Data Structure (CS100) | Unit I: Introduction & Linear Data Structures

DATA STRUCTURE

CS10২

UNIT I — Introduction & Linear Data Structures

Part-I: Stack & Queue

BTechX

1. Introduction to Data & Data Structure

1.1 What is Data?

Data is a collection of facts, such as numbers, words, measurements, observations or descriptions of things. It can come in the form of text, observations, figures, images, numbers, graphs, or symbols.

- Examples: prices, weights, addresses, ages, names, temperatures, dates, distances

 Key Point: Processed data is called Information. Representation of data is called Structure.

1.2 What is a Data Structure?

A data structure is a specialized format for organizing, processing, retrieving and storing data. It is the logical or mathematical model of a particular organization of data.

- Provides an efficient way of storing and organizing data in the computer.
- Makes data usage efficient (quick access, easy traversal, easy searching).

1.3 Why Data Structures?

- Applications are becoming more complex and the amount of data is increasing every day.
- Problems with data searching, processing speed, multiple requests handling arise without proper structure.
- Data Structures support different methods to organize, manage, and store data efficiently.

Benefits of Data Structures

- Easier to traverse data items
- Accessing data becomes quick and efficient

- Searching of data becomes easy
- Provides efficient memory utilization

2. Types of Data Structures

2.1 Primitive vs Non-Primitive

Type	Description
Primitive	Predefined data types supported directly by the programming language. Can store only a single value. Examples: int, char, float, double, pointers.
Non-Primitive	Derived from primitive data types. Can hold multiple values in contiguous or random locations. Examples: Arrays, Stacks, Queues, Linked Lists.

2.2 Linear vs Non-Linear

Type	Description
Linear	Data stored in sequential or linear fashion. Each element is attached to its previous and next adjacent elements. Single run can traverse the structure. Types: Static and Dynamic. Examples: Array, Stack, Queue, Linked List.
Non-Linear	Data elements are not placed sequentially. Multiple paths between elements. Data elements connected to one or more elements. Examples: Tree, Graph.

2.3 Static vs Dynamic

Type	Description
Static	Fixed memory size. Easier to access elements. Example: Array.
Dynamic	Size is not fixed. Memory can be increased or decreased, providing more flexibility. Examples: Stack, Queue, Linked List.

3. Linear Data Structures — Overview

3.1 Array

- Collection of similar data types stored at contiguous memory locations.
- Uses index-based data structure to identify each element.
- Used to implement Stacks, Queues, Heaps, Hash tables, etc.

 Key Point: Data type of elements may be char, int, float, or double.

3.2 Stack (LIFO)

Stack is a linear data structure that follows the LIFO (Last In First Out) principle. Insertion (PUSH) and deletion (POP) operations are performed only from the top end.

3.3 Queue (FIFO)

Queue is a linear data structure that follows the FIFO (First In First Out) principle. Insertion at one end (REAR) and deletion at the other end (FRONT).

3.4 Linked List

Elements are stored non-contiguously. A pointer is used to link elements together. Head node stores the address of the first element.

3.5 Non-Linear — Tree & Graph

Structure	Description
Tree	Stores data on multiple levels. Top node is Root. Branches are Children. Represents hierarchical relationships. Consists of nodes connected by edges.
Graph	Pictorial representation of elements (vertices) connected by edges. Unlike trees, graphs can have cycles.

Real-Life Applications of Data Structures

- Stack — Browser back/forward buttons, Undo/Redo in text editors
- Queue — Printing in printers, CPU scheduling, ticket counters
- Tree — Social network graphs, file systems
- Graph — Google Maps (shortest path), social graphs

4. Array — In Depth

4.1 Types of Arrays

Type	Description
1D Array	Needs only one subscript to specify elements. Example: int arr[5];
2D Array	Needs two subscripts (rows and columns). Represents a matrix/table.
3D Array	Needs three subscripts. Used for complex data representations.

4.2 Operations on Array

Operation	Description
Traversal	Visiting every element of the array to print or process them.
Insertion	Adding an element at a particular index (beginning, end, or any position).
Deletion	Removing an element from a particular index.
Search	Finding an element using the given index or value.
Update	Replacing element at a particular index with a new value.

4.3 Insertion Algorithm

 Note: Array length(N), Element (ITEM) to insert, Position to insert (K)

```
1. Start
2. Set J = N-1
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5.    Set Arr[J+1] = Arr[J]
6.    Set J = J-1
7. Set Arr[K] = ITEM
8. Stop
```

4.4 Deletion Algorithm

 Note: Array length(N), Position to delete (K)

```
1. Start
2. Set ITEM = Arr[K]
3. Set J = K
4. Repeat steps 5 and 6 while J < N-1
5.    Set Arr[J] = Arr[J+1]
6.    Set J = J+1
7. Set N = N-1
8. Stop
```

4.5 Search Algorithm

 Note: Array length(N), Element (ITEM) to search

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
```

```
4.   If Arr[J] == ITEM Then Goto Step 6
5.   Set J = J+1
6. Print J, ITEM (found at position J)
7. Stop
```

4.6 Update Algorithm

 Note: Array length(N), Position K, New value NEW_ITEM

```
1. Start
2. Set Arr[K] = NEW_ITEM
3. Stop
```

4.7 Advantages & Disadvantages

Advantages	Disadvantages
Easier to declare and use	Number of elements must be predefined
Stores multiple elements of same type with same name	Array size cannot be changed after declaration
2D array represents tabular data efficiently	Allocating more memory than required leads to memory wastage

4.8 Address Calculations in 2D Array

Order	Formula
Row-Major Order	Address(A[i][j]) = B + (i*c + j) * w where B=base address, c=column size, w=element size
Column-Major Order	Address(A[i][j]) = B + (j*r + i) * w where B=base address, r=row size, w=element size

5. Stack — Complete Notes

5.1 Fundamentals of Stack

A stack is a linear data structure that follows the principle of Last In First Out (LIFO). Insertion (PUSH) and deletion (POP) operations are performed only from the top of the stack.

 Key Point: Think of a stack of plates — you always add and remove from the top!

5.2 Operations on Stack

Operation	Description
Push()	Add an element to the top of a stack
Pop()	Remove an element from the top of a stack
Peep() / Peek()	Get the value of the top element without removing it
IsEmpty()	Check if the stack is empty
IsFull()	Check if the stack is full
top()	Returns the top element of the stack
size()	Returns the size of the stack

5.3 PUSH Operation Algorithm

 Note: Input: STACK (linear array), MAX_SIZE. Initial: TOP = -1

```
PUSH (STACK, TOP, MAX_SIZE, ITEM)

Step 1: If TOP = MAX_SIZE - 1, then
           Print "Stack Overflow" and exit.
Step 2: Set TOP = TOP + 1
Step 3: Set STACK[TOP] = ITEM
Step 4: Exit
```

5.4 POP Operation Algorithm

 Note: Input: STACK (linear array), MAX_SIZE. Initial: TOP = -1

```
POP (STACK, TOP, MAX_SIZE, ITEM)

Step 1: If TOP = -1, then
           Print "Stack Underflow" and exit.
Step 2: Set ITEM = STACK[TOP]
Step 3: Set TOP = TOP - 1
Step 4: Exit
```

5.5 PEEP (Peek) Operation Algorithm

 Note: Returns the top element without removing it

```
PEEP (STACK, TOP, MAX_SIZE)

Step 1: If TOP = -1, then
```

```
            Print "Stack is Empty" and exit.
Step 2: Return STACK[TOP]
Step 3: Exit
```

5.6 Applications of Stack

Applications of Stack

- Browser history — back button saves previously visited URLs in a stack
- Undo/Redo operations in text editors
- Function calls and recursion (activation records)
- Expression evaluation (Postfix, Prefix)
- Conversion of expressions (Infix to Postfix / Prefix)
- Balanced Parentheses checking
- Tower of Hanoi
- To reverse a word or string

6. Recursion & Expression Conversion

6.1 Function Calls and Recursion

When a function calls itself directly or indirectly, it is called recursion. Every time a function is called, a structure called an activation record is created in memory, containing all local variables and parameters.

Example — Factorial using recursion:

```
long factorial(int n) {
    if (n > 1)
        return (n * factorial(n-1));
    else
        return 1;
}
```

 Key Point: Each function call creates a new stack frame (activation record). When the function returns, its frame is popped off the stack.

6.2 Arithmetic Expression Notations

Notation	Description	Example
Infix	Operator written between operands. Human-readable but needs precedence rules.	A + B
Prefix (Polish)	Operator written before operands. Machine-friendly, no parentheses needed.	+ A B
Postfix (Reverse Polish)	Operator written after operands. Used in compilers and calculators.	A B +

 Key Point: Prefix and Postfix notations are computationally efficient — they do not require parentheses or operator precedence tracking.

6.3 Operator Precedence & Associativity

Priority	Operator	Associativity
Highest	^ (Exponentiation)	Right to Left
High	*, / (Multiply, Divide)	Left to Right
Low	+, - (Add, Subtract)	Left to Right
Lowest	( ) Parentheses	Left to Right (override all)

6.4 Infix to Postfix Conversion (Manual Method)

Steps: Parenthesize from left to right. Sub-expressions converted to postfix are treated as single operands. Remove parentheses at the end.

Example: (A+B)*C/D+E^F/G

```
(A+B)*C/D+E^F/G
→ (AB+)*C/D+(EF^)/G         [() and ^ handled first]
→ ((AB+)C*)/D+(EF^)/G       [* left to right]
→ (((AB+)C*)D/)+(EF^)/G     [/ left to right]
→ (((AB+)C*)D/)+((EF^)G/)   [/ right side]
→ (((AB+)C*)D/)((EF^)G/)+   [+ last]

Postfix: AB+C*D/EF^G/+
```

6.5 Infix to Postfix Using Stack Algorithm

 Note: X = infix expression, Y = postfix output

```
1. Push '(' onto Stack, and add ')' to the end of X.
2. Scan X from left to right, repeat steps 3-6 until Stack is empty.
3. If operand encountered  → add it to Y.
4. If '(' encountered      → push it onto Stack.
5. If operator encountered →
   a. Pop from Stack and add to Y all operators with same/higher
precedence.
   b. Push current operator onto Stack.
6. If ')' encountered      →
   a. Pop from Stack and add to Y until '(' encountered.
   b. Remove '(' from Stack.
7. Exit
```

6.6 Infix to Postfix — Worked Example

Convert: A*(B+C+D)

Symbol Scanned	Stack	Postfix Expression
(	(	
A	(	A
*	(*	A
(	(*(	A
B	(*(	AB
+	(*(+	AB
C	(*(+	ABC
+	(*(+	ABC+ ← pop + (same precedence)
D	(*(+	ABC+D
)	(*	ABC+D+ ← pop until (
)		ABC+D+* ← pop until (

Result Postfix: ABC+D+*

6.7 Infix to Prefix Conversion

Steps:

- Step 1: Reverse the infix expression (swap '(' with ')' and vice versa)
- Step 2: Apply Infix-to-Postfix algorithm on the reversed expression
- Step 3: Reverse the resulting postfix expression to get the prefix

Example: a*(b+c+d)

