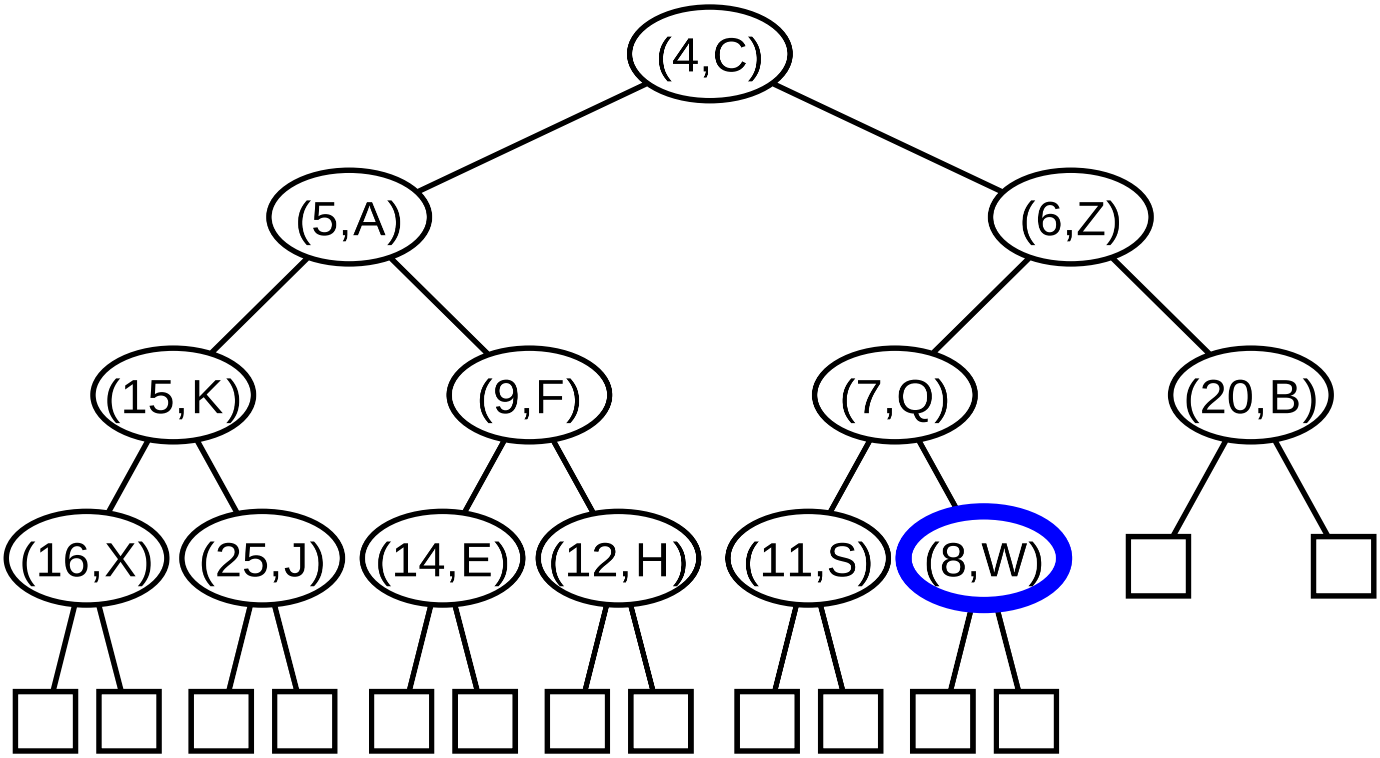
Heigh of a Heap (contd.)

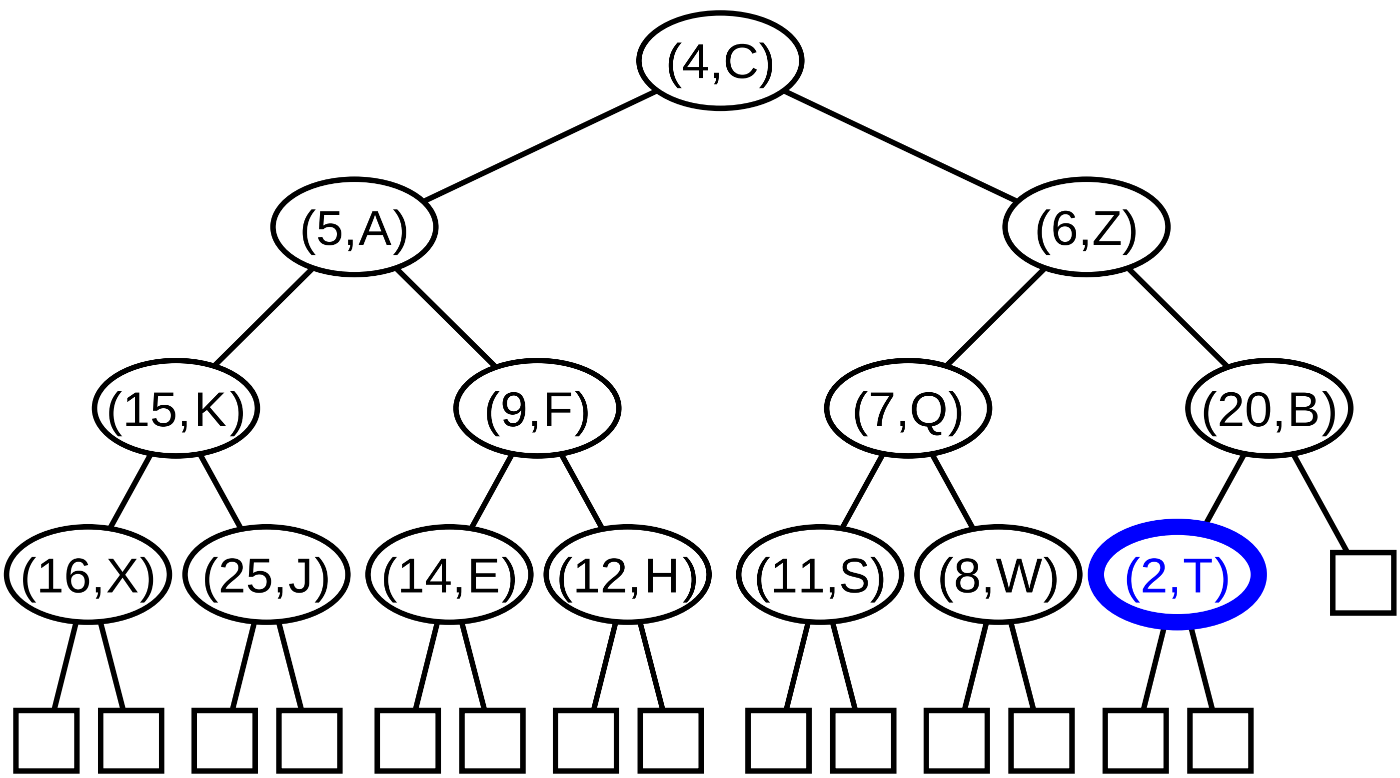
- Let's look at that graphically:



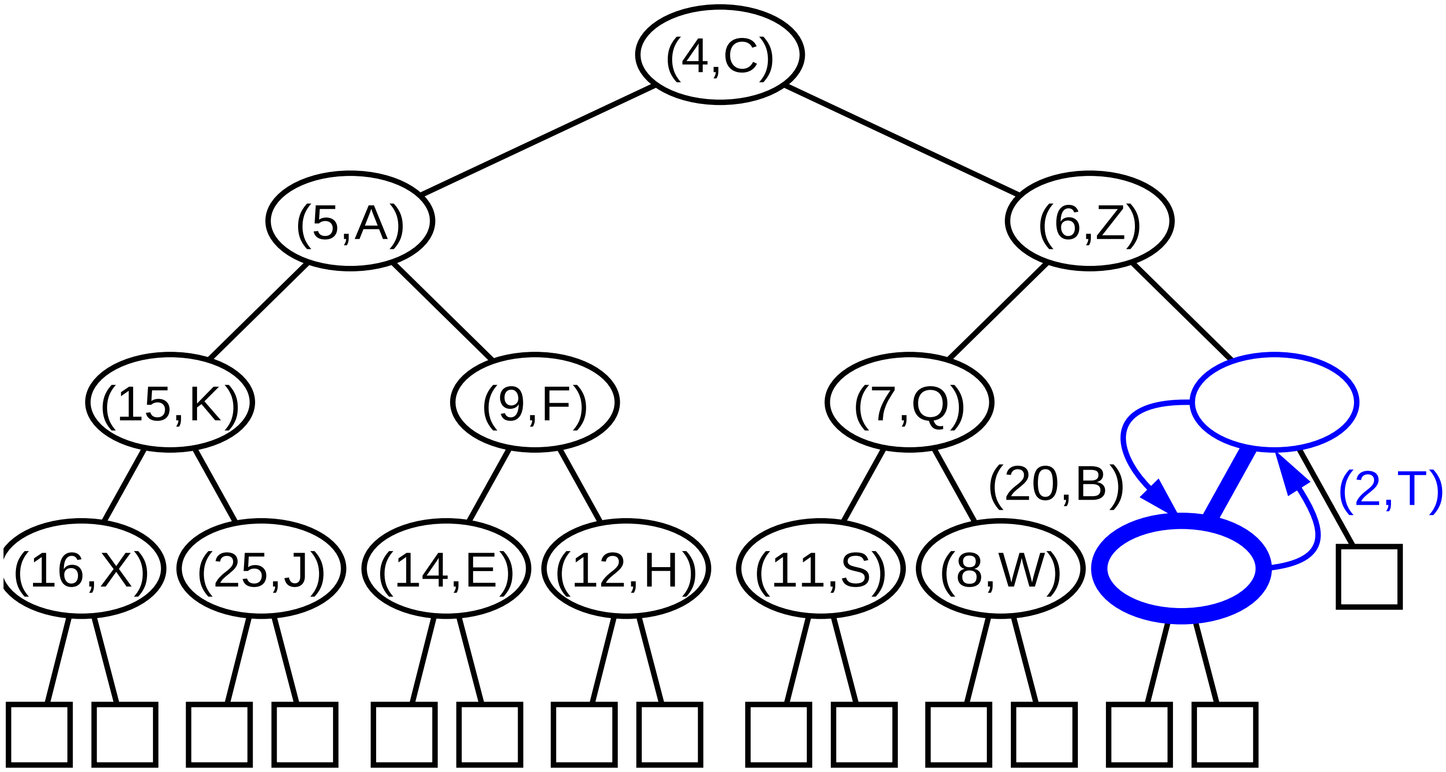
- Consider this heap which has height $h = 4$ and $n = 13$
- Suppose two more nodes are added. To maintain completeness of the tree, the two external nodes in level 4 will become internal nodes: i.e.
  $n = 15$, $h = 4 = \log(15+1)$
- Add one more: $n = 16$, $h = 5 = \lceil \log(16+1) \rceil$

