CBCS SCHEME

USN

MMC204

Second Semester MCA Degree Examination, June/July 2025
Software Engineering

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Explain the need and importance of software engineering in modern systems development.	10	L2	C1
	b.	Discuss the various attributes of software quality. Why are they essential for successful software delivery?	10	L2	C1
		OR			
Q.2	a.	Explain specialized process models such as Component-Based Development and Concurrent Model.	10	L2	C2
	b.	Describe the advantages and challenges of adopting agile methodology.	5	L2	C2
	c.	Explain the differences between Scrum and Extreme Programming.	5	L2	C1
		Module – 2			
Q.3	a.	Explain the need for requirement analysis and specification in software development.	10	L3	C1
	b.	Describe the process of requirements gathering and analysis.	10	L3	C2
		OR			
Q.4	a.	Define formal system specification and discuss its advantages	5	L2	C1
	b.	How are FSMs used in software modeling and system design?	5	L2	C2
	c.	What are CASE tools? Describe their types and applications.	10	L2	C1
		Module – 3			
Q.5	a.	Explain the importance of software design in the software development life cycle. Describe the various activities involved in the design process.	10	L3	C2
	b.	Describe the Model-View-Controller (MVC) design pattern. How does MVC promote separation of concerns?	10	L3	C3
		OR			
Q.6	a.	Describe the Client-Server architecture. Discuss how it is different from a Tiered architecture.	10	L3	C4
	b.	Discuss the challenges in designing user interfaces. How can UI design impact user experience?	10	L3	C4

		Module – 4			
Q.7	a.	Describe various black box testing techniques. How are equivalence partitioning and boundary value analysis used in black box testing?	10	L3	C3
	b.	Explain the difference between integration testing and system testing with real-world examples.	10	L2	C2
		OR			
Q.8	a.	Describe the challenges of regression testing in large software projects. How can automated tools help?	10	L3	C2
	b.	Discuss common debugging techniques and tools used by software developers.	10	L3	C3
		Module – 5			
Q.9	a.	Explain the importance of Software Project Management in the development life cycle of a software product. Discuss the key responsibilities of a project manager.	10	L2	C2
	b.	Discuss the Critical Path Method (CPM) and its significance in determining the project duration and key tasks.	10	L3	C3
		OR			
Q.10	a.	Discuss the role of the cloud as a platform in implementing DevOps. How does it enhance scalability, availability, and automation?	10	L3	C3
	b.	Present a case study that demonstrates the challenges faced in project scheduling and how those challenges were addressed using software project management principles.	10	L3	C3

* * * * *

VTU Second Semester Examination – June/July 2025

Data Structures and Algorithms							Sub Code:	MMC203
01/09/2025	Duration:	3 hrs	Max Marks:	100	Sem:	II	Branch:	MCA

MODULE- 1

Q1.a) Explain data structures and its classification with neat diagrams

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally classified into

Primitive data Structures

Non-primitive data Structures

Primitive data Structures: Primitive data structures are the fundamental data types which are supported by a programming language. Basic data types such as integer, real, character and Boolean are known as Primitive data Structures. These data types consists of characters that cannot be divided and hence they also called simple data types.

Non- Primitive data Structures: Non-primitive data structures are those data structures which are created using primitive data structures. Examples of non-primitive data structures is the processing of complex numbers, linked lists, stacks, trees, and graphs Based on the structure and arrangement of data, non-primitive data structures is further classified into

1. Linear Data Structure

2. Non-linear Data Structure

1. Linear Data Structure:

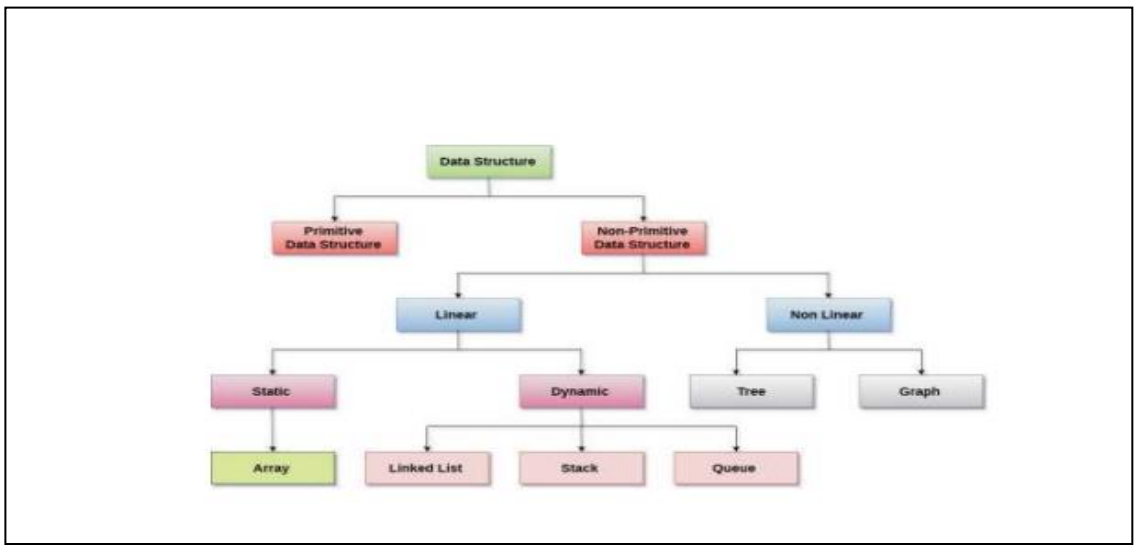
A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.

1. One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.

2. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The common examples of linear data structure are Arrays, Queues, Stacks, Linked lists

2. Non-linear Data Structure:
A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear data structure.



Arrays:
The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2, 3 . . . . . *n*. if **A** is chosen the name for the array, then the elements of **A** are denoted by subscript notation a1, a2, a3….. an
by the bracket notation A [1], A [2], A [3] . . . . . A [n]

Trees
Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or a tree. Some of the basic properties of tree are explained by means of examples

1. Stack: A stack, also called a fast-in first-out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the top. This structure is similar in its operation to a stack of dishes on a spring system as shown in fig.

Note that new 4 dishes are inserted only at the top of the stack and dishes can be deleted only from the top of the Stack

Queue: A queue, also called a first-in first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "from'' of the list, and insertions can take place only at the other end of the list, the “rear” of the list.

This structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. the first person in line is the first person to board the bus. Another analogy is with automobiles waiting to pass through an intersection the first car in line is the first car through.

Graph: Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. For example, suppose an airline flies only between the cities connected by lines in Fig. The data structure which reflects this type of relationship is called a graph.

DATA STRUCTURES OPERATIONS

The data appearing in data structures are processed by means of certain operations. The following four operations play a major role in this text:

1. **Traversing:** accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “**visiting**” the record.)

2. **Searching:** Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.

3. **Inserting:** Adding a new node/record to the structure.

4. **Deleting:** Removing a node/record from the structure.

The following two operations, which are used in special situations:

1. **Sorting:** Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)

Merging: Combining the records in two different sorted files into a single sorted file.

Q1.b) Write functions in c program to demonstrate the following operations on a singly linked list a) Insert an element at front b) Insert a node at end c) Display all the elements.

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

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

// Head pointer to the list
struct Node* head = NULL;

// Function to insert node at the beginning
void insertAtBeginning(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;
    printf("Node inserted at beginning.\n");
}

// Function to insert node at the end
void insertAtEnd(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    printf("Node inserted at end.\n");
}


// Function to display the linked list
void displayList() {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("Linked List: ");
```

```
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function with menu
int main() {
    int choice, value, position;
    while (1) {
        printf("\n--- Menu ---\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Delete at position\n");
        printf("4. Display list\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
       scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert at beginning: ");
                scanf("%d", &value);
                insertAtBeginning(value);
                break;
            case 2:
                printf("Enter value to insert at end: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;
                break;
            case 3:
                displayList();
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
}
```

Sample Output:

