

Data Structure Using C (313301)

Model Question Paper — Set A · Model Answers

Q.1

a) Define data structure. Give two examples of linear data structure.

2 M

Data structure: A data structure is a way of organizing and storing data in computer memory so that it can be accessed and used efficiently (insert, delete, search, sort).

Examples of linear data structure: Array, Stack, Queue, Linked list.

(Definition 1 M, any two examples 1 M)

b) Differences between linear search and binary search.

2 M

Linear search	Binary search
Elements need not be sorted.	Elements must be in sorted order.
Compares the key with every element one by one.	Compares the key with the middle element and discards half the list each time.
Time complexity $O(n)$.	Time complexity $O(\log n)$.
Works on arrays and linked lists.	Works efficiently only on arrays (direct access).

(Any two points, 1 M each)

c) Define Abstract Data Type (ADT).

2 M

Abstract Data Type (ADT) is a logical description of a data type in which only the **data values** and the **operations** allowed on them are specified, without specifying how the data is stored or how the operations are implemented.

Example: Stack ADT — operations PUSH, POP, PEEK, isEmpty, isFull; it can be implemented using an array or a linked list.

d) LIFO principle of stack with example.

2 M

LIFO (Last In First Out): In a stack, the element inserted last is the first element to be removed. Insertion (PUSH) and deletion (POP) are done only at one end called **TOP**.

Real-life example: A pile of plates — the plate placed last on the top is taken out first. (Also: stack of books, undo operation in an editor.)

e) Types of queue.

2 M

1. Linear (simple) queue
2. Circular queue
3. Priority queue
4. Double ended queue (Deque)

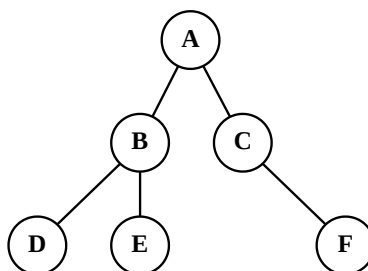
(½ M each)

f) Leaf node and degree of a node.

2 M

i) Leaf node: A node which has no child (degree 0) is called a leaf node or terminal node.

ii) Degree of a node: The number of children (sub-trees) of a node is called the degree of that node.



Leaf nodes: D, E, F · Degree of A = 2, degree of C = 1

g) Advantages of linked list over array.

2 M

- 1. Dynamic size:** Memory is allocated at run time, so the list can grow or shrink as needed; an array has fixed size.
- 2. Easy insertion and deletion:** Only pointers are changed; no shifting of elements as in an array.
- 3.** No memory wastage, since memory is allocated only when a node is needed.

(Any two, 1 M each)

Q.2

a) Bubble sort with example (40, 20, 50, 10, 30).

4 M

Working: Bubble sort compares adjacent elements and swaps them if they are in the wrong order. After every pass the largest remaining element 'bubbles up' to its correct position at the end. For n elements, at most n – 1 passes are needed.

Example: Given: 40, 20, 50, 10, 30

Pass	Comparisons / swaps	Array after pass
1	(40,20) swap; (40,50) no; (50,10) swap; (50,30) swap	20, 40, 10, 30, 50
2	(20,40) no; (40,10) swap; (40,30) swap; 50 fixed	20, 10, 30, 40, 50
3	(20,10) swap; (20,30) no; 40, 50 fixed	10, 20, 30, 40, 50
4	(10,20) no swap — array is sorted	10, 20, 30, 40, 50

Sorted list: 10, 20, 30, 40, 50

(Working 2 M, example with passes 2 M)

b) Concept of linked list with terms.

4 M

A **linked list** is a linear collection of data elements called **nodes**, where the order is given by pointers. Each node points to the next node; nodes need not be in continuous memory.

- Node:** Each element of the list. It has two parts — *data* (information) and *next* (address of the next node).
- Next pointer:** The address field of a node that stores the address of the next node in the list.

- **Null pointer:** The next field of the last node contains NULL, which shows the end of the list.
- **Empty list:** A list with no nodes; the START (head) pointer contains NULL.

