Short Question & Answer

1. What is a data structure?

A data structure is a specific way of organizing data in a computer so that it can be used efficiently. It defines the relationship between data elements and provides operations for accessing and managing them.

2. Define an abstract data type (ADT).

An ADT is a model for data structures that specifies the type of data stored, the operations supported, and the types of parameters of the operations. It provides a theoretical framework for understanding how data structures work.

3. List four basic types of data structures.

The four basic types of data structures are arrays, linked lists, stacks, and queues. These structures provide different ways to store and organize data for efficient access and modification.

4. What is the difference between primitive and non-primitive data structures?

Primitive data structures are basic types like int, char, and float, directly operated upon by machine instructions. Non-primitive data structures are more complex and are derived from primitive types, including arrays, linked lists, stacks, and queues.

5. How do data structures improve the efficiency of computer programs?

Data structures improve efficiency by organizing data in ways that enhance speed of access and modification. Proper choice of data structure can significantly reduce time complexity of algorithms and optimize resource use.

6. What is a linear list?

A linear list is a sequence of elements arranged in a linear order, where each element has a unique predecessor and successor. Examples include arrays and linked lists.

7. Explain the concept of a singly linked list.

A singly linked list is a linear data structure where each element, called a node, contains a data part and a reference (or link) to the next node in the sequence. It allows dynamic memory allocation and efficient insertions/deletions.

8. How does a singly linked list differ from an array?

A singly linked list consists of nodes with data and a link to the next node, allowing dynamic memory usage. An array is a fixed-size sequence of elements stored in contiguous memory locations, allowing direct access by index.

9. What operations can be performed on a linear list?

Operations on a linear list include insertion, deletion, traversal, searching, and updating elements. These operations allow for dynamic management and modification of the list's content.

10. Describe the process of inserting a new element into a singly linked list.

To insert a new element, create a new node, update its link to point to the next node, and adjust the link of the preceding node to point to the new node. This maintains the integrity of the list structure.

11. How is a node in a singly linked list represented in C?

In C, a node is represented using a struct, containing data and a pointer to the next node: `struct Node { int data; struct Node* next; };`. This structure defines the basic unit of a singly linked list.

12. What does the 'head' pointer represent in a linked list?

The 'head' pointer in a linked list points to the first node of the list. It is used as a reference to access and traverse the entire linked list.

13. How do you search for an element in a singly linked list?

To search for an element, start from the head node and traverse the list by following the next pointers, comparing each node's data with the target value until the element is found or the end of the list is reached.

14. Describe the process of deleting a node from a singly linked list.

To delete a node, adjust the link of the previous node to bypass the node to be deleted, and free the memory allocated to the node. Ensure proper linkage to maintain the list structure.

15. What are the advantages of using a linked list over an array?

Linked lists offer dynamic memory allocation, efficient insertions and deletions, and flexible memory usage. Unlike arrays, they do not require a predetermined size and can grow or shrink as needed.

16. Explain how to insert a node at the beginning of a linked list.

Create a new node, set its link to the current head node, and update the head pointer to the new node. This ensures the new node becomes the first node in the list.

17. How can a node be inserted at the end of a linked list?

Traverse to the last node, create a new node, set the last node’s link to the new node, and set the new node's link to NULL. This appends the new node at the end of the list.

18. Describe how to delete the first node of a linked list.

Update the head pointer to point to the second node, free the memory allocated to the first node, and adjust any necessary links. This removes the first node from the list.

19. Explain the deletion of the last node in a linked list.

Traverse to the second-last node, set its link to NULL, free the memory of the last node, and adjust any necessary links. This removes the last node from the list.

20. How do you perform a search operation in a linked list?

Traverse the list from the head, comparing each node’s data with the target value until the element is found or the end of the list is reached. Return the node or NULL if not found.

21. What is a stack?

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. Elements are added (pushed) and removed (popped) from the top of the stack.

22. List the basic operations performed on a stack.

Basic operations include PUSH (inserting an element), POP (removing the top element), PEEK (accessing the top element without removing it), and checking if the stack is empty or full.

23. Explain the LIFO principle with an example.

LIFO (Last-In-First-Out) means the last element added to the stack is the first to be removed. For example, stacking plates; the last plate placed on top is the first one taken off.

