Algorithm

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in certain order to get the desired output. In term of data structures, following are the categories of algorithms.

Here are five essential properties of algorithms:

Finiteness: An algorithm must terminate after a finite number of steps.

Definiteness: Each step of the algorithm must be precisely defined and unambiguous.

Input: An algorithm should have zero or more inputs.

Output: An algorithm should produce one or more outputs that represent the solution to the problem or task it is designed to solve.

Effectiveness (or Feasibility): An algorithm should be effective in solving the problem for which it is designed.

Big Oh Notation, O

The O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or longest amount of time an algorithm can possibly take to complete.

Omega Notation, Ω

The Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best-case time complexity or best amount of time an algorithm can possibly take to complete.

Theta Notation, θ

The θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following.

Relationship Between Time and Space Complexities

Trade-offs: Often, there is a trade-off between time and space complexities. An algorithm that uses more memory (higher space complexity) might be able to reduce the time it takes to run (lower time complexity). Conversely, an algorithm that uses less memory might take more time to execute.

Optimization: The relationship between time and space complexities is a key consideration in algorithm optimization. Developers often aim to minimize both time and space complexities, but in some cases, it might be necessary to prioritize one over the other based on the specific requirements of the application.

Cache Efficiency: In some cases, the relationship between time and space complexities can be influenced by cache efficiency. Algorithms that use more memory might be more cache-friendly, leading to faster execution times on modern hardware with large caches.

Algorithm Design: The choice of algorithm can significantly impact both its time and space complexities. For example, sorting algorithms vary widely in their time and space complexities, with some being more efficient in terms of time at the expense of space, and vice versa.

Hardware Limitations: The relationship between time and space complexities can also be influenced by hardware limitations. For example, algorithms that are optimized for parallel processing might have different time and space complexities compared to those designed for sequential processing.

Search Techniques

Linear Search

The Linear Search algorithm searches through an array and returns the index of the value it searches for.

Let's try to do the searching manually, just to get an even better understanding of how Linear Search works before actually implementing it in a programming language. We will search for value 11.

Step 1: We start with an array of random values: [12, 8, 9, 11, 5, 11]

Step 2: We look at the first value in the array, is it equal to 11?

[12, 8, 9, 11, 5, 11]

Step 3: We move on to the next value at index 1, and compare it to 11 to see if it is equal:

[12, 8, 9, 11, 5, 11]

Step 4: We check the next value at index 2: [12, 8, 9, 11, 5, 11]

Step 5: We move on to the next value at index 3. Is it equal to 11?

[12, 8, 9, 11, 5, 11]

We have found it!

Value 11 is found at index 3.

Returning index position 3.

Linear Search is finished.

Linear Search Time Complexity

Linear Search compares each value with the value it is looking for. If the value is found, the index is returned, and if it is not found -1 is returned.

To find the time complexity for Linear Search, let's see if we can fins out how many compare operations are needed to find a value in an array with n values.

Best Case Scenario is if the value we are looking for is the first value in the array. In such a case only one compare is needed and the time complexity is O(1).

Worst Case Scenario is if the whole array is looked through without finding the target value. In such a case all values in the array are compared with the target value, and the time complexity is O(n).

Average Case Scenario is not so easy to pinpoint. What is the possibility to finding the target value? That depends on the values in the array right? But if we assume that exactly one of the values in the array is equal to the target value, and that the position of that value can be anywhere, the average time needed for Linear Search is half of the time time needed in the worst case scenario.

Time complexity for Linear Search is O(n).

Binary Search

The Binary Search algorithm searches through an array and returns the index of the value it searches for.

Let's try to do the searching manually, just to get an even better understanding of how Binary Search works before actually implementing it in a programming language. We will search for value 11.

Step 1: We start with an array: [2, 3, 7, 7, 11, 15, 25]

Step 2: The value in the middle of the array at index 3, is it equal to 11?

[2, 3, 7, 7, 11, 15, 25]

Step 3: 7 is less than 11, so we must search for 11 to the right of index 3. The values to the right of index 3 are [11, 15, 25]. The next value to check is the middle value 15, at index 5.

[2, 3, 7, 7, 11, 15, 25]

Step 4: 15 is higher than 11, so we must search to the left of index 5. We have already checked index 0-3, so index 4 is only value left to check: [2, 3, 7, 7, 11, 15, 25]

We have found it!

Value 11 is found at index 4.

Returning index position 4.

Binary Search is finished.

Binary Search Time Complexity

Binary Search finds the target value in an already sorted array by checking the centre value. If the centre value is not the target value, Linear Search selects the left or right sub-array and continues the search until the target value is found.

To find the time complexity for Binary Search, let's see how many compare operations are needed to find the target value in an array with n values.

The best case scenario is if the first middle value is the same as the target value. If this happens the target value is found straight away, with only one compare, so the time complexity is O(1) in this case.

The worst case scenario is if the search area must be cut in half over and over until the search area is just one value. When this happens, it does not affect the time complexity if the target value is found or not.

Let's consider array lengths that are powers of 2, like 2, 4, 8, 16, 32 64 and so on.

How many times must 2 be cut in half until we are looking at just one value? It is just one time, right?

How about 8? We must cut an array of 8 values in half 3 times to arrive at just one value.

An array of 32 values must be cut in half 5 times.

We can see that $2=2^1$, $8=2^3$ and $32=2^5$. So the number of times we must cut an array to arrive at just one element can be found in the power with base 2. Another way to look at it is to ask "how many times must I multiply 2 with itself to arrive at this number?". Mathematically we can use the base-2 logarithm, so that we can find out that an array of length n can be split in half $\log_2(n)$ times.

This means that time complexity for Binary Search is $O(\log_2 n)$

The average case scenario is not so easy to pinpoint, but since we understand time complexity of an algorithm as the upper bound of the worst case scenario, using Big O notation, the average case scenario is not that interesting.

Note: Time complexity for Binary Search $O(\log_2 n)$ is a lot faster than Linear Search O(n), but it is important to remember that Binary Search requires a sorted array, and Linear Search does not.

ARRAY

Let the size of the element stored in a matrix A[8][3] be 4 bytes, if base address is 3500, find address of A[5][2] in Row and Column order.

Row-major order calculation:

The formula to calculate the address of A[i][j] in row-major order is:

$\text{Address}_{\text{row-major}}(A[i][j]) = \text{Base address} + \text{Offset}$

Where:

- Base address is the starting address of the matrix.
- Offset is calculated as follows:
- Offset = row × number of columns + column

  Here, row = 5, column = 2, and number of columns = 3.

Plugging in the values: Offset = 5 × 3 + 2 = 15 + 2 = 17

Therefore, $\text{Address}_{\text{row-major}}(A[5][2]) = 3500 + (17 \times 4) = 3500 + 68 = 3568$

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

Column-major order calculation:

The formula to calculate the address of A[i][j] in column-major order is:

$\text{Address}_{\text{column-major}}(A[i][j]) = \text{Base address} + \text{Offset}$

The offset for column-major order is calculated differently:

Offset = column × number of rows + row

Here, row = 5, column = 2, and number of rows = 8.

Plugging in the values:

Offset = 2 × 8 + 5 = 16 + 5 = 21

Therefore, $\text{Address}_{\text{column-major}}(A[5][2]) = 3500 + (21 \times 4) = 3500 + 84 = 3584$

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

Define the term array. How 2D Array stored in memory?

An array is a contiguous block of memory locations, each of which holds an element of the same data type. The elements of an array are indexed starting from zero (for the first element) up to the size of the array minus one.

Storage of 2D Array in Memory:

A 2D (two-dimensional) array is essentially an array of arrays, where each element is itself an array. In memory, elements of a 2D array are stored in a row-major or column-major order depending on the programming language and storage convention.

Let's consider a 2D array A[m][n]:

- 'm' represents the number of rows and 'n' represents the number of columns.

Row-major order storage (most common):

In row-major order, the elements of each row are stored contiguously in memory. The entire array is stored row by row. To compute the memory address of an element 'A[i][j]' in a row-major order:

- Each element occupies a fixed size (e.g., 4 bytes for an integer).
- The address calculation is: 'address(A[i][j]) = base_address + (i * n + j) * element_size'

  -> 'base_address' is the starting address of the array.

  -> 'i' is the row index. -> 'j' is the column index.

  -> 'n' is the number of columns. -> 'element_size' is the size (in bytes) of each element.

Column-major order storage:

In column-major order, the elements of each column are stored contiguously in memory. The entire array is stored column by column. The address calculation for A[i][j] in column-major order is similar to row-major order but takes into account the number of rows 'm':

- 'address(A[i][j]) = base_address + (j * m + i) * element_size'
- 'base_address' is the starting address of the array.
- 'i' is the row index, 'j' is the column index, 'm' is the number of rows.
- 'element_size' is the size (in bytes) of each element.

Represent 3 Tuple form. What is sparse matrix?

A 3-tuple representation, often used in the context of representing a sparse matrix, consists of three components: (row index, column index, value). This representation is particularly useful for matrices where most of the elements are zero, as it allows us to efficiently store and manipulate only the non-zero elements.

Let's break down the components of this 3-tuple representation:

1. Row Index: This refers to the row number of a non-zero element in the matrix.

2. Column Index: This refers to the column number of a non-zero element in the matrix.

3. Value: This is the actual value of the non-zero element located at the specified row and column indices.

For example, consider a sparse matrix represented as:

[[0, 0, 0, 0],

[5, 0, 0, 0],

[0, 0, 3, 0],

[0, 0, 0, 0]]

[(1, 0, 5), (2, 2, 3)]

Here, the tuple (1, 0, 5) indicates that there is a non-zero value (5) at row 1 and column 0, and (2, 2, 3) indicates a non-zero value (3) at row 2 and column 2.