```
Original infix:          a*(b+c+d)
Step 1 — Reverse:        )d+c+b(*a   → replace brackets: (d+c+b)*a
Step 2 — Postfix of (d+c+b)*a:  dc+b+a*
Step 3 — Reverse:        *a+b+cd

Prefix: *a+b+cd
```

7. Expression Evaluation using Stack

7.1 Postfix Expression Evaluation

 Note: Scan postfix expression from left to right

1. Scan expression from left to right.
2. If operand encountered → PUSH it onto stack.
3. If operator (#) encountered →
   a. Pop top element → A (top), Pop next → B (top-1).
   b. Perform operation: B # A
   c. Push result back to stack.
4. Final result is the remaining element in stack.

Example: 7 8 2 * 4 / +

```
Scan 7  → Stack: [7]
Scan 8  → Stack: [7, 8]
Scan 2  → Stack: [7, 8, 2]
Scan *  → Pop 2 and 8, compute 8*2=16 → Stack: [7, 16]
Scan 4  → Stack: [7, 16, 4]
Scan /  → Pop 4 and 16, compute 16/4=4 → Stack: [7, 4]
Scan +  → Pop 4 and 7, compute 7+4=11  → Stack: [11]

Result: 11
```

7.2 Prefix Expression Evaluation

 Note: Reverse prefix expression first, then scan left to right

1. Reverse the prefix expression.
2. Scan from left to right.
3. If operand encountered → PUSH it onto stack.
4. If operator (#) encountered →
   a. Pop top element → A (top), Pop next → B (top-1).
   b. Perform operation: A # B (Note: order reversed vs postfix)
   c. Push result back to stack.
5. Final result is the remaining element in stack.

 Note: For Postfix: operation is B # A (TOP-1 operator TOP). For Prefix (after reversing): operation is A # B (TOP operator TOP-1).

8. Queue — Complete Notes

8.1 Fundamentals of Queue

Queue is a linear data structure that follows the FIFO (First In First Out) principle. A queue enables insert operations at one end called REAR and delete operations at another end called FRONT.

 Key Point: Think of a queue of people waiting at a ticket counter — First come, first served!

8.2 Representation using Array

- Create an array of size n.
- Two variables: front and rear, both initialized to -1 (empty queue).
- REAR: index up to which elements are stored.
- FRONT: index of the first element of the array.

8.3 Types of Queue

Type	Description
Linear Queue	Simple FIFO queue. Limitation: memory wastage due to unutilized space after deletion.
Circular Queue	Last node is connected to the first node. Solves memory wastage problem. Uses modulo operation.
Double-Ended Queue (Deque)	Insertion and deletion from both ends. Hybrid of stack and queue.
Priority Queue	Elements are served based on priority, not order of insertion.

8.4 Insertion (Enqueue) Algorithm — Linear Queue

 Note: MAX = max size, FRONT and REAR initialized to -1

```
Step 1: If REAR = MAX - 1
           Print "OVERFLOW / QUEUE FULL" and EXIT
Step 2: If FRONT = -1 and REAR = -1
           Set FRONT = REAR = 0
        Else
           Set REAR = REAR + 1
Step 3: Set Q[REAR] = NUM
Step 4: EXIT
```

8.5 Deletion (Dequeue) Algorithm — Linear Queue

 Note: Delete from FRONT of queue

```
Step 1: If FRONT = -1 and REAR = -1
```

```
            Print "UNDERFLOW / QUEUE EMPTY" and EXIT
Step 2: Set NUM = Q[FRONT]
Step 3: If FRONT == REAR
            Set FRONT = REAR = -1
        Else
            Set FRONT = FRONT + 1
Step 4: EXIT
```

9. Circular Queue

9.1 Why Circular Queue?

Drawback of linear queue is wastage of memory due to unutilized memory space after deletions. A Circular Queue is an extended version of a linear queue where the last node is connected to the first node, making a circular link.

 Key Point: Circular Queue uses the Modulo (%) operation to wrap around: Rear = (Rear + 1) % MAX

9.2 Modulo Operation (Used in Circular Queue)

The modulus operator (%) returns the remainder when one number is divided by another.

```
Example 1:  7 % 5 = ?
7 = 5 * 1 + 2  →  Remainder = 2

Example 2: -7 % 5 = ?
Largest number <= -7 divisible by 5 is -10
Remainder = (-7) - (-10) = 3
```

9.3 Insertion Algorithm — Circular Queue

 Note: max = Maximum Queue Size

```
Step 1: Check for queue full:
   If (rear == max-1 AND front == 0) OR front == (rear+1) % max
   Then → OVERFLOW, Exit
Step 2: Check position of rear:
   If front == -1 AND Rear == -1 → set front = rear = 0
   Else if rear == max-1 AND front != 0 → set rear = 0
   Else → set rear = (rear+1) % max
Step 3: Q[rear] = Data
Step 4: Exit
```

9.4 Deletion Algorithm — Circular Queue

 Note: Delete from FRONT of circular queue

```
Step 1: If front == -1 → UNDERFLOW, Exit
Step 2: Data = Q[front]
Step 3: Update position:
   If front == Rear → set front = rear = -1
   Else if front = max-1 → set front = 0
   Else → set front = (front+1) % max
Step 4: Exit
```

Applications of Circular Queue

- Memory management
- Round Robin CPU Scheduling
- Traffic Light System
- Clock systems

10. Double-Ended Queue (Deque)

10.1 What is Deque?

Deque is a linear data structure where the insertion and deletion operations are performed from both ends. It is a hybrid linear structure that provides all the capabilities of stacks and queues in a single data structure.

10.2 Types of Deque

Type	Description
Input Restricted Deque	Insertion restricted at single end (REAR) only. Deletion allowed at BOTH ends (front and rear).
Output Restricted Deque	Deletion restricted at single end (FRONT) only. Insertion allowed at BOTH ends (front and rear).

10.3 Why Deque?

- Does not require the strict LIFO/FIFO ordering enforced by stack/queue.
- When random access is required but you also care about insertion and removal at both ends.
- Provides all capabilities of stacks and queues in a single data structure.

10.4 Operations on Deque

Operation	Description
Insertion at Rear	rear = rear + 1

Operation	Description
Insertion at Front	front = front - 1 (or wrap around using modulo)
Deletion at Front	front = front + 1
Deletion at Rear	rear = rear - 1 (when Rear reaches 0, set REAR = MAXSIZE - 1)

Applications of Deque

- Browser history: Recently visited URLs added to front; old URLs removed from back after a limit.
- Ticket purchasing line: Someone can re-join the front of the queue.
- Undo-Redo operations
- Palindrome checking

11. Quick Revision Summary

11.1 Key Terminologies

Term	Meaning
LIFO	Last In First Out — principle of Stack
FIFO	First In First Out — principle of Queue
PUSH	Insert element into Stack (from top)
POP	Delete element from Stack (from top)
Enqueue	Insert element into Queue (at REAR)
Dequeue	Delete element from Queue (at FRONT)
Overflow	Trying to insert into a full Stack/Queue
Underflow	Trying to delete from an empty Stack/Queue
Infix	Operator between operands: A+B
Prefix	Operator before operands: +AB
Postfix	Operator after operands: AB+
Deque	Double-Ended Queue — insert/delete at both ends
Modulo (%)	Returns remainder: 7 % 5 = 2; used in circular queue to wrap around

11.2 Practice Questions

Important Questions

- What is a Dequeue? Distinguish between LIFO and FIFO.

- Write PUSH and POP algorithms with examples.
- Convert (A+B)*C/D+E^F/G to Postfix and Prefix form.
- Evaluate postfix expression: 7 8 2 * 4 / +
- Evaluate postfix expression: 6 2 3 + - 3 8 2 / + *
- Evaluate prefix expression: + - * 3 2 / 8 4 1
- Explain circular queue with insertion and deletion example.
- Write an algorithm to insert/delete an element in a linear queue.
- Write algorithms to insert at beginning and end of an array of size 5.
- What are the applications of Stack and Queue?

12. References & Resources

12.1 Text Books

- "Data Structure: A Pseudo code approach with C" — Richard F. Gilberg & Behrouz A. Forouzan, Thomson Publication, 2nd Ed.
- "Data Structure in C" — A.M. Tanenbaum, Y. Langsam, M.J. Augenstein, PHI/Pearson, 7th Ed.

12.2 Reference Books

- "Data Structures with C" (Schaum's Series) — Seymour Lipschutz, Tata-McGraw-Hill, 1st Ed.
- "Fundamentals of Data Structure in C" — Horowitz, Sahani & Freed, Computer Science Press, 2nd Ed.
- "Data Structures Using C" — Reema Thareja, Oxford University Press, 2nd Ed.

12.3 Online Sources

- https://nptel.ac.in/courses/106104128 (Introduction to Programming in C)
- https://archive.nptel.ac.in/courses/106/102/106102064 (Data Structures and Algorithms)
- https://www.geeksforgeeks.org/data-structures/
- https://www.javatpoint.com/data-structure-introduction
- https://btechx.infinityfreeapp.com

— End of Unit I Notes —

BTechX

Data Structure (CS100) | Unit II: Linear Data Structures — Linked List

DATA STRUCTURE (CS10২)

UNIT II — Linear Data Structures

LINKED LIST

BTechX

Syllabus — Unit II

- Single Linked List: Operations and Implementation
- Double Linked List: Operations and Implementations
- Circular Linked List: Concepts and Implementation
- Applications: Stack & Queue Implementation using Linked List
- Polynomial Manipulation using Linked List

1. Introduction to Linked List

1.1 What is a Linked List?

Definition: A linked list is a linear data structure in which elements (nodes) are NOT stored at contiguous memory locations. Nodes are connected using pointers.

- Each node contains two parts: DATA (value) and LINK (pointer to next node).
- The first node is called the HEAD node — used to traverse the entire list.
- Nodes can be stored at any random memory location and are linked using pointers.
- Such structures which contain a member field pointing to the same structure type are called self-referential structures.

1.2 Why Linked List? (Array Limitations)

Array Limitation	Linked List Advantage
Fixed Size — size must be decided in advance; cannot grow or shrink.	Dynamic Size — nodes can be added or removed at runtime; no memory wasted.
Slow Insertion/Deletion — requires shifting of all elements when inserting or deleting in the middle.	Fast Insertion/Deletion — only pointers are updated; no shifting required.

Array Limitation	Linked List Advantage
Contiguous Memory — requires a single continuous block of memory; fails on fragmented memory.	Non-Contiguous Memory — nodes can be at any memory location, connected via pointers.

1.3 Structure of a Node

 Definition: A node is the basic building block of a linked list. It contains a DATA part (stores the value) and an ADDRESS part (pointer to the next node).

In C, a node is defined using a structure:

```
struct node {
    int data;           // data part
    struct node *next;  // pointer to next node
};
```

 Key Point: The pointer variable 'next' holds the address of the successor node, or NULL (0) if it is the last node.

1.4 Types of Linked Lists

Type	Description
Singly Linked List	Each node has one pointer (next). Traversal is one-directional (forward only). Last node points to NULL.
Doubly Linked List	Each node has two pointers (prev + next). Traversal is bi-directional. First node's prev = NULL, last node's next = NULL.
Circular Linked List	Last node points back to the first node (head) instead of NULL, forming a closed loop.
Circular Doubly Linked List	Combination of Doubly + Circular. Each node has both prev and next pointers, and the list forms a circle.

2. Singly Linked List

2.1 Definition & Structure

 Definition: A singly linked list consists of nodes where each node has a data field and a reference (next) to the next node. The next pointer of the last node is NULL.

Visual Structure: HEAD → [A|→] → [B|→] → [C|→] → [D|NULL]

2.2 Operations on Singly Linked List

Four Main Operations

- Insertion — Add a new node to the linked list
- Deletion — Remove an existing node from the list
- Search — Find a node in the linked list
- Traversal — Access each node of the linked list

2.3 Insertion in Singly Linked List

There are 5 cases for inserting a node in a singly linked list:

Case 1: Insertion at the Beginning

- Allocate space for new node and store data into the data part.
- Make the NEXT pointer of new node point to the existing HEAD (first node).
- Make the new node the new HEAD of the list.

