DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE ENGINEERING

LIST OF EXPERIMENTS

Ex.No.	Name of the Experiment	Date of Experiment	Date of Submission	Page No	Faculty Signature
1.	Write code and understand the concept Variable, Data type, and Data Object				
2.	Write code and understand the concept List in data structure				
3.	Write code and understand the concept Queue in data structure				
4.	Write code and understand the concept Array in data structure				
5.	Write code and understand the concept Graph, Trees in data structure				
6.	Write code and understand the concept Hashing, Hash tables in data structure				
7.	Write code and understand the concept Search Algorithms (linear Search, Binary Search)				
8.	Write code and understand the concept Sorting Algorithm (Bubble Sort, Insertion Sort)				
9.	Write code and understand the concept Algorithms Technique on Greedy Approach				

Experiment No-1

Aim:

Write code and understand the concept Variable, Data Type and Data Object in C

Theory:

Variables

- A variable is a named storage location that holds a value.

- Variables have a data type, which determines the type of value they can store.

- Variables are key building elements of the C programming language used to store and modify data in computer programs.

-Each variable has a **unique identifier**, its ***name***, and a ***data type*** describing the type of data it may hold.

Rules for Naming Conventions in C

- Names can contain letters, digits and underscores.
- Names must begin with a letter or an underscore (_)
- Names are case-sensitive (myVar and myvar are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (such as int) cannot be used as names.

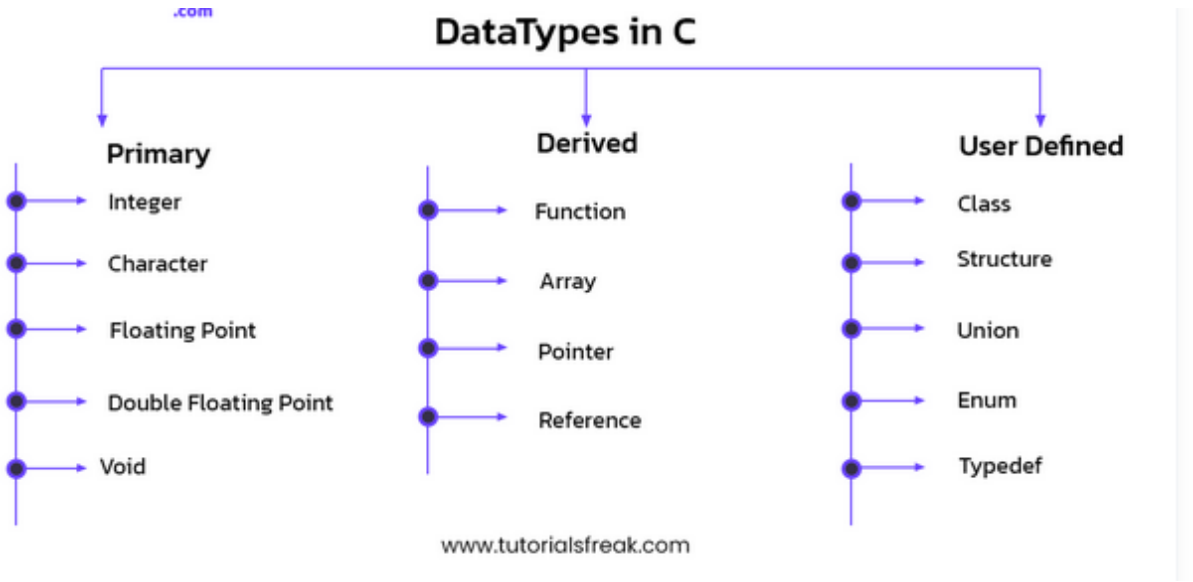
Syntax:

```
data_type variable_name;
```

Data Types-

-A data type is a characteristic of the data that tells the compiler how to interpret the value

-Data types are used when defining variables and functions



Primitive Data Types -Primitive data types are the most basic data types that are used for representing simple values such as **integers, float, characters**, **float, double, void** etc.

Derived Data Types- The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types such as **array, pointers, function**

User Defined Data Types- The user-defined data types are defined by the user himself such as **structure, union, Enum**

Primitive Data Types:

1)Integer Data Type

The integer datatype in C is used to store the integer numbers (any number including positive, negative and zero without decimal part).

Octal values, hexadecimal values, and decimal values can be stored in int data type in C.

- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 4 bytes
- **Format Specifier:** %d

Syntax of Integer

We use int keyword to declare the integer variable:

```
int var_name;
```

2)Character Data Type

-Character data type allows its variable to store only a single character.

-The size of the character is 1 byte. It is the most basic data type in C.

- It stores a single character and requires a single byte of memory in almost all compilers.

- **Range:** (-128 to 127) or (0 to 255)
- **Size:** 1 byte
- **Format Specifier:** %c

Syntax of char

The **char keyword** is used to declare the variable of character type:

char *var_name;*

3)Float Data Type

-In C programming float data type is used to store floating-point values.

- Float in C is used to store decimal and exponential values.

- It is used to store decimal numbers (numbers with floating point values) with single precision.

- Range: 1.2E-38 to 3.4E+38
- Size: 4 bytes
- Format Specifier: %f

Syntax of float

The **float** keyword is used to declare the variable as a floating point:

float *var_name;*

4)Double Data Type

A Double data type in C is used to store decimal numbers (numbers with floating point values) with double precision. It is used to define numeric values which hold numbers with decimal values in C.

The double data type is basically a precision sort of data type that is capable of holding 64 bits of decimal numbers or floating points.

Since double has more precision as compared to that float then it is much more obvious that it occupies twice the memory occupied by the floating-point type.

It can easily accommodate about 16 to 17 digits after or before a decimal point.

- Range: 1.7E-308 to 1.7E+308
- Size: 8 bytes
- Format Specifier: %lf

Syntax of Double

The variable can be declared as double precision floating point using the double keyword:

double *var_name;*

5)Void Data Type

The void data type in C is used to specify that no value is present.

It does not provide a result value to its caller. It has no values and no operations

. It is used to represent nothing. Void is used in multiple ways as function return type, function arguments as void, and pointers to void.

Syntax:

void *name_of_pointer;

2)Derived Data Types:

1. Functions

A function is called a C language construct which consists of a function-body associated with a function-name.

In every program in C language, execution begins from the main function, which gets terminated after completing some operations which may include invoking other functions.

Function Declaration

return_type function_name(data_type *param1*, data_type *param2*, ...);

2. Arrays

Array in C is a fixed-size collection of similar data items stored in contiguous memory locations.

An array is capable of storing the collection of data of primitive, derived, and user-defined data types.

Array Declaration

data_type array_name [*size*];

3.Pointer

A pointer in C language is a data type that stores the address where data is stored. Pointers store memory addresses of variables, functions, and even other pointers.

Pointer Declaration

data_type * *ptr_name*;

3)User-Defined Data Types:

1. Structure

As we know, C doesn't have built-in object-oriented features like C++ but structures can be used to achieve encapsulation to some level.

Structures are used to group items of different types into a single type.

The "struct" keyword is used to define a structure.

The size of the structure is equal to or greater than the total size of all of its members.

Syntax-

struct *structure_name* {

data_type *member_name1;*

data_type *member_name1;*

....

....

}

2. Union

Unions are similar to structures in many ways. What makes a union different is that all the members in the union are stored in the same memory location resulting in only one member containing data at the same time. The size of the union is the size of its largest member. Union is declared using the “union” keyword.

Syntax

union *union_name* {

datatype *member1;*

datatype *member2;*

...

};

3.Enumeration (enums)

Enum is short for “Enumeration”. It allows the user to create custom data types with a set of named integer constants. The “enum” keyword is used to declare an enumeration. Enum simplifies and makes the program more readable.

Syntax

enum *enum_name* {const1, const2, ..., constN};

4)**Typedef**

typedef is used to redefine the existing data type names. Basically, it is used to provide new names to the existing data types. The “typedef” keyword is used for this purpose;

Syntax

typedef *existing_name alias_name;*

Data Objects:

- A data object is a region of memory that stores a value.

- Data objects can be variables, constants, or expressions.

Conclusion:

In C programming, understanding variables, data types, and data objects is crucial for effective coding.

- Variables provide named storage for values.

- Data types determine the type of value a variable can hold.

- Data objects, including variables, constants, and expressions, store values in memory.

Sample program-

```
#include <stdio.h>
int main() {
    // Variable declarations (data objects)
    int age;            // Variable of type int
    float height;       // Variable of type float
    char initial;       // Variable of type char

    // Assigning values to variables
    age = 25;                // Assigning an integer value
    height = 5.9;            // Assigning a floating-point value
    initial = 'A';           // Assigning a character value

    // Printing the values
    printf("Age: %d years\n", age);              // %d is used for int
    printf("Height: %.2f feet\n", height);       // %.2f is used for float
    printf("Initial: %c\n", initial);            // %c is used for char
```

```
    // Using an array (derived data type)
    int scores[5] = {90, 85, 78, 92, 88};          // Array of integers
    printf("Scores: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", scores[i]);                  // Accessing array elements
    }
    printf("\n");

    return 0;  // Indicating successful execution
}
```