Sparse matrix and how to store it?:

A sparse matrix is a matrix that is composed mostly of zero elements. The opposite of a sparse matrix is a dense matrix, which has a significant number of non-zero elements. Sparse matrices arise commonly in various applications like scientific computing, engineering simulations, and machine learning, where the underlying data is often sparse.

Sparse matrices are typically stored and manipulated using specialized data structures and representations (like the 3-tuple form mentioned above) that efficiently store and operate on only the non-zero elements. Storing sparse matrices in this way can save memory and computation time compared to dense representations, especially when dealing with large matrices where the majority of elements are zero.

What is an Abstract data type?

An Abstract Data Type (ADT) is a theoretical model or concept in computer science that defines a set of operations and behaviors without specifying the implementation details. It focuses on what operations can be performed on the data, what the behavior of these operations should be, and what properties the data structure should have, rather than how these operations are implemented internally.

Key characteristics of an Abstract Data Type include:

Encapsulation: The data and operations are encapsulated together into a single unit. The internal details of how the data is stored or manipulated are hidden from the user.

Defined Operations: An ADT specifies a set of operations that can be performed on the data. These operations define the interface of the ADT.

Behavior Specification: For each operation defined by the ADT, there is a specification of what the operation does, including its input parameters, its effect on the data structure, and its output or return value.

Data Abstraction: The user interacts with the ADT through its operations without needing to know the specific implementation details. This abstraction allows for modularity and separation of concerns in software design.

Common examples of Abstract Data Types include:

Stack: An ADT that allows operations like push (to add an element), pop (to remove the top element), and peek (to view the top element), typically following Last-In-First-Out (LIFO) behavior.

Queue: An ADT that supports operations like enqueue (to add an element to the back) and dequeue (to remove an element from the front), typically following First-In-First-Out (FIFO) behavior.

List: An ADT that represents a collection of elements with operations like insert, delete, search, and traversal.

Tree: An ADT that represents hierarchical relationships among elements, with operations like insertion, deletion, and traversal (e.g., inorder, preorder, postorder).

Graph: An ADT that represents a set of vertices and edges, with operations for adding vertices, edges, and traversing the graph.

Differentiate Linear vs Non-linear data structure.

Linear Data Structure: Represents a sequential arrangement of elements where each element has a unique predecessor and successor (except for the first and last elements). Operations like traversal and access are typically straightforward, and memory allocation is often simpler compared to non-linear structures.

Non-linear Data Structure: Represents data in a hierarchical manner where elements can have multiple predecessors or successors. This includes structures like trees and graphs. Traversal and manipulation of non-linear structures often involve more complex algorithms due to the hierarchical relationships among elements.

Aspect	Linear Data Structure	Non-linear Data Structure
Definition	Data elements are arranged in a sequential order.	Data elements are not arranged in a sequential order.
Representation	Typically represented in a linear sequence (e.g., arrays, linked lists).	Represented using hierarchical relationships (e.g., trees, graphs).
Memory Allocation	Uses contiguous memory allocation.	Uses non-contiguous memory allocation.
Traversal	Linear traversal such as iteration or recursion.	Non-linear traversal, such as Depth-First or Breadth-First
Examples	Arrays, Linked Lists, Stacks, Queues.	Trees, Graphs, Heaps
Operations	Operations like insertion and deletion can be complex	Operations may be complex depending on the structure
Memory Utilization	Less flexible in memory utilization.	More flexible in memory utilization

STACK

Define Stack. Write push() and pop() function.

Definition of Stack:

- A stack is a linear data structure consisting of a collection of elements.
- Elements are added and removed based on the Last In, First Out (LIFO) principle.

'push()' Function

```
int push(Stack *stack, int value) {
    if (stack->top >= MAX_SIZE - 1) {
        printf("Stack overflow: Cannot push element, stack is full.\n");
        return 0;    // Return 0 to indicate failure
    }
    stack->data[++stack->top] = value;
    return 1;    // Return 1 to indicate success
}
```

'pop()' Function

```
int pop(Stack *stack) {
    if (stack->top < 0) {
        printf("Stack underflow: Cannot pop element, stack is empty.\n");
        return -1;    // Return -1 (or any other appropriate error code) for failure
    }
    int value = stack->data[stack->top--];
    return value;    // Return the popped element
}
```

Describe the algorithm to evaluate postfix expression.

1. Initialize an Empty Stack:

Start by creating an empty stack. This stack will be used to temporarily hold operands as we process the postfix expression.

2. Iterate Through the Postfix Expression:

Traverse the postfix expression from left to right. Each element in the postfix expression can be an operand (like numbers) or an operator (+, -, *, /, etc.).

3. Process Each Element:

For each element in the postfix expression:

- If the element is an operand (i.e., a number):

  Convert it from string format to a numeric value and push it onto the stack.

- If the element is an operator:

  - Pop the top two elements (operands) from the stack.

  - Apply the operator to these operands in the order: (second operand) operator (first operand).

  - Push the result back onto the stack.

4. Continue Until End of Expression:

Repeat the above steps for each element in the postfix expression until you reach the end of the expression.

5. Final Result:

After processing all elements of the postfix expression, the stack will contain only one element, which is the final result of the expression.

6. Handle Errors:

- Ensure that the postfix expression is valid.
- Ensure that there are enough operands for each operator encountered.
- Handle any potential errors such as division by zero.

Describe the algorithm to convert infix to postfix expression.

Let's convert the infix expression 3 + 4 * 5 - 7 to postfix notation:

1. Start with an empty stack and an empty output stack.

2. Process each token:

- 3: Directly add to the output stack.
- +: Push to the stack.
- 4: Directly add to the output stack.
- *: Pop + from stack to output stack (since * has higher precedence). Push * to stack.
- 5: Directly add to the output stack.
- -: Pop * from stack to output stack (since * has same precedence but * is left-associative). Push - to stack.
- 7: Directly add to the output stack.

3. Pop any remaining operators from stack to output stack (-).

4. The output stack becomes 3 4 5 * + 7 -, which is the postfix equivalent of the infix expression 3 + 4 * 5 - 7.

Evaluate the postfix expression 3 1 + 2 ^ 7 4 -2 * + 5 -

Step-by-step Evaluation:

1. Push '3' onto the stack. -> Stack: [3]

2. Push 1 onto the stack. -> Stack: [3, 1]

3. Encounter '+' (addition): Pop '1' and '3' (operands) from the stack, calculate '3 + 1 = 4', and push '4' back onto the stack. -> Stack: [4]

4. Push '2' onto the stack. -> Stack: [4, 2]

5. Encounter '^' (exponentiation): Pop '2' and '4' (operands) from the stack, calculate '2 ^ 4 = 16', and push '16' back onto the stack. -> Stack: [16]

6. Push '7' onto the stack. -> Stack: [16, 7]

7. Push '4' onto the stack. -> Stack: [16, 7, 4]

8. Encounter '-' (subtraction): Pop '4' and '7' (operands) from the stack, calculate '7 - 4 = 3', and push '3' back onto the stack. -> Stack: [16, 3]

9. Encounter '2' (push onto stack): Push '2' onto the stack. -> Stack: [16, 3, 2]

10. Encounter '*' (multiplication): Pop '2' and '3' (operands) from the stack, calculate '3 * 2 = 6', and push '6' back onto the stack. -> Stack: [16, 6]

11. Encounter '+' (addition): Pop '6' and '16' (operands) from the stack, calculate '16 + 6 = 22', and push '22' back onto the stack. -> Stack: [22]

12. Push '5' onto the stack. -> Stack: [22, 5]

13. Encounter '-' (subtraction): Pop '5' and '22' (operands) from the stack, calculate '22 - 5 = 17', and push '17' back onto the stack. -> Stack: [17]

Final Result: The final result of evaluating the postfix expression '3 1 + 2 ^ 7 4 - 2 * + 5 - is 17'.

Convert the infix expression in postfix expression 9 + 5 * 7 - 6 ^ 2 + 15/3

For the infix expression 9 + 5 * 7 - 6 ^ 2 + 15/3:

Tokens: 9, +, 5, *, 7, -, 6, ^, 2, +, 15, /, 3

Use the shunting yard algorithm to convert:

- '9' (operand) goes directly to output.
- '+' (operator) is pushed onto the stack.
- '5' (operand) goes directly to output.
- '*' (operator) is pushed onto the stack (higher precedence than '+').
- '7' (operand) goes directly to output.
- When encountering - (operator), '*' is popped to output (since '*' has higher precedence than '-').
- '-' (operator) is pushed onto the stack.
- '6' (operand) goes directly to output.
- '^' (operator) is pushed onto the stack (highest precedence).
- '2' (operand) goes directly to output.
- 'W'hen encountering '+' (operator), both '^' and '-' are popped to output (higher precedence than '+').
- '+' (operator) is pushed onto the stack.
- '15' (operand) goes directly to output.
- '/' (operator) is pushed onto the stack.
- '3' (operand) goes directly to output.
- Pop remaining operators (/) to output.

The postfix expression equivalent to '9 + 5 * 7 - 6 ^ 2 + 15/3' is '9 5 7 * + 6 2 ^ - 15 3 / +'.

QUEUE

Write an algorithm to insert an element in Linear Queue.

1. Initialize:

- 'front' = 0 (start of the queue).
- 'rear' = -1 (empty queue).
- 'queue' with sufficient size.

2. Check if Queue is Full:

- If 'rear' is at the end of the queue ('rear' == size_of_queue - '1'), the queue is full.

3. Enqueue Operation:

- Increment 'rear' by '1'.
- Set queue['rear'] to the new element.

Data Structures:

- 'queue': Array to hold queue elements.
- 'front': Index of the front element.
- 'rear': Index of the rear element.

Operations:

- 'enqueue(element)': Add an element to the rear of the queue.
- 'isFull()': Check if the queue is full.
- 'isEmpty()': Check if the queue is empty.

Write an algorithm to delete an element in Linear Queue.

1. Initialize:

- 'front' = 0 (start of the queue).
- 'rear' = -1 (empty queue).
- 'queue' with sufficient size.