```
newnode = malloc(sizeof(struct node));
if (newnode == NULL) { printf("Memory allocation failed"); return 1; }
scanf("%d", &item);
newnode->data = item;
newnode->next = head;     // new node points to old head
head = newnode;           // new node becomes new head
```

 Remember: ptr->next = head; head = ptr; — Just two pointer updates, no shifting needed!

Case 2: Insertion at the End

- Create new node with given data and set new_node->next = NULL.
- If head == NULL (empty list): make new node the head.
- If head != NULL: traverse to find the last node, then set last_node->next = new_node.

```
newNode = (struct Node *)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = NULL;
if (head == NULL) { head = newNode; }
else {
    temp = head;
    while (temp->next != NULL) { temp = temp->next; }  // traverse to last
    temp->next = newNode;   // link new node at end
}
```

Case 3: Insertion at a Specific Position

- Create new node using malloc. If memory fails, display error and stop.
- If position == 1: set newNode->next = head and update head = newNode.

- Otherwise, traverse to the (position-1)th node using temp pointer.
- If temp becomes NULL, display 'Invalid position' and stop.
- Set newNode->next = temp->next and temp->next = newNode.

```
for (i = 1; i < pos - 1 && temp != NULL; i++) { temp = temp->next; }
if (temp == NULL) { printf("Invalid position\n"); return; }
newNode->next = temp->next;
temp->next = newNode;
```

Case 4: Insertion After a Given Node

- Traverse the list from HEAD to find the given node (key).
- If key not found, print 'Node not found' and stop.
- If found: set new_node->next = current->next and current->next = new_node.

```
while (temp != NULL) {
    if (temp->data == key) break;
    temp = temp->next;
}
if (temp == NULL) { printf("Node not found\n"); return; }
newNode->next = temp->next;
temp->next = newNode;
```

Case 5: Insertion Before a Given Node

- If list is empty (head == NULL): print 'List is empty'.
- If first node has the key (head->data == key): set newNode->next = head; head = newNode.
- Otherwise, use two pointers: prev = NULL, temp = head. Traverse until temp->data == key.
- Adjust links: prev->next = newNode and newNode->next = temp.

```
if (head->data == key) { newNode->next = head; head = newNode; }
else {
    prev = head; temp = head->next;
    while (temp != NULL && temp->data != key) { prev = temp; temp = temp-
>next; }
    if (temp == NULL) { printf("Key not found\n"); return; }
    newNode->next = temp;
    prev->next = newNode;
}
```

2.4 Traversal of Singly Linked List

Traversal means visiting every node of the list from HEAD to the last node.

- Initialize a pointer 'temp' pointing to HEAD of the list.
- Iterate until 'temp' becomes NULL.

- At each step: print temp->data, then move temp = temp->next.

```
struct node *temp = head;
while (temp != NULL) {
    printf("%d --->", temp->data);
    temp = temp->next;
}
```

 Key Point: Output example for list [12→20→28]: 12--->20--->28--->NULL

2.5 Deletion in Singly Linked List

Case 1: Deletion at Beginning

- If head == NULL: print 'List is empty' and stop.
- Store address of first node in temp = head.
- Move head pointer to next node: head = head->next.
- Free memory of deleted node: free(temp).

```
void deleteFirst() {
    if (head == NULL) { printf("List is empty\n"); return; }
    struct Node *temp = head;
    head = head->next;    // move head to next node
    free(temp);           // free old head
    temp = NULL;
}
```

Case 2: Deletion at End

- Case A — Only ONE node (head->next == NULL): set head = NULL and free the node.
- Case B — Multiple nodes: traverse using two pointers (prev, current) until current->next == NULL. Set prev->next = NULL and free current.

```
while (current->next != NULL) { prev = current; current = current->next; }
if (prev == NULL) { free(head); head = NULL; }  // only one node
else { prev->next = NULL; free(current); }
```

Case 3: Deletion at Specific Position

- If head == NULL: print 'List is empty'.
- If position == 1: move head = head->next and free old head.
- Traverse to (position-1)th node using temp.
- Store: del = temp->next; adjust: temp->next = del->next; free(del).

```
struct Node* nodeToDelete = temp->next;
```

```
temp->next = nodeToDelete->next;
free(nodeToDelete);
```

3. Doubly Linked List

3.1 Definition & Structure

 Definition: A Doubly Linked List (two-way linked list) is a linked list where each node contains THREE fields: PREV pointer (address of previous node), DATA (value), and NEXT pointer (address of next node).

Visual Structure: NULL ← [prev|A|next] ⇄ [prev|B|next] ⇄ [prev|C|next] ⇄ [prev|D|next] → NULL

- Allows traversal in BOTH directions (forward and backward).
- Easier to insert or delete nodes at any point in the list.
- HEAD node's prev = NULL; LAST node's next = NULL.

```
struct node {
    int data;
    struct node *next;   // pointer to next node
    struct node *prev;   // pointer to previous node
};

// Connecting 3 nodes (1, 2, 3):
one->next = two;    one->prev = NULL;
two->next = three;  two->prev = one;
three->next = NULL; three->prev = two;
```

3.2 Traversal of Doubly Linked List

Doubly linked list supports both forward (using next) and backward (using prev) traversal.

```
// Forward traversal
Node* temp = head;
while (temp != NULL) { printf("%d ", temp->data); temp = temp->next; }

// Backward traversal – first go to the last node
while (temp->next != NULL) { temp = temp->next; }
while (temp != NULL) { printf("%d ", temp->data); temp = temp->prev; }
```

3.3 Insertion in Doubly Linked List

Case 1: Insertion at Beginning

- Allocate space for new node and store data.
- Set ptr->prev = NULL and ptr->next = head.
- If head != NULL: set head->prev = ptr.
- Update head = ptr.

```
newnode->data = item;
if (head == NULL) {
    newnode->next = NULL; newnode->prev = NULL; head = newnode;
} else {
    newnode->prev = NULL;
    newnode->next = head;
    head->prev = newnode;
    head = newnode;
}
```

Case 2: Insertion at End

- Allocate space for new node and store data.
- If list is empty: make new node the head node.
- Otherwise: traverse to last node using temp, then set temp->next = ptr and ptr->prev = temp.

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

Case 3: Insertion Between Two Nodes

- Traverse to the node after which insertion is needed.
- Set newNode->next = temp->next and newNode->prev = temp.
- Update temp->next->prev = newNode and temp->next = newNode.

3.4 Deletion in Doubly Linked List

Case 1: Deletion at Beginning

Store first node in ptr. Move head to next node. Set head->prev = NULL. Free ptr.

```
void deleteBeginning() {
    if (head == NULL) { printf("List is empty\n"); return; }
    struct node *temp = head;
    head = head->next;
    if (head != NULL) head->prev = NULL;
```

```
    free(temp);
}
```

Case 2: Deletion at End

- Case A — Only ONE node: free(head); head = NULL.
- Case B — Multiple nodes: traverse to last node. Set temp->prev->next = NULL and free(temp).

```
if (head->next == NULL) { free(head); head = NULL; return; }
struct node *temp = head;
while (temp->next != NULL) { temp = temp->next; }
temp->prev->next = NULL;   // Disconnect last node
free(temp);
```

Case 3: Deletion Between Two Nodes

- Traverse to find the node to delete.
- Set del->prev->next = del->next and del->next->prev = del->prev.
- Free the deleted node.

4. Circular Linked List

4.1 Definition & Structure

 Definition: A Circular Linked List is a data structure where the LAST node points back to the FIRST node (HEAD) instead of NULL, forming a closed loop.

Visual Structure: → [1|next] → [2|next] → [3|next] → (back to 1)

- No node contains NULL pointer in the next field (except in empty list).
- Can be traversed starting from any node.
- Both singly and doubly linked lists can be made circular.

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

/* Connect as circular: three->next = one; */
```

4.2 Insertion in Circular Linked List

Case 1: Insertion at Beginning

- Allocate new node and store data.
- Traverse to find the LAST node (temp) where temp->next == head.

- Set ptr->next = head (new node points to old head).
- Set temp->next = ptr (last node now points to new node).
- Set head = ptr (new node becomes new head).

```
if (head == NULL) { newNode->next = newNode; head = newNode; }
else {
    struct Node *temp = head;
    while (temp->next != head) { temp = temp->next; }  // find last node
    newNode->next = head;
    temp->next = newNode;
    head = newNode;
}
```

Case 2: Insertion at End

- If list is empty: set newNode->next = newNode and head = newNode.
- Otherwise: traverse to last node (temp->next == head). Set temp->next = newNode and newNode->next = head.

```
void insertEnd(int value) {
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = value;
    if (head == NULL) { head = newNode; newNode->next = head; }
    else {
        temp = head;
        while (temp->next != head) { temp = temp->next; }
        temp->next = newNode;
        newNode->next = head;
    }
}
```

4.3 Deletion in Circular Linked List

Case 1: Deletion at Beginning

- Traverse using pointer ptr to reach the LAST node.
- Set ptr->next = head->next (last node now points to second node).
- Free the old head node.
- Set head = ptr->next (second node becomes new head).

```
if (head->next == head) { free(head); head = NULL; return; }
struct node *temp = head, *current = head;
while (current->next != head) { current = current->next; }
current->next = head->next;
head = head->next;
free(temp);
```

Case 2: Deletion at End

- Case A — ONE node (head->next == head): free(head); head = NULL.
- Case B — Multiple nodes: use two pointers (prev, ptr). Traverse until ptr->next == head. Set prev->next = head and free(ptr).

```
void deleteEnd() {
    if (head == NULL) { printf("List is empty\n"); return; }
    if (head->next == head) { free(head); head = NULL; return; }
    temp = head; prev = NULL;
    while (temp->next != head) { prev = temp; temp = temp->next; }
    prev->next = head;  // second-last now points to head
    free(temp);
}
```

4.4 Circular Doubly Linked List

 Definition: A Circular Doubly Linked List is a mixture of doubly linked list and circular linked list. Each node has TWO pointers (prev + next), and the last node's next points to the first node while the first node's prev points to the last node.

