10
Priority Queues
Chapter 09

ODTÜ
METU

Priority Queues

The queue ADT is a collection of objects where items are added and dropped according to the first-in first-out (FIFO) principle.

However, there might be cases where an item can have priority higher than the other items.

- Loyalty customers
- Frequent fliers
- Kernel processes

Priority Queues

For these kind of problems we can use priority queues.

Similar to queues in the sense that any type of data item can be added to the queue. Elements in a queue has to have two parts:

- Value: The data/object item that will be stored in the element.
- Key: The value designating the priority of an element. The element with a lower key will have a higher priority.

Priority Queue ADT

P.add(k,v) Insert an item with a key-value pair into priori queue P.

P.min() Return a tuple (k,v), representing the key and value of an item in priority queue P with minimum key (but do not remove the item); an error occurs if the priority queue is empty.

P.remove_min() Remove an item with minimum key from priority queue P, and return a tuple, (k,v), representing the key and value of the removed item; an error occurs if the priority queue is empty.

P.is empty() Return True if priority queue P does not contain any items.

len(P) Return the number of items in priority queue P.

Priority Queue ADT Example

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min()	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min()	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min()	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min()	(7,D)	{(9,C)}
P.remove_min()	(9,C)	{ }
P.is_empty()	True	{ }
P.remove_min()	"error"	{ }

Implementation with Unsorted List

Internally maintain a list.

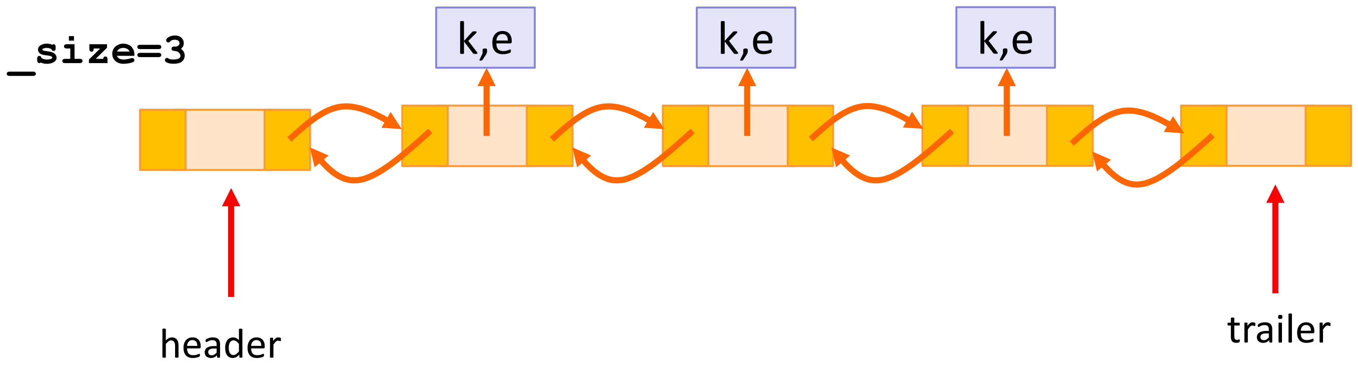
Perform insert operations at the end.

For the min and remove_min operations, find the item with the minimum key and return and drop depending on the action.

For the len and is_empty operations, we can use length of list.

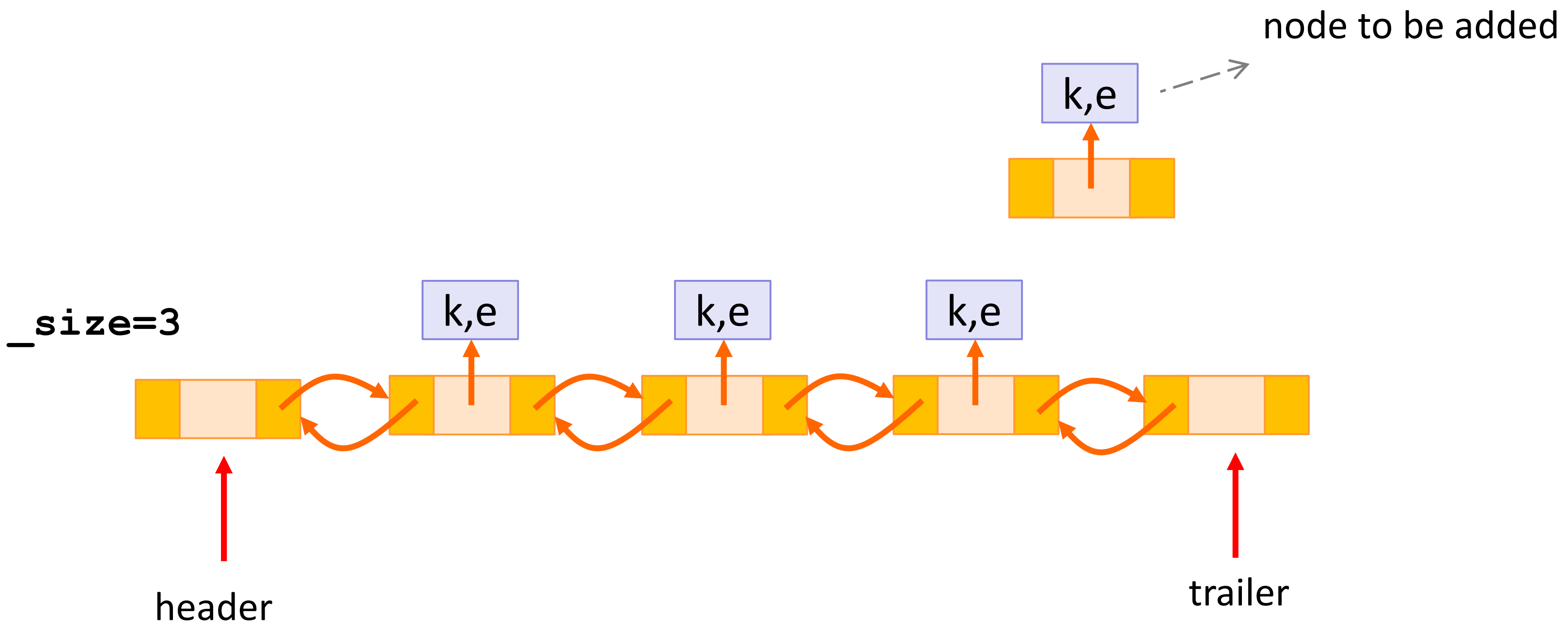
Implementation with Unsorted List

len and **is_empty** can simply be done by checking the `_size` data member. Hence they both are O(1).



Implementation with Unsorted List

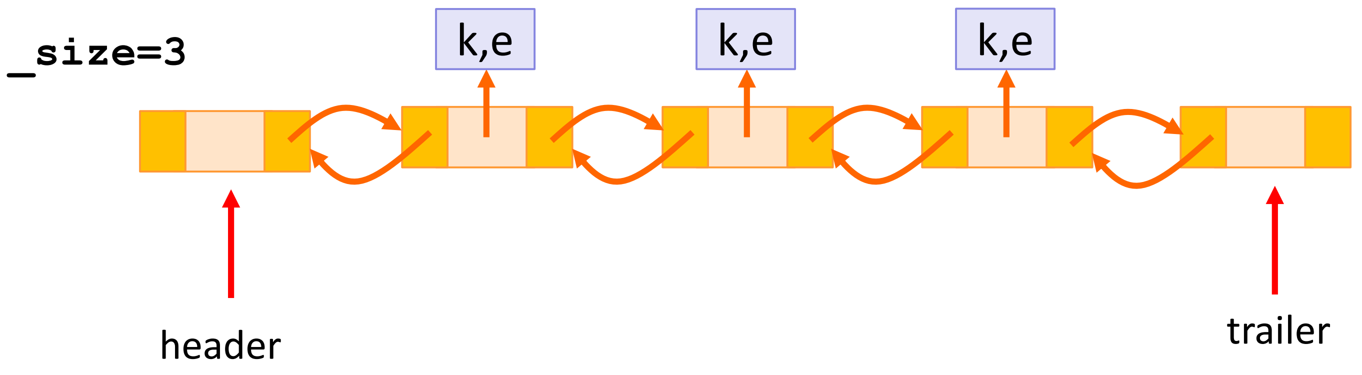
add operation is performed at the end of the doubly linked list. This operation is clearly O(1).



Implementation with Unsorted List

min operation tries to find the item with the minimum key. This is performed by simply checking each item's key starting from the header going onward up until to the trailer.

min is O(n).

