

CBCS SCHEME

USN

BCS304

**Third Semester B.E./B.Tech. Degree Supplementary Examination,
June/July 2024**

Data Structures and Applications

Time: 3 hrs.

Max. Marks: 100

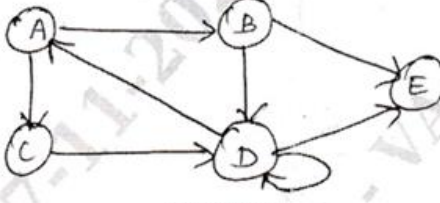
*Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.*

Module – 1			M	L	C
Q.1	a.	Define Data Structures.	04	L1	CO1
	b.	Explain the classification of Data Structures with example.	10	L2	CO1
	c.	Explain all operations of Data Structures.	06	L2	CO1
OR					
Q.2	a.	Explain any five string handling functions supported by 'c' with syntax and example.	10	L2	CO1
	b.	Convert the following infix expression to postfix expression using stack: $A + (B * C - (D/E \wedge F) * G) * H$	10	L3	CO1
Module – 2					
Q.3	a.	List the disadvantages of linear queue and how is it solved in circular queues. Give the algorithm to insert and delete an element in circular queues.	12	L2	CO2
	b.	Explain in detail about multiple queues with relevant functions in 'C'.	08	L2	CO2
OR					
Q.4	a.	Develop a linked list with the basic operations performed on Singly Linked List (SLL) and different types of linked list.	12	L3	CO2
	b.	Examine a node structure for linked representation of polynomial. Explain algorithm to add two polynomial represented using linked list.	08	L2	CO2
Module – 3					
Q.5	a.	Summarize Sparse Matrix. For the given sparse matrix, write the diagrammatic linked list representation. $\begin{bmatrix} 8 & 0 & 0 & 0 \\ 5 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 8 \\ 0 & 0 & 9 & 1 \end{bmatrix}$	08	L3	CO3
	b.	Define Doubly linked list. Write the functions to perform the following operations on doubly linked list. (i) Insert a node at rear end of the list (ii) Delete a node at rear end of the list (iii) Search a node with a given key value	12	L3	CO3
OR					
Q.6	a.	Define Tree with any six tree terminology.	06	L1	CO3
	b.	Write the function for copying and testing of binary tree.	06	L3	CO3
	c.	Draw a binary tree and find out the binary tree traversals for the following expression $3 + 4 * (7 - 6) / 4 + 3$.	08	L3	CO3

Module – 4

Q.7	a.	Construct binary search tree for the given set of values 14, 15, 4, 9, 7, 18, 3, 5, 16, 20 Also perform inroder, preorder and post order traversals of the obtained tree.	08	L3	CO4
	b.	Build a linked list representation of disjoint sets in detail.	06	L3	CO4
	c.	Simplify recursive search algorithm for a binary search tree.	06	L3	CO4

OR

Q.8	a.	Compare a graph with tree. For the graph shown in Fig.Q8(a), show the adjacency matrix and adjacency list representation.  Fig.Q8(a)	08	L3	CO4
	b.	Explain all methods used for traversing a graph with suitable example and write 'C' function for the same.	12	L3	CO4

Module – 5

Q.9	a.	Differentiate between static hashing and dynamic hashing in detail with operations.	10	L2	CO5
	b.	Describe double ended priority queue.	04	L2	CO5
	c.	Explain Hashing with any three Hash functions.	06	L2	CO5

OR

Q.10	a.	What is collision? Explain the method to resolve collision with suitable algorithm of linear probing. Insert keys 72, 27, 36, 24, 63, 81, 92, 101 into % [size 10].	10	L3	CO5												
	b.	Construct an optimal binary search tree for the following keys with the probabilities as <table border="1" data-bbox="284 1294 798 1370"> <tr> <td>Keys</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr> <tr> <td>Probability</td><td>0.25</td><td>0.2</td><td>0.05</td><td>0.2</td><td>0.3</td></tr> </table>	Keys	A	B	C	D	E	Probability	0.25	0.2	0.05	0.2	0.3	10	L3	CO5
Keys	A	B	C	D	E												
Probability	0.25	0.2	0.05	0.2	0.3												

1 a. Define Data Structures.

Data structure is a representation of the logical relationships existing between individual elements of data. A data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

A data structure is a way to organize, store, and manage data efficiently for use in algorithms and computations. Examples include arrays, linked lists, stacks, queues, trees, and graphs, each designed to handle specific types of data operations and access patterns.

1 b. Explain the classification of Data Structures with example.

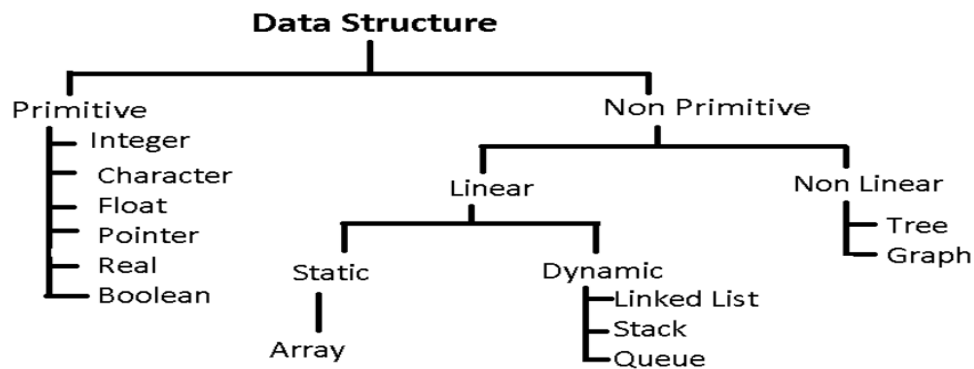


Figure: Classification of Data Structures

Types of Data Structures

1. Linear Data Structures

Linear data structures organize data sequentially, where elements are arranged one after the other.

1. **Array** : A collection of elements of the same data type stored in contiguous memory locations.

Example: Storing marks of 5 students.

2. **Linked List**: A collection of nodes, where each node contains data and a reference (or link) to the next node in the sequence.

Example: Managing dynamic lists like playlists or to-do lists.

3. **Stack**: A collection of elements following the **Last In First Out (LIFO)** principle.