Output-

```
Age: 25 years
Height: 5.90 feet
Initial: A
Scores: 90 85 78 92 88


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

Experiment No-2

Aim:

Write code and understand the concept List in data Structure.

Theory:

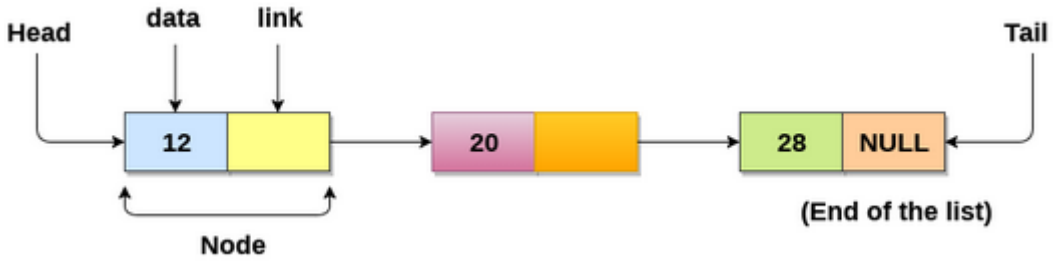
List:

A list is a linear data structure that stores a collection of elements in a specific order.

The list can be defined as an abstract data type in which the elements are stored in an ordered manner for easier and efficient retrieval of the elements

List Data Structure allows repetition that means a single piece of data can occur more than once in a list

It is very much similar to the array but the major difference between the array and the list data structure is that array stores only homogenous data in them whereas the list (in some programming languages) can store heterogeneous data items in its object. List Data Structure is also known as a sequence.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node cannot be present in the linked list.
- We can store values of primitive types or objects in the singly linked list**.**

Types of Lists:

1)singly linked list

2)Doubly linked list

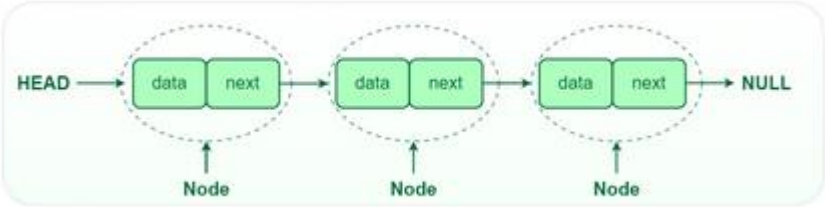
3) Circular linked list

1)singly linked list

A **singly linked list** is a fundamental data structure in computer science and programming, it consists of **nodes** where each node contains a **data** field and a **reference** to the next node in the node.

The last node points to **null**, indicating the end of the list.

This linear structure supports efficient insertion and deletion operations, making it widely used in various applications. In this tutorial, we'll explore the node structure, understand the operations on singly linked lists (traversal, searching, length determination, insertion, and deletion), and provide detailed explanations and code examples to implement these operations effectively.



Node Structure: A node in a linked list typically consists of two components:

1. **Data:** It holds the actual value or data associated with the node.
2. **Next Pointer or Reference:** It stores the memory address (reference) of the next node in the sequence.

Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

2)Doubly linked list.

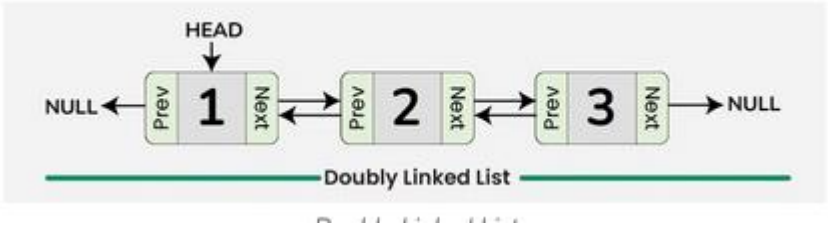
A **doubly linked list** is a data structure that consists of a set of nodes, each of which contains a **value** and **two pointers**, one pointing to the **previous node** in the list and one pointing to the **next node** in the list.

This allows for efficient traversal of the list in **both directions**, making it suitable for applications where frequent **insertions** and **deletions** are required.

A **doubly linked list** is a more complex data structure than a singly linked list, but it offers several advantages.

The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node.

This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.



Representation of Doubly Linked List in Data Structure

In a data structure, a doubly linked list is represented using nodes that have three fields:

1. Data
2. A pointer to the next node (**next**)
3. A pointer to the previous node (**prev**)

3) Circular linked list

A **circular linked list** is a special type of linked list where all the nodes are connected to form a circle.

Unlike a regular linked list, which ends with a node pointing to **NULL**, the last node in a circular linked list points back to the first node.

This means that you can keep traversing the list without ever reaching a **NULL** value.

Types of Circular Linked Lists

1)Circular Singly Linked List

2) Circular Doubly Linked List

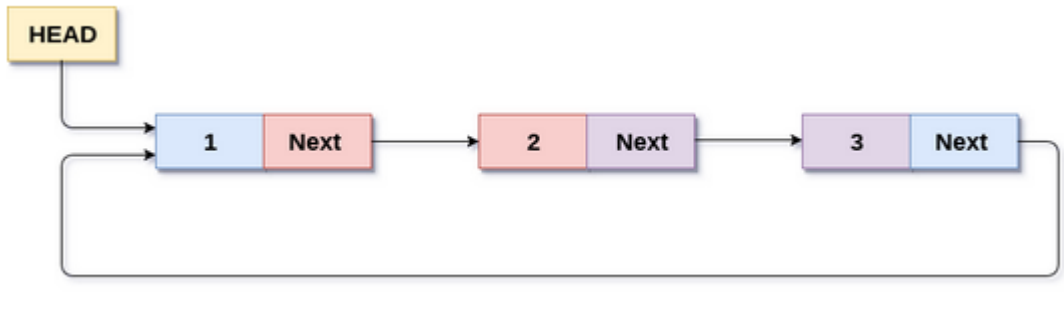
1)Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

Circular linked list is mostly used in task maintenance in operating systems. There are many examples where circular linked list is being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button

The following image shows a circular singly linked list.



2)Circular Doubly Linked List

.

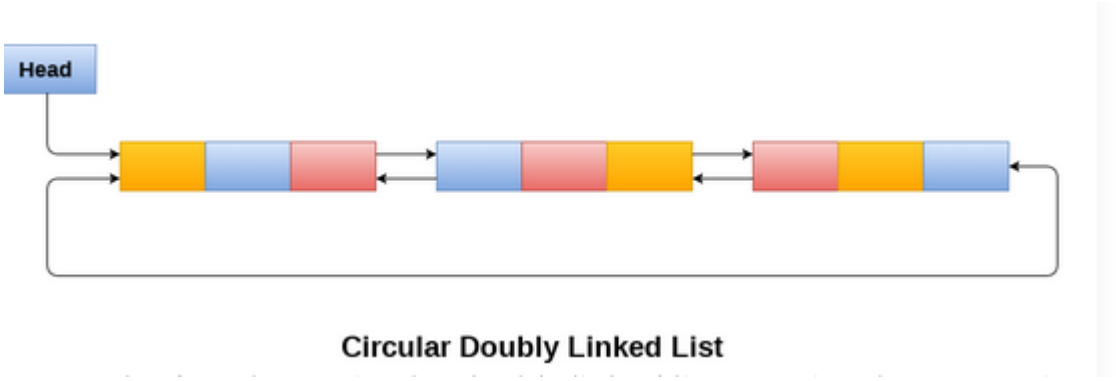
Circular doubly linked list is a more complexed type of data structure in which a node contains pointers to its previous node as well as the next node.

Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list.

The first node of the list also contains address of the last node in its previous pointer.

Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient

A circular doubly linked list is shown in the following figure.



Circular Doubly Linked List

.

Operations on Lists:

1. Insertion (at beginning, end, or specific position)
2. Deletion (at beginning, end, or specific position)
3. Traversal (printing or accessing elements)
4. Search (finding a specific element)

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Sample Program-

Insert a node in linked list

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

// Structure for a Node
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the linked list
void insertAtBeginning(struct Node** head_ref, int new_data) {
    // Create a new node
    struct Node* new_node = createNode(new_data);

    // Make next of new node as head
    new_node->next = *head_ref;

    // Move the head to point to the new node
    *head_ref = new_node;
}
```

```
// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

// Main function to test the linked list operations
int main() {
    struct Node* head = NULL; // Start with an empty linked list

    // Inserting nodes into the linked list at the beginning
    insertAtBeginning(&head, 40);
    insertAtBeginning(&head, 30);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 10);

    // Print the linked list
    printList(head);

    // Free memory (not shown here for simplicity)
    return 0;
}
```

Output-

```
10 -> 20 -> 30 -> 40 -> NULL


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

Delete a node in linked list

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

// Node structure for the linked list
struct Node {
    int data;
    struct Node* next;
};



// Function to delete the head node
struct Node* deleteHead(struct Node* head)
{
    // Base case if linked list is empty
    if (head == NULL)
        return NULL;

    // Store the current head in a temporary variable
    struct Node* temp = head;

    // Move the head to the next node
    head = head->next;
```

```
    // Free the memory of the old head node
    free(temp);

    // Return the new head
    return head;
}

// Function to print the linked list
void printList(struct Node* head)
{
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

// Function to create a new node
struct Node* createNode(int data)
{
    struct Node* node
        = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->next = NULL;
    return node;
}

int main()
{
```

```
    // Creating a linked list
    // 1 -> 2 -> 3 -> 4 -> 5 -> NULL
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    head->next->next->next = createNode(4);
    head->next->next->next->next = createNode(5);

    printf("Original list: ");
    printList(head);

    // Deleting the head node
    head = deleteHead(head);

    printf("List after deleting the head: ");
    printList(head);

    return 0;
}
```

Output-

```
Original list: 1 -> 2 -> 3 -> 4 -> 5 -> NULL
List after deleting the head: 2 -> 3 -> 4 -> 5 -> NULL


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

Conclusion:

Lists are essential data structures for storing and manipulating collections of elements.

Experiment No-3

Aim:

Write code and understand the concept, Queue in data structure

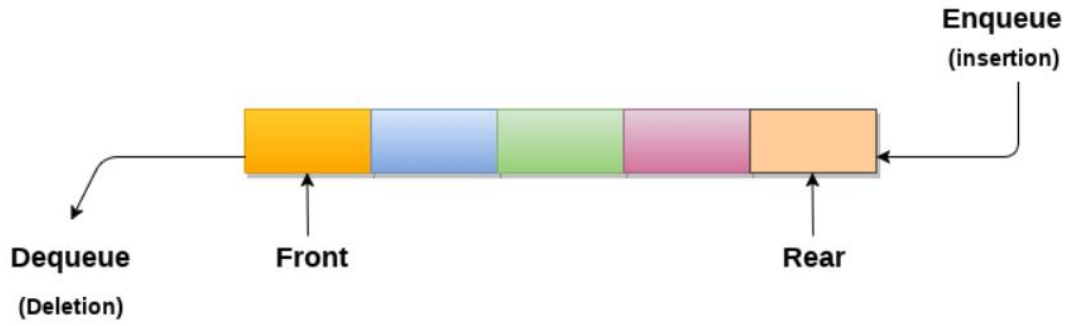
Theory:

Queue:

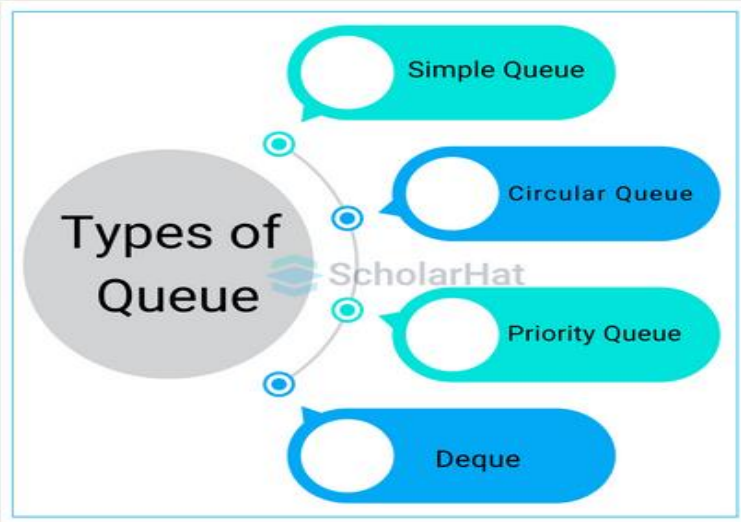
- A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

- Queue is referred to be as First In First Out list.

- For example, people waiting in line for a rail ticket form a queue.



Types of Queue-



Types of Queue-

1. Simple Queue/Linear Queue

2. Circular Queue

3. Priority Queue

1. Simple Queue/Linear Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end.

It strictly follows the **FIFO** rule.

The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

2. Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element.

It is also known as Ring Buffer, as all the ends are connected to another end.

The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization

3.Priority Queue

It is a special type of queue in which each element has a priority assigned to it.
The element with the highest priority is removed first.
This is useful in situations where certain elements need to be processed before others

Operations on Queues:

1)Enqueue

2)Dequeue

3)QueueFull

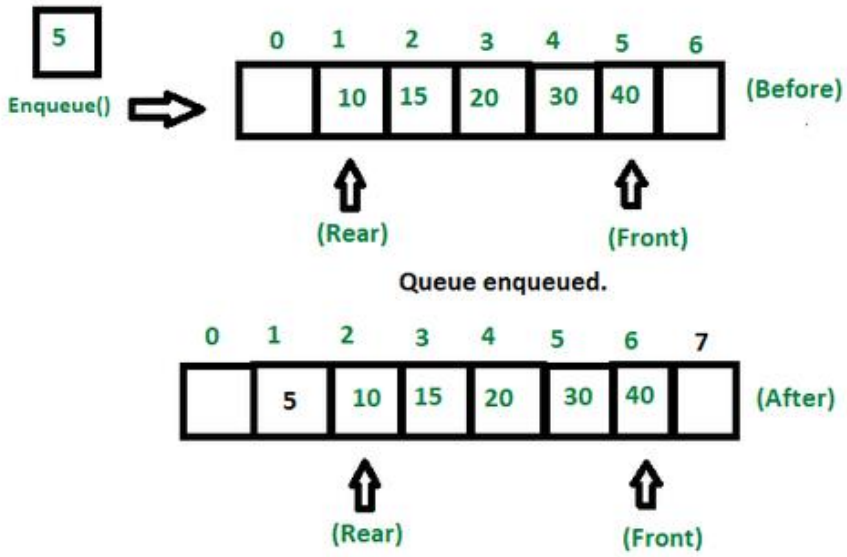
4)QueueEmpty

1)Enqueue-

Inserts an element at the end of the queue i.e. at the rear end.

The following steps should be taken to enqueue (insert) data into a queue:

- Check if the queue is full.
- If the queue is full, return overflow error and exit.
- If the queue is not full, increment the rear pointer to point to the next empty space.
- Add the data element to the queue location, where the rear is pointing.
- return success.

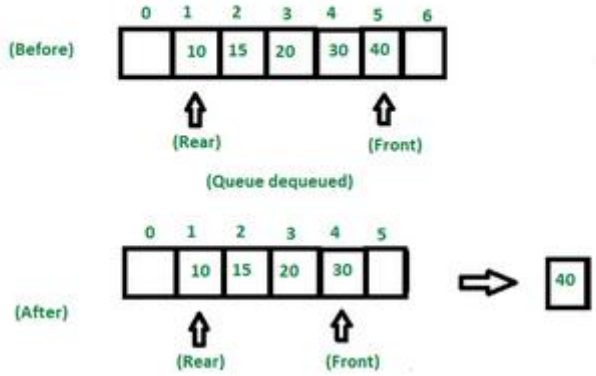


2)Dequeue

This operation removes and returns an element that is at the front end of the queue.

The following steps are taken to perform the dequeue operation:

- Check if the queue is empty.
- If the queue is empty, return the underflow error and exit.
- If the queue is not empty, access the data where the front is pointing.
- Increment the front pointer to point to the next available data element.
- The Return success.



3)QueueFull/ isFull()

This operation returns a boolean value that indicates whether the queue is full or not.

The following steps are taken to perform the isFull() operation:

- Check if front value is equal to zero and rear is equal to the capacity of queue if yes then return true.
- otherwise return false

4)QueueEmpty/isEmpty()

This operation returns a Boolean value that indicates whether the queue is empty or not.

The following steps are taken to perform the Empty operation:

- check if front value is equal to -1 or not, if yes then return true means queue is empty.
- Otherwise return false, means queue is not empty

Conclusion:

Queues are essential data structures for managing elements in a FIFO order.

Sample Program

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5  // Maximum size of the queue
// Queue structure
struct Queue {
    int arr[MAX];  // Array to hold queue elements
    int front;     // Index of the front element
    int rear;      // Index of the rear element
};
// Function to initialize the queue
void initializeQueue(struct Queue* q) {
    q->front = -1;
    q->rear = -1;
}
// Check if the queue is full
int isFull(struct Queue* q) {
    return (q->rear == MAX - 1);
}
// Check if the queue is empty
int isEmpty(struct Queue* q) {
```

```
    return (q->front == -1);
}
// Enqueue operation (insert an element)
void enqueue(struct Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue %d\n", value);
        return;
    }

    if (q->front == -1) {  // First element to be inserted
        q->front = 0;
    }

    q->rear++;
    q->arr[q->rear] = value;
    printf("Enqueued %d\n", value);
}

// Dequeue operation (remove an element)
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;  // Return a sentinel value
    }

    int dequeuedValue = q->arr[q->front];
    if (q->front == q->rear) {  // Only one element left
        q->front = -1;
        q->rear = -1;
    } else {
```

```
        q->front++;
    }

    return dequeuedValue;
}
// Display the elements of the queue
void display(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Queue elements: ");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d ", q->arr[i]);
    }
    printf("\n");
}

int main() {
    struct Queue q;
    initializeQueue(&q);

    // Enqueue some elements
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50);
```

```
    // Try to enqueue when the queue is full
    enqueue(&q, 60);

    // Display queue
    display(&q);

    // Dequeue some elements
    printf("Dequeued %d\n", dequeue(&q));
    printf("Dequeued %d\n", dequeue(&q));

    // Display queue after dequeuing
    display(&q);

    // Enqueue more elements
    enqueue(&q, 60);
    enqueue(&q, 70);

    // Display queue after enqueuing
    display(&q);

    return 0;
}
```