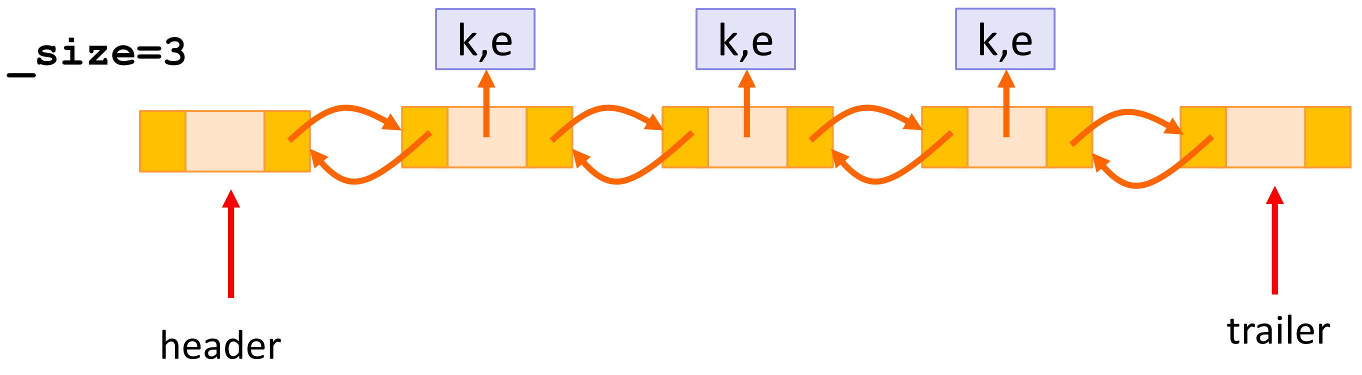


Implementation with Unsorted List

remove_min

With the **min** operation we find the item with the minimum key. When the item is found, it is removed.

Search is O(n), removing is O(1), hence **remove_min** is O(n).



Implementation with Sorted List

Internally maintain an ordered list such that the item with smallest key is always at the beginning of the list (i.e., `header.next`)

For the **min** and **remove_min** operations, get the item with the minimum key (which is simply, `header.next`) and return and drop depending on the action.

add operation has to make sure that the list is ordered after insertion of the new iten.

Due to this reason, it makes a search starting from the end (`trailer.before`) backward to the beginning to find the position whose key is smaller than the new item's key. After locating the position, insertion is made.

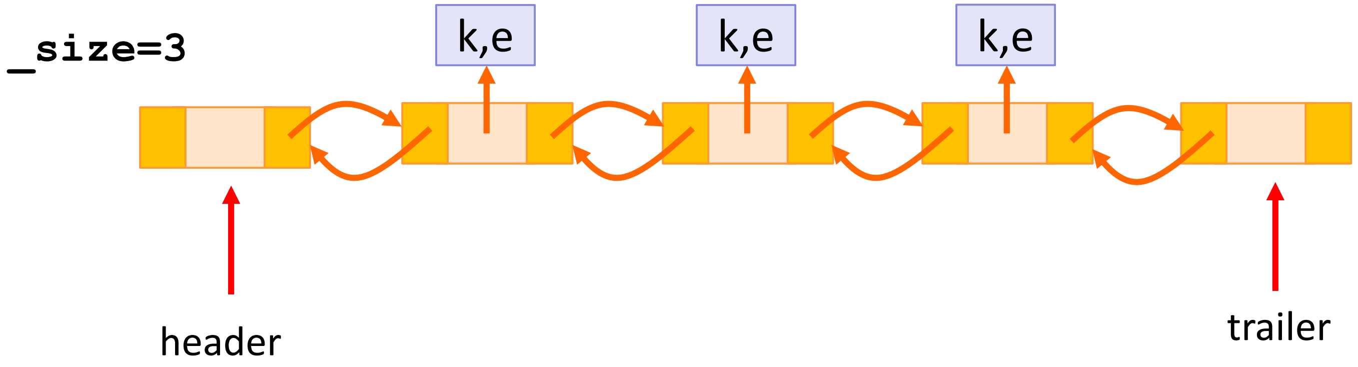
Implementation with Unsorted List

min and **remove_min**

With the **min** operation we find the item with the minimum key.

In an ordered list, the item with the smallest key is always at the beginning (header.next).

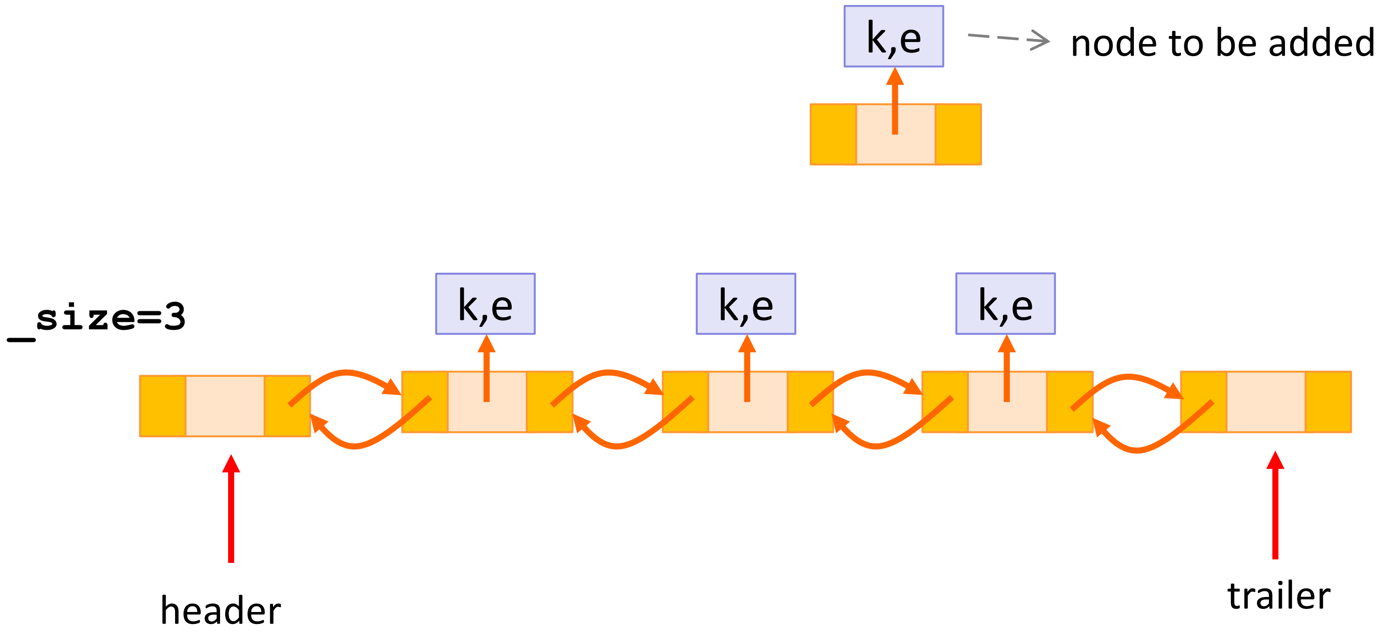
min is O(1), and **remove_min** is O(1).



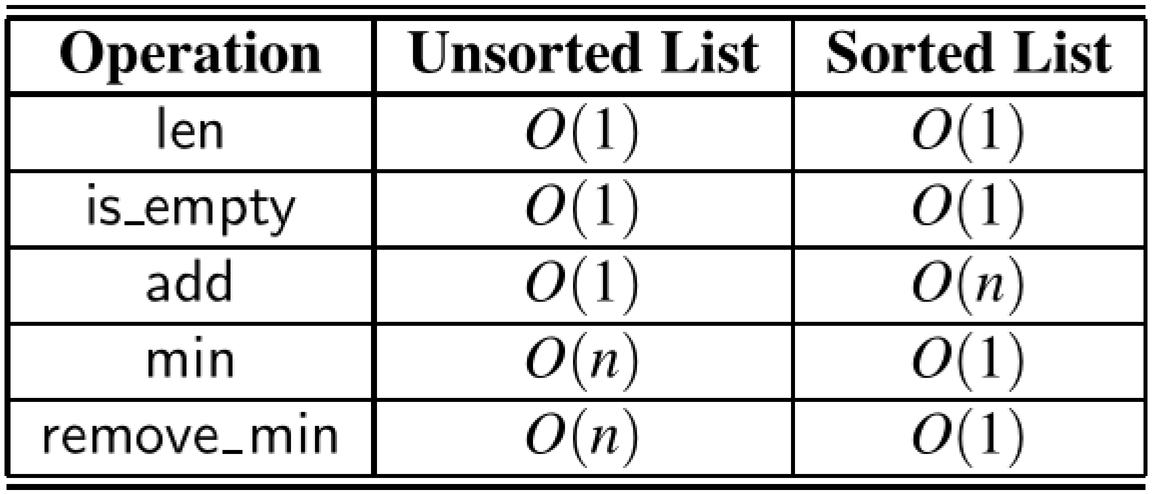
Implementation with Unsorted List

add operation starts search from the trailer and continues up to the header to find the item whose key is smaller the new item's key. And then insert operation is performed.

