Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Heaps and Priority Queues

""

CS2013: Programming with Data Structures

Heaps

Heap

- ***heap***: a binary tree with two additional properties:
  - ***Heap-Order Property***
  - ***Complete Binary Tree Property***

- NOTE: A heap is a binary tree, NOT a binary search tree. Therefore the left less, right greater property of a binary search tree does not apply.

- NOTE: A heap data structure has nothing to do with the memory heap used to store objects in Java and other such languages.

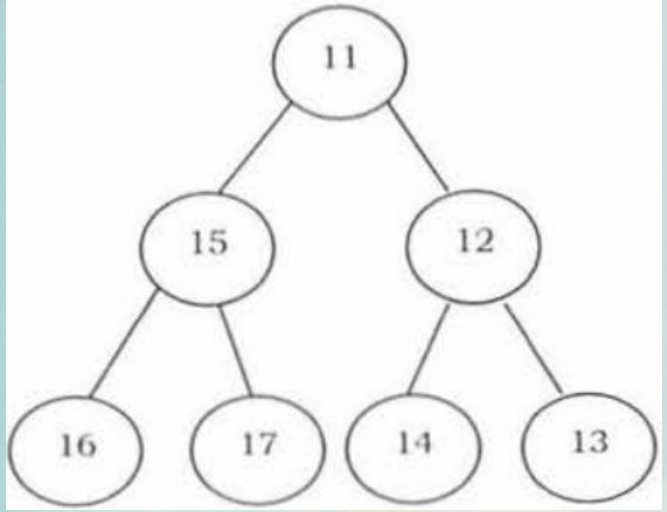
Heap-Order Property

- ***heap-order property***:
  - min heap: the key of a node must be less than or equal to the key of its children.
  - max heap: the key of a node must be greater than or equal to the key of its children.

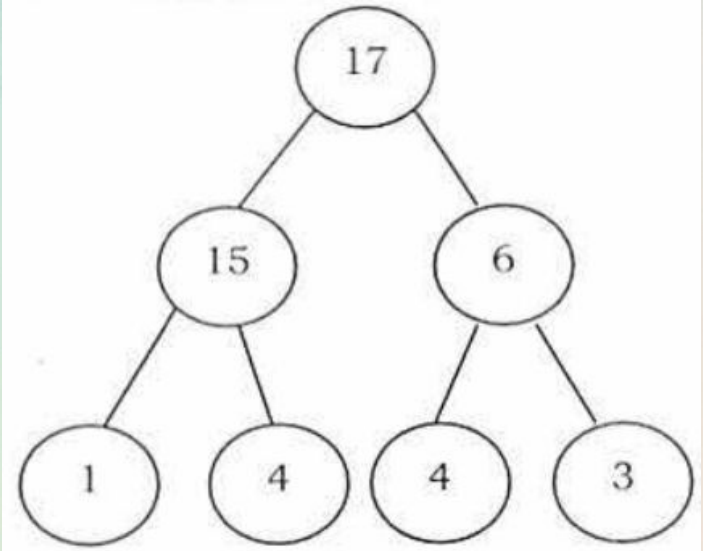
- the keys encountered on a path from the root to a leaf are in increasing order (min heap), or decreasing order (max heap).
  - the minimum or maximum value in the heap will always be the root of the tree.

Min vs Max Heaps

- Depending on which of the two heap properties a heap utilizes a heap can be a min heap or a max heap:
  - ***min heap***: value of a node must be <= the values of its children. minimum value is the root.



  - ***max heap***: the value of a node must be >= the values of its children. maximum value is the root.

