CSE 373/L-8/03.03.2024/

Chapter-6

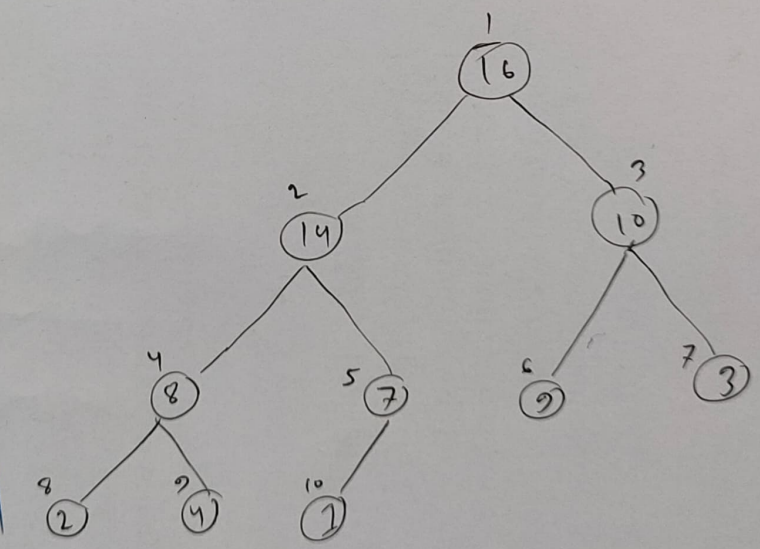
Heapsort

- Combine the advantages of both merge sort and insertion sort.
  - $O(n \lg n)$ worst-case running time
  - sorts in place
  - when array nearly sorted, runs fast in practice

- Heap is combined of a binary tree and an array

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

```
PARENT (i)
    return ⌊i/2⌋
LEFT (i)
    return 2i
RIGHT (i)
    return 2i+1
```



- Always fill from the left.

⇒ In a max heap, parent will be larger or equal to the child

parent ≥ child

⇒ In min heap

parent ≤ child

* Height of a node
  ⇒ ~~long~~ longest simple downward path from node to a leaf

* Height of the heap/root ⇒ $\Theta(\lg n)$

* List of function:

MAX-HEAPZFY ⇒ $O(\lg n)$

BUZLD-MAX-HEAP ⇒ $O(n)$

HEAPSORT ⇒ $O(n \lg n)$

MAX-HEAP-ZNSERT ⇒ $O(\lg n)$

HEAP-EXTRACT-MAX ⇒ $O(\lg n)$

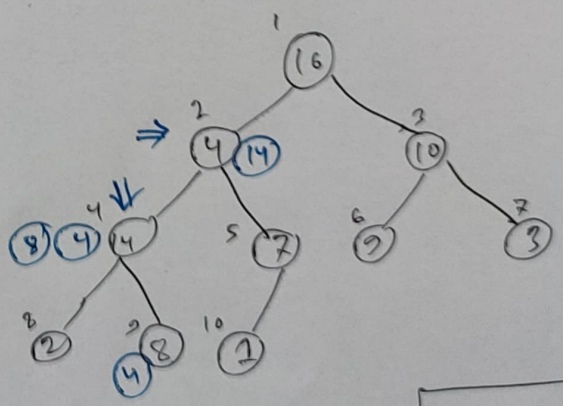
HEAP-ZNCREASE-KEY ⇒ $O(\lg n)$

HEAP-MAXZMUM ⇒ $O(\lg n)$

* MAX-HEAPZFY (A, i)

```
l = LEFT(i)
r = RZGHT(i)        → l exist or not
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else
    largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPZFY (A, largest)
```

$\Theta(1)$



MAX-HEAPZFY (A, 2)
MAX-HEAPZFY (A, 4)

→ can be called up to $\frac{2n}{3}$ times
→ maximum number of children

Run time,
$T(n) = T(\frac{2n}{3}) + \Theta(1)$
→ only one side will be call & divided int $\frac{2n}{3}$
$= O(\lg n)$

