Contents

fb.com/prodhan24 github.com/prodhan2

Data Structure -2021

Section A

1.(a) What is the benefit of binary search over linear search?

The binary search algorithm uses the divide-and-conquer approach, it does not scan every element in the list, it only searches half of the list instead(পরিবর্তে) of going through each element, Hence said to be the best searching algorithm because it is faster to execute as compared to Linear search.

The linear search algorithm, also known as sequential search, is a straightforward searching algorithm that scans each element in a list or array until the target value is found. The main advantage of the linear search algorithm is its simplicity and ease of implementation.

(b) Maze (1:6, -4:1, 5:10) is a 3D array with base-100, w=4, calculate Maze (3, -2, 8] address in a row major order and column major order.

(c) We often store sparse matrix in a ID array to save spaces. What is the memory saving if we store a sparse matrix in a ID array rather than a 2D array?

2. (a) What is a linked list? Explain the main differences between the linked list and linear array?

(b) Briefly discuss the terms garbage collection, overflow and underflow.

Certainly, here's a brief overview of the terms you mentioned:

1. **Garbage Collection:** Garbage collection is a process in computer programming where the system automatically identifies and frees up memory that is no longer being used by a program. This helps prevent memory leaks, where memory is allocated but not properly deallocated,

leading to inefficient use of system resources over time. Garbage collection ensures that memory is properly managed and reclaimed, improving the efficiency and stability of programs.

2. **Overflow:** Overflow occurs when a computation or operation results in a value that exceeds the maximum representable value within a given data type. For example, if you're using a 8-bit integer data type, the maximum value that can be represented is 255. If you perform an operation that results in a value larger than 255, an overflow occurs, and the result "wraps around" to a value within the valid range. Overflow can lead to unexpected behavior and incorrect results in programs.
3. **Underflow:** Underflow is the opposite of overflow. It happens when a computation or operation results in a value that is smaller than the minimum representable value within a given data type. Similar to overflow, underflow can lead to unexpected behavior and incorrect results. Modern programming languages and systems often handle overflow and underflow conditions by providing mechanisms to detect and handle these situations, such as raising exceptions or using special values to indicate these conditions.

(c) One of the advantages of linked list is the ability to insert data into the list easily. Explain with your own words and figures how to insert data at the beginning, after a given node, at the end and to a sorted list.

3.

(**a) Define stack. Explain the usage of stack in recursive algorithm implementation.**

(b) Simulate the infix to postfix transformation algorithm for Q: A+(BC-(D/ETF) G) H by showing the stack's contents as each element is scanned.

(c) Write down a routine to insert an element onto a queue.

4.

(a) What is a hash function?

A hash function usually means a function that compresses, meaning the output is shorter than the input

► A hash function takes a group of characters (called a key) and maps it to a value of a certain length (called a hash value or hash).

►The hash value is representative of the original string of characters, but is normally smaller than the original.

►This term is also known as a hashing algorithm or message digest function.

‣ Hash functions also called message digests or one-way encryption or hashing algorithm.

(

Section B

1. **(a) Define the terms (i) siblings, (ii) ancestor, and (iii) depth of a binary tree.**

(i) Siblings: if two nodes are the children of the same parent node, they are siblings to each other.

(ii) Ancestor: an ancestor of a node is any node that is higher up in the tree hierarchy and is traversed when moving from the root to the given node.

(iii) Depth of a Binary Tree: The depth of a node in a binary tree is the length of the path from the root node to that specific node. The depth of the root node is typically defined as 0. The depth of a binary tree is the maximum depth of any node in the tree. It represents the length of the longest path from the root node to any leaf node in the tree.

(b) Tree traversal (also known as walking the tree) refers to the process of visiting each node exactly once. Simulate the preorder traversal algorithm for the following tree.

6. (a) What is adjacency matrix? How is it formed? (b) Define the terms (i) Isolated node, (ii) Simple Path and (iii) Weighted Graph.

(a) Adjacency Matrix: An adjacency matrix is a square matrix used to represent a graph's connections or relationships between its vertices. In an undirected graph, the adjacency matrix is symmetric, while in a directed graph, it might not be symmetric. Each row and column of the matrix corresponds to a vertex, and the entries indicate whether there is an edge between the corresponding vertices.

In an unweighted graph, the entries are typically binary: 0 indicates no edge, and 1 indicates the presence of an edge between the vertices. In a weighted graph, the entries contain the weight or cost of the edge between the vertices. Adjacency matrices are useful for quickly determining whether two vertices are connected and for various graph algorithms that require matrix operations.

An adjacency matrix for an undirected graph with 4 vertices might look like this:

0 1 2 3 0 [0 1 1 0] 1 [1 0 1 1] 2 [1 1 0 0] 3 [0 1 0 0]

In this matrix, a 1 in the (i, j) position indicates an edge between vertices i and j.

(b) Definitions:

(i) **Isolated Node:** An isolated node, also known as an isolated vertex, is a vertex in a graph that has no edges connecting it to any other vertex in the graph. In other words, it is not part of any

edge. Isolated nodes have a degree of 0 since the degree of a vertex represents the number of edges connected to it.

(ii) **Simple Path:** A simple path in a graph is a sequence of vertices in which no vertex is repeated, and no edge is repeated. In other words, a simple path is a non-repetitive sequence of vertices connected by edges. It starts from a vertex and ends at another vertex while traversing through different vertices and edges. Simple paths are important for understanding connectivity and paths between vertices in a graph.

(iii) **Weighted Graph:** A weighted graph is a type of graph in which each edge is assigned a numerical value or weight. These weights can represent various properties such as distances, costs, capacities, etc., depending on the context of the graph. Weighted graphs are used to model situations where the relationships between vertices have associated values that are relevant to the problem being analyzed. Weighted graphs are used in various applications, including network optimization, routing, and decision-making problems.

Understanding these graph theory concepts is essential for working with graphs and solving graph-related problems in various field

(c) How can you search '5' in the following binary search tree? However if '5' is not in the tree just insert to its appropriate place and show the resultant tree.

7.(a) What is meant strongly and weakly connected in a graph?

Strongly Connected. A directed graph is strongly connected if there is a path from a to b and from b to a whenever a and b are vertices in the graph.

Weakly Connected. A directed graph is weakly connected if there is a path between every two vertices in the underlying undirected graph.

(b) Prove that the maximum number of edges that a graph with n vertices is n*(n-1)/2.

To prove that the maximum number of edges in a graph with n vertices is n*(n-1)/2, we can use the concept of a complete graph. A complete graph is a graph in which there is an edge between every pair of distinct vertices. Let's use this concept to prove the given statement.

Consider a graph with n vertices. For each vertex, there are (n - 1) possible edges that can be connected to other vertices (since we can't connect a vertex to itself). Therefore, the total number of edges that can be formed in the graph is:

Total edges = n * (n - 1)

However, since each edge is counted twice (once for each of its endpoints), we need to divide this total by 2 to avoid counting each edge twice:

Maximum number of edges = n * (n - 1) / 2

This formula gives us the maximum number of edges that can be present in a graph with n vertices.

To further illustrate this, let's consider a few values of n and calculate the maximum number of edges:

- For n = 2, the formula gives 1 edge, which is correct (a single edge between two vertices).
- For n = 3, the formula gives 3 edges, which is correct (a triangle with 3 vertices and 3 edges).
- For n = 4, the formula gives 6 edges, which is correct (a complete graph with 4 vertices and 6 edges).

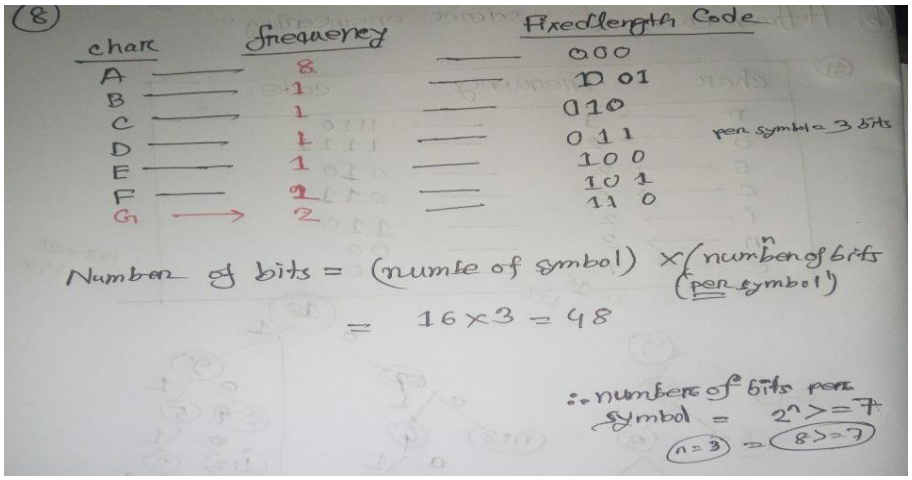
This pattern continues, and the formula holds true for all positive integers n.

Therefore, we have successfully proved that the maximum number of edges in a graph with n vertices is n * (n - 1) / 2.

