**1. Time Complexity Overview**

* **Binary Tree Search:** Best-case time complexity is O(log(n))O(log(n)) for balanced trees, but can degrade to O(n)O(n) for unbalanced trees.
* **Binary Search Tree Insertion:**
  + **Best Case:** O(log(n))O(log(n)) for balanced trees.
  + **Worst Case:** O(n)O(n) for unbalanced trees.
* **Map/Hash Table Lookup:**
  + **Best Case:** O(1)O(1) using efficient hash functions.
* **Single For Loop:** O(n)O(n), iterates through all elements once.
* **Nested For Loops:** O(n2)O(n2), loops through all pairs.
* **Fastest Runtime:** O(log(n))O(log(n)), as seen in balanced search operations.

**2. Trees**

* **Binary Search Tree (BST):**
  + **Definition:** Organizes nodes where Left<Node<Right
  + **Efficiency:** Balanced trees offer O(log(n))O(log(n)) operations.
  + **Applications:** Data hierarchy and sorting, such as databases.
  + **Traversals:**
    - **In-order:** Produces sorted output (Left→Root→Right)(Left→Root→Right).
    - **Pre-order:** Root→Left→Right
    - **Post-order:** Left→Right→Root
    - **Level-order:** Traverses level by level, top to bottom.
* **AVL Tree:**
  + **Definition:** Self-balancing BST with height difference criteria of -1, 0, or 1.
  + **Balancing Rotations:**
    - **Left-Left Imbalance:** Perform right rotation.
    - **Right-Right Imbalance:** Perform left rotation.
    - **Left-Right or Right-Left:** Perform double rotations.
  + **Comparison to Heaps:**
    - AVL is better for key-based retrieval.
    - Heaps excel in maintaining hierarchical order for priorities.

**3. Linked Lists**

* **Structure:** Composed of nodes stored non-contiguously, each pointing to the next node.
* **Advantages:**
  + **Dynamic Size:** Grows or shrinks as needed.
  + **Efficient Insertions/Deletions:** O(1)O(1) when modifying the head or tail.
* **Comparison to Vectors:**
  + Linked Lists: Efficient for frequent insertions/deletions.
  + Vectors: Faster random access O(1)O(1), thanks to contiguous memory.

**4. Hash Tables**

* **Efficiency:** O(1)O(1) on average for insertion, deletion, and lookup.
* **Collision Resolution Techniques:**
  + **Linear Probing:** Sequentially search for the next free slot.
    - **Example:**
      * hash(x)=xmod  10hash(x)=xmod10:
        + Insert 12: Index 2.
        + Insert 22: Index 2 → move to 3.
        + Insert 32: Index 2 → 3 → 4.
  + **Separate Chaining:** Use linked lists within each slot to handle collisions.
* **Use Cases:** Ideal for non-sequential key-value storage like dictionaries.

**5. Recursion**

* **Key Component:**
  + A **base case** to terminate recursion and avoid infinite loops.
* **Applications:**
  + Traversing tree structures.
  + Solving computational problems like the N-Queens problem.

**6.**4.The **Rule of Three** refers to three important functions in classes that manage dynamic memory:

1. **Copy Constructor**: Creates a new object as a copy of an existing object.
2. **Assignment Operator**: Assigns one object to another of the same type.
3. **Destructor**: Cleans up when an object is no longer needed.

**Why It's Important**: If your class handles dynamic memory, you need to define all three to avoid memory leaks or issues like shallow copies, which can lead to unexpected behavior when objects share the same resources.Without defining these, your program could have memory management issues. For example, when two objects share a resource, like memory, changing one object could accidentally affect the other. A deep copy with a copy constructor ensures that each object has its own copy of the data. Without the Rule of Three, default constructors provided by the compiler will create shallow copies, which is often not what you want.

**7. Queue Operations**

* Access the first element using queue.front().
* Efficient for FIFO (First In, First Out) operations.

**8. Binary Tree vs. Linked List**

* **Binary Tree:**
  + Efficient search for balanced trees: O(log(n))O(log(n)).
  + Can degrade to O(n)O(n) for unbalanced.
* **Linked List:** Sequential search is O(n)O(n), less efficient for lookups.

**9. Like-Love-Meh**

* **Maps (Love):** Efficient O(1)O(1) key-value storage for large datasets.
* **Hash Tables (Like):** Fast operations O(1)O(1), but require good hash functions to avoid collisions.
* **Binary Trees (Meh):** Useful for sorted data but slower O(n)O(n) in worst-case unless balanced.

**10. Coding Questions**

**Reverse Singly Linked List:**

**Node\* reverseList(Node\* head) {**

Node\* prev = nullptr; // `prev` will track the previous node, initially null.

Node\* current = head; // `current` starts at the head of the list.

Node\* next = nullptr; // `next` will temporarily hold the next node.

while (current != nullptr) {

next = current->next; // Store the next node.

current->next = prev; // Reverse the link direction.

prev = current; // Move `prev` forward to the current node.

current = next; // Move `current` forward to the next node.

}

return prev; // After the loop, `prev` will be the new head.

}

**Simple Coding Interview Question:**

**Question:**Write a function that takes a string as input and returns the first non-repeating character in the string. If no such character exists, return null.

Data Structure Best for the Solution:

Hash Map (or Dictionary):A hash map is ideal for this problem because it allows us to store the frequency of each character as we iterate through the string.

Using a hash map, we can quickly check the frequency of each character and find the first one with a frequency of 1

.The time complexity of this approach is O(n), where n is the length of the string, because we only need to traverse the string once

to build the frequency map, and checking the map for each character is an O(1) operation.

Describe the Components that Make Up a Binary Tree

A binary tree consists of:

• Root: The topmost node.

• Nodes: Each node has a data element and two children (left and right).

• Leaf Nodes: Nodes with no children.

• Subtrees: The left and right subtrees rooted at any node.

Who Would Win in a Battle to the Death, Stacks, Deques, or Queues?

Deques would win because they offer the most flexibility and can handle both stack-like and queue-like operations. Whether you need to add/remove from the front or back,deques can do it all.

**Using the Rule of Three, Make a .h File and Include Two Functions and Two Variables**

#include <string>

class MyClass {

private:

std::string\* data; // Dynamically allocated resource

int id; // Example of a second variable

public:

// Constructor

MyClass(const std::string& initData, int initId);

// Destructor

~MyClass();

// Copy Constructor

MyClass(const MyClass& other);

// Copy Assignment Operator

MyClass& operator=(const MyClass& other);

// Member Functions

std::string getData() const; // Function to get the data

void setData(const std::string& newData); // Function to set the data

};

**In Your Own Words, Describe the Difference Between Two Data Structures**

• Array vs. Linked List:

An array is a collection of elements stored in contiguous memory locations. It allows for constant-time access to any element by its index (O(1)), but inserting or removing elements can be inefficient (O(n)) due to the need for shifting elements.

• A linked list, on the other hand, consists of nodes where each node contains data and a pointer to the next node. It allows for efficient insertions and deletions (O(1)) because no elements need to be shifted. However, accessing elements requires O(n) time, as you have to traverse the list from the head.

**Benefits of Implementing a Hash Table Compared to a Linked List**

**• Hash Table:**

**Hash tables provide faster access times with an average case of O(1) for insertion, deletion, and lookup operations. This is much faster than the O(n) time complexity of linked lists for searching, since a hash table maps keys to values directly via a hash function.**

**• Linked List:**

**Linked lists are efficient for scenarios where frequent insertions and deletions are required in the middle of the collection. However, they do not provide fast access to individual elements compared to hash tables**