2. Check if Queue is Empty:

- If 'front' > 'rear', then queue is empty. Throw an underflow error if trying to dequeue from an empty queue.

3. Dequeue Operation:

- Increment 'front' by '1' to remove the front element.
- Return the removed element.

Data Structures:

- 'queue': Array to hold queue elements.
- 'front': Index of the front element.
- 'rear': Index of the rear element.

Operations:

- 'enqueue(element)': Add an element to the rear of the queue.
- 'dequeue()': Remove and return the element at the front of the queue.

Write an algorithm to display an element in Linear Queue.

1. Check if Queue is Empty:

- If 'front' is greater than 'rear', the queue is empty. Display an appropriate message and return.

2. Display Queue Elements:

- Initialize a loop variable 'i' starting from 'front'.
- Iterate while 'i' is less than or equal to 'rear':

    -> Print 'queue[i]'.

    -> Increment 'i' to move to the next element.

Data Structures:

- 'queue': Array to hold queue elements.
- 'front': Index of the front element.
- 'rear': Index of the rear element.

Operations:

- 'displayQueue()': Display all elements currently in the queue.

Write an algorithm to insert an element in Circular Queue.

1. Initialize:

- 'front' = -1 (empty queue indicator).
- 'rear' = -1 (empty queue indicator).
- 'size_of_queue' (maximum capacity of the queue).
- 'queue' array of 'size size_of_queue'.

2. Check if Queue is Full:

- If 'rear' is immediately followed by 'front' (considering circular wrapping), the queue is full.

3. Enqueue Operation:

- If the queue is not full ('!isFull()'):

    -> If the queue is initially empty ('front is -1'):

        -> Set 'front' to 0.

    -> Increment 'rear' using 'getNextIndex()' to handle circular wrapping.

    -> Insert the element at 'queue[rear]'.

Data Structures:

- 'queue': Array to hold queue elements.
- 'front': Index of the front element and 'rear': Index of the rear element.
- 'size_of_queue': Maximum capacity of the queue.

Operations:

- 'enqueue(element)': Add an element to the rear of the queue.
- 'isFull()': Check if the queue is full.
- 'isEmpty()': Check if the queue is empty.
- 'getNextIndex(index)': Helper function to get the next circular index.

Write an algorithm to delete an element in Circular Queue.

1. Check if Queue is Empty:

- If 'isEmpty()', then the queue is empty. Throw an underflow error.

2. Dequeue Operation:

- Save the front element ('queue[front]') to a variable to return later.
- If 'front' is equal to 'rear' (indicating the queue has only one element):

    -> Reset 'front' and 'rear' to '-1' (empty queue).

- Otherwise, increment 'front' using 'getNextIndex()' to handle circular wrapping.

Data Structures:

- 'queue': Array to hold queue elements.
- 'front': Index of the front element.
- 'rear': Index of the rear element.
- 'size_of_queue': Maximum capacity of the queue.

Operations:

- 'dequeue()': Remove and return the element at the front of the queue.
- 'isEmpty()': Check if the queue is empty.
- 'getNextIndex(index)': Helper function to get the next circular index.

Write an algorithm to display an element in Circular Queue.

1. Check if Queue is Empty:

- If 'isEmpty()', then the queue is empty. Display an appropriate message.

2. Display Queue Elements:

- Initialize an index 'current' starting from 'front'.
- Loop while 'current' is less than or equal to 'rear' (taking into account circular wrapping using 'getNextIndex()').

    -> Print 'queue[current]'.

    -> Update 'current' to the next index using 'getNextIndex(current)'.

Data Structures:

- 'queue': Array to hold queue elements.
- 'front': Index of the front element.
- 'rear': Index of the rear element.
- 'size_of_queue': Maximum capacity of the queue.

Operations:

- 'displayQueue()': Display all elements currently in the circular queue.

What is Priority Queue?

A Priority Queue is a type of abstract data structure that operates similar to a regular queue or stack, but with a key distinction: elements in a priority queue are assigned a priority. This priority determines the order in which elements are processed or removed from the queue. Elements with a higher priority are processed or removed before elements with a lower priority, and elements with the same priority are typically processed or removed based on other criteria, such as order of insertion.

LINKED LIST

Write algorithms to do the following operations on singly-linked list.

(i) insert a node at beginning (ii) insert a node at the end

(iii) delete a node at beginning (iv) delete a node ate the end

(v) search a node (vi) count number of nodes

(i) Algorithm to insert a node at the beginning of a singly-linked list:

1. Allocate Memory for the New Node: Use malloc to allocate memory for the new node.

2. Assign Data to the New Node: Set the data of the new node to the value you want to insert.

3. Link the New Node to the Current Head: Set the next pointer of the new node to point to the current head of the list.

4, Update the Head of the List: Make the new node the head of the list by updating the head pointer to point to the new node.

(ii) Algorithm to insert a node at the end of a singly-linked list:

1. Define the Node Structure: First, define the structure for the nodes of the singly-linked list. Each node will contain some data and a pointer to the next node.

2. Create a Function to Insert a Node: This function will take the head of the list and the data to be inserted as parameters. It will create a new node, set its data, and then traverse the list to find the last node and insert the new node after it.

(iii) Algorithm to delete a node at the beginning of a singly-linked list:

1. Define the Node Structure: If you haven't already, define the structure for the nodes of the singly-linked list. Each node will contain some data and a pointer to the next node.

2. Create a Function to Delete a Node: This function will take the head of the list as a

parameter. It will check if the list is empty. If it's not, it will update the head to point to the second node in the list and then free the memory of the original head node.

(iv) Algorithm to delete a node at the end of a singly-linked list:

1. Define the Node Structure: If you haven't already, define the structure for the nodes of the singly-linked list. Each node will contain some data and a pointer to the next node.

2. Create a Function to Delete a Node: This function will take the head of the list as a parameter. It will traverse the list to find the last node and then update the second-to-last node's next pointer to NULL, effectively removing the last node from the list.

(v) Algorithm to search for a node in a singly-linked list:

1. Define the Node Structure: If you haven't already, define the structure for the nodes of the singly-linked list. Each node will contain some data and a pointer to the next node.

2. Create a Function to Search for a Node: This function will take the head of the list and the data to be searched as parameters. It will traverse the list, comparing each node's data with the search data. If a match is found, it will return a pointer to that node. If no match is found, it will return NULL.

(vi) Algorithm to count the number of nodes in a singly-linked list:

1. Define the Node Structure: If you haven't already, define the structure for the nodes of the singly-linked list. Each node will contain some data and a pointer to the next node.

2. Create a Function to Count the Nodes: This function will take the head of the list as a parameter. It will traverse the list, incrementing a counter for each node it encounters. Once it reaches the end of the list (indicated by a NULL next pointer), it will return the count.

Write algorithms to do the following operations on singly-linked list.

(i) insert a node at beginning (ii) insert a node at the end

(iii) delete a node at beginning (iv) delete a node ate the end

(v) search a node (vi) count number of nodes

(i) Algorithm to insert a node at the beginning of a doubly-linked:

1. Define the Node Structure: First, define the structure for the nodes of the doubly-linked list. Each node will contain some data and pointers to the next and previous nodes.

2. Create a Function to Insert a Node: This function will take the head of the list and the data to be inserted as parameters. It will create a new node, set its data, and then insert it at the beginning of the list by updating the pointers of the new node and the current head node.

(ii) Algorithm to insert a node at the end of a doubly-linked list:

1. Define the Node Structure: First, define the structure for the nodes of the doubly-linked list. Each node will contain some data and pointers to the next and previous nodes.

2. Create a Function to Insert a Node: This function will take the head of the list and the data to be inserted as parameters. It will create a new node, set its data, and then traverse the list to find the last node and insert the new node after it by updating the pointers of the new node and the last node.

(iii) Algorithm to delete a node at the beginning of a doubly-linked list:

1. Define the Node Structure: First, define the structure for the nodes of the doubly-linked list. Each node will contain some data and pointers to the next and previous nodes.

2. Create a Function to Delete a Node: This function will take the head of the list as a parameter. It will check if the list is empty. If it's not, it will update the head to point to the second node in the list and then free the memory of the original head node.

(iv) Algorithm to delete a node at the end of a doubly-linked list:

1. Define the Node Structure: First, define the structure for the nodes of the doubly-linked list. Each node will contain some data and pointers to the next and previous nodes.

2. Create a Function to Delete a Node: This function will take the head of the list as a parameter. It will traverse the list to find the last node and then update the second-to-last node's next pointer to NULL, effectively removing the last node from the list.

(v) Algorithm to search for a node in a doubly-linked list:

1. Define the Node Structure: First, define the structure for the nodes in the doubly-linked list. Each node will have a data field and two pointers, one pointing to the next node and another pointing to the previous node.

2. Create the Doubly-Linked List: Implement functions to create a new node, insert a node at the beginning, insert a node at the end, and display the list.

3. Search for a Node: Implement a function to search for a node in the doubly-linked list. This function will traverse the list from the head node to the tail node, comparing each node's data with the target data.

(vi) Algorithm to count the number of nodes in a doubly-linked list:

1. Define the Node Structure: If you haven't already, define the structure for the nodes in the doubly-linked list. Each node will have a data field and two pointers, one pointing to the next node and another pointing to the previous node.

2. Create the Doubly-Linked List: Implement functions to create a new node, insert a node at the beginning, insert a node at the end, and display the list. These functions were covered in the previous response.

3. Count the Nodes: Implement a function to count the number of nodes in the doubly-linked list. This function will traverse the list from the head node to the tail node, incrementing a counter for each node it encounters.

Write algorithms to merge two sorted linked list.

Step 1: Define the Node Structure

First, define the structure for a node in the linked list. Each node contains an integer value and a pointer to the next node.

Step 2: Function to Create a New Node

This function will be used to create a new node with a given value.