Building a Heap

⇒ BUILD-MAX-HEAP (A, n)

A. heap-size = n

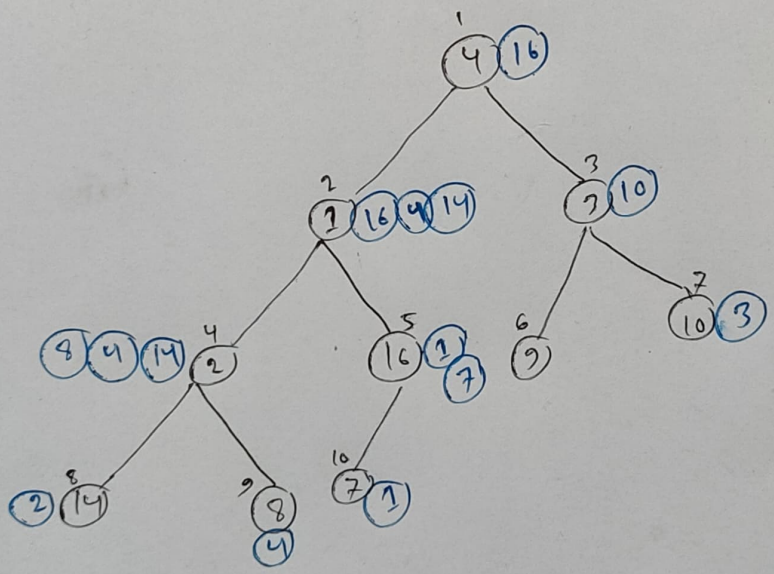
for i = $\lfloor n/2 \rfloor$ downto 1

$\lfloor \frac{n}{2} \rfloor = \frac{10}{2} = 5 \Rightarrow 5+1 = 6$ to 10 all leaf

= already a heap, no need to modify

MAX-HEAPIFY (A, i)

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7
16	14	10	8	7	9	3	2	4	1



MAX-HEAPIFY (A,5)

4

3

2

1

Analysis 1:

MAX-HEAPIFY cost $O(\lg n)$

need to call $O(n)$

∴ run time = $O(n \lg n)$

Analysis 2:

for n element, height is $\lfloor \lg n \rfloor$

nodes, in max $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

Time required for MAX-HEAPIFY = $O(h)$

Then run time = $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$

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

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h+1}}\right)$$

$$= O\left(n \cdot \frac{1/2}{(1-1/2)^2}\right)$$

$$= O(n \cdot 2)$$

$$= O(n)$$

HEAPSORT (A, n)

