1. Explain the algorithm of Selection sort with example.

Selection sort is a simple sorting algorithm that repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted portion of the array and moves it to the beginning. Here's the algorithm for selection sort:

1. **Start**: Begin with the entire array of elements.
2. **Find the minimum**: Scan the array to find the smallest element.
3. **Swap**: Swap the smallest element found in step 2 with the first element of the array.
4. **Repeat**: Repeat steps 2 and 3 for the remaining unsorted portion of the array, excluding the elements that have already been sorted.
5. **End**: Continue this process until the entire array is sorted.

SelectionSort(A)

```
1. for i = 0 to length(A) - 2 do
2.     minIndex = i
3.     for j = i + 1 to length(A) - 1 do
4.         if A[j] < A[minIndex] then
5.             minIndex = j
6.     swap(A[i], A[minIndex])
```

Let's illustrate this with an example:

Consider the array: **[5, 3, 8, 2, 1]**

Initial Array: [5, 3, 8, 2, 1]

Iteration	Unsorted Array	Smallest Element	Swap
1	[5, 3, 8, 2, 1]	1	[1, 3, 8, 2, 5]
2	[3, 8, 2, 5]	2	[2, 8, 3, 5]
3	[8, 3, 5]	3	[3, 8, 5]
4	[8, 5]	5	[5, 8]

2. Explain the algorithm of Bubble sort with example.

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. Here's the algorithm for bubble sort:

1. **Start**: Begin with the entire array of elements.
2. **Pass through the array**: Iterate through the array from the first element to the second-to-last element.
3. **Compare and swap**: Compare each pair of adjacent elements. If they are in the wrong order (e.g., the current element is greater than the next element for ascending order), swap them.
4. **Repeat**: Repeat steps 2 and 3 for each pass through the array until no swaps are needed.
5. **End**: The array is now sorted.

BubbleSort(A)

```
1. for i = 0 to length(A) - 1 do
2.     for j = 0 to length(A) - i - 2 do
3.         if A[j] > A[j + 1] then
4.             swap(A[j], A[j + 1])
```

Initial Array: [5, 3, 8, 2, 1]

Pass	Array	Swaps Made
1	[3, 5, 2, 1, 8]	5 ↔ 3, 8 ↔ 2, 8 ↔ 1
2	[3, 2, 1, 5, 8]	5 ↔ 2, 5 ↔ 1
3	[2, 1, 3, 5, 8]	3 ↔ 2
4	[1, 2, 3, 5, 8]	No swaps

3. Explain the algorithm of Quick sort with example.

Quick sort is a divide-and-conquer algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

1. **Choose a pivot**: Select an element from the array to serve as the pivot. There are various methods for choosing the pivot; commonly used ones include selecting the first element, the last element, or a random element.
2. **Partitioning**: Rearrange the array so that all elements less than the pivot come before it, and all elements greater than the pivot come after it. After this operation, the pivot is in its final sorted position. This is called the partition operation.
3. **Recursively sort sub-arrays**: Apply the above steps recursively to the sub-arrays of elements less than the pivot and elements greater than the pivot.
4. **Combine**: The sub-arrays are combined to form the final sorted array.

QuickSort(A, low, high)

```
1. if low < high then
2.     pivotIndex = Partition(A, low, high)
3.     QuickSort(A, low, pivotIndex - 1)
4.     QuickSort(A, pivotIndex + 1, high)
```

Partition (A, low, high)

```
1. pivot = A[high]
2. i = low - 1
3. for j = low to high - 1 do
4.     if A[j] <= pivot then
5.         i = i + 1
6.         swap A[i] and A[j]
7. swap A[i + 1] and A[high]
8. return i + 1
```

Initial Array: [7, 2, 1, 6, 8, 5, 3, 4]
Pivot: 4
Partitioned Array: [2, 1, 3, 4, 8, 5, 7, 6]
Sub-arrays after Partitioning:

Left Sub-array	Pivot	Right Sub-array
[2, 1, 3]	4	[8, 5, 7, 6]

Sorted Left Sub-array: [1, 2, 3]
Sorted Right Sub-array: [5, 6, 7, 8]
Final Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8]
This is how the quick sort algorithm works. It's efficient for large datasets and has an average time complexity of O(n log n).

4. Explain the algorithm of Merge sort with example.

Merge sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts each half, and then merges the sorted halves to produce a single sorted array.

1. **Divide**: Divide the unsorted array into two halves.
2. **Conquer**: Recursively sort each half.
3. **Combine**: Merge the sorted halves to produce the final sorted array.

Now, let's illustrate this with an example and then represent it in a table format:

Consider the array: **[38, 27, 43, 3, 9, 82, 10]**.