24. How is a stack implemented using an array?

An array-based stack uses an array and a top index. PUSH increments the top index and adds an element; POP decrements the top index and removes the top element.

25. Describe how a stack can be implemented using a linked list.

A linked list stack uses nodes with a top pointer. PUSH creates a new node, sets its link to the current top, and updates the top pointer. POP removes the top node and updates the top pointer.

26. What does the PUSH operation do in a stack?

The PUSH operation adds an element to the top of the stack. It involves increasing the top index (array) or creating a new node and updating the top pointer (linked list).

27. Explain the POP operation in a stack.

The POP operation removes the top element from the stack. It involves decreasing the top index (array) or removing the top node and updating the top pointer (linked list).

28. How do you check if a stack is full in array implementation?

Check if the top index is equal to the maximum size minus one. If true, the stack is full and cannot accommodate more elements without resizing.

29. Describe how to check if a stack is empty.

Check if the top index is -1 (array) or if the top pointer is NULL (linked list). If true, the stack is empty and has no elements to POP.

30. What is a stack overflow?

Stack overflow occurs when trying to PUSH an element onto a full stack. In array implementation, it happens when the top index exceeds the array's maximum size.

31. Compare array and linked list implementations of stacks.

Array stacks have fixed size, efficient indexing, and potential overflow. Linked list stacks offer dynamic size, efficient insertions/deletions, but require extra memory for node pointers.

32. What are the advantages of using a linked list to implement a stack?

Linked list stacks offer dynamic resizing, efficient memory usage, no overflow, and easy insertions/deletions. They adapt to varying storage needs without fixed size constraints.

33. How is the top element accessed in an array-based stack?

Access the top element by using the top index to retrieve the value from the array. This provides constant time access to the stack's top element.

34. Explain the dynamic nature of a linked list implementation of a stack.

Linked list stacks grow and shrink as needed, allocating memory dynamically. They do not require pre-defined size and efficiently use memory based on current storage needs.

35. What is a real-world application of stacks?

Stacks are used in function call management, where each function call is pushed onto the call stack and popped when the function returns. They manage execution order and local variables.

36. How are stacks used in function calls in programming languages?

Function calls are managed using a call stack, storing return addresses and local variables. Each call pushes a new frame onto the stack, and returns pop the frame, resuming execution.

37. Explain how stacks can be used for expression evaluation.

Stacks evaluate expressions by handling operators and operands. Operands are pushed, and operators process the top elements, pushing results back for further operations, enabling efficient evaluation.

38. What role do stacks play in undo mechanisms in software applications?

Stacks track changes for undo operations by storing previous states. Each action pushes a state onto the stack, and undo pops the stack to revert to the previous state.

39. What is a queue?

A queue is a linear data structure following the First-In-First-Out (FIFO) principle. Elements are added (enqueued) at the rear and removed (dequeued) from the front.

40. Describe the FIFO principle.

FIFO (First-In-First-Out) means the first element added to the queue is the first one to be removed. It ensures ordered processing, like waiting in line.

41. List the basic operations of a queue.

Basic operations include ENQUEUE (inserting an element at the rear), DEQUEUE (removing an element from the front), PEEK (accessing the front element without removing it), and checking if the queue is empty or full.

42. How is a queue different from a stack?

A queue follows the FIFO (First-In-First-Out) principle, while a stack follows the LIFO (Last-In-First-Out) principle. Queues are used for ordered processing, whereas stacks are typically used for function call management and undo mechanisms.

43. Explain how a circular queue works.

A circular queue is a queue structure implemented using a fixed-size array where the rear and front pointers wrap around when they reach the end of the array, enabling efficient use of space and continuous insertion/deletion operations.

44. Describe the ENQUEUE operation in a queue.

ENQUEUE adds an element to the rear of the queue. It involves incrementing the rear index (or adjusting pointers in a linked list implementation) and placing the element at that position.

45. What is the DEQUEUE operation in a queue?

DEQUEUE removes the front element from the queue. It involves incrementing the front index (or adjusting pointers in a linked list implementation) and returning or removing the element from that position.

46. How do you check if a queue is full?

In array implementation, check if `(rear + 1) % max_size == front`. If true, the queue is full and cannot accommodate more elements without removing some.

47. Explain how to check if a queue is empty.

Check if `front == rear`. If true, the queue is empty and contains no elements to dequeue or peek; otherwise, there are elements in the queue available for operations.

