Stack

- Baseball game

```
class Solution {
    public int calPoints(String[] operations) {
        Stack<Integer> st=new Stack<>();
        for(String str:operations)
        {
            if(Character.isDigit(str.charAt(0)) || str.charAt(0)
== '-')
            {
                st.push(Integer.valueOf(str));
            }
            else if(str.equals("+"))
            {
                int last=st.pop();
                int l2=st.peek();
                st.push(last);
                st.push(last+l2);
            }
            else if(str.equals("C"))
                st.pop();
            else if(str.equals("D"))
            {
                st.push(st.peek()*2);
            }
        }
```

```
        int sum=0;
        while(!st.isEmpty())
        {
            sum+=st.pop();
        }

        return sum;
    }
}
```

- Valid parenthesis

```
class Solution {
    public boolean isValid(String s) {
        Stack<Character> st = new Stack<>();

        // early rejection for odd length strings
        if (s.length() % 2 != 0)
            return false;

        for (char ch : s.toCharArray()) {
            if (ch == '(' || ch == '{' || ch == '[') {
                st.push(ch);
            } else {
                //  check before peek
                if (st.isEmpty()) return false;

                char top = st.peek();
                if ((ch == ')' && top == '(') ||
```

```
                    (ch == '}' && top == '{') ||
                    (ch == ']' && top == '[')) {
                    st.pop();
                } else {
                    return false;
                }
            }
        }

        return st.isEmpty();
    }
}
```

- Implement stack using queue

```
class MyStack {

    Queue<Integer> q1;
    Queue<Integer> q2;

    public MyStack() {
        q1 = new LinkedList<>();
        q2 = new LinkedList<>();
    }

    public void push(int x) {
        // Step 1: Move all elements from q1 to q2
        while (!q1.isEmpty())
```

```
            q2.offer(q1.poll());

        // Step 2: Push new element into q1
        q1.offer(x);

        // Step 3: Move all elements back from q2 to q1
        while (!q2.isEmpty())
            q1.offer(q2.poll());
    }

    public int pop() {
        return q1.poll();
    }

    public int top() {
        return q1.peek();
    }

    public boolean empty() {
        return q1.isEmpty();
    }
}
```

Or

```
class MyStack {

    Queue<Integer> q1;
```

```
    public MyStack() {
        q1 = new LinkedList<>();
    }

    public void push(int x) {

        // Push new element into q1
        q1.offer(x);

        // reverse until last input element
        for(int i=q1.size()-1;i>0;i--)
            q1.offer(q1.poll());
    }

    public int pop() {
        return q1.poll();
    }

    public int top() {
        return q1.peek();
    }

    public boolean empty() {
        return q1.isEmpty();
    }
}
```

- Implement queue using stacks

```java
class MyQueue {

    Stack<Integer> st1;
    Stack<Integer> st2;

    public MyQueue() {
        st1=new Stack<>();
        st2=new Stack<>();
    }

    public void push(int x) {
        while(!st1.isEmpty())
            st2.push(st1.pop());  // move everything to st2
(reverse order)
        st2.push(x);              // add new element
        while(!st2.isEmpty())
            st1.push(st2.pop());  // move everything back to st1
    }

    public int pop() {
        return st1.pop();
    }

    public int peek() {
        return st1.peek();
    }

    public boolean empty() {
        return st1.isEmpty();
    }
}
```

- Minimum stack

```
class MinStack {

    Stack<Integer> st;
    Stack<Integer> min;

    public MinStack() {
        st=new Stack<>();
        min=new Stack<>();
    }

    public void push(int val) {
        // Push to min stack if empty or val <= current min
        if(min.isEmpty() || val<=min.peek())
        {
            min.push(val);
        }
        st.push(val);
    }

    public void pop() {
        // Remove from min stack if it's the current minimum
        if(min.peek().equals(st.peek()))
        {
            min.pop();
```

```
        }
        st.pop();
    }

    public int top() {
        return st.peek();
    }

    public int getMin() {
        return min.peek();
    }
}
```