- Combines bi-directional traversal (doubly) with no NULL end (circular).
- Useful in applications like music players (prev/next song in loop).

5. Comparison: Types of Linked Lists

Feature	Singly Linked List	Doubly Linked List
Pointers per node	1 (next only)	2 (prev + next)
Traversal direction	Forward only	Forward + Backward
Memory usage	Less (one pointer)	More (two pointers)
Deletion efficiency	Need prev node explicitly	Easy — prev pointer available
Insertion at beginning	O(1)	O(1)
Insertion at end	O(n) — traverse to last	O(n) — traverse to last
NULL pointers	Last node's next = NULL	Head's prev = NULL, Last's next = NULL
Circular variant	Yes (last→head)	Yes (last→head, head←last)

6. Stack Implementation Using Linked List

6.1 Concept

 Definition: A Stack is a linear data structure following LIFO (Last In First Out). When implemented using a linked list, PUSH adds a node at the beginning and POP removes from the beginning.

- No overflow condition (unlike array) — limited only by heap memory.
- TOP of stack = HEAD of linked list.
- PUSH = Insert at beginning of linked list.
- POP = Delete from beginning of linked list.

 Remember: In linked list stack: newNode->next = top; top = newNode; (for PUSH)

6.2 Node Structure for Stack

```
struct node {
    int data;
    struct node *next;
};
struct node *top = NULL;
```

6.3 PUSH Operation (Insert at Beginning)

Algorithm: Allocate new node → store value → set newNode->next = top → set top = newNode.

```
void push(int value) {
    struct node *newNode = (struct node*)malloc(sizeof(struct node));
    if (newNode == NULL) { printf("Stack Overflow\n"); return; }
    newNode->data = value;
    newNode->next = top;   // new node points to current top
    top = newNode;         // new node becomes the top
}
```

6.4 POP Operation (Delete from Beginning)

Algorithm: If top == NULL → Underflow. Else, store top in temp, move top = top->next, print temp->data, free(temp).

```
void pop() {
    if (top == NULL) { printf("Stack Underflow\n"); return; }
    struct node *temp = top;
    printf("Deleted element: %d\n", top->data);
    top = top->next;
    free(temp);
}
```

6.5 PEEK & Display Operations

```
void peek() {
    if (top == NULL) printf("Stack is empty\n");
    else printf("Top element: %d\n", top->data);
}

void display() {
    if (top == NULL) { printf("Stack is empty\n"); return; }
    struct node *temp = top;
    while (temp != NULL) { printf("%d\n", temp->data); temp = temp->next; }
}
```

6.6 Example — Stack [10, 20, 30, 40, 50]

Stack (TOP = 50): 50 → 40 → 30 → 20 → 10 → NULL
After PUSH(60): 60 → 50 → 40 → 30 → 20 → 10 → NULL
After POP(): removes 60 → 50 → 40 → 30 → 20 → 10 → NULL

Advantages of Linked List Stack over Array Stack

- No fixed size limit — grows dynamically
- No overflow (unless memory is exhausted)
- Efficient memory usage — only allocated when needed
- No need to pre-define size

7. Queue Implementation Using Linked List

7.1 Concept

 Definition: A Queue is a linear data structure following FIFO (First In First Out). Using linked list: ENQUEUE (insert) at the REAR end, DEQUEUE (delete) from the FRONT end.

- Two pointers maintained: FRONT (head/start) and REAR (tail/end).
- FRONT points to the first element (dequeue from here).
- REAR points to the last element (enqueue here).
- No overflow condition unlike array queue.

Visual: FRONT → [22|→] → [-5|→] → [19|→] → [7|NULL] ← REAR

7.2 ENQUEUE Operation (Insert at Rear)

Algorithm: Allocate new node → store value → set newNode->next = NULL → if queue empty: front = rear = newNode → else: rear->next = newNode; rear = newNode.

```
void enqueue(int value) {
    struct node *newNode = (struct node*)malloc(sizeof(struct node));
    if (newNode == NULL) { printf("Queue Overflow\n"); return; }
```

```
    newNode->data = value;
    newNode->next = NULL;
    if (front == NULL) { front = rear = newNode; }
    else { rear->next = newNode; rear = newNode; }
    printf("%d inserted\n", value);
}
```

7.3 DEQUEUE Operation (Delete from Front)

Algorithm: If front == NULL → Underflow. Else store front in temp, print front->data, move front = front->next, if front == NULL then rear = NULL, free(temp).

```
void dequeue() {
    if (front == NULL) { printf("Queue Underflow\n"); return; }
    struct node *temp = front;
    printf("%d deleted\n", front->data);
    front = front->next;
    if (front == NULL) rear = NULL;  // queue became empty
    free(temp);
}
```

7.4 Display Queue

```
void display() {
    if (front == NULL) { printf("Queue is empty\n"); return; }
    struct node *temp = front;
    printf("Queue elements:\n");
    while (temp != NULL) { printf("%d ", temp->data); temp = temp->next; }
    printf("\n");
}
```

7.5 Example — Queue using Linked List

Initial Queue: FRONT → [3|→]→[4|→]→[5|→]→[6|→]→[7|→]→[8|NULL] ← REAR
After ENQUEUE(9): FRONT → [3]→[4]→[5]→[6]→[7]→[8]→[9|NULL] ← REAR
After DEQUEUE(): removes 3 → FRONT → [4]→[5]→[6]→[7]→[8]→[9|NULL] ← REAR

Advantages of Linked List Queue over Array Queue

- No fixed size — grows and shrinks dynamically
- No wasted memory (unlike circular queue complexity)
- No overflow unless memory exhausted
- Simple and clean implementation

8. Polynomial Manipulation Using Linked List

8.1 What is Polynomial Manipulation?

 Definition: A polynomial is an expression containing a sum of powers in one or more variables multiplied by coefficients. Example: $5x^2 + 2x - 3$

- Linked lists are ideal for representing polynomials since terms may have non-consecutive powers.
- Each NODE of the linked list represents ONE TERM of the polynomial.

8.2 Node Structure for Polynomial

Each node has THREE fields:

Field	Purpose
Coefficient	The numeric value of the term (e.g., 5 in $5x^2$)
Degree (Power)	The exponent/power of the variable x (e.g., 2 in $5x^2$)
Address (next)	Pointer to the next term node in the linked list

```
struct poly {
    int coeff;          // coefficient
    int degree;         // power of x
    struct poly *next; // pointer to next term
};
```

8.3 Example — Polynomial Representation

Polynomial: $4x^3 + 6x^2 + 10x + 6$

Linked List: poly → [4|3|→] → [6|2|→] → [10|1|→] → [6|0|NULL]

Polynomial: $5x^2 + 4x^1 + 2$

Linked List: poly → [5|2|→] → [4|1|→] → [2|0|NULL]

8.4 Addition of Two Polynomials

Rules for polynomial addition:

- Scan both polynomials from left to right simultaneously.
- If degrees are EQUAL: add the coefficients and create a new node in result list.
- If degree of P1 > degree of P2: copy term from P1 to result list.
- If degree of P1 < degree of P2: copy term from P2 to result list.
- Continue until both lists are exhausted.

Example 1: Add $5x^2 + 4x^1 + 2$ and $-5x^1 - 5$

List 1: [5|2] → [4|1] → [2|0]

List 2: [-5|1] → [-5|0]

```
Step 1: Degree of L1 = 2, L2 = 1 → L1 higher → copy [5|2] to result
Step 2: Both degree 1 → add: 4 + (-5) = -1 → add [-1|1] to result
Step 3: Both degree 0 → add: 2 + (-5) = -3 → add [-3|0] to result

Result: 5x² - x - 3
Result List: [5|2] → [-1|1] → [-3|0]
```

Example 2: Add $5x^3 + 4x^2 + 2$ and $5x^1 - 5$

List 1: [5|3] → [4|2] → [2|0]
List 2: [5|1] → [-5|0]
Result: $5x^3 + 4x^2 + 5x - 3$
Result List: [5|3] → [4|2] → [5|1] → [-3|0]

General Example: Add $(5x^4 + 6x^2 + 10x - 6)$ and $(7x^3 + 3x^2 + 2x + 7)$

Result: $5x^4 + 7x^3 + 9x^2 + 12x + 1$

Polynomial Addition Algorithm Summary

- Traverse both polynomial lists simultaneously using two pointers p1 and p2
- Compare degrees: equal → add coefficients; p1 higher → copy p1 term; p2 higher → copy p2 term
- Move the pointer(s) whose term was just processed
- Continue until both lists are exhausted
- Append any remaining terms from either list to the result

9. Quick Revision Summary

9.1 Key Terminologies

Term	Meaning
Head / HEAD	Pointer to the first node of the linked list
Tail / Last Node	The last node; its next = NULL (or HEAD in circular)
NULL	End marker — indicates no further node
Traversal	Visiting each node one by one from HEAD to last
Self-referential structure	Struct containing a pointer to its own type
malloc()	Dynamic memory allocation in C for new nodes
free()	Deallocate/release memory of a deleted node
LIFO	Last In First Out — Stack principle
FIFO	First In First Out — Queue principle

Term	Meaning
PUSH	Insert element at TOP of stack (beginning of linked list)
POP	Delete element from TOP of stack (beginning of linked list)
Enqueue	Insert element at REAR of queue (end of linked list)
Dequeue	Delete element from FRONT of queue (beginning of linked list)
Coefficient	Numeric value of a polynomial term
Degree / Power	Exponent of variable in a polynomial term

9.2 Operations Summary Table

Operation	Singly Linked List	Doubly Linked List
Insert at Beginning	newNode->next = head; head = newNode;	Also set head->prev = newNode; newNode->prev = NULL;
Insert at End	Traverse to last; last->next = newNode;	Also set newNode->prev = last;
Delete at Beginning	head = head->next; free(old head);	Also set head->prev = NULL;
Delete at End	Traverse; prev->next = NULL; free(last);	last->prev->next = NULL; free(last);
Traversal	while(temp!=NULL){...temp=temp->next;}	Forward and backward using prev/next

9.3 Important Practice Questions

Questions to Practice

- Write a C program to insert a node at beginning, end, and specific position in SLL.
- Write a C program to delete a node from beginning, end, and specific position in SLL.
- Explain the difference between Singly, Doubly, and Circular Linked List with diagrams.
- Implement Stack using Linked List with PUSH, POP, and PEEK operations.
- Implement Queue using Linked List with ENQUEUE and DEQUEUE operations.
- Add two polynomials ($5x^3 + 4x^2 + 2$) and ($5x - 5$) using linked list.
- What are the advantages of linked list over array? Explain with examples.
- Write a C function to traverse and display all elements of a circular linked list.
- How is deletion at beginning different in Singly vs Doubly Linked List?
- Add polynomials: ($3x^4 + 2x^2 + 5$) and ($4x^3 + x^2 + 3x - 2$).

