UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Priority Queues and Heaps

Mai Dahshan

October 11, 2024

Learning Objectives

- Understand the concept of priority queues and their implementations
- Inserting and deleting in binary heap
- Analyzing time complexity of binary heap

Priority Queue

is an abstract Data type (ADT)

- Priority Queue is an abstract data type where each element has a "priority" associated with it.
- In a priority queue, an element with high priority is served before an element with low priority.
- A priority queue stores a collection of items. An item is a pair (key, element)
- Several elements may have the same priority

Queue

Priority Queue

UVA DATA SCIENCE

Priority Queue Operations

- Main methods of the Priority Queue ADT
  - **construct():** build a priority queue from a set of items
  - **peek():** the highest priority element in the queue without removing it from the queue
  - **enqueue():** insert new data into the queue
  - **dequeue():** removes the element with the highest priority from the queue
- Additional methods
  - **minKey()/maxKey()**: returns, but does not remove, the smallest or largest key of an item
  - **minElement()/maxKey()**: returns, but does not remove, the element of an item with smallest or largest key
  - **size(), isEmpty()**

Priority Queue

- The most important element deserving priority can be **Min or Max** depending on the nature of application
  - A student with the highest grade 'max' could have top priority
  - A worker with the lowest execution time 'min' could have the highest priority
  - A job with shortest burst time 'min' could have highest priority
- If two elements have the same priority, they are served according to their *order in the queue*
- Priority Queue does **not** follow FIFO pattern

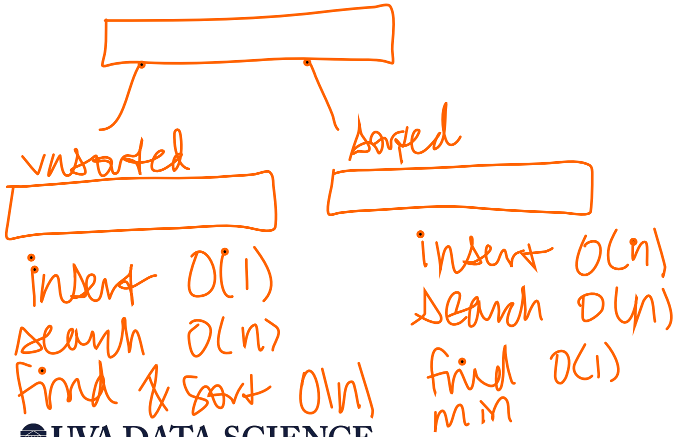
Priority Queue Applications

- Emergency room: Highest priority => most severe condition
- Operating systems: scheduling of processes on the CPU
- Airline check-in: first class, business class, coach

Priority
queue implementation
is FIFO

Priority Queue Implementation

- **Array-based** implementation of priority queue
- **List-based** implementation of priority queue
- **Binary Heap** implementation of priority queue



Array-based implementation

- An **array-based implementation of a priority queue** can be done in two main ways: using **unsorted arrays** or **sorted arrays**
    - Priority Queue Using an Unsorted Array: Elements are inserted in no particular order
        - **Insertion**: append the new element at the end of the array. This operation is O(1) in average case
        - **Find/Remove** the Highest Priority Element: To remove/find the element with the highest priority, you need to scan the entire array to find it, which takes O(n) in average case

Array-based implementation

- Priority Queue Using a Sorted Array: elements are inserted in sorted order
  - **Insertion**: The array is kept sorted, so you need to find the correct position to insert the new element, which takes O(n) in average case
  - **Find/Remove** the Highest Priority Element: the element with the highest priority is always at one end, and you can access or remove it in O(1) in average case

List-based implementation

- A **list-based implementation of a priority queue** can be done using either an **unsorted list** or a **sorted list**
  - Priority Queue Using an Unsorted List: elements are inserted without considering their priority
    - **Insertion**: inserting an element at the end of the list, which takes O(1) in average case
    - **Remove**: scan the entire list to find the minimum or maximum, and then remove it, which takes O(n) in average case

List-based implementation

- **Priority Queue Using a Sorted List:** the list is kept sorted at all times
  - **Insertion**: scan the list to find the correct insertion point, which is O(n) in average case
  - **Remove**: pop the first element, which is O(1) in average case

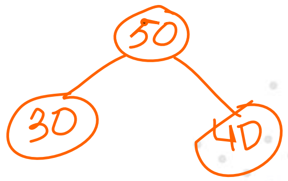
Priority Queue Implementations

- The **unsorted array/list** allows for **quick insertion**
- The **sorted array/list** is more efficient for **deletion**
- Although these implementations (array/list) are straightforward and provide a basic understanding of priority queues, they are not highly efficient
- A more efficient way to implement a priority queue is by using a **binary heap**, which is a specialized type of binary tree with unique structural and heap-order properties

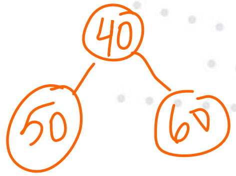
Binary Heap

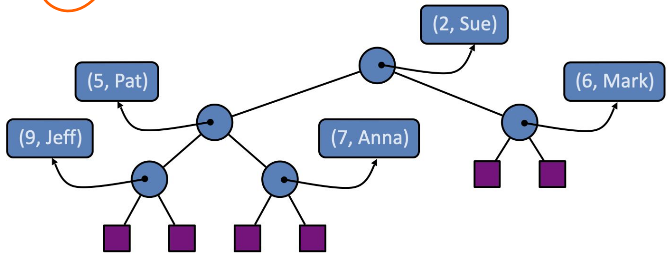
- A heap is a binary tree that satisfies
  - shape property (complete binary tree): the tree is completely filled except possibly the bottom level, which is filled from left to right
  - heap order property
    - every node is less than or equal to its children or every node is greater than or equal to its children
    - The root node is always the smallest node or the largest, depending on the heap order
- A binary heap is **NOT** a binary search tree

max heap:
value of root is more than its children



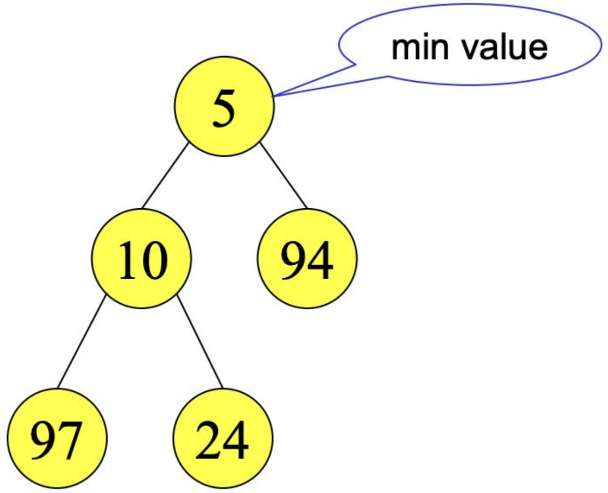
min heap:
value of root is less than its children



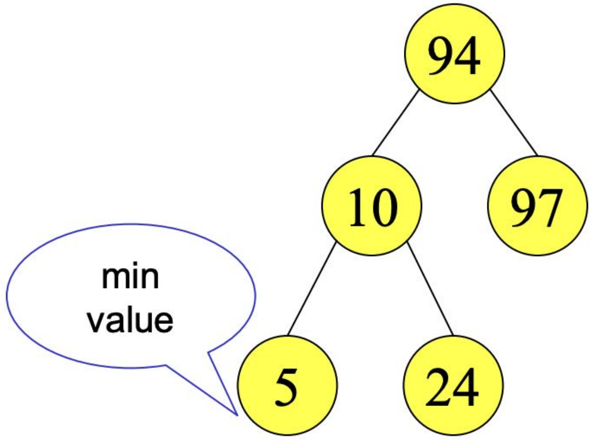


UVA DATA SCIENCE

Binary Heap



Binary Heap



Binary Search Tree

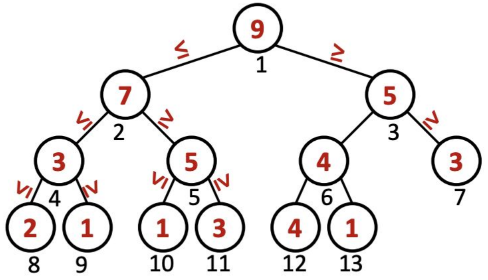
UVA DATA SCIENCE

Binary Heap Invariants

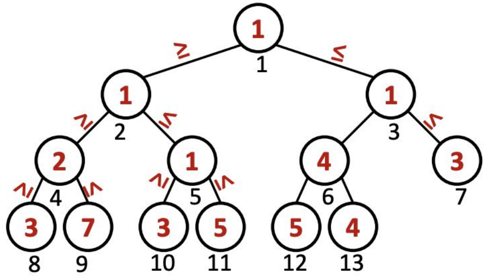
- Two types of heaps
  - Max Heap
    - every parent node stores a value that is greater than or equal to the value of either of its children.
    - the root stores the maximum of all values in the tree.
    - bigger numbers represent higher priorities
  - Min Heap
    - every parent node stores a value that is less than or equal to that of its children
    - The root stores the minimum of all values in the tree
    - smaller numbers represent higher priorities

