Total Order & Comparator

Total Order
\( \leq \), defined on every pair of elements, such that

- **Reflexive**: \( k \leq k \).
- **Anti-symmetric**: \( k_1 \leq k_2 \) and \( k_2 \leq k_1 \Rightarrow k_1 = k_2 \).
- **Transitive**: \( k_1 \leq k_2 \) and \( k_2 \leq k_3 \Rightarrow k_1 \leq k_3 \).
Total Order & Comparator

Total Order
≤, defined on every pair of elements, such that

- Reflexive: \( k \leq k \).
- Anti-symmetric: \( k_1 \leq k_2 \) and \( k_2 \leq k_1 \) \( \Rightarrow k_1 = k_2 \).
- Transitive: \( k_1 \leq k_2 \) and \( k_2 \leq k_3 \) \( \Rightarrow k_1 \leq k_3 \).

Comparators
A comparator is an object that defines a total order on elements in the following way:

- isLess\((a,b)\), isLessOrEqualTo\((a,b)\)
- isEqualTo\((a,b)\)
- isGreater\((a,b)\), isGreaterOrEqualTo\((a,b)\)
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by keys. Each element is associated with a key.
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by **keys**. Each element is associated with a **key**.

- **insert**\((k, e)\): insert an element \(e\) with key \(k\) into PQ.
- **removeMin**(): Return and remove from PQ an element with the **smallest** key. **min-priority queue**.
- **removeMax**(): Return and remove from PQ an element with the **largest** key. **max-priority queue**.
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by keys. Each element is associated with a key.

- insert\((k, e)\): insert an element \(e\) with key \(k\) into PQ.
- removeMin(): Return and remove from PQ an element with the smallest key. min-priority queue.
- removeMax(): Return and remove from PQ an element with the largest key. max-priority queue.

Simple Implementation on top of Queues

- Store Keys.
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by keys. Each element is associated with a key.

▶ insert\((k, e)\): insert an element \(e\) with key \(k\) into PQ.
▶ removeMin(): Return and remove from PQ an element with the smallest key. min-priority queue.
▶ removeMax(): Return and remove from PQ an element with the largest key. max-priority queue.

Simple Implementation on top of Queues

▶ Store Keys.
▶ insert\((k, e)\): \(O(1)\), removeMin()\(]/\)removeMax(): \(O(n)\). How?
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by keys. Each element is associated with a key.

- insert($k, e$): insert an element $e$ with key $k$ into PQ.
- removeMin(): Return and remove from PQ an element with the smallest key. min-priority queue.
- removeMax(): Return and remove from PQ an element with the largest key. max-priority queue.

Simple Implementation on top of Queues

- Store Keys.
- insert($k, e$): $O(1)$, removeMin()/removeMax(): $O(n)$. How?
- insert($k, e$): $O(n)$, removeMin()/removeMax(): $O(1)$. How?
Algorithm PQ-sort($C, P$)
Input: an $n$-element sequence $C$, a priority queue $P$.
Output: the sequence $C$ sorted by the total order relation.

while ! $C$.isEmpty() do
  $e \leftarrow C$.removeFirst()
  $P$.insert($e, e$).
end while

while ! $P$.isEmpty() do
  $e \leftarrow P$.removeMin(). $P$.removeMax()
  $C$.insertLast($e$).
end while
PQ-based Sorting

**Algorithm** PQ-sort(\(C, P\))

- **Input:** an \(n\)-element sequence \(C\), a priority queue \(P\).
- **Output:** the sequence \(C\) sorted by the total order relation.

```plaintext
while ! C.isEmpty() do
    e ← C.removeFirst()
    P.insert(e, e).
end while

while ! P.isEmpty() do
    e ← P.removeMin(). P.removeMax()
    C.insertLast(e).
end while
```

**Correctness?**
PQ-based Sorting: Simple Implementation

- $\text{insert}(k, e)$: $O(1)$, $\text{removeMin}()$: $O(n)$. Total running time $O(n^2)$. Also known as "selection-sort".

Improvement on efficiency?

- $\text{insert}(k, e)$: $O(\log n)$, $\text{removeMin}()$: $O(\log n)$. Total running time $O(n \log n)$. Also known as "heap-sort".

Optimal running time? Yes for comparison-based sorting.

Week 10!
PQ-based Sorting: Simple Implementation

- \text{insert}(k, e): O(1), \text{removeMin}(): O(n).
  Total running time \( O(n^2) \). Also known as "selection-sort".

- \text{insert}(k, e): O(n), \text{removeMin}(): O(1).
  Total running time \( O(n^2) \). Also known as "insertion-sort".
PQ-based Sorting: Simple Implementation

- insert($k, e$): $O(1)$, removeMin(): $O(n)$.
  Total running time $O(n^2)$. Also known as "selection-sort".
- insert($k, e$): $O(n)$, removeMin(): $O(1)$.
  Total running time $O(n^2)$. Also known as "insertion-sort".

Improvement on efficiency?

- insert($k, e$): $O(\log n)$, removeMin(): $O(\log n)$.
  Total running time $O(n \log n)$. Also known as "heap-sort".
PQ-based Sorting: Simple Implementation

- `insert(k, e): O(1), removeMin(): O(n).`
  Total running time `O(n^2)`. Also known as "selection-sort".
- `insert(k, e): O(n), removeMin(): O(1).`
  Total running time `O(n^2)`. Also known as "insertion-sort".

Improvement on efficiency?

