Data Structure

- Array
- Linked-List
- Stack
- Queue

void enqueue(Q, x)

if (rear == Q.len-1)
Overflow
Q[++rear] = x;

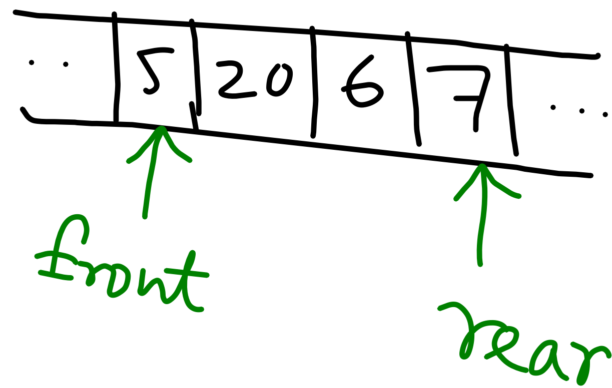
rear ← rear+1
Q[rear] ← x

int dequeue(Q)

if (rear == front)
underflow
return Q[front++]

Queue

(FIFO)
↳ First-in-First-Out



initialization

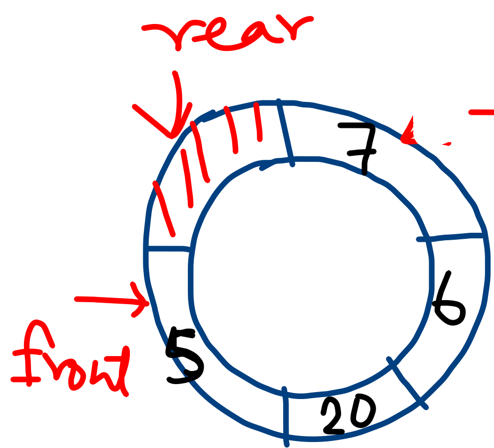
front = -1
rear = -1

Circular Queue

front = -1, rear = -1 } X

{ front = 0, rear = 0 }

- Wrapping up



→ Have to Sacrifice One place → o/w underflow ≡ overflow

enqueue(Q, x)

if ((rear+1) mod n == front) // overflow condition

rear ← (rear + 1) mod len

Q[rear] ← x;

int dequeue(Q)

if ((rear+1) mod n == front) // underflow cond

front' = front

front ← (front + 1) mod len

return Q[front']

enqueue(Q, x)

if ((rear+1) mod N == front) Overflow

Q[rear] = x;

rear = (rear+1) mod N

deque(x)

if (front == rear) underflow

Priority Queue

↳ Elements will be deleted based on priority

- Scheduling Algo
- Prim's Algo (MST)

Insert (S, x)

Max (S)

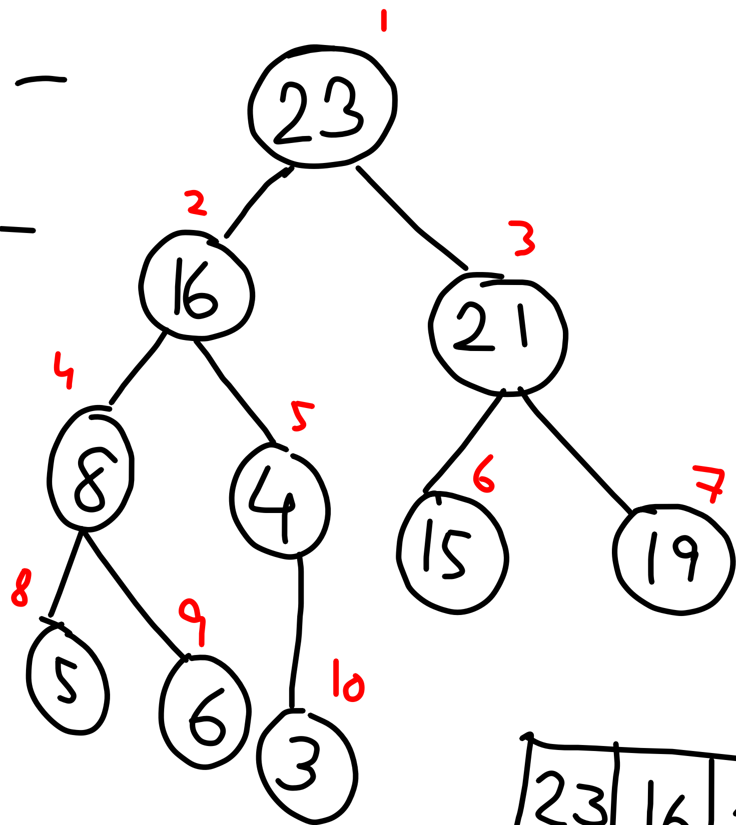
Extract_Max (S) → Report max & remove that element

Increase_Key (S, i, key) → increase the value of ith element to key.

