Priority Queues ADT and Heaps

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

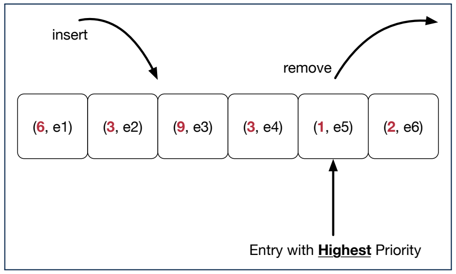
Learning Outcomes of this Lecture

This module is designed to help you understand:

- The ***Priority Queue*** (***PQ***) ADT
- The ***Heap*** Data Structure (Properties & Operations)
- Time Complexities of ***Heap***-Based ***PQ***

What is a Priority Queue?

- A ***Priority Queue (PQ)*** stores a collection of ***entries***.



  - Each ***entry*** is a pair: an ***element*** and its ***key***.
  - The ***key*** of each ***entry*** denotes its ***element***'s "priority".
  - ***Keys*** in a Priority Queue (PQ) are **not** used for uniquely identifying an entry.
- In a PQ, the next entry to remove has the "***highest***" priority.
  - e.g., In the stand-by queue of a fully-booked flight, ***frequent flyers*** get the higher priority to replace any cancelled seats.
  - e.g., A network router, faced with insufficient bandwidth, may only handle ***real-time tasks*** (e.g., streaming) with highest priorities.
  - e.g., When performing Dijkstra's ***shortest path algorithm*** on a weighted graph, the vertex with the minimum *D* value gets the highest priority to be visited next.

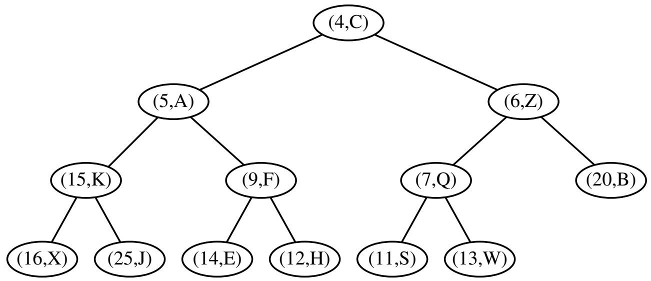
The Priority Queue (PQ) ADT

- ***min***
  [***precondition***: PQ is not empty]
  [***postcondition***: return entry with **highest priority** in PQ]
- ***size***
  [***precondition***: **none**]
  [***postcondition***: return number of entries inserted to PQ]
- ***isEmpty***
  [***precondition***: **none**]
  [***postcondition***: return whether there is no entry in PQ]
- ***insert(k, v)***
  [***precondition***: PQ is not full]
  [***postcondition***: insert the input entry into PQ]
- ***removeMin***
  [***precondition***: PQ is not empty]
  [***postcondition***: remove and return a **min** entry in PQ]

Heaps

A ***heap*** is a ***binary tree*** which:

1. Stores in each node an ***entry*** (i.e., ***key*** and ***value***).

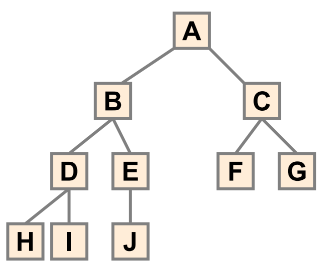


2. Satisfies a ***structural*** property of tree **organization**
3. Satisfies a ***relational*** property of stored **keys**

BT Terminology: Complete BTs

A *binary tree* with ***height*** h is considered as ***complete*** if:

- Nodes with ***depth*** $\leq h - 2$ has two children.
- Nodes with ***depth*** $h - 1$ may have zero, one, or two child nodes.
- ***Children*** of nodes with ***depth*** $h - 1$ are filled from left to right.



