DATA STRUCTURE & ALGORITHM, DATABASE SYTEM AND OPERATING SYSTEM (ACtE07)

7.1 INTRODUCTION TO DATA STRUCTURE, LIST, LINKED LISTS AND TREES

Introduction to Data Structure:

Data structure is a way of organizing and storing data in a computer for efficient access and manipulation.

- Data Structure: Efficient way of storing and organizing data in a computer.
- Based on computer's ability to fetch and store data at any memory address.
- Structural representation of logical relationship between data elements.
- Defined as triple (D, F, A):
  D: Data
  F: Set of functions
  A: Set of axioms.

Data Structure is the representation of logical relationship existing between individual elements of data. The data structure is a way of organizing all the data items that considers not only the elements but also their relationship to each other.

Algorithm + Data Structure = Program

Permitted Data values + Operations = Data Type

Organized Data + Allowed Operations = Data Structure

Classification of Data Structure

Depending upon the arrangement of data, data structures can be classified as arrays, records, link lists, stack, queues, trees, graphs etc.

1. Primitive and Non-Primitive Data Structure
2. According to the Nature of Size
3. According to its occurrence

4. Homogeneous and Non-Homogeneous

1. **Primitive and Non-Primitive Data Structure**

Primitive Data Structure:

- Basic data structure that can be directly operated by machine instruction.
- Representation of primitive data structure is different on different computers.
- Examples: character, integer, floating point numbers, strings, etc.

Non-Primitive Data Structure:

- Complex data structure derived from the basic data structure.
- Emphasis on structuring homogeneous and heterogeneous data items.
- Examples: arrays, linked lists, stacks, queues, trees, graphs, etc.

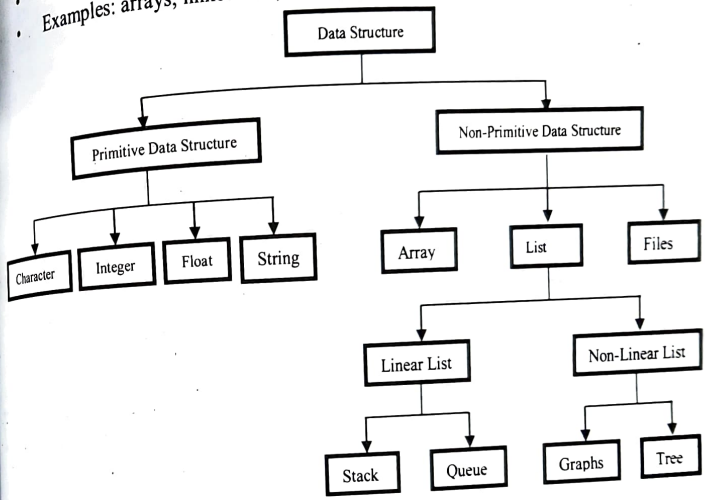


Figure: Classification of Data Structure

2. **According to the nature of size**

Static data structure: The size of data items (data elements) are fixed in static data structure. For example: array

Dynamic data structure: The size of data items (data elements) are not fixed, can be change during the execution of the program in dynamic data structure. For example: linked list, tree, graph etc.

3. **Homogeneous and Non-Homogeneous**

- **Homogeneous:** The data items of homogeneous data structure are of same type. For example, array.
- **Non-homogeneous:** The data items of Non-homogeneous data structure are different.

4. **According to its occurrence**

Linear data structure: In linear data structure the data elements are stored in a consecutive memory location or data are stored in a sequential manner. For example: array, stack, queue etc.

Non-linear data structure: In non-linear data structure the data elements are not stored in a consecutive memory location or stored in non-sequential manner. These data structures are used to represent the hierarchical relationship between data elements. For example: tree, graph etc.

Linear data structures:

- Arrays
- Stacks
- Linked lists
- Queues

Non-linear data structures:

- Trees
- Hash tables
- Graphs
- Heaps

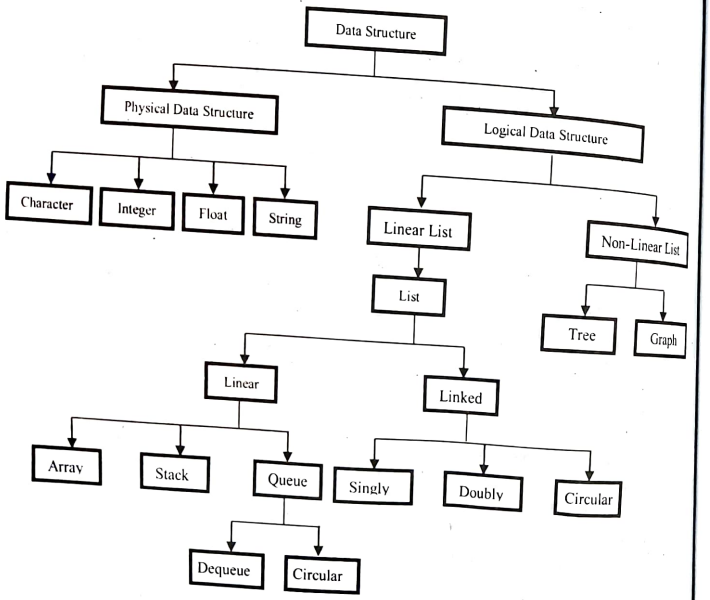


Figure: Classification on the basis of occurrence

Operations on Data Structure

1. **Creation**: Process of Creating a data structure
2. **Insertion**: Process of inserting an element to a data structure.
3. **Deletion**: Process of removing an element from a data structure.
4. **Traversing**: Process of visiting each and every element (data items) of a data structure.
5. **Searching**: Process of finding a data items in a data structure.
6. **Sorting:** The process of arranging data items either in ascending or descending order
7. **Concatenation**: Process of appending two data structure.
8. **Display**: Process of showing (displaying) data items of a data structure.
9. **Destroying:** This operation is applying at the last of data structure, when the data structure is no more needed. It is a process of collapse of a Data Structure.

LIST:

List: Data structure of ordered elements (nodes) in linear order.

Implemented using arrays or linked lists.

Array-based: Elements stored in contiguous memory, accessed by index.

Linked: Separate nodes linked by pointers, head node points to first element, last node has null pointer.

Efficient insertion/deletion at arbitrary positions, constant-time access to first last elements. Access time to other elements depends on implementation.

LINKED LIST:

Linked List: Ordered collection of homogeneous data elements (nodes) with linear order maintained by links/pointers.

Each node has:

a) Data element information

b) Link field (next pointer) containing address of next node in list.

Representation of Linear Linked List

```
struct node
{
  int data;
  struct node *pnext;
};
struct node *pfirst, *pthis, *pnew, ;
```

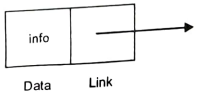


Figure: Node of a linked list

Manisha

Terminology Used in Linked List

a. Data Field

The data field contains an actual value to be stored and processed.

b. Link Field

The link field contains the address of the next data item in the linked list. The address used to access a particular node is known as a pointer.

c. Null Pointer

The link field of last node contains NULL rather than a valid address. The Null Pointer indicates the end of list.

d. External Pointer

External pointer points to a very first node of a linked list. It enables to access the whole linked list.

e. Empty List

If the nodes are not present in the list then it is called empty linked list or simply empty list. A linked list can be made an empty list by assigning NULL value to the external pointer.

Advantages

- Size can grow/shrink during program execution
- Efficient memory utilization (allocation/deallocation)
- Easy node insertion/deletion
- Flexible insertion at specified position, efficient deletion
- Reduced access time, expandable in real time without memory overhead.

Disadvantages:

a) Uses more memory than arrays (requires 2 fields per data item).

b) Access to arbitrary data item is cumbersome and time-consuming.

Operations on Linked List

a) Creation: Creating a linked list.

b) Insertion: Adding a new node at specified position.

c) Deletion: Removing a node.

d) Traversing: Visiting every node in list (forward/backward).

e) Searching: Finding desired node in list.

f) Concatenation: Appending one list to another.

g) Display: Showing/displaying nodes.

Various Data Structure

1. Array
2. Stack
3. Queue
4. Linked List
5. Tree
6. Graph
7. Sorting
8. Searching

1. Array

The simplest data structure is array data structure. Array is a collection of homogeneous elements, in the form of index, value pairs and store in a consecutive memory location. The size of an array is defined at the designing time or committed time.

char name[7] = {'A', 'B', 'C', 'D', 'E'};

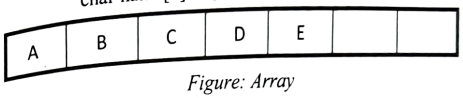


Figure: Array

2. Stack

Stack is a linear data structure in which the insertion and deletion of data element is done from only one end called the Top of the Stack (TOS). Stack works on the principle of LIFO (Last in First Out).

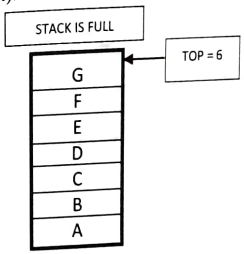


Figure: Stack

3. Queue

Queue is a linear data structure in which the data elements are inserted from one end called rear of a queue and deleted from another end called front of a queue. The Queue works on the principle FIFO (First in First Out).

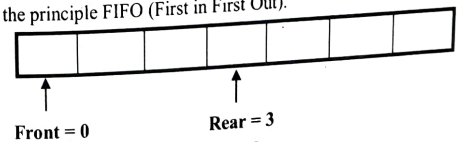


Figure: Queue

4. Linked List

Lists are the most commonly used non-primitive data structures. List can be defined as a collection of variable number of data items. An element of a list is called node and each

node of a list contains at least two fields one for storing actual information and another field is used for storing address of next element (use pointer).

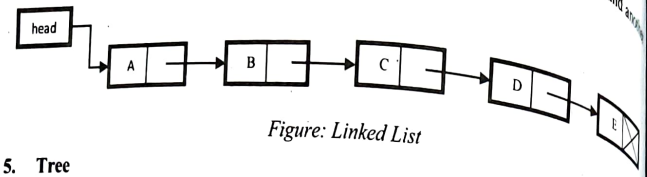


Figure: Linked List

5. Tree

Tree is a nonlinear data structures in which data elements are arranged or stored in a sorted order. A tree is a finite set of data elements (nodes). The data elements in a tree are represent the hierarchical relationship between various elements.

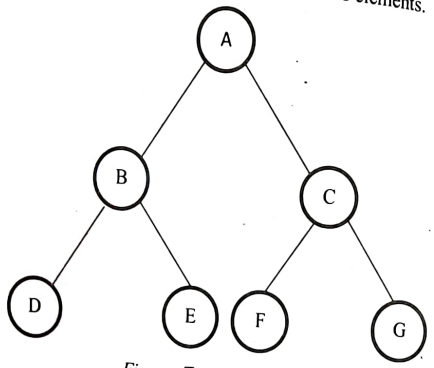


Figure: Tree Structure

6. Graph

Graph is a nonlinear data structure used to represent many kinds of physical structures. Its application is in Computer network, Engineering Science, Mathematics, Chemistry etc.

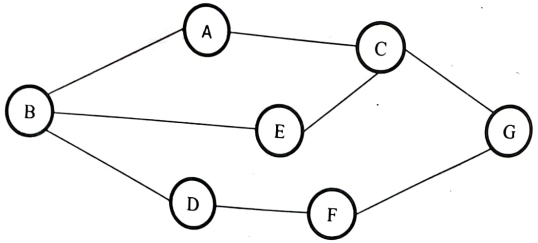


Figure: Graph Structure

7. Sorting

A sorting is a process of arranging data-elements in some particular order. The order may be either in ascending or descending or some priority order.