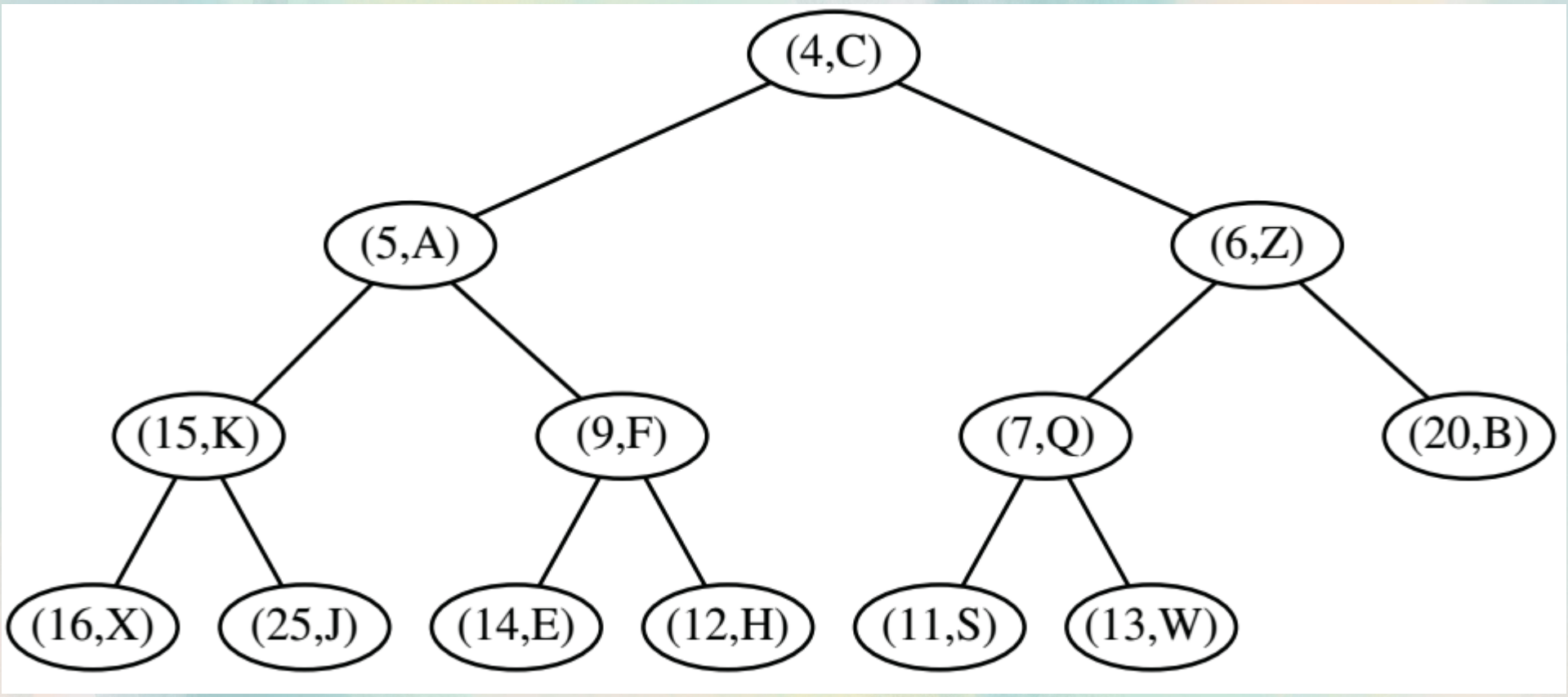


Complete Binary Tree Property

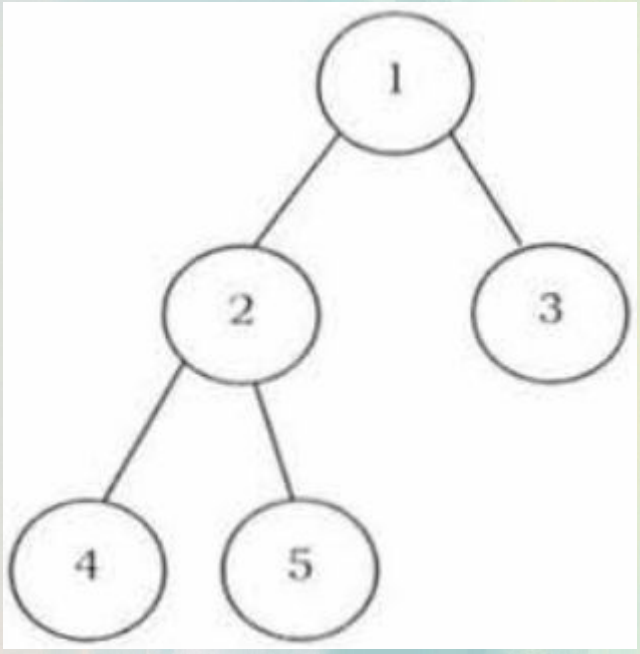
- ***complete binary tree property***:
  - a heap must always form a complete binary tree
  - recall: a complete binary tree is one where all levels are filled, except maybe the last level, but all nodes in the last level are as leftmost as possible.

- This property helps to enforce that the heap has the minimum height possible by keeping the heap balanced.
  - see last week's lecture for the need for balancing.

