DATA STRUCTURE

A data structure is a way of organizing and storing data in a computer so that it can be accessed and manipulated efficiently. It defines the relationship between the data and the operations that can be performed on it. Examples include arrays, linked lists, trees, and hash tables.

Data structures can be broadly classified into two main categories: linear data structures and non-linear data structures.

1. Linear Data Structures:

A. Arrays: A collection of elements stored in contiguous memory locations, accessed using an index.

B. Linked Lists: Elements are stored in nodes, where each node contains a data field and a reference (pointer) to the next node.

C. Stacks: Follows the Last-In-First-Out (LIFO) principle, where elements are added and removed from the same end (top).

D. Queues: Follows the First-In-First-Out (FIFO) principle, where elements are added at one end (rear) and removed from the other end (front).

E. Hash Tables: Stores data in key-value pairs, allowing for efficient retrieval of values based on keys.

2. Non-linear Data Structures:

A. Trees: Hierarchical structures consisting of nodes connected by edges, with a single root node and child nodes branching out from it. Examples include binary trees, binary search trees, AVL trees, etc.

Graphs: Collections of nodes (vertices) and edges connecting these nodes. Graphs can be directed or undirected and may have weighted edges.

C. Heaps: Specialized tree-based structures that satisfy the heap property, allowing for efficient retrieval of the maximum (max heap) or minimum (min heap) element.

D. Tries: Also known as digital trees or prefix trees, they are tree-based data structures used to store a dynamic set where keys are strings.

Q. What is Abstract Data Types?

Abstract data types (ADTs) are high-level models that define a set of operations and their behavior on data without specifying the implementation details. They provide a logical description of data and operations, allowing for modular and reusable code. Examples include stacks, queues, lists, and trees.

Q. Difference between Linear and Non-linear Data Structures.

Differentiation between Linear and Non-linear Data Structures:

Property	Linear Data Structures	Non-linear Data Structures
Organization	Elements arranged sequentially.	Elements organized hierarchically or interconnected.
Access	Accessible sequentially (one after another).	Accessible non-sequentially.
Traversal	Traversed linearly.	Traversal may involve recursive or iterative methods.
Relationship	Elements are related linearly (preceding and succeeding).	Elements may have multiple connections.
Examples	Arrays, Linked Lists, Stacks, Queues.	Trees, Graphs, Heaps, Tries.

ALGORITHM

Algorithms are step-by-step procedures or sets of instructions designed to solve specific problems or perform specific tasks. They define a sequence of operations to be performed to achieve a desired outcome, typically optimized for efficiency and correctness. Algorithms are fundamental to computer science and are used in various fields such as mathematics, data processing, artificial intelligence, and more.

Time complexity refers to the measure of the amount of time an algorithm takes to complete as a function of the size of the input data. It describes how the runtime of an algorithm increases with the size of the input.

Space complexity, on the other hand, refers to the measure of the amount of memory space an algorithm requires to execute as a function of the size of the input data. It describes how much memory an algorithm needs to run to completion.

Big O notation is a mathematical notation used to describe the upper bound of the time or space complexity of an algorithm in terms of the input size. It represents the worst-case scenario for the growth rate of an algorithm's performance as the input size approaches infinity.

Q. What are the five essential properties of Algorithm?

1. Input: An algorithm must have zero or more inputs, which are the data provided to it for processing.

2. Output: It must produce at least one output, which is the result or solution obtained after processing the input data.

3. Definiteness: Each step of the algorithm must be precisely defined and unambiguous, leaving no room for interpretation.

4. Finiteness: The algorithm must terminate after a finite number of steps. It should not run indefinitely.

5. Effectiveness: The steps of the algorithm must be executable using a finite amount of resources (time and space). It should be practical and feasible to implement in real-world scenarios.

ARRAY

Q. Define the term array. How 2D arrays are stored in memory?

An array is a data structure that stores a collection of elements of the same type in a contiguous block of memory. Each element in the array is accessed by its index.

In 2D arrays, elements are arranged in rows and columns. The way 2D arrays are stored in memory can follow two common conventions:

1. Row Major Order (Row-wise): In row major order, elements of each row are stored sequentially in memory. This means that elements of the first row are stored first, followed by the elements of the second row, and so on. Within each row, elements are stored adjacent to each other.

For example, consider a 2D array A with dimensions m x n. If A[i][j] represents the element at the i-th row and j-th column:

A[0][0], A[0][1], A[0][2], ..., A[0][n-1] are stored consecutively.

A[1][0], A[1][1], A[1][2], ..., A[1][n-1] are stored consecutively after the first row, and so on.

