Array Vs Linked List

Array	Linked List
A collection of elements stored in contiguous memory locations.	A collection of nodes where each node contains data and a reference (or pointer) to the next node.
Fixed size (must be defined at the time of creation).	Dynamic size (can grow or shrink at runtime).
Contiguous memory allocation.	Non-contiguous memory allocation (nodes scattered in memory).
Fast random access using an index (O(1)).	Slow access (O(n)), must traverse the list to access an element.
Expensive for large arrays (O(n)) as elements need to be shifted.	Efficient (O(1) for insertion/deletion at the beginning), but O(n) for searching or modifying nodes.
Requires less memory (only stores the data).	Requires more memory (stores both data and pointers to other nodes).
Simple to use for basic operations and fixed-size data.	More complex due to pointer manipulation.
Cache-friendly due to contiguous memory storage.	Not cache-friendly, as nodes are scattered in memory.
int arr[5] = {1, 2, 3, 4, 5};	Node* head = new Node(1); head->next = new Node(2);

Curious_.Programmer

Liner Search Vs Binary Search

Linear Search	Binary Search
Searches for an element by checking each element one by one.	Searches by repeatedly dividing the search interval in half.
Works on both unsorted and sorted arrays.	Requires the array to be sorted.
O(n), where n is the number of elements.	O(log n), where n is the number of elements.
Sequentially checks each element until the target is found.	Divides the array into two halves and eliminates half in each step.
O(1) if the element is at the first position.	O(1) if the middle element is the target.
O(n) when the element is at the last position or not present.	O(log n) in the worst case.
Can be used on any array, sorted or unsorted.	Can only be used on sorted arrays.
Simpler to implement but slower for large datasets.	Faster for large datasets but requires sorting.
Searching for a contact in an unsorted phone book by going page by page.	Searching for a word in a dictionary by looking in the middle and narrowing the search.

Curious_.Programmer

Merge Sort vs Quick Sort

Merge Sort	Quick Sort
A divide-and-conquer algorithm that splits the array into two halves, sorts them, and merges them back together.	A divide-and-conquer algorithm that selects a pivot, partitions the array, and sorts the subarrays.
O(n log n) in all cases (best, average, worst).	O(n log n) on average, but $O(n^2)$ in the worst case.
O(n), requires extra space for merging.	O(log n) due to recursion, but no extra space for partitioning.
Stable (maintains relative order of equal elements).	Not stable (may not maintain relative order of equal elements).
Preferred for large datasets or linked lists due to stability and guaranteed performance.	Preferred for smaller datasets or when speed is more important and the worst case can be avoided.
Merges sorted subarrays back together.	Partitions around a pivot and sorts smaller parts recursively.
Always O(n log n).	$O(n^2)$ when pivot selection is poor (e.g., always picking the smallest or largest element as pivot).
Not in-place (requires additional space).	In-place (sorts within the original array).

Curious_.Programmer

Singly Linked List vs Doubly Linked List

Singly Linked List	Doubly Linked List
A linked list where each node has a reference (pointer) to the next node only.	A linked list where each node has references (pointers) to both the next and previous nodes.
Can be traversed in one direction (forward only).	Can be traversed in both directions (forward and backward).
Uses less memory as each node stores only one pointer (to the next node).	Uses more memory as each node stores two pointers (to both next and previous nodes).
Efficient for insertion/deletion at the head (O(1)), but less efficient at the tail or middle (O(n)).	More efficient for insertion/deletion at both ends (head and tail) or in the middle (O(1) for both directions).
Simpler to implement and manage.	More complex due to extra pointers and two-way traversal.
Suitable for simple use cases where only forward traversal is needed.	Suitable for scenarios where bidirectional traversal is required, such as navigating back and forth.
Not possible without additional operations (like recursion or re-traversing the list).	Can be easily traversed in reverse direction.
A single-lane road where cars can only move forward.	A two-way street where cars can move in both directions.

Curious_.Programmer

Array Vs Stack

Array	Stack
A collection of elements stored in contiguous memory locations.	A linear data structure that follows the LIFO (Last In, First Out) principle.
Can access any element directly using an index (random access).	Can only access the top element (no direct access to other elements).
Elements can be added or removed at any position using an index.	Elements are added and removed only at the top.
Maintains the order of insertion but allows random access.	Follows the LIFO order (last element added is the first to be removed).
Accessing an element: O(1), Insertion/Deletion: O(n) (in the worst case, if shifting is needed).	Push and pop operations are O(1), since they occur only at the top.
Fixed size or dynamic (depending on the language and implementation).	Dynamic in size, growing as elements are added.
Used for storing multiple elements where random access is required.	Used for managing function calls, undo operations, and evaluating expressions.
A list of songs where you can play any song at any time.	A stack of plates: the last plate placed is the first to be removed.

