Priority Queues Continued…

HEAPS

HEAPS

- The two strategies for implementing a priority queue ADT in the previous section demonstrate an interesting trade-off.

HEAPS

- The two strategies for implementing a priority queue ADT in the previous section demonstrate an interesting trade-off.
- When using an **unsorted list** to store entries, we can perform **insertions in O(1)** time, but **finding or removing** an element with minimum key requires an **O(n)**-time loop through the entire collection.
- Ifusing a **sorted list**, we can trivially **find or remove** the **minimum** element in **O(1)** time, but **adding** a new element to the queue may require **O(n)** time to restore the sorted order.

HEAPS

When using an unsorted list to store entries, we can perform

insertions in O(1) time,

but

finding or removing an element with minimum key requires an O(n)-time loop through the entire collection.

HEAPS

- In contrast, if using a sorted list, we can trivially

find or remove the minimum element in O(1) time,

but

adding a new element to the queue may require O(n) time to restore the sorted order.

Heaps

- In this section, we provide a more efficient realization of a priority queue using a data structure called a **binary heap.**
- This data structure allows us to perform both insertions and removals in logarithmic time, which is a significant improvement over the list-based implementations discussed

Heaps

- This data structure allows us to perform both

insertions and removals in logarithmic time,

which is a significant improvement over the list-based implementations discussed

Heap

- A heap is a tree-based structure,
- A binary heap only specifies the relationship between a parent and its children.
- For a **min-heap**, **a node must be less than all of its children** and it, in turn, must be greater than its parent (if any).
- Thus, a binary min-heap is a binary tree that satisfies the min-heap property.

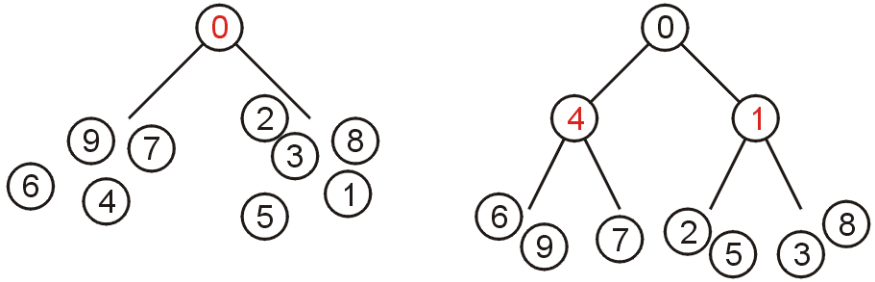
Important: there is no other relationship between the children of a node other than they are all greater than their common parent. The failure to comprehend this has previously been the greatest source of errors.

Min Heaps

Definition

A non-empty binary tree is a min-heap if

- The key associated with the root is less than or equal to the keys associated with either of the sub-trees (if any)
- Both of the sub-trees (if any) are also binary min-heaps



From this definition:

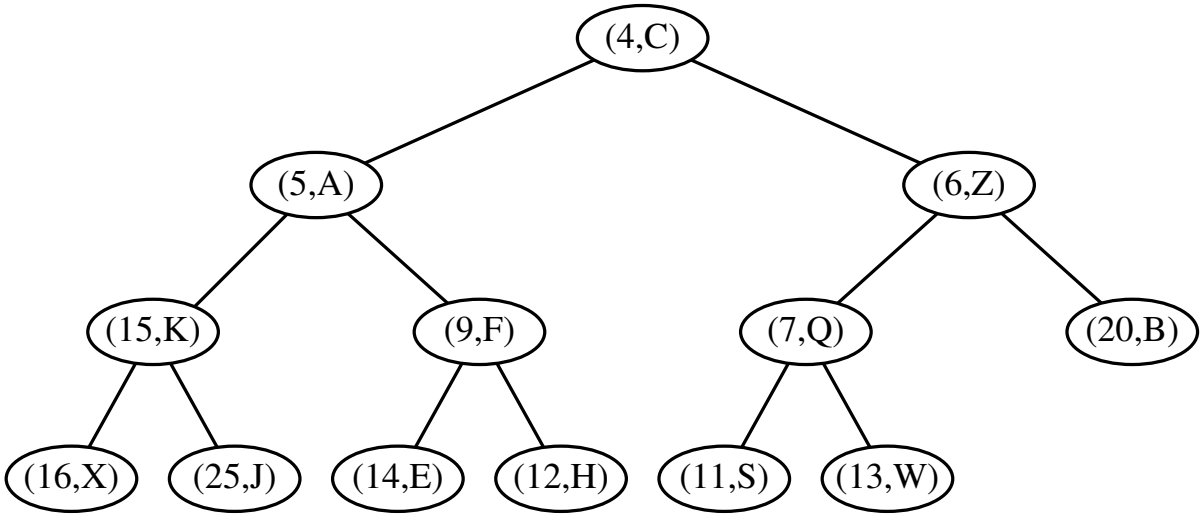
- A single node is a min-heap
- All keys in either sub-tree are greater than the root key

BST vs Binary Heaps

Binary Search Tree	Binary Heap
Given a node: All objects in the left sub-tree less than the node, All objects in the right sub-tree are greater than the node, and both sub trees are also a binary search tree.	Given a node: All strict descendants are greater than the node, And both sub-trees are also binary heaps.

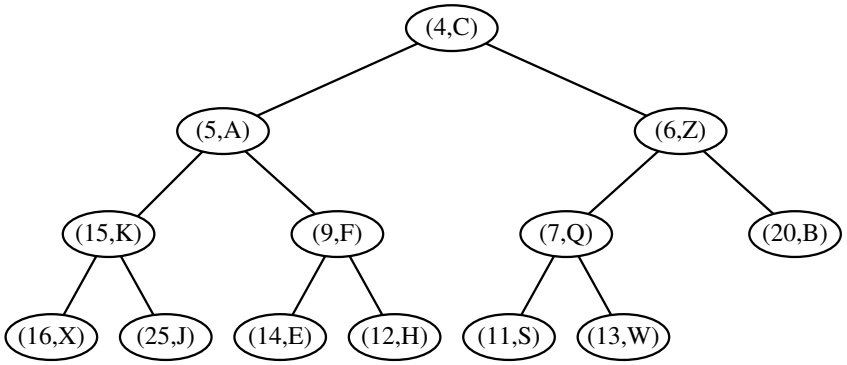
Implementing the Heap Structure

The Heap Data Structure with the key value pair



Example of a heap storing 13 entries with integer keys. The last position is the one storing entry (13,W).

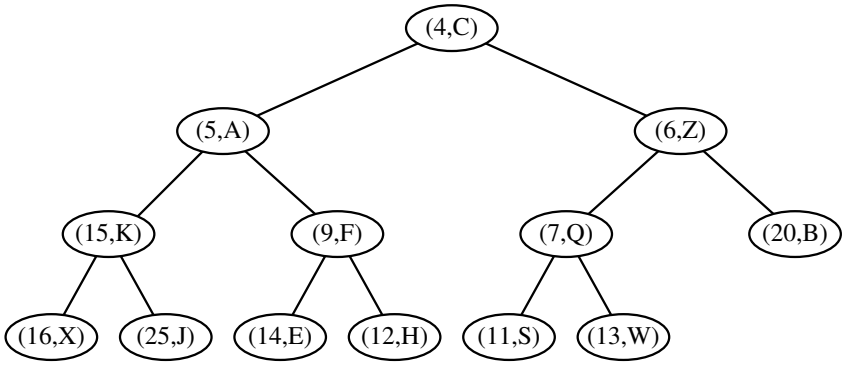
The Heap Data Structure



- This binary tree satisfies 2 additional properties
  - Relational property (definition of the min heap as discussed earlier)
  - Structural property
- **The relational property** is : In a heap T , for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent.

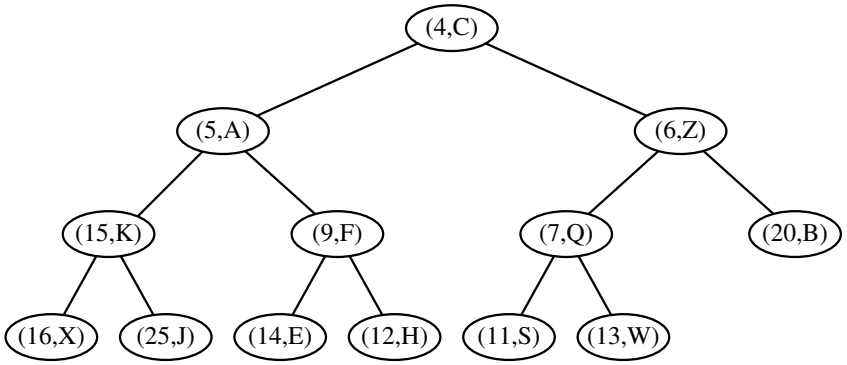
This relational property is the **Heap-Order Property itself.**

The Heap Data Structure



- This binary tree satisfies 2 additional properties
  - Relational property
  - Structural property
- **The structural property** is : The binary tree must be **complete**.

The Height of a Heap



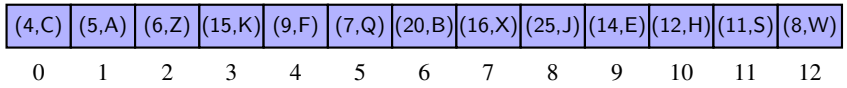
- Let h denote the height of T. Insisting that T be **complete** :
- A heap T storing n entries has height

$$h = \lfloor \log n \rfloor$$

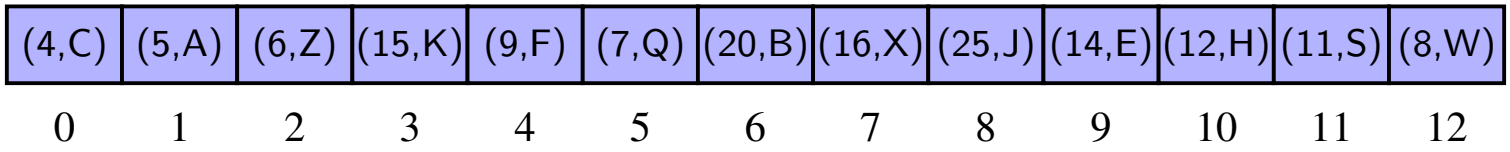
Remember : Array Based Representation of a Complete Binary Tree

Array Based Representation of a Complete Binary Tree

- The array-based representation of a binary tree is especially suitable for a complete binary tree T .
- We recall that in this implementation, the elements of T are stored in an array-based list A such that the element at position p in T is stored in A with index equal to the level number f(p) of p, defined as follows:



Array Based Representation of a Complete Binary Tree



- If p is the root of T, then f(p)=0. If p is the left child of position q, then f(p) = 2f(q)+1.
- If p is the right child of position q, then f(p) = 2f(q)+2.

What is the index value for (9,F)'s left child?

Implementing a Priority Queue with a Min Heap

Implementing a Priority Queue with a Heap Design Decisions

- We will use and store **<key ,value>** pairs as items in the heap.
- The **len** and **is empty** methods can be implemented based on examination of the tree, and
- the **min** operation is equally trivial because the heap property assures that the element at the root of the tree has a minimum key.
- The interesting algorithms are those for implementing the **add** and **remove min** methods.

Adding an Item to the Heap

- Let us consider how to perform **add(k,v)** on a priority queue implemented with a heap T.

1. **We store the pair (k,v) as an item at a new node of the tree.**
2. To maintain the **complete binary tree property,**
   1. that new node should be placed at a position p just beyond the rightmost node at the bottom level of the tree,
   2. or as the leftmost position of a new level, if the bottom level is already full (or if the heap is empty).

However as demonstrated earlier Executing Step 1 may cause a violation on **heap-order property.**

Adding an Item to the Heap
Up Heap Bubbling (Percolate Up)

- Let us consider how to perform **add(k,v)** on a priority queue implemented with a heap T.

1. **We store the pair (k,v) as an item at a new node of the tree.**
2. To maintain the complete binary tree property, that new node should be placed at a position p just beyond the rightmost node at the bottom level of the tree,
3. or as the leftmost position of a new level, if the bottom level is already full (or if the heap is empty).

