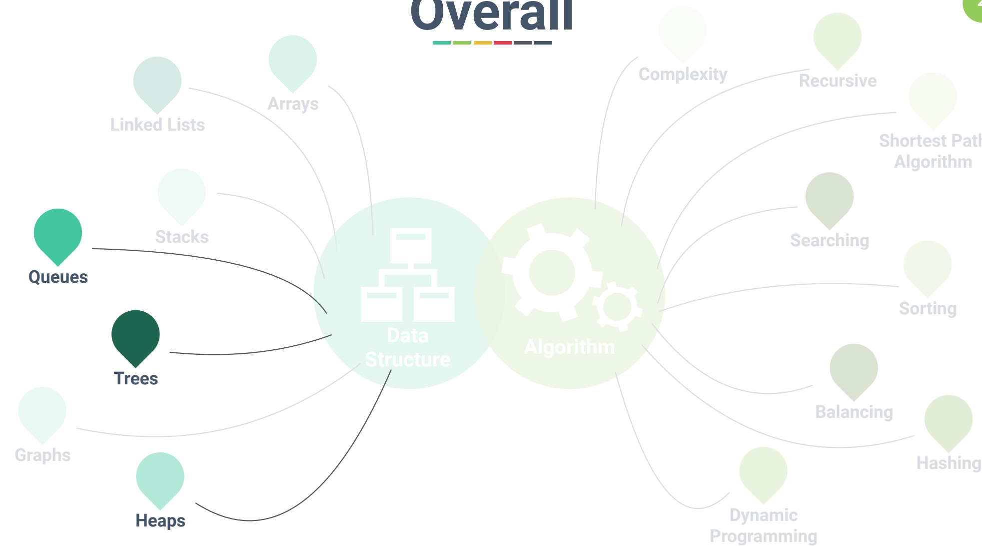
Chapter 7: Trees (Part 2)

(Priority Queues and Binary Heaps)

Dr. Sirasit Lochanachit

Overall



Outline

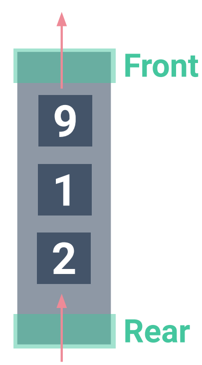
Priority Queues:

- Properties and methods
- Operation example

Binary Heaps:

- Definition, properties, examples
- Insertion and Deletion
- Consideration and Implementation in Array

Queues



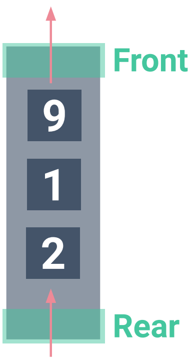
Queue is a collection, which keeps objects in a sequence, that are inserted and removed according to the **first-in first-out** (FIFO) principle [1].

A FIFO queue with priorities?

- Airline's waiting queue: First Class, Business Class, Economy Class

[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Queues

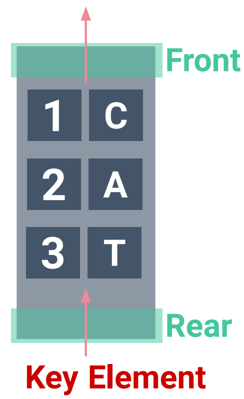


Queue is a collection, which keeps objects in a sequence, that are inserted and removed according to the **first-in first-out** (FIFO) principle [1].

Another variation is a **priority queue**. It stores prioritised elements that allows arbitrary element insertion and allows the deletion of the element that has first priority [1].

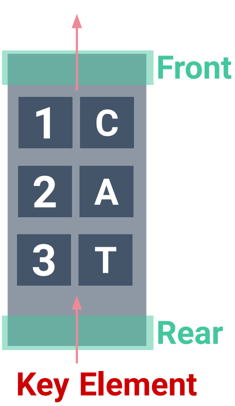
[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Priority Queues Properties



- When a new element is added in the queue, its priority is determined by an associated **key**.
- The order of elements is determined by their priority.
- The highest priority items are at the front of the queue.
- The lowest priority elements are at the back.
- A new item that is enqueued could be moved to the front of the queue.

Priority Queue Methods



Formally, there are two main operations for priority queues P :

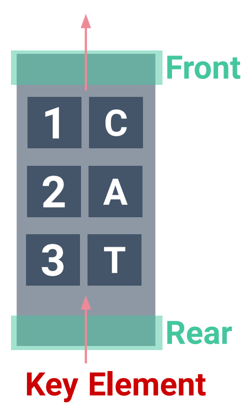
1) P.add(*key, value*)

Insert an element with *key* and *value* into the queue.

