DATA STRUCTURES AND ALGORITHMS

STUDY MATERIAL

UNIT I: LISTS

UNIT I

LISTS

1.1 Abstract Data Types (ADTs)

An **Abstract Data Type (ADT)** is a mathematical model for a data type, defined by its behaviour (operations) from the user's point of view, rather than its implementation. It specifies *what* operations are performed but not *how* they are performed.

Properties of ADT

- Encapsulation: hides internal implementation from the user.
- Examples of ADTs: List, Stack, Queue, Tree, Graph.
- An ADT defines: Data stored + Operations on data + Error conditions.
- Separation of interface from implementation enables code reuse and flexibility.

1.2 List ADT

A **List** is an ordered sequence of elements. The List ADT supports:

Operation	Description	Time Complexity
insert(pos, val)	Insert element at given position	O(n) array / O(1) linked
delete(pos)	Remove element at given position	O(n) array / O(1) linked
find(val)	Search for an element	O(n)
get(pos)	Access element at position	O(1) array / O(n) linked
size()	Return number of elements	O(1)

isEmpty()	Check if list is empty	O(1)

1.3 Array-Based Implementation

In an **array-based list**, elements are stored in contiguous memory locations. A variable tracks the current size.

Feature	Description
Storage	Contiguous memory block
Random Access	O(1) – direct index access
Insertion/Deletion	O(n) – shifting of elements required
Size	Fixed at creation (static array) or dynamic resizing
Memory	Pre-allocated; may waste space or need reallocation

```
struct ArrayList {
    int data[MAX_SIZE];
    int size;
};
// Insert at position i: shift elements right, then insert
// Delete at position i: shift elements left
```

1.4 Linked List Implementation

A **linked list** stores elements in nodes, where each node contains data and a pointer to the next node. Elements are not stored contiguously in memory.

1.4.1 Singly Linked List

Each node has: **data** + **next pointer**. Traversal is only in one direction.

```
struct Node {
    int data;
    struct Node* next;
};
Head → [10|*] → [20|*] → [30|NULL]
```

Operation	Steps	Time Complexity
Insert at head	Create node, set next=head, update head	O(1)
Insert at tail	Traverse to end, set last->next=new node	O(n)
Delete node	Find previous node, set prev->next=node->next	O(n)

Search	Traverse from head, compare data	O(n)
Traverse	Visit each node from head to NULL	O(n)

1.4.2 Circularly Linked List

The last node's **next pointer** points back to the **head** node, forming a circle. Useful for round-robin scheduling and circular buffers.

```
Head → [10|*] → [20|*] → [30|*]
                          ↑_____________|
```

Circular Linked List

- ■ No NULL pointer – traversal wraps around.
- ■ Can start traversal from any node.
- ■ Useful: CPU scheduling (round robin), circular buffer.
- ■ Care needed to avoid infinite loops during traversal.

1.4.3 Doubly Linked List

Each node has: **prev pointer + data + next pointer**. Traversal in both directions.

```
struct DNode {
    struct DNode* prev;
    int data;
    struct DNode* next;
};
NULL ← [10|*] ↔ [20|*] ↔ [30|*] → NULL
```

Feature	Singly	Doubly	Circular
Direction	Forward only	Both directions	Forward (wraps)
Pointers per node	1 (next)	2 (prev+next)	1 (next to head)
Reverse traversal	Not possible	O(n)	Not directly
Memory	Less	More	Same as singly
Delete node	Need prev pointer	O(1) with node ref	Need care
Use case	Simple lists	Browser history	Round-robin

1.5 Applications of Lists

Application	List Type Used	Description
Polynomial ADT	Singly Linked List	Represent polynomial terms as nodes

Radix Sort	Array of lists (buckets)	Sort integers digit by digit
Multi-lists	Linked list of lists	Elements belong to multiple lists
Memory management	Free list	OS tracks free memory blocks
Undo/Redo	Doubly Linked List	Navigate history in both directions
Music playlist	Circular Linked List	Loop through songs continuously

1.5.1 Polynomial ADT

A polynomial like **$4x^3 + 3x^2 + 2x + 1$** can be represented as a linked list where each node stores: **(coefficient, exponent, next)**.

```
struct PolyNode {
    float coeff;
    int exp;
    struct PolyNode* next;
};
4x^3 + 3x^2 + 2x + 1:
[4,3] → [3,2] → [2,1] → [1,0] → NULL
```

1.5.2 Radix Sort

Radix Sort is a non-comparative sorting algorithm that sorts integers by processing individual digits. It uses a list of buckets (0–9) at each pass.