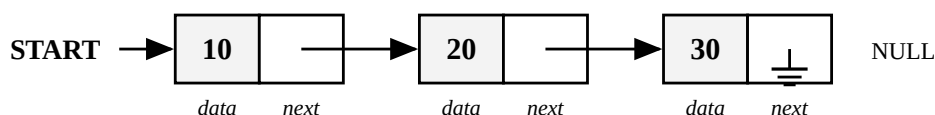


Fig.: Singly linked list (START points to first node, last node next = NULL)

START



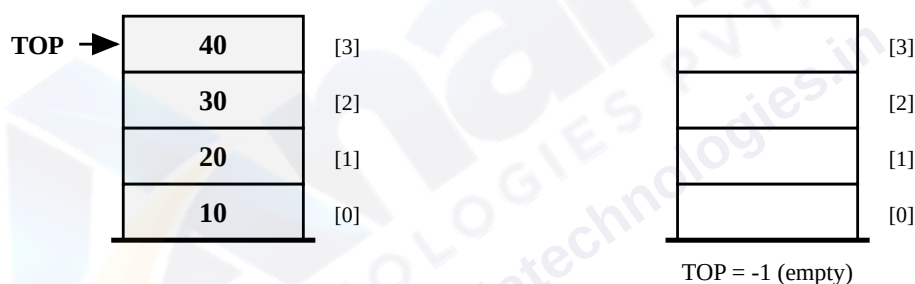
Empty list: START = NULL

c) Stack overflow and underflow.

4 M

Stack overflow: If we try to PUSH an element when the stack is already full ($TOP = MAX - 1$), the condition is called stack overflow. The new element cannot be inserted.

Stack underflow: If we try to POP an element when the stack is empty ($TOP = -1$), the condition is called stack underflow. There is no element to delete.



Overflow: stack full ($MAX = 4$), PUSH 50 not possible Underflow: stack empty, POP not possible

```
PUSH: if (top == MAX-1) printf("Stack Overflow"); else stack[++top] = x;
POP : if (top == -1)      printf("Stack Underflow"); else x = stack[top--];
```

(Each condition with example 2 M)

d) Expression tree for $(4p - 3q)^2 * (a + 6b)^3$

4 M

The operator with the lowest precedence that is applied last (*) becomes the root. Each sub-expression becomes a sub-tree; operands are leaf nodes. (4p means $4 * p$.)

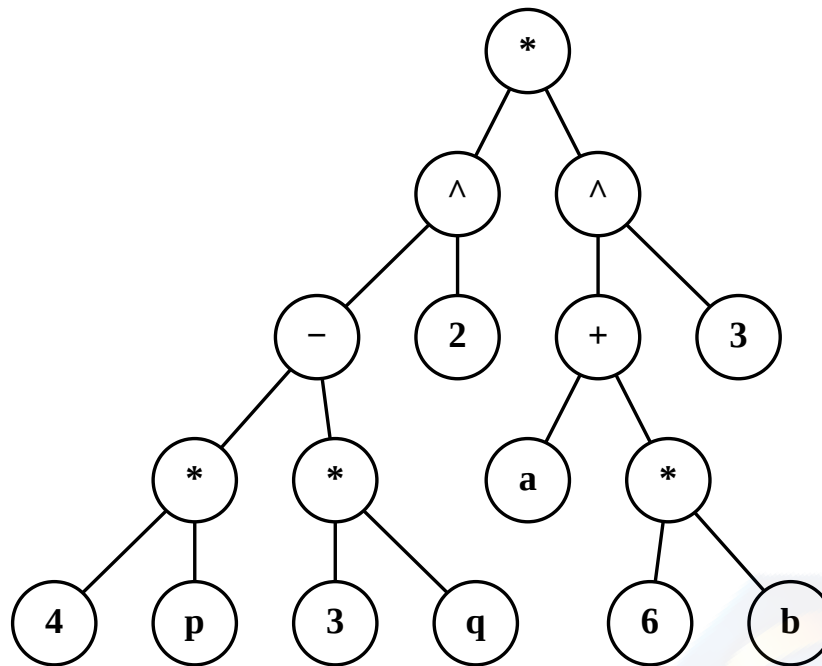


Fig.: Expression tree

(Correct tree 4 M)

Q.3

a) C program — binary search.