2. Column Major Order (Column-wise): In column major order, elements of each column are stored sequentially in memory. This means that elements of the first column are stored first, followed by the elements of the second column, and so on. Within each column, elements are stored adjacent to each other.

For example, considering the same 2D array A with dimensions m x n:

A[0][0], A[1][0], A[2][0], ..., A[m-1][0] are stored consecutively.

A[0][1], A[1][1], A[2][1], ..., A[m-1][1] are stored consecutively after the first column, and so on.

Q. Let the size of the elements stored in a matrix A[8][3] is 4 bytes, if the base address is 3500 then address of A[5][2] in both row and column major order.

Given: Size of each element = 4 bytes

Base address of matrix A = 3500

1. Row Major Order:

Address of A[i][j] = Base Address + (i * Number of columns + j) * Size of each element

Address of A[5][2] = Base address + (5 * 3 + 2) * Size of each element

= 3500 + (17 * 4)

= 3500 + 68

= 3568

So, the address of A[5][2] in row-major order is 3568.

2. Column Major Order:

Address of A[i][j] = Base Address + (j * Number of rows + i) * Size of each element

Address of A[5][2] = Base address + (2 * 8 + 5) * Size of each element

= 3500 + (21 * 4)

= 3500 + 84

= 3584

So, the address of A[5][2] in column-major order is also 3584.

Q. What is Sparse Matrix? Represent 3 Tuple form.

A sparse matrix is a matrix that contains mostly zero elements. In a sparse matrix, the majority of elements have the value of zero, making it inefficient to store them all. Instead, sparse matrices typically store only the non-zero elements along with their row and column indices, allowing for more efficient memory usage and computational operations on matrices with large dimensions.

In the context of representing a sparse matrix, a 3-tuple form (also known as triplet form or coordinate list) typically includes the following information for each non-zero element:

1. Row index
2. Column index
3. Value of the element

For example, if we have a sparse matrix with non-zero elements at positions (1, 2), (2, 3), and (3, 1), and their respective values are 5, 7, and 9, then the 3-tuple form representation would be:

(1, 2, 5)

(2, 3, 7)

(3, 1, 9)

Each tuple represents a non-zero element's row index, column index, and value, respectively.

Q. Write a c program to add two polynomial.

```
#include <stdio.h>
#define MAX_TERMS 100

typedef struct {
   int coef;
   int exp;
} Term;

void addPolynomials(Term poly1[], int size1, Term poly2[], int size2, Term result[]) {
   int index1 = 0, index2 = 0, indexResult = 0;

   while (index1 < size1 && index2 < size2) {
      if (poly1[index1].exp == poly2[index2].exp) {
         result[indexResult].coef = poly1[index1].coef + poly2[index2].coef;
         result[indexResult].exp = poly1[index1].exp;
         index1++;
         index2++;
      } else if (poly1[index1].exp > poly2[index2].exp) {
         result[indexResult] = poly1[index1];
         index1++;
      } else {
         result[indexResult] = poly2[index2];
         index2++;
      }
      indexResult++;
   }

   while (index1 < size1) {
      result[indexResult] = poly1[index1];
```

```
        index1++;
        indexResult++;
    }

    while (index2 < size2) {
        result[indexResult] = poly2[index2];
        index2++;
        indexResult++;
    }
}
```

STACK

Q. Describe the algorithm for postfix evaluation.

The algorithm for evaluating postfix expressions, also known as reverse Polish notation (RPN), involves scanning the expression from left to right and performing operations based on encountered operands and operators. Here's a brief description of the algorithm:

1. Initialize an empty stack to store operands.
2. Iterate through each token (operand or operator) in the postfix expression.
3. If the token is an operand, push it onto the stack.
4. If the token is an operator:
   - Pop the required number of operands from the stack (typically 2 for binary operators).
   - Perform the operation on the popped operands.
   - Push the result back onto the stack.
5. Once all tokens have been processed, the final result will be the only element left in the stack.

Q. Describe the algorithm to convert infix to postfix.

The algorithm to convert an infix expression to a postfix expression involves using a stack to manage operators and operands. Here's a brief description of the algorithm:

1. Create an empty stack to hold operators.
2. Iterate through each symbol (operand or operator) in the infix expression from left to right.
3. If the symbol is an operand, output it directly to the postfix expression.
4. If the symbol is an operator:
   a. Pop operators from the stack and output them to the postfix expression until either:
      - The stack is empty.
      - An operator with lower precedence than the current operator is encountered.
   b. Push the current operator onto the stack.
5. If the symbol is an opening parenthesis '(', push it onto the stack.
6. If the symbol is a closing parenthesis ')', pop operators from the stack and output them to the postfix expression until an opening parenthesis is encountered. Discard the opening parenthesis.
7. Once all symbols have been processed, pop any remaining operators from the stack and output them to the postfix expression.
8. The resulting postfix expression is the desired output.

