Task (1): What are the 13 rules of clean code?

The "Clean Code" principles are a set of guidelines and best practices for writing clean, maintainable, and readable code. While there isn't an official list of 13 rules, there are several principles and guidelines that are commonly associated with writing clean code.

1. **Meaningful Names:** Choose descriptive and meaningful names for variables, functions, classes, and modules to make the code self-explanatory.
2. **Keep Functions Small:** Functions should be short and focused on doing one thing. They should have a single responsibility and be easy to understand at a glance.
3. **Use Descriptive Comments:** Use comments to explain why code is written, not what it does. In most cases, aim to write code that's self-explanatory without the need for comments.
4. **Single Responsibility Principle (SRP):** Each module, class, or function should have a single responsibility and should only change for one reason.
5. **Don't Repeat Yourself (DRY):** Avoid duplicating code. Reuse code through functions, classes, or libraries to eliminate redundancy.
6. **Keep Lines Short:** Limit the length of lines in your code to improve readability and make it easier to understand.
7. **Avoid Deep Nesting:** Limit the level of nesting in your code. Deeply nested code is harder to read and understand.
8. **Open-Closed Principle (OCP):** Code should be open for extension but closed for modification. You should be able to add new functionality without altering existing code.
9. **Avoid Magic Numbers and Strings:** Replace magic numbers and strings with named constants or variables to make your code more self-explanatory and maintainable.
10. **Unit Testing:** Write unit tests to ensure that your code functions correctly and can be easily maintained.
11. **Separation of Concerns:** Separate different concerns in your code, such as user interface, business logic, and data access, into distinct modules or layers.
12. **Code Conventions:** Follow a consistent coding style and naming convention within your project to make the codebase uniform and more readable.
13. **Version Control:** Use version control systems (e.g., Git) to track changes, collaborate with others, and easily revert to previous states if needed.

Task (2): What is the red – black trees?

Red black trees are a type of balanced binary search tree that use a specific set of rules to ensure that the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n).

Red Black Tree is a binary search tree in which every node is colored with either red or black. It is a type of self-balancing binary search tree. It has a good efficient worst case running time complexity.

Properties of Red Black Tree:

The red black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties:

1. The root is black.
2. Every leaf (leaf is a NULL child of a node) is black.
3. The children of a red node are black. Hence possible parent of red node is a black node.
4. All the leaves have the same black depth.
5. Every simple path from root to descendant leaf node contains same numbers of black nodes.

Rules That Every Red-Black Tree Follows:

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. Every leaf (e.i. NULL node) must be colored BLACK.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete... etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Task (3): What is scheduling algorithms and how it works?

Scheduling algorithms are a critical component of operating systems that manage the allocation of resources, particularly CPU time, to different processes or threads. These algorithms determine the order in which processes or threads are executed and how CPU time is distributed among them. The goal is to maximize system efficiency, ensure fairness, and provide responsive performance to users and applications.

Here's an overview of how scheduling algorithms work:

Process Queue: Processes or threads in an operating system are typically placed in a queue, often referred to as a ready queue, ready list, or run queue. This queue contains all the processes or threads that are ready to execute.

Dispatcher: The dispatcher is a component of the operating system responsible for selecting the next process or thread to run from the ready queue. The dispatcher uses the scheduling algorithm to make this decision.

Scheduling Algorithm: The scheduling algorithm is the heart of the scheduling process. There are various scheduling algorithms, each with its own characteristics and objectives. Here are some common scheduling algorithms:

- **First-Come, First-Served (FCFS):** The processes are executed in the order they arrive in the ready queue. It's a non-pre-emptive algorithm.
- **Shortest Job First (SJF):** The process with the shortest expected execution time is given priority. It's often non-pre-emptive but can be pre-emptive.
- **Priority Scheduling:** Each process has an associated priority, and the process with the highest priority is executed next. It can be pre-emptive or non-pre-emptive.
- **Round Robin (RR):** Processes are executed in a round-robin fashion with a fixed time quantum. If a process doesn't complete within its time quantum, it's moved to the end of the queue. It's a pre-emptive algorithm.
- **Multilevel Queue Scheduling:** Processes are divided into multiple queues, and each queue has its own scheduling algorithm. This approach is often used to categorize and prioritize processes.
- **Multilevel Feedback Queue Scheduling**: Similar to multilevel queue scheduling, but processes can move between different queues based on their behaviour. This approach adapts to the behaviour of processes.

Pre-emption: Pre-emption refers to the ability to interrupt the execution of a currently running process or thread and switch to another. Pre-emptive scheduling algorithms allow for this interruption based on certain criteria (e.g., time quantum expiration, higher-priority process arriving).

Context Switching: When a scheduling decision is made, and the dispatcher selects a new process or thread to run, a context switch occurs. This involves saving the state of the currently running process and loading the state of the newly selected process. Context

switching has some overhead but is necessary for maintaining process isolation and allowing multiple processes to run on a single CPU.

Scheduling Criteria: Scheduling algorithms consider various criteria, such as CPU burst time, priority, fairness, response time, and throughput, depending on the specific algorithm's objectives.

The choice of scheduling algorithm can have a significant impact on system performance, and the selection depends on the specific requirements and goals of the operating system and the applications it supports. Different scenarios may Favor different scheduling algorithms, and real-world operating systems may use a combination of scheduling policies to meet various needs efficiently.

Task (4): What is Hash map/Hash Table?

Hash map/Hash table is a data structure that implements an associative array, also called a dictionary, which is an abstract data type that maps keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key. Such collisions are typically accommodated in some way.

In a well-dimensioned hash table, the average time complexity for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key–value pairs, at amortized constant average cost per operation.

Hashing is an example of a space-time trade off. If memory is infinite, the entire key can be used directly as an index to locate its value with a single memory access. On the other hand, if infinite time is available, values can be stored without regard for their keys, and a binary search or linear search can be used to retrieve the element.

In many situations, hash tables turn out to be on average more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.