10	20	30	40	50	60	70

Figure: data sorted in ascending order

8. Searching

A Searching is a process of identifying or finding a data element from a set of data with a help of ***key***. Where key may be either internal key or external key.

Types of Linked List

1. Singly Linked List
2. Circular Singly Linked List
3. Doubly Linked List
4. Circular Doubly Linked List

TREE:

Tree is non-linear data structure in which data elements are arranged in a sorted sequence. It is used to represent the hierarchical relationships existing among several data elements.

Data Type:

Data types in data structures define the type of values that a particular data structure element can hold.

- Integer: whole numbers such as 1, 2, 3, etc.
- Character: a single character such as 'A', 'B', 'C', etc.
- Float: floating-point numbers such as 1.23, 2.34, 3.45, etc.
- Boolean: values that can be either true or false.

Abstract Data Type:

ADT: Abstract Data Type, mathematical model for a data structure defining values and operations.

Defines values and operations without specifying memory representation.

Stack ADT: Collection of elements with "push" and "pop" operations, operates on Last-In-First-Out (LIFO) principle.

Here are some key notes on ADT:

a) Independence: Describes behavior and properties of data structures without implementation details.
b) Common ADTs: Stack, queue, linked list, tree, graph, map/dictionary, set, hash table.
c) Importance: Crucial for designing and analyzing algorithms using data structures.
d) Implementation: Can be done using various data structures (arrays, linked lists, trees, hash tables) based on requirements and trade-offs.
e) Operations: Adding elements, removing elements, searching, checking if empty, specific to each ADT.

Time and space analysis of algorithms (Big oh, omega and theta notations):

Measure of efficiency of an algorithm, based on time taken to complete as a function of input size.

Importance: Determines scalability and helps choose appropriate algorithm.

There are typically three scenarios of Complexity analysis:

a) Best case: Least amount of time required to execute.

b) Worst case: Maximum amount of time required to execute.

c) Average case: Average amount of time required to execute.

Algorithm complexity analysis help compare cost associated with each algorithm/code.

Space Complexity:

Measure of resource usage of an algorithm, based on memory required to complete as a function of input size.

Importance: Determines feasibility and helps choose appropriate algorithm.

Asymptotic Analysis

- Simplest way to describe running time of an algorithm.
- Represents efficiency and performance in systematic manner.
- Theoretical analysis of algorithm.
- Used in asymptotic analysis.
- Asymptotic analysis is the theoretical analysis of an algorithm. The following notations are used for asymptotic analysis:
  a) Big Oh(O) notation
  b) Omega (Ω) notation
  c) Theta (θ) notation

Big Oh (O) – notation (Upper Bound):

- Expresses upper bound of running time of an algorithm.
- Denoted by 'O'.
- Computes maximum possible amount of time for completion.
- f(n) <= Cg(n) for positive constant C and integer n0, where n > n0.

Here, f(n) = O g((n))

Big Omega (Ω) – notation (Lower Bound)

- Gives lower bound of a function within a constant factor.
- Written as f(n) = Ω(g(n)).
- Positive constant n0 and C exist such that f(n) >= Cg(n) for n > n0.

Big Theta (θ) – notation (Tightly Bound)

- Theta notation, denoted by θ, computes average completion time of an algorithm by expressing running time between lower and upper bounds.

- It's defined as f(n) = θ(g(n)) where f(n) and g(n) are positive functions of n (input size) and C1 and C2 are positive constants with C1 g(n) ≤ f(n) ≤ C2 g(n).

Linear data structure (Stack and queue implementation):

Linear Data Structure:

Linear data structures are structures that store elements in a linear sequence, like arrays, linked lists, stacks, and queues.

Array:

- Arrays are linear data structures that store elements of the same data type in a contiguous block of memory. They are simple and efficient for accessing elements by index.

Linked List:

- A linked list is a linear data structure that consists of a collection of nodes, where each node has a reference to the next node in the list.
- Linked lists are dynamic in nature, meaning they can grow or shrink in size during runtime.

Stack:

- Stack is a linear data structure with LIFO (Last-In-First-Out) property
- Push operation for inserting elements [Top=Top+1]
- Pop operation for removing elements [Top=Top-1]
- Underflow occurs when there are no elements in the stack
- Overflow occurs when the stack is full
- Peek operation returns the topmost element
- Is Empty operation checks if the stack is empty [Top=-1]
- Is Stack Full operation checks if the stack is full [Top=MAXSIZE-1]
- Real-life examples: deck of cards, piles of books, stacks of money, etc.

Stack Implementation:

a) Static Implementation/array/contiguous

Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (i.e., increased or decreased).

The array implementation requires the following

- an array
- a variable top to hold the top most element of the stack.

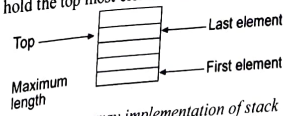


Figure: an array implementation of stack

The array implementation has the following disadvantages.

- Memory space is wasted (because of storing number of items in the array than maximum size of array)
- There is limitation in storing the items into the stack.
- Insertion and deletion operation in the stack using an array involves more data movements.

b) Dynamic implementation (using linked list):

- Insertion and deletion operations can be performed easily in linked list implementation of a stack.
- Memory utilization is efficient as memory is allocated dynamically in linked list implementation.
- No need to pre-allocate memory as in arrays.
- Easily scalable as compared to arrays.
- Overcoming the size limitations of arrays.

Advantage:

- Stack does not need to be of fixed size
- Size can grow or shrink during program execution
- Linked list does not waste memory space
- Insertion and deletion operations are efficient
- Data can be moved efficiently, allowing for efficient rearrangement of data items

Disadvantages:

- Linked list uses more storage than array
- Access to arbitrary data item is cumbersome and time consuming

Stack Applications:

a) It is very useful to evaluate arithmetic expression (postfix expression)

- Infix-→a+b (operator in between operand)
- Prefix→ +ab (operator before operand)
- Postfix→ab+ (operator after operand)
- Where '+' symbol is called operator and a and b are called the operand.

b) Infix to postfix transformation

c) Function calls in C/C++

d) Validity of arithmetic expressions

e) Recursive programs

f) Compiler design in operating system

g) String reversal

Infix to postfix conversion:

The algorithm for converting an infix expression to a postfix expression is as follows:

1. Initialize an empty stack
2. Scan infix expression from left to right
3. If current character is operand, add to output string
4. If current character is operator, pop operators with lower precedence and add to output string
5. If current character is left parenthesis, push onto stack
6. If current character is right parenthesis, pop operators and add to output string until left parenthesis is found
7. Repeat steps 3 to 6 for each character in infix expression
8. Pop any remaining operators from the stack and add to output string
9. The postfix expression is now complete.

Convert infix to postfix expression a$b*c – d + e/f/ (g+h)

Solution,

a$b*c – d + e/f/ (g+h)	
= a$b*c-d+e/f/(g+h)	infix form
=a$b*c-d+e/f/(gh+)	convert the parenthesized operation
= (ab$)*c-d+e/f/(gh+)	convert the exponentiation.
= (ab$c*)-d+e/f/(gh+)	convert the multiplication.
= (ab$c*)-d+ (ef/)/(gh+)	convert the division (left most).
= (ab$c*)-d+ (ef/gh+/)	convert the division
= (ab$c*d-) + (ef/gh+/)	convert the subtraction.
= (ab$c*d-ef/gh+/) +	convert the addition.
=ab$c*d-ef/gh+/+	postfix form

Evaluation of postfix expression:

Algorithm:

1. Declare all necessary variables. Initialize an empty stack called result stack.
2. Read a character in postfix expression if a character is an operand, push it into a stack. else Pop two operand from result stack and store in operand 1 and operand 2. Evaluate applying the expression opnd1 operator opnd2
3. Repeat step (2) until the end of postfix expression
4. Pop the element in result stack.
5. Stop

Example 2: 7 5 3 2 ^ * 9 2 2 ^ - / + 6 4 * + (Evaluate the postfix Expression)

Symbol 1	Action	Stack
7	Push 7	7
5	Push 5	7,5
3	Push 3	7,5,3
2	Push 2	7,5,3,2
^	Pop 2 and 3, push 3^2	7,5,9
*	Pop 9 and 5, push 5*9	7,45
9	Push 9	7,45,9
2	Push 2	7,45,9,2
2	Push 2	7,45,9,2,2
^	Pop 2 and 2, push 2^2	7,45,9,4
-	Pop 4 and 9, push 9-4	7,45,5
/	Pop 5 and 45, push 45/5	7,9
+	Pop 9 and 7, push 7+9	16
6	Push 6	16,6
4	Push 4	16,6,4
*	Pop 4 and 6, push 6*4	16,24
+	Pop 24 and 16, push 16+24	40

Order of precedence (PEMDAS)

a) Parenthesis ( )
b) Exponential (^ or ↑, $)
c) Multiplication (*, x)
d) Division (/or ÷)
e) Addition (+)
f) Subtraction (–)

QUEUE:

A queue is a non-primitive linear data structure. It is a homogeneous collection of elements in which new elements are added at one end called the Rear end and the existing elements are deleted from the other end called the front of the queue.

Queue Related Terms

Rear: A variable indicating the end where new data will be added (in the queue).

Front: It is a variable indicating the end from where the data will be retrieved.

- The end from which items are deleted is known as **front** of the queue.
- Since, first inserted item is removed at first and last inserted item is removed at last. That is why queue is called **FIFO (First In First Out)** data structure.

- Insertion process is called **enqueue** operation. When we insert an element in the queue, the value of rear is incremented by one: ***Rear = Rear+1***
- Deletion process is called **dequeue** operation. When we delete/remove an element from the queue the value of front/head is incremented by one: ***Front = front +1***
- The process of adding an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.
- The following figure shows queue graphically during insertion and deletion operation

Implementation of Queue

Queue can be implemented in two different ways

a) **Static Implementation (Using an array):**

The array implementation of queues involves allocation of fixed size array in the memory. Both queue operations (insert and delete) are performed on this array with a constant check being made to ensure that the array does not go out of bounds.

Advantages:

- The implementation of a Queue using an array is very simple and easy.

Disadvantages:

- It is possible to store only a fixed number of elements in the queue.
- Wastage of memory space can occur:
- For example, if the size of the queue is 15 but the number of elements stored in the queue may be just 5 then the space allocated for the 10 elements will be wastage.

b) **Dynamic implementation (Using Linked List or Pointer):**

The linked list implementation of queues is based on dynamic memory allocation technique, which allows allocation and de-allocation of memory space at run time.

Advantages:

- There is no wastage of memory space.
- Insertion and deletion operation are easier to perform.
- Size of the queue is not fixed. That means any number of elements can be placed in a queue dynamically. The size of a Queue can be grow or Shrink at the execution mode.

Disadvantages:

- Some extra pointer fields are used along with each data item in the queue.
- They take extra memory space.

Operations on a queue:

The operations that can be performed on a queue are:

1. Insertion Operation
2. Deletion Operation
3. Is empty
4. Isfull

Insertion / Enqueue Operation

- An element can be inserted into the queue only when there is some space in the queue or only when the queue is not full.
- Before inserting an element into the queue, check whether the queue is full on not.
- If the Queue is not full, insert the elements into the Queue otherwise insertion is not possible.
- The insertion operation can be specified as follow:
- Insert (q, element)
- Where q = is the queue in which insertion operation has to be performed and 'element' is the data or item that has to be inserted at the rear end of the queue.