4 M

```
#include <stdio.h>

int binarySearch(int a[], int n, int key)
{
    int low = 0, high = n - 1, mid;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (a[mid] == key)
            return mid;          /* element found */
        else if (key < a[mid])
            high = mid - 1;       /* search left half */
        else
            low = mid + 1;        /* search right half */
    }
    return -1;                   /* element not found */
}

int main()
{
    int a[50], n, i, key, pos;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements in ascending order:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("Enter element to search: ");
    scanf("%d", &key);
    pos = binarySearch(a, n, key);
    if (pos == -1)
```

```
printf("Element not found\n");
else
    printf("Element found at position %d (index %d)\n", pos + 1, pos);
return 0;
}
```

Output:

```
Enter number of elements: 8
Enter 8 elements in ascending order:
11 18 25 32 47 56 63 78
Enter element to search: 47
Element found at position 5 (index 4)
```

b) Infix to postfix: $(A + B) * C - D / E \wedge F$

4 M

Rules: operands go to the postfix string; '(' is pushed; on ')' pop till '('; an operator pops operators of higher or equal precedence ($\wedge$ is right associative) and is then pushed. Precedence: $\wedge > *, / > +, -$.

Symbol	Stack	Postfix expression
(	(	
A	(	A
+	(+	A
B	(+	AB
)	—	AB+
*	*	AB+
C	*	AB+C
-	-	AB+C*
D	-	AB+C*D
/	-/	AB+C*D
E	-/	AB+C*DE
$\wedge$	-/ $\wedge$	AB+C*DE
F	-/ $\wedge$	AB+C*DEF
(end)	—	AB+C*DEF $\wedge$ -

Postfix expression: $AB+C*DEF\wedge/-$

c) Linear vs non-linear data structure.

4 M

Linear data structure	Non-linear data structure
Elements are arranged in a sequential (linear) order.	Elements are arranged in a hierarchical or network order.
Every element has a unique predecessor and successor (except first and last).	An element may be connected to many elements.
All elements can be traversed in a single run.	All elements cannot be traversed in a single run.

0	1	2	3	4	5	6	7
11	18	25	32	47	56	63	78

↑
LOW=HIGH=MID

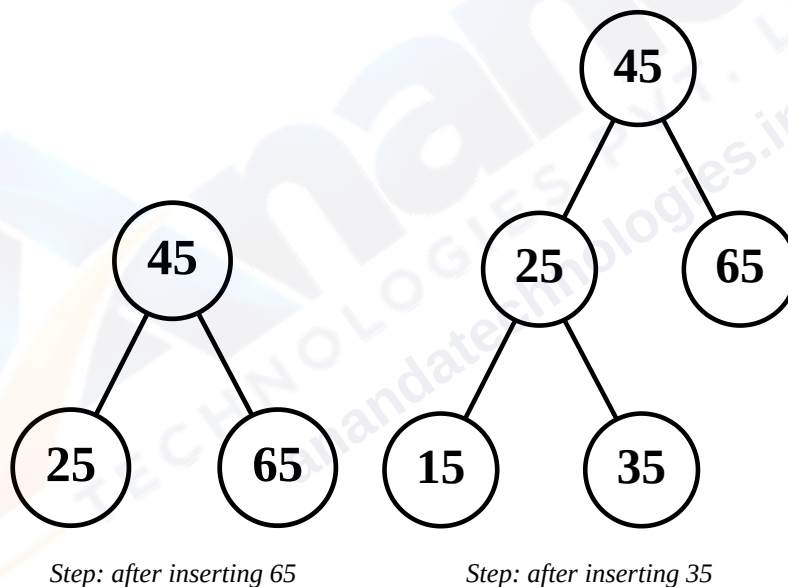
Element 47 is found at index 4, i.e. position 5.

