BCS304- DSA SIMP Answers - 2025

1. **What is Data Structures? Classify and Explain them briefly. Also explain the basic operations that can be performed on data structures. List out the applications.**

A data structure is a way of organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.[1] Data structures can be classified into two types: linear and non-linear.

- **Linear:** A data structure is said to be linear if its elements form a sequence or a linear list. Examples: Array. Linked List, Stacks, and Queues.[2]
- **Non-linear:** A non-linear data structure is a data structure in which the data elements are not arranged in a sequential order. Examples: Trees and graphs.

Basic operations that can be performed on data structures:

- **Insertion:** Add a new data item in the given collection of data items.
- **Deletion:** Delete an existing data[3] item from the given collection of data items.
- **Traversal:** Access each data item exactly once so that it can be processed.
- **Searching:** Find out the location of the data item if it exists in the given collection of data items.
- **Sorting:** Arranging the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary[4] order in case of alphanumeric data.[5]

Applications of data structures:

- **Arrays:** Storing and accessing a collection of elements of the same data type, such as a list of numbers or a list of names.
- **Linked lists:** Implementing dynamic data structures, such as stacks and queues, where the size of the data structure can change at runtime.
- **Trees:** Representing hierarchical relationships between data elements, such as a family tree or a file system.
- **Graphs:** Representing relationships between data elements that are not necessarily hierarchical, such as a social network or a road network.

2. **Define Pointers. Give advantages and disadvantages of pointers.**

Pointers are variables that store the address of another variable. The data type of the pointer and the variable must be the same.

Advantages:

- Pointers are used for dynamic memory allocation.
- Pointers are used to implement stacks, queues, linked lists, and binary trees.
- Pointers are used to pass parameters by reference.

Disadvantages:

- Pointers are a bit complex to understand and use.
- If pointers are not used carefully, they can cause memory leaks and other problems.

How do you declare and initialize the pointer?

Declaration:

```C
data_type *pointer_name;
```

Initialization:

```C
pointer_name = &variable_name;
```

How do you access the value pointed to by a pointer?

You can access the value pointed to by a pointer using the * operator. For example, if ptr is a pointer to an integer variable x, then *ptr will give the value of x.

3. **Differentiate between static and dynamic memory allocations. What are the different types of memory Allocation? Explain the Different functions that supports Dynamic Memory Allocation - 3+7M**

Static memory allocation is performed at compile time. The size of the memory allocated is fixed and cannot be changed during runtime. **Dynamic memory allocation** is performed at runtime. The size of the memory allocated can be changed during runtime.

Types of memory allocation:

- **Static memory allocation:** The memory is allocated from the stack.
- **Dynamic memory allocation:** The memory is allocated from the heap.

Functions that support dynamic memory allocation:

- **malloc():** Allocates a block of memory of the specified size.
- **calloc():** Allocates a block of memory of the specified size and initializes[6] all the bytes to zero.
- **realloc():** Resizes the previously allocated memory block.
- **free():** Releases the previously allocated memory block.

4. **Explain Traversing, inserting, deleting, searching, and sorting operations with a programming example or an algorithm in an array.**

Traversing: Traversing an array means accessing each element of the array exactly once.

Inserting: Inserting an element into an array means adding a new element to the array.

Deleting: Deleting an element from an array means removing an element from the array.

Searching: Searching for an element in an array means finding the location of the element in the array.

Sorting: Sorting an array means arranging the elements of the array in some order.

Programming example:

C

```c
#include <stdio.h>

int main() {
    int arr[100], n, i, pos, val;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    printf("Enter the elements: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Traversing the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Inserting an element
    printf("Enter the position to insert: ");
    scanf("%d", &pos);
    printf("Enter the value to insert: ");
    scanf("%d", &val);
    for (i = n - 1; i >= pos; i--) {
        arr[i + 1] = arr[i];
    }
    arr[pos] = val;
    n++;

    // Traversing the array after insertion
    printf("The elements of the array after insertion are: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    // Deleting an element
    printf("Enter the position to delete: ");
    scanf("%d", &pos);
    for (i = pos; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    n--;

    // Traversing the array after deletion
    printf("The elements of the array after deletion are: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Searching for an element
    printf("Enter the value to search: ");
    scanf("%d", &val);
    for (i = 0; i < n; i++) {
        if (arr[i] == val) {
            printf("The value is found at position %d\n", i);
            break;
        }
    }
    if (i == n) {
        printf("The value is not found\n");
    }

    // Sorting the array
    for (i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }

    // Traversing the array after sorting
    printf("The elements of the array after sorting are: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

5. **Define Arrays. Explain different types of arrays. How can a one-dimensional array be initialized? Explain with example.**

An array is a collection of elements of the same data type stored in contiguous memory locations.

Types of arrays:

- **One-dimensional array:** An array with a single row or column.
- **Multi-dimensional array:** An array with multiple rows and columns.

Initializing a one-dimensional array:

C