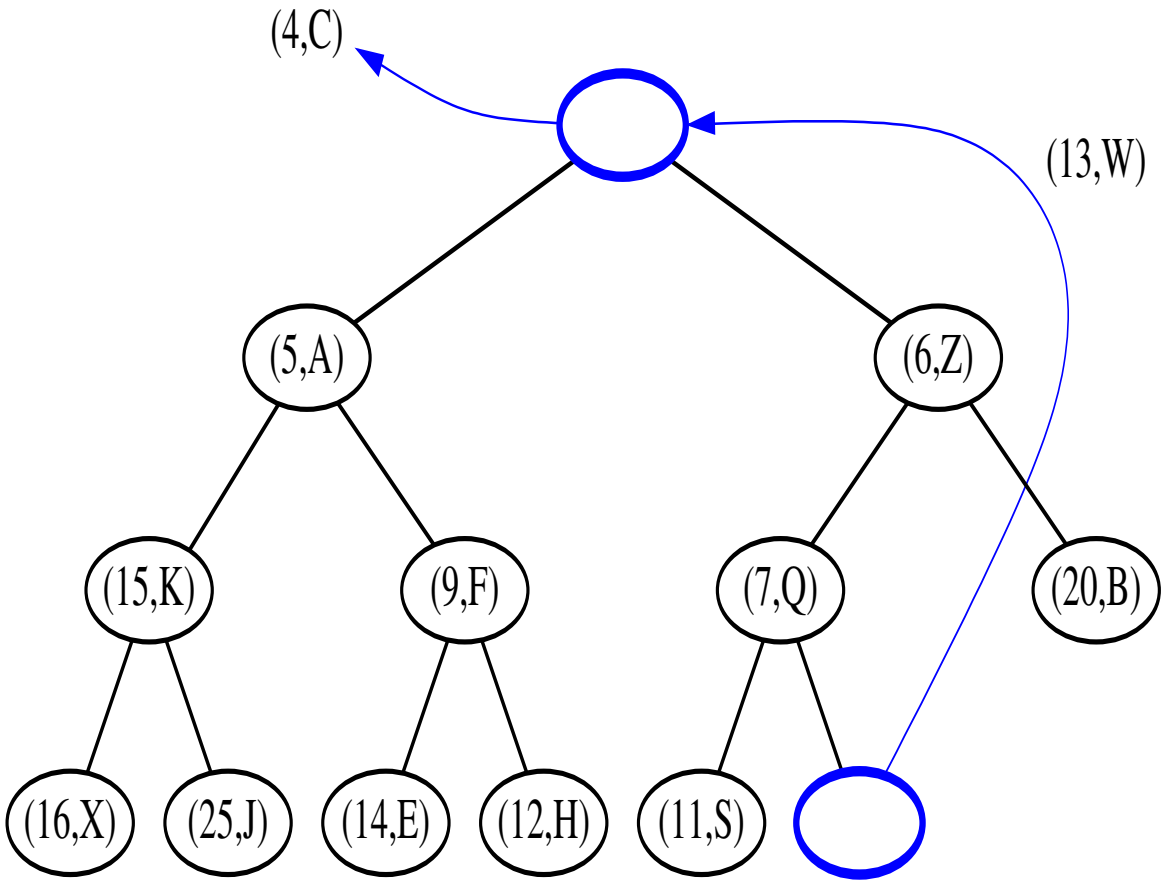
Executing Step 1 may cause a violation on **heap-order property.**

We must apply : Up heap bubbling (percolate up) as shown earlier in simulation slides.

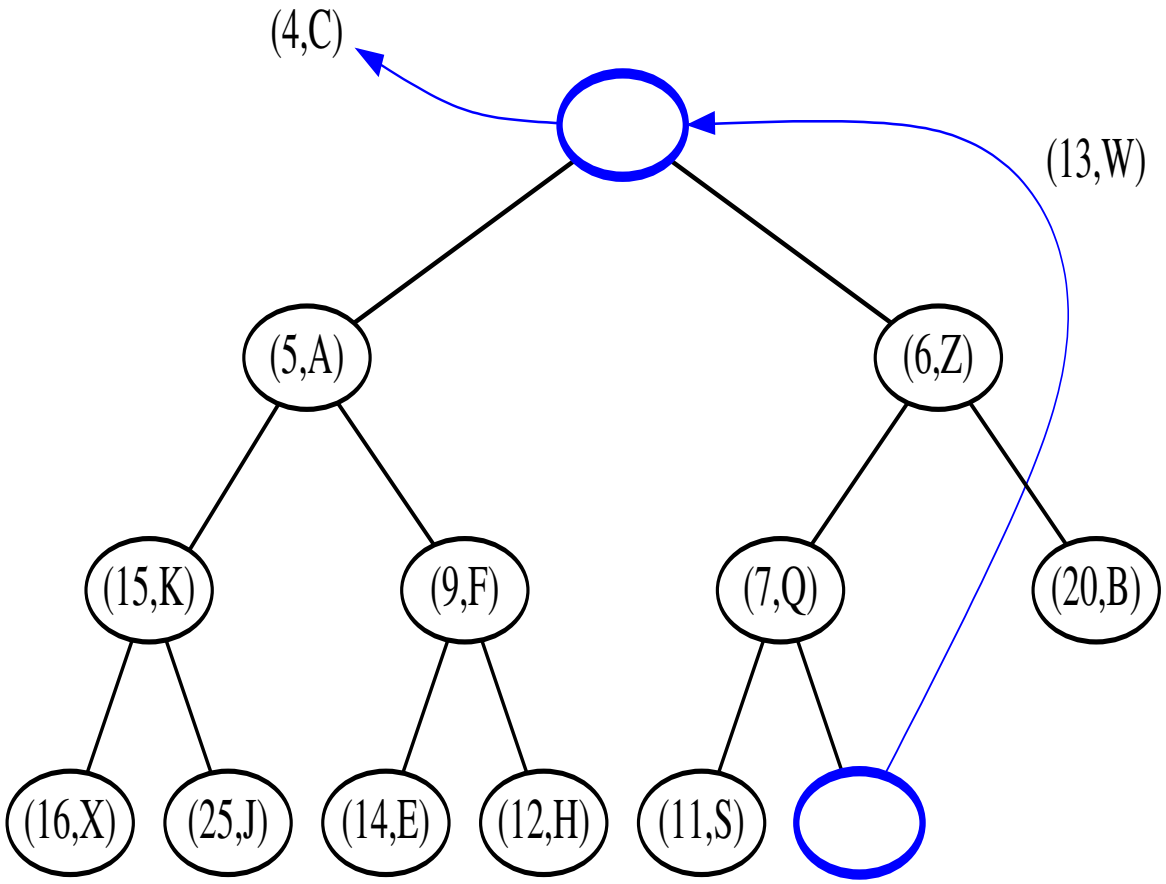
Removing the Item with Min key
Down Heap Bubbling

- The min element resides at the root so when we remove that:
  - We must ensure that the shape of the heap respects the complete binary tree property by deleting the leaf at the last position p of T , defined as the rightmost position at the bottommost level of the tree.
  - To preserve the item from the last position p, we copy it to the root r (in place of the item with minimum key that is being removed by the operation).
  - Just simply satisfying the tree is complete not good enough, it must remain as a min heap therefore we must find the accurate place for that child that replaced the root.

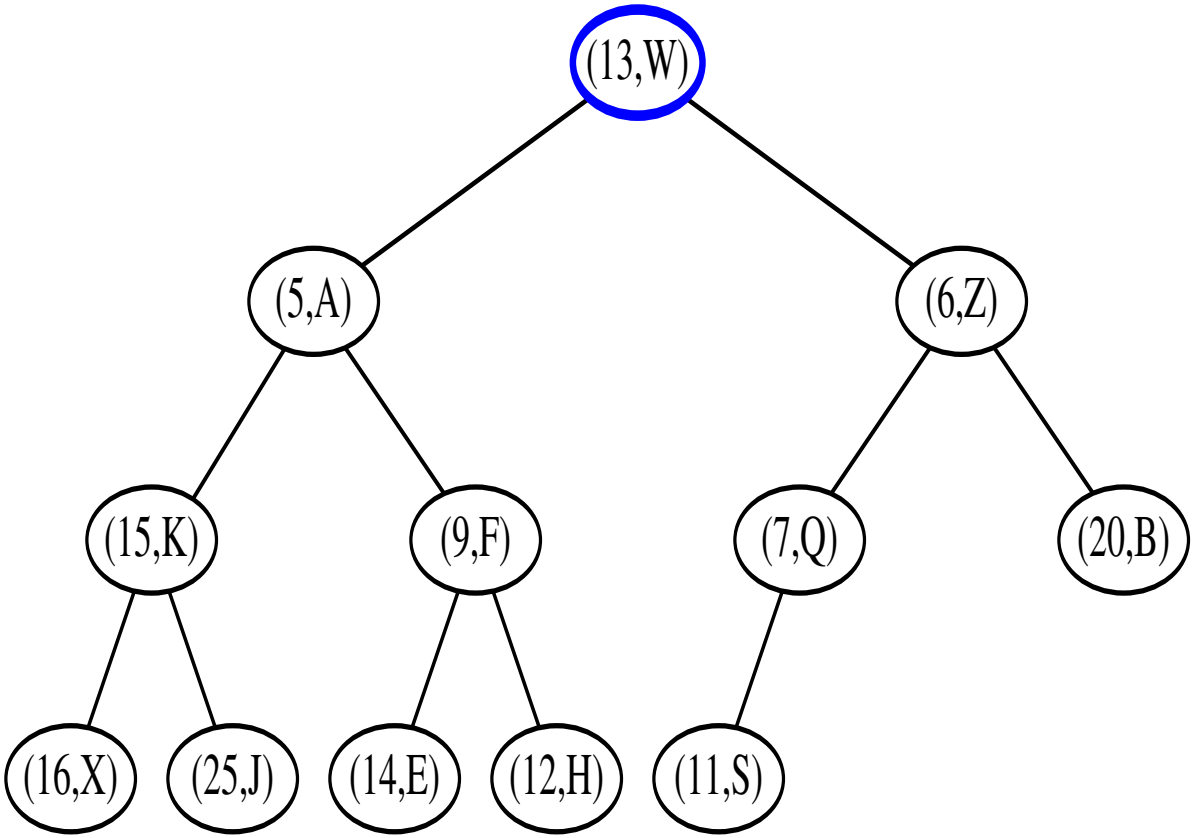
Let's work on remove
How do we remove an element and keeping at a complete tree at the same time?