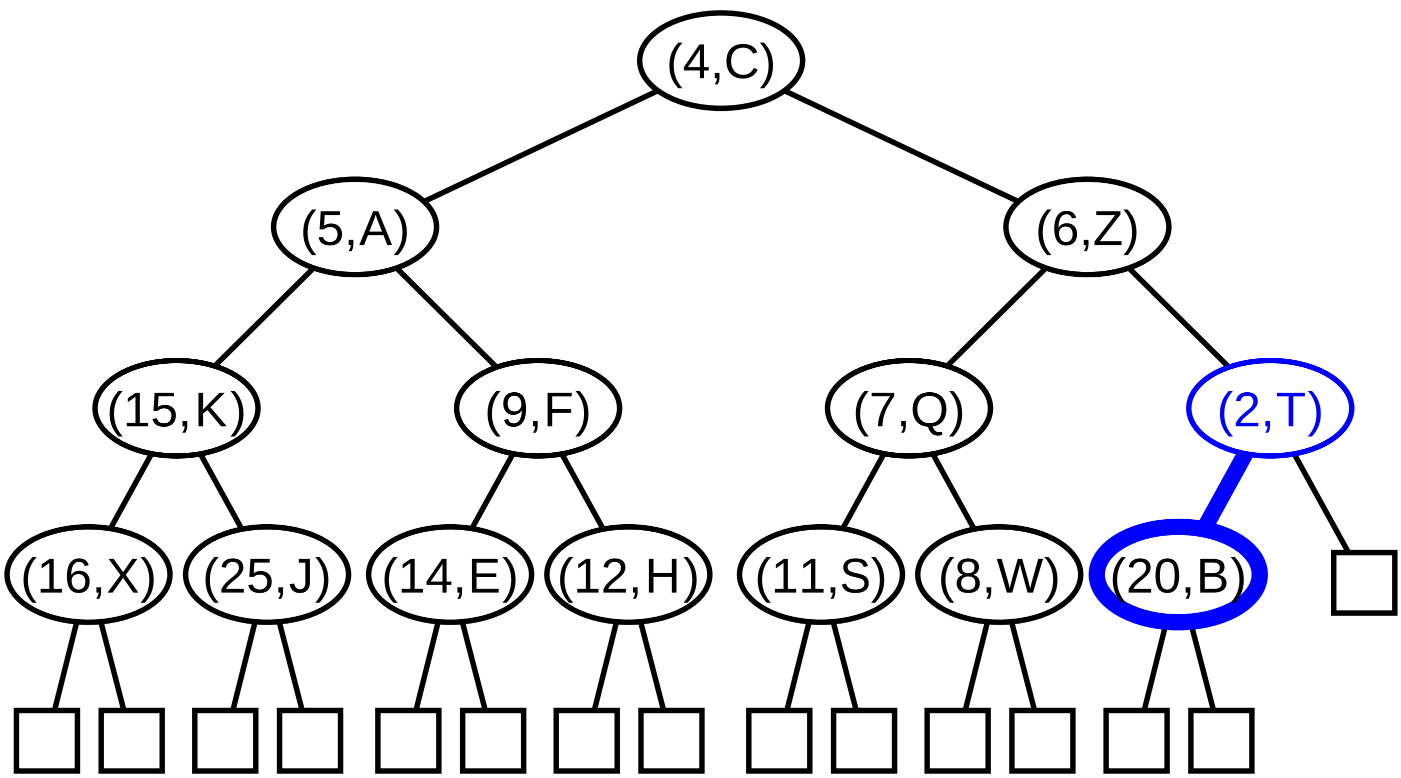
Insertion into a Heap



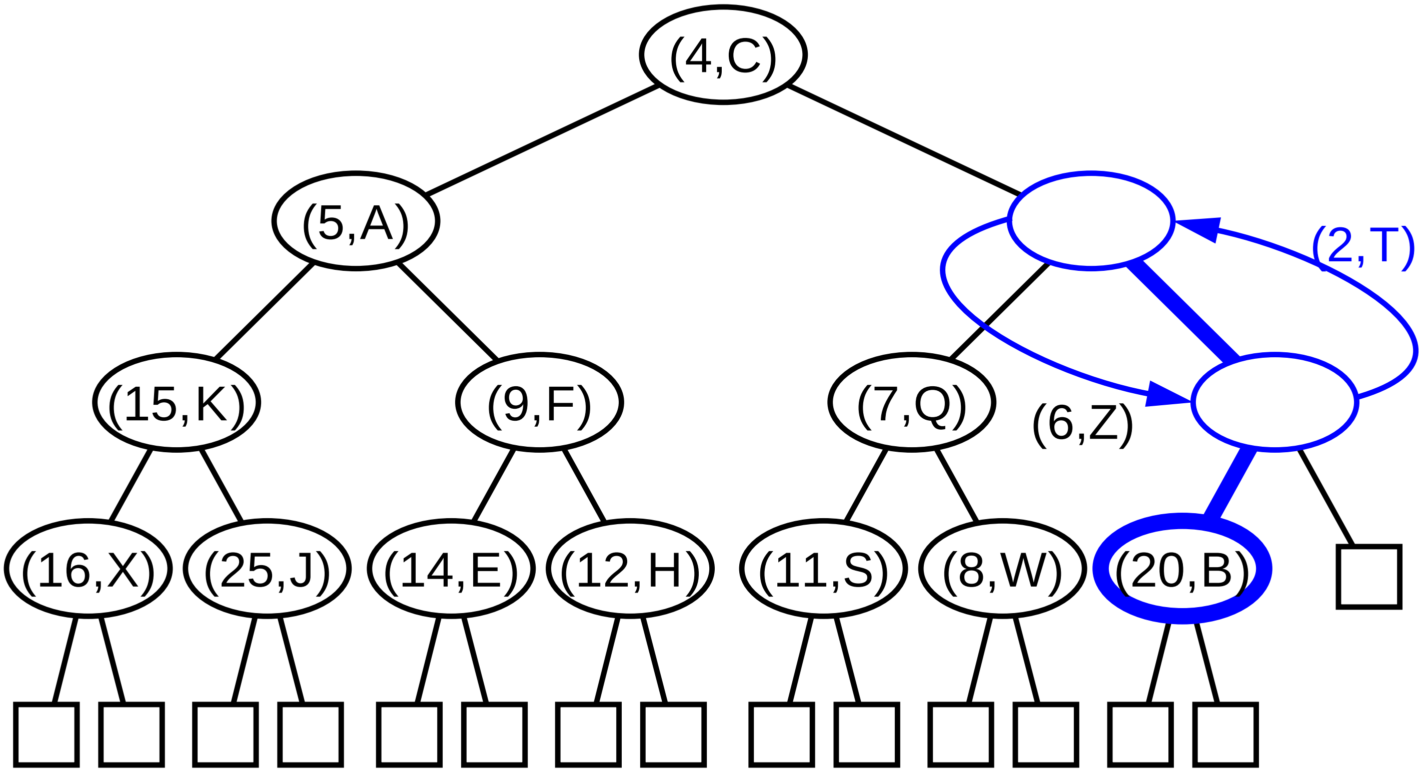


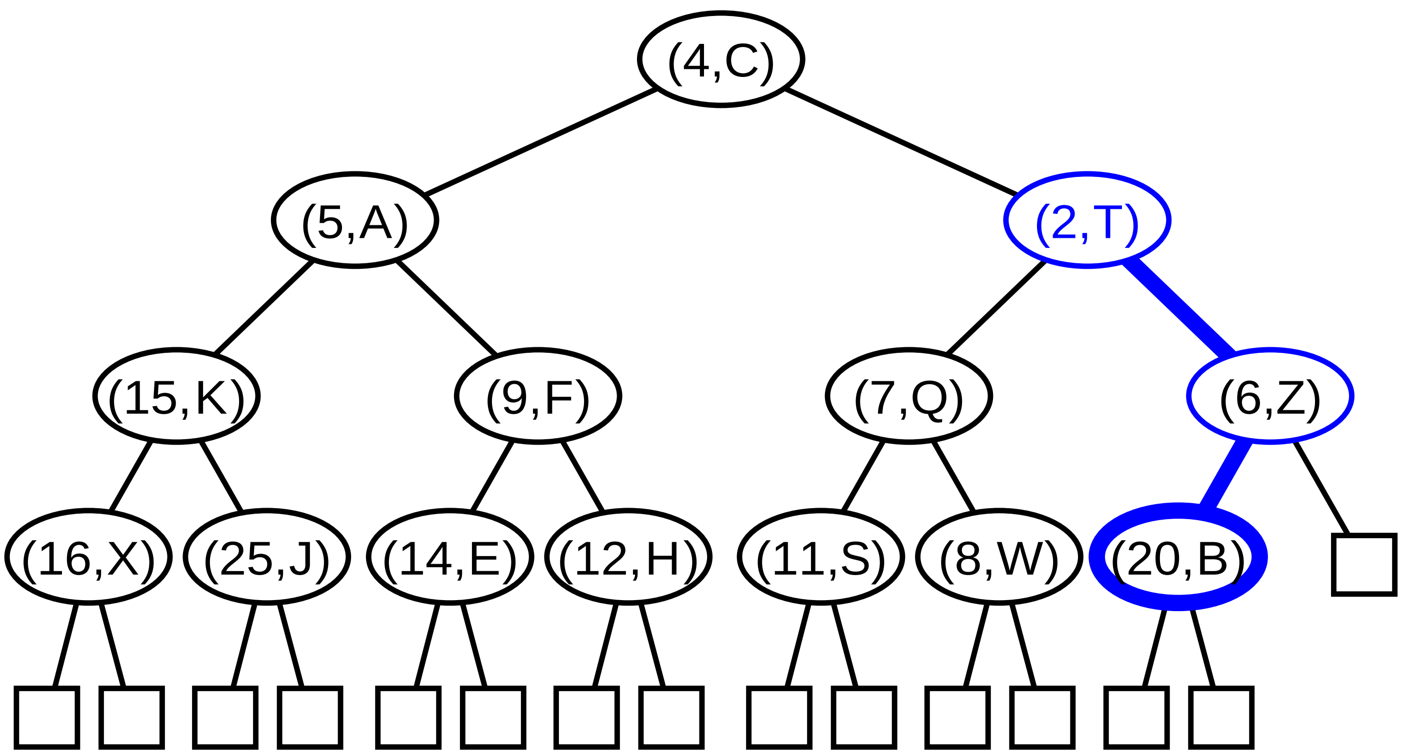
Insertion into a Heap (cont.)