2) P.remove_min()

Remove and return the *key* and *value* that have the lowest priority; return error if the queue is empty.

Priority Queue Methods



Additional operations for priority queues P :

1) **P.min()**

   Return (but do not remove) the key and value that have the lowest priority; return error if the queue is empty.

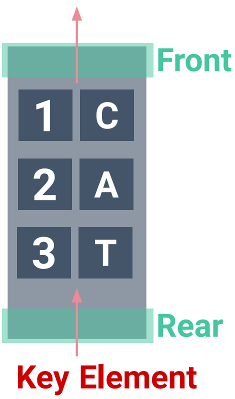
2) **P.is_empty()**

   Check whether a priority queue is empty.

3) **len(P)**

   Return the number of elements in a priority queue P.

Priority Queue



Before implementing a priority queue, the following assumptions should be considered:

- Keys of multiple elements can be either equal or unique.
- Once a key is assigned to the element and it has been added to a priority queue, it should be either fixed or adjustable.

Operation Example

Operation	Return Value	Priority Queue [first, ..., last]		
P.add(5, A)	-	(5, A)		
P.add(9, C)	-	(5, A)	(9, C)	
P.min()	(5, A)	(5, A)	(9, C)	
P.add(2, B)	-	(2, B)	(5, A)	(9, C)
P.remove_min()	(2, B)	(5, A)	(9, C)	
P.is_empty()	False	(5, A)	(9, C)	
len(P)	2	(5, A)	(9, C)	

Operation Exercise

Operation	Return Value	Priority Queue [first, ..., last]
P.add(5, A)	-	
P.add(4, B)	-	
P.remove_min()		
P.add(7, F)	-	
P.add(3, J)		
P.remove_min()		
P.remove_min()		

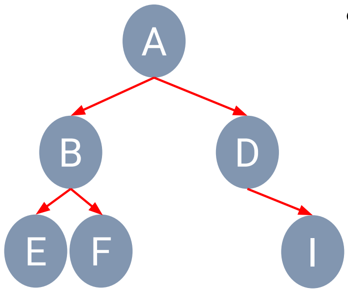
Asymptotic Performance

n denotes the number of elements or nodes.

Operation	Running Time (unsorted list)	Running Time (sorted list)
P.add(k, v)	$O(1)$	**O(n)**
P.remove_min()	$O(n)$	$O(1)$
P.min()	$O(n)$	$O(1)$
P.is_empty()	$O(1)$	$O(1)$
len(P)	$O(1)$	$O(1)$

A solution that is more efficient?

Binary Heap

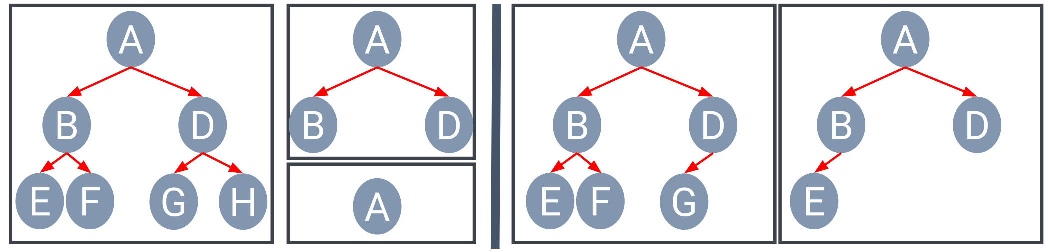


- A **heap** is a binary tree that stores elements and has the following properties:
  - For every node except the root, the key stored in a node is greater than or equal to the key stored at a node's parent - **min heap**.
  - A minimum key is located at the root node.
  - A heap has to be a **complete** or **nearly complete** binary tree.