Example: Undo operation in a text editor.

4. **Queue**: A collection of elements following the **First In First Out (FIFO)** principle.

Example: Managing tasks in a printer queue.

2. Non-Linear Data Structures

Non-linear data structures allow hierarchical relationships among elements.

1. **Tree**: A hierarchical structure where each node is connected to children nodes. A tree starts with a root node.

Example: Representing file systems or organizational hierarchies.

2. **Graph**: A collection of nodes (vertices) connected by edges. Can be **directed** or **undirected**.

Example: Representing social networks or roadmaps.

3. **Hash Table**: A data structure that maps keys to values for efficient lookup using a hash function.

Example: Implementing dictionaries or caches.

Primitive Operations on Data Structures

1. Insertion : Adding a new element to the data structure.

- **Example:** Adding 5 to an array.

```
array = [1, 2, 3, 4]
```

```
array.append(5)
```

Output: [1, 2, 3, 4, 5]

2. Deletion : Removing an element from the data structure.

- **Example:** Deleting 3 from a linked list.

```
linked_list = [1, 2, 3, 4]
```

```
linked_list.remove(3)
```

Output: [1, 2, 4]

3. Traversal : Accessing each element of the data structure sequentially.

- **Example:** Traversing an array to print elements.

```
array = [1, 2, 3, 4]
```

```
for element in array:
```

```
    print(element)
```

Output: 1 2 3 4

4. Search : Finding a specific element in the data structure.

- **Example:** Searching for 3 in an array.

```
array = [1, 2, 3, 4]
```

```
print(3 in array)
```

Output: True

5. Sorting : Arranging elements in a specific order (ascending/descending).

- **Example:** Sorting an array in ascending order.

```
array = [4, 1, 3, 2]
```

```
array.sort()
```

Output: [1, 2, 3, 4]

6. Access : Retrieving the value of a specific element by its position or key.

- **Example:** Accessing the second element in an array.

```
array = [1, 2, 3, 4]
```

```
print(array[1])
```

Output: 2

1 c. Explain all operations of Data Structures.

1. Traversal

- **Description:** Traversal operations are used to visit each node in a data structure in a specific order.
- **Example** (Array Traversal):

```
array = [1, 2, 3, 4]
```

```
for element in array:
```

```
    print(element)
```

Output: 1 2 3 4

2. Insertion

- **Description:** Adding a new element to the data structure.
- **Example** (Insert into a List):

```
array = [1, 2, 3]
```

```
array.append(4) # Adds 4 to the end
```

```
print(array)
```

Output: [1, 2, 3, 4]

3. Deletion

- **Description:** Removing an element from the data structure.
- **Example** (Remove from a List):

```
array = [1, 2, 3, 4]
```

```
array.remove(3) # Removes 3
```

```
print(array)
```

Output: [1, 2, 4]

4. Search

- **Description:** Finding whether a specific element exists in the data structure.
- **Example** (Search in a List):

```
array = [1, 2, 3, 4]
```

```
if 3 in array:
```

```
    print("Found")
```

```
else:
```

```
    print("Not Found")
```

Output: Found

5. Sorting

- **Description:** Arranging elements in a specific order (ascending or descending).
- **Example** (Sort a List):

```
array = [4, 2, 3, 1]

array.sort() # Sorts in ascending order

print(array)
```

Output: [1, 2, 3, 4]

2. a. Explain any five string handling functions supported by 'c' with syntax and 10 example.

1. strlen

- **Description:** Determines the length of a string (excluding the null character \0).
- **Syntax:**

```
size_t strlen(const char *str);
```

- **Example:**

```
int main()
{
    char str[] = "Hello";
    printf("Length of the string: %lu\n", strlen(str));
    return 0;
}
```

Output: 5

2. strcpy

- **Description:** Copies the content of one string into another.
- **Syntax:**

```
char *strcpy(char *dest, const char *src);
```

- **Example:**

```
int main()
{
    char src[] = "World";
    char dest[20];
    strcpy(dest, src);
    printf("Copied string: %s\n", dest);
    return 0;
}
```

Output: World

3. strcat

- **Description:** Concatenates (joins) two strings.
- **Syntax:**

```
char *strcat(char *dest, const char *src);
```

- **Example:**

```
int main()
{
    char str1[20] = "Hello ";
    char str2[] = "World";
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

Output: Hello World

4. strcmp

- **Description:** Compares two strings lexicographically.
- **Syntax:**

```
int strcmp(const char *str1, const char *str2);
```

- **Example:**

```
int main()
{
    char str1[] = "Hello";
    char str2[] = "World";
    int result = strcmp(str1, str2);
    if (result == 0)
        printf("Strings are equal.\n");
    else if (result < 0)
        printf("str1 is less than str2.\n");
    else
        printf("str1 is greater than str2.\n");
    return 0;
}
```

Output: str1 is less than str2.

5. strrev (*Not part of the standard library, but widely available in compilers like Turbo C*)

- **Description:** Reverses a string.

- **Syntax:**

