CBCS SCHEME

USN

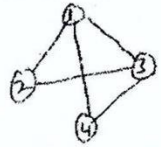
BCS304

Third Semester B.E./B.Tech. Degree Examination, Dec.2025/Jan.2026
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 Structure. Explain with neat diagram different types of data structure with examples. What are the primitive operations that can be performed?	10	L2	CO1
	b.	Define structure and union? Explain how they are different from each other, with suitable example.	5	L2	CO1
	c.	What do you mean by pattern matching? Outline Kruth, pattern matching algorithm.	5	L2	CO1
		OR			
Q.2	a.	Define stack. Give the implementation of push ( ) POP ( ) and Display ( ) functions by considering its empty and full conditions.	7	L2	CO1
	b.	Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 6, 2, 13, -4, 2, ×, +	7	L3	CO1
	c.	Write the postfix form of the following using stack, (i) $A*(B*C+D*E)+F$ (ii) $(A+(B*C)/(D-E))$	6	L3	CO1
		Module – 2			
Q.3	a.	What are the disadvantages of ordinary queue? Discuss the implementation of circular queue.	8	L2	CO2
	b.	Write a note on multiple stacks and priority queue.	6	L2	CO2
	c.	Define Queue. Discuss how to represent Queue using dynamic arrays.	6	L2	CO2
		OR			
Q.4	a.	What are Linked list? Explain the different types of Linked List with neat diagram.	4	L2	CO2
	b.	Give the structure definition for Singly Linked List (SSL). Write a C function to, (i) Insert an element at the end of SSL. (ii) Delete at node at the end of SSL.	8	L3	CO2

	c.	Write a C function to add two polynomials show the Linked List representation of below two polynomials $p(x)=3x^{14}+2x^7+1$ $q(x)=8x^{14}+5x^5+3x^2+2$	8	L3	CO2
		Module – 3			
Q.5	a.	Write a C-function for the following operation on doubly Linked List (DLL) : (i) Addition of a DLL node. (ii) Concatenation of two DLL.	8	L3	CO3
	b.	Write a C-function for the following operations on circular Linked List (i) Inserting at the front of a List. (ii) Find the number of nodes in circular list.	8	L3	CO3
	c.	Represent the given Sparse matrix using linked list representation. $A=\begin{bmatrix}2&0&0&0\\4&0&0&3\\0&0&0&0\\7&0&1&1\\0&0&6&0\end{bmatrix}$	4	L3	CO3
		OR			
Q.6	a.	Explain the different types binary tree representation with example.	8	L3	CO3
	b.	Define Threaded Binary tree. Discuss in threaded binary tree.	4	L3	CO3
	c.	Discuss Inorder, preorder, postorder and level order traversal with suitable recursive function for each.	8	L2	CO3
		Module – 4			
Q.7	a.	Write a function to perform the following operations on Binary Search Tree (BST) : (i) Inserting on element into BST. (ii) Recursive search of a BST.	8	L3	CO4
	b.	Discuss selection Trees with suitable example.	8	L2	CO4
	c.	Explain transforming a forest into a binary tree with an example.	4	L2	CO4
		OR			
Q.8	a.	Define graph. Show the adjacency matrix and adjacency. List representation of the graph given below.  Fig. Q8 (a)	6	L3	CO4

	b.	Define the following Terminologies with examples : (i) Vertex (node) (ii) Self loop (iii) Weighted graph (iv) Parallel edges	7	L1	CO4
	c.	Explain in detail elementary graph operations.	7	L1	CO4
		Module – 5			
Q.9	a.	What is collision? What are the methods to resolve collision? Explain linear probing with example.	8	L2	CO5
	b.	Explain in details about static and dynamic hashing.	6	L2	CO5
	c.	Discuss Leftist Trees with an example.	6	L2	CO5
		OR			
Q.10	a.	Explain different types of HASH functions with example.	6	L2	CO5
	b.	Discuss different types of rotations with suitable examples.	6	L3	CO5
	c.	Define Red-Black Tree, Splay tree. Discuss the method to insert an element into Red-Black tree.	8	L2	CO5

*****

VTU Jan 2026 : Solution

Sub:	Data Structures and Applications					Sub Code:	BCS304	Branch:	AIML
Date:		Duration:	3 hrs	Max Marks:	100	Sem / Sec:	III (A, B)		**OBE**

	Answer Key	**MARKS**	**L**	**C**
1.a.	~~Wha~~ Data Structure. Explain with neat diagram different types of data ıcture with examples. What are the primitive operations that can be formed?	**10**	**L2**	**CO1**