1. **Divide**:
   - Divide the array into two halves: **[38, 27, 43, 3]** and **[9, 82, 10]**.
2. **Conquer**:
   - Recursively sort each half.
   - For **[38, 27, 43, 3]**, we further divide it into **[38, 27]** and **[43, 3]**.
   - For **[9, 82, 10]**, we further divide it into **[9]**, **[82]**, and **[10]**.
3. **Combine**:
   - Merge the sorted halves.
   - For **[38, 27]**, merge them into **[27, 38]**.
   - For **[43, 3]**, merge them into **[3, 43]**.
   - For **[9]**, **[82]**, and **[10]**, they are already sorted.
   - Merge **[27, 38]** and **[3, 43]** into **[3, 27, 38, 43]**.
   - Merge **[9]**, **[82]**, and **[10]** into **[9, 10, 82]**.
   - Finally, merge **[3, 27, 38, 43]** and **[9, 10, 82]** into the final sorted array **[3, 9, 10, 27, 38, 43, 82]**.

MergeSort(A, low, high)

```
1. if low < high then
2.     mid = (low + high) / 2
3.     MergeSort(A, low, mid)
4.     MergeSort(A, mid + 1, high)
5.     Merge(A, low, mid, high)
```

Merge(A, low, mid, high)

```
1. n1 = mid - low + 1
2. n2 = high - mid
3. Let L[1..n1] and R[1..n2] be two temporary arrays
4. for i = 1 to n1 do
5.     L[i] = A[low + i - 1]
6. for j = 1 to n2 do
7.     R[j] = A[mid + j]
```

```
8. L[n1 + 1] = ∞  // Infinity sentinel value
9. R[n2 + 1] = ∞  // Infinity sentinel value
10. i = 1, j = 1
11. for k = low to high do
12.     if L[i] ≤ R[j] then
13.         A[k] = L[i]
14.         i = i + 1
15.     else
16.         A[k] = R[j]
17.         j = j + 1
```

Initial Array: [38, 27, 43, 3, 9, 82, 10]

Step 1: Divide the array into smaller sub-arrays

Divided Arrays	**Merged Arrays**
[38, 27, 43, 3], [9, 82, 10]	[27, 38, 3, 43], [9, 10, 82]
[38, 27], [43, 3] , [9], [82], [10]	[3, 27, 38, 43], [9, 10, 82]

Step 2: Sort each divided sub-array

Divided Arrays	**Merged Arrays**
[27, 38, 3, 43], [9, 10, 82]	[3, 27, 38, 43], [9, 10, 82]

Step 3: Merge sorted sub-arrays

Divided Arrays	**Merged Arrays**
[3, 27, 38, 43], [9, 10, 82]	[3, 9, 10, 27, 38, 43, 82]

5. Explain the algorithm for Sequential Search in detail with example.

Sequential searching, also known as linear search, is a straightforward method of finding a target value within a list. It involves iterating through each element of the list until the desired element is found or the end of the list is reached.

Algorithm:

1. Start from the first element of the list.
2. Compare the target value with the current element.
3. If the current element matches the target value, return its index.
4. If the end of the list is reached without finding the target value, return a "not found" indication.

```
function sequentialSearch(list, target):
    for each element in the list:
        if element equals target:
            return index of element
    return "not found"
```

Example:

Consider the following list: **[14, 7, 25, 3, 9, 21]**.

Let's search for the value **9** using sequential search:

Iteration	Current Element	Index
1	14	0
2	7	1
3	25	2
4	3	3
5	9	4 (Target found)

The target value **9** is found at index **4**.

6. Explain the algorithm for Binary Search in detail with example.

Binary searching is a more efficient search algorithm compared to sequential search, but it requires the list to be sorted beforehand. It follows the "divide and conquer" strategy, repeatedly dividing the search interval in half until the target value is found.

Algorithm:

1. Sort the list if it's not already sorted.
2. Initialize two pointers, **low** and **high**, to the first and last elements of the list, respectively.
3. Compute the middle index as **(low + high) / 2**.
4. Compare the target value with the element at the middle index.
5. If the target matches the middle element, return its index.
6. If the target is less than the middle element, continue the search on the lower half.
7. If the target is greater than the middle element, continue the search on the upper half.
8. Repeat steps 3-7 until the target is found or **low** becomes greater than **high**, indicating the target is not in the list.

```
function binarySearch(sortedList, target):
    low = 0
    high = length of sortedList - 1
    while low <= high:
        middle = (low + high) / 2
        if sortedList[middle] equals target:
            return middle
        else if sortedList[middle] < target:
            low = middle + 1
        else:
            high = middle - 1
    return "not found"
```

Consider the sorted list: **[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]**.
Let's search for the value **13** using binary search:

Iteration	Low	High	Middle	Middle Element	Comparison
1	0	9	4	9	13 > 9
2	5	9	7	15	13 < 15
3	5	6	5	11	13 > 11
4	6	6	6	13	13 = 13 (Target found)

9. As we can see, the target value **13** is found at index **6**.

7. Explain Trie structure.

A Trie, also known as a prefix tree, is a tree-like data structure used for efficiently storing and retrieving a dynamic set of strings. It is particularly useful for tasks like autocomplete, spell checking, and dictionary implementations. The key characteristic of a Trie is that the path from the root to each node represents a string.