C function for insertion operation in Queue:

```
queue is declared as
queue [5], front= -1 and rear = -1;
void queue( )
{
    int item ;
    if(rear<4 )
{   printf( "enter the number");
    scanf( "%d", &item);
    if(front== -1)
    {
        front = 0;
        rear= 0;
    }
    else
{
        rear= rear +1;
}
queue [rear] = item;
}
else
printf("queue is full");
}
```

Deletion / Dequeue / Removal operation

- An element can be removed from the Queue only when the Queue is not empty.
- Before removing an element from the Queue check whether the queue is empty (or) not.
- If the Queue is not empty then remove operation can be done at the front end of the queue, otherwise the remove operation cannot be performed.
- The remove operation can be specified as follows:

- item = remove (q)
- Where, q is the queue from which the remove operation is done and 'item' is the variable in which the removed item is stored.

C function for Deletion/Dequeue operation in Queue:

```
void delete ( )
{
    int item;
    if(front != -1)
    {
        item= queue[front];
        if(front==rear)
        {
            front= -1;
            rear= -1;
        }
        else
            front= front +1;
    printf(" no deleted is =%d", item);
}
    else
        printf(" queue is empty");
}
```

Is empty operation

- The empty condition of the Queue can be identified by executing this operation. The empty condition can be specified as follows.
- bv = Is empty (q)
- Where, q is the queue name whose empty condition has to be verified.
- This empty condition can either return a true or false value.
- If the queue is empty, TRUE is returned otherwise it returns FALSE.
- The return value will get stored in the variable bv.

```
isempty functions
int isempty (rear,front)
{
  if (rear == -1 && front == -1)
    return 1;
  else
    return 0;
}
```

Is full operation

- The full condition of the queue can be identified by performing the IS full operation.
- It's general from is
- Fv = ISFull (q).
- Where 'q' is the queue name whose full condition is verified.
- This operation can return either a TRUE or FALSE value.
- If the queue is FULL, TRUE is returned otherwise it returns FALSE value.
- Here, the returned value is stored in the variable Fv.

```
is full functions
int isfull (int front)
{
  if (front == MAXSIZE - 1)
return 1;
  else
return 0;
}
```

Circular Queue: It is not other than linear queue where the unused space of array is reused.

a) Data Addition:

- Check if the queue is not full
- Rear = (Rear + 1) % Q-Size
- Insert the data at the rear position

b) Data Deletion:

- Check if the queue is not empty
- Retrieve the data at the head position
- Head = (Head + 1) % Q-Size

Application of circular queue:

- Used in real-time systems, efficient memory utilization important.

Types of Queues:

a) Linear Queue:

- FIFO (First-In, First-Out)
- Elements added at the back, removed from the front
- Time complexity: O(1) for insertion, O(1) for deletion

b) Circular Queue:

- Last position connected to first, forming a circular buffer
- Efficient memory utilization
- Time complexity: O(1) for insertion, O(1) for deletion

c) Priority Queue:

- Elements removed based on priority, not order added
- Time complexity: O(log n) for insertion, O(1) for deletion (with a binary heap)

d) Double Ended Queue (Deque):

- Elements can be added/removed from front and back
- Time complexity: O(1) for insertion, O(1) for deletion

Array implementation of lists:

Lists are linear data structures where elements are stored in a consecutive manner and can be accessed sequentially, from the first item (head) to the last (tail).

Basic Operations on a List

- Creating a list: Allocating memory
- Traversing: Accessing elements
- Inserting: Adding element at position
- Deleting: Removing element
- Concatenating: Combining two lists into one.

Here is an example of array implementation of a list in C:

```
#define MAX_SIZE 100

int list[MAX_SIZE];
int length = 0; // number of elements in the list

// Insert an element into the list at a specified index
void insert(int item, int index) {
if (index < 0 || index > length) {
printf("Invalid index\n");
return;
}
if (length == MAX_SIZE) {
printf("List is full\n");
return;
}
for (int i = length; i > index; i--) {
list[i] = list[i-1];
}
list[index] = item;
length++;
```

```
}

// Delete an element from the list at a specified index
void delete(int index) {
if (index < 0 || index >= length) {
printf("Invalid index\n");
return;
}
for (int i = index; i < length-1; i++) {
list[i] = list[i+1];
}
length--;
}

// Print the elements of the list
void printList() {
for (int i = 0; i < length; i++) {
printf("%d ", list[i]);
}
printf("\n");
}
```

In this example, the list is implemented as an array with a fixed size of 100. The length variable keeps track of the number of elements in the list, and the insert and delete functions allow for adding and removing elements at a specified index. The printList function displays the elements of the list.

Stack and Queues as list:

A stack can be implemented as a list with two main operations: push and pop. The push operation adds an element to the top of the stack, and the pop operation removes the element at the top of the stack. The top of the stack is represented by the end of the list.

Here is a short example of a stack as a list in C:

```
#define MAX_SIZE 100
int stack[MAX_SIZE];
int top = -1; // index of the top element in the stack
// Push an element to the top of the stack
void push(int item) {
if (top == MAX_SIZE-1) {
printf("Stack overflow\n");
```



```
return;
}
stack[++top] = item;
}
// Pop an element from the top of the stack
int pop() {
if (top == -1) {
printf("Stack underflow\n");
return -1;
}
return stack[top--];
}
```

Queue as a List in C:

A queue can be implemented as a list with two main operations: enqueue and dequeue. The enqueue operation adds an element to the back of the queue, and the dequeue operation removes the element at the front of the queue.

Here is a short example of a queue as a list in C:

```
#define MAX_SIZE 100
int queue[MAX_SIZE];
int front = -1, rear = -1; // indices of the front and rear elements in the queue
// Enqueue an element to the back of the queue
void enqueue(int item) {
if (rear == MAX_SIZE-1) {
printf("Queue overflow\n");
return;
}
queue[++rear] = item;
}

// Dequeue an element from the front of the queue
int dequeue() {
if (front == rear) {
printf("Queue underflow\n");
return -1;
}
return queue[++front];
}
```

Static and dynamic list structure:

- Static list structure: fixed size at creation, memory allocated at compile-time (example: array)
- Dynamic list structure: size can change dynamically, memory allocated at run-time (example: linked list)

Linked List:

- Linked list: Collection of nodes with linear order maintained by links/pointers
- Nodes: Two parts, one contains element information and other contains address of next node.

Types of Linked List with Time Complexity:

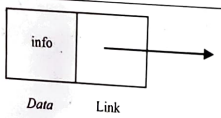
a) Singly Linked List:
   - Each node has a single link to the next node
   - Time complexity: O(n) for traversal, O(1) for insertion, O(1) for deletion

b) Doubly Linked List:
   - Each node has two links, one to the previous node and one to the next node
   - Time complexity: O(n) for traversal, O(1) for insertion, O(1) for deletion

c) Circular Linked List:
   - Last node points to the first node, forming a circular loop
   - Time complexity: O(n) for traversal, O(1) for insertion, O(1) for deletion

Representation of Linear Linked List

```
struct node
{
    int data;
    struct node *pnext;
};
struct node *pfirst, *pthis, *pnew, ;
```



Terminology Used in Linked List

a) Data Field
- The data field contains an actual value to be stored and processed.

b) Link Field
- The link field contains the address of the next data item in the linked list. The address used to access a particular node is known as a pointer.

Null Pointer
- The link field of last node contains NULL rather than a valid address. The Null Pointer indicates the end of list.

External Pointer
- External pointer points to a very first node of a linked list. It enables to access the whole linked list.

Empty List
- If the nodes are not present in the list then it is called empty linked list or simply empty list. A linked list can be made an empty list by assigning NULL value to the external pointer.

Advantages
- Linked list are dynamic data structure:
- Efficient Memory utilization:
- Node insertion and deletion operations are very easy and simple:

Disadvantages
- They use more memory than array. To store a single data at least two fields are required.
- Access to an arbitrary data item is little bit cumbersome and also time consuming.

Dynamic Implementation of Linked List:
- Dynamic allocation: Memory for each node can change dynamically
- Nodes and pointers: Each node contains data and pointer to next node, last node has null pointer
- Adding node: Allocate memory, update pointers to link new node into list
- Removing node: Update pointers to bypass node, free memory
- Flexibility: Dynamic implementation allows for flexible manipulation without fixed memory allocation.

Types of Linked List

a) Singly Linked List

Singly linked list is one in which all nodes are linked together in some sequential manner. Hence it is a Linear Linked List.

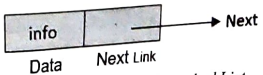


Figure: Node of Singly Linked List

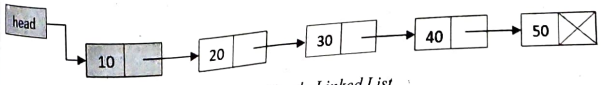


Figure: Singly Linked List

Node Insertion Operation in Singly Linked List

- Node insert from the beginning
- Node insert from the end
- Node insert before Xth position
- Node insert after Xth position

b) Circular Singly Linked List:

A circular linked list is one which has no beginning and no end. In circular singly linked list the address field of last node of a list points to the first node.

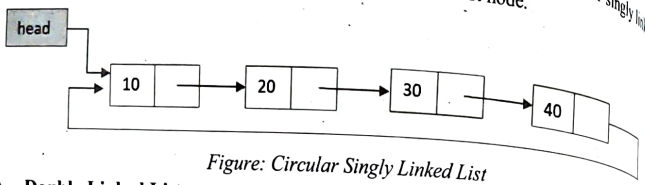


Figure: Circular Singly Linked List

c) Doubly Linked List:

A doubly linked list is a type of linked list where each node has two links, one pointing to the next node and another pointing to the previous node, allowing for bi-directional traversal.

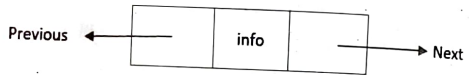


Figure: Node of a doubly linked List

```
struct node
{
    int info;
    struct node *previous, *next;
};
struct node *pfirst = null, *plast = null, *pthis;
```

d) Circular Doubly Linked List

Circular doubly linked list is a doubly linked list in which the next address field of a last node points to the first node and the previous address field of first node points to the last node of a linked list.

The list shown in the figure is a doubly linked circular list.

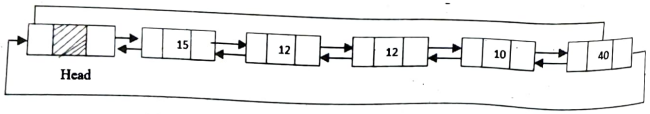


Figure: Circular doubly linked list with header node

Like in Linear Doubly Linked list in circular doubly linked list we can perform Insertion and Deletion Operation

Insertion Operation

- Node insertion from the beginning
- Node insertion from the end
- Node insertion before Xth Node
- Node insertion after Xth Node

Concept of Tree:

- Tree: Hierarchical data structure with nodes connected by edges
- Nodes: Multiple child nodes, root node is topmost, leaf nodes have no children
- Uses: Represent hierarchical relationships, such as in file systems or org charts.
- Trees: Non-linear data structure with branching data elements
- Operations: Efficient searching, insertion, deletion
- Applications: Algorithms such as searching, sorting, traversal
- Types: Binary trees, AVL trees, B-trees, etc.