48. What is queue overflow and underflow?

Queue overflow occurs when trying to ENQUEUE an element into a full queue. Queue underflow occurs when trying to DEQUEUE or PEEK from an empty queue.

49. Compare the array and linked list implementations of queues.

Array queues have fixed size, efficient indexing, and potential overflow issues. Linked list queues offer dynamic sizing, efficient insertions/deletions, and no overflow, but use more memory for node pointers.

50. What are the benefits of implementing queues using linked lists?

Linked list queues provide dynamic memory allocation, no size restrictions, efficient insertions/deletions from both ends, and adaptability to varying storage needs without fixed size constraints.

51. What is a dictionary in the context of data structures?

A dictionary is an abstract data type that stores a collection of key-value pairs, allowing efficient insertion, deletion, and lookup operations based on keys.

52. How is a dictionary implemented using a linear list?

A dictionary implemented with a linear list uses an array or linked list where each element is a key-value pair. Operations involve searching by key and updating or deleting elements based on key matches.

53. Explain the concept of a skip list.

A skip list is a probabilistic data structure that allows for quick search, insertion, and deletion operations. It uses multiple levels of linked lists with skip pointers to efficiently skip over nodes during search operations.

54. How does a skip list improve search efficiency?

A skip list improves search efficiency by allowing logarithmic time complexity for search operations similar to balanced trees, but with simpler implementation and efficient space usage.

55. What are the basic operations performed on dictionaries?

Basic operations on dictionaries include insertion (adding a key-value pair), deletion (removing a key-value pair), searching (finding a value based on key), and updating (modifying the value associated with a key).

56. Describe the insertion process in a dictionary using a linear list.

To insert in a linear list dictionary, append a new key-value pair at the end of the list or replace an existing value if the key matches. This operation maintains the order or replaces the old value.

57. How is deletion handled in a skip list?

Deletion in a skip list involves locating the node to delete by traversing through levels using skip pointers, adjusting pointers to bypass the node, and freeing its memory. This maintains skip list properties.

58. What is a hash table?

A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

59. Explain the role of a hash function in a hash table.

A hash function maps keys of arbitrary size to fixed-size values, typically integers, which index into an array of buckets. It determines the location where the key-value pair will be stored or retrieved.

60. What is a collision in the context of hash tables?

A collision occurs in a hash table when two or more keys hash to the same index or bucket. This requires handling to maintain data integrity and efficient access.

61. Describe separate chaining as a method for collision resolution.

Separate chaining resolves collisions by storing multiple elements (collisions) at each index of the hash table as a linked list. Each index points to a linked list of elements that hashed to the same location.

62. How does open addressing differ from separate chaining?

Open addressing resolves collisions by finding an alternative location within the hash table itself (often through probing techniques like linear probing, quadratic probing, or double hashing) to store the colliding element.

63. Explain linear probing in hash tables.

Linear probing is a method of open addressing where collisions are resolved by placing the colliding element in the next available slot sequentially following the hashed index in the table.

64. What is quadratic probing, and how does it work?

Quadratic probing is an open addressing technique where the interval between successive probes is determined by a quadratic function. It reduces clustering and can distribute elements more evenly across the hash table.

65. Describe double hashing as a collision resolution technique.

Double hashing uses a secondary hash function to determine the step size for probing. It offers better performance than linear probing and reduces clustering of elements in the hash table.

66. What is rehashing in hash tables?

Rehashing is the process of creating a new hash table with a larger size and re-inserting all existing key-value pairs into the new table. It helps reduce collisions and maintain efficient operations as the hash table grows.

67. Explain the concept of extendible hashing.

Extendible hashing dynamically adjusts the size of the hash table by splitting or merging buckets based on the hash value of keys. It minimizes collisions and provides efficient lookup times.

68. Compare linear probing and quadratic probing in terms of efficiency.

Linear probing has simpler implementation but can suffer from clustering. Quadratic probing reduces clustering but requires more computation for collision resolution, potentially impacting performance.

69. How does separate chaining handle collisions differently from open addressing?

Separate chaining stores collided elements in separate data structures (linked lists) at the same index in the hash table. Open addressing attempts to find alternative locations within the table for collided elements.

70. What are the advantages of using a hash table for dictionary operations?

Hash tables offer average constant-time complexity for search, insertion, and deletion operations, making them efficient for managing large datasets with dynamic keys and values.