Binary Heap Invariants

Binary Max-Heap



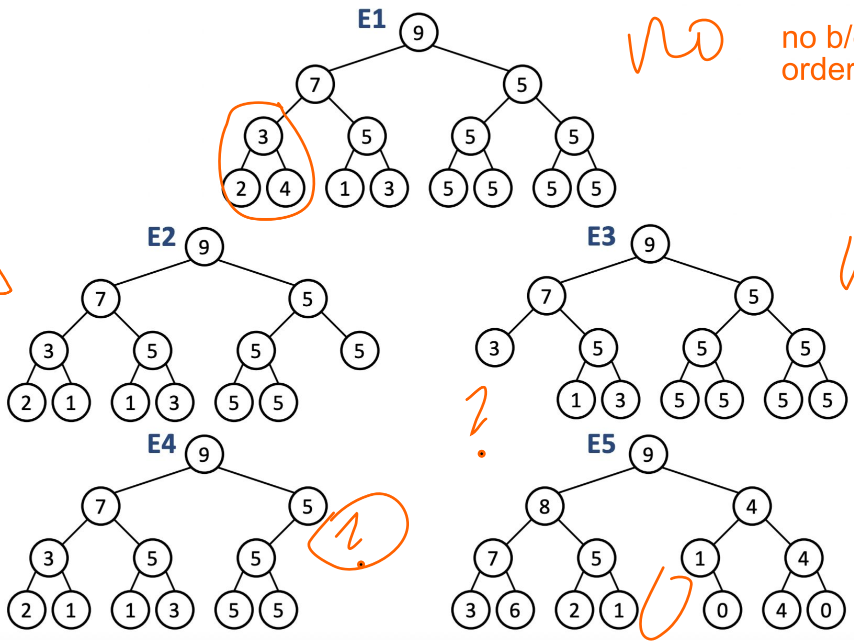
Binary Min-Heap



Binary Heap Visualization: https://visualgo.net/en/heap

Quiz

Which of these binary trees follows the max heap property?



no

no b/c it doesn't satisfy the heap order property

yes

?

no

no b/c it does not satisfy the first condition of being a complete binary tree (doesn't fill from left to right)

no b/c we have a level that was skipped

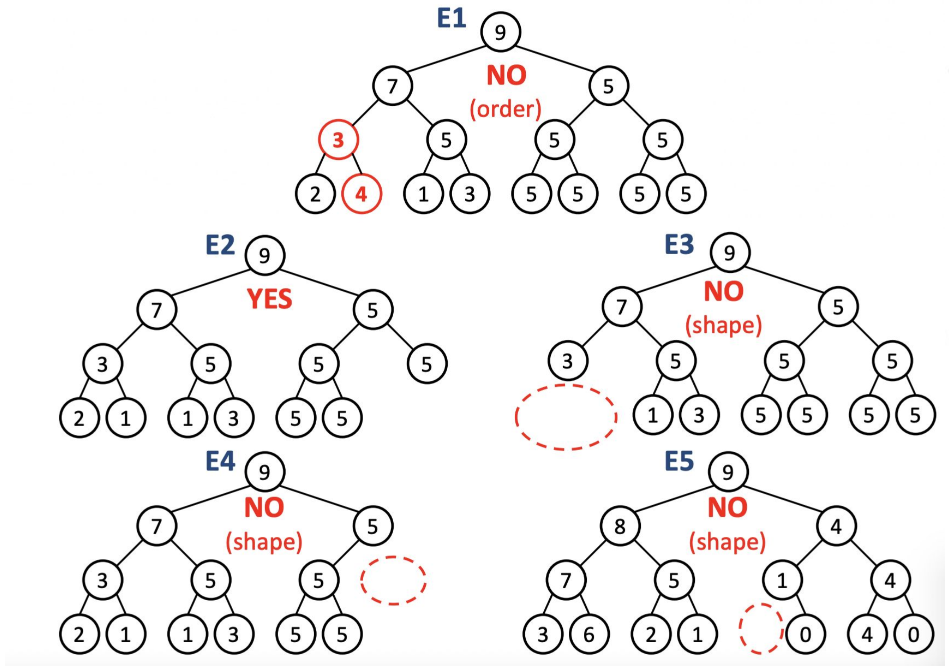
no

no

no b/c it does not satisfy the first condition of a complete binary tree

Quiz

Which of these binary trees follows the max heap property?



UVA DATA SCIENCE

Binary Heap Operations

- The two most important methods on heaps are:
    - **inserting** a new value into the heap. New Nodes added *from left to right*
    - **retrieving the smallest or largest** value from the heap (*removing the root*)
    is O(1) b/c getting the min value is O(1)
- Node insertion must ensure that the insertion maintains both the **order** and **shape** properties of the heap.
- Removing node removes and *returns the root value* in the heap. This method also **rebuilds the heap** because it removes the root, and all nonempty trees must have a root by definition

UVA DATA SCIENCE

Inserting a node into a Heap

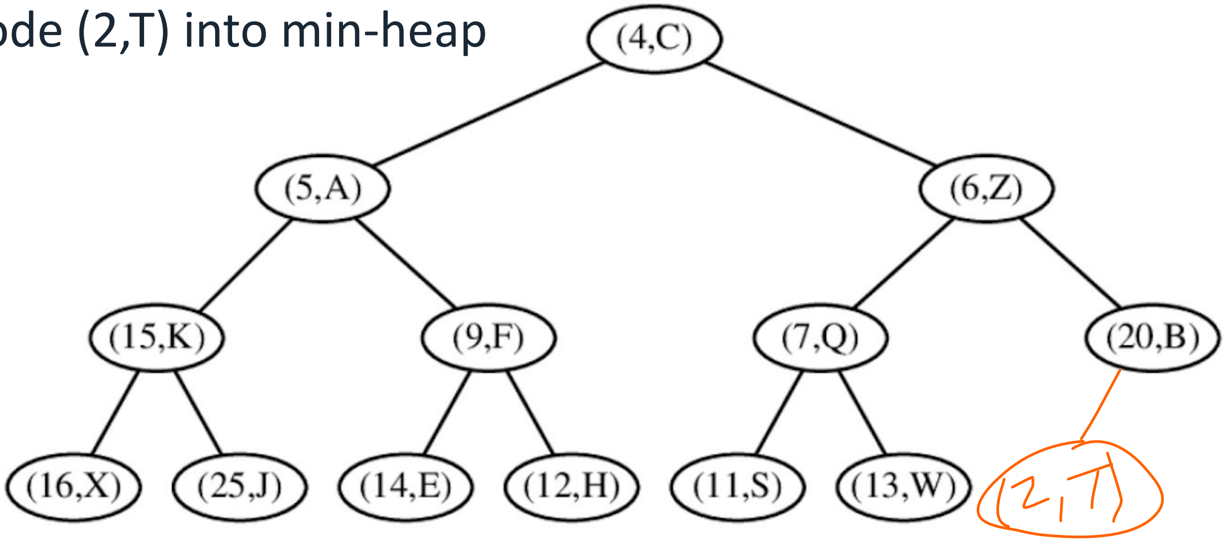
- Insert a new element to the end (***bottom level***) of the heap – *maintaining the* **shape property**
- Fix up to restore the order property
- The number of operations required is dependent on the number of levels the new element must rise to satisfy the heap property
- **Time complexity in worst case: O(log *n*)**

Inserting a node into a Heap

- After the insertion of a new element, the heap-order property may be violated
- Upheap (bubble up) restores the heap-order property by swapping ***new node*** along an upward path from the insertion node
- Upheap terminates when the new element reaches the root or a node whose parent has a key smaller/larger than or equal to ***new element***
- Since a heap has height ***O***(log ***n***), upheap runs in ***O***(log ***n***) time in worst case

Inserting a node into a Heap

Insert new node (2,T) into min-heap

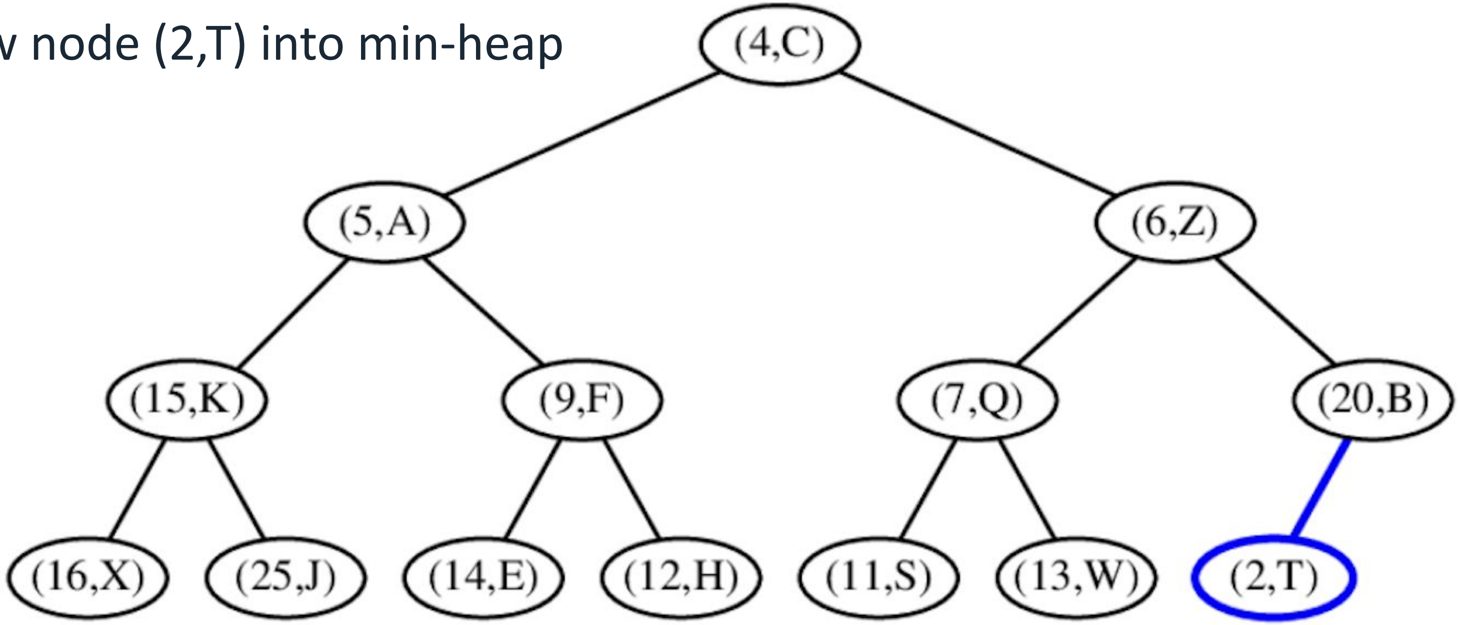


initial heap

The node would insert here, but we need to make sure our element is correctly placed in the heap