```
--- Menu ---
1. Insert at beginning
2. Insert at end
3. Display list
4. Exit
Enter your choice: 1
```

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

--- Menu ---
1. Insert at beginning
2. Insert at end
3. Display list
4. Exit
Enter your choice: 2
Enter value to insert at end: 20
Node inserted at end.

--- Menu ---
1. Insert at beginning
2. Insert at end
3. Display list
4. Exit
Enter your choice: 3
Linked List: 10 -> 20 -> NULL

--- Menu ---
1. Insert at beginning
2. Insert at end
3. Display list
4. Exit
Enter your choice: 4

=== Code Execution Successful ===

Q1 C) . Show that: If $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$ then $t1(n) + t2(n) \in O(\max(g1(n), g2(n)))$.

Proof:

1. By definition of Big-O:
   - Since $t1(n) \in O(g1(n))$, there exist constants $c1 > 0$ and $n1$ such that $t1(n) \leq c1\ g1(n), \quad \forall\ n \geq n1$.
   - Similarly, since $t2(n) \in O(g2(n))$, there exist constants $c2 > 0$ and $n2$ such that $t2(n) \leq c2\ g2(n), \quad \forall\ n \geq n2$.

2. For $n \geq \max(n1, n2)$, we can combine:
   $t1(n) + t2(n) \leq c1\ g1(n) + c2\ g2(n)$.

3. Since $g1(n) \leq \max(g1(n), g2(n))$ and $g2(n) \leq \max(g1(n), g2(n))$, we get:
   $t1(n) + t2(n) \leq (c1 + c2)\ \max(g1(n), g2(n))$.

4. Hence, by the definition of Big-O:
   $t1(n) + t2(n) \in O(\max(g1(n), g2(n)))$.

Q2.a) Explain Asymptotic notations with examples

Asymptotic notations are mathematical tools used in **algorithm analysis** to describe the **time complexity** (or space complexity) of an algorithm in terms of input size **n**.They tell us how an algorithm **scales** as input size grows, ignoring machine-dependent constants and focusing on growth rate.

1. Big-O Notation (O) – Upper Bound

Describes the **worst-case** time complexity.
It tells the maximum time an algorithm can take for input size **n**.

Example:

- Linear Search in an array of size **n**:
  - In the worst case (element at the end or not present), it will take **n** comparisons.
  - Time Complexity = **O(n)**.

More examples:

- Binary Search → **O(log n)**
- Bubble Sort (worst case) → **$O(n^2)$**

2. Omega Notation (Ω) – Lower Bound

Describes the **best-case** time complexity.
It shows the minimum time required by an algorithm.

Example:

- Linear Search:
  - If the element is found at the **first position**, only **1** comparison is needed.
  - Best Case = **Ω(1)**.

More examples:

- Binary Search (best case) → **Ω(1)** (when middle element is the target)
- Bubble Sort (best case, already sorted) → **Ω(n)**

3. Theta Notation (Θ) – Tight Bound

Describes the **average/expected case**, or when the algorithm has the same order in best and worst case.It gives an **exact bound**.

Example:

- Linear Search:
  - On average, the element will be found halfway through the list.
  - Average comparisons = **n/2** = **Θ(n)**.

More examples:

- Binary Search → **Θ(log n)**
- Merge Sort → **Θ(n log n)**

Q2. b) Write C functions to insert and delete an array.

```
int main() {
   int arr[100] = {10, 20, 30, 40, 50};
   int n = 5;

   printf("Original Array: ");
   for (int i = 0; i < n; i++) printf("%d ", arr[i]);
   printf("\n");

   // Insertion
   insert(arr, &n, 2, 99);  // Insert 99 at index 2
   printf("After Insertion: ");
   for (int i = 0; i < n; i++) printf("%d ", arr[i]);
   printf("\n");

   // Deletion
   delete(arr, &n, 4);  // Delete element at index 4
   printf("After Deletion: ");
   for (int i = 0; i < n; i++) printf("%d ", arr[i]);
   printf("\n");

   return 0;
}
```

Output:

Original Array: 10 20 30 40 50
After Insertion: 10 20 99 30 40 50
After Deletion: 10 20 99 30 50

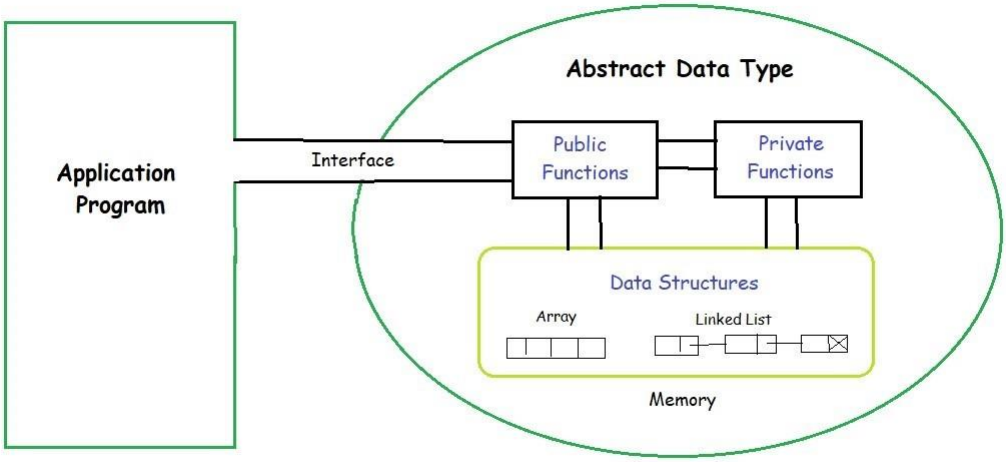
Q2. c) Explain abstract types with examples:

An Abstract Data Type (ADT) is a conceptual model that defines a set of operations and behaviors for a data structure, without specifying how these operations are implemented or how data is organized in memory. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it provides an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

Features of ADT

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation:** ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction:** ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence:** ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity:** ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.



This image demonstrates how an Abstract Data Type (ADT) hides internal data structures (like arrays, linked lists) using public and private functions, exposing only a defined interface to the application program.

Example of Abstraction

For example, we use primitive values like int, float, and char with the understanding that these data types can operate and be performed on without any knowledge of their implementation details. ADTs operate similarly by defining **what operations are possible without detailing their implementation.**

MODULE- 2

Q3.a. What is a Stack? Write functions in C to implement push and POP operations in a stack.

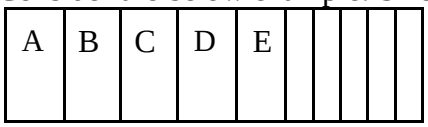
Stack is a linear data structure which follows a particular order in which the operations are performed. In stack, insertion and deletion of elements happen only at one end, i.e., the most recently inserted element is the one deleted first from the set.

Few real world examples are given as:

Stack of plates in a buffet table. The plate inserted at last will be the first one to be removed out of stack. Stack of Compact Discs

Push Operation : The push operation is used to insert an element into the stack. The element is added always at the topmost position of stack. Since the size of array is fixed at the time of declaration, before inserting the value, check if top=Size-1, if so stack is full and no more insertion is possible. In case of an attempt to insert a value in full stack, an OVERFLOW message gets displayed.

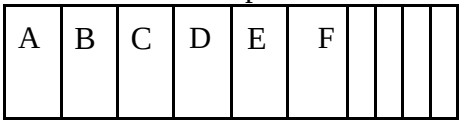
Consider the below example: Size=10



0 1 2 3 Top=4 5 6 7 8 9

Fig.3.2.3.2 Array Representation of Stack, Push(E)

To insert a new element F, first check if Top==Size-1. If the condition fails, then increment the Top and store the value. Thus the updated stack is:



0 1 2 3 4 Top=5 6 7 8 9

Fig.3.2.3.3 Array Representation of Stack, Push(F)

Algorithm:

Push(X)

if Top == Size-1
Write “Stack Overflow”

else

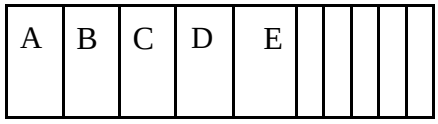
End

Pop Operation

return Top = Top + 1 // St represents an Array with maximum limit as Size St[Top] = X // X element to be inserted .

The pop operation is used to remove the topmost element from the stack. In this case, first check the presence of element, if top == -1 (indicates no array elements), then it indicates empty stack and thereby deletion not possible. In case of an attempt to delete a value in an empty stack, an UNDERFLOW message gets displayed.

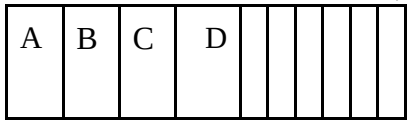
Consider the below example: Size=10



0 1 2 3 Top=4 5 6 7 8 9

Fig.3.2.3.4 Array Representation of Stack, Pop()

To delete the topmost element, first check if Top== -1. If the condition fails, then decrement the Top.Thus the updated stack is:



0 1 2 Top=3 4 5 6 7 8 9

Fig.3.2.3.5 Array Representation of Stack, Pop()

Algorithm:

Pop()

if Top == -1

Write “Stack Underflow”

return

else

X = St[Top] // St represents an Array with maximum limit as Size

Top = Top-1 // X represents element removed

return X

End

Program code:

```
#include <stdio.h>
#define SIZE 100

int stack[SIZE];
int top = -1;

void push(int value) {
   if (top == SIZE - 1)
      printf("Stack Overflow!\n");
   else {
      top++;
      stack[top] = value;
      printf("Pushed %d\n", value);
   }
}

void pop() {
   if (top == -1)
      printf("Stack Underflow!\n");
   else {
      printf("Popped %d\n", stack[top]);
      top--;
   }
}

void display() {
   if (top == -1)
      printf("Stack is empty\n");
   else {
      printf("Stack: ");
      for (int i = top; i >= 0; i--)
         printf("%d ", stack[i]);
      printf("\n");
```

```
    }
}

int main() {
    int choice, value;
    while (1) {
        printf("\n1. PUSH\n2. POP\n3. DISPLAY\n4. EXIT\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                return 0;
            default:
                printf("Invalid choice\n");
        }
    }
}
```

Sample output:

```
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to push: 10
Pushed 10

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to push: 20
Pushed 20

1. PUSH
```

2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to push: 30
Pushed 30

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 2
Popped 30

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 3
Stack: 20 10

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 4

=== Code Execution Successful ===

Q3.b. Write a program to implement tower of Hanoi using recursion and trace the output for 3 disks

Recursive case:
Move n - 1 disks from A to B using C as spare.
Move the nth disk from A to C using B as spare.
Move the n - 1 disks from B to C using A as spare.

Program code:

```
#include <stdio.h>

// Recursive function to solve Tower of Hanoi
void towerOfHanoi(int n, char source, char auxiliary, char destination) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", source, destination);
        return;
    }
```

```
    // Move n-1 disks from source to auxiliary
    towerOfHanoi(n - 1, source, destination, auxiliary);

    // Move nth disk from source to destination
    printf("Move disk %d from %c to %c\n", n, source, destination);