Q1: ***Minimum*** # of nodes of a ***complete*** BT? $(2^h - 1) + 1 = 2^h$

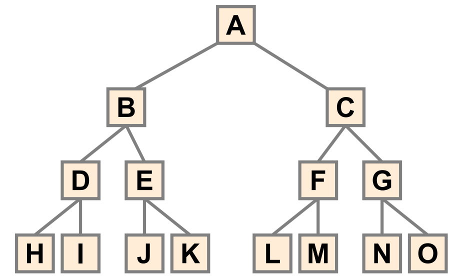
Q2: ***Maximum*** # of nodes of a ***complete*** BT? $2^{h+1} - 1$

BT Terminology: Full BTs

A *binary tree* with ***height*** h is considered as ***full*** if:

Each node with ***depth*** $\leq h - 1$ has two child nodes.

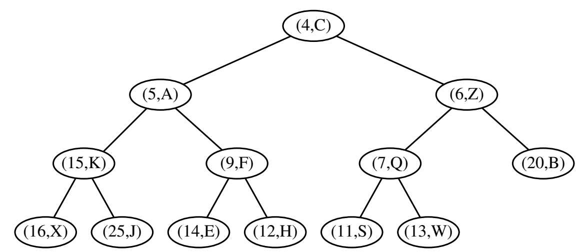
That is, all ***leaves*** are with the same ***depth*** h.



Q1: ***Minimum*** # of nodes of a complete BT? $2^{h+1} - 1$

Q2: ***Maximum*** # of nodes of a complete BT? $2^{h+1} - 1$

Heap Property 1: Structural



A ***heap*** with ***height h*** satisfies the ***Complete BT Property***:

- Nodes with ***depth*** $\leq$ **h – 2** has two child nodes.
- Nodes with ***depth h - 1*** may have zero, one, or two child nodes.
- Nodes with ***depth h*** are filled from left to right.

Q. When the # of nodes is n, what is h? $\lfloor \log_2 n \rfloor$

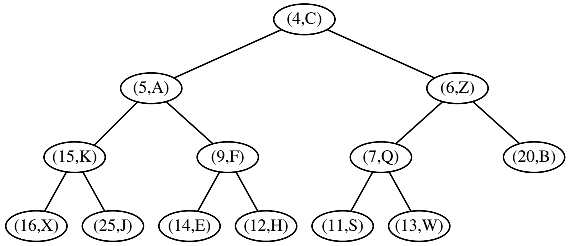
Q. # of nodes from Level 0 through Level $h - 1$? $2^h - 1$

Q. # of nodes at Level h? $n - (2^h - 1)$

Q. **Minimum** # of nodes of a complete BT? 2^h

Q. **Maximum** # of nodes of a complete BT? $2^{h+1} - 1$

Heap Property 2: Relational



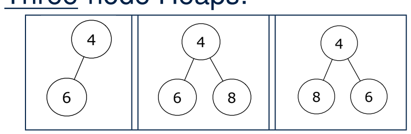
Keys in a ***heap*** satisfy the ***Heap-Order Property*** :

- Every node ***n*** (other than the root) is s.t. ***key***$(n) \geq$ ***key***(**parent**(n))
  - $\Rightarrow$ ***Keys*** in a ***root-to-leaf path*** are sorted in a non-descending order.
    e.g., Keys in entry path $\langle (4, C), (5, A), (9, F), (14, E) \rangle$ are sorted.
  - $\Rightarrow$ The ***minimal key*** is stored in the ***root***.
    e.g., Root $(4, C)$ stores the minimal key 4.
- ***Keys*** of nodes from ***different subtrees*** are **not** constrained at all.
  e.g., For node $(5, A)$, key of its ***LST***'s root (15) is not minimal for its ***RST***.

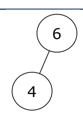
Heaps: More Examples

- The ***smallest*** *heap* is just an empty binary tree.
- The ***smallest*** **non-empty** *heap* is a one-node heap.
  e.g.,

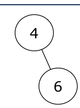
- Two-node and Three-node Heaps:



- These are **not** two-node heaps:



relational property violated



structural property violated

Heap Operations

- There are three main operations for a ***heap*** :
  1. **Extract the Entry with Minimal Key**:
     Return the stored entry of the ***root***. [***O(1)***]
  2. **Insert a New Entry**:
     A single ***root-to-leaf path*** is affected. [***O(h)*** or ***O(log n)***]
  3. **Delete the Entry with Minimal Key**:
     A single ***root-to-leaf path*** is affected. [***O(h)*** or ***O(log n)***]
- After performing each operation,
  both ***relational*** and ***structural*** properties must be maintained.