Output-

```
Enqueued 10
Enqueued 20
Enqueued 30
Enqueued 40
Enqueued 50
Queue is full. Cannot enqueue 60
Queue elements: 10 20 30 40 50
Dequeued 10
Dequeued 20
Queue elements: 30 40 50
Queue is full. Cannot enqueue 60
Queue is full. Cannot enqueue 70
Queue elements: 30 40 50


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

Experiment No-4

Aim:

Write code and understand the concept Array in data Structure

Theory:

Array: An array is a collection of elements of the same data type stored in contiguous memory locations.

It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

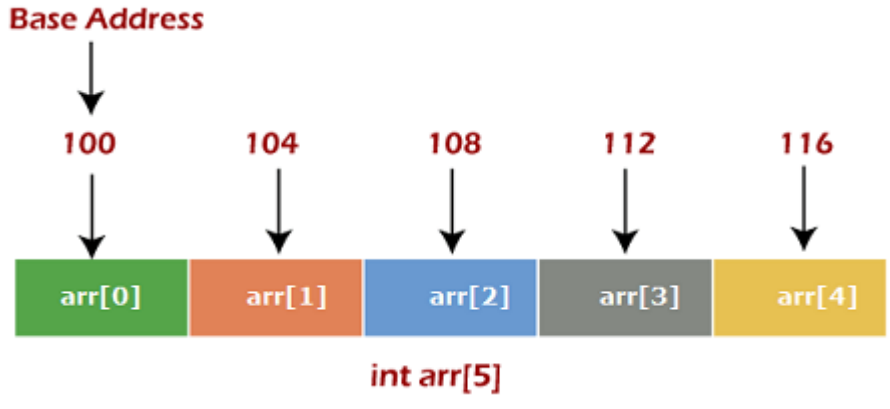
Each element in an array is of the same data type and carries the same size that is 4 bytes.

Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.

Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

We can define the indexing of an array in the below ways -

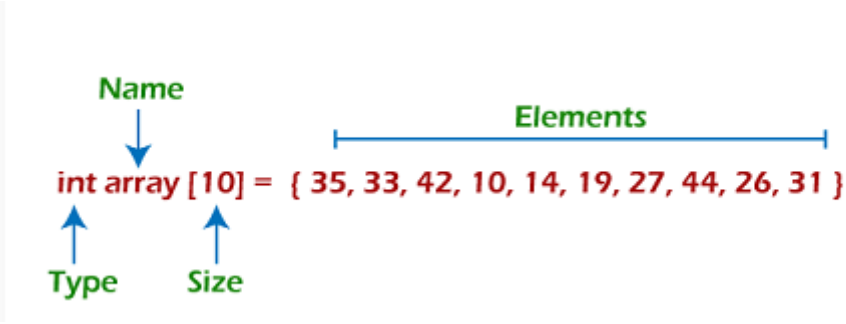
1. 0 (zero-based indexing): The first element of the array will be arr[0].
2. 1 (one-based indexing): The first element of the array will be arr[1].
3. n (n - based indexing): The first element of the array can reside at any random index number.



In the above image, we have shown the memory allocation of an array arr of size 5. The array follows a 0-based indexing approach. The base address of the array is 100 bytes. It is the

address of arr[0]. Here, the size of the data type used is 4 bytes; therefore, each element will take 4 bytes in the memory.

Representation of an array



As per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

Syntax of Array-

datatype Array_Name[size] = { value1, value2, value3, ….. valueN };

Properties of array

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantages of Array

- Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

Disadvantages of Array

- Array is homogenous. It means that the elements with similar data type can be stored in it.
- In array, there is static memory allocation that is size of an array cannot be altered.
- There will be wastage of memory if we store less number of elements than the declared size.

Operations on Arrays:

1.Insertion.

2.Traversal

3. Search.

4.Delete

1)Insertion Operation

In the insertion operation, we are adding one or more elements to the array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

This is done using input statements of the programming languages.

Algorithm

Following is an algorithm to insert elements into a Linear Array until we reach the end of the array −

1. Start
2. Create an Array of a desired datatype and size.
3. Initialize a variable 'i' as 0.
4. Enter the element at ith index of the array.
5. Increment i by 1.
6. Repeat Steps 4 & 5 until the end of the array.
7. Stop

2)Traversal Operation

This operation traverses through all the elements of an array. We use loop statements to carry this out.

Algorithm

Following is the algorithm to traverse through all the elements present in a Linear Array −

```
1 Start
2. Initialize an Array of certain size and datatype.
3. Initialize another variable ‘i’ with 0.
4. Print the I th value in the array and increment i.
5. Repeat Step 4 until the end of the array is reached.
6. End
```

3)Search Operation

Searching an element in the array using a key; The key element sequentially compares every value in the array to check if the key is present in the array or not.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

4)Deletion Operation-

In this array operation, we delete an element from the particular index of an array. This deletion operation takes place as we assign the value in the consequent index to the current index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the K^{th} position of LA.

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

Conclusion:

- Arrays have homogeneous elements and fixed sizes.
- Elements are accessed using indexes.
- Operations include traversal, insertion, deletion, search, and sorting.

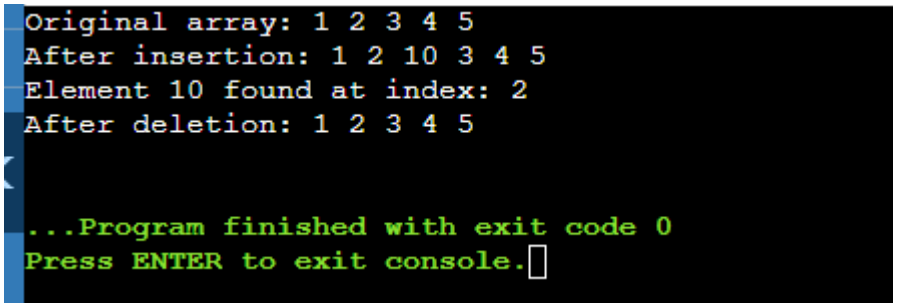
Sample Program

```
#include <stdio.h>

void insert(int arr[], int *n, int pos, int value);

void traverse(int arr[], int n);

int search(int arr[], int n, int value);

void delete(int arr[], int *n, int pos);


int main() {

   int arr[100] = {1, 2, 3, 4, 5};

   int n = 5; // current number of elements
```

```
    printf("Original array: ");
    traverse(arr, n);
    printf("\n");

    // Insertion
    insert(arr, &n, 2, 10);
    printf("After insertion: ");
    traverse(arr, n);
    printf("\n");

    // Searching
    int index = search(arr, n, 10);
    if (index != -1) {
        printf("Element 10 found at index: %d\n", index);
    } else {
        printf("Element 10 not found\n");
    }

    // Deletion
    delete(arr, &n, 2);
    printf("After deletion: ");
    traverse(arr, n);
    printf("\n");

    return 0;
}

void insert(int arr[], int *n, int pos, int value) {
    for (int i = *n; i > pos; i--) {
```

```
        arr[i] = arr[i - 1];
    }
    arr[pos] = value;
    (*n)++;
}
void traverse(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
}
int search(int arr[], int n, int value) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == value) {
            return i; // return index if found
        }
    }
    return -1; // return -1 if not found
}