    // Move n-1 disks from auxiliary to destination
    towerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
    int n;
    printf("Enter number of disks: ");
    scanf("%d", &n);

    printf("\nSteps to solve Tower of Hanoi:\n");
    towerOfHanoi(n, 'A', 'B', 'C');  // A = source, B = auxiliary, C = destination

    return 0;
}
```

Sample output:

```
Enter number of disks: 3

Steps to solve Tower of Hanoi:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C


=== Code Execution Successful ===
```

Q3.c.What is a Queue? Write a C Program to implement queue of integers using arrays?

Queue is a linear data structure where elements are ordered in special fashion i.e. FIFO (First In First Out). Which means element inserted first to the queue will be removed first from the queue.
In real life you have come across various queue examples. Such as queue of persons at ticket counter, where the first person entering queue gets ticket first.

C Program: Queue of Integers using Arrays

```
#include <stdio.h>
#define SIZE 5   // maximum size of queue

int queue[SIZE];
int front = -1, rear = -1;
```

```
// Function to check if queue is empty
int isEmpty() {
    return (front == -1);
}

// Function to check if queue is full
int isFull() {
    return (rear == SIZE - 1);
}

// Function to add element to queue
void enqueue(int value) {
    if (isFull()) {
        printf("Queue Overflow! Cannot insert %d\n", value);
        return;
    }
    if (front == -1) front = 0;  // first element
    queue[++rear] = value;
    printf("%d inserted into queue\n", value);
}

// Function to remove element from queue
void dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow! Cannot remove element\n");
        return;
    }
    printf("%d removed from queue\n", queue[front]);
    front++;
    if (front > rear) {
        // Reset queue when empty
        front = rear = -1;
    }
}

// Function to display queue elements
void display() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

// Main driver function
int main() {
```

```
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();

    dequeue();
    display();

    enqueue(40);
    enqueue(50);
    enqueue(60); // Will show overflow if SIZE = 5
    display();

    dequeue();
    dequeue();
    display();

    return 0;
}
```

Sample Output:

```
10 inserted into queue
20 inserted into queue
30 inserted into queue
Queue elements: 10 20 30
10 removed from queue
Queue elements: 20 30
40 inserted into queue
50 inserted into queue
Queue Overflow! Cannot insert 60
Queue elements: 20 30 40 50
20 removed from queue
30 removed from queue
Queue elements: 40 50
```

Q4.a.What is Circular Queue? Write functions in C to implement Insert and delete operations in a circular queue?

A **Circular Queue** is an advanced version of a linear queue that **connects the rear back to the front** of the queue in a circular manner.

 It solves the problem of wasted space in a simple queue by reusing empty slots created by deletions.

- **Enqueue (Insert)** → Add element at `rear`.
- **Dequeue (Delete)** → Remove element from `front`.
- **Rear wraps around to 0** when it reaches the end of the array.

C Implementation of Insert & Delete in Circular Queue

```
#include <stdio.h>
#define SIZE 5  // maximum size of queue

int cqueue[SIZE];
int front = -1, rear = -1;

// Function to check if queue is full
int isFull() {
   return ((front == 0 && rear == SIZE - 1) || (rear + 1) % SIZE == front);
}

// Function to check if queue is empty
int isEmpty() {
   return (front == -1);
}

// Function to insert element (Enqueue)
void enqueue(int value) {
   if (isFull()) {
      printf("Queue Overflow! Cannot insert %d\n", value);
      return;
   }
   if (isEmpty()) {
      front = rear = 0;  // first element
   } else {
      rear = (rear + 1) % SIZE;  // circular increment
   }
   cqueue[rear] = value;
   printf("%d inserted into Circular Queue\n", value);
}

// Function to delete element (Dequeue)
void dequeue() {
   if (isEmpty()) {
      printf("Queue Underflow! Cannot delete\n");
      return;
   }
   printf("%d deleted from Circular Queue\n", cqueue[front]);

   if (front == rear) {
      // Only one element was present
      front = rear = -1;
   } else {
      front = (front + 1) % SIZE;  // circular increment
   }
}

// Function to display elements
void display() {
```

```
    if (isEmpty()) {
        printf("Circular Queue is empty!\n");
        return;
    }
    printf("Circular Queue elements: ");
    int i = front;
    while (1) {
        printf("%d ", cqueue[i]);
        if (i == rear) break;
        i = (i + 1) % SIZE;
    }
    printf("\n");
}

// Driver code
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    display();

    dequeue();
    dequeue();
    display();

    enqueue(50);
    enqueue(60);  // Uses free space at beginning
    display();

    enqueue(70);  // Overflow since queue is full
    return 0;
}
```

Sample Output:

```
10 inserted into Circular Queue
20 inserted into Circular Queue
30 inserted into Circular Queue
40 inserted into Circular Queue
Circular Queue elements: 10 20 30 40
10 deleted from Circular Queue
20 deleted from Circular Queue
Circular Queue elements: 30 40
50 inserted into Circular Queue
60 inserted into Circular Queue
Circular Queue elements: 30 40 50 60
Queue Overflow! Cannot insert 70
```

Q4.b.Evaluate the following postfix expression : 6 8 3 *+5 4 2 / + * by showing the contents of a stack

Postfix rules:

- **Operands** → push onto stack
- **Operators** → pop operands, perform operation, push result back

Token	Action	Stack (Top → Bottom)
6	Operand → push	6
8	Operand → push	8, 6
3	Operand → push	3, 8, 6
*****	Operator → pop 3 & 8 → 8×3=24 → push	24, 6
+	Operator → pop 24 & 6 → 6+24=30 → push	30
5	Operand → push	5, 30
4	Operand → push	4, 5, 30
2	Operand → push	2, 4, 5, 30
/	Operator → pop 2 & 4 → 4/2=2 → push	2, 5, 30
+	Operator → pop 2 & 5 → 5+2=7 → push	7, 30
*****	Operator → pop 7 & 30 → 30×7=210 → push	210

Ans : 210

Q4.c.Write the General plan for Analyzing the time efficiency of recursive algorithms

Analysis of Recursive Algorithms:-

General plan for analyzing the time efficiency of recursive algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
4. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Compute the factorial function F(n) = n! for an arbitrary nonnegative integer n. Since

n! = 1 . . . . . .(n-1) . n = (n-1)! . n for n>=1

and 0! = 1 by definition, we can compute F (n) = F(n – 1) * n with the following recursive al Algorithm.

Algorithm
F(n)
//Computes n! recursively
//input: A nonnegative integer n
//Output: the value of n!
if n = 0 return I
else return F(n— l) * n

Since the function *F(n)* is computed according to the formula

F (n) = F(n – 1) * n for n>0

Let's analyze the efficiency of this algorithm:

1. **Time Complexity:** The time complexity of this algorithm is O(n), where 'n' is the input number for which we want to find the factorial. This is because the loop runs 'n' times, and each iteration involves a constant amount of work (multiplication and assignment).
   - **Time Complexity: O(n) – The number of basic operations grows linearly with the input 'n'.**
2. **Space Complexity:** The space complexity of this algorithm is O(1), which means it uses a constant amount of additional memory regardless of the input value 'n'. It only requires a few extra variables (e.g., 'factorial' and 'i').
   - **Space Complexity: O(1) – It uses a constant amount of memory regardless of 'n'.**

MODULE- 3

Q5.a.What is binary tree? Write a note on array and linked list representation of a binary tree.

A **Binary Tree** is a hierarchical data structure in which each node has **at most two children**:

- **Left child**
- **Right child**

The **topmost node** is called the **root**, and nodes with no children are called **leaves**.

 It is widely used in searching (Binary Search Tree), sorting (Heaps), and expression evaluation.

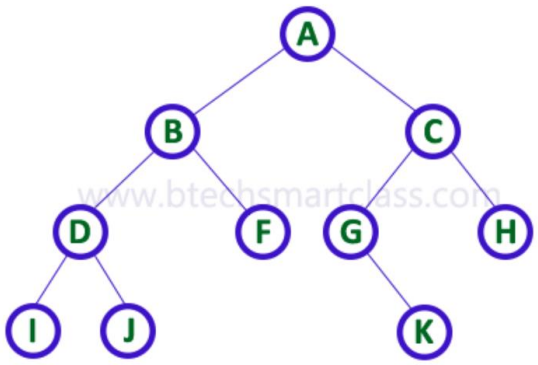
Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**

2. Linked List Representation

Consider the following binary tree...



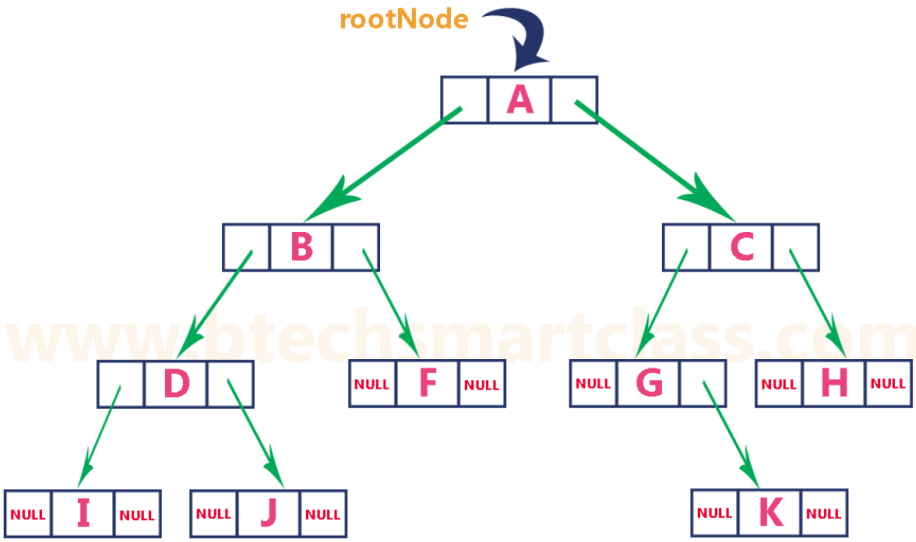
1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree. Consider the above example of a binary tree and it is represented as follows...

2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...



Q5.b. Construct a binary search tree for the following numbers and traverse in preorder, In order and post order 14,15,4,9.7.18,3,5,16,20,17

Step 1: Insert the given numbers

Sequence: **14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17**

Insert 14 → root