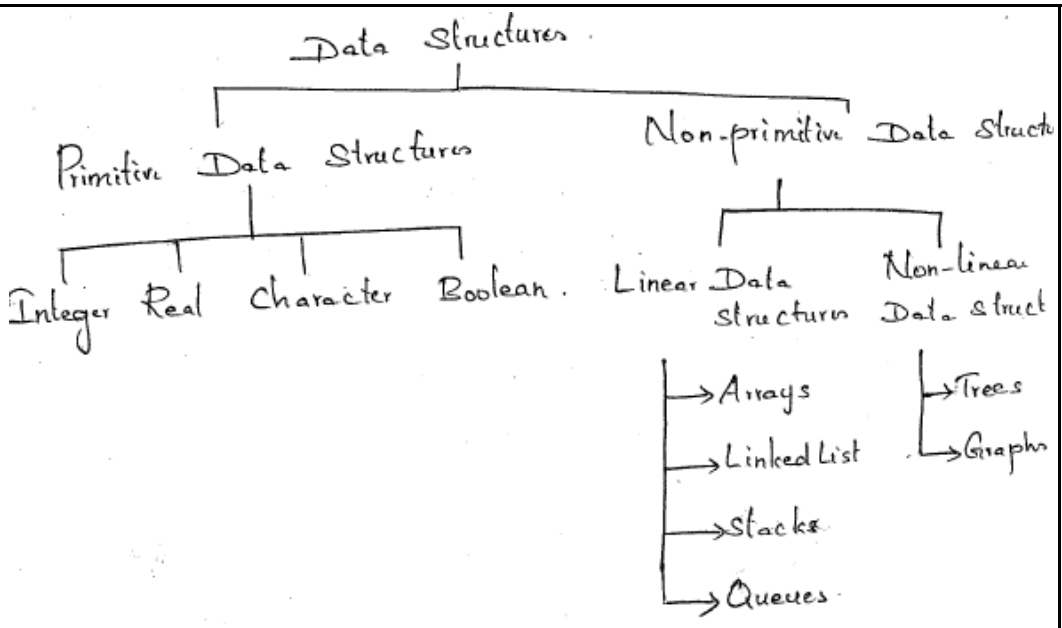
Data Structure:

Data structure is a collection of set of elements and various operations that can be performed on these elements.

Classification of Data Structures:

Data structures are classified as

1) Primitive Data structure.
2) Non-primitive Data Structures.



Data Structure Operations:

The data appearing in the data structures are processed by means of certain operations.

The operations are.

1. Traversing.
2. Searching.
3. Inserting.
4. Deleting.

Other operations : 1) Sorting.
2) Merging.

1.b	Define structure and union? Explain how they are different from each other, with suitable example.	**5**	**L2**	**CO1**

Group of data that permits the data of various types.

* keyword used struct.

Definition:

A structure is a collection of dataitem where each item is identified as to its type and name.

Difference Between Structure and Union

STRUCTURE	UNION
1) Keyword struct is used to define structure	1) Keyword Union is used to define a union
2) When a variable is associated with a structure the compiler allocates memory to each of the variable.	2) When a variable associated with a union the compiler allocates memory by considering the size of the largest memory.
3) The size of structure will be greater than or equal to the sum of sizes of its members	3) The size of union will be equal to size of largest member
4) Each member with in a structure is assigned unique storage area of locations	4) Memory allocated is shared by individual members.
5) The address of each member will be in ascending order. This indicates that memory for each member will start at different offset values.	6) The address is same for all the members of a union. every member begins at different offset value.
6) Individual member can be accessed at a time	6) Only one member can be accessed

1.c	What do you mean by pattern matching? Outline Kruth, pattern matching algorithm.	5	L2	CO1
2a	Define stack. Give the implementation of push ( ) POP ( ) and Display ( ) functions by considering its empty and full conditions.	7	L2	CO1

Definition: If $p = p_0 p_1 \cdots p_{n-1}$ is a pattern, then its *failure function*, f, is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1 \cdots p_i = p_{j-i} p_{j-i+2} \cdots p_j & \text{if such an } i \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases} \quad \square$$

```
int pmatch(char *string, char *pat)
{
/* Knuth, Morris, Pratt string matching algorithm */
  int i = 0, j = 0;
  int lens = strlen(string);
  int lenp = strlen(pat);
  while ( i < lens && j < lenp ) {
    if (string[i] == pat[j]) {
      i++; j++; }
    else if (j == 0) i++;
        else j = failure[j-1]+1;
  }
  return ( (j == lenp) ? (i-lenp) : -1);
}
```

Definition:

A stack is an ordered list in which insertion and deletions (push and pop) are made at one end called the top.

* Stack is also known as Last-In-First-Out (LIFO) since, last element inserted into a stack is the first element to be removed.

Full condition:

top >= MaxSize - 1

Empty condition:

top = -1

ADT stack:

ADT *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for **all** *stack* ∈ *Stack*, *item* ∈ *element*, *maxStackSize* ∈ positive integer
    Stack CreateS(*maxStackSize*) ::=
                create an empty stack whose maximum size is *maxStackSize*
    Boolean IsFull(*stack*, *maxStackSize*) ::=
                if (number of elements in *stack* == *maxStackSize*)
                return *TRUE*
                else return *FALSE*
    Stack Push(*stack*, *item*) ::=
                if (IsFull(*stack*)) *stackFull*
                else insert *item* into top of *stack* and **return**
    Boolean IsEmpty(*stack*) ::=
                if (*stack* == CreateS(*maxStackSize*))
                return *TRUE*
                else return *FALSE*
    Element Pop(*stack*) ::=
                if (IsEmpty(*stack*)) **return**
                else remove and return the element at the top of the stack.

ADT 3.1: Abstract data type *Stack*

```
void push(int item)
{
  if (top >= (Max-size - 1))
     printf("stack is full");
  else
     {
       stack[++top] = item;
       top = top + 1;
       stack[top].key = item;
     }
}
```

```
element pop()
{
  if (top == -1)
     printf("empty stack");
  else
     return stack[top--].key;
}
```

2b	Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 6, 2, 13, -4, 2,×,+ **Wrong Question.**	**7**	**L3**	**CO1**
2c	Write the postfix form of the following using stack, (i) $A*(B*C+D*E)+F$ (ii) $(A+(B*C)/(D-E))$	**7**	**L3**	**CO1**