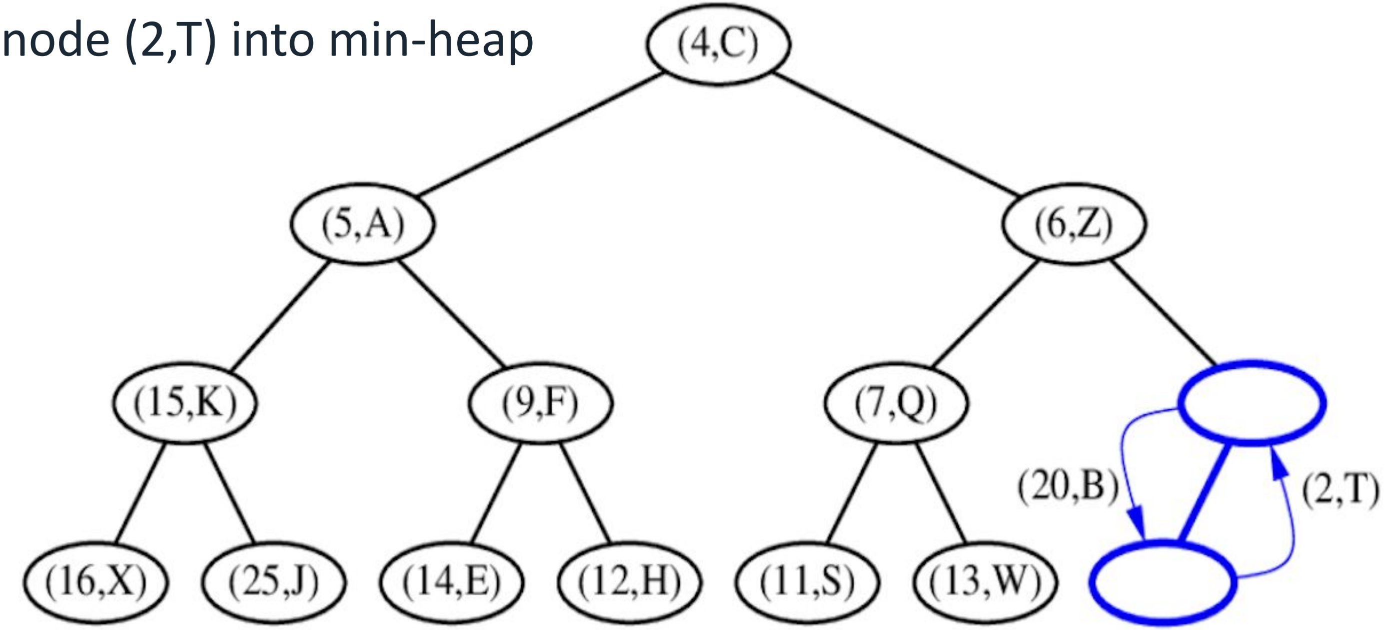
Inserting a node into a Heap

Insert new node (2,T) into min-heap



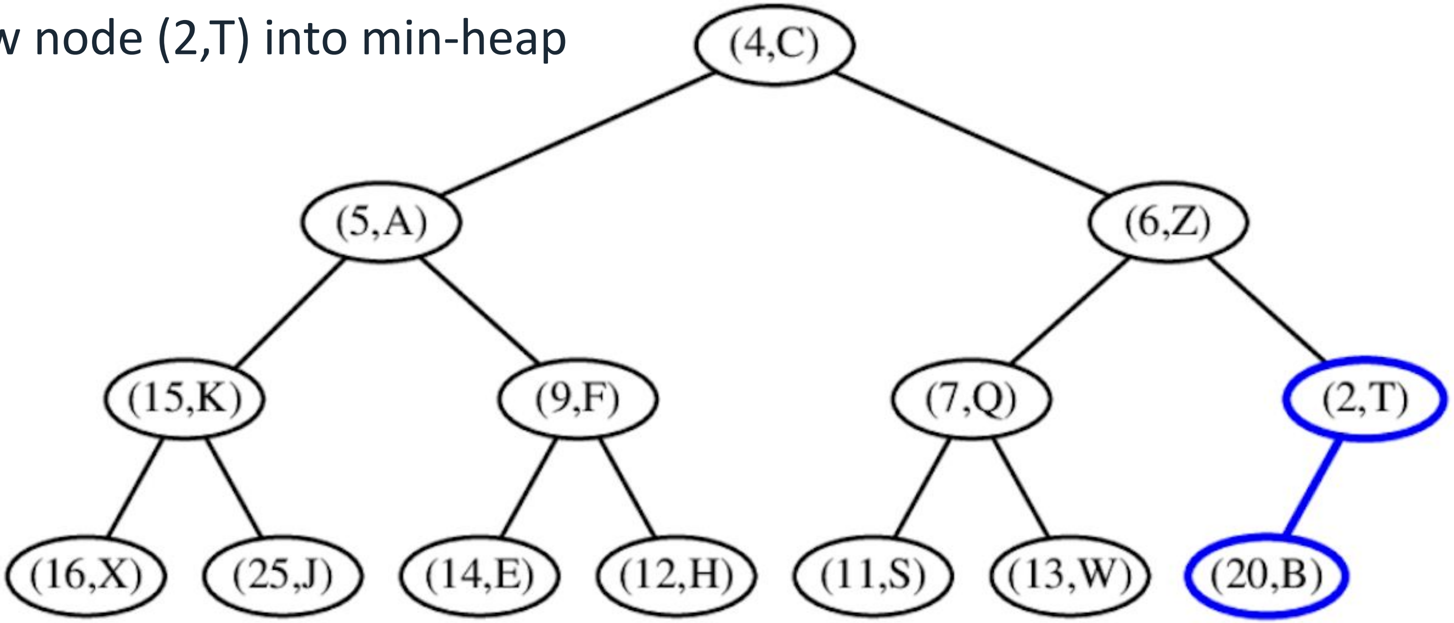
Inserting a node into a Heap

Insert new node (2,T) into min-heap



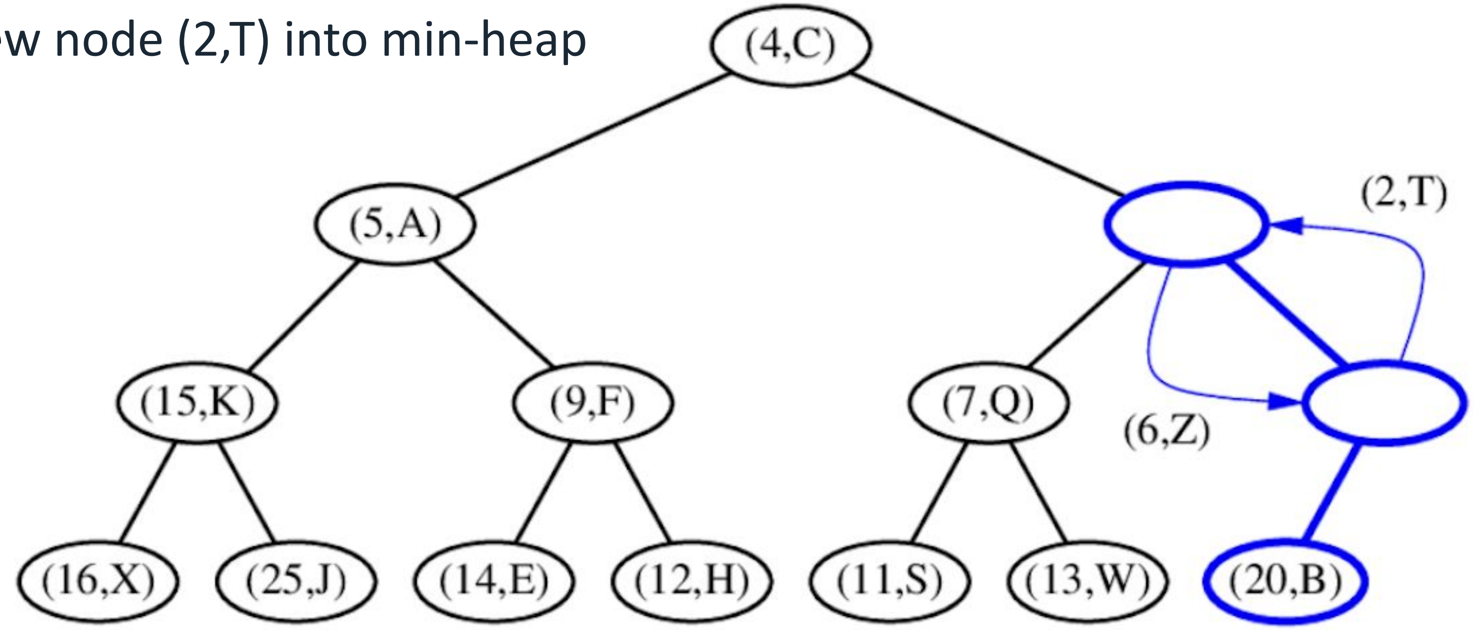
Inserting a node into a Heap

Insert new node (2,T) into min-heap



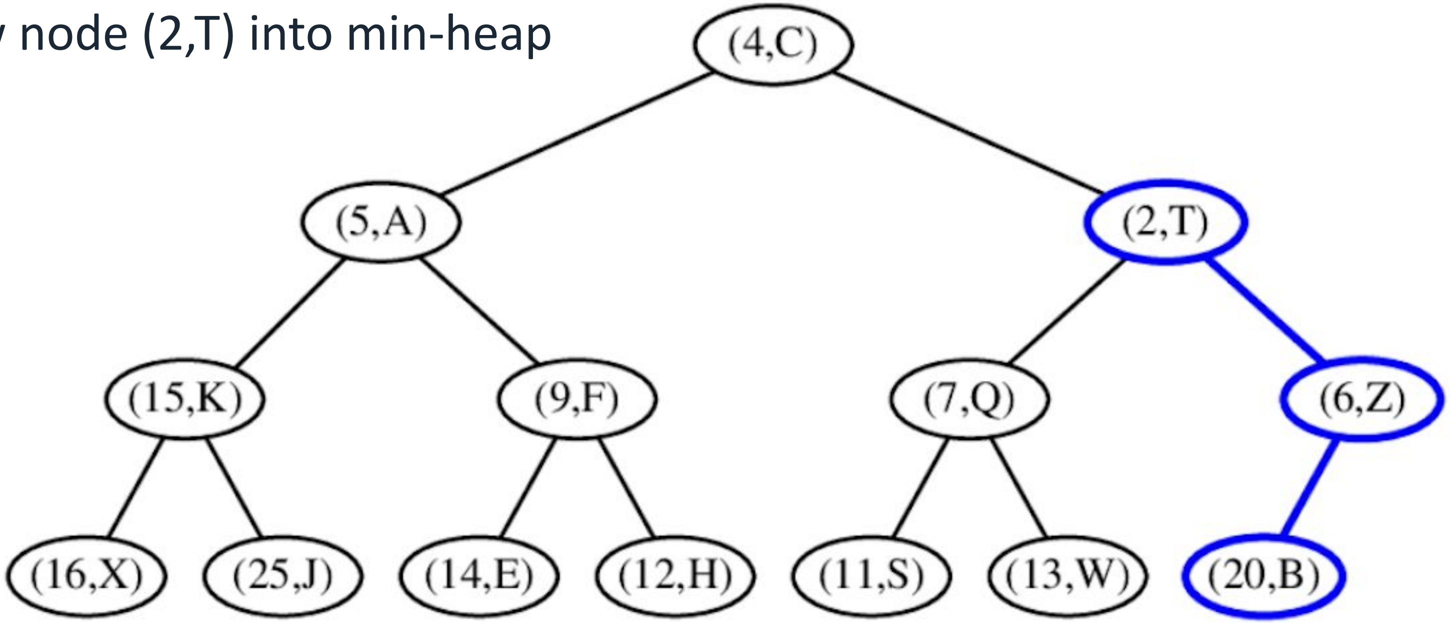
Inserting a node into a Heap

Insert new node (2,T) into min-heap



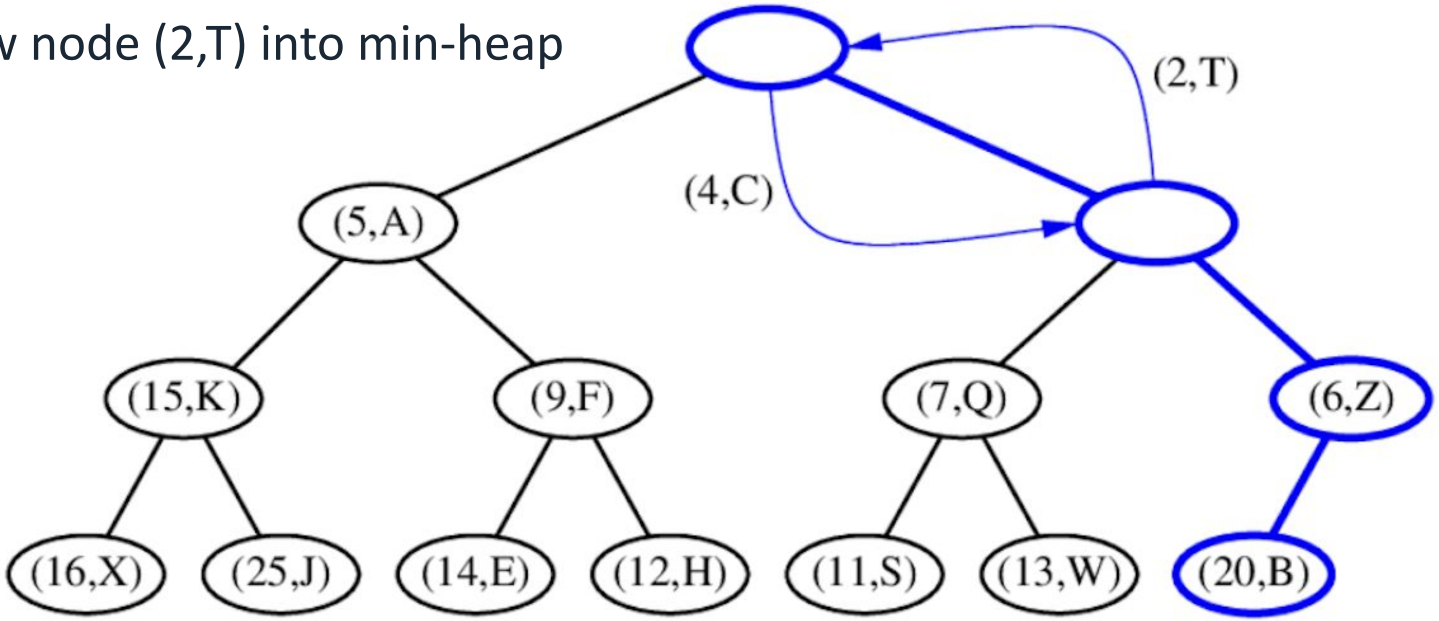
Inserting a node into a Heap

Insert new node (2,T) into min-heap



Inserting a node into a Heap

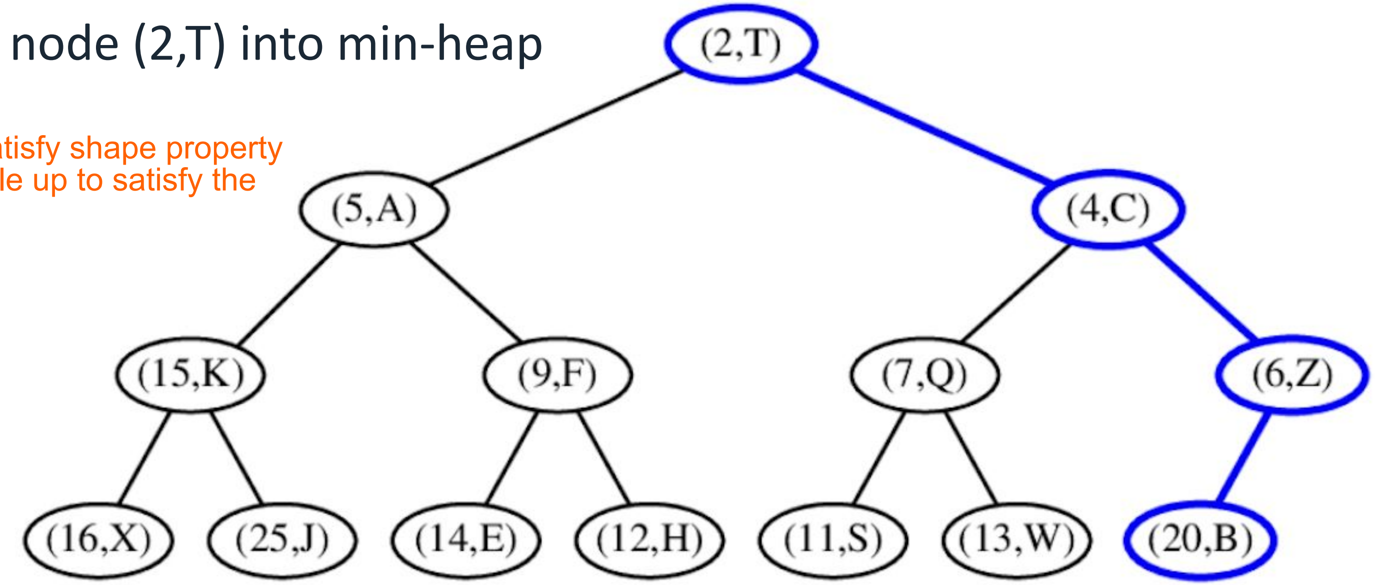
Insert new node (2,T) into min-heap



Inserting a node into a Heap

