Data Structures and Algorithms

Lab 8
Heap Sort

Agenda

- Complete Binary Tree
- Heap
  - Min/ Max
  - Array Example
  - Building a Max Heap
- Heap sort
- Sorting Comparison

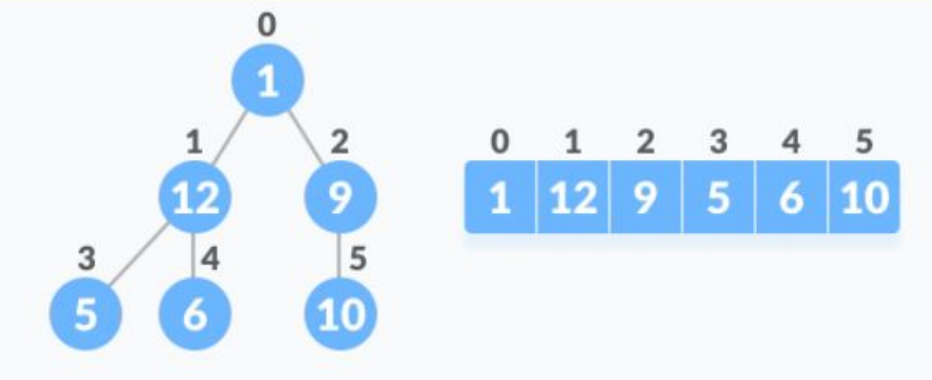
Complete Binary Tree (CBT)

8.1 What is a complete Binary Tree?

Complete Binary Tree (CBT)

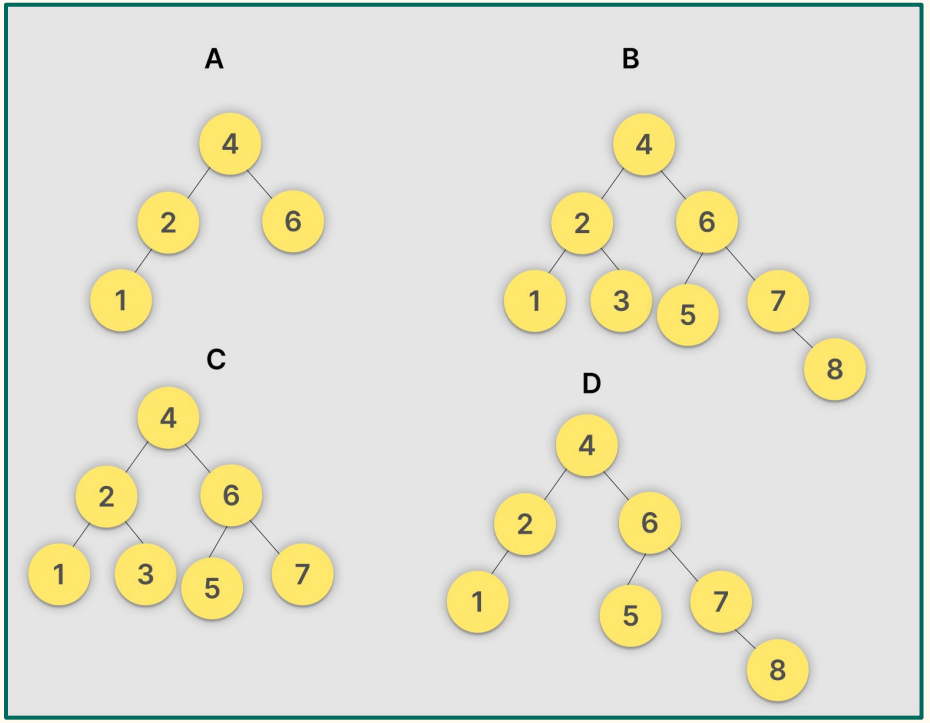
8.1 What is a complete Binary Tree?

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.



Complete Binary Tree (CBT)

8.2 Which of A,B,C,D is considered CBT? and Why?



Complete Binary Tree (CBT)

Binary tree construction:

1. Select the first element of the list to be the root node.
2. Put the second element as a left child of the root node and the third element as the right child. (no. of elements on level-II: 2)
3. Put the next two elements as children of the left node of the second level. Again, put the next two elements as children of the right node of the second level (no. of elements on level-III: 4 elements).
4. Keep repeating until you reach the last element.

CBT as Array

Relationship between array indexes and tree element

- `Root = 0`
- `Parent = (i-1) / 2`
- `Left = 2*i + 1`
- `Right = 2*i + 2`

Heap data structure

8.3-What is Heap Data structure?