```
data_type array_name[size] = {value1, value2, ..., valueN};
```

Example:

C

```
int arr[5] = {1, 2, 3, 4, 5};
```

6. **Define structure and different type of structures. Explain how a structure can be represented in C and also define union.**

A structure is a collection of variables of different data types grouped under a single name.

Types of structures:

- **Simple structure:** A structure that contains only basic data types.
- **Nested structure:** A structure that contains another structure as a member.
- **Array of structures:** An array where each element is a structure.

Representing a structure in C:

C

```
struct structure_name {
   data_type member1;
   data_type member2;
   ...
};
```

Union: A union is a special data type that allows different data types to be stored in the same memory location.

7. **What is a sparse matrix? Briefly explain it with an example.**

A sparse matrix is a matrix in which most of the elements are zero.

Example:

```
0 0 0 0
0 0 1 0
0 2 0 0
0 0 0 0
```

8. **Explain String function with an example.**

String functions are functions that are used to manipulate strings. Some common string functions in C are:

- **strlen():** Returns the length of a string.
- **strcpy():** Copies one string to another.
- **strcat():** Concatenates two strings.
- **strcmp():** Compares two strings.[7]

Example:

C

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str1[100] = "Hello";
    char str2[100] = "World";

    printf("Length of str1: %ld\n", strlen(str1));

    strcpy(str1, str2);
    printf("str1 after copying str2: %s\n", str1);

    strcat(str1, str2);
    printf("str1 after concatenating str2: %s\n", str1);

    printf("Comparison of str1 and str2: %d\n", strcmp(str1, str2));

    return 0;
}
```

9. **Define stack and its application and List and implement pop, push, isempty, isfull operations in stack using C, briefly explain how array can be implemented in a stack -12M**

A stack is a linear data structure that follows the LIFO (Last-In, First-Out) principle. This means that the last element added to the stack is the first element to be removed.[1]

Applications of stacks:

- Function calls
- Evaluating expressions
- Undo/Redo functionality
- Backtracking algorithms

Stack operations:

- **push(x):** Adds an element x to the top of the stack.
- **pop():** Removes the top element from the stack.
- **isempty():** Returns true if the stack is empty, false otherwise.
- **isfull():** Returns true if the stack is full, false otherwise.

Implementing stack operations in C:

C

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

void push(int x) {
  if (top == MAX_SIZE - 1) {
    printf("Stack Overflow\n");
    exit(1);
  }
  stack[++top] = x;
}

int pop() {
  if (top == -1) {
    printf("Stack Underflow\n");
    exit(1);
  }
```

```
    return stack[top--];
}

int isempty() {
    return top == -1;
}

int isfull() {
    return top == MAX_SIZE - 1;
}

int main() {
    push(10);
    push(20);
    push(30);

    printf("Popped element: %d\n", pop());

    printf("Is stack empty? %s\n", isempty() ? "Yes" : "No");

    return 0;
}
```

Implementing a stack using an array:

An array can be used to implement a stack by using a variable to keep track of the top of the stack. The push operation increments the top variable and adds the new element to the array. The pop operation returns the element at the top variable and decrements the top variable.

10. **What is recursion? Give two conditions to be followed for successive working of recursive programs. Write a 'c' recursive program to solve the tower of Hanoi problem.**

Recursion is a technique where a function calls itself.

Conditions for recursion:

- **Base case:** There must be a base case that stops the recursion.
- **Recursive case:** The recursive case must move the solution towards the base case.

Recursive program for Tower of Hanoi:

C