10. References & Resources

10.1 Text Books

- "Data Structure: A Pseudo code approach with C" — Richard F. Gilberg & Behrouz A. Forouzan, Thomson Publication, 2nd Ed.
- "Data Structure in C" — A.M. Tanenbaum, Y. Langsam, M.J. Augenstein, PHI/Pearson, 7th Ed.

10.2 Reference Books

- "Data Structures with C" (Schaum's Series) — Seymour Lipschutz, Tata-McGraw-Hill, 1st Ed.
- "Fundamentals of Data Structure in C" — Horowitz, Sahani & Freed, Computer Science Press, 2nd Ed.
- "Data Structures Using C" — Reema Thareja, Oxford University Press, 2nd Ed.

10.3 Online Sources

- https://www.javatpoint.com/data-structure-introduction
- https://www.geeksforgeeks.org/data-structures/linked-list/
- https://www.geeksforgeeks.org/doubly-linked-list/
- https://www.geeksforgeeks.org/circular-linked-list/
- https://btechx.infinityfreeapp.com

— End of Unit II Notes —

BTechX

Data Structure (CS102) | Unit II: Tree

DATA STRUCTURE

CS102

UNIT III — Tree

Non Linear Data Structure

BTechX

Non-Linear Data Structure

Key Characteristics
▸ Data elements are NOT placed sequentially or linearly
▸ Multiple paths exist for an element to reach another element
▸ Data elements are connected to one or more elements
▸ Examples: Tree, Graph

Tree Data Structure

What is a Tree?

- **Definition:** A hierarchical, non-linear data structure that stores data on multiple levels.
- **Root Node:** The topmost node of the tree — every tree has exactly one root.
- **Children:** Branches in the tree are known as children.
- **Structure:** Tree consists of nodes (circles) connected by edges (lines).

Why Use Trees?

Advantages of Tree Data Structures

- Enable efficient data searching and sorting
- Simple insertion and deletion operations
- Hierarchical nature represents relationships in a structured manner
- Used in file systems for quick data retrieval and updates

Applications of Tree Data Structure

Application	
D	Organizes files/folders in a hierarchical structure
e	Navigate and organize web pages efficiently
s	Model decision-making processes (e.g., weather forecast)
c	Represent arithmetic expressions with operators
r	Describes routing paths in networks
i	Quickly organizes and accesses large datasets
p	
t	
i	
o	
n	

Basic Terminologies

Node Types

- **Root:** The topmost node; has no parent. Every tree has exactly one root.
- **Node / Key:** Fundamental unit containing data and references to child nodes.

- **Parent Node:** A tree node with one or more child nodes.
- **Child Node:** Any node that has a parent. Node 'c' is child of 'p' if directly connected via an edge.
- **Siblings:** Nodes sharing the same parent are known as sibling nodes.
- **Leaf Node:** A node with no children. Also called External Node or Terminal Node.
- **Internal Node:** A node with at least one child (Non-Terminal Node). Root is internal if tree has more than one node.
- **Subtree:** A portion of a tree that can be viewed as a complete tree in itself.

Measurements

- **Edge:** Connecting link between two nodes. A tree with N nodes has at most N-1 edges.
- **Degree:** Total number of children a node has. The highest degree in a tree is the Degree of Tree.
- **Path:** A sequence of consecutive nodes and edges from source to destination.
- **Length of Path:** Total number of nodes in a path.
- **Level:** Count of edges from root to that node. Root node is at Level 1.
- **Height:** Total edges from a leaf node to a particular node in the longest path. Height of all leaf nodes = 0.
- **Depth:** Total edges from root to a particular node. Depth of root = 0.
- **Forest:** A collection of disjoint trees. Created by removing the root of a tree.

```
Level Rule: If node is at Level 'L', its children are at Level 'L+1'
```

```
Height of root node = Height of the tree
```

Types of Tree

Tree Types Overview
▸ Binary Tree — Each node has at most 2 children
▸ Binary Search Tree (BST) — Ordered binary tree for searching in O(log n)
▸ AVL Tree — Self-balancing BST with balance factor -1, 0, or +1
▸ Heap Tree — Complete binary tree (Min-Heap / Max-Heap)

- General Tree — Nodes can have any number of children

Binary Tree

Definition

A Binary Tree is a type of tree data structure where each node can have a maximum of two child nodes — a left child node and a right child node.

Binary Tree Rules
▸ Each node can have either 0, 1, or 2 children
▸ Maximum degree of any node is at most two
▸ Node structure: Left Child Address | Data | Right Child Address

C Code — Node Structure

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

Types of Binary Tree

1. Strictly Binary Tree

Each non-terminal node must have both a non-empty left subtree and a non-empty right subtree.

- Example: Arithmetic expression trees with binary operators and operands, e.g., (A+B)*C

2. Complete Binary Tree

All levels are completely filled except possibly the last level, and the last level has all keys as left as possible.

- The leftmost side of the leaf node must always be filled first
- The last leaf node does not need to have a right sibling

```
At level L, maximum number of nodes = 2^L
```

3. Extended Binary Tree

Every null (missing) child is replaced by a special external node.

- Internal nodes → Original nodes with data (represented by circles)
- External nodes → Dummy nodes replacing NULL children (represented by squares)

4. Perfect Binary Tree

All internal nodes have exactly two children nodes. All leaf nodes are on the same level. All levels are completely filled.

5. Degenerate / Pathological Binary Tree

Each node contains only one child node — either on the left side or the right side of the tree.

Left Skewed Tree	
R	All children are RIGHT children
i	Looks like a linked list to the right
g	Maximum height for n nodes = n-1
h	
t	
S	
k	
e	
w	
e	
d	

T	
r	
e	
e	

6. Balanced Binary Tree

The difference between the height of the left and the right subtree for each node is either 0 or 1.

```
|Height(Left Subtree) - Height(Right Subtree)| ≤ 1  for every node
```

Representation of a Binary Tree

Array Representation	
L	Each node has: Left ptr | Data | Right ptr
i	Root has no parent pointer (NULL)
n	Leaf nodes have both pointers = NULL
k	Flexible — no wasted space
e	Better for dynamic insertions
d	Uses more memory per node (2 pointers)
L	
i	
s	
t	
R	
e	
p	

r	
e	
s	
e	
n	
t	
a	
t	
i	
o	
n	

Tree Traversal

Tree Traversal is a way to visit each node exactly once in a systematic manner. A key application is representing arithmetic expressions.

Three Types of Tree Traversal
▸ Preorder — Root, Left Child, Right Child (Prefix notation)
▸ Inorder — Left Child, Root, Right Child (Infix notation)
▸ Postorder — Left Child, Right Child, Root (Postfix notation)

Preorder Traversal (Root → Left → Right)

Visit the root node first, then recursively traverse the left subtree, then the right subtree.

```
Example result: A B D E C F G
```

Inorder Traversal (Left → Root → Right)

Recursively traverse the left subtree first, then visit the root, then traverse the right subtree. For a BST, this gives sorted order.

Example result: D B E A F C G

Postorder Traversal (Left → Right → Root)

Recursively traverse the left subtree, then the right subtree, then visit the root.

Example result: D E B F G C A

Traversal C Code Templates

```c
// Preorder
void printPreOrder(struct node *root) {
    if (root != NULL) {
        printf("%d ", root->data);
        printPreOrder(root->left);
        printPreOrder(root->right);
    }
}
```

Binary Expression Tree

An expression tree is a binary tree in which each internal node corresponds to an operator and each leaf node corresponds to an operand.

- Preorder traversal → Prefix expression (Polish notation)
- Inorder traversal → Infix expression (with parentheses)
- Postorder traversal → Postfix expression (Reverse Polish notation)

Example: (A+B)*(C+D) → Postorder: A B + C D + *

Binary Search Tree (BST)

Definition & Properties

A Binary Search Tree is a data structure that quickly allows us to maintain a sorted list of numbers.

BST Properties
▸ All nodes of LEFT subtree are LESS THAN the root node
▸ All nodes of RIGHT subtree are MORE THAN the root node
▸ Both subtrees of each node are also valid BSTs
▸ Search operation runs in O(log n) time

BST Insertion Algorithm

✓ Start at the root node
✓ Compare the value to insert with the current node's value
✓ If value is smaller → move to LEFT child
✓ If value is greater → move to RIGHT child
✓ Repeat until an empty spot is found
✓ Insert the new node at the empty spot

BST Deletion — Three Cases

Case	
A	Replace the leaf with NULL and free the space
c	Replace the node with its child; remove child from original position
t	Replace with inorder successor, remove successor from original position
i	
o	
n	

AVL Tree

Definition

AVL tree is a self-balancing Binary Search Tree in which each node maintains extra information called a balance factor, whose value is either -1, 0, or +1.

Named after inventors: Georgy Adelson-Velsky and Landis.

```
Balance Factor = Height(Left Subtree) - Height(Right Subtree)
```

AVL Tree Key Points
▸ Self-balancing property is maintained by the balance factor
▸ Controls height of BST by preventing skewed trees
▸ Rotation is performed when Balance Factor is other than -1, 0, or +1

Four Types of AVL Rotations

Rotation Type	
C	New node in LEFT subtree of LEFT subtree → Clockwise rotation (BF = 2)
o	New node in RIGHT subtree of RIGHT subtree → Anti-clockwise rotation (BF = -2)
n	New node in RIGHT subtree of LEFT subtree → RR on subtree + LL on full tree
d	New node in LEFT subtree of RIGHT subtree → LL on subtree + RR on full tree
i	
t	
i	
o	
n	
&	
A	
c	

t	
i	
o	
n	

LL Rotation (Clockwise)

When a node is inserted into the left subtree of the left subtree of A, making BF = +2. A single clockwise rotation is applied.

RR Rotation (Anti-clockwise)

When a node is inserted into the right subtree of the right subtree of A, making BF = -2. A single anti-clockwise rotation is applied.

LR Rotation (Double)

First apply RR rotation on the subtree, then apply LL rotation on the full tree (the first unbalanced ancestor node).

RL Rotation (Double)

First apply LL rotation on the subtree, then apply RR rotation on the full tree.