Symbol	Stack	Postfix
A		A
*	*	A
(	* (	A
B	* (	AB
*	* (*	AB
C	* (*	ABC
+	* (+	ABC*
D	* (+	ABC*D
*	* (+ *	ABC*D
E	* (+ *	ABC*DE
)	*	ABCDE+
+	+	ABCDE+*
F	+	ABCDE+*F
(end)		ABCDE+*F+

——----------

Symbol	Stack	Postfix
A		A
+	+	A
(	+ (	A
B	+ (	AB
*	+ (*	AB
C	+ (*	ABC
)	+	ABC*
/	+ /	ABC*
(	+ / (	ABC*
D	+ / (	ABC*D
-	+ / (-	ABC*D
E	+ / (-	ABC*DE
)	+ /	ABC*DE-
(end)		ABC*DE-/+

3a	What are the disadvantages of ordinary queue? Discuss the implementation of circular queue.	**8**	**L2**	**CO2**

Disadvantages of Ordinary (Linear) Queue

1. **Wastage of Memory**
    - In an array implementation, when elements are deleted from the front, the freed space cannot be reused.
    - This causes unused spaces at the beginning of the array.
2. **False Overflow Condition**
    - Even if empty spaces exist at the front, the queue reports overflow when the rear reaches the end of the array.

3. **Inefficient Utilization**

- Requires shifting elements to reuse space, which increases time complexity.

```
void addq(int front, int *rear, element item)
{
/* add an item to the queue */
   *rear = (*rear+1) % MAX_QUEUE_SIZE;
   if (front == *rear) {
      queue_full(rear); /* reset rear and print error*/
      return;
   }
   queue[*rear] = item;
}
```

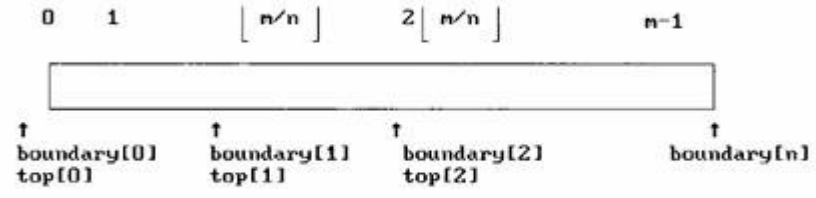
```
element deleteq(int *front, int rear)
{
   element item;
   /* remove front element from the queue and put it in
  item */
      if (*front == rear)
         return queue_empty(); /* queue_empty returns an
        error key */
      *front = (*front+1) % MAX_QUEUE_SIZE;
      return queue[*front];
}
```

3.b Write a note on multiple stacks and priority queue. **6** **L2** **CO2**

```
#define MEMORY_SIZE 100 /* size of memory */
#define MAX_STACKS 10 /* max number of stacks plus 1 */
/* global memory declaration */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int boundary[MAX_STACKS];
int n;  /* number of stacks entered by the user */
```

To divide the array into roughly equal segments we use the following code:

```
top[0] = boundary[0] = -1;
for (i = 1; i < n;i++)
   top[i] = boundary[i] = (MEMORY_SIZE/n)*i;
boundary[n] = MEMORY_SIZE-1;
```



All stacks are empty and divided into roughly equal segments.

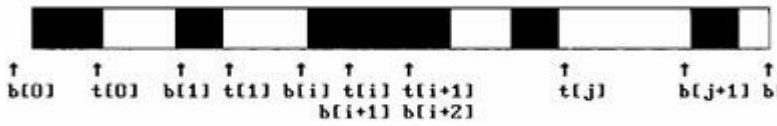
Figure 3.18: Initial configuration for *n* stacks in *memory* [*m*].

```
void add(int i, element item)
{
/* add an item to the ith stack */
   if (top[i] == boundary[i+1])
      stack_full(i);
   memory[++top[i]] = item;
}
```

Program 3.12: Add an *item* to the stack *stack_no*

```
element delete(int i)
{
/* remove top element from the ith stack */
   if (top[i] == boundary[i])
      return stack_empty(i);
   return memory[top[i]--];
}
```

Program 3.13: Delete an *item* from the stack *stack_no*



b = boundary, t = top

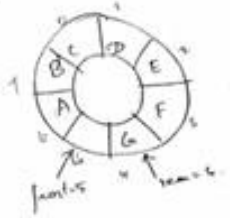
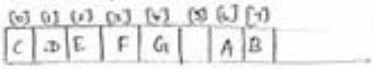
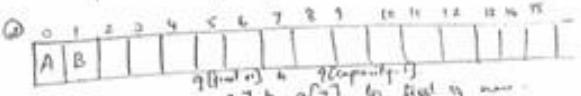
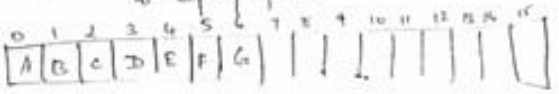
Figure 3.19: Configuration when stack i meets stack $i + 1$, but the memory is not full

3.c	Define Queue. Discuss how to represent Queue using dynamic arrays.	**6**	**L2**	**CO2**

Definition:

A queue is an ordered list in which insertion takes place through one end called the rear and deletion takes place at the other end called front