8.4- What is Min Heap?

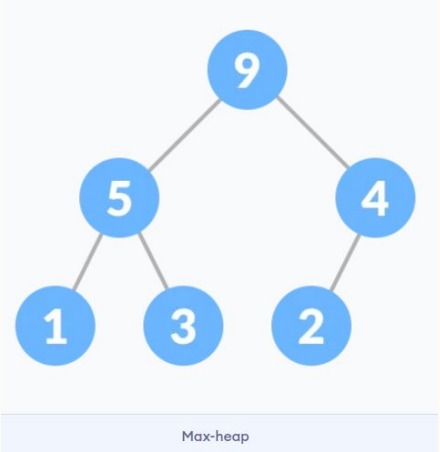
8.5- What is Max Heap?

8.6- Can we implement a Heap using Array? How to calculate index of each Node?

Heap data structure

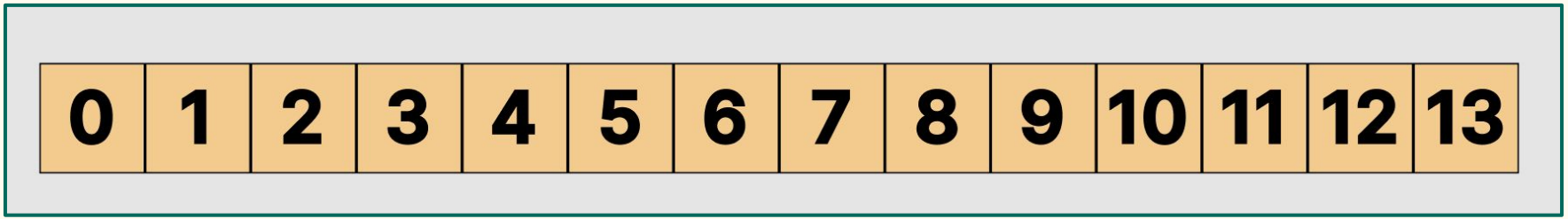
Heap data structure is a *complete binary tree* that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called **max heap** property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called **min heap** property.

