Stacks / Queues :

Implementation of stack using array

```
class Stack {
private:
    int top;
    static const int MAX = 100;
    // Define maximum size of stack {It has to pre-defined}
    int stack[MAX];

public:
    Stack() {
        top = -1; // Initialize top to -1 indicating stack is empty
    }

    void push(int item) {
        if (top == (MAX - 1)) {
            cout << "Stack Overflow" << endl;
        } else {
            stack[++top] = item; // Increment top and add item
        }
    }

    int pop() {
        if (top < 0) {
            cout << "Stack Underflow" << endl;
            return -1; // Return -1 indicating stack is empty
        } else {
            return stack[top--]; // Return top item and decrement top
        }
    }

    int peek() {
        if (top < 0) {
            cout << "Stack is empty" << endl;
            return -1; // Return -1 indicating stack is empty
        } else {
            return stack[top]; // Return top item
        }
    }

    bool isEmpty() {
        return top < 0; // Return true if stack is empty
    }
```

```
    int size() {
        return top + 1; // Return size of stack
    }
};
```

Implementation of queue using array

```
class Queue {
private:
    int front, rear, size; // added at rear and removed from front
    static const int MAX = 100; // Define maximum size of queue
    int queue[MAX];

public:
    Queue() {
        front = 0;
        rear = -1;
        size = 0;
    }

    void enqueue(int item) {
        if (size == MAX) {
            cout << "Queue Overflow" << endl;
        } else {
            rear = (rear + 1) % MAX; // Circular increment
            queue[rear] = item;
            size++;
        }
    }

    int dequeue() {
        if (size == 0) {
            cout << "Queue Underflow" << endl;
            return -1; // Return -1 indicating queue is empty
        } else {
            int item = queue[front];
            front = (front + 1) % MAX; // Circular increment
            // {IT IS NOT MANDATORY THAT REAR IS ALWASY AHEAD OF FRONT}
            size--;
            return item;
        }
    }

    int peek() {
        if (size == 0) {
            cout << "Queue is empty" << endl;
            return -1; // Return -1 indicating queue is empty
        } else {
```

```
            return queue[front]; // Return front item
        }
    }

    bool isEmpty() {
        return size <= 0; // Return true if queue is empty
    }

    int getSize() {
        return size; // Return size of queue
    }
};
```

Implement stack using linked lists :

```
// Node structure
struct Node {
    int data;
    Node* next;
};

// Stack class using linked list
class Stack {
private:
    Node* top;

public:
    Stack() {
        top = nullptr; // Initialize top as NULL
    }

    void push(int item) {
        Node* newNode = new Node(); // Create a new node
        newNode->data = item;        // Assign data
        newNode->next = top;         // Link new node to the top
        top = newNode;               // Update the top to new node
    }

    int pop() {
        if (isEmpty()) {
            cout << "Stack Underflow" << endl;
            return -1; // Return -1 indicating stack is empty
        } else {
            Node* temp = top;
            int popped = top->data;
            top = top->next; // Move top to the next node
            delete temp;     // Delete old top
            return popped;
```

```
        }
    }

    int peek() {
        if (isEmpty()) {
            cout << "Stack is empty" << endl;
            return -1; // Return -1 indicating stack is empty
        } else {
            return top->data; // Return top item
        }
    }

    bool isEmpty() {
        return top == nullptr; // Return true if stack is empty
    }
};
```

