236. Lowest Common Ancestor of a Binary Tree
We will use post-order traverse all trees and return the node if left and right is not None, if left or right, we will return left or right. TC is O(n)

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        if not root:
            return None
        left, right = None, None
        if root.left:
            left = self.lowestCommonAncestor(root.left, p, q)
        if root.right:
            right = self.lowestCommonAncestor(root.right, p, q)

        if root == p or root == q:
            return root
        elif left and right:
            return root
        elif left or right:
            return left or right
        else:
            return None
```

42. Trapping Rain Water
We will start from both ends. And always remember the highest left side and right side. Then move the lowest side to center and accumulate all empty cells. TC is O(n)

```
class Solution:
    def trap(self, height: List[int]) -> int:
        left, right = 0, len(height) - 1
        count, left_bound, right_bound = 0, 0, 0

        if not height:
            return 0

        while left < right:
            if height[left] <= height[right]:
                count += max(left_bound - height[left], 0)
                left_bound = max(left_bound, height[left])
                left += 1
            else:
                count += max(right_bound - height[right], 0)
                right_bound = max(right_bound, height[right])
                right -= 1
```

```
        return count
```

146. LRU Cache
We will use map and double linked list to remember all key-value pairs. We will use map to memorize whether these keys exists and use dll to remember least used order. TC is O(1), O(1)

```
class Node:
  def __init__(self, key, val):
    self.key = key
    self.val = val
    self.next = None
    self.prev = None

class LRUCache:

    def __init__(self, capacity: int):
        self.capacity = capacity
        self.dummy = Node(0, 0)
        self.dummy.next = self.dummy
        self.dummy.prev = self.dummy
        self.map = {}

    def get(self, key: int) -> int:
        if key in self.map:
          node = self.map[key]
          node.prev.next = node.next
          node.next.prev = node.prev
          node.next = self.dummy.next
          self.dummy.next = node
          node.next.prev = node
          node.prev = self.dummy
          return node.val
        else:
          return -1

    def put(self, key: int, value: int) -> None:
        if key in self.map:
          node = self.map[key]
          node.prev.next = node.next
          node.next.prev = node.prev
          node.val = value
        else:
          if self.capacity > 0:
            self.capacity -= 1
```

```
      else:
        del_node = self.dummy.prev
        del_node.prev.next = self.dummy
        self.dummy.prev = del_node.prev
        del self.map[del_node.key]
      node = Node(key, value)
    self.map[key] = node
    node.next = self.dummy.next
    self.dummy.next = node
    node.next.prev = node
    node.prev = self.dummy
```

208. Implement Trie (Prefix Tree)
We will use nested hashmap to implement Trie. And set ‘isWord’ to True if it’s the end of a word.
TC is O(len(word))

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.trie = {}

    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
        """
        node = self.trie
        for c in word:
            if c not in node:
                node[c] = {}
            node = node[c]
        node['isWord'] = True


    def search(self, word: str) -> bool:
        """
        Returns if the word is in the trie.
        """
        node = self.trie
        for c in word:
            if c not in node:
                return False
```

```
            node = node[c]
        return 'isWord' in node


    def startsWith(self, prefix: str) -> bool:
        """
        Returns if there is any word in the trie that starts with the given prefix.
        """
        node = self.trie
        for c in prefix:
            if c not in node:
                return False
            node = node[c]
        return True
```

5. Top K elements in a data stream

Use database

Mapreduce

SQL

Multiple SQL partition all data and use central SQL to get top K from all top k sequence.