Step 3: Function to Merge Two Sorted Linked Lists

This function takes two sorted linked lists and merges them into a single sorted linked list.

Step 4: Function to Print the Linked List

This function is used to print the elements of the linked list.

Step 5: Main Function

Finally, write a main function to test the merging of two sorted linked lists.

TREE

What is Binary Search Tree?

A Binary Search Tree (BST) is a tree data structure in which each node has at most two children, referred to as the left child and the right child. For each node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater than the node. This property makes the BST an ordered or sorted binary tree.

Terms

Path – Path refers to sequence of nodes along the edges of a tree.

Root – Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.

Parent – Any node except root node has one edge upward to a node called parent.

Child – Node below a given node connected by its edge downward is called its child node.

Leaf – Node which does not have any child node is called leaf node.

Subtree – Subtree represents descendents of a node.

Visiting – Visiting refers to checking value of a node when control is on the node.

Traversing – Traversing means passing through nodes in a specific order.

Levels – Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.

keys – Key represents a value of a node based on which a search operation is to be carried out for a node.

Here's a basic structure of a node in a Binary Search Tree:

```
struct node {
    int data;
    struct node *leftChild;
    struct node *rightChild;
};
```

Key Characteristics of a Binary Search Tree:

1. Order Property: For any given node, the value of all nodes in its left subtree is less than the value of the node, and the value of all nodes in its right subtree is greater than the value of the node.
2. Search Operation: Searching for a value in a BST is efficient. Starting from the root, you can eliminate half of the tree from consideration at each step.
3. Insertion: Inserting a new node into a BST involves finding the correct location in the tree where the new node should be placed to maintain the order property.
4. Deletion: Deleting a node from a BST involves finding the node to be deleted, removing it, and then rearranging the tree to maintain the order property.
5. Balanced vs Unbalanced: A balanced BST is one where the heights of the two child subtrees of any node differ by at most one. This is important for maintaining the efficiency of operations.

Preorder Traversal: It is a simple three step process.

- visit root node
- traverse left subtree
- traverse right subtree

Inorder Traversal: It is a simple three step process.

- traverse left subtree
- visit root node
- traverse right subtree

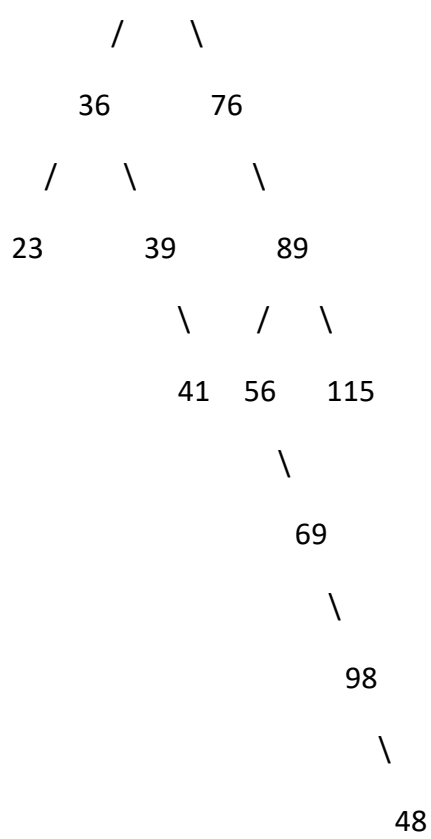
Postorder Traversal: It is a simple three step process.

- traverse left subtree
- traverse right subtree
- visit root node

Make a BST for the sequence: 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48. Traverse the tree in Pre-order, In-order, Post-order.

Inserting sequence: 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

BST after inserting:



Traversing the BST

Pre-order Traversal (Root-Left-Right)

Pre-order traversal: 45, 36, 23, 39, 41, 76, 89, 56, 69, 98, 115, 48

In-order Traversal (Left-Root-Right)

In-order traversal: 23, 36, 39, 41, 45, 56, 69, 76, 89, 98, 115, 48

Post-order Traversal (Left-Right-Root)

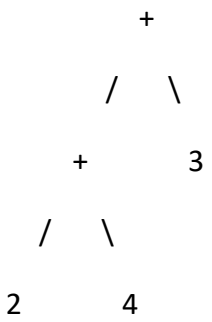
Post-order traversal: 23, 41, 39, 36, 69, 56, 98, 48, 115, 89, 76,

Represent the the expression 2 + 4 + 3 in expression table. Show the Pre-order, In-order and Post-order.

To represent the expression 2+4+32+4+3 in an expression tree and then demonstrate the Pre-order, In-order, and Post-order traversals, we will construct a binary tree where the operators (+) are internal nodes and the operands (numbers) are leaf nodes.

Expression Tree Construction

We will construct the expression tree for 2+4+32+4+3 as follows:



Pre-order Traversal (Root-Left-Right)

In pre-order traversal, we visit the root node first, then recursively do a pre-order traversal of the left subtree, followed by a pre-order traversal of the right subtree.

Pre-order traversal: +, +, 2, 4, 3

In-order Traversal (Left-Root-Right)

In in-order traversal, we recursively do an in-order traversal of the left subtree, then visit the root node, followed by an in-order traversal of the right subtree.

In-order traversal: 2, +, 4, +, 3

Post-order Traversal (Left-Right-Root)

In post-order traversal, we recursively do a post-order traversal of the left subtree, then recursively do a post-order traversal of the right subtree, followed by visiting the root node.

Post-order traversal: 2, 4, +, 3, +

Construct a Binary Tree whose nodes In-order: 10, 15, 17, 18, 20, 25, 30, 35, 38, 40, 50 Pre-order: 20, 15, 10, 18, 17, 30, 25, 40, 35, 38, 50

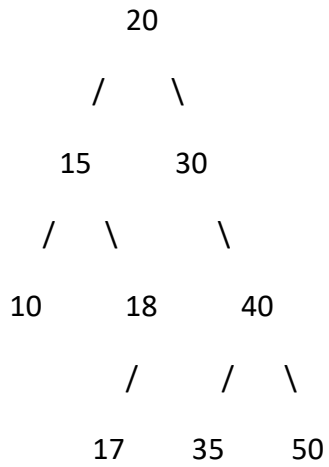
In-order: 10, 15, 17, 18, 20, 25, 30, 35, 38, 40, 50

Pre-order: 20, 15, 10, 18, 17, 30, 25, 40, 35, 38, 50

Binary Tree Construction:

1. **Identify the Root:** The root of the binary tree is 20 (first node in Pre-order).
2. **Locate Root in In-order Traversal:** Root node 20 is located at the center of the In-order traversal. Nodes to the left belong to the left subtree, and nodes to the right belong to the right subtree.
3. **Recursively Construct Left and Right Subtrees:**
    - For the left subtree:
        - In-order: 10, 15, 17, 18
        - Pre-order: 15, 10, 18, 17
    - For the right subtree:
        - In-order: 25, 30, 35, 38, 40, 50
        - Pre-order: 30, 25, 40, 35, 38, 50

Constructed Binary Tree:



In-order Traversal: E I C F J B G D K H L A, Pre-order Traversal: A B C E I F J D G H K L, Draw the tree and write Post-order.

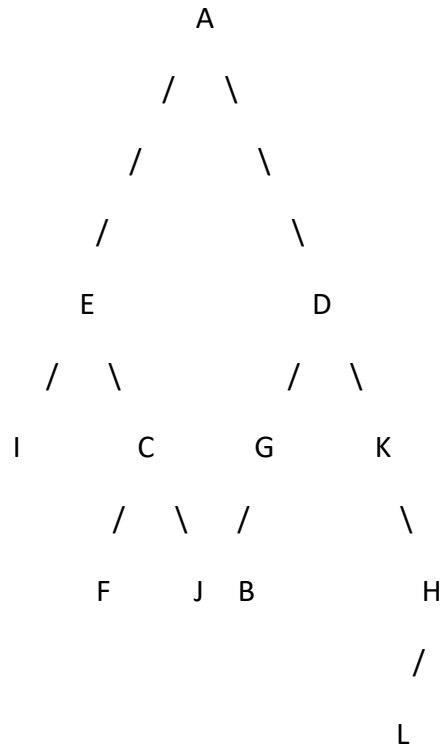
Identify the Root: The root of the binary tree is the first node in the Pre-order traversal (**A**).

Locate Root in In-order Traversal: Find the root node **A** in the In-order traversal. Nodes to the left of **A** belong to the left subtree, and nodes to the right belong to the right subtree.

Recursively Construct Left and Right Subtrees:

- For the left subtree:
    - In-order: **E I C F J B G D K H L**, Pre-order: **A B C E I F J D G H K L**
- For the right subtree:
    - In-order: **B G D K H L**, Pre-order: **D G B K H L**

Constructed Binary Tree:



Post-order Traversal (Left-Right-Root):

Post-order traversal: I F J C E B G L H K D A

In-order Traversal: H D B I E A F J C K G L, Pre-order Traversal: H D I E B J F K L G C A, Draw the tree and write Post-order.

Identify the Root: The root of the binary tree is the first node in the Pre-order traversal (**H**).

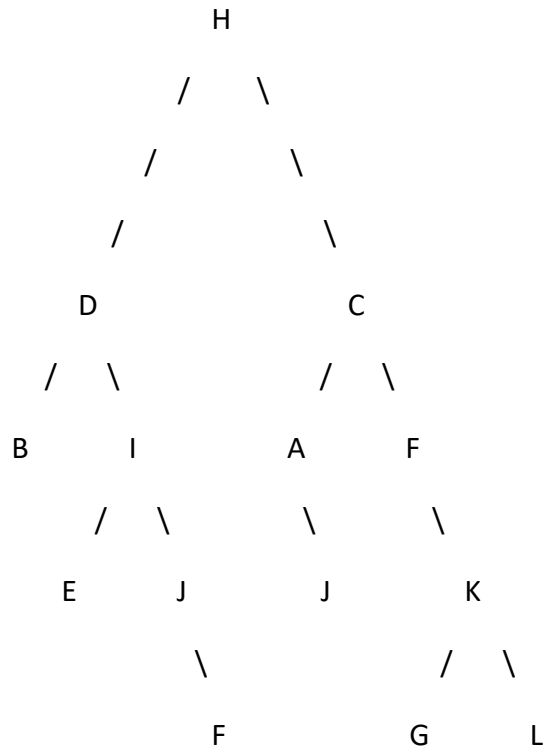
Locate Root in In-order Traversal: Find the root node **H** in the In-order traversal. Nodes to the left of **H** belong to the left subtree, and nodes to the right belong to the right subtree.