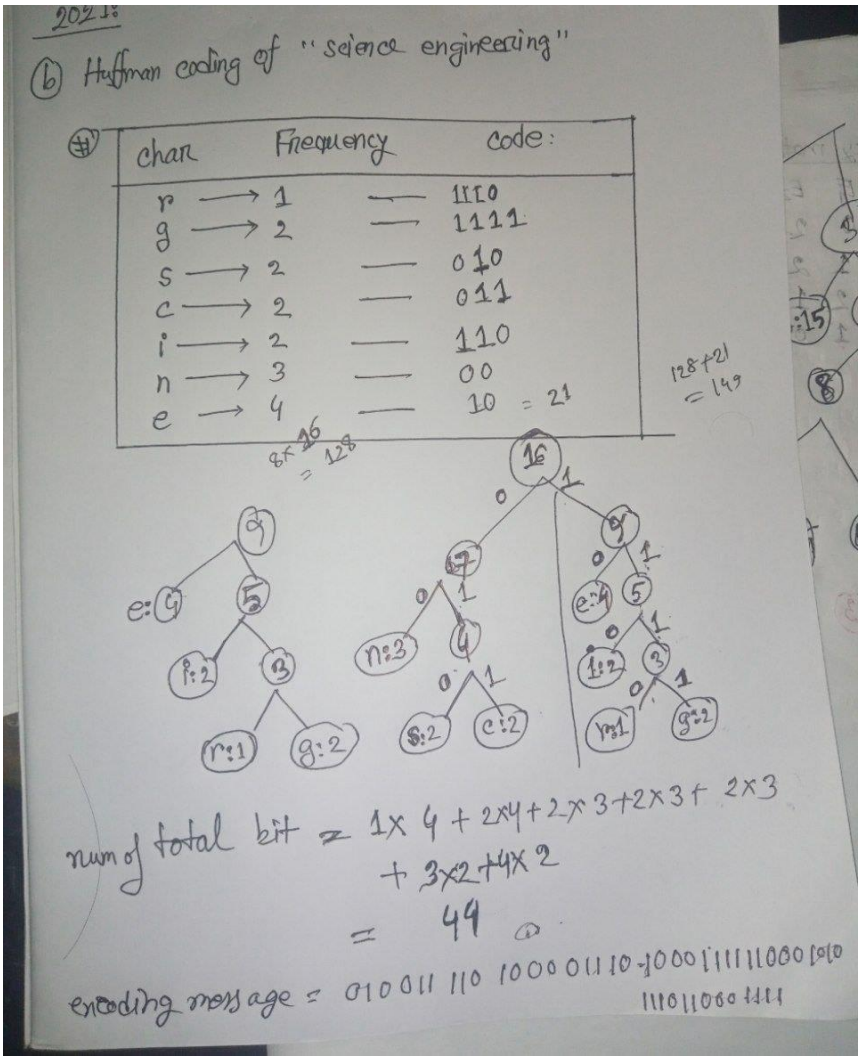
(c) Explain Breadth First search algorithm with example.

8.

(a) Suppose we want to encode a message constructed from the symbols A, B, C, D, E, F and G using a fixed-length code. How many bits are required to encode the message FDEGAACAAGAAFABA?



(b) Build the Huffman coding tree for the message 'science engineering'.



(c) Suppose you are to insert a node 'X' as the right child of node 'P'. Discuss the inserting mechanisms with figures.

i. When the right subtree of 'P' is empty.

ii. When the right subtree of 'P' is not empty.

i. When the right subtree of 'P' is empty:

In this case, we can directly assign 'X' as the right child of 'P'. The right pointer of 'P' will now point to 'X'.

Before insertion:

```
    P
   / \
```

```
  L   R
```

After insertion:

```
    P
   / \
  L   R
   /
   X
```

ii. When the right subtree of 'P' is not empty:

In this case, we need to find the rightmost node in the right subtree of 'P' and insert 'X' as its right child. This ensures that the tree remains balanced and follows the binary search tree property.

Before insertion:

```
    P
   / \
  L   R
   /
   Q
```

After insertion:

```
    P
   / \
  L   R
   / \
   Q   X
```

Here, we traverse the right subtree of 'P' until we reach the rightmost node 'Q'. Then, we make 'X' the right child of 'Q'. If there are further nodes in the right subtree of 'Q', they will remain unaffected.

Data Structure -2020

1.

a) How can you differ linear data structures from non-linear data structures?

Linear Data Structure	Non-linear Data Structure
In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

b) Simulate the binary search algorithm on the following data: 10 30 50 78 90 95 100 105 110 (suppose we search for item 110).

1.

Start with the sorted list: 10 30 50 78 90 95 100 105 110.

2. Set the lower bound (start index) to 0 and the upper bound (end index) to the length of the list minus 1.
    - Lower bound: 0
    - Upper bound: 8
3. Calculate the middle index as the average of the lower and upper bounds (integer division).
    - Middle index: 4
4. Compare the middle element (90) with the target element (110).
    - Since 90 < 110, the target element must be in the upper half of the list.
    - Update the lower bound to the middle index plus 1.
    - Lower bound: 5
5. Recalculate the middle index based on the updated lower and upper bounds.
    - Middle index: 6
6. Compare the middle element (100) with the target element (110).
    - Since 100 < 110, the target element must still be in the upper half of the list.
    - Update the lower bound to the middle index plus 1.
    - Lower bound: 7
7. Recalculate the middle index based on the updated lower and upper bounds.
    - Middle index: 7
8. Compare the middle element (105) with the target element (110).
    - Since 105 < 110, the target element must still be in the upper half of the list.
    - Update the lower bound to the middle index plus 1.
    - Lower bound: 8
9. Recalculate the middle index based on the updated lower and upper bounds.

- Middle index: 8

10. Compare the middle element (110) with the target element (110).
    - The middle element is equal to the target element.
    - The target element (110) has been found at index 8.

The binary search algorithm successfully found the target element 110 in the list at index 8.

c) How are the elements of a 2D array stored in the memory?

There are two main techniques for storing two-dimensional array elements in memory: Row major order. Column major order.

A 2D array is stored in the memory as a contiguous block of memory, where each row of the array is stored one after another. The elements of a 2D array are stored in row-major order, which means that the elements of each row are stored together, one after another, and the rows themselves are stored one after another in memory.

For example, let's say we have a 2D array A with dimensions 3x4, where A[0][0] = 1, A[0][1] = 2, A[0][2] = 3, A[0][3] = 4, A[1][0] = 5, A[1][1] = 6, and so on. In memory, the elements of the array would be stored as follows:

1 2 3 4 5 6 7 8 9 10 11 12

The first row of the array (1, 2, 3, 4) is stored first, followed by the second row (5, 6, 7, 8), and finally the third row (9, 10, 11, 12).

Note that in some programming languages, such as C and C++, 2D arrays are implemented as arrays of arrays, where each row is itself an array. In this case, the rows of the 2D array are stored as separate arrays in memory, but the order of storage is still row-major.

2.

a) Score is a 20x5 matrix with base address 200 and w=4. Find out the address of the element Score[10,2] for both row major order and column major order.

• If the elements of an Array A with m rows and n columns are stored

in row major order, the location of any elements A[i, j] can be found from the formula.

- **loc(A[i, j])=Base(A)+w[n(i-1)+(j-1)]**

  = 200+4[5x(10-1)+(2-1)] = 384

Similarly for column major order the location is

- **loc(A[i, j])=Base(A)+w[m(j-1)+(i-1)]**

  = 200+4[(10-1)+20x(2-1)] = 316

b) What is sparse matrix?

What is the difference between triangular matrix and tridiagonal matrix?

Sparse Matrices • Matrices with a high proportion of zero entries are called Sparse Matrices. • It is two types: •

(Lower) Triangular Matrix: All elements above the main diagonal are zero or, equivalently, nonzero entries can only occur on or bellow the main diagonal.

• **Tridiagonal Matrix:** Non zero entries can only occur on the diagonal or on elements immediately above or below the diagonal

Property	Triangular Matrix	Tridiagonal Matrix
Non-zero elements	Only above or below the main diagonal	On the main diagonal and immediately above or below it
Shape	Rectangular or square	Square
Sparsity	Can be sparse or dense	Always sparse

Applications	Linear algebra, numerical analysis, differential equations	Numerical analysis, differential equations, optimization

Here are some additional details about each type of matrix:

- A triangular matrix is a matrix in which all of the elements above or below the main diagonal are zero. Triangular matrices can be either upper triangular or lower triangular, depending on whether the non-zero elements are above or below the main diagonal.
- A tridiagonal matrix is a special type of triangular matrix in which the non-zero elements are only on the main diagonal and immediately above or below it. Tridiagonal matrices are a type of sparse matrix, which means that they have a relatively small number of non-zero elements.

Tridiagonal matrices are often used in numerical analysis and differential equations because they can be solved more efficiently than other t

c) How can you locate element aij of a sparce matrix from a 1D array?

3.

a) Mention some differences between arrays and linked lists. (CONTIGOUS =PASAPASI

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

No, the binary search algorithm cannot be directly applied to data stored in a linked list. The binary search algorithm relies on random access to elements, which is not possible in a linked list. Binary search requires the ability to access elements in constant time based on their index, but in a linked list, accessing an element requires traversing the list from the beginning until the desired element is reached.

Binary search works efficiently on data structures that support random access, such as arrays, where elements can be accessed in constant time using their indices. In a linked list, each element only contains a reference to the next element, so random access is not possible. Traversing a linked list to access a specific element has a time complexity of O(n), where n is the length of the list.

If you want to search for an element in a linked list, a different algorithm such as linear search would be more appropriate, where you iterate through the list sequentially until the desired element is found or the end of the list is reached. Linear search has a time complexity of O(n), where n is the length of the list.

c) Briefly discuss inserting mechanism of an item at the beginning, after a given node, at the end and also to a sorted list.

Inserting an item at the beginning of a linked list

To insert an item at the beginning of a linked list, you can follow these steps:

1. Create a new node and set its data to the value you want to insert.
2. Set the new node's next pointer to the current head of the list.
3. Set the head of the list to the new node.

Inserting an item after a given node in a linked list

To insert an item after a given node in a linked list, you can follow these steps:

1. Find the node after which you want to insert the new item.
2. Create a new node and set its data to the value you want to insert.
3. Set the new node's next pointer to the node's next pointer.
4. Set the node's next pointer to the new node.