- `insert(k, e): O(log n), removeMin(): O(log n).`
  Total running time `O(n log n)`. Also known as "heap-sort".
- Optimal running time? Yes for comparison-based sorting.
  Week 10!
How to achieve $O(\log n)$ for both insertion and removal?

Heap

Instead of storing elements in sequences, store in complete binary trees satisfying the Heap-Order Property.
How to achieve $O(\log n)$ for both insertion and removal?

Heap

Instead of storing elements in sequences, store in complete binary trees satisfying the **Heap-Order Property**.

- **Heap-Order Property:**
  - **(min-heap)** for every node $v$ other than the root, its key $\geq$ the key of its parent.
  - **(max-heap)** for every node $v$ other than the root, its key $\leq$ the key of its parent.
How to achieve $O(\log n)$ for both insertion and removal?

Heap

Instead of storing elements in sequences, store in complete binary trees satisfying the **Heap-Order Property**.

- **Heap-Order Property:**
  
  - *(min-heap)* for every node $v$ other than the root, its key $\geq$ the key of its parent.
  
  - *(max-heap)* for every node $v$ other than the root, its key $\leq$ the key of its parent.

- **Complete Binary Trees:** binary tree with height $h$ and maximum number of nodes in all levels $0, \cdot \cdot \cdot, h - 1$. In level $h - 1$, the internal nodes are to the left of the external nodes.
How to achieve $O(\log n)$ for both insertion and removal?

Heap

Instead of storing elements in sequences, store in complete binary trees satisfying the **Heap-Order Property**.

- **Heap-Order Property:**
  - *(min-heap)* for every node $v$ other than the root, its key $\geq$ the key of its parent.
  - *(max-heap)* for every node $v$ other than the root, its key $\leq$ the key of its parent.

- **Complete Binary Trees:** binary tree with height $h$ and maximum number of nodes in all levels $0, \cdots, h-1$. In level $h-1$, the internal nodes are to the left of the external nodes.

- **Last Node:** as the rightmost internal node on level $h-1$. 
Heap Example: only keys
Vector-based Implementation

Binary-Tree

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$. 

Application to heaps

- The last node of a heap of $n$ keys is indexed $n$ in the array.
- The first node of a heap is indexed 1 in the array.
Vector-based Implementation

Binary-Tree

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$. 

Application to heaps

- The last node of a heap of $n$ keys is indexed $n$ in the array.
- The first node of a heap is indexed 1 in the array.
Vector-based Implementation

Binary-Tree

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$.

Application to heaps

- The last node of a heap of $n$ keys is indexed $n$ in the array.
- The first node of a heap is indexed 1 in the array.
Heap Example: only keys

The image shows a binary heap with the following keys:

- Root node: 4
- Left child of root: 5
  - Left child of 5: 15
    - Left child of 15: 16
    - Right child of 15: 25
  - Right child of 5: 9
    - Left child of 9: 14
    - Right child of 9: 12
    - Right child of 9: 11
    - Right child of 9: 8
- Right child of root: 6
  - Left child of 6: 20
- Right child of 15: 7
  - Left child of 7: 11
  - Right child of 7: 8
Heap Example: only keys

[4, 5, 6, 15, 9, 7, 20, 16, 25, 14, 12, 11, 8]
Heap Property

Theorem (Exercise 6.1-2)

A heap $T$ storing $n$ keys has height $h = \Theta(\log(n))$.
Heap Property

Theorem (Exercise 6.1-2)

A heap $T$ storing $n$ keys has height $h = \Theta(\log(n))$

Proof.

\[
2^0 + \cdots + 2^{h-1} \leq n \leq 2^0 + \cdots + 2^h
\]
Heap Property

Theorem (Exercise 6.1-2)
A heap $T$ storing $n$ keys has height $h = \Theta(\log(n))$

Proof.

$2^0 + \cdots + 2^{h-1} \leq n \leq 2^0 + \cdots + 2^h$

$2^h - 1 \leq n \leq 2^{h+1} - 1$.

$h = \Theta(\log(n))$. 

Remark: if updates $\sim$ height $h$, then $\Theta(\log(n))$. 
Heap Property

**Theorem (Exercise 6.1-2)**

A heap $T$ storing $n$ keys has height $h = \Theta(\log(n))$

**Proof.**

\[2^0 + \cdots + 2^{h-1} \leq n \leq 2^0 + \cdots + 2^h\]

\[2^h - 1 \leq n \leq 2^{h+1} - 1.\]

\[h = \Theta(\log(n)).\]

**Remark:** if updates $\sim$ height $h$, then $\Theta(\log(n))$. 
Insertion

Goals

- Maintain properties of heap.
- Cost $\sim$ the height of the heap. i.e., $\Theta(h) = \Theta(\log(n))$. 
Insertion

Goals

- Maintain properties of heap.
- Cost ∼ the height of the heap. i.e., $\Theta(h) = \Theta(\log(n))$.
- Heap-Order Property:
  (min-heap) for every node $v$ other than the root, its key $\geq$ the key of its parent.
  (max-heap) for every node $v$ other than the root, its key $\leq$ the key of its parent.
Insertion

Goals

- Maintain properties of heap.
- Cost $\sim$ the height of the heap. i.e., $\Theta(h) = \Theta(\log(n))$.

- Heap-Order Property:
  - \textit{(min-heap)} for every node $v$ other than the root, its key $\geq$ the key of its parent.
  - \textit{(max-heap)} for every node $v$ other than the root, its key $\leq$ the key of its parent.

- Complete Binary Trees: binary tree with height $h$ and maximum number of nodes in all levels $0, \ldots, h - 1$. In level $h - 1$, the internal nodes are to the left of the external nodes.