```
#include <stdio.h>

void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
    if (n == 1) {
        printf("Move disk 1 from rod %c to rod %c\n", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    printf("Move disk %d from rod %c to rod %c\n", n, from_rod, to_rod);
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

int main() {
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return 0;
}
```

11. **Write a function to evaluate the postfix expression. Illustrate the same for the given postfix expression PQR-+ assuming P=5, Q=3 and R=2*

C

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

void push(int x) {
    if (top == MAX_SIZE - 1) {
        printf("Stack Overflow\n");
        exit(1);
    }
    stack[++top] = x;
}

int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        exit(1);
    }
    return stack[top--];
}
```

```
int evaluatePostfix(char* exp) {
    for (int i = 0; exp[i]; ++i) {
        if (isdigit(exp[i]))
            push(exp[i] - '0');
        else {
            int val1 = pop();
            int val2 = pop();
            switch (exp[i]) {
                case '+': push(val2 + val1); break;
                case '-': push(val2 - val1); break;
                case '*': push(val2 * val1); break;
                case '/': push(val2 / val1); break;
            }
        }
    }
    return pop();
}

int main() {
    char exp[] = "532-*+";
    printf("Postfix evaluation: %d\n", evaluatePostfix(exp));
    return 0;
}
```

**Illustration for PQR-+ assuming P=5, Q=3 and R=2:*

1. Push P (5) onto the stack.
2. Push Q (3) onto the stack.
3. Push R (2) onto the stack.
4. Pop R (2) and Q (3) from the stack.
5. Multiply R and Q (2 * 3 = 6).
6. Push the result (6) onto the stack.
7. Pop 6 and P (5) from the stack.
8. Subtract P from 6 (6 - 5 = 1).
9. Push the result (1) onto the stack.
10. Pop 1 from the stack. This is the final result.
11. **Differentiate between Circular queue, Linear queue and Priority queue with a c program representing different functions (Insert,delete.display-Write syntax only), advantages and its working -12M**

Linear Queue: A linear queue is a data structure that follows the FIFO (First-In, First-Out) principle. Elements are added to the rear and removed from the front. It can suffer from a limitation where the queue appears full even if there is space available at the front because the rear has reached the end of the allocated memory.

Circular Queue: A circular queue is a linear queue that connects the rear and front ends to form a circle. This allows elements to be added to the front of the queue even if the rear is at the end of the allocated memory, as long as there is space available.

Priority Queue: A priority queue is a queue where each element has a priority associated with it. Elements with higher priority are dequeued before elements with lower priority,[2] regardless of their order of insertion.

C program syntax for queue operations:

C

```c
// Linear Queue
void insert_linear(int queue[], int* front, int* rear, int size, int value);
int delete_linear(int queue[], int* front, int* rear);
void display_linear(int queue[], int front, int rear);

// Circular Queue
void insert_circular(int queue[], int* front, int* rear, int size, int value);
int delete_circular(int queue[], int* front, int* rear, int size);
void display_circular(int queue[], int front, int rear, int size);

// Priority Queue (implementation-dependent)
void insert_priority(int queue[], int* front, int* rear, int size, int value, int priority);
int delete_priority(int queue[], int* front, int* rear, int size);
void display_priority(int queue[], int front, int rear, int size);
```

13. **Explain how to implement a queue using dynamically allocated array(Take circular queue as example)**

To implement a circular queue using a dynamically allocated array:

1. **Initialization:**
   - Allocate memory for an array of the desired size using malloc() or calloc().
   - Initialize front and rear to -1.
2. **Insertion:**
   - If the queue is full ((rear + 1) % size == front), reallocate memory for a larger array using realloc().
   - Increment rear using the modulo operator (rear = (rear + 1) % size) to wrap around the array.
   - Add the new element at the rear index.
3. **Deletion:**
   - If the queue is empty (front == -1), return an error.
   - Retrieve the element at the front index.
   - Increment front using the modulo operator (front = (front + 1) % size) to wrap around the array.
   - If the queue becomes empty (front == rear), reset front and rear to -1.
4. **Display:**
   - If the queue is empty (front == -1), return an error.

    ○ Iterate from front to rear (using the modulo operator to wrap around) and print each element.

5. **Give differences between SLL and DLL. How are they represented (Order), Explain different functions of SLL and DLL using syntax of a programming example**

SLL (Singly Linked List):

- Each node has a data field and a pointer to the next node.
- Nodes are traversed in one direction only (from head to tail).

DLL (Doubly Linked List):

- Each node has a data field, a pointer to the next node, and a pointer to the previous node.
- Nodes can be traversed in both directions (from head to tail and from tail to head).

Representation:

- SLL: Linear, unidirectional.
- DLL: Linear, bidirectional.

Programming example syntax:

C

```c
// SLL
struct Node {
    int data;
    struct Node* next;
};

void insert_sll(struct Node** head, int data);
void delete_sll(struct Node** head, int key);
void display_sll(struct Node* head);

// DLL
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

void insert_dll(struct Node** head, int data);
void delete_dll(struct Node** head, int key);
void display_dll(struct Node* head);
```

15. **Distinguish arrays and linked lists Explain advantages of circular lists with respect to other**

lists.

