STUDY MATERIAL FOR B.Sc., COMPUTER SCIENCE WITH ARTIFICIAL INTELLIGENCE

DATA STRUCTURES AND ALGORITHMS

SEMESTER – II

ACADEMIC YEAR 2025-26

PREPARED BY

COMPUTER SCIENCE DEPARTMENT

KAMARAJ COLLEGE

INDEX

UNIT	CONTENT	PAGE NO
I	ABSTRACT DATA TYPES	03-21
II	THE STACK ADT	22-34
III	TREE ADT	35-42
IV	GRAPH DATA STRUCTURE	43-48
V	LINEAR SEARCH	49-62

KAMARAJ WOMENS COLLEGE

UNIT - I

Abstract Data Types (ADTs)

An Abstract Data Type (ADT) is a conceptual model that defines a collection of data and the operations that can be performed on that data, without specifying how those operations are implemented. It focuses on what an operation does, not how it is carried out. Examples of ADTs include lists, stacks, queues, sets, and graphs—just like built-in types such as integers and booleans, but more complex and user-defined. For instance, a set ADT might include operations like add, remove, contains, or union, depending on the specific design.

In C++, ADTs are implemented using classes, which allow developers to hide internal implementation details through the use of private data members and public methods. This means other parts of the program interact with the ADT only through well-defined interfaces, making the internal workings irrelevant and easily replaceable. This approach promotes encapsulation, modularity, and abstraction, enabling cleaner, reusable, and maintainable code.

The design of an ADT is flexible; there are no strict rules about which operations must be included. Decisions such as how to handle errors or resolve conflicts (like duplicate elements in a set) are left to the programmer. If implemented correctly, programs using an ADT do not need to know or care how it is implemented internally. This allows for easy changes or optimizations in the future without affecting the rest of the program. In summary, ADTs define what a data structure can do, while classes in C++ define how it does it, keeping the details hidden and the design clean.

List ADT

The List Abstract Data Type (List ADT) is a sequential collection of elements that allows for the storage and management of data in an ordered manner. It provides a defined set of operations for accessing, inserting, modifying, and removing elements, while hiding the internal implementation details such as whether the list is backed by an array or a linked structure. This abstraction allows users to work with lists conceptually without worrying about how they are maintained in memory. The key operations supported by the List ADT include get(), which retrieves an element from a specific position; insert(), which adds a new element at any given location; and remove() or removeAt(), which delete either the first occurrence of a specific element or an element at a specific index, respectively. The replace() function updates an element at a given position, while size() returns the current number of elements in the list. Utility methods such as isEmpty() check if the list has no elements, and isFull() determines whether a fixed-size list (like an array-based list) has reached its capacity. These operations provide a consistent interface for interacting with lists in a logical and organized way, regardless of the underlying implementation.

Array based implementation

An array is a way to store many values of the same type in a single variable. All the values are kept next to each other in memory, and we can find any value using its index, starting from 0. For example, if we have an array of 5 numbers, we can access the first number using index 0, the

KAMARAJ WOMENS COLLEGE

second with index 1, and so on. Arrays make it easy to handle a lot of data without creating separate variables for each value.

There are two types of arrays based on size: fixed-size arrays, which cannot change their size after they are created, and dynamic arrays, which can grow or shrink as needed. Arrays can also be one-dimensional (like a simple list), two-dimensional (like a table), or even more complex with multiple dimensions.

We can do many operations with arrays:

Traversal: Go through each item in the array.

Insertion: Add a new item at a specific position.

Deletion: Remove an item from the array.

Searching: Find where a specific value is in the array.

There are two common ways to Search:

Linear Search – checks each element one by one.

Binary Search – much faster but works only on sorted arrays.

Arrays are simple and powerful, and they are used in almost every program to store and manage data easily.

Linked list

A Linked List is a linear data structure in which the elements are stored in nodes, and each node is connected to the next using a pointer. Unlike arrays, linked list elements are not stored in contiguous memory. Instead, each node consists of two parts:

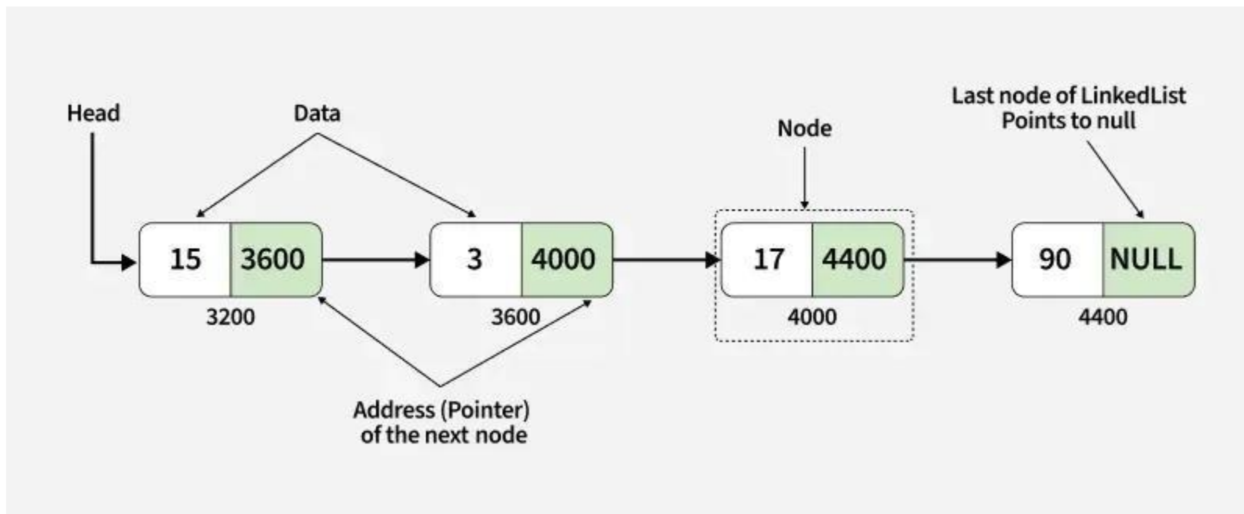
i) Data – stores the actual value.

ii) Pointer (or reference) – holds the address of the next node in the sequence.

The first node is called the head, and the last node points to NULL (or to the head in the case of circular linked lists). Linked lists allow dynamic memory allocation, meaning their size can grow or shrink during runtime, making them more flexible than arrays when it comes to memory management. Operations such as insertion and deletion are more efficient, especially at the beginning or middle of the list, as they don't require shifting elements like in arrays.

Linked lists are widely used to implement other complex data structures such as stacks, queues, hash tables, and graphs. They are particularly useful when the number of data elements is not known in advance or changes frequently.

KAMARAJ WOMENS COLLEGE



Types of Linked Lists

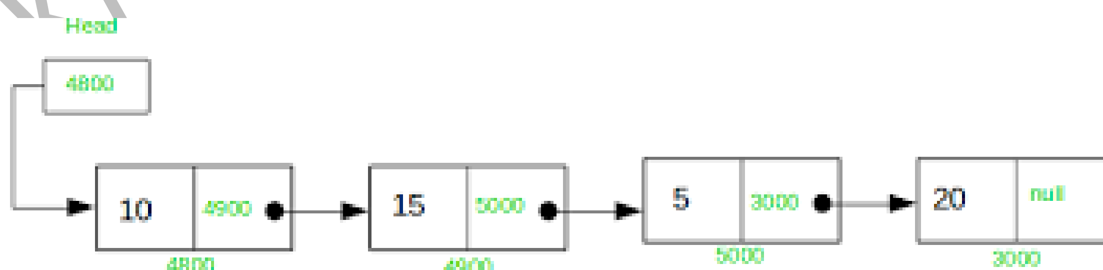
- Singly Linked List
- Doubly Linked List
- Circular Linked List

Singly linked lists

A Singly Linked List is a linear data structure in which each element, called a node, contains two parts: the data and a pointer to the next node in the sequence. The first node is known as the head, and the last node points to NULL, indicating the end of the list. It allows traversal in only one direction and supports efficient insertion and deletion operations, especially when compared to arrays.

Syntax:

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



Operations

- Insertion
- Deletion
- Traversal

PROGRAM :

```
#include <iostream>
#include <cstdlib> // for malloc and free using namespace std;

// Node structure struct node {
int data; node* next;
};

class LinkedList {
node* head = NULL; // Pointer to the head of the list

public:
// Insert node at the beginning void insertAtBeginning(int val) {
node* newNode = (node*)malloc(sizeof(node)); if (newNode == NULL) {
cout << "Out of memory\n"; return;
}
newNode->data = val; newNode->next = head; head = newNode;
cout << "Inserted " << val << " at the beginning\n";
}

// Display the list void display() {
if (head == NULL) {
cout << "List is empty\n"; return;
}


node* temp = head;
cout << "Elements in the list: ";
 while (temp != NULL)
{
cout << temp->data << " "; temp = temp->next;
}
cout << endl;
}
// Insert at specific position
```

KAMARAJ WOMENS COLLEGE

```
void insertAtPos(int pos, int val) { if (pos < 0) {
cout << "Invalid position\n"; return;
}
node* newNode = (node*)malloc(sizeof(node)); if (newNode == NULL) {
cout << "Out of memory\n"; return;
}
newNode->data = val;
if (pos == 0) {
newNode->next = head; head = newNode;
cout << "Inserted " << val << " at position " << pos << endl; return;
}
node* temp = head;
for (int i = 1; i < pos; i++) { if (temp == NULL) {
cout << "Invalid position\n"; free(newNode);
return;
}
temp = temp->next;
}
if (temp == NULL) {
cout << "Invalid position\n"; free(newNode);
return;
}
newNode->next = temp->next; temp->next = newNode;
cout << "Inserted " << val << " at position " << pos << endl;
}
// Delete at specific position void deleteAtPos(int pos) {
if (head == NULL) {
cout << "List is empty\n"; return;
}
if (pos == 0) {
node* temp = head; head = head->next;
cout << "Deleted " << temp->data << " from position 0\n"; free(temp);
return;
}
node* temp = head; node* prev = NULL;
for (int i = 0; i < pos; i++) { prev = temp;
temp = temp->next; if (temp == NULL) {
cout << "Invalid position\n"; return;
}
}
prev->next = temp->next;
```

KAMARAJ WOMENS COLLEGE

```
cout << "Deleted " << temp->data << " from position " << pos << endl; free(temp);
}
};
int main() {
int choice, val, pos; LinkedList list1;
while (true)
 {
cout << "\n -------- Linked List Menu   \n";
cout << "1. Insert at Beginning\n"; cout << "2. Display the list\n";
cout << "3. Insert at specified position\n"; cout << "4. Delete from specified position\n"; cout << "5. Exit\n";
cout << "        \n";    -------------------------
cout << "Enter your choice:\t"; cin >> choice;
switch (choice) { case 1:
cout << "Enter the data: "; cin >> val; list1.insertAtBeginning(val); break;
case 2:
list1.display(); break;
case 3:
cout << "Enter the position (starts at 0): "; cin >> pos;
cout << "Enter the data: "; cin >> val; list1.insertAtPos(pos, val); break;
case 4:
cout << "Enter the position to delete: "; cin >> pos;
list1.deleteAtPos(pos); break;
case 5:
cout << "Exiting program...\n"; return 0;
default:
cout << "Invalid choice. Try again.\n";
}
}
```