Searching is O(n), insertion is O(1), hence **add** operation is O(n).



Comparison of Sorted and Unsorted



Operation	Unsorted List	Sorted List
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

A trade-off between add and remove operations exists.

There is also another approach where both add and remove operations are O(logn).

Heaps

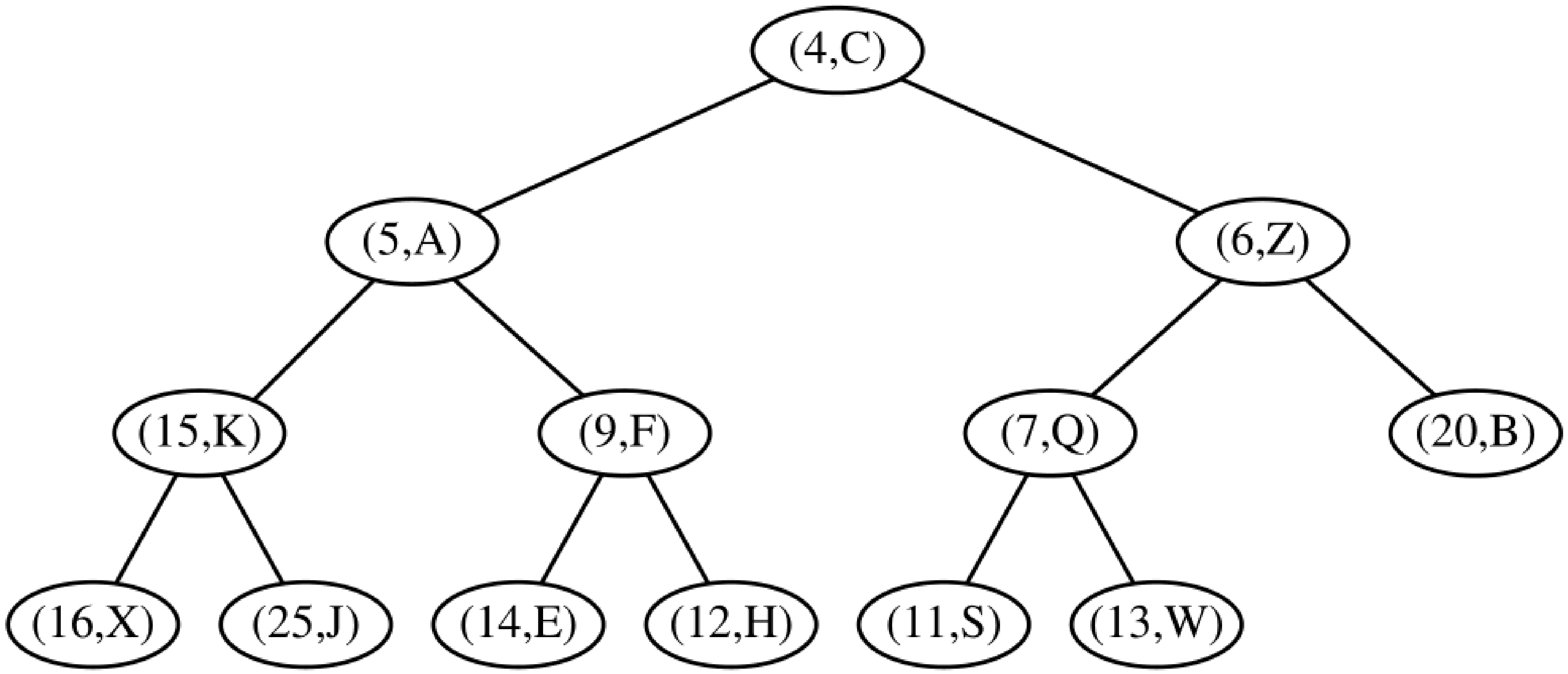
Allows insertions and deletions to be performed in O(logn) time.

A heap is a binary tree satisfying the following properties:

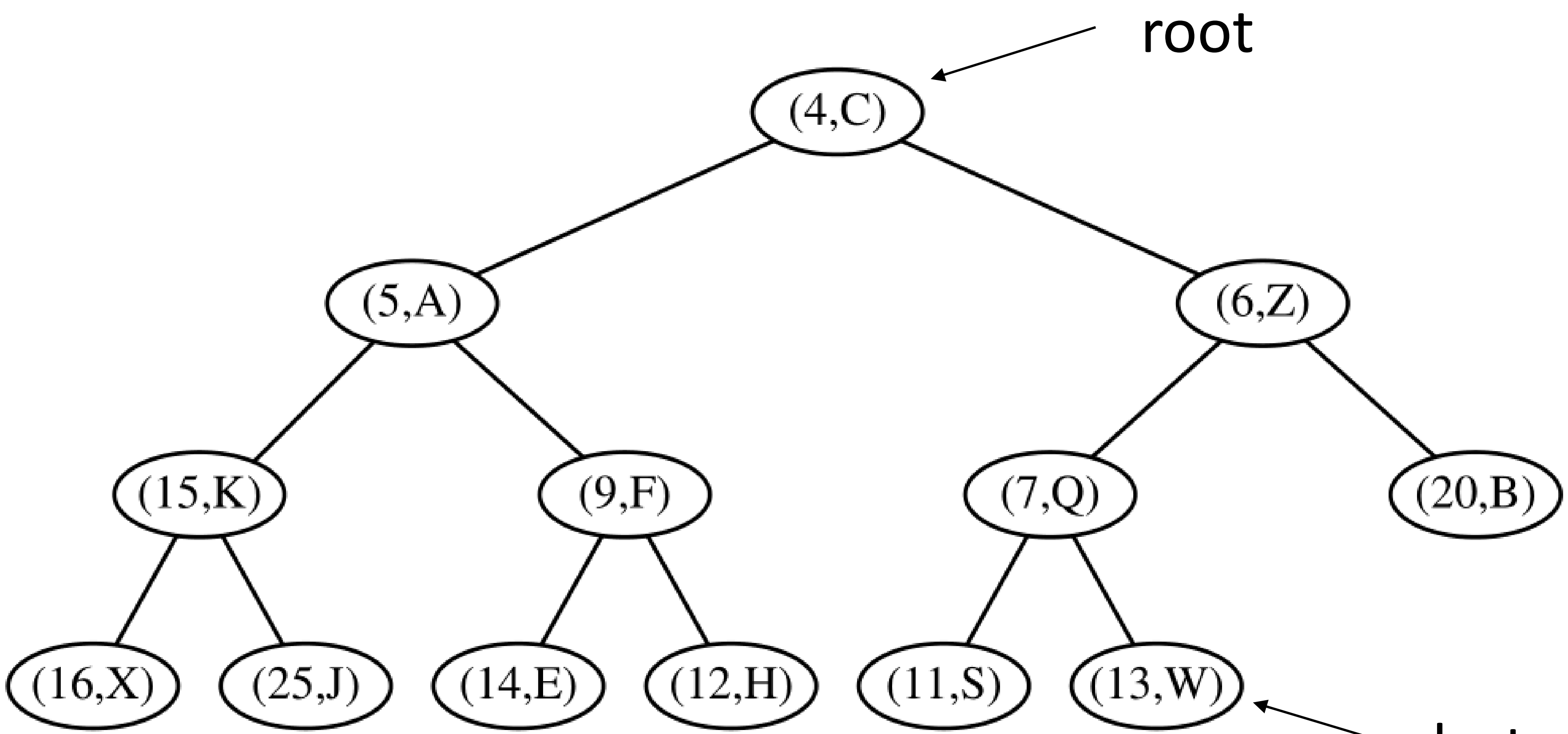
Heap-order Property: In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p’s parent.

