**Stack & Queue**

1. Implement Stack using Arrays
   1. **Naïve Approach**
      1. **Time Complexity: O(n)**
      2. **Space Complexity: O(n)**

**ALGO:**

class Stack{

    public:

        int \*arr;

        int top;

        int size;

    Stack(int size){

        this->size = size;

        arr = new int[size];

        top = -1;

    }

    void push(int element){

        if(size-top>1){

            top++;

            arr[top] = element;

        }else{

            cout<<"Stack Overflow"<<endl;

        }

    }

    void pop(){

        if(top >= 0){

            top--;

        }else{

            cout<<"Stack Underflow"<<endl;

        }

    }

    int peek(){

        if(top>=0){

            return arr[top];

        }else{

            cout<<"Stack is empty"<<endl;

            return -1;

        }

    }

    bool isEmpty(){

        if(top == -1){

            return true;

        }else{

            return false;

        }

    }

};

1. Implement Queue using Array
   1. **Naïve Approach:**
      1. **Time Complexity:**
         1. **Pop Function : O(1);**
         2. **Push: O(1)**
         3. **Top: O(1)**
         4. **Size: O(1)**
      2. **Space Complexity: O(n)**
2. Implement stack using Queue
   1. **Time Complexity: O(n)**
   2. **Space Complexity: O(n)**

**Algo:**

class Stack{

    queue<int> q;

    public:

        void push(int x){

            int s = q.size();

            q.push(x);

            for(int i = 0;i<s;i++){

                q.push(q.front());

                q.pop();

            }

        }

        int pop(){

            int n = q.front();

            q.pop();

            return n;

        }

        int top(){

            return q.front();

        }

        int size(){

            return q.size();

        }

};

1. Implement Queue using Stack
   1. **Naïve Approach:** 
      1. Time Complexity: O(n)
      2. Space Complexity: O(1)

Algo:

struct Queue{

    stack<int> input,output;

    void push(int data){

        while(!input.empty()){

            output.push(input.top());

            input.pop();

        }

        cout<<"The Element pushed is: "<<data<<endl;

        input.push(data);

        while(!output.empty()){

            input.push(output.top());

            output.top();

        }

    }

    int pop(){

        if(input.empty()){

            cout<<"Stack is Empty";

            exit(0);

        }

        return input.top();

    }

    int size(){

        return input.size();

    }

};

* 1. Efficient Approach:
     1. Time Complexity: O (1)
     2. Space Complexity: O(2\*n)

Algo:

class Queue{

    public:

        stack<int> input,output;

    void push(int x){

        cout<<"The element pushed is "<<x<<endl;

        input.push(x);

    }

    int pop(){

        if(output.empty()){

            while(input.size()){

                output.push(input.size());

                input.pop();

            }

        }

        int x = output.top();

        output.pop();

        return x;

    }

    int top(){

        if(output.empty()){

            while(input.size()){

                output.push(input.top());

                input.pop();

            }

        }

        return output.top();

    }

    int size(){

        return (output.size()+input.size());

    }

};

1. Check for balanced parenthesis
   1. Naïve Approach:
      1. Time Complexity: O(n);
      2. Space complexity: O(n);

Algo:

bool isValid(string s){

    stack<char> st;

    for(auto it:s){

        if(it == '(' || it == '[' || it=='{'){

            st.push(it);

        }

        else{

            if(st.size() == 0){

                return false;

            }

            char ch = st.top();

            if((it == ')' && ch == '(') || ((it == ']') && ch == '[') || (it == '}' && it == '{')){

                continue;

            }else{

                return false;

            }

        }

    }

    return st.empty();

}

1. Next Greater Element
   1. Naïve Approach:
      1. Time Complexity: O(n)
      2. Space Complexity: O(n)

Algo:

class Solution{

    public:

        vector<int> nexGreaterElement(vector<int>& nums){

            int n = nums.size();

            vector<int> ans(n,-1);

            stack<int> st;

            for(int i = 2\*n-1;i>=0;i--){

                while(!st.empty() && st.top() <= nums[i%n]){

                    st.pop();

                }

                if(i<n){

                    if(!st.empty()){

                        ans[i] = st.top();

                    }

                    st.push(nums[i%n]);

                }

            }

            return ans;

        }

};

1. Sorted Stack
   1. Naïve approach
      1. Time Complexity: O(n)
      2. Space Complexity: O(1)

Algo:

void sortedInsert(stack<int>& st,int num){

    if(st.empty() || (!st.empty() && st.top() < num)){

        st.push(num);

        return;

    }

    int n = st.top();

    st.pop();

    sortedInsert(st,num);

    st.push(n);

}

void sortedStack(stack<int>& st){

    if(st.empty()){

        return;

    }

    int num = st.top();

    st.pop();

    sortedStack(st);

    sortedInsert(st,num);

}