```
char *strrev(char *str);
```

- **Example:**

```
int main()
{
    char str[] = "Hello";
    printf("Original string: %s\n", str);
    printf("Reversed string: %s\n", strrev(str));
    return 0;
}
```

Output: olleH

2 b. Convert the following infix expression to postfix expression using stack: $A + (B * C - (D / E ^ F) G) * H$.

Expression : $A + (B * C - (D / E ^ F) * G) * H$

Symbol	Stack	Postfix
A		A
+	+	A
(+, (A
B	+, (AB
*	+, (, *	AB
C	+, (, *	ABC
-	+, (, -	ABC*
(+, (, -, (ABC*
D	+, (, -, (ABC*D
/	+, (, -, (, /	ABC*D
E	+, (, -, (, /	ABC*DE
^	+, (, -, (, /, ^	ABC*DE
F	+, (, -, (, /, ^	ABC*DEF
)	+, (, -, (, /, ^,)	ABC*DEF
	+, (, -, (, /, ^,))	ABC*DEF^/
*	+, (, -, *	ABC*DEF^/
G	+, (, -, *	ABC*DEF^/G
)	+, (, -, *))	ABC*DEF^/G*-
*	+, *	ABC*DEF^/G*-
H	+, *	ABC*DEF^/G*-H
Final Result		ABC*DEF^/G*-H*+

3a. List the disadvantages of linear queue and how is it solved in circular queues. Give the algorithm to insert and delete an element in circular queues.

Disadvantages of Linear Queue

1. Wasted Space:

- In a linear queue, when elements are dequeued, the freed space at the front of the queue cannot be reused, even though it is no longer occupied.

2. Queue Overflow:

- Even if there is unused space at the front, the queue may still show as full when the rear pointer reaches the maximum size of the array.

3. Inefficient Memory Utilization:

- The fixed size of the array in a linear queue can lead to underutilization of memory when elements are dequeued.

How Circular Queue Solves These Issues

1. Reusability of Space:

- In a circular queue, the rear pointer wraps around to the beginning of the array when it reaches the end, reusing the freed space at the front.

2. Efficient Memory Utilization:

- Circular queues use all available space efficiently by reusing the memory locations freed during deletions.

3. No Overflow Until Full:

- Overflow occurs only when all slots are occupied, regardless of the position of front and rear.

// Function to insert an item into circular queue	// Function to delete an element from queue
<pre>void insert_rear (int item) { // Check for overflow of queue if (count === Q_SIZE) { printf ("Queue Overflow"); return; } // Increment rear by 1 rear = (rear + 1) % Q_SIZE; // Insert an item into the queue queue[rear] = item; // Update count by 1 count++; }</pre>	<pre>void delete_front() { // Check for underflow of queue if (count == 0) { printf ("Queue Underflow"); return; } // Delete the item from circular queue printf ("Item deleted :%d", queue[front]); // Increment front by 1 front (front + 1)% Q_SIZE; // Update count by 1 count = count - 1; }</pre>

3b. Explain in detail about multiple queue with relevant functions in 'C'.

A **multiple queue** is a data structure where two or more queues are implemented within a single memory array. These queues may be used to manage multiple independent data streams or processes efficiently within shared storage.

Advantages of Multiple Queues

1. Efficient Memory Usage:

- Avoids allocating separate memory for each queue.

2. Concurrent Data Management:

- Useful in situations like CPU scheduling, where different processes have separate queues.

3. Flexibility:

- Dynamic sharing of memory allows better utilization in some designs.

Types of Multiple Queues

1. Fixed-Sized Multiple Queues:

- Each queue is allocated a fixed size within the array.

2. Dynamic Multiple Queues:

- Each queue can grow or shrink dynamically as needed, sharing the array space.
- In a dynamic implementation, additional logic is required to handle overlapping queues or reallocate memory when one queue grows beyond its segment size. This can be achieved using linked lists or a dynamic array.

Implementation of Fixed-Sized Multiple Queues in C

Structure for Multiple Queues

```
#define MAX 100

typedef struct
{
    int front;
    int rear;
    int start;
    int end;
} Queue;

typedef struct
{
    int data[MAX];
    Queue queues[2];
} MultiQueue;
```

Example Program

```
int main()
{
    MultiQueue mq;
    int numQueues = 2;
    int segmentSize = MAX / numQueues;

    initializeQueues(&mq, numQueues, segmentSize);

    // Queue 0 operations
    enqueue(&mq, 0, 10);
    enqueue(&mq, 0, 20);
    enqueue(&mq, 0, 30);
    printf("Queue 0: ");
    displayQueue(&mq, 0);

    // Queue 1 operations
    enqueue(&mq, 1, 100);
    enqueue(&mq, 1, 200);
    printf("Queue 1: ");
    displayQueue(&mq, 1);

    // Dequeue operations
    printf("Dequeue from Queue 0: %d\n", dequeue(&mq, 0));
    printf("Queue 0 after dequeue: ");
    displayQueue(&mq, 0);

    printf("Dequeue from Queue 1: %d\n", dequeue(&mq, 1));
    printf("Queue 1 after dequeue: ");
    displayQueue(&mq, 1);
    return 0;
}
```

4a. Develop a linked list with the basic operations performed on Singly Linked List (SLL) and different types of linked list.

A **Singly Linked List (SLL)** is a data structure where each element (node) points to the next node in the sequence. Each node contains two parts:

1. **Data:** Holds the actual data.
2. **Next:** Points to the next node in the list (or NULL if it is the last node).

Basic Operations on Singly Linked List

The basic operations on a singly linked list are:

1. **Insertion**
 - Insert at the beginning
 - Insert at the end
 - Insert after a given node
2. **Deletion**
 - Delete the first node
 - Delete the last node
 - Delete a node at a specific position
3. **Traversal**
 - Traverse the list to print all elements
4. **Search**
 - Search for an element in the list
5. **Reversal**
 - Reverse the list

Types of Linked Lists

There are three main types of linked lists:

1. **Singly Linked List (SLL):**
 - Each node points to the next node.
2. **Doubly Linked List (DLL):**
 - Each node contains two pointers: one points to the next node, and the other points to the previous node.
3. **Circular Linked List:**
 - A variation where the last node points back to the first node, forming a circle.

Insert on element at the front end of SLL	Insert on element at the rear end of SLL
<pre> NODE insert_front(int item, NODE first) { NODE temp; temp = getnode(); temp->info = item; temp->link = first; return temp; } </pre>	<pre> NODE insert_rear(int item, NODE first) { NODE temp; NODE cur; temp = getnode(); temp->info = item; temp->link = NULL; if (first == NULL) return temp; cur = first; while (cur->link != NULL) { cur = cur->link; } cur->link = temp; return first; } </pre>
Delete a node at the beginning of SLL.	Delete a node at the end of SLL.
<pre> NODE delete_front(NODE first) { NODE temp; if (first == NULL) { printf("List is empty cannot delete\n"); return NULL; } temp = first; temp = temp->link printf("Item deleted = %d\n",first->info); free(first); return temp; } </pre>	<pre> NODE delete_rear(NODE first) { NODE cur, prev; if (first == NULL) { printf("List is empty cannot delete\n"); return first; } if (first->link == NULL) { printf ("The item to deleted is %d\n",first->info); free(first); return NULL; } prev = NULL; cur = first; while(cur->link != NULL) { prev = cur; cur = cur->link; } printf("The item deleted is %d\n",cur->info); free(cur); prev->link = NULL; return first; } </pre>

4b. Examine a node structure for linked representation of polynomial. Explain algorithm to add two polynomial represented using linked list.

Linked Representation of Polynomials

A polynomial can be represented as a sequence of terms where each term has two parts:

1. **Coefficient:** The numerical value associated with the term.
2. **Exponent:** The power of the variable (usually represented as x).

In the linked list representation, each node of the linked list can store a term of the polynomial. The node structure might include:

- **Coefficient:** The numerical coefficient of the term.
- **Exponent:** The exponent of the term.
- **Next Pointer:** A pointer to the next term in the polynomial.

Node Structure for Polynomial Linked List