Radix Sort Steps

- ■ Step 1: Sort by least significant digit (LSD) using counting/bucket sort.
- ■ Step 2: Repeat for each digit position up to most significant digit.
- ■ Time Complexity: O(d × (n+k)) where d=digits, n=elements, k=base(10).
- ■ Example: Sort [170, 45, 75, 90, 802, 24, 2, 66]
- ■ Pass 1 (units): [170,90][802,2][24][45,75][66]→[170,90,802,2,24,45,75,66]
- ■ Pass 2 (tens): [802,2][24][45,66][170,75][90]→[802,2,24,45,66,170,75,90]
- ■ Pass 3 (hundreds): [2,24,45,66,75,90][170][802]→[2,24,45,66,75,90,170,802]

DATA STRUCTURES AND ALGORITHMS

STUDY MATERIAL

UNIT II: STACKS AND QUEUES

UNIT II

STACKS AND QUEUES

2.1 Stack ADT

A **Stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle. Elements are added and removed from the same end called the **top**.

Operation	Description	Time Complexity
push(x)	Insert element x onto the top of stack	O(1)
pop()	Remove and return top element	O(1)
peek() / top()	Return top element without removing	O(1)
isEmpty()	Check if stack is empty	O(1)
isFull()	Check if stack is full (array-based)	O(1)
size()	Return number of elements	O(1)

```
// Array-based stack
struct Stack {
    int data[MAX];
    int top;  // -1 when empty
};
push(x): if(!isFull) data[++top] = x;
```

```
  pop():   if(!isEmpty) return data[top--];
```

2.2 Applications of Stack

2.2.1 Balancing Symbols

Stacks are used to check if brackets/parentheses in an expression are **balanced**. For every opening symbol push it; for every closing symbol pop and check for match.

```
  Algorithm BalanceCheck(expr):
    for each char c in expr:
      if c is '(' or '[' or '{': push(c)
      if c is ')': pop and check if '(' matches
      if c is ']': pop and check if '[' matches
      if c is '}': pop and check if '{' matches
    return stack.isEmpty()  // true = balanced
```

Expression	Balanced?	Reason
(a + b) * { c }	Yes	All symbols match
(a + [b * c)	No	Mismatch:) closes [
{ ( ) [ ] }	Yes	Correct nesting
((a + b)	No	Unclosed parenthesis

2.2.2 Infix, Postfix, and Prefix Notations

Notation	Form	Example for A+B*C
Infix	Operator between operands	A + B * C
Postfix (RPN)	Operator after operands	A B C * +
Prefix (Polish)	Operator before operands	+ A * B C

2.2.3 Infix to Postfix Conversion

Algorithm uses a stack to hold operators. Rules based on operator precedence:

Operator	Precedence
^ (exponent)	3 (highest)
* /	2
+ -	1 (lowest
( )	0 (in stack)

```
  Algorithm InfixToPostfix(infix):
    for each token in infix:
      if token is operand: append to output
```

```
    if token is '(': push to stack
    if token is ')':
      pop and output until '(' is found
    if token is operator:
      while stack not empty AND prec(stack.top) >= prec(token):
        pop and output
      push token
  pop all remaining operators to output
```

Infix Expression	Postfix Equivalent
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A * B + C * D	A B * C D * +
(A + B) * (C - D)	A B + C D - *

2.2.4 Evaluating Postfix Expressions

```
Algorithm EvalPostfix(postfix):
  for each token:
    if token is operand: push to stack
    if token is operator:
      pop operand2 = pop()
      pop operand1 = pop()
      result = operand1 operator operand2
      push result
  return pop()  // final answer

Example: 5 3 2 * + = ?
  push 5 → [5]
  push 3 → [5,3]
  push 2 → [5,3,2]
  * : pop 2,3 → 3*2=6, push 6 → [5,6]
  + : pop 6,5 → 5+6=11, push 11 → [11]
  Result = 11
```

2.2.5 Function Calls (Call Stack)

When a function is called, the system pushes a **stack frame** (activation record) onto the call stack containing: return address, local variables, and parameters. When the function returns, the frame is popped.