Heap

(binary) heap

$$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$
$$\text{left_child}(i) = 2i$$
$$\text{right_child}(i) = 2i+1$$



23	16	21	8	4	15	19	5	6	3

- (near) Complete binary tree
  - All levels except last one is complete
  - Last level: filled up from left
- max heap: $\forall i,\ A[\text{parent}(i)] \geq A[i]$
- min_heap: $\forall i,\ A[\text{parent}(i)] \leq A[i]$

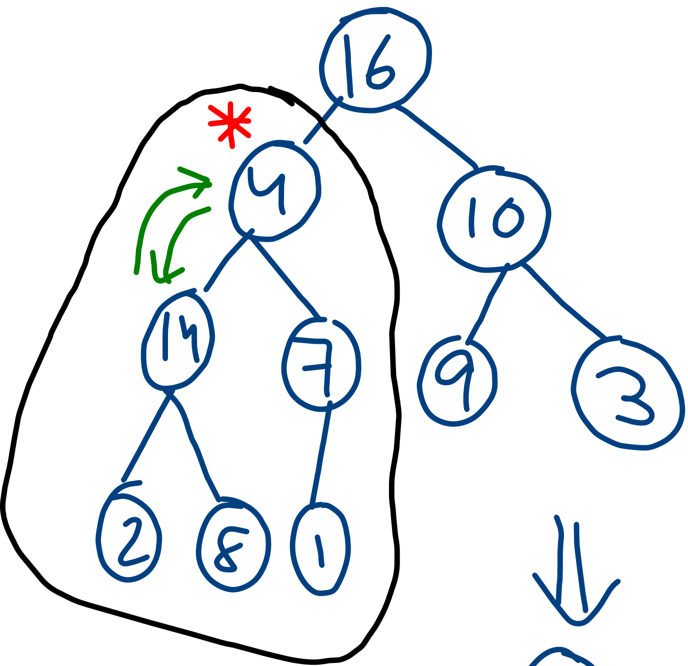
Opearations on max_heap

① Heapify (A, i) :

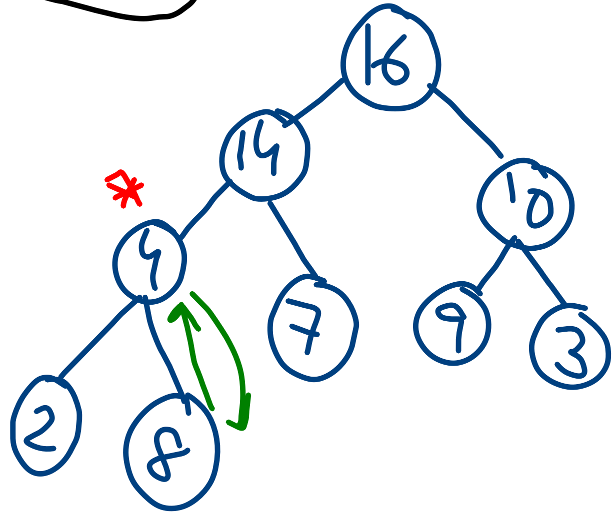
- heap A.
- A[i] is not maintaining the heap property
- A[left(i)], A[right(i)] are max heaps

⇓

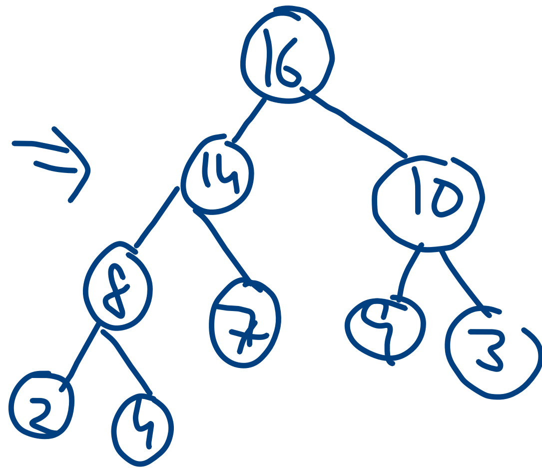
max-heap



⇓



⇒



Max_heapify(A, i)

```
l ← Left(i)
r ← right(i)
if (l ≤ A.heapsize && A[l] > A[i])
        largest = l
else
        largest = i
if (r ≤ A.heapsize && A[r] > A[largest])
        largest = r
if (largest ≠ i)
        Swap(A[i], A[largest])
        Max_heapify(A, largest)
```

Time Complexity

$$T(n) = T(2n/3) + c$$

$$T(n) = O(\lg n)$$

② Build_Maxheap(A)

For $i = \lfloor n/2 \rfloor$ (-1) 1

Max_heapify(A, i)

Loop Invariant:

"At i^{th} iteration, each node $(i+1), (i+2), \ldots n$ is the root of a max_heap".

Time Complexity: $O(n \lg n)$

- n element heap has $\lfloor \log n \rfloor$ levels.
- At level h at most $\lceil n/2^{h+1} \rceil$ nodes.