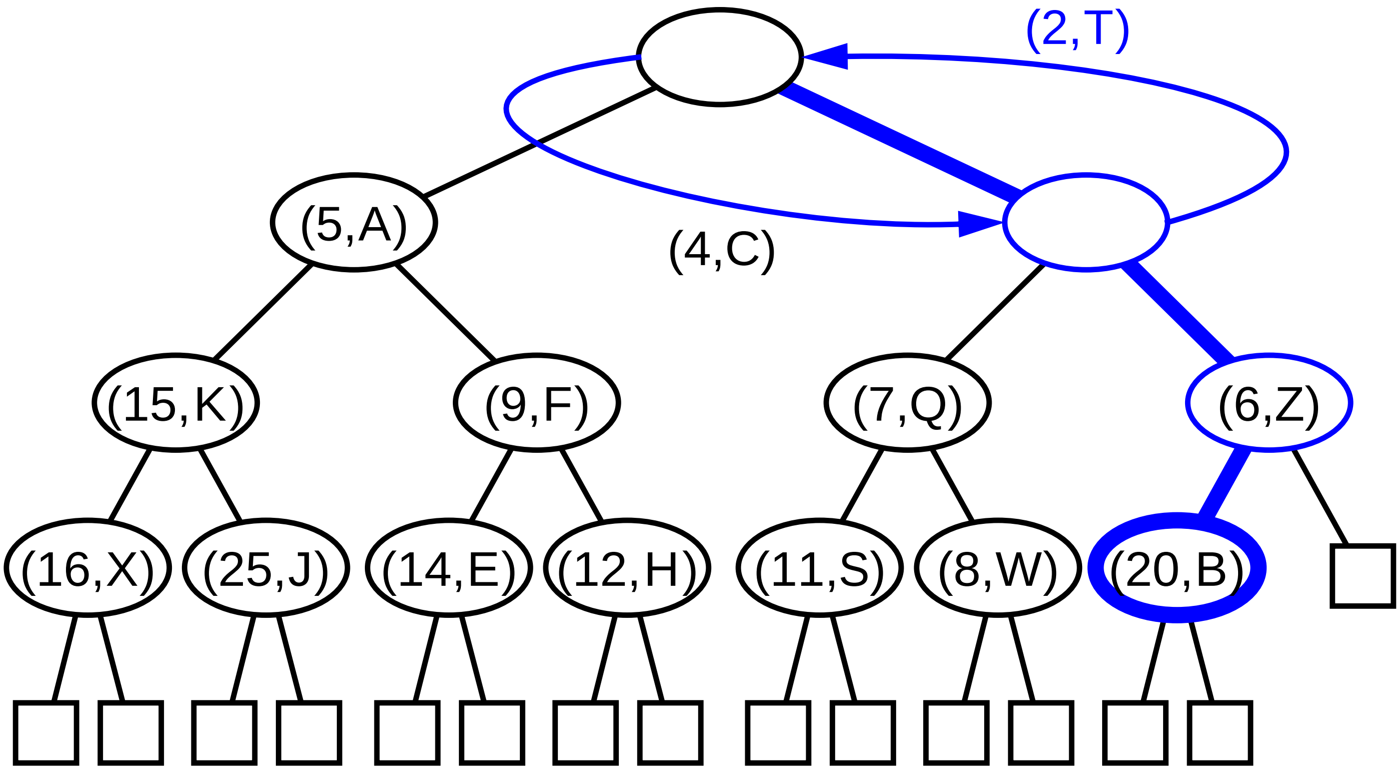


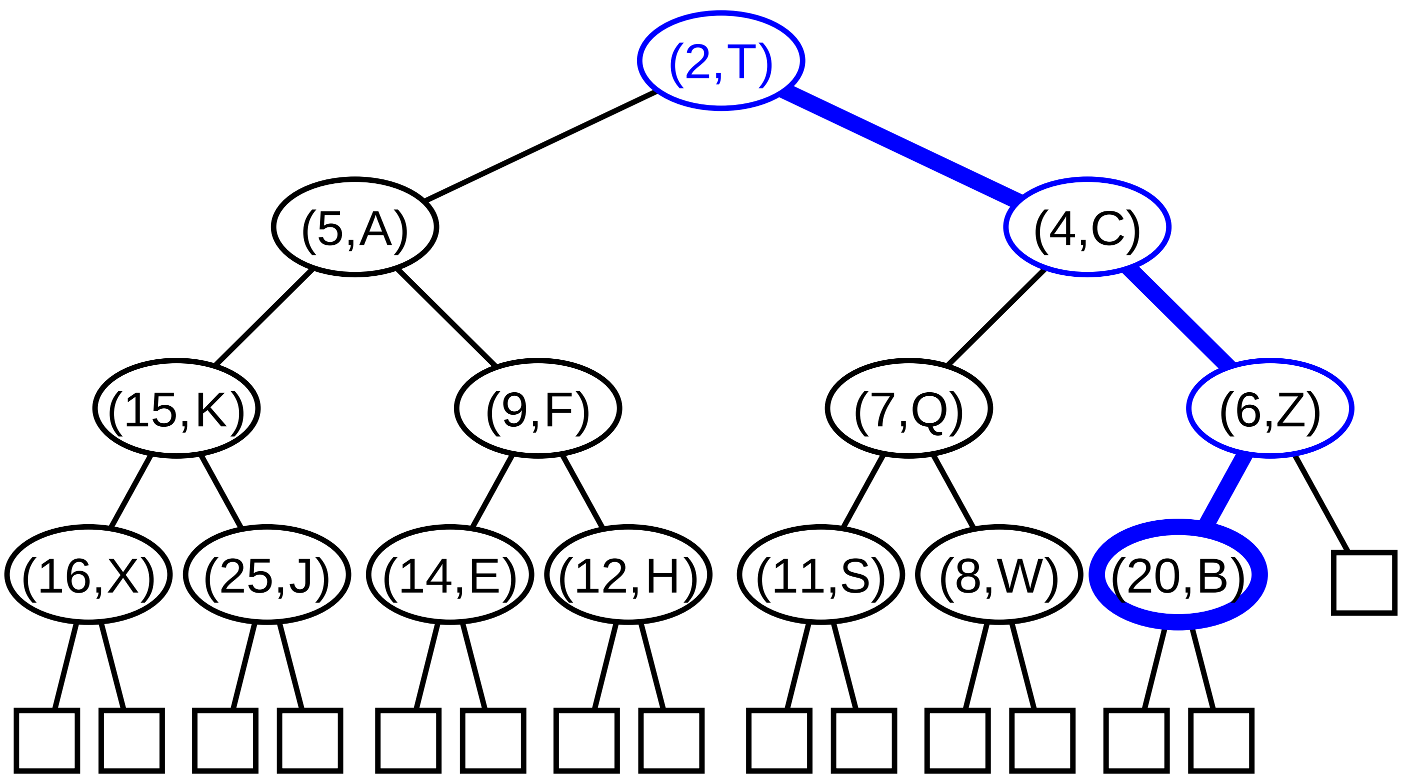
Insertion into a Heap (cont.)



