calm the mind

IPU NOTES

B.TECH CSE

3RD SEM

SNAPED
COMMUNITY

DATA STRUCTURES

MidTerm Solution

"MINDSET FIRST, ACTION SECOND"

JOIN US:

MidTerm Solution			
Subject	**Data Structures**	**Code**	**CIC-209**
Community	**SnapED codeCampus**		

Ques 1: (a) Illustrate sequential search.

(b) Explain merge sort with an example.

(c) Illustrate sorting concept with an example.

(d) Explain spanning tree property with an example.

Ans-1:

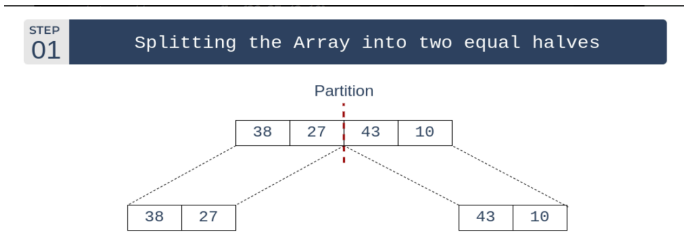
(a) Sequential search, also known as linear search, is a simple searching algorithm that finds the position of a target value within a list. It sequentially checks each element of the list until a match is found or the entire list has been searched.

(b) Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

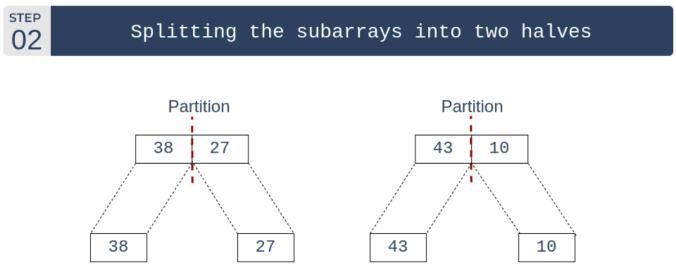
For Example:-

consider an array arr[] = {38, 27, 43, 10}

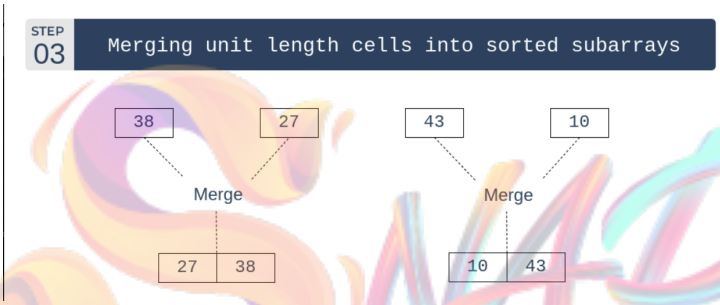
- Initially divide the array into two equal halves:



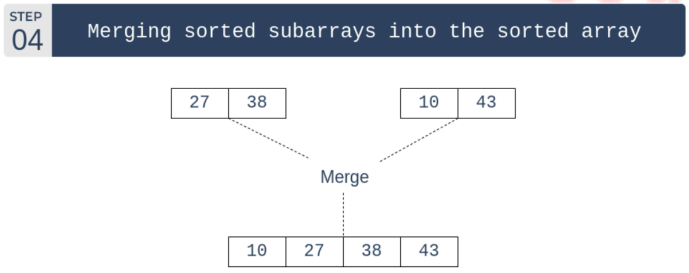
- These subarrays are further divided into two halves. Now they become arrays of unit length that can no longer be divided and array of unit length are always sorted.



- These sorted subarrays are merged together, and we get bigger sorted subarrays.



- This merging process is continued until the sorted array is built from the smaller subarrays.



(c) Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

For Example:-

Consider the following array as an example: arr[] = {64, 25, 12, 22, 11}

First pass:

For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing the whole array it is clear that 11 is the lowest value.

64 25 12 22 11

Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

11 25 12 22 64

Second Pass:

For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

11 25 12 22 64

After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

11 12 25 22 64

Third Pass:

Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array.

11 12 25 22 64

While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.