Tree Terminologies

• Node	• 8. Non-Terminal Node
• Parent	• 9. Siblings
• Child	• 10. Level
• Root	• 11. Edge or Brach
• Degree of a node	• 12. Path
• Degree of a Tree	• 13. Depth
• Terminal Nodes or Leaf Node	• 14. Forest

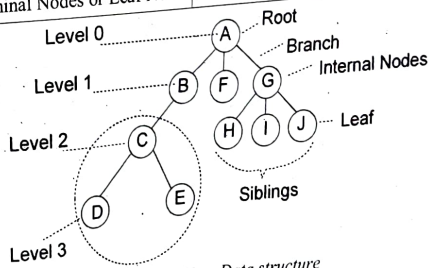


Figure: A Tree Data structure

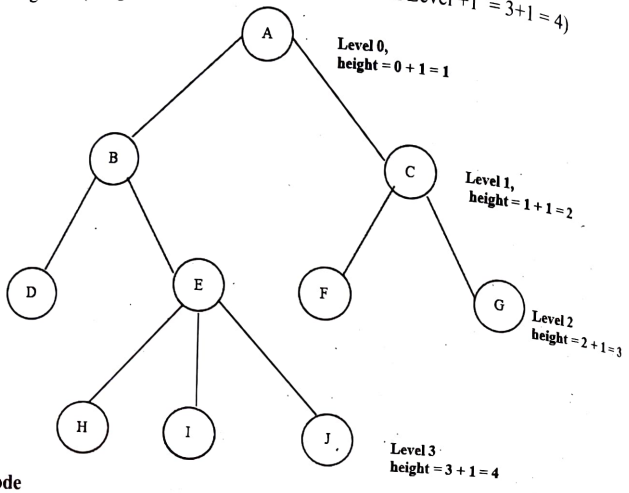
Parents : A, B, C, G

Children : B, F, G, C, H, I, J, D, E

Siblings : {B, F, G}, {D, E}, {H, I, J}

Leaves : F,D,E,H,I,J

Length : 4 (Length or height of the tree = Maximum Level +1 = 3+1 = 4)



- **Node**

  Each data elements in a tree is called a node. It is a basic and main important component of any tree structure. It stores the actual information along with the links to other nodes. In above figure each circle representing a node.

- **Parent Node**

  The parent node is the immediate predecessor node of that node. For example, node B is a parent node of node D and node E. Similarly, node A is a parent node of node B and node C etc.

- **Child Node**

  The immediate Successor nodes of a tree are called child nodes. The child node placed at the left is known as **Left Child** and the child node placed at right side is known as **Right Child** node of a Tree. In above figure node B and node C are the left child and right child node of node A.

- **Root**

  The Root node is a special node of a tree. It is the first node of a tree and it has no parent node. There can be only one root node in a tree. In the figure above node A is a root node.

- **Degree of a node**

  The degree of a node is defined as the maximum number of child that a node has. For example, in above figure the node E has three children and the other non-terminal node

has two children. So, the degree of node E is 3, the degree of other non-terminal node has 2 and the degree of leaf node has a degree of a node 0.

- ***Degree of a Tree***

  The degree of a tree is defined as the maximum number of degree of a node in that tree. For example, in above tree the maximum degree of a node is 3. So, the degree of a tree is also 3.

- ***Terminal node / Leaf nodes***

  The terminal nodes are those nodes that don't have any children. Terminal nodes are also called the Leaf nodes. In above figure node D, F, G, H, I, J are the leaf nodes.

- **Non-Terminal Nodes**

  The non-terminal nodes are those nodes which have at least one child nodes. In above figure node A, B, C and D are the non-terminal nodes.

- **Siblings**

  The child nodes of a given parent node are called *Siblings*. They are also called the brother of nodes. For example, in above figure node C is the Siblings of node B.

- **Level**

  The level of a node is the number of edges along the unique path between it and the root. The level of the root is zero. If the node is at level I then its child is at level I+1 and its parent is at level I-1. This rule is common for all nodes except the root node.

- **Edge**

  It is the connection line between two nodes. The line drawn from one node to another node is called edge.

- **Path**

  It is the sequence of consecutive edges from the source node to the destination node. In the above figure the path between node A and node H is given by the node pairs.

  (A, B), (B, E) and (E, H)

- **Depth**

  The maximum level of any node in a given tree. In the above tree, the root node A has a maximum level. The term **Height** is also used to denote the depth.

- **Forest**

  It is a set of disjoint trees. If you remove a root node in the above tree it becomes a forest. The forest with two trees.

Binary tree:

A binary tree is a type of tree data structure in which each node has at most two children. The children are referred to as the left child and the right child. Binary trees are used to represent hierarchical relationships where each node has only two children, such as in decision making processes where a choice is made based on certain conditions. Types of binary tree are:

a) Strictly Binary Tree: Every node has exactly two children, results in symmetrical and balanced tree

b) Complete Binary Tree: Every level filled except possibly last, results in balanced tree

c) Almost Complete Binary Tree: Similar to Complete Binary Tree but last level not necessarily filled, also balanced.

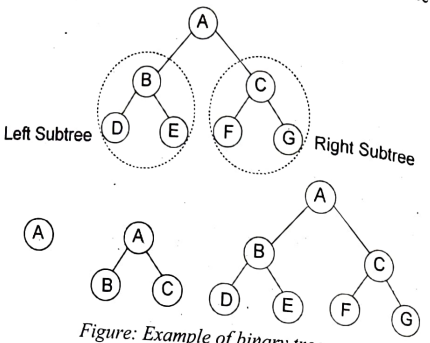


Figure: Example of binary tree

Binary Search Tree (BST):

- Type of binary tree data structure
- Each node has a value
- Left child has value less than parent node
- Right child has value greater than parent node
- Efficient for searching, insertion, deletion operations
- O(log n) time for search and delete operations.

The following are some of the common operations that can be performed on a Binary Search Tree (BST):

- Insertion: Adding a node based on its value
- Searching: Starting at root, comparing node value to search key, moving to left/right subtree as necessary
- Deletion: Finding node to delete, rearranging tree to maintain BST property.

Here is the algorithm for inserting a node in a Binary Search Tree (BST):

1. Start at the root node.
2. Compare the value of the new node with the value of the current node.
3. If the value of the new node is less than the value of the current node, move to the left child node.
4. If the value of the new node is greater than the value of the current node, move to the right child node.
5. Repeat steps 2-4 until an empty position is found.
6. Insert the new node at the empty position.
7. Repeat the process until all nodes have been inserted.

Here is the algorithm for searching a node in a BST:

1. Start at the root node.
2. Compare the value of the current node with the search key.
3. If the value of the current node is equal to the search key, the search is successful.
4. If the value of the current node is less than the search key, move to the right child node.
5. If the value of the current node is greater than the search key, move to the left child node.
6. Repeat steps 2-5 until either the search key is found or an empty position is encountered.

Here is the algorithm for deleting a node in a BST:

1. Start at the root node.
2. Search for the node to be deleted and keep track of its parent node.
3. If the node to be deleted has no children, simply delete the node.
4. If the node to be deleted has one child, delete the node and replace it with its child.
5. If the node to be deleted has two children, find the inorder successor of the node and replace the node to be deleted with the inorder successor.
6. Repeat the process until all nodes have been deleted.

Tree Traversal:

Tree traversal is the process of visiting each node in a tree data structure exactly once. The objective of tree traversal is to access and process the data elements stored in the nodes of the tree in a specific order

Types of Binary Tree Traversal

Tree traversal types:

- Pre-order: Visit root, then left subtree, then right subtree.
- In-order: Visit left subtree, then root, then right subtree.
- Post-order: Visit left subtree, then right subtree, then root.

Complexity: O(n) for all three, where n is the number of nodes in the tree.

AVL Tree ((Height Balanced Tree)

- The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows

- An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.
- Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor BF (T) of a node T is a binary tree is defined to be

Balanced Factor (N) = Height (Left Subtree (N)) – Height (Right Subtree (N))

AVL Tree Rotations

We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

- Rotation operations are used to make a tree balanced.
- Rotation is the process of moving the nodes to either left or right to make tree balanced.
- There are four rotations and they are classified into two types.
  1. Left Rotation
  2. Right Rotation
  3. Left Right Rotation
  4. Right left Rotation

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O (log n) time complexity. In AVL Tree, new node is always inserted as a leaf node.

The insertion operation is performed as follows

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the Balance Factor of every node.

Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

Complexity Analysis of AVL Tree and Binary Search Tree:

	AVL Tree		Binary Search Tree	
	Average	Worst	Average	Worst
Insertion	O(logn)	O(logn)	O(logn)	O(n)
Deletion	O(logn)	O(logn)	O(logn)	O(n)
Searching	O(logn)	O(logn)	O(logn)	O(n)
Traversal	O(n)	O(n)	O(n)	O(n)

7.2 SORTING, SEARCHING, AND GRAPHS

Sorting:

Sorting refers to the process of arranging a set of elements in a specific order.

- Internal sorting algorithms are designed to sort data that can fit entirely into memory. They work by rearranging the data within the same memory space. Here are some of the most common types of internal sorting algorithms:

a) **Insertion Sort:** Insertion sort is a simple sorting algorithm that works by sorting one element at a time, and inserting each element into its proper place in a sorted sub list. It is often used to sort small datasets, as it is easy to implement.

b) **Selection Sort:** Selection sort works by finding the smallest element in the list and swapping it with the first element. It then finds the second smallest element and swaps it with the second element, and so on, until the entire list is sorted.

c) **Bubble Sort**: Bubble sort works by comparing adjacent pairs of elements in the list and swapping them if they are in the wrong order. It continues to make passes over the list until the list is sorted.

d) **Quick Sort**: Quick sort is a divide-and-conquer algorithm that works by selecting a pivot element, partitioning the list around the pivot, and then recursively sorting the resulting sub lists.

e) **Merge Sort**: Merge sort works by dividing the list into two halves, sorting each half, and then merging the two sorted halves back together.

f) **Heap Sort**: Heap sort is based on the heap data structure, which is a binary tree where each node is greater than or equal to its children. The algorithm works by building a heap from the input data, and then repeatedly removing the largest element and inserting it into the sorted portion of the list.

g) **Shell Sort**: Shell sort is based on insertion sort and works by comparing elements that are farther apart in a list first, and then moving closer together. This makes it more efficient than simple insertion sort for large datasets.

- **External sorting**: is a technique used to sort data that is too large to fit entirely into memory. It works by dividing the data into smaller pieces that can be sorted separately, and then merging the sorted pieces together. External sorting is used in applications such as database management and external storage devices. The most common external sorting algorithms are External Merge Sort and Distribution Sort.

Efficiency of Sorting Algorithm:

The efficiency of a sorting algorithm is typically measured in terms of its time complexity and space complexity.

- Measured by speed & memory usage
- Factors: data size, type, algorithm used.

a) **Time Complexity**:

- Measures amount of time taken to run based on input size
- Can have different worst/best/average-case time complexity
- Usually expressed in Big O notation.

b) **Space Complexity:**

- Measures amount of memory required to run based on input size
- Can sort in place or require additional memory
- Usually expressed in Big O notation.

Stability: A sorting algorithm is stable if it maintains the relative order of equal elements in the input data. Stable algorithms are useful in applications where the original order of equal elements is important, such as sorting by multiple criteria.