Q. Evaluate the postfix expression: 3 1 + 2 ^ 7 4 - 2 * + 5 –

3 1 + 2 ^ 7 4 – 2 * + 5 –

= 4 2 ^ 7 4 – 2 * + 5 –

= 16 7 4 – 2 * + 5 –

= 16 3 2 * + 5 –

= 16 6 + 5 –

= 22 5 –

= 17

Q. Convert the infix expression into postfix expression: 9 + 5 * 7 – 6 ^ 2 + 15 / 3

9 + 5 * 7 – 6 ^ 2 + 15 / 3

= 9 + 5 * 7 – [6 2 ^] + 15 / 3

= 9 + [5 7 *] – [6 2 ^] + 15 / 3

= 9 + [5 7 *] – [6 2 ^] + [15 3 /]

= [9 5 7 * -] – [6 2 ^] + [15 3 /]

= [9 5 7 * - 6 2 ^ -] + [15 3 /]

= [9 5 7 * - 6 2 ^ - 15 3 / +]

Q.Define stack. Write push() and pop function.

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, meaning that the last element added to the stack will be the first one to be removed. It supports two main operations: push, which adds an element to the top of the stack, and pop, which removes the top element from the stack.

Here's how you can define a stack and implement push and pop functions:

```
#include <stdio.h>

#include <stdlib.h>

#define SIZE 4


int top = -1, inp_array[SIZE];


void push() {

   int x;

   if (top == SIZE - 1) {

      printf("\nOverflow!!");

   } else {

      printf("\nEnter the element to be added onto the stack: ");
```

```
        scanf("%d", &x);
        top = top + 1;
        inp_array[top] = x;
    }
}

void pop() {
    if (top == -1) {
        printf("\nUnderflow!!");
    } else {
        printf("\nPopped element: %d", inp_array[top]);
        top = top - 1;
    }
}
```

QUEUE

Q. Write algorithm to insert an element in linear queue.

Here's an algorithm to insert an element into a linear queue:

```
Insert(Queue, element):
    1. if Queue is full
          return "Queue Overflow"
    2. else
          increment rear pointer
          Queue[rear] = element
```

Q. Write algorithm to delete an element in linear queue.

Here's the algorithm to delete an element from a linear queue:

```
delete_element():
    if front == -1 and rear == -1:
        display "Queue is empty (Underflow)"
    else:
        removed_element = queue[front]
        if front == rear:
            front = rear = -1
        else:
            front = (front + 1) % MAX_SIZE
        display "Element removed from queue:", removed_element
```

In this algorithm:

- `front` and `rear` are pointers to the front and rear of the queue, respectively.
- `queue` is the array used to implement the queue.
- `MAX_SIZE` is the maximum size of the queue.

Q. Write algorithm to insert an element in a circular queue.

Here's the algorithm to insert an element into a circular queue:

```
insert_element(value):
    if ((rear + 1) % MAX_SIZE) == front:
        display "Queue is full (Overflow)"
    else:
        if front == -1:
            front = rear = 0
        else:
            rear = (rear + 1) % MAX_SIZE
        queue[rear] = value
        display "Element inserted into queue:", value
```

In this algorithm:

- `front` and `rear` are pointers to the front and rear of the circular queue, respectively.
- `queue` is the array used to implement the circular queue.
- `MAX_SIZE` is the maximum size of the circular queue.

Q. Write algorithm to display an element in a circular queue.

Here's the algorithm to display an element in a circular queue:

```
display_element():
    if front == -1 and rear == -1:
        display "Queue is empty (Underflow)"
    else:
        current_index = front
        display "Elements in the circular queue:"
        repeat until current_index reaches rear:
            display queue[current_index]
            current_index = (current_index + 1) % MAX_SIZE
        display queue[rear]
```

In this algorithm:

- `front` and `rear` are pointers to the front and rear of the queue, respectively.
- `queue` is the array used to implement the queue.

- `MAX_SIZE` is the maximum size of the queue.

Q. What is priority queue?

A priority queue is an abstract data type similar to a regular queue or stack but with an additional feature: each element in the priority queue has an associated priority. Elements with higher priorities are dequeued before elements with lower priorities.