Inserting an item at the end of a linked list

To insert an item at the end of a linked list, you can follow these steps:

1. Create a new node and set its data to the value you want to insert.
2. Set the new node's next pointer to NULL.
3. Find the last node in the list.
4. Set the last node's next pointer to the new node.

Inserting an item to a sorted linked list

To insert an item to a sorted linked list, you can follow these steps:

1. Create a new node and set its data to the value you want to insert.
2. Start at the head of the list.
3. Compare the data in the new node to the data in the current node.
4. If the data in the new node is less than the data in the current node, then insert the new node before the current node.
5. Otherwise, continue to the next node.
6. Repeat steps 3-5 until you reach the end of the list.

If the data in the new node is greater than the data in all of the nodes in the list**, then insert the new node at the end of the list.**

To delete an item in a linked list, you can follow these steps:

1. Find the node containing the item you want to delete.
2. Adjust the previous node's next pointer to skip the node you want to delete.
3. Free the memory allocated for the node you want to delete.

4.

a) <u>What is hash table?</u>

A hash table is a data structure that provides fast access to data by using a hash function to map keys to specific locations in an array.

What can be the techniques to avoid collision?

There are several techniques that can be used to avoid collisions in a hash table, including:

1. Separate chaining: This technique involves maintaining a linked list at each hash table index to store multiple values that hash to the same index.
2. Open addressing: This technique involves searching for another available slot within the hash table when a collision occurs.
3. Linear probing: This technique involves searching for the next available slot in the hash table when a collision occurs.
4. Quadratic probing: This technique involves searching for the next available slot using a quadratic sequence when a collision occurs.
5. Double hashing: This technique involves using a second hash function to find an alternative index when a collision occurs.

The choice of collision avoidance technique depends on the specific application and the characteristics of the data being stored in the hash table.

b) Draw a hash table with chaining and a size of 9. Use the hash function "k%9" to

insert the keys 5, 29, 20, 0, 27, 26, 17 and 18 into your table. c) A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as

shown below: 1

0

2

42

3 4 23

34

5 52

6

46

7

8

9

Which one of the following choices gives a possible order in which the key values could have been inserted in the table? Explain.

i. **46, 42, 34, 52, 23, 33**

ii. 34, 42, 23, 52, 33, 46

iii. 46, 34, 42, 23, 52, 33

iv. 42, 46, 33, 23, 34, 52

b) Here is an example of a hash table with chaining and a size of 9 using the hash function "k%9" to insert the keys 5, 29, 20, 0, 27, 26, 17 and 18:

Index	Value
0	0
1	29
2	18
3	27
4	5
5	20
6	26
7	
8	17

c) The correct choice is (iii) 46, 34, 42, 23, 52, 33.

When using linear probing, items are inserted sequentially into the next available slot in the hash table starting from the hashed index.

Assuming the first value inserted is 46, it would be hashed to index 6. Then 42 would be inserted into index 7, 34 would be inserted into index 8, 23 would be inserted into index 3, 52 would be inserted into index 2, and finally 33 would be inserted into index 4.

This results in the following hash table:

Index	Value
0	
1	
2	52
3	23
4	33
5	
6	46
7	42
8	34
9	

Section B

5. a) What is a queue, how is it different from a stack and how is it implemented?

b) Write the steps involved in the insertion and deletion of an element in a stack.

e) Simulate the postfix expression evaluation algorithm using 18, 6, 7, 6, 2, 1, +3,-, 4, by showing Stack's contents as each element is scanned.

6.

b) **Traverse the following tree in preorder, inorder and postorder.**

c) How can binary search tree overcome the disadvantages of array and linked list? Explain the process of inserting a new unique item in a binary search tree with a simple example.

7. a) Suppose the following eight numbers are inserted in order into an empty binary search tree: 40, 60, 50, 33, 35, 55, 11, 12. Show the tree as each number is inserted into a binary search tree.

b) Consider the following adjacency matrix below: A = 0101 1010 0 1 0 1 Now find out A2, A3, A, B, and from that make the path matrix and tell whether this is strongly connected or not.

8

a) What is directed graph?

b**)Discuss the sequential representation of graph with example.**

Sequential representation of a graph refers to storing the graph's data in sequential data structures such as arrays or lists. There are two commonly used sequential representations for graphs: the adjacency matrix and the adjacency list.

Adjacency Matrix:

In an adjacency matrix, a 2D array is used to represent the graph. The rows and columns of the matrix represent the vertices of the graph, and the cells of the matrix indicate whether there is an edge between the corresponding vertices. If there is an edge between vertices i and j, then the cell (i, j) contains a 1; otherwise, it contains a 0.

Example of an undirected graph:

```
  A---B
 /    \
C-------D
```

The adjacency matrix for this graph is:

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

	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0

Adjacency List:

In an adjacency list, each vertex of the graph is associated with a list that contains its neighboring vertices. The graph is represented as an array of lists (or a list of lists), where each index corresponds to a vertex, and the list at that index contains the vertices that share an edge with the corresponding vertex.

The adjacency list for this graph is:

A: [B, C]

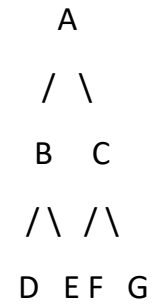
B: [A, D]

C: [A, D]

a)**When do we call a binary tree to be a 2-tree?**

A binary tree is called a 2-tree if every internal node in the tree has exactly two children. In other words, a 2-tree is a type of binary tree where each non-leaf node has exactly two child nodes, and all leaf nodes are at the same level.

For example, here's an illustration of a 2-tree:



In this 2-tree, every internal node (A, B, and C) has exactly two children, and all leaf nodes (D, E, F, and G) are at the same level.

What is the depth of a complete binary tree with 10,00,000 nodes?

The depth of a complete binary tree with n nodes can be calculated using the formula:

Depth = (log2(n)) + 1
For a complete binary tree with 1,000,000 nodes:

Depth = (log2(1000000)) + 1 =](19.93156) + 1 = 20+ 1 = 21

c) Use the Warshall's algorithm to find the shortest path matrix of the adjacency matrix given below:

Data Structure -2019

Section A

1. **(a) Define data structure. Why data structure is necessary?**
   A data structure is a way of organizing and storing data in a computer so that it can be accessed and manipulated efficiently. It provides a way to organize data in a logical and efficient manner, making it easier to access and process data as needed.

   Data structures are necessary for several reasons, including:

Need for Data Structure

1. Data structures are important way of organizing data in a computer
2. It has a different way of storing and organizing data so that it can be used efficiently.
3. It helps us to understand relationship of one data element with other
4. It helps to store a data in logical order.
5. It stores a data that may grow and shrink dynamically over a time and allows efficient access.

(b) What do you mean by Linear data structure and Nonlinear data structure? Give example.

Linear data structure and nonlinear data structure are two broad categories of data structures. Linear data structures are those in which data elements are arranged in a sequential manner, while nonlinear data structures are those in which data elements are not arranged in a sequential manner.

Examples of linear data structures include arrays, linked lists, stacks, and queues, while examples of nonlinear data structures include trees and graphs.

Here is a table summarizing the main differences between linear and nonlinear data structures:

Category	Characteristics	Examples
Linear	Elements are arranged sequentially	Arrays, linked lists, stacks, queues
Nonlinear	Elements are not arranged sequentially	Trees, graphs

It is important to note that this table only provides a high-level overview of the differences between linear and nonlinear data structures, and there are many different variations and implementations of each type.

(c) What is 2D array? How can you represent 2D array in memory?

A 2D array, also known as a two-dimensional array, is a data structure that represents a collection of elements arranged in a grid of rows and columns. In programming, a 2D array is typically used to represent matrices or tables of data.

In memory, a 2D array is represented as a contiguous block of memory that is divided into rows and columns. The elements of the array are stored in a row-major or column-major order, depending on the programming language and the platform.

To access an element in a 2D array, the programmer must specify the row and column indices. For example, to access the element at row i and column j of a 2D array A, the programmer would write A[i][j].

Here is an example of a 2D array in C:

```
int matrix[3][3] = {
                    {1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9}
};
```

In this example, the 2D array has three rows and three columns, and the elements are initialized with the values 1 to 9. In memory, the array is represented as a contiguous block of memory with nine integer elements.

In summary, a 2D array is a data structure that represents a collection of elements arranged in a grid of rows and columns. It is represented in memory as a contiguous block of memory that is divided into rows and columns. To access an element in a 2D array, the programmer must specify the row and column indices.

2.

(a) Suppose Book(2:10, -4:-1, 5:8) is a 3D array with base=200, w=4, calculate Book[5, -1, 7] address in a row major order and column major order. The dimensions of the 3D array Book are:

x: 2 to 10
y: -4 to -1
z: 5 to 8
The base address of the array is 200, and the size of each element (w) is 4 bytes.

To calculate the address of Book[5, -1, 7] in row-major order, we can use the following formula:

```
address = base + w * ((i - x_min) * (y_max - y_min + 1) * (z_max - z_min + 1) + (j - y_min) * (z_max - z_min + 1) + (k - z_min))
```

Substituting the values, we get:

```
address = 200 + 4 * ((5 - 2) * (4 + 1) * (8 - 5 + 1) + (-1 - (-4)) * (8 - 5 + 1) + (7 - 5))
address = 200 + 4 * (3 * 5 * 4 + 3 * 3 + 2)
address = 200 + 4 * 67
address = 200 + 268
address = 468
```

Therefore, the address of Book[5, -1, 7] in row-major order is 468.