Implement Queue using Linked Lists :

```
// Node structure
struct Node {
    int data;
    Node* next;
};

// Queue class using linked list
class Queue {
private:
    Node* front;
    Node* rear;

public:
    Queue() {
        front = rear = nullptr; // Initialize both front and rear as NULL
        // {IT DOESN'T MATTER IF WE ARE CHECKING rear or front AS BOTH
        // WOULD BE nullptr SIMULTANEOUSLY}
    }

    void enqueue(int item) {
        Node* newNode = new Node(); // Create a new node
        newNode->data = item;
        newNode->next = nullptr;    // New node will be the last node

        if (rear == nullptr) {      // If queue is empty
            front = rear = newNode; // Both front and rear are the new node
            return;
        }
```

```
        rear->next = newNode;       // Link new node to the last node
        rear = newNode;             // Update rear to new node
    }

    int dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow" << endl;
            return -1; // Return -1 indicating queue is empty
        }

        Node* temp = front;
        int dequeued = front->data;
        front = front->next; // Move front to the next node

        if (front == nullptr) { // If front becomes NULL, rear should also be NULL
            rear = nullptr;
        }

        delete temp; // Delete old front
        return dequeued;
    }

    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return -1; // Return -1 indicating queue is empty
        } else {
            return front->data; // Return front item
        }
    }

    bool isEmpty() {
        return front == nullptr; // Return true if queue is empty
    }
};
```

Implementation of stack using 1 queue

```
class Stack {
private:
    queue<int> q;

public:
    void push(int x) {
        int size = q.size(); // Get current size of queue
        q.push(x); // Push the element into the queue

        // Move all elements before the newly inserted
```

```
        // element to the back of the queue
        for (int i = 0; i < size; i++) {
            q.push(q.front());
            q.pop();
        }
    }

    int pop() {
        if (q.empty()) {
            cout << "Stack Underflow" << endl;
            return -1;
        }

        int popped = q.front();
        q.pop();
        return popped;
    }

    int top() {
        if (q.empty()) {
            cout << "Stack is empty" << endl;
            return -1;
        }
        return q.front(); // The front of the queue is the top of the stack
    }

    bool isEmpty() {
        return q.empty(); // Return true if the queue (stack) is empty
    }
};
```

Implement Queue using 1 stack

```
class Queue {
private:
    stack<int> s1, s2;
    // s1 used to keep on pushing, s2 is used to keep on popping / topping

public:
    void enqueue(int x) {
        s1.push(x); // Push the element onto s1
    }

    int dequeue() {
        if (s1.empty() && s2.empty()) {
            cout << "Queue Underflow" << endl;
            return -1;
        }
```

```
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top()); // Move all elements from s1 to s2
                s1.pop();
            }
        }

        int dequeued = s2.top(); // The front of the queue is the top of s2
        s2.pop();

        return dequeued;
    }

    int front() {
        if (s1.empty() && s2.empty()) {
            cout << "Queue is empty" << endl;
            return -1;
        }

        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top()); // Move all elements from s1 to s2
                s1.pop();
            }
        }

        return s2.top(); // The front of the queue is the top of s2
    }

    bool isEmpty() {
        return s1.empty() && s2.empty(); // Return true if both stacks are empty
    }
};
```

Valid parenthesis :

```
bool isValid(string s) {
  stack<char> st;
  for(auto i : s) {
      if(i == '{' || i == '(' || i == '[') {
          st.push(i);
      }
      else {
          if(st.empty()) return false;
          // similar to count = -1 in greedy approach
          char top = st.top();
```

```
            if((i == '}' && top == '{') || (i == ')' &&
            top == '(') || (i == ']' && top == '[')) {
                st.pop();
            } else {
                return false; // missmatch
            }
        }
    }
    return st.empty(); // only opening brackets
  }
```

Implement min-stack : (standard stack supporting an extra method get_Min())

```
class MinStack {
private:
    stack<std::pair<int, int>> s;

public:
    MinStack() {}

    void push(int x) {
        int minVal = s.empty() ? x : std::min(x, s.top().second);
        s.push({x, minVal});
    }

    void pop() {
        s.pop();
    }

    int top() {
        return s.top().first;
    }

    int getMin() {
        return s.top().second;
    }
};
```

- when we try to store elements in stack in such a way that it follows a particular order we call it monotonic stack