[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Types of Binary Trees

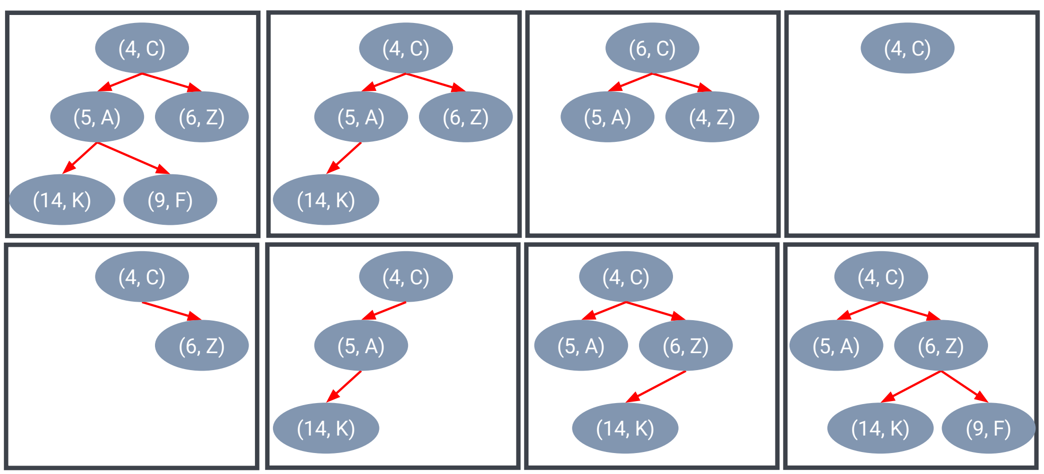
- A binary tree is nearly complete when every level, except the last, is completely filled and all leaf nodes are as far left as possible.



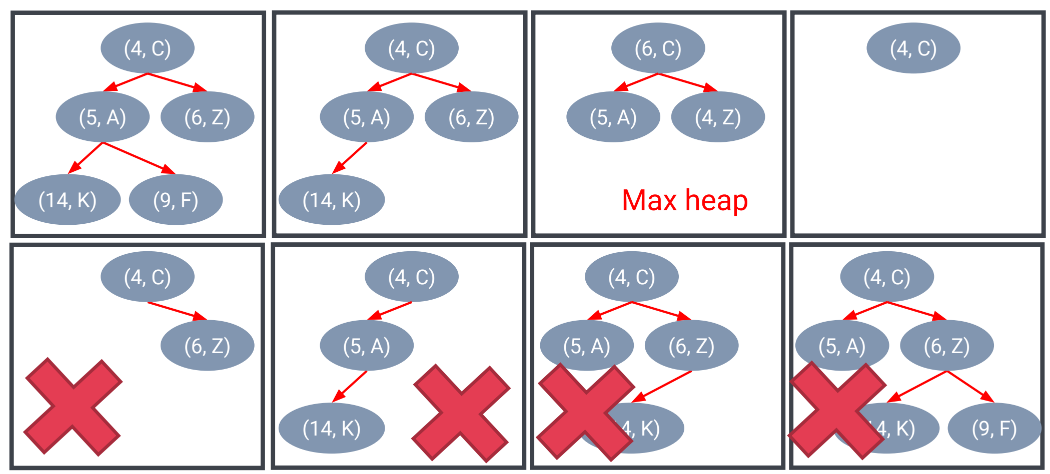
(a) *Complete/Perfect* Binary Tree

(b) *Nearly Complete* Binary Tree at level 2

Binary Heap



Binary Heap



Binary Heap

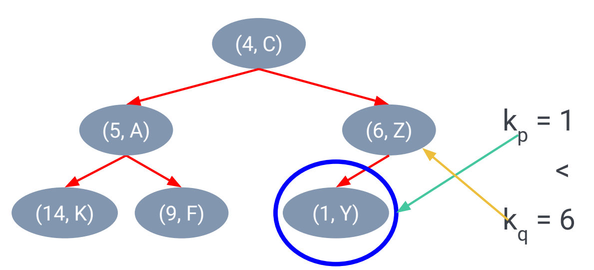
- To allow a binary heap to have a logarithmic performance when performing insertion and deletion, a tree needs to be balanced or **nearly complete**.
- Insertions and removals in logarithmic time is a significant improvement over the list-based implementations.
- Can be implemented using a linked list of an array.

Binary Heap Insertion

- To preserve the nearly complete property, the new node should be inserted at a position p, after the rightmost node at the bottom level, or as the leftmost node of a new level, if the bottom level is full.
- However, the **heap order property** may be violated where the key p is less than key of p's parent, which is denoted as q ($k_p < k_q$).
  - The heap order needs to be restored by swapping key-value pairs, moving the new item up one level.
  - This is done until there is no violation of the heap order property.