Call Stack

- Recursion relies entirely on the call stack.
- Stack overflow occurs when recursion is too deep (stack runs out of space).
- Each recursive call creates a new stack frame.
- Example: factorial(3) → factorial(2) → factorial(1) → unwind.

2.3 Queue ADT

A **Queue** is a linear data structure that follows the **FIFO (First In First Out)** principle. Elements are added at the **rear (enqueue)** and removed from the **front (dequeue)**.

Operation	Description	Time Complexity
enqueue(x)	Insert element at rear	O(1)
dequeue()	Remove and return front element	O(1)
peek() / front()	Return front element without removing	O(1)
isEmpty()	Check if queue is empty	O(1)
isFull()	Check if queue is full	O(1)
size()	Number of elements	O(1)

2.4 Circular Queue

A **Circular Queue** solves the problem of wasted space in a linear array-based queue. The rear wraps around to the front when it reaches the end of the array.

```
struct CircularQueue {
    int data[MAX];
    int front, rear, size;
};
enqueue(x): rear = (rear + 1) % MAX; data[rear] = x;
dequeue():  front = (front + 1) % MAX; return data[front];
isFull():   (rear + 1) % MAX == front
isEmpty():  front == rear
```

Circular Queue Properties

- No wasted space – array positions are reused.
- Full condition: (rear+1) % MAX == front.
- Empty condition: front == rear.
- Capacity = MAX - 1 (one slot kept empty to distinguish full/empty).

2.5 Deque (Double-Ended Queue)

A **Deque (DeQueue)** allows insertion and deletion at **both ends**. It is a generalisation of both stack and queue.

Operation	Description
insertFront(x)	Insert at front
insertRear(x)	Insert at rear
deleteFront()	Delete from front
deleteRear()	Delete from rear
getFront()	Peek at front
getRear()	Peek at rear

Type	Restriction	Behaviour
Input-Restricted Deque	Insertion only at rear	Delete from both ends
Output-Restricted Deque	Deletion only from front	Insert at both ends
Full Deque	No restriction	Insert/delete at both ends

2.6 Applications of Queues

Application	Queue Type	Description
CPU Scheduling	Simple Queue	Process scheduling (FCFS algorithm)
Print Spooler	Simple Queue	Print jobs processed in order
BFS (Graph Traversal)	Simple Queue	Explores nodes level by level
Keyboard Buffer	Circular Queue	Stores keystrokes in order
Sliding Window	Deque	Max/min in window problems
Call Center	Priority Queue	Handle calls by priority
Network Packet Buffer	Circular Queue	Manage incoming packets

Feature	Stack	Queue
Order	LIFO	FIFO
Insertion end	Top	Rear
Deletion end	Top	Front
Application	Recursion, undo	Scheduling, BFS
Pointer(s)	top	front + rear

DATA STRUCTURES AND ALGORITHMS

STUDY MATERIAL

UNIT IV: MULTIWAY SEARCH TREES AND GRAPHS

UNIT IV

MULTIWAY SEARCH TREES AND GRAPHS

4.1 B-Tree

A **B-Tree of order m** is a self-balancing multiway search tree generalising a BST. It is designed to work efficiently with disk storage by minimising I/O operations.

B-Tree of Order m: Properties

- ■ Every node has at most m children.
- ■ Every non-root node has at least ■m/2■ children.
- ■ Root has at least 2 children (unless it is a leaf).
- ■ All leaves appear at the same level (perfectly balanced).
- ■ A node with k children has k–1 keys.
- ■ Keys within a node are stored in sorted order.