Recursively Construct Left and Right Subtrees:

For the left subtree: In-order: **D B I E A F J C K G L**, Pre-order: **H D I E B J F K L G C A**

For the right subtree: In-order: **A F J C K G L**, Pre-order: **C A F J K G L**

Constructed Binary Tree:



Post-order Traversal (Left-Right-Root):

Performing a Post-order traversal on the constructed binary tree:

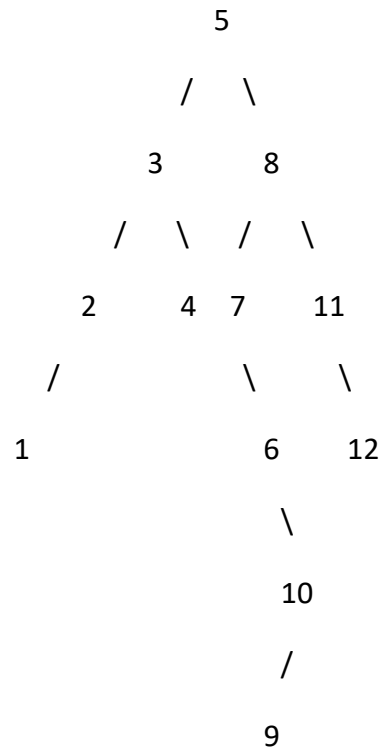
Post-order traversal: B E J F I D A J G L K C H

What is AVL Tree?

Construct AVL Tree in order 3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10, 9

An AVL tree is a self-balancing binary search tree (BST) where the heights of the two child subtrees of any node differ by at most one. This balancing property helps in maintaining efficient search, insertion, and deletion operations, which ensures that the tree remains balanced and operations run in O(log n) time complexity on average.

Let's construct an AVL tree from the given sequence: 3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10, 9.



In this AVL tree:

- The heights of the left and right subtrees of every node differ by at most one.
- All insertion operations maintain the AVL tree balancing property through rotation operations when necessary.

Define Balance Factor.

The balance factor of a node in an AVL tree is a numerical value that indicates the difference in height between the left subtree and the right subtree of that node. Specifically, the balance factor is calculated as:

Balance Factor = height of left subtree − height of right subtree

In the context of AVL trees, where the goal is to maintain the tree's balance (i.e., keep the heights of left and right subtrees nearly equal), the balance factor plays a crucial role. The balance factor can have one of three possible values:

1. **Balance Factor = 0**: The heights of the left and right subtrees are equal.
2. **Balance Factor = +1**: The left subtree is taller by one level compared to the right subtree.
3. **Balance Factor = -1**: The right subtree is taller by one level compared to the left subtree.

The balance factor of a node helps determine whether the tree needs to be rebalanced after an insertion or deletion operation to maintain the AVL tree properties (i.e., ensuring that the balance factor of every node is within the range [-1, 0, +1]).

During tree rotations (such as left rotation, right rotation, left-right rotation, or right-left rotation), adjustments are made to the balance factors of affected nodes to restore balance to the tree.

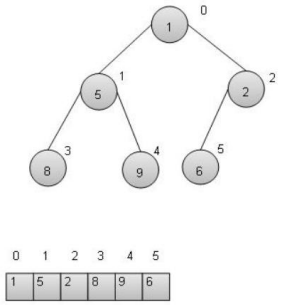
HEAP

Heap represents a special tree-based data structure used to represent priority queue or for heap sort. We'll going to discuss binary heap tree specifically.

Binary heap tree can be classified as a binary tree with two constraints −

Completeness − Binary heap tree is a complete binary tree except the last level which may not have all elements but elements from left to right should be filled in.

Heapness − All parent nodes should be greater or smaller to their children. If parent node is to be greater than its child then it is called Max heap otherwise it is called Min heap. Max heap is used for heap sort and Min heap is used for priority queue. We're considering Min Heap and will use array implementation for the same.



Following are basic primary operations of a Min heap which are following.

Insert − insert an element in a heap.

Get Minimum − get minimum element from the heap.

Remove Minimum − remove the minimum element from the heap

GRAPH

What is Graph?

A graph is a non-linear data structure that consists of nodes (or vertices) and edges. The nodes are the entities of the graph, and the edges are the connections between these nodes. Graphs are used to represent many real-world problems, such as social networks, web pages, and transportation networks.

There are two main types of graphs in DSA:

Directed Graph (Digraph): In a directed graph, the edges have a direction, meaning they go from one node to another in a specific direction. This is similar to the concept of one-way streets in a city.

Undirected Graph: In an undirected graph, the edges do not have a direction. This means that if there is an edge between two nodes, you can travel from one node to the other in both directions.

Graphs can be classified into several categories based on their properties:

Weighted Graph: Each edge has a weight or cost associated with it. This is useful in scenarios where the cost of traversing an edge is not the same for all edges.

Unweighted Graph: The edges do not have any weight associated with them. This is the simplest form of a graph.

Cyclic Graph: A graph is cyclic if there is a path that starts and ends at the same vertex.

Acyclic Graph: A graph is acyclic if it does not contain any cycles.

Connected Graph: A graph is connected if there is a path between every pair of vertices.

Disconnected Graph: A graph is disconnected if there is at least one pair of vertices for which there is no path.

Bipartite Graph: A bipartite graph is a graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set.

Why Graph is called Non-linear data structure?

In a graph, data elements (nodes) are connected to each other in a non-linear fashion. The connections between nodes can be direct (edges) or indirect (through other nodes), and these connections do not follow a strict linear order. Nodes in a graph can be connected to any number of other nodes, and the connections can form complex structures that do not resemble a straight line.

In summary, graphs are called non-linear data structures because they do not arrange data elements in a linear sequence. Instead, they represent relationships between entities in a complex, non-linear manner, allowing for the representation of a wide range of real-world problems and scenarios.

Explain adjacency Matrix.

An adjacency matrix is a method used to represent a graph in Data Structures and Algorithms (DSA). It is a square matrix used to represent the connections between the vertices (or nodes) of a graph. The size of the matrix is determined by the number of vertices in the graph. Each cell in the matrix corresponds to an edge between two vertices.

Advantages

Simplicity: The adjacency matrix is straightforward to understand and implement. It provides a clear visual representation of the graph.

Ease of Access: It is easy to check if an edge exists between two vertices by simply looking at the corresponding cell in the matrix.

Disadvantages

Space Complexity: The space complexity of an adjacency matrix is O(V^2), where V is the number of vertices. This can be inefficient for large, sparse graphs (graphs with few edges) because most of the matrix will be filled with zeros.

Inefficiency for Sparse Graphs: As mentioned, adjacency matrices can be inefficient for sparse graphs because they allocate space for all possible edges, even if many of them do not exist.

Example: Consider a simple undirected graph with 4 vertices (A, B, C, D) and the following edges: (A, B), (A, C), (B, C), (C, D).

The adjacency matrix for this graph would be:

A B C D

A 0 1 1 0

B 1 0 1 0

C 1 1 0 1

D 0 0 1 0

In this matrix, a 1 indicates an edge between the vertices, and a 0 indicates no edge. For example, the cell at the intersection of row A and column B is 1, indicating an edge between vertices A and B.

Explain adjacency list with an example graph or given graph.

An adjacency list is another method used to represent a graph in Data Structures and Algorithms (DSA). Unlike an adjacency matrix, which uses a two-dimensional array to represent the connections between vertices, an adjacency list represents each vertex as a list of its adjacent vertices. This representation is particularly efficient for sparse graphs, where the number of edges is much less than the number of vertices squared.

Advantages

Space Efficiency: The adjacency list is more space-efficient than the adjacency matrix for sparse graphs, as it only stores the actual edges.

Flexibility: It can easily represent both directed and undirected graphs, and it can also represent weighted graphs by storing additional information about the edges.

Disadvantages

Complexity in Accessing Edges: Unlike the adjacency matrix, accessing the adjacency list requires traversing the list of a vertex to find its adjacent vertices. This can be less efficient for certain operations, such as checking if an edge exists between two vertices.

Example: Consider a simple undirected graph with 4 vertices (A, B, C, D) and the following edges: (A, B), (A, C), (B, C), (C, D).

The adjacency list for this graph would be:

A: B, C

B: A, C

C: A, B, D

D: C

In this representation, each vertex is followed by a list of its adjacent vertices. For example, vertex A is adjacent to vertices B and C, as indicated by the list A: B, C.

Explain DFS and BFS according to the Graph. Consider a as source.

Depth-First Search (DFS):

DFS explores as far as possible along each branch before backtracking. It uses a stack data structure to keep track of vertices to visit next.

Algorithm:

- Start from the source vertex (A in this case).
- Mark the source vertex as visited and push it onto the stack.
- While the stack is not empty, pop a vertex from the stack.
- For each unvisited neighbor of the popped vertex, mark it as visited and push it onto the stack.
- Repeat steps 3-4 until the stack is empty.

Example Traversal:

- Start at A.
- Visit B (A -> B).
- Visit D (B -> D).
- Backtrack to B (D -> B).
- Visit C (B -> C).
- Visit E (C -> E).
- Backtrack to C (E -> C).
- Backtrack to A (C -> A).
- Visit D (A -> D).
- Backtrack to A (D -> A).