void delete(int arr[], int *n, int pos) {
    for (int i = pos; i < *n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    (*n)--; // decrease the size
}
```

Output-


Original array: 1 2 3 4 5
After insertion: 1 2 10 3 4 5
Element 10 found at index: 2
After deletion: 1 2 3 4 5
...Program finished with exit code 0
Press ENTER to exit console.

Experiment No-5

Aim:

Write code and understand the concept Graphs, Trees in data Structure

Theory:

Graphs

Graph Data Structure is a non-linear data structure

A graph G can be defined as an ordered set G (V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A graph can be defined as group of vertices and edges that are used to connect these vertices.

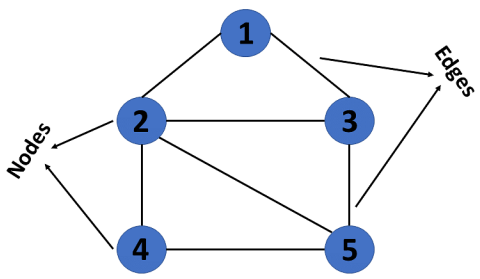
A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Graph Data Structure is a collection of nodes connected by edges.

It's used to represent relationships between different entities.

Graph algorithms are methods used to manipulate and analyse graphs, solving various problems like finding the shortest path **or** detecting cycles.

It is useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structure can be used to analyse and understand the dynamics of team performance and player interactions on the field.



Types of Graphs:

1. Directed Graph

2.UnDirected Graph

3. Weighted Graph

4. Unweighted Graph

5.Cyclic Graph

6. Acyclic Graph

7.Connected Graph

8.Disconnected Graph

9. Complete Graph

10. Subgraph

1. Directed Graph-

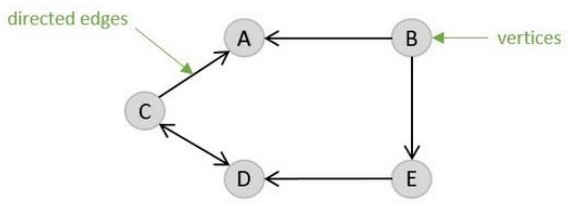
Another name for the directed graphs is digraphs.

A graph is called a directed graph or digraph if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

By directed edges, we mean the edges of the graph that have a direction to determine from which node it is starting and at which node it is ending.

All the edges for a graph need to be directed to call it a directed graph or digraph.

All the edges of a directed graph or digraph have a direction that will start from one vertex and end at another.



Graph Data Structure

Directed Graph

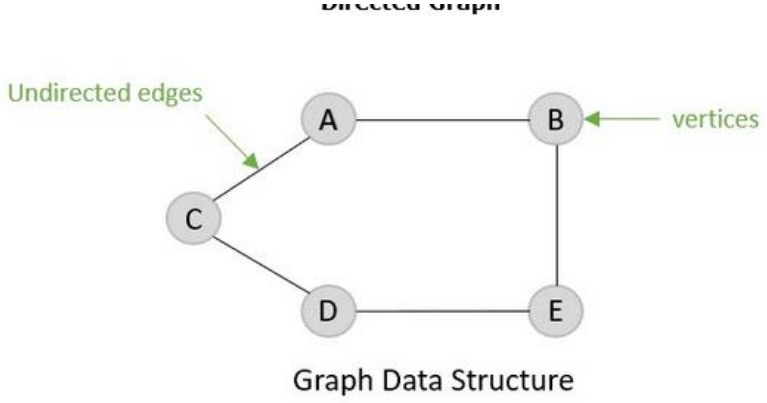
2.Undirected graph:

An undirected graph is a fundamental structure in graph theory, consisting of a set of vertices (or nodes) connected by edges. Unlike directed graphs, where edges have a direction (from one vertex to another),

In undirected graphs, the edges do not have any orientation.

This means that an edge between two vertices indicates a bidirectional relationship.

an undirected graph, edges do not have a direction. They simply connect two nodes without any particular order.



Graph Data Structure

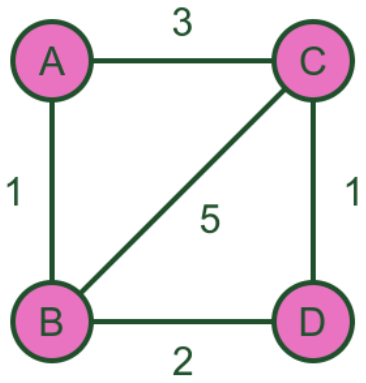
3. Weighted Graph:

A graph in which edges have weights or costs associated with them. Example: A road network graph where the weights can represent the distance between two cities.

weighted graph is a type of graph in which each edge has a numerical value, called a weight. These weights can represent various quantities, such as distances, costs, or any metric that quantifies the relationship between the vertices (nodes) connected by the edges.

Weighted graphs are widely used in various fields, including:

- **Network routing:** To find the shortest path in telecommunications.
- **Transportation:** To optimize routes based on distance or time.
- **Game development:** To manage costs in pathfinding algorithms



4.Unweighted Graph-

-An unweighted graph is a graph in which the edges do not have weights or costs associated with them. Instead, they simply represent the presence of a connection between two vertices.

-Unweighted graphs are used to represent data that are not related in terms of magnitude.

- Unweighted graphs are used to represent computation flow.

- Representation of image segmentation, where pixels are represented as nodes and edges represent adjacency relationships.

-Representation of state spaces in decision-making processes and problem-solving in AI.

- Representation of information networks, such as the World Wide Web.

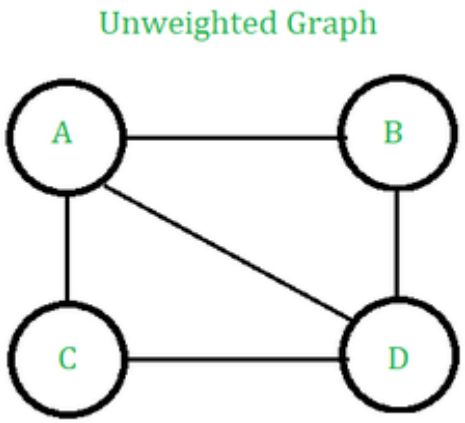
Unweighted graphs can be used to solve puzzles.

-It can be used to represent a circuit diagram.

-It can be used in social media sites to find whether two users are connected or not.

-It is used in Hamiltonian graphs which have many practical applications like genome mapping to combine many tiny fragments of genetic code.

-Used in computer networks as it represents the connections between computers in a network as an unweighted graph.



5.Cyclic Graph

A cyclic graph is defined as a graph that contains at least one cycle which is a path that begins and ends at the same node, without passing through any other node twice.

Formally, a cyclic graph is defined as a graph G = (V, E) that contains at least one cycle, where V is the set of vertices (nodes) and E is the set of edges (links) that connect them.

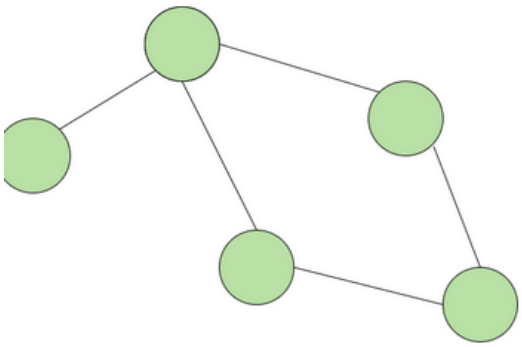
A cyclic graph contains one or more cycles or closed paths, which means that you can traverse the graph and end up where you started.

A cyclic graph can be either directed or undirected. In a directed cyclic graph, the edges have a direction, and the cycle must follow the direction of the edges. In an undirected cyclic graph, the edges have no direction, and the cycle can go in any direction.

A cyclic graph may have multiple cycles of different lengths and shapes. Some cycles may be contained within other cycles.

A cyclic graph is bipartite if and only if all its cycles are of even length.

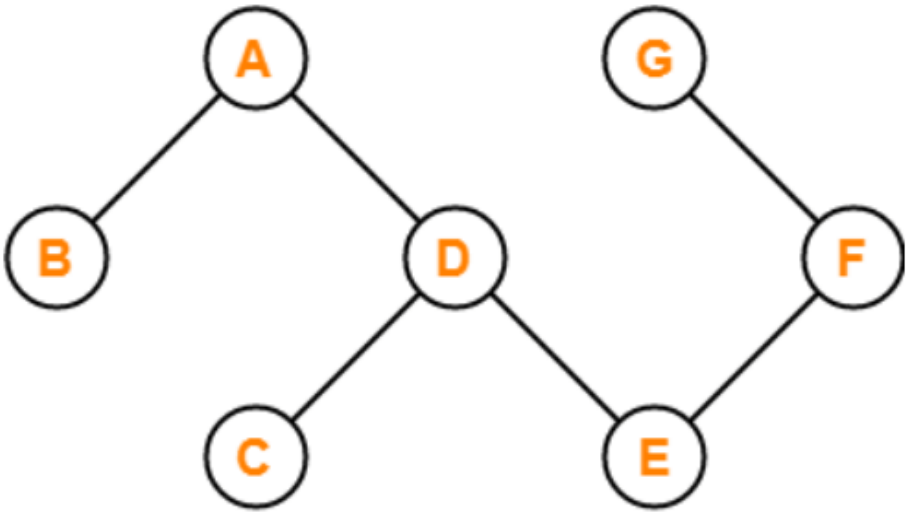
Note: Cyclic Graph and Cycle Graph are not the same.



Example of Cyclic Graph

6.Acyclic Graph-

A graph is called an acyclic graph if zero cycles are present, and an acyclic graph is the complete opposite of a cyclic graph.



The graph shown in the above image is acyclic because it has zero cycles present in it. That means if we begin traversing the graph from vertex B, then a single path doesn't exist that will traverse all the vertices and end at the same vertex that is vertex B.

An acyclic graph is **a directed graph that contains absolutely no cycle**; that is, no node can be traversed back to itself. Here, there are no paths which connect a node back to itself in the graph.

Graph Operations:

Operations of Graphs

The primary operations of a graph include creating a graph with vertices and edges, and displaying the said graph. However, one of the most common and popular operation performed using graphs are Traversal, i.e. visiting every vertex of the graph in a specific order.

There are two types of traversals in Graphs −

- Depth First Search Traversal
- Breadth First Search Traversal

Depth First Search Traversal

Depth First Search is a traversal algorithm that visits all the vertices of a graph in the decreasing order of its depth. In this algorithm, an arbitrary node is chosen as the starting point and the graph is traversed back and forth by marking unvisited adjacent nodes until all the vertices are marked.

The DFS traversal uses the stack data structure to keep track of the unvisited nodes.

Breadth First Search Traversal

Breadth First Search is a traversal algorithm that visits all the vertices of a graph present at one level of the depth before moving to the next level of depth. In this algorithm, an arbitrary node is chosen as the starting point and the graph is traversed by visiting the adjacent vertices on the same depth level and marking them until there is no vertex left.

The DFS traversal uses the queue data structure to keep track of the unvisited nodes.

Tree-

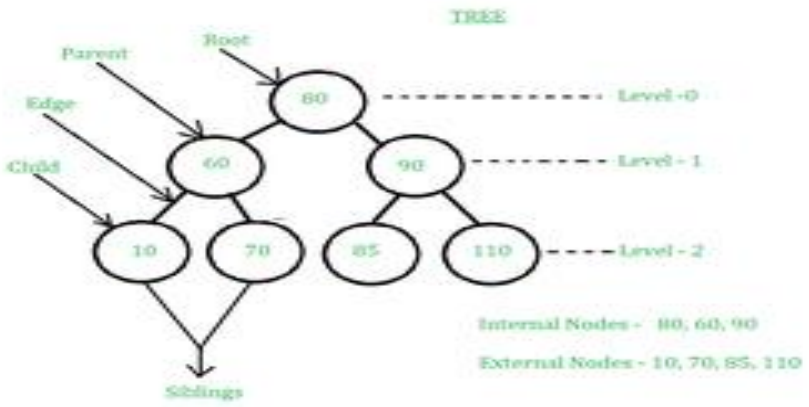
Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search.

It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

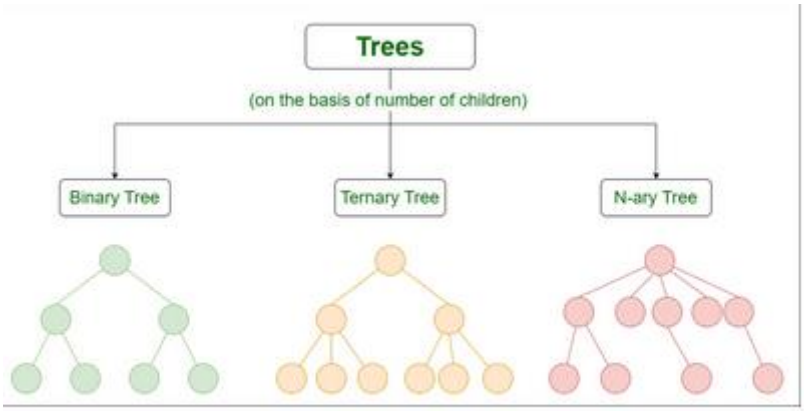
The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure**.**

Basic Terminologies in Tree Data Structure:

- **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {I, J, K, F, G, H} are the leaf nodes of the tree.
- **Ancestor of a Node**: Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node**: Parent or child nodes of that node are called neighbours of that node.
- **Subtree:** Any node of the tree along with its descendant.



Types of Trees



Types of Trees in Data Structure based on the number of children

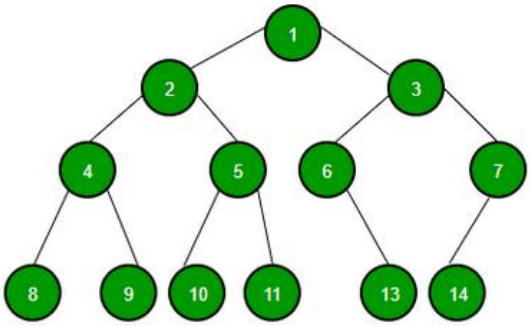
1)Binary tree:

In a binary tree, each node can have a maximum of two children linked to it.

Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.

A Binary Tree Data Structure is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is commonly used in computer science for efficient storage and retrieval of data, with various operations such as insertion, deletion, and traversal.

Examples of Binary Tree are Binary Search Tree and Binary Heap.



2)Ternary Tree

A Ternary Tree is a type of tree data structure where each node can have up to three child nodes.

This is different from a binary tree, where each node can have at most two child nodes.

In a ternary tree, the first child node is called the "left" child, the second child node is called the "middle" child, and the third child node is called the "right" child.

A **Ternary Tree** is a special type of tree data structure. Unlike a regular **binary tree** where each node can have up to two child nodes. The Experiment explains the basic structure and properties of ternary trees, such as the number of possible children per node, tree height, and node depth.

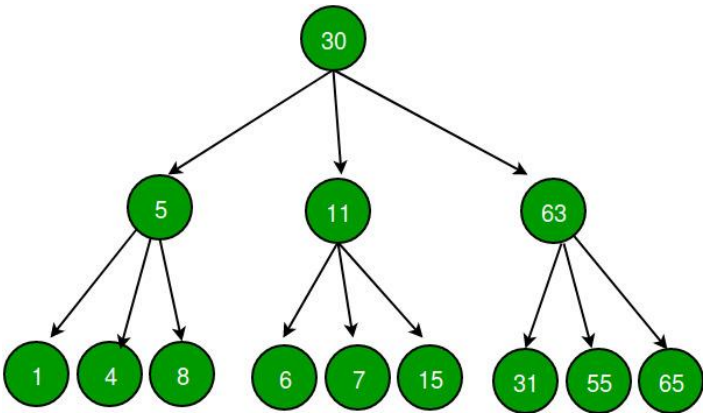
It also discusses why ternary trees can be useful, highlighting applications in areas like string searching and database indexing. It also introduces some common problems and algorithms related to ternary trees.

Basic Structure of a Ternary Tree

In a ternary tree:

- Each node has three possible children: a left child, a middle child, and a right child.
- The nodes are connected by edges that represent the parent-child relationships.

Here's a simple visualization of a ternary tree:



3)Generic Trees (N-ary Trees)-

Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed).

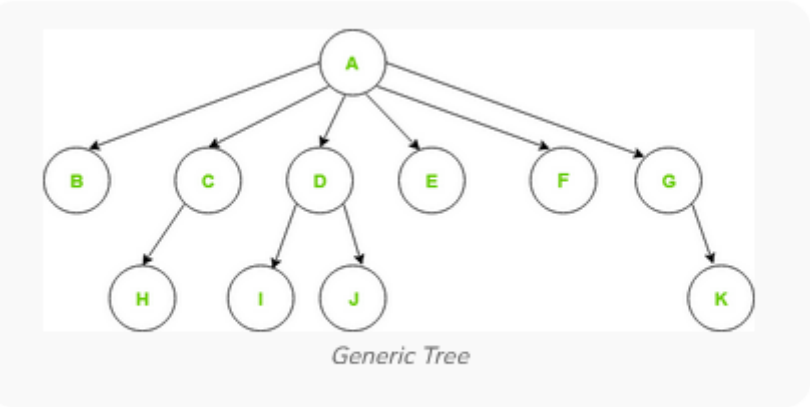
Unlike the linked list, each node stores the address of multiple nodes.

Every node stores address of its children and the very first node’s address will be stored in a separate pointer called root.

The Generic trees are the N-ary trees which have the following properties:

1. Many children at every node.

2. The number of nodes for each node is not known in advance.



To represent the above tree, we have to consider the worst case, that is the node with maximum children (in above example, 6 children) and allocate that many pointers for each node.

Conclusion:

- Graphs represent relationships between nodes.

- Trees represent hierarchical relationships.

Sample Program-(Graph)