Operation	Time Complexity	Notes
Search	O(log n)	Similar to BST but multiway
Insert	O(log n)	May cause node splits propagating up
Delete	O(log n)	May cause merges or redistribution

4.2 B+ Tree

A **B+ Tree** is a variation of B-Tree where **all data records are stored in leaf nodes**. Internal nodes store only keys for routing. Leaf nodes are linked as a sorted linked list.

Feature	B-Tree	B+ Tree
Data storage	In all nodes (internal + leaf)	Only in leaf nodes
Internal nodes	Store keys + data pointers	Store keys only
Leaf nodes	Not linked	Linked as sorted list
Sequential access	Requires full traversal	Efficient via leaf linked list
Space usage	Less efficient	More efficient for range queries
Use case	General databases	File systems, RDBMS indexes (MySQL InnoDB)

4.3 Graph Definition and Representation

A **Graph G = (V, E)** consists of a set of **vertices V** (nodes) and a set of **edges E** (connections between vertices).

Term	Definition
Undirected Graph	Edges have no direction: (u,v) = (v,u)
Directed Graph (Digraph)	Edges have direction: (u,v) ≠ (v,u)
Weighted Graph	Each edge has an associated weight/cost
Degree	Number of edges incident to a vertex
In-degree / Out-degree	For digraphs: incoming / outgoing edges
Path	Sequence of vertices connected by edges
Cycle	Path that starts and ends at the same vertex
Connected Graph	Every vertex is reachable from every other vertex
Spanning Tree	Subgraph that is a tree connecting all vertices

Graph Representations

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

```
// Adjacency Matrix (V=4: 0,1,2,3)
   0 1 2 3
0 [0,1,0,1]
1 [1,0,1,0]
2 [0,1,0,1]
3 [1,0,1,0]

// Adjacency List
```

```
0 → [1, 3]
1 → [0, 2]
2 → [1, 3]
3 → [0, 2]
```

4.4 Graph Traversals

4.4.1 Breadth-First Search (BFS)

BFS explores all neighbours at the current depth before moving deeper. Uses a **Queue**.

```
BFS(G, start):
  mark start as visited; enqueue(start)
  while queue not empty:
    v = dequeue()
    visit v
    for each neighbour u of v:
      if u not visited: mark visited; enqueue(u)
```

4.4.2 Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking. Uses a **Stack** (or recursion).

```
DFS(G, v):
  mark v as visited; visit v
  for each neighbour u of v:
    if u not visited: DFS(G, u)
```

Feature	BFS	DFS
Data Structure	Queue	Stack / Recursion
Time Complexity	O(V+E)	O(V+E)
Space Complexity	O(V)	O(V)
Shortest Path	Yes (unweighted)	No
Cycle Detection	Yes	Yes
Use case	Shortest path, level-order	Topological sort, SCC

4.5 Topological Sort

A **topological sort** of a Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (u→v), u comes before v.

Topological Sort

- ■ Only possible for DAGs (no cycles).
- ■ Algorithm (Kahn's): Find all vertices with in-degree 0, enqueue them.

■ Repeatedly dequeue, output vertex, reduce in-degree of neighbours.

■ If output count < V, graph has a cycle.

■ Use case: Build systems, course prerequisites, task scheduling.

4.6 Shortest Path – Dijkstra's Algorithm

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

```
Dijkstra(G, src):
  dist[src]=0; dist[all others]=INFINITY
  priority queue PQ with (0, src)
  while PQ not empty:
    (d, u) = extract_min from PQ
    for each neighbour v of u:
      if dist[u] + weight(u,v) < dist[v]:
        dist[v] = dist[u] + weight(u,v)
        add (dist[v], v) to PQ
```

Implementation	Time Complexity
Adjacency Matrix + linear scan	$O(V^2)$
Adjacency List + Binary Heap	O((V+E) log V)
Adjacency List + Fibonacci Heap	O(E + V log V)

4.7 Minimum Spanning Tree (MST)

A **Minimum Spanning Tree** of a connected weighted graph is a spanning tree with the minimum total edge weight. (V-1 edges for V vertices, no cycles.)