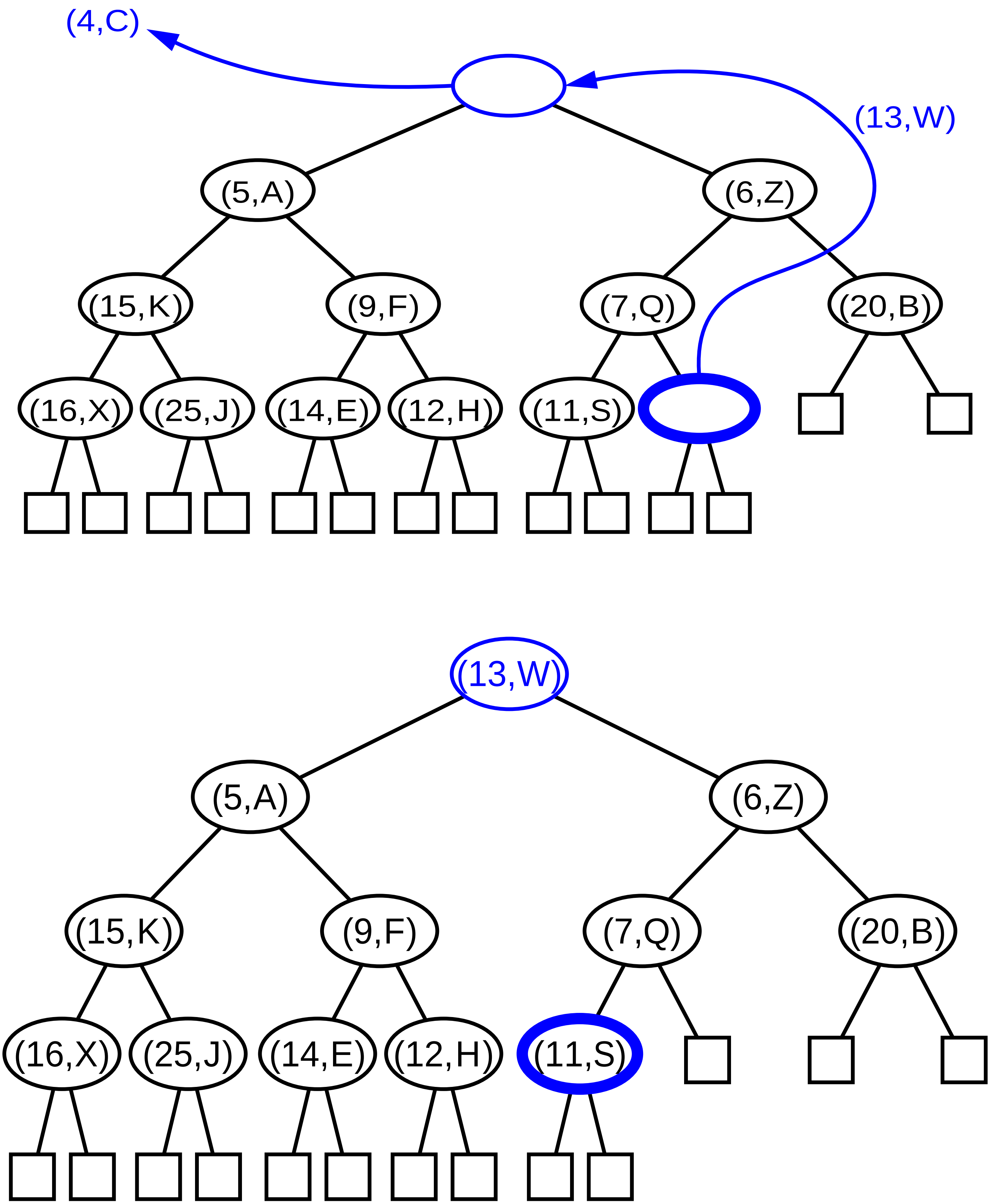


Insertion into a Heap (cont.)