```
struct Node
```

```
{  
  
    int coefficient;  
  
    int exponent;  
  
    struct Node* next;  
  
};
```

The linked list representation of a polynomial is ordered by exponents in decreasing order. For example, the polynomial $5x^3 + 4x^2 + 3x + 2$ will be represented as:

Head -> [5, 3] -> [4, 2] -> [3, 1] -> [2, 0] -> NULL

Here:

- Node [5, 3] represents the term $5x^3$.
- Node [4, 2] represents the term $4x^2$.
- Node [3, 1] represents the term $3x$.
- Node [2, 0] represents the constant term 2.

Algorithm to Add Two Polynomials Using Linked List

The process to add two polynomials involves iterating through both polynomial linked lists, comparing the exponents, and combining terms with the same exponent. If the exponents are different, the term with the higher exponent is added to the result list.

Steps:

1. **Initialize two pointers:** One for each polynomial (poly1 and poly2).
2. **Iterate through the lists:**

- If the exponent of poly1 is greater than poly2, add the term of poly1 to the result list and move poly1 to the next node.
 - If the exponent of poly1 is smaller than poly2, add the term of poly2 to the result list and move poly2 to the next node.
 - If the exponents are equal, add the coefficients and insert the new term with the same exponent.
3. **Add remaining terms:** Once one list is fully traversed, append the remaining terms from the other list to the result list.
 4. **Return the resulting polynomial.**

// C function of add two polynomials

```
struct Node* addPolynomial(struct Node* head1, struct Node* head2)
{
    if (head1 == NULL) return head2;
    if (head2 == NULL) return head1;
    if (head1->pow > head2->pow)
    {
        struct Node* nextPtr = addPolynomial(head1->next, head2);
        head1->next = nextPtr;
        return head1;
    }
    else if (head1->pow < head2->pow)
    {
        struct Node* nextPtr = addPolynomial(head1, head2->next);
        head2->next = nextPtr;
        return head2;
    }
    struct Node* nextPtr = addPolynomial(head1->next, head2->next);
    head1->coeff += head2->coeff;
    head1->next = nextPtr;
    return head1;
}
```

5a. Summarize Sparse Matrix. For the given sparse matrix, write the diagrammatic linked list representation.

$$\begin{bmatrix} 8 & 0 & 0 & 0 \\ 5 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 8 \\ 0 & 0 & 9 & 1 \end{bmatrix}$$

To represent the given sparse matrix as a linked list diagrammatically, you need to create a representation where each non-zero element is stored along with its row and column indices. Here's how you can summarize it:

Linked List Representation:

Each node in the linked list contains:

- **Value:** The non-zero element.
- **Row index:** The row position of the element.
- **Column index:** The column position of the element.
- **Pointer:** A pointer to the next node.

For this matrix:

1. The non-zero elements are: 8, 5, 3, 4, 8, 9, 1.
2. Their respective positions are:
 - 8 at (0, 0)
 - 5 at (1, 0)
 - 3 at (1, 3)
 - 4 at (3, 0)
 - 8 at (3, 3)
 - 9 at (4, 2)
 - 1 at (4, 3)

Linked List Nodes:

Each node can be represented as:

[value, row, column] -> next_node

Diagram Representation:

[8, 0, 0] -> [5, 1, 0] -> [3, 1, 3] -> [4, 3, 0] -> [8, 3, 3] -> [9, 4, 2] -> [1, 4, 3] -> NULL

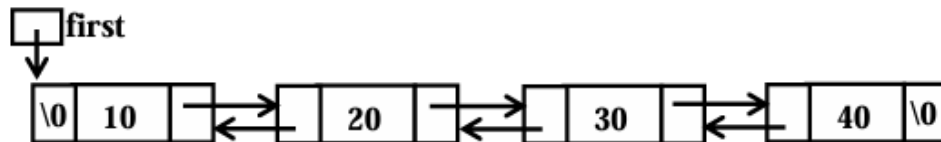
This linked list representation is sequentially connected, storing only non-zero elements, saving space compared to the full matrix. Let me know if you need further elaboration or a drawing for better visualization!

5b. Define Doubly linked list. Write the functions to perform the following operations on doubly linked list.

Definition: A doubly-linked list is a linear collection of nodes where each node is divided into three parts:

- info – This is a field where the information has to be stored
- llink – This is a pointer field which contains address of the left node or previous node in the list
- rlink – This is a pointer field which contains address of the right node or next node in the list

The pictorial representation of a doubly linked list is shown in figure below:



(i) Insert a node at rear end of the list

```
void insertAtRear(int data)
{
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    if (head == NULL)
    {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
```

(ii) Delete a node at rear end of the list

```
void deleteAtRear()
{
    if (head == NULL)
    {
        printf("List is empty.\n");
        return;
    }
    Node* temp = head;
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
}
```

```

        if (temp->prev != NULL)
        {
            temp->prev->next = NULL;
        }
        else
        {
            head = NULL;
        }
        free(temp);
    }

```

(iii) Search a node with a given key value