Heap Examples

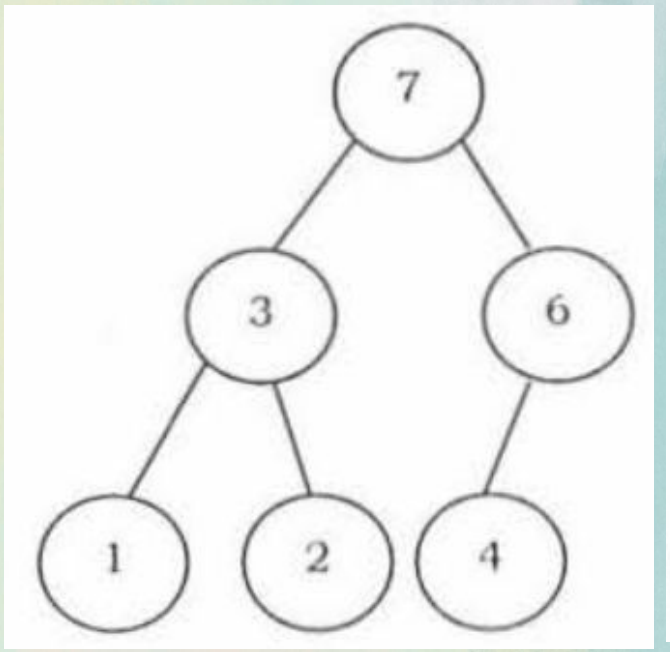


Heap Examples



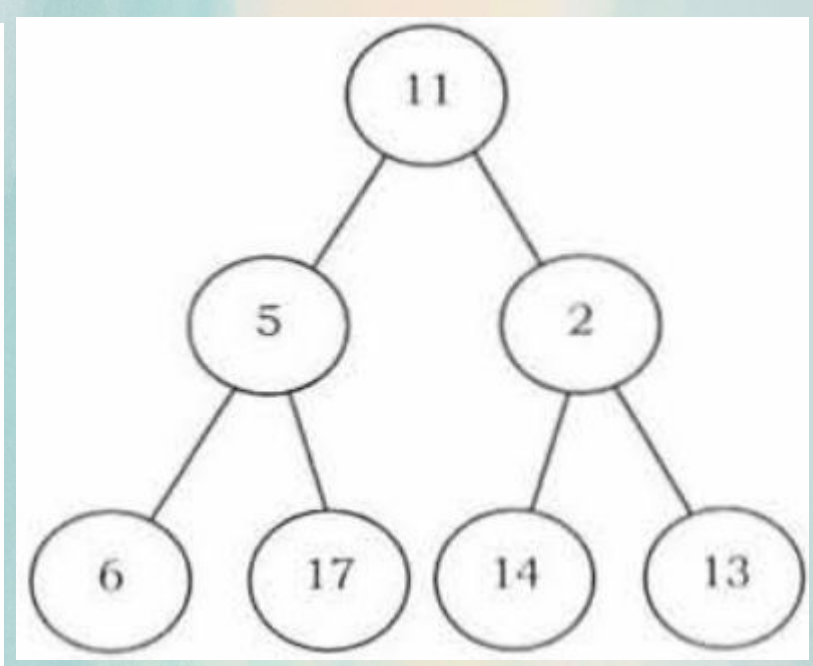
this **is** a heap:

- all children are greater than their parents.
- is a complete binary tree



this **is** a heap:

- all children are less than their parents.
- is complete binary tree



this **is not** a heap:

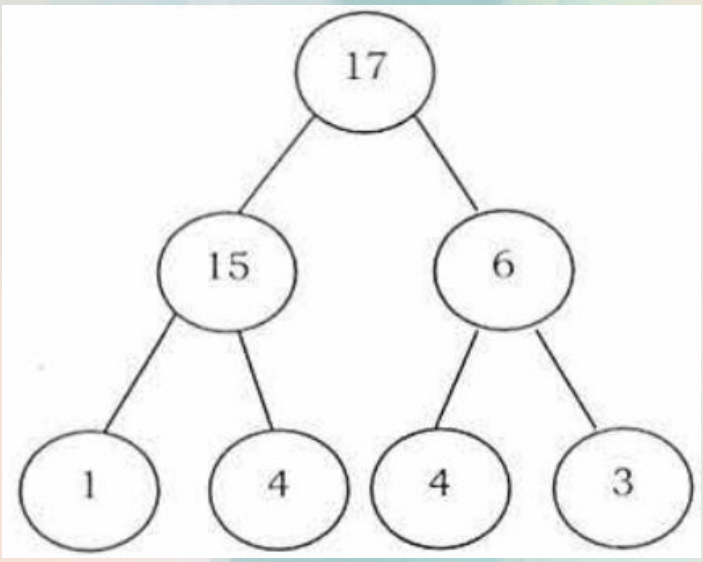
- not all children are less than or greater than the parent. (11 is greater than 5 and 2)
- is a complete binary tree

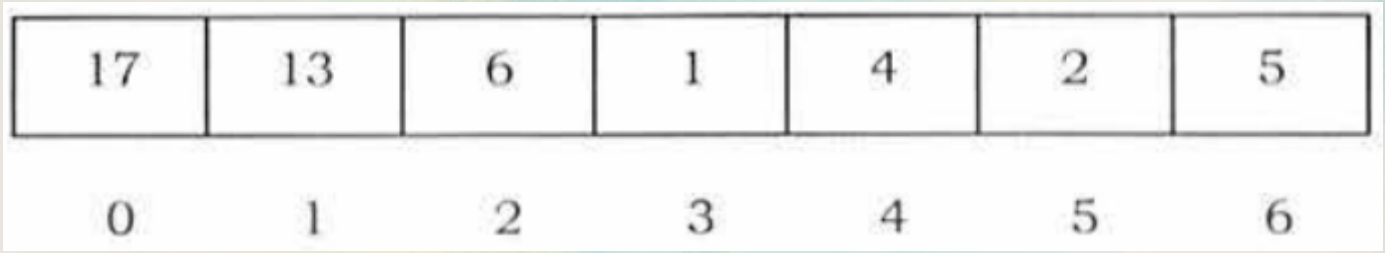
Heap Uses

- **Heapsort**: in place sorting with no quadratic worst-case scenarios.
- **Selection algorithms**: O(1) access to min or max element. Other operations (such as median or kth-element) can be done in sub-linear time.
- **Graph algorithms:** Prim's minimal-spanning-tree, Dijkstra's shortest-path algorithm
- **Priority Queue**
- **Order statistics:** efficiently find the kth smallest (or largest) element in an array.

Heap Implementations

- heaps can be represented using a linked tree structure.
- they can also be represented using an array, or list.
  - this is a good option, since the heap has to be a complete binary tree, there is no wasting of locations in the array.
  - The only downside would be possible resize operations.



