Double-Ended Queue (Deque)
A double-ended queue, or deque, is a linear data structure that allows insertion and deletion at both ends, making it more flexible than a traditional queue. Structurally, a deque can be implemented using a circular array, doubly linked list, or another type of dynamic storage, enabling access from both ends. This dual accessibility is beneficial in scenarios that require quick insertions and deletions from both front and back. The time complexity for both insertion and deletion at either end is O(1) while accessing elements is O(n) for linear searches. Deques excel in situations where operations need to occur at both ends, though they are less efficient for arbitrary position access compared to arrays or linked lists. Deques are commonly used in task scheduling, caching algorithms, and maintaining lists of recently accessed elements.

Red-Black Tree
A Red-Black Tree is a self-balancing binary search tree with specific properties to maintain balance after insertions and deletions. Each node in a Red-Black Tree is colored either red or black, and the tree follows rules such as ensuring that no two red nodes are adjacent and that every path from the root to a leaf has the same number of black nodes. These properties ensure the tree remains approximately balanced, providing O(logn) time complexity for search, insertion, and deletion operations. Red-Black Trees are ideal for maintaining sorted data, as they balance themselves efficiently during updates. However, they can be complex to implement and manage due to the color properties. Red-Black Trees are widely used in databases, file systems, and memory management systems, where frequent insertions, deletions, and lookups need balanced performance.

Segment Tree
A Segment Tree is a specialized data structure that efficiently stores information about segments or intervals within a dataset, making it useful for range queries and updates. Structurally, a Segment Tree is a binary tree where each node represents a specific segment of an array, allowing operations like finding minimum, maximum, or sum within a range. Building a Segment Tree has O(n) complexity, and querying or updating a range takes O(logn), making it efficient for handling large datasets with frequent range-based queries. Although Segment Trees use significant memory due to their recursive structure, they are valuable in applications requiring quick range computations, such as computer graphics, computational geometry, and range-sum queries in competitive programming.