LINKED LIST

Q. Write algorithm to do the following operations on a singly linked list:

1. Insert a node at beginning

2. Insert a node at end

3. Delete a node from beginning

4. Delete a node from end

5. Search a node

6. Count no. of nodes.

Here are algorithms to perform the specified operations on a singly linked list:

1. Insertion at the beginning:

InsertAtBeginning(List, data):

1. Create a new node with the given data

2. Set the next pointer of the new node to the current head of the list

3. Update the head of the list to point to the new node

2. Insertion at the end:

InsertAtEnd(List, data):

1. Create a new node with the given data

```
   2. If the list is empty, set the head of the list to point to the new node
   3. Otherwise, traverse the list until the last node
   4. Set the next pointer of the last node to point to the new node
```

3. Deletion from the beginning:

```
DeleteFromBeginning(List):
   1. If the list is empty, return "List is empty"
   2. Otherwise, set a temporary pointer to the current head of the list
   3. Update the head of the list to point to the next node
   4. Free the memory allocated for the old head node
```

4. Deletion from the end:

```
DeleteFromEnd(List):
   1. If the list is empty, return "List is empty"
   2. If the list has only one node, set the head of the list to NULL and free the memory allocated
for the node
   3. Otherwise, traverse the list until the second-to-last node
   4. Set the next pointer of the second-to-last node to NULL
   5. Free the memory allocated for the last node
```

5. Search a node:

```
Search(List, key):
   1. Set a pointer to the head of the list
   2. While the pointer is not NULL and the data of the current node is not equal to the key:
         Move the pointer to the next node
   3. If the pointer is NULL, return "Node not found"
   4. Otherwise, return "Node found"
```

6. Count number of nodes:

```
CountNodes(List):
  1. Set a counter variable to 0
  2. Set a pointer to the head of the list
  3. While the pointer is not NULL:
       Increment the counter variable
       Move the pointer to the next node
  4. Return the counter variable
```

Q. Write algorithm to merge 2 sorted linked list.

Here's an algorithm to merge two sorted linked lists into a single sorted linked list:

```
MergeSortedLists(list1, list2):
  1. Create a new empty linked list to store the merged result, let's call it mergedList
  2. Set two pointers, one for each input list: pointer1 to the head of list1 and pointer2 to the head of list2
  3. While both pointer1 and pointer2 are not NULL:
       If the value of the node pointed to by pointer1 is less than or equal to the value of the node pointed to by pointer2:
         Append the value of the node pointed to by pointer1 to mergedList
         Move pointer1 to the next node in list1
       Else:
         Append the value of the node pointed to by pointer2 to mergedList
         Move pointer2 to the next node in list2
  4. If there are any remaining nodes in list1, append them to mergedList
  5. If there are any remaining nodes in list2, append them to mergedList
  6. Return mergedList
```

TREE

Q. What is Binary Search Tree?

A Binary Search Tree (BST) is a binary tree data structure where each node has at most two children (left and right), and the key (value) of each node follows a specific order:

1. The key of the left child node is less than the key of its parent node.
2. The key of the right child node is greater than the key of its parent node.

Q. What is AVL Tree?

An AVL tree is a self-balancing binary search tree named after its inventors Adelson-Velsky and Landis. It is a binary search tree with the additional property that for each node in the tree, the heights of the two child subtrees differ by at most one. This balancing property ensures that the tree remains balanced and maintains its efficiency for operations such as insertion, deletion, and searching, which all have average time complexity of O(log n), where n is the number of nodes in the tree.

Q. Define Balance Factor.

The balance factor of a node in an AVL tree is a numerical value that represents the difference in height between the left and right subtrees of that node. It is calculated as follows:

Balance Factor (BF) = Height of the left subtree - Height of the right subtree

A balance factor of 0 indicates that the left and right subtrees of the node have equal height, meaning the tree is balanced at that node. A positive balance factor indicates that the left subtree is taller than the right subtree, while a negative balance factor indicates the opposite.

Q. What is the condition of Balance Factor to make an AVL Tree? Explain the four types of rotation in AVL Tree.

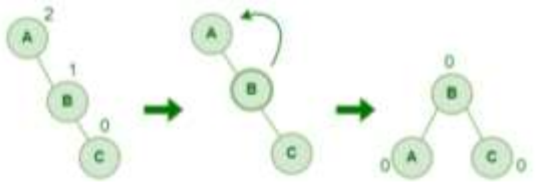
In an AVL tree, the balance factor of each node must satisfy the following condition to maintain balance:

1. **The balance factor of every node in the tree must be in the range [-1, 0, 1].**

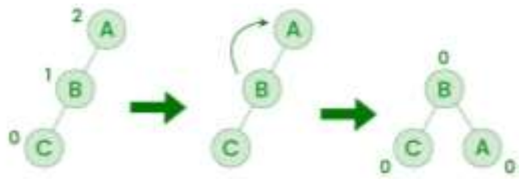
If the balance factor of any node violates this condition, the tree is considered unbalanced at that node, and rotations are performed to restore balance.

