CBCS SCHEME

USN | | | | | | | | | | |

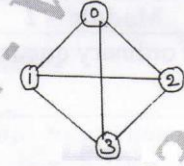
BCS304

Third Semester B.E./B.Tech. Degree Examination, Dec.2023/Jan.2024
Data Structures and Applications

Time: 3 hrs. Max. Marks: 100

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

		Module – 1	M	L	C
Q.1	a.	Define Data Structures. Explain with neat block schematic different type of data structures with examples. What are the primitive operations that can be performed?	10	L2	CO1
	b.	Differentiate between structures and unions shown examples for both.	5	L1	CO1
	c.	What do you mean by pattern matching? Outline knuth, Morris, Pratt pattern matching algorithm.	5	L2	CO1
		OR			
Q.2	a.	Define stack. Give the implementation of Push ( ), POP ( ) and display ( ) functions by considering its empty and full conditions.	7	L2	CO1
	b.	Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 6, 2, /, 3, -, 4, 2, *, +	7	L3	CO1
	c.	Write the Postfix form of the following using stack : (i) A*(B*C+D*E) + F (ii) (a + (b*c) / (d-e))	6	L3	CO1
		Module – 2			
Q.3	a.	What are the disadvantages of ordinary queue? Discuss the implementation of circular queue.	8	L2	CO2
	b.	Write a note on multiple stacks and priority queue.	6	L2	CO2
	c.	Define Queue. Discuss how to represent queue using dynamic arrays.	6	L2	CO2
		OR			
Q.4	a.	What is a linked list? Explain the different types of linked lists with neat diagram.	4	L2	CO2
	b.	Give the structure definition for singly linked list (SLL). Write a C function to, (i) Insert on element at the end of SLL. (ii) Delete a node at the beginning of SLL.	8	L3	CO2
	c.	Write a C-function to add two polynomials show the linked list representation of below two polynomials $p(x) = 3x^{14} + 2x^8 + 1$ $q(x) = 8x^{14} - 3x^{10} + 10x^6$	8	L3	CO2
		Module – 3			
Q.5	a.	Write a C-function for the following operations on Doubly Linked List (DLL): (i) addition of a node. (ii) concatenation of two DLL.	8	L3	CO3
	b.	Write C functions for the following operations on circular linked list : (i) Inserting at the front of a list. (ii) Finding the length of a circular list.	8	L3	CO3

	c.	For the given sparse matrix, give the diagrammatic linked representation. $A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$.	4	L3	CO3
		OR			
Q.6	a.	Discuss how binary tree are represented using, (i) Array (ii) Linked list	6	L2	CO3
	b.	Discuss inorder, preorder, postorder and level order traversal with suitable recursive function for each.	8	L2	CO3
	c.	Define Threaded Binary Tree. Discuss In-Threaded binary Tree.	6	L2	CO3
		Module – 4			
Q.7	a.	Write a function to perform the following operations on Binary Search Tree (BST) : (i) Inserting an element into BST. (ii) Recursive search of a BST.	8	L3	CO4
	b.	Discuss selection Trees with an example.	8	L2	CO4
	c.	Explain Transforming a first into a binary tree with an example.	4	L2	CO4
		OR			
Q.8	a.	Define graph. Show the adjacency matrix and adjacency list representation of the graph given below (Refer Fig. Q8 (a)).  Fig. Q8 (a)	6	L3	CO4
	b.	Define the following Terminologies with examples, (i) Digraph (ii) Weighted graph (iii) Self loop (iv) Parallel edges	8	L1	CO4
	c.	Explain in detail elementary graph operations.	6	L2	CO4
		Module – 5			
Q.9	a.	What is collision? What are the methods to resolve collision? Explain linear probing with an example.	7	L2	CO5
	b.	Explain in detail, about static and dynamic hashing.	6	L2	CO5
	c.	Discuss Leftist Trees with an example.	7	L2	CO5
		OR			
Q.10	a.	Explain different types of HASH function with example.	6	L2	CO5
	b.	Discuss AVL tree with an example. Write a function for insertion into an AVL Tree.	6	L3	CO5
	c.	Define Red-black Tree, Splay tree. Discuss the method to insert an element into Red-Black tree.	8	L2	CO5

* * * * *

1 a. Define Data Structures. Explain with neat block schematic different type of data structures with examples. What are the primitive operations that can be performed?

