Questions for data structures

Question

Here are 6 hard questions about data structures, designed to challenge even experienced programmers:

1. **Designing a Cache Replacement Policy:** You need to design a cache replacement policy for a web server that handles a massive, unpredictable stream of requests. Standard LRU or FIFO are not performing adequately. Describe a hybrid algorithm that combines aspects of several known algorithms (e.g., LRU, LFU, Clock) to improve performance under high load and varying request patterns. Justify your choices and analyze its potential weaknesses. Consider the trade-off between complexity and performance.
2. **Concurrent Skip List Implementation:** Design a thread-safe implementation of a skip list data structure suitable for high-concurrency environments. Detail the synchronization mechanisms you would use (e.g., locks, atomic operations) and explain how you would handle potential race conditions during insertion, deletion, and search operations. Analyze the performance implications of your chosen synchronization methods.
3. **Bloom Filter Optimization for Dynamic Set Sizes:** A Bloom filter is used to determine if an element is *possibly* in a set. However, the optimal size of a Bloom filter depends on the expected number of elements. Describe an algorithm and data structure to efficiently manage a Bloom filter whose size needs to dynamically adjust as the number of elements in the underlying set changes significantly over time. Address the trade-offs between accuracy, memory usage, and the cost of resizing.
4. **Self-Balancing Tree under Non-Uniform Distributions:** Traditional self-balancing trees (AVL, red-black) assume relatively uniform insertion/deletion patterns. Design a self-balancing tree data structure optimized for scenarios where key insertions are heavily skewed towards a particular range or follow a predictable non-uniform distribution (e.g., Zipfian distribution). Analyze its performance advantages and disadvantages compared to standard self-balancing trees under such conditions.
5. **Space-Efficient Trie for Very Large String Sets:** You need to store and efficiently search a massive set of strings (millions or billions) in a space-constrained environment. Standard tries can be memory-intensive. Design a space-optimized trie variant that minimizes memory usage while maintaining reasonable search performance. Consider techniques like compression, node sharing, or alternative data structures for representing the trie.
6. **Persistent Data Structure for Version Control:** Design a persistent data structure (allowing access to previous versions) suitable for implementing a version control system for large text files. Explain your choice of data structure