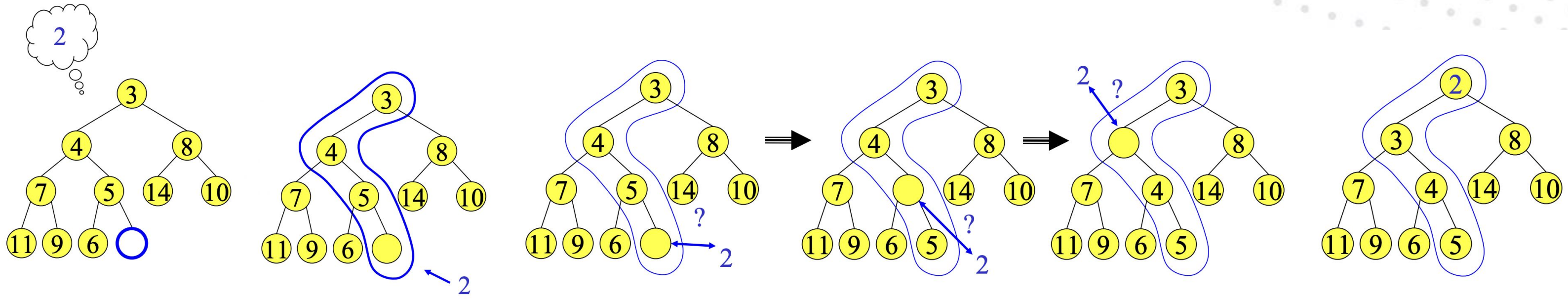
Insert new node (2,T) into min-heap

so step #1 is to satisfy shape property
step #2 is to bubble up to satisfy the order property



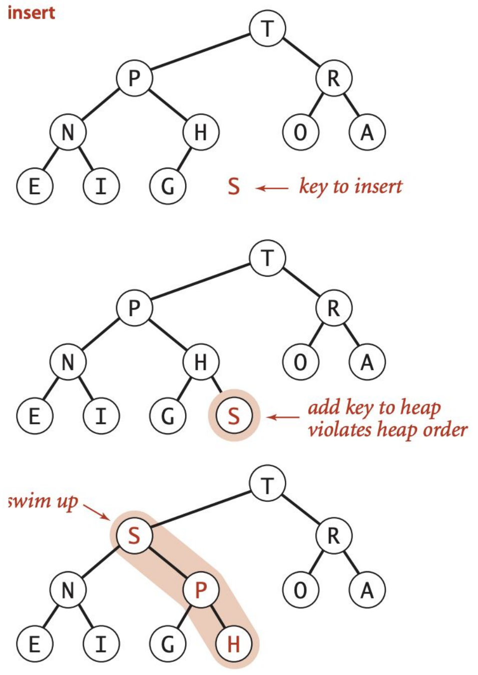
Inserting a node into a Heap

Insert new node into min-heap



bubbling up the tree
is O(log n) b/c
branching up towards root

Inserting a node into a Heap



UVA DATA SCIENCE

Deleting a node into a Heap

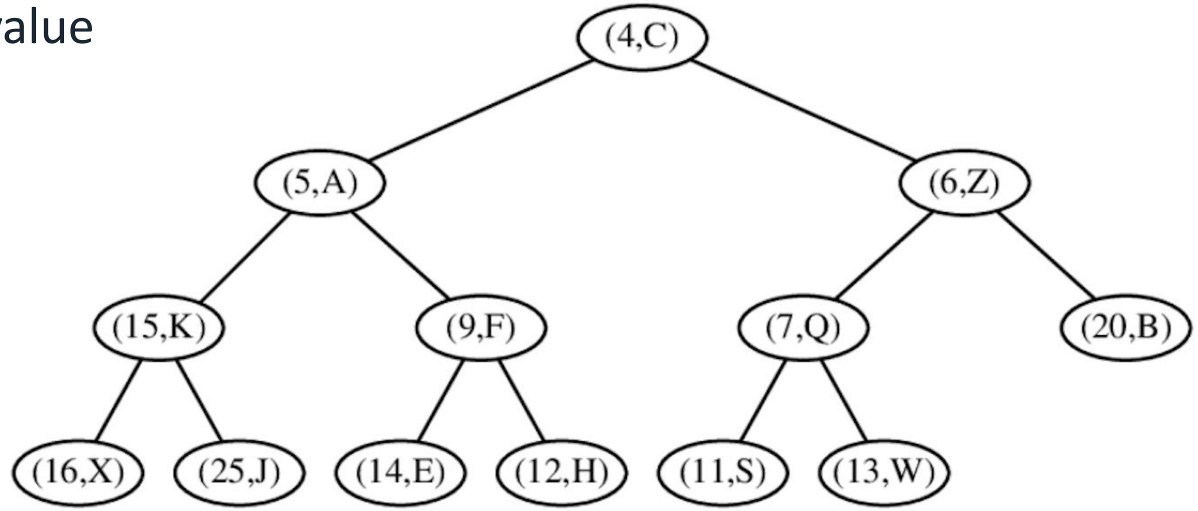
- Replace the root of the heap with the *last element on the last level – maintaining the* shape property
- In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree.
- Time complexity in worst case: O(log *n*)

Deleting a node into a Heap