Complete Binary Tree Property: A heap T with heigh h is a complete binary tree s.t. at each level 0, 1, ..., h-1 there are maximum number of nodes and at level h all the nodes are in the leftmost possible position.

Heap



Heap



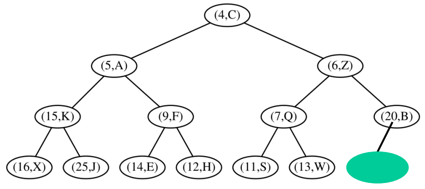
A heap T storing n entries has height h = ⌊logn⌋.

Heap Operations

Adding Item

If heap is empty, the item is added as the root node, and the operation is done.

Otherwise, the item is added to the rightmost available position at the bottom level of the tree.



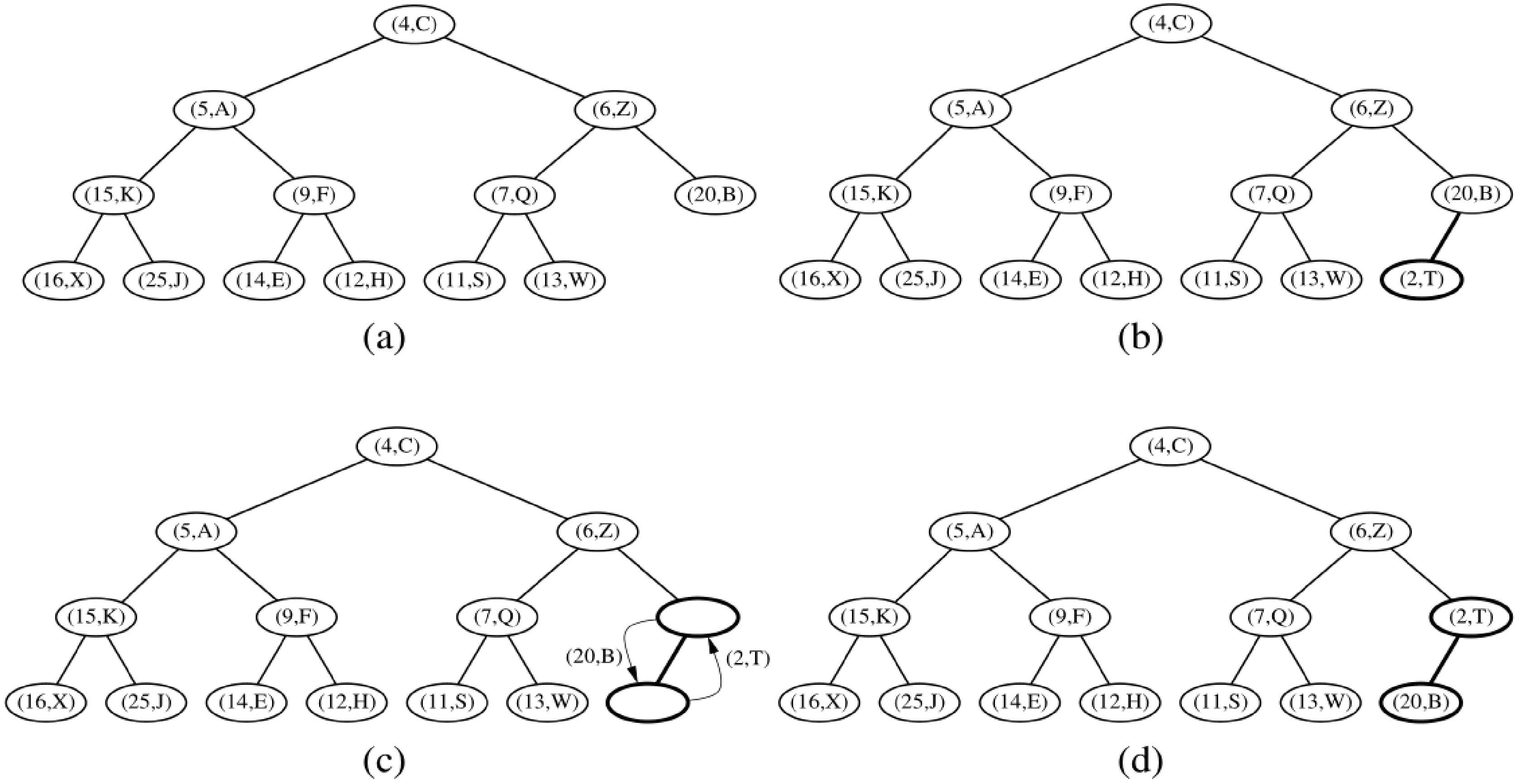
At this state,

Complete Binary Tree Property is maintained.

In order to maintain the heap order property, the up-heap bubbling is performed.

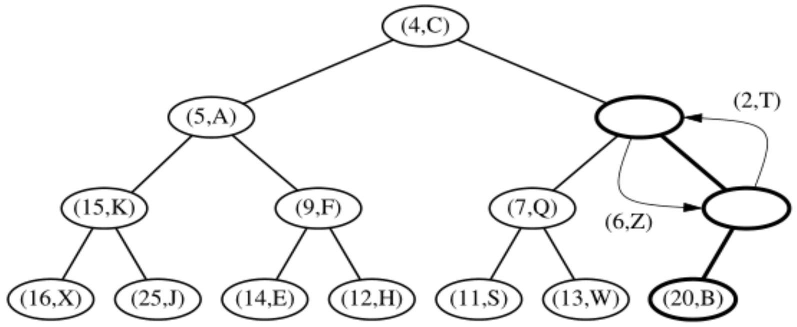
Heap Operations

Up-Heap Bubbling

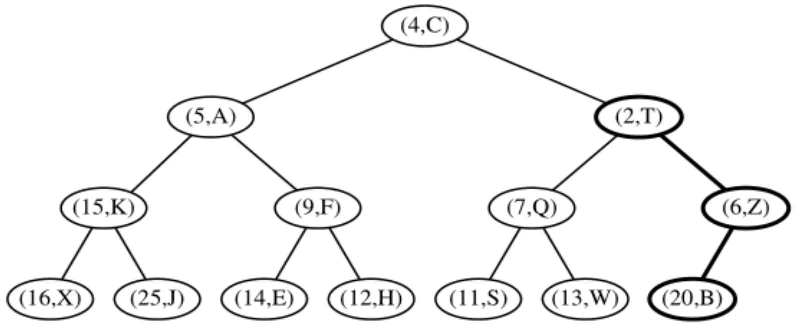


Heap Operations

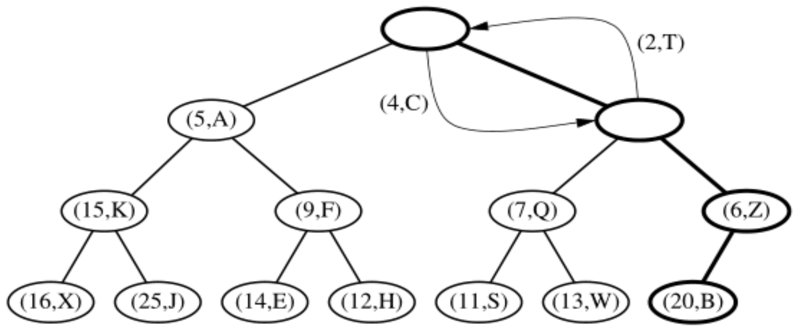
Up-Heap Bubbling



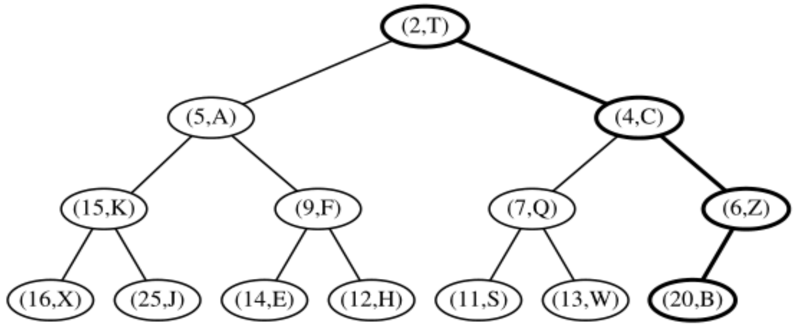
(e)



(f)



(g)



(h)

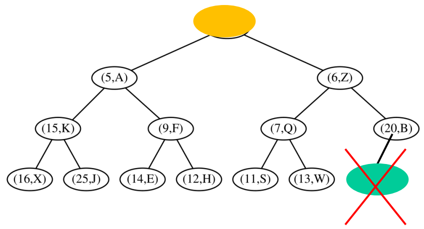
Heap Operations

Removing Item

If heap is empty, an error should be thrown.