Here are some of the most common sorting algorithms and their worst-case time and space complexities:

a) **Insertion Sort:** O(n^2) time complexity and O(1) space complexity. It is an efficient algorithm for small data sets.

b) **Selection Sort:** O(n^2) time complexity and O(1) space complexity. It is a simple algorithm to understand and implement.

c) **Bubble Sort**: O(n^2) time complexity and O(1) space complexity. It is an inefficient algorithm and is rarely used in practice.

d) **Quick Sort:** O(n^2) time complexity in the worst case and O(nlogn) time complexity in the average case, with O(logn) space complexity. It is a commonly used algorithm due to its efficiency on average and best-case inputs.

e) **Merge Sort:** O(nlogn) time complexity and O(n) space complexity. It is an efficient algorithm for large data sets and is widely used in practice.

f) **Heap Sort:** O(nlogn) time complexity and O(1) space complexity. It is an efficient algorithm for large data sets.

g) **Shell Sort:** O(n(logn)^2) time complexity and O(1) space complexity. It is an efficient algorithm for small data sets.

Sorting Algorithm Applications:

- Bubble sort: Small data sets, education
- Insertion sort: Small data sets, partially sorted data
- Selection sort: Small data sets, data with duplicate values
- Quick sort: Large data sets, randomly ordered data
- Merge sort: Large data sets, partially sorted data
- Heap sort: Efficient, data with duplicate values
- Radix sort: Efficient for data with a small number of possible values

Choice depends on data size, type, desired time/space complexity.

Sequential search, Binary search and Tree search:

Searching in refers to the process of finding the location or presence of a specific value or element within a collection of data.

Types of searching:

a) Sequential Search: Checks each element one by one, O(n) time complexity.

b) Binary Search: Uses sorted data, checks middle element and divides, O(log n) time complexity.

c) Tree Search: Uses tree structure, moves down the tree based on comparison, O(log n) average and O(n) worst case time complexity.

Choice depends on problem, data structure, and desired efficiency.

Hashing: Hash function and hash tables:

a) Hashing: Mapping elements to indices in hash table using hash function.

b) *Hash Function*: Calculates index for each element in hash table based on its value. *Aims to distribute elements evenly and minimize collisions.*

c) *Hash Table*: Data structure storing elements in array-like structure, with index calculated by hash function.

d) *Uses*: Database indexing, password validation, cache memory management, etc.

Some common types of hash functions include:

a) Division Method: h(x) = x mod m, where x is the element and m is the size of the hash table.

b) Folding Method: h(x) = ∑(parts of x), where the element is divided into parts and combined.

c) Mid Square Method: h(x) = middle digits of x^2, where x is the element.

d) Multiplication Method: h(x) = fractional part of (kx), where k is a constant and x is the element.

Collision resolution technique:

Collision Resolution in Hashing: Process of resolving conflicts when multiple elements are assigned the same hash value and mapped to the same index in a hash table.

Techniques:

a) Chaining: Linked list associated with each index in hash table.

b) Open addressing: Find next available slot in hash table.

There are several methods for finding the next available slot in open addressing, including:

- Linear probing: In this method, the algorithm checks the next index in the hash table until an empty slot is found. h(x, i) = (h'(x) + i) mod m, checks next index.
- Quadratic probing: In this method, the algorithm checks indices that are the result of a quadratic function of the number of probes. h(x, i) = (h'(x) + c1i + c2i^2) mod m, checks indices based on quadratic function of probes.
- Double hashing: In this method, the algorithm uses two hash functions to find the next available slot. h(x, i) = (h1(x) + ih2(x)) mod m, uses two hash functions.

c) Rehashing: Resize hash table and rehash elements to new indices.

Graph:

A graph is a non-linear data structure that represents a collection of vertices (also known as nodes) and edges.

Vertices: The vertices in a graph represent objects or entities, and they are connected by edges.

Edges: Edges represent the relationships or connections between the vertices. An edge can be directed (with an arrow) or undirected (without an arrow).

Types of Graph:

a) Null Graph: A graph with no vertices or edges.

b) Trivial Graph: A graph with only one vertex and no edges.

c) Non-directed Graph: A graph with edges that have no direction, also known as undirected graphs.
d) Directed Graph: A graph with edges that have a direction, also known as directed graphs or digraphs.
e) Connected Graph: A graph in which there exists a path between any two vertices.
f) Disconnected Graph: A graph in which there is no path between some vertices.
g) Regular Graph: A graph in which each vertex has the same number of edges.
h) Complete Graph: A graph in which there is an edge between every pair of vertices.
i) Cycle Graph: A graph in which there exists at least one cycle.
j) Acyclic Graph: A graph without any cycles, also known as a forest.
k) Finite Graph: A graph with a finite number of vertices and edges.
l) Infinite Graph: A graph with an infinite number of vertices and edges.
m) Bipartite Graph: A graph in which the vertices can be divided into two disjoint sets such that every edge connects a vertex from one set to a vertex in the other set.
n) Planar Graph: A graph that can be drawn on a plane without any edges crossing.
o) Simple Graph: A graph without self-loops or multiple edges between the same pair of vertices.
p) Multi Graph: A graph with multiple edges between the same pair of vertices.
q) Pseudo Graph: A graph with self-loops but without multiple edges between the same pair of vertices.
r) Euler Graph: A graph that has an Euler tour, which is a path that visits every edge exactly once.
s) Hamiltonian Graph: A graph that has a Hamiltonian cycle, which is a cycle that visits every vertex exactly once.

Representation of Graph:

In graph theory, representation of a graph refers to the method of encoding the structure of a graph into the memory of a computer for efficient storage and manipulation.

There are several ways to represent a graph, including:

a) Adjacency Matrix: An adjacency matrix is a two-dimensional matrix in which each row and column represent a vertex, and the entries in the matrix indicate the presence or absence of an edge between the vertices.
b) Adjacency List: An adjacency list is a collection of linked lists, where each linked list represents the vertices adjacent to a particular vertex.
c) Incidence Matrix: An incidence matrix is a two-dimensional matrix in which each row represents a vertex and each column represents an edge, and the entries in the matrix indicate the presence or absence of a connection between the vertex and the edge.

Transitive closure of graph:

a) Definition: The transitive closure of a directed graph represents the reachability between all vertex pairs (i, j) in the graph.

b) Reachability: The transitive closure determines if vertex j is reachable from vertex i for every pair of vertices in the graph.
c) Representation: The transitive closure is represented by a reachability matrix, indicating the existence of a path from vertex i to vertex j.
d) Algorithms: The transitive closure can be found using algorithms like Warshall's algorithm or Floyd-Warshall algorithm.
e) Applications: The transitive closure is useful in finding the shortest path, detecting cycles, and solving reachability problems.

Warshall's algorithm:

a) Purpose: To find the transitive closure of a graph.
b) Reachability Matrix: The algorithm updates a reachability matrix to represent the reachability between all pairs of vertices in the graph.
c) Iterative Updates: The reachability matrix is updated in a series of iterations, where each iteration updates the reachability between two vertices if there exists a path that passes through a third vertex.
d) Termination: The algorithm terminates when the reachability matrix reaches its final form, which represents the transitive closure of the graph.
e) Time Complexity: O(n^3), where n is the number of vertices in the graph.
f) Applications: Used in finding the shortest path, detecting cycles, and solving reachability problems in various domains such as computer networks, databases, and social networks.
g) Implementation: Simple to implement and widely used in practice.

Depth First Traversal and Breadth First Traversal of Graph:

Graph traversal is the process of visiting all the vertices in a graph in a systematic manner. There are two main methods of graph traversal: Depth First Traversal (DFT) and Breadth First Traversal (BFT).

Depth First Traversal (DFT): This method visits vertices by going deeper into the graph, exploring as far as possible along each branch before backtracking. In DFT, a vertex is marked as visited when it is first encountered and is added to a stack data structure. The algorithm then visits the vertices adjacent to the current vertex and repeats the process until all vertices have been visited.

Breadth First Traversal (BFT): This method visits vertices by exploring all neighbors of a vertex before moving on to the next level of neighbors. In BFT, a vertex is marked as visited when it is first encountered and is added to a queue data structure. The algorithm then visits all unvisited neighbors of the current vertex before moving on to the next vertex in the queue.

Topological sorting (Depth first, Breadth first topological sorting):

Topological sorting: A linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u, v), vertex u comes before vertex v. Topological sorting can be implemented using depth-first search (DFS) or breadth-first search (BFS). The time

complexity of topological sorting is O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Types:

a) Depth First Topological Sorting (DFTS): Uses DFT algorithm to visit vertices and maintain a stack to keep track of order.

b) Breadth First Topological Sorting (BFTS): Uses BFT algorithm to visit vertices and maintain a queue to keep track of order.

Applications: Scheduling tasks, order of operations in a compiler, etc.

Minimum spanning trees (Prim's, Kruskal's and Round- Robin algorithms):

Minimum Spanning Tree (MST): A subgraph of a graph that connects all vertices with minimum total edge weight and has no cycles.

Types:

a) Prim's: Starts with a vertex, adds edges with lowest weight to the tree.

b) Kruskal's: Sorts edges by weight, adds them to the tree if they don't form a cycle.

c) Round-Robin: A parallel version of Prim's for distributed computing.

Time complexity: O(E log E) or O(E log V).

Applications: shortest path, efficient communication networks, optimization problems.

Shortest-path algorithm (Greedy algorithm, and Dijkstra's Algorithm):

Shortest-path Algorithms: Find the shortest path between two vertices in a graph.

Algorithms:

a) Greedy Algorithm: Selects next vertex based on lowest edge weight.

b) Dijkstra's Algorithm: Uses priority queue to minimize total path weight.

Time complexity: O(V^2) or O(E + V log V).

7.3 INTRODUCTION TO DATA MODELS, NORMALIZATION, AND SQL

Introduction to data Models:

Data Model: A conceptual representation of data structures and relationships used by a DBMS to store, manage and manipulate data.

Types of Data Models:

a) Conceptual Data Model: High-level representation of data and relationships between entities and attributes.

b) Logical Data Model: Representation of data relationships, entities, attributes, and constraints.

c) Physical Data Model: Representation of physical storage of data, structures, and storage techniques.

d) Object-Oriented Data Model: Representation of data as objects with attributes, methods, and object relationships.

e) Relational Data Model: Based on mathematical relations, data is organized into tables with rows and columns.

f) Hierarchical Data Model: Representation of data as a tree-like structure with parent-child relationships.

g) Network Data Model: Representation of data as a network with multiple relationships between elements.

Normalization: Process of organizing data in a database to reduce redundancy and improve data integrity by breaking down large tables into smaller, more manageable tables.

Types of Normalization:

a) 1NF: Each column must contain atomic values, example: student information table with columns for name, address, and phone number.

b) 2NF: No partial dependencies, all non-key columns depend on the entire primary key, example: student and course information table linked by student ID.

c) 3NF: No transitive dependencies, non-key columns do not depend on other non-key columns, example: course information and instructor information table linked by course ID.

d) BCNF:
- Stands for Boyce-Codd Normal Form
- Requires that every non-trivial functional dependency in the relation is a dependency on a superkey
- Eliminates redundancy and anomalies in the data
- Example: Consider a table with columns for student ID, name, course, and instructor name. In BCNF, the table would be split into two tables, one for student information and one for course information, linked by student ID.

e) Fourth Normal Form (4NF):
- Stands for Fourth Normal Form
- Requires that there are no multi-valued dependencies
- Eliminates redundancy and anomalies in multi-valued relationships
- Example: Consider a table with columns for student ID, name, course, and instructor name. In 4NF, the table would be split into three tables, one for student information, one for course information, and one for instructor information, linked by student ID and course.

f) Fifth Normal Form (5 NF):
- Stands for Fifth Normal Form
- Requires that there are no join dependencies
- Eliminates redundancy and anomalies in join relationships
- Example: Consider a table with columns for student ID, name, course, and instructor name. In 5NF, the table would be split into multiple tables, each representing a single relationship between entities, such as student-course and course-instructor.