```
14
```

Insert 15 → greater than 14 → right child

```
 14
   \
    15
```

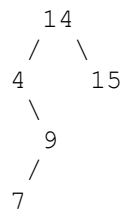
Insert 4 → less than 14 → left child

```
    14
   /  \
  4    15
```

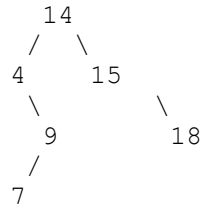
Insert 9 → less than 14 → go left → greater than 4 → right child of 4

```
    14
   /  \
  4    15
   \
```

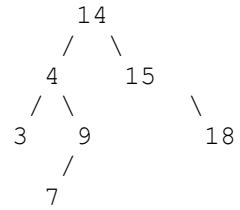
Insert 7 → less than 14 → go left → greater than 4 → go right to 9 → less than 9 → left child



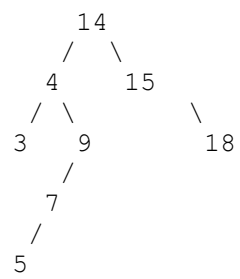
Insert 18 → greater than 14 → go right → greater than 15 → right child



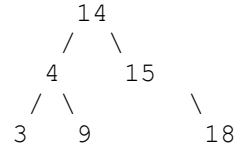
Insert 3 → less than 14 → go left → less than 4 → left child

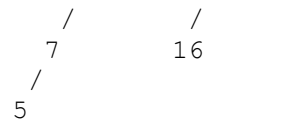


Insert 5 → less than 14 → go left to 4 → greater than 4 → go right to 9 → less than 9 → go left to 7 → less than 7 → left child

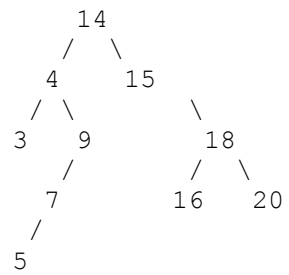


Insert 16 → greater than 14 → go right to 15 → greater than 15 → go right to 18 → less than 18 → left child

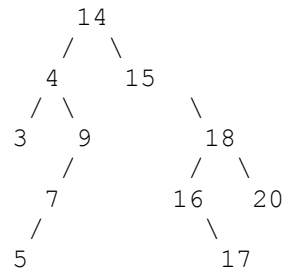




Insert 20 → greater than 14 → go right to 15 → right to 18 → greater than 18 → right child



Insert 17 → greater than 14 → go right to 15 → right to 18 → left to 16 → greater than 16 → right child



Step 2: Traversals

1. Preorder (Root → Left → Right)

```
14, 4, 3, 9, 7, 5, 15, 18, 16, 17, 20
```

2. Inorder (Left → Root → Right)

Inorder of BST gives sorted order:

```
3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20
```

3. Postorder (Left → Right → Root)

```
3, 5, 7, 9, 4, 17, 16, 20, 18, 15, 14
```

Final Answer:

- **Preorder:** 14, 4, 3, 9, 7, 5, 15, 18, 16, 17, 20
- **Inorder:** 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20
- **Postorder:** 3, 5, 7, 9, 4, 17, 16, 20, 18, 15, 14

Q5.c. What is a graph? Give adjacency list and adjacency matrix reprsentation of the graph in Fig Q5. C

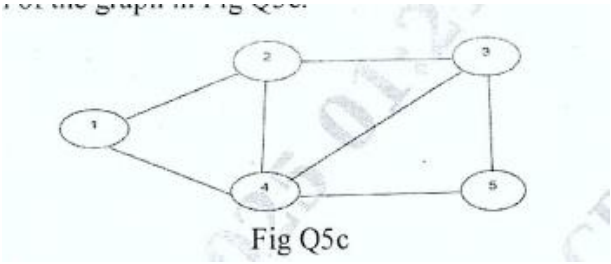


Fig Q5c

A **graph** is a non-linear data structure consisting of:

- **Vertices (nodes)** → represent entities.
- **Edges** → connections between vertices.

Graphs can be **directed/undirected** and **weighted/unweighted**.

The given graph (Fig Q5c) is **undirected and unweighted**.

Vertices

The graph has **5 vertices**: {1, 2, 3, 4, 5}.

Edges (from the figure)

- (1,2), (1,4)
- (2,3), (2,4)
- (3,4), (3,5)
- (4,5)

1. Adjacency List Representation

Each vertex stores a list of all vertices it is connected to.