```
#include <stdio.h>

#include <stdlib.h>


// Structure to represent a node in the adjacency list

struct Node {

   int dest;

   struct Node* next;

};


// Structure to represent the graph

struct Graph {
```

```
    int V;              // Number of vertices
    struct Node** adjList; // Array of adjacency lists
};

// Function to create a new adjacency list node
struct Node* createNode(int dest) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}

// Function to create a graph
struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;

    graph->adjList = (struct Node**)malloc(V * sizeof(struct Node*));

    for (int i = 0; i < V; i++) {
        graph->adjList[i] = NULL;
    }

    return graph;
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct Node* newNode = createNode(dest);
```

```
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    // For undirected graph, add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}

// Function to print the graph
void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->V; v++) {
        struct Node* temp = graph->adjList[v];
        printf("\n Adjacency list of vertex %d\n head ", v);
        while (temp) {
            printf("-> %d", temp->dest);
            temp = temp->next;
        }
        printf("\n");
    }
}

// Main function to test the implementation
int main() {
    int V = 5; // Number of vertices
    struct Graph* graph = createGraph(V);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
```

```
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    printGraph(graph);

    return 0;
}
```

Output-

```
Adjacency list of vertex 0
head -> 4-> 1

Adjacency list of vertex 1
head -> 4-> 3-> 2-> 0

Adjacency list of vertex 2
head -> 3-> 1

Adjacency list of vertex 3
head -> 4-> 2-> 1

Adjacency list of vertex 4
head -> 3-> 1-> 0
```

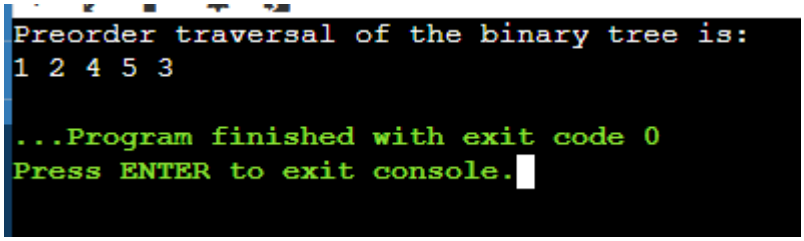
Sample Program (Tree)

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

// Structure to represent a node in the binary tree
struct Node {
    int data;
    struct Node* left;
```

```
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to print the tree in preorder traversal
void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

// Main function to test the implementation
int main() {
    // Creating a simple binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
```

```
    printf("Preorder traversal of the binary tree is:\n");
    preorderTraversal(root);

    return 0;
}
```

Output



Experiment No-6

Aim:

Write code and understand the concept Hashing, Hast Tables in data Structure.

Theory:

Hashing:

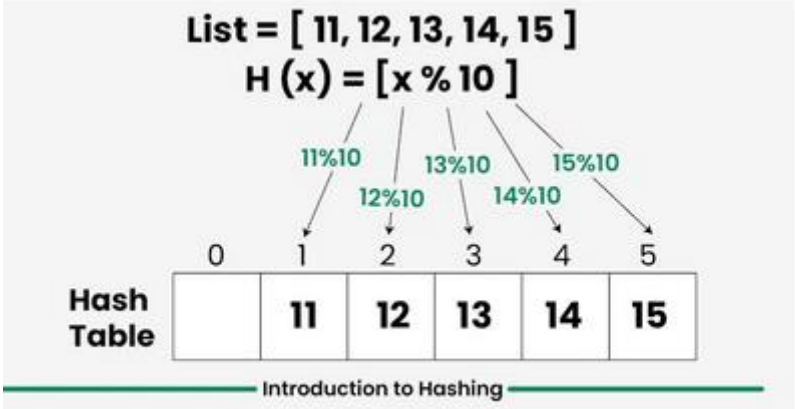
Hashing is a popular technique in computer science that involves mapping large data sets to fixed-length values.

It is a process of converting a data set of variable size into a data set of a fixed size.

The ability to perform efficient lookup operations makes hashing an essential concept in data structures.

A hashing algorithm is used to convert an input (such as a string or integer) into a fixed-size output (referred to as a hash code or hash value). The data is then stored and retrieved using this hash value as an index in an array or hash table. The hash function must be deterministic, which guarantees that it will always yield the same result for a given input.

Hashing is commonly used to create a unique identifier for a piece of data, which can be used to quickly look up that data in a large dataset. For example, a web browser may use hashing to store website passwords securely. When a user enters their password, the browser converts it into a hash value and compares it to the stored hash value to authenticate the user.



Hashing in Data Structure

Applications of Hashing

1. Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc.
2. Database indexing: Hashing is used to index and retrieve data efficiently in databases and other data storage systems.
3. Dictionaries: To implement a dictionary so that we can quickly search a word

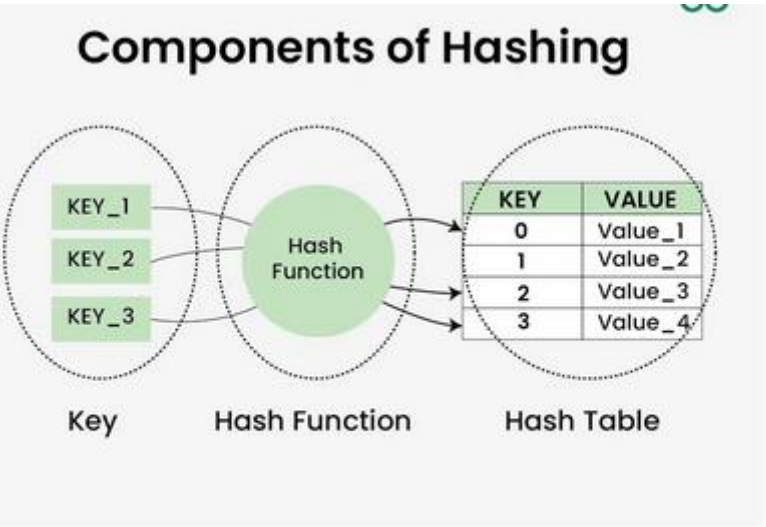
4. Password storage: Hashing is used to store passwords securely by applying a hash function to the password and storing the hashed result, rather than the plain text password.
5. Network Routing: Determining the best path for data packets
6. Bloom Filters : Bloom filter is a space optimized and probabilistic version of hashing and has huge applications like spam filtering, recommendations.
7. Cryptography: Hashing is used in cryptography to generate digital signatures, message authentication codes (MACs), and key derivation functions.
8. Load balancing: Hashing is used in load-balancing algorithms, such as consistent hashing, to distribute requests to servers in a network.
9. Blockchain: Hashing is used in blockchain technology, such as the proof-of-work algorithm, to secure the integrity and consensus of the blockchain.
10. Image processing: Hashing is used in image processing applications, such as perceptual hashing, to detect and prevent image duplicates and modifications.

Hash Table:

A hash table is a data structure that stores key-value pairs.

A Hash table is defined as a data structure used to insert, look up, and remove key-value pairs quickly. It operates on the hashing concept, where each key is translated by a hash function into a distinct index in an array.

The index functions as a storage location for the matching value. In simple words, it maps the keys with the value



Hash Function and Table

What is Load factor?

A hash table's load factor is determined by how many elements are kept there in relation to how big the table is. The table may be cluttered and have longer search times and collisions if the load factor is high. An ideal load factor can be maintained with the use of a good hash function and proper table resizing.

Hash Function:

A hash function generates an index from a key.

A Function that translates keys to array indices is known as a hash function. The keys should be evenly distributed across the array via a decent hash function to reduce collisions and ensure quick lookup speeds.

- **Integer universe assumption:** The keys are assumed to be integers within a certain range according to the integer universe assumption. This enables the use of basic hashing operations like division or multiplication hashing.
- **Hashing by division:** This straightforward hashing technique uses the key's remaining value after dividing it by the array's size as the index. When an array size is a prime number and the keys are evenly spaced out, it performs well.
- **Hashing by multiplication:** This straightforward hashing operation multiplies the key by a constant between 0 and 1 before taking the fractional portion of the outcome. After that, the index is determined by multiplying the fractional component by the array's size. Also, it functions effectively when the keys are scattered equally.

Choosing a hash function:

Selecting a decent hash function is based on the properties of the keys and the intended functionality of the hash table. Using a function that evenly distributes the keys and reduces collisions is crucial.

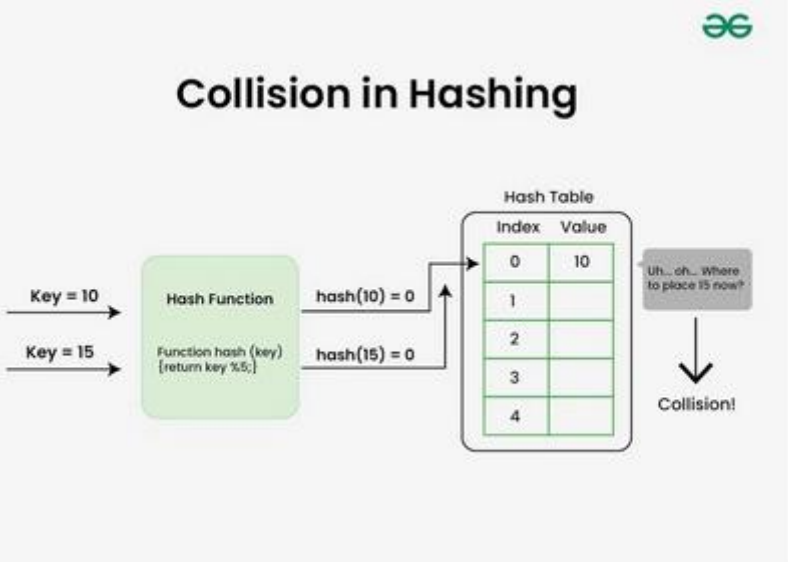
Criteria based on which a hash function is chosen:

- To ensure that the number of collisions is kept to a minimum, a good hash function should distribute the keys throughout the hash table in a uniform manner. This implies that for all pairings of keys, the likelihood of two keys hashing to the same position in the table should be rather constant.
- To enable speedy hashing and key retrieval, the hash function should be computationally efficient.
- It ought to be challenging to deduce the key from its hash value. As a result, attempts to guess the key using the hash value are less likely to succeed.

- A hash function should be flexible enough to adjust as the data being hashed changes. For instance, the hash function needs to continue to perform properly if the keys being hashed change in size or format.

Collision resolution techniques:

Collisions happen when two or more keys point to the same array index. Chaining, open addressing, and double hashing are a few techniques for resolving collisions.



- **Open addressing:** collisions are handled by looking for the following empty space in the table. If the first slot is already taken, the hash function is applied to the subsequent slots until one is left empty. There are various ways to use this approach, including double hashing, linear probing, and quadratic probing.
- **Separate Chaining:** In separate chaining, a linked list of objects that hash to each slot in the hash table is present. Two keys are included in the linked list if they hash to the same slot. This method is rather simple to use and can manage several collisions.
- **Robin Hood hashing:** To reduce the length of the chain, collisions in Robin Hood hashing are addressed by switching off keys. The algorithm compares the distance between the slot and the occupied slot of the two keys if a new key hashes to an already-occupied slot. The existing key gets swapped out with the new one if it is closer to its ideal slot. This brings the existing key closer to its ideal slot. This method has a tendency to cut down on collisions and average chain length.

Dynamic resizing:

This feature enables the hash table to expand or contract in response to changes in the number of elements contained in the table. This promotes a load factor that is ideal and quick lookup times.

Types of Hash Functions

1. Division Method.
2. Multiplication Method
3. Mid-Square Method
4. Folding Method
5. Cryptographic Hash Functions
6. Universal Hashing
7. Perfect Hashing

1. Division Method

The division method involves dividing the key by a prime number and using the remainder as the hash value.

$h(k)=k \bmod m$

Where k is the key and mm is a prime number.

Advantages:

- Simple to implement.
- Works well when mm is a prime number.

Disadvantages:

- Poor distribution if mm is not chosen wisely.

2.Multiplication Method

In the multiplication method, a constant AA ($0 < A < 1$) is used to multiply the key. The fractional part of the product is then multiplied by mm to get the hash value.

$h(k)=\lfloor m(kA \bmod 1) \rfloor$

Where $\lfloor \rfloor$ denotes the floor function.

Advantages:

- Less sensitive to the choice of mm.