Practice Questions

Q1. Tree Terminologies

For the given tree (A as root, B and C as children of A, D and E as children of B, F and G as children of C, H as child of E, I and J as children of G), find:

- Siblings of Node D
- Degree of Node E
- Height of Tree
- Ancestors of Node H
- Descendants of Node C

- Level of Node G
- Depth of Tree
- Leaf Nodes
- Internal Nodes
- Left subtree of B

Q2. Tree Traversals

Find Preorder, Inorder, and Postorder traversals for the following trees:

- Tree 1: A → B (D, E→G), C (F)
- Tree 2: F → A (E, K→C), D (H→B, G)
- Tree 3: 30 → 20 (10→15,23, 25), 39 (35, 42)

Q3. Construct BST

Construct a Binary Search Tree with the following elements in the given sequence:

```
45, 15, 79, 90, 10, 55, 12, 20, 50
```

Q4. AVL Tree Construction

Construct AVL trees for the following sequences, applying rotations as needed:

```
Exercise 1: 40, 20, 10, 25, 30, 22, 50
```

```
Exercise 2: 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20
```

```
Exercise 3: 50, 20, 60, 10, 8, 15, 32, 46, 11, 48
```

Q5. Multiple Choice

How many rotations are required during the construction of an AVL tree if the following elements are added in the given sequence?

```
35, 50, 40, 25, 30, 60, 78, 20, 28
```

- 1. 2 left rotations, 2 right rotations
- 2. 2 left rotations, 3 right rotations
- 3. 3 left rotations, 2 right rotations

- 4. 3 left rotations, 1 right rotation

Construction of Binary Tree from Traversal

Given any two traversals (one of which must be Inorder), we can uniquely construct a binary tree.

From Preorder + Inorder

- First element of Preorder = Root
- Find Root in Inorder → splits into Left and Right subtrees
- Recursively repeat for each subtree

```
Preorder: A B D E C F   |   Inorder: D B E A F C
```

From Postorder + Inorder

- Last element of Postorder = Root
- Find Root in Inorder → splits into Left and Right subtrees
- Recursively repeat for each subtree

```
Postorder: 9 15 7 20 3   |   Inorder: 9 3 15 20 7
```

References

1. https://www.knowledgehut.com/blog/programming/types-of-trees-in-data-structure
2. https://www.codingninjas.com/studio/library/application-of-tree-in-data-structure
3. https://prepinsta.com/data-structures-algorithms/inorder-postorder-preorder-examples/
4. https://www.hackerearth.com/practice/algorithms/sorting/heap-sort/tutorial/
5. https://www.javatpoint.com/avl-tree

— End of Unit V Notes —

BTechX

UNIT – 4

Graph Algorithms

Detailed Notes

CS 102

1. Fundamentals of Graph

A graph G = (V, E) consists of a set of vertices V (also called nodes) and a set of edges E connecting pairs of vertices.

Types of Graphs

Graph Type	Description
Undirected Graph	Edges have no direction. If (u,v) is an edge, v is adjacent to u.
Directed Graph (Digraph)	Edges have direction. Edge (u,v) goes from u to v only.
Weighted Graph	Each edge has a weight/cost assigned to it.
Unweighted Graph	All edges have equal weight (or no weight).
Cyclic Graph	Contains at least one cycle (path starting and ending at same vertex).
Acyclic Graph	No cycles. DAG = Directed Acyclic Graph.
Complete Graph	Every pair of distinct vertices is connected by a unique edge.
Connected Graph	There is a path between every pair of vertices.
Tree	A connected, acyclic graph with n vertices and n-1 edges.

Key Terms

- **Vertex (Node):** A fundamental unit represented by a point.
- **Edge:** A connection between two vertices.
- **Degree:** Number of edges incident to a vertex. In directed graphs: in-degree and out-degree.
- **Path:** A sequence of vertices where each adjacent pair is connected by an edge.
- **Cycle:** A path that starts and ends at the same vertex.
- **Neighbour:** Vertices connected by an edge to a given vertex.

- **Self-loop:** An edge from a vertex to itself.
- **Multi-edge:** Multiple edges between the same pair of vertices.

2. Representation of Graphs

Adjacency Matrix

A 2D array of size V × V where matrix[i][j] = 1 if an edge exists between vertices i and j, else 0. For weighted graphs, matrix[i][j] = weight, or ∞ if no edge.

```
matrix[i][j] = 1  (edge exists)   |   matrix[i][j] = 0  (no edge)
```

Example — Undirected Graph (4 vertices)

```
     0  1  2  3
   +------------
0  | 0  1  0  1
1  | 1  0  1  0
2  | 0  1  0  1
3  | 1  0  1  0

Matrix[0][1] = 1  →  edge exists between vertex 0 and vertex 1
```

Python Implementation

```
class GraphMatrix:
    def __init__(self, vertices):
        self.V = vertices
        self.matrix = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v, weight=1):
        self.matrix[u][v] = weight
        self.matrix[v][u] = weight  # undirected

    def display(self):
        for row in self.matrix:
            print(row)
```

Adjacency Matrix — Complexity
▸ Adding edge: O(1)
▸ Checking edge: O(1)
▸ Finding all neighbours: O(V)
▸ Space Complexity: $O(V^2)$
▸ Best for: Dense graphs (many edges), fast edge lookup, small V

Adjacency List

Each vertex maintains a list of its neighbouring vertices. Much more space-efficient for sparse graphs.

Example

```
0 -> [1, 3]
1 -> [0, 2]
2 -> [1, 3]
3 -> [0, 2]
```

Python Implementation

```
from collections import defaultdict

class GraphList:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def add_edge(self, u, v, weight=1):
        self.graph[u].append((v, weight))
        self.graph[v].append((u, weight))  # undirected

    def display(self):
        for v in self.graph:
            print(f"{v} -> {self.graph[v]}")
```

Adjacency List — Complexity

- Adding edge: O(1)
- Checking edge: O(degree) = O(V) worst case
- Finding all neighbours: O(degree) = O(V) worst case
- Space Complexity: O(V + E)
- Best for: Sparse graphs (few edges), iterating neighbours efficiently

Comparison: Matrix vs List

Criteria	Adjacency Matrix	Adjacency List
Space	$O(V^2)$	O(V + E)
Check Edge	O(1)	O(degree)
Iterate Neighbours	O(V)	O(degree)
Best For	Dense graphs	Sparse graphs

3. Path Matrix (Reachability Matrix)

A path matrix P (or reachability matrix) of size V × V where P[i][j] = 1 if there exists a path from vertex i to vertex j, else 0.

The path matrix can be computed using the Floyd-Warshall algorithm by repeatedly updating reachability through intermediate vertices.

Floyd-Warshall Algorithm for Path Matrix

```
def path_matrix(graph, V):
    P = [[0]*V for _ in range(V)]

    # Initialize with direct edges
    for i in range(V):
        for j in range(V):
            if graph[i][j]:
                P[i][j] = 1

    # Find transitive closure (all reachable pairs)
    for k in range(V):   # intermediate vertex
        for i in range(V):   # source
            for j in range(V):   # destination
                P[i][j] = P[i][j] or (P[i][k] and P[k][j])

    return P

# Time: O(V³)    Space: O(V²)
```

4. Breadth First Search (BFS)

BFS explores a graph level by level, visiting all neighbours before moving to the next level. It uses a Queue (FIFO).

Algorithm

```
BFS(G, start):
    visited[] = false for all vertices
    queue = empty
    visited[start] = true
    queue.enqueue(start)

    while queue is not empty:
        v = queue.dequeue()
        visit(v)                    # process vertex v
        for each neighbour u of v:
            if not visited[u]:
                visited[u] = true
                queue.enqueue(u)
```

Properties

BFS Properties

- Time Complexity: O(V + E)
- Space Complexity: O(V)
- Produces shortest path in unweighted graphs
- Level-order traversal
- Uses Queue (FIFO) data structure

Example

Graph: 0-1-2, 0-3, 1-4, 2-4, 3-4

```
BFS from vertex 0:
  Level 0:  0
  Level 1:  1, 3
  Level 2:  2, 4
  Order:    0 -> 1 -> 3 -> 2 -> 4
```

Complete Python Implementation

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue   = deque([start])
    visited.add(start)
    result  = []

    while queue:
        vertex = queue.popleft()
        result.append(vertex)
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
    return result

graph = {0:[1,3], 1:[0,2,4], 2:[1,4], 3:[0,4], 4:[1,2,3]}
print(bfs(graph, 0))   # Output: [0, 1, 3, 2, 4]
```

5. Depth First Search (DFS)

DFS explores a graph as deep as possible along each branch before backtracking. It uses a Stack (LIFO) or recursion.

Algorithm

```
DFS(G, start):
```

```
    visited[] = false for all vertices
    stack = empty
    stack.push(start)

    while stack is not empty:
        v = stack.pop()
        if not visited[v]:
            visited[v] = true
            visit(v)                          # process vertex v
            for each neighbour u of v (reverse order):
                if not visited[u]:
                    stack.push(u)
```

Properties

DFS Properties

- Time Complexity: O(V + E)
- Space Complexity: O(V) (recursion stack or explicit stack)
- Backtracking-based exploration
- Useful for cycle detection, path finding, topological sort
- Uses Stack (LIFO) data structure (or recursion)

Example

Graph: 0-1-2, 0-3, 1-4, 2-4, 3-4

```
DFS from vertex 0 (using stack):
  Visit 0, push neighbours: push 1, 3
  Pop 3,  push its neighbours: 4
  Pop 4,  push its neighbours: 2, 1 (already visited)
  Pop 2,  push its neighbours: (none new)
  Order:  0 -> 3 -> 4 -> 2 -> 1
```

Complete Python Implementation

```
# Iterative DFS
def dfs_iterative(graph, start):
    visited = set()
    stack   = [start]
    result  = []
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            result.append(vertex)
            for neighbour in reversed(graph[vertex]):
                if neighbour not in visited:
                    stack.append(neighbour)
    return result

# Recursive DFS
```

```
def dfs_recursive(graph, vertex, visited, result):
    visited.add(vertex)
    result.append(vertex)
    for neighbour in graph[vertex]:
        if neighbour not in visited:
            dfs_recursive(graph, neighbour, visited, result)

graph = {0:[1,3], 1:[0,2,4], 2:[1,4], 3:[0,4], 4:[1,2,3]}
print(dfs iterative(graph, 0))   # Output: [0, 1, 2, 4, 3]
```

6. Spanning Trees

A spanning tree of a graph G is a subgraph that is a tree and contains all vertices of G.

Spanning Tree: V vertices | Exactly V-1 edges | Connected & Acyclic

Properties

Spanning Tree Properties
▸ Contains ALL V vertices of the original graph ▸ Has exactly V-1 edges ▸ Is a connected, acyclic graph
▸ Every connected graph has at least one spanning tree ▸ Can be found using BFS or DFS traversal

Spanning Tree from BFS / DFS

When traversing a graph with BFS or DFS, the edges used to discover new vertices (tree edges) form a spanning tree. Tree edges connect a vertex to a previously undiscovered vertex.

7. Minimum Spanning Trees (MST)

A Minimum Spanning Tree in a weighted, connected graph is a spanning tree with the minimum total edge weight.

Kruskal's Algorithm

A greedy algorithm that sorts all edges by weight and adds the smallest edge that does not form a cycle.