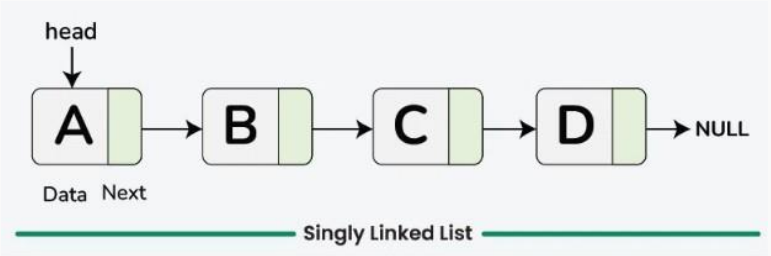
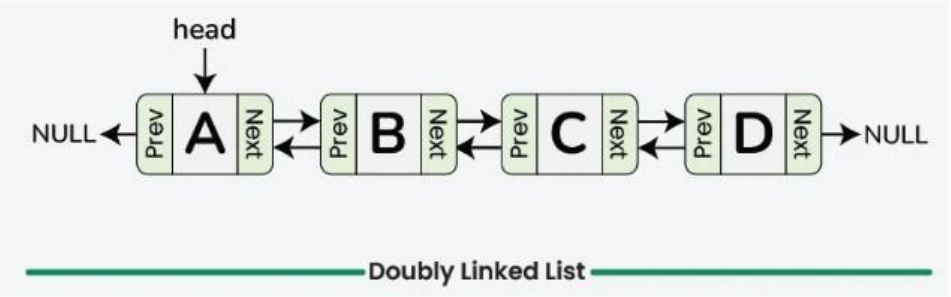
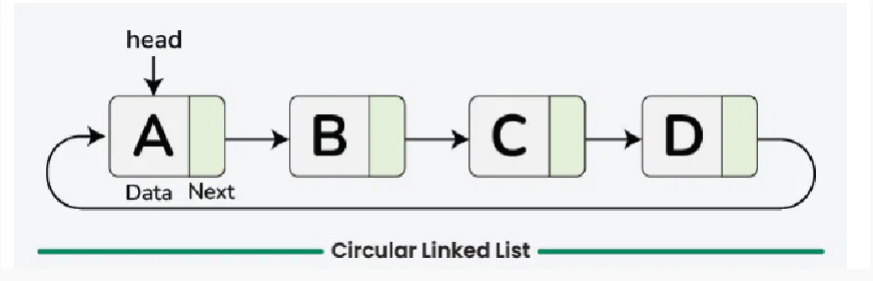
The queue is called First-In-First-Out (FIFO)

	19 Circular queue : Dynamic Allocation  ① doubling ② q[front] to q[capacity-1] ③ q[0] to q[rear] at capacity-front-1 (8 - 5-1) = 2 Array Representation  Array Doubling  new queue Capacity = capacity x 2  copy q[0] to q[7] to first of new copy second segment ③ queue[0] to queue[rear] at capacity - front - 1 			
4.a.	What are Linked list? Explain the different types of Linked List with neat diagram.	**4**	**L2**	**CO2**

Definition:

Linear collection of data elements called nodes, where the linear order is given by means of pointer

* Each node is divided into two parts
  - first part contains the information of the element
  - second part contain link field or next pointer field contains the address of the next node in the list.

	Singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.  *A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the **next** as well as the **previous** node in sequence.*  *A circular linked list is a type of linked list in which the last node's next pointer points back to the first node of the list, creating a circular structure. This design allows for continuous traversal of the list, as there is no null to end the list.*  			
4b	Give the structure definition for Singly Linked List (SSL). Write a C function to, (i) Insert an element at the end of SSL. (ii) Delete at node at the end of SSL. **C Representation of Linked List:** typedef struct LinkedList node; typedef struct LinkedList { int data; node *link; }node; node *first,*temp,*ptr;	**8**	**L3**	**CO2**

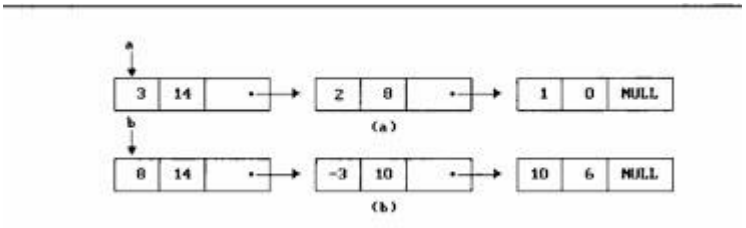
(i)

. Insert an element at the end of list:

```
void insert_last(int ele)
{
   create_node(ele);
   ptr=first;
   while(ptr->link!=NULL)
   {
      ptr=ptr->link;
   }
   ptr->link=temp;
}
```

(ii)

```
void delete_last()
{
node *prev;
ptr=first;
while(ptr->link!=NULL)
{
prev=ptr;
ptr=ptr->link;
}
temp=ptr;
prev->link=NULL;
free(temp);
}
```

4c	Write a C function to add two polynomials show the Linked List representation of below two polynomials $p(x) = 3x^{14} + 2x^{7} + 1$ $q(x) = 8x^{14} + 5x^{5} + 3x^{2} + 2$ 	**8**	**L3**	**CO2**

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
/* return a polynomial which is the sum of a and b */
   poly_pointer front, rear, temp;
   int sum;
   rear = (poly_pointer)malloc(sizeof(poly_node));
   if (IS_FULL(rear)) {
      fprintf(stderr, "The memory is full\n");
      exit(1);
   }
   front = rear;
   while (a && b)
      switch (COMPARE(a->expon,b->expon)) {
         case -1: /* a->expon < b->expon */
               attach(b->coef,b->expon,&rear);
               b = b->link;
               break;
         case 0: /* a->expon = b->expon */
               sum = a->coef + b->coef;
               if (sum) attach(sum,a->expon,&rear);
               a = a->link;  b = b->link; break;
         case 1: /* a->expon > b->expon */
               attach(a->coef,a->expon,&rear);
               a = a->link;
      }
   /* copy rest of list a and then list b */
   for (; a; a = a->link) attach(a->coef,a->expon,&rear);
   for (; b; b = b->link) attach(b->coef,b->expon,&rear);
   rear->link = NULL;
   /* delete extra initial node */
   temp = front; front = front->link;  free(temp);
   return front;
}
```

Program 4.10: Add two polynomials

5a	Write a C-function for the following operation on doubly Linked List (DLL) : (i) Addition of a DLL node. (ii) Concatenation of two DLL.	**8**	**L3**	**CO3**