- After replacing the root key with the element e of the last node, the heap-order property may be violated
- Downheap restores the heap property by swapping element e with the child with the smallest/largest key along a downward path from the root
- Downheap terminates when element e reaches a leaf or a node whose children have values greater/less than or equal to ***element e***
- Since a heap has height ***O***(log ***n***), downheap runs in ***O***(log ***n***) time in worst case

Deleting a node into a Heap

Remove min value

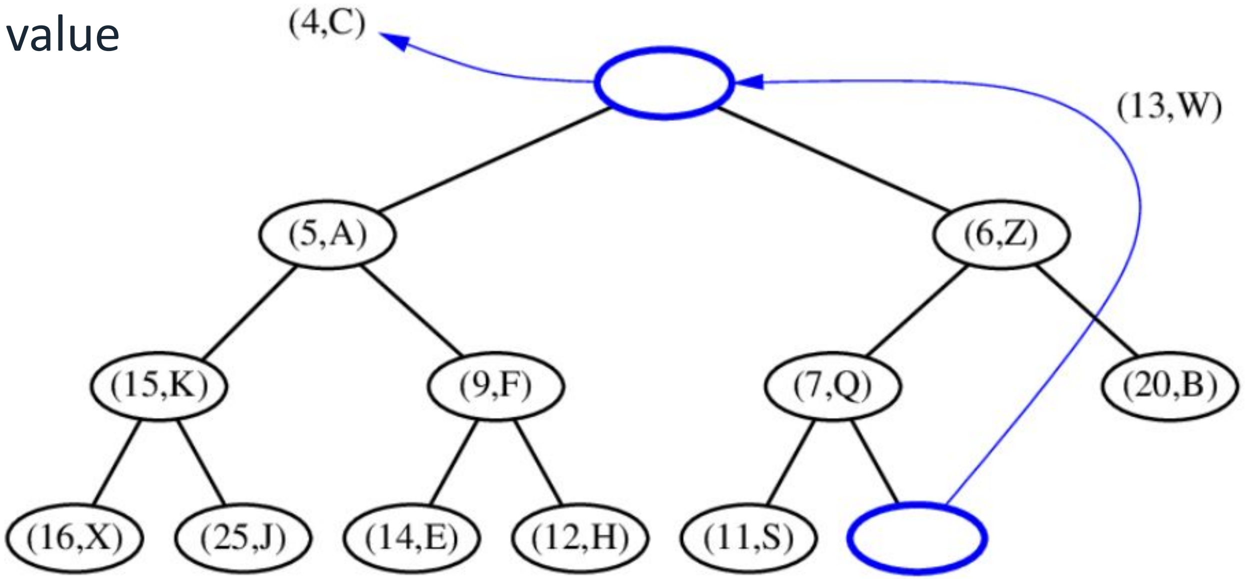


initial heap

we choose 13 b/c its the newest node in the tree (if we fill the bottom level left to right)