11 12 22 25 64

Fourth pass:

Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array

As 25 is the 4th lowest value hence, it will place at the fourth position.

11 12 22 25 64

Fifth Pass:

At last the largest value present in the array automatically get placed at the last position in the array

The resulting array is the sorted array.

11 12 22 25 64

(d) The Spanning Tree Property is a fundamental concept in graph theory, particularly in the context of network design and communication. In a connected, undirected graph, a spanning tree is a subgraph that includes all the vertices of the original graph and forms a tree (i.e., it is acyclic and connected). The Spanning Tree Property states that every connected, undirected graph has at least one spanning tree.

❖ Spanning Tree Property:

- **Connected:** All vertices in G are connected in T.
- **Acyclic:** T is acyclic, meaning it contains no cycles.
- **Includes all vertices:** T includes all vertices of G.
- **Minimally connected:** If any edge is removed from T, it becomes disconnected.

Ques-2: (a) Compare merge sort, insertion sort and selection sort.Also discuss the role of sorting in data structure in terms of space and time complexity.

(b)Discuss the shortest path algorithm with an example.

Ans-2:

(a)

	Selection Sort	Insertion Sort	Merge Sort
Algorithm	It iteratively selects the minimum element from the unsorted part and swaps it with the first unsorted element, gradually building the sorted part.	It builds the sorted array one element at a time by repeatedly taking elements from the unsorted part and inserting them into their correct position in the sorted part.	It follows the divide-and-conquer paradigm, dividing the array into halves, recursively sorting each half, and then merging them.
Time Complexity	O(n^2) in the worst, average, and best cases. It always performs the same number of comparisons and swaps.	O(n^2) in the worst and average cases, O(n) in the best case for already sorted arrays.	O(n log n) in the worst, average, and best cases.
Space Complexity	O(1) as it requires a constant amount of additional space.	O(1) as it requires a constant amount of additional space.	O(n) due to the need for additional space for merging.

- Role of Sorting in Data Structures:

1. Search Operations: Sorted data allows for more efficient search operations. Techniques like binary search can be applied to sorted data, reducing search time complexity to O(log n).

2.Efficient Merging: Sorting is often a fundamental step in various algorithms and data structures. For example, in merge sort, sorting is explicitly used during the merging step.

3. Data Retrieval: For applications where data retrieval in a specific order is necessary, sorting becomes essential. For instance, a sorted list of names allows for quick alphabetical retrieval.

4. Enhanced Performance: Many algorithms and data structures perform better or have simpler implementations when applied to sorted data. For instance, priority queues and binary search trees leverage the ordered nature of sorted data.

Space and Time Complexity Considerations:

Time Complexity: Sorting algorithms have different time complexities, influencing their suitability for different data sizes and contexts. Merge sort, with a time complexity of O(n log n), is generally more efficient for large datasets compared to O(n^2) algorithms like insertion sort and selection sort.

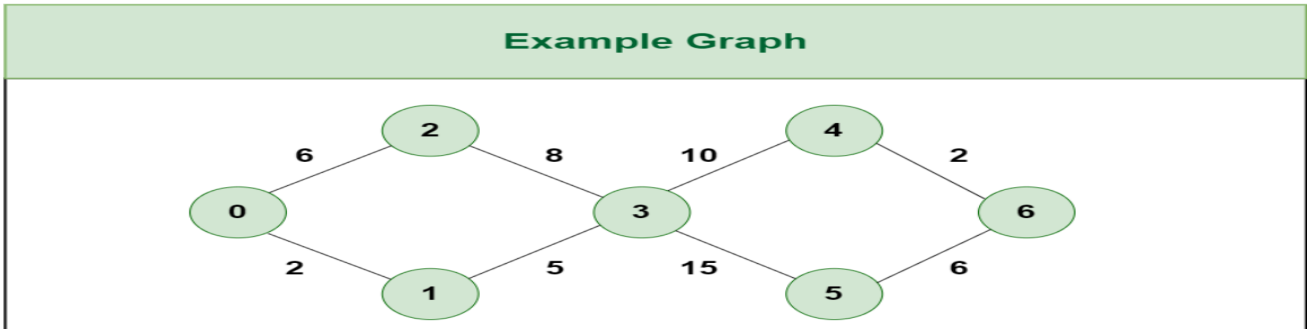
Space Complexity: The space complexity of sorting algorithms varies as well. Merge sort requires additional space for merging, making it less memory-efficient than in-place algorithms like insertion sort and selection sort, which have O(1) space complexity.