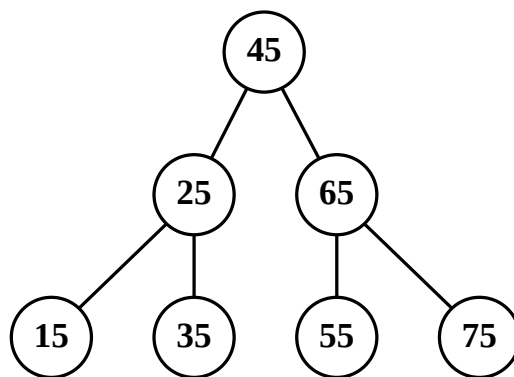
b) BST for 45, 25, 65, 15, 35, 55, 75, 30.

4 M

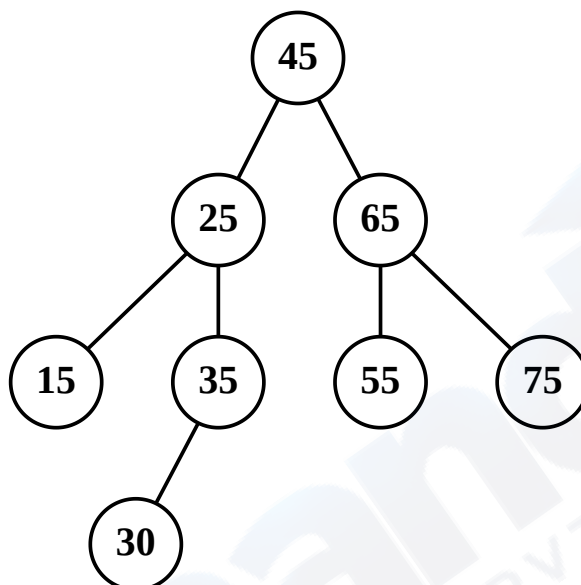
Rule: the first element is the root; a smaller value goes to the left sub-tree, a greater value goes to the right sub-tree.

1. 45 → root.
2. 25 < 45 → left of 45.
3. 65 > 45 → right of 45.
4. 15 < 45, 15 < 25 → left of 25.
5. 35 < 45, 35 > 25 → right of 25.
6. 55 > 45, 55 < 65 → left of 65.
7. 75 > 45, 75 > 65 → right of 65.
8. 30 < 45, 30 > 25, 30 < 35 → left of 35.





Step: after inserting 75

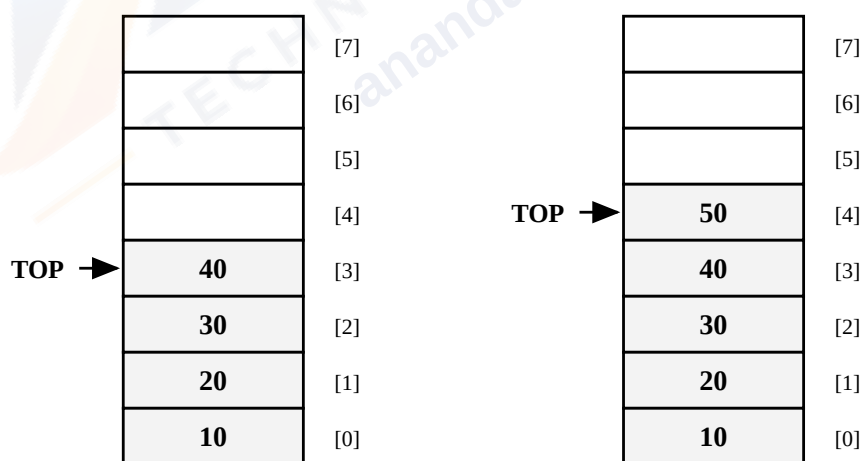


Final binary search tree

c) Effect of PUSH and POP on stack.

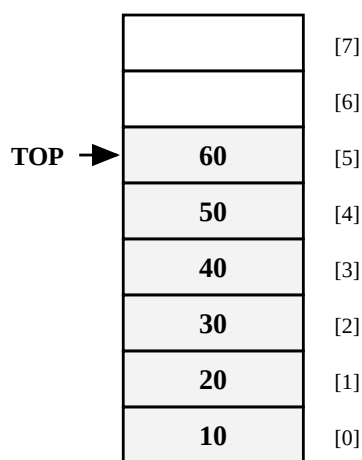
4 M

Stack size = 8 (index 0 to 7). Initially TOP = 3 (element 40).