To calculate the address of Book[5, -1, 7] in column-major order, we can use the following formula:

```
address = base + w * ((k - z_min) * (x_max - x_min + 1) * (y_max - y_min + 1) + (j - y_min) * (x_max - x_min + 1) + (i - x_min))
```

Substituting the values, we get:

```
address = 200 + 4 * ((7 - 5) * (10 - 2 + 1) * (4 - (-1) + 1) + (-1 - (-4)) * (10 - 2 + 1) + (5 - 2))
address = 200 + 4 * (2 * 9 * 6 + 3 * 9 + 3)
address = 200 + 4 * 189
address = 956
```

Therefore, the address of Book[5, -1, 7] in column-major order is 956.

In summary, the address of Book[5, -1, 7] in row-major order is 468, and the address in column-major order is 956.

(b) What is pointer array? How a pointer can save memory space when storing a 2D array?

A pointer array is an array of pointers, where each pointer points to a value or an object in memory. In other words, a pointer array stores the memory addresses of other variables or objects, rather than storing the values directly.

When storing a 2D array, a pointer array can save memory space by using a technique called "dynamic memory allocation". In dynamic memory allocation, the programmer can allocate memory for the 2D array at runtime, rather than at compile-time. This means that the size of the array can be determined at runtime, based on the input or other factors.

To allocate memory for a 2D array using dynamic memory allocation, the programmer can create a pointer array that points to each row of the array, and then allocate memory for each row separately. For example, the following code creates a 2D array of integers with dimensions 3x4 using dynamic memory allocation:

```
int **arr;
arr = (int **)malloc(3 * sizeof(int *));
for (int i = 0; i < 3; i++) {
   arr[i] = (int *)malloc(4 * sizeof(int));
}
```

In this example, the first line creates a pointer array arr with 3 elements, each pointing to an integer pointer. The second line uses the malloc() function to allocate memory for each row of the array, with 4 integers per row.

Using a pointer array with dynamic memory allocation can save memory space because it allows the programmer to allocate memory only for the required number of rows and columns, rather than allocating a fixed size at compile-time. This can be useful when dealing with large or sparse arrays, where most of the elements are empty or not required. Additionally, the memory allocated for the array can be released using the free() function, which can help prevent memory leaks and optimize the use of memory resources.

(c) Is there any differences between a record and a linear array?

Yes, there are differences between a record and a linear array.
A linear array is a data structure that stores a collection of homogeneous elements of the same data type in contiguous memory locations, accessed using a single index or subscript.
A record, on the other hand, is a data structure that stores a collection of related and possibly heterogeneous data elements in a contiguous block of memory, accessed using field names or labels. Each field in the record may have a different data type, and each field can be accessed using its own label.
In summary, the main difference between a record and a linear array is that a record stores a collection of related and possibly heterogeneous data elements with their own field names or labels, while a linear array stores a collection of homogeneous elements of the same data type, accessed using a single index or subscript.

3.

(a) What are the advantages of Linked List over array?

Advantages	Linked Lists	Arrays
Dynamic Size	Linked lists can grow or shrink dynamically during runtime	Arrays have a fixed size determined at compile time
Efficient Insertions and Deletions	Inserting or deleting an element in a linked list only requires changing the pointers to the adjacent nodes	In an array, inserting or deleting an element requires shifting all the elements in the array after the insertion or deletion point
Memory Efficiency	Linked lists can be more memory-efficient than arrays for small data structures because each node only needs to store the data value and a pointer to the next node	In an array, each element needs to store the actual data value

In summary, linked lists offer dynamic size, efficient insertions and deletions, and memory efficiency over arrays. However, arrays have faster access times and can be more memory-efficient for large data structures or when random access is required. The choice of data structure depends on the specific use case and requirements of the program

(b) Briefly discuss inserting mechanism of an item at the beginning, after a given node and at the end.

Inserting an item in a linked list involves creating a new node with the item and then updating the pointers of the relevant nodes to include the new node.

Inserting an item at the beginning:
To insert an item at the beginning of a linked list, we create a new node with the item and set its next pointer to the current head node. Then, we set the head pointer to point to the new node. This operation has a time complexity of O(1) since it only involves updating a constant number of pointers.

Inserting an item after a given node:
To insert an item after a given node, we create a new node with the item and set its next pointer to the next node of the given node. Then, we update the next pointer of the given node to point to the new node. This operation also has a time complexity of O(1) since it only involves updating a constant number of pointers.

Inserting an item at the end:
To insert an item at the end of a linked list, we first traverse the list to find the last node. Then, we create a new node with the item and set the next pointer of the last node to point to the new node. This operation has a time complexity of O(n) since we need to traverse the entire list to find the last node.

In summary, inserting an item in a linked list involves creating a new node with the item and updating the pointers of the relevant nodes. Inserting an item at the beginning or after a given node has a time complexity of O(1), while inserting an item at the end has a time complexity of O(n) because we need to traverse the list to find the last node.

© What is two way lists? Why it is important? Explain with Schematic diagram.

4. **(a) Briefly discuss different hash collision resolution techniques.**

Hash collision resolution techniques are used to handle the situation where two or more keys produce the same hash value in a hash table. The most commonly used techniques are:

1. Separate Chaining: In this technique, each bucket in the hash table is represented by a linked list. Whenever a collision occurs, the new key-value pair is added to the linked list in the corresponding bucket. This technique is simple to implement

and is space-efficient, but it can cause performance degradation when the linked list becomes too long.

2. Open Addressing: In this technique, when a collision occurs, the hash table looks for the next available slot in the table by following a predefined sequence of steps, such as linear probing or quadratic probing. This technique has good cache locality and can be faster than separate chaining, but it can cause performance degradation when the hash table becomes too full.
3. Robin Hood Hashing: This technique is a variation of open addressing. It uses the same sequence of steps as open addressing, but when a collision occurs, it checks the distance between the current slot and the original slot where the key should have been placed. If the distance is greater than that of the existing key in that slot, it swaps the two keys. This technique tends to reduce the average length of the probe sequence and is faster than separate chaining and open addressing.
4. Cuckoo Hashing: This technique uses multiple hash functions and two or more hash tables. When a collision occurs, the key-value pair is moved to the next hash table using the next hash function. If all hash tables are full, some keys may need to be evicted to make room for new keys. This technique has good worst-case performance but can be complicated to implement.

Overall, the choice of a collision resolution technique depends on the specific requirements of the application, such as the expected number of collisions, the size of the hash table, and the desired performance characteristics.

(c) Draw a hash table with open addressing and a size of 9. Use the hash function "k%9" and quadratic probing Insert the keys: 5, 29, 20, 0, 27, 26, 17 and 18 into your table.

Initialize the hash table with 9 empty slots:

Table: [_, _, _, _, _, _, _, _, _]

Insert key 5:

Hash value: 5 % 9 = 5

Table: [_, _, _, _, _, 5, _, _, _]

Insert key 29:

Hash value: 29 % 9 = 2

Table: [_, _, 29, _, _, 5, _, _, _]

Insert key 20:

Hash value: 20 % 9 = 2 (collision)

Quadratic probing: 2 + 1^2 = 3

Table: [_, _, 29, 20, _, 5, _, _, _]

Insert key 0:

Hash value: 0 % 9 = 0

Table: [0, _, 29, 20, _, 5, _, _, _]

Insert key 27:

Hash value: 27 % 9 = 0 (collision)

Quadratic probing: 0 + 1^2 = 1

Table: [0, 27, 29, 20, _, 5, _, _, _]

Insert key 26:

Hash value: 26 % 9 = 8

Table: [0, 27, 29, 20, _, 5, _, _, 26]

Insert key 17:

Hash value: 17 % 9 = 8 (collision)

Quadratic probing: 8 + 1^2 = 9 (wraps around)

Table: [0, 27, 29, 20, _, 5, _, _, 26]

Quadratic probing: 0 + 2^2 = 4

Table: [0, 27, 29, 20, 17, 5, _, _, 26]

Insert key 18:

Hash value: 18 % 9 = 0 (collision)

Quadratic probing: 0 + 1^2 = 1

Table: [0, 18, 29, 20, 17, 5, _, _, 26]

After inserting all the keys, the final hash table will look like this:

Table: [0, 18, 29, 20, 17, 5, _, _, 26]

Note: "_" represents an empty slot in the hash table.

(e) Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function x mod 10, which of the following statements are true? Explain.

i. 9679, 1989, 4199 hashes to the same value

ii. 1471, 6171 has to the same value

iii. All elements hash to the same value

iv. Each element hashes to a different value

v. None of the above.

The given hash function is x mod 10, which means that the hash table has 10 slots numbered from 0 to 9. To determine which statements are true, we need to calculate the hash values for each input element and see if any of them collide (i.e., hash to the same slot).

4322 mod 10 = 2

1334 mod 10 = 4

1471 mod 10 = 1

9679 mod 10 = 9

1989 mod 10 = 9

6171 mod 10 = 1

6173 mod 10 = 3

4199 mod 10 = 9

Based on these calculations, we can see that:

i. 9679, 1989, and 4199 all hash to the same value (9), so this statement is true.

ii. 1471 and 6171 both hash to the same value (1), so this statement is also true.

iii. All elements do not hash to the same value, so this statement is false.

iv. Each element does not hash to a different value, so this statement is also false.

v. Therefore, the correct answer is (i) and (ii) are true, and (iii), (iv), and (v) are false.

Section B

1. (a) Differentiate between stack and queue.

Here is a table comparing the key differences between stack and queue:

Feature	Stack	Queue
Data Structure	LIFO (Last In First Out)	FIFO (First In First Out)

Feature	Stack	Queue
Insertion/Deletion	Elements are inserted and deleted at the top of the stack	Elements are inserted at the rear and deleted from the front of the queue
Operations	Push (insertion), pop (deletion), peek (view top element)	Enqueue (insertion), dequeue (deletion), peek (view front element)
Implementation	Implemented using a linked list or an array	Implemented using a linked list or an array
Applications	Expression evaluation, function call implementation, backtracking, etc.	Job scheduling, resource sharing, buffering, etc.