Updating a Heap: Insertion

To insert a new entry (k, v) into a heap with ***height h***:

1. Insert (k, v), possibly **temporarily** breaking the ***relational property***.
   - **1.1** Create a new entry **e** $= (k, v)$.
   - **1.2** Create a new ***right-most*** node ***n*** at ***Level h***.
   - **1.3** Store entry **e** in node ***n***.

   After steps **1.1** and **1.2**, the ***structural property*** is maintained.
2. Restore the **heap-order property** (**HOP**) using *Up-Heap Bubbling* :
   - **2.1** Let ***c*** = ***n***.
   - **2.2** While **HOP** is not restored and ***c*** is not the root:
     - **2.2.1** Let ***p*** be ***c***'s parent.
     - **2.2.2** **If** *key*(***p***) $\leq$ *key*(***c***), then **HOP** is restored.
       Else, swap nodes ***c*** and ***p***. ["upwards" along ***n***'s ***ancestor path***]

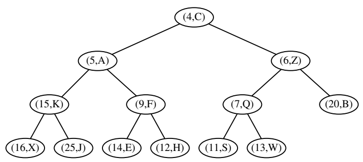
Running Time?

- All sub-steps in **1**, as well as steps **2.1**, **2.2.1**, and **2.2.2** take ***O(1)***.
- Step **2.2** may be executed up to ***O(h)*** (or ***O(log n)***) times.

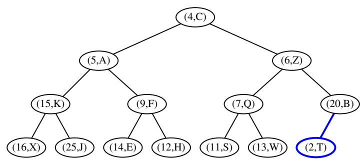
[*O(log n)*]

Updating a Heap: Insertion Example (1.1)

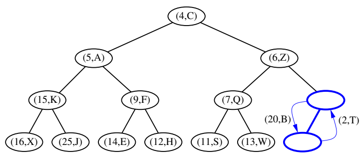
(0) A heap with height 3.



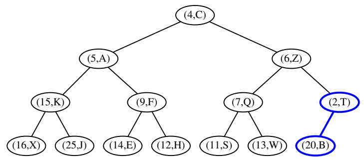
(1) Insert a new entry $(2, T)$ as the ***right-most*** node at Level 3. Perform *up-heap bubbling* from here.



(2) **HOP** violated ∵ 2 < 20 ∴ Swap.

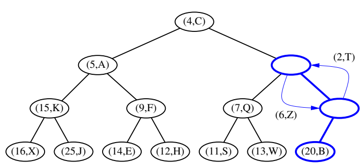


(3) After swap, entry $(2, T)$ prompted up.

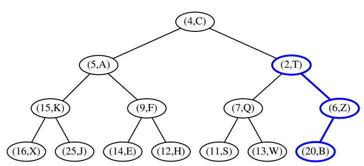


Updating a Heap: Insertion Example (1.2)

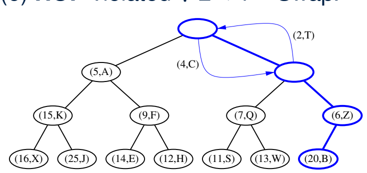
(4) **HOP** violated ∵ 2 < 6 ∴ Swap.



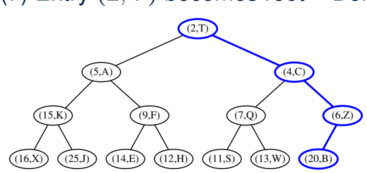
(5) After swap, entry $(2, T)$ prompted up.



(6) **HOP** violated ∵ 2 < 4 ∴ Swap.



(7) Entry $(2, T)$ becomes root ∴ Done.



Updating a Heap: Deletion

To delete the **root** (with the ***minimal*** key) from a heap with ***height h***:

1. Delete the **root**, possibly <u>**temporarily**</u> breaking **HOP**.
   - **1.1** Let the ***right-most*** node at ***Level h*** be ***n***.
   - **1.2** Replace the **root**'s entry by ***n***'s entry.
   - **1.3** Delete ***n***.