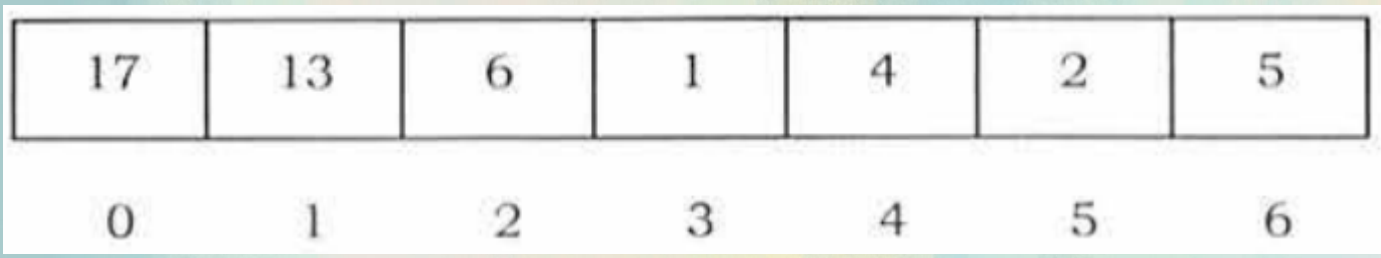


- For the following discussion, we will assume the heap is stored in an array or list.

Computing Heap Positions

- Computing the parent of a node:
  - for a node at i^{th} location its parent is at (i - 1) / 2

- Compute the children of a node:
  - for a node at i^{th} location its children are at:
    - left child: 2i + 1
    - right child: 2i + 2



- Examples:
  - parent of index 4 is: (4 – 1) / 2 = 1
  - left child of index 1 is: 2 * 1 + 1 = 3
  - right child of index 1 is: 2 * 1 + 2 = 4

Heapifying an Element

- After an element is inserted or deleted into / from the heap, the heap may no longer satisfy the **heap order property**.
  - We need to adjust the heap by **heapifying**

- ***heapifying****:* swapping nodes around the heap until its heap-order property is satisfied.

- heapify can happen from top to bottom (percolateDown) or from bottom to top (percolateUp).

percolateUp()

- NOTE 1: The input for this algorithm will vary, if your heap is represented as an array, the input is the array and index of the element you want to heapify. If the heap is a tree, then the input is the node you wish to heapify.
- NOTE 2: The algorithm will also vary depending on if you are heapifying a min heap vs a max heap.

```
percolateUpMin(current):
  if current is the root:
    return

  if current.key < parent.key:
    swap parent with current
    percolateUpMin(current)

percolateUpMax(current):
  if current is the root:
    return

  if current.key > parent.key:
    swap parent with current
    percolateUpMin(current)
```

percolateDown()

- NOTE 1: The input for this algorithm will be the root of the heap, no matter how it is represented.
- NOTE 2: The algorithm will also vary depending on if you are heapifying a min heap vs a max heap.

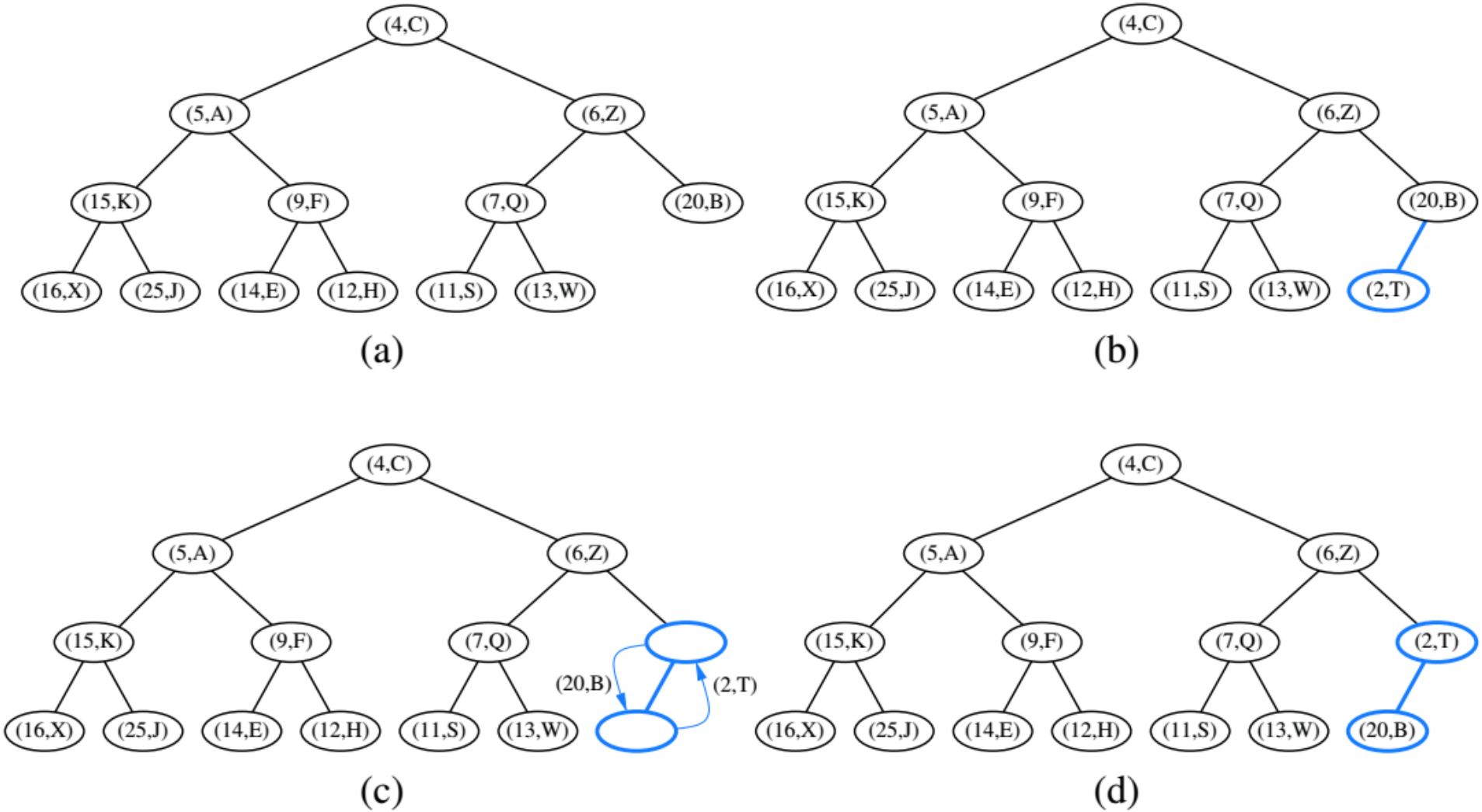
```
percolateDownMin(current):
  if current is a leaf:
    return

  child = min(left child, right child)

  if current.key > child.key:
    swap current with child
    percolateDownMin(current)

percolateDownMax(current):
  if current is a leaf:
    return

  child = max(left child, right child)

  if current.key < child.key:
    swap current with child
    percolateDownMax(current)
```