```

int search(int key)
{
    Node* temp = head;
    int position = 0;
    while (temp != NULL)
    {
        if (temp->data == key)
        {
            return position;
        }
        temp = temp->next;
        position++;
    }
    return -1;
}

```

6a. Define Tree with any six tree terminology.

A **Tree** is a hierarchical data structure consisting of nodes, where each node may have a parent and zero or more children. It is defined as a collection of nodes such that:

1. There is a distinguished node called the **root**.
2. Every other node is connected by an edge from exactly one parent node.
3. The structure has no cycles, making it a connected, acyclic graph.

Six Tree Terminologies

1. **Root:**

- The topmost node of the tree.
- It has no parent.
- Example: In a tree representing a family, the root could represent the oldest ancestor.

2. **Parent and Child:**

- A **parent** is a node with one or more children.
- A **child** is a node that descends from a parent node.
- Example: In a binary tree, a node can have up to two children.

3. Leaf:

- A **leaf** is a node that has no children.
- It represents the end of a branch in the tree.
- Example: In a file system tree, files (not folders) are typically leaf nodes.

4. Height of a Tree:

- The **height** of a tree is the length of the longest path from the root to a leaf.
- Example: If a tree has three levels, its height is 2 (considering 0-based indexing).

5. Degree of a Node:

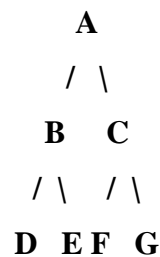
- The **degree** of a node is the number of children it has.
- Example: In a binary tree, the maximum degree of a node is 2.

6. Subtree:

- A **subtree** is any node in the tree along with its descendants.
- Example: If you remove a node and its children from the tree, that part forms a subtree.

Tree Diagram Example

Consider the following tree structure:



- **Root:** A
- **Parent:** A is the parent of B and C.
- **Children:** B and C are children of A.
- **Leaf:** D, E, F, G are leaf nodes.
- **Height:** The height of this tree is 2 (from A to the deepest leaf D, E, F, or G).
- **Subtree:** The tree rooted at B (with children D and E) is a subtree.

6 b. Write the function for copying and testing of binary tree.

- ♦ Copying a tree – copy one binary tree to other tree
- ♦ Test for equality – check whether two trees are equal or not

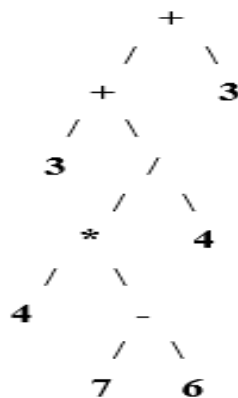
// Function to get the exact copy of a tree	// Function to get the exact copy of a tree
<pre>NODE copy(NODE root) { NODE temp; if (root == NULL) return NULL; temp = getnode(); temp->info = root->info; temp->lptr = copy(root->lptr); temp->rptr = copy(root->rptr); return temp; }</pre>	<pre>int equal (NODE r1, NODE r2) { if (r1 == NULL && r2 == NULL) return 1; if (r1 == NULL && r2 != NULL) return 0; if (r1 != NULL && r2 == NULL) return 0; if (r1->info != r2->info) return 0; if (r1->info == r2->info) return 1; return equal(r1->llink, r2->llink) && equal(r1->rlink, r2->rlink); }</pre>

6 C. Draw a binary tree and find out the binary tree traversals for the following expression

3+4*(7-6)/4+3.

In-Order Traversal (Left, Root, Right):

The in-order traversal produces the original expression.



Let us traverse the tree in preorder and postorder as shown below:

Preorder traversal

Postorder traversal

<p>$- 7 6 = A$</p>	<p>$7 6 - = A$</p>
<p>$* 4 A = B$</p>	<p>$4 A * = B$</p>
<p>$/ B 4 = C$</p>	<p>$B 4 / = C$</p>
<p>$+ 3 C = D$</p>	<p>$3 C + = D$</p>
<p>$+ D 3 = E$</p>	<p>$D 3 + = E$</p>

E
 + D 3
 ++ 3 C 3
 ++ 3 / B 4 3
 ++ 3 / * 4 A 4 3
 ++ 3 / * 4 - 7 6 4 3
 (Prefix Expression)

E
 D 3 +
 3 C + 3 +
 3 B 4 / + 3 +
 3 4 A * 4 / + 3 +
 3 4 7 6 - * 4 / + 3 +
 (Postfix Expression)

7 a.. Construct binary search tree for the given set of values 14, 15, 4, 9, 7, 18, 3, 5, 16, 20. Also perform inroder, preorder and post order traversals of the obtained tree.

Constructing the Binary Search Tree (BST)

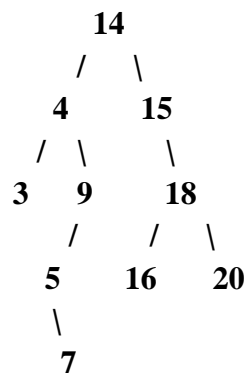
Rules for BST construction:

- The left subtree of a node contains only nodes with values less than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.

Steps to Construct the Tree

1. Start with the first value (14) as the root.
2. Insert each value one by one, comparing it with the current node and moving left or right accordingly.

Values to insert: 14, 15, 4, 9, 7, 18, 3, 5, 16, 20



Now, let's perform the traversals:

Inorder Traversal (Left, Root, Right):

- Traverse the left subtree.
- Visit the root node.
- Traverse the right subtree.

Inorder Traversal Output: 3, 4, 5, 7, 9, 14, 15, 16, 18, 20

Preorder Traversal (Root, Left, Right):

- Visit the root node.
- Traverse the left subtree.
- Traverse the right subtree.

Preorder Traversal Output: 14, 4, 3, 9, 5, 7, 15, 18, 16, 20

Postorder Traversal (Left, Right, Root):

- Traverse the left subtree.
- Traverse the right subtree.
- Visit the root node.

Postorder Traversal Output: 3, 7, 5, 9, 4, 16, 20, 18, 15, 14

7 b. Build a linked list representation of disjoint sets in detail.

To build a **linked list representation of disjoint sets**, we use a method called **linked list-based disjoint-set representation**. In this representation, each set is implemented as a linked list, and the head of the list serves as the representative of the set.

Key Concepts in Linked List Representation of Disjoint Sets

1. Node Structure:

- Each node stores:
 - The element.
 - A pointer to the next element in the linked list.
 - A pointer to the head of the linked list (for quick access to the set representative).

2. Set Operations:

- **Make-Set(x)**: Create a new set with a single element x.
- **Find(x)**: Find the representative (head) of the set containing x.
- **Union(x, y)**: Merge two sets containing x and y.

Implementation Steps

1. Node Structure

Each node in the linked list can be represented as:

1. value: The element's value.
2. next: A pointer to the next node in the list.
3. head: A pointer to the head of the list, representing the set's representative.

2. Disjoint Set Operations

1. Make-Set(x):

- Create a new linked list containing a single node.
- The head pointer of the node points to itself.

2. Find(x):

- Return the head pointer of the node.

3. Union(x, y):

- Combine the two linked lists of the sets containing x and y.
- Append the second list to the end of the first list and update the head pointer of all nodes in the second list.

Characteristics and Limitations

1. Advantages:

- Simple implementation.
- Direct representation of disjoint sets.

2. Disadvantages:

- **Union operation is expensive** because updating the head pointer for all nodes in the second list takes $O(n)$ time.
- Does not use the path compression optimization.

7 c. Simplify recursive search algorithm for a binary search tree.

A recursive search algorithm for a binary search tree (BST) can be simplified by directly utilizing the binary search tree properties:

- If the value matches the root, return the node.
- If the value is smaller, recursively search in the left subtree.
- If the value is larger, recursively search in the right subtree.

Here's the simplified recursive search algorithm:

Implementation in C