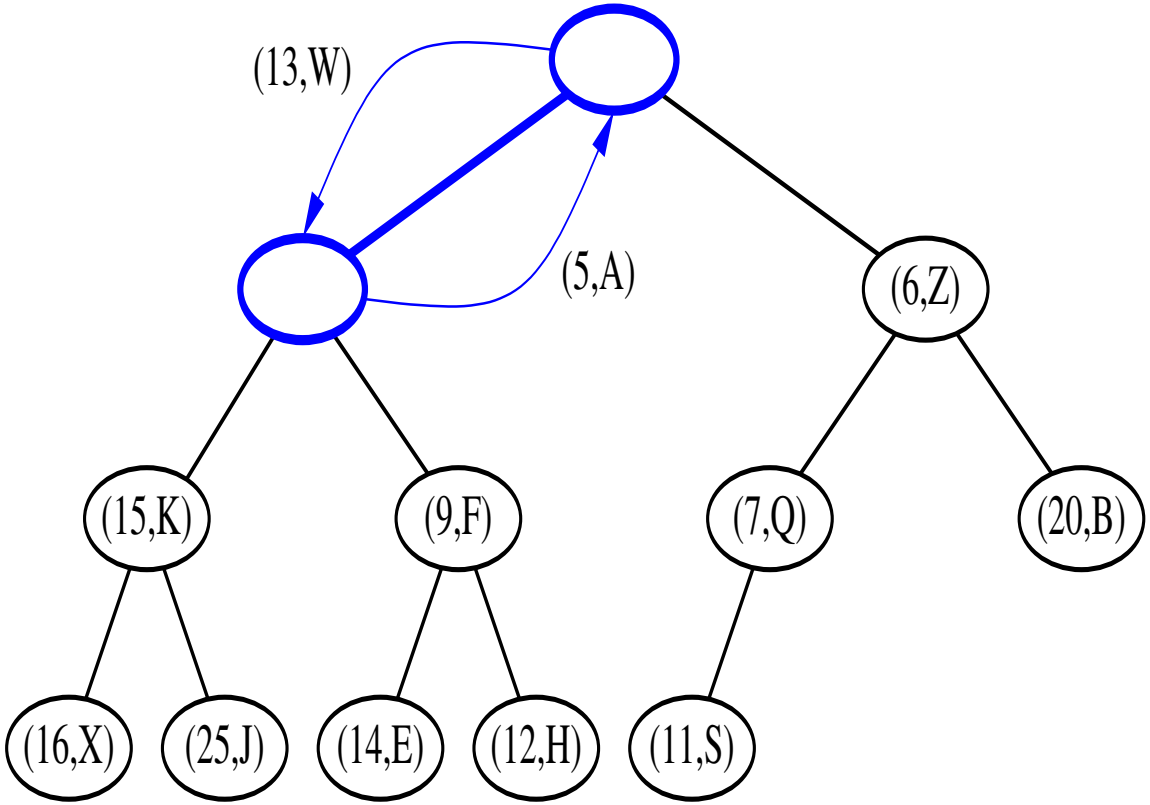
Removing the Item with Min key



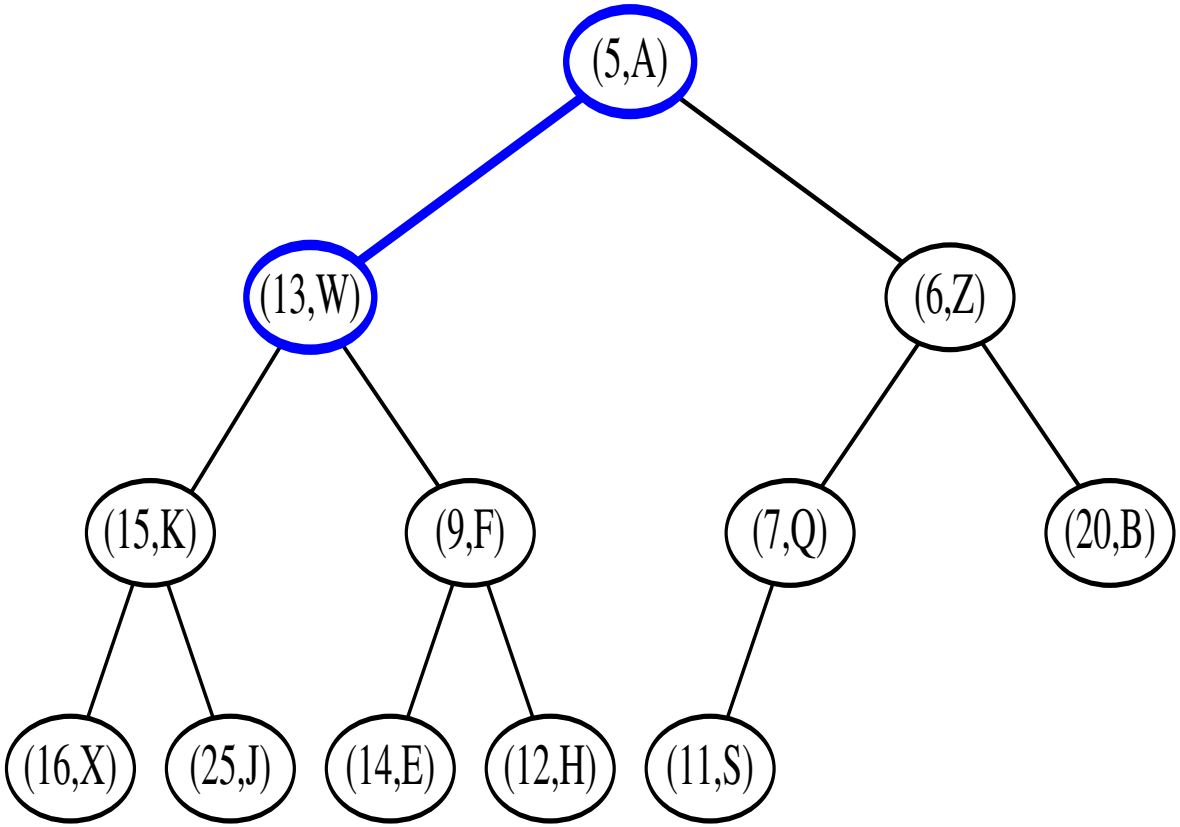
Removing the Item with Min key



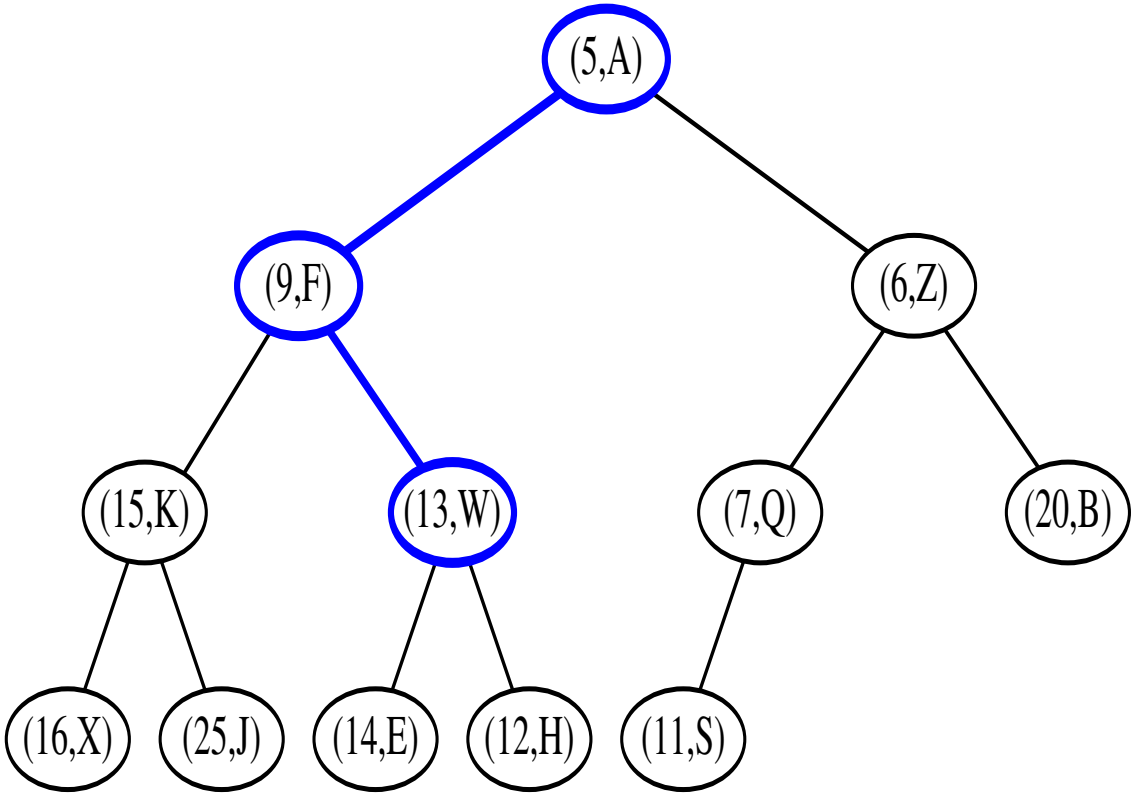
Removing the Item with Min key



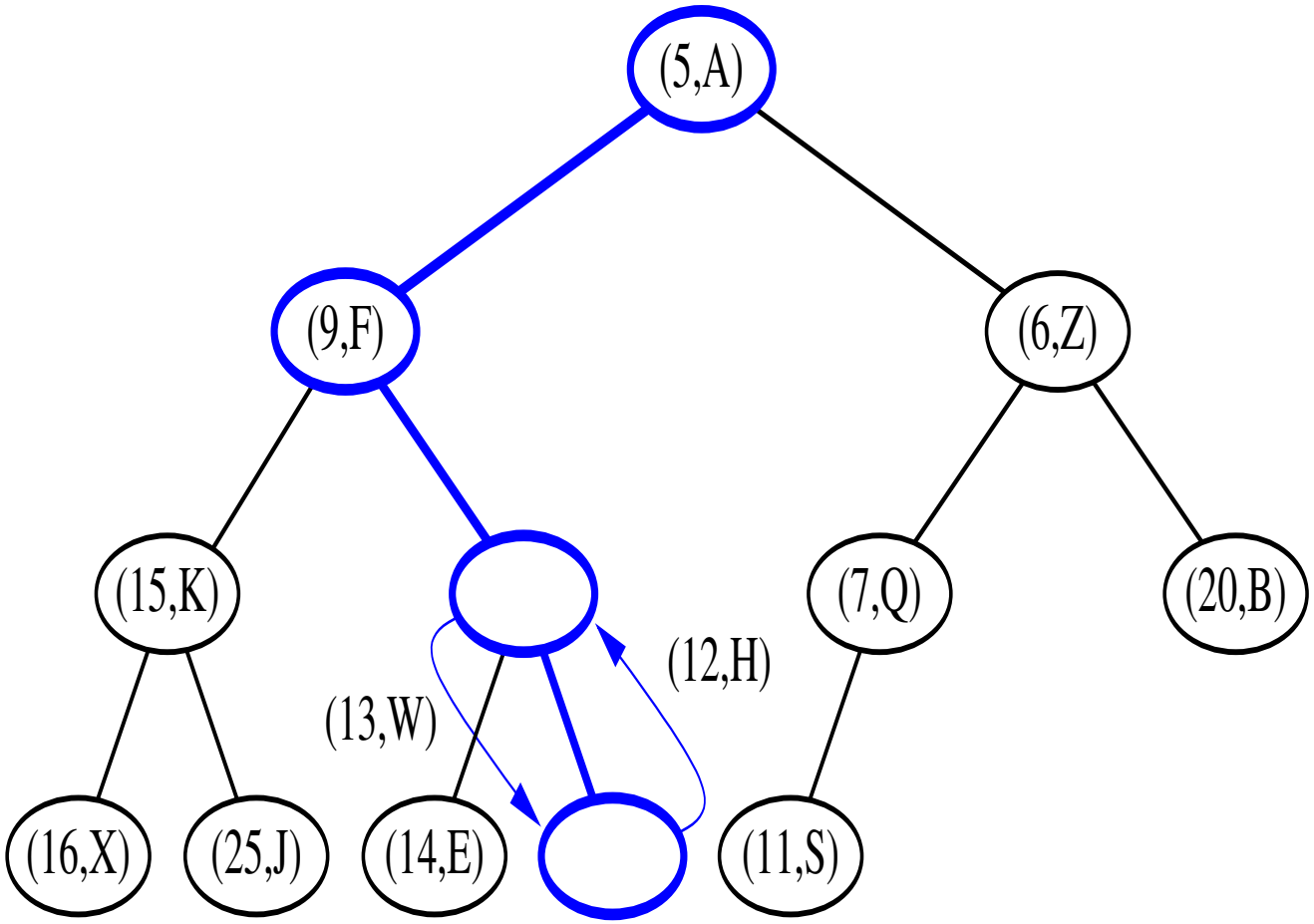
Removing the Item with Min key



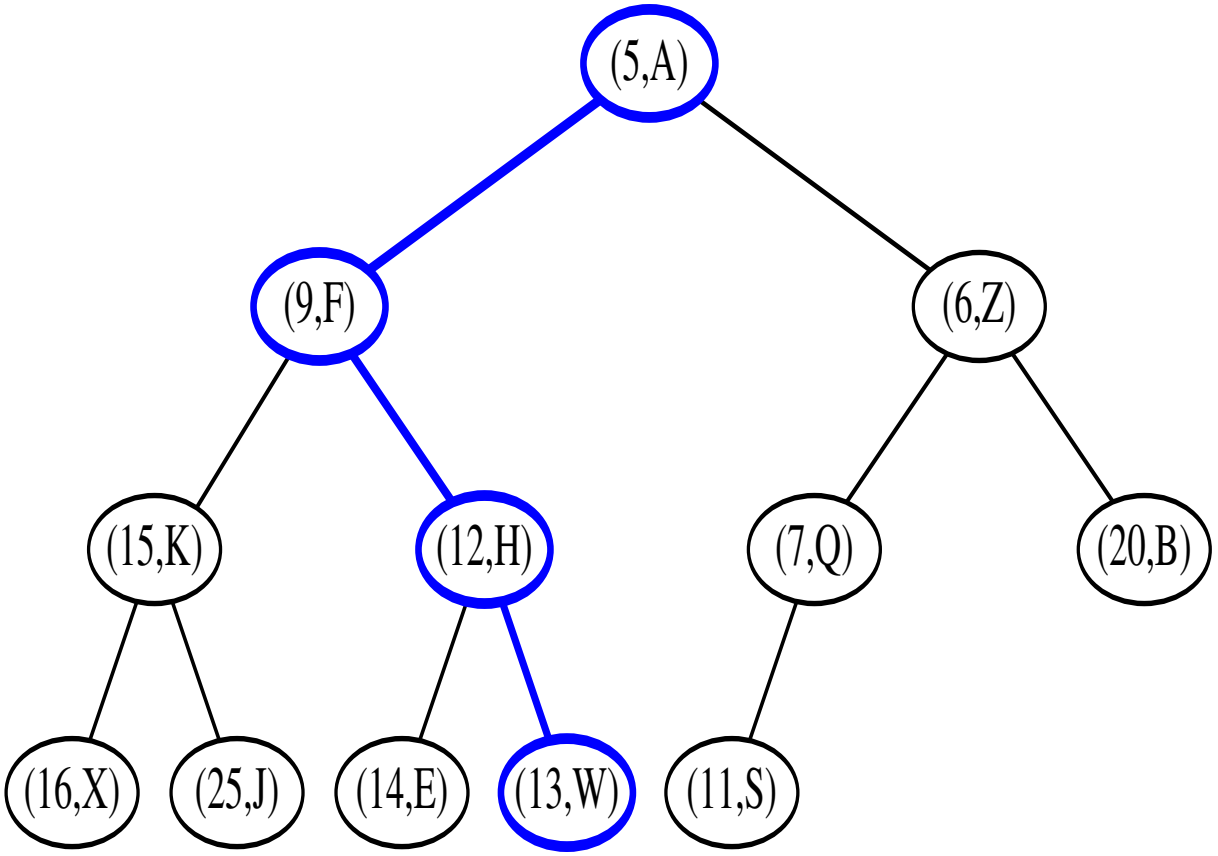
Removing the Item with Min key