```
void dinsert(node_pointer node, node_pointer newnode)
{
/* insert newnode to the right of node */
   newnode->llink = node;
   newnode->rlink = node->rlink;
   node->rlink->llink = newnode;
   node->rlink = newnode;
}
```

Program 4.28: Insertion into a doubly linked circular list

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

// Node structure
typedef struct Node {
```

```
    int data;
    struct Node* next;
    struct Node* prev;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}

// 1. Function to concatenate two DLLs: appends list2 to the end of list1
Node* concatenateDLL(Node* head1, Node* head2) {
    if (head1 == NULL) return head2;
    if (head2 == NULL) return head1;

    Node* temp = head1;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = head2;
    head2->prev = temp;
    return head1;
}

// 2. Function to add two DLLs (as numbers)
// Assumes lists are equal length; returns sum list
Node* addDLL(Node* h1, Node* h2, int* carry) {
    if (h1 == NULL) return NULL;

    Node* res = (Node*)malloc(sizeof(Node));
    res->next = addDLL(h1->next, h2->next, carry);

    int sum = h1->data + h2->data + *carry;
    *carry = sum / 10;
    res->data = sum % 10;
```

```
    res->prev = NULL;

    if (res->next != NULL) {
        res->next->prev = res;
    }
    return res;
}

// Helper: Print DLL
void printList(Node* head) {
    while (head) {
        printf("%d <-> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    // List 1: 1 <-> 2 <-> 3
    Node* head1 = createNode(1);
    head1->next = createNode(2);
    head1->next->prev = head1;
    head1->next->next = createNode(3);
    head1->next->next->prev = head1->next;

    // List 2: 4 <-> 5
    Node* head2 = createNode(4);
    head2->next = createNode(5);
    head2->next->prev = head2;
    printf("List 1: "); printList(head1);
    printf("List 2: "); printList(head2);
    // Concatenation
    Node* concatHead = concatenateDLL(head1, head2);
    printf("Concatenated: "); printList(concatHead);
    return 0;
}
```

5b	Write a C-function for the following operations on circular Linked List
	(i) Inserting at the front of a List.
	(ii) Find the number of nodes in circular list.
	
	`void insert_front(list_pointer *ptr, list_pointer node)`
	`/* insert node at the front of the circular list ptr,`
	`where ptr is the last node in the list */`
	`{`
	`  if (IS_EMPTY(*ptr)) {`
	`  /* list is empty, change ptr to point to new entry */`
	`  *ptr = node;`
	`  node->link = node;`
	`  }`
	`  else {`
	`  /* list is not empty, add new entry at front */`
	`  node->link = (*ptr)->link;`
	`  (*ptr)->link = node;`
	`  }`
	`}`
	
	`int length(list_pointer ptr)`
	`{`
	`/* find the length of the circular list ptr */`
	`  list_pointer temp;`
	`  int count = 0;`
	`  if (ptr) {`
	`  temp = ptr;`
	`  do {`
	`  count++;`
	`  temp = temp->link;`
	`  } while (temp != ptr);`
	`  }`
	`  return count;`
	`}`
	
	Program 4.20: Finding the length of a circular list
		8	**L3**	**CO3**

5c	Represent the given Sparse matrix using linked list representation. $A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 7 & 0 & 1 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$ **Linked List Representation (Triplet Form)** Each node in the linked list contains: (row, column, value, next) Header Node HEAD → (rows=5, cols=4, nonzeros=7)	**4**	**L3**	**CO3**

Data Nodes (row-wise order)

HEAD

↓

(1,1,2) → (2,1,4) → (2,4,3) → (4,1,7) → (4,3,1) → (4,4,1) → (5,3,6) → NULL

Step : Diagrammatic View

[5, 4, 7]

↓

[1,1,2] → [2,1,4] → [2,4,3] → [4,1,7] → [4,3,1] → [4,4,1] → [5,3,6] → NULL

Step : Node Structure (C-like)

```
struct node {
  int row;
  int col;
  int value;
  struct node *next;
};
```

6a	Explain the different types binary tree representation with example.
	
	Linked-Node Representation
	The most common and flexible method for representing any kind of binary tree uses dynamically allocated nodes and pointers (or references).
	
	• Structure: Each node in the tree is a structure or class that contains three components:
	  ○ `Data`: The value stored in the node.
	  ○ `Left Child Pointer`: A pointer or reference to the left child node.
	  ○ `Right Child Pointer`: A pointer or reference to the right child node.
	• Example (Conceptual):
	A node with value `A` would have a left pointer to a node with value `B` and a right pointer to a node with value `C`.
	
	Array Representation
	This method uses a one-dimensional array to store the tree nodes and is particularly efficient for representing complete binary trees.
	