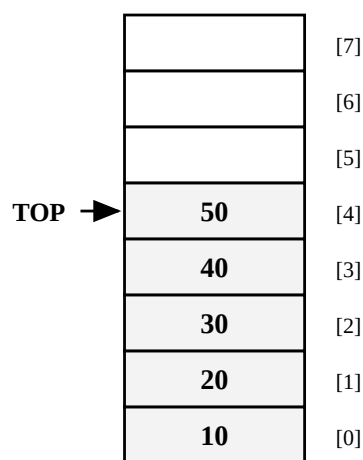


Initial

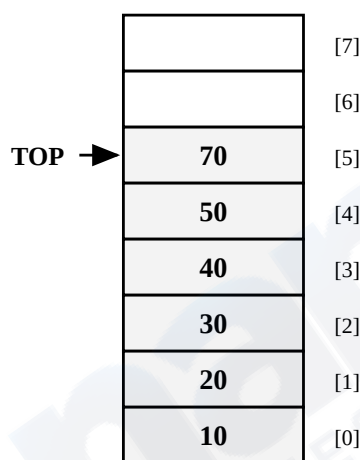
i) PUSH 50



ii) PUSH 60



iii) POP (60 removed)



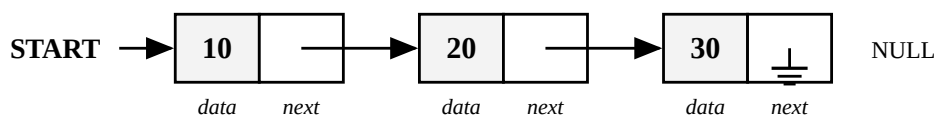
iv) PUSH 70

i) PUSH 50: TOP = 4 · ii) PUSH 60: TOP = 5 · iii) POP removes 60: TOP = 4 · iv) PUSH 70: TOP = 5.

Final stack: 10, 20, 30, 40, 50, 70 (70 at TOP).

d) Structures of linked lists.

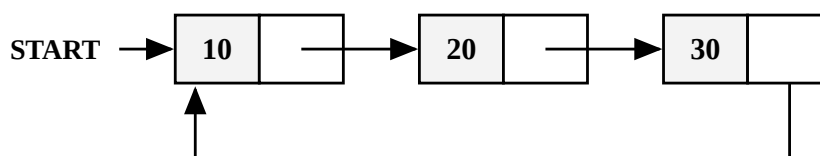
4 M



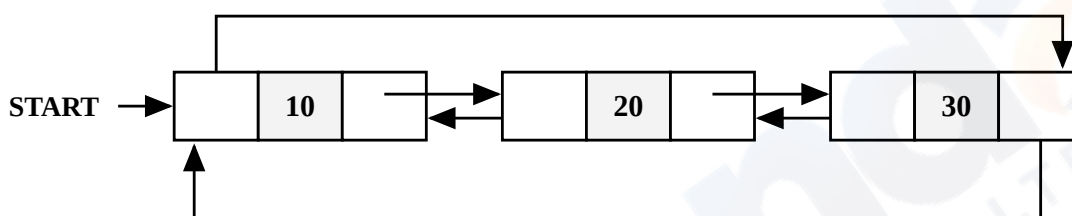
i) Singly linked list



ii) Doubly linked list



iii) Circular singly linked list — last node points to first node



iv) Circular doubly linked list — last next → first, first prev → last

e) Circular queue with diagram; advantage.

4 M

A **circular queue** is a queue in which the last position is connected back to the first position, forming a circle. When REAR reaches the last index and there is free space at the beginning, the next element is inserted at index 0.

Insert: $REAR = (REAR + 1) \% MAX$ · Delete: $FRONT = (FRONT + 1) \% MAX$ · Full when $(REAR + 1) \% MAX = FRONT$.