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h)$$

$$= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

$$= O(n)$$

③ Heap Sort (A)

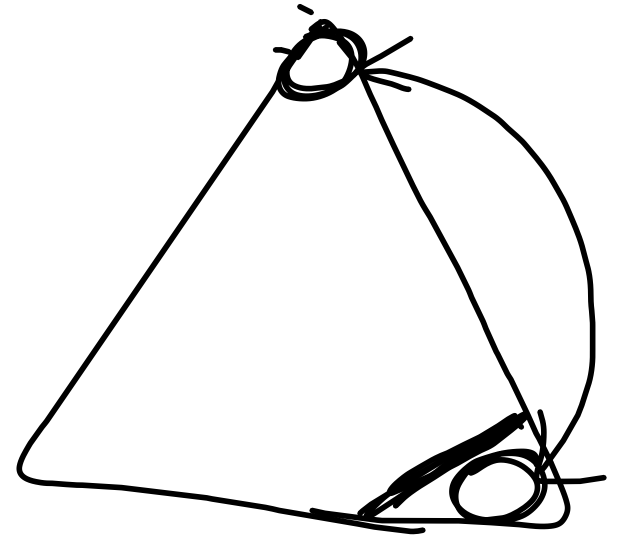
Build_Maxheap (A)

For = i = A.len (-1) 2

Swap(A[1], A[i])

A.heapsize = A.heapsize - 1

Max_heapify (A, 1)



In-place

Time Complexity

O(n log n)

Loop Invariant

A[1..i] → smallest i elements

A[(i+1)..n] → height (n-i) elements in sorted order

Priority Queue using heap

Max(A) → return A[1] — $O(1)$

Extract_max (A)

max ← A[1]

A[1] ← A[A.heapsize] — $O(\log n)$

A.heapsize = A.heapsize - 1

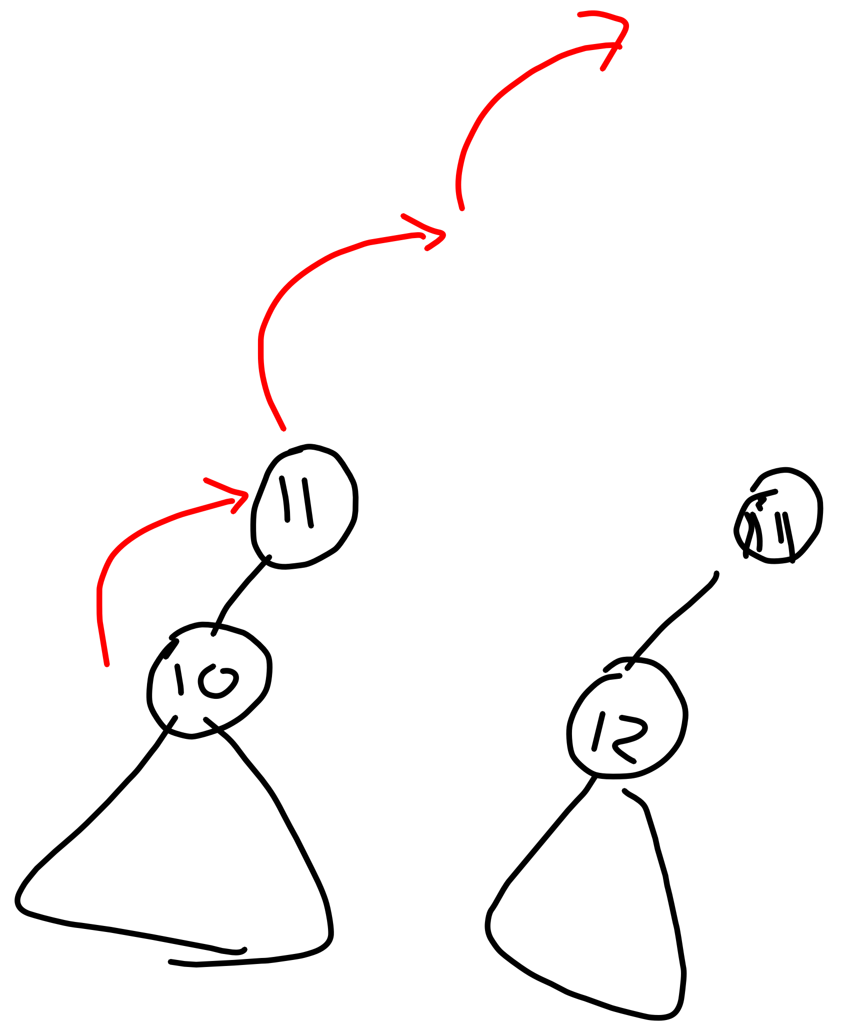
Max_heapify (A, 1)

return max

Increase_Key (A, i, key)

```
A[i] = key
while (A[Parent(i)] < A[i])
    Swap(A[i], A[parent(i)])
    i = Parent(i)
```

→ $O(\log n)$



Insert (A, key)

```
A.heapsize = A.heapsize + 1
A[A.heapsize] = -∞
IncreaseKey(A, A.heapsize, key)
```

→ $O(\log n)$