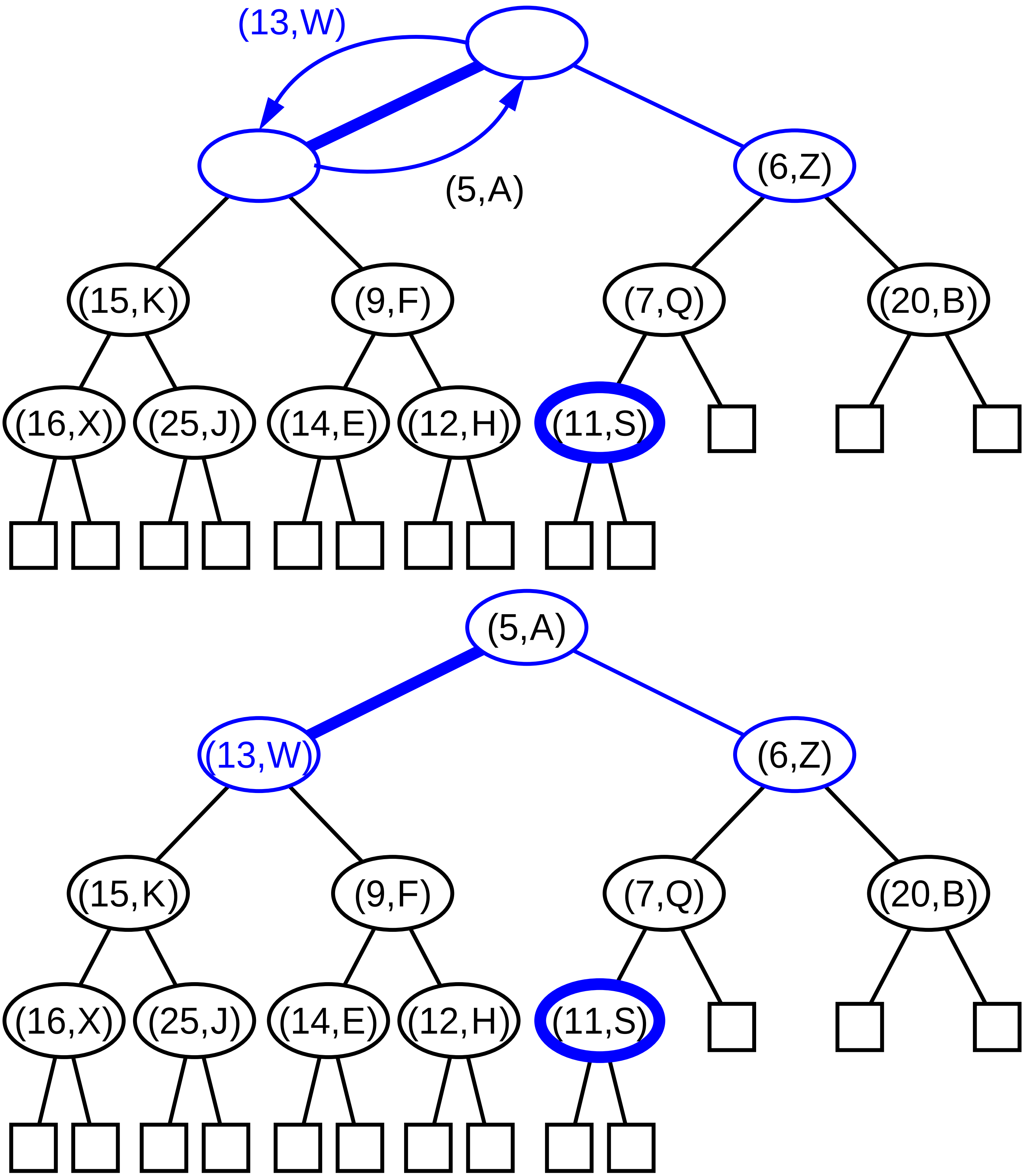




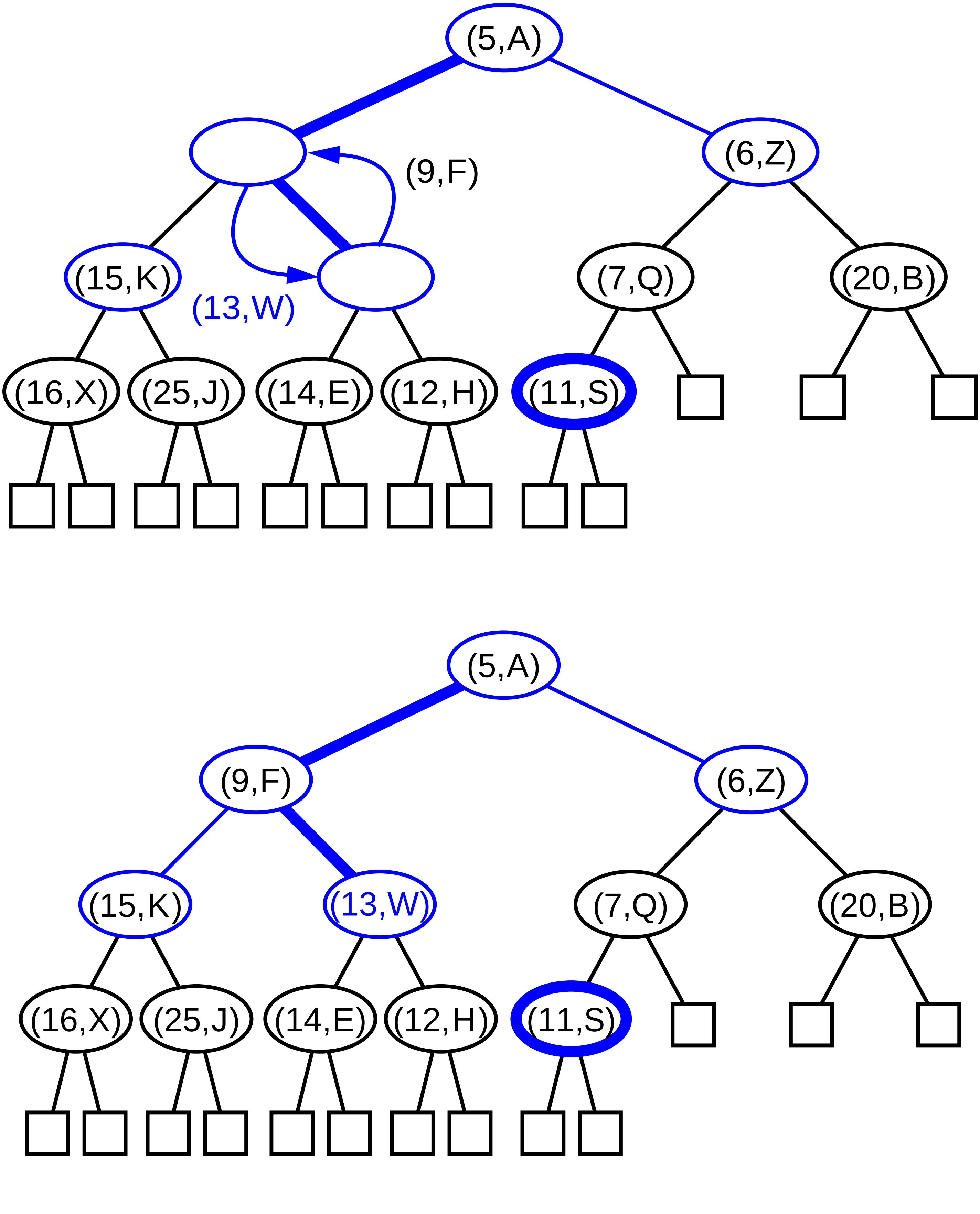
Removal from a Heap



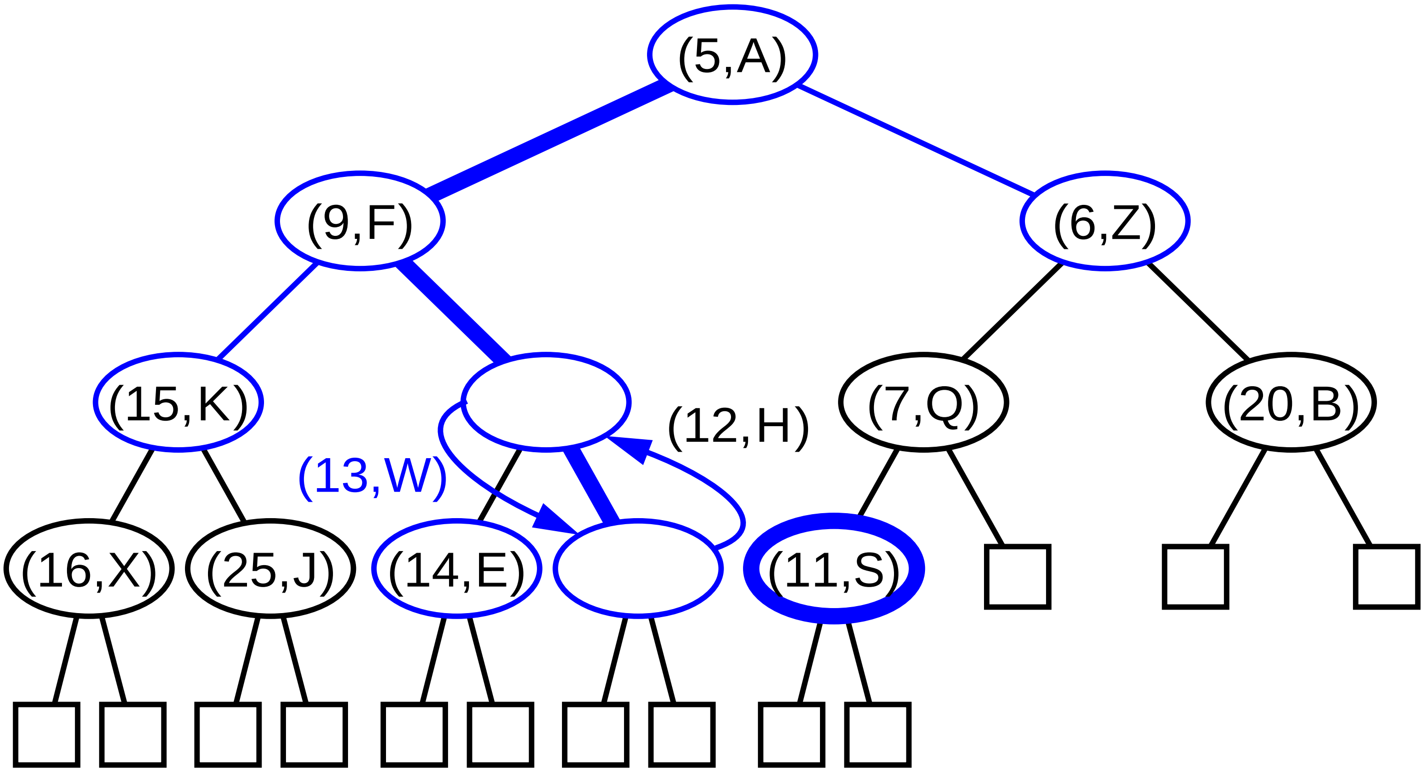
Removal from a Heap (cont.)

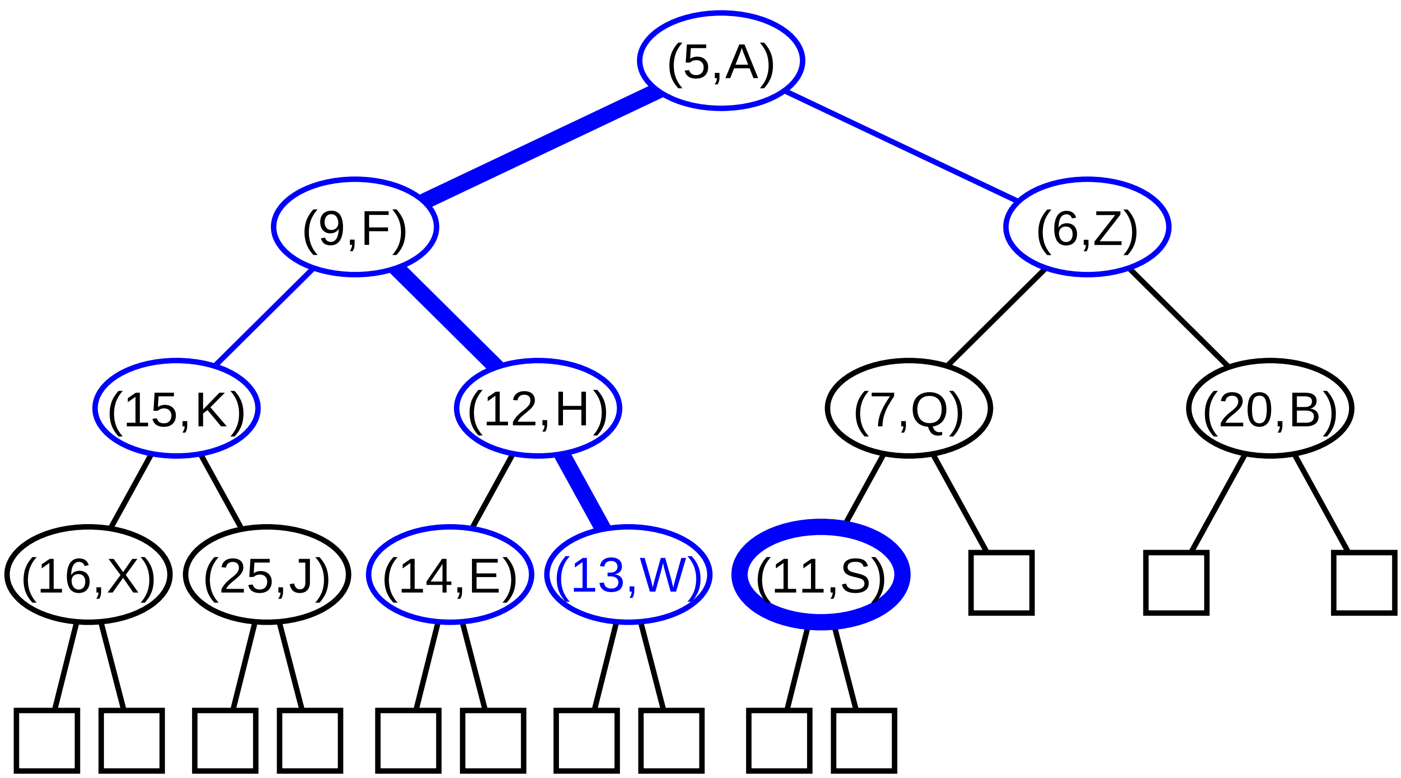


Removal from a Heap (cont.)



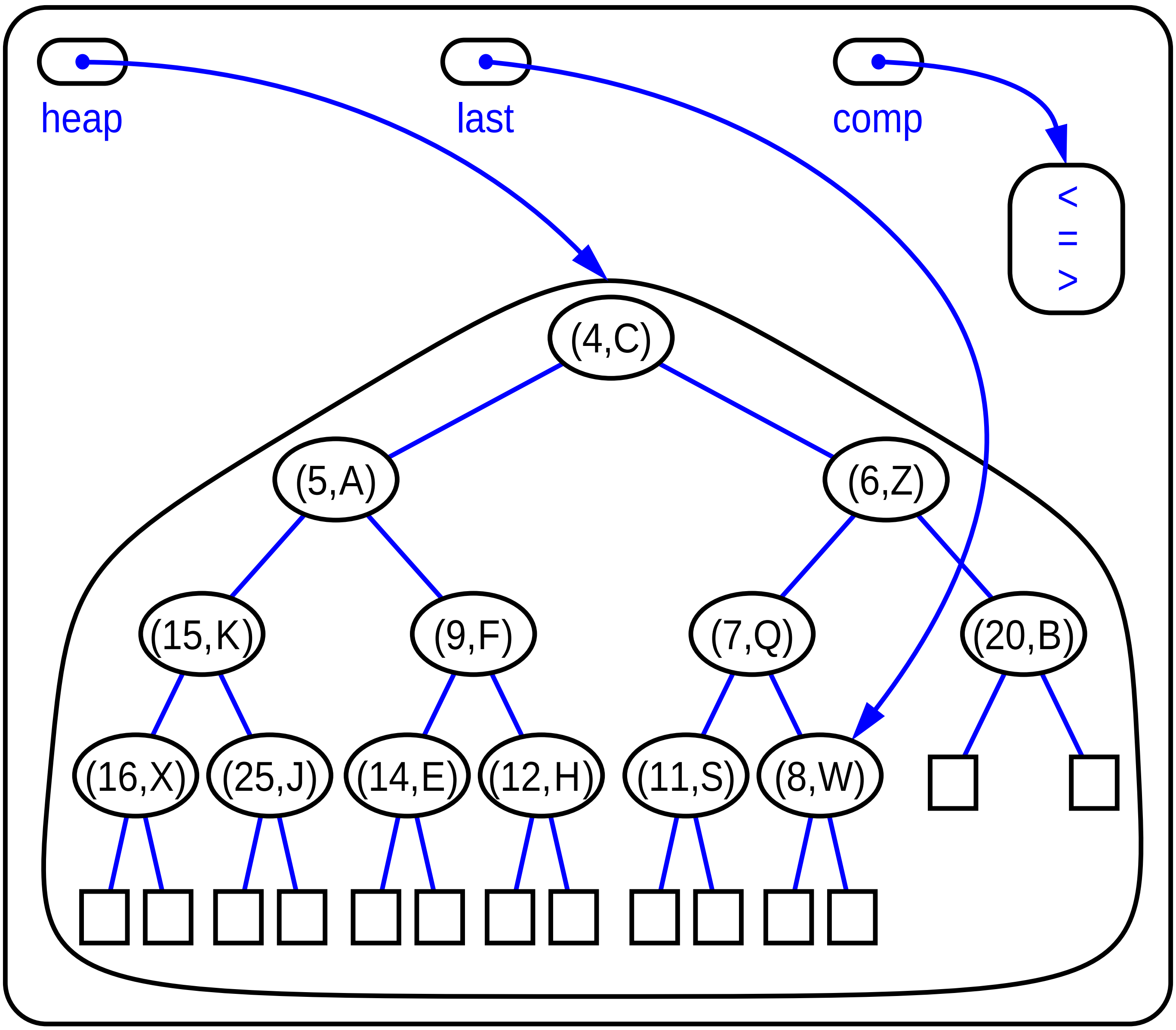
Removal from a Heap(cont.)