If there is only one item, it should simply be removed, leaving the heap empty.

Otherwise, the item at the root is replacced with the item at the last position, and the last item deleted.



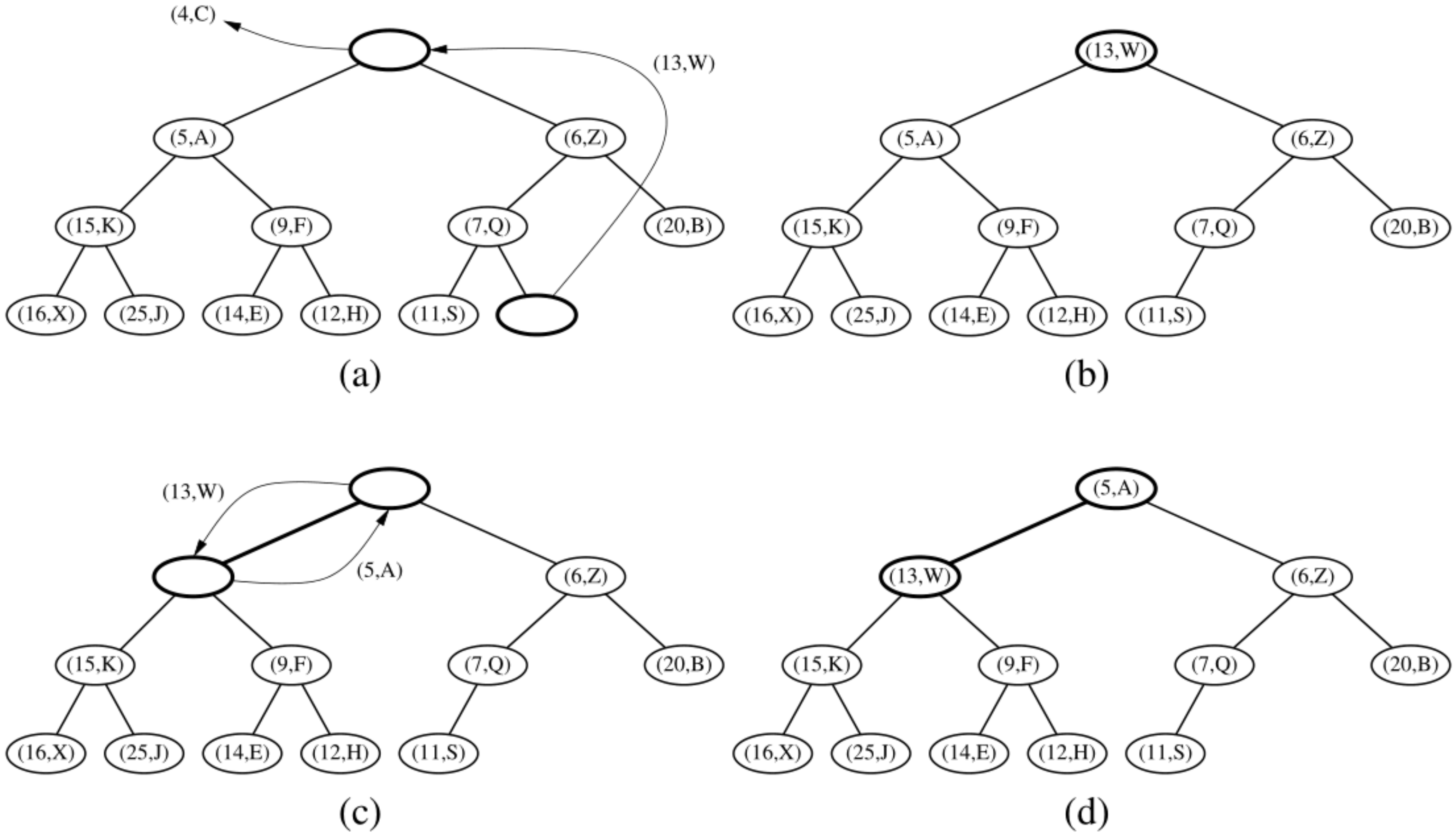
At this state,

Complete Binary Tree Property is maintained.

In order to maintain the heap order property, the down-heap bubbling is performed.

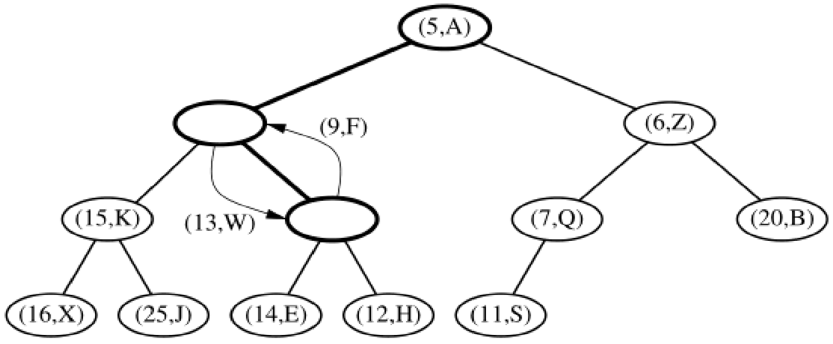
Heap Operations

Down-Heap Bubbling

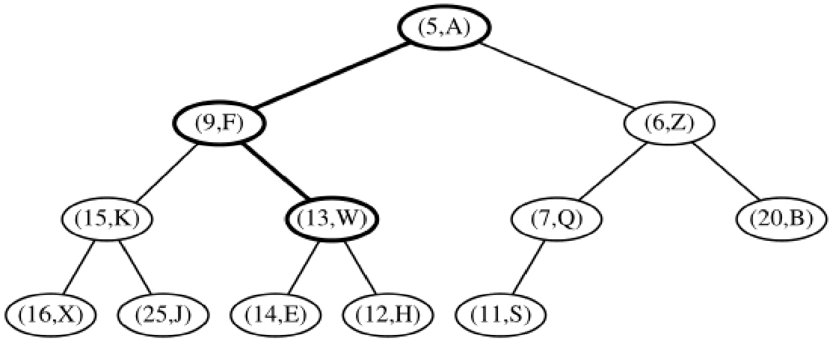


Heap Operations

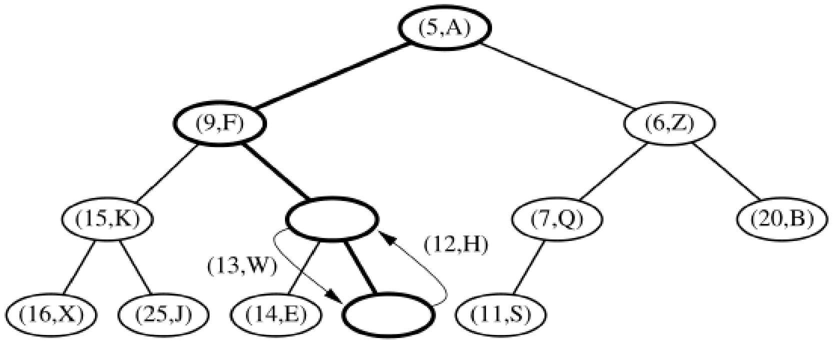
Down-Heap Bubbling



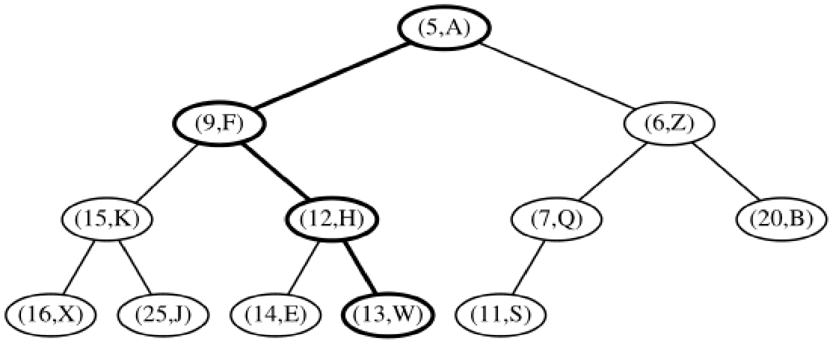
(e)