71. How do you choose a good hash function?

A good hash function minimizes collisions by evenly distributing keys across the hash table. It should be fast to compute, deterministic, and generate uniformly distributed hash codes for different inputs.

72. What are the challenges associated with hashing?

Challenges include collision resolution, choosing an appropriate hash function, handling resizing and rehashing, and ensuring performance under varying load factors and input distributions.

73. How can hash tables be resized, and why is this important?

Hash tables are resized by creating a new, larger table and rehashing all existing elements into the new table. Resizing is important to maintain efficient performance as the number of elements or load factor changes.

74. Describe a real-world application of hash tables.

Hash tables are used in databases for indexing and fast retrieval of records, in compilers for symbol tables, in caching mechanisms for quick data access, and in implementing associative arrays in various programming languages.

75. How does a hash table perform insertion operations?

Insertions in a hash table involve computing the hash value of the key, mapping it to a bucket, and inserting the key-value pair into that bucket. Collision handling ensures data integrity and efficient storage.

76. Explain the deletion process in a hash table.

Deletion in a hash table involves locating the key-value pair using its hash value, removing it from the corresponding bucket, and adjusting any necessary pointers or structures to maintain integrity.

77. How is searching implemented in a hash table?

Searching in a hash table involves computing the hash value of the key, accessing the corresponding bucket, and comparing keys until a match is found or the end of the bucket (or linked list) is reached.

78. What is the load factor in the context of hash tables?

The load factor is the ratio of the number of stored elements to the size of the hash table. It affects performance; higher load factors increase the likelihood of collisions, impacting efficiency.

79. How does the load factor affect a hash table's performance?

A higher load factor increases the number of collisions, potentially degrading performance as search, insertion, and deletion operations take longer. A lower load factor reduces collisions but increases memory usage.

80. Describe how extendible hashing dynamically adjusts to the data set size.

Extendible hashing dynamically splits or merges buckets based on the hash values of keys. It adapts the hash table size to maintain efficient performance as the number of elements grows or shrinks.

81. What is the significance of choosing the right probing sequence in open addressing?

Choosing the right probing sequence (linear, quadratic, double hashing) affects how efficiently collisions are resolved and how evenly elements are distributed across the hash table, impacting overall performance.

82. How does double hashing minimize clustering in hash tables?

Double hashing reduces clustering by using a secondary hash function to determine the step size for probing. It can distribute elements more evenly across the hash table, reducing collisions.

83. Compare the efficiency of separate chaining and open addressing for different load factors.

Separate chaining is generally more efficient for higher load factors as it avoids clustering and maintains constant-time operations. Open addressing can be more space-efficient for lower load factors but may suffer from clustering.

84. Why might a skip list be preferred over a traditional linked list for dictionary implementations?

Skip lists offer logarithmic time complexity for search, insertion, and deletion operations similar to balanced trees but with simpler implementation and efficient use of memory, making them suitable for dictionary implementations.

85. Describe an instance where rehashing would be necessary in a hash table.

Rehashing becomes necessary when the load factor of the hash table exceeds a certain threshold, indicating an increase in collisions and degraded performance. It involves resizing the table to reduce collisions.

86. How can extendible hashing be beneficial for databases?

Extendible hashing allows databases to dynamically adjust storage size, minimizing disk accesses and improving query performance by maintaining efficient data organization based on hash values.

87. What is spatial locality, and why is it important in the context of hashing?

Spatial locality refers to the tendency of data accessed together to be stored together. In hashing, it ensures that elements with similar hash values are stored close together, optimizing memory access patterns.

88. How does a hash table contribute to the efficiency of data retrieval?

Hash tables provide average constant-time complexity for retrieval operations, ensuring quick access to stored elements based on their keys. This efficiency makes them suitable for applications requiring fast data access.

89. Describe the process of key transformation in hashing.

Key transformation converts keys of variable length or content into fixed-length values (hash codes) using hash functions. This process ensures uniform distribution and efficient storage in hash tables.

90. How does a hash table support quick insertion and deletion?

A hash table computes hash values for keys, maps them to bucket indices, and stores key-value pairs in buckets. Insertions and deletions are quick as they involve accessing buckets directly, typically in constant time.