(b) Dijkstra's algorithm is a popular algorithm for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956.

Algorithm for Dijkstra's Algorithm:

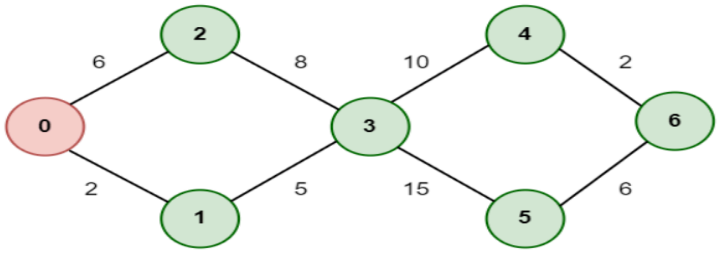
1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes that are unvisited.

For Example:-



STEP 1

Start from Node 0 and mark Node 0 as Visited and check for adjacent nodes

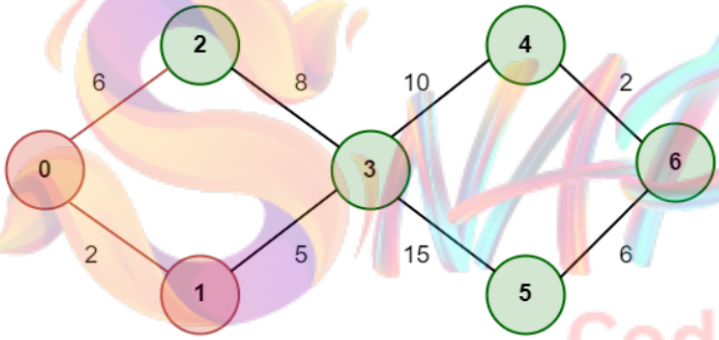


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: ∞
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

STEP 2

Mark Node 1 as Visited and add the Distance

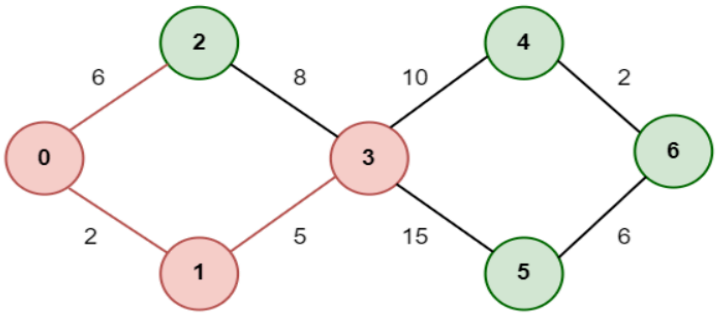


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: 2 ✓
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

STEP 3

Mark Node 3 as Visited after considering the Optimal path and add the Distance

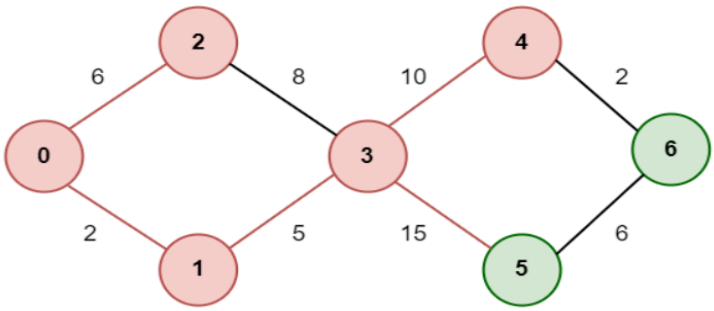


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✓
1: 2 ✓
2: 6 ✓
3: 7 ✓
4: ∞
5: ∞
6: ∞