Implementation of a Heap

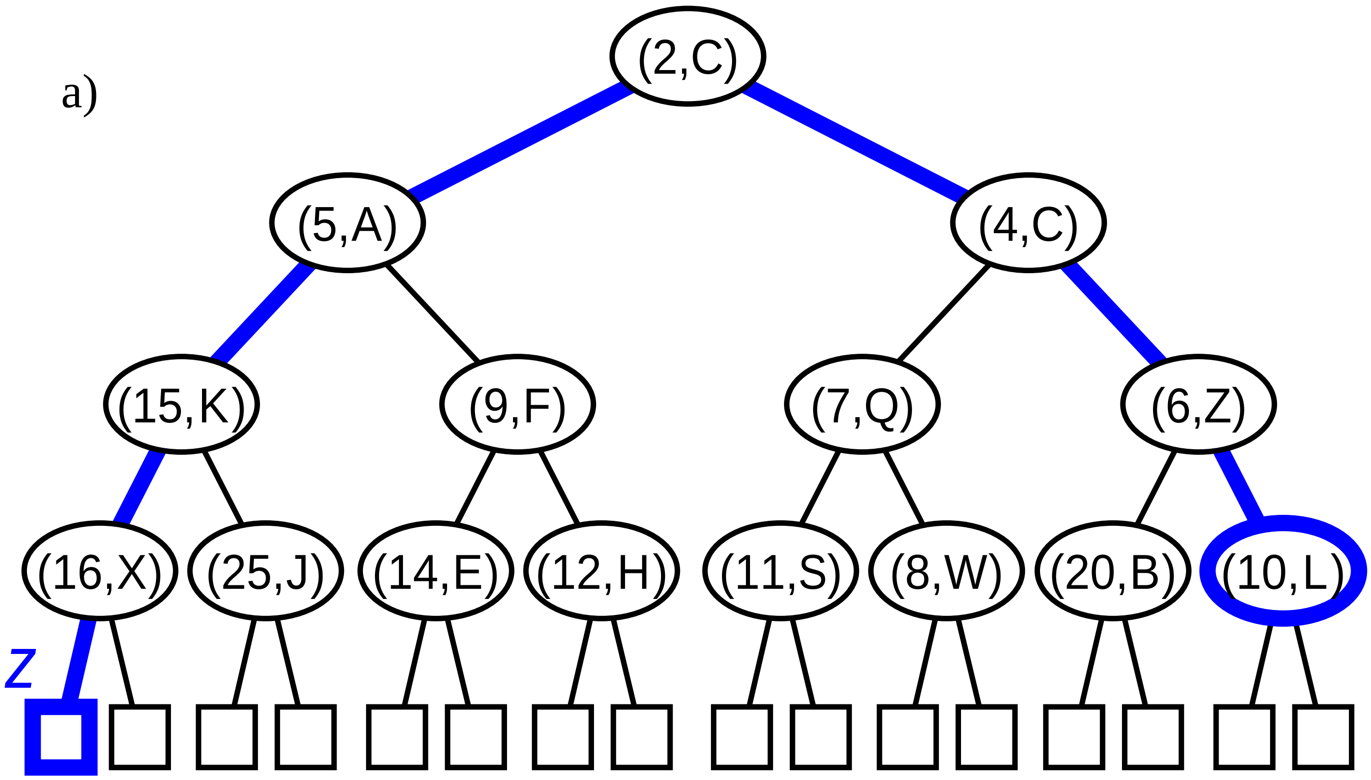
```
public class HeapSimplePriorityQueue implements
    SimplePriorityQueue {
    BinaryTree T;
    Position last;
    Comparator comparator;
    ...
}
```



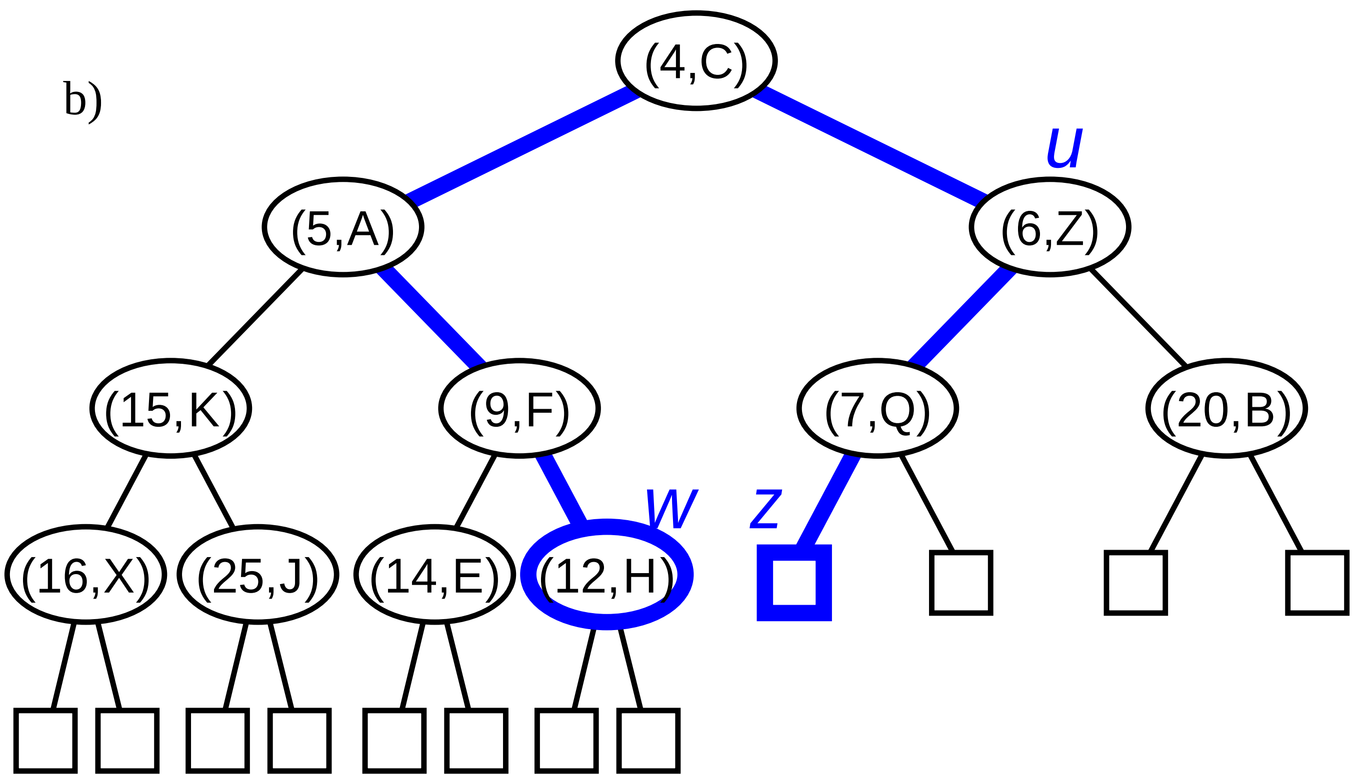
Implementation of a Heap(cont.)

- 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.