Structure of a Trie:

- Each node of the Trie represents a single character of a string.
- The root of the Trie is typically an empty node.
- Each node has links to its child nodes, representing the next character in the string.
- Nodes may also contain additional information, such as a boolean flag indicating whether the string represented by the node is a complete word.

Operations:

1. **Insertion**: Insert a string into the Trie.
2. **Search**: Search for a string in the Trie.
3. **Prefix Search**: Find all strings with a given prefix in the Trie.
4. **Deletion**: Remove a string from the Trie.

8. Explain height-balanced tree in detail.

Height-balanced tree is a tree data structure in which the height of the left and right subtrees of any node differ by at most one. These trees ensure that the depth of the tree is kept as small as possible, which leads to more efficient operations like search, insert, and delete.

Here are the details of a height-balanced tree:

1. **Definition**:

   A height-balanced tree is a binary tree in which the height difference between the left and right subtrees of any node is at most one.

2. **Properties**:

   The height difference between the left and right subtrees of any node (also known as the balance factor) is either L,B and R.
   The height of an empty tree (or a single node) is defined as -1.
   The height of a leaf node is 0.
   The height of the tree is the maximum height among all nodes.

3. **Advantages**:

   Provides efficient operations: Search, Insert, Delete.
   Guarantees logarithmic time complexity for these operations.
   Ensures the tree remains balanced, preventing worst-case scenarios that could lead to performance degradation.

4. **Implementation**:

   Height-balanced trees are often implemented using AVL trees or Red-Black trees.
   AVL trees use balance indicators to ensure balance after each insertion or deletion operation.
   Red-Black trees use color-coding and rotation operations to maintain balance.

   Balance indicators
   L (Left-Heavy): Indicates that the tree is leaning towards the left, meaning that the height of the left sub tree of a node is greater than the height of its right subtree.
   R (Right- Heavy): Indicates that the tree is leaning towards the right, meaning that the height of the right sub tree of a node is greater than the height of its left subtree.
   B (Balanced): Indicates that the tree is balanced, meaning that the heights of the left and right subtrees of a node differ by at most one.

5. **Operations**:

   Insertion: After inserting a node, rebalancing might be necessary to maintain the height balance property. If the balance indicators are not B,L,R then tree is considered as unbalanced tree,
   Deletion: Similar to insertion, deletion might require rebalancing the tree.
   Search: Search operations can be performed efficiently in logarithmic time due to the balanced nature of the tree.
   Traversal: In-order, pre-order, and post-order traversals can still be performed efficiently.

Unbalanced Trees:

An unbalanced tree occurs when the balance indicators of nodes violate the balance property, either by being left-heavy or right-heavy. Unbalanced trees can lead to inefficient operations and degraded performance.

Balancing Strategies:

To address unbalanced trees, various balancing strategies can be employed:

1. **Rotation Operations**: Rotation operations can be used to rebalance the tree by adjusting the structure to maintain balance indicators within acceptable ranges.
2. **Rebalancing Algorithms**: These algorithms identify unbalanced portions of the tree and apply appropriate balancing operations to restore balance.
3. **Proper Insertion and Deletion Procedures**: Ensuring that insertion and deletion procedures include steps for rebalancing the tree can prevent or mitigate imbalance.

9. Explain Weight-balanced tree in detail.

Weight-Balanced Tree:

A weight-balanced tree is a type of binary search tree where the balance of the tree is maintained based on the weights of its nodes rather than the heights. In a weight-balanced tree, the weight of a node is defined as the number of nodes in its subtree, including itself. The aim of a weight-balanced tree is to ensure that for any node, the weight of its left and right subtrees differ by at most one.

Structure of a Weight-Balanced Tree:

1. **Node Structure**: Each node in a weight-balanced tree typically contains the following information:
   - Key: The value stored in the node.
   - Left Child Pointer: Points to the left child node.
   - Right Child Pointer: Points to the right child node.
   - Weight: The number of nodes in the subtree rooted at the current node.
2. **Balance Condition**: The balance condition for a weight-balanced tree is that for any node, the absolute difference in the weights of its left and right subtrees should be at most one.