KAMARAJ WOMENS COLLEGE

```
return 0;
}
```

output

-------- Linked List Menu -----------

Insert at Beginning

Display the list

Insert at specified position

Delete from specified position

Exit

Enter your choice: 1 Enter the data: 50

Inserted 50 at the beginning

-------- Linked List Menu ----------- Enter your choice: 1

Enter the data: 40

Inserted 40 at the beginning

-------- Linked List Menu ----------- Enter your choice: 2

Elements in the list are: 40 50

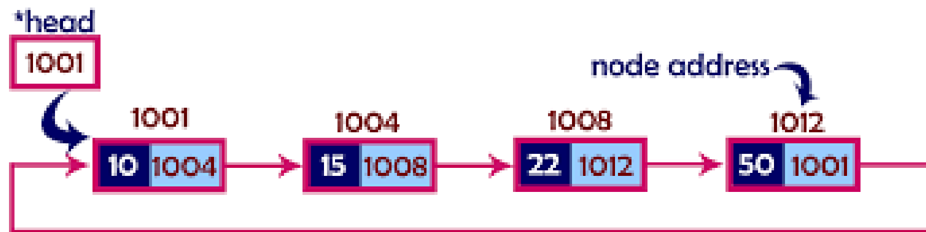
Circular Linked List

A Circular Linked List is a variation of the traditional linked list in which the last node of the list points back to the first node instead of having a NULL pointer. This creates a closed loop structure, where all the nodes are connected in a circular manner. In other words, it forms a circle of nodes, making it possible to traverse the entire list starting from any node and eventually reaching the same node again.

Syntax:

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

KAMARAJ WOMENS COLLEGE



Operations

- Insertion
  - At beginning
  - At End
- Deletion
  - At beginning
  - At End
- Traversal

PROGRAM :

```
#include<iostream>
 #include<cstdlib>
using namespace std;
// Define the node structure struct lnode {
int data;
struct lnode* next;
};
typedef struct lnode node; node* last = NULL;
// Function to insert at the beginning of the list void insertAtBeginning(int val) {
node* newNode = (node*)malloc(sizeof(node)); if(newNode == NULL) {
cout << "Out of memory\n"; return;
}
newNode->data = val; if(last == NULL) {
newNode->next = newNode; last = newNode;
} else {
newNode->next = last->next; last->next = newNode;
}
cout << val << " inserted at beginning.\n";
}
// Function to insert at the end of the list void insertAtEnd(int val) {
```

KAMARAJ WOMENS COLLEGE

```
node* newNode = (node*)malloc(sizeof(node)); if(newNode == NULL) {
cout << "Out of memory\n"; return;
}
newNode->data = val; if(last == NULL) {
newNode->next = newNode; last = newNode;
} else {
newNode->next = last->next; last->next = newNode;
last = newNode;
}
cout << val << " inserted at end.\n";
}
// Function to delete node from beginning void deleteAtBeginning() {
if(last == NULL) {
cout << "The list is empty.\n"; return;
}
node* del;
if(last->next == last) { del = last;
last = NULL;
} else {
del = last->next;
last->next = del->next;
}
cout << del->data << " is deleted from beginning.\n"; free(del);
}

// Function to delete node from end void deleteAtEnd() {
if(last == NULL) {
cout << "The list is empty.\n"; return;
}
node *del, *temp; if(last->next == last) {
del = last; last = NULL;
}
 else {
temp = last->next; while(temp->next != last) {
temp = temp->next;
}
del = last;
temp->next = last->next; last = temp;
}
cout << del->data << " is deleted from end.\n"; free(del);
}
```

KAMARAJ WOMENS COLLEGE

```
// Function to display the circular linked list void display() {
if(last == NULL) {
cout << "List is empty.\n"; return;
}
node* temp = last->next; cout << "List: ";
do {
cout << temp->data << " "; temp = temp->next;
} while(temp != last->next); cout << endl;
}
// Main function with menu using switch-case int main() {
int choice, value;


while(true) {
cout << "\n--- Circular Linked List Menu ---\n"; cout << "1. Insert at Beginning\n";
cout << "2. Insert at End\n";
cout << "3. Delete at Beginning\n"; cout << "4. Delete at End\n";
cout << "5. Display\n"; cout << "6. Exit\n";
cout << "Enter your choice: "; cin >> choice;
switch(choice) { case 1:
cout << "Enter value to insert at beginning: "; cin >> value;
insertAtBeginning(value); break;
case 2:
cout << "Enter value to insert at end: "; cin >> value;
insertAtEnd(value); break;
case 3:
deleteAtBeginning(); break;
case 4:
deleteAtEnd(); break;
case 5:
display(); break;
case 6:
```

KAMARAJ WOMENS COLLEGE

```
cout << "Exiting...\n"; return 0;

default:

cout << "Invalid choice. Please try again.\n";

}

}

return 0;

}
```

Output

--- Circular Linked List Menu ---

Insert at Beginning

Insert at End

Delete at Beginning

Delete at End

Display

Exit

Enter your choice: 1

Enter value to insert at beginning: 10 10 inserted at beginning.

--- Circular Linked List Menu --- Enter your choice: 1

Enter value to insert at beginning: 20 20 inserted at beginning.

--- Circular Linked List Menu --- Enter your choice: 2

Enter value to insert at end: 30 30 inserted at end.

--- Circular Linked List Menu --- Enter your choice: 5

List: 20 10 30

--- Circular Linked List Menu --- Enter your choice: 3

20 is deleted from beginning.

KAMARAJ WOMENS COLLEGE

--- Circular Linked List Menu --- Enter your choice: 5

List: 10 30

--- Circular Linked List Menu --- Enter your choice: 4

30 is deleted from end.

--- Circular Linked List Menu --- Enter your choice: 5

List: 10

--- Circular Linked List Menu --- Enter your choice: 6

Exiting...

Circular linked lists

Doubly Linked List

A Doubly Linked List (DLL) is a type of linear data structure that consists of a sequence of elements, called nodes, where each node contains three components:

A data field that stores the actual value or information.

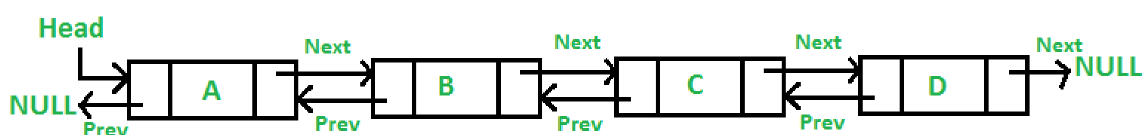
A pointer (or reference) to the next node in the sequence.

A pointer (or reference) to the previous node in the sequence.

Unlike a singly linked list, which can only be traversed in one direction (forward), a doubly linked list allows traversal in both directions — forward (from head to tail) and backward (from tail to head) — due to the presence of both next and prev pointers in each node.

Syntax

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



KAMARAJ WOMENS COLLEGE

PROGRAM

```
#include <iostream> #include <cstdlib> using namespace std;


// Define node structure for Doubly Linked List struct lnode {
int data; lnode* prev; lnode* next;
};
typedef struct lnode node;
// Global head and tail pointers node* head = NULL;
node* tail = NULL;
// Insert a node at the beginning of the list void insertAtBeginning(int val) {
node* newNode = (node*)malloc(sizeof(node)); if (newNode == NULL) {
cout << "Error: Memory allocation failed." << endl; return;
}
newNode->data = val; newNode->prev = NULL; newNode->next = head;
if (head == NULL) { tail = newNode;
} else {
head->prev = newNode;
}
head = newNode;
cout << "Inserted " << val << " at the beginning." << endl;
}
// Insert a node at a specific position void insertAtPos(int pos, int val) {
if (pos == 0) { insertAtBeginning(val); return;
}
node* newNode = (node*)malloc(sizeof(node)); if (newNode == NULL) {
cout << "Error: Memory allocation failed." << endl; return;
}
node* temp = head;
```

KAMARAJ WOMENS COLLEGE

```
for (int i = 1; i <= pos - 1; i++) { if (temp == NULL) {
cout << "Invalid position: " << pos << endl; free(newNode);
return;
}
temp = temp->next;
}
newNode->data = val; newNode->next = temp->next; newNode->prev = temp;

if (temp->next == NULL) { tail = newNode;
} else {
temp->next->prev = newNode;
}

temp->next = newNode;

cout << "Inserted " << val << " at position " << pos << "." << endl;
}
// Delete a node from a specific position
void deleteAtPos(int pos) {
if (head == NULL) {
cout << "List is empty. Nothing to delete." << endl; return;
}
node* temp = head; if (pos == 0) {
head = head->next; if (head == NULL) {
tail = NULL;
} else {
head->prev = NULL;
}
```

KAMARAJ WOMENS COLLEGE

```
cout << "Deleted element: " << temp->data << " from position 0." << endl; free(temp);
return;
}
node* prev;
for (int i = 1; i <= pos; i++) { prev = temp;
temp = temp->next; if (temp == NULL) {
cout << "Invalid position: " << pos << endl; return;
}
}
prev->next = temp->next; if (temp->next == NULL) {
tail = prev;
} else {
temp->next->prev = prev;
}
cout << "Deleted element: " << temp->data << " from position " << pos << "." << endl; free(temp);
}
// Display list from head to tail void display() {
if (head == NULL) {
cout << "List is empty." << endl; return;
}
node* temp = head;
cout << "List (Forward): "; while (temp != NULL) {
cout << temp->data << " "; temp = temp->next;
}
cout << endl;
}

// Display list from tail to head void displayRev() {
```

KAMARAJ WOMENS COLLEGE

```
if (tail == NULL) {
cout << "List is empty." << endl; return;
}
node* temp = tail;
cout << "List (Reverse): "; while (temp != NULL) {
cout << temp->data << " "; temp = temp->prev;
}
cout << endl;
}
// Main menu-driven function int main() {
int choice, val, pos;
cout << "=== Doubly Linked List Program (C++) ===" << endl; cout << "Commands:\n";
cout << "1 - Insert at Beginning\n"; cout << "2 - Insert at Position\n"; cout << "3 - Delete at Position\n";
cout << "4 - Display List (Forward)\n";
cout << "5 - Display List (Reverse)\n"; cout << "6 - Exit\n";
while (true) {
cout << "\nEnter your choice: "; cin >> choice;
switch (choice) { case 1:
cout << "Enter value to insert: "; cin >> val; insertAtBeginning(val);
break; case 2:
cout << "Enter position: "; cin >> pos;
cout << "Enter value: "; cin >> val; insertAtPos(pos, val); break;
case 3:
cout << "Enter position to delete: "; cin >> pos;
deleteAtPos(pos); break;
case 4:
display(); break;
```

KAMARAJ WOMENS COLLEGE

```
case 5:
displayRev(); break;
case 6:
cout << "Exiting program." << endl; return 0;
default:
cout << "Invalid choice. Please enter a number between 1 and 6." << endl;
}
}
return 0;
}
```

OUTPUT

=== Doubly Linked List Program (C++) === Commands:

- Insert at Beginning
- Insert at Position
- Delete at Position
- Display List (Forward)
- Display List (Reverse)
- Exit

Enter your choice: 1 Enter value to insert: 10

Inserted 10 at the beginning.