Structured Query Language (SQL):

Standard language for managing relational databases. It allows users to interact with a database by creating, reading, updating, and deleting data stored in tables.

- Performs operations such as insertion, retrieval, modification, and deletion of data
- Used to interact with databases and structure and manipulate data stored in tables
- Used to create, modify, and delete databases, tables, indexes, views, and other objects
- Manages relationships between tables and enforces data integrity with constraints.

SQL can be broadly classified into two types:

a) **Data Definition Language (DDL)**
   - Used to define the database schema and to specify the structure of the database.
   - Commands: CREATE, ALTER, DROP, TRUNCATE.

b) **Data Manipulation Language (DML)**
   - Used to manage the data stored in the database.
   - Commands: SELECT, INSERT, UPDATE, DELETE.

c) **Data Control Language (DCL)**
   - Used to control access to the data stored in the database.
   - Commands: GRANT, REVOKE.

d) **Transaction Control Language (TCL)**
   - Used to manage transactions in the database.
   - Commands: COMMIT, ROLLBACK, SAVEPOINT.

Syntax:

a) **Data Definition Language (DDL):**
   - CREATE TABLE table_name (column_name1 data_type, column_name2 data_type, ...);
   - ALTER TABLE table_name ADD column_name data_type;
   - DROP TABLE table_name;
   - TRUNCATE TABLE table_name;

b) **Data Manipulation Language (DML):**
   - SELECT column_name1, column_name2, ... FROM table_name;
   - INSERT INTO table_name (column_name1, column_name2, ...) VALUES (value1, value2, ...);
   - UPDATE table_name SET column_name1 = value1, column_name2 = value2, ... WHERE condition;
   - DELETE FROM table_name WHERE condition;

c) **Data Definition Langaue (DCL):**
   - GRANT privilege_name ON object_name TO user_name;
   - REVOKE privilege_name ON object_name FROM user_name;

d) ***Transaction Control Language (TCL):***
   - COMMIT;
   - ROLLBACK;
   - SAVEPOINT savepoint_name;

The three levels of data abstraction in database management are:

Data Abstraction:

Data abstraction is a technique in computer science to separate the implementation details of a system from its external behavior. It involves hiding the underlying complexity and presenting only essential features to the user.

a) View Level: Represents the user's perception of the data and provides a simplified view.

b) Conceptual Level: Provides a high-level view of the data and its relationships, independent of any specific database management system.

c) Physical Level: Deals with the storage and retrieval of data, specifying details such as file structures, storage formats, and access methods.

Data independence:

Data independence allows a database management system to change the underlying data storage structure or relationships without affecting the applications that use the data. There are two types of data independence:

a) Physical (ability to change physical storage without affecting applications) and

b) Logical (ability to change logical structure without affecting applications).

This makes the database more flexible and scalable, allowing for adaptability to changing needs.

Schema: Schema defines the structure of the database.

Database schema: blueprint or structure that defines the relationships and constraints between data in a database.

Database instance: specific set of data stored in the database at a given point in time. Instance is the actual data stored and manipulated in the database.

E-R Diagram:

An Entity-Relationship (E-R) diagram is a graphical representation of entities and their relationships to each other in a database. The following are the components of an E-R diagram:

a) **Entity**: Represents a real-world object or concept, such as a customer or an order. An entity is represented by a rectangle in the E-R diagram.

b) **Attribute**: Represents a characteristic of an entity, such as the name or address of a customer. Attributes are represented by ovals in the E-R diagram. There are three types of attributes:

- **Simple attribute**: a single characteristic of an entity, such as the name of a customer.
- **Composite attribute:** a combination of multiple characteristics of an entity, such as the address of a customer.

- **Derived attribute**: an attribute that is calculated from other attributes, such as total cost of an order.

c) **Relationship**: Represents the association between two or more entities, such as relationship between a customer and an order. Relationships are represented diamonds in the E-R diagram.

Strong and Weak Entity:

In a database, an entity set is a collection of entities of the same type, such as a set customers or a set of orders. There are two types of entity sets: strong and weak.

a) Strong Entity Set: has a primary key, unique identification for each entity in the set.

b) Weak Entity Set: lacks a primary key, dependent on another entity set for identification

Types of Attribute:

Four types of attributes in a database: simple, composite, derived, and multivalued.

a) Simple/Single: single characteristic, e.g. customer_ID.

b) Composite: combination of multiple characteristics, e.g. customer address.

c) Derived: calculated from other attributes, e.g. total cost of an order.

d) Multivalued: multiple values for a single entity, e.g. multiple phone numbers for a customer.

Key: A key in a database is an attribute or set of attributes that uniquely identifies each entity in an entity set. Keys serve as unique identifiers for entities and are used to enforce relationships and constraints between entities. There are several types of keys in a database, including:

a) Primary Key: unique identifier, cannot be null, enforces relationships/constraints.

b) Alternate Key: alternative identifier, enforces relationships/constraints.

c) Foreign Key: refers to primary key in another entity set, enforces relationships/integrity.

d) Candidate Key: potential primary key, only one primary key per entity set.

e) Super key: unique identifier including additional attributes beyond primary key.

Functional Dependencies:

- Relationships between attributes in a database.
- One attribute determines the value of another.
- Value of one attribute can be used to determine the value of the other.
- Helps enforce relationships/constraints and ensure data integrity.

Types of functional dependencies:

a) Trivial: attribute dependent on itself.

b) Full: one attribute determines value of another, second attribute cannot be determined without first.

c) Partial: one attribute determines value of another, but second attribute can also be determined by other attributes.

d) Transitive: one attribute determines value of another, second determines value of a third.

Integrity constraints and domain constraints:

Integrity constraints and domain constraints are used to ensure the validity and accuracy of data in a database.

Integrity Constraints: Integrity constraints are used to enforce rules and relationships between entities in a database. They ensure that the data entered into the database is consistent and accurate. Examples of integrity constraints include:

- Primary Key Constraint: Ensures that each entity has a unique identifier.
- Foreign Key Constraint: Ensures that relationships between entities are maintained.
- Not Null Constraint: Ensures that a value is entered for an attribute.
- Unique Constraint: Ensures that the values for an attribute are unique.
- Check Constraint: Ensures that the values entered for an attribute meet certain criteria.

Domain Constraints: Domain constraints are used to define the valid values for an attribute. They ensure that the data entered into the database is valid and within a certain range. Examples of domain constraints include:

- Range Constraint: Ensures that the values entered for an attribute are within a specified range.
- Set Constraint: Ensures that the values entered for an attribute are from a specified set of values.

Relations (Joined, Derived):

Relations:

- Collection of data organized into rows and columns.
- Joined: combine two or more relations based on common attribute.
- Derived: relation derived from one or more existing relations.

Examples:

- Joined relation: combining customer and order relations based on customer ID.
- Derived relation: calculating total sales for each customer by aggregating order data.

DDL and DML Commands:

DDL and DML are types of SQL commands used in a relational database management system.

DDL (Data Definition Language) Commands: DDL commands are used to define the structure of a database, including the tables, attributes, and relationships. Examples of DDL commands include:

- CREATE: used to create a new table, view, or index.
- ALTER: used to modify the structure of a table.
- DROP: used to delete a table, view, or index.

DML (Data Manipulation Language) Commands: DML commands are used to manipulate the data stored in a database. Examples of DML commands include:

- SELECT: used to retrieve data from one or more tables.

- INSERT: used to add new data to a table.
- UPDATE: used to modify existing data in a table.
- DELETE: used to delete data from a table.

View:

A view in a database management system is a virtual table that is derived from one or more existing tables. Views provide a way to access data in a database without having to physically store the data in a separate table.

Views can be used to:

- Simplify complex queries by breaking them down into smaller, more manageable pieces.
- Provide a way to access data that is otherwise difficult or impossible to access.
- Restrict access to sensitive data by presenting only a subset of the data in a table.
- Simplify data maintenance by encapsulating complex data structures and relationships.

Assertions and Triggers:

Used to enforce constraints and ensure data integrity.

- Assertions: statement defining constraint on data in database, system checks data to ensure it meets constraint.
- Triggers: set of instructions automatically executed when specified event occurs in database, used to enforce constraints and ensure data integrity by executing actions.

Types of Relational Algebra:

Unary Relational Operations: These operations operate on a single relation and include:

- SELECT: filters the rows of a relation based on a condition.
- PROJECT: selects a subset of the columns in a relation.

Binary Relational Operations: These operations operate on two relations and include:

- JOIN: combines two relations based on a common attribute.
- UNION: combines two relations into a single relation.
- INTERSECTION: returns the rows that are common to two relations.
- DIFFERENCE: returns the rows that are in one relation but not in another.

Aggregate Functions:

These operations are used to summarize data and include:

- SUM: calculates the sum of values in a column.
- AVG: calculates the average of values in a column.
- MIN: returns the minimum value in a column.
- MAX: returns the maximum value in a column.

Query Cost Estimation:

Query cost estimation is the process of determining the resources, such as time and memory, required to execute a database query. The goal of query cost estimation is to determine the most efficient way to execute a query and return the results to the user.

Query cost estimation is performed by the database management system and involves the following steps:

- *Parsing the query to understand its structure and requirements.*
- *Examining the database schema and statistics to determine the data distribution and relationships.*
- *Analyzing the query to determine the most efficient plan for executing the query.*
- *Estimating the resources, such as time and memory, required to execute the query based on the chosen plan.*

Steps in Query Processing:

- Parsing: understanding structure and requirements of query
- Optimization: determining most efficient plan for executing query
- Execution: retrieving data from relevant tables and applying necessary operations
- Returning Results: final result returned to user.

Query Optimization and Decomposition:

- Query Optimization: determining most efficient way to execute query
- Query Decomposition: technique to simplify complex queries into smaller parts
- Goal: minimize resources required to execute query and return results to user.

7.4 TRANSACTION PROCESSING, CONCURRENCY CONTROL AND CRASH RECOVERY

Transaction:

- Unit of work manipulating data in database
- Goal: ensure data consistency and correctness, even in case of failures
- Consists of series of database operations (inserts, updates, deletes)
- If any operation fails, entire transaction rolled back
- ACID properties ensure transactions are executed as single unit of work, data remains consistent, transactions are isolated, and results of completed transactions are durable.

ACID Properties:

- Atomicity: transaction executed as single, indivisible unit of work (All/None).
- Consistency: data remains consistent after transaction execution
- Isolation: transactions isolated from each other
- Durability: results of completed transactions permanent and survive failures.

Concurrency Control Protocols:

Lock-Based, Time-Stamp, and Validation-Based, each with its own method of controlling access to data. The choice of protocol depends on the requirements of the database management system.

- Lock-Based: uses locks to control access to data
- Time-Stamp: uses timestamps to control access to data
- Validation-Based: uses validation techniques to control access to data

Lock Based Protocol:

Transactions must acquire a lock on data before reading or writing it.