Operations:

1. **Insertion**: When a node is inserted into a weight-balanced tree, the weights of affected nodes along the insertion path are updated to maintain the balance condition.
2. **Deletion**: Similar to insertion, when a node is deleted from a weight-balanced tree, the weights of affected nodes along the deletion path are adjusted to ensure balance.
3. **Search**: Search operations can be performed similarly to other binary search trees, leveraging the balanced structure of the tree.

Advantages:

1. **Balanced Structure**: Weight-balanced trees maintain balance based on the weights of nodes, ensuring efficient operations.
2. **Optimal Height**: By maintaining balance, weight-balanced trees keep the height of the tree close to optimal, resulting in efficient search, insert, and delete operations.
3. **Adaptability**: Weight-balanced trees can adapt to changes in the data set dynamically, ensuring continued balance and optimal performance.

Implementation:

1. **Maintaining Weights**: During insertion and deletion operations, the weights of nodes are updated recursively as necessary to maintain balance.
2. **Rotation Operations**: Rotation operations may be employed to adjust the structure of the tree and maintain balance if needed.

Example:

Consider the following weight-balanced tree:

```
          10(6)
         /     \
     5(2)       15(3)
    /   \       /   \
 2(1)  8(1)  12(1)  20(1)
```

In this example, each node is annotated with its weight in parentheses. The weights of the left and right sub trees of each node differ by at most one, satisfying the balance condition.

Complexity:

1. **Search Time**: Search operations in a weight-balanced tree have a time complexity of O(log n), where n is the number of nodes in the tree.
2. **Insertion and Deletion Time**: Insertion and deletion operations also have a time complexity of O(log n) on average, with occasional rebalancing steps.

Weight-balanced trees offer a balanced structure and efficient operations, making them suitable for dynamic data sets where the number of elements may vary over time.

A 2-3 tree is a type of balanced search tree where each node can have either two or three children. It's a self-balancing tree data structure that ensures the tree remains balanced after insertions and deletions. 2-3 trees are specifically designed to provide efficient operations, including search, insert, and delete, with a worst-case time complexity of O(log n), where n is the number of elements in the tree.

10. Explain 2-3 tree in detail.

Structure of a 2-3 Tree:

1. **Node Structure**: Each node in a 2-3 tree can have either two or three children. The node can be one of two types:
   - 2-node: Contains one key and two children.
   - 3-node: Contains two keys and three children.
2. **Properties**:
   - All leaf nodes are at the same level, ensuring balanced height.
   - Each node (except the root) has either one or two keys.
   - For a node with k keys, it has k+1 children.
   - Keys within a node are stored in sorted order.

Operations:

1. **Search**: Search operations in a 2-3 tree are performed similarly to other binary search trees, navigating through the tree based on key values to locate the desired element.
2. **Insertion**: Insertion operations in a 2-3 tree involve finding the appropriate location for the new key and inserting it while maintaining the properties of the tree. If a node becomes overfull (having three keys), it is split into two nodes, and the middle key is promoted to the parent.
3. **Deletion**: Deletion operations in a 2-3 tree involve finding the key to be deleted and handling various cases to maintain the properties of the tree. If a node becomes underfull (having only one key), it may borrow a key from a neighboring sibling or merge with a sibling to maintain balance.

Advantages:

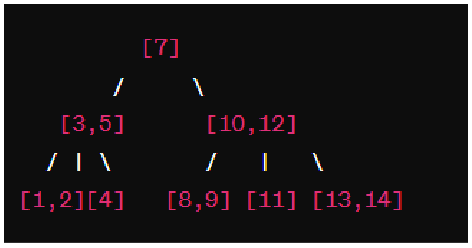
1. **Balanced Height**: 2-3 trees ensure that all leaf nodes are at the same level, maintaining a balanced height and ensuring efficient search operations.
2. **Efficient Operations**: Search, insert, and delete operations in 2-3 trees have a worst-case time complexity of O(log n), making them suitable for dynamic data sets.

Implementation:

1. **Node Splitting**: When a node becomes overfull during insertion, it is split into two nodes, and the middle key is promoted to the parent node.
2. **Node Merging**: When a node becomes underfull during deletion, it may merge with a sibling node to maintain balance.
3. **Recursive Algorithms**: Many operations in 2-3 trees are implemented recursively, traversing down the tree to find the appropriate location for operations.

Example:

Consider the following 2-3 tree:



```
          [7]
         /    \
   [3,5]       [10,12]
  / | \       /   |    \
[1,2][4]   [8,9] [11] [13,14]
```

In this example, each node is represented by square brackets, and keys within each node are separated by commas. This 2-3 tree maintains balance with all leaf nodes at the same level.