Enter your choice: 1 Enter value to insert: 20

Inserted 20 at the beginning.

Enter your choice: 4 List (Forward): 20 10

Enter your choice: 5 List (Reverse): 10 20

Enter your choice: 2 Enter position: 1

Enter value: 15

KAMARAJ WOMENS COLLEGE

Inserted 15 at position 1.

Enter your choice: 4 List (Forward): 20 15 10

Enter your choice: 3 Enter position to delete: 0

Deleted element: 20 from position 0.

Enter your choice: 4 List (Forward): 15 10

Enter your choice: 6

Exiting program.

Applications of Linked Lists

Applications in Computer Science

- Used to implement Stacks and Queues.
- Graph Representation using adjacency lists.
- Dynamic memory allocation (free memory blocks tracked by linked list).
- Symbol tables in compilers.
- Performing arithmetic on large integers (each digit in a node).
- Polynomial operations (each term as a node).
- Sparse matrix representation (only non-zero values stored).
- File and directory management in file systems.
- Managing processes/tasks in operating systems.
- Used in hash tables, trees, and other data structures.

Real-World Applications

- Image viewers – Navigate between previous/next images.
- Web browsers – Back and forward navigation history.
- Music players – Track next and previous songs in a playlist.
- GPS navigation systems – Store route points as linked locations.

KAMARAJ WOMENS COLLEGE

- Speech recognition – Phoneme possibilities as nodes.
- Simulation systems – Discrete time steps stored in a list.
- Undo/Redo operations – Actions stored in a doubly linked list.
- Game engines – Manage game objects and animations.

Applications of Circular Linked Lists

- Round-robin scheduling in operating systems.
- Circular queues for memory management.
- Animation systems in games (loop through frames).
- Traffic light control systems – Phases cycle repeatedly.
- Streaming/buffer systems – For audio and video data.
- Signal processing – Circular buffer for continuous input.

Applications of Doubly Linked Lists

- Undo/Redo functionality in editors.
- Browser history – Navigate back and forward.
- Most Recently Used (MRU) caches.
- Operating system scheduling – Easy traversal in both directions.
- Graph algorithms with enhanced traversal.
- Game object interactions.

KAMARAJ WOMENS COLLEGE

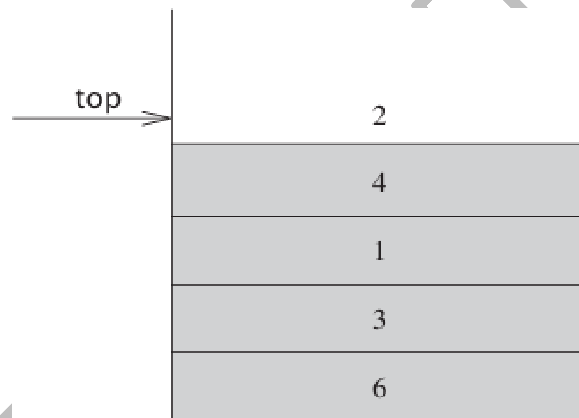
UNIT - II

The Stack ADT

A stack is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the top.

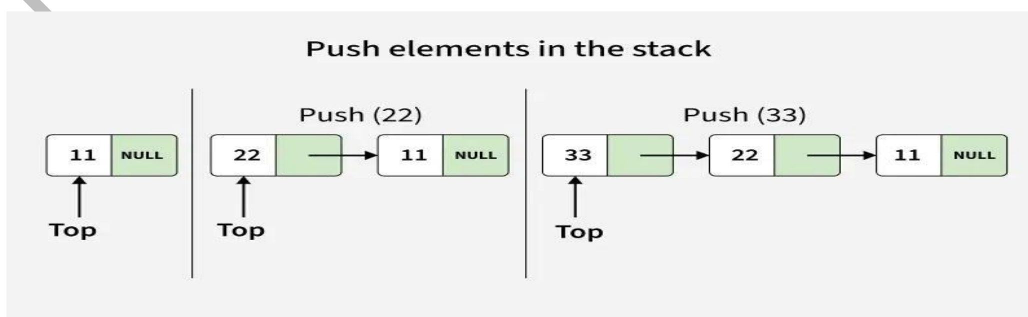
Stack Model

The fundamental operations on a stack are push, which is equivalent to an insert, and pop, which deletes the most recently inserted element. The most recently inserted element can be examined prior to performing a pop by use of the top routine. A pop or top on an empty stack is generally considered an error in the stack ADT. On the other hand, running out of space when performing a push is an implementation limit but not an ADT error. Stacks are sometimes known as LIFO (last in, first out) lists. The model depicted in signifies only that pushes are input operations and pops and tops are output. The usual operations to make empty stacks and test for emptiness are part of the repertoire, but essentially all that you can do to a stack is push and pop. An abstract stack after several operations. The general model is that there is some element that is at the top of the stack, and it is the only element that is visible.



Linked List Implementation of Stacks

The first implementation of a stack uses a singly linked list. We perform a push by inserting at the front of the list. We perform a pop by deleting the element at the front of the list. A top operation merely examines the element at the front of the list, returning its value. Sometimes the pop and top operations are combined into one.



KAMARAJ COLLEGE

```
#include <iostream>
#include <cstdlib>
using namespace std;
struct node {
int data;
struct node *next;
};
typedef struct node Node; Node *top = NULL;
void push(int val) {
Node *newNode = (Node*)malloc(sizeof(Node)); if (newNode == NULL) {
cout << "Stack Overflow (Out of Memory)!" << endl;
return;
}
newNode->data = val; newNode->next = top; top = newNode;
cout << val << " pushed into stack." << endl;
}
void pop() {
if (top == NULL) {
cout << "Stack Underflow (Empty Stack)!" << endl; return;
}
Node *temp = top;
cout << temp->data << " popped from stack." << endl; top = top->next;
free(temp);
}
void peek() {
if (top == NULL)
cout << "Stack is Empty!" << endl; else
cout << "Top element: " << top->data << endl;
```

KAMARAJ WOMENS COLLEGE

```
}
void display() {
if (top == NULL) {
cout << "Stack is Empty!" << endl; return;
}
cout << "Stack elements: "; Node *temp = top;
while (temp != NULL) { cout << temp->data << " "; temp = temp->next;
}
cout << endl;
}
int main() {
int choice, val; while (true) {
cout << "\n--- Stack Menu ---" << endl;
cout << "1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit" << endl;
cout << "Enter your choice: "; cin >> choice;
switch (choice) { case 1:
cout << "Enter value to push: "; cin >> val;
push(val); break;
case 2:
pop(); break;
case 3:
peek(); break;
case 4:
display(); break;
case 5:
cout << "Exiting..." << endl; exit(0);
default:
cout << "Invalid Choice!" << endl;
```

KAMARAJ WOMENS COLLEGE

```
}
}
return 0;
}
```

Applications of Stacks:

Function calls: Stacks are used to keep track of the return addresses of function calls, allowing the program to return to the correct location after a function has finished executing.

Recursion: Stacks are used to store the local variables and return addresses of recursive function calls, allowing the program to keep track of the current state of the recursion.

Expression evaluation: Stacks are used to evaluate expressions in postfix notation (Reverse Polish Notation).

Syntax parsing: Stacks are used to check the validity of syntax in programming languages and other formal languages.

Memory management: Stacks are used to allocate and manage memory in some operating systems and programming language.

Arithmetic Expression Evaluation Using Stack

The stack data structure is highly effective for evaluating arithmetic expressions because it naturally follows the LIFO (Last In, First Out) principle. This property is ideal for managing nested operations, parentheses, and operator precedence in arithmetic expressions.

Expressions can be written in different notations based on the placement of operators and operands.

Infix

Operator is written between operands.

Example:

- A + B
- (A + B) * C
- A + (B * C)

Requires parentheses or operator precedence to determine order of evaluation.

Commonly used in mathematical expressions and programming languages.

KAMARAJ WOMENS COLLEGE

Prefix

Operator is written before operands.

Example:

- +AB → (A + B)
- *+ABC → (A + B) * C

No parentheses are required.

Order of operations is clear and unambiguous.

Postfix

Operator is written after operands.

Example:

- AB+ → (A + B)
- ABC+* → A * (B + C)

No parentheses are needed.

Very efficient for evaluation using a stack.

Operator Precedence and Associativity

Precedence Level	Operators	Description
Highest	^	Exponentiation
Medium	*, /	Multiplication and Division
Lowest	+, -	Addition and Subtraction

Infix to Postfix Conversion Using Stack

In infix notation, operators are written between operands (e.g., A + B). However, computers find it easier to evaluate postfix (Reverse Polish) expressions where operators come after operands (e.g., AB+).

To evaluate expressions efficiently using stacks, infix expressions must be converted to postfix.

Why Conversion is Needed

- Infix expressions are ambiguous without parentheses or precedence rules.
- Postfix expressions require no parentheses.

KAMARAJ WOMENS COLLEGE

- Evaluation of postfix expressions is easier and faster using a stack.

Advantages

- Removes ambiguity of operator precedence.
- Postfix expression can be directly evaluated using a stack.
- Simplifies compiler parsing and expression evaluation.

Limitations

- Conversion must be done before evaluation.
- Requires careful handling of associativity and parentheses.

Rules

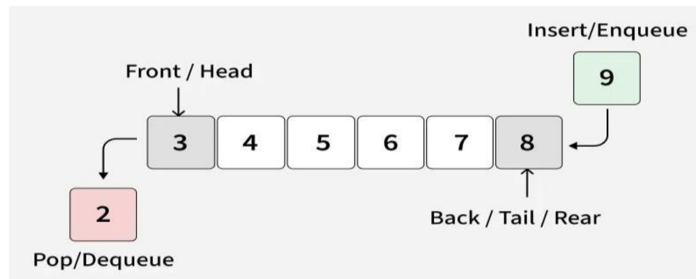
1. Operands are added directly to the postfix expression in the same order.
2. '(' is always pushed onto the stack.
3. ')' causes popping from the stack until a matching '(' is found.
4. Operator precedence order:
   a. Highest → ^
   b. Next → *, /
   c. Lowest → +, -
5. Associativity:
   a. Left to Right → +, -, *, /
   b. Right to Left → ^
6. When an operator appears:
   a. Pop from the stack while the top has higher precedence, or
   b. Equal precedence and the new operator is left-associative,
   c. Then push the new operator.
7. After the entire expression is scanned, pop all remaining operators from the stack and add them to the postfix expression.