DFS Order: A, B, D, C, E, D

Breadth-First Search (BFS)

BFS explores all the vertices at the current level before moving on to the vertices at the next level. It uses a queue data structure to keep track of vertices to visit next.

Algorithm:

- Start from the source vertex (A in this case).
- Mark the source vertex as visited and enqueue it.
- While the queue is not empty, dequeue a vertex.
- For each unvisited neighbor of the dequeued vertex, mark it as visited and enqueue it.
- Repeat steps 3-4 until the queue is empty.

Example Traversal:

- Start at A.
- Visit B (A -> B).
- Visit C (A -> C).
- Visit D (B -> D).
- Visit E (C -> E).
- Visit D (C -> D).

BFS Order: A, B, C, D, E, D

Comparison

- DFS is more suitable for scenarios where you need to go deep into the graph, such as finding a path to a specific vertex or exploring all possible paths.
- BFS is more suitable for scenarios where you need to find the shortest path between two vertices or when you need to explore all vertices at a certain distance from the source vertex.

What is Pendant Vertex?

A pendant vertex (also known as a pendant node) is a vertex that has exactly one edge. In other words, it is connected to exactly one other vertex in the graph. Pendant vertices are significant in graph theory because they can simplify the structure of the graph and are often used in graph algorithms to identify and handle specific cases.

Characteristics of Pendant Vertices

- Degree: The degree of a pendant vertex is 1. This is because it has only one edge connecting it to another vertex.
- Isolation: Pendant vertices are isolated from the rest of the graph in terms of their degree. They do not contribute to the overall connectivity of the graph in the same way that vertices with higher degrees do.
- Impact on Graph Structure: Removing a pendant vertex from a graph does not disconnect the graph. It simply removes one edge and one vertex, leaving the rest of the graph intact.

Applications in Graph Algorithms

- Graph Simplification: Pendant vertices can be removed without affecting the connectivity of the graph. This can simplify the graph for further analysis or manipulation.
- Path Finding: In some algorithms, pendant vertices might be bypassed or ignored during the traversal or search process, as they do not add to the complexity of the path.
- Graph Analysis: Pendant vertices can be identified and analyzed separately from other vertices in the graph, which can be useful in certain graph analysis tasks.

Example: Consider a graph with vertices A, B, C, D, and E, and the following edges: (A, B), (B, C), (C, D), (D, E), (E, F), (F, G), (G, H), (H, I), (I, J), (J, K), (K, L), (L, M), (M, N), (N, O), (O, P), (P, Q), (Q, R), (R, S), (S, T), (T, U), (U, V), (V, W), (W, X), (X, Y), (Y, Z).

In this graph, vertices A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, and Z are all pendant vertices because they each have exactly one edge.

Differentiate Prisms and Krushkal Agorithm. Wite down complexities of those.

Prims and Kruskal's algorithms are two different approaches to solving the Minimum Spanning Tree (MST) problem. The MST problem involves finding a tree that spans all the vertices in a graph and has the minimum possible total edge weight. Both algorithms aim to solve this problem, but they do so in different ways and have different complexities.

Prims Algorithm

Overview:

- Prims algorithm starts with an arbitrary vertex and grows the MST by adding the minimum weight edge that connects a vertex in the MST to a vertex outside the MST.
- It uses a priority queue to select the next edge to add to the MST.

Complexity:

- Time Complexity: O(E log E) or O(E log V), where E is the number of edges and V is the number of vertices. This is because each edge is processed once, and the time complexity of extracting the minimum element from a priority queue is O(log E) or O(log V).
- Space Complexity: O(V), as it needs to store the MST and the visited vertices.

Kruskal's Algorithm

Overview:

- Kruskal's algorithm starts with all the vertices in separate sets and adds the smallest edge that connects two different sets.
- It uses a disjoint set data structure to efficiently check if two vertices are in the same set and to merge sets.

Complexity:

- Time Complexity: O(E log E) or O(E log V), where E is the number of edges and V is the number of vertices. This is because each edge is processed once, and the time complexity of sorting the edges and performing union operations in a disjoint set is O(E log E) or O(E log V).
- Space Complexity: O(V), as it needs to store the MST and the disjoint set data structure.

Find Minimal cost spanning tree using prism's Algorithm.

To find the Minimal Cost Spanning Tree (MST) using Prim's algorithm in Data Structures and Algorithms (DSA), we'll follow a step-by-step approach. Prim's algorithm is a greedy algorithm that starts with an arbitrary vertex and grows the MST by adding the minimum weight edge that connects a vertex in the MST to a vertex outside the MST.

Step 1: Initialize

- Start with an arbitrary vertex.
- Create a set of vertices not yet included in the MST.
- Initialize the key values of all vertices to infinity, except for the starting vertex, which has a key value of 0.

Step 2: Grow the MST

- While there are vertices not yet included in the MST:
- Select the vertex with the minimum key value from the set of vertices not yet included in the MST.
- Add the selected vertex to the MST.
- Update the key values of its adjacent vertices. If the weight of an edge connecting the selected vertex to an adjacent vertex is less than the current key value of the adjacent vertex, update the key value of the adjacent vertex.

Step 3: Repeat Step 2

- Continue the process until all vertices are included in the MST.

Find Minimal cost spanning tree Krushkal Algorithm.

To find the Minimal Cost Spanning Tree (MST) using Kruskal's algorithm in Data Structures and Algorithms (DSA), we'll follow a step-by-step approach. Kruskal's algorithm is a greedy algorithm that starts with all vertices in separate sets and adds the smallest edge that connects two different sets.

Step 1: Initialize

- Create a set of vertices not yet included in the MST.
- Initialize a disjoint set data structure to keep track of the sets of vertices.

Step 2: Sort All Edges

- Sort all the edges in non-decreasing order of their weight.

Step 3: Grow the MST

- While there are edges not yet included in the MST:
- Select the smallest edge.
- If the edge connects two different sets, add it to the MST and merge the two sets.

Step 4: Repeat Step 3

- Continue the process until all vertices are included in the MST.

SORTING

Bubble Sort

Bubble Sort is an algorithm that sorts an array from the lowest value to the highest value.

The word 'Bubble' comes from how this algorithm works, it makes the highest values 'bubble up'.

Before we implement the Bubble Sort algorithm in a programming language, let's manually run through a short array only one time, just to get the idea.

Step 1: We start with an unsorted array: [7, 12, 9, 11, 3]

Step 2: We look at the two first values. Does the lowest value come first? Yes, so we don't need to swap them: [7, 12, 9, 11, 3]

Step 3: Take one step forward and look at values 12 and 9. Does the lowest value come first? No: [7, 12, 9, 11, 3]

Step 4: So, we need to swap them so that 9 comes first: [7, 9, 12, 11, 3]

Step 5: Taking one step forward, looking at 12 and 11: [7, 9, 12, 11, 3]

Step 6: We must swap so that 11 comes before 12: [7, 9, 11, 12, 3]

Step 7: Looking at 12 and 3, do we need to swap them? Yes:

[7, 9, 11, 12, 3]

Step 8: Swapping 12 and 3 so that 3 comes first: [7, 9, 11, 3, 12]

Selection Sort

The Selection Sort algorithm finds the lowest value in an array and moves it to the front of the array.

The algorithm looks through the array again and again, moving the next lowest values to the front, until the array is sorted.

Before we implement the Selection Sort algorithm in a programming language, let's manually run through a short array only one time, just to get the idea.

Step 1: We start with an unsorted array: [7, 12, 9, 11, 3]

Step 2: Go through the array, one value at a time. Which value is the lowest? 3, right?

[7, 12, 9, 11, 3]

Step 3: Move the lowest value 3 to the front of the array: [3, 7, 12, 9, 11]

Step 4: Look through the rest of the values, starting with 7. 7 is the lowest value, and already at the front of the array, so we don't need to move it: [3, 7, 12, 9, 11]

Step 5: Look through the rest of the array: 12, 9 and 11. 9 is the lowest value: [3, 7, 12, 9, 11]

Step 6: Move 9 to the front: [3, 7, 9, 12, 11]

Step 7: Looking at 12 and 11, 11 is the lowest: [3, 7, 9, 12, 11]

Step 8: Move it to the front: [3, 7, 9, 11, 12]

Finally, the array is sorted.

Insertion Sort

The Insertion Sort algorithm uses one part of the array to hold the sorted values, and the other part of the array to hold values that are not sorted yet.

The algorithm takes one value at a time from the unsorted part of the array and puts it into the right place in the sorted part of the array, until the array is sorted.

Before we implement the Insertion Sort algorithm in a programming language, let's manually run through a short array, just to get the idea.

Step 1: We start with an unsorted array: [7, 12, 9, 11, 3]

Step 2: We can consider the first value as the initial sorted part of the array. If it is just one value, it must be sorted, right?

[7, 12, 9, 11, 3]

Step 3: The next value 12 should now be moved into the correct position in the sorted part of the array. But 12 is higher than 7, so it is already in the correct position: [7, 12, 9, 11, 3]

Step 4: Consider the next value 9: [7, 12, 9, 11, 3]

Step 5: The value 9 must now be moved into the correct position inside the sorted part of the array, so we move 9 in between 7 and 12: [7, 9, 12, 11, 3]

Step 6: The next value is 11: [7, 9, 12, 11, 3]

Step 7: We move it in between 9 and 12 in the sorted part of the array: [7, 9, 11, 12, 3]

Step 8: The last value to insert into the correct position is 3: [7, 9, 11, 12, 3]

Step 9: We insert 3 in front of all other values because it is the lowest value: [3,7, 9, 11, 12]

Finally, the array is sorted.

Quick Sort

As the name suggests, Quicksort is one of the fastest sorting algorithms.

The Quicksort algorithm takes an array of values, chooses one of the values as the 'pivot' element, and moves the other values so that lower values are on the left of the pivot element, and higher values are on the right of it.

Before we implement the Quicksort algorithm in a programming language, let's manually run through a short array, just to get the idea.