	• Structure: Nodes are stored in an array based on their level-order traversal (top to bottom, left to right).
	• Indexing Logic: For any node stored at index `P`:
	  ○ The left child is at index `2 * P + 1`.
	  ○ The right child is at index `2 * P + 2`.
	  ○ The parent is at index `floor((P - 1) / 2)`
		8	**L3**	**CO3**

6b	Define Threaded Binary tree. Discuss in threaded binary tree. A threaded binary tree is a variant of a binary tree that utilizes the otherwise null pointers in a standard linked representation to facilitate faster, non-recursive, and stack-free traversals. These extra pointers are called threads, which point to the in-order predecessor or successor of a node. Types of Threaded Binary Trees Threaded binary trees are generally classified into two main types: • Single Threaded: Either the left or right null pointer is used to point to the in-order predecessor or successor, respectively.	**4**	**L3**	**CO3**

	• Double Threaded (Fully Threaded): Both null pointers are used to point to the in-order predecessor (left) and successor (right)			

6c	Discuss Inorder, preorder, postorder and level order traversal with suitable recursive function for each.
	`void inorderTraversal(struct node* root) {`
	`  if (root == NULL) return;`
	`  inorderTraversal(root->left);  // Traverse left subtree`
	`  printf("%d ->", root->item);  // Visit root`
	`  inorderTraversal(root->right);  // Traverse right subtree`
	`}`
	`void printPreorder(struct Node* node) {`
	`  if (node == NULL) return;`
	`  printf("%d ", node->data);  // Root`
	`  printPreorder(node->left);  // Left`
	`  printPreorder(node->right); // Right`
	`}`
	`void printPostorder(struct Node* node) {`
	`  if (node == NULL) return;`
	`  printPostorder(node->left);  // Left`
	`  printPostorder(node->right); // Right`
	`  printf("%d ", node->data);  // Root`
	`}`
		8	**L2**	**CO3**

7a	Write a function to perform the following operations on Binary Search Tree (BST): (i) Inserting on element into BST. (ii) Recursive search of a BST. **(i)**	**8**	**L3**	**CO4**

```
void insert(int num)
{
      p=(node*)malloc(sizeof(node));
      p->info=num;
      p->left=p->right=NULL;
      if(root==NULL)
      {
            root=p;
            return;
      }
      temp=root;
      while(temp!=NULL)
      {
            if(num>=temp->info)
            {
                  prev=temp;
                  temp=temp->right;
            }
            else
            {
                  prev=temp;
                  temp=temp->left;
            }
      }
      if(num>=prev->info)
            prev->right=p;
      else
            prev->left=p;
      }
```

(ii)

```
void search(node *temp,int data)
{
if(temp==NULL)
      printf("\nElement not found.........");
else if(data<temp->info)
      search(temp->left,data);
else if(data>temp->info)
      search(temp->right,data);
else
      printf("\nElement found......");
}
```

7b	Discuss selection Trees with suitable example.	**8**	**L2**	**CO4**

Selection Trees

* Assume k ordered sequences called runs.
* Runs consists of records and the keys of the records are in non-decreasing order.
* Merging is done by repeatedly outputting records with the smallest key.
* This is achieved by using by a selection tree by reducing the number of comparisons needed to find the next smallest element.

Two kinds of Selection Trees:

1. Winner Tree
2. Loser Tree

Winner Tree:

* A complete binary tree in which each node represent the smaller of its two children.
* The root represent the smallest node in the tree.

Example:

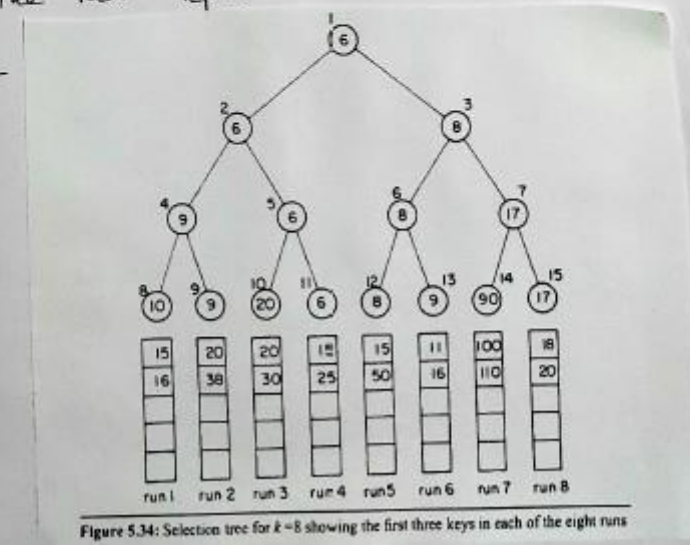
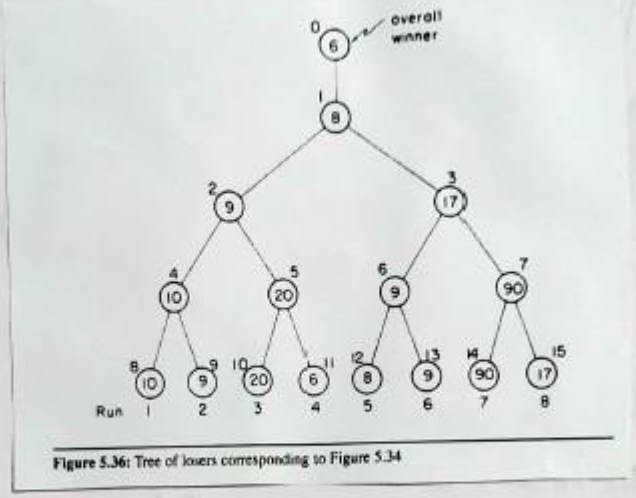


Figure 5.34: Selection tree for $k = 8$ showing the first three keys in each of the eight runs