- Evaluate reverse polish notation

```
class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> st=new Stack<>();

        for(String str:tokens)
        {
            if(!str.equals("+") && !str.equals("-")
&& !str.equals("/") && !str.equals("*"))
            {
                st.push(Integer.valueOf(str));
            }
            else
            {
```

```
                // Pop top two elements
                int b = st.pop();
                int a = st.pop();

                switch (str) {
                    case "+": st.push(a + b); break;
                    case "-": st.push(a - b); break;
                    case "*": st.push(a * b); break;
                    case "/": st.push(a / b); break;
                }
            }
        }

        return st.peek();
    }
}
```

- Asteroid collision

```
class Solution {
    public int[] asteroidCollision(int[] asteroids) {
        Stack<Integer> st = new Stack<>();

        for (int i : asteroids) {

            // push if stack empty OR i moving right
            if (st.isEmpty() || i > 0) {
                st.push(i);
```

```
            }
            else {

                // handle collisions for left-moving asteroid
                while (!st.isEmpty()) {
                    int top = st.peek();

                    // top also left → no collision
                    if (top < 0) {
                        st.push(i);
                        break;
                    }

                    int abs = Math.abs(i);

                    // both same size → both destroy
                    if (abs == top) {
                        st.pop();
                        break;
                    }
                    // current asteroid smaller → destroyed
                    else if (abs < top) {
                        break;
                    }
                    // current larger → destroy top & continue
checking
                    else {
                        st.pop();

                        // if stack empty → no more collisions
                        if (st.isEmpty()) {
                            st.push(i);
                            break;
```

```
                        }
                    }
                }
            }
        }

        // build result in correct order
        int[] arr = new int[st.size()];
        for (int i = st.size() - 1; i >= 0; i--)
            arr[i] = st.pop();

        return arr;
    }
}
```

- Daily temperatures

```
class Solution {
    public int[] dailyTemperatures(int[] temperatures) {
        Stack<Integer> st=new Stack<>();
        int[] arr=new int[temperatures.length];

        for(int i=0;i<temperatures.length;i++)
        {
            // If stack empty OR current temp <= top temp → push
index
            if(st.isEmpty() ||
temperatures[st.peek()]>=temperatures[i])
                st.push(i);
```

```
            else
            {
                // Found a warmer temperature → resolve all
previous colder ones
                while(!st.isEmpty() &&
temperatures[i]>temperatures[st.peek()])
                {
                    // If current is warmer → compute days
difference
                    arr[st.peek()]=i-st.peek();
                    st.pop();
                }
                st.push(i); // always push current index after
finishing comparisons
            }
        }

        return arr;
    }
}
```

- Online stock span

```
class StockSpanner {

    // Monotonic Stack
    Stack<Integer> st;        // stores indices of prices in
decreasing order
```

```
    ArrayList<Integer> arr;   // stores prices

    public StockSpanner() {
        st=new Stack<>();
        arr=new ArrayList<>();
    }

    public int next(int price) {

        arr.add(price); // store current price

        // pop all previous smaller/equal prices
        while(!st.isEmpty() && arr.get(st.peek())<=price)
            st.pop();

        int previousGreaterIdx=(st.isEmpty())?-1:st.peek();
        int currentIdx=arr.size()-1;
        int span=currentIdx-previousGreaterIdx;  // span =
distance from previous greater

        st.push(currentIdx);

        return span;    // push current index
    }
}
```