QUEUE ATD

A Queue is a linear data structure used to store and manage data in a specific order following the FIFO (First In, First Out) principle.

KAMARAJ WOMENS COLLEGE

The first element inserted is the first one to be removed.



FIFO (First In, First Out) means:

The element added first will be removed first.

Real-life Example:

A queue of people waiting for tickets — the person who comes first is served first.

→ First Come, First Serve

Applications of Queue

1. Buffer management
   a. Used as a buffer when there is a speed mismatch between two devices (e.g., CPU and keyboard, network devices).
2. Operating Systems
   a. Used in CPU Scheduling, Memory Management, and Disk Scheduling.
3. Data Structures & Algorithms
   a. Used in Breadth First Search (BFS) for graphs.
   b. Used in Level Order Traversal of trees.
4. Communication Systems
   a. Used in message queues, task scheduling, and packet processing.

```
#include <iostream>
 using namespace std;
struct Node {
 int data;
Node* next;
```

KAMARAJ WOMENS COLLEGE

```
};
class Queue {
Node *front, *rear;
public:
Queue() {
front = rear = NULL;
}
void enqueue(int val) {
Node* newNode = new Node(); newNode->data = val; newNode->next = NULL;
if (rear == NULL) {
front = rear = newNode;
} else {
rear->next = newNode; rear = newNode;
}
cout << val << " inserted\n";
}
void dequeue() {
if (front == NULL) {
cout << "Queue is Empty\n"; return;
}
Node* temp = front;
cout << front->data << " deleted\n"; front = front->next;
if (front == NULL)
rear = NULL; delete temp;
}
void display() {
if (front == NULL) {
cout << "Queue is Empty\n"; return;
```

KAMARAJ WOMENS COLLEGE

```
}
Node* temp = front; while (temp != NULL) {
cout << temp->data << " "; temp = temp->next;
}
cout << endl;
}
};
int main()  {
Queue q;
q.enqueue(10);
q.enqueue(20);
q.enqueue(30);
q.display();
q.dequeue();
q.display();
}
```

Basic Operations

Operation	**Description**
Enqueue	Add (insert) an element to the rear (end) of the queue
Dequeue	Remove (delete) an element from the front of the queue
Peek / Front	View the element at the front without removing it
isEmpty	Check if the queue is empty
isFull	Check if the queue is full (for fixed-size queues)

Circular Queue

Definition

A Circular Queue is an advanced form of a linear queue in which the last position is connected back to the first position to make a circle.

KAMARAJ WOMENS COLLEGE

It overcomes the major drawback of the simple queue — wasted memory space after several enqueue and dequeue operations.

A circular queue is a linear data structure that overcomes the limitations of a simple queue. In a normal array implementation, dequeue() can be O(n) or we may waste space. Using a circular array, both enqueue() both enqueue() and dequeue() can be done in O(1).

```
#include <iostream> using namespace std;

#define SIZE 5

int queue[SIZE];

int front = -1, rear = -1;

// Function to check if queue is full bool isFull() {

return (front == (rear + 1) % SIZE);

}

// Function to check if queue is empty bool isEmpty() {

return (front == -1);

}

// Function to insert element void enqueue(int element) {

if (isFull()) {

cout << "Queue is Full\n"; return;

}

if (isEmpty()) { front = rear = 0;

} else {

rear = (rear + 1) % SIZE;

}

queue[rear] = element;

cout << element << " inserted\n";

}

// Function to delete element void dequeue() {

if (isEmpty()) {

cout << "Queue is Empty\n"; return;
```

KAMARAJ WOMENS COLLEGE

```
}
cout << queue[front] << " deleted\n"; if (front == rear) {
// Only one element was present front = rear = -1;
} else {
front = (front + 1) % SIZE;
}
}
// Function to display queue void display() {
if (isEmpty()) {
cout << "Queue is Empty\n"; return;
}
cout << "Queue elements: "; int i = front;
while (true) {
cout << queue[i] << " "; if (i == rear)
break;
i = (i + 1) % SIZE;
}
cout << endl;
}
int main()
{
enqueue(10);
enqueue(20);
enqueue(30);
enqueue(40);
enqueue(50);
display();
dequeue();
```

KAMARAJ WOMENS COLLEGE

```
dequeue();
enqueue(60);
enqueue(70);
display();
return 0;
}
```

Priority Queue

A Priority Queue is a special type of queue in which each element has a priority value associated with it.

- The element with higher priority is served first.
- If two elements have the same priority, they are served according to their order of insertion (FIFO).
- Unlike a normal queue (which follows FIFO), a priority queue follows:
- Elements are dequeued in order of priority, not arrival time.

In a priority queue, generally, the value of an element is considered for assigning the priority. For example, the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority. The reverse case can also be used i.e., the element with the lowest value can be assigned the highest priority. Also, the priority can be assigned according to our needs.

Operations on a Priority Queue

A typical priority queue supports the following operations:

Insertion : If the newly inserted item is of the highest priority, then it is inserted at the top. Otherwise, it is inserted in such a way that it is accessible after all higher priority items are accessed.

Deletion : We typically remove the highest priority item which is typically available at the top. Once we remove this item, we need not move next priority item at the top.

Peek : This operation only returns the highest priority item (which is typically available at the top) and does not make any change to the priority queue.

Deque (Double-Ended Queue)

A Deque (pronounced “deck”) is a linear data structure that allows insertion and deletion of elements from both ends — the front and the rear.

KAMARAJ WOMENS COLLEGE

It is a generalization of a queue, capable of functioning as both a Queue (FIFO) and a Stack (LIFO).

Feature	Description
Double-Ended Operations	Insertions and deletions can be performed at both ends — front and rear.
Flexibility	Can behave as both stack and queue.
Dynamic Behavior	Suitable for situations where data needs to be added or removed from either side.

Types of Deque

Type	Description
Input-Restricted Deque	Insertion allowed only at one end, but deletion at both ends.
Output-Restricted Deque	Deletion allowed only at one end, but insertion at both ends.

Applications

- **Undo/Redo Functionality:**

  Storing a history of actions for redo and undo operations in software applications.
- **Browser History:**

  Managing recent URLs, with new pages added to the front and older ones removed from the back.
- **Graph Traversal:**

  Used in algorithms like Breadth-First Search (BFS) to store nodes for processing.
- **Task Management:**

  Handling tasks with varying priorities by adding or removing them from different ends of the deque.

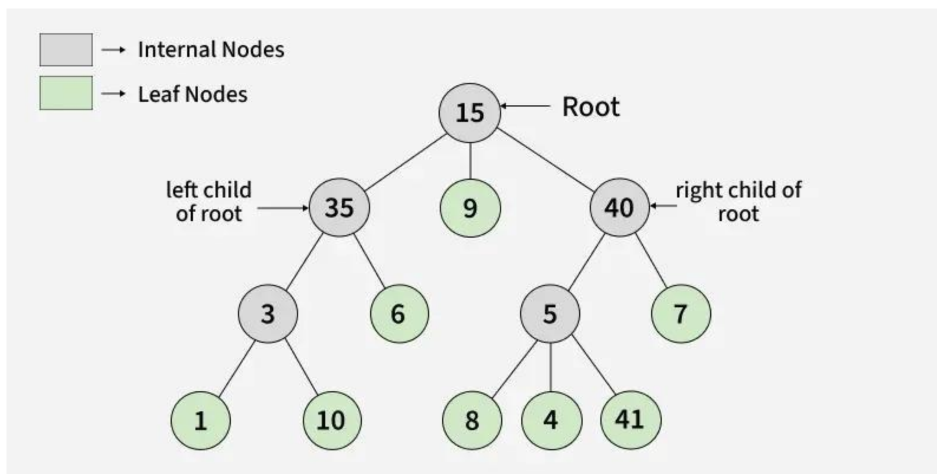
KAMARAJ WOMENS COLLEGE

UNIT - III

Tree ADT

A tree is a hierarchical data structure used to organize and represent data in a parent – child relationship.

It consists of nodes, where the topmost node is called the root, and every other node can have one or more child nodes



Basic Terminologies In Tree Data Structure:

Parent Node: A node that is an immediate predecessor of another node. Example: 35 is the parent of 3 and 6.

Child Node: A node that is an immediate successor of another node. Example: 3 and 6 are children of 35.

Root Node: The topmost node in a tree, which does not have a parent. Example: 15 is the root node.

Leaf Node (External Node): Nodes that do not have any children. Example: 1, 10, 12, 5, 7, 7 are leaf nodes.

Ancestor: Any node on the path from the root to a given node (excluding the node itself).

Example: 15 and 35 are ancestors of 10.

Descendant: A node x is a descendant of another node y if y is an ancestor of x. Example: 1, 10, and 6 are descendants of 35.

Sibling: Nodes that share the same parent. Example: 1 and 10 are siblings, and 5 and 7 are siblings.

KAMARAJ WOMEN'S COLLEGE

Level of a Node: The number of edges in the path from the root to that node. The root node is at level 0.

Internal Node: A node with at least one child.

Neighbor of a Node: The parent or children of a node.

Subtree: A node and all its descendants form a subtree

Tree Traversal

Tree traversal means visiting each node in a tree exactly once in a specific order. Traversal helps perform operations like displaying, searching, inserting, or evaluating expressions on tree data.