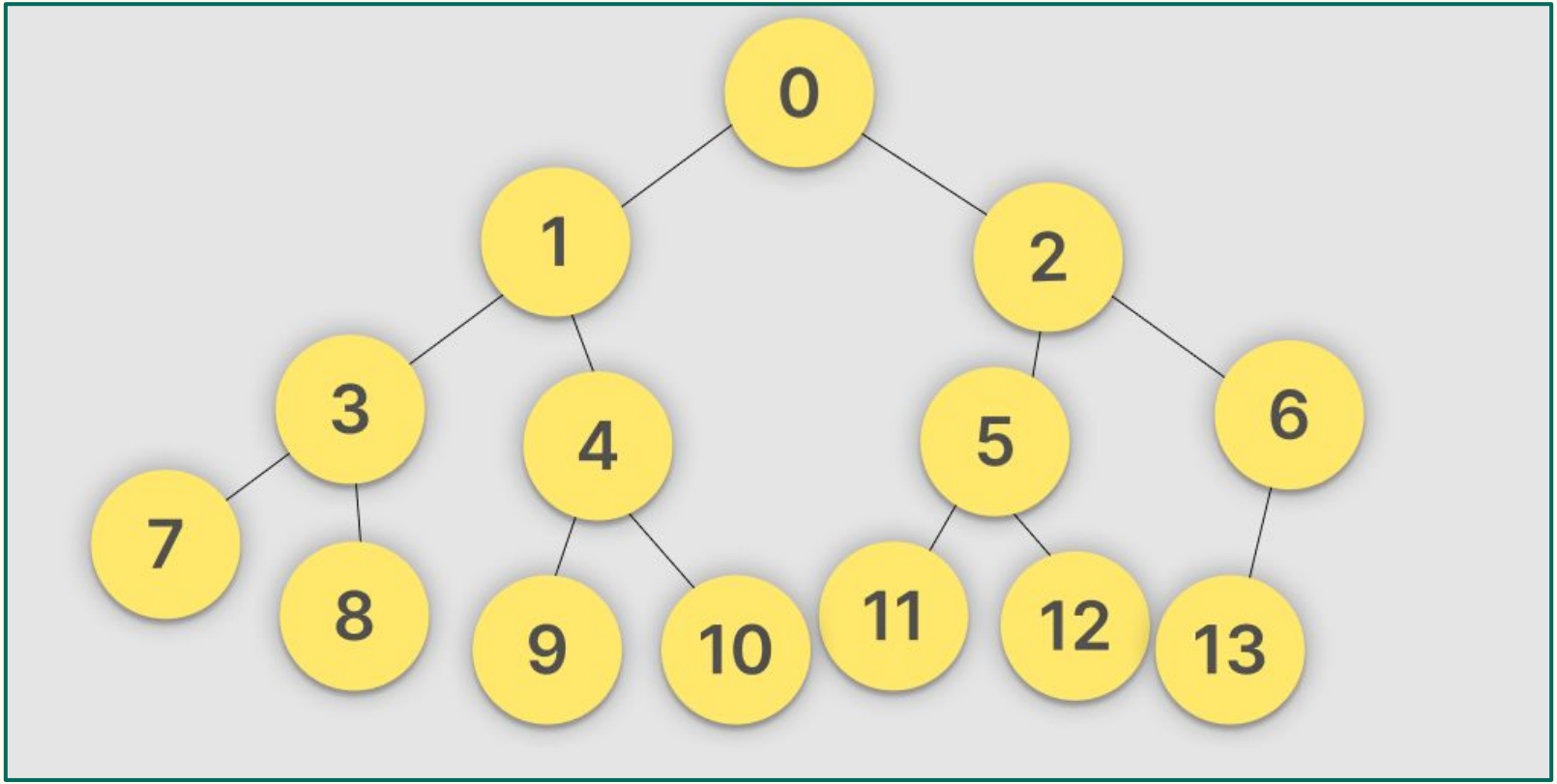


Heap data structure

8.7- Build a **Min Heap** from the Following Array:

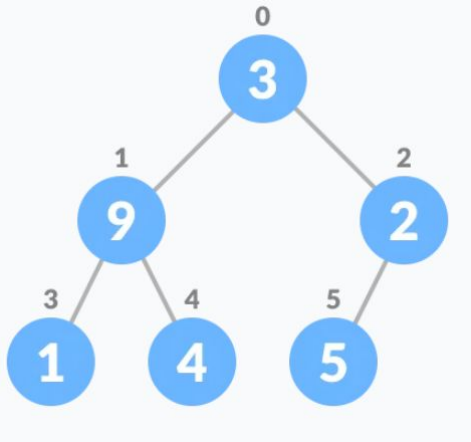


8.7-Solution



Heapify

- Create a complete binary tree from the array

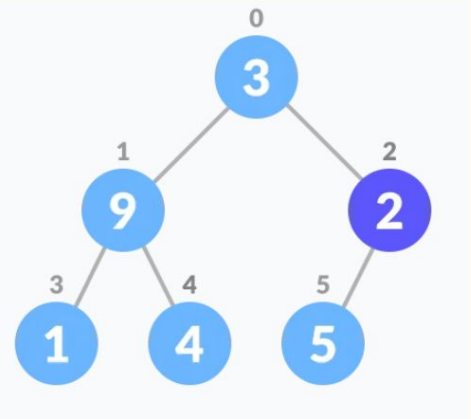


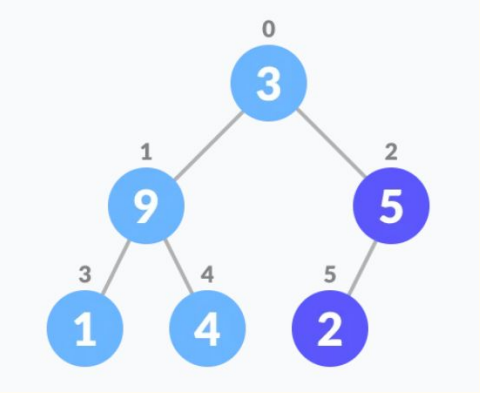
Heapify

- Start from the first index of non-leaf node whose index is given by n/2 - 1.

1. Set current element i as *largest*.
2. The index of left child is given by 2i + 1 and the right child is given by 2i + 2.
   - If *leftChild* is greater than *currentElement* (i.e. element at ith index), set *leftChildIndex* as largest.
   - If *rightChild* is greater than element in *largest*, set *rightChildIndex* as *largest*.
3. Swap *largest* with *currentElement*

- Repeat the above steps starting from the non-leaf nodes until the subtrees are also heapified.