	Loser Tree: A selection tree in which each non-leaf node retains a pointer to the loser is called loser tree. Example:  Figure 5.36: Tree of losers corresponding to Figure 5.34 * Each node contains key value of a record. * leaf node represent first record in each run. * Additional node : node 0 — represent overall winner.			
7c	Explain transforming a forest into a binary tree with an example.	**4**	**L2**	**CO4**

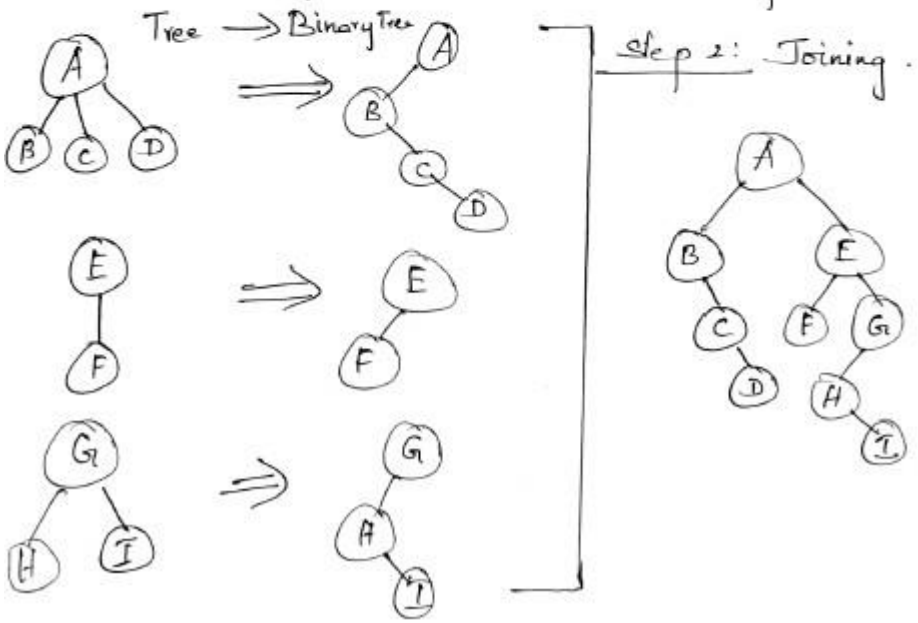
Transforming a Forest into Binary Tree:

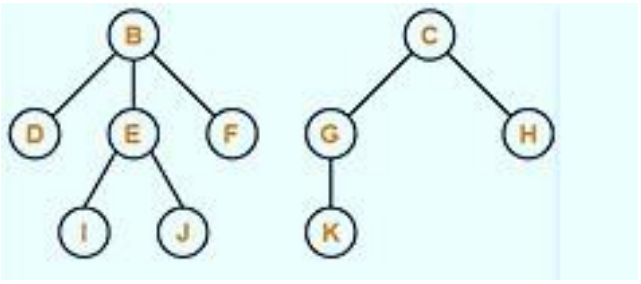
Step 1: Convert each tree into a binary tree using Left-child – Right Sibling Method.

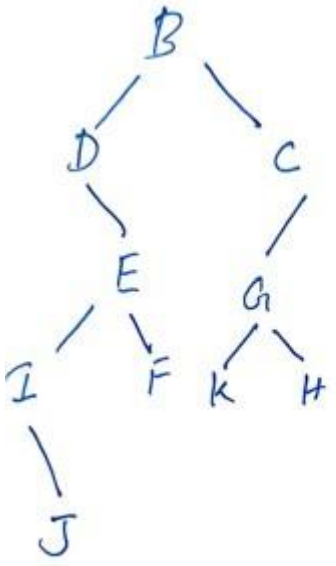
Step 2: Link Binary trees together through right child field of the root nodes.

Example:

The above forest is converted to Binary Tree.







8a	Define graph. Show the adjacency matrix and adjacency. List representation of the graph given below.  Fig. Q8 (a)	**6**	**L3**	**CO4**

Graph:

A graph, G consists of two sets V and E.

V - a finite non-empty set of vertices
[each node is called a vertex]

E - a set of pairs of vertices and these pairs are called Edges.

Adjacency Matrix:

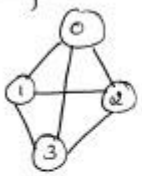
. Let $G=(V, E)$ be a graph with n vertices.

$$n \geq 1$$

Adjacency matrix is a 2-D array with the following property.

> ⊛ a[i][j] = 1, iff the edge (i, j) is in E(G).
> ⊛ a[i][j] = 0, if there is no such edge.

Example:



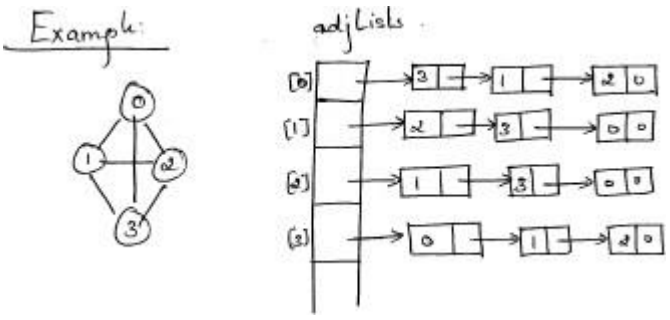
Graph.

Adjacency Matrix:

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

Adjacency List:

* The n rows of the adjacency matrix is represented as 'n' chains.
* There is one chain for each vertex in G.
* The nodes in chain i represent the vertices that are adjacent from vertex i.
* The data field of a chain node stores the index of an adjacent vertex.
* adjList[i] is a pointer to the first node in the adjacency list for vertex i.