HashMap vs TreeMap

HashMap	TreeMap
Uses a hash table.	Uses a Red-Black Tree (a balanced binary search tree).
Does not maintain any specific order of keys.	Maintains the natural order of keys (or a custom order via comparator).
O(1) on average for put(), get(), and remove() (constant time).	O(log n) for put(), get(), and remove() (logarithmic time).
Allows one null key and multiple null values.	Does not allow null keys, but allows multiple null values.
Generally faster due to constant time operations.	Slightly slower because of tree balancing (logarithmic time).
Use when fast access with no order is needed.	Use when sorted order of keys is important.
Not sorted.	Automatically sorts by keys (natural or custom order).
A dictionary where you don't care about the order of words.	A phone book where entries are sorted by names.

BFS vs DFS

Breadth-First Search (BFS)	Depth-First Search (DFS)
Explores all nodes at the present depth level before moving on to nodes at the next depth level.	Explores as far as possible along one branch before backtracking.
Uses a queue to keep track of nodes to explore.	Uses a stack (or recursion) to explore nodes.
Level-by-level (breadth-wise).	Depth-wise (explores one path fully before backtracking).
Queue.	Stack (or recursion).
O(V + E), where V is vertices and E is edges.	O(V + E), where V is vertices and E is edges.
O(V), since it needs to store all vertices in memory at each level.	O(V), due to recursion or stack space.
When you want the shortest path in an unweighted graph.	When you want to explore all possible paths in a graph.
Finding shortest paths, peer-to-peer networks, and social networks.	Solving puzzles, detecting cycles, and maze traversal.
No backtracking; explores all neighbors before moving on.	Involves backtracking when a dead-end is reached.
Finding people level by level in a social network.	Exploring a maze by going down one path fully before trying another.

Curious_.Programmer

Binary Tree Vs Binary Search Tree (BST)

Binary Tree	Binary Search Tree (BST)
A tree data structure where each node has at most two children (left and right).	A special type of binary tree where the left child contains values smaller than the parent, and the right child contains values greater than the parent.
No specific order between parent and child nodes.	Maintains a specific order: left child < parent < right child.
Can contain duplicate values.	Usually does not allow duplicate values (depends on implementation).
O(n) in the worst case, as no particular order is maintained.	O(log n) in the best/average case (if balanced), O(n) in the worst case (if unbalanced).
Can become unbalanced, leading to poor performance.	Should ideally be balanced (e.g., AVL or Red-Black trees) to ensure optimal performance.
No specific traversal order is enforced.	In-order traversal results in sorted data (left to right).
Used for representing hierarchical data structures where order is not important.	Used for efficient searching, insertion, and deletion when order of elements is important.
Representing general tree structures like organizational charts, file systems.	Used in applications requiring sorted data or fast lookup, such as databases and search engines.
A family tree where the order of children doesn't matter.	A sorted bookshelf where books are arranged by title.

Curious_.Programmer

Stack Vs Queue

Stack	Queue
A linear data structure that follows LIFO (Last In, First Out) principle.	A linear data structure that follows FIFO (First In, First Out) principle.
Elements are added and removed from the same end (called the top).	Elements are added at the rear and removed from the front.
Push (insert) and Pop (remove) operations happen at the top.	Enqueue (insert) happens at the rear, and Dequeue (remove) happens at the front.
Last element added is the first to be removed.	First element added is the first to be removed.
Used for problems like recursion, backtracking, and function call management.	Used in scheduling, buffering, and managing tasks in the order they arrive.
A stack of plates: the last plate added is the first to be taken out.	A queue in a supermarket: the first person in line is the first to be served.
push(), pop(), peek()	enqueue(), dequeue(), peek()
O(1) for push and pop operations.	O(1) for enqueue and dequeue operations.

Curious_.Programmer

HashMap Vs TreeMap

HashMap	TreeMap
Uses a hash table.	Uses a Red-Black Tree (self-balancing binary search tree).
Does not maintain any specific order of keys.	Maintains the natural order of keys (or custom order using a comparator).
O(1) on average for put(), get(), and remove() (constant time).	O(log n) for put(), get(), and remove() (logarithmic time).
Allows one null key and multiple null values.	Does not allow null keys but allows multiple null values.
Generally faster due to constant time operations.	Slightly slower due to tree balancing overhead.
Use when fast access without sorting is needed.	Use when sorted order of keys is important.
Unordered, no sorting applied.	Automatically sorts keys by their natural order (or custom order).
Less memory usage, as no tree structure is needed.	More memory usage due to maintaining the tree structure.
A dictionary where you don't care about the order of words.	A sorted phone book where entries are kept in alphabetical order.

Curious_.Programmer