Disadvantages:

- More complex than the division method.

3. Mid-Square Method

In the mid-square method, the key is squared, and the middle digits of the result are taken as the hash value.

Steps:

1. Square the key.
2. Extract the middle digits of the squared value.

Advantages:

- Produces a good distribution of hash values.

Disadvantages:

- May require more computational effort.

4. Folding Method

The folding method involves dividing the key into equal parts, summing the parts, and then taking the modulo with respect to *mm*.

Steps:

1. Divide the key into parts.
2. Sum the parts.
3. Take the modulo *mm* of the sum.

Advantages:

- Simple and easy to implement.

Disadvantages:

- Depends on the choice of partitioning scheme.

Operations on Hash Tables:

1. Insertion (adding key-value pairs)
2. Deletion (removing key-value pairs)
3. Search (finding a key-value pair)

1. Insertion

- **Operation**: Add a key-value pair to the hash table.
- **Process**:
    - Compute the hash code of the key using a hash function.
    - Map the hash code to an index in the underlying array.
    - Insert the key-value pair at that index.
    - Handle collisions using techniques like chaining (linked lists) or open addressing (probing).

2. Searching

- **Operation**: Retrieve the value associated with a given key.
- **Process**:
    - Compute the hash code of the key.
    - Map the hash code to an index in the array.
    - Check that index for the key. In case of collisions, search through the entries at that index depending on the collision resolution method used.

3. Deletion

- **Operation**: Remove a key-value pair from the hash table.
- **Process**:
    - Compute the hash code of the key to find the index.
    - Check the entry at that index for the key.
    - If the key is found, remove the associated value. Again, handle chains or probing accordingly.

Conclusion:

- Hashing maps keys to indices.
- Hash tables store key-value pairs.
- Hash functions generate indices.

Sample Program

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

#define TABLE_SIZE 10

// Node structure for chaining
typedef struct Node {
    char *key;
    int value;
    struct Node *next;
} Node;

// Hash table structure
typedef struct HashTable {
    Node **table;
} HashTable;

// Hash function
unsigned int hash_function(const char *key) {
    unsigned int hash = 0;
    while (*key) {
        hash = (hash << 5) + *key; // shift left 5 bits and add the current character
        key++;
    }
    return hash % TABLE_SIZE; // Return a value in the size of the table
}

// Create a new hash table
```

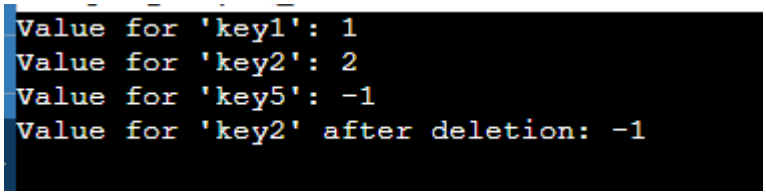
```
HashTable* create_table() {
    HashTable *hashtable = malloc(sizeof(HashTable));
    hashtable->table = malloc(TABLE_SIZE * sizeof(Node*));
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashtable->table[i] = NULL;
    }
    return hashtable;
}

// Create a new node
Node* create_node(const char *key, int value) {
    Node *new_node = malloc(sizeof(Node));
    new_node->key = strdup(key);
    new_node->value = value;
    new_node->next = NULL;
    return new_node;
}

// Insert a key-value pair into the hash table
void insert(HashTable *hashtable, const char *key, int value) {
    unsigned int index = hash_function(key);
    Node *new_node = create_node(key, value);

    if (hashtable->table[index] == NULL) {
        hashtable->table[index] = new_node;
    } else {
        Node *current = hashtable->table[index];
        while (current->next != NULL) {
            if (strcmp(current->key, key) == 0) {
                current->value = value;  // Update if key already exists
```

```
            free(new_node->key);
            free(new_node);
            return;
        }
        current = current->next;
    }
    current->next = new_node; // Insert at end of linked list
  }
}

// Search for a key in the hash table
int search(HashTable *hashtable, const char *key) {
  unsigned int index = hash_function(key);
  Node *current = hashtable->table[index];

  while (current != NULL) {
    if (strcmp(current->key, key) == 0) {
      return current->value; // Return value if key found
    }
    current = current->next;
  }

  return -1; // Return -1 if key not found
}

// Delete a key from the hash table
void delete(HashTable *hashtable, const char *key) {
  unsigned int index = hash_function(key);
  Node *current = hashtable->table[index];
  Node *prev = NULL;
```

```
    while (current != NULL) {
        if (strcmp(current->key, key) == 0) {
            if (prev == NULL) {
                hashtable->table[index] = current->next; // Remove the first node
            } else {
                prev->next = current->next; // Bypass the current node
            }
            free(current->key);
            free(current);
            return;
        }
        prev = current;
        current = current->next;
    }
}

// Free the hash table
void free_table(HashTable *hashtable) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        Node *current = hashtable->table[i];
        while (current != NULL) {
            Node *temp = current;
            current = current->next;
            free(temp->key);
            free(temp);
        }
    }
    free(hashtable->table);
    free(hashtable);
```

```
}

// Main function to demonstrate the hash table
int main() {
    HashTable *hashtable = create_table();

    insert(hashtable, "key1", 1);
    insert(hashtable, "key2", 2);
    insert(hashtable, "key3", 3);
    insert(hashtable, "key4", 4);

    printf("Value for 'key1': %d\n", search(hashtable, "key1"));
    printf("Value for 'key2': %d\n", search(hashtable, "key2"));
    printf("Value for 'key5': %d\n", search(hashtable, "key5")); // Key not found

    delete(hashtable, "key2");
    printf("Value for 'key2' after deletion: %d\n", search(hashtable, "key2"));