Binary Heap Insertion

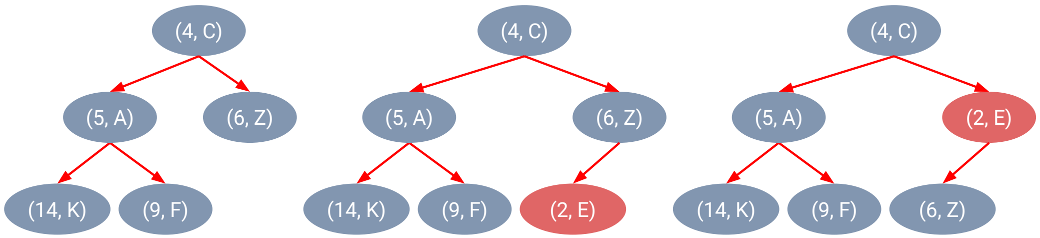
Example of when $k_p < k_q$ after inserting a new node (1, Y).



Binary Heap Insertion

- If key p is greater than key q, then the heap order is preserved ($k_p >= k_q$).
- The upward movement of the newly inserted node through swapping is **up heap bubbling** or **reheap up**.
- The worst case scenario of upheap is moving the new item all the way to the root of heap.
  - The number of swaps executed in method add() is equal to the **height of heap** or **floor of log n** (i.e. $\lfloor \log n \rfloor$).

Upheap

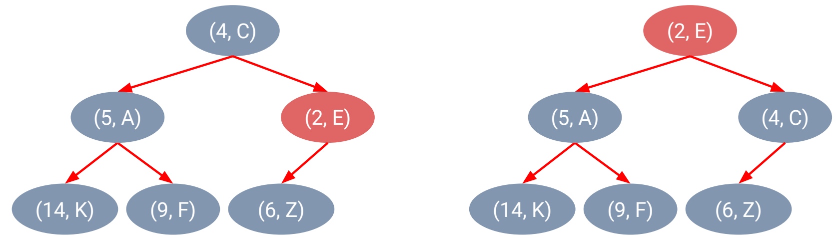


Current Binary Heap

Adding new node
(key, value)

Up heap #1

Upheap



Up heap #1
(cont. from previous slide)

Up heap #2

Binary Heap Removal

- Due to the property of min heap, the item with the **smallest key** is stored at the **root node**.
- However, removing the root node directly would leave two disconnected subtrees.
- To ensure that the heap remains as the nearly complete binary tree, the leaf node at the rightmost position of the bottom level of the tree is deleted.
  - Before removing, the item in the last position is copied to the root node.

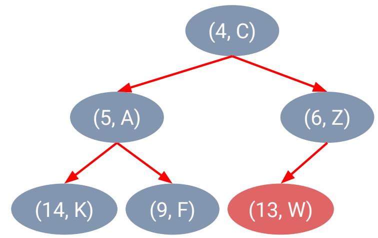
Binary Heap Removal

- However, the **heap order property** may be violated where the key p is greater than key of p's children, which is denoted as q ($k_p > k_c$).
  - The heap order needs to be restored by swapping key-value pairs, moving the new item down one level.
  - When p has two children, the smaller key determines the direction of moving down.
  - This is done until there is no violation of the heap order property.

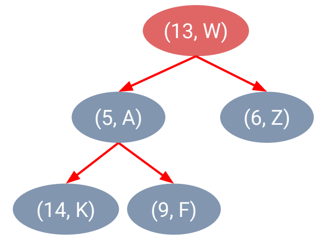
Binary Heap Removal

- If key p is less than key c, then the heap order is preserved (k_p <= k_c).
- The downward movement of the new root node through swapping is **down heap bubbling** or **reheap down**.
- The worst case scenario of downheap is moving the root item all the way to the bottom level of the tree.
  - The number of swaps executed in method remove_min() is equal to the **height of heap** or **floor of log n** (i.e. ⌊log n⌋).

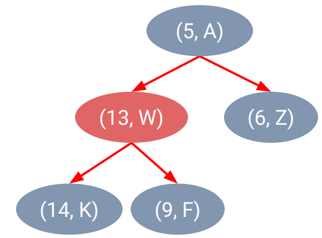
Downheap



Current Binary Heap

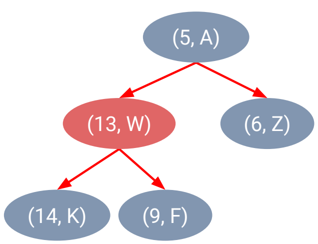


Remove a node with smallest key & copy a rightmost leafnode to the root

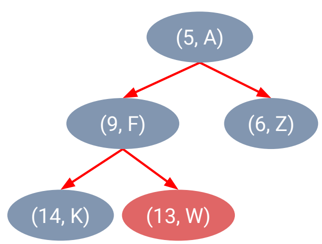


Down heap #1

Downheap



Down heap #1
(cont. from previous slide)