4.7.1 Prim's Algorithm

Builds MST by **growing one vertex at a time**. At each step, add the cheapest edge connecting the tree to a new vertex.

```
Prim(G, start):
  key[start]=0; key[all others]=INF
  parent[start]=-1
  priority queue PQ with all vertices
  while PQ not empty:
    u = extract_min(PQ)
    for each neighbour v of u:
      if v in PQ and weight(u,v) < key[v]:
        parent[v] = u; key[v] = weight(u,v)
```

4.7.2 Kruskal's Algorithm

Builds MST by **adding edges in increasing weight order**, skipping edges that create cycles. Uses Union-Find (Disjoint Set Union) to detect cycles.

```
Kruskal(G):
  sort all edges by weight
  initialise DSU (each vertex its own component)
  MST = []
  for each edge (u, v, w) in sorted order:
    if find(u) != find(v):  // no cycle
      MST.append((u, v, w))
      union(u, v)
```

Feature	Prim's	Kruskal's
Approach	Vertex-based (greedy)	Edge-based (greedy)
Data Structure	Priority Queue	Sorted edges + DSU
Time Complexity	O(E log V)	O(E log E)
Best for	Dense graphs	Sparse graphs
Handles disconnected?	No	Yes (finds MST forest)

DATA STRUCTURES AND ALGORITHMS

STUDY MATERIAL

UNIT III: TREES

UNIT III

TREES

3.1 Tree ADT

A **Tree** is a non-linear hierarchical data structure consisting of nodes connected by edges. It has one **root** node and every node (except root) has exactly one parent.

Term	Definition
Root	Top-most node with no parent
Edge	Connection between two nodes
Parent	Node with children below it
Child	Node connected below a parent
Leaf / External Node	Node with no children
Internal Node	Node with at least one child
Depth of node	Number of edges from root to that node
Height of tree	Maximum depth among all nodes
Degree of node	Number of children of a node
Subtree	A node and all its descendants

3.2 Tree Traversals

Traversal visits every node exactly once. Three standard DFS traversals for binary trees:

Traversal	Order	Pseudocode

Inorder	Left → Root → Right	inorder(left); visit(root); inorder(right)
Preorder	Root → Left → Right	visit(root); preorder(left); preorder(right)
Postorder	Left → Right → Root	postorder(left); postorder(right); visit(root)
Level-order (BFS)	Level by level	Use a queue; enqueue root, process level by level

Traversal Uses

- Inorder of BST gives sorted sequence.
- Preorder used to copy a tree.
- Postorder used to delete a tree.
- Level-order used in BFS and finding shortest path in trees.

3.3 Binary Tree ADT

A **Binary Tree** is a tree where each node has **at most two children**: left child and right child.

Type	Definition
Full Binary Tree	Every node has 0 or 2 children
Complete Binary Tree	All levels filled except last; last level filled left to right
Perfect Binary Tree	All internal nodes have 2 children; all leaves at same level
Skewed Tree	All nodes have only one child (left or right)
Balanced Binary Tree	Height difference between left and right subtrees ≤ 1

For a binary tree with n nodes: **Maximum height = n–1, Minimum height = log2(n)**

3.4 Expression Trees

An **Expression Tree** is a binary tree where: **leaves** are operands (numbers/variables) and **internal nodes** are operators (+, –, *, /).

Traversal	Result for (a+b)*(c-d)
Inorder	a + b * c - d (infix – needs parentheses for correctness)
Preorder	* + a b - c d (prefix notation)
Postorder	a b + c d - * (postfix notation)

3.5 Binary Search Tree (BST)

A **BST** is a binary tree where for each node: all values in the **left subtree** are less than the node, and all values in the **right subtree** are greater.

Operation	Average Case	Worst Case (Skewed)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Inorder Traversal	O(n)	O(n)

```
BST Search Algorithm:
if root == NULL: return NOT FOUND
if key == root.data: return FOUND
if key < root.data: search(root.left, key)
else: search(root.right, key)

Example BST for [50, 30, 70, 20, 40, 60, 80]:
         50
        /  \
       30   70
      / \ / \
     20 40 60 80
```

3.6 AVL Trees

An **AVL Tree** is a **self-balancing BST** where the height difference (Balance Factor = height(left) – height(right)) of every node is at most **±1**. Rebalancing is done using rotations.