- Inorder Traversal (Left, Root, Right)
- Preorder Traversal (Root, Left, Right)
- Postorder Traversal (Left, Right, Root)

Inorder Traversal (Left → Root → Right)

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

Preorder Traversal (Root → Left → Right)

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

Postorder Traversal (Left → Right → Root)

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

Applications of Tree Traversals

Traversal Applications

Inorder

Retrieves data in sorted order for Binary Search Trees

KAMARAJ WOMENS COLLEGE

Preorder

Used to create a copy of the tree / prefix expression

Postorder

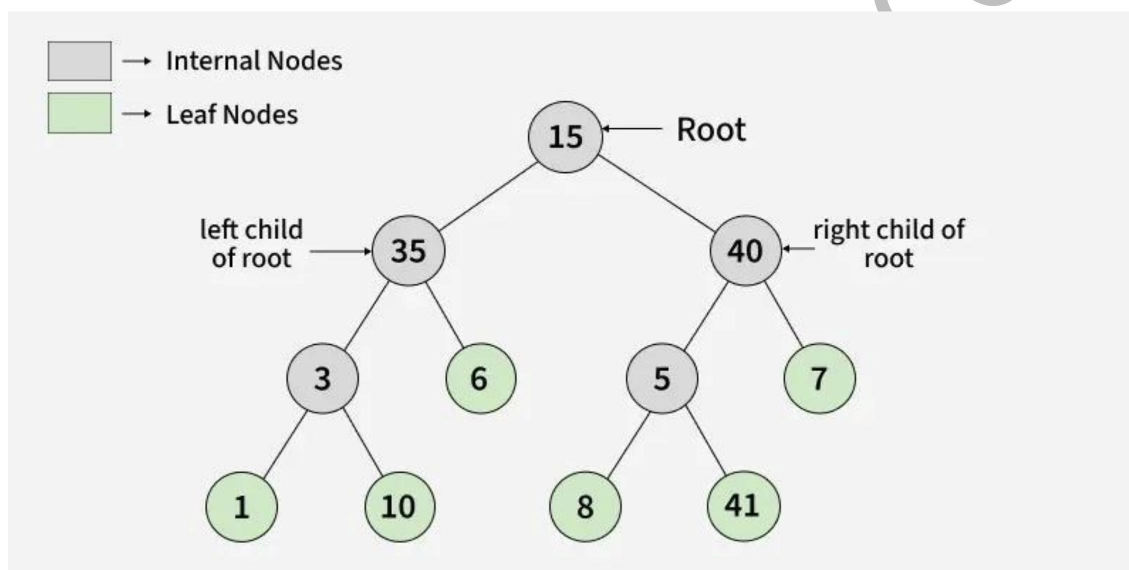
Used to delete a tree / evaluate postfix expressions

Level Order

Used in Breadth-First Search (BFS), shortest-path algorithms.

Binary Tree

Binary Tree is a non-linear and hierarchical data structure where each node has at most two children referred to as the left child and the right child. The topmost node in a binary tree is called the root, and the bottom-most nodes(having no children) are called leaves.



Properties of a Binary Tree

- Maximum number of nodes at level ‘l’ = 2^l
- Maximum number of nodes in a tree of height ‘h’ = (2^(h+1)) − 1
- Minimum possible height for ‘n’ nodes = $\lceil \log_2(n + 1) \rceil - 1$
- Binary Tree of height h has between h + 1 and 2^(h+1) − 1 nodes.

Advantages of Binary Trees

1. Structured Organization: Offers a clear, hierarchical data structure. Efficient Searching and Sorting: BSTs facilitate fast data operations.

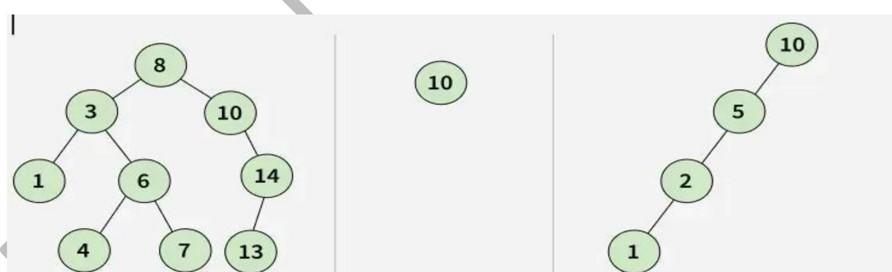
2. Balanced Storage: Variants like AVL and Red-Black trees ensure balanced performance (O(log n)).
3. Flexibility: Adaptable to various specialized structures (e.g., heaps, BSTs). Recursion Support: Naturally aligns with recursive algorithms.
4. Scalability: Suitable for managing large dynamic datasets. Disadvantages of Binary Trees
5. Skewed Trees: Unbalanced trees can degrade performance to O(n), similar to linked lists.
6. Memory Overhead: Additional pointers in each node increase memory usage.
7. Complex Implementation: Balancing trees (e.g., AVL, Red-Black) requires sophisticated rotations.
8. Limited Degree: Restricts each node to two children, which might not be ideal for some applications.

Binary Search Tree

A Binary Search Tree (BST) is a type of binary tree data structure in which each node contains a unique key and satisfies a specific ordering property:

- All nodes in the left subtree of a node contain values strictly less than the node's value.
- All nodes in the right subtree of a node contain values strictly greater than the node's value.

This structure enables efficient operations for searching, insertion, and deletion of elements, especially when the tree remains balanced.



Characteristics of a BST:

1. Hierarchical Structure: A BST is composed of nodes, each having up to two children, forming a tree-like hierarchy with a single root node at the top.
2. Ordering Property: For every node in the BST, all values in the left subtree are smaller, and all values in the right subtree are larger than the node's value. This rule holds recursively for all subtrees.

KAMARAJ WOMENS COLLEGE

3. Efficient Operations: In a balanced BST, operations like search, insertion, and deletion can be performed in O(log n) time. In the worst-case (unbalanced), these degrade to O(n). With self-balancing BSTs like AVL and Red Black Trees, we can ensure the worst case as O(Log n).
4. Recursive Nature: Each left or right subtree of a node in a BST is itself a BST, allowing recursive algorithms to naturally process the tree.
5. Practical Applications: BSTs are widely used in database indexing, symbol tables, range queries, and are foundational for advanced structures like AVL trees, Red-Black trees. In problem solving, BSTs are used in problems where we need to maintain sorted stream of data

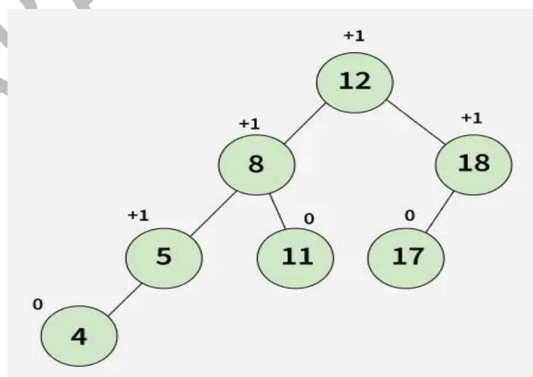
Applications of BST

Application	Description
Searching	Quick lookup due to ordered structure
Sorting	Inorder traversal gives sorted data
Range Queries	Count or print elements between two values
Dynamic Sets	Supports insertion/deletion efficiently
Symbol Tables	Used in compilers and interpreters

AVL Tree

An AVL tree defined as a self-balancing Binary Search Tree (BTS) where the difference between heights of left and right subtrees for any node cannot be more than one.

Balance Factor = left subtree height - right subtree height For a Balanced Tree(for every node): -1 ≤ Balance Factor ≤ 1



AVL Tree:

Rotations: rotations are designed to restore balance in O(1) time while ensuring the overall time complexity remains O(log n).

KAMARAJ WOMENS COLLEGE

AVL Trees use four types of rotations to rebalance themselves after insertions and deletions:

- Left-Left (LL) Rotation
- Right-Right (RR) Rotation
- Left-Right (LR) Rotation
- Right-Left (RL) Rotation

Insertion and Deletion:

While insertion is followed by upward traversals to check balance and apply rotations, deletion can be more complex due to multiple rotations possibly being required.

AVL Trees may require multiple rebalancing steps during deletion, unlike Red-Black Trees which limit this better.

Use Cases: AVL Trees are particularly useful when you need frequent and efficient lookups, like in database indexing, memory-intensive applications, or where predictable time complexity is crucial.

Drawbacks Compared to Other Trees:

Although faster in lookups than Red-Black Trees, AVL Trees might incurslightly more overhead on insertions and deletions due to stricter balancing requirements.

As a result, Red-Black Trees are more common in standard libraries like TreeMap or TreeSet in Java or map in C++ STL.

In-order Traversal:

An in-order traversal of an AVL Tree still gives you elements in sorted order, just like any Binary Search Tree.

Rotating the subtrees (Used in Insertion and Deletion)

An AVL tree may rotate in one of the following four ways to keep itself balanced while making sure that the BST properties are maintained.

Left-Left Rotation:

Occurs when a node is inserted into the left subtree of the left child, causing the balance factor to become more than +1.

Fix: Perform a single right rotation.

Right-Right Rotation:

Occurs when a node is inserted into the right subtree of the right child, making the balance factor less than -1.

KAMARAJ WOMENS COLLEGE

Fix: Perform a single left rotation. 2 / 3

Left-Right Rotation:

Occurs when a node is inserted into the right subtree of the left child, which disturbs the balance factor of an ancestor node, making it left-heavy.

Fix: Perform a left rotation on the left child, followed by a right rotation on the node.

Right-Left Rotation:

Occurs when a node is inserted into the left subtree of the right child, which disturbs the balance factor of an ancestor node, making it right-heavy.

Fix: Perform a right rotation on the right child, followed by a left rotation on the node.

Heap Data Structure

A Heap is a complete binary tree that satisfies the heap property: Every node follows a specific order with respect to its children:

- Max Heap: Key of parent ≥ keys of children
- Min Heap: Key of parent ≤ keys of children
- Complete Binary Tree: All levels are fully filled except possibly the last, which is filled from left to right.

Types of Heaps

Type	Heap Property
Max Heap	Parent ≥ Children Maximum element
Min Heap	Parent ≤ Children Minimum element

Heap Operations

Insertion

- Insert new element at the end of the heap.
- Heapify-up / Bubble-up: Swap with parent until heap property is restored.
- Time Complexity: O(log n)

Deletion (Extract Max / Min)

- Remove the root element (max for max-heap, min for min-heap).
- Replace root with last element.

KAMARAJ WOMENS COLLEGE

- Heapify-down / Bubble-down: Swap with largest/smallest child until heap property is restored.
- Time Complexity: O(log n)

Peek / Get Max or Min

- Return the root element without removing it.
- Time Complexity: O(1)

Heapify / Build Heap

- Rearrange an arbitrary array into a heap.
- Bottom-up heapify can build a heap in O(n) time.

Applications of Heap

Application	**Description**
Priority Queue	Tasks executed based on priority
Heap Sort	Sort array in O(n log n)
Graph Algorithms	Dijkstra's algorithm, Prim's algorithm
Median Finding	Maintain two heaps for streaming median

KAMARAJ WOMENS COLLEGE

UNIT - IV

Graph Data Structure

Graph Data Structure is a collection of nodes connected by edges. It's used to represent relationships between different entities.

If you are looking for topic-wise list of problems on different topics like DFS, BFS, Topological Sort, Shortest Path, etc.

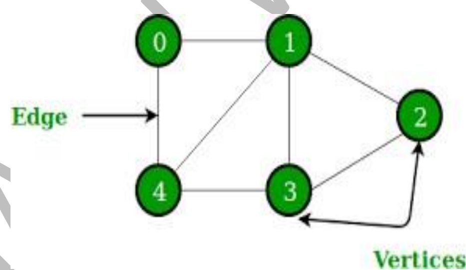
Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).

Imagine a game of football as a web of connections, where players are the nodes and their interactions on the field are the edges. This web of connections is exactly what a graph data structure represents, and it's the key to unlocking insights into team performance and player dynamics in sports.

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way.

There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled / unlabelled.