```
// Recursive function to search in the BST
Node* searchBST(Node* root, int value)
{
    if (root == NULL || root->value == value)
    {
        return root;
    }
    if (value < root->value)
    {
        return searchBST(root->left, value);
    }
    return searchBST(root->right, value);
}
```

Explanation of the Code

1. Base Case:

- If the current node (root) is NULL, the value is not found.
- If the value matches the current node's value, the node is returned.

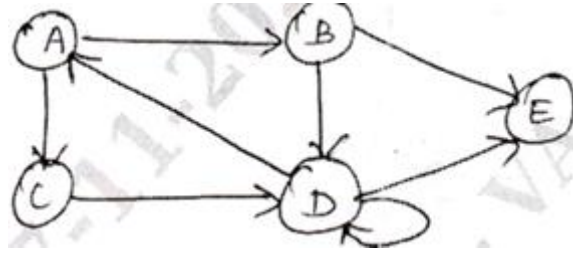
2. Recursive Steps:

- If the value is smaller than the current node's value, the search continues in the left subtree.
- If the value is larger, the search continues in the right subtree.

Advantages of This Algorithm

1. It is simple and concise.
2. Takes advantage of the BST properties for efficient searching ($O(\log n)$ for balanced trees).

8 a. Compare a graph with tree. For the graph shown in Fig.Q8(a), show the adjacency matrix and adjacency list representation.



The given graph represents a directed graph. To solve the problem, let's compute both the **adjacency matrix** and the **adjacency list representation** for the graph.

Adjacency Matrix

The adjacency matrix is a 2D matrix where each entry $M[i][j]$ represents whether there is an edge from vertex i to vertex j (1 if there is an edge, 0 otherwise).

Adjacency List

The adjacency list representation associates each vertex with a list of its direct neighbors (vertices it has outgoing edges to).

Let's first analyze the graph:

Vertices: A,B,C,D,E

Edges (from the diagram):

- $A \rightarrow B, A \rightarrow C, A \rightarrow D$
- $B \rightarrow E$
- $C \rightarrow D$
- $D \rightarrow E, D \rightarrow D$ (self-loop)

Now I'll compute the adjacency matrix and adjacency list:

Adjacency Matrix:

	A	B	C	D	E
A	[0, 1, 1, 1, 0]				
B	[0, 0, 0, 0, 1]				
C	[0, 0, 0, 1, 0]				
D	[0, 0, 0, 1, 1]				
E	[0, 0, 0, 0, 0]				

Adjacency List:

A: B, C, D
B: E
C: D
D: D, E
E: (no neighbors)

8.b. Explain all methods used for traversing a graph with suitable example and write 'C' function for the same.

Graph Traversal Methods

Graph traversal refers to the process of visiting each vertex and edge in a graph. There are two primary methods to traverse a graph:

1. Depth-First Search (DFS)

Description:

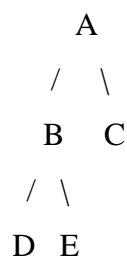
- DFS explores as far as possible along each branch before backtracking. It uses a stack (explicitly or via recursion) to keep track of the vertices to visit.

Steps:

1. Start at a source vertex.
2. Mark it as visited.
3. Recursively visit all unvisited neighbors.
4. Backtrack when there are no more unvisited neighbors.

Example:

Graph:



Traversal (starting from A):

A -> B -> D -> E -> C

2. Breadth-First Search (BFS)

Description:

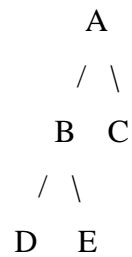
- BFS explores all the neighbors of a vertex before moving to the next level of vertices. It uses a queue to keep track of the vertices to visit.

Steps:

1. Start at a source vertex.
2. Mark it as visited and enqueue it.
3. Dequeue a vertex, process it, and enqueue all its unvisited neighbors.
4. Repeat until the queue is empty.

Example:

Graph:



Traversal (starting from A):

A -> B -> C -> D -> E

C Functions for DFS and BFS

Here's the code implementation in C:

1. Depth-First Search (DFS)

```
void DFS(int graph[MAX][MAX], int visited[], int vertex, int n)
{
    printf("%c ", vertex + 'A');
    visited[vertex] = 1;

    for (int i = 0; i < n; i++)
    {
        if (graph[vertex][i] == 1 && !visited[i])
        {
            DFS(graph, visited, i, n);
        }
    }
}
```

2. Breadth-First Search (BFS)