There are four types of rotations used in AVL trees to restore balance:

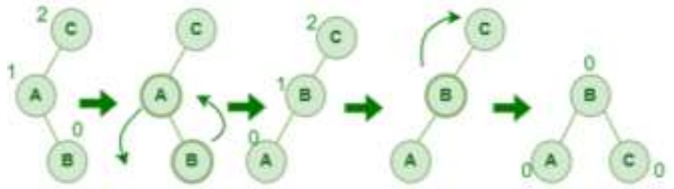
1. Left Rotation (LL Rotation): This rotation is performed when a node has a balance factor of +2 and its right child has a balance factor of +1 or 0. It involves rotating the unbalanced node to the left to rebalance the tree.



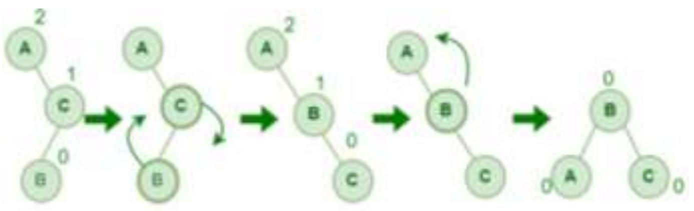
2. Right Rotation (RR Rotation): This rotation is performed when a node has a balance factor of -2 and its left child has a balance factor of -1 or 0. It involves rotating the unbalanced node to the right to rebalance the tree.



3. Left-Right Rotation (LR Rotation): This rotation is performed when a node has a balance factor of +2 and its right child has a balance factor of -1. It involves first performing a right rotation on the right child and then a left rotation on the unbalanced node to rebalance the tree.



4. Right-Left Rotation (RL Rotation): This rotation is performed when a node has a balance factor of -2 and its left child has a balance factor of +1. It involves first performing a left rotation on the left child



and then a right rotation on the unbalanced node to rebalance the tree.

GRAPH

Q. Define Graph. Why Graph is called non-linear Data Structure? Explain Adjacent Matrix & Adjacent List with example.

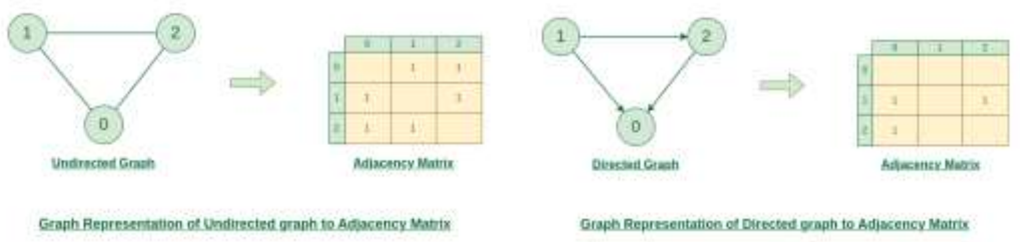
A graph is a data structure that consists of a set of vertices (also called nodes) and a set of edges (also called links) that connect pairs of vertices. Graphs are used to represent relationships between objects, where vertices represent the objects and edges represent the connections between them. Graphs can model a wide range of real-world phenomena, such as social networks, transportation networks, and computer networks.

A graph is called a non-linear data structure because its elements (vertices and edges) are not arranged in a linear sequence like arrays or linked lists. Instead, vertices can be connected in any arbitrary manner, leading to a non-linear arrangement of data.

Now, let's explain two common ways to represent a graph:

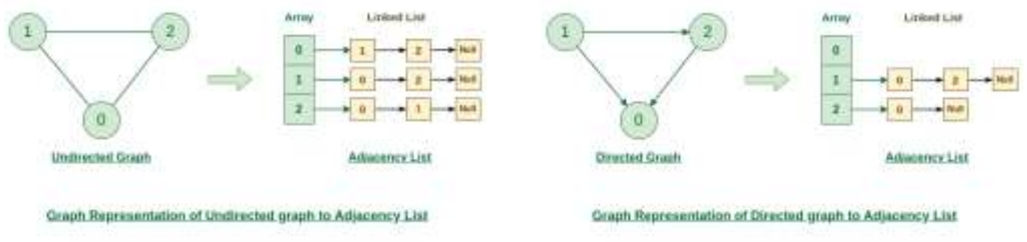
1. Adjacency Matrix: An adjacency matrix is a 2D array (matrix) where the rows and columns represent the vertices of the graph, and the presence or absence of an edge between two vertices is indicated by a 1 or 0, respectively. For an undirected graph, the matrix is symmetric across the diagonal.