Down heap #2

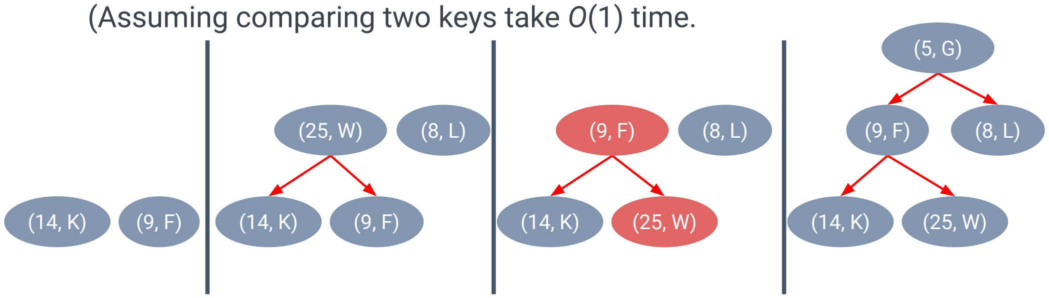
Asymptotic Performance

n denotes the number of elements or nodes.
The height of heap is **log n**.

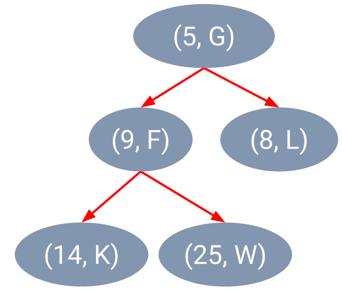
Operation	Running Time (unsorted list)	Running Time (sorted list)	Running Time (Binary Heap)
P.add(k, v)	$O(1)$	$O(n)$	**$O(\log n)$**
P.remove_min()	$O(n)$	$O(1)$	**$O(\log n)$**
P.min()	$O(n)$	$O(1)$	$O(1)$
P.is_empty()	$O(1)$	$O(1)$	$O(1)$
len(P)	$O(1)$	$O(1)$	$O(1)$

Binary Heap Consideration

- If start with an empty heap, calling add operations *n* successive times will run in ***O*(n log n)** time - top-down approach.
- Alternatively, bottom-up construction method runs in ***O*(n)** time (Assuming comparing two keys take *O*(1) time.

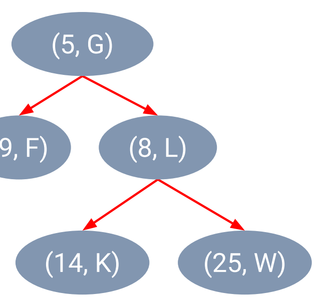


Binary Heap Implementation in Array



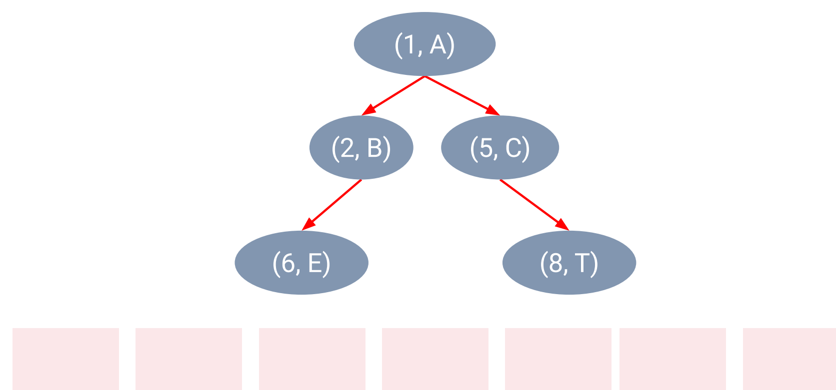
(5, G)	**(9, F)**	**(8, L)**	**(14, K)**	**(25, W)**

Binary Heap Implementation in Array



(5, G)	**(9, F)**	**(8, L)**			**(14, K)**	**(25, W)**

Exercise



Individual Assignment

- Assignment#5: Trees
- Due 09.00 am, Tuesday 15/09/2020.
- Submission
  - Email: sirasit@it.kmitl.ac.th (PDF format preferred)
  - Paper: in classroom next week
- Can be either written by hand or typing.
- Make sure to submit on time!!
  - Late submission has penalty on the score.
- If unable to submit on time for reasonable reasons, let me know asap.