```
void BFS(int graph[MAX][MAX], int n, int startVertex)
{
    int queue[MAX], front = 0, rear = 0;
    int visited[MAX] = {0};

    queue[rear++] = startVertex;
    visited[startVertex] = 1;

    while (front < rear)
    {
        int vertex = queue[front++];
        printf("%c ", vertex + 'A');

        for (int i = 0; i < n; i++)
        {
            if (graph[vertex][i] == 1 && !visited[i])
            {
                queue[rear++] = i;
                visited[i] = 1;
            }
        }
    }
}
```

9. a. Differentiate between static hashing and dynamic hashing in detail with operations.

1. Static Hashing

Definition:

In static hashing, the hash table has a fixed size. Once the table is created, the number of buckets remains constant throughout its usage.

Characteristics:

1. **Fixed Table Size:** The number of buckets is predefined and does not change even if data grows beyond capacity.
2. **Overflow Handling:** Collisions and overflow are managed using techniques like **chaining** (linked lists) or **overflow buckets**.
3. **Efficiency:** Works well when the size of the data is known and does not change significantly.
4. **Memory Usage:** Can waste memory if the hash table is underutilized or cause performance degradation if it's overfull.

5. **Collisions:** More frequent in static hashing as data grows.

Operations:

1. **Insertion:**

- Compute the hash value of the key using a hash function.
- Place the key-value pair in the corresponding bucket.
- If the bucket is full, manage overflow using chaining or other methods.

2. **Search:**

- Compute the hash value of the key.
- Search within the bucket to find the value.

3. **Deletion:**

- Compute the hash value of the key.
- Remove the key-value pair from the bucket.

Example:

- Hash Table with 5 buckets (bucket[0] to bucket[4]).
- Hash function: $h(\text{key}) = \text{key} \bmod 5$.

For keys 12, 22, 32:

- **Insertion:** All map to bucket[2] causing overflow, managed via chaining.

2. Dynamic Hashing

Definition:

In dynamic hashing, the size of the hash table grows or shrinks dynamically based on the data. It uses a directory structure to map keys to buckets.

Characteristics:

1. **Variable Table Size:** Buckets are split or merged dynamically as data increases or decreases.
2. **Collision Reduction:** Reduces collisions as new buckets are created when required.
3. **Scalable:** Ideal for applications where data size is unpredictable.
4. **Memory Usage:** Optimized as the structure adapts to data requirements.
5. **Uses Directory:** A directory level manages the mapping of keys to buckets, and it grows as buckets are split.

Operations:

1. **Insertion:**

- Compute the hash value of the key using the hash function.
- Place the key-value pair in the corresponding bucket.
- If the bucket overflows, **split the bucket** and update the directory.

2. **Search:**

- Compute the hash value of the key.
- Use the directory to find the corresponding bucket and retrieve the value.

3. Deletion:

- Compute the hash value of the key.
- Remove the key-value pair from the bucket.
- Optionally merge buckets if utilization falls below a threshold.

Example:

- Initial directory has 2 buckets (size doubles dynamically).
- Insert keys: 5, 15, 25 (hash function $h(\text{key}) = \text{key} \bmod 4$).
- After splitting due to overflow:
 - Directory grows to accommodate more buckets.
 - Keys are redistributed based on new hash values.

Comparison Table

Aspect	Static Hashing	Dynamic Hashing
Table Size	Fixed. Defined during initialization.	Dynamic. Adjusts as data grows or shrinks.
Overflow Handling	Managed via chaining or overflow buckets.	Handled by splitting buckets dynamically.
Memory Usage	Can waste memory if underutilized or degrade performance when overfilled.	Efficient as it adjusts to the data.
Scalability	Limited. Not suitable for unpredictable or growing datasets.	Highly scalable for dynamic or unpredictable datasets.
Collision Frequency	Higher, especially as data grows.	Lower, as buckets split dynamically.
Ease of Implementation	Simpler to implement and understand.	Complex due to directory management and bucket splitting logic.
Use Case	Suitable for static or small datasets where size is predictable.	Suitable for dynamic or large datasets with unpredictable size.

9 b. Describe double ended priority queue.

A **Double-Ended Priority Queue (DEPQ)**, also known as a **Two-ended Priority Queue**, is a specialized data structure that supports the following operations:

1. **Insert:** Insert an element into the queue.
2. **Delete-Min:** Remove and return the smallest element in the queue.
3. **Delete-Max:** Remove and return the largest element in the queue.
4. **Peek-Min:** Return the smallest element without removing it.
5. **Peek-Max:** Return the largest element without removing it.

Key Characteristics:

- Two Ends
- Efficient Operations

Operations in a Double-Ended Priority Queue (DEPQ)

1. Insert
2. Delete-Min
3. Delete-Max
4. Peek-Min
5. Peek-Max

Implementation Details

A typical implementation of a **Double-Ended Priority Queue** can use two **heaps** (min-heap and max-heap).

This structure allows both ends to be efficiently accessed:

1. Min-Heap
2. Max-Heap

Advantages of Double-Ended Priority Queue (DEPQ):

- Efficient
- Versatile

Applications:

- Scheduling Systems
- Data Streams
- Game Engines

9 c. Explain Hashing with any three Hash functions.

Hashing is a technique used to map data (keys) to fixed-size values, typically integers, called **hash values** or **hash codes**. This mapping is performed by a **hash function**.

Different Hashing Functions

A **hash function** takes an input (key) and produces a fixed-size output (hash value). There are different types of hash functions depending on the requirements, such as:

1. Division Method:

- The simplest and most commonly used method.
- The hash value is computed by taking the modulus of the key with a prime number m (size of the table).

Hash function:

$$h(k) = k \bmod m$$

where k is the key, and m is the size of the table (preferably a prime number).

Example:

- Given a table size of 10, hash function $h(k) = k \bmod 10$
- For key 15, the hash value is $h(15) = 15 \bmod 10 = 5$

2. Folding Method:

- In this method, the key is divided into several parts, which are then added together to produce the hash value.

Hash function:

- Split the key into equal-sized parts, then sum the parts and take modulo mm.

Example:

- Given key 123456, split it into parts: 12, 34, 56.
- Sum the parts: $12+34+56=102$.
- Take $102 \bmod 10=2$.
- The hash value is 2.

3. Mid Square Method:

- In this method, the key is squared, and the middle digits of the result are extracted as the hash value.

Hash function:

$h(k)=\text{middle digits of } k^2$

Example:

- For key 23, square it: $23^2=529$.
- Extract the middle digit(s): 5.
- The hash value is 5.

10 a. What is collision? Explain the method to resolve collision with suitable algorithm of linear probing. Insert keys 72, 27, 36, 24, 63, 81, 92, 101 into % [size 10].

A **collision** in hashing occurs when two or more keys map to the same index in a hash table. Since each index in a hash table can only store one element, when two keys hash to the same index, a **collision** happens. This can occur when the hash function produces the same hash value for different keys.

Collision Resolution Methods

There are several methods to resolve collisions in hash tables, including:

1. **Chaining:** Store multiple elements at the same hash index using a linked list or another dynamic data structure.
2. **Open Addressing:** All elements are stored within the hash table itself. When a collision occurs, the algorithm searches for the next available slot based on a specific probing technique. Some common open addressing methods include:
 - **Linear Probing**
 - **Quadratic Probing**
 - **Double Hashing**

Linear Probing for Collision Resolution

In **Linear Probing**, when a collision occurs at a given index, the algorithm checks the next index (i.e., index + 1) in the hash table. If that index is occupied, it checks the next one (index + 2), and so on, until an empty slot is found. This technique is simple but suffers from **primary clustering**, where groups of consecutive occupied slots form, leading to performance degradation.

Linear Probing Algorithm

1. Compute the hash index for the key.
2. If the slot at the computed index is empty, insert the key there.
3. If the slot is occupied, move to the next slot (index + 1).
4. Repeat the process until an empty slot is found.

Example: Insert keys into a hash table using Linear Probing

Let's create a hash table of size 10 and insert the following keys:

72, 27, 36, 24, 63, 81, 92, 101

Hash Function

$$h(\text{key}) = \text{key} \% 10$$

This will produce a hash value between 0 and 9 (since the table size is 10).

Step-by-Step Insertion with Linear Probing

1. **Insert key 72:**

$$h(72) = 72 \% 10 = 2$$

Slot 2 is empty, so insert **72** at index 2.

2. **Insert key 27:**

$$h(27) = 27 \% 10 = 7$$

Slot 7 is empty, so insert **27** at index 7.

3. **Insert key 36:**

$$h(36) = 36 \% 10 = 6$$

Slot 6 is empty, so insert **36** at index 6.

4. **Insert key 24:**

$$h(24) = 24 \% 10 = 4$$

Slot 4 is empty, so insert **24** at index 4.

5. **Insert key 63:**

$$h(63)=63\%10=3$$

Slot 3 is empty, so insert **63** at index 3.

6. **Insert key 81:**

$$h(81)=81\%10=1$$

Slot 1 is empty, so insert **81** at index 1.

7. **Insert key 92:**

$$h(92)=92\%10=2$$

Slot 2 is occupied (by **72**), so use linear probing. Check the next index (index 3):

- Slot 3 is occupied (by **63**), so move to index 4.
- Slot 4 is occupied (by **24**), so move to index 5.
- Slot 5 is empty, so insert **92** at index 5.

8. **Insert key 101:**

$$h(101)=101\%10=1$$

Slot 1 is occupied (by **81**), so use linear probing. Check the next index (index 2):

Final Hash Table

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

Index	Value
0	Empty
1	81
2	72
3	63
4	24
5	92
6	36
7	27
8	101
9	Empty

10 b. Construct an optimal binary search tree for the following keys with the probabilities as

Keys	A	B	C	D	E
Probability	0.25	0.2	0.05	0.2	0.3

To construct an **Optimal Binary Search Tree (OBST)** for the given keys and probabilities, we will use the **dynamic programming approach**.

Given:

- **Keys:** A, B, C, D, E
- **Probabilities:**
 - $P(A) = 0.25$
 - $P(B) = 0.20$
 - $P(C) = 0.05$
 - $P(D) = 0.20$
 - $P(E) = 0.30$

Objective:

We want to find an optimal binary search tree such that the expected search cost is minimized.

Steps:

1. Define Variables:

- $C[i][j]$: Minimum cost of constructing a binary search tree from keys i to j .
- $W[i][j]$: Sum of probabilities from keys i to j , including the probabilities of the keys themselves.
- **Key Indexes:** A = 1, B = 2, C = 3, D = 4, E = 5.

2. Initialize the Probability Table: For each pair of keys i and j , calculate $W[i][j]$, the sum of probabilities for keys i to j :

$$W[i][j] = P[i] + P[i+1] + \dots + P[j]$$

The table of probabilities looks like this:

i/j	1 (A)	2 (B)	3 (C)	4 (D)	5 (E)
1 (A)	0.25	0.45	0.50	0.70	1.00
2 (B)		0.20	0.25	0.45	0.75
3 (C)			0.05	0.25	0.55
4 (D)				0.20	0.50
5 (E)					0.30

3. Initialize the Cost Table: For a single key (when $i=j$), the cost is simply the probability of the key:

$$C[i][i] = P[i]$$

The table of costs for individual keys:

i/j	1 (A)	2 (B)	3 (C)	4 (D)	5 (E)
1 (A)	0.25				
2 (B)		0.20			
3 (C)			0.05		
4 (D)				0.20	
5 (E)					0.30

4. **Fill the Table Using Dynamic Programming:** We now fill the table for $C[i][j]$, the minimum cost for constructing a binary search tree from keys i to j , using the recurrence relation:

$$C[i][j] = \min_{r=i}^j (C[i][r-1] + C[r+1][j] + W[i][j])$$

where r is the root of the tree for the subarray from i to j .

For example, for $C[1][2]$, we compute the cost for both possible roots (A or B), and similarly for all other ranges of keys.

Calculation:

Now let's go step-by-step to calculate the values of $C[i][j]$ for increasing subarray sizes.

Cost Table Calculations:

1. For $C[1][2]$:

$$C[1][2] = \min (C[1][1] + C[2][2] + W[1][2])$$

$$C[1][2] = \min (0.25 + 0.20 + 0.45) = 0.90$$

2. For $C[2][3]$:

$$C[2][3] = \min (C[2][2] + C[3][3] + W[2][3])$$

$$C[2][3] = \min (0.20 + 0.05 + 0.25) = 0.50$$

3. For $C[3][4]$:

$$C[3][4] = \min (C[3][3] + C[4][4] + W[3][4])$$

$$C[3][4] = \min (0.05 + 0.20 + 0.25) = 0.50$$

4. For $C[4][5]$:

$$C[4][5] = \min (C[4][4] + C[5][5] + W[4][5])$$

$$C[4][5] = \min (0.20 + 0.30 + 0.50) = 1.00$$

Now we continue this for larger ranges (for example, $C[1][3]$, $C[2][4]$, etc.), and continue applying the recurrence relation.

Finally, the optimal binary search tree will be built by choosing the root that minimizes the cost at each stage.