91. What is the impact of hash function selection on collision frequency?

A well-chosen hash function minimizes the likelihood of generating the same hash value for different keys (collisions). It evenly distributes keys across the hash table, reducing collision frequency and improving performance.

92. Explain the advantage of quadratic probing over linear probing.

Quadratic probing reduces clustering by using a quadratic function to determine step sizes, distributing elements more evenly across the hash table compared to linear probing, which has a fixed step size.

93. How do dynamic hashing techniques like extendible hashing work?

Dynamic hashing adjusts hash table size dynamically by splitting or merging buckets based on hash values of keys. It ensures efficient collision resolution and maintains performance as data set sizes change.

94. What are the trade-offs involved in selecting a hash table size?

Choosing a larger hash table size reduces collisions and improves performance but increases memory usage. Smaller sizes save memory but may increase collisions and degrade performance for larger data sets.

95. How does separate chaining allow for the direct addressing of collisions?

Separate chaining stores collided elements in linked lists at the same index in the hash table. It directly accesses these lists to handle collisions, maintaining efficient operations even with higher load factors.

96. Describe how a dictionary can be implemented in a distributed system.

In a distributed system, a dictionary can be implemented using distributed hash tables (DHTs). Key-value pairs are distributed across multiple nodes using consistent hashing to ensure load balancing and fault tolerance.

97. Explain the advantages of using a skip list for dictionary implementations with large data sets.

Skip lists offer efficient logarithmic search, insertion, and deletion operations, similar to balanced trees but with simpler implementation and less memory overhead, making them suitable for large-scale dictionary implementations.

98. How does hashing facilitate faster search operations compared to other data structures?

Hashing computes direct indices (hash values) for keys, enabling constant-time average search operations. This efficiency makes hashing ideal for applications requiring quick access to stored data based on unique identifiers.

99. What strategies can be employed to reduce the impact of collisions in a hash table?

Strategies include choosing an effective hash function, resizing the hash table dynamically (rehashing), using collision resolution techniques like separate chaining or open addressing, and maintaining an optimal load factor.

100. Compare the performance implications of different collision resolution techniques.

Collision resolution techniques impact performance by affecting average time complexity, memory usage, and implementation complexity. Techniques like separate chaining and double hashing offer different trade-offs in handling collisions effectively.

101. Define a binary search tree (BST).

A binary search tree (BST) is a binary tree where each node has at most two children, known as the left child and right child. The key (or value) of each node is greater than the keys in its left subtree and less than the keys in its right subtree.

102. How is a binary search tree implemented?

A BST is implemented using a node structure where each node contains data (key/value), pointers to its left and right children (which are also BST nodes), and an optional pointer to its parent node.

103. Describe the process of searching for an element in a BST.

To search in a BST, compare the target key with the current node's key. If they match, the node is found. If the target key is less, search in the left subtree; if greater, search in the right subtree. Repeat until the node is found or the subtree is null.

104. Explain how insertion is performed in a BST.

Insertion in a BST starts from the root node. Compare the key of the new node with the current node's key. If it is less, move to the left subtree; if greater, move to the right subtree. Repeat until an appropriate null position is found, then insert the new node.

105. How is a node deleted from a BST?

Deleting a node in a BST involves three cases:

Case 1: Node has no children (leaf node): Simply remove the node.

Case 2: Node has one child: Replace the node with its child.

Case 3: Node has two children: Find the inorder successor (or predecessor), copy its value to the node to be deleted, and recursively delete the successor node.

106. What is a B-Tree and how does it differ from a BST?

A B-Tree is a self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and searching operations. Unlike BSTs, B-Trees have multiple keys per node (typically more than two), providing better balance and handling large datasets.

107. Explain the structure of a B+ Tree.

A B+ Tree is a variant of a B-Tree where:

All keys are present in leaf nodes.

Leaf nodes are linked together.

Non-leaf nodes (internal nodes) store keys to guide searches.

This structure enhances range queries and sequential access.

108. What are the advantages of using AVL trees over BSTs?

AVL trees are self-balancing binary search trees where the height difference between left and right subtrees (balance factor) is at most one. This balance ensures O(log n) time complexity for search, insert, and delete operations, unlike BSTs that can degenerate into linear structures.

109. Define the height of an AVL tree.

The height of an AVL tree is the maximum number of edges from the root to any leaf node. It ensures the tree remains balanced by maintaining a balance factor of -1, 0, or 1 for every node.