Deleting a node into a Heap

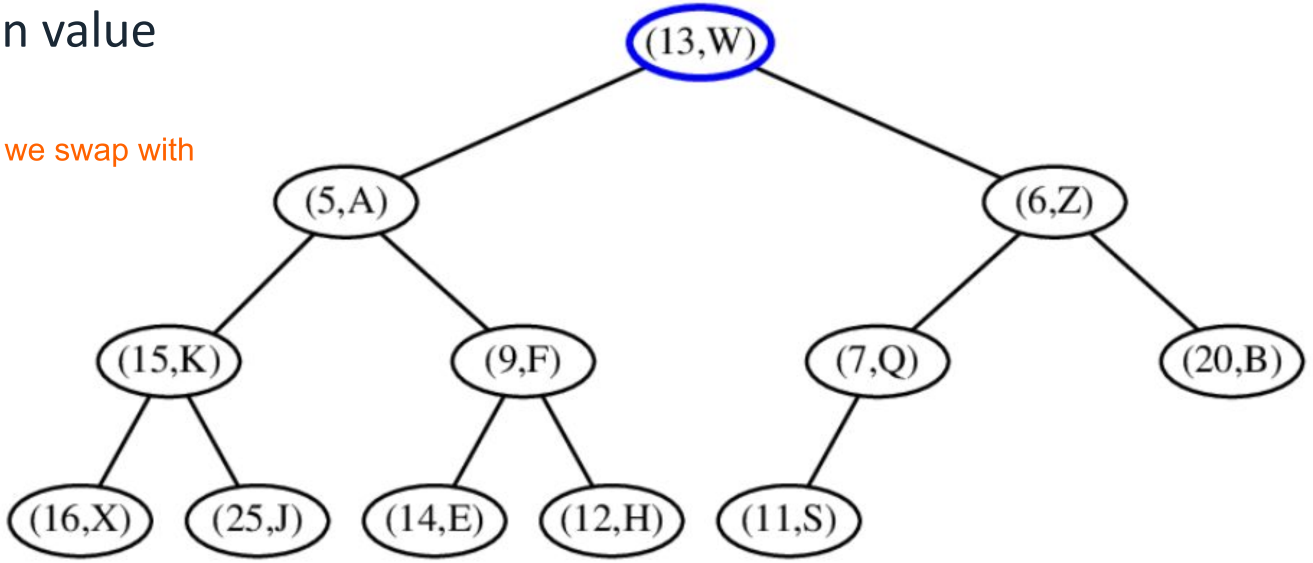
Remove min value



Deleting a node into a Heap

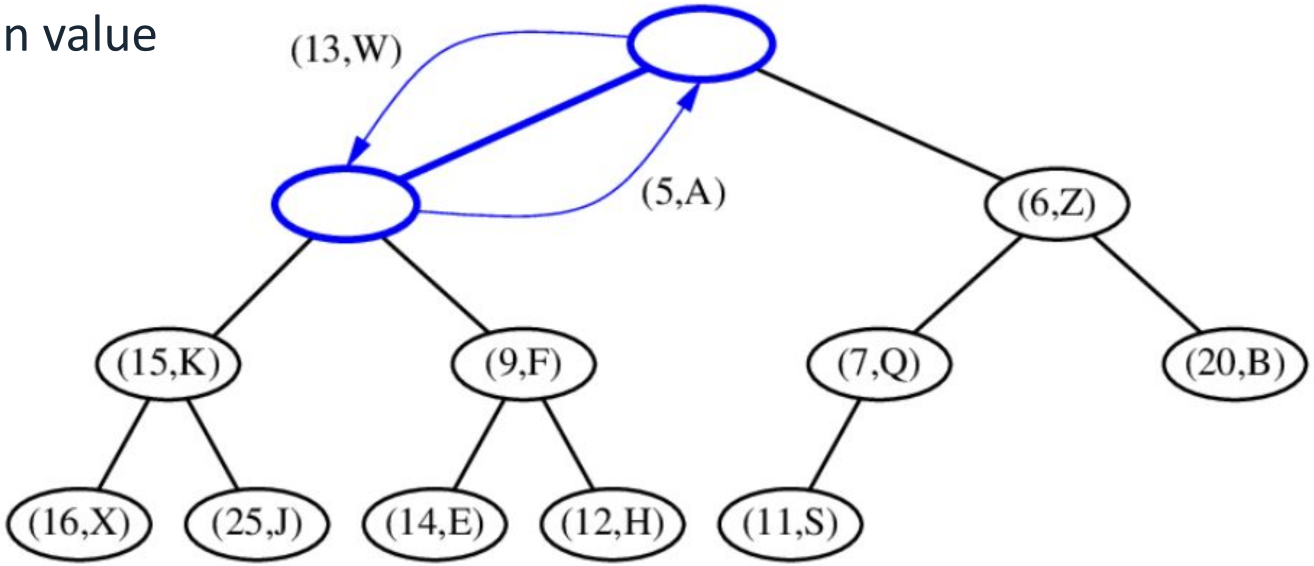
Remove min value

with these swaps, we swap with the largest child



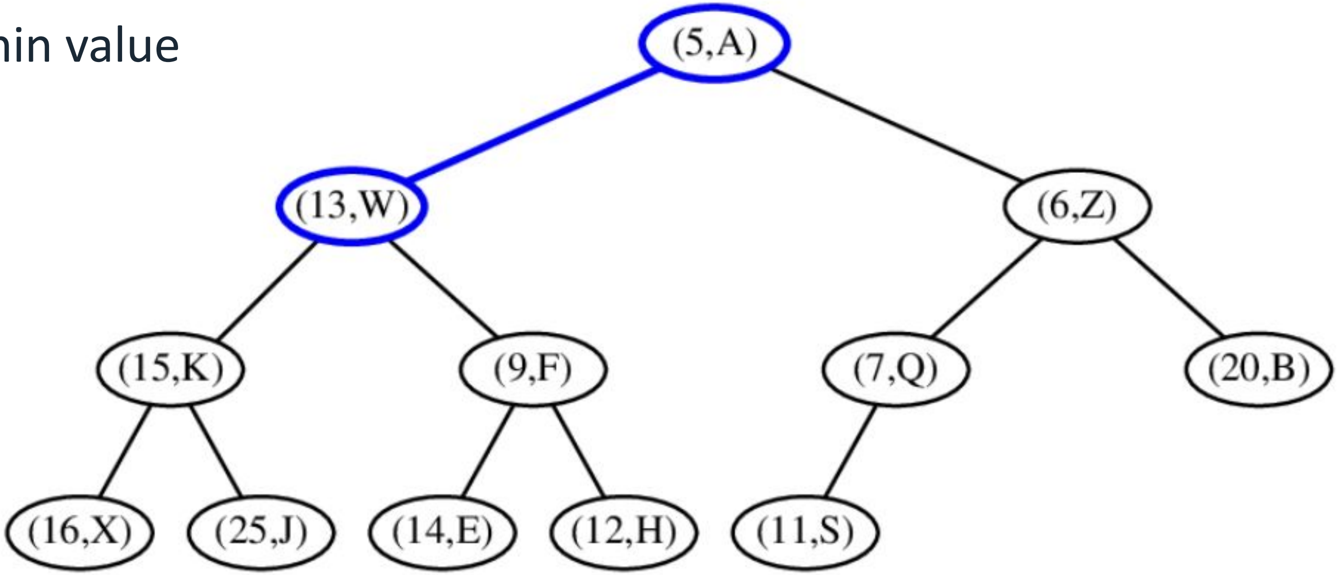
Deleting a node into a Heap

Remove min value



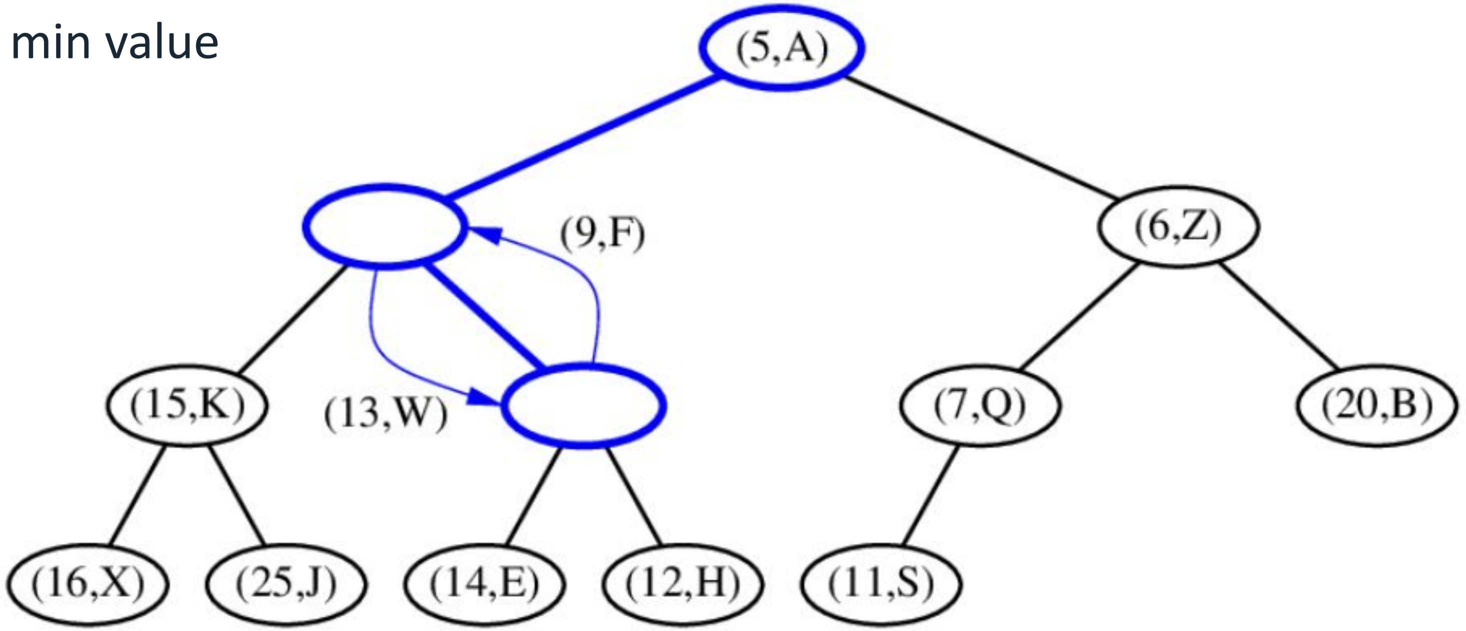
Deleting a node into a Heap

Remove min value



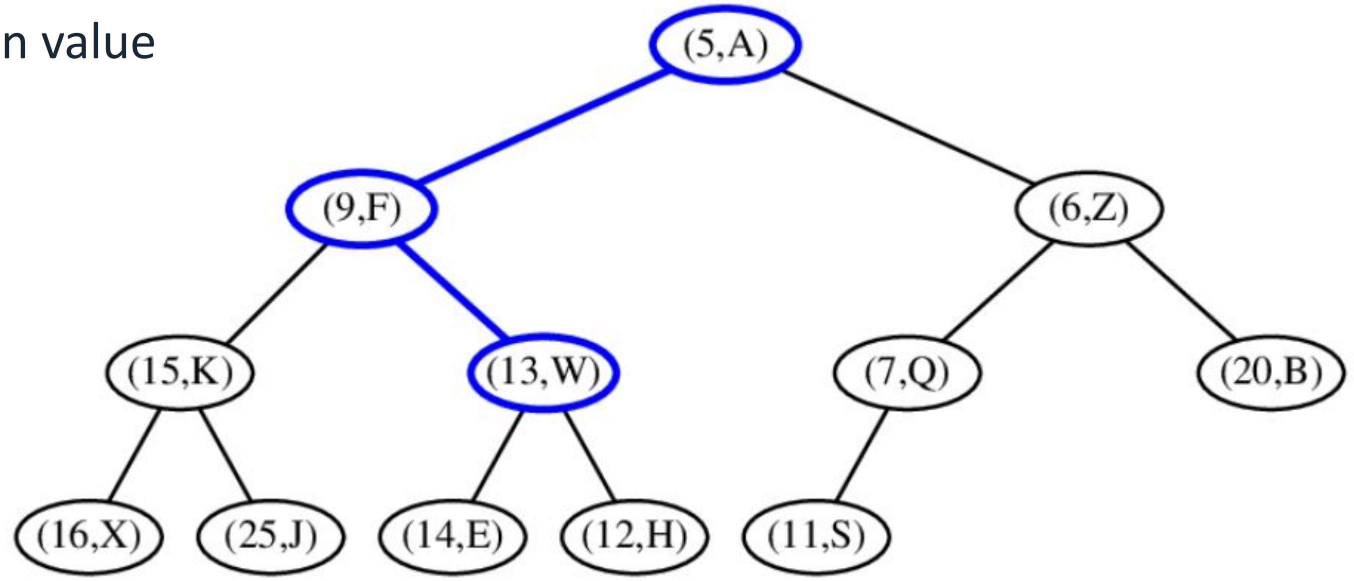
Deleting a node into a Heap

Remove min value



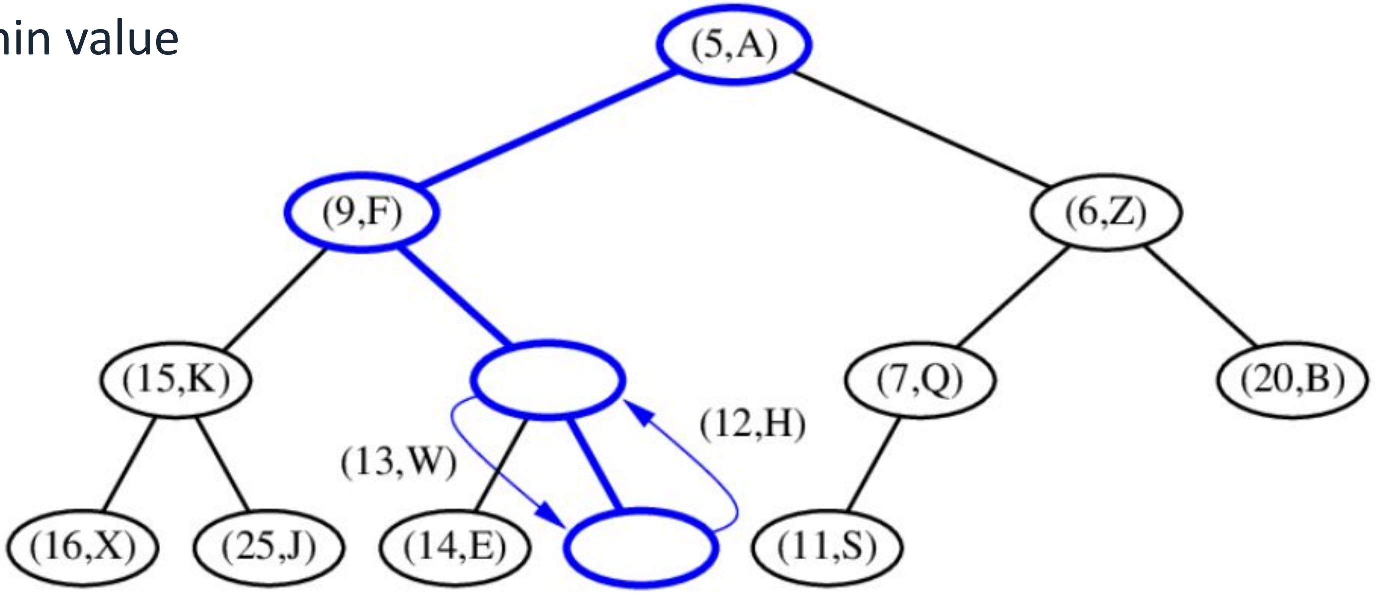
Deleting a node into a Heap

Remove min value



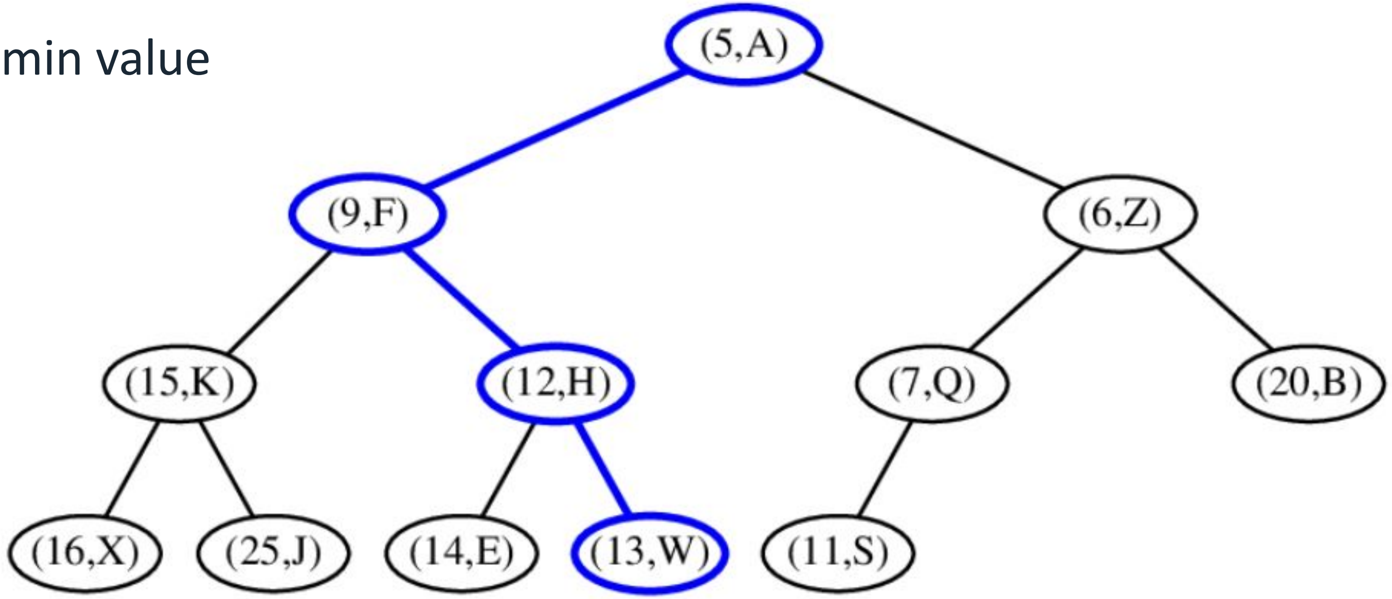
Deleting a node into a Heap

Remove min value



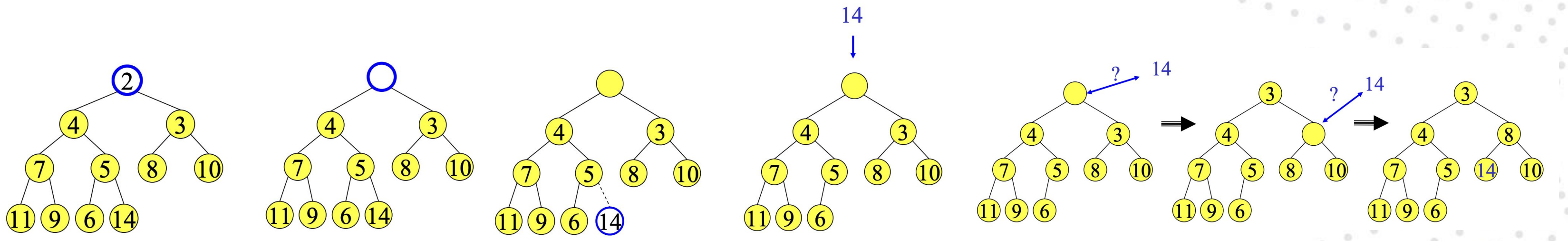
Deleting a node into a Heap

Remove min value



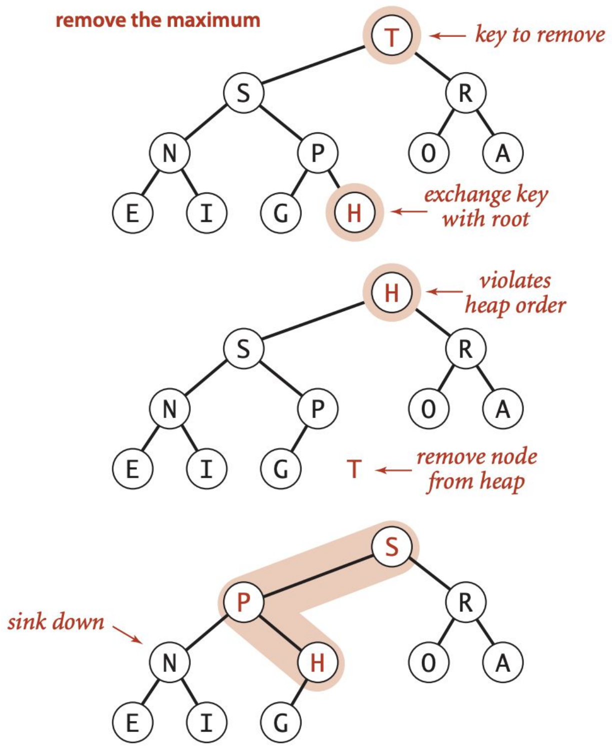
Deleting a node into a Heap