STEP 4

Mark Node 4 as Visited after considering the Optimal path and add the Distance

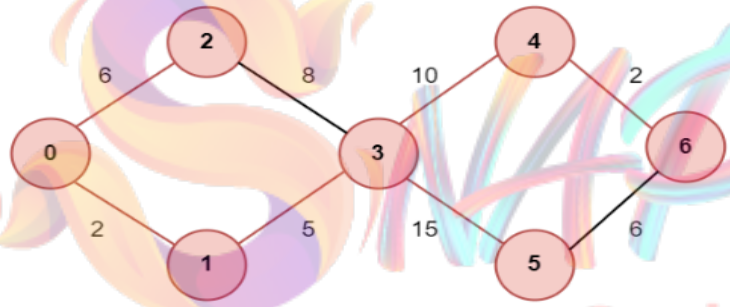


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
- 0: 0 ✓
- 1: 2 ✓
- 2: 6 ✓
- 3: 7 ✓
- 4: 17 ✓
- 5: ∞
- 6: ∞

STEP 5

Mark Node 6 as Visited and add the Distance



Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
- 0: 0 ✓
- 1: 2 ✓
- 2: 6 ✓
- 3: 7 ✓
- 4: 17 ✓
- 5: 22 ✓
- 6: 19 ✓

Ques-3: (a)Implement an algorithm for binary search.Search item 23 from the following sorted elements from the binary search:

2,5,8,12,16,23,38,56,72,91

(b)Implement an algorithm of quick sort with an example.

Ans-3:

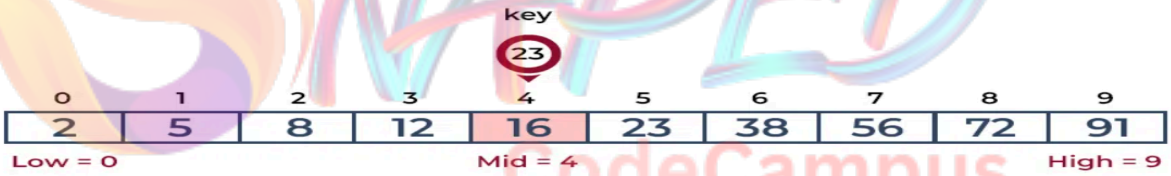
(a) Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

An array arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, and the target = 23.

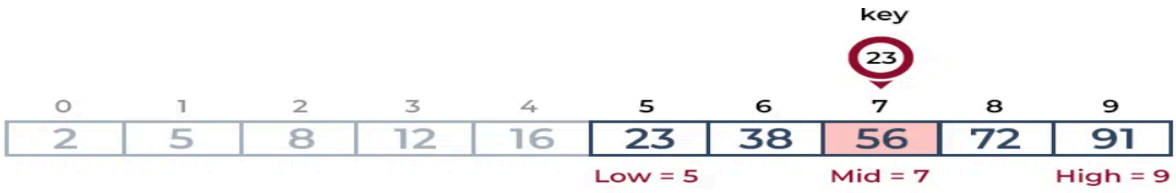
First Step:

Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.

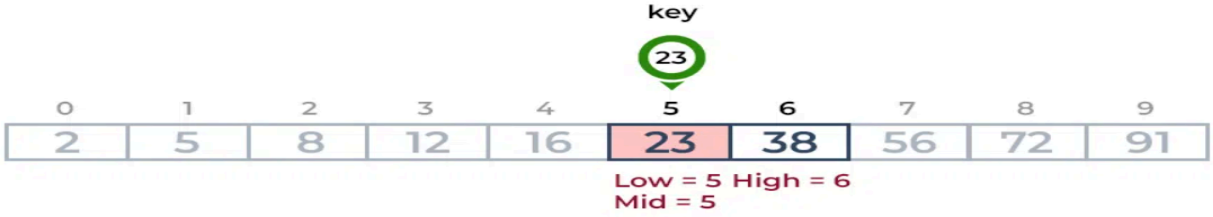
Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.