Rotation	When Applied	Description
LL Rotation (Single Right)	Left-Left imbalance	Rotate right at imbalanced node
RR Rotation (Single Left)	Right-Right imbalance	Rotate left at imbalanced node
LR Rotation (Double)	Left-Right imbalance	Rotate left at child, then right at node
RL Rotation (Double)	Right-Left imbalance	Rotate right at child, then left at node

Feature	BST	AVL Tree
Balance	Not guaranteed	Always balanced (BF = ±1)
Search (worst)	O(n) skewed	O(log n) guaranteed
Insert overhead	O(log n) avg	O(log n) + rotation

Use case	Simple lookups	Frequent search-heavy operations

3.7 Priority Queue and Binary Heap

A **Priority Queue** is an ADT where each element has a priority. The element with the highest (or lowest) priority is served first.

Binary Heap

A **Binary Heap** is a complete binary tree satisfying the **heap property**:

Heap Type	Property	Use
Min-Heap	Parent ≤ Children (root is minimum)	Dijkstra, Prim, Priority Queue
Max-Heap	Parent ≥ Children (root is maximum)	Heap Sort, scheduling

Operation	Time Complexity	Description
Insert	O(log n)	Add at end, sift up (heapify up)
Delete Min/Max	O(log n)	Remove root, replace with last, sift down
Get Min/Max	O(1)	Root element
Build Heap	O(n)	Heapify all non-leaf nodes bottom-up

```
Array representation of Heap (1-indexed):
  Parent of node i = i/2
  Left child of i  = 2*i
  Right child of i = 2*i + 1

Min-Heap example: [1, 3, 6, 5, 9, 8]
          1
        /   \
       3     6
      / \   /
     5   9 8
```

DATA STRUCTURES AND ALGORITHMS

STUDY MATERIAL

UNIT V: SEARCHING SORTING AND HASHING

UNIT V

SEARCHING, SORTING AND HASHING

5.1 Searching Algorithms

5.1.1 Linear Search

Sequentially checks each element until the target is found or the list ends.

```
LinearSearch(arr, n, key):
  for i from 0 to n-1:
    if arr[i] == key: return i
  return -1  // not found
```

Case	Time Complexity
Best	O(1) – element at first position
Average	O(n/2) = O(n)
Worst	O(n) – element at last position or not found

5.1.2 Binary Search

Binary Search works on **sorted arrays**. Repeatedly halves the search space by comparing with the middle element.

```
BinarySearch(arr, low, high, key):
  while low <= high:
    mid = (low + high) / 2
    if arr[mid] == key: return mid
    else if arr[mid] < key: low = mid + 1
```

```
    else: high = mid - 1
  return -1
```

Feature	Linear Search	Binary Search
Pre-requisite	None (unsorted works)	Array must be sorted
Time (Best)	O(1)	O(1)
Time (Worst)	O(n)	O(log n)
Space	O(1)	O(1) iterative, O(log n) recursive
Data access	Sequential	Random (index-based)

5.2 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
Shell Sort	O(n log n)	O(n^1.5)	$O(n^2)$	O(1)	No
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No

5.2.1 Bubble Sort

Repeatedly compares adjacent elements and swaps if out of order. Largest element bubbles to end each pass.

```
BubbleSort(arr, n):
  for i = 0 to n-2:
    for j = 0 to n-2-i:
      if arr[j] > arr[j+1]: swap(arr[j], arr[j+1])

Example: [64, 34, 25, 12, 22, 11, 90]
Pass 1: [34,25,12,22,11,64,90] → 90 in place
Pass 2: [25,12,22,11,34,64,90] → 64 in place
```

5.2.2 Selection Sort

Finds the minimum element from the unsorted part and places it at the correct position.

```
SelectionSort(arr, n):
  for i = 0 to n-2:
    min_idx = i
    for j = i+1 to n-1:
```

```
      if arr[j] < arr[min_idx]: min_idx = j
    swap(arr[i], arr[min_idx])
```

5.2.3 Insertion Sort

Builds sorted array one element at a time by inserting each element into its correct position. Efficient for small or nearly-sorted data.

```
InsertionSort(arr, n):
  for i = 1 to n-1:
    key = arr[i]; j = i-1
    while j >= 0 and arr[j] > key:
      arr[j+1] = arr[j]; j--
    arr[j+1] = key
```