Remove min value



Deleting a node into a Heap

Remove max value



Why keep a balanced Tree?

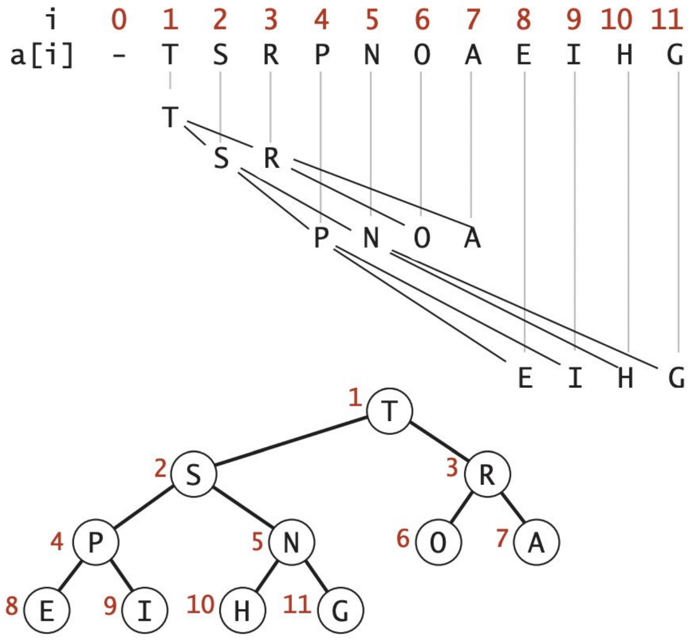
- Height of the tree determines the time required for adding and removing elements - keeping the tree balanced maximizes performance
- Adding an element requires 1 step for every level of height
- A tree of height h contains $2^{h-1} <= n < 2^h$ elements or: $h-1 <= \log_2(n) < h$
- Therefore: adding and removing elements is O(log(n))

Binary Heap ~ 1D Array

- We can store the elements of our heap in a one-dimensional array in strict left-to- right, **level order *(“breadth-first traversal”)***
- That is, we store all of the nodes on level *i* from left to right before storing the nodes on level *i* + 1. This one-dimensional array representation of a heap i

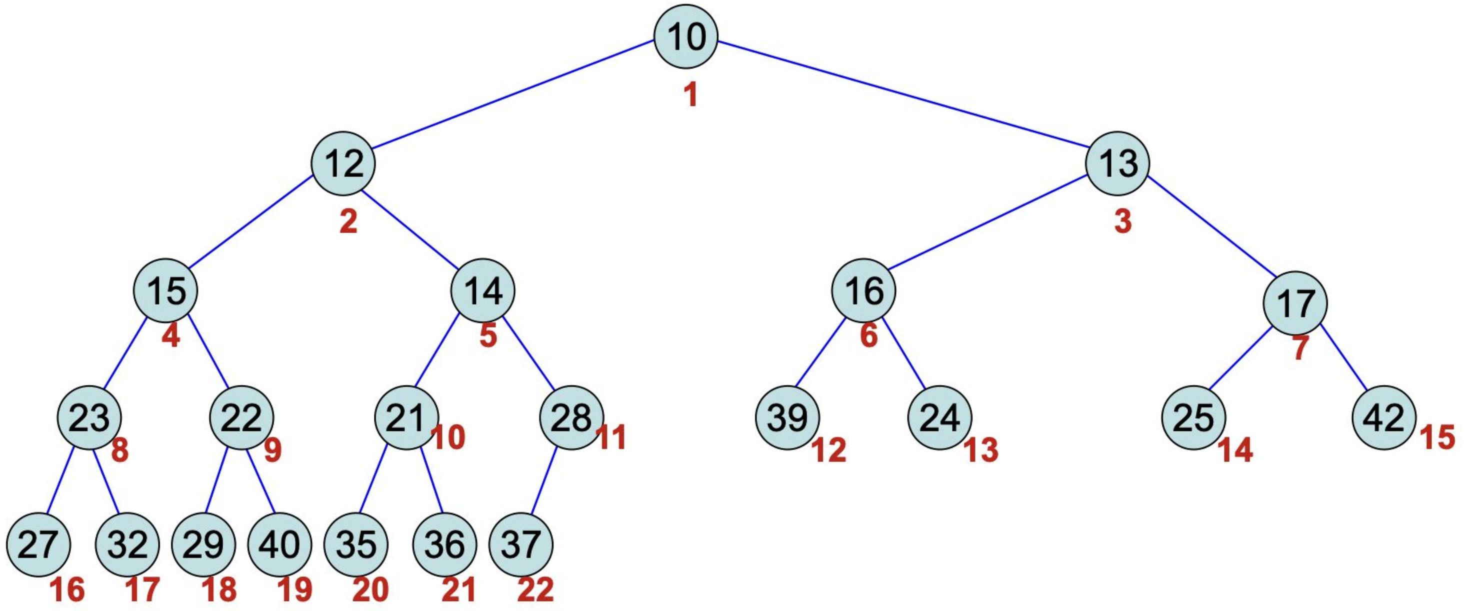