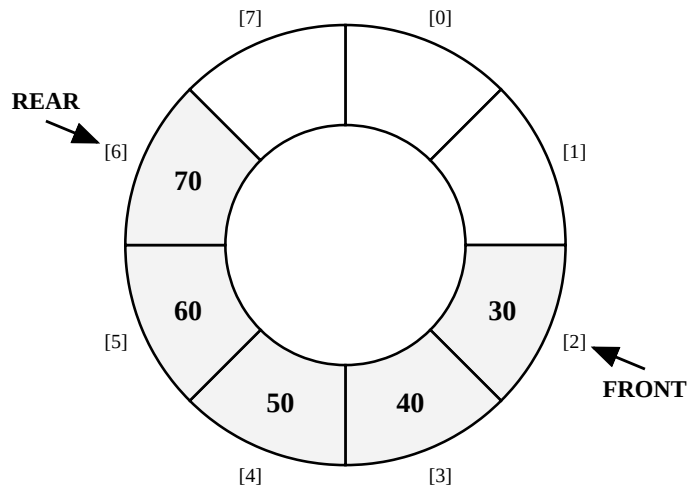


Fig.: Circular queue of size 8 (rear wraps from [7] to [0])

Advantage over linear queue: In a linear queue, the empty locations at the front (after deletions) cannot be reused, so 'queue full' may occur even when space is free. A circular queue reuses those empty locations, so memory is used efficiently.

(Description 1 M, diagram 2 M, advantage 1 M)

Q.5

a) C program — linear queue using array.

6 M

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
int front = -1, rear = -1;

void insert(int item)
{
    if (rear == MAX - 1)
    {
        printf("Queue Overflow\n");
        return;
    }
    if (front == -1)
        front = 0;
    rear = rear + 1;
    queue[rear] = item;
    printf("%d inserted\n", item);
}

void delete()
{
    if (front == -1 || front > rear)
    {
        printf("Queue Underflow\n");
        return;
    }
}
```

```
}
printf("%d deleted\n", queue[front]);
front = front + 1;
}

void display()
{
    int i;
    if (front == -1 || front > rear)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");
}