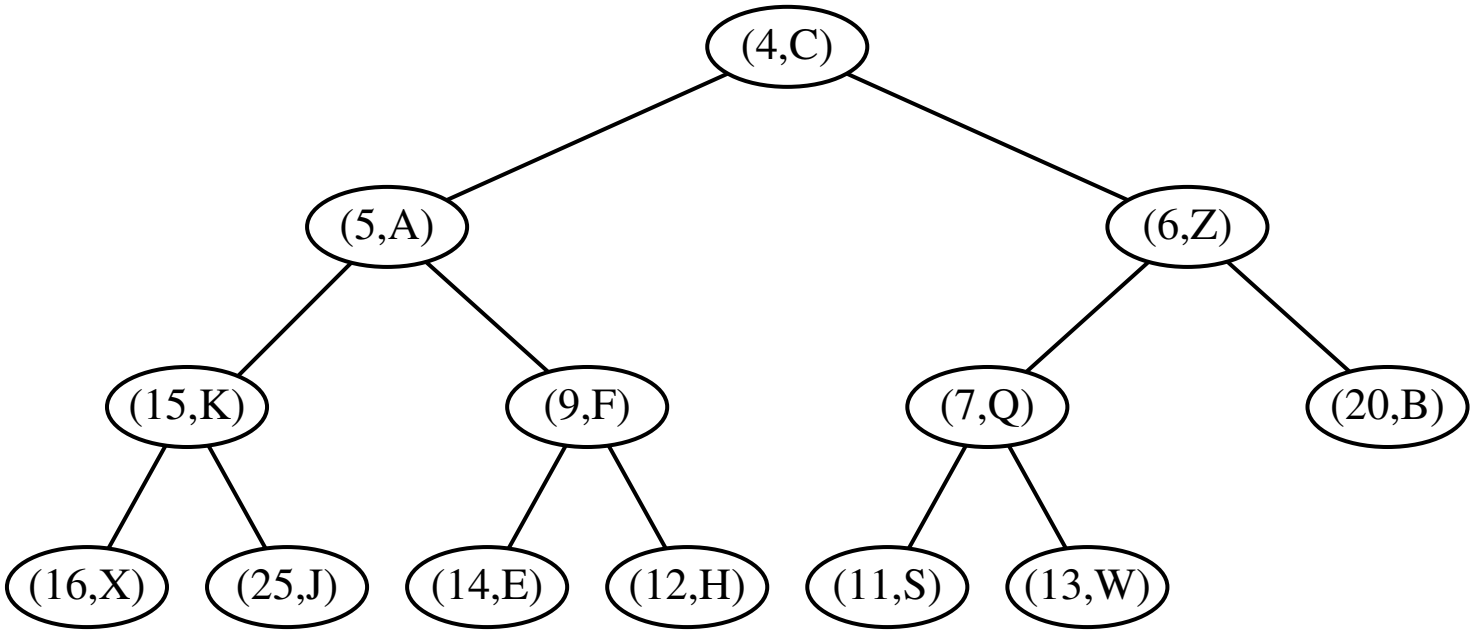
Removing the Item with Min key



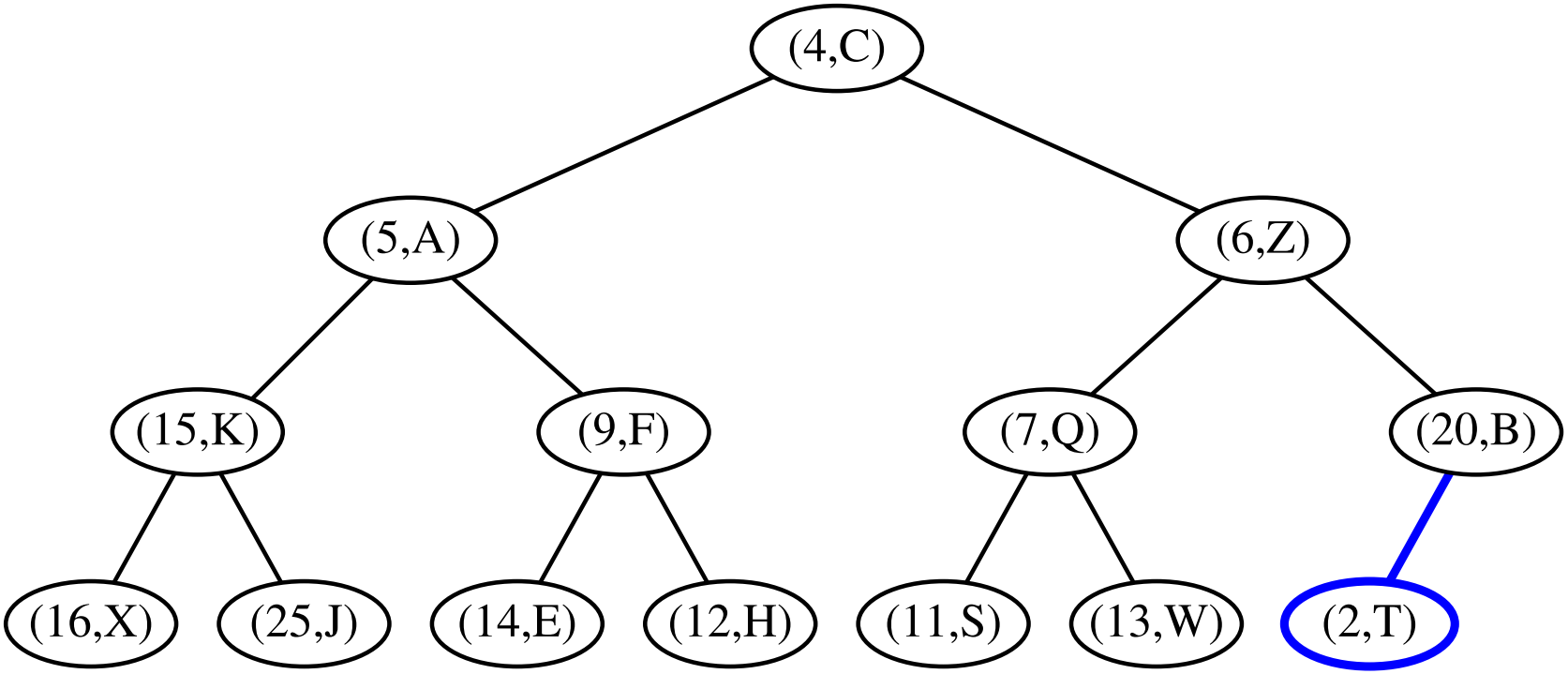
Removing the Item with Min key



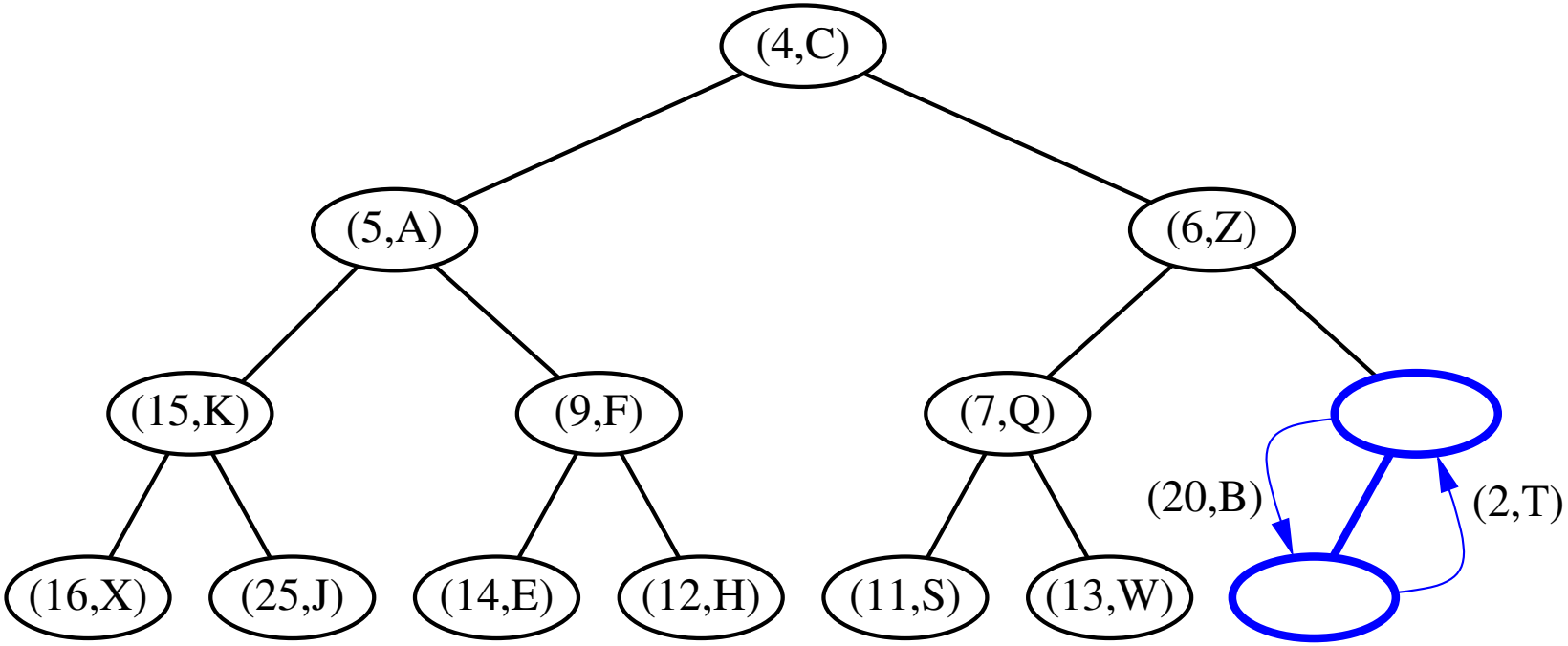
Adding a new item to Heap : Up heap Bubbling (percolate up)



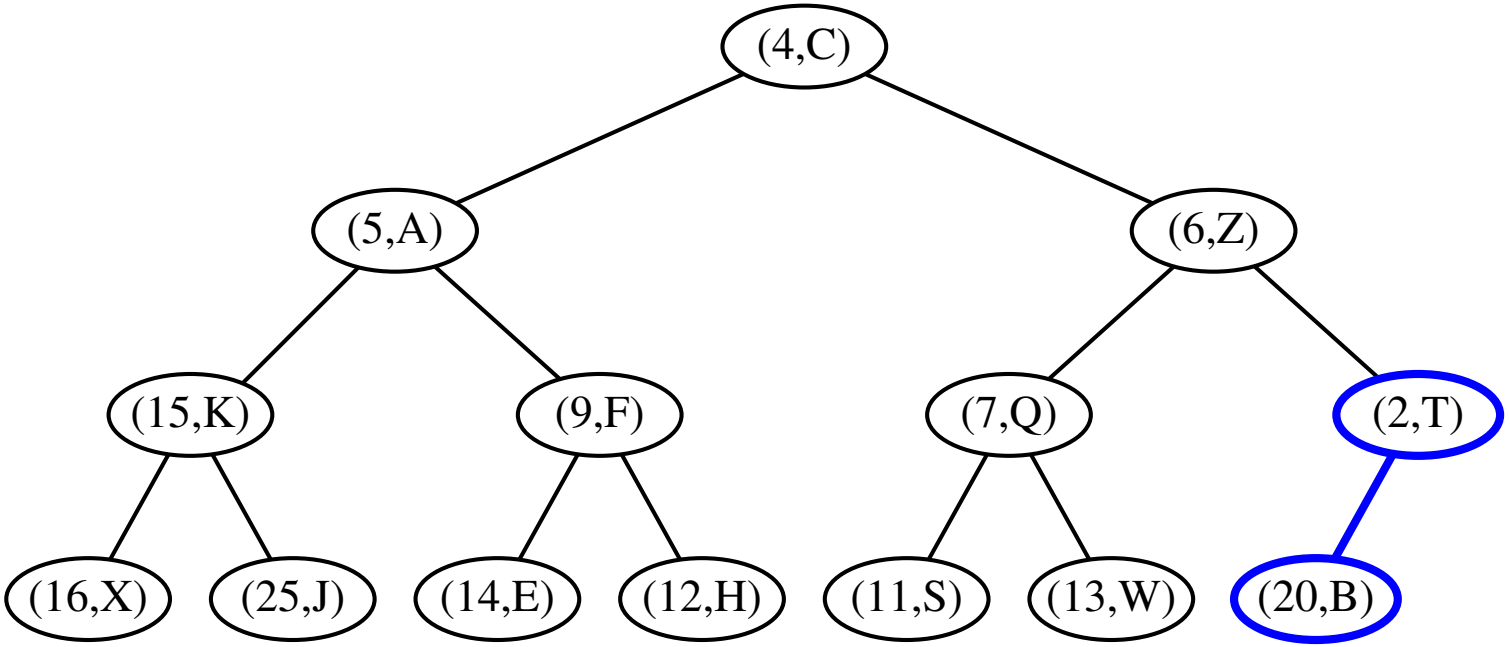
Adding a new item to Heap : Up heap Bubbling (percolate up)