5.2.4 Shell Sort

Shell Sort is an improved Insertion Sort. It sorts elements far apart first, then reduces the gap until gap=1 (becoming insertion sort). Proposed by Donald Shell.

```
ShellSort(arr, n):
  gap = n/2
  while gap > 0:
    for i = gap to n-1:
      temp = arr[i]; j = i
      while j >= gap and arr[j-gap] > temp:
        arr[j] = arr[j-gap]; j -= gap
      arr[j] = temp
    gap = gap/2
```

5.2.5 Merge Sort

Merge Sort is a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts each half, then merges the sorted halves.

```
MergeSort(arr, left, right):
  if left < right:
    mid = (left + right) / 2
    MergeSort(arr, left, mid)
    MergeSort(arr, mid+1, right)
    Merge(arr, left, mid, right)

Example: [38,27,43,3,9,82,10]
Split:   [38,27,43] [3,9,82,10]
Sort:    [27,38,43] [3,9,10,82]
```

```
Merge:   [3,9,10,27,38,43,82]
```

5.3 Hashing

Hashing is a technique that maps keys to positions in a hash table using a **hash function**. Provides O(1) average-case search, insert, and delete.

5.3.1 Hash Functions

Hash Function	Formula	Notes
Division Method	h(k) = k mod m	m should be prime; simple and widely used
Multiplication Method	h(k) = floor(m × (k×A mod 1))	A ≈ 0.618 (golden ratio); works for any m
Folding Method	Split key, add parts	h(123456) = 12+34+56=102; mod m
Mid-Square	Square key, extract middle digits	Spreads keys uniformly

5.3.2 Collision Resolution

A **collision** occurs when two keys hash to the same index. Resolution methods:

Separate Chaining

Each table position holds a **linked list** of elements that hash to it.

```
Hash Table with Separate Chaining (m=7):
h(k) = k mod 7
Insert: 50,700,76,85,92,73,101
Index 0: [700] → [NULL]
Index 1: [50] → [92] → [NULL]
Index 2: [NULL]
Index 3: [73] → [NULL]
Index 4: [76] → [NULL]
Index 5: [85] → [NULL]
Index 6: [101] → [NULL]
```

Open Addressing

All elements stored in the hash table itself. On collision, probe for next available slot.

Probing Method	Formula	Problem
Linear Probing	h(k,i) = (h(k)+i) mod m	Primary clustering
Quadratic Probing	h(k,i) = (h(k)+i^2) mod m	Secondary clustering
Double Hashing	h(k,i) = (h1(k)+i×h2(k)) mod m	Best distribution, no clustering

5.3.3 Rehashing

Rehashing is the process of creating a new, larger hash table and re-inserting all existing elements when the load factor (n/m) exceeds a threshold (typically 0.5–0.75).

Rehashing

- Load factor $\lambda = n/m$ (n=elements, m=table size).
- When $\lambda >$ threshold: create new table of size 2m (next prime), rehash all elements.
- Expensive: O(n) operation, but amortised O(1) per insertion.
- Necessary to maintain performance of open addressing.

5.3.4 Extendible Hashing

Extendible Hashing is a dynamic hashing scheme for databases that grows the hash table one bucket at a time, avoiding full rehashing.

Extendible Hashing

- Uses a directory of pointers to buckets.
- Global depth: number of bits used to index directory.
- Local depth: number of bits a specific bucket uses.
- On bucket overflow: split the bucket, double directory only if needed.
- Efficient for disk-based storage systems (databases, file systems).

Method	Collision Handling	Space	Performance
Separate Chaining	Linked list at each slot	O(n+m)	O(1) avg, O(n) worst
Linear Probing	Next available slot	O(m)	Clustering reduces performance
Quadratic Probing	Quadratic jump	O(m)	Less clustering than linear
Double Hashing	Second hash function	O(m)	Best open addressing performance
Rehashing	New larger table	O(2m)	Restores O(1) performance
Extendible	Directory + buckets	Dynamic	O(1) avg for databases