BUILD-MAX-HEAP (A, n) $\Longrightarrow O(n)$

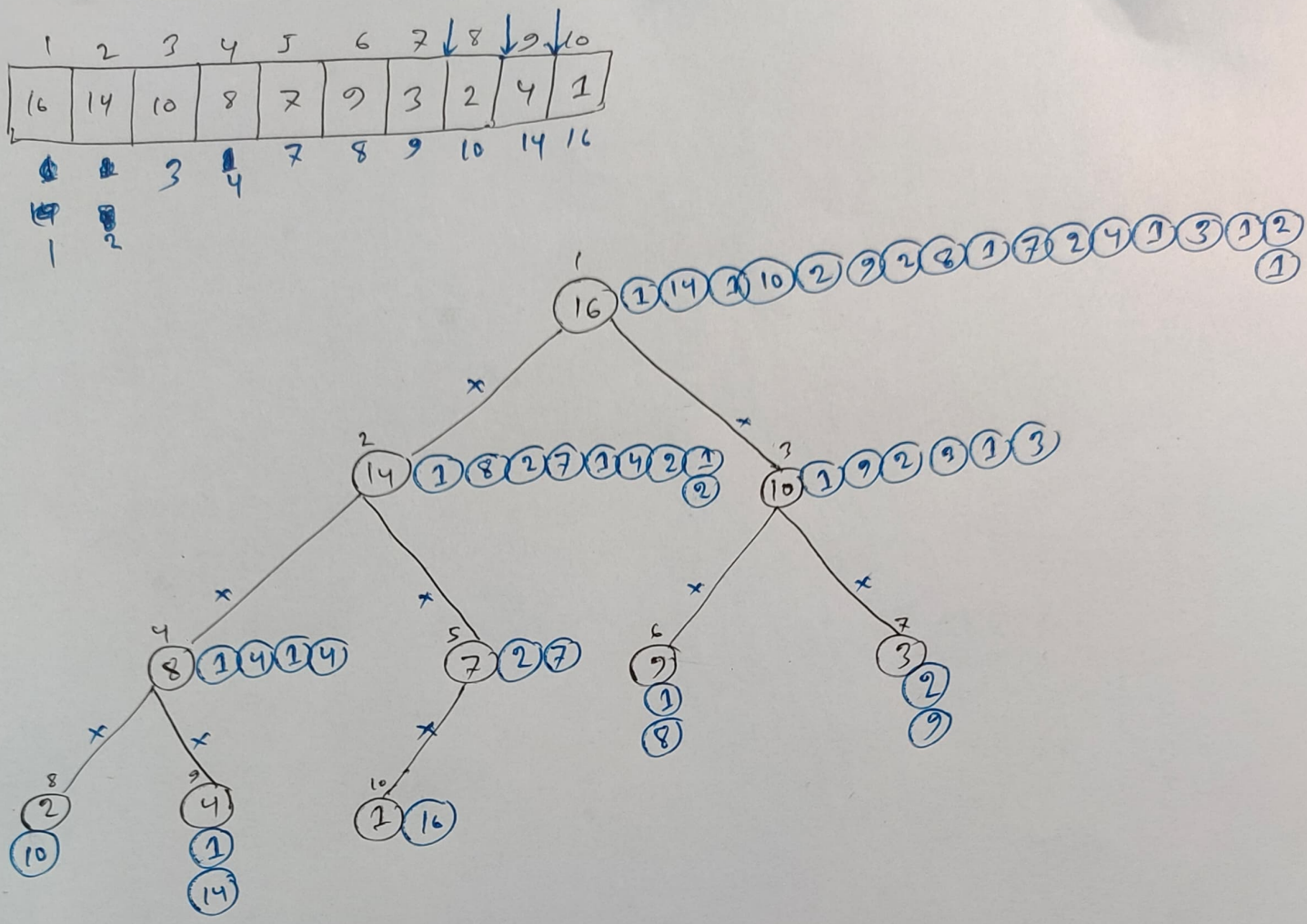
for i = n down to 2

  exchange A[1] with A[i]

  A.heap-size = A.heap-size - 1

  MAX-HEAPIFY (A, 1) $\Rightarrow O(\lg n)$

$O(n \lg n)$



Priority Queue

(i) max-priority ⇒ max-heap base

(ii) min-priority ⇒ min-heap base

MAX-HEAP-MAXIMUM (A)

if A.heap-size < 1

error "heap underflow"

return A[1]

$\Theta(1)$

MAX-HEAP-EXTRACT-MAX(A)

$\Theta(1)$
- max = MAX-HEAP-MAXIMUM (A)
- A[1] = A[A.heap-size]
- A.heap-size = A.heap-size − 1

$O(\lg n)$ ← MAX-HEAPIFY (A, 1)

return max

$O(\lg n)$

MAX-HEAP-INCREASE-KEY (A, x, k)

x: previous value, k: new value ⇒ $x \leq k$

$\Theta(1)$
- if k < x.key
  - error "new key is smaller than current key"
- x.key = k