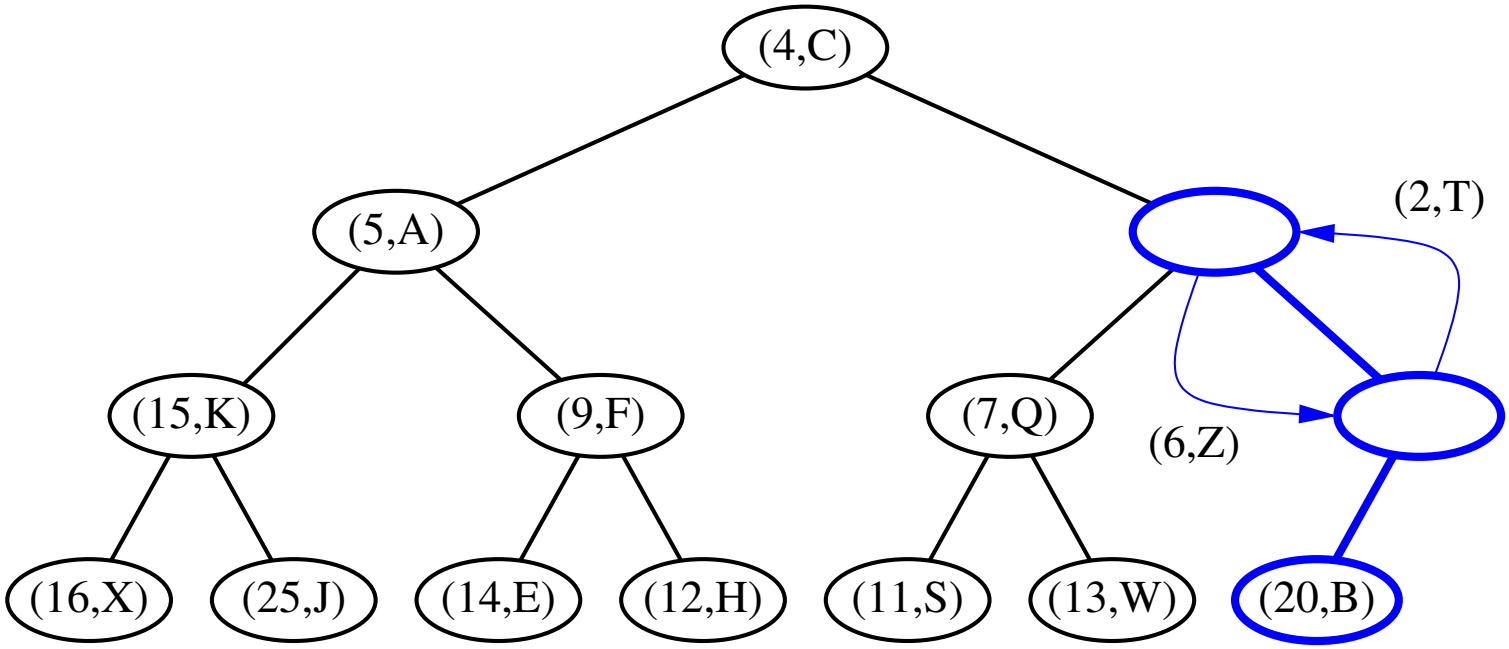
Adding a new item to Heap : Up heap Bubbling (percolate up)



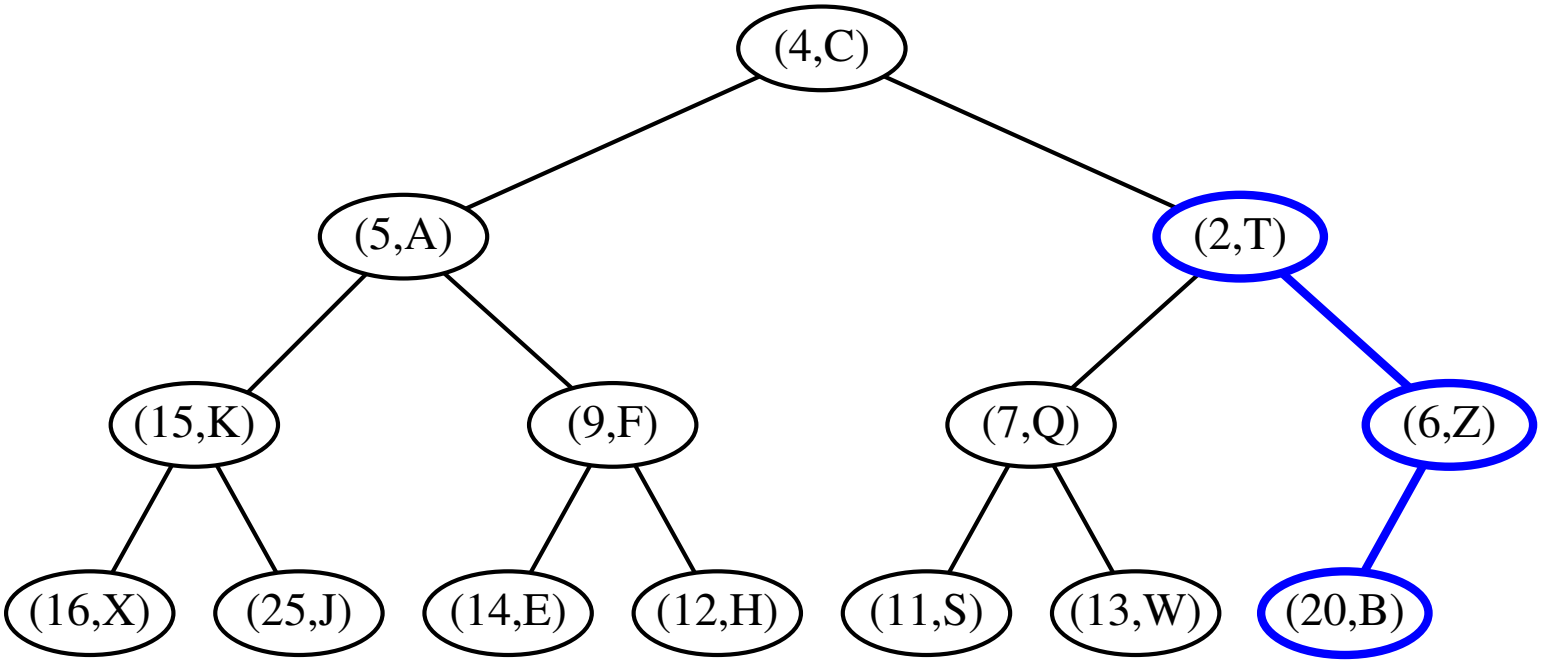
Adding a new item to Heap : Up heap Bubbling (percolate up)



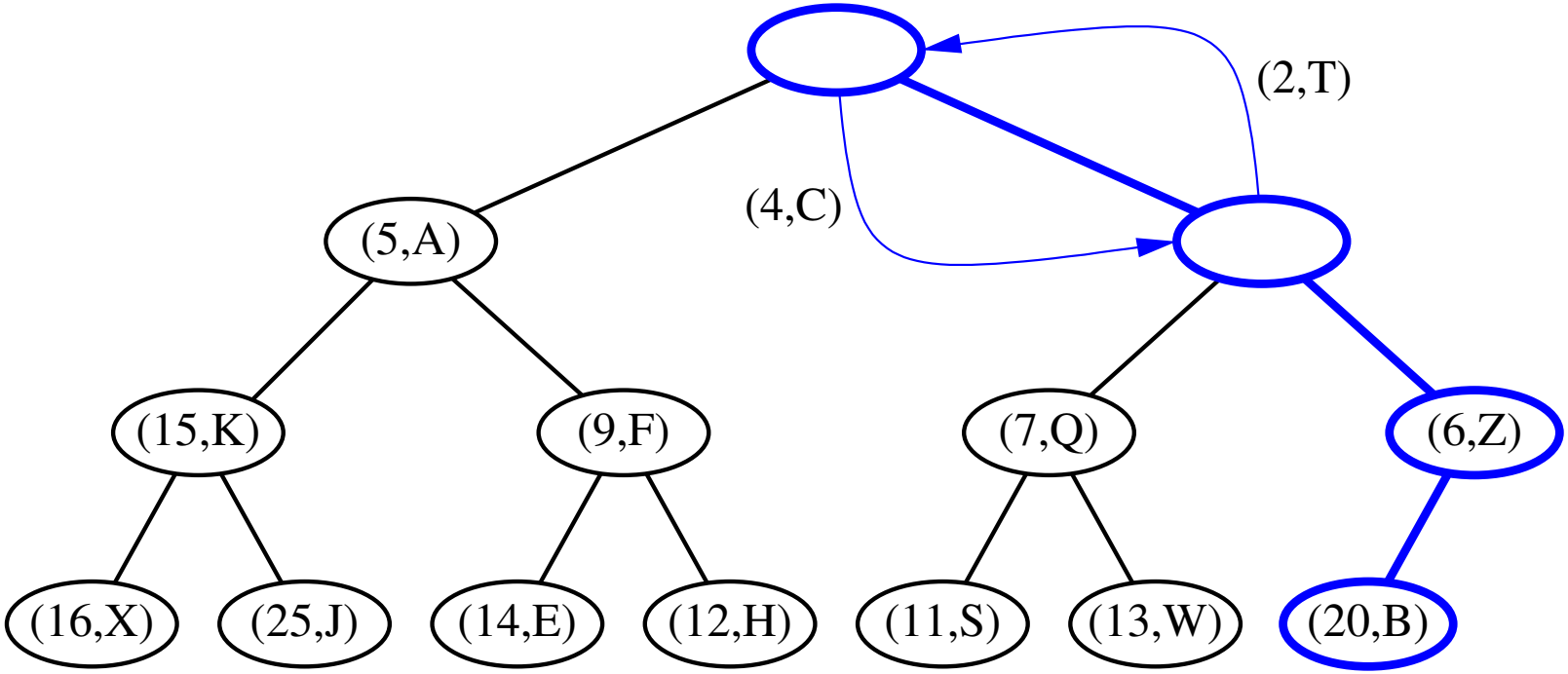
Adding a new item to Heap : Up heap Bubbling (percolate up)



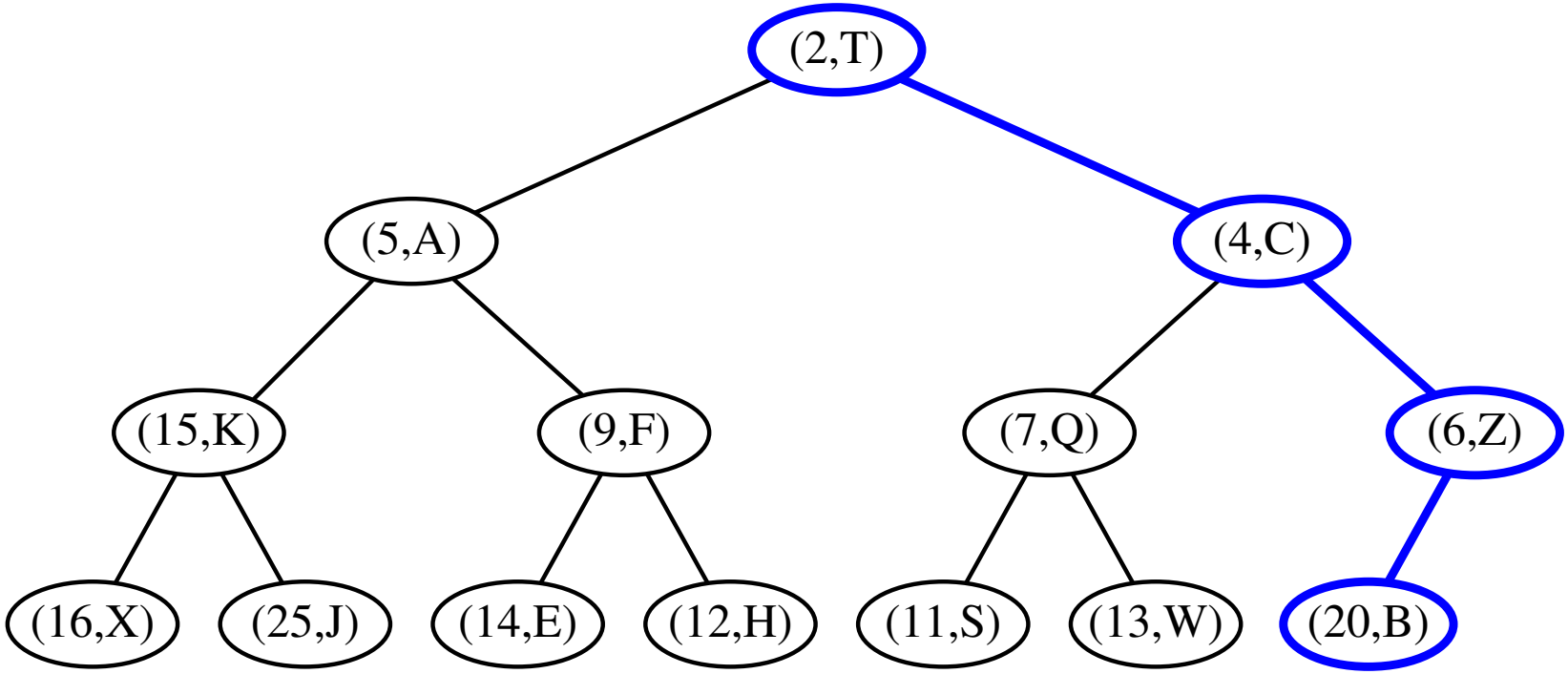
Adding a new item to Heap : Up heap Bubbling (percolate up)



