Hash Search vs. Tree Search

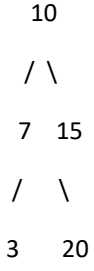
Aspect	Hash Search	Tree Search
How it Works	Uses a "hash function" to turn data into a number (key) and looks it up directly in a table, like finding a book by its exact shelf number.	Works like a family tree: starts at the top (root) and moves through branches based on conditions (e.g., greater or smaller) until it finds the item.
Speed	Very fast (almost instant) if the hash is good, as it jumps straight to the location.	Fast with sorted data, but slower than Hash Search because it follows a path step-by-step.
Example	Imagine a phonebook. You hash the name "Ali" to number 5, so you go straight to slot 5 in a table and find Ali's number.	Imagine finding 7 in a list: Start at 5 (root), go right to 8 (bigger numbers), then left to 7 (smaller than 8). You've found it!
Advantage	Super quick lookups with no step-by-step process.	Great for sorted data and allows range searches (e.g., find all numbers between 5 and 10).
Disadvantage	If two items hash to the same spot (collision), it gets tricky and slower to resolve.	If the tree is unbalanced (one side much longer), it can slow down like Linear Search.

Hash Search Example:

- **Scenario:** You're looking for a student's grade in a class of 100 students.
- **Data:** Student "Sara" → Hash function turns "Sara" into key 42 → Go to slot 42 in a table → Grade is 95.
- **Result:** Found in 1 step, no matter how big the list is!

Tree Search Example:

- **Scenario:** You're searching for the number 15 in a sorted list: [3, 7, 10, 15, 20].



What is Complexity?

Complexity tells us how an algorithm performs as the amount of data grows. It's like figuring out how long it'll take you to find something in a messy room versus a huge warehouse. There are two main types:

1. **Time Complexity:** How much time the algorithm takes to finish.
2. **Space Complexity:** How much memory (or space) it needs to work.

For now, I'll focus on **Time Complexity** since it's the most common, and I'll cover **Space Complexity** if you want later.

Time Complexity - Explained

Time Complexity is measured using **Big O Notation** (written as O), which describes the worst-case scenario for how many steps an algorithm takes as "n" (the input size) grows. It's all about how the time scales with more data.

Common Time Complexity Types:

1. **O(1) - Constant Time:**
    - **What it means:** The algorithm takes the same amount of time no matter how big "n" is. It's a fixed number of steps.
    - **Example:** Imagine you know exactly where your keys are in your house—you go straight to them. Doesn't matter if your house has 2 rooms or 20.
    - **Why it's great:** It's the fastest possible case.
2. **O(n) - Linear Time:**
    - **What it means:** The time grows directly with "n". If "n" doubles, the time doubles too.
    - **Example:** You're looking for a pen in your bag. You check every item one by one until you find it. For 10 items, it's 10 steps; for 1000, it's 1000 steps.
    - **Why it's like that:** You're checking every single thing once.
3. **O(log n) - Logarithmic Time:**
    - **What it means:** The time grows very slowly. Each step cuts the problem in half.

- **Example:** Think of finding a name in a sorted phonebook. You open the middle, decide if the name is before or after, and keep splitting until you find it. For 1000 names, it might take just 10 steps.
- **Why it's good:** Super efficient for big data.

4. **O(n log n) - Linearithmic Time:**
   - **What it means:** A mix of linear and logarithmic. The time grows a bit faster than linear but still manageable.
   - **Example:** Sorting a list (like with QuickSort). You divide the list and sort it step by step. For 1000 items, it's about 10,000 steps.
   - **Why it's used:** Common in sorting algorithms.
5. **$O(n^2)$ - Quadratic Time:**
   - **What it means:** The time grows with the square of "n". If "n" doubles, the time increases by 4x.
   - **Example:** Comparing every student in a class to every other student. For 5 students, it's 25 steps (5×5). For 1000, it's a million steps!
   - **Why it's slow:** You're doing tons of comparisons.
6. **$O(2^n)$ - Exponential Time:**
   - **What it means:** The time doubles with every new item. It explodes quickly.
   - **Example:** Solving a puzzle where each step opens up new possibilities, like calculating all moves in chess. For 10 items, it's over 1000 steps; for 20, it's a million!
   - **Why it's bad:** It's impractical for large "n".

Best, Average, and Worst Case:

- **Best Case:** The easiest scenario—like finding what you want on the first try.
- **Average Case:** What happens on average, assuming normal conditions.
- **Worst Case:** The hardest scenario—like searching a whole list and finding the item at the end (or not at all).

General Example:

- Searching for a number in a list of 10 items:
    - **O(1):** You find it in 1 step.
    - **O(n):** You check all 10 items and find it at the end.
    - **O(log n):** You split the list in half each time and find it in 3 steps.

Why Do We Care About Complexity?

It helps us pick the right algorithm:

- For small data (like 10 items), any complexity works fine.
- For big data (like a million items), you need something like O(1) or O(log n), because $O(n^2)$ could take hours!