Next Greater Element : nums2 is the given array, nums1 has the elements to which we have to find the NGE

```
vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
  unordered_map<int, int> mp;
  stack<int> st;
  int n = nums2.size();
```

```
  // Traverse nums2 from right to left
  for(int i = n - 1; i >= 0; i--) {
      // Maintain the stack such that it contains
      // only elements greater than nums2[i]
      while(!st.empty() && st.top() <= nums2[i]) {
          st.pop();
      }

      // If the stack is empty, no greater element exists
      if(st.empty()) {
          mp[nums2[i]] = -1;
      } else {
          mp[nums2[i]] = st.top();
      }

      // Push the current element to the stack
      st.push(nums2[i]);
  }

  // Generate the result for nums1
  vector<int> ans;
  for(auto num : nums1) {
      ans.push_back(mp[num]);
  }

  return ans;
}
```

- What if in the previous question we are allowed to wrap around ?
- Assume you make a copy of array and put it beside end. Then we can apply the same logic as before.

```
vector<int> nextGreaterElements(vector<int>& a) {
  int n = a.size();
  vector<int>v(n,-1);

  stack<int>st;
  for(int i = 2*n - 1; i >= 0; i--)
  {
      // we pop out all elements smaller than current element
      while(!st.empty() && (a[i%n] >= st.top()))
      // modulo is required as we are hypothetically assuming there was a copy
      {
          st.pop();
      }

      // if stack is empty means no greater element is there
```

```
        // if not empty we make answer at that index equal to top element
        if(!st.empty() && (i < n))
        {
            v[i] = st.top();
        }

        st.push(a[i%n]);
    }

    return v;
}
```

- So when it comes to find the previous smallest element we apply the reverse logic.

- Rain water trapping
- Brute force → on the ith index find the min(leftmax, rightmax) and ans += height[i] - mini
- We could pre-calculate the leftmax, rightmax for all the indices
- A two-pointer approach

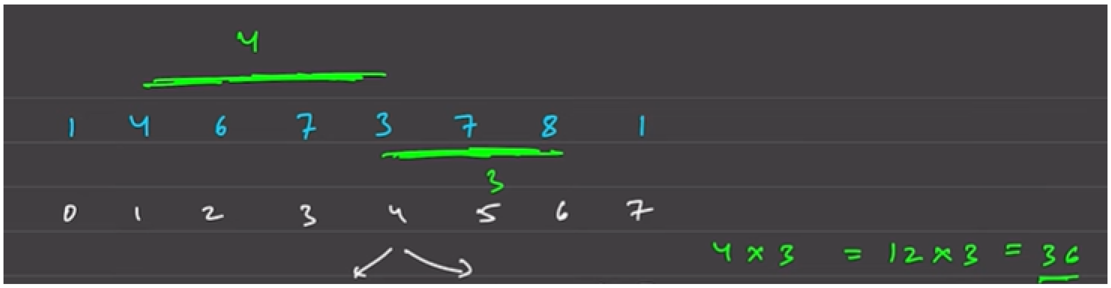
```
int trap(int heights[], int n) {
    int left = 0, right = n - 1;
    int left_max = 0, right_max = 0;
    int water = 0;

    while (left < right) {
        if (heights[left] < heights[right]) {
        // we are for sure right is bounded by a building whose height is
        // >= current index height
            if (heights[left] >= left_max) // Boundary case
                left_max = heights[left];
            else
                water += left_max - heights[left];
                // Right max pe depend nhi karti as we are only
                // moving the smaller ones first, making sure it is
                / less than right wale big towers
            left++;
        } else {
            if (heights[right] >= right_max)
                right_max = heights[right];
            else
                water += right_max - heights[right];
            right--;
        }
    }
```

```
    return water;
}
```

Sum of Subarray Minimum :

- Calculate all subarrays, find minimum in them and return sum of all those minimums



- For a given index find the nse, pse (4*3 because each element in left part can have three elements to add and extend from right)
- But we fail in case there are duplicacies, to avoid that we convert on of nse or pse to allowing even equality symbol.