Adding a new item to Heap : Up heap Bubbling (percolate up)



Adding a new item to Heap : Up heap Bubbling (percolate up)

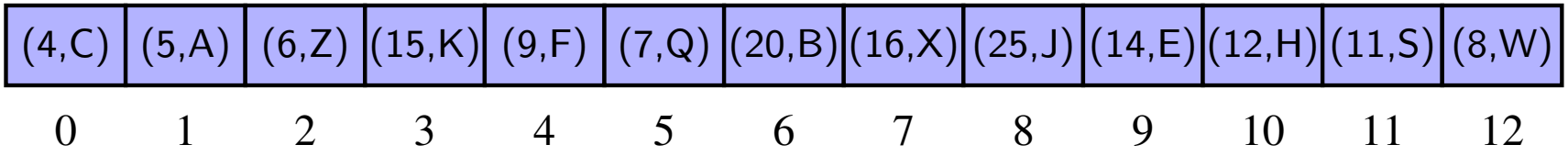


Before we implement our P.Queue using a min heap ….

Remember : Array Based Representation of a Complete Binary Tree

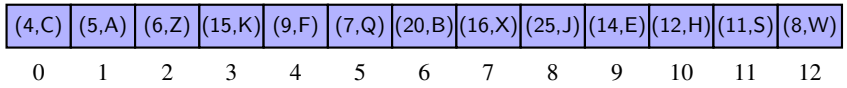
Array Based Representation of a Complete Binary Tree

- The array-based representation of a binary tree is especially suitable for a complete binary tree T .



Recall that we are using an Array Based Representation of a Complete Binary Tree

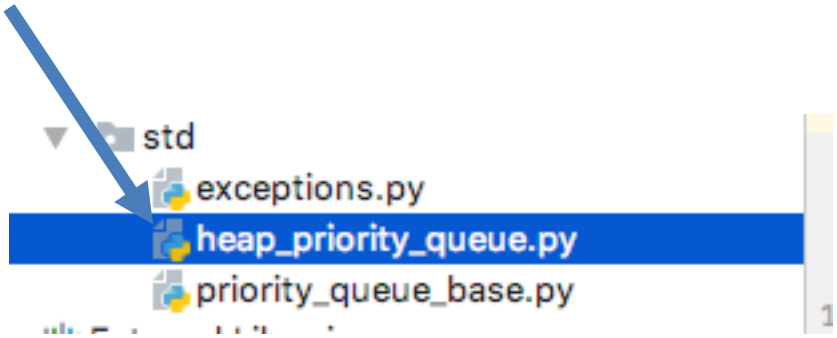
- **If p is the root of T, then f(p)=0. If p is the left child of position q, then f(p) = 2f(q)+1.**
- **If p is the right child of position q, then f(p) = 2f(q)+2.**



Let’s code

GO TO D2L

- Download the Lecture incomplete Python code.
- Start up Pycharm and copy the py files under a new project.
- Go to



- Complete _parent, _left and _right functions

Constructors

```
#---------------------------- public behaviors ----------------------------

def __init__(self):
  """Create a new empty Priority Queue."""
  self._data = []

def __len__(self):
  """Return the number of items in the priority queue."""
  return len(self._data)
```

Parent, left and right

```
def _parent(self, j):
  return (j-1) // 2
#----------------------------
def _left(self, j):
  return 2*j + 1
#----------------------------
def _right(self, j):
  return 2*j + 2
```

Why are we tapping into implementing a priority queue using an array-based heap ?

- Implementing a priority queue using an array-based heap representation allows us to avoid some complexities of a node-based tree structure.
- In particular, the **add and remove min** operations of a priority queue both **depend on locating the last index of a heap of size n.**
- **With the array-based representation, the last position is at index n − 1 of the array.**
- Locating the last position of a complete binary tree implemented with a linked structure requires more effort.

Why are we tapping into implementing a priority queue using an array-based heap ?

- Also
- If the size of a priority queue is not known in advance, use of an array-based representation does introduce the need to dynamically resize the array on occasion, as is done with a Python list.
- ***The space usage of such an array-based representation of a complete binary tree with n nodes is O(n), and the time bounds of methods for adding or removing elements become amortized.***

Finally: Python Heap Implementation

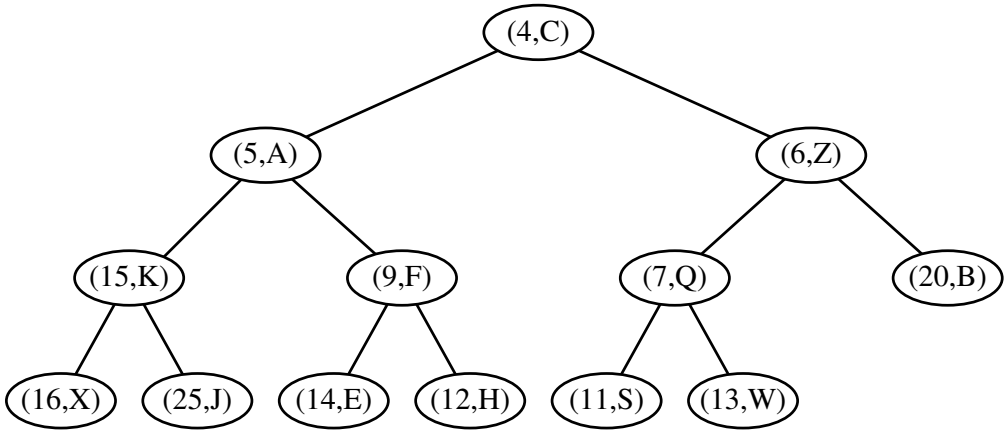
Class: HeapPriorityQueue

```
from priority_queue_base import PriorityQueueBase
from exceptions import Empty
```

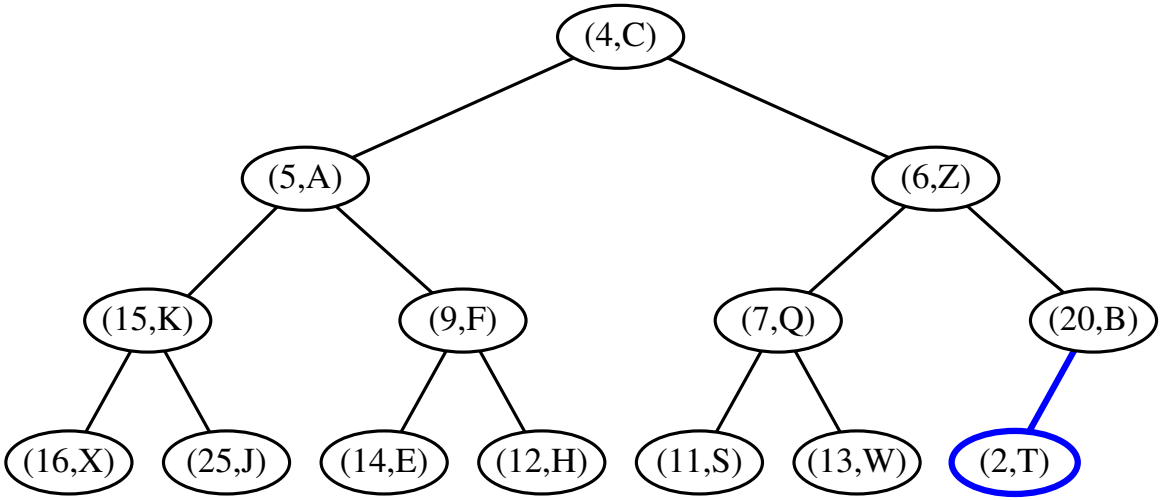

```
class HeapPriorityQueue(PriorityQueueBase):  # base class defines _Item
    """A min-oriented priority queue implemented with a binary heap."""

    # ------------------------------ nonpublic behaviors ------------------------------
    def _parent(self, j):
        return (j - 1) // 2

    def _left(self, j):
        return 2 * j + 1

    def _right(self, j):
        return 2 * j + 2

    def _has_left(self, j):
        return self._left(j) < len(self._data)  # index beyond end of list?

    def _has_right(self, j):
        return self._right(j) < len(self._data)  # index beyond end of list?
```

Let's implement **add**
which means we must implement upheap
bubbling (percolate up)!

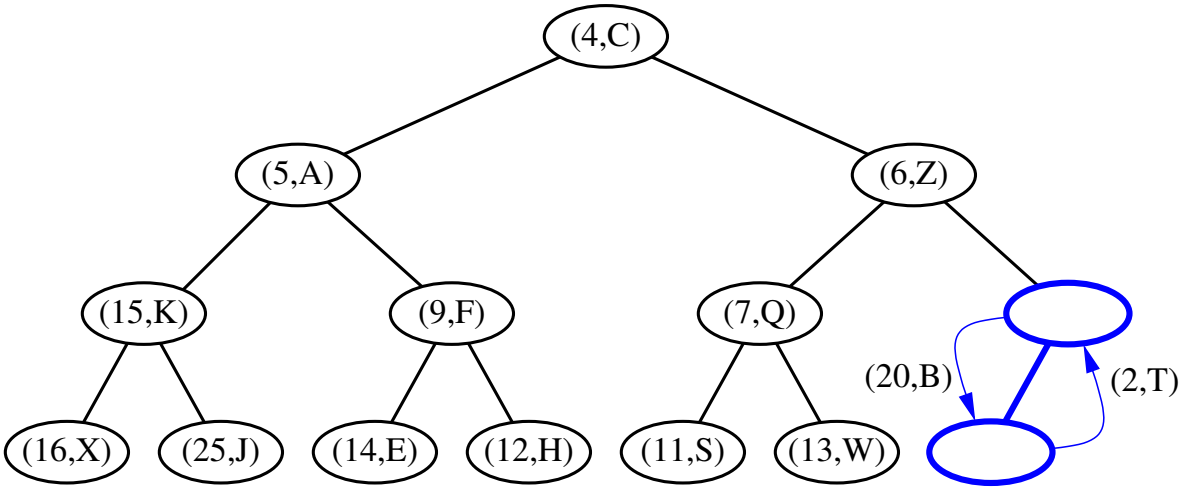
Adding a new item to Heap : Up heap Bubbling (percolate up)



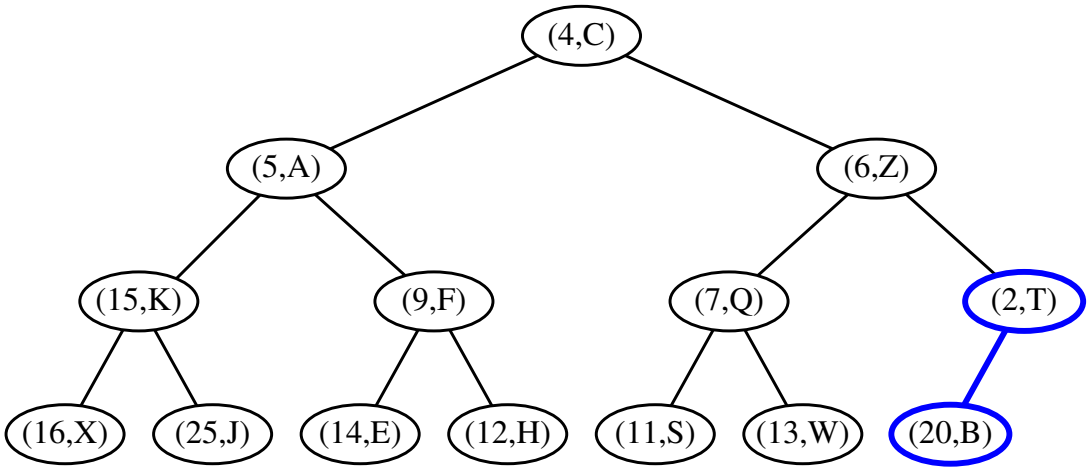
Adding a new item to Heap : Up heap Bubbling (percolate up)