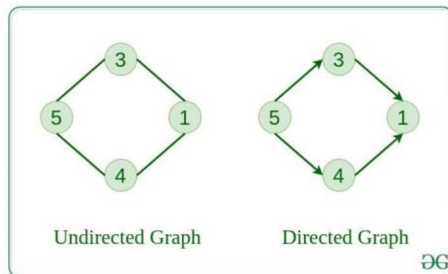


Type	Description
Directed Graph (Digraph)	Edges have a direction (u → v)
Undirected Graph	Edges have no direction (u — v)
Weighted Graph	Edges have weights/costs
Unweighted Graph	All edges treated equally
Cyclic Graph	Graph contains a cycle

KAMARAJ WOMENS COLLEGE

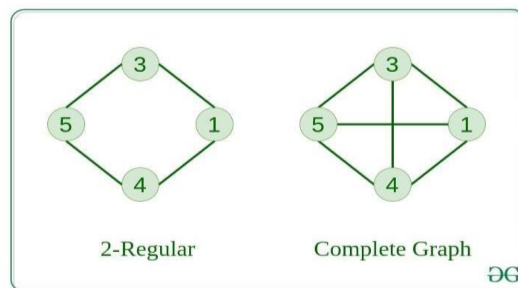
Directed Graph

A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.



Complete Graph

The graph in which from each node there is an edge to each other node.



Representations of Graphs

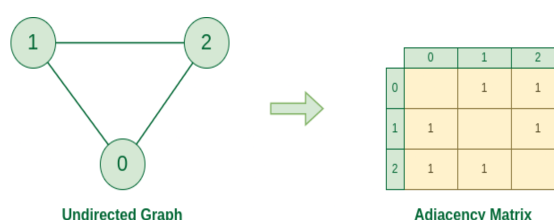
Graphs can be represented in multiple ways, but the two most common representations are:

1. Adjacency Matrix
2. Adjacency List

We will focus on unweighted graphs for simplicity.

Adjacency Matrix

An adjacency matrix is a 2D array of size n × n, where n is the number of vertices. It uses 0s and 1s to indicate the presence or absence of edges between vertices.



KAMARAJ WOMENS COLLEGE

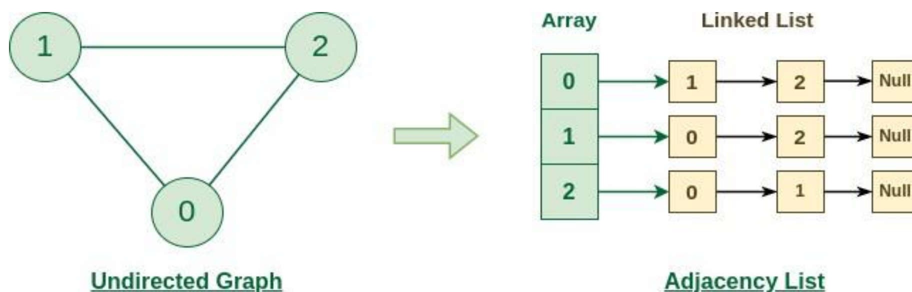
Adjacency List

An adjacency list represents a graph as an array of lists. Each vertex has a list containing all vertices connected to it.

How it works:

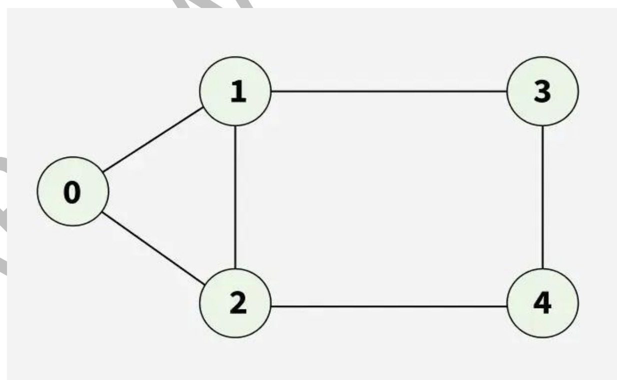
Create an array of size n, where each element is a list.

For vertex i, the list contains all vertices j such that there is an edge i-j.



Breadth First Search or BFS

Given a undirected graph represented by an adjacency list adj, where each adj[i] represents the list of vertices connected to vertex i. Perform a Breadth First Search (BFS) traversal starting from vertex 0, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph



Algorithm Steps (for a graph G = (V, E))

1. Start at a source vertex s.
2. Mark s as visited and enqueue it into a queue.
3. While the queue is not empty:
    a. Dequeue a vertex v from the front.

KAMARAJ WOMENS COLLEGE

b. Visit v (process it, e.g., print it).

c. For each neighbour u of v:

d. If u is not visited, mark it visited and enqueue it.

Applications of BFS in Graphs

BFS has various applications in graph theory and computer science, including:

Shortest Path Finding: BFS can be used to find the shortest path between two nodes in an unweighted graph. By keeping track of the parent of each node during the traversal, the shortest path can be reconstructed.

Cycle Detection: BFS can be used to detect cycles in a graph. If a node is visited twice during the traversal, it indicates the presence of a cycle.

Connected Components: BFS can be used to identify connected components in a graph. Each connected component is a set of nodes that can be reached from each other.

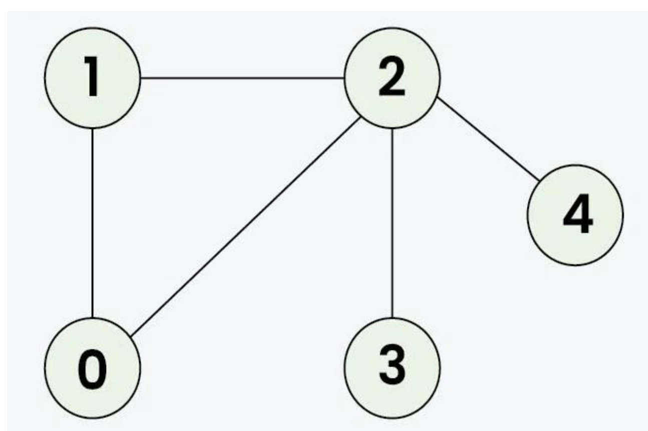
Topological Sorting: BFS can be used to perform topological sorting on a directed acyclic graph (DAG). Topological sorting arranges the nodes in a linear order such that for any edge (u, v), u appears before v in the order.

Level Order Traversal of Binary Trees: BFS can be used to perform a level order traversal of a binary tree. This traversal visits all nodes at the same level before moving to the next level.

Network Routing: BFS can be used to find the shortest path between two nodes in a network, making it useful for routing data packets in network protocols

Depth First Search or DFS

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. This is similar to a depth first tree traversal, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.



KAMARAJ WOMENS COLLEGE

Algorithm Steps (Recursive Version)

- Start at a source vertex v.
- Mark v as visited.
- Process v (e.g., print it).
- For each neighbour u of v:
  - If u is not visited, recursively perform DFS on u.

Applications of DFS (Depth-First Search)

1. **Topological Sorting**
   - DFS is used in Directed Acyclic Graphs (DAGs) to find an order of tasks or jobs.
   - Example: Scheduling tasks in project management.
2. **Cycle Detection**
   - DFS can detect cycles in both directed and undirected graphs using recursion stack.
3. **Connected Components**
   - DFS can explore all vertices in a connected component of a graph.
4. **Path Finding / Maze Solving**
   - DFS can explore all possible paths from a source to a destination in mazes or puzzles.
5. **Strongly Connected Components (SCC)**
   - DFS is the backbone of Kosaraju's algorithm and Tarjan's algorithm to find SCCs in directed graphs.
6. **Solving Games / Backtracking Problems**
   - DFS is used in n-queens problem, sudoku, and other constraint satisfaction problems.

Applications of Graphs

Computer Networks

Representation: Nodes = computers/routers, Edges = connections (wired/wireless).

Use: Network routing, shortest path algorithms (e.g., BFS/Dijkstra) for data transmission.

KAMARAJ WOMENS COLLEGE

Social Networks

Representation: Nodes = people/accounts, Edges = friendships or followers.

Use: Finding friends of friends, degrees of separation, community detection.

Web and Internet

Representation: Nodes = web pages, Edges = hyperlinks.

Use: Web crawling, PageRank algorithm (Google search), link analysis.

Maps and Navigation

Representation: Nodes = intersections, Edges = roads.

Use: Shortest path algorithms (BFS, Dijkstra, A*), GPS navigation, traffic flow optimization.

Project Scheduling

Representation: Nodes = tasks, Edges = dependencies.

Use: Topological sorting to find the order of tasks in a project (e.g., PERT, CPM).

Transport Networks

Representation: Nodes = stations/airports, Edges = train routes or flights.

Use: Optimal routing, shortest distance, connectivity analysis.

Electrical Circuits

Representation: Nodes = components, Edges = connections/wires.

Use: Circuit analysis, detecting loops or isolated components.

Biology

Representation: Nodes = species or proteins, Edges = interactions.

Use: Analyzing protein-protein interaction networks, ecological food webs.

Game Development

Representation: Nodes = game states, Edges = moves/actions.

Use: AI pathfinding (DFS/BFS/A*), puzzle solving.

Recommendation Systems

Representation: Nodes = users/items, Edges = interactions (likes, purchases).

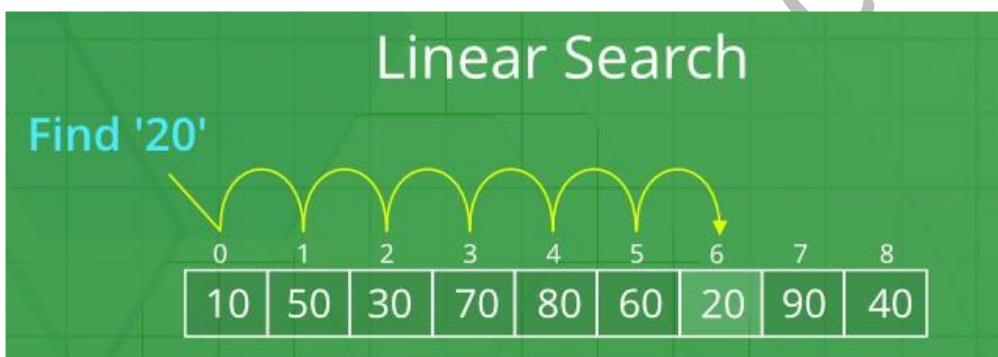
Use: Suggesting products, movies, or friends based on connections.