In summary, a stack is a LIFO data structure, where elements are inserted and deleted at the top of the stack using push and pop operations, while a queue is a FIFO data structure, where elements are inserted at the rear and deleted from the front of the queue using enqueue and dequeue operations. Both data structures can be implemented using a linked list or an array and have different applications based on their properties.

(b) Convert the following infix expression to its equivalent prefix and postfix expression.

A+(B*C/5-(D/E|F)* G)* H

Infix Expression: A+(B*C/5-(D/E|F)*G)*H

1. Convert to Postfix: We'll use the concept of the stack to convert the expression to postfix notation. Postfix Expression: ABC*5/DE/F|G*-H*+
2. Convert to Prefix: Reverse the postfix expression obtained from the previous step to get the prefix expression. Prefix Expression: +A*/*BC5*-|/DEF*GH

Therefore, the equivalent prefix and postfix expressions for the given infix expression "A+(B*C/5-(D/E|F)*G)*H" are:

Prefix Expression: +A*/*BC5*-|/DEF*GH Postfix Expression: ABC*5/DE/F|G*-H*+

(C) Simulate the postfix expression evaluation algorithm using 18, 6, /, 6, 2, +,*, 12, 4, /,- by showing Stack's contents as each element is scanned.

Postfix Expression: 18 6 / 6 2 + * 12 4 / -

Let's go through the expression step by step:

1. Scan "18": Stack: [18]
2. Scan "6": Stack: [18, 6]
3. Scan "/": Pop two elements from the stack: 6 and 18 Perform the division: 18 / 6 = 3 Push the result back to the stack Stack: [3]
4. Scan "6": Stack: [3, 6]
5. Scan "2": Stack: [3, 6, 2]
6. Scan "+": Pop two elements from the stack: 2 and 6 Perform the addition: 6 + 2 = 8 Push the result back to the stack Stack: [3, 8]
7. Scan "*": Pop two elements from the stack: 8 and 3 Perform the multiplication: 3 * 8 = 24 Push the result back to the stack Stack: [24]
8. Scan "12": Stack: [24, 12]
9. Scan "4": Stack: [24, 12, 4]
10. Scan "/": Pop two elements from the stack: 4 and 12 Perform the division: 12 / 4 = 3 Push the result back to the stack Stack: [24, 3]

11. Scan "-": Pop two elements from the stack: 3 and 24 Perform the subtraction: 24 - 3 = 21 Push the result back to the stack Stack: [21]

After scanning all the elements, the final result is 21. The stack contains only one element, which is the evaluated result.

(d) What is deque?

1. A deque, short for "double-ended queue," is a data structure that allows insertion and removal of elements from both ends. It is an abstract data type that supports the following operations:
2. Insertion at the front: Adds an element to the front of the deque.
3. Insertion at the rear: Adds an element to the rear of the deque.
4. Removal from the front: Removes and returns the element from the front of the deque.
5. Removal from the rear: Removes and returns the element from the rear of the deque.
6. Accessing the front element: Retrieves the element at the front of the deque without removing it.
7. Accessing the rear element: Retrieves the element at the rear of the deque without removing it.
8. Checking if the deque is empty: Determines whether the deque is empty or not.

6(a) What is complete binary tree? What is the parent child relationship?

A complete binary tree is a binary tree in which the nodes are filled from left to right at each level.

1. The left child: This is the node that is positioned to the left of the parent node. It can be identified as the "left child" of the parent node.
2. The right child: This is the node that is positioned to the right of the parent node. It can be identified as the "right child" of the parent node.

1. In a complete binary tree of height h:
    - The number of nodes at the last level (h) is between 1 and 2^h.
    - The total number of nodes in the tree is between 2^h and 2^(h+1) - 1.

- Its left child has an index of 2i.
- Its right child has an index of 2i + 1.

(b) Discuss the linked representation of binary tree in memory.

(c) Simulate the maxheap algorithm for the following values: 100, 29, 90, 148, 12, 34, 90, 89, 120, 99 and 12.

Step 1: Start with an empty heap.

Step 2: Add the first value, 100, as the root of the heap.

Max Heap:

100

Step 3: Add the next value, 29, as the left child of the root.

Max Heap:

```
  100
 /
29
```

Step 4: Add the next value, 90, as the right child of the root.

Max Heap:

```
  100
 / \
29  90
```

Step 5: Add the next value, 148, as the left child of the node with value 29.

Max Heap:

```
   100
  / \
148  90
 /
29
```

Step 6: Add the next value, 12, as the right child of the node with value 148.

Max Heap:

100

/ \

148 90

/ \

29 12

Step 7: Add the next value, 34, as the left child of the node with value 90.

Max Heap:

100

/ \

148 90

/ \ /

29 12 34

Step 8: Add the next value, 90, as the right child of the node with value 12.

Max Heap:

100

/ \

148 90

/ \ / \

29 12 34 90

Step 9: Add the next value, 89, as the left child of the node with value 34.

Max Heap:

100

/ \

148 90

/ \ / \

29 12 34 90

/

89

Step 10: Add the next value, 120, as the right child of the node with value 34.

Max Heap:

100

/ \

148 90

/ \ / \

29 12 34 90

/ \

89 120

Step 11: Add the next value, 99, as the left child of the node with value 90.

Max Heap:

100

/ \

148 99

/\ /\

29 12 34 90

/ \

89 120

Step 12: Add the final value, 12, as the right child of the node with value 99.

Max Heap:

100

/ \

148 99

/\ /\

29 12 34 90

/ \ /

89 120

12

The resulting binary tree satisfies the max heap property, where the value of each node is greater than or equal to the values of its children

7.

(a) What is Directed Graph? Explain.

A directed graph, also known as a digraph, is a type of graph that consists of a set of vertices or nodes connected by directed edges or arcs. In a directed graph, each edge has a specific direction associated with it, indicating the flow or direction of the relationship between the vertices.

(b) Briefly discuss the advantages of binary search tree over array and linked list.

(c) Use the Warshall's algorithm to find the shortest path matrix of the weighted matrix given below.W=

2 5 0 0

2 0 1 4

5 3 3 3

4 2 3 0

Step 2: Apply Warshall's algorithm by considering each vertex as an intermediate vertex one by one.

For k = 1: W(1) =

2 5 0 0

2 0 1 4

5 3 3 3

4 2 3 0

For k = 2: W(2) =

2 5 1 4

2 0 1 4

5 3 3 3

4 2 3 0

For k = 3: W(3) =

2 5 1 4

2 0 1 4

5 3 3 3

4 2 3 0

For k = 4: W(4) =

2 5 1 4

2 0 1 4

5 3 3 3

4 2 3 0

Step 3: The final matrix obtained after applying Warshall's algorithm is the shortest path matrix.

Shortest Path Matrix:

2 5 1 4

2 0 1 4

5 3 3 3

4 2 3 0

9. (a) Discuss the sequential Representation of Graph with example.

Sequential representation of a graph refers to storing the graph's data in sequential data structures such as arrays or lists. There are two commonly used sequential representations for graphs: the adjacency matrix and the adjacency list.

Adjacency Matrix:

In an adjacency matrix, a 2D array is used to represent the graph. The rows and columns of the matrix represent the vertices of the graph, and the cells of the matrix indicate whether there is an edge between the corresponding vertices. If there is an edge between vertices i and j, then the cell (i, j) contains a 1; otherwise, it contains a 0.

Example of an undirected graph:

A---B

/ \

C-------D

The adjacency matrix for this graph is:

| A | B | C | D |

--|---|---|---|---|

A | 0 | 1 | 1 | 0 |

--|---|---|---|---|

B | 1 | 0 | 0 | 1 |

--|---|---|---|---|

C | 1 | 0 | 0 | 1 |

--|---|---|---|---|

D | 0 | 1 | 1 | 0 |

Adjacency List:

In an adjacency list, each vertex of the graph is associated with a list that contains its neighboring vertices. The graph is represented as an array of lists (or a list of lists), where each index corresponds to a vertex, and the list at that index contains the vertices that share an edge with the corresponding vertex.

Example of an undirected graph:

```
  A---B
 /    \
C-------D
```

The adjacency list for this graph is:

A: [B, C]

B: [A, D]

C: [A, D]

(b) How many ways a graph G can be traversed? What is the significance of the STATUS field?

There are two main ways to traverse a graph:

1. Depth-First Search (DFS): In DFS, we start from a source vertex and explore as far as possible along each branch before backtracking. This is done by maintaining a stack of vertices to be visited. **DFS can be implemented recursively or using an iterative approach.**
2. Breadth-First Search (BFS): In BFS, we start from a source vertex and explore all vertices at **the current depth before moving on to vertices at the next depth.** This is done by maintaining a queue of vertices to be visited.

Both DFS and BFS can be used to visit all vertices and edges in a graph, but they differ in the order in which they visit the vertices.

The STATUS field is often used in graph traversal algorithms to keep track of the state of each vertex. In DFS, the STATUS field can be used to mark a vertex as visited, discovered, or unexplored. In BFS, the STATUS field can be used to mark a vertex as discovered or explored. This helps to avoid revisiting vertices and to ensure that all vertices are visited exactly once during the traversal. The STATUS field can be implemented as a boolean array or an integer variable with different values representing different states.

(e) Consider the adjacency list of the Graph G in the following table. Find the nodes that are reachable from node C using Depth First Search.