Heapify algorithm: (the left and right indexes depends on the array)

Algorithm 1: Max-Heapify Pseudocode

```
Data: B: input array; s: an index of the node
Result: Heap tree that obeys max-heap property
Procedure Max-Heapify(B, s)
    left = 2s;
    right = 2s + 1;
    if left ≤ B.length and B[left] > B[s] then
        largest =left;
    else
        largest =s;
    end
    if right ≤ B.length and B[right] > B[largest] then
        largest =right;
    end
    if largest ≠ s then
        swap(B[s], B[largest]);
        Max-Heapify(B, largest);
    end
end
```

Building a max heap algorithm:

Algorithm 2: Building a Max-Heap Pseudocode

```
Data: B: input array
Result: Heap tree
Procedure Max-Heap-Building(B)
    B.heapsize = B.length;
    for k = B.length/2 down to 1 do
        Max-Heapify(B, k);
    end
end
```

Heap sort

You have the following number in Order

15, 20, 7, 9, 30, 31, 50, 8

8.8- Build a Heap data structure from them.

8.9- Represent them in Array.

8.10- Heapify the array (max heap)

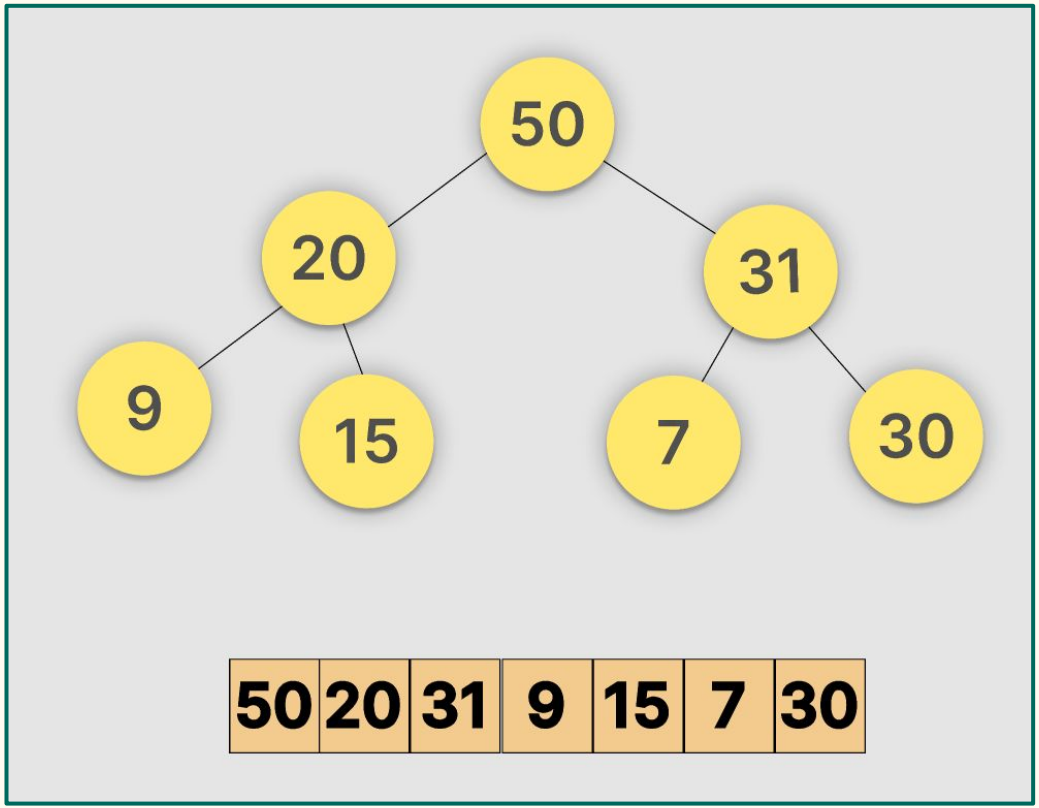
8.11- Remove the Root Node and put it at the end of the array, put the last item at the vacant place (do this 4 times), (each time check the Heap properties), save the deleted Nodes in an Array

8.12- What do you observe?

