Your Ultimate Guide To Landing Top AI roles

DECODE AiML

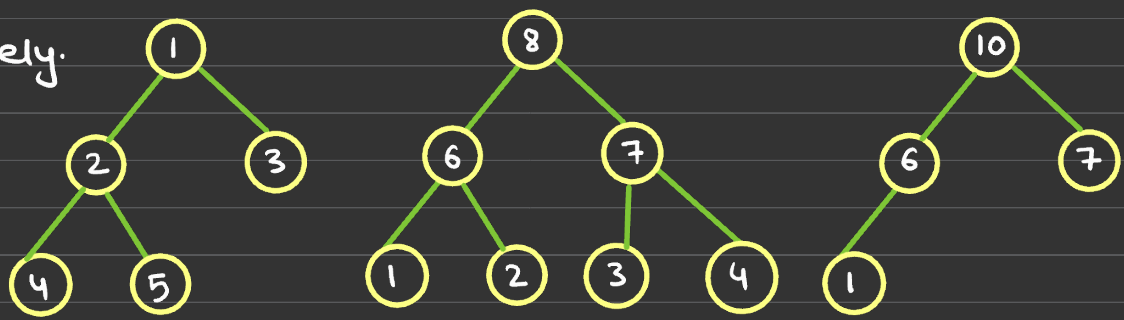
2.17

Introduction to Heaps

→ If we need a data structure where time complexity of insertion, find min, delete min should be minimum, there comes the heap.

→ Heap is a Complete binary Tree with some properties

1. Min heap :- Parent node value should be lesser than value of it's children.
2. Max heap :- Parent node value should be greater than value of its children.
3. Applies recursively.

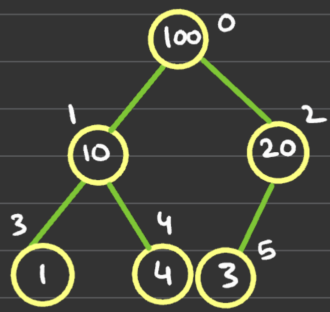


→ It is a Complete binary Tree ⇒ a Binary Tree where

1. all (n-1) levels are Completely filled.
2. Nodes at last level are filled left to right

Implementation of Heap

→ Though Heap is a Tree data structure, we can represent heap using a list data structure and interpret list as a Tree.



100	10	20	1	4	3
0	1	2	3	4	5

→ If parent index $= i$

→ Index of left child $= 2 \times i + 1$

→ Index of right child $= 2 \times i + 2$

→ If child index $= i$

→ Index of Parent $= \lfloor (i-1)/2 \rfloor$

→ A sorted list will always be a heap.

① 14, 13, 12, 10, 8

② 8, 10, 12, 13, 14

Key operations in Heap

→ If we just say Heap, it can either mean min or max heap.

① Build Heap

→ Given an array of number, you need to build max heap. How?

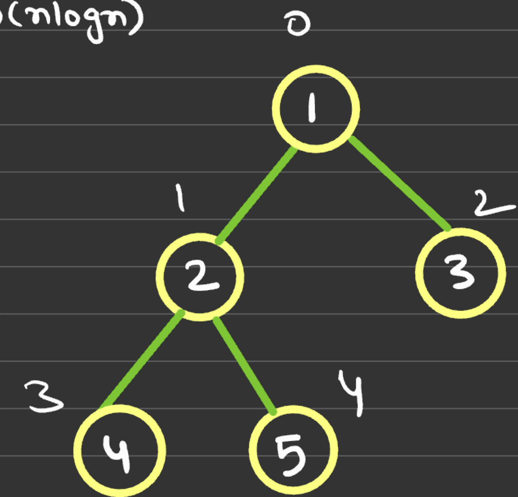
Approach 1:

Sort in reverse sorted order. But $T(n) \rightarrow O(n\log n)$

Approach 2:

```
def build_heap(arr):
    n = len(arr)
    for i in range(n//2-1, -1, -1):
        heapify(arr, n, i)
    return arr
```