- Car Fleet

```
class Solution {
    public int carFleet(int target, int[] position, int[] speed) {
```

```
        HashMap<Integer,Double> map=new HashMap<>();

        // time = distance / speed for each car
        for(int i=0;i<position.length;i++)
            map.put(position[i],(double)(target-
position[i])/speed[i]);

        // sort cars by position
        Arrays.sort(position);

        Stack<Integer> st=new Stack<>();

        for(int i=position.length-1;i>=0;i--)
        {
            if(st.isEmpty())    // first car forms a fleet
                st.push(position[i]);
            else
            {
                // if current car reaches later → forms new fleet
                if(map.get(position[i])>map.get(st.peek()))
                    st.push(position[i]);
            }
        }

        return st.size();
    }
}
```

- Simplify path

```java
class Solution {
    public String simplifyPath(String path) {
        String str[] = path.split("/");   // split path by '/'
        Stack<String> st = new Stack<>();

        for (String s : str) {
            if (s.equals("..") && !st.isEmpty())
                st.pop();                 // go one directory back
            else if (!s.equals("") && !s.equals(".")
&& !s.equals(".."))
                st.push(s);               // valid folder name → push
        }

        String wd = "";

        for (String s : st)
            wd = wd + "/" + s;            // rebuild canonical path

        return (st.size() == 0) ? "/" : wd; // root if empty
    }
}
```

- Decode string

```java
class Solution {
    public String decodeString(String s) {
        Stack<String> st = new Stack<>();

        for (char ch : s.toCharArray()) {
```

```
            // push characters until ']' found
            if (ch != ']') {
                st.push(String.valueOf(ch));
            }
            else {
                // get the substring inside [...]
                String subs = "";
                while (!st.peek().equals("["))
                    subs = st.pop() + subs;
                st.pop(); // remove '['

                // get the number before '['
                String num = "";
                while (!st.isEmpty() &&
Character.isDigit(st.peek().charAt(0)))
                    num = st.pop() + num;
                int k = Integer.parseInt(num);

                // repeat substring and push back
                String repeated = subs.repeat(k);   // instead of a
loop
                st.push(repeated);
            }
        }

        String ans="";
        for (String str : st)
            ans+=str;

        return ans;
    }
}
```

- Maximum frequency stack

```
class FreqStack {

    // freq → list/stack of values having this frequency
    HashMap<Integer, Stack<Integer>> st;

    // value → its frequency count
    HashMap<Integer,Integer> fmap;

    // current maximum frequency in the structure
    int maxc;

    public FreqStack() {
        st = new HashMap<>();
        fmap = new HashMap<>();
        maxc = 0;
    }

    public void push(int val) {

        // increase value's frequency
        int c = fmap.getOrDefault(val, 0) + 1;
        fmap.put(val, c);

        // create stack for this frequency if not present
        if(!st.containsKey(c))
            st.put(c, new Stack<>());

        // push value into its frequency group
        st.get(c).push(val);
```

```
        // update global maximum frequency
        maxc = Math.max(maxc, c);
    }

    public int pop() {

        // pop the most recent element from highest frequency stack
        int mx = st.get(maxc).pop();

        // decrease its frequency
        fmap.put(mx, fmap.get(mx) - 1);

        // if highest frequency stack becomes empty, reduce maxc
        if(st.get(maxc).isEmpty())
            maxc--;

        return mx;
    }
}
```

- Largest rectangle in histogram

```
class Solution {
    public int largestRectangleArea(int[] heights) {

        Stack<Integer> st = new Stack<>(); // stores indices of
increasing bars
        int max = 0;

        for(int i = 0; i <= heights.length; i++) {
```

```
            // treat i == n as height = 0 → force emptying the stack
            int curr = (i == heights.length) ? 0 : heights[i];

            // pop while current bar is smaller (meaning rectangle
ends here)
            while(!st.isEmpty() && heights[st.peek()] > curr) {

                int h = heights[st.pop()];          // height of
rectangle
                int left = st.isEmpty() ? -1 : st.peek(); // nearest
smaller left
                int width = i - left - 1;           // width between
smaller bars

                max = Math.max(max, h * width);
            }

            st.push(i); // push current index
        }

        return max;
    }
}
```