Key is less than the current mid 56. The search space moves to the left.



Second Step: If the key matches the value of the mid element, the element is found and stops searching.



(b) QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Algorithm:-

QUICKSORT (array A, start, end)

{

1 if (start < end)

2 {

3 p = partition(A, start, end)

4 QUICKSORT (A, start, p - 1)

5 QUICKSORT (A, p + 1, end)

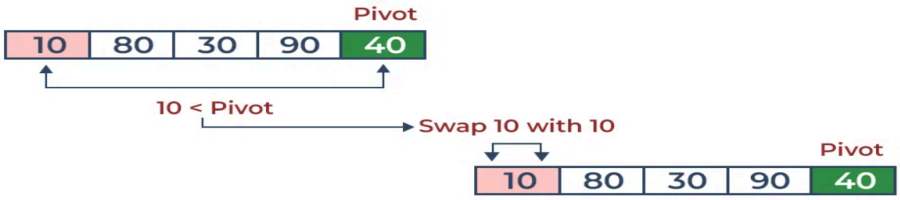
6 }

}

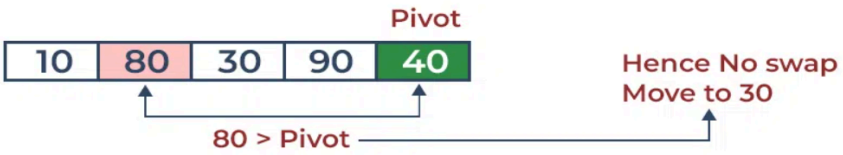
For Example:-

Consider: arr[] = {10, 80, 30, 90, 40}.

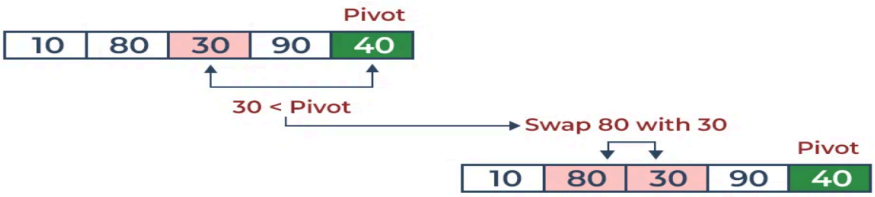
1. Compare 10 with the pivot and as it is less than pivot arrange it accordingly.



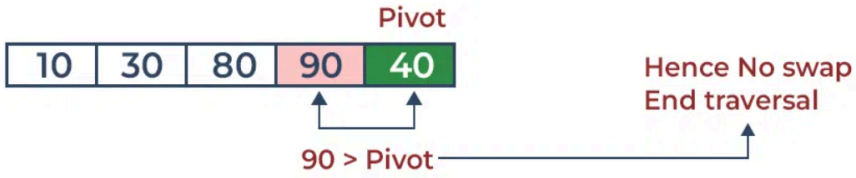
2. Compare 80 with the pivot. It is greater than pivot.



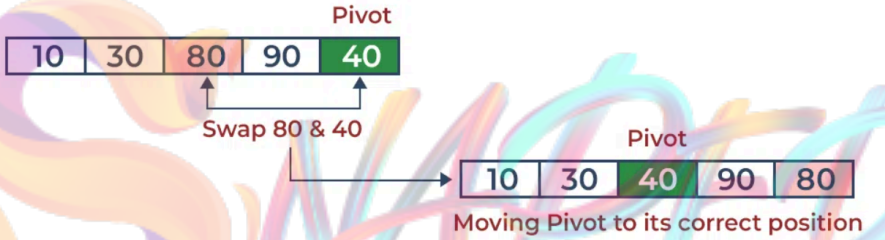
3. Compare 30 with pivot. It is less than pivot so arrange it accordingly.



4. Compare 90 with the pivot. It is greater than the pivot.



5. Arrange the pivot in its correct position.



Ques-4: (a) Compare different hashing techniques to avoid collision

(b) Compare BFS and DFS Graph traversal based on the searching techniques.