[1, 0]



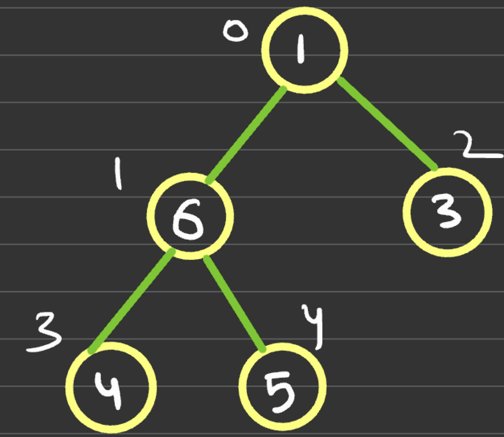
$5//2 - 1 \Rightarrow 2 - 1 = 1$

② Heapify

→ The heapify operation takes a node as input and ensures that the subtree rooted at that node satisfies the heap property.

→ It assumes subtree are already heaps but current node might be violating heap property.

```
def heapify(arr, n, i):
    largest = i
    left = 2xi+1
    right = 2xi+2
    if left<n and arr[left] > arr[largest]:
        largest = left
    if right<n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest],
                               arr[i]
        heapify(arr, n, largest)
```



$T(\text{Tree}) = T(\text{LST}/\text{RST}) + O(1)$

Time & Space Complexity

① Heapify

↳ Time Complexity $= O(\log n)$ ← Height of Tree is $\log n$

↳ Space Complexity $= O(\log n)$ ← Auxiliary space for stack frame allocation due to Recursion

② Build Heap

↳ Time Complexity = heapify() called $(n/2)$ times

$= O(\log n) \times n/2 = O(n \cdot \log n)$ ✗ wrong.

but, actually.

$\boxed{\text{Time Complexity} = O(n)}$ ← because lower level nodes take less work.

↳ Space Complexity $= O(\log n)$ ← Space taken by heapify()

③ Get Max

→ Return the root element of Heap

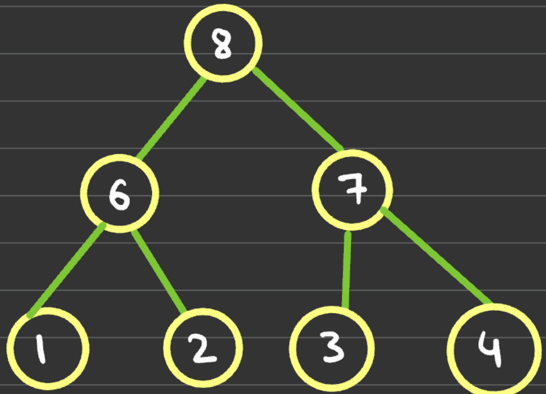
→ Time Complexity = O(1)

Space Complexity = O(1)

④ Extract Max

→ Extract max operation remove this root and restore the heap property.

```
def extract_max(arr, n):
    ans = arr[0]
    arr[0] = arr[n-1]
    n -= 1
    heapify(arr, n, 0)
    return ans
```



→ Time Complexity = O(logn)

Space Complexity = O(logn)

5. Insert

→ Add a new key at the end of heap.

→ Bubble up until heap property is restored.

→ $T(n) = O(\log n)$

6. Increase-Key / Decrease-key

→ Increase-key in max heap may require bubbling up.

→ Decrease-key may require bubbling down.

→ $T(n) = O(\log n)$

7. Delete

→ Remove a node at index i

→ Replace it with last element, shrink heap and restore heap property with heapify.

→ $T(n) = O(\log n)$

Heap Implementation in Python

```python
import heapq
```
← default min-heap

```python
nums = [5, 3, 1, 8, 2]
heapq.heapify(nums)
```
← Build Heap

```python
heapq.heappush(nums, 0)
```
← Insert

```python
heapq.heappop(nums)
```
← pop smallest element

→ max-heap simulated using negation.

DECODE AiML

Like Subscribe