Example:


Undirected Graph
Adjacency Matrix
Directed Graph
Adjacency Matrix
Graph Representation of Undirected graph to Adjacency Matrix
Graph Representation of Directed graph to Adjacency Matrix

2. Adjacency List: An adjacency list is a collection of lists (or arrays) where each list corresponds to a vertex in the graph. Each list contains the vertices adjacent to the corresponding vertex. In the case of directed graphs, the list may also contain information about the direction of edges.

Example:


Undirected Graph
Array
Linked List
Adjacency List
Directed Graph
Array
Linked List
Adjacency List
Graph Representation of Undirected graph to Adjacency List
Graph Representation of Directed graph to Adjacency List

Q. What is Pendent Vertex?

A pendent vertex (also known as a pendant vertex) in a graph is a vertex that is connected to exactly one other vertex by an edge. In other words, a pendent vertex has a degree of 1, meaning it has only one adjacent vertex.

Q. Explain DFS & BFS.

Depth First Search (DFS) and Breadth First Search (BFS) are two fundamental graph traversal algorithms used to explore or search through a graph data structure. Both algorithms visit every vertex in the graph, but they do so in different orders.

1. Depth First Search (DFS):

DFS explores as far as possible along each branch before backtracking.

It starts at a chosen vertex and explores as far as possible along each branch before backtracking.

DFS uses a stack (either explicitly or implicitly through recursion) to keep track of vertices to visit next.

It's often implemented recursively but can also be implemented iteratively using a stack data structure.

DFS is typically used for topological sorting, cycle detection, and solving problems like maze traversal.

It may not necessarily find the shortest path between two vertices.

Example of DFS traversal:

```
A -- B -- C
|    |
D    E
```

DFS traversal order: A -> B -> E -> C -> D

2. Breadth First Search (BFS):

BFS explores all the vertices at the current depth before moving to the vertices at the next depth level.

It starts at a chosen vertex and explores all the neighboring vertices at the current depth before moving to the vertices at the next depth level.

BFS uses a queue data structure to keep track of vertices to visit next.

It guarantees the shortest path between the starting vertex and any other reachable vertex in an unweighted graph.

BFS is often used in algorithms for finding shortest paths, network flow, and spanning trees.

Example of BFS traversal:

```
  A -- B -- C
|    |
D    E
```

BFS traversal order: A -> B -> D -> E -> C

Q. Differentiate Prims' and Krushkal's Algorithm. Write down complexities of those.

Prim's algorithm and Kruskal's algorithm are both used to find the minimum spanning tree (MST) of a connected, weighted graph. However, they differ in their approach to constructing the MST and their complexities.

1. Prim's Algorithm:

Prim's algorithm starts from an arbitrary vertex and grows the MST by adding the shortest edge that connects a vertex in the MST to a vertex outside the MST.

It continuously selects the minimum weight edge that connects a vertex in the MST to a vertex outside the MST and adds it to the MST.

Prim's algorithm can be implemented using a priority queue or a min-heap data structure to efficiently select the minimum weight edge.

Complexity:

A. *Time Complexity:* O(V^2) with adjacency matrix representation, O(E log V) with adjacency list representation using binary heap or Fibonacci heap.
B. *Space Complexity:* O(V) for storing the MST and O(V) for the priority queue.

2. Kruskal's Algorithm:

Kruskal's algorithm initially treats each vertex as a separate component and repeatedly adds the shortest edge that connects two different components until all vertices are connected.

It sorts all the edges in non-decreasing order of weight and iteratively selects the minimum weight edge that does not create a cycle in the current forest of trees.

Kruskal's algorithm uses disjoint-set data structure (such as Union-Find) to efficiently detect cycles and merge components.

Complexity:

A. *Time Complexity*: O(E log E) for sorting the edges, O(E α(V)) for detecting cycles and merging components using disjoint-set data structure, where α(V) is the inverse Ackermann function and practically a constant.

B. *Space Complexity:* O(V) for storing the MST and O(E) for storing the edges.

SORTING

Q. Take a set of integers 19, 16, 15, 11, 31, 81, 12, 20. Show the steps to sort using 1. Merge Sort, 2. Quick Sort, 3. Insertion Sort, 4. Selection Sort & 5. Bubble Sort.

Let's go through the steps of sorting the set of integers {19, 16, 15, 11, 31, 81, 12, 20} using each of the specified sorting algorithms.

1. Merge Sort:

Step 1: Split the array into smaller subarrays recursively until each subarray contains only one element.