    free_table(hashtable);
    return 0;
}
```

Output-



Experiment No-7

Aim:

Write code and understand the concept Search Algorithms (Linear Search, Binary Search.)

Theory:

Search Algorithm-

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data.

This collection of data can take various forms, such as arrays, lists, trees, or other structured representations.

These algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various applications such as databases, web search engines, and more.

The primary objective of searching is to determine whether the desired element exists within the data, and if so, to identify its precise location or retrieve it.

It plays an important role in various computational tasks and real-world applications, including information retrieval, data analysis, decision-making processes, and more.

Importance of Searching in DS

- **Efficiency:** Efficient searching algorithms improve program performance.
- **Data Retrieval:** Quickly find and retrieve specific data from large datasets.
- **Database Systems:** Enables fast querying of databases.
- **Problem Solving:** Used in a wide range of problem-solving tasks.

Applications of Searching:

Searching algorithms have numerous applications across various fields. Here are some common applications:

- **Information Retrieval:** Search engines like Google, Bing, and Yahoo use sophisticated searching algorithms to retrieve relevant information from vast amounts of data on the web.
- **Database Systems:** Searching is fundamental in database systems for retrieving specific data records based on user queries, improving efficiency in data retrieval.
- **E-commerce:** Searching is crucial in e-commerce platforms for users to find products quickly based on their preferences, specifications, or keywords.

- **Networking:** In networking, searching algorithms are used for routing packets efficiently through networks, finding optimal paths, and managing network resources.
- **Artificial Intelligence:** Searching algorithms play a vital role in AI applications, such as problem-solving, game playing (e.g., chess), and decision-making processes
- **Pattern Recognition:** Searching algorithms are used in pattern matching tasks, such as image recognition, speech recognition, and handwriting recognition.

Characteristics of Searching

Understanding the characteristics of searching in data structures and algorithms is crucial for designing efficient algorithms and making informed decisions about which searching technique to employ. Here, we explore key aspects and characteristics associated with searching:

1. Target Element:

In searching, there is always a specific target element or item that you want to find within the data collection. This target could be a value, a record, a key, or any other data entity of interest.

2. Search Space:

The search space refers to the entire collection of data within which you are looking for the target element. Depending on the data structure used, the search space may vary in size and organization.

3. Complexity:

Searching can have different levels of complexity depending on the data structure and the algorithm used. The complexity is often measured in terms of time and space requirements.

4. Deterministic vs non-deterministic:

Some searching algorithms, like binary search, are deterministic, meaning they follow a clear and systematic approach. Others, such as linear search, are non-deterministic, as they may need to examine the entire search space in the worst case.

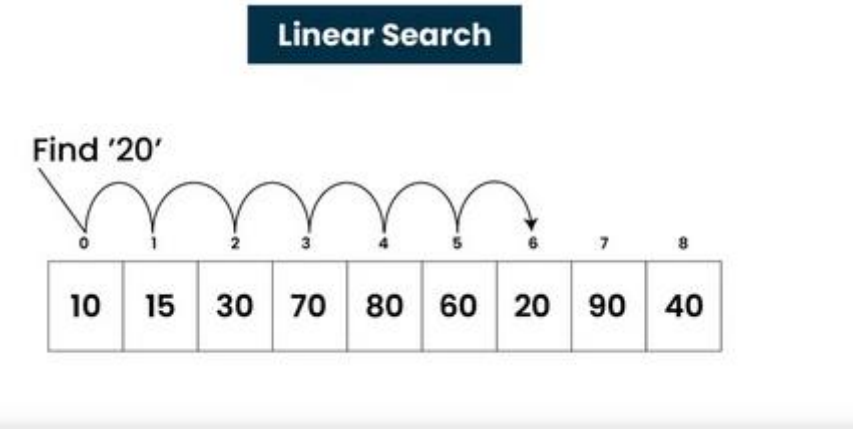
Below are some searching algorithms:

1. Linear Search
2. Binary Search
3. Ternary Search
4. Jump Search
5. Interpolation Search
6. Fibonacci Search
7. Exponential Search

1)Linear Search

Linear search is a simple search algorithm that checks each element in a list until it finds the target value.

Linear Search, also known as Sequential Search, is one of the simplest and most straightforward searching algorithms. It works by sequentially examining each element in a collection of data(array or list) until a match is found or the entire collection has been traversed.



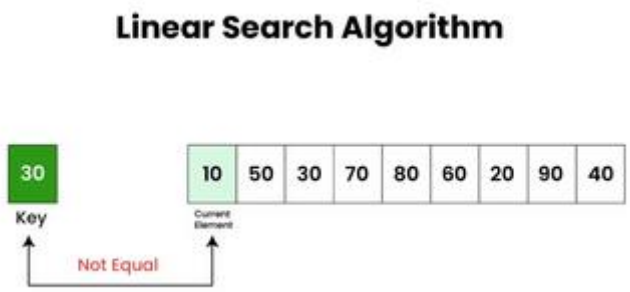
Algorithm of Linear Search:

- The Algorithm examines each element, one by one, in the collection, treating each element as a potential match for the key you're searching for.
- If it finds any element that is exactly the same as the key you're looking for, the search is successful, and it returns the index of key.
- If it goes through all the elements and none of them matches the key, then that means "No match is Found".

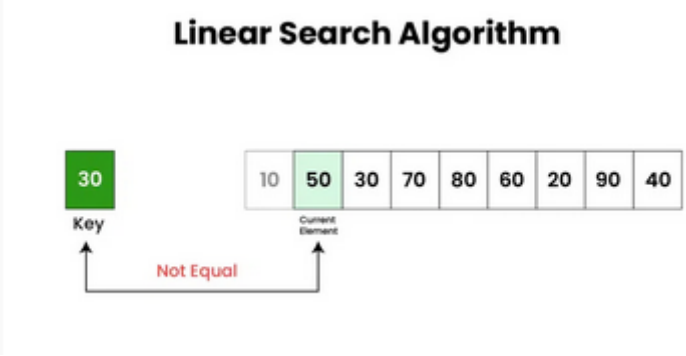
Illustration of Linear Search:

Consider the array **arr[] = {10, 50, 30, 70, 80, 20, 90, 40}** and **key = 30**

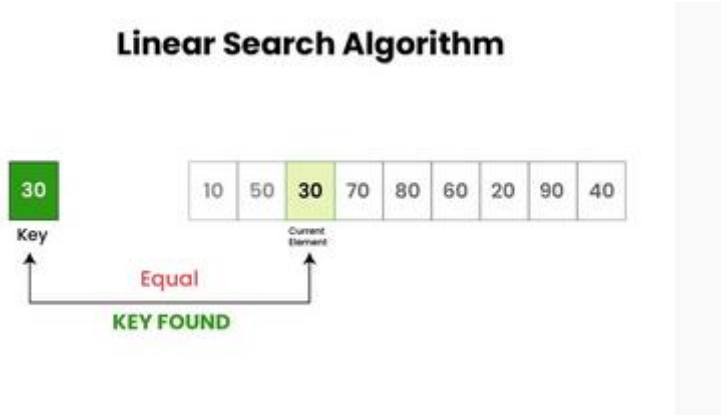
Start from the first element (index 0) and compare key with each element (arr[i]). Comparing key with first element arr[0]. Since not equal, the iterator moves to the next element as a potential match.



Comparing key with next element arr[1]. Since not equal, the iterator moves to the next element as a potential match.



Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found.



Pseudo Code for Linear Search:

Linear Search (collection, key):

for each element in collection:

if element is equal to key:

return the index of the element

return "Not found"

Complexity Analysis of Linear Search:

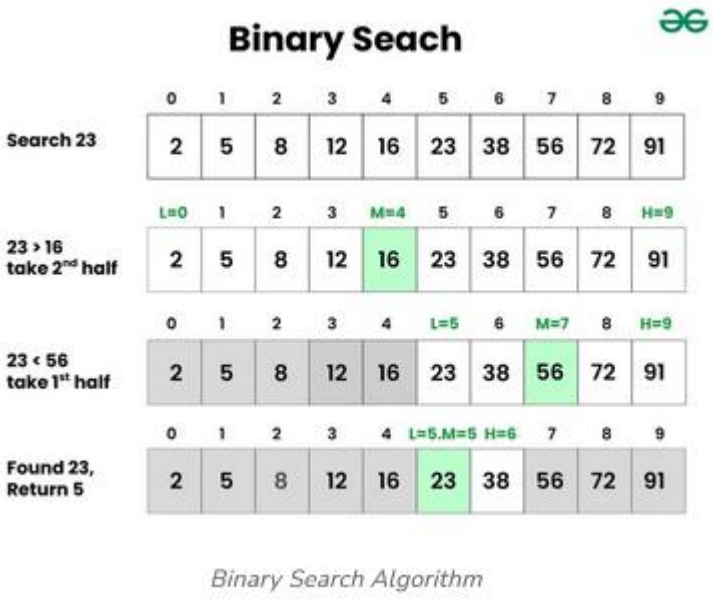
- **Time Complexity:**
    - **Best Case**: In the best case, the key might be present at the first index. So the best case complexity is O(1)
    - **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
    - **Average Case:** O(N)
- **Auxiliary Space:** O(1) as except for the variable to iterate through the list, no other variable is used.

When to use Linear Search:

- When there is small collection of data.
- When data is unordered.

2)Binary Search:

Binary Search is defined as a searching algorithm used in a **sorted array** by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to **O(log N).**



Binary Search Algorithm

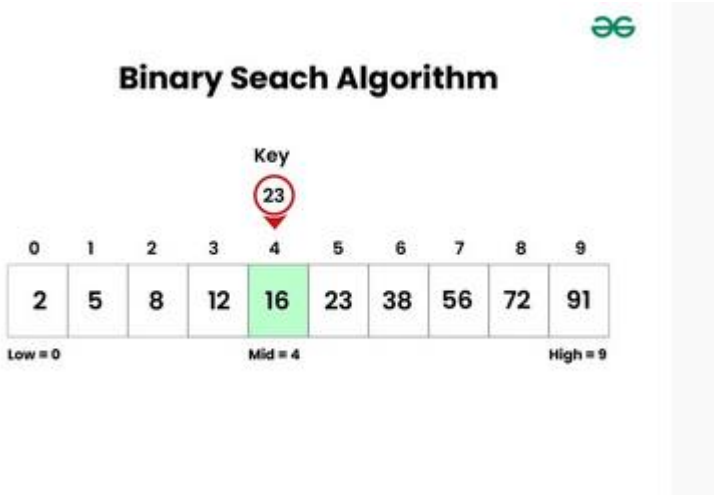
Algorithm of Binary Search:

- Divide the search space into two halves by finding the middle index "**mid**".
- Compare the middle element of the search space with the **key**.
- If the **key** is found at middle element, the process is terminated.
- If the **key** is not found at middle element, choose which half will be used as the next search space.
    - If the key is smaller than the middle element, then the **left** side is used for next search.
    - If the key is larger than the middle element, then the **right** side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

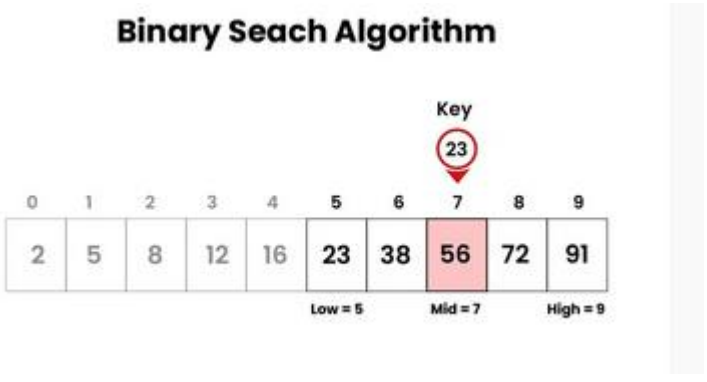
Illustration of Binary Search:

Consider an array **arr[]** = **{2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target** = **23**.

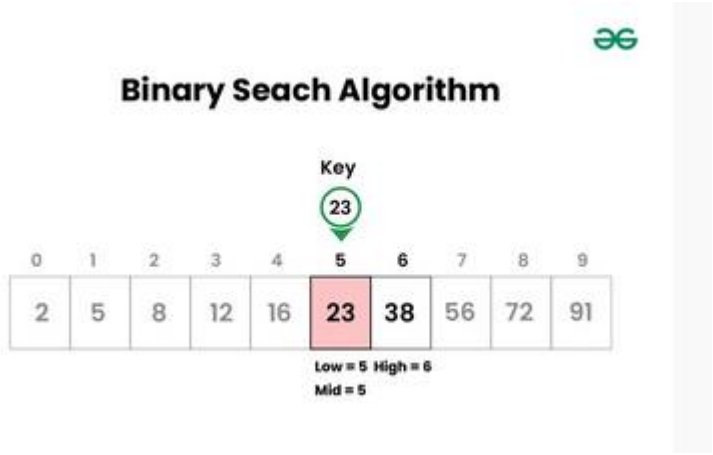
- Calculate the **mid** and compare the mid element with the key. If the key is less than **mid** element, move to **left** and if it is greater than the **mid** then move search space to the **right**.
- Key (i.e., 23) is greater than current **mid** element (i.e., 16). The search space moves to the **right**.



- **Key** is less than the current mid **56**. The search space moves to the **left**.



If the **key** matches the value of the mid element, the element is found and stop search.



Pseudo Code for Binary Search:

Below is the pseudo code for implementing binary search:

binarySearch(collection, key):

left = 0

right = length(collection) – 1

while left <= right:

mid = (left + right) // 2

if collection[mid] == key:

return mid

elif collection[mid] < key:

left = mid + 1

else:

right = mid – 1

return “Not found”

Complexity Analysis of Binary Search:

- **Time Complexity:**
    - **Best Case:** O(1) – When the key is found at the middle element.
    - **Worst Case:** O(log N) – When the key is not present, and the search space is continuously halved.
    - **Average Case:** O(log N)
- **Auxiliary Space**: O(1)

When to use Binary Search:

- When the data collection is monotonic (essential condition) in nature.
- When efficiency is required, specially in case of large datasets.

Conclusion:

Search algorithms are essential in data structures.

- Linear search checks each element.

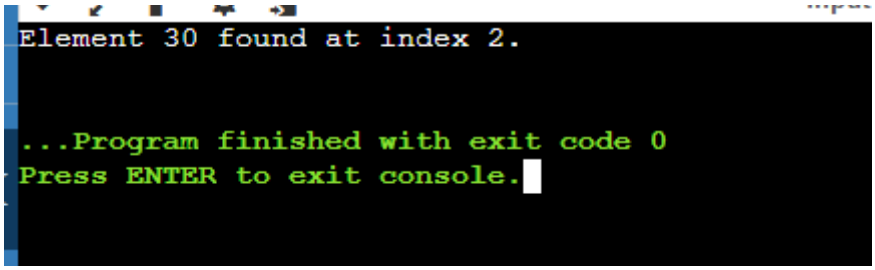
- Binary search divides the list in half.

Sample Program (Linear Search)

```
#include <stdio.h>
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;  // Return the index if element is found
        }
    }
    return -1;  // Return -1 if element is not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};  // Example array
    int n = sizeof(arr) / sizeof(arr[0]);  // Number of elements in the array
    int target = 30;  // Element to search for