(f)



(g)

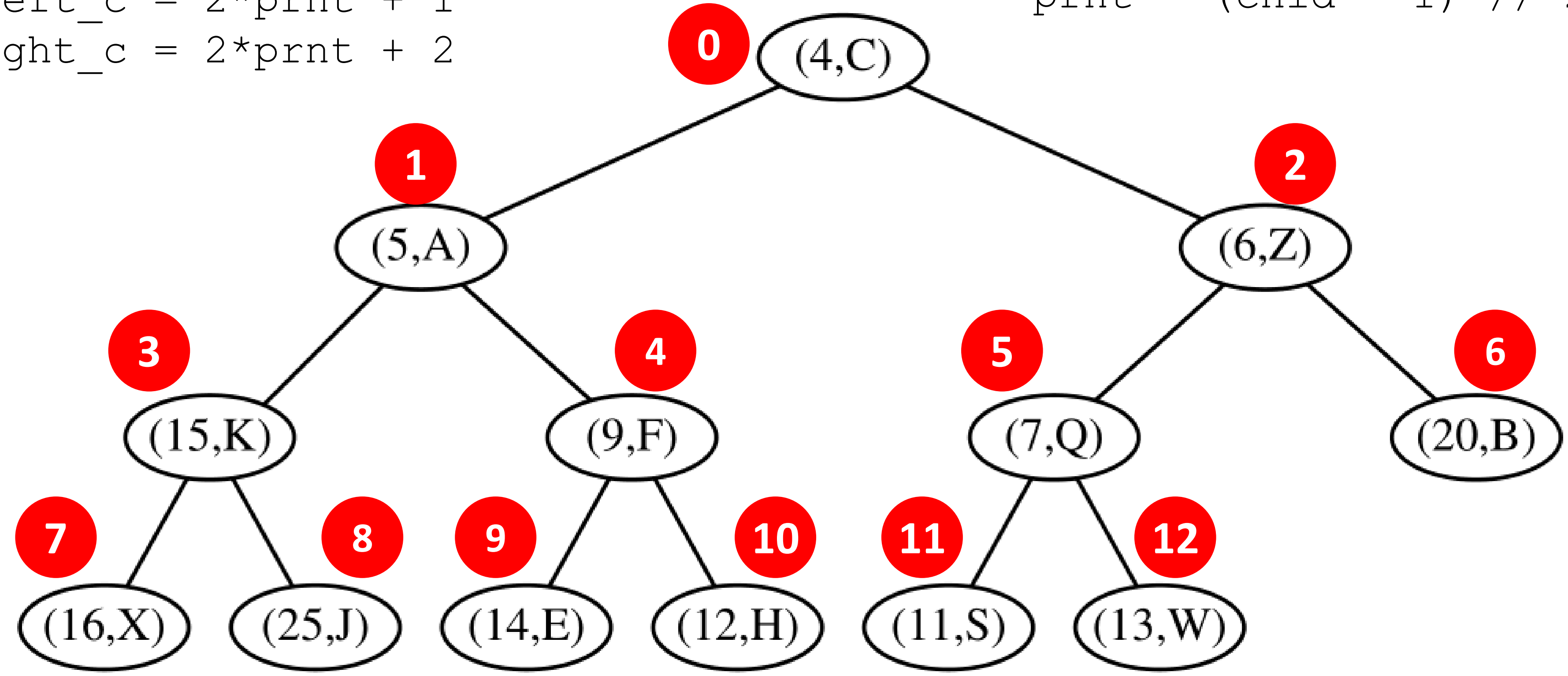


(h)

Heap Implementation

```
left_c = 2*prnt + 1
rght_c = 2*prnt + 2
```

```
prnt = (chld - 1) // 2
```



0	1	2	3	4	5	6	7	8	9	10	11	12
(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)

Heap Implementation

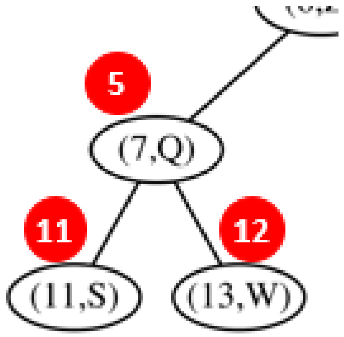
```
class PriorityQueueBase:
  """Abstract base class for a priority queue."""

  class _Item:
    """Lightweight composite to store priority queue items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __lt__(self, other):
      return self._key < other._key          # compare items based on their keys

  def is_empty(self):                        # concrete method assuming abstract len
    """Return True if the priority queue is empty."""
    return len(self) == 0
```

Heap Implementation

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

  #------------------------------ 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)
```

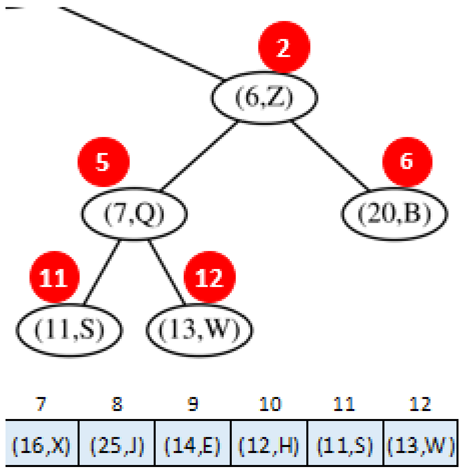
Heap Implementation

```
def _parent(self, j):
  return (j−1) // 2

def _left(self, j):
  return 2∗j + 1

def _right(self, j):
  return 2∗j + 2
```



Heap Implementation

```
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?
```

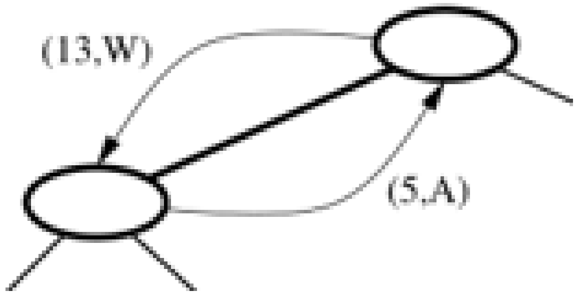


7	8	9	10	11	12
(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)

The index of left child of 6 would be 13, which is out of bounds of the internal array.

Heap Implementation

```
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]
```



Only the values are changing. No other link operations whatsoever take place.

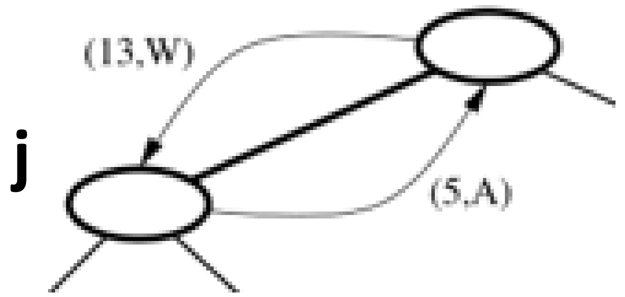
0	1	2	3
(13,W)	**(5,A)**	(6,Z)	(15,K)

Heap Implementation

```
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
```

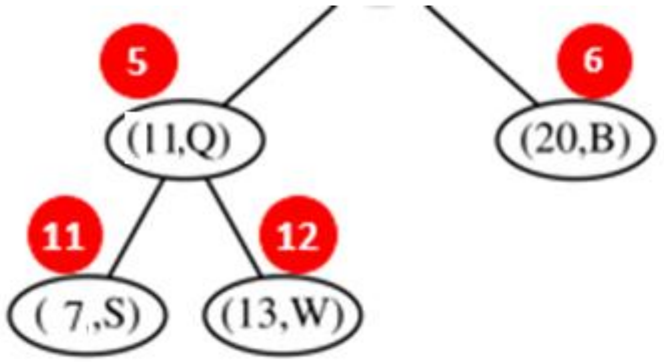
If the item is not root.

parent(j)



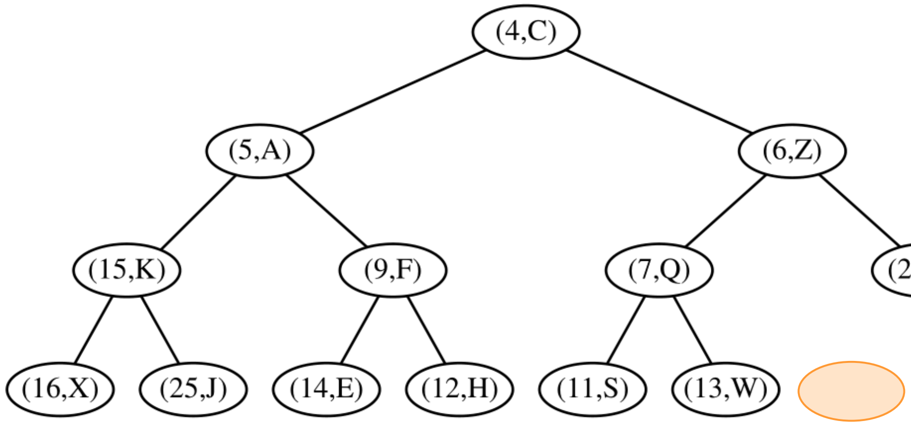
Heap Implementation