Complexity:

- **Search Time**: Search operations in a 2-3 tree have a time complexity of O(log n), where n is the number of elements in the tree.
- **Insertion and Deletion Time**: Insertion and deletion operations also have a time complexity of O(log n) on average, ensuring efficient dynamic operations.

2-3 trees offer a balanced structure and efficient operations, making them suitable for various applications where dynamic data sets are prevalent.

11. Explain hash table, hashing functions and Hashing searching technique in detail.

Hash Table:

A hash table is a data structure that stores key-value pairs where keys are mapped to values through a hashing function. It provides efficient insertion, deletion, and lookup operations by using a technique called hashing.

Components of a Hash Table:

1. **Array**: A fixed-size array is used to store elements of key-value pairs.
2. **Hash Function**: A hash function maps keys to indices in the array, allowing for fast access to values.
3. **Collision Resolution**: Since different keys may map to the same index (collision), collision resolution techniques are employed to handle such scenarios efficiently.

Hashing Functions:

A hashing function is a mathematical function that converts an input (or key) into a fixed-size value (hash code or hash value) typically of a fixed length. The hash code is then used to index a hash table.

Properties of a Good Hashing Function:

1. **Deterministic**: For the same input, the hashing function should always produce the same hash code.
2. **Uniform Distribution**: The hashing function should evenly distribute the hash codes across the hash table to minimize collisions.
3. **Efficient**: The hashing function should be computationally efficient to calculate.

Hashing-based Searching Techniques:

1. **Direct Addressing**: In this technique, each key is directly mapped to an index in the hash table using its hash code. If collisions occur, they are handled using collision resolution techniques like chaining or open addressing.
2. **Chaining**: In chaining, each slot in the hash table contains a linked list of key-value pairs that hash to the same index. When a collision occurs, the new key-value pair is added to the linked list at the corresponding index.
3. **Open Addressing**: In open addressing, collisions are resolved by probing for an empty slot in the hash table. Different probing strategies include linear probing, quadratic probing, and double hashing.

Linear Probing:

- In linear probing, if a collision occurs at index i, the algorithm checks index i+1, i+2, and so on until an empty slot is found.

Quadratic Probing:

- Quadratic probing uses a quadratic function to probe for empty slots after collisions. If a collision occurs at index i, the algorithm checks indices i+1, i+4, i+9, and so on.

Double Hashing:

- Double hashing uses two hashing functions to calculate probe sequences. If a collision occurs at index i, the algorithm calculates the next index using a second hash function.

We'll use a hash table of size 10 and insert the keys {12, 32, 45, 72, 85}.

Hash Table:

- Table Size: 10

1. Linear Probing:

In linear probing, if a collision occurs at index i, the algorithm checks index i+1, i+2, and so on until an empty slot is found.

Key	Hash Code	Index	Collision Resolution (Linear Probing)
12	12 % 10 = 2	2	2 (empty)
32	32 % 10 = 2	2 + 1 = 3	3 (empty)
45	45 % 10 = 5	5	5 (empty)
72	72 % 10 = 2	2 + 2 = 4	4 (empty)
85	85 % 10 = 5	5 + 1 = 6	6 (empty)

Resulting hash table: [__, __, 12, 32, 72, 45, 85, __, __, __]

2. Quadratic Probing:

In quadratic probing, if a collision occurs at index i, the algorithm checks indices i+1^2, i+2^2, i+3^2, and so on until an empty slot is found.

Key	Hash Code	Index	Collision Resolution (Quadratic Probing)
12	12 % 10 = 2	2	2 (empty)
32	32 % 10 = 2	2 + 1^2 = 3	3 (empty)
45	45 % 10 = 5	5	5 (empty)

Key	Hash Code	Index	Collision Resolution (Quadratic Probing)
72	72 % 10 = 2	2 + 2^2 = 6	6 (empty)
85	85 % 10 = 5	5 + 1^2 = 6	7 (empty)

Resulting hash table: [__, __, 12, 32, __, __, 72, 45, 85, __]

3. Double Hashing:

In double hashing, if a collision occurs at index i, the algorithm calculates a second hash using a secondary hash function and probes at index i + hash2(key), where hash2(key) is the result of the secondary hash function.

For simplicity, let's assume the secondary hash function is hash2(key) = 7 - (key % 7).