```
1 → 2, 4
2 → 1, 3, 4
3 → 2, 4, 5
4 → 1, 2, 3, 5
5 → 3, 4
```

2. Adjacency Matrix Representation

For **5 vertices**, we use a 5×5 matrix.

- If an edge exists between vertex *i* and *j*, then `matrix[i][j] = 1`, else `0`.
- Since the graph is undirected → matrix is **symmetric**.

	1	**2**	**3**	**4**	**5**
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

So,

- **Adjacency List** gives memory-efficient representation.
- **Adjacency Matrix** gives faster edge lookup (O(1)).

Q6.a. What is an AVL Tree? Explain the different rotations of an AVL tree with example

An AVL Tree (Adelson-Velsky and Landis, 1962) is a self-balancing Binary Search Tree (BST). It ensures that the heights of the left and right subtrees of any node differ by at most 1. This guarantees O(log n) performance for search, insert, and delete operations.

Balance Factor

(BF) BF(node) = Height(left subtree) - Height(right subtree)

• BF = -1, 0, +1 → Balanced

• BF < -1 or BF > +1 → Unbalanced, requires rotation

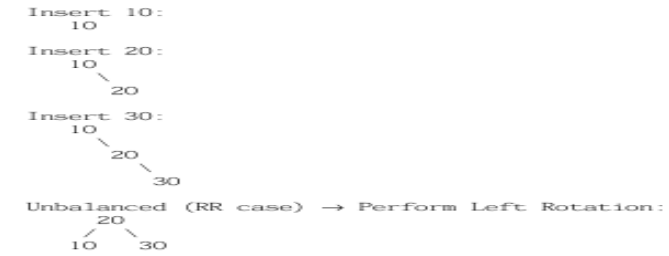
Rotations in AVL Tree

1. LL Rotation (Right Rotation)
2. RR Rotation (Left Rotation)
3. LR Rotation (Left-Right Rotation)
4. RL Rotation (Right-Left Rotation)

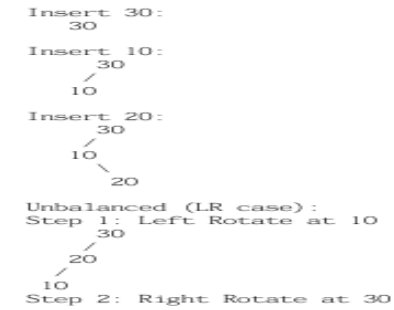
Example 1: Insertion (10, 20, 30)

Example 2: LR Case (30, 10, 20)

Example 1: Insertion (10, 20, 30)



Example 2: LR Case (30, 10, 20)



Q6.b. Apply Dijikstra's algorithm to find single source shortest path assuming 1 as the source vertex

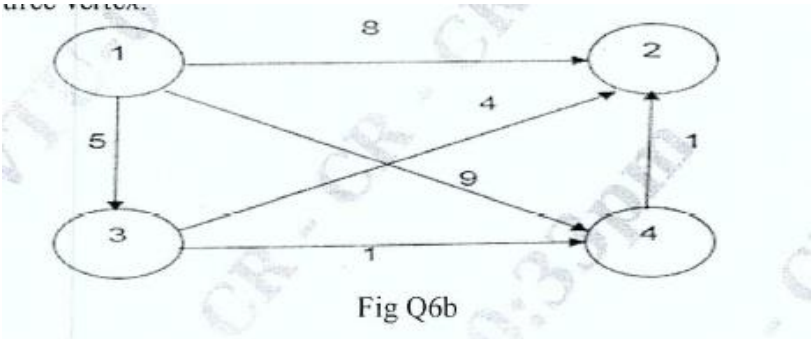


Fig Q6b

Graph edges (from figure Q6b):

- 1→2 = 8
- 1→3 = 5
- 1→4 = 9
- 3→2 = 4
- 3→4 = 1
- 4→2 = 1

Source = **1**

Step-wise Dijkstra

Step 1:
Source set S={1}S = \{1\}S={1}, Pending set P={2,3,4}P=\{2,3,4\}P={2,3,4}

Initial distances:

- d(1→2) = 8
- d(1→3) = 5
- d(1→4) = 9

Minimum = 5 → choose **vertex 3**

Step 2:
Now S={1,3},P={2,4}S=\{1,3\}, P=\{2,4\}S={1,3},P={2,4}

Relax edges from 3:

- To 2: via 3 → 5+4=9 (but current d(2)=8, keep 8)
- To 4: via 3 → 5+1=6 (better than 9 → update d(4)=6)
-

Distances:

- d(2)=8, d(4)=6

Minimum = 6 → choose **vertex 4**

Step 3:
Now S={1,3,4},P={2}S=\{1,3,4\}, P=\{2\}S={1,3,4},P={2}

Relax edges from 4:

- To 2: via 4 → 6+1=7 (better than 8 → update d(2)=7)

Distances:

- d(2)=7

So pick **vertex 2**

Step 4:
Now S={1,2,3,4}S=\{1,2,3,4\}S={1,2,3,4}, all vertices chosen. **Stop**

Final shortest distances from source 1

- d(1)=0
- d(2)=7
- d(3)=5
- d(4)=6

Shortest paths

- 1→3 (cost 5)
- 1→3→4 (cost 6)
- 1→3→4→2 (cost 7)

Q6. c. Explain BFS and DFS traversal of a graph. List the differences between them.

DFS (Depth-First Search)
Depth-First Search (DFS) is a popular algorithm used for traversing or searching through graph or tree data structures. It starts at a source node (or vertex) and explores as far along each branch or path as possible before backtracking.

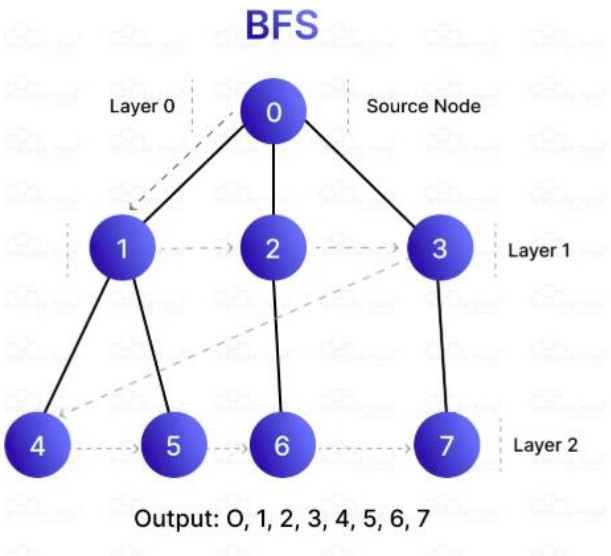
This means it dives deep into the graph before moving to other branches, hence the name "depth-first." DFS can be implemented using either recursion or a stack.

BFS (Breadth-First Search)
BFS (Breadth-First Search) is a graph traversal algorithm that explores nodes level by level, starting from a given source node. It visits all neighboring nodes at the current depth (or level) before moving on to nodes at the next depth level. BFS is commonly used for traversing or searching tree or graph data structures.

Difference Between DFS and BFS with Example
We will take an example to understand the difference between DFS and BFS:

1. BFS
Approach: BFS explores the graph level by level, starting from the source node. It visits all the nodes at one level before moving to the next. BFS uses a queue to keep track of nodes that need to be explored.

How it works: From the source node, BFS moves to all its direct neighbors (Layer 1), then moves to the next layer (Layer 2), continuing until all nodes are visited.



Traversal Order:

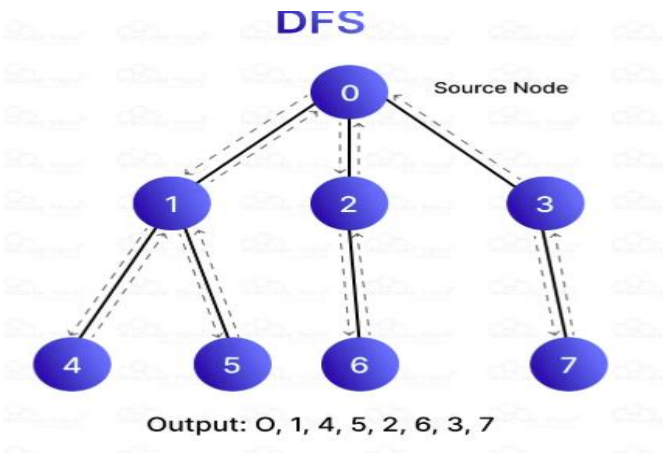
- Starting from node 0:
- Layer 0: Visit node 0 (source node).
- Layer 1: Visit nodes 1, 2, 3 (neighbors of node 0).
- Layer 2: Visit nodes 4, 5, 6, 7 (neighbors of nodes 1, 2, 3).

- Output: 0, 1, 2, 3, 4, 5, 6, 7.

2. DFS

Approach: DFS explores as deep as possible along a branch before backtracking. It uses a stack (or recursion) to keep track of nodes and backtracks when it reaches a dead end.

How it works: Starting from the source node, DFS goes deeper into the graph, visiting one branch of the graph first before backtracking to explore others.



Traversal Order:

- Starting from node 0:
- Visit node 0 → Visit node 1 → Visit node 4 (deepest) → Backtrack to node 1 → Visit node 5 → Backtrack to node 0 → Visit node 2 → Visit node 6 → Backtrack to node 2 → Visit node 3 → Visit node 7.
- **Output:** 0, 1, 4, 5, 2, 6, 3, 7

Difference between DFS and BFS

Find all the differences between DFS and BFS algorithms:

Criteria	**BFS (Breadth-First Search)**	**DFS (Depth-First Search)**
Traversal Method	Explores all neighbors at the current level before moving to the next level.	Explores as deep as possible along one branch before backtracking.
Data Structure Used	Queue (FIFO)	Stack (LIFO) or recursion stack.

Time Complexity	O(V + E) (V = vertices, E = edges)	O(V + E) (V = vertices, E = edges)
Space Complexity	O(V) (due to queue storing nodes at the current level)	O(V) (due to recursion or explicit stack)
Best for	Finding the shortest path in an unweighted graph.	Exploring all possible paths or solving puzzles like mazes.
Use Case	- Shortest path in unweighted graphs (like social networks, web crawlers). - Level-order traversal of trees.	- Solving maze or puzzle problems. - Detecting cycles in graphs, topological sorting.
Performance on Deep Graphs	May require more memory for wide/deep graphs, as the queue holds many nodes.	Efficient for deep graphs, as it traverses a single path at a time.
Performance on Wide Graphs	Efficient for shallow/wide graphs, as it processes level by level.	May require more memory for wide graphs, as the recursion stack can grow large.
Shortest Path	Guaranteed to find the shortest path in an unweighted graph.	Not guaranteed to find the shortest path, but explores all paths.
Path finding in Weighted Graphs	Not suitable for weighted graphs.	Not suitable for weighted graphs (Dijkstra's or Bellman-Ford should be used).
Cycle Detection	Can detect cycles, but less commonly used for it.	Useful for detecting cycles in directed or undirected graphs.
Recursive/Iterative	Iterative (uses a queue)	Can be recursive (via recursion stack) or iterative (using an explicit stack).
Graph Representation	Can be used with both adjacency lists and matrices.	Can be used with both adjacency lists and matrices.

Tree Traversal	Used for level-order traversal in trees.	Used for pre-order, in-order, and post-order traversal in trees.
Backtracking	No backtracking.	Uses backtracking to explore new branches.
Algorithm Type	Greedy approach (explores level by level).	Backtracking approach (explores depth first).
Application in AI	Used in algorithms like breadth-first search for game AI (level exploration).	Used in algorithms like depth-first search for solving constraint satisfaction problems.
Example of Use	Finding the shortest path from a source to all nodes in an unweighted graph.	Exploring all possible paths to solve a maze

MODULE- 4

Q7. a. Write a C Program to implement bubble sort. Obtain its time complexity.

. The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item ―bubbles‖ up to the location where it belongs. The shaded items are being compared to see if they are out of order. If there are n items in the list, then there are n - 1n - 1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. It is a stable sorting algorithm with a time complexity of $O(n^2)$ in the average and worst cases – and O(n) in the best case.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1**5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1**4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.
Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1**2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

C program for bubble sort.

```
#include<stdio.h>
#include<stdlib.h>
void display(int a[],int n);
void bubble_sort(int a[],int n);
int main()
{
	int n,choice,i;
	char ch[20];
	printf("Enter no. of elements u want to sort : ");
	scanf("%d",&n);
	int arr[n];
	for(i=0;i<n;i++)
	{
		printf("Enter %d Element : ",i+1);
		scanf("%d",&arr[i]);
	}
	printf("Please select any option Given Below for Sorting : \n");
	while(1)
	{
		printf("\n1. Bubble Sort\n 2. Display Array.\n3. Exit the Program.\n");
		printf("\nEnter your Choice : ");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1:bubble_sort(arr,n);
				break;
			case 2:display(arr,n);
```

```
                        break;
                case 3:
                        return 0;
                default:
                        printf("\nPlease Select only 1-3 option ----\n");
                }
        }
        return 0;
}
//-----------End of main function----------------------