Node	Adjacency	Node	Adjacency
A	G, E	E	C
B	C	F	A, B
C	F	G	B, C, E
D	C	H	D

 Ans: We want to find all the nodes reachable from the node C. The steps of Depth -

First Search are given bellow:

a) Initially, push C onto the stack as follows: STACK: C

b) Pop and print the top element C and then push onto stack all the neighbors of C as follows: Print C STACK: F

c) Pop and print the top element F and then push onto stack all the neighbors of F as follows: Print F STACK: A, B

d) Pop and print the top element B and then push onto stack all the neighbors of B as follows: Print B STACK: A

e) Pop and print the top element A and then push onto stack all the neighbors of A as follows: Print A STACK: G, E

f) Pop and print the top element E and then push onto stack all the neighbors of E as follows: Print E STACK: G

g) Pop and print the top element G and then push onto stack all the neighbors of G as follows: Print G STACK:

Now stack is empty. So depth-first search is complete. The output is C, F, B, A, E, G

Hence, The nodes C, F, B, A, E, G are reachable from C

Data Structure -2018

1.

a) **How can you sort the elements of a matrix? Explain.**

There are various ways to sort the elements of a matrix, but here are two simple approaches:

1. Sort each row or column individually: To sort the rows of a matrix in ascending order, you can apply a sorting algorithm (such as quicksort, mergesort, or heapsort) to each row separately. Similarly, if you want to sort the columns of a matrix, you can apply a sorting algorithm to each column.
2. Flatten the matrix and sort the resulting array: Another approach is to flatten the matrix into a one-dimensional array and then apply a sorting algorithm to that array. For example, you can concatenate all the rows of the matrix into a single array, and then sort that array using a sorting algorithm.

These approaches are relatively straightforward and can be implemented using basic programming constructs like loops and conditionals. The specific approach you choose will depend on the specific problem you are trying to solve and the constraints you are working with.

b) **How does a pointer array can save memory when store a variable sized group of data? Discuss with necessary figures.**

c) **Maze(2:6, -4:-1, 1:5) is a 3D array with base=100, w=4. Calculate Maze[5, -1, 4] address in a row major order and column major order.**

The dimensions of the Maze array are as follows:

- The first dimension ranges from 2 to 6, for a total of 5 elements.
- The second dimension ranges from -4 to -1, for a total of 4 elements.
- The third dimension ranges from 1 to 5, for a total of 5 elements.

So the total number of elements in the array is 5 x 4 x 5 = 100.

To find the address of Maze[5, -1, 4] in row major order, we need to calculate the linear index of this element, which is given by:

index = (5 - 2) * 4 * 5 + (-1 - (-4)) * 5 + (4 - 1) = 3 * 4 * 5 + 3 * 5 + 3 = 63

So the address of Maze[5, -1, 4] in row major order is 63.

To find the address of Maze[5, -1, 4] in column major order, we need to reverse the order of the dimensions, so that the third dimension changes most rapidly. Then the linear index is given by:

index = (4 - 1) * 5 * 5 + (-1 - (-4)) * 5 + (5 - 2) = 3 * 5 * 5 + 15 + 3 = 78

So the address of Maze[5, -1, 4] in column major order is 78.

2.

a) **What is linked list? Discuss different types of linked list with diagram**.

b) **Discuss the advantages and disadvantages of linked list?**

Linked lists are a data structure that stores a collection of elements, where each element points to the next element in the list. Here are the advantages and disadvantages of using a linked list:

Advantage	Explanation
Dynamic size	**Linked lists can grow or shrink in size as needed, making them flexible for managing memory.**
Efficient insertion and deletion	**Inserting or deleting an element in a linked list requires only updating the pointers to the adjacent elements, which is an O(1) operation. This is much faster than inserting or deleting an element in an array, which requires shifting all the remaining elements.**
No wasted space	**Linked lists only use the memory they need to store the elements and the pointers, without any unused space between elements.**

Advantage	Explanation
Ease of implementation	Linked lists are easy to implement and understand, with a simple node structure consisting of an element and a pointer to the next node.
Versatility	Linked lists can be used to implement various other data structures, such as stacks, queues, and hash tables.

Disadvantage	Explanation
Slow random access	Unlike arrays, linked lists do not allow for direct access to an element by its index. Accessing an element in a linked list requires traversing the list from the beginning, which can be slow for large lists.
More memory overhead	Linked lists require extra memory to store the pointers that link the nodes together. This overhead can be significant for small elements or when the list is large.
No cache locality	Linked lists do not have contiguous memory locations, so traversing a linked list can result in frequent cache misses, which can be slow on modern CPUs.
Limited parallelism	Linked lists do not allow for efficient parallelism, as multiple threads modifying the list can result in race conditions and synchronization issues.
Difficulty in reverse traversal	Traversing a linked list in reverse order is not as efficient as traversing it in forward order, as each node only has a pointer to the next node.

c) **What is Garbage collection? When does it take place?**

Garbage collection is a process used by programming languages to automatically free up memory that is no longer being used by a program. It is a way to manage memory

dynamically, without requiring the programmer to manually allocate or deallocate memory.

Garbage collection typically takes place at runtime, while a program is running. During the process, the garbage collector identifies and removes objects in memory that are no longer being used by the program. This frees up memory that can then be used for other purposes.

The timing of garbage collection varies depending on the programming language and implementation. In some languages, garbage collection may be triggered when the program runs out of available memory, while in others it may be performed periodically or in response to certain events.

Garbage collection is an important feature of many programming languages, as it helps to prevent memory leaks and other issues that can arise when a program does not properly manage its memory usage.

3.

a) **What is sparse matrix? How can you locate element as of aij sparse matrix from a 1D array?**

A sparse matrix is a matrix that contains mostly zero values. Sparse matrices are used in many scientific and engineering applications where the matrix is very large and sparse, and storing all the elements would be inefficient or infeasible.

To store sparse matrices efficiently, specialized data structures are used that only store the non-zero values and their locations. One common data structure used for sparse matrices is the Compressed Sparse Row (CSR) format. In this format, the non-zero values of the matrix are stored in a 1D array, along with two additional arrays that store the column indices and row pointers.

The row pointer array stores the indices of the first element of each row in the 1D array. The column index array stores the column indices of each non-zero element. The values array stores the non-zero values themselves.

To locate an element in a sparse matrix given its indices (i, j), we need to search the corresponding row of the matrix to find the non-zero element. We can use the row

pointers to locate the starting index of the row in the values array. Then, we can perform a binary search on the column index array to locate the position of the element in the values array.

The time complexity of locating an element in a sparse matrix using the CSR format is O(log nnz), where nnz is the number of non-zero elements in the matrix. This is a significant improvement over a dense matrix representation, where locating an element would take O(1) time, but storing and manipulating the matrix would take O(n^2) space and time, respectively.

Overall, the CSR format is an efficient way to store and manipulate sparse matrices, and it is widely used in numerical computing and scientific simulations.

b) Discuss the array representation mechanism of stack.

To represent a stack using an array, we need to define the following operations:

- Push: This operation adds an element to the top of the stack. To implement this operation using an array, we need to insert the new element at the end of the array and update the top pointer to point to the new element.
- Pop: This operation removes and returns the element at the top of the stack. To implement this operation using an array, we need to remove the element at the end of the array and update the top pointer to point to the new top element.
- Peek: This operation returns the element at the top of the stack without removing it. To implement this operation using an array, we simply return the element at the position pointed to by the top pointer.
- IsEmpty: This operation returns a Boolean value indicating whether the stack is empty. To implement this operation using an array, we check whether the top pointer is equal to -1, which indicates that the stack is empty.

c) **Convert the following infix expression to its equivalent prefix and postfix expression.**

i) A+(B-C+D)* E/F+G|H

ii) 1+2-(3*4/5|6)*7

i) A+(B-C+D)* E/F+G|H

Prefix expression: +A*+-BCD/EFG|H

Postfix expression: ABC-D+E*F/-GH|+

ii) 1+2-(3*4/5|6)*7

Prefix expression: -+123/45|67

Postfix expression: 12+3456|/7-

4.

a) What is polish notation? What are the benefits of polish notation?

Polish notation, also known as prefix notation, is a way of writing mathematical expressions in which operators are placed before their operands. For example, the expression 2 + 3 in infix notation would be written as + 2 3 in prefix notation.

The benefits of using Polish notation include:

1. Elimination of parentheses: In infix notation, parentheses are used to specify the order of operations. In prefix notation, the order of operations is determined by the order in which the operators are written, eliminating the need for parentheses.
2. Reduced ambiguity: In infix notation, the order of operations can be ambiguous, leading to confusion and errors. Prefix notation removes this ambiguity by specifying the order of operations explicitly.
3. Simplified parsing: Parsing infix notation requires a complex algorithm, whereas parsing prefix notation is much simpler and can be done using a simple recursive algorithm.
4. Easy to evaluate: Once a prefix expression is parsed, it can be evaluated using a simple stack-based algorithm, which makes it very efficient for evaluating mathematical expressions.
5. Flexibility: Prefix notation allows for the expression of complex mathematical expressions using a compact and flexible syntax.

Overall, Polish notation provides a simple and efficient way to write and evaluate mathematical expressions, eliminating many of the complexities and ambiguities of traditional infix notation. It is widely used in computer science and mathematics, and is a valuable tool for anyone working with complex mathematical expressions.

b) Simulate the infix to postfix transformation algorithm for Q: A+ (B C-D/E1F) G) H by showing the stack's contents as each element is scanned.

c) Suppose 15 elements are maintained by array and another 15 are by Linked List. Which 3 methods take longer time to access 9 element. Justify your answer.

here are 3 methods that take longer time to access the 9th element in an array and a linked list:

1. Using a linear search. A linear search is a simple algorithm that scans through the entire list or array until it finds the desired element. This means that the time complexity of a linear search is O(n), where n is the number of elements in the list or array. So, if there are 15 elements in the array or linked list, then the linear search will take O(15) time to find the 9th element.
2. Using a binary search. A binary search is a more efficient algorithm that can be used to search for an element in a sorted list or array. The binary search works by repeatedly dividing the list or array in half and searching the half that is more likely to contain the desired element. This means that the time complexity of a binary search is O(log n), which is much faster than the O(n) time complexity of a linear search. However, the binary search can only be used if the list or array is sorted.
3. Using a recursive function. A recursive function is a function that calls itself to solve a problem. This can be a useful way to implement a search algorithm, but it can also be inefficient. The time complexity of a recursive function depends on the way that the function is implemented. However, in general, recursive functions tend to be slower than iterative functions.