Key	Hash Code	Index	Secondary Hash	Collision Resolution (Double Hashing)
12	12 % 10 = 2	2	7 - (12 % 7) = 7 - 5 = 2	2 (empty)
32	32 % 10 = 2	2 + (7 - (32 % 7)) = 2 + 4 = 6	2 - (32 % 2) = 2 - 0 = 2	6 (empty)
45	45 % 10 = 5	5	7 - (45 % 7) = 7 - 3 = 4	5 (empty)
72	72 % 10 = 2	2 + (7 - (72 % 7)) = 2 + 6 = 8	2 - (72 % 2) = 2 - 0 = 2	8 (empty)
85	85 % 10 = 5	5 + (7 - (85 % 7)) = 5 + 3 = 8	5 - (85 % 2) = 5 - 1 = 4	8 + 4 = 12 (empty)

Resulting hash table: [__, __, 12, 32, __, __, 72, 45, 85, __]

In each probing technique, we handle collisions by probing for an empty slot in the hash table. Linear probing, quadratic probing, and double hashing employ different strategies to determine the next probing location after a collision, leading to different distributions of keys in the hash table.

Advantages of Hashing:

1. **Fast Lookup**: Hash tables provide constant-time (O(1)) average case lookup complexity.
2. **Efficient Insertion and Deletion**: Insertion and deletion operations are also efficient on average, assuming a good hashing function and collision resolution technique.
3. **Versatility**: Hash tables can be used in various applications, including symbol tables, caches, and databases.

Disadvantages of Hashing:

1. **Collision Handling Overhead**: Dealing with collisions adds overhead, especially in scenarios with high collision rates.
2. **Space Overhead**: Hash tables may require more memory than the actual data stored due to collision resolution and unused slots.
3. **Deterministic Behavior**: Hash tables may not perform well in applications where deterministic ordering is required, as the order of elements is not guaranteed.

Suppose we have a set of keys: {10, 20, 30, 40, 50}. We'll hash these keys into a hash table using three different hashing functions:

1. **Division Method**: The simplest hashing function, which calculates the remainder when dividing the key by the table size.
2. **Multiplication Method**: This method multiplies the key by a constant A, extracts the fractional part, and multiplies by the table size.
3. **Folding Method**: The key is divided into equal-sized parts, which are then summed to produce the hash code.

Let's create a hash table with a size of 10 and hash the keys using these three hashing functions:

1. Division Method:

- Table Size: 10

Key	Hash Code (Division Method)
10	10 % 10 = 0
20	20 % 10 = 0
30	30 % 10 = 0
40	40 % 10 = 0
50	50 % 10 = 0

Since all keys hash to the same index, we encounter clustering issues.

2. Multiplication Method:

- Constant A: 0.618 (a typical choice for A)
- Table Size: 10

Key	Hash Code (Multiplication Method)
10	floor((10 * 0.618) % 10) = 6
20	floor((20 * 0.618) % 10) = 2
30	floor((30 * 0.618) % 10) = 9

Key	Hash Code (Multiplication Method)
40	floor((40 * 0.618) % 10) = 5
50	floor((50 * 0.618) % 10) = 1

3. Folding Method:

- Table Size: 10

Key	Hash Code (Folding Method)
10	1 + 0 = 1
20	2 + 0 = 2
30	3 + 0 = 3
40	4 + 0 = 4
50	5 + 0 = 5

In this example, we've applied three different hashing functions to the same set of keys to generate hash codes. Each hashing function produces different hash codes, which demonstrates how the choice of hashing function can affect the distribution of keys in the hash table and potential clustering issues.

12. Explain heap sort in detail with example.

Heap sort is a comparison-based sorting algorithm that builds a heap from the input array and then repeatedly extracts the maximum (for a max-heap) or minimum (for a min-heap) element from the heap and places it at the end of the sorted array. Heap sort has a time complexity of O(n log n) and is an in-place sorting algorithm, meaning it does not require additional storage proportional to the size of the input.

Steps of Heap Sort:

1. **Build Heap**: Build a heap from the input array. This step is also known as heapify, where we arrange the elements of the array into a heap data structure.
2. **Extract Maximum (or Minimum)**: Repeatedly remove the maximum (or minimum) element from the heap and place it at the end of the sorted array. After each removal, adjust the heap to maintain its heap property.

Example of Heap Sort in Tabular Format:

3. **Step 1: Build Max-Heap**

Step	Max-Heap
1	[5, 3, 8, 4, 2, 9, 1, 6, 7]
2	[5, 3, 9, 4, 2, 8, 1, 6, 7]
3	[5, 3, 9, 4, 2, 8, 1, 6, 7]
4	[9, 4, 8, 5, 2, 3, 1, 6, 7]