//-------------------Display Function-----------------
void display(int arr[],int n)
{
        for(int i=0;i<n;i++)
        {
                printf(" %d ",arr[i]);
        }
}
//---------------------Bubble Sort Function-----------
void bubble_sort(int arr[],int n)
{
        int i,j,temp;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(arr[j]>arr[j+1])
                        {
                                temp=arr[j];
                                arr[j]=arr[j+1];
                                arr[j+1]=temp;
                        }
                }
        }
        printf("After Bubble sort Elements are : ");
        display(arr,n);
}
```

Sample Output:

Enter no. of elements u want to sort : 5
Enter 1 Element : 34
Enter 2 Element : 78
Enter 3 Element : 54
Enter 4 Element : 98
Enter 5 Element : 90

Please select any option Given Below for Sorting :
1. Bubble Sort 2. Display Array.3. Exit the Program.
Enter your Choice : 1
After Bubble sort Elements are : 34 54 78 90 98
1. Bubble Sort
2. Display Array.
3. Exit the Program.

Q7. b. Implement the hash function h(k)=k%11 on the numbers 25,46,10,36,18,29 and 43.Show the hash table

Step 1: Compute Hash Values

- h(25)=25%11=3h(25) = 25 \% 11 = 3h(25)=25%11=3
- h(46)=46%11=2h(46) = 46 \% 11 = 2h(46)=46%11=2
- h(10)=10%11=10h(10) = 10 \% 11 = 10h(10)=10%11=10
- h(36)=36%11=3h(36) = 36 \% 11 = 3h(36)=36%11=3 → collision with 25
- h(18)=18%11=7h(18) = 18 \% 11 = 7h(18)=18%11=7
- h(29)=29%11=7h(29) = 29 \% 11 = 7h(29)=29%11=7 → collision with 18
- h(43)=43%11=10h(43) = 43 \% 11 = 10h(43)=43%11=10 → collision with 10

Step 2: Construct Hash Table (Separate Chaining Method)

Index	Elements (after hashing)
0	–
1	–
2	46
3	25 → 36
4	–
5	–
6	–
7	18 → 29
8	–
9	–
10	10 → 43

Final Hash Table Representation

- At index **3**, collision handled by chaining: 25 → 36
- At index **7**, collision handled by chaining: 18 → 29

- At index **10**, collision handled by chaining: 10 → 43

Q7.c. Write an algorithm for insertion sort. Sort the following numbers using insertion sort.35,10, 15,45,25,20 and 40.Obtain its time complexity.

Algorithm for Insertion Sort

Insertion Sort (Ascending Order):

```
Algorithm InsertionSort(A, n)
// A is the array, n is number of elements
1. for i ← 1 to n-1 do
2.     key ← A[i]
3.     j ← i - 1
4.     while j ≥ 0 and A[j] > key do
5.         A[j+1] ← A[j]
6.         j ← j - 1
7.     end while
8.     A[j+1] ← key
9. end for
```

Sorting the given numbers: 35, 10, 15, 45, 25, 20, 40

Initial Array: **[35, 10, 15, 45, 25, 20, 40]**

Pass 1: (Insert 10 in sorted {35})

- Compare 10 < 35 → shift 35
- Array: **[10, 35, 15, 45, 25, 20, 40]**

Pass 2: (Insert 15 in sorted {10, 35})

- Compare 15 < 35 → shift 35
- Compare 15 > 10 → insert
- Array: **[10, 15, 35, 45, 25, 20, 40]**

Pass 3: (Insert 45 in sorted {10, 15, 35})

- 45 > 35 → place as is
- Array: **[10, 15, 35, 45, 25, 20, 40]**

Pass 4: (Insert 25 in sorted {10, 15, 35, 45})

- Compare 25 < 45 → shift 45
- Compare 25 < 35 → shift 35

- Compare 25 > 15 → insert
- Array: **[10, 15, 25, 35, 45, 20, 40]**

Pass 5: (Insert 20 in sorted {10, 15, 25, 35, 45})

- Compare 20 < 45 → shift 45
- Compare 20 < 35 → shift 35
- Compare 20 < 25 → shift 25
- Compare 20 > 15 → insert
- Array: **[10, 15, 20, 25, 35, 45, 40]**

Pass 6: (Insert 40 in sorted {10, 15, 20, 25, 35, 45})

- Compare 40 < 45 → shift 45
- Compare 40 > 35 → insert
- Array: **[10, 15, 20, 25, 35, 40, 45]**

Final Sorted Array: [10, 15, 20, 25, 35, 40, 45]

Time Complexity of Insertion Sort

- **Best Case (Already Sorted):** O(n) → only comparisons, no shifts
- **Worst Case (Reverse Sorted):** $O(n^2)$ → maximum comparisons + shifts
- **Average Case:** $O(n^2)$
- **Space Complexity:** O(1) (in-place sorting)

Q8.a. Sort the following numbers using radix sort and show the table of various passes of radix sort.338,249,112,589,699,478,728,246,532

Pass	Sorted Sequence
Initial	338, 249, 112, 589, 699, 478, 728, 246, 532
Pass 1 (Units)	112, 532, 246, 338, 478, 728, 249, 589, 699
Pass 2 (Tens)	112, 532, 338, 246, 249, 478, 728, 589, 699
Pass 3 (Hundreds)	**112, 246, 249, 338, 478, 532, 589, 699, 728**

Final Sorted Order:
112, 246, 249, 338, 478, 532, 589, 699, 728

Q8.b. Write a C Program to implement linear search. Obtain Best case, Worst case and Average case efficiency

```
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {

    for (int i = 0; i < n; i++) {
```

```
        if (arr[i] == key) {
            return i;   // return index if found
        }
    }
    return -1;  // return -1 if not found
}

int main() {
    int arr[] = {10, 25, 35, 40, 50, 60};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key;

    printf("Enter the element to search: ");
    scanf("%d", &key);

    int result = linearSearch(arr, n, key);

    if (result == -1)
        printf("Element %d not found in array.\n", key);
    else
        printf("Element %d found at position %d.\n", key, result + 1);

