Challenges in TypeScript

Task 1 :

Implement the type version of binary tree inorder traversal.

For example:

```
const tree1 = {
  val: 1,
  left: null,
  right: {
    val: 2,
    left: {
      val: 3,
      left: null,
      right: null,
    },
    right: null,
  },
} as const


type A = InorderTraversal<typeof tree1> // [1, 3, 2]
```

```
export class BinarySearchTreeNode<T> {


  data: T;
```

```
  leftNode?: BinarySearchTreeNode<T>;

  rightNode?: BinarySearchTreeNode<T>;


  constructor(data: T) {

    this.data = data;

  }

}


export class BinarySearchTree<T> {

  root?: BinarySearchTreeNode<T>;

  comparator: (a: T, b: T) => number;


  constructor(comparator: (a: T, b: T) => number) {
```

```
    this.comparator = comparator;

  }


  insert(data: T): BinarySearchTreeNode<T> | undefined {

    if (!this.root) {

      this.root = new BinarySearchTreeNode(data);

      return this.root;

    }


    let current = this.root;


    while (true) {

      if (this.comparator(data, current.data) === 1) {
```

```
      if (current.rightNode) {

        current = current.rightNode;

      } else {

        current.rightNode = new BinarySearchTreeNode(data);

        return current.rightNode;

      }

    } else {

      if (current.leftNode) {

        current = current.leftNode;

      } else {

        current.leftNode = new BinarySearchTreeNode(data);

        return current.leftNode;
```

```
        }


      }}}
  inOrderTraversal(node: BinarySearchTreeNode<T> | undefined): void {


    if (node) {
      this.inOrderTraversal(node.leftNode);
      console.log(node.data);


      this.inOrderTraversal(node.rightNode);
    }
  }
}
function comparator(a: number, b: number) {
  if (a < b) return -1;
  if (a > b) return 1;
  return 0;
}
const bst = new BinarySearchTree(comparator);
bst.insert(1);
```

```
bst.insert(2);

bst.insert(3);

bst.inOrderTraversal(bst.root);
```

```
E:\htmlcssexample\all_files>node homework.js
1
2
3

E:\htmlcssexample\all_files>
```

Task 2 :

Implement CamelCase<T> which converts snake_case string to camelCase.

For example

type camelCase1 = CamelCase<'hello_world_with_types'> // expected to be 'helloWorldWithTypes'

type camelCase2 = CamelCase<'HELLO_WORLD_WITH_TYPES'> // expected to be same as previous one

```
const toCamel = (s) => {
```

```
  return s.replace(/([-_][a-z])/ig, ($1) => {

    return $1.toUpperCase()

      .replace('-', '')

      .replace('_', '');

  });
```

```
};




console.log(toCamel("hello_vineet_type_script")) ;

console.log(toCamel("hello_world_with_types")) ;

console.log(toCamel("'HELLO_WORLD_WITH_TYPES'")) ;
```

```
E:\htmlcssexample\all_files>node homework.js
helloVineetTypeScript
helloWorldWithTypes
'HELLOWORLDWITHTYPES'

E:\htmlcssexample\all_files>
```

Task 3:

Merge two types into a new type. Keys of the second type overrides keys of the first type.

For example

type foo = {

  name: string;

  age: string;

}

type coo = {

  age: number;

  sex: string

}

type Result = Merge<foo,coo>; // expected to be {name: string, age: number, sex: string}

```
 const foo = {
  name: "Vineet Verma",
  age: "23"
}
const coo = {
  age: 23 ,
  sex : "Male"
}

const Result = {...foo, ...coo};

console.log(Result);
```

```
E:\htmlcssexample\all_files>tsc homework.ts

E:\htmlcssexample\all_files>node homework.js
{ name: 'Vineet Verma', age: 23, sex: 'Male' }

E:\htmlcssexample\all_files>
```

Task 4:

Implement a generic Last<T> that takes an Array `T` and returns its last element. For example

type arr1 = ['a', 'b', 'c']

type arr2 = [3, 2, 1]

type tail1 = Last<arr1> // expected to be 'c'

type tail2 = Last<arr2> // expected to be 1

```
E:\htmlcssexample\all_files>tsc homework.ts

E:\htmlcssexample\all_files>node homework.js
the tail1 is :: c
the tail2 is :: 1

E:\htmlcssexample\all_files>
```

```
interface Array<T> {

  last(): T | undefined;

}

if (!Array.prototype.last) {

  Array.prototype.last = function () {

      if (!this.length) {

          return undefined;

      }

      return this[this.length - 1];

  };

}




let arr1 : Array<string> = ['a'  ,'b' , 'c'] ;


let arr2: Array<number> = [3,2,1];




let tail1 = arr1.last() ;

let tail2  = arr2.last() ;


console.log("the tail1 is ::"  , tail1) ;

console.log("the tail2 is ::"  , tail2) ;
```

Task 5:

Implement Capitalize<T> which converts the first letter of a string to uppercase and leave the rest as-is.
For example

type capitalized = Capitalize<'hello world'> // expected to be 'Hello world'

Task 6:

For given function type Fn, and any type A (any in this context means we don't restrict the type, and I don't have in mind any type ���) create a generic type which will take Fn as the first argument, A as the second, and will produce function type G which will be the same as Fn but with appended argument A as a last one.
For example,

type Fn = (a: number, b: string) => number

type Result = AppendArgument<Fn, boolean>

// expected be (a: number, b: string, x: boolean) => number

Task 7:

Implement Python liked any function in the type system. A type takes the Array and returns true if any element of the Array is true. If the Array is empty, return false.
For example:

type Sample1 = AnyOf<[1, '', false, [], {}]> // expected to be true.

type Sample2 = AnyOf<[0, '', false, [], {}]> // expected to be false.

Task 8:

Implement a generic PartialByKeys<T, K> which takes two type argument T and K.

K specify the set of properties of T that should set to be optional. When K is not provided, it should make all properties optional just like the normal Partial<T>.
For example

```
interface User {

  name: string

  age: number

  address: string

}


type UserPartialName = PartialByKeys<User, 'name'> // { name?:string; age:number; address:string }
```

Task 9:

Implement a generic Pop<T> that takes an Array T and returns an Array without its last element.
For example

```
type arr1 = ['a', 'b', 'c', 'd']

type arr2 = [3, 2, 1]


type re1 = Pop<arr1> // expected to be ['a', 'b', 'c']

type re2 = Pop<arr2> // expected to be [3, 2]
```

Extra: Similarly, can you implement Shift, Push and Unshift as well?

```
E:\htmlcssexample\all_files\assignment1>tsc hw11.ts

E:\htmlcssexample\all_files\assignment1>node hw11.js
[ 'a', 'b', 'c' ]
[ 3, 2 ]

E:\htmlcssexample\all_files\assignment1>
```

```ts
function newArr<T>(arr: T[]): T[] {

  arr.pop();

 return arr ;

}




const arr1  = newArr(['a' , 'b' ,'c' , 'd']) ;




const  arr2  = newArr( [3, 2, 1]) ;


console.log(arr1 , arr2) ;
```

Task 10:

Implement the type version of Array.reverse
For example:

type a = Reverse<['a', 'b']> // ['b', 'a']

type b = Reverse<['a', 'b', 'c']> // ['c', 'b', 'a']

```ts
function newArr<T>(arr: T[]): T[] {
    arr.reverse();
   return arr ;
}

const a = newArr(['a', 'b']) // ['b', 'a']
const b = newArr(['a', 'b', 'c']) // ['c', 'b', 'a']

console.log(a ,  b ) ;
```

```
E:\htmlcssexample\all_files\assignment1>tsc hw11.ts

E:\htmlcssexample\all_files\assignment1>node hw11.js
[ 'b', 'a' ] [ 'c', 'b', 'a' ]

E:\htmlcssexample\all_files\assignment1>
```