Step 1: We start with an unsorted array: [11, 9, 12, 7, 3]

Step 2: We choose the last value 3 as the pivot element: [11, 9, 12, 7, 3]

Step 3: The rest of the values in the array are all lower than 3, an must be on the right side of 3. Swap 3 with 11: [3, 9, 12, 7, 11]

Step 4: Value 3 is now in the correct position. We need to sort the values to the right of 3. We choose the last value 11 as the new pivot element: [3, 9, 12, 7, 11]

Step 5: The value 7 must be to the left of pivot value 11, and 12 must be to the right of it. Move 7 and 12: [3, 9, 7, 12, 11]

Step 6: Swap 11 with 12 so that lower values 9 anf 7 are on the left side of 11, and 12 is on the right side: [3, 9, 7, 11, 12]

Step 7: 11 and 12 are in the correct positions. We choose 7 as the pivot element in sub-array [9, 7], to the left of 11: [3, 9, 7, 11, 12]

Step 8: We must swap 9 with 7: [3, 7, 9, 11, 12]

And now, the array is sorted.

Merge Sort

The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.

Divide: The algorithm starts with breaking up the array into smaller and smaller pieces until one such sub-array only consists of one element.

Conquer: The algorithm merges the small pieces of the array back together by putting the lowest values first, resulting in a sorted array.

Let's try to do the sorting manually, just to get an even better understanding of how Merge Sort works before actually implementing it in a programming language.

Step 1: We start with an unsorted array, and we know that it splits in half until the sub-arrays only consist of one element. The Merge Sort function calls itself two times, once for each half of the array. That means that the first sub-array will split into the smallest pieces first.

[12, 8, 9, 3, 11, 5, 4]

[12, 8, 9] [3, 11, 5, 4]

[12] [8, 9] [3, 11, 5, 4]

[12] [8] [9] [3, 11, 5, 4]

Step 2: The splitting of the first sub-array is finished, and now it is time to merge. 8 and 9 are the first two elements to be merged. 8 is the lowest value, so that comes before 9 in the first merged sub-array: [12] [8, 9] [3, 11, 5, 4]

Step 3: The next sub-arrays to be merged is [12] and [8, 9]. Values in both arrays are compared from the start. 8 is lower than 12, so 8 comes first, and 9 is also lower than 12: [8, 9, 12] [3, 11, 5, 4]

Step 4: Now the second big sub-array is split recursively: [8, 9, 12] [3, 11, 5, 4]

[8, 9, 12] [3, 11] [5, 4]

[8, 9, 12] [3] [11] [5, 4]

Step 5: 3 and 11 are merged back together in the same order as they are shown because 3 is lower than 11: [8, 9, 12] [3, 11] [5, 4]

Step 6: Sub-array with values 5 and 4 is split, then merged so that 4 comes before 5:

[8, 9, 12] [3, 11] [5] [4]

[8, 9, 12] [3, 11] [4, 5]

Step 7: The two sub-arrays on the right are merged. Comparisons are done to create elements in the new merged array:

3 is lower than 4

4 is lower than 11

5 is lower than 11

11 is the last remaining value

[8, 9, 12] [3, 4, 5, 11]

Step 8: The two last remaining sub-arrays are merged. Let's look at how the comparisons are done in more detail to create the new merged and finished sorted array:

3 is lower than 8: Before [8, 9, 12] [3, 4, 5, 11]

After: [3, 8, 9, 12] [4, 5, 11]

Step 9: 4 is lower than 8: Before [3, 8, 9, 12] [4, 5, 11]

After: [3, 4, 8, 9, 12] [5, 11]

Step 10: 5 is lower than 8: Before [3, 4, 8, 9, 12] [5, 11]

After: [3, 4, 5, 8, 9, 12] [11]

Step 11: 8 and 9 are lower than 11: Before [3, 4, 5, 8, 9, 12] [11]

After: [3, 4, 5, 8, 9, 12] [11]

Step 12: 11 is lower than 12: Before [3, 4, 5, 8, 9, 12] [11]

After: [3, 4, 5, 8, 9, 11, 12]

The sorting is finished!

Case Study of best, worst and average time complexities of (i) merge sort (ii) Quick sort (iii) insertion sort (iv) selection sort (v) Bubble sort

(i) Merge Sort

Time complexity can be calculated based on the number of split operations and the number of merge operations:

$O((n-1)+n \cdot \log_2 n)=O(n \cdot \log_2 n)$

The number of splitting operations $(n-1)$ can be removed from the Big O calculation above because $n \cdot \log_2 n$ will dominate for large n, and because of how we calculate time complexity for algorithms.

The figure below shows how the time increases when running Merge Sort on an array with n values.

- **Best Case:** O(n log n)
    - In the best case, Merge Sort always divides the array into two halves and merges them recursively. The time complexity remains the same as the worst-case due to its consistent nature.
- **Worst Case:** O(n log n)
    - Merge Sort divides the array into halves until each sub-array contains a single element. Then, it merges these sub-arrays back together in sorted order. The worst-case time complexity occurs when the array is divided log n times and each division takes linear time to merge.
- **Average Case:** O(n log n)
    - Merge Sort's average-case time complexity is the same as the worst-case scenario. This is because it always divides the array into halves and merges them back together, regardless of the distribution of elements.

(ii) Quick Sort

- **Best Case:** O(n log n)
    - The best-case scenario for Quick Sort occurs when the partition process divides the array into roughly equal halves. This results in balanced recursive calls and a time complexity of O(n log n).
- **Worst Case:** O(n^2)
    - The worst-case time complexity of Quick Sort happens when the pivot chosen is always the smallest or largest element in the current array partition. This leads to unbalanced partitions and results in a time complexity of O(n^2), although this can be mitigated with randomized pivot selection.

      Worst case time complexity is: $O(n \cdot (n/2)) = O(n^2)$
- **Average Case:** O(n log n)
    - On average, Quick Sort performs efficiently at O(n log n) time complexity, making it one of the fastest sorting algorithms for average cases. Randomized pivot selection can further improve its performance.

(iii) Insertion Sort

- **Best Case:** O(n)
    - Insertion Sort performs optimally when the array is already sorted or nearly sorted. In this scenario, each element requires constant time to insert into the correct position, resulting in a best-case time complexity of O(n).
- **Worst Case:** O(n^2)
    - The worst-case time complexity of Insertion Sort occurs when the array is in reverse order. Each element must be compared and moved to the beginning of the array, resulting in a time complexity of O(n^2).
- **Average Case:** O(n^2)
    - On average, Insertion Sort has a time complexity of O(n^2). While it can be efficient for small datasets or nearly sorted arrays, its performance degrades for larger or more disorderly datasets.

(iv) Selection Sort

- **Best Case:** O(n^2)
    - The best-case time complexity for Selection Sort is O(n^2), as it requires scanning the entire array to find the minimum element in each iteration, regardless of the input distribution.
- **Worst Case:** O(n^2)
    - The worst-case time complexity also remains O(n^2), occurring when the array is in reverse order or nearly sorted. Selection Sort repeatedly finds the minimum element and swaps it with the current position, resulting in quadratic time complexity.
- **Average Case:** O(n^2)
    - Selection Sort's average-case time complexity is O(n^2), as it involves nested loops for finding the minimum element and swapping it with the current position.

(v) Bubble Sort

- **Best Case:** O(n)
    - The best-case time complexity of Bubble Sort occurs when the array is already sorted. In this case, Bubble Sort makes a single pass through the array without any swaps, resulting in a time complexity of O(n).
- **Worst Case:** O(n^2)
    - The worst-case time complexity of Bubble Sort happens when the array is in reverse order. In each pass, Bubble Sort compares and potentially swaps adjacent elements throughout the entire array, resulting in a time complexity of O(n^2).
- **Average Case:** O(n^2)
    - On average, Bubble Sort's time complexity is O(n^2). While it can be efficient for small datasets or nearly sorted arrays, its performance degrades for larger or more disorderly datasets.

Write the Algorithm of merge sort (i) merge sort (ii) Quick sort (iii) insertion sort (iv) selection sort (v) Bubble sort

Algorithm: Merge Sort

1. **Procedure mergeSort(arr):**
    - Input: An array **arr** of elements to be sorted.
    - Output: The array **arr** sorted in non-decreasing order.
2. **Base Case:** If the length of **arr** is less than or equal to 1, return **arr** (already sorted).
3. **Divide Step:** Split the array **arr** into two halves:
    - **left** = first half of **arr**
    - **right** = second half of **arr**
4. **Recursive Calls:** Recursively apply **mergeSort** to the **left** and **right** halves:
    - **left** = **mergeSort(left)**
    - **right** = **mergeSort(right)**
5. **Merge Step:** Merge the two sorted halves (**left** and **right**) into a single sorted array:
    - Initialize empty list **result**
    - Initialize pointers **i** (for **left**) and **j** (for **right**) to 0
    - While **i** < length of **left** and **j** < length of **right**:
        - If **left[i]** ≤ **right[j]**, append **left[i]** to **result** and increment **i**
        - Otherwise, append **right[j]** to **result** and increment **j**
    - Append remaining elements of **left** (if any) to **result**
    - Append remaining elements of **right** (if any) to **result**
6. **Return Result:** Return the merged **result** as the sorted array.