4. **Step 2: Extract Maximum (or Minimum)**

Step	Max-Heap	Sorted Array
1	[7, 4, 8, 5, 2, 3, 1, 6]	[9]
2	[8, 4, 7, 5, 2, 3, 1, 6]	[9]
3	[6, 4, 7, 5, 2, 3, 1]	[9, 8]
4	[7, 4, 6, 5, 2, 3, 1]	[9, 8]
5	[1, 4, 6, 5, 2, 3]	[9, 8, 7]
6	[6, 4, 3, 5, 2, 1]	[9, 8, 7]
7	[5, 4, 3, 1, 2]	[9, 8, 7, 6]
8	[4, 2, 3, 1]	[9, 8, 7, 6, 5]
9	[3, 2, 1]	[9, 8, 7, 6, 5, 4]
10	[2, 1]	[9, 8, 7, 6, 5, 4, 3]
11	[1]	[9, 8, 7, 6, 5, 4, 3, 2]
12	[]	[9, 8, 7, 6, 5, 4, 3, 2, 1]

13. Show time complexity analysis for different sorting Algorithms?

Algorithm	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)
Heap Sort	O(n log n)	O(n log n)	O(n log n)

14. Show time the time complexity analysis for Sequential Search and Binary Search?

Algorithm	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity
Sequential Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

For Binary Search, it's important to note that the array must be sorted in order to apply the algorithm, which may add an additional O(n log n) time complexity for sorting the array, making the overall complexity for binary search in the worst case O(n log n).

15. Explain load factor in hashing.

The load factor in hashing refers to the ratio of the number of elements stored in a hash table to the size of the hash table. It's a measure of how full the hash table is.

Mathematically, the load factor α is calculated as:

$\alpha = m/n$

Where:

- n is the number of elements currently stored in the hash table.
- m is the size of the hash table (i.e., the number of buckets or slots).

A higher load factor means the hash table is more densely populated, while a lower load factor means it's less densely populated.

The load factor is an important factor in determining the performance of a hash table. A higher load factor increases the likelihood of collisions, which can degrade the performance of hash table operations such as insertion, deletion, and searching. Typically, a hash table is resized (rehashed) when the load factor exceeds a certain threshold to maintain a balance between efficiency and memory usage.

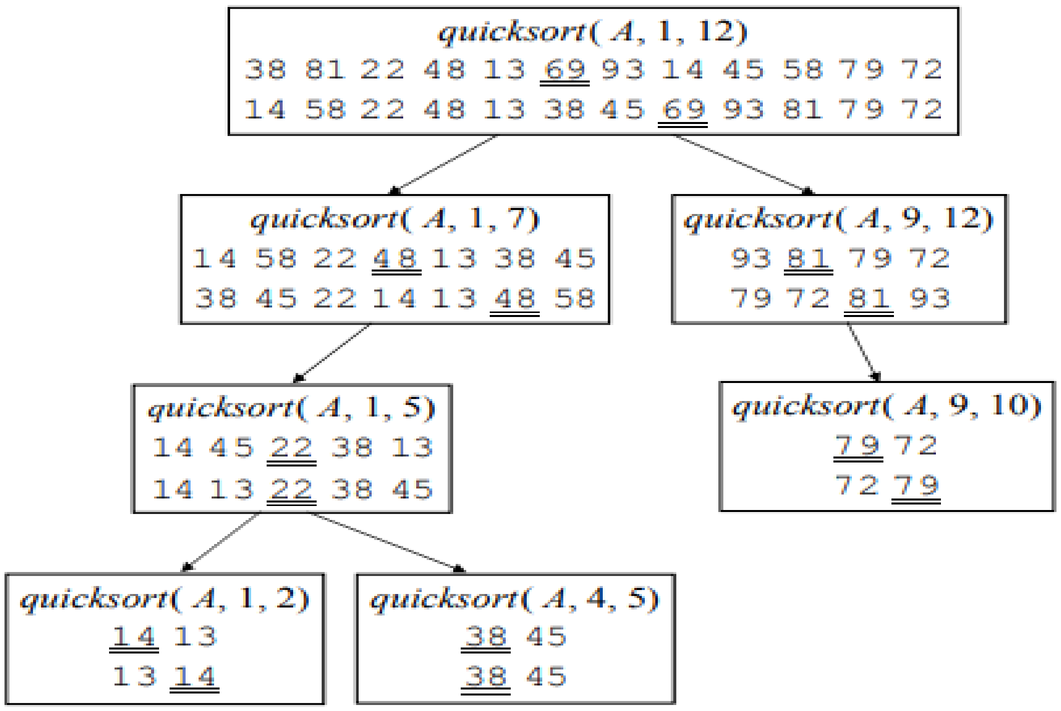
16. Discuss Difference/comparison of Bubble Sort and Insertion Sort.