				
8b	Define the following Terminologies with examples : (i) Vertex (node) (ii) Self loop (iii) Weighted graph (iv) Parallel edges • Vertex (Node): The basic component of a graph used to represent data points, often labeled, and typically connected by edges. • Self-loop: An edge where the starting and ending vertex is the same. • Parallel Edges (Multiple Edges): Two or more edges that connect the same pair of vertices, allowing multiple, distinct paths between them. • Weighted Graph: A graph where each edge is associated with a numerical value (e.g., cost, distance, or time)	**7**	**L1**	**CO4**

8c	Explain in detail elementary graph operations
	
	BFS:
	
	DFS:
	
	void dfs(int start,int n)
	{
	int visited[n]={0};
	int stack[MAX];
	int top=-1,i;
	printf("%d->",start);
	visited[start]=1;
	stack[++top]=start;
	while(top!=-1)
	{
	start=stack[top];
	for(i=0;i<MAX;i++)
	{
	if(adj[start][i] && visited[i]==0)
	{
	stack[++top]=i;
	printf("%d->",i);
	visited[i]=1;
	break;
	}
	}
	if(i==MAX)
	top--;
	}
		7	**L1**	**CO4**

	}			
9a	What is collision? What are the methods to resolve collision? Explain linear probing with example.	**8**	**L2**	**CO5**

Collision:

The situation in which the hash function returns same hash key for more than one record is called collisim.

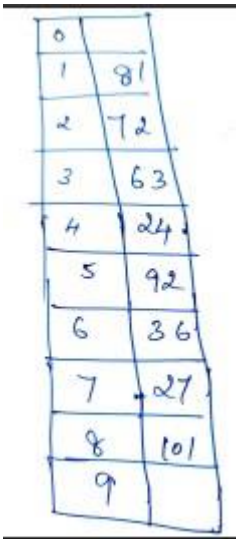
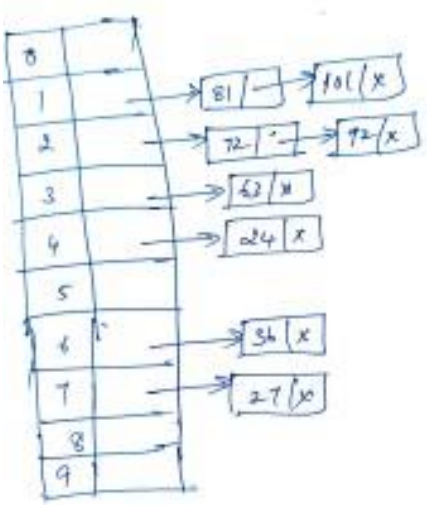
Linear Probing:

* Easiest method for handling collisim.
* When collisim occurs, the second record is placed linearly down, wherever an empty bucket is found.
* Hash table is a single-dimensional array ranging from 0 to tablsize-1.

Example:

Consider the following keys to be inserted in a hash table. 131, 4, 8, 7, 21, 5, 31

* Hash function:
  H(key) = key / tablsize

	  **Linear Probing** **Chaining**			
9b	Explain in details about static and dynamic hashing.	**6**	**L2**	**CO5**

Hashing:

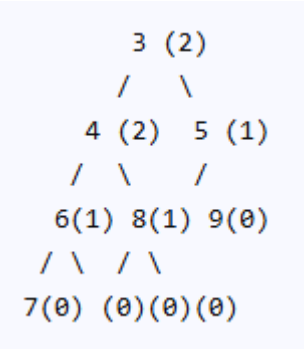
* Hashing is an effective way to store the elements in some data structure.
* It reduces the number of comparisons.
* Two elements of hashing.
  - Hash table.
  - Hash function.

Static Hashing:

Static hashing is a technique in which the resultant bucket size remains the same.

Dynamic Hashing:

Hashing technique in which bucket is size is not fixed. It can grow or shrink.

9c	Discuss Leftist Trees with an example. A leftist tree is a type of binary tree, often used to implement a mergeable priority queue (heap), that satisfies two main properties: 1. Heap Property: The key (value) of any node is less than or equal to the keys of its children in a min-leftist tree, or greater than or equal in a max-leftist tree. 2. Leftist Property: For every node, the null path length (NPL) of its left child is greater than or equal to the null path length of its right child. The NPL of a node is the length of the shortest path from that node to an external (null or empty) node in its subtree. 	**6**	**L2**	**CO5**
10a	Explain different types of HASH functions with example.	**6**	**L2**	**CO5**

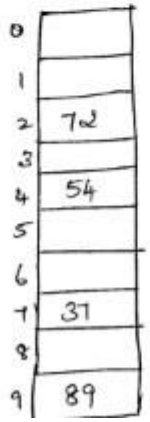
① Division Method:

* The hash function depends upon the remainder of division.
* Typically the divisor is table length.

Example:

Record : 54, 72, 89, 37

h(key) : record % table size



2) Mid-Square :

In the mid square method, the key is squared and the middle or mid part of the result is used as the index.

Example: Record : 3111.

$3111^2 = 9678321$

Hash table size 1000.

H(3111) = 783 (middle 3 - digits).

	3) Digit Folding: The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key. Example: Record : 1236541 2 ① Record divided into 123 654 12 ② Added together 123+654+12 = 0789 ⟹ H(1236541 2) = 789			
10b	Discuss different types of rotations with suitable examples. **Out of Syllabus**	**6**	**L3**	**CO5**
10c	Define Red-Black Tree, Splay tree. Discuss the method to insert an element into Red-Black tree. **Out of Syllabus**	**8**	**L2**	**CO5**