Two types of locks:

Shared (read-only)

- Shared lock allows multiple transactions to read the data simultaneously

Exclusive (read and write)

- Exclusive lock allows only one transaction to modify the data at a time
- Pessimistic concurrency control, transactions may be blocked waiting for locks to be released.

There are four types of lock protocols available:

- Simplistic Lock Protocol: basic concurrency control, but not prevent deadlocks or ensure serializability.
- Pre-Claiming Lock Protocol: reduce lock requests, improve performance, but not guarantee serializability.
- Two-Phase Locking (2PL): ensures serializability, prevents deadlocks, but may lead to performance issues.
- Strict Two-Phase Locking (Strict-2PL): ensures serializability by strict locking protocol, but may lead to performance issues if transactions are blocked waiting for locks to be released.

Necessary conditions to occur deadlock:

Deadlocks occur when the following conditions are met simultaneously:

- Mutual Exclusion: Exclusive access to resources
- Hold and Wait: Holding non-shareable resources and waiting for additional ones
- No Preemption: Voluntary release of resources only after completion of task
- Circular Wait: Processes in a circular chain holding resources requested by others.

Deadlock handling methods:

- Deadlock Prevention
- Deadlock Avoidance (Banker's Algorithm)
- Deadlock Detection & Recovery
- Deadlock Ignorance (Ostrich Method)

Deadlock Prevention:

- *Wait-Die scheme:* This scheme prevents deadlocks by allowing a transaction to wait if the resource it requires is currently held by another transaction. If the waiting transaction has a lower priority, it will be rolled back (the "die" part of the scheme).
- *Wound-Wait scheme:* This scheme is similar to the Wait-Die scheme, but instead of rolling back the waiting transaction, it rolls back the transaction holding the resource. This ensures that the resource is freed as soon as possible and that deadlocks are prevented.

Failure Classification:

a) Transaction failure: A failure that occurs within a transaction, causing it to not complete successfully. This can be caused by logical errors in the transaction logic or syntax errors in the transaction code.

b) System crash: A failure of the entire system, including hardware, software, and network components. This can result in loss of data and disruptions to the system's operations.

c) Disk failure: A failure of the disk storage system, which can result in loss of data and require the system to use backup data to recover.

Log-based Recovery:

- The log is a sequence of records for each transaction
- Maintained in stable storage
- Used for recovery in case of failure

The basic steps involved in log-based recovery are as follows:

a) Write-ahead logging: Before making changes to the database, the transaction writes a log record describing the change to the log file.

b) Checkpointing: System periodically writes the current state of the database to disk and updates the log.

c) Recovery: In case of a crash, the system reads the log to determine active transactions and rolls back incomplete transactions and redoes committed transactions not reflected in the database.

d) Consistency Checking: After recovery, the system checks database consistency and repairs any inconsistencies.

7.5 INTRODUCTION TO OPERATING SYSTEM & PROCESS MANAGEMENT

An operating system (OS) is a program that manages the resources of a computer. It acts as an intermediary between the computer's hardware and software, providing a user-friendly interface and common services for programs. The main functions of an OS include:

- Resource management: Allocating and managing hardware and software resources.
- Memory management: Allocating memory to programs and data.
- Process management: Controlling and managing program execution.

- File management: Managing storage, organization, and retrieval of files.
- Security: Providing security measures to protect the computer and its data.

Types of Operating System:

a) Single-tasking operating system:
   - Example: MS-DOS
   - Only one task can be executed at a time.

b) Multitasking operating system:
   - Example: Windows, macOS, Linux
   - Multiple tasks can be executed simultaneously.

c) Multithreading operating system:
   - Example: Windows, macOS, Linux
   - The operating system allows multiple threads to run within a single process.

d) Multiprogramming operating system:
   - Example: UNIX, Linux
   - Multiple programs can run at the same time, but only one is executed at a time.

e) Multiprocessing operating system:
   - Example: Windows, macOS, Linux
   - Multiple processors can be utilized to execute multiple processes simultaneously.

f) Real-time OS:
   - An operating system that is designed to respond to events within a guaranteed time frame, such as real-time embedded systems.
   - Examples include VxWorks and QNX.

Operating System Component:

a) Process Management: Manages the creation and execution of processes, including the allocation of resources and coordination of input/output operations.

b) I/O Device Management: Manages the input/output operations of the computer, including the allocation of resources and coordination of data transfer.

c) File Management: Manages the organization, storage, and retrieval of files and data on the computer's hard drive.

d) Network Management: Manages the communication between the computer and other devices on a network, including the allocation of network resources and coordination of data transfer.

e) Main Memory Management: Manages the allocation of main memory to processes and programs, ensuring that each program has the necessary resources to run.

f) Secondary Storage Management: Manages the organization, storage, and retrieval of data on secondary storage devices, such as hard drives and floppy disks.

g) Security Management: Provides security measures to protect the computer and its data from unauthorized access, viruses, and other security threats.

h) Command Interpreter System: Provides a user-friendly interface for executing commands and managing the computer's resources.

Operating System Structure:

Operating System Structure: There are two main structures for an operating system:

- Simple Structure: This is a basic structure where all the components of the operating system are integrated into a single module, providing a simple and straightforward solution.
- Layered Structure: This structure divides the operating system into a series of layers, each layer performing a specific set of functions. This structure provides a more modular and flexible solution, allowing for easier maintenance and future upgrades.

Operating System Services:

a) Program Execution: The operating system provides a platform for executing programs, including the allocation of processing time and memory resources.

b) Input/Output Operations: The operating system manages the communication between the computer's hardware and software, allowing programs to interact with input/output devices such as keyboards, mice, and displays.

c) File Management: The operating system provides a file system to store and organize data, including the allocation of disk space and the management of file access and protection.

d) Error Handling: The operating system provides error detection and handling mechanisms, including error messages and the ability to recover from crashes or other failures.

e) Resource Management: The operating system manages the allocation of resources such as memory, processing power, and input/output devices, ensuring that each program has the resources it needs to run.

f) Communication between Processes: The operating system provides mechanisms for processes to communicate with each other, including interprocess communication and message passing.

Introduction to Process and Process Description:

A process is an instance of a computer program that is being executed by one or many threads. It contains the program code and its current activity.

Key aspects of a process include:

- State: The current state of the process, such as running, blocked, or terminated.
- Program counter: The current location within the program code that is being executed.
- CPU register: The values stored in the CPU's registers, which hold data and intermediate results.

- Memory: The data and code of the process, including the program stack, heap, and other memory segments.
- System resources: The resources that the process has acquired, such as open files, network connections, and other system objects.

Process State:

a) New: The process is about to be created, but has not yet been created.
b) Ready: The process has been created and is loaded into main memory, waiting for CPU time.
c) Run: The process is chosen by the CPU for execution and its instructions are executed.
d) Blocked/Wait: The process requests access to I/O or user input, or needs access to a critical region. It waits in main memory without requiring CPU.
e) Terminated/Completed: The process is killed and its process control block (PCB) is deleted.
f) Suspend Ready: The process was in the ready state but was swapped out of main memory and placed onto external storage.
g) Suspend Wait/Suspend Blocked: The process was performing an I/O operation and lack of main memory caused it to move to secondary memory. Once the work is finished, it may transition to Suspend Ready.

Process Control Block (PCB) – It is a data structure that contains information about a process, including its:

a) Pointer: A stack pointer saved during state transitions to retain the current position of the process.
b) Process state: Stores the state of the process.
c) Process number: A unique ID assigned to each process for identification.
d) Program counter: Stores the address of the next instruction to be executed.
e) Register: CPU registers including accumulator, base, general purpose registers, etc.
f) Memory limits: Information about memory management system used by the OS, including page tables, segment tables, etc.
g) Open files list: A list of files opened for a process.

Threads:

A thread is a lightweight, independent unit of execution within a process. It shares the same memory space and system resources with other threads within the same process, but has its own program counter, register values, and stack.

Types of threads:

a) User-level threads: Threads created by the user in a program, managed by the application rather than the operating system.
b) Kernel-level threads: Threads created and managed by the operating system, which provide a more efficient way of managing multiple tasks.

Components of Thread:

- Program counter
- Register set
- Stack space

Scheduling Algorithm:

These are the six popular process scheduling algorithms in operating systems:

a) First-Come, First-Served (FCFS) Scheduling: The process that arrives first is executed first.
b) Shortest-Job-Next (SJN) Scheduling: The process with the shortest execution time is executed first.
c) Priority Scheduling: The process with the highest priority is executed first.
d) Shortest Remaining Time: The process with the shortest remaining time is executed first.
e) Round Robin (RR) Scheduling: A time-sharing algorithm in which each process is assigned a certain time slice in a cyclic way.
f) Multiple-Level Queues Scheduling: Processes are separated into different queues based on their priority, and each queue is scheduled using one of the algorithms such as FCFS or SJN.

Concurrency:

- Concurrency refers to the simultaneous execution of multiple instruction sequences
- Occurs in an operating system with multiple process threads running in parallel
- Process threads communicate through shared memory or message passing
- Results in sharing of resources, leading to issues such as deadlocks and resource starvation
- Coordinates execution of processes, memory allocation, and execution scheduling for maximum throughput.

Principle of Concurrency:

- Interleaved processes: Multiple processes are interleaved in execution, giving the illusion of simultaneous execution.
- Overlapped processes: Processes overlap in execution, with one process starting before the previous process has finished.
- Speed of execution: Cannot be predicted as it depends on other processes, operating system handling of interrupts, and scheduling policies.

Critical Region:

That part of the program which accesses the shared memory or file is called the critical region or critical section.

Race Condition:

a) Race condition: A situation where the outcome of a program depends on the timing of events such as the order of execution of instructions, interrupts, or access to shared resources.

b) Causes: Two or more processes access and manipulate the same shared data simultaneously, leading to unexpected results.

c) Example: Two threads accessing the same bank account, leading to incorrect balance updates.

d) Effects: Inconsistent data, unexpected behavior, and incorrect results.

e) Mitigation: Use synchronization techniques such as semaphores, monitors, and locks to enforce mutual exclusion and ensure that only one process accesses the shared data at a time.

Mutual Exclusion:

a) Ensures that only one process at a time can access a shared resource

b) Prevents race conditions by ensuring that multiple processes do not attempt to modify a shared resource simultaneously

c) Implemented through locks or semaphores to synchronize access to shared resources

d) Essential for maintaining data consistency and avoiding unpredictable results.

Semaphores

a) A variable shared between threads

b) Non-negative value

c) Used for signaling mechanism

d) Another thread can signal a waiting thread

Types:

There are two types of semaphores:

- Binary Semaphore: A binary semaphore can only have two values, 0 and 1, representing a locked and unlocked state.
- Counting Semaphore: A counting semaphore allows a resource to be locked multiple times. The value of the semaphore represents the number of available resources.

Mutex:

- Mutex is a synchronization object used to control access to shared resources
- Mutex is created with a unique name at the start of a program
- Mutex locking mechanism ensures only one thread can enter the critical section at a time
- The thread releases the mutex when it exits the critical section
- Mutex is a special type of binary semaphore
- Mutex includes priority inheritance mechanism to minimize effects of priority inversion
- Mutex minimizes the time higher priority tasks are kept in the blocked state

Message Passing:

a) Allows processes to communicate and synchronize actions without sharing same address space

b) Message passing provides two operations: send message and receive message

c) Messages can be fixed or variable size

d) Fixed size messages require a straight forward system level implementation but more difficult to program

e) Variable size messages require more system level implementation but easier to program

f) Communication link exists between two processes, P1 and P2, that want to communicate

g) Send() and receive() operations are used to implement the communication link.