```
def _downheap(self, j):
  if self._has_left(j):
    left = self._left(j)
    small_child = left
    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)
```



Heap Implementation

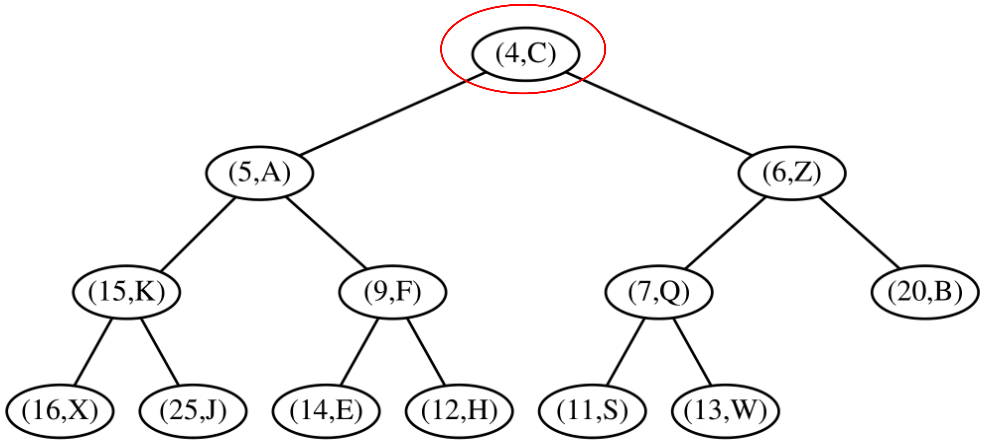
```
def add(self, key, value):
  """Add a key-value pair to the priority queue."""
  self._data.append(self._Item(key, value))
  self._upheap(len(self._data) − 1)          # uphea
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13
(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)	X

Heap Implementation

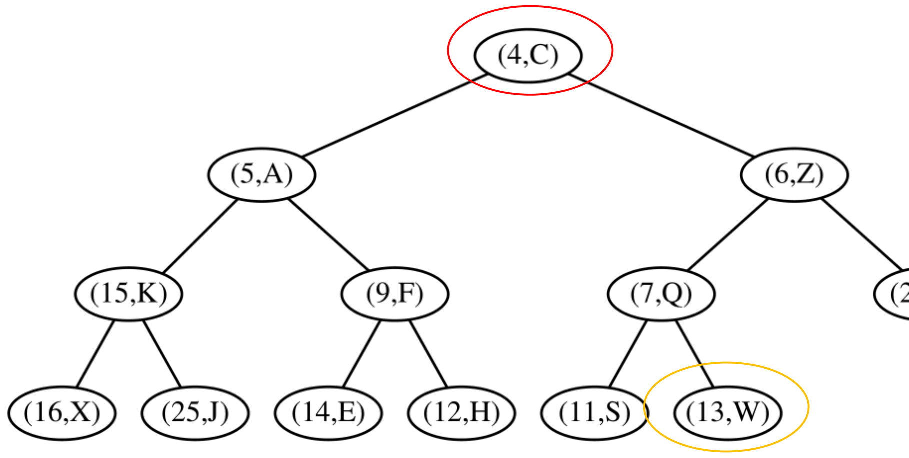
```
def min(self):
  """Return but do not remove (k,v) tuple with min

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



0	1	2	3	4	5	6	7	8	9	10	11	12
(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)

Heap Implementation

```
def remove_min(self):
  if self.is_empty( ):
    raise Empty('Priority queue is empty.')
  self._swap(0, len(self._data) − 1)
  item = self._data.pop( )
  self._downheap(0)
  return (item._key, item._value)
```



0	1	2	3	4	5	6	7	8	9	10	11	12
(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)

Analysis of Heap Operations

Assumption

Two keys can be compared in O(1) time. (Think of a case how that can be higher than O(1) as a HW!)

T is implemented with array- or linked-based tree.

T has n nodes.

Complexities

Height of T is O(logn) because it is a complete binary tree.

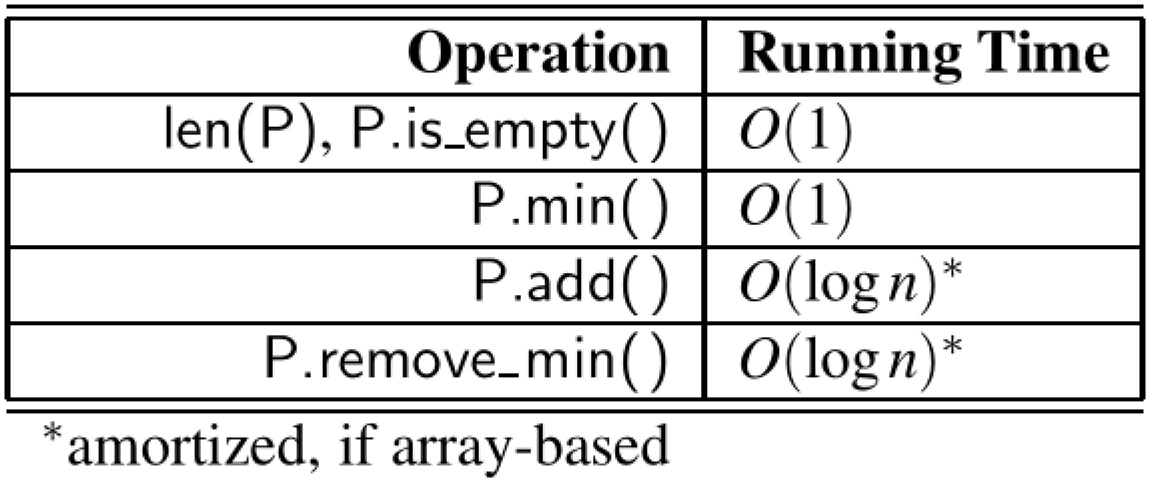
min operation runs in O(1) since it simply returns the root item.

Finding the last position is O(1) for array-based, and O(logn) for tree-based approach (How?).

Analysis of Heap Operations

Complexities

Up-heap and down-heap operations can be performed in O(logn).



Operation	Running Time
len(P), P.is_empty()	$O(1)$
P.min()	$O(1)$
P.add()	$O(\log n)^*$
P.remove_min()	$O(\log n)^*$

*amortized, if array-based

Sorting with Priority Queues

By making use of priority queues, we can easily sort items in a given list. First, insert each item to the priority queue, then apply remove_min operation.