Arrays:

- Fixed size.
- Elements are stored in contiguous memory locations.
- Direct access to elements using index.

Linked Lists:

- Dynamic size.
- Elements can be scattered in memory.
- Sequential access to elements.

Advantages of circular lists:

- Efficient for implementing queues and circular buffers.
- Allows for continuous traversal of the list.
- Can represent data that is naturally circular (e.g., scheduling algorithms).

16. **Explain how queue and stack are represented using SLL**

Queue using SLL:

- Enqueue: Insert new nodes at the tail of the list.
- Dequeue: Delete nodes from the head of the list.

Stack using SLL:

- Push: Insert new nodes at the head of the list.

Pop: Delete nodes from the head of the list.

25. **Construct a binary tree from the given preorder and inorder sequence:**
Preorder: ABDG CHIEF
Inorder: DGBAHEICF
Inorder: 4-8-2-5-1-6-3-7
Postorder: 8-4-5-2-6-7-3-1
To construct a binary tree from its preorder and inorder traversals, we follow these steps:
    1. The first element in the preorder traversal is the root of the tree.
    2. Find the root in the inorder traversal. This divides the inorder traversal into left and right subtrees.
    3. Recursively construct the left and right subtrees using the corresponding sub-sequences in the preorder and inorder traversals.

Applying these steps to the given sequences, we get the following binary tree: A

```
   /  \
  B    C
 / \  / \
D  G H  I
  / \  /
  E  F K
```

26. **Find the Inorder, Preorder and Postorder traversal for the following:** [Image 1]
    Inorder: DBFEAGCLHJK
    Preorder: ABDECFGHLJK
    Postorder: DFBGECHLJK
27. **With an example describe Binary Tree Level order traversal. Write the C function to perform Binary Tree Level order traversal.**
    Level order traversal of a binary tree traverses the tree level by level, starting from the root, and going down to the deepest level. Within each level, nodes are visited from left to right.
    Example:
    For the binary tree in question 25, the level order traversal would be: A B C D G H I E F K
    C function for level order traversal:
    C

```
#include <stdio.h>
#include <stdlib.h>

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

// Function to create a new node
struct Node* newNode(int data) {
    struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to print level order traversal of a tree
void printLevelOrder(struct Node* root) {
    int h = height(root);
    int i;
    for (i = 1; i <= h; i++) {
        printCurrentLevel(root, i);
    }
}

// Function to print nodes at a current level
void printCurrentLevel(struct Node* root, int level) {
    if (root == NULL) {
        return;
    }
    if (level == 1) {
        printf("%d ", root->data);
    } else if (level > 1) {
        printCurrentLevel(root->left, level - 1);
        printCurrentLevel(root->right, level - 1);
    }
}

// Function to get the height of the tree
int height(struct Node* node) {
    if (node == NULL) {
        return 0;
    } else {
        int lheight = height(node->left);
        int rheight = height(node->right);
        if (lheight > rheight) {
            return (lheight + 1);
        } else {
            return (rheight + 1);
        }
    }
}

int main() {
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
    printLevelOrder(root);
    return 0;
}
```

28. **Define a threaded binary tree. What are the advantages of threaded binary tree over binary trees?**
A threaded binary tree is a binary tree variant where the NULL pointers in nodes are replaced with threads. These threads point to either the inorder predecessor or successor of the node, optimizing inorder traversal.
Advantages of threaded binary trees:
   - Efficient inorder traversal without recursion or a stack.
   - Threads can be used to implement other tree operations more efficiently.
   - Reduced space consumption compared to traditional inorder traversal methods.

29. **Write recursive functions for the following operations on BST:**
   - **Insert_key()**
C
```
#include <stdio.h>
#include <stdlib.h>

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

// Function to create a new node
struct Node* newNode(int data) {
    struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to insert a new key in BST
struct Node* insert_key(struct Node* node, int key) {
    if (node == NULL) {
        return newNode(key);
    }

    if (key < node->data) {
        node->left = insert_key(node->left, key);
    } else if (key > node->data) {
        node->right = insert_key(node->right, key);
    }

    return node;
}

int main() {
    struct Node *root = NULL;
    root = insert_key(root, 50);
    insert_key(root, 30);
    insert_key(root, 20);
    insert_key(root, 40);
    insert_key(root, 70);
    insert_key(root, 60);
    insert_key(root, 80);

    return 0;
}
```