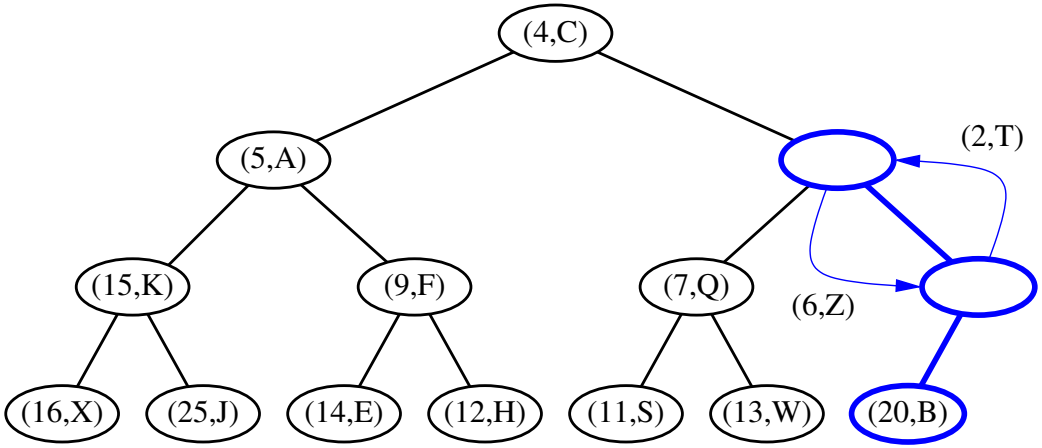
Adding a new item to Heap : Up heap Bubbling (percolate up)



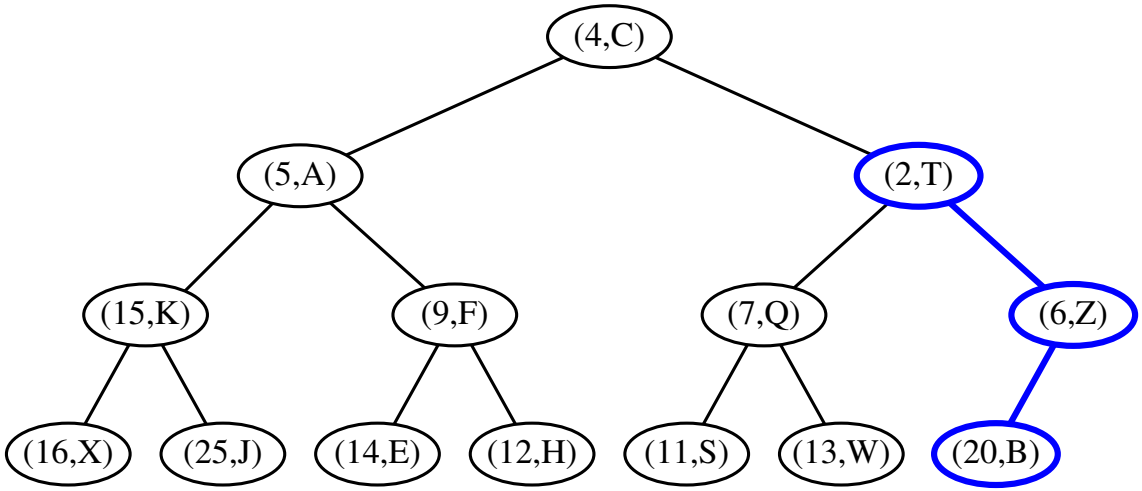
Adding a new item to Heap : Up heap Bubbling (percolate up)



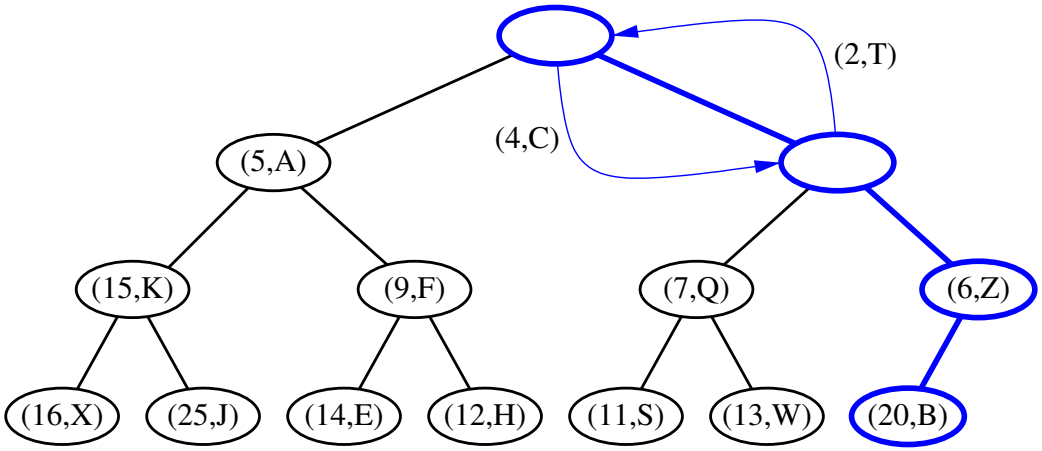
Adding a new item to Heap : Up heap Bubbling (percolate up)



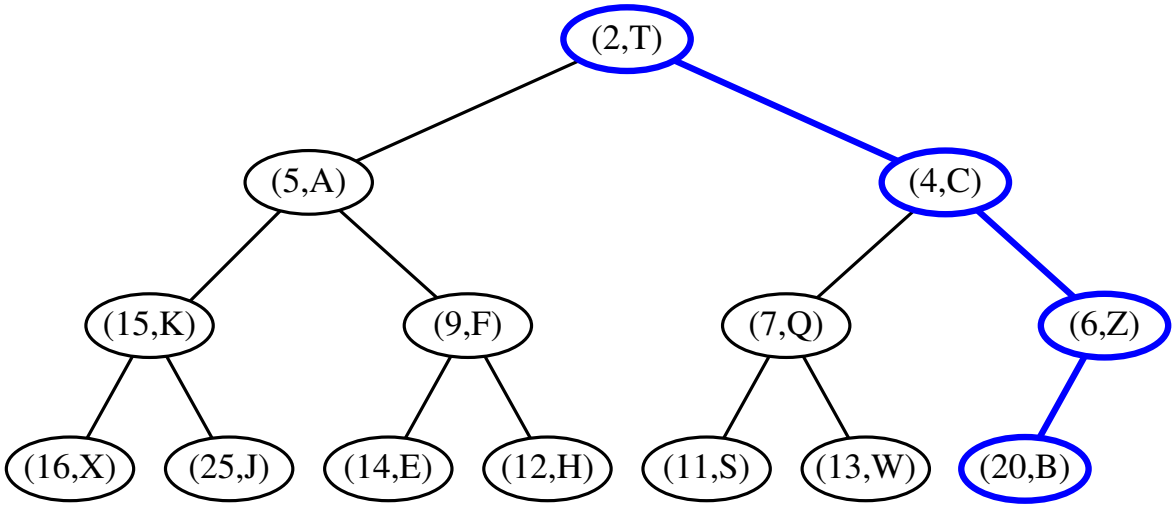
Adding a new item to Heap : Up heap Bubbling (percolate up)



Adding a new item to Heap : Up heap Bubbling (percolate up)



Adding a new item to Heap : Up heap Bubbling (percolate up)



Step 1 implement upheap → used in add

Complete the upheap bubbling
upheap bubbling : non public method will be used during add

Program this!

```
def _swap(self, i, j):
    """Swap the elements at indices i and j of array."""


def _upheap(self, j):
  #local variable parent gets this object's parent j th index
  # check to see:   j > 0  also this object's data[j] is less than  this object's data[parent]:
    #then swap(j, parent)
    # recur at position of parent
```

Completed version
upheap bubbling : nonpublic method will be used during add

```
def _swap(self, i, j):
    """Swap the elements at indices i and j of array."""
    self._data[i], self._data[j] = self._data[j], self._data[i]

def _upheap(self, j):
    parent = self._parent(j)
    if j > 0 and self._data[j] < self._data[parent]:
        self._swap(j, parent)
        self._upheap(parent)  # recur at position of parent
```

Step 2 implement add

Now Complete the add method

```
def add(self, key, value):
    """Append a key-value pair to the priority queue."""
    #append item( key value ) pair
    # upheap newly added position
```

Completed the add method

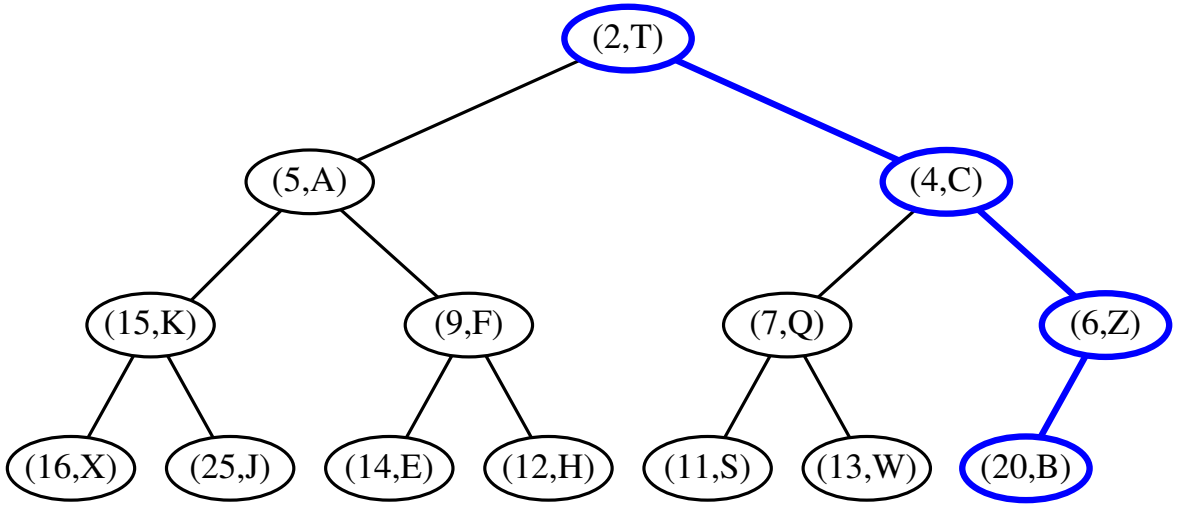
```
def add(self, key, value):
    """Add a key-value pair to the priority queue."""
    self._data.append(self._Item(key, value))
    self._upheap(len(self._data) - 1)  # upheap newly added position
```

Test Add method !

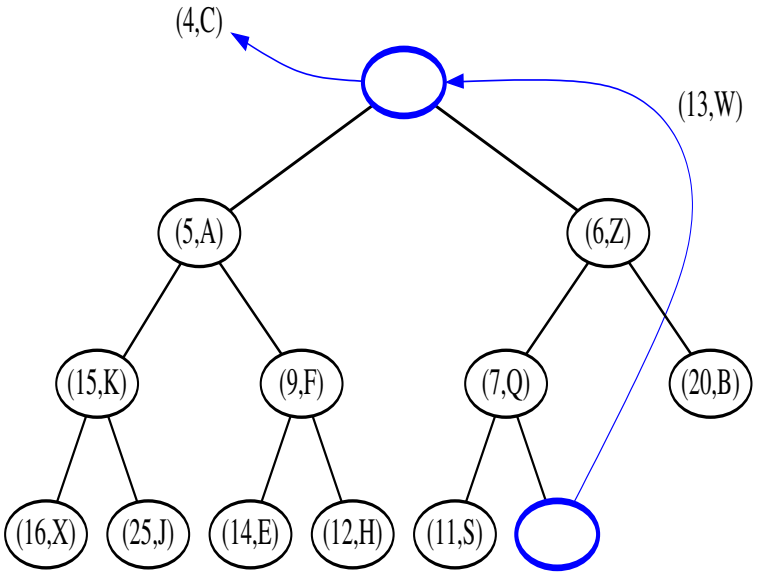
Test the add, use the break point

```
pqueu = HeapPriorityQueue()
pqueu.add(5, 'A')
pqueu.add(4,'B')
pqueu.add(3,'B')
pqueu.add(9, 'C')
pqueu.add(3, 'B')
pqueu.add(1,'A')
```

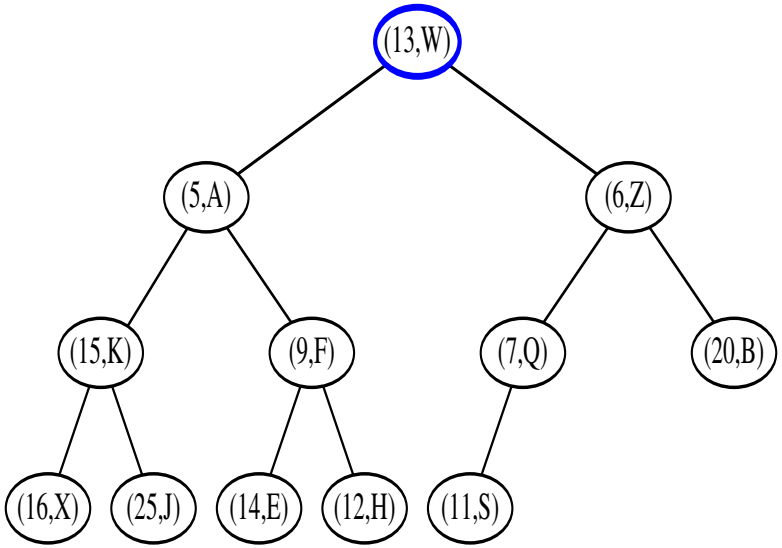
Let's work on remove
How do we remove an element and keeping at a complete tree at the same time?