```
Kruskal(G):
```

```
    sort all edges by weight (ascending)
    for each edge (u, v) in sorted order:
        if u and v are in different components:
            add edge (u, v) to MST
            union the two components

  Time Complexity: O(E log E)  or  O(E log V)
  Space:           O(V + E)
```

Prim's Algorithm

A greedy algorithm that grows the MST from a starting vertex, always adding the minimum weight edge connecting a visited vertex to an unvisited vertex.

```
Prim(G, start):
    visited[start] = true
    result = empty
    while not all vertices visited:
        find minimum weight edge (u, v)
            where u is visited, v is not
        add edge (u, v) to result
        mark v as visited

  Time Complexity: O(E log V)  with priority queue
  Space:           O(V + E)
```

Example

Graph: 4 vertices, edges: 0-1(10), 0-2(6), 0-3(5), 1-2(3), 2-3(4)

```
Kruskal's MST:
  Step 1: Pick 1-2 (weight 3)  →  add
  Step 2: Pick 2-3 (weight 4)  →  add
  Step 3: Pick 0-3 (weight 5)  →  add
  MST edges: (1,2), (2,3), (0,3)
  Total weight: 3 + 4 + 5 = 12
```

8. Dijkstra's Shortest Path Algorithm

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

Algorithm

```
Dijkstra(G, source):
    dist[]  = infinity for all vertices
    dist[source] = 0
    visited[] = false for all vertices
    priority_queue = min-heap with (distance, vertex)

    while priority_queue is not empty:
```

```
        (d, u) = extract_min(priority_queue)
        if visited[u]: continue
        visited[u] = true
        for each neighbour v of u with weight w:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                add (dist[v], v) to priority_queue

    return dist[]

  Time: O((V + E) log V) with priority queue
  Space: O(V)
```

Example

Graph: 0→1(4), 0→2(2), 1→2(1), 1→3(5), 2→3(8), 2→4(10), 3→4(2)

```
Dijkstra from source 0:
  Initial:           dist = [0, ∞,  ∞,  ∞,  ∞]
  After processing 0: dist = [0, 4,  2,  ∞,  ∞]
  After processing 2: dist = [0, 3,  2, 10, 12]
  After processing 1: dist = [0, 3,  2,  8, 12]
  After processing 3: dist = [0, 3,  2,  8, 10]

  Shortest paths from 0:
    0→0 = 0    0→2 = 2    0→1 = 3    0→3 = 8    0→4 = 10
```

Important Notes — Dijkstra

- Does NOT work with negative edge weights
- Greedy algorithm — makes locally optimal choice at each step
- Produces shortest paths in weighted graphs with non-negative weights
- Equivalent to BFS when all edge weights are equal

9. Bellman-Ford Algorithm

The Bellman-Ford algorithm finds shortest paths from a source to all vertices, handling graphs with negative edge weights. It can also detect negative weight cycles.

Algorithm

```
Bellman-Ford(G, source):
    dist[] = infinity for all vertices
    dist[source] = 0

    # Relax all edges V-1 times
    repeat V-1 times:
        for each edge (u, v) with weight w:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
```

```
    # Check for negative weight cycles
    for each edge (u, v) with weight w:
        if dist[u] + w < dist[v]:
            return "Negative cycle detected"

    return dist[]

Time Complexity: O(V × E)
Space Complexity: O(V)
```

Why V-1 Iterations?

In the worst case, the shortest path between any two vertices can have at most V-1 edges (simple path without cycles). Each iteration finds shortest paths with at most one more edge than before.

Example

Graph: 0→1(4), 0→2(2), 1→2(-3), 1→3(5), 2→3(2)

```
Bellman-Ford from source 0:
  Initial:         dist = [0, ∞, ∞, ∞]
  After iteration 1: dist = [0, 4, 2, 4]
  After iteration 2: dist = [0, 1, 2, 4]   (negative edge improved path)
  No further improvement → no negative cycle

  Shortest paths: 0→0=0, 0→2=2, 0→1=1 (via 2), 0→3=4 (via 2)
```

Comparison: Dijkstra vs Bellman-Ford

Criteria	Dijkstra	Bellman-Ford
Time Complexity	O((V+E) log V)	O(V × E)
Edge Weights	Non-negative only	Can be negative
Negative Cycle	Not detected	Detected
Algorithm Type	Greedy	Dynamic Programming

10. Summary & Quick Reference

Algorithm	Time Complexity	Space	Use Case
BFS	O(V + E)	O(V)	Shortest path (unweighted)
DFS	O(V + E)	O(V)	Cycle detection, path finding
Kruskal (MST)	O(E log E)	O(V + E)	Minimum spanning tree
Prim (MST)	O(E log V)	O(V + E)	Minimum spanning tree

Dijkstra	O((V+E) log V)	O(V)	Shortest path (non-neg weights)
Bellman-Ford	O(V × E)	O(V)	Shortest path (any weights)
Floyd-Warshall	$O(V^3)$	$O(V^2)$	All-pairs shortest path

Data Structure (CS102) | Unit V: Sorting and Searching

DATA STRUCTURE

CS102

UNIT V — Sorting and Searching

Sorting Algorithms & Searching Techniques

BTechX

1. Introduction to Sorting

1.1 What is Sorting?

Sorting is the process of rearranging a sequence of elements in a specific order — either ascending or descending. It is one of the most fundamental operations in computer science.

Formally: Given a sequence of n numbers a_1 , a_2 , …, a□, sorting produces a permutation a_1 ', a_2 ', …, a□' such that:

$a_1' \leq a_2' \leq \cdots \leq a\square'$ (non-decreasing order)

□ **Key Point:** Processed/rearranged data helps in faster searching, better performance, and cleaner data management.

1.2 Sorting Example

Original list: 10, 30, 20, 80, 70, 10, 60, 40, 70

Non-decreasing: 10, 10, 20, 30, 40, 60, 70, 70, 80

Non-increasing: 80, 70, 70, 60, 40, 30, 20, 10, 10

1.3 Issues in Sorting

- How to rearrange a given set of data?
- Which data structures are most suitable to store data prior to sorting?
- How fast can sorting be achieved? (Time Complexity)
- How to sort in a memory-constrained situation? (Space Complexity)
- How to sort various types of data? (Generalization)

1.4 Categories of Sorting Algorithms

Category	Basic Idea	Examples
Sorting by Comparison	Data items are compared with each other to find the correct position.	Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort
Sorting by Distribution	No key comparison. Items are distributed over auxiliary storage based on digits/radix, then grouped.	Radix Sort, Counting Sort, Hash-based Sort

2. Bubble Sort

2.1 How It Works

Bubble Sort is a simple comparison-based sorting algorithm. It works by repeatedly comparing and swapping adjacent elements if they are in the wrong order. In every pass, the largest unsorted element 'bubbles up' to its correct position at the end.

- In Pass 1: consider index 0 to n-1
- In Pass 2: consider index 0 to n-2
- In Pass 3: consider index 0 to n-3
- In Pass n-1: consider index 0 to 1

 Key Point: In every pass, the largest element sinks to the bottom (rightmost). Total passes needed = n-1.

2.2 Step-by-Step Example

Input: 3 12 -5 6 72 21 -7 45

Pass 1:

```
3   12  -5   6  72  21  -7  45
3   -5  12   6  72  21  -7  45   ← swap(12, -5)
3   -5   6  12  72  21  -7  45   ← swap(12, 6)
3   -5   6  12  72  21  -7  45
3   -5   6  12  21  72  -7  45   ← swap(72, 21)
3   -5   6  12  21  -7  72  45   ← swap(72, -7)
3   -5   6  12  21  -7  45  72   ← swap(72, 45)  → 72 placed ✓
```

Pass 2:

```
-5   3   6  12  21  -7  45  72
-5   3   6  12  -7  21  45  72   ← swap(21, -7)
Final after Pass 2: -5  3  6  12  -7  21  45  72
```

2.3 C Code — Bubble Sort

```
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

void bubble_sort(int x[], int n) {
    int i, j;
    for (i = n-1; i > 0; i--)
        for (j = 0; j < i; j++)
            if (x[j] > x[j+1])
                swap(&x[j], &x[j+1]);
}
```

Output Example: -45 89 -65 87 0 3 -23 19 56 21 76 -50

After sorting: -65 -50 -45 -23 0 3 19 21 56 76 87 89

2.4 Complexity Analysis

Case	Time Complexity	Reason
Best Case	O(n)	Array already sorted (with optimization flag)
Average Case	$O(n^2)$	Partial comparisons and swaps
Worst Case	$O(n^2)$	Array sorted in reverse order
Space Complexity	O(1)	In-place sorting, no extra space needed

3. Selection Sort

3.1 How It Works

Selection Sort divides the array into a sorted portion and an unsorted portion. In each pass, it finds the minimum element from the unsorted portion and swaps it with the first element of the unsorted portion.

- Step 1: Find the smallest element (mval) in x[k … size-1]
- Step 2: Swap it with x[k]
- Step 3: Increment k and repeat

 Key Point: Selection Sort makes exactly n-1 swaps regardless of input — better than Bubble Sort in terms of swaps.

3.2 Step-by-Step Example

Input: -17 12 -5 6 142 21 3 45

```
Pass 1: Minimum = -17 (already in place)     → -17  12  -5  6  142  21   3
45
Pass 2: Minimum = -5, swap with 12           → -17  -5  12  6  142  21   3
45
Pass 3: Minimum =  3, swap with 12           → -17  -5   3  6  142  21  12
45
Pass 4: Minimum =  6 (already in place)      → -17  -5   3  6  142  21  12
45
Pass 5: Minimum = 12, swap with 142          → -17  -5   3  6   12  21 142
45
Pass 6: Minimum = 21 (already in place)      → -17  -5   3  6   12  21 142
45
Pass 7: Minimum = 45, swap with 142          → -17  -5   3  6   12  21  45
142
```

3.3 C Code — Selection Sort

```
int findMinLoc(int x[], int k, int size) {
    int j, pos = k;
    for (j = k+1; j < size; j++)
        if (x[j] < x[pos]) pos = j;
    return pos;
}

void selectionSort(int x[], int size) {
    int k, m, temp;
    for (k = 0; k < size-1; k++) {
        m = findMinLoc(x, k, size);
        temp = x[k];
        x[k] = x[m];
        x[m] = temp;
    }
}
```

3.4 Complexity Analysis

Case	Time Complexity	Space Complexity
Best Case	$O(n^2)$	O(1)
Average Case	$O(n^2)$	O(1)
Worst Case	$O(n^2)$	O(1)

4. Insertion Sort

4.1 How It Works

Insertion Sort builds the sorted array one element at a time. It picks each element and inserts it into its correct position in the already-sorted portion of the array (to the left).

- Pick the element at index i (starting from i=1)
- Compare it with elements to its left
- Shift elements that are greater than the picked element one position to the right
- Insert the picked element at the correct position

 Key Point: Insertion Sort is like sorting a hand of playing cards — you pick one card at a time and insert it in the right place.

4.2 Step-by-Step Example

Input: 54 26 93 17 77 31 44 55 20