KAMARAJ WOMENS COLLEGE

UNIT - V

Linear Search

- Linear Search is a sequential searching algorithm used to find the position of a target element (key) in a given array or list.
- It checks each element one by one until the key is found or the end of the array is reached.
- The search begins from the first element (index 0).
- Each element is compared with the key value.
- If a match is found → search is successful.
- If the end of the array is reached without finding the key → search is unsuccessful.




```
#include <iostream>
using namespace std;
int linearSearch(int arr[], int n, int key) { for (int i = 0; i < n; i++) {
if (arr[i] == key) { return i;
}
}
return -1;
}

int main() { int n, key;
cout << "Enter number of elements: "; cin >> n;
int arr[n];
cout << "Enter " << n << " elements:\n"; for (int i = 0; i < n; i++) {
cin >> arr[i];
}
cout << "Enter element to search: "; cin >> key;
```

KAMARAJ WOMEN'S COLLEGE

```
int result = linearSearch(arr, n, key); if (result == -1)
cout << "Element not found in the array." << endl; else
cout << "Element found at index " << result << " (position " << result + 1 << ")." << endl;
return 0;
}
```

Advantages

- Simple and easy to implement.
- Works on both sorted and unsorted data.

Disadvantages

- Inefficient for large datasets.
- Time-consuming since it checks each element sequentially.

Binary Search

Binary Search is an efficient searching algorithm used to find the position of a target (key) element within a sorted array.

It repeatedly divides the search space in half, reducing the number of comparisons. Works only on sorted data (ascending or descending order).

- Compare the key with the middle element of the array:
- If key == middle element, search is successful.
- If key < middle element, search the left half.
- If key > middle element, search the right half.
- Repeat the process until the element is found or the range becomes invalid.

RAJ WOMENS COLLEGE

```
#include <iostream>
using namespace std;
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; // Key found else if (key < arr[mid])
high = mid - 1; // Search left half else
low = mid + 1; // Search right half
}
return -1; // Key not found
}
int main() { int n, key;
cout << "Enter number of elements: "; cin >> n;
int arr[n];
cout << "Enter " << n << " elements in sorted order:\n"; for (int i = 0; i < n; i++) {
cin >> arr[i];
}

cout << "Enter element to search: "; cin >> key;
int result = binarySearch(arr, n, key); if (result == -1)
cout << "Element not found in the array." << endl; else
cout << "Element found at index " << result << " (position " << result + 1 << ")." << endl;
return 0;
}
```

Advantages

- Much faster than Linear Search for large sorted datasets.
- Requires fewer comparisons.

Disadvantages

- Works only on sorted data.
- More complex logic compared to Linear Search.

Bubble Sort

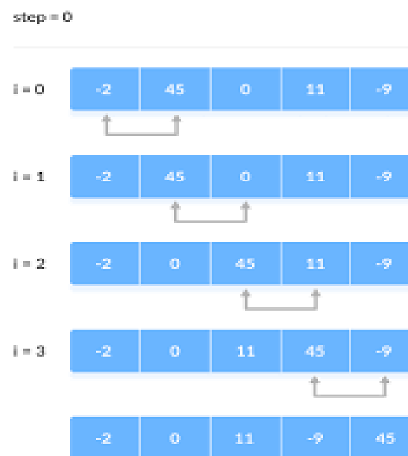
Bubble Sort is a simple comparison-based sorting algorithm.

It repeatedly compares adjacent elements and swaps them if they are in the wrong order.

This process continues until the entire array is sorted.

KAMARAJ WOMENS COLLEGE

- Start from the first element and compare it with the next.
- If the current element > next element, swap them.
- Continue this for all elements → after the first pass, the largest element will "bubble up" to the end.
- Repeat the process for the remaining elements (excluding the last sorted part).




```
#include <iostream>
using namespace std;
void bubbleSort(int arr[], int n) { for (int i = 0; i < n - 1; i++) { bool swapped = false;
for (int j = 0; j < n - i - 1; j++) { if (arr[j] > arr[j + 1]) {
int temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp; swapped = true;
}
}
if (!swapped) break;
}
}

int main() { int n;
cout << "Enter number of elements: "; cin >> n;
int arr[n];
cout << "Enter " << n << " elements:\n"; for (int i = 0; i < n; i++) {
cin >> arr[i];
}
bubbleSort(arr, n);
cout << "Sorted array (Ascending Order): "; for (int i = 0; i < n; i++)
cout << arr[i] << " "; cout << endl;
return 0;
}
```

KAMARAJ WOMEN'S COLLEGE

Advantages

- Simple and easy to understand.
- Works well for small datasets.
- In-place sorting (no extra memory needed).

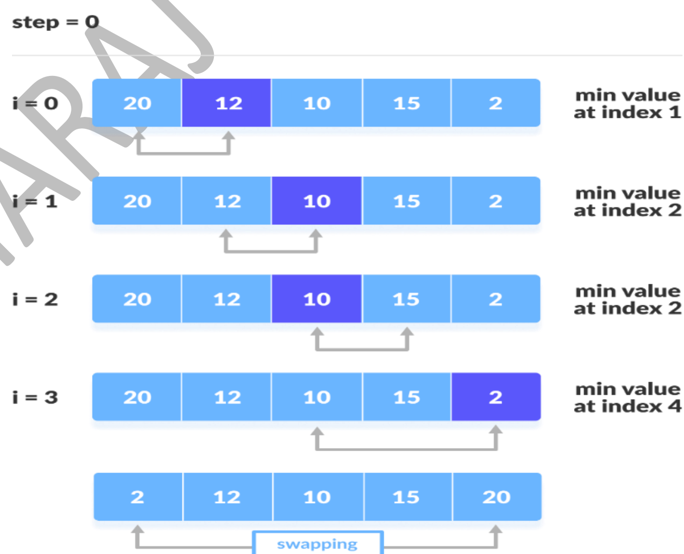
Disadvantages

- Very slow for large datasets ($O(n^2)$ time).
- Inefficient compared to advanced algorithms like Quick Sort or Merge Sort.

Selection Sort

Selection Sort is a comparison-based sorting algorithm that sorts an array by repeatedly finding the smallest (or largest) element from the unsorted part of the array and placing it at the beginning of the unsorted section.

- It divides the array into two parts:
- Sorted part (initially empty)
- Unsorted part (initially the whole array)
- Find the minimum element in the unsorted portion of the array.
- Swap it with the first element of the unsorted part.
- Move the boundary between the sorted and unsorted parts forward. Repeat until the entire array is sorted.



KAMARAJ WOMENS COLLEGE

Insertion sort

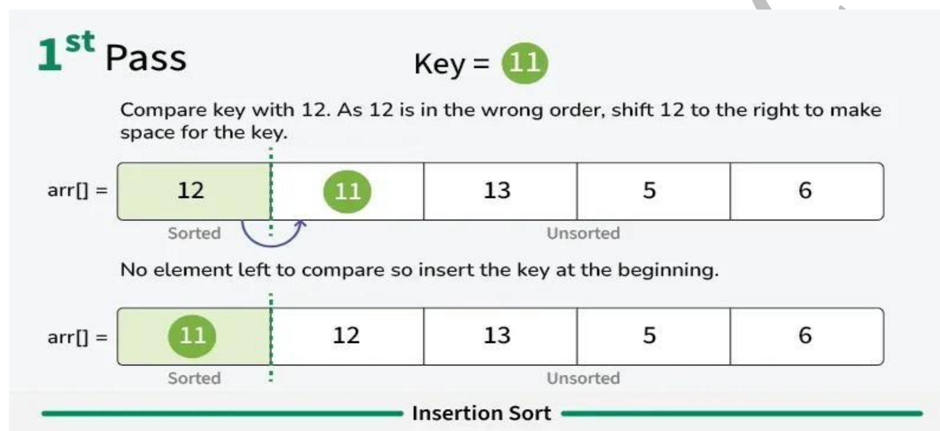
Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

We start with the second element of the array as the first element is assumed to be sorted.

Compare the second element with the first element if the second element is smaller then swap them.

Move to the third element, compare it with the first two elements, and put it in its correct position

Repeat until the entire array is sorted.




```
#include <iostream> using namespace std;
void insertionSort(int arr[], int n) { for (int i = 1; i < n; i++) {
int key = arr[i]; int j = i - 1;




while (j >= 0 && arr[j] > key) { arr[j + 1] = arr[j];
j--;
}
arr[j + 1] = key;
}
}
int main() {
int arr[] = {5, 3, 4, 1, 2};
```

KAMARAJ ... COLLEGE

```
int n = sizeof(arr) / sizeof(arr[0]); insertionSort(arr, n);
cout << "Sorted array: "; for (int i = 0; i < n; i++) cout << arr[i] << " ";
cout << endl; return 0;
}
```

Advantages of Insertion Sort

- Simple and Easy to Implement
- The algorithm is straightforward and easy to understand for beginners.
- In-Place Sorting
- Requires no extra space; sorting is done within the original array (space complexity = O(1)).
- Adaptive (Efficient for Nearly Sorted Data)
- If the array is already or almost sorted, the time complexity approaches O(n) — very fast in that case.
- Stable Sorting
- Preserves the relative order of equal elements (important in sorting records or objects).
- Online Algorithm
- Can sort data as it comes — useful when elements are received in a stream.
- Good for Small Datasets
- Performs well on small lists compared to more complex algorithms like Quick Sort or Merge Sort.

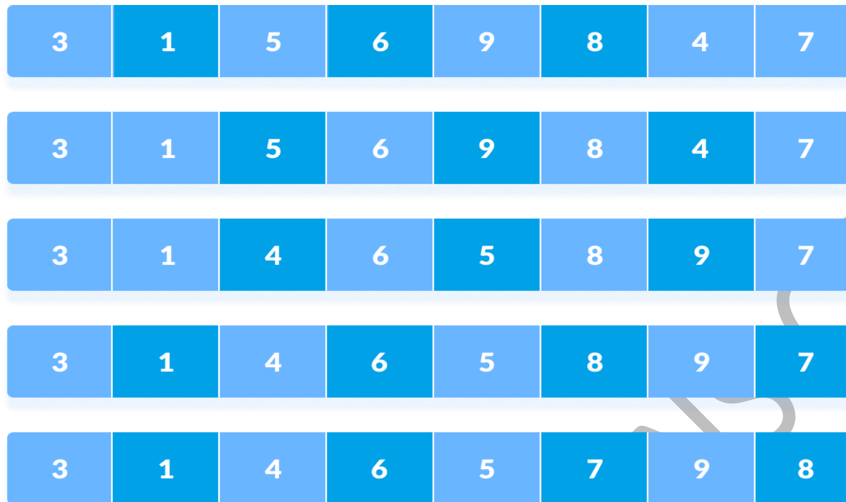
Disadvantages of Insertion Sort

- Inefficient for Large Data Sets
- Time complexity in the average and worst case is $O(n^2)$ — much slower than efficient algorithms like Merge Sort or Quick Sort.
- More Comparisons and Shifts
- Each insertion may require shifting many elements, especially when the smallest element is near the end.
- Not Suitable for Complex or Large Inputs
- For very large datasets, it becomes impractical due to high processing time.