Heap Insert and Delete

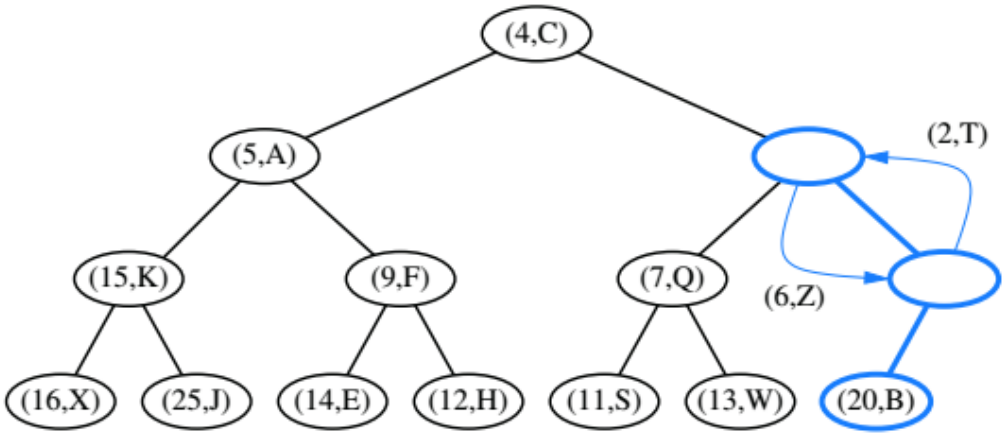
- Insertion into a heap is very easy:
  - add the new element as the next leaf in the last row of the tree (or the first open position in the end of the array)
  - percolateUp the new element.

- Deletion is also very easy:
  - With a heap, we always remove from the root because we are usually interested in the max or min element.
  - Swap the root with the rightmost leaf in the last row of the heap.
  - Delete the rightmost leaf that we just swapped into.
  - percolateDown the new root.

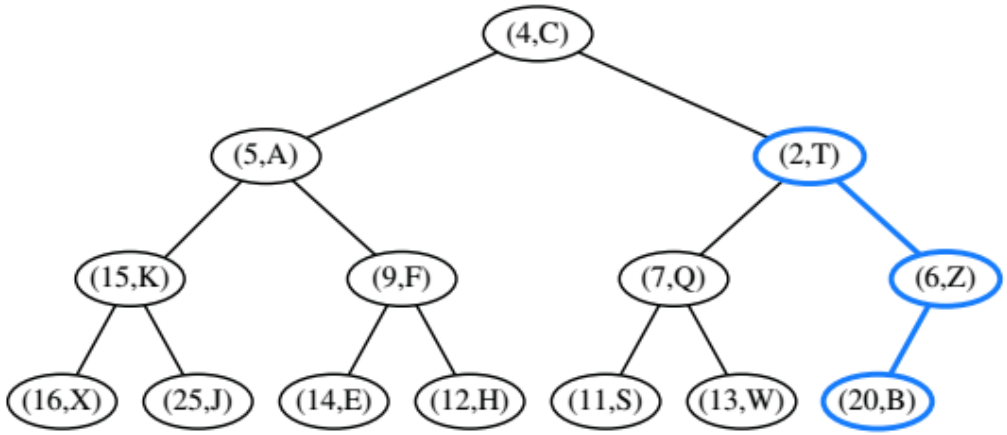
Heap Insert


(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)
(a)
(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)
(2,T)
(b)
(4,C)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(2,T)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(13,W)
(c)
(4,C)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(2,T)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(13,W)
(20,B)
(d)