- Split {19, 16, 15, 11, 31, 81, 12, 20} into {19, 16, 15, 11} and {31, 81, 12, 20}

- Split {19, 16, 15, 11} into {19, 16} and {15, 11}

- Split {31, 81, 12, 20} into {31, 81} and {12, 20}

Step 2: Merge the sorted subarrays back together.

- Merge {19, 16} and {15, 11} to get {11, 15, 16, 19}

- Merge {31, 81} and {12, 20} to get {12, 20, 31, 81}

- Merge {11, 15, 16, 19} and {12, 20, 31, 81} to get the sorted array {11, 12, 15, 16, 19, 20, 31, 81}

2. Quick Sort:

Step 1: Choose a pivot (e.g., the last element, 20) and partition the array around the pivot such that elements smaller than the pivot are on the left, and elements greater than the pivot are on the right.

- Partition {19, 16, 15, 11, 31, 81, 12, 20} around pivot 20 to get {19, 16, 15, 11, 12, 20, 81, 31}

Step 2: Recursively apply Quick Sort to the left and right partitions.

- Apply Quick Sort to {19, 16, 15, 11, 12} and {81, 31}

- Partition {19, 16, 15, 11, 12} around pivot 12 to get {11, 12, 15, 16, 19}

- Partition {81, 31} around pivot 31 to get {31, 81}

- Merge the sorted partitions to get the sorted array {11, 12, 15, 16, 19, 20, 31, 81}

3. Insertion Sort:

- Start from the second element (16) and insert it into its correct position among the already sorted elements to its left.

- Insert 16 into {19} to get {16, 19}

- Repeat this process for each subsequent element in the array.

- Insert 15 into {16, 19} to get {15, 16, 19}

- Insert 11 into {15, 16, 19} to get {11, 15, 16, 19}

- Insert 31 into {11, 15, 16, 19} to get {11, 15, 16, 19, 31}

- Insert 81 into {11, 15, 16, 19, 31} to get {11, 15, 16, 19, 31, 81}

- Insert 12 into {11, 15, 16, 19, 31, 81} to get {11, 12, 15, 16, 19, 31, 81}

- Insert 20 into {11, 12, 15, 16, 19, 31, 81} to get the sorted array {11, 12, 15, 16, 19, 20, 31, 81}

4. Selection Sort:

- Find the smallest element in the unsorted part of the array and swap it with the first element.
- Swap 11 with 19 to get {11, 16, 15, 19, 31, 81, 12, 20}
- Repeat this process for the remaining unsorted part of the array.
- Swap 12 with 16 to get {11, 12, 15, 19, 31, 81, 16, 20}
- Swap 15 with 19 to get {11, 12, 15, 16, 31, 81, 19, 20}
- Swap 16 with 31 to get {11, 12, 15, 16, 19, 81, 31, 20}
- Swap 19 with 81 to get {11, 12, 15, 16, 19, 81, 31, 20}
- Swap 20 with 81 to get the sorted array {11, 12, 15, 16, 19, 20, 31, 81}

Q. Case study of best, worst & average time complexities of all sort.

Algorithm	**Best**	**Worst**	**Average**
Merge Sort	O (nlogn)	O (nlogn)	O (nlogn)
Quick Sort	O (nlogn)	O (nlogn)	O (n^2)
Insertion Sort	O (n)	O (n^2)	O (n^2)
Selection Sort	O (n^2)	O (n^2)	O (n^2)
Bubble Sort	O (n)	O (n^2)	O (n^2)
Heap Sort	O (nlogn)	O (nlogn)	O (nlogn)

Q. Write algorithm of 1. Insertion Sort, 2. Selection Sort & 3. Bubble Sort.

Here are the algorithms for Insertion Sort, Selection Sort, and Bubble Sort:

1. Insertion Sort:

```
InsertionSort(arr)
    n = length of arr
    for i from 1 to n-1:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j = j - 1
        arr[j+1] = key
```

2. Selection Sort:

```
SelectionSort(arr)
    n = length of arr
    for i from 0 to n-1:
        min_index = i
        for j from i+1 to n-1:
            if arr[j] < arr[min_index]:
                min_index = j
        swap arr[min_index] with arr[i]
```

3. Bubble Sort:

```
BubbleSort(arr)
    n = length of arr
    for i from 0 to n-1:
        swapped = False
        for j from 0 to n-1-i:
            if arr[j] > arr[j+1]:
                swap arr[j] with arr[j+1]
                swapped = True
        if swapped is False:
```

break

<u>HASHING</u>

Q. Define Hashing. What you mean by Hash Table and Hash Function? Explain 3 Hash Function used with examples.

Hashing is a technique used in computer science to efficiently store and retrieve data in a data structure called a hash table. It involves converting a data item (such as a key) into a fixed-size value (hash code) using a hash function. This hash code is then used as an index to access the corresponding data item in the hash table.