KAMARAJ WOMENS COLLEGE

Shell Sort

Shell sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element are sorted.




```
#include <iostream>
 using namespace std;
void shellSort(int arr[], int n) {
for (int gap = n / 2; gap > 0; gap /= 2) { for (int i = gap; i < n; i++) {
int temp = arr[i]; int j;
location for arr[i]
for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) { arr[j] = arr[j - gap];
}
arr[j] = temp;
}
}
}
int main() {
int arr[] = {23, 12, 1, 8, 34, 54, 2, 3};
int n = sizeof(arr) / sizeof(arr[0]); shellSort(arr, n);
cout << "Sorted array: "; for (int i = 0; i < n; i++) cout << arr[i] << " ";
cout << endl; return 0;
}
```

KAMARAJ WOMEN'S COLLEGE

Advantages of Shell Sort

- Faster than Insertion Sort
- Far-apart comparisons help reduce the total number of movements.
- In-place sorting
- No extra memory required.
- Flexible gap sequences
- Different sequences (Knuth, Hibbard, Sedgewick) can make it faster.
- Efficient for Medium-Sized Datasets
- Performs well for arrays of moderate size (better than $O(n^2)$ sorts).

Disadvantages of Shell Sort

- Not Stable
- Equal elements may get swapped and lose their original order.
- Complex to Analyze
- The exact time complexity depends on the chosen gap sequence.
- Not Ideal for Very Large Data
- Still slower than advanced algorithms like Merge Sort or Quick Sort.

Radix sort

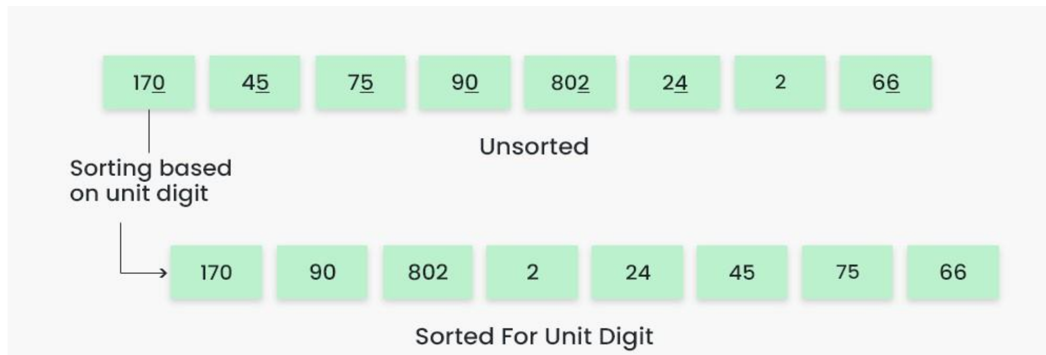
Radix Sort is a linear sorting algorithm (for fixed length digit counts) that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

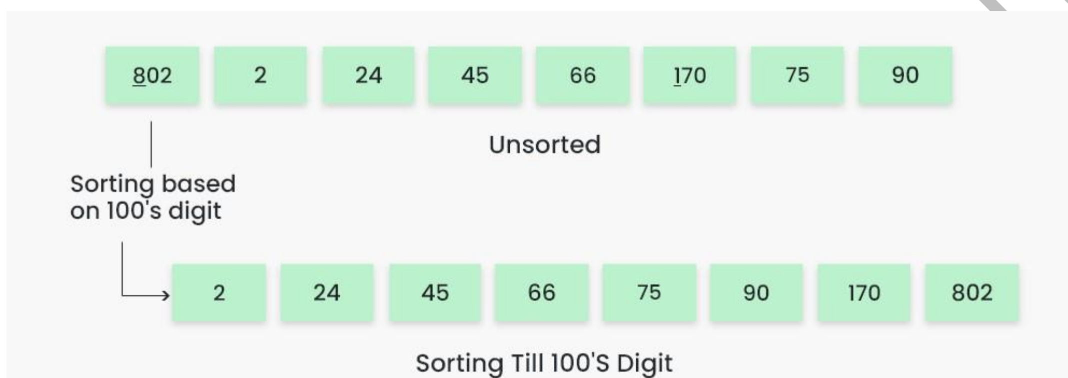
It repeatedly distributes the elements into buckets based on each digit's value. This is different from other algorithms like Merge Sort or Quick Sort where we compare elements directly.

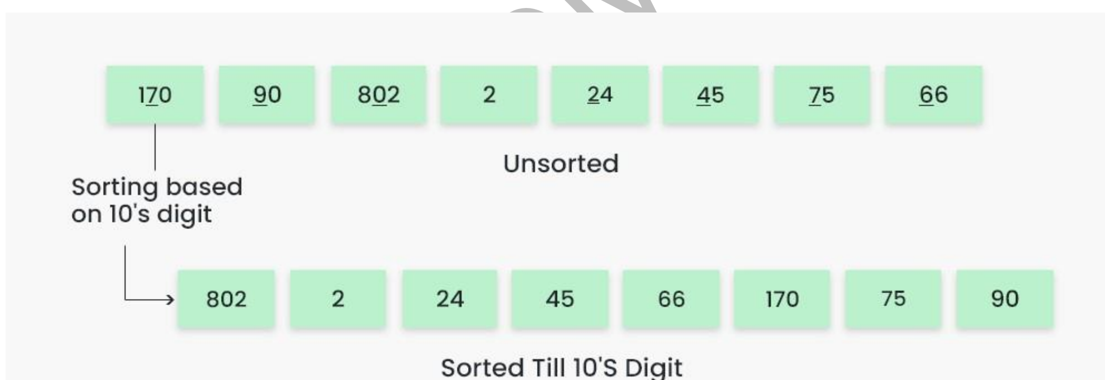
By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, it achieves the final sorted order.

We use a stable algorithm like Counting Sort to sort the individual digits so that the overall algorithm remains stable.

KAMARAJ WOMENS COLLEGE








```
#include <iostream>
 using namespace std;
int getMax(int arr[], int n) { int maxVal = arr[0];
for (int i = 1; i < n; i++) if (arr[i] > maxVal)
maxVal = arr[i]; return maxVal;
}
void countingSort(int arr[], int n, int exp) { int output[n]; // output array
int count[10] = {0};
for (int i = 0; i < n; i++) count[(arr[i] / exp) % 10]++;
for (int i = 1; i < 10; i++) count[i] += count[i - 1];
```

```
for (int i = n - 1; i >= 0; i--) { output[count[(arr[i] / exp) % 10] - 1] = arr[i]; count[(arr[i] / exp) % 10]--
;
}
for (int i = 0; i < n; i++) arr[i] = output[i];
}
void radixSort(int arr[], int n) { int maxVal = getMax(arr, n);
for (int exp = 1; maxVal / exp > 0; exp *= 10) countingSort(arr, n, exp);
}

int main() {
int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
int n = sizeof(arr) / sizeof(arr[0]); radixSort(arr, n);
cout << "Sorted array: "; for (int i = 0; i < n; i++) cout << arr[i] << " ";
cout << endl; return 0;
}
```

Advantages of Radix Sort

- Very Fast for Integers and Strings
- Does not use comparisons, unlike Quick Sort or Merge Sort.
- Stable Sort
- Preserves the relative order of elements with equal keys.
- Good for Large Numbers with Limited Digits
- Efficient when number of digits (k) is much smaller than number of elements (n).
- Predictable Performance
- Linear time for fixed digit length (like 32-bit integers).

Disadvantages of Radix Sort

- Restricted Use
- Works only for data that can be broken into digits (integers, strings, etc.).
- Extra Space Needed
- Requires temporary arrays (unlike in-place sorts).
- Complex Implementation
- More complicated than simple sorts like Insertion or Bubble Sort.
- Dependent on Digit Count
- Efficiency decreases if the number of digits is large.

KAMARAJ WOMENS COLLEGE

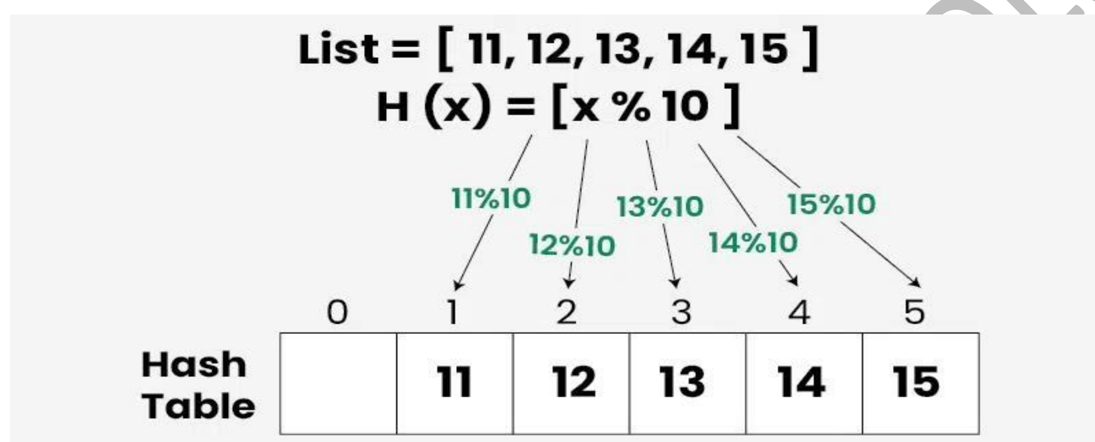
Hashing in Data Structure

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access.

Hashing involves mapping data to a specific index in a hash table (an array of items) using a hash function. It enables fast retrieval of information based on its key.

The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average.

Hashing is mainly used to implement a set of distinct items (only keys) and dictionaries (key value pairs).



Separate Chaining

Separate Chaining is a collision handling technique. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions. In this article, we will discuss about what is Separate Chain collision handling technique, its advantages, disadvantages, etc.

There are mainly two methods to handle collision:

- Separate Chaining
- Open Addressing

All those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

KAMARAJ WOMEN COLLEGE

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Open Addressing

Open Addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): Delete operation is interesting. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Rehashing

Rehashing is the process of increasing the size of a hashmap and redistributing the elements to new buckets based on their new hash values. It is done to improve the performance of the hashmap and to prevent collisions caused by a high load factor.

When a hashmap becomes full, the load factor (i.e., the ratio of the number of elements to the number of buckets) increases. As the load factor increases, the number of collisions also increases, which can lead to poor performance. To avoid this, the hashmap can be resized and the

KAMARAJ WOMENS COLLEGE

elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions.

During rehashing, all elements of the hashmap are iterated and their new bucket positions are calculated using the new hash function that corresponds to the new size of the hashmap. This process can be time-consuming but it is necessary to maintain the efficiency of the hashmap.

Rehashing is needed in a hashmap to prevent collision and to maintain the efficiency of the data structure.

As elements are inserted into a hashmap, the load factor (i.e., the ratio of the number of elements to the number of buckets) increases. If the load factor exceeds a certain threshold (often set to 0.75), the hashmap becomes inefficient as the number of collisions increases. To avoid this, the hashmap can be resized and the elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions. This process is known as rehashing.

Rehashing can be costly in terms of time and space, but it is necessary to maintain the efficiency of the hashmap.

KAMARAJ WOMENS COLLEGE