Binary Heap ~ 1D Array

- Heaps are often represented as arrays, where for a node at index i. Starting at index 1
  - Left(i)= 2*i
  - Right(i)= 2*i+1
  - Parent(i) = floor (i/2)



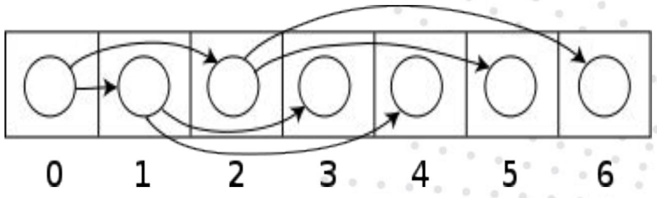
Heap representations

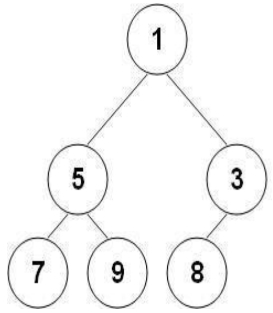
Binary Heap ~ 1D Array

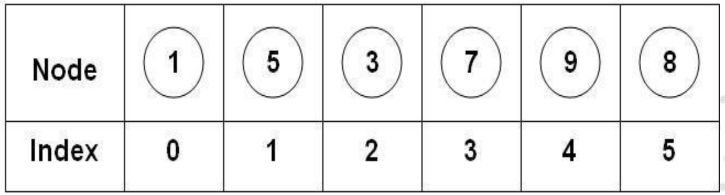


Binary Heap ~ 1D Array

- Heaps are often represented as arrays, where for a node at index i. Starting at index 0
    - Left(i)= 2*i+1
    - Right(i)= 2*i+2
    - Parent(i) = floor ((i-1)/2)





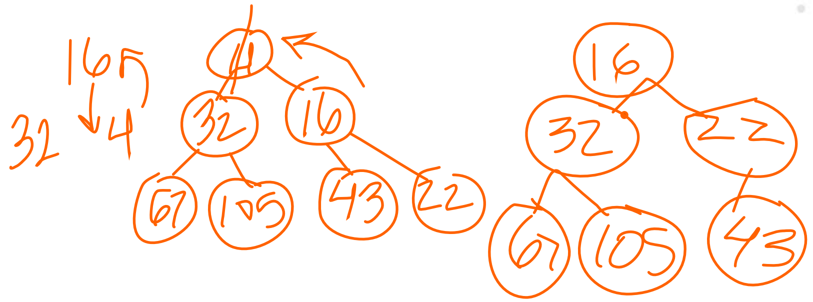


Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Activity

Starting with an empty array-based min binary heap, which is the result after

1. insert (in this order): 16, 32, 4, 67, 105, 43, 2 2
2. Remove once



16
5
32
4
4
32
16
67
105
43
22
16
32
22
67
105
43