Solution

8.8- Build a Max Heap from them:

8.9- Represent them in Array:



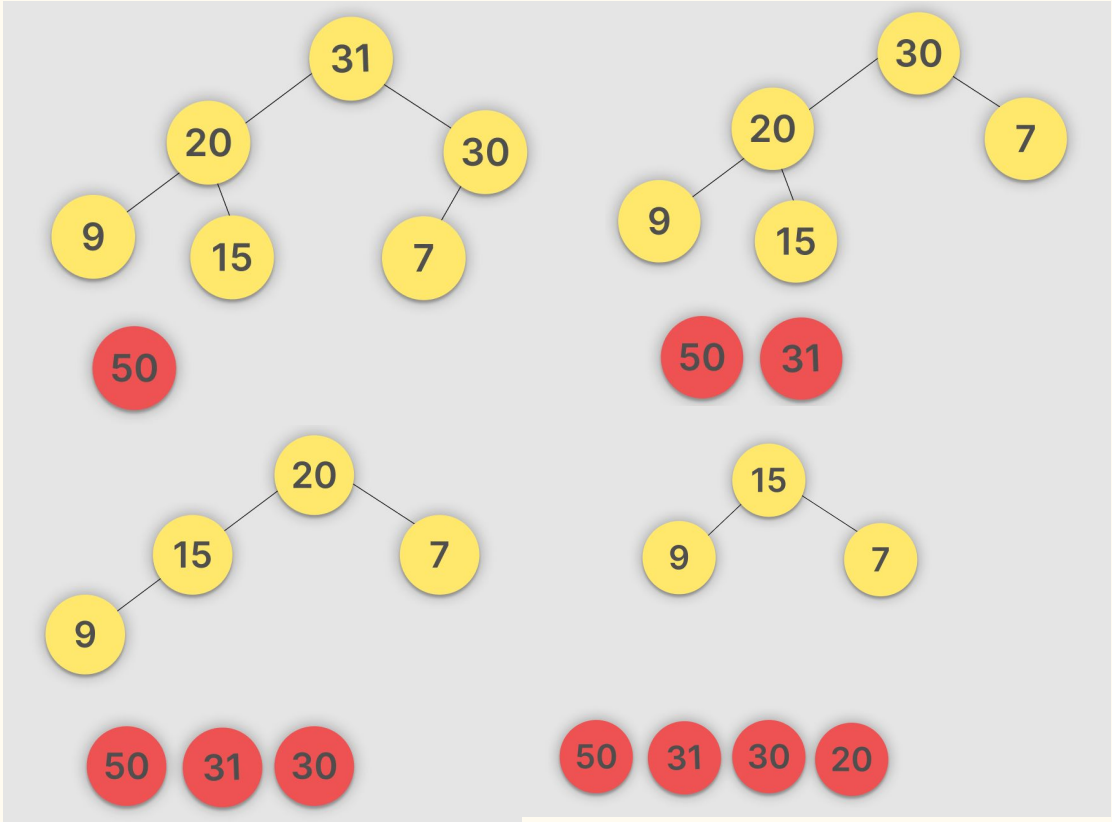
Heap sort

Swap and remove the Root Node 4 times:

Deleted nodes are: (50,31,30,20)

8.12- What do you observe?

You are Sorting !!!



Heap Sort Steps

1. Build a Max Heap (the largest item is at the root node)
2. Swap and delete The Root Node
3. Reduce the size of the heap by 1
4. Heapify the root element again so that we have the highest element at root.
5. While heapSize $>$1 goto 2

Heap Sort

8.14- Sort the Following numbers Using Heap Sort

15, 5, 20, 1, 17, 10, 30, 40, 3, 50, 13, 40, 1, 5

Show The Max Heap you build and the Deleted nodes

Comparison

8.15- Is Heap sort faster than **Quick Sort** and **Merge Sort**? Explain.

Comparison

8.15- Is Heap sort faster than **Quick Sort** and **Merge Sort**? Explain.

- the height of a complete binary tree containing n elements is *log n*
- to heapify an element we will need to move an element from the root to the leaf node making a multiple of *log(n)* comparisons and swaps.
- During the *build_max_heap* stage, we do that for *n/2* elements so the worst case complexity of the *build_heap* step is *n/2*log n ~ nlog n*.
-
- During the sorting step, we swap and heapify the root element. this again takes log n worst time because. Since we repeat this n times, the *heap_sort* step is also *nlog n*.
- Also since the *build_max_heap* and *heap_sort* steps are executed one after another, the algorithmic complexity is not multiplied and it remains in the order of *nlog* n.

Comparison

Sorting Algorithm	Best case	Average case	Worst case	Space complexity
Merge sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	O(n)
Quick sort	O(n*log(n))	O(n*log(n))	O(n^2)	O(log(n))
Heap sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	O(1)

See You next week!