- **Delete_key()**

C

```
#include <stdio.h>
#include <stdlib.h>

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

```
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to find the node with minimum key value in a BST
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}

// Function to delete a node with a given key in BST
struct Node* delete_key(struct Node* root, int key) {
    if (root == NULL) {
        return root;
    }

    if (key < root->data) {
        root->left = delete_key(root->left, key);
    } else if (key > root->data) {
        root->right = delete_key(root->right, key);
    } else {
        if (root->left == NULL) {
            struct Node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node *temp = root->left;
            free(root);
            return temp;
        }

        struct Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = delete_key(root->right, temp->data);
    }
    return root;
}

int main() {
  struct Node *root = NULL;
  root = insert_key(root, 50);
  insert_key(root, 30);
  insert_key(root, 20);
  insert_key(root, 40);
  insert_key(root, 70);
  insert_key(root, 60);
  insert_key(root, 80);

  printf("Delete 20\n");
  root = delete_key(root, 20);

  printf("Delete 30\n");
  root = delete_key(root, 30);

  printf("Delete 50\n");
  root = delete_key(root, 50);

  return 0;
}
```

- **Search_key()**

C

```
#include <stdio.h>
#include <stdlib.h>

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

// Function to create a new node
struct Node* newNode(int data) {
  struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
  temp->data = data;
  temp->left = temp->right = NULL;
  return temp;
}

// Function to search a given key in BST
struct Node* search_key(struct Node* root, int key) {
  if (root == NULL || root->data == key) {
    return root;
  }

  if (root->data < key) {
    return search_key(root->right, key);
  }

  return search_key(root->left, key);
}

int main() {
  struct Node *root = NULL;
  root = insert_key(root, 50);
  insert_key(root, 30);
  insert_key(root, 20);
  insert_key(root, 40);
  insert_key(root, 70);
  insert_key(root, 60);
  insert_key(root, 80);

  struct Node *result = search_key(root, 20);
  (result == NULL) ? printf("Data Not Found\n") : printf("Data Found %d\n", result->data);

  result = search_key(root, 100);
  (result == NULL) ? printf("Data Not Found\n") : printf("Data Found %d\n", result->data);

  return 0;
}
```

30. **Construct BST for the following:** 22, 28, 20, 25, 22, 15, 18, 10, 14

To construct a Binary Search Tree (BST) from the given sequence, we insert each element into the tree while maintaining the BST property (left subtree < node < right subtree).

Following the insertion order, the resulting BST is:

```
    22
   / \
  20   28
 / \  /
15  22  25
/ \
10  18
  /
 14
```

31. **How do you create a note and delete it in BST, State the rules and also design a C program to create a Binary Search Tree of N Integers.**

Creating a node in BST:

1. Allocate memory for a new node.
2. Assign the desired data to the node's data field.
3. Set the left and right child pointers of the node to NULL.

Deleting a node in BST:

1. **Leaf node:** Simply remove the node.
2. **Node with one child:** Replace the node with its child.

3. **Node with two children:** Find the inorder successor (smallest[1] node in the right subtree), replace the node's data with the successor's data, and then delete the successor node.

C program to create a BST of N integers:C

```
#include <stdio.h>
#include <stdlib.h>

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

// Function to create a new node
struct Node* newNode(int data) {
    struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to insert a new key in BST
struct Node* insert(struct Node* node, int key) {
    if (node == NULL) {
        return newNode(key);
    }

    if (key < node->data) {
        node->left = insert(node->left, key);
    } else if (key > node->data) {
        node->right = insert(node->right, key);
    }

    return node;
}

int main() {
    struct Node *root = NULL;
    int n, i, data;

    printf("Enter the number of nodes: ");
    scanf("%d", &n);

    printf("Enter the data for each node:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &data);
        root = insert(root, data);
    }
    return 0;
}
```

Define the Graph, for the given graph show the adjacency matrix and adjacency linked list representation of the graph [Image 3]
A graph is a non-linear data structure consisting of a set of vertices (nodes) connected by edges. Edges can be directed or undirected, and may have weights associated with them.
Adjacency matrix representation:
An adjacency matrix is a 2D array where matrix[i][j] is 1 if there is an edge from vertex i to vertex j, and 0 otherwise.
For the given graph, the adjacency matrix is:

```
|A|B|C|D|E|F|G|
---------------
A|0|1|0|1|1|0|0|
B|1|0|1|0|1|0|0|
C|0|1|0|0|1|1|1|
D|1|0|0|0|1|0|0|
E|1|1|1|1|0|1|0|
F|0|0|1|0|1|0|1|
G|0|0|1|0|0|1|0|
```

Adjacency list representation:
An adjacency list is an array of linked lists, where each linked list represents the neighbors of a vertex.
For the given graph, the adjacency list is:

```
A: B -> D -> E
B: A -> C -> E
C: B -> E -> F -> G
D: A -> E
E: A -> B -> C -> D -> F
F: C -> E -> G
G: C -> F
```

33. **Obtain DFS and BFS traversals for the given graph.** [Image 3]
    - **DFS (Depth-First Search):** A C B E D F G
    - **BFS (Breadth-First Search):** A B D E C F G
34. **Explain the following terminologies with respect to a graph?**
    - **Degree of a node:** The degree of a node is the number of edges incident to it.
    - **Weighted graph:** A weighted graph is a graph where each edge has a weight associated with it.
    - **Adjacency matrix:** An adjacency matrix is a 2D array used to represent a graph where matrix[i][j] is 1 if there is an edge from vertex i to vertex j, and 0 otherwise.
    - **Connected graph:** A connected graph is a graph where there is a path between any two vertices.
    - **Complete graph:** A complete graph is a graph where every pair of distinct vertices is connected by an edge.
    - **Directed Graph:** A directed graph is a graph where each edge has a direction.
    - **Subgraph:** A subgraph is a graph that is a subset of another graph.
    - **Multigraph:** A multigraph is a graph that can have multiple edges between the same pair of

vertices.

35. **What is the Spanning tree of a graph? Explain with an example how a spanning tree is constructed using DFS traversal.**
    A spanning tree of a graph is a subgraph that is a tree and includes all the vertices of the graph.
    To construct a spanning tree using DFS traversal:
    1. Start from any vertex in the graph.
    2. Perform DFS traversal, marking visited vertices.
    3. For each visited vertex, add the edge that led to its discovery to the spanning tree.
    4. The resulting set of edges forms a spanning tree.
36. **What is hashing? What are the key components of hashing? List the different types of hashing functions. Briefly explain each of them.**
    Hashing is a technique used to map data of arbitrary size to fixed-size values. This is done using a hash function, which takes the input data and produces a hash code.
    Key components of hashing:
    - **Hash function:** A function that maps the input data to a hash code.
    - **Hash table:** A data structure used to store the data using the hash codes as keys.

    Types of hash functions:

    - **Division method:** Divides the input data by a number and uses the remainder as the hash code.
    - **Multiplication method:** Multiplies the input data by a number and uses the fractional part of the result as the hash code.
    - **Mid-square method:** Squares the input data and uses the middle digits of the result as the hash code.
    - **Folding method:** Divides the input data into parts, combines the parts (e.g., by addition or XOR), and uses the result as the hash code.
37. **What is a priority queue? How can a priority queue be implemented? How do you decrease the priority at any point, Explain?**
    A priority queue is a data structure that allows elements to be inserted with a priority and retrieved in order of their priority.[1]
    Implementations of priority queue:
    - **Binary heap:** A binary tree-based structure that satisfies the heap property (parent node has higher priority than its children).
    - **Fibonacci heap:** A more complex heap structure with better performance for some operations.

    Decreasing priority:
    1. Locate the element in the priority queue.
    2. Decrease the priority of the element.
    3. Restore the heap property by moving the element towards the root until its priority is less than or equal to its parent's priority.
38. **What is an Optimal Binary Search Tree (OBST)? Explain the concept of "search cost" in the context of Optimal Binary Search Trees, How is the cost calculated**
    An Optimal Binary Search Tree (OBST) is a BST that minimizes the average search time for a given set of keys and their probabilities of being searched.
    Search cost:
    The search cost for a key in an OBST is the number of comparisons required to find the key. The average search cost is the sum of the search costs for all keys, weighted by their probabilities.
    Cost calculation:
    The cost of an OBST is calculated as the sum of the products of each key's probability and its depth

in the tree. The optimal BST is the one with the minimum cost.

39. **What is the time complexity for constructing an Optimal Binary Search Tree using dynamic programming?**
    The time complexity for constructing an Optimal Binary Search Tree using dynamic programming is O(n^3), where n is the number of keys.
40. **Explain Static and Dynamic Search keys in OBST**
    - **Static search keys:** The probabilities of searching for the keys are known in advance and do not change over time.

Dynamic search keys: The probabilities of searching for the keys can change over time.