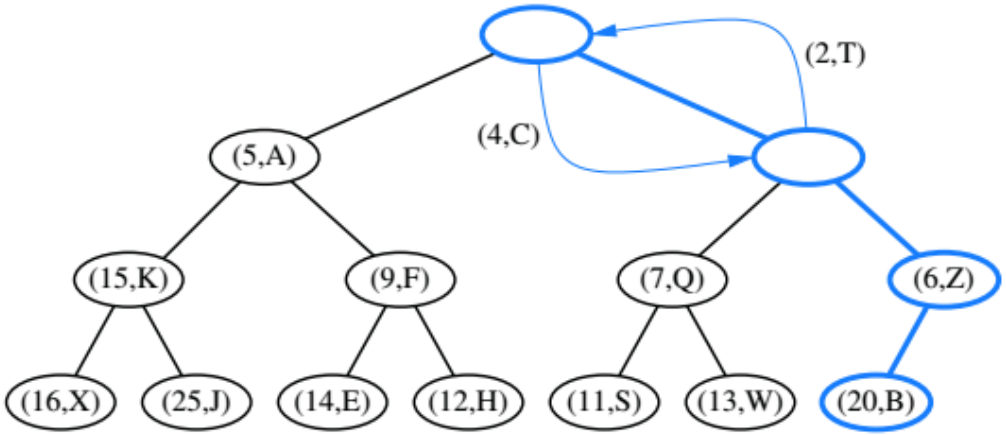
Heap Insert



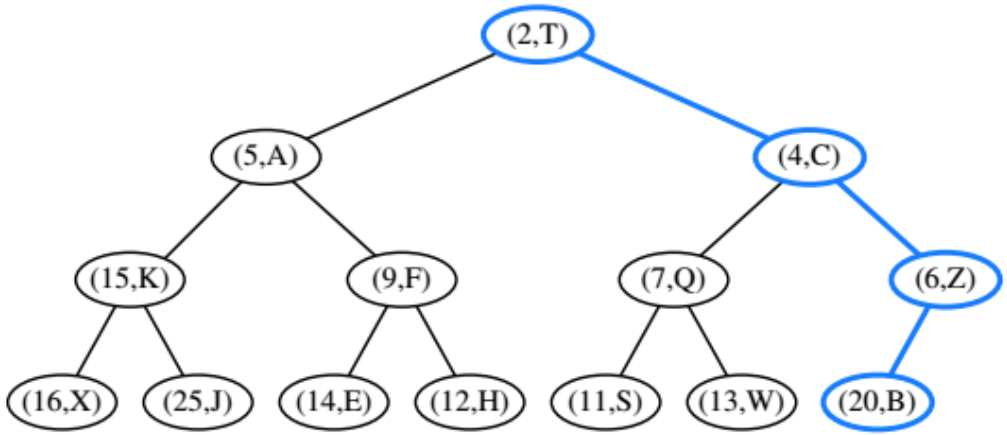
(e)



(f)