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

    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
```

```
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Output

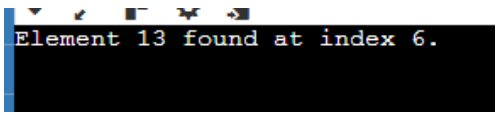


Sample Program (Binary search)

```
#include <stdio.h>

// Function to perform binary search
int binarySearch(int arr[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;  // Calculate the middle index

        // Check if the target is at the mid position
        if (arr[mid] == target) {
            return mid;  // Element found, return its index
        }

        // If the target is smaller than mid, ignore the right half
        if (arr[mid] > target) {
            right = mid - 1;
        }
```

```
        // If the target is larger than mid, ignore the left half
        else {
            left = mid + 1;
        }
    }

    return -1;  // Element not found
}

int main() {
    // Example sorted array
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int n = sizeof(arr) / sizeof(arr[0]);  // Number of elements in the array
    int target = 13;  // Element to search for

    int result = binarySearch(arr, 0, n - 1, target);

    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Output



Experiment No-8

Aim:

Write code and understand the concept Sorting Algorithms (Bubble Sort, Insertion Sort)

Theory:

Sorting Algorithms

A Sorting Algorithm is used to rearrange a given array or list of elements in an order. Sorting is provided in library implementation of most of the programming languages

Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements.

The comparison operator is used to decide the new order of elements in the respective data structure.

The sorting algorithm is important in Computer Science because it reduces the complexity of a problem.

There is a wide range of applications for these algorithms, including searching algorithms, database algorithms, divide and conquer methods, and data structure algorithms.

In the following sections, we list some important scientific applications where sorting algorithms are used

- When you have hundreds of datasets you want to print, you might want to arrange them in some way.
- Once we get the data sorted, we can get the k-th smallest and k-th largest item in O(1) time.
- Searching any element in a huge data set becomes easy. We can use Binary search method for search if we have sorted data. So, Sorting become important here.
- They can be used in software and in conceptual problems to solve more advanced problems.

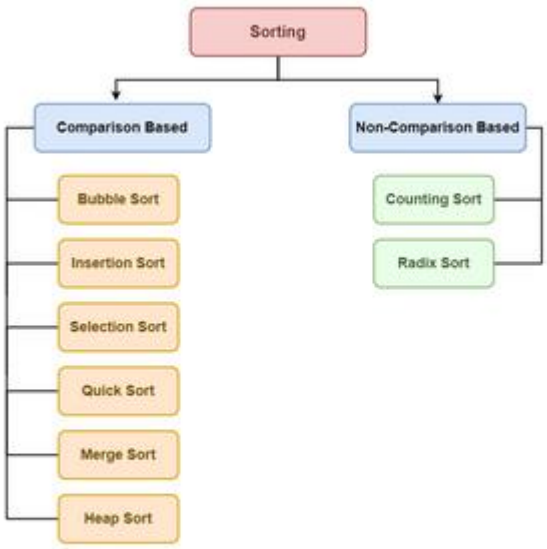
Sorting Basics:

- **In-place Sorting:** An in-place sorting algorithm uses **constant space** for producing the output (modifies the given array only. Examples: Selection Sort, Bubble Sort, Insertion Sort and Heap Sort.
- **Internal Sorting:** Internal Sorting is when all the data is placed in the **main memory** or **internal memory**. In internal sorting, the problem cannot take input beyond allocated memory size.
- **External Sorting :** External Sorting is when all the data that needs to be sorted need not to be placed in memory at a time, the sorting is called external sorting. External Sorting is used for the massive amount of data. For example Merge sort can be used in external sorting as the whole array does not have to be present all the time in memory,
- **Stable sorting:** When two same items appear in the same order in sorted data as in the original array called stable sort. Examples: Merge Sort, Insertion Sort, Bubble Sort.
- **Hybrid Sorting:** A sorting algorithm is called Hybrid if it uses more than one standard sorting algorithms to sort the array. The idea is to take advantages of multiple sorting algorithms. For example IntroSort uses Insertions sort and Quick Sort.

Types of Sorting Techniques:

There is various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
2. **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)



Sorting algorithm

1)Bubble Sort

In this Experiment, we will discuss the Bubble sort Algorithm. The working procedure of bubble sort is simplest. This article will be very helpful and interesting to students as they might face bubble sort as a question in their examinations. So, it is important to discuss the topic.

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets

Bubble short is majorly used where -

- complexity does not matter
- simple and shortcode is preferred

Algorithm

In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

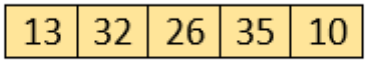
```
1. begin BubbleSort(arr)
2.    for all array elements
3.       if arr[i] > arr[i+1]
4.          swap(arr[i], arr[i+1])
5.       end if
6.    end for
7.    return arr
8. end BubbleSort
```

Example

Now, let's see the working of Bubble sort Algorithm.

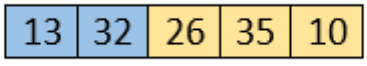
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **O(n^2).**

Let the elements of array are -

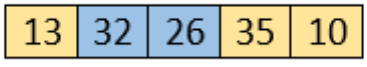


First Pass

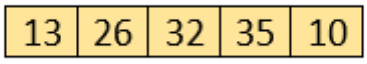
Sorting will start from the initial two elements. Let compare them to check which is greater.



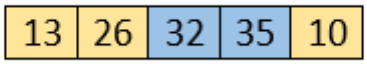
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

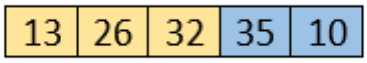


Now, compare 32 and 35.

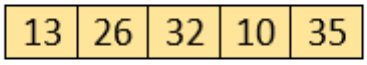


Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



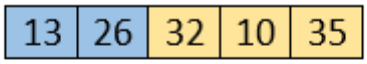
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

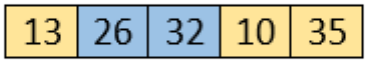


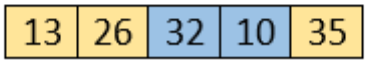
Now, move to the second iteration.

Second Pass

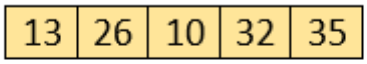
The same process will be followed for second iteration.

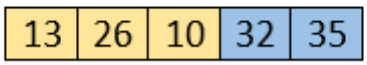






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

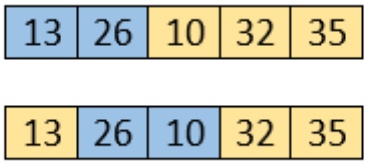




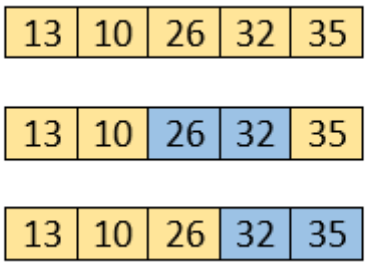
Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.



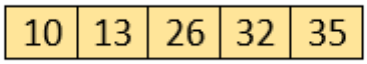
Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



Now, move to the fourth iteration.

Fourth Pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity:

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1. Time Complexity

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

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is O(n).
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is $O(n^2)$.
    - **Worst case complexity-**Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is $O(n^2)$.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.
- The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

Now, let's see the algorithm of selection sort.

Advantages of Bubble Sort

1. Easily understandable.
2. Does not necessitate any extra memory.
3. The code can be written easily for this algorithm.
4. Minimal space requirement than that of other sorting algorithms.

Disadvantages of Bubble Sort

1. It does not work well when we have large unsorted lists, and it necessitates more resources that end up taking so much of time.
2. It is only meant for academic purposes, not for practical implementations.
3. It involves the n^2 order of steps to sort an algorithm.

2)Insertion sort:

In this Experiment, we will discuss the Insertion sort Algorithm. The working procedure of insertion sort is also simple. This article will be very helpful and interesting to students as they might face insertion sort as a question in their examinations. So, it is important to discuss the topic.

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **$O(n^2)$**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

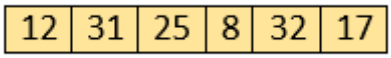
Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

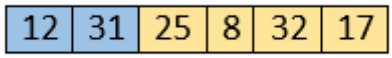
Step 6 - Repeat until the array is sorted.

Example

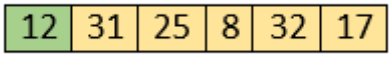
Let the elements of array are -



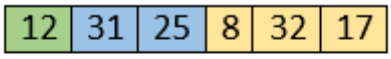
Initially, the first two elements are compared in insertion sort.

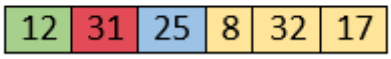


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



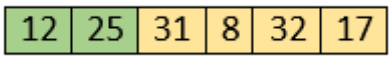
Now, move to the next two elements and compare them.



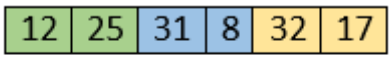


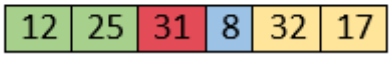
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

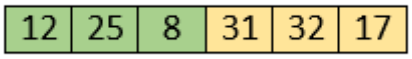


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

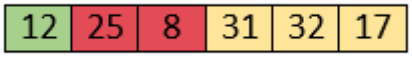




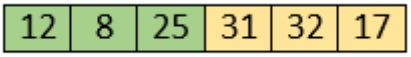
Both 31 and 8 are not sorted. So, swap them.



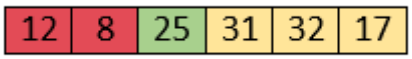
After swapping, elements 25 and 8 are unsorted.



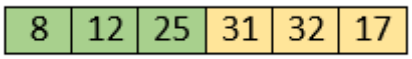
So, swap them.



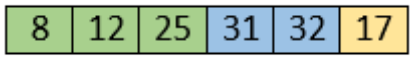
Now, elements 12 and 8 are unsorted.



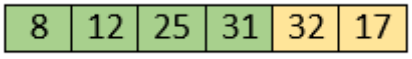
So, swap them too.



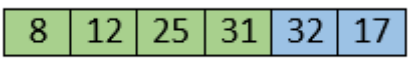
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



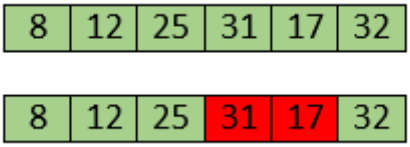
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



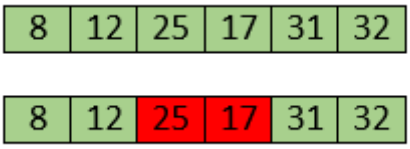
Move to the next elements that are 32 and 17.



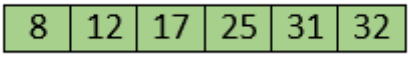
17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

1)Time Complexity

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

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is O(n).

- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is $O(n^2)$.
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is $O(n^2)$.

2)Space Complexity

Space Complexity
O(1)
Stable
YES

Advantages Of Insertion sort

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Disadvantages Of Insertion sort

- Inefficient for large data sets
- Requires more writes
- Worst-case time complexity

Conclusion:

Sorting algorithms are essential in data structures.

- Bubble sort compares adjacent elements.

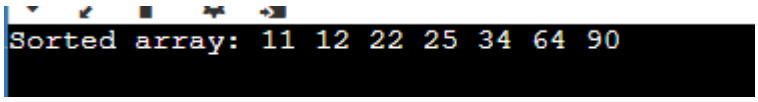
- Selection sort compare data and sort them

Sample Program

(Bubble sort)

```
#include <stdio.h>
void bubble_sort(int arr[], int n) {
  int i, j;
  for (i = 0; i < n - 1; i++) {
   for (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;
      }
    }
  }
}
int main() {
  int arr[] = {64, 34, 25, 12, 22, 11, 90};
  int n = sizeof(arr) / sizeof(arr[0]);
  bubble_sort(arr, n);
  printf("Sorted array: ");
  for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }
  return 0;
}
```

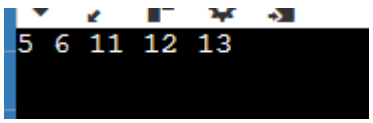
Output



Insertion sort

```
#include <stdio.h>
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);
    insertionSort(arr, n);
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Output



Experiment No-9

Aim:

Write code and understand the concept Algorithm Technique on Greedy Approach

Theory:

Greedy Approach

The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results. Let's understand through some terms.

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favourable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions

Components of Greedy Algorithm:

The components that can be used in the greedy algorithm are:

- **Candidate set:** A solution that is created from the set is known as a candidate set.
- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
- **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- **Objective function:** A function is used to assign the value to the solution or the partial solution.
- **Solution function:** This function is used to intimate whether the complete function has been reached or not.

Applications of Greedy Algorithm:

1. We use Greedy Algorithms in our day to day life to find minimum number of coins or notes for a given amount. We fist begin with largest denomination and try to use maximum number of the largest and then second largest and so on.

2. **Dijkstra's shortest path algorithm:** Finds the shortest path between two nodes in a graph.
3. **Kruskal's and Prim's minimum spanning tree algorithm:** Finds the minimum spanning tree for a weighted graph. Minimum Spanning Trees are used in Computer Networks Designs and have many real world applications
4. **Huffman coding:** Creates an optimal prefix code for a set of symbols based on their frequencies.
5. **Fractional knapsack problem:** Determines the most valuable items to carry in a knapsack with a limited weight capacity.
6. **Activity selection problem:** Chooses the maximum number of non-overlapping activities from a set of activities.
7. Job Sequencing and Job Scheduling Problems.
8. Finding close to the optimal solution for NP-Hard problems like TSP. ide range of network design problems, such as routing, resource allocation, and capacity planning.
9. **Machine learning**: Greedy algorithms can be used in machine learning applications, such as feature selection, clustering, and classification. In feature selection, greedy algorithms are used to select a subset of features that are most relevant to a given problem. In clustering and classification, greedy algorithms can be used to optimize the selection of clusters or classes
10. **Image processing**: Greedy algorithms can be used to solve a wide range of image processing problems, such as image compression, denoising, and segmentation. For example, Huffman coding is a greedy algorithm that can be used to compress digital images by efficiently encoding the most frequent pixels.

Key Characteristics:

- Greedy algorithms are simple and easy to implement.
- They are efficient in terms of time complexity, often providing quick solutions. Greedy Algorithms are typically preferred over Dynamic Programming for the problems where both are applied. For example, Jump Game problem and Single Source Shortest Path Problem (Dijkstra is preferred over Bellman Ford where we do not have negative weights)..
- These algorithms do not reconsider previous choices, as they make decisions based on current information without looking ahead.

Advantages of Greedy Algorithms

- **Simple and easy to understand:** Greedy algorithms are often straightforward to implement and reason about.
- **Efficient for certain problems:** They can provide optimal solutions for specific problems, like finding the shortest path in a graph with non-negative edge weights.
- **Fast execution time:** Greedy algorithms generally have lower time complexity compared to other algorithms for certain problems.
- **Intuitive and easy to explain:** The decision-making process in a greedy algorithm is often easy to understand and justify.
- **Can be used as building blocks for more complex algorithms:** Greedy algorithms can be combined with other techniques to design more sophisticated algorithms for challenging problems.

Disadvantages of the Greedy Approach:

- **Not always optimal:** Greedy algorithms prioritize local optima over global optima, leading to suboptimal solutions in some cases.
- **Difficult to prove optimality:** Proving the optimality of a greedy algorithm can be challenging, requiring careful analysis.
- **Sensitive to input order:** The order of input data can affect the solution generated by a greedy algorithm.
- **Limited applicability:** Greedy algorithms are not suitable for all problems and may not be applicable to problems with complex constraints.

Fractional Knapsack Problem Using a Greedy Algorithm

Given the weights and profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get the maximum total profit in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

Input: arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50
Output: 240
Explanation: By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.
Hence total price will be 60+100+(2/3)(120) = 240

Input: arr[] = {{500, 30}}, W = 10
Output: 166.667

Conclusion:

The Greedy Approach is a useful algorithm technique for solving optimization problems.

- Makes locally optimal choices.
- Hopes for global optimality.
- No backtracking.

Sample Program

Fractional Knapsack Problem Using a Greedy Algorithm

```
#include <stdio.h>
struct Item {
    int value;
    int weight;
    float ratio;
};

// Function to compare two items based on their value-to-weight ratio
int compare(const void* a, const void* b) {
    struct Item* item1 = (struct Item*)a;
    struct Item* item2 = (struct Item*)b;

    if (item1->ratio < item2->ratio) {
        return 1;
    } else if (item1->ratio > item2->ratio) {
        return -1;
    }
    return 0;
}
```

```
// Function to solve the Fractional Knapsack problem
float fractionalKnapsack(int W, struct Item items[], int n) {
    // Sort items based on value-to-weight ratio in descending order
    qsort(items, n, sizeof(struct Item), compare);

    int currentWeight = 0;  // Current weight of the knapsack
    float totalValue = 0.0; // Total value in the knapsack

    // Loop through all items
    for (int i = 0; i < n; i++) {
        // If adding the full item doesn't exceed capacity, take the full item
        if (currentWeight + items[i].weight <= W) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        } else {
            // If we can't take the full item, take the fraction of it
            int remainingWeight = W - currentWeight;
            totalValue += items[i].value * ((float)remainingWeight / items[i].weight);
            break; // Knapsack is full
        }
    }

    return totalValue;
}

int main() {
    int n, W;

    // Input number of items and the capacity of the knapsack
    printf("Enter number of items: ");
```

```
    scanf("%d", &n);

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &W);

    struct Item items[n];

    // Input values and weights of the items
    printf("Enter value and weight for each item (value weight):\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &items[i].value, &items[i].weight);
        // Calculate value-to-weight ratio
        items[i].ratio = (float)items[i].value / items[i].weight;
    }

    // Calculate maximum value achievable
    float maxValue = fractionalKnapsack(W, items, n);

    printf("Maximum value in Knapsack = %.2f\n", maxValue);

    return 0;
}
```

Output