Algorithm: Quick Sort

1. **Procedure quickSort(arr, low, high):**
    - Input: An array **arr** of elements to be sorted, and indices **low** and **high** indicating the range of elements to sort.
    - Output: The array **arr** sorted in non-decreasing order.
2. **Base Case:**
    - If **low** ≥ **high**, return (base case for recursion).
3. **Partitioning Step:**
    - Choose a pivot element from the array (e.g., **arr[high]**).
    - Rearrange the array elements such that all elements less than the pivot are moved to its left and all elements greater than the pivot are moved to its right.
        - Use two pointers **i** and **j**:
            - Initialize **i** to **low** - 1 (points to the end of the smaller elements).
            - Iterate **j** from **low** to **high - 1**:
                - If **arr[j]** ≤ pivot, increment **i** and swap **arr[i]** with **arr[j]**.
        - Finally, swap **arr[i + 1]** with **arr[high]** (placing the pivot in its correct position).
4. **Recursive Calls:**
    - Recursively apply **quickSort** to the subarrays **[low...i]** and **[i+2...high]**:
        - **quickSort(arr, low, i)**
        - **quickSort(arr, i + 2, high)**

Algorithm: Insertion Sort

1. **Procedure insertionSort(arr):**
    - Input: An array **arr** of elements to be sorted.
    - Output: The array **arr** sorted in non-decreasing order.
2. **Iterative Insertion:**
    - Iterate over each element **arr[i]** in the array, starting from the second element (**i = 1**):
        - **i** represents the index of the element to be inserted into the sorted subarray **[0...i-1]**.
3. **Insertion Step:**
    - For each element **arr[i]**, compare it with the elements in the sorted subarray **[0...i-1]** from right to left (**j = i-1** to **0**):
        - If **arr[j]** is greater than **arr[i]**, shift **arr[j]** one position to the right (**arr[j+1] = arr[j]**).
        - Continue this process until you find the correct position to insert **arr[i]**.
4. **Place Element:**
    - Insert **arr[i]** into its correct position in the sorted subarray **[0...i-1]**:
        - Set **arr[j+1] = arr[i]**.

Algorithm: Selection Sort

1. **Procedure selectionSort(arr):**
    - Input: An array **arr** of elements to be sorted.
    - Output: The array **arr** sorted in non-decreasing order.
2. **Iterative Selection:**
    - Iterate over each element **arr[i]** in the array, starting from the first element (**i = 0**) up to the second-to-last element (**i = n-1**, where **n** is the length of the array).
3. **Find Minimum Element:**
    - For each iteration, find the index **min_idx** of the minimum element in the unsorted subarray **[i...n-1]**:
        - Initialize **min_idx** to **i**.
        - Traverse the subarray **[i+1...n-1]** to find the index of the smallest element:
            - If **arr[j] < arr[min_idx]**, update **min_idx** to **j**.
4. **Swap Elements:**
    - Swap the element at index **min_idx** with the element at index **i** to place the minimum element in its correct position in the sorted array.

Algorithm: Bubble Sort

1. **Procedure bubbleSort(arr):**
    - Input: An array **arr** of elements to be sorted.
    - Output: The array **arr** sorted in non-decreasing order.
2. **Iterative Passes:**
    - Repeat the following steps for **n-1** passes, where **n** is the length of the array:
        - In each pass (**pass_num** from **1** to **n-1**):
            - Iterate through the array from index **0** to **n-1-pass_num**:
                - Compare each pair of adjacent elements (**arr[j]** and **arr[j+1]**).
                - If **arr[j]** is greater than **arr[j+1]**, swap them.
3. **Optimization:**
    - During each pass, the largest element in the unsorted portion of the array "bubbles up" to its correct position at the end of the array.
    - After each pass, the last **pass_num** elements become sorted and do not need to be checked again.

HASHING

Define Hashing. What do you mean by hash table and hash function. Explain 3 h ash functions used with example.

Hashing: Hashing is a technique used in computer science to convert a range of key values into a range of indexes of an array. It is a way of mapping data of arbitrary size to fixed-size values. Hashing is used in various applications such as database indexing, caching, and password storage. The primary goal of hashing is to distribute data evenly across an array to minimize collisions (where two different keys produce the same hash value).

Hash Table: A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Ideally, the hash function will assign each key to a unique bucket, but in practice, collisions can occur.

Hash Function: A hash function is a function that takes an input (or 'message') and returns a fixed-size string of bytes. The output is typically a 'digest' that is unique to each unique input. It is used in hash tables to compute the index for storing or retrieving data.

Examples of Hash Functions

1. Division Method: One simple hash function is the division method, where the key is divided by a prime number and the remainder is used as the index. For example, if we have an array of size 10 and a key k, the hash function could be hash(k) = k mod 10.

Example:

- Key: 123
- Array size: 10
- Hash function: hash(k) = k mod 10
- Index: 123 mod 10 = 3

2. Multiplication Method: Another method is the multiplication method, where the key is multiplied by a fractional number between 0 and 1, and the fractional part is used as the index.

Example:

- Key: 123
- Array size: 10
- Hash function: hash(k) = (k * 0.1) mod 1
- Index: (123 * 0.1) mod 1 = 0.123
- Since we need an integer index, we could round this to the nearest integer, but in practice, we would use the fractional part directly as the index.

3. Mid-Square Method: The mid-square method involves squaring the key, taking the middle part of the result, and using that as the index. This method is particularly useful for keys that are integers.

Example:

- Key: 123
- Array size: 10
- Hash function: hash(k) = (k^2 / 100) mod 10
- Index: (123^2 / 100) mod 10 = (15129 / 100) mod 10 = 151 mod 10 = 15

What is collision?

What are the types of collision resolution techniques in open addressing? explai n with example.

What is the chain method of collision resolution?

Collision

A collision in the context of hashing occurs when two different keys produce the same hash value. This is a common issue in hash tables because the number of possible keys is typically much larger than the number of slots available in the hash table. Collisions can lead to inefficiencies in data retrieval and storage.

Types of Collision Resolution Techniques in Open Addressing

1. Linear Probing: In linear probing, when a collision occurs, the hash table looks for the next available slot in a linear sequence. For example, if the hash function computes the index i for a key, and i is already occupied, the table looks at index i+1, then i+2, and so on, until it finds an empty slot.

Example:

- Hash table size: 10
- Key k hashes to index i
- If i is occupied, look at i+1, then i+2, and so on.

2. Quadratic Probing: Quadratic probing is similar to linear probing but uses a quadratic function to find the next slot. This can help distribute the keys more evenly in the table.

Example:

- Hash table size: 10
- Key k hashes to index i
- If i is occupied, look at i+1^2, then i+2^2, and so on.

3. Double Hashing: Double hashing uses a second hash function to determine the step size for probing. This can help avoid clustering and improve the distribution of keys.

Example:

- Hash table size: 10
- Key k hashes to index i
- If i is occupied, use a second hash function h2(k) to determine the step size.
- Look at i+h2(k), then i+2*h2(k), and so on.

Chain Method of Collision Resolution: The chain method, also known as separate chaining, is a collision resolution technique where each slot in the hash table points to a linked list of elements that hash to the same slot. When a collision occurs, the new element is simply added to the end of the list. This method is particularly effective for handling collisions because it allows for dynamic resizing of the hash table and does not require rehashing of elements when the table size changes.

Example:

- Hash table size: 10
- Key k hashes to index i
- If i is occupied, add the new element to the linked list at index i.

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

1. Calculate the hash value for each key: We use the hash function h(k) = k mod 10.
2. Insert the key into the hash table: If the calculated index is empty, insert the key there. If the index is occupied, use linear probing to find the next available slot.

Let's go through the process for each key:

- Key 12: h(12) = 12 mod 10 = 2. Index 2 is empty, so 12 is inserted at index 2.
- Key 18: h(18) = 18 mod 10 = 8. Index 8 is empty, so 18 is inserted at index 8.
- Key 13: h(13) = 13 mod 10 = 3. Index 3 is empty, so 13 is inserted at index 3.
- Key 2: h(2) = 2 mod 10 = 2. Index 2 is occupied, so we use linear probing. The next available slot is index 3, so 2 is inserted at index 3.
- Key 3: h(3) = 3 mod 10 = 3. Index 3 is occupied, so we use linear probing. The next available slot is index 4, so 3 is inserted at index 4.
- Key 23: h(23) = 23 mod 10 = 3. Index 3 is occupied, so we use linear probing. The next available slot is index 5, so 23 is inserted at index 5.
- Key 5: h(5) = 5 mod 10 = 5. Index 5 is empty, so 5 is inserted at index 5.
- Key 15: h(15) = 15 mod 10 = 5. Index 5 is occupied, so we use linear probing. The next available slot is index 6, so 15 is inserted at index 6.

Resultant Hash Table

After inserting all keys, the hash table looks like this:

Index: 0 1 2 3 4 5 6 7 8 9

Value: - 18 12 2 3 23 5 15 - -

- The dashes (-) represent empty slots.
- The numbers represent the keys that have been inserted into the hash table.
- The keys are distributed across the hash table based on the hash function and the linear probing technique used to resolve collisions.

Write a C program to add two polynomials (use array or linked list).

```
#include <stdio.h>

void addPolynomials(int poly1[], int poly2[], int result[], int degree) {
	for (int i = 0; i <= degree; i++) {
		result[i] = poly1[i] + poly2[i];
	}
}

int main() {
	int degree = 2; // Degree of the polynomials
	int poly1[3] = {1, 2, 3}; // Coefficients of the first polynomial: 1x^2 + 2x + 3
	int poly2[3] = {4, 5, 6}; // Coefficients of the second polynomial: 4x^2 + 5x + 6
	int result[3] = {0}; // Array to store the result
	addPolynomials(poly1, poly2, result, degree);
	printf("The resultant polynomial is: ");
	for (int i = degree; i >= 0; i--) {
		if (i != degree || result[i] != 0) {
			printf("%dx^%d", result[i], i);
			if (i > 0)
				printf(" + ");
		}
	}
	printf("\n");
	return 0;
}
```

Write a C program to print the 100th fibonacci number.

```
#include <stdio.h>

int main() {

	long long fib[101]; // Array to store Fibonacci numbers

	fib[0] = 0; // F(0) = 0

	fib[1] = 1; // F(1) = 1

	// Calculate Fibonacci numbers up to the 100th

	for (int i = 2; i <= 100; i++) {

		fib[i] = fib[i-1] + fib[i-2];

	}

	// Print the 100th Fibonacci number

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

	return 0;

}
```