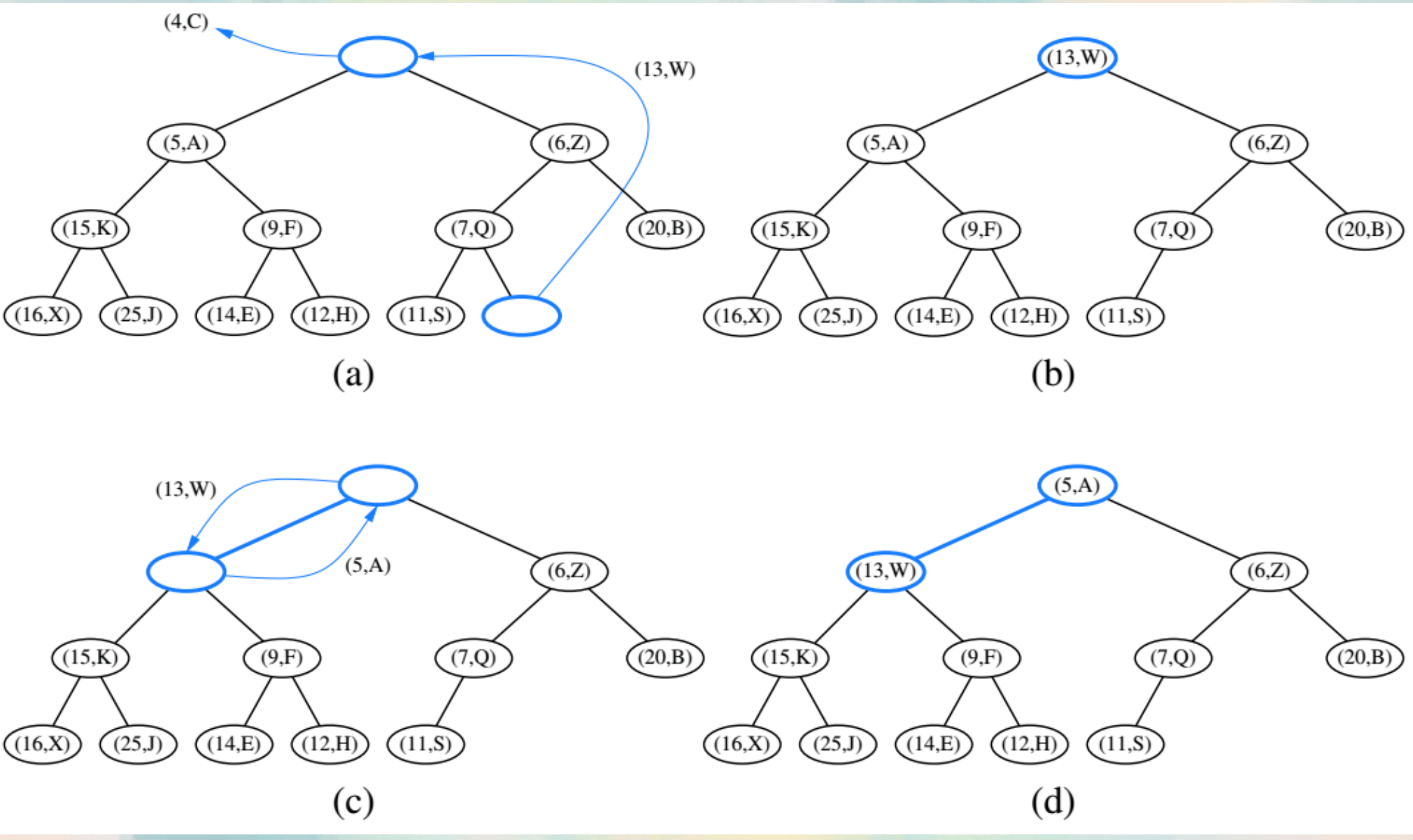


(g)

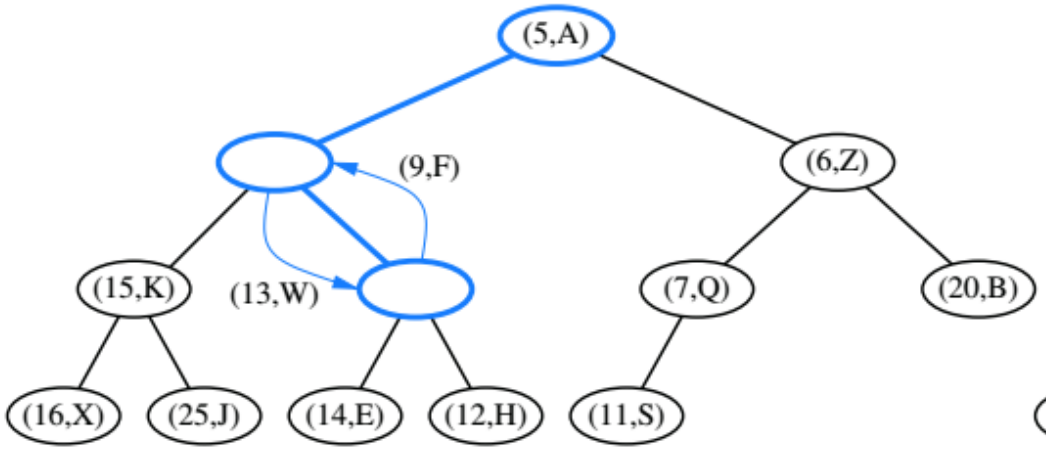


(h)

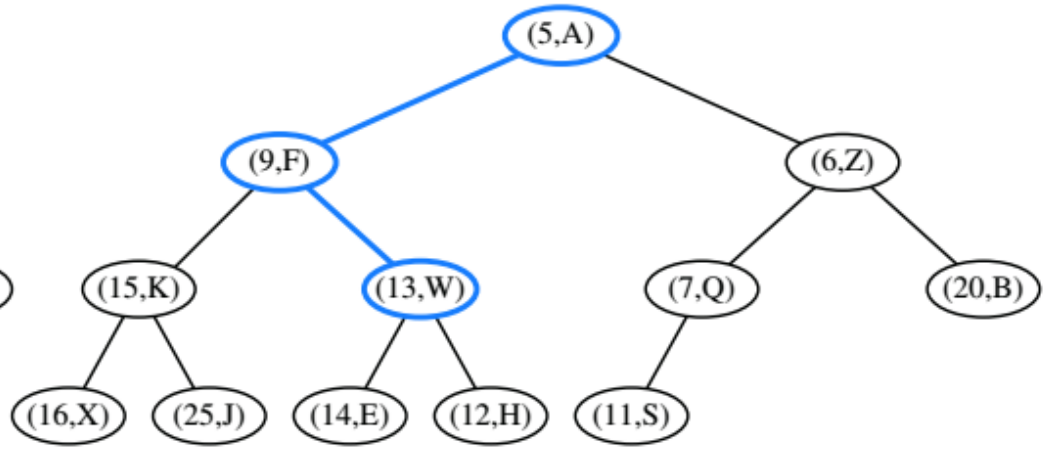
Heap Delete


(4,C)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(a)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(b)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(c)
(5,A)
(13,W)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(d)