A hash table is a data structure that stores key-value pairs where each key is mapped to a unique index (hash code) in the table. It typically consists of an array of fixed size, where each element in the array is called a bucket. Each bucket can store multiple key-value pairs, and collisions may occur when multiple keys are mapped to the same index. To handle collisions, different strategies such as chaining or open addressing are employed.

A hash function is a mathematical function that takes an input (or key) and produces a fixed-size value (hash code). The hash function should be deterministic, meaning it always produces the same hash code for the same input. It should also distribute hash codes evenly across the range of possible indices to minimize collisions.

Here are three commonly used hash functions:

1. Division Method:

- This hash function computes the hash code by taking the remainder of dividing the key by a prime number (usually the size of the hash table).

- Example: **hash(key) = key % table_size**

- For example, if the key is 27 and the table size is 10, the hash code would be 27 % 10 = 7.

2. Multiplication Method:

- This hash function multiplies the key by a constant value (usually a fraction) and takes the fractional part of the result.

- Example: **hash(key) = floor(table_size * (key * A % 1))** ; where A is a constant between 0 and 1.

- For example, if the key is 27, the constant A is 0.618033 (the golden ratio), and the table size is 10, the hash code would be floor(10 * (27 * 0.618033 % 1)) = floor(10 * 0.468933) = 4.

3. Polynomial Rolling Hash:

- This hash function treats the key as a polynomial and evaluates it at a certain point using Horner's rule.

- Example: **hash(key) = (key[0] * p^(n-1) + key[1] * p^(n-2) + ... + key[n-1]) % table_size**; where key[i] represents the ASCII value of the ith character in the key, p is a prime number (base), and n is the length of the key.

- For example, if the key is "hello" (ASCII values: 104, 101, 108, 108, 111), the base is 31, and the table size is 10, the hash code would be (104*31^4 + 101*31^3 + 108*31^2 + 108*31^1 + 111*31^0) % 10.

Q. What is Collision? What are the resolution in both open and close addressing.

A collision in hashing occurs when two different keys generate the same hash code, leading to a situation where multiple keys are mapped to the same index in the hash table. Collisions are inevitable in hash tables, especially when the number of possible keys exceeds the number of buckets in the hash table.

There are two main strategies for resolving collisions in hash tables:

1. Open Addressing (Closed Hashing):

In open addressing, when a collision occurs, the algorithm searches for an alternative location within the hash table to place the collided key-value pair. This is done through a sequence of probes until an empty slot (or a slot with a matching key) is found.

Common methods of open addressing include:

1. Linear Probing: The algorithm searches linearly through the hash table, checking consecutive slots until an empty slot is found.

2. Quadratic Probing: The algorithm searches through the hash table using a quadratic sequence, which reduces clustering compared to linear probing.

3. Double Hashing: The algorithm uses a secondary hash function to calculate the step size for probing, providing more flexibility in finding alternative locations.

2. Closed Addressing (Chaining):

In closed addressing, each bucket in the hash table stores a linked list (or another data structure) of key-value pairs that hash to the same index. When a collision occurs, the collided key-value pair is added to the linked list corresponding to its hash code.

Common methods of closed addressing include:

Separate Chaining: Each bucket contains a linked list of collided key-value pairs.

Q. The Keys 12, 18, 13, 2,3, 23, 5, 15 are inserted into an initially empty hash table of length to using open addressing with hash function h(K) = k mod 10 and linear probing. What is the resultant hash table?

To insert the given keys {12, 18, 13, 2, 3, 23, 5, 15} into an initially empty hash table of length 10 using open addressing with the hash function h(K) = K mod 10 and linear probing, we'll perform the following steps:

1. Compute the hash value for each key using the given hash function h(K) = K mod 10.

2. If the computed index is already occupied, perform linear probing until an empty slot is found.

Index	Values
0	
1	
2	***12***
3	***13***
4	***2***
5	***3***
6	***23***
7	***5***
8	***18***
9	***15***

MISCELLANEOUS

Q. Write a c program to print 100th Fibonacci Number.

```
#include <stdio.h>

void fibonacci(int n, long long fib[]) {
   fib[0] = 0; // First Fibonacci number
   fib[1] = 1; // Second Fibonacci number

   for (int i = 2; i <= n; i++) {
      fib[i] = fib[i - 1] + fib[i - 2];
   }
}

int main() {
   int n = 100;
   long long fib[n + 1];

   fibonacci(n, fib);

   printf("The 100th Fibonacci number is: %lld\n", fib[n]);

   return 0;
}
```