```
[54]  26  93  17  77  31  44  55  20  ← 54 is sorted list of 1
[26 54]  93  17  77  31  44  55  20   ← inserted 26
[26 54 93]  17  77  31  44  55  20    ← inserted 93
[17 26 54 93]  77  31  44  55  20     ← inserted 17
[17 26 54 77 93]  31  44  55  20      ← inserted 77
[17 26 31 54 77 93]  44  55  20       ← inserted 31
[17 26 31 44 54 77 93]  55  20        ← inserted 44
[17 26 31 44 54 55 77 93]  20         ← inserted 55
[17 20 26 31 44 54 55 77 93]          ← inserted 20 ✓
```

4.3 C Code — Insertion Sort

```
void insertionSort(int list[], int size) {
    int i, j, item;
    for (i = 1; i < size; i++) {
        item = list[i];
        for (j = i-1; (j >= 0) && (list[j] > item); j--)
            list[j+1] = list[j];
```

```
            list[j+1] = item;
        }
    }
```

4.4 Complexity Analysis

Case	Time Complexity	Reason
Best Case	O(n)	Already sorted array
Average Case	$O(n^2)$	Partial comparisons and shifts
Worst Case	$O(n^2)$	Reverse sorted array
Space Complexity	O(1)	In-place, no extra memory

5. Quick Sort

5.1 How It Works

Quick Sort is a divide-and-conquer sorting algorithm. It selects a pivot element and partitions the array into two sub-arrays: elements less than or equal to the pivot (left), and elements greater than the pivot (right). It then recursively sorts both sub-arrays.

- Step 1: Select a pivot (usually first element)
- Step 2: Partition array — smaller elements to the left, larger to the right
- Step 3: Recursively apply Quick Sort to left and right sub-arrays
- Step 4: Combine — no explicit merge needed!

 Key Point: Quick Sort is generally the fastest in practice with average time complexity O(n log n). It is an in-place algorithm.

5.2 Partition Logic

Given pivot = a[l] (leftmost), two pointers i and j move inward:

- i moves right until it finds an element greater than pivot
- j moves left until it finds an element less than or equal to pivot
- If i < j: swap a[i] and a[j], then continue
- If i >= j: swap pivot with a[j] — pivot is now in its final position

5.3 Step-by-Step Example

Input: 45 -56 78 90 -3 -6 123 0 -3 45 69 68

```
Pivot = 45:
→ Left part:  -6  -56  -3   0  -3
→ Pivot 45 placed at correct position
→ Right part: 123  90  78  45  69  68

Recursively sort left:  -56  -6  -3  -3  0
Recursively sort right: 45  68  69  78  90  123

Final: -56  -6  -3  -3  0  45  45  68  69  78  90  123
```

5.4 C Code — Quick Sort

```
int partition(int a[], int l, int r) {
    int pivot = a[l], i = l, j = r+1, t;
    while (1) {
        do { ++i; } while (a[i] <= pivot && i <= r);
        do { --j; } while (a[j] > pivot);
        if (i >= j) break;
        t = a[i]; a[i] = a[j]; a[j] = t;
    }
    t = a[l]; a[l] = a[j]; a[j] = t;
    return j;
}

void quickSort(int a[], int l, int r) {
    if (l < r) {
        int j = partition(a, l, r);
        quickSort(a, l, j-1);
        quickSort(a, j+1, r);
    }
}
```

5.5 Complexity Analysis

Case	Time Complexity	Reason
Best Case	O(n log n)	Pivot always divides array into equal halves
Average Case	O(n log n)	Random pivot selection
Worst Case	$O(n^2)$	Sorted/reverse-sorted array (bad pivot choice)
Space Complexity	O(log n)	Recursive call stack

6. Merge Sort

6.1 How It Works

Merge Sort is a classic divide-and-conquer algorithm credited to John von Neumann. It recursively splits the array into halves until each sub-array has 1 element, then merges them back in sorted order.

- Step 1: If the list has 0 or 1 element — it is already sorted (base case)
- Step 2: If the list has more than 1 element — split it into two halves
- Step 3: Recursively apply Merge Sort to each half
- Step 4: Merge the two sorted halves into a single sorted list

 Key Point: Merge Sort guarantees O(n log n) time in ALL cases — best, average, and worst. It is a stable sort.

6.2 Visual Tree (Example)

Input: 38 27 43 3 9 82 10

```
                [38 27 43 3 9 82 10]
               /                    \
       [38 27 43 3]                [9 82 10]
       /           \               /        \
   [38 27]       [43 3]        [9 82]       [10]
   /     \       /    \        /    \
 [38]   [27]   [43]   [3]    [9]   [82]
   \     /       \     /       \     /
  [27 38]       [3 43]        [9 82]
       \           /               \
       [3 27 38 43]              [9 10 82]
              \                    /
           [3  9  10  27  38  43  82]  ← Final Sorted
```

6.3 Merge Operation

The core of Merge Sort is the merge step. Given two sorted arrays L and R, we compare their front elements and pick the smaller one into the result array repeatedly.

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
```

```
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```

6.4 Complexity Analysis

Case	Time Complexity	Space Complexity
Best Case	O(n log n)	O(n)
Average Case	O(n log n)	O(n)
Worst Case	O(n log n)	O(n)

 Note: Unlike Quick Sort, Merge Sort requires O(n) extra space for the temporary arrays. But it is stable and guaranteed O(n log n) always.

7. Comparison of Sorting Algorithms

Algorithm	Best	Average	Worst	Space	Stable?
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes

 Note: A sorting algorithm is 'stable' if it preserves the relative order of equal elements.

8. Searching Algorithms

8.1 What is Searching?

Searching is the process of finding a specific element (called the key) in a collection of data. The result is either the position (index) of the element, or an indication that it was not found.

Type	Description	Pre-requisite
Linear Search	Check each element one by one from start to end until key is found.	None — works on any array
Binary Search	Divide and conquer. Compare key with the middle element and reduce the search range by half each time.	Array must be sorted

8.2 Linear Search

Linear Search (also called Sequential Search) scans through the array element by element until the target is found or the end is reached.

 Note: Input: Array A[ ] of size n, Search key K

```
Step 1: Start from index i = 0
Step 2: Compare A[i] with K
          If A[i] == K → return i (found at position i)
          If A[i] != K → set i = i + 1
Step 3: Repeat Step 2 while i < n
Step 4: If not found → return -1
```

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++)
        if (arr[i] == key) return i;
    return -1;
}
```

Case	Time Complexity
Best Case	O(1) — Key found at first position
Average Case	O(n) — Key found somewhere in middle
Worst Case	O(n) — Key not found or at last position

8.3 Binary Search

Binary Search works on a sorted array. It repeatedly divides the search interval in half. If the key equals the middle element, it's found. If the key is smaller, search the left half; if larger, search the right half.

 Note: Input: Sorted Array A[ ], Low = 0, High = n-1, Search Key K

```
Step 1: Set LOW = 0, HIGH = n-1
Step 2: While LOW <= HIGH:
          Set MID = (LOW + HIGH) / 2
          If A[MID] == K → return MID (found)
          If A[MID] < K  → Set LOW = MID + 1 (search right half)
          If A[MID] > K  → Set HIGH = MID - 1 (search left half)
Step 3: If LOW > HIGH → Key not found, return -1
```

```
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Binary Search Example: Search key = 40 in sorted array [10, 20, 30, 40, 50, 60, 70]

```
Step 1: LOW=0, HIGH=6 → MID=3 → A[3]=40 == 40 → FOUND at index 3 ✓

Search key = 25:
Step 1: LOW=0, HIGH=6 → MID=3 → A[3]=40 > 25 → HIGH = 2
Step 2: LOW=0, HIGH=2 → MID=1 → A[1]=20 < 25 → LOW = 2
Step 3: LOW=2, HIGH=2 → MID=2 → A[2]=30 > 25 → HIGH = 1
Step 4: LOW=2 > HIGH=1 → NOT FOUND → return -1
```

Case	Time Complexity
Best Case	O(1) — Key is at the middle
Average Case	O(log n)
Worst Case	O(log n) — Key not present

 Key Point: Binary Search is much faster than Linear Search for large sorted arrays. For n=1,000,000 elements, Linear takes up to 1,000,000 steps, Binary takes at most 20 steps!

8.4 Linear vs Binary Search

Feature	Linear Search	Binary Search
Data Requirement	Works on any array	Array must be sorted
Time Complexity	O(n)	O(log n)
Comparisons	Up to n comparisons	At most $\log_2$ (n) comparisons
Implementation	Simple	Slightly complex
Best Use	Small or unsorted data	Large sorted datasets

9. Quick Revision Summary

9.1 Key Terminologies

Term	Meaning
Sorting	Rearranging elements in a specific order (ascending/descending)
Searching	Finding a specific element (key) in a data collection
Pivot	The selected reference element used in Quick Sort partitioning
Pass	One complete scan through the unsorted portion of the array
Stable Sort	Preserves the relative order of equal elements
In-place Sort	Uses O(1) extra space — modifies the original array
Divide & Conquer	Strategy of breaking problem into sub-problems (Quick Sort, Merge Sort)
Linear Search	Sequential element-by-element search — O(n)
Binary Search	Halving-based search on sorted array — O(log n)
Overflow	No room to insert (relevant in array-based structures)

9.2 Practice Questions

Important Questions

- What is sorting? Explain the types of sorting algorithms with examples.
- Trace Bubble Sort on: 64, 34, 25, 12, 22, 11, 90. Show all passes.
- Write and explain the Selection Sort algorithm with C code.

- Differentiate between Bubble Sort, Selection Sort, and Insertion Sort.
- Explain Quick Sort with the partition algorithm. Trace on: 7, 12, 1, -2, 0, 15, 4, 11, 9.
- Draw the Merge Sort tree for: 38, 27, 43, 3, 9, 82, 10.
- Compare all sorting algorithms based on time and space complexity.
- Write and explain Linear Search algorithm with C code.
- Trace Binary Search for key=40 in: 10, 20, 30, 40, 50, 60, 70.
- When would you prefer Binary Search over Linear Search? Justify with complexity analysis.

10. References & Resources

10.1 Text Books

- "Data Structure: A Pseudo code approach with C" — Richard F. Gilberg & Behrouz A. Forouzan, Thomson Publication, 2nd Ed.
- "Data Structure in C" — A.M. Tanenbaum, Y. Langsam, M.J. Augenstein, PHI/Pearson, 7th Ed.

10.2 Reference Books

- "Data Structures with C" (Schaum's Series) — Seymour Lipschutz, Tata-McGraw-Hill, 1st Ed.
- "Fundamentals of Data Structure in C" — Horowitz, Sahani & Freed, Computer Science Press, 2nd Ed.
- "Data Structures Using C" — Reema Thareja, Oxford University Press, 2nd Ed.

10.3 Online Sources

- https://nptel.ac.in/courses/106104128
- https://www.geeksforgeeks.org/sorting-algorithms/
- https://www.javatpoint.com/binary-search
- https://btechx.infinityfreeapp.com

— End of Unit V Notes —

BTechX