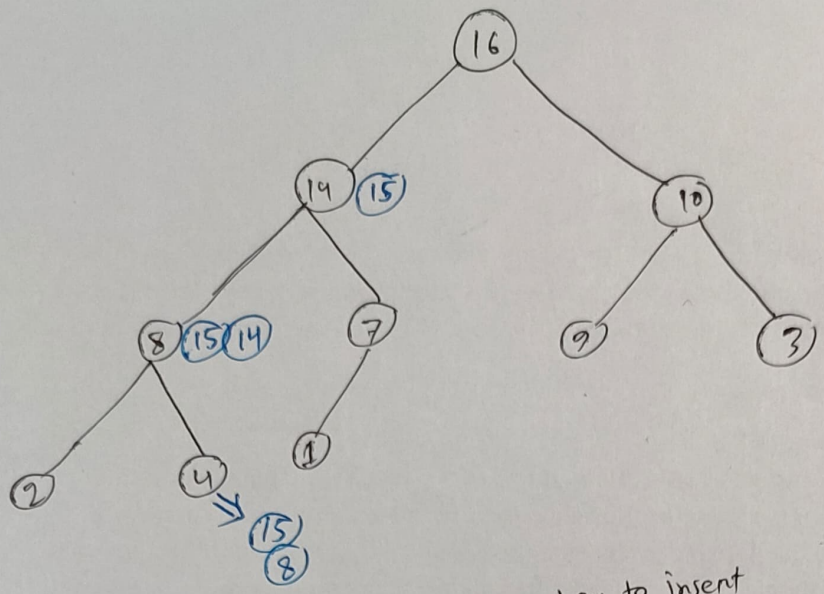
$\Theta(n)$ ← find the index i in array A where object x occurs

$O(\lg n)$ ← while i > 1 and A[PARENT(i)].key < A[i].key

exchange A[i] with A[PARENT(i)], updating the information that maps priority queue object to array indices

i = PARENT(i)

⇒ Total runtime = $O(\lg n)$



key to insert

MAX-HEAP-INSERT (A, x, n)

heap-size

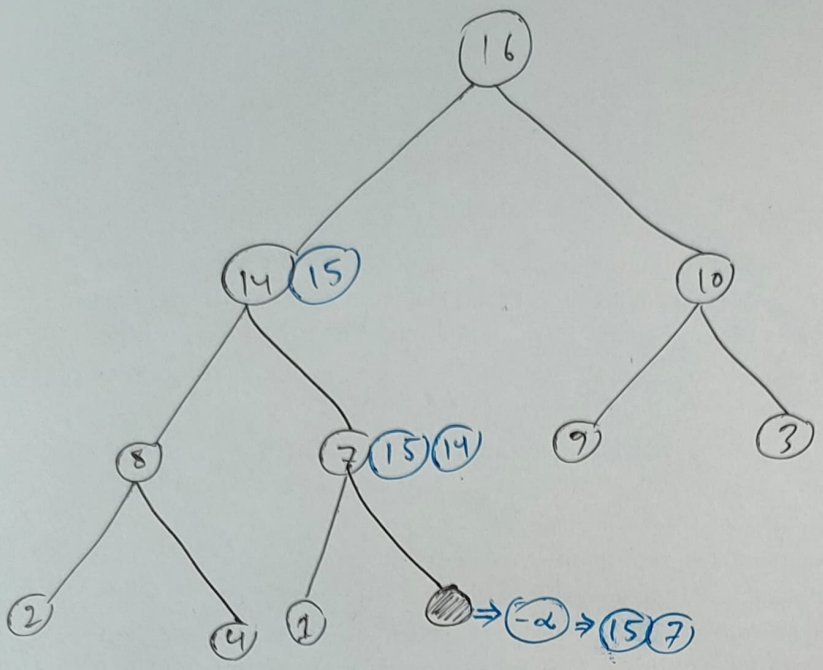
$\Theta(1)$

```
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = -∞
A[A.heap-size] = x
```

$O(n)$ ← map x to index heap-size in the array

$O(\lg n)$ ← MAX-HEAP-INCREASE-KEY (A, x, k)

Total runtime = $O(\lg n)$



Quize- Question

- pseudocode with some changes
- Draw the operation set
- Time analysis
- comparison between two algorithm