     After steps **1.1** – **1.3**, the ***structural property*** is maintained.
2. Restore **HOP** using *Down-Heap Bubbling*:
   - **2.1** Let ***p*** be the **root**.
   - **2.2** While **HOP** is not restored <u>and</u> ***p*** is <u>not</u> external:
     - **2.2.1** <u>**IF**</u> ***p*** has <u>no</u> **right child**, let ***c*** be ***p***'s ***left child***.

       <u>**Else**</u>, let ***c*** be ***p***'s child with a ***smaller key value***.
     - **2.2.2** <u>**If**</u> $key(p) \leq key(c)$, then **HOP** is restored.

       <u>**Else**</u>, swap nodes ***p*** and ***c***. ["downwards" along a ***root-to-leaf path***]

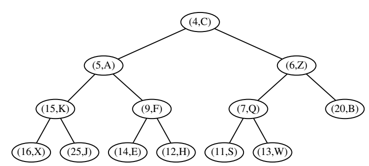
<u>Running Time?</u>

- All sub-steps in **1**, as well as steps **2.1**, **2.2.1**, and **2.2.2** take ***O(1)***.
- Step **2.2** may be executed up to ***O(h)*** (or ***O(log n)***) times.

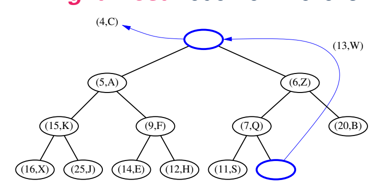
[*O(log n)*]

Updating a Heap: Deletion Example (1.1)

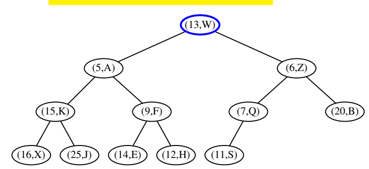
(0) Start with a heap with height 3.



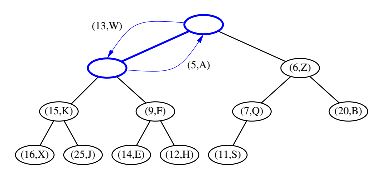
(1) Replace root with $(13, W)$ and delete ***right-most*** node from Level 3.



(2) $(13, W)$ becomes the root. Perform *down-heap bubbling* from here.

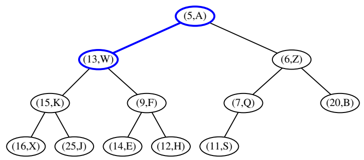


(3) Child with smaller key is $(5, A)$. **HOP** violated ∵ 13 > 5 ∴ Swap.

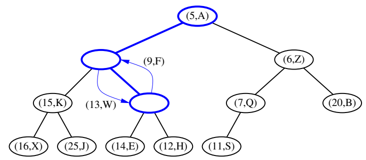


Updating a Heap: Deletion Example (1.2)

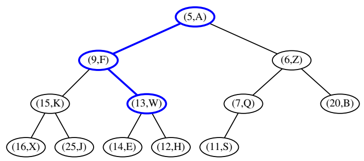
(4) After swap, entry $(13, W)$ demoted down.



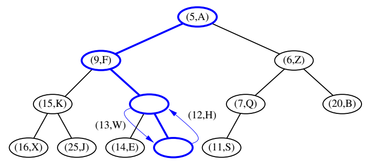
(5) Child with smaller key is $(9, F)$.
HOP violated $\because$ 13 > 9 $\therefore$ Swap.



(6) After swap, entry $(13, W)$ demoted down.

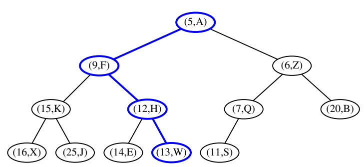


(7) Child with smaller key is $(12, H)$.
HOP violated $\because$ 13 > 12 $\therefore$ Swap.



Updating a Heap: Deletion Example (1.3)

(8) After swap, entry $(13, W)$ becomes an external node $\therefore$ Done.



Heap-Based Implementation of a PQ

PQ Method	Heap Operation	RT
min	root	*O(1)*
insert	insert then up-heap bubbling	*O(log n)*
removeMin	delete then down-heap bubbling	*O(log n)*

Index (1)

Index (2)