int a = 10;

int \*ptr = &a; // ptr holds the address of variable a

int b = \*ptr; // dereferencing ptr gives the value of a (which is 10) <-(\*ptr)

if I have a structure instance marks then if also shortcut ptr->

 m is a **pointer to a struct**, so \*m gives you the struct itself.

 But m->arr[i] is a **shorthand** for (\*m).arr[i] → accessing the arr[i] from the struct that m points to.

**🔹 Q2: Why even use a pointer?**

✅ Reasons we use a pointer in fillMarks(struct Marks \*m):

* If we **pass the struct by value** (i.e., fillMarks(struct Marks m)), it makes a **copy** of the entire struct. Changes won't affect the original.
* We want to **modify the actual struct** (not a copy), so we pass a **pointer**.
* Passing by pointer is also more memory-efficient, especially for large structs.

**\*is value at operator**

### When use which sold and search

You're absolutely right to think critically about time complexity — good thinking! Let’s break it down so you understand **when sorting + binary search** is worth it and when it’s not.

### 🔁 Linear Search

* **Time complexity:** O(n)
* **Works on:** **Unsorted** arrays
* **Use when:**
  + You have to search only **once or a few times**
  + The array is **small or unsorted**
* ✅ No need to sort

### 🔁 Sorting + Binary Search

* **Sorting time complexity:**
  + **O(n^2)** for bubble/selection/insertion sort (bad for large arrays)
  + **O(n log n)** for quicksort, mergesort, etc. (efficient)
* **Binary search time complexity:** O(log n)
* **Use when:**
  + You need to search **many times** (reusing the sorted array)
  + The array is **already sorted** or can be sorted once up front

### 🤔 So, is sorting "not good"?

It depends:

| **Scenario** | **Best Choice** |
| --- | --- |
| One-time search on unsorted data | **Linear search** |
| Many searches on same data | **Sort once + Binary search** |
| Data is already sorted | **Binary search** |

### 🔧 Example:

Let’s say you have an array of 1,000 elements.

* **1 search:** O(n) = 1,000 steps (linear is fine)
* **Sort + 1 binary search:** O(n log n + log n) ≈ 10,000 + 10 = 10,010 steps (worse)
* **Sort + 100 binary searches:** 10,010 total steps → now **much better** than 100,000 linear steps

### ✅ Final Advice for You:

If you're doing **just one search**, use **linear search**.  
If you'll search the same array **many times**, then **sort once** and use **binary search**.

Want me to show you both versions in your code?

##### You said:

so for bin search nlogn and sort logn still it is more then o(n) in lineare search

###### ChatGPT said:

Great observation again, Nitin! You're on the right track, but let’s clear up a small misunderstanding:

### Let's clarify:

When we say:

* **Sorting takes**: O(n log n)
* **Binary search takes**: O(log n)

We **do not** add these together to get O(n log n + log n), because O(log n) is so small compared to O(n log n) that it **doesn’t matter** in Big-O terms.

### So really:

* **Sorting + binary search** = O(n log n) total
* **Linear search** = O(n)

### ✅ When is linear search better?

If you're doing just **1 search**, yes:

* O(n) (linear search) is better than O(n log n) (sort + binary)

### ✅ When is binary search better?

If you're doing **many searches**, binary search becomes worth it **after sorting once**:

Let’s take an example:

| **Approach** | **Sort Time** | **Per Search Time** | **Total Time (for 1, 10, or 100 searches)** |
| --- | --- | --- | --- |
| Linear Search | 0 | O(n) | O(n), O(10n), O(100n) |
| Sort + Binary Search | O(n log n) | O(log n) | O(n log n + k log n) (k = # of searches) |

So:

* For **1 search**, linear is better.
* For **10+ searches**, binary becomes competitive.
* For **100+ searches**, binary is **definitely better**.

### Summary:

| **Case** | **Best Option** |
| --- | --- |
| 1-time search, unsorted | Linear search (O(n)) |
| Many searches, unsorted | Sort once + Binary (O(n log n)) |
| Already sorted | Binary search (O(log n)) |

Would you like me to give you code examples for **linear search** and **binary search with sorting** so you can compare both?

##### You said:

O(n log n + k log n) how it is good as comp to O(kn)

###### ChatGPT said:

Fantastic question, Nitin — and now you're really thinking like a computer scientist! Let's dig into:

### Your comparison:

* **Linear search for k searches**: O(k × n)
* **Sort once + k binary searches**: O(n log n + k × log n)

You’re asking:

How is O(n log n + k log n) better than O(kn)?

Let’s walk through with **examples** and real numbers:

### Example 1: Small array (n = 1000), k = 1 (one search)

* **Linear:** k × n = 1 × 1000 = 1000
* **Sort + Binary:**  
  n log n + k log n ≈ 1000 × 10 + 1 × 10 = 10,010

✅ Linear is better

### Example 2: n = 1000, k = 100 searches

* **Linear:** k × n = 100 × 1000 = 100,000
* **Sort + Binary:**  
  1000 × 10 + 100 × 10 = 10,000 + 1,000 = 11,000

✅ **Binary is better now**

### General Comparison

| **What matters** | **Linear (O(kn))** | **Sort + Binary (O(n log n + k log n))** |
| --- | --- | --- |
| k small (e.g., 1–5) | Better | Slower due to sort |
| k large (e.g., 100–1000) | Much slower | **Faster overall** |

### ✅ Key Takeaway:

* Sorting takes time **upfront** (O(n log n)) but saves time **later**.
* As k increases, the savings from log n searches **quickly outweigh** the cost of sorting.

### Real-world analogy:

Imagine you're:

* Finding a name in an **unsorted** stack of papers: you flip through every time. (Linear)
* You **sort** the papers once into alphabetical order: now every future search is super fast. (Binary)

If you're only looking once — don't sort.  
If you're looking often — sorting saves time.

In C, inside a struct you **cannot initialize a member directly** like struct Node \*top = NULL;.

**Why?**

* In C, **member variables inside structs can only be declared**, not initialized.
* Initialization of members happens **when you create an instance** of the struct, not inside its definition.

### struct Stack {

### struct Node \*top; // just declaration, no initialization here

### };

### // declare a Stack variable

### struct Stack myStack; // this creates an instance

### // then initialize top separately, e.g.,

### myStack.top = NULL;

### Or, if you want a global pointer to top directly (without struct):

### struct Node\* top = NULL; // this is a pointer to Node initialized to NULL