int main()
{
    int ch, item;
    do
    {
        printf("\n1.Insert 2.Delete 3.Display 4.Exit\nEnter choice: ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("Enter element: ");
                scanf("%d", &item);
                insert(item);
                break;
            case 2:
                delete();
                break;
            case 3:
                display();
                break;
            case 4:
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (ch != 4);
    return 0;
}
```

Sample output:

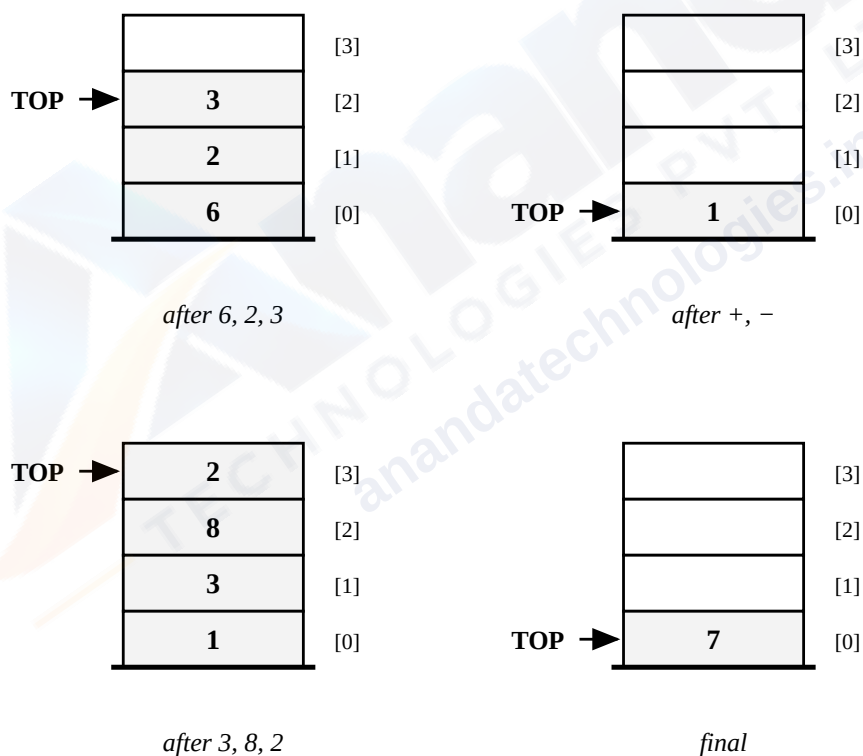
```
1 → Enter element: 10 → 10 inserted
1 → Enter element: 20 → 20 inserted
3 → Queue elements: 10 20
2 → 10 deleted
```

b) Evaluate postfix: 6, 2, 3, +, -, 3, 8, 2, /, +, *

6 M

Scan from left to right. Operand → PUSH. Operator → POP two operands (first popped = B, second = A), compute A op B, PUSH result.

Symbol	Operation	Stack (bottom → top)
6	Push 6	6
2	Push 2	6, 2
3	Push 3	6, 2, 3
+	Pop 3, 2; $2 + 3 = 5$; push 5	6, 5
-	Pop 5, 6; $6 - 5 = 1$; push 1	1
3	Push 3	1, 3
8	Push 8	1, 3, 8
2	Push 2	1, 3, 8, 2
/	Pop 2, 8; $8 / 2 = 4$; push 4	1, 3, 4
+	Pop 4, 3; $3 + 4 = 7$; push 7	1, 7
*	Pop 7, 1; $1 * 7 = 7$; push 7	7



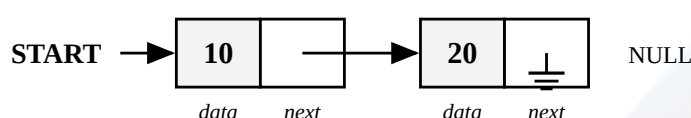
Result = 7

c) Singly vs doubly linked list.

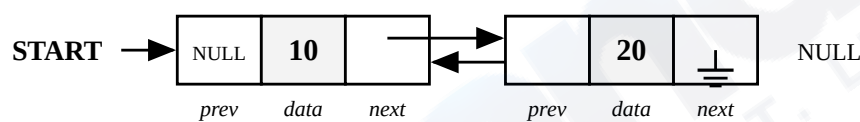
6 M

Singly linked list	Doubly linked list
Each node has two parts: data and next.	Each node has three parts: prev, data and next.

Traversal only in forward direction.	Traversal in both forward and backward directions.
Uses less memory (one pointer per node).	Uses more memory (two pointers per node).
Deleting a given node needs the address of its previous node (found by traversing).	A given node can be deleted directly using its prev pointer.
Insertion/deletion code is simpler (one pointer to change).	Insertion/deletion needs two pointers to be changed.
struct node { int data; struct node *next; };	struct node { struct node *prev; int data; struct node *next; };



Singly



Doubly

Q.6

a) Insertion sort: {28, 12, 45, 6, 33, 19}

6 M

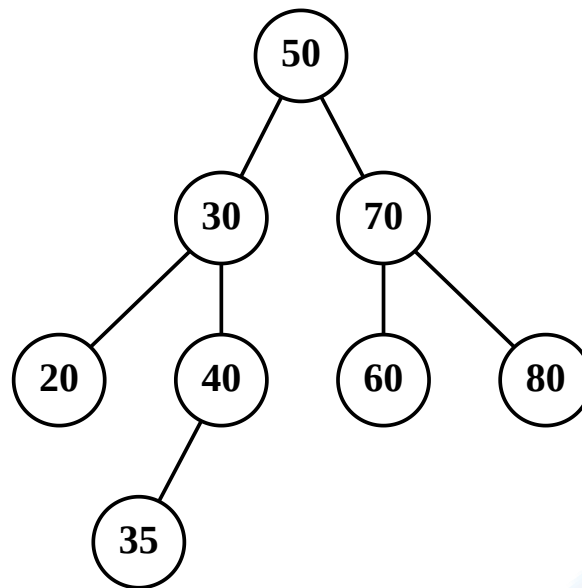
Working: The list is divided into a sorted part (left) and unsorted part (right). In each iteration the next element (key) is picked and inserted at its correct position in the sorted part by shifting greater elements one place to the right.

Iteration	Key	Array after iteration
Initial	—	28, 12, 45, 6, 33, 19
1	12	12, 28, 45, 6, 33, 19
2	45	12, 28, 45, 6, 33, 19
3	6	6, 12, 28, 45, 33, 19
4	33	6, 12, 28, 33, 45, 19
5	19	6, 12, 19, 28, 33, 45

Sorted list: 6, 12, 19, 28, 33, 45

b) BST for 50, 30, 70, 20, 40, 60, 80, 35 and traversals.

6 M



Binary search tree

Inorder (Left, Root, Right): 20, 30, 35, 40, 50, 60, 70, 80

Preorder (Root, Left, Right): 50, 30, 20, 40, 35, 70, 60, 80

Postorder (Left, Right, Root): 20, 35, 40, 30, 60, 80, 70, 50

(Tree 3 M, each traversal 1 M)

c) C program — delete from beginning and end of singly linked list.

6 M