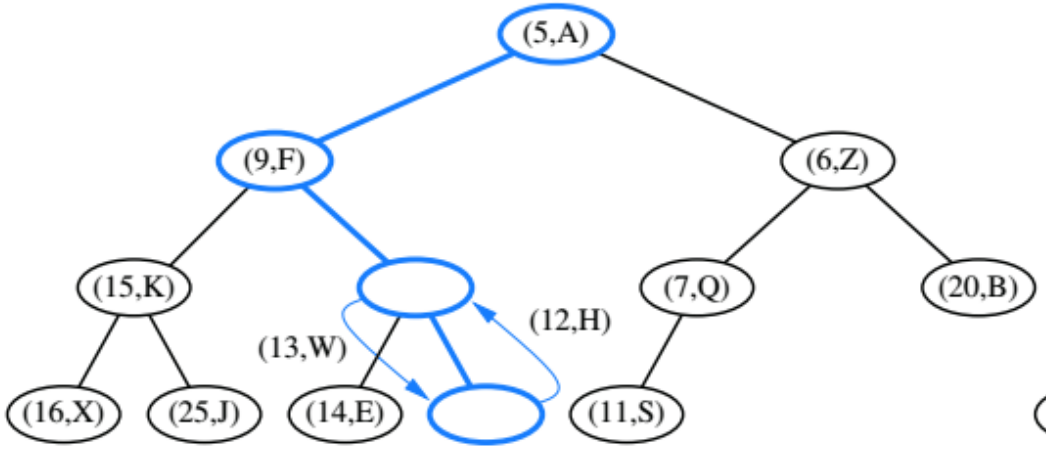
Heap Delete



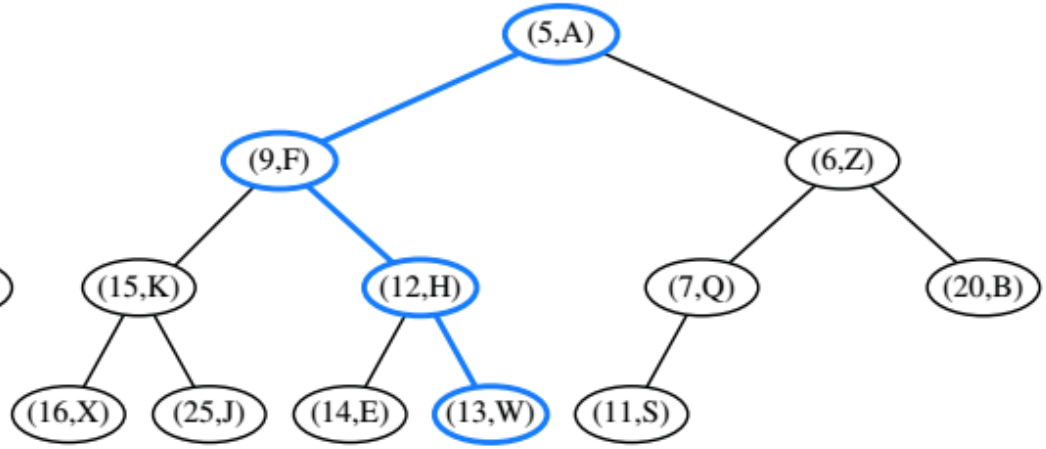
(e)



(f)



(g)



(h)

Heap Operation Runtimes

- size, isEmpty: O(1)
- findMin() or findMax(): O(1)
- insert(): O(logn) due to heapify operation
- removeMin() or removeMax(): O(logn) due to heapify operation.

Priority Queues

What do we mean by priority?

- Recall that queue is a First-in, First-out data structure.

- Sometimes we may want to have a First-come, First-served data structure:
  - managing which planes get to land in an air traffic control sitution.
  - scheduling algorithms which need to determine which application gets CPU time and how much time to give.
  - assigning patients in an emergency room based on severity of injury or illness.

- An item with a high priority gets moved to the front of the queue
  - priorities can be assigned based on a number of conditions.

Priority Queue

- ***priority queue***: a queue which allows removal based on an element's current priority in the queue.
  - elements are added in no particular order and are removed based on the min or max value of the priority (or key)

- priority queues are commonly implemented using a heap underneath.
  - heaps always organize things based on minimum or maximum value, where the min/max is always the root of the heap.
  - the root of the heap is the item with the highest priority.

- NOTE: Two elements can have the same priority, in this case it does not matter which is removed first.

Additional Operations

- Priority queues may also support the changing of priorities as the situation changes:
  - a person who was thought to be only minorly injured is more seriously injured than anticipated.
  - an airplane that was a lower priority may need to be bumped ahead due to malfunction or other issues.

- When a priority of an element changes, the queue must be adjusted.
  - in the case of a heap, this means heapifying the tree.
  - and easy way to do this is to find the node to change, change its priority, and then percolate up or down based on how the new value compares to its parent or children.

- NOTE: Heaps do not generally support a search feature since searching through a heap is O(n) time at its worst.
  - If the heap is another type of Binary Tree, why is the search time **not** O(logn)