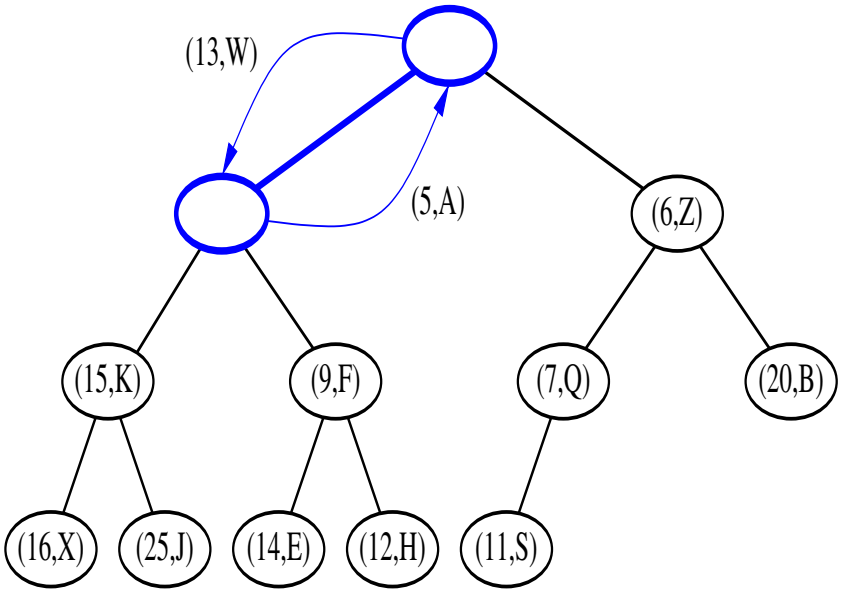
Removing the Item with Min key



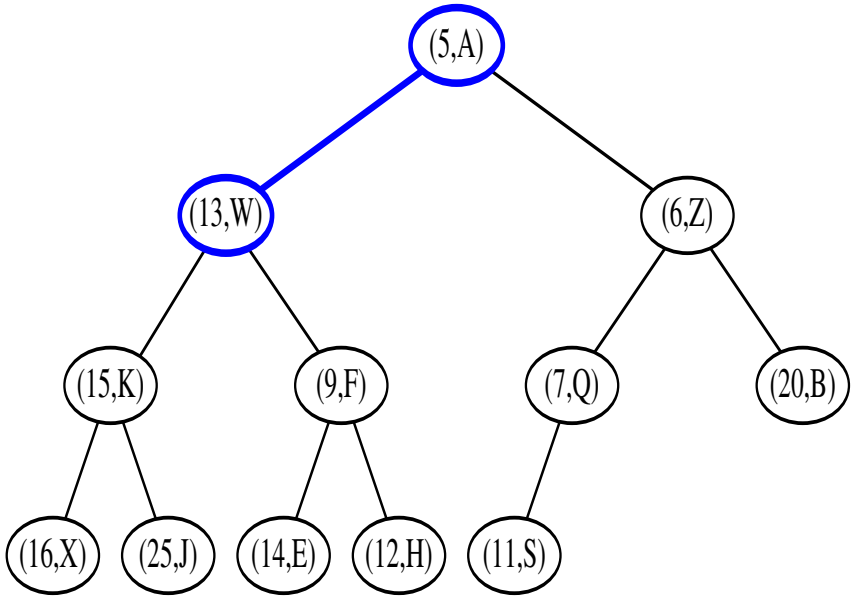
Removing the Item with Min key



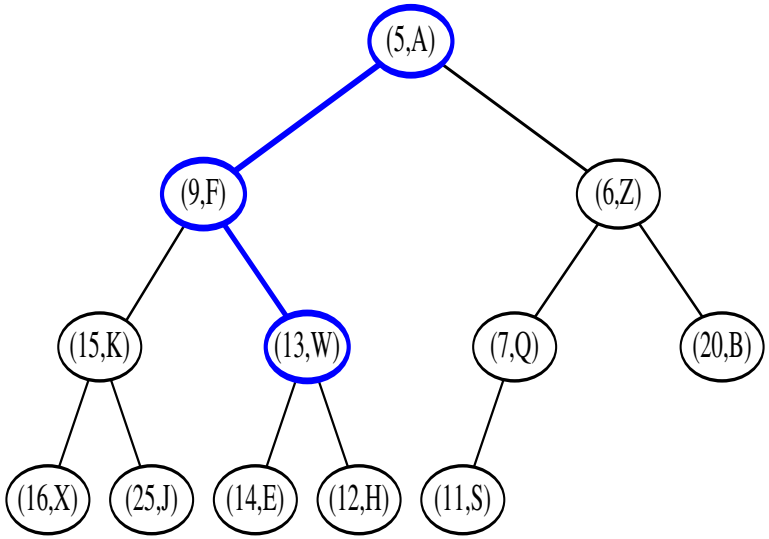
Removing the Item with Min key



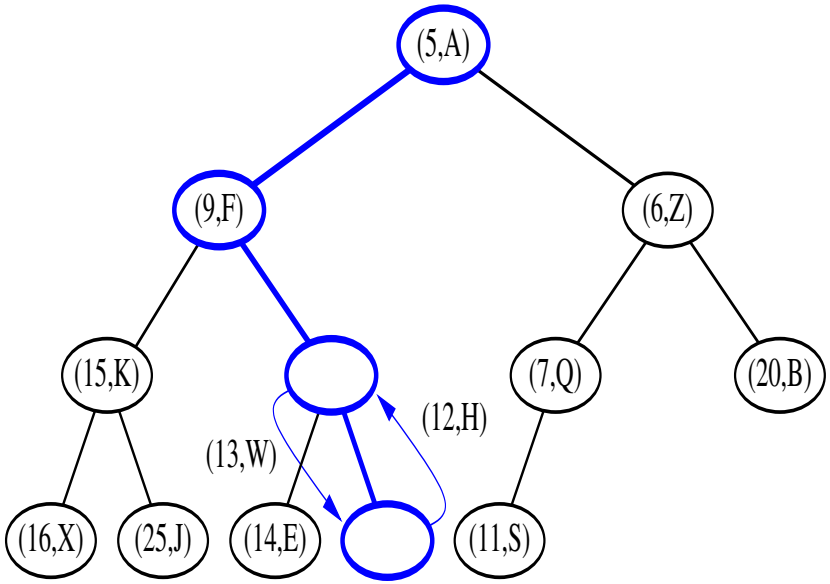
Removing the Item with Min key



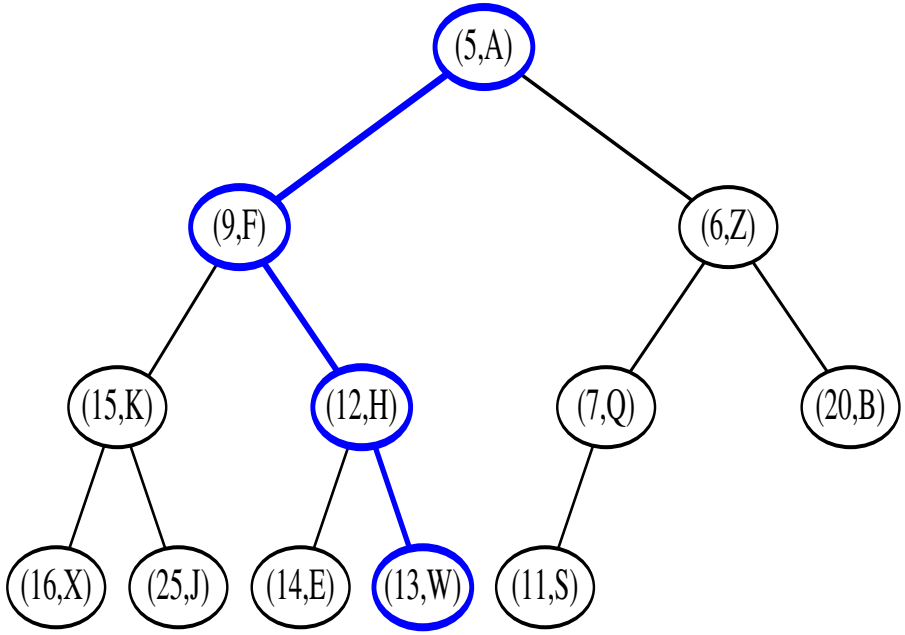
Removing the Item with Min key



Removing the Item with Min key



Removing the Item with Min key



Step 1 : downheap bubbling (percolate down) →used in remove_min

Complete the code for **downheap** bubbling

Complete downheap bubbling : non public method will be used during remove_min

```
def _downheap(self, j):

    if current object has left at j :
      then local variable left gets  current object's left j
      local variable small gets  left            # although right may be smaller
      if current object has right at j:
        then local variable right  gets current object right j
        if current object data at right index is less than  < current data left indext:
          small gets right
      if current data at small_child index is less than < current data j th index:
        then swap j with  small
        current object downheap(small_child)    # recur at position of small child
```

Completed version
downheap bubbling : non public method
will be used during remove_min

```
def _downheap(self, j):
    if self._has_left(j):
        left = self._left(j)
        small_child = left  # although right may be smaller
        if self._has_right(j):
            right = self._right(j)
            if self._data[right] < self._data[left]:
                small_child = right
        if self._data[small_child] < self._data[j]:
            self._swap(j, small_child)
            self._downheap(small_child)  # recur at position of small child
```

Step 2 : complete remove_min

Complete remove_min

```
def remove_min(self):

    """Remove and return (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if current object is Empty:
        raise Empty('Priority queue is empty.')
    current object swap(0, len(self._data) - 1)  # put minimum item at the
end
   # and remove it from the list;
   # then fix new root
    return (item._key, item._value)
```

remove_min

```
def remove_min(self):
    """Remove and return (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if self.is_empty():
        raise Empty('Priority queue is empty.')
    self._swap(0, len(self._data) - 1)  # put minimum item at the end
    item = self._data.pop()  # and remove it from the list;
    self._downheap(0)  # then fix new root
    return (item._key, item._value)
```

Test remove_min method !

Test remove_min method !

```
pqueu = HeapPriorityQueue()
pqueu.add(5, 'A')
pqueu.add(4, 'B')
pqueu.add(3, 'B')
pqueu.add(9, 'C')
pqueu.add(3, 'B')
pqueu.add(1, 'A')

print(str(pqueu.min()))
pqueu.remove_min()
print(str(pqueu.min()))
```

Complete min method

Complete min method

```
def min(self):
    """Return but do not remove (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if current object is Empty:
        raise Empty('Priority queue is empty.')
```

Completed version min method

```
def min(self):
    """Return but do not remove (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if self.is_empty():
        raise Empty('Priority queue is empty.')
    item = self._data[0]
    return (item._key, item._value)
```

Analysis of a Heap-Based Priority Queue

- In short, each of the priority queue ADT methods can be performed in O(1) or in O(logn) time, where n is the number of entries at the time the method is executed. The analysis of the running time of the methods is based on the following:
- The heap T has n nodes, each storing a reference to a key-value pair.
- The height of heap T is O(log n), since T is complete
- The min operation runs in O(1) because the root of the tree contains such an element.
- Locating the last position of a heap, as required for add and remove min, can be performed in O(1) time for an array-based representation, or O(log n) time for a linked-tree representation.
- In the worst case, up-heap and down-heap bubbling perform a number of swaps equal to the height of T .

Analysis of a Heap-Based Priority Queue

Operation	Running Time
len(P), P.is_empty()	$\Theta(1)$
P.min()	$\Theta(1)$
P.add()	O(logn)*
P.remove_min()	O(logn)*

*amortized, if array-based