Ans-4:

(a) While collisions are inevitable in hashing, various techniques aim to minimize their occurrence and efficiently handle them. Here's a comparison of some popular techniques:

Collision Resolution Techniques:

(i) Open Hashing (Separate Chaining):

Concept: Stores data in linked lists at each hash table index.

Advantages:

Handles collisions efficiently without affecting other elements.

Easy to implement and understand.

Disadvantages:

Requires additional memory for the linked lists.

Searching for specific data within a chain can be slower than open addressing.

(ii) Closed Hashing (Open Addressing):

Concept: Probes for an empty slot in the hash table when a collision occurs.

Advantages:

Requires less memory than separate chaining.

Disadvantages:

Collision resolution can be time-consuming, especially with a high load factor.

Clustering can occur, leading to performance degradation.

Open Addressing Techniques:

Linear Probing:

Concept: Probes adjacent slots until an empty one is found.

Advantages:

Simple to implement.

Performs well with low load factors.

Disadvantages:

* Can lead to clustering, especially with high load factors.

* Worst-case performance can be poor.

Quadratic Probing:

Concept: Probes slots using a quadratic function to avoid clustering.

Advantages:

Performs better than linear probing with high load factors.

Avoids clustering effectively.

Disadvantages:

More complex to implement than linear probing.

(iii)Double Hashing:

Concept: Uses two hash functions to probe for an empty slot.

Advantages:

Performs well with high load factors and avoids clustering.

More flexible than linear or quadratic probing.

Disadvantages:

Requires two hash functions, increasing computational cost.

Other Techniques:

(iv) Perfect Hashing:

Concept: Creates a custom hash function that maps each key to a unique index.

Advantages:

Eliminates collisions entirely.

Can be very fast for specific data sets.

Disadvantages:

Requires complex precomputation to generate the hash function.

Not suitable for dynamic data sets.

(v) Probabilistic Hashing:

Concept: Uses hashing functions with a random component to minimize collisions.

Advantages:

Simple to implement and can handle large data sets.

Provides good collision resolution for moderate load factors.

Disadvantages:

Cannot guarantee zero collisions.

Performance depends on the chosen random function.

(vi) Coalesced Hashing:

Concept: Stores multiple data elements in a single table slot.

Advantages:

Highly efficient for data with similar characteristics.

Can significantly reduce memory usage.

Disadvantages:

Complex to implement and requires specific data types.

Collision resolution can be more involved.

(vii) Choosing the Right Technique:

The choice of hashing technique depends on several factors, including:

Expected load factor: High load factors benefit from techniques like double hashing or probabilistic hashing.

Performance requirements: For time-critical applications, open addressing or perfect hashing might be preferred.

Memory constraints: Separate chaining might not be optimal if memory is limited.

Data characteristics: Techniques like coalesced hashing can be advantageous if data shares certain features.

(b)

	BFS	DFS
Traversal Order	Explores all nodes at the current level before moving to the next level. This results in a "layer-by-layer" exploration.	Explores a single branch as deep as possible before backtracking and exploring other branches. This can lead to a more "zig-zag" traversal.
Data Structure	Utilizes a Queue to store and process nodes. The nodes are added to the back of the queue and processed from the front, ensuring a level-by-level exploration.	Uses a Stack to keep track of explored nodes and backtracking paths. Nodes are pushed onto the stack when explored and popped when backtracking is needed.
Applications	Finding shortest paths in unweighted graphs, finding connected components, checking bipartiteness, finding minimum spanning trees.	Topological sorting, finding cycles in graphs, finding strongly connected components, finding paths in directed acyclic graphs.
Strengths and Weaknesses	Efficient for finding shortest paths in unweighted graphs, lower memory usage, but not ideal for finding cycles in graphs or topological sorting.	Efficient for finding cycles in graphs or topological sorting, but can be inefficient for finding shortest paths and may require higher memory usage.
Choosing the Right Technique	Ideal for finding shortest paths, connected components, bipartiteness, or minimum spanning trees.	Useful for finding cycles, strongly connected components, topological order, or paths in directed acyclic graphs.