    return 0;
```

}

Case	Comparisons	Time Complexity
Best Case	1	O(1)
Worst Case	n	O(n)
Average Case	n/2	O(n)

Q8.c. What is hash collision? Explain linear probing and separate chaining methods

The situation in which the hash function returns the same hash key (home bucket) for more than one record is called collision and two same hash keys returned for different records is called synonym.

COLLISION RESOLUTION TECHNIQUES

If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique.

1.Chaining
2.Open addressing (linear probing)
3.Quadratic probing
4.Double hashing
6. Rehashing

OPEN ADDRESSING – LINEAR PROBING

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

For example:

Consider that following keys are to be inserted in the hash table

131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table.
We will use Division hash function. That means the keys are placed using the formula

H(key) = key % tablesize
H(key) = key % 10

For instance the element 131 can be placed at

H(key) = 131 % 10
= 1

Index 1 will be the home bucket for 131. Continuing in this fashion we will place 4, 8, 7.

Now the next key to be inserted is 21. According to the hash function

H(key)=21%10
H(key) = 1

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element. Therefore 21 will be placed at the index 2. If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5.

Index	Key	Key	Key
0	NULL	NULL	NULL
1	131	131	131
2	NULL	21	21
3	NULL	NULL	31
4	4	4	4
5	NULL	5	5
6	NULL	NULL	61
7	7	7	7
8	8	8	8
9	NULL	NULL	NULL

after placing keys 31, 61

The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0^{th} index.

Problem with linear probing:

One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

19%10 = 9
18%10 = 8
39%10 = 9
29%10 = 9
8%10 = 8

cluster is formed

rest of the table is empty

Key
39
29
8
18
19

this cluster problem can be solved by quadratic probing.

QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H(key) = (Hash(key) + i^2) \% m)$$

where m can be table size or any prime number.

for eg; If we have to insert following elements in the hash table with table size 10:

37, 90, 55, 22, 17, 49, 87

37 % 10 = 7
90 % 10 = 0
55 % 10 = 5
22 % 10 = 2
11 % 10 = 1

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

Now if we want to place 17 a collision will occur as 17%10 = 7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

H_i (key) = (Hash(key) + i^2) % m

Consider i = 0 then
(17 + 0^2) % 10 = 7

(17 + 1^2) % 10 = 8, when i =1

The bucket 8 is empty hence we will place the element at index 8. Then comes 49 which will be placed at index 9.

49 % 10 = 9

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	49
9	

Now to place 87 we will use quadratic probing.

(87 + 0) % 10 = 7
(87 + 1) % 10 = 8… but already occupied
(87 + 2^2) % 10 = 1.. already occupied
(87 + 3^2) % 10 = 6

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	

DOUBLE HASHING

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert.
There are two important rules to be followed for the second function:

- it must never evaluate to zero.
- must make sure that all cells can be probed.

The formula to be used for double hashing is

CHAINING

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

For eg;

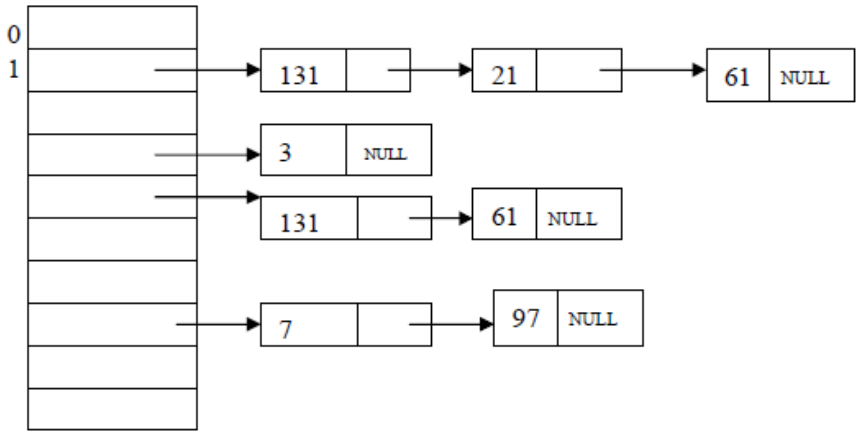
Consider the keys to be placed in their home buckets are

131, 3, 4, 21, 61, 7, 97, 8, 9

then we will apply a hash function as H(key) = key % D

Where D is the size of table. The hash table will be

Here D = 10



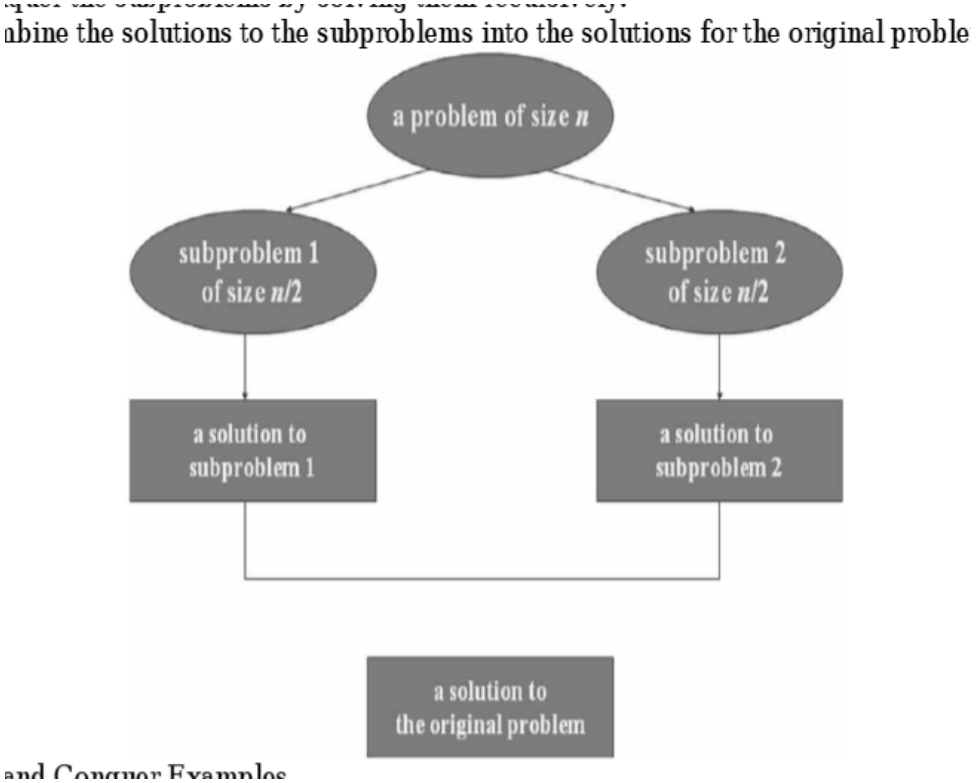
A chain is maintained for colliding elements. for instance 131 has a home bucket (key) 1. similarly key 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

MODULE- 5

Q9.a. Write short notes on a) Greedy techniques b) Divide and conquer

The most well known algorithm design strategy is Divide and Conquer Method. It

- Divide the problem into two or more smaller subproblems.
- Conquer the subproblems by solving them recursively.
- Combine the solutions to the subproblems into the solutions for the original problem.



Divide and Conquer Examples

- Sorting: mergesort and quicksort
- Tree traversals
- Binary search
- Matrix multiplication-Strassen‘s algorithm

Greedy Technique

- **Definition:**
  The Greedy method builds a solution step by step, always choosing the option that looks **best at the current moment** (locally optimal choice) with the hope that this leads to a **global optimum**.
- **Characteristics:**
  - Makes decisions in sequence.
  - No backtracking is used.
  - Works well when the problem has the **greedy-choice property** and **optimal substructure**.
- **Applications:**

1. **Activity Selection Problem** – choose maximum non-overlapping activities.
2. **Huffman Coding** – optimal prefix-free codes.
3. **Minimum Spanning Tree** – Kruskal's and Prim's algorithms.
4. **Dijkstra's Algorithm** – single-source shortest path.

- **Time Complexity:** Depends on implementation, often **O(n log n)** due to sorting.

Q9.b. What is a heap? Explain the Construction of max heap by taking the following numbers :

13,86,43,38,54,23,63 using bottom-up approach

Heap Definition

- A **Heap** is a special complete binary tree stored as an array.
- Two types:
  - **Max Heap** → Each parent ≥ its children.
  - **Min Heap** → Each parent ≤ its children.
- For **n elements**, if stored in array `A[1..n]`:
  - Parent of node `i` = `i/2`
  - Left child = `2i`
  - Right child = `2i+1`

Construction of Max Heap (Bottom-Up Approach)

Array given:

```
13, 86, 43, 38, 54, 23, 63
```

Step 1: Represent as a Complete Binary Tree

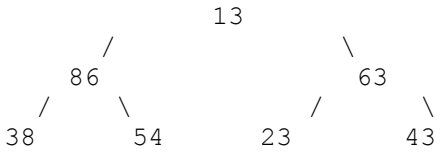
```
13
```

Step 2: Apply Heapify (Bottom-Up)

We start from the last **non-leaf node** = index ⌊n/2⌋ = 7/2 = 3.

Heapify at index 3 (value 43):

- Children: 23 (left), 63 (right).
- Largest = 63 → Swap.

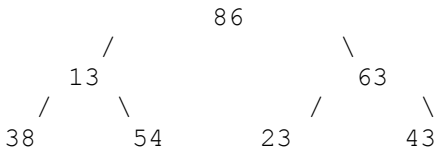


Heapify at index 2 (value 86):

- Children: 38, 54.
- Largest = 86 → Already valid.
  (No change)

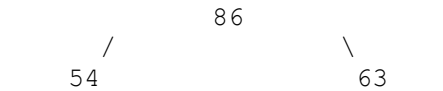
Heapify at index 1 (value 13):

- Children: 86, 63.
- Largest = 86 → Swap with 13.



Now, heapify index 2 (value 13):

- Children: 38, 54.
- Largest = 54 → Swap with 13.



Heapify index 4 (value 38): leaf → no change.

Final Max Heap:

Heap array representation:

```
[86, 54, 63, 38, 13, 23, 43]
```

Q10.a. Write short notes on a) Dynamic programming b) Trie

Definition

- **Dynamic Programming** is a method of solving problems by **breaking them down into overlapping subproblems** and solving each subproblem only once, storing the results (usually in a table or array) for future use.
- It is mainly used to optimize **recursive problems** that recompute the same results multiple times

Key Characteristics

1. **Optimal Substructure:**
   - A problem has optimal substructure if its solution can be constructed from solutions of subproblems.
2. **Overlapping Subproblems:**
   - A problem has overlapping subproblems if the same subproblems occur multiple times during recursion.

Difference between Recursion and Dynamic Programming

Aspect	Recursion	Dynamic Programming
Definition	Problem solved by breaking into subproblems, solved via function calls.	Enhances recursion by storing subproblem results to avoid recomputation.
Efficiency	May recompute the same subproblem many times → exponential time (O(2^n) for Fibonacci).	Avoids recomputation by storing results → polynomial time (O(n) for Fibonacci).
Memory Usage	Uses call stack (may lead to stack overflow for deep recursions).	Uses additional memory (table/array) to store results.
Approach	*Top-down only*.	Can be implemented *Top-down*

		(Memoization) or *Bottom-up (Tabulation).*

Example 1: Fibonacci Numbers

Recursive approach (without DP):

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

- Time Complexity: O(2^n) (recomputes many subproblems).

DP approach (Memoization – Top-down):

```
memo = {}
def fib_dp(n):
    if n <= 1:
        return n
    if n not in memo:
        memo[n] = fib_dp(n-1) + fib_dp(n-2)
    return memo[n]
```

- Time Complexity: O(n).
- Stores results of subproblems in `memo`.

Example 2: 0/1 Knapsack Problem

- **Problem:** Given `n` items with weights and values, and a knapsack capacity `W`, maximize the total value without exceeding the capacity.

Recursive approach:

```
def knapsack(wt, val, W, n):
    if n == 0 or W == 0:
        return 0
    if wt[n-1] > W:
        return knapsack(wt, val, W, n-1)
    else:
        return max(val[n-1] + knapsack(wt, val, W-wt[n-1], n-1),
                   knapsack(wt, val, W, n-1))
```

- Time Complexity: O(2^n).

DP approach (Bottom-up – Tabulation):