In the case of the 9th element, the linear search and the recursive function will both take O(15) time to find the element. However, the binary search will only take O(log 15) time, which is much faster. Therefore, the binary search is the most efficient way to access the 9th element in an array or linked list.

Here is a table that summarizes the time complexity of the three methods:

Method	Time Complexity
Linear search	O(n)
Binary search	O(log n)
Recursive function	O(n)

As you can see, the binary search is the most efficient way to access the 9th element in an array or linked list. The linear search and the recursive function are both less efficient, but they may be easier to implement in some cases.

Section-B

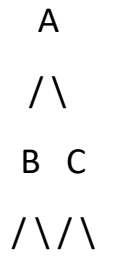
5.

a**) Define the following terms: siblings, successor, and level.**

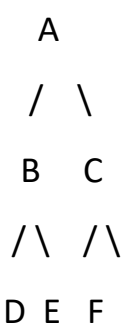
Siblings: Two nodes in a tree are siblings if they have the same parent node. For example, in the following tree, the nodes B, C, D, E, and F are all siblings:

A

Successor: The successor of a node in a tree is the node that comes after it in the tree. For example, in the following tree, the successor of the node B is the node C:



Level: The level of a node in a tree is the number of edges that separate it from the root node. The root node is at level 0, its children are at level 1, and so on. For example, in the following tree, the node B is at level 1, and the node F is at level 2:



b) What is binary search tree. How can binary search tree overcome the disadvantage of array and linked list?

c) Traverse the following tree in preorder, inorder and postorder.

6.

a**) Discuss the linked representation of Graph with example.**

b) Define the following terms with example: (i) Strongly Connected Graph (ii) Directed Graph (iii) Weighted Graph

c) Consider the following adjacency matrix below: A =

1011

0100

0011

1100

Now find out A2, A3, A4, B4 and from that make the path matrix and tell whether this is strongly connected or not.

7)

a) What is heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children.

What are the advantages of heap?

advantages of using a heap data structure:

1. Efficient Insertion and Deletion: O(log n) time complexity for both insertion and deletion operations.
2. Constant-time Access to Extreme Elements: Quick access to the minimum or maximum element (root) of the heap.
3. Priority Queue Implementation: Heaps are commonly used to implement priority queues, allowing efficient management of priorities.
4. Heap Sort: Heap sort algorithm with O(n log n) time complexity for sorting data.
5. Space Efficiency: Heaps are relatively space-efficient compared to other tree-based data structures.
6. Partial Sorting: Efficiently find top k largest or smallest elements without fully sorting the entire dataset.

7. Heap-Based Algorithms: Various algorithms are based on the heap data structure, contributing to optimized solutions in other domains.

b) Use the Warshall's algorithm to find the shortest path matrix of the weighted matrix given w =

2 4 1 0

3 0 0 5

5 1 3 6

3 2 3 0

c)Suppose the following six numbers are inserted in order into an empty binary search tree. 10, 50, 49, 33, 55, 13. Show the tree as each number is inserted into a binary search tree.

Inserting 10:

10

Inserting 50:

10

\

50

Inserting 49:

10

\

50

/

49

Inserting 33:

10

\

50

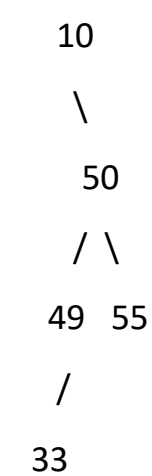
/

49

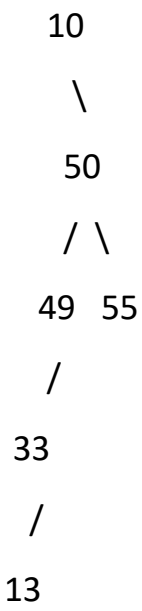
/

33

Inserting 55:



Inserting 13:



❖ Note that in a binary search tree, the left child of a node is always less than the node itself, and the right child of a node is always greater than the node itself. This property allows for efficient searching, insertion, and deletion of elements in the tree.

8.

a) Define hash function and hash table and load factor.

- **Hash function:** A hash function is a function that takes an input value and produces a unique output value. The output value is called the hash code. Hash functions are used in hash tables to map keys to values.
- **Hash table**: A hash table is a data structure that uses hash functions to store and retrieve data. Hash tables are very efficient for storing and retrieving data, because the hash function ensures that the keys are distributed evenly throughout the hash table.
- **Load factor:** The load factor of a hash table is the ratio of the number of elements in the hash table to the size of the hash table. The load factor is

important, because it affects the performance of the hash table. If the load factor is too high, then the hash table may become inefficient.

c) Briefly discuss open addressing quadratic probing method with example.

Quadratic probing is one of the methods used in open addressing to resolve collisions. It determines the next available slot by using a quadratic sequence of probe positions based on the original hash index. The formula used is:

probe_position = (hash_index + c1 * i + c2 * i^2) % table_size

Here, hash_index is the initial index calculated by the hash function, c1 and c2 are constants, i is the number of probes, and table_size is the size of the hash table.

The quadratic probing method incrementally probes slots in a quadratic manner, which helps to distribute the keys more evenly throughout the hash table and reduce clustering. If a slot is occupied, it tries the next probe position until an empty slot is found.

Let's consider an example using quadratic probing with a hash table of size 9. We will insert the keys 3, 5, 29, 20, 0, 27, 26, 17, and 18 into the hash table:

Index	Elements
0	0
1	27
2	18
3	3
4	4
5	5
6	26
7	17
8	20

c) Draw a hash table with chaining and a size of 9. Use the hash function "k%9" to insert the keys 3 5, 29, 20, 0, 27, 26, 17 and 18 into your table.

Bucket| Keys

------- | --------

0 | 0, 27, 18

1 | Empty

2 | 20

3 | 29

4 | Empty

5 | 5

6 | 29

7 | empty

8 | 17

Data Structure -2017 :

Part-A

1. **a) What is linear array? How can you find the no. of elements in any linear array?**

A linear array is a collection of data elements of the same type that are arranged in a sequence or a linear order. Each element in the array is identified by its index or position, which is an integer value starting from 0 and increasing by 1 for each element in the array.

To find the number of elements in a linear array, you can use the following formula:

Number of elements = (Size of the array in bytes) / (Size of each element in bytes)

For example, if you have an array of integers with 20 elements, and each integer takes up 4 bytes of memory, the size of the array in bytes would be:

Size of the array = 20 * 4 = 80 bytes

Therefore, the number of elements in the array would be:

Number of elements = 80 / 4 = 20

So, the array has 20 elements. Note that this formula assumes that each element in the array occupies the same amount of memory. If the elements are of different data types or have varying sizes, the formula will need to be modified accordingly.

b) Simulate the binary search algorithm on the following data: 11 22 33 44 55 66 77 88 99 110 121 132 143 (suppose we search for item 120).

To simulate the binary search algorithm, we need to follow these steps:

1. Set the lower bound and upper bound of the array. lower bound = 0 upper bound = 12
2. Calculate the mid-point of the array. mid-point = (lower bound + upper bound) / 2 = (0 + 12) / 2 = 6
3. Compare the middle element with the search element. Since the middle element is 77 and the search element is 120, we know that the search element is greater than the middle element.
4. If the search element is greater than the middle element, we ignore the left half of the array and search in the right half. Therefore, we set the lower bound to mid-point + 1. lower bound = 6 + 1 = 7
5. Calculate the new mid-point of the array. mid-point = (lower bound + upper bound) / 2 = (7 + 12) / 2 = 9
6. Compare the middle element with the search element. Since the middle element is 110 and the search element is 120, we know that the search element is greater than the middle element.
7. If the search element is greater than the middle element, we ignore the left half of the array and search in the right half. Therefore, we set the lower bound to mid-point + 1. lower bound = 9 + 1 = 10
8. Calculate the new mid-point of the array. mid-point = (lower bound + upper bound) / 2 = (10 + 12) / 2 = 11
9. Compare the middle element with the search element. Since the middle element is 132 and the search element is 120, we know that the search element is less than the middle element.
10. If the search element is less than the middle element, we ignore the right half of the array and search in the left half. Therefore, we set the upper bound to mid-point - 1. upper bound = 11 - 1 = 10
11. Calculate the new mid-point of the array. mid-point = (lower bound + upper bound) / 2 = (7 + 10) / 2 = 8
12. Compare the middle element with the search element. Since the middle element is 99 and the search element is 120, we know that the search element is greater than the middle element.

13. If the search element is greater than the middle element, we ignore the left half of the array and search in the right half. Therefore, we set the lower bound to mid-point + 1. lower bound = 8 + 1 = 9
14. Calculate the new mid-point of the array. mid-point = (lower bound + upper bound) / 2 = (9 + 10) / 2 = 9
15. Compare the middle element with the search element. Since the middle element is 110 and the search element is 120, we know that the search element is less than the middle element.
16. If the search element is less than the middle element, we ignore the right half of the array and search in the left half. Therefore, we set the upper bound to mid-point - 1. upper bound = 9 - 1 = 8
17. Since the lower bound is now greater than the upper bound, we know that the search element is not in the array.