Aspect	Bubble Sort	Insertion Sort
Basic Principle	Repeatedly compare adjacent elements and swap if necessary	Build the sorted array one element at a time by inserting each element into its correct position
Time Complexity	O(n^2) in average and worst cases	O(n^2) in average and worst cases
Best-case Time Complexity	O(n) when the array is already sorted	O(n) when the array is already sorted
Space Complexity	O(1)	O(1)
Stability	Stable	Stable
Adaptive Nature	Not adaptive	Adaptive
Suitability	Rarely used in practice, mainly for small datasets or educational purposes	Practical for small datasets or nearly sorted arrays

17. Discuss how following numbers are sorted using Quick Sort.
38,81,22,48,13,69,93,14,45,58,79,72 (WINTER-2022)

***the underlined element is a pivot element**

We will partition the array into subarrays using pivot element and the sort as follows.



18. Discuss various rehashing techniques.

Rehashing is the process of creating a new hash table and transferring all elements from the old hash table to the new one when the load factor exceeds a certain threshold. This helps in maintaining a balance between the number of elements and the size of the hash table, ensuring efficient performance. There are several techniques used for rehashing, each with its own advantages and considerations:

1. **Linear Probing**:
- In linear probing, when a collision occurs during insertion, the algorithm searches for the next available slot in the hash table by linearly probing successive slots until an empty slot is found.
- During rehashing, elements from the old hash table are inserted into the new hash table using linear probing. If collisions occur in the new hash table, the probing sequence is recalculated to find the next available slot.

2. **Quadratic Probing**:
- Quadratic probing is similar to linear probing, but instead of probing consecutive slots, it probes slots using a quadratic function.
- During rehashing, elements are inserted into the new hash table using quadratic probing. This helps in reducing primary clustering compared to linear probing.

3. **Double Hashing**:
- Double hashing uses two hash functions: one to compute the initial hash index and another to calculate the interval between successive probes.
- During rehashing, elements are inserted into the new hash table using double hashing, which helps in distributing elements more evenly across the hash table and reducing clustering.

4. **Chaining**:
- Chaining is a technique where each bucket in the hash table is a linked list, and elements with the same hash value are stored in the same bucket.
- During rehashing, a new hash table with a larger size is created, and elements from the old hash table are redistributed among the buckets based on their new hash values.

5. **Cuckoo Hashing**:
- Cuckoo hashing uses multiple hash functions and two hash tables. Each element is stored in one of the two hash tables based on the output of the hash functions.
- During rehashing, elements from the old hash tables are inserted into the new hash tables using cuckoo hashing, which helps in achieving better load balancing and reducing collisions.

6. **Extendible Hashing**:
- Extendible hashing dynamically adjusts the size of the directory and the buckets based on the number of elements in the hash table.
- During rehashing, if the load factor exceeds a certain threshold, the directory is resized, and elements are redistributed among the buckets based on their new hash values.

Each rehashing technique has its own trade-offs in terms of memory usage, insertion and lookup times, and collision resolution. The choice of rehashing technique depends on the specific requirements of the application and the characteristics of the data being stored.

19. Enlist different hashing functions and briefly explain them.

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

1. **Division Method.**
2. **Mid Square Method.**
3. **Folding Method.**
4. **Multiplication Method.**

Let's begin discussing these methods in detail.

1. Division Method:

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:
h(K) = k mod M
Here,
***k** is the key value, and*
***M** is the size of the hash table.*

It is best suited that **M** is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Example:
k = 12345
M = 95
h(12345) = 12345 mod 95
= 90

k = 1276
M = 11
h(1276) = 1276 mod 11
= 0

Pros:
1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

Cons:
1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to choose the value of M.

2. Mid Square Method:

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

1. Square the value of the key k i.e. k^2

2. Extract the middle **r** digits as the hash value.

Formula:
h(K) = h(k x k)
Here,
***k** is the key value.*

The value of **r** can be decided based on the size of the table.

Example:
Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

k = 60
k x k = 60 x 60
= 3600
h(60) = 60

The hash value obtained is 60

Pros:
1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:
1. The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

<u>3. Digit Folding Method:</u>
This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:
k = k1, k2, k3, k4, ….., kn
s = k1+ k2 + k3 + k4 +….+ kn
h(K)= s
Here,
s** is obtained by adding the parts of the key **k

Example:
k = 12345
k1 = 12, k2 = 34, k3 = 5
s = k1 + k2 + k3
= 12 + 34 + 5
= 51
h(K) = 51

Note:
The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

4. Multiplication Method

This method involves the following steps:

1. Choose a constant value A such that 0 < A < 1.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:
h(K) = floor (M (k*A mod 1))
Here,
***M** is the size of the hash table.*
***k** is the key value.*
***A** is a constant value.*

Example:
k = 12345
A = 0.357840
M = 100

*h(12345) = floor[100 (12345*0.357840 mod 1)]*
= floor[100 (4417.5348 mod 1)]
= floor[100 (0.5348)]
= floor[53.48]
= 53

Pros:
The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

Cons:
The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.