```
def knapsack_dp(wt, val, W, n):
    dp = [[0]*(W+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(1, W+1):
            if wt[i-1] <= w:
                dp[i][w] = max(val[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][W]
```

- Time Complexity: O(nW).
- Stores solutions in a **2D table**.

Trie is an efficient information re***Trie***val data structure. The term tries comes from the word retrieval

Definition of a Trie

· Data structure for representing a collection of strings

· In computer science , a trie also called digital tree or radix tree or prefix tree. · Tries support fast string matching.

Properties of Tries

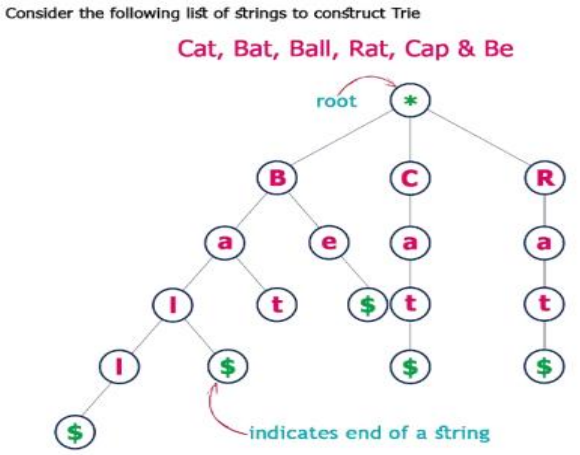
· A Multi way tree

· Each node has from 1 to n children

· Each edge of the tree is labeled with a character

· Each leaf node corresponds to the stored string which is a concatenation of characters on a path from the root to this node.

EXAMPLE

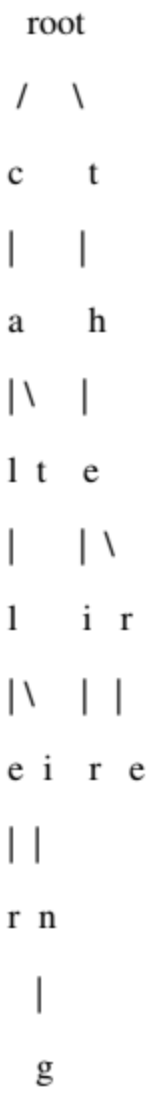


Trie | (Insert and Search)

Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to an optimal limit (key length).Given multiple strings. The task is to insert the string in a Trie

Examples:

Example 1: str = {"cat", "there", "caller", "their", "calling", “bat”}



Approach: An efficient approach is to treat every character of the input key as an individual trie node and insert it into the trie. Note that the children are an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array of children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Trie deletion Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.

2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.

3. Key is prefix key of another long key in trie. Unmark the leaf node.

4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

Time Complexity: The time complexity of the deletion operation is O(n) where n is the key length

Advantages of Trie Data Structure

Tries is a tree that stores strings. The maximum number of children of a node is equal to the size of the alphabet. Trie supports search, insert and delete operations in O(L) time where L is the length of the key.

Hashing:- In hashing, we convert the key to a small value and the value is used to index data. Hashing supports search, insert and delete operations in O(L) time on average.

Self Balancing BST : The time complexity of the search, insert and delete operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(L * Log n) where n is total number words and L is the length of the word. The advantage of Self-balancing BSTs is that they maintain order which makes operations like minimum, maximum, closest (floor or ceiling) and kth largest faster.

APPLICATIONS OF TRIES

String handling and processing are one of the most important topics for programmers. Many real time applications are based on the string processing like:

1. Search Engine results optimization

2. Data Analytics

3. Sentimental Analysis

The data structure that is very important for string handling is the Trie data structure that is based on prefix of string

TYPES OF TRIES

Tries are classified into three categories:

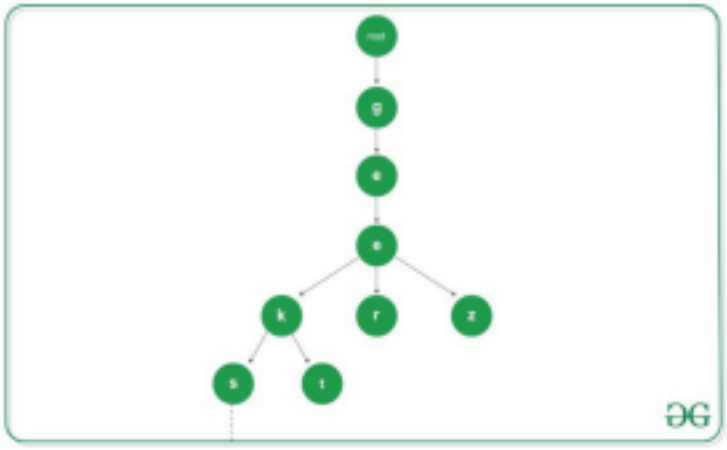
1. Standard Tries
2. Compressed Tries
3. Suffix Tries

STANDARD TRIES

A standard trie have the following properties:}

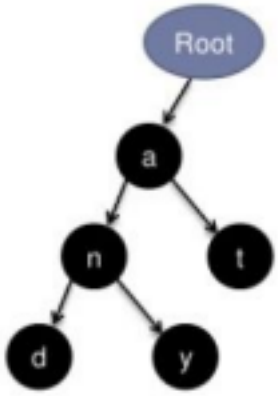
· It is an ordered tree like data structure.

· Each node(except the root node) in a standard trie is labeled with a character. · The children of a node are in alphabetical order.
· Each node or branch represents a possible character of keys or words. · Each node or branch may have multiple branches.
· The last node of every key or word is used to mark the end of word or node. · The path from external node to the root yields the string of S. Below is the illustration of the Standard Trie



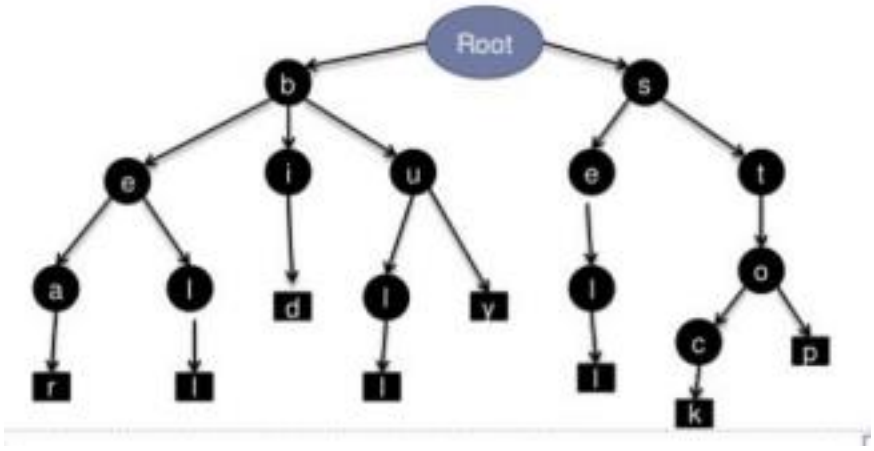
Standard Trie Insertion

Strings={ a,an,and,any}



Example of Standard Trie

Standard trie for the following strings S={ bear, bell, bid, bull, buy, sell, stock, stop}



Handling Keys(strings)

· When a key is prefix of another key
How can we know that “an “ is a word
Example : an, and

COMPRESSED TRIE

A Compressed trie have the following properties:

1. A Compressed Trie is an advanced version of the standard trie.

2. Each nodes(except the leaf nodes) have atleast 2 children.

3. It is used to achieve space optimization.

4. To derive a Compressed Trie from a Standard Trie, compression of chains of redundant nodes is performed.

It consists of grouping, re-grouping and un-grouping of keys of characters.

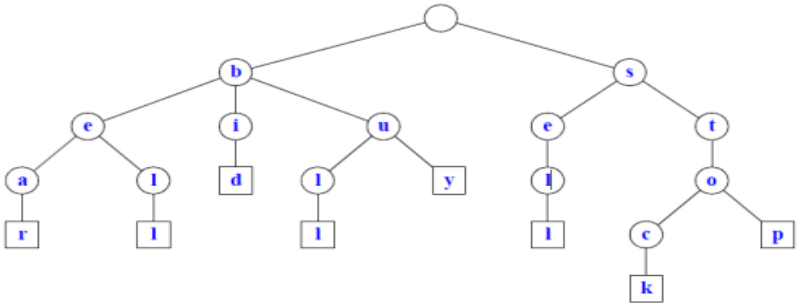
6. While performing the insertion operation, it may be required to un-group the already

grouped characters.

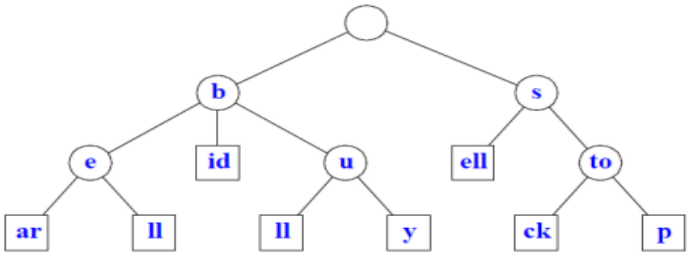
7. While performing the deletion operation, it may be required to re-group the already grouped characters.

Compressed trie is constructed from standard trie

- Standard Trie:



- Compressed Trie:



Q10. b. Explain segment and Fenwick tree with an example

1. Segment Tree

Definition

- A **Segment Tree** is a binary tree used for storing information about intervals (segments).
- It allows efficient **range queries** (sum, min, max, gcd, etc.) and **updates** on an array.

Construction

- Each node represents an interval `[L, R]`.
- The root node represents the entire range.
- Each internal node = merge of its children.
- Leaf nodes = single elements.

Operations

1. **Build Tree** → O(n)
2. **Query (Range Sum/Min/Max)** → O(log n)
3. **Update** (change element) → O(log n)

Example: Range Sum using Segment Tree

Array: **[2, 4, 5, 7, 8, 9]**

- Root stores sum of entire array = 35.
- Break into left `[2, 4, 5]` (sum=11), right `[7, 8, 9]` (sum=24).
- Continue dividing until single elements.

Query: Sum of range [2..4] (elements 4+5+7 = 16).

- The tree helps us compute this quickly by combining relevant nodes.

2. Fenwick Tree (Binary Indexed Tree - BIT)

Definition

- A **Fenwick Tree** is a data structure that provides efficient methods for **prefix sum queries** and **updates**.
- It uses clever indexing with binary representation of numbers.

Operations

1. **Update(index, value)** → O(log n)

2. **Query(prefix sum 1..i)** → O(log n)
3. **Range Sum (l..r)** = Query(r) – Query(l-1)

Example: Prefix Sums with Fenwick Tree

Array: **[2, 4, 5, 7, 8, 9]**

- Internal tree stores cumulative sums at different ranges.
- Suppose we want **sum(1..4)** → result = 18.
- The tree jumps through indices using the **last set bit** in binary.