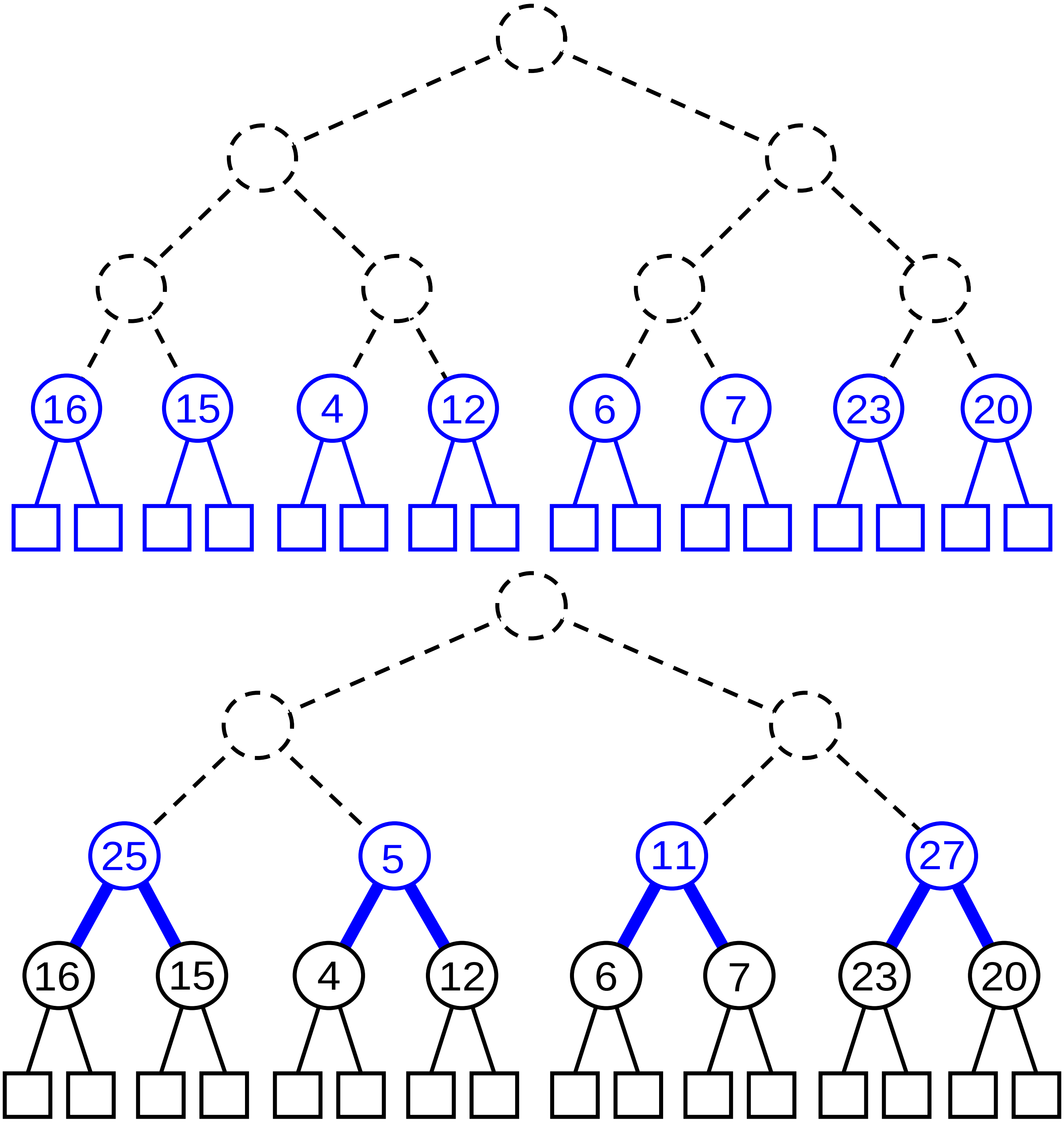
- Steps:

  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.

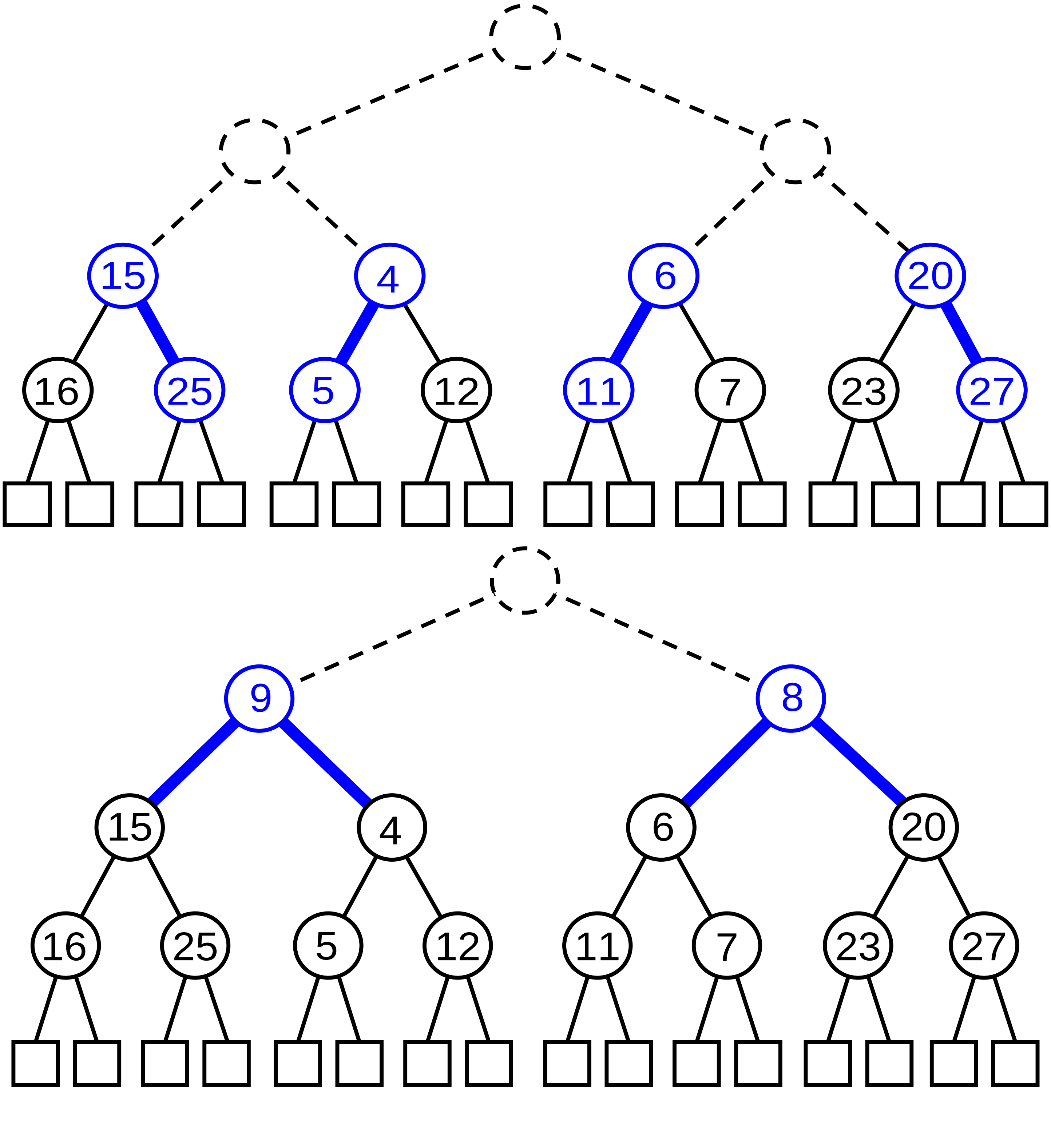
  ...

  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.

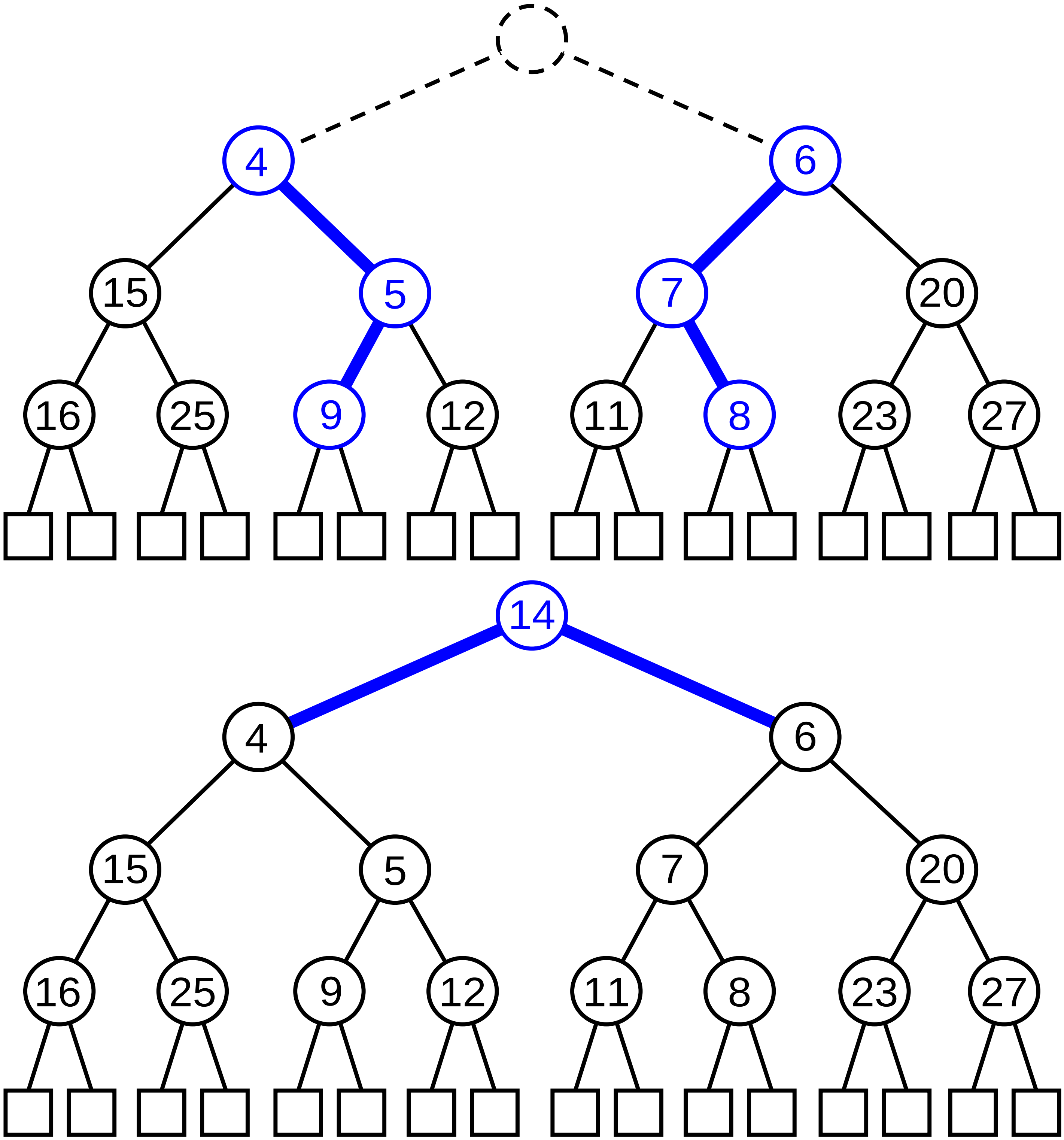
Bottom-Up Heap Construction (cont.)



