Outline

1. Priority Queue
2. Heap
3. Heap Sort
Priority Queue is a data structure maintaining a set of elements, each element has an index and a key value.

One only can remove the first element and insert a new element to the end.
Priority Queue

- **Extract**: remove the first element.
- **Insert**: Add a new element to the end.

```
Index: 1 2 3 4 5 6 7
key value: 2 1 9 7 9 0 5
```
Priority Queue

- **Extract**: remove the first element.
- **Insert**: Add a new element to the end.

```
Index: 1 2 3 4 5 6 7
key value: 2 1 9 7 9 0 5
```
Priority Queue

- **Extract**: remove the first element.
- **Insert**: Add a new element to the end.

Index: 1 2 3 4 5 6

Key value: 1 9 7 9 0 5
Priority Queue

- **Extract**: remove the first element.
- **Insert**: Add a new element to the end.

Index: 1 2 3 4 5 6 7

**key value**: 1 9 7 9 0 5 6
Outline

1. Priority Queue
2. Heap
3. Heap Sort
Heap is a special priority queue:


Heap can be visualized by a **nearly complete binary tree**.

```
25
/   \
22   16
/   /   \
20   17  10  15
/ \      / \      / \
3   12  13
```
Build A Heap

**BuildMaxHeap(A)**

1: **For** $i = n/2$ to 1  
2: MaxHeapify(A, i)  
3: **End For**

**MaxHeapify(A, i)**

1: **if** $A[i] \geq \max\{A[2i], A[2i + 1]\}$ **then**  
2: Do nothing  
3: **else**  
4: **if** $A[2i] = \max\{A[2i], A[2i + 1]\}$ **then**  
5: Swap the key values of $A[i]$ and $A[2i]$  
6: MaxHeapify(A, 2i)  
7: **else**  
8: Swap the key values of $A[i]$ and $A[2i + 1]$  
9: MaxHeapify(A, 2i+1)  
10: **end if**  
11: **end if**
Build A Heap: Complexity

- **MaxHeapify(A, i):** $O(\text{height}(A[i])) = O(\log n)$, \( \text{height}(A[i]) \) is the height of \( A[i] \) in the corresponding visualizing tree.

- **BuildMaxHeap(A):** $O(n \log n)$, but actually is \( \theta(n) \) by a more careful analysis.
Outline

1. Priority Queue
2. Heap
3. Heap Sort
Heap Sort

- **Basic idea:**
  1. Build a MaxHeap.
  2. Repeat the following steps $n$ times:
     1. Swap the key values of the first and final elements.
     2. Reduce the size of the heap by one.
     3. Run MaxHeapify(A, 1).

- **Complexity:** $\theta(n \log n)$ time and $\theta(1)$ space.
Comparison of Three Sorting Algorithms

- **Insertion Sort**: $\theta(n^2)$ time in worst case and $\theta(n)$ time in best case, $\theta(1)$ extra space.

- **Merge Sort**: $\theta(n \log n)$ time and $\theta(n)$ extra space; easy to be parallel.

- **Heap Sort**: $\theta(n \log n)$ time and $\theta(1)$ extra space.