Therefore, the binary search algorithm

c) What is 2D array? How can you represent 2D array in memory?

A 2D array, also known as a two-dimensional array, is an array in which each element is itself an array, forming a grid or matrix of values. A 2D array is often used to represent data that has two dimensions, such as an image, a table, or a map.

In memory, a 2D array is stored as a contiguous block of memory, where each row is stored one after the other. The elements of the array are typically stored in row-major order, which means that the elements of each row are stored together, and the rows themselves are stored one after the other.

To represent a 2D array in memory, we use a pointer to the first element of the array. Since each element of the array is itself an array, the pointer points to the first element of the first row of the array. To access a particular element of the array, we need to calculate its position in memory using its row and column indices.

For example, suppose we have a 2D array of integers with three rows and four columns, as follows:

Copy code

1 2 3 4

5 6 7 8

9 10 11 12

To represent this array in memory, we allocate a block of memory that is large enough to hold all the elements of the array, which in this case would be 3 * 4 = 12 integers. We then store the elements of each row of the array one after the other, in row-major order. The memory layout of this array would look like the following:

Address | Value

--------|------

1000 | 1

1004 | 2

1008 | 3

1012 | 4

1016 | 5

1020 | 6

1024 | 7

1028 | 8

1032 | 9

1036 | 10

1040 | 11

1044 | 12

In this example, the first element of the array is stored at memory address 1000, and each subsequent element is stored at the next consecutive address. To access a particular element of the array, we use its row and column indices to calculate its memory address. For example, to access the element at row 2, column 3 (which has the value 7), we would calculate its memory address as follows:

yaml

Copy code

Address = 1016 + (2 * 4) + 3 = 1025

This calculation takes into account the size of each element of the array (4 bytes in this case) and the number of rows and columns in the array. By adding the appropriate offset to the address of the first element of the array, we can access any element of the array using its row and column indices.

2. **a) Room (1:8, -4:1, 6:10) is a 3D array with base-400, w-2, calculate Room [4, -2, 7) address in a row major order and column major order.**

b) What is sparse matrix? How can you locate element as of a sparse matrix from a ID array? memory saving if we store a sparse matrix in a ID array rather than a 2D

c) What is array?

3. a) What is a Linked List? Discuss with example.

b) Define overflow and underflow? How can you handle them?

c) Briefly discuss inserting mechanism of an item at the beginning, after a given node, at the end and to a sorted list.

d) What is Header Linked List?

4. a) Differentiate between stack and queue. postfix expression.

b) Convert the following infix expression to its equivalent prefix and

(i) A+(B-C) D/E+F/GTH

(ii) (1+2) 3-4/5 617

c) Simulate the postfix expression evaluation algorithm using 20, 5, 2,*,/, 2, 3, 1. 1, 4. /,- by showing Stack's contents as each element is scanned.

Part-B

5. a) What is complete binary tree? A complete binary tree has 1129 nodes, find out the depth of the tree.

A complete binary tree is a binary tree where all levels are completely filled except possibly the last level, which is filled from left to right. In other words, all nodes at a given level are as far left as possible.

To find out the depth of a complete binary tree with 1129 nodes, we can use the formula for the maximum number of nodes in a complete binary tree of depth d:

2^(d+1) - 1

We can use this formula to find the largest depth d such that the number of nodes in the tree is less than or equal to 1129:

2^(d+1) - 1 <= 1129

Simplifying this inequality, we get:

2^(d+1) <= 1130

Taking the logarithm base 2 of both sides, we get:

d + 1 <= log2(1130)

d <= log2(1130) - 1

Using a calculator, we get:

d <= 9.137

Since the depth of a binary tree must be a positive integer, the depth of a complete binary tree with 1129 nodes is 9.

b) Discuss the linked representation of binary tree in memory.

A binary tree can be represented using a linked data structure where each node contains a data element, a left child pointer, and a right child pointer.

To represent a binary tree using linked representation, we can define a struct or class for the nodes, where each node has a data element and two pointers, one for the left child and one for the right child. The root of the binary tree is represented by a pointer to the first node.

Here is an example C++ struct for a node in a binary tree:

```
struct Node {
   int data;
   Node* left;
   Node* right;
};
```

To create a binary tree using linked representation, we can create nodes for each element of the tree and set the left and right pointers to the appropriate child nodes. For example, to create the following binary tree:

```
markdown
    5
   / \
  3   8
 / \
2   4
```

We can create the nodes for each element and set the pointers as follows:

```
Node* root = new Node{5, nullptr, nullptr};
root->left = new Node{3, nullptr, nullptr};
root->right = new Node{8, nullptr, nullptr};
root->left->left = new Node{2, nullptr, nullptr};
root->left->right = new Node{4, nullptr, nullptr};
```

In this representation, the left child pointer of a node points to the left child node, and the right child pointer points to the right child node. If a node has no child, the pointer is set to nullptr.

This linked representation of binary trees has several advantages. It allows for efficient insertion and deletion of nodes and enables traversal algorithms such as inorder, preorder, and postorder traversals. However, it requires more memory than an array representation of the binary tree and can have slower access times due to the use of pointers

c) Simulate the maxheap algorithm for the following values: 67, 29, 90, 48, 12, 34, 90, 9, and 12.

6.

a) Illustrate similar and copies with example.

b) What is an extended tree?

An extended binary tree is a binary tree in which all the leaf nodes are at the same level. If a leaf node is missing in the original binary tree, it is represented in the extended binary tree as a null node.

For example, consider the following binary tree:

```
markdown
     5
    / \
   3   8
  / \
 2   4
      \
       6
```

In the extended binary tree, we add null nodes to the level where the leaf node 6 is missing:

```
javascript
       5
      / \
     3   8
    / \    \
   2   4    null
  /   \
null   6
```

The extended binary tree has all the leaf nodes at the same level, which simplifies certain algorithms on binary trees. For example, the extended binary tree can be used to efficiently represent heaps, where all the leaf nodes must be at the same level. Additionally, the extended binary tree can be used to represent expressions, where the operands are the leaf nodes and the operators are the internal nodes.

c) Simulate the postorder traversal algorithm for the following tree.

7. **a) Define the following terms: Connected Graph, Path, and Weighted Graph.**

Here's a table defining the terms Connected Graph, Path, and Weighted Graph:

Term	Definition
Connected Graph	A connected graph is a graph where there is a path between every pair of vertices in the graph. In other words, there are no disconnected components in the graph.
Path	A path in a graph is a sequence of vertices connected by edges. The length of a path is the number of edges in the path.
Weighted Graph	A weighted graph is a graph where each edge is assigned a weight or cost. The weight represents some quantitative value associated with the edge, such as distance, time, or cost.

b) How many ways a graph G can be traversed? What is the significance of the STATUS field?

c) Consider the adjacency list of the Graph G in the following table. Draw the graph and find out the path from A to F with minimum number of nodes along that path using Breadth First Search.

8. a) Define deque and priority queue with example.

b) Write the steps of preorder and postorder traversal of a binary tree.

c) What is binary search tree? Mention the advantages of a binary search tree.

Data Structure -2016

Part-A

(a) Define data structure. Why is data structure necessary? (b) What do you mean by Linear data structure and Nonlinear data structure? Give example.

(c) What is sparse matrix? What is the difference between triangular matrix and Tridiagonal matrix?

2.

(a) What are the advantages of Linked List?

(b) Suppose 10 elements are maintained by array and another 10 are by Linked List. Which methods take longer time to access 7th element. Justify your answer. (c) What is two way lists? Why is it important? Explain with schematic diagram.

3.

(a) Discuss the array representation mechanism of stack. (b) What is polish notation? What are the benefits of polish notation?

(c) Convert the following infix expression to its equivalent prefix and postfix expression. (i) A*B/C+D1(E-F*G)/H (ii) 1+2*3/415*6-7*8

4.

(a) Simulate the postfix expression evaluation algorithm using 12, 6, 1, 6, 2, +, *, 12, 4, /, - by showing Stack's contents as each element is scanned.

(b) What is recursion? Explain the use of recursion.

(c) Explain the operations on queue with example.

Part-B

(a) Briefly discuss inserting and deleting mechanism of an item in the linked list.

(b) Define deque and priority queue with example. (c) What is garbage collection? When does it take place?

6.

(a) Simulate the preorder traversal algorithm for the following tree.

-D

C

M

2

1.1

(b) What is binary search tree? Why binary search tree is important?

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

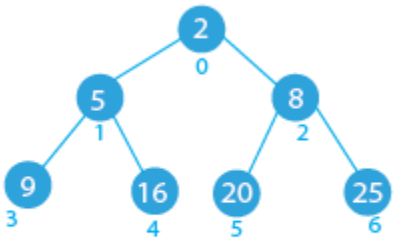
- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

IMPORTANCE OF BST

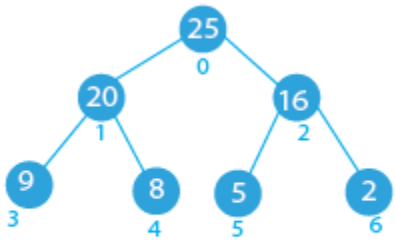
The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

- ❖Insert a key value.
- ❖Determine whether a key value is in the tree.
- ❖Remove a key value from the tree.
- ❖Print all the key values in sorted order.

(f) What is the difference between maxheap and minheap?



Min Heap



Max Heap

Heap

- A *max tree* is a tree in which the key value in each node is larger than the key values in its children. A *max heap* is a complete binary tree that is also a max tree.

- A *min tree* is a tree in which the key value in each node is smaller than the key values in its children. A *min heap* is a complete binary tree that is also a min tree.

fb.com/prodhan24 github.com/prodhan2