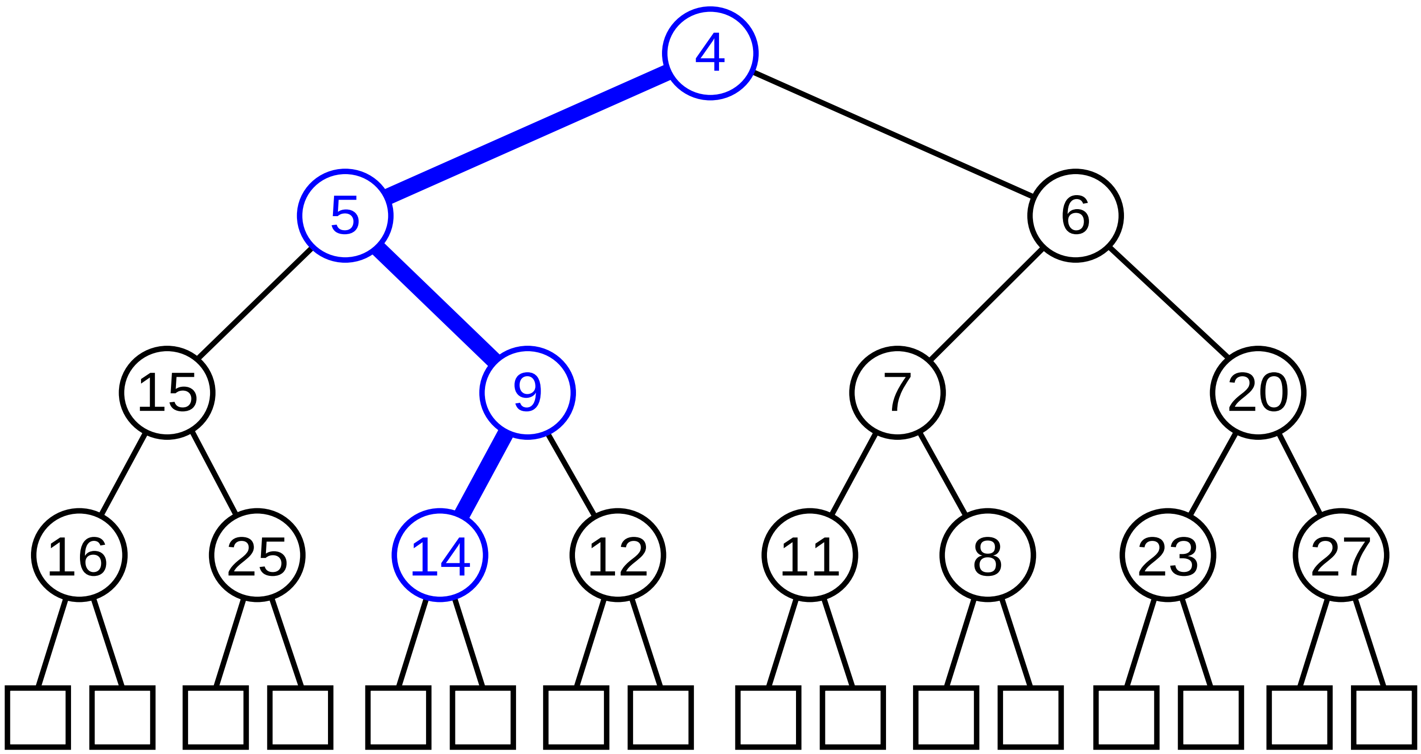
Bottom-Up Heap Construction (cont.)



Bottom-Up Heap Construction (cont.)



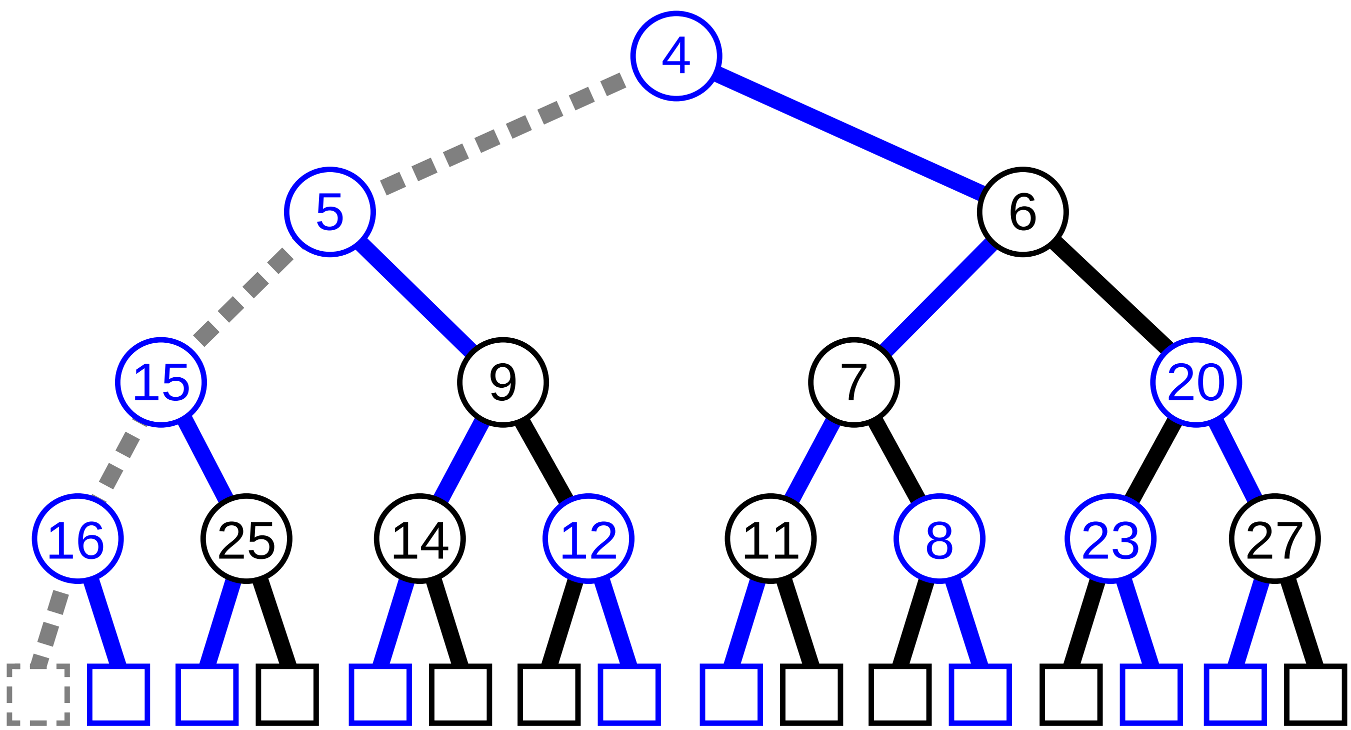
Bottom-Up Heap Construction (cont.)



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 = $\boldsymbol{O}(n)$

Locators

- Locators can be used to keep track of elements in a container
- A locator sticks with a specific key-lement pair, even if that element "moves around".
- The Locator ADT supports the following fundamental methods:

element(): Return the element of the item associated with the Locator.
Input: None; **Output**: Object

key(): Return the key of the item assocated with the Locator.
Input: None; **Output**: Object

isContained(): Return true if and only if the Locator is associated with a container.
Input: None; **Output**: boolean

container(): Return the container associated with the Locator.
Input: None; **Output**: boolean

Priority Queue with Locators

- It is easy to extend the sequence-based and heap-based implementations of a Priority Queue to support Locators.
- The Priority Queue ADT can be extended to implement the Locator ADT
- In the heap implementation of a priority queue, we store in the locator object a key-element pair and a reference to its position in the heap.
- All of the methods of the Locator ADT can then be implemented in $O(1)$ time.

A Java Implementation of a Locator

```
public class LocItem extends Item implements Locator
{
  private Container cont;
  private Position pos;
  LocItem (Object k, Object e, Position p, Container c) {
    super(k, e);
    pos = p;
    cont = c;
  }
  public boolean isContained() throws
InvalidLocatorException {
    return cont != null;
  }
  public Container container() throws
InvalidLocatorException { return cont; }
  protected Position position() {return pos; }
  protected void setPosition(Position p) { pos = p; }
  protected void setContainer(Container c) { cont = c; }
}
```

A Java Implementation of a Locator-Based Priority Queue

```
public class SequenceLocPriorityQueue
  extends SequenceSimplePriorityQueue implements PriorityQueue {

    //priority queue with locators implemented
    //with a sorted sequence

  public SequenceLocPriorityQueue (Comparator comp) { super(comp); }

    // auxiliary methods
protected LocItem locRemove(Locator loc) {
    checkLocator(loc);
    seq.remove(((LocItem) loc).position());
    ((LocItem) loc).setContainer(null);
    return (LocItem) loc;
  }
```

Locator-Based PQ (contd.)

```
protected Locator locInsert(LocItem locit) throws
InvalidKeyException {
    Position p, curr;
    Object k = locit.key();
    if (!comp.isComparable(k))
  throw new InvalidKeyException("The key is not valid");
    else
      if (seq.isEmpty())
        p = seq.insertFirst(locit);
      else if
           (comp.isGreaterThan(k,extractKey(seq.last())))
        p = seq.insertAfter(seq.last(),locit);
      else {
        curr = seq.first();
        while (comp.isGreaterThan(k,extractKey(curr)))
          curr = seq.after(curr);
        p = seq.insertBefore(curr,locit);
      }
     locit.setPosition(p);
     locit.setContainer(this);
     return (Locator) locit;
}
```

Locator-Based PQ (contd.)

```
   public void insert(Locator loc) throws
InvalidKeyException {
      locInsert((LocItem) loc);
   }
   public Locator insert(Object k, Object e) throws
InvalidKeyException {
      LocItem locit = new LocItem(k, e, null, null);
      return locInsert(locit);
   }
   public void insertItem (Object k, Object e) throws
InvalidKeyException {
      insert(k, e);
   }
   public void remove(Locator loc) throws
InvalidLocatorException {
      locRemove(loc);
   }
public Object removeMinElement () throws
EmptyContainerException {
      Object toReturn = minElement();
      remove(min());
      return toReturn;
   }
```