Monitors:

a) Monitors are used for process synchronization

b) Monitors control access to shared data structures and procedures, and enforce mutual exclusion.

c) Monitors have 4 components: Initialization, Private Data, Monitor Procedure, and Monitor Entry Queue.

d) Initialization only occurs once when the monitor is created.

e) Private Data contains data and procedures that can only be accessed within the monitor.

f) Monitor Procedures can be called from outside the monitor.

g) Monitor Entry Queue holds the threads that have called monitor procedures.

Classical Problems of Synchronization:

a) Bounded-buffer (or Producer-Consumer) Problem: Coordination between a producer and consumer to ensure that a buffer does not overflow or underflow.

b) Dining-Philosophers Problem: Coordination between multiple philosophers trying to eat from a shared plate.

c) Readers and Writers Problem: Coordination between multiple readers and writers accessing a shared resource.

d) Sleeping Barber Problem: Coordination between a barber and customers waiting for haircuts in a barbershop.

7.6 MEMORY MANAGEMENT, FILE SYSTEMS AND SYSTEM ADMINISTRATION

Memory Management:

- Memory management manages primary memory (RAM) usage.
- It keeps track of memory location status and provides abstractions for efficient usage.
- Swapping between primary and secondary memory enables execution of large programs on limited memory.
- Memory management protects allocated memory from corruption and enables sharing of memory space between processes.

There are two main categories of memory management techniques:

a) Contiguous memory management schemes

b) Non-contiguous memory management schemes

Contiguous memory management schemes:

Contiguous memory management schemes involve allocating and managing memory in contiguous (adjacent) blocks. Examples of contiguous memory management schemes include:

- Single contiguous allocation: This technique involves allocating a single block of contiguous memory to a process.
- Partitioned allocation: This technique involves dividing memory into fixed-size partitions and allocating a partition to a process.
- Paged memory allocation: This technique involves dividing memory into fixed-size pages and allocating pages to a process as needed. Pages do not need to be contiguous in physical memory.

Segmented memory allocation: This technique involves dividing memory into variable-sized segments and allocating segments to a process. Segments may or may not be contiguous in physical memory.

Multiple partitioning:

Multiple partitioning is a memory management technique that divides memory into fixed or dynamic partitions.

- Fixed partitioning assigns fixed-size partitions to each process, resulting in internal fragmentation.
- Dynamic partitioning assigns variable-sized partitions based on a process's memory requirements to minimize internal fragmentation.

Example: In dynamic partitioning, if a process needs 10 MB of memory, it will be assigned a partition of exactly 10 MB, rather than being assigned a fixed-size partition that may have unused memory.

Non-contiguous memory management schemes:

- Non-contiguous memory allocation is a technique for allocating memory.
- It doesn't require physically contiguous memory locations.
- It allows different parts of a single process to be stored in a non-contiguous fashion.
- This results in more efficient and effective memory utilization.

Two popular techniques for non-contiguous memory allocation are Paging and Segmentation.

Paging:

- Memory is divided into fixed-size pages.
- Programs are allocated memory in page-sized chunks.
- Each page is mapped to a physical memory location.
- The operating system keeps track of the mapping between virtual and physical memory.
- Paging allows programs to use memory that is not physically contiguous.
- Provides more efficient memory allocation and management.

Segmentation:

- Memory is divided into logical segments or sections.
- Each segment is allocated a separate physical memory location.
- The operating system keeps track of the mapping between logical and physical memory.
- Segmentation allows programs to use memory that is not contiguous.
- Provides more flexibility in memory allocation and management than paging.

File System:

A file system is an essential component of an operating system that manages the storage, organization, and retrieval of files and directories on a storage device. Some common types of file systems include:

- FAT (File Allocation Table): An older file system used by earlier versions of Windows and other operating systems.
- NTFS (New Technology File System): A modern file system used by Windows, offering features such as file and folder permissions, compression, and encryption.
- ext (Extended File System): A file system commonly used on Linux and Unix-based operating systems.
- HFS (Hierarchical File System): A file system used by macOS.
- APFS (Apple File System): A new file system introduced by Apple for their Macs and iOS devices, providing enhanced performance, security, and reliability.

Virtual Memory Management:

- Virtual memory is a memory management technique used by an operating system.
- It allows the operating system to use more memory than is physically available in the computer's RAM.
- The technique involves temporarily transferring data from the RAM to the hard drive or SSD when it is not actively being used.
- When data is needed again, it is brought back from the disk and placed back into the RAM.
- This allows larger programs or multiple programs to be run simultaneously.
- Virtual memory is critical for efficient memory utilization in modern computer systems.

Demand Paging:

- Demand paging is a popular method of virtual memory management.
- In demand paging, the pages of a process that are least used are stored in secondary memory.
- When a page fault occurs, or when a page is demanded, it is copied from secondary memory to main memory.
- Various page replacement algorithms are used to determine which pages should be replaced.

- The selection of a page replacement algorithm depends on factors such as the number of page faults, available memory, and the size of the process.

Page Replacement Algorithm:

Here are the main points about different page replacement algorithms:

a) **FIFO Page Replacement Algorithm:**
   - Selects the page that was loaded into memory first.
   - Maintains a queue of pages in the order they were loaded.
   - When a page needs to be evicted, it removes the page at the front of the queue.

b) **LIFO Page Replacement Algorithm:**
   - Selects the page that was loaded into memory last.
   - Maintains a stack of pages in the order they were loaded.
   - When a page needs to be evicted, it removes the page at the top of the stack.

c) **LRU Page Replacement Algorithm:**
   - Selects the page that has not been used for the longest time.
   - Assumes that the least recently used page is less likely to be used again soon.
   - Maintains a list of pages in the order they were last used.

d) **Optimal Page Replacement Algorithm:**
   - Selects the page that will not be used for the longest time.
   - Assumes that the optimal page to replace is the one that will not be needed for the longest time in the future.
   - Requires knowledge of the future page requests, which is impossible to obtain in practice.

e) **Random Page Replacement Algorithm:**
   - Selects a random page to evict from memory.
   - Does not take into account the frequency of page usage or any other information about the pages in memory.
   - Simple to implement but often not very effective.

Introduction to File, Directory and File Paths:

File:

- A file is a collection of related information that is stored on a computer's storage device.
- Files can be of different types, such as text, images, audio, and video.
- Operating systems use a file management system to organize, store, and retrieve files.
- Each file has a unique name, which helps to identify and locate it on the storage device.
- Files have attributes such as size, creation date, modification date, and access permissions that define how they can be accessed and used.

Directory:

- A file directory contains a collection of files with attributes, location, and ownership information.
- The operating system manages much of the storage-related information.
- Directories are accessible by various file management routines and can be organized hierarchically.
- Directories play a crucial role in organizing and managing files in a computer system.
- Users can create, move, copy, and delete files and directories using OS tools.

File Allocation Method:

a) Contiguous Allocation:
   - Allocates storage space for a file in a contiguous block of space on a storage device.
   - Simple and efficient but can lead to fragmentation when files are deleted or resized.

b) Linked Allocation:
   - Allocates storage space for a file by linking together blocks of space on a storage device.
   - Each block contains a pointer to the next block in the file.
   - Efficient use of storage space but can result in slower file access times due to the need to follow the chain of pointers.

c) Indexed Allocation:
   - Allocates storage space for a file using an index block that contains pointers to the blocks of space used by the file.
   - Faster file access times than linked allocation but requires additional space for the index block.

d) Combined Allocation:
   - Combines the advantages of contiguous, linked, and indexed allocation methods.
   - Allocates the first few blocks of space to a file using contiguous allocation and then switches to linked or indexed allocation for the remaining blocks.

Fragmentation:

Fragmentation in an operating system can occur in three types: external, internal, and data fragmentation.

a) External fragmentation occurs when available memory becomes divided into small, non-contiguous blocks due to file and program allocation and deallocation.

b) Internal fragmentation occurs when allocated space remains unused and wasted.

c) Data fragmentation occurs when a file is stored non-contiguously on a storage device.

Fragmentation can impact system performance, but can be reduced through appropriate memory allocation methods, optimizing disk usage, and using defragmentation tools.

Mapping File Blocks on Disk Platter (Free Disk Space Management):

- Files are stored on disks and require allocation of disk space.
- The operating system maintains a Disk Allocation Table (DAT) to track free disk space.
- When a file is created, the OS searches the free space list and allocates space for the file.
- When a file is deleted, its space is added back to the free space list.
- DAT helps to manage disk space efficiently and avoid disk fragmentation.

Linked Free Space List:

- Free blocks are linked together using a linked list of disk blocks.
- Each disk block holds a maximum number of free blocks that can fit.
- Free blocks are used to store the free list.
- During allocation, the first block in the free space list is allocated.
- No disk fragmentation occurs.
- Reading each block can be time-consuming.
- Pointers used in the linked list consume disk space.

Bitmap or Bit Vector:

- A bitmap is a memory allocation technique that represents each block on a storage device with a single bit.
- A value of 1 represents a free block, and a value of 0 represents an allocated block.
- A disk with n blocks requires a bitmap with n bits.
- Bitmaps require less space and are simple and efficient.
- However, they may require hardware support to perform bit operations.

File System Performance:

- Disk access is slower than memory access due to seek time and latency time.
- Disk performance is worse than memory performance.
- Disk performance can be improved by using techniques such as block caching, block read-ahead, and reducing disk arm motion.

MULTIPLE CHOICE QUESTIONS

Data Structure

1. **What is the difference between a primitive data type and an abstract data type in programming?**
   A. Primitive data types are basic and cannot be changed, while abstract data types can be modified.
   B. Primitive data types are simple, while abstract data types are complex.
   C. Primitive data types are defined by the programming language, while abstract data types are defined by the programmer.
   D. Primitive data types are stored in memory, while abstract data types are stored on disk.
2. **Which of the following is a linear data structure?**
   A. Graph B. Queue
   C. Tree D. Hash Table
3. **In the big O notation, O (1) represents:**
   A. A constant time algorithm
   B. A linear time algorithm
   C. A logarithmic time algorithm
   D. An exponential time algorithm
4. **What does big O notation represent in algorithm analysis?**
   A. The worst-case time complexity of an algorithm
   B. The best-case time complexity of an algorithm
   C. The average time complexity of an algorithm
   D. The exact time complexity of an algorithm
5. **What does a big O notation of O(n) indicate about an algorithm?**
   A. The algorithm takes a constant amount of time regardless of the size of the input.
   B. The algorithm takes a linear amount of time proportional to the size of the input.
   C. The algorithm takes a logarithmic amount of time relative to the size of the input.
   D. The algorithm takes an exponential amount of time based on the size of the input.
6. **What is the big O notation for a binary search algorithm?**
   A. O(1) B. O(n)
   C. O(log n) D. O(n log n)
7. **Which of the following algorithms has a big O notation of O(n^2)?**
   A. Binary search B. Quick sort
   C. Bubble sort D. Linear search
8. **What does big O notation tell us about the efficiency of an algorithm?**
   A. It gives us an exact measurement of the time complexity of an algorithm.
   B. It gives us an approximation of the time complexity of an algorithm, taking into account the worst-case scenario.
   C. It gives us an estimate of the space complexity of an algorithm.
   D. It gives us a comparison of the time complexity of different algorithms.