```
def pq_sort(C):
  """Sort a collection of elements stored in a positional list."""
  n = len(C)
  P = PriorityQueue()
  for j in range(n):
    element = C.delete(C.first())
    P.add(element, element)          # use element as key and value
  for j in range(n):
    (k,v) = P.remove_min()
    C.add_last(v)                    # store smallest remaining element in C
```

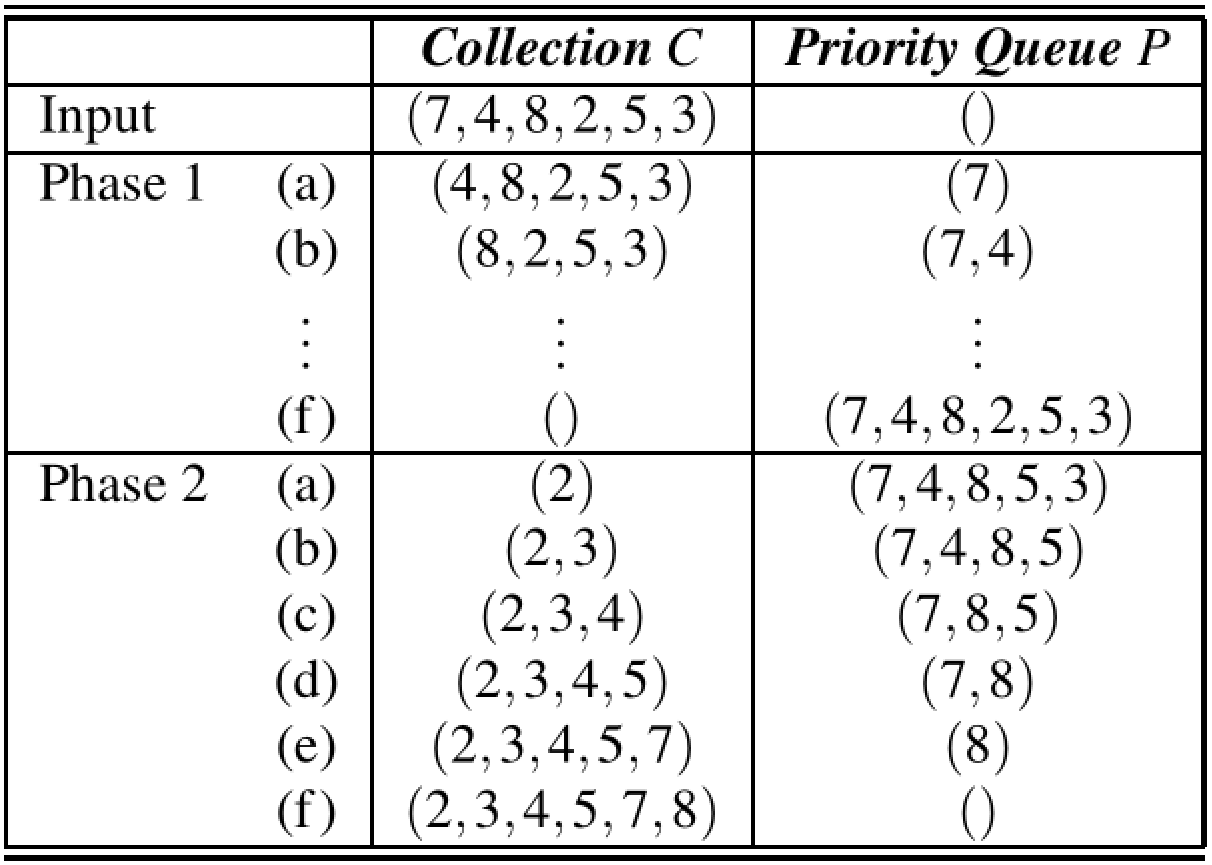
Sorting with Priority Queues

If PriorityQueue is implemented with unsorted list, the code below does **selection sort**.

But instead, if it uses a sorted list, then what the code does would be **insertion sort**.

```
def pq_sort(C):
  """Sort a collection of elements stored in a positional list."""
  n = len(C)
  P = PriorityQueue()
  for j in range(n):
    element = C.delete(C.first())
    P.add(element, element)            # use element as key and value
  for j in range(n):
    (k,v) = P.remove_min()
    C.add_last(v)                      # store smallest remaining element in C
```

Selection Sort



		Collection C	*Priority Queue P*
Input		(7,4,8,2,5,3)	()
Phase 1	(a)	(4,8,2,5,3)	(7)
	(b)	(8,2,5,3)	(7,4)
	⋮	⋮	⋮
	(f)	()	(7,4,8,2,5,3)
Phase 2	(a)	(2)	(7,4,8,5,3)
	(b)	(2,3)	(7,4,8,5)
	(c)	(2,3,4)	(7,8,5)
	(d)	(2,3,4,5)	(7,8)
	(e)	(2,3,4,5,7)	(8)
	(f)	(2,3,4,5,7,8)	()

Insert each item to an unsorted list

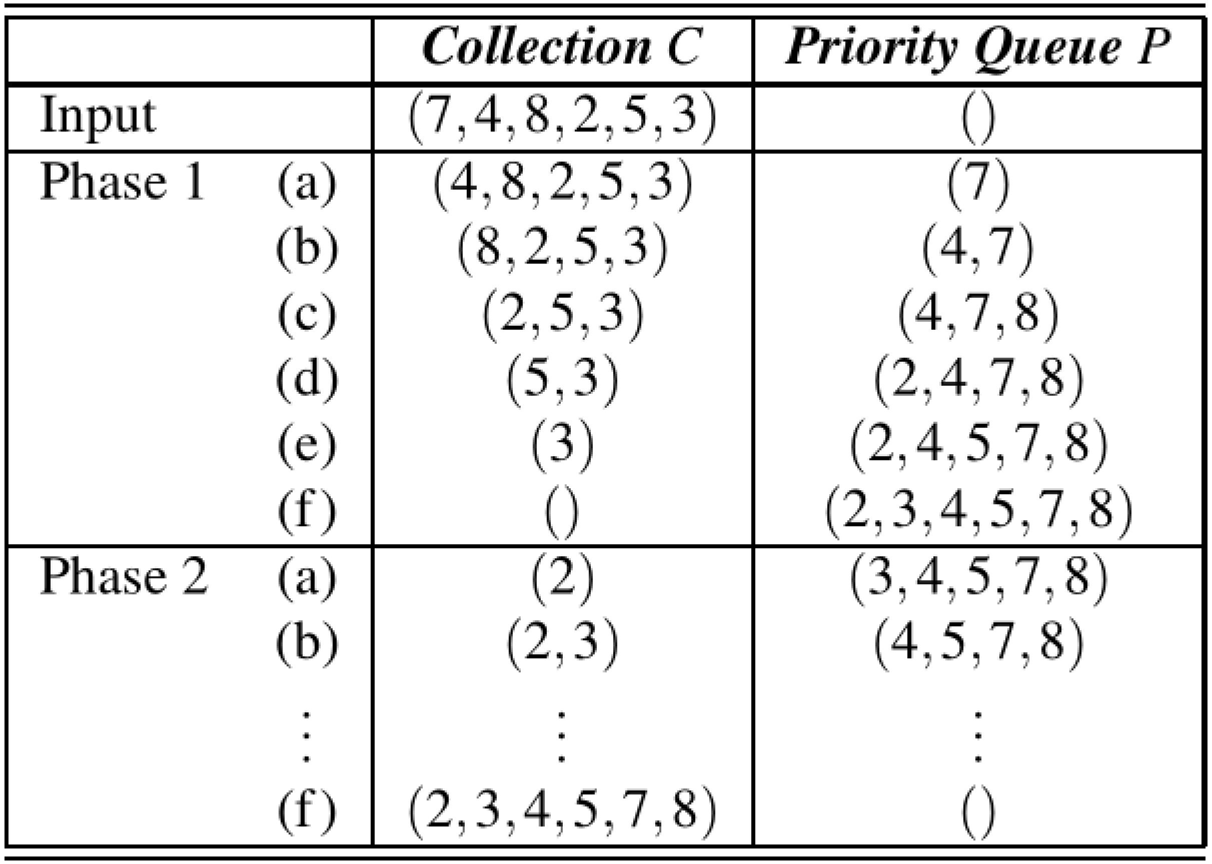
n * O(1)

"Select" and remove the minimum item from the list $O(n^2)$.

Worst-case: n(n-1)/2

Selection sort is $O(n^2)$.

Insertion Sort



		Collection C	*Priority Queue P*
Input		(7,4,8,2,5,3)	()
Phase 1	(a)	(4,8,2,5,3)	(7)
	(b)	(8,2,5,3)	(4,7)
	(c)	(2,5,3)	(4,7,8)
	(d)	(5,3)	(2,4,7,8)
	(e)	(3)	(2,4,5,7,8)
	(f)	()	(2,3,4,5,7,8)
Phase 2	(a)	(2)	(3,4,5,7,8)
	(b)	(2,3)	(4,5,7,8)
	⋮	⋮	⋮
	(f)	(2,3,4,5,7,8)	()

Insert each item to a sorted list requires n(n-1)/2 operations in the worst-case so this part is O(n^2).

Remove the minimum item from the list n * O(1) = O(n).

Selection sort is O(n^2).

Heap Sort

Instead of unsorted or sorted list, if we use a heap as the underlying data structure for priority queue ordering:

In the <u>insertion part</u>, we would do n add operations each of which would cost O(logn), in the worst case.

In the <u>removing part</u>, we would do n remove_min operations each of which, again, would cost O(logn) in the worst case.

So in general, heap sort would work in O(nlogn):

n * O(logn) + n * O(logn).