110. Describe the insertion process in an AVL tree.

Insertion in an AVL tree involves:

Performing a standard BST insertion.

Updating the balance factor of nodes from the inserted node to the root.

Rotating the tree if necessary to maintain AVL balance criteria.

This ensures the tree remains balanced after insertion.

111. How does deletion work in an AVL tree?

Deletion in an AVL tree involves:

Performing a standard BST deletion.

Updating the balance factor of nodes from the deleted node to the root.

Rotating the tree if necessary to restore AVL balance.

This ensures the tree remains balanced after deletion.

112. What is a Red-Black Tree?

A Red-Black Tree is a self-balancing binary search tree where each node contains an extra bit for color (red or black). It maintains balanced properties:

Every node is either red or black.

Root is black.

No two adjacent red nodes.

Every path from a node to its descendant leaves has the same number of black nodes.

113. Explain the properties of a Red-Black Tree.

Red-Black Trees maintain balanced properties ensuring O(log n) time complexity for operations:

Root is black.

No two adjacent red nodes.

Every path from root to null nodes has the same number of black nodes.

These properties enforce balanced structure during insertions and deletions.

114. How do you insert a node into a Red-Black Tree?

Insertion in a Red-Black Tree involves:

Inserting the node as in a BST.

Coloring the node red.

Adjusting the tree to maintain Red-Black properties (rotations and color flips).

This ensures the tree remains balanced and maintains Red-Black properties.

115. Describe the deletion process in a Red-Black Tree.

Deletion in a Red-Black Tree involves:

Performing a standard BST deletion.

Replacing the node with its successor/predecessor.

Adjusting colors and performing rotations to maintain Red-Black properties.

This ensures the tree remains balanced and adheres to Red-Black properties.

116. Define what a Splay Tree is.

A Splay Tree is a self-adjusting binary search tree where recently accessed elements are moved closer to the root. It optimizes access times for frequently accessed elements by restructuring the tree based on access patterns.

117. Explain the splaying operation in Splay Trees.

Splaying in a Splay Tree moves a recently accessed node to the root using rotations. It involves single rotations, double rotations, and zig-zag or zig-zig restructurings to maintain the tree's balance and optimize access times.

118. How does searching work in a Splay Tree?

Searching in a Splay Tree involves:

Performing a standard BST search.

Splaying the accessed node to the root to optimize future accesses.

This improves access times for frequently accessed elements.

119. Compare the insertion process in BSTs and AVL trees.

BST insertion is straightforward but can lead to unbalanced trees. AVL tree insertion maintains balance by performing rotations to ensure the tree's height remains logarithmic, ensuring O(log n) operations.

120. How do B-Trees handle large amounts of data efficiently?

B-Trees handle large data sets efficiently by:

Storing multiple keys per node.

Balancing the tree after insertions and deletions.

Minimizing disk accesses and maintaining order for efficient range queries and data retrieval.

This structure makes them ideal for databases and filesystems.

121. Describe the balancing mechanism of AVL trees.

AVL trees balance themselves by maintaining a balance factor (-1, 0, or 1) for every node. After insertions or deletions that disturb balance, rotations (single or double) are performed to restore AVL properties, ensuring logarithmic time complexity for operations.

122. What makes Red-Black Trees unique in handling insertions and deletions?

Red-Black Trees handle insertions and deletions by maintaining color properties (red or black) and performing rotations to restore balance. They ensure O(log n) time complexity for operations, making them efficient for dynamic data structures.

123. How do B+ Trees improve upon B-Trees in terms of disk access?

B+ Trees improve upon B-Trees by storing all keys in leaf nodes and linking leaf nodes together. This structure enhances sequential access and range queries by reducing disk seeks, making them efficient for database indexing.

124. Explain the concept of rotation in AVL tree balancing.

Rotation in AVL trees is a technique to maintain balance after insertions or deletions. Single rotations (left or right) and double rotations (left-right or right-left) adjust node positions while preserving BST properties and balancing factors.

125. What is the difference between AVL trees and Splay Trees in terms of balancing?

AVL trees maintain strict balance using rotations to ensure logarithmic time complexity for operations. Splay Trees adaptively adjust structure based on access patterns, bringing frequently accessed nodes closer to the root for optimized access times.