```

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

struct node
{
    int data;
    struct node *next;
};

struct node *start = NULL;

void insertEnd(int x)
{
    struct node *temp, *p;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = x;
    temp->next = NULL;
    if (start == NULL)
        start = temp;
    else
    {
        p = start;
        while (p->next != NULL)
            p = p->next;
        p->next = temp;
    }
}
  
```

```
void deleteBeg()
{
    struct node *temp;
    if (start == NULL)
    {
        printf("List is empty\n");
        return;
    }
    temp = start;
    start = start->next;
    printf("Deleted %d from beginning\n", temp->data);
    free(temp);
}

void deleteEnd()
{
    struct node *p, *q;
    if (start == NULL)
    {
        printf("List is empty\n");
        return;
    }
    if (start->next == NULL)          /* only one node */
    {
        printf("Deleted %d from end\n", start->data);
        free(start);
        start = NULL;
        return;
    }
    p = start;
    while (p->next != NULL)          /* q follows p */
    {
        q = p;
        p = p->next;
    }
    q->next = NULL;
    printf("Deleted %d from end\n", p->data);
    free(p);
}

void display()
{
    struct node *p = start;
    printf("List: ");
    while (p != NULL)
    {
        printf("%d -> ", p->data);
        p = p->next;
    }
    printf("NULL\n");
}

int main()
{
    insertEnd(10);
    insertEnd(20);
    insertEnd(30);
    insertEnd(40);
    display();
}
```

```

deleteBeg();
display();
deleteEnd();
display();
return 0;
}

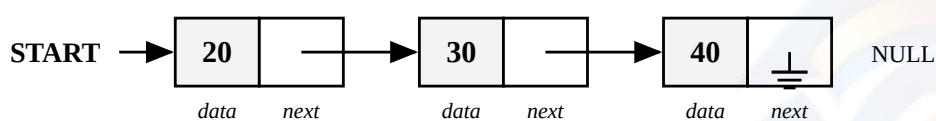
```

Output:

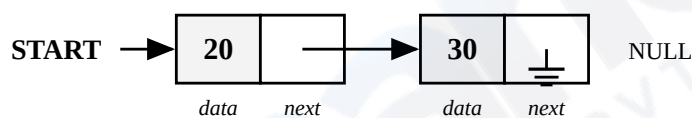
```

List: 10 -> 20 -> 30 -> 40 -> NULL
Deleted 10 from beginning
List: 20 -> 30 -> 40 -> NULL
Deleted 40 from end
List: 20 -> 30 -> NULL

```



Before deleting from end



After deleting 40 from end

Note: This is a **model question paper prepared by Ananda Technologies** only for practice and exam preparation, based on the pattern of previous MSBTE question papers and the K-scheme syllabus. It is **not an official MSBTE paper**, and it does **not** mean that these questions will come in the exam. Answers follow standard textbook points; marks shown are indicative. Free for students — anandatechnologies.in