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

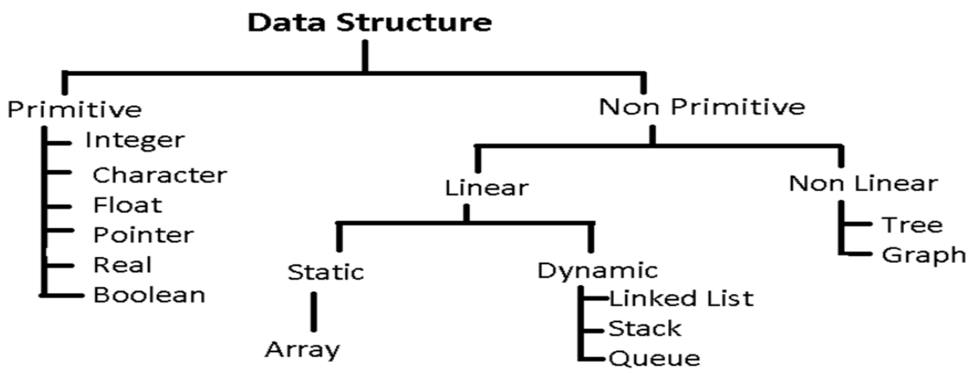


Figure: Classification of Data Structures

Types of Data Structures

1. Linear Data Structures

Linear data structures organize data sequentially, where elements are arranged one after the other.

1. **Array** : A collection of elements of the same data type stored in contiguous memory locations.

   Example: Storing marks of 5 students.

2. **Linked List**: A collection of nodes, where each node contains data and a reference (or link) to the next node in the sequence.

   Example: Managing dynamic lists like playlists or to-do lists.

3. **Stack**: A collection of elements following the **Last In First Out (LIFO)** principle.

   Example: Undo operation in a text editor.

4. **Queue**: A collection of elements following the **First In First Out (FIFO)** principle.

   Example: Managing tasks in a printer queue.

2. Non-Linear Data Structures

Non-linear data structures allow hierarchical relationships among elements.

1. **Tree**: A hierarchical structure where each node is connected to children nodes. A tree starts with a root node.
   Example: Representing file systems or organizational hierarchies.

2. **Graph**: A collection of nodes (vertices) connected by edges. Can be **directed** or **undirected**.

   Example: Representing social networks or roadmaps.

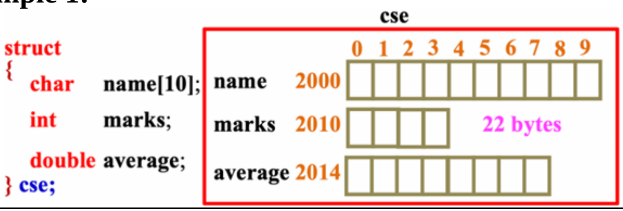
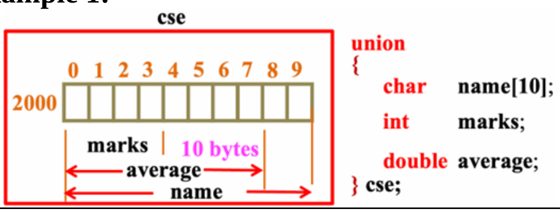
3. **Hash Table:** A data structure that maps keys to values for efficient lookup using a hash function.

Example: Implementing dictionaries or caches.

Primitive Operations on Data Structures

1. **Insertion**: Add an element to the data structure.
2. **Deletion**: Remove an element from the data structure.
3. **Traversal**: Visit each element in the data structure.
4. **Searching**: Find an element in the data structure.
5. **Sorting**: Arrange elements in a specific order.
6. **Updating**: Modify an element in the data structure.

1 b. Differentiate between structures and unions shown examples for both.

Structure	Union
• Separate memory locations are allocated for every member of the structure.	• The memory is allocated and its size is equal to maximum size of a member.
• Each member within a structure is assigned unique address.	• The address is same for all members.
• The address of each member is greater than the address of its previous member.	• The address is same for all members.
• Altering the value of one member will not affect other members of the structure.	• Altering the value of one member affects other member as the memory is shared.
• Several members of a structure can be initialized.	• Only the first member of the union can be initialized.
• Size of structure is >= sum of sizes of its members. (Greater because of slack bytes)	• Size of union is = size of largest member
• **Example 1:**  struct { char name[10]; int marks; double average; } cse; cse 0 1 2 3 4 5 6 7 8 9 name 2000 marks 2010 average 2014 22 bytes	• **Example 1:**  union { char name[10]; int marks; double average; } cse; cse 0 1 2 3 4 5 6 7 8 9 2000 marks 10 bytes average name

• **Example 2:**
#include <stdio.h>
// Define a structure
struct Employee
{
    char name[50];
    int age;
    float salary;
};
 	
int main()
{
    // Create an instance of Employee
    struct Employee emp = {"John Doe", 30, 50000.0};
 	
    // Access structure members
    printf("Name: %s\n", emp.name);
    printf("Age: %d\n", emp.age);
    printf("Salary: %.2f\n", emp.salary);
 	
    return 0;
}
	• **Example 2:**
	#include <stdio.h>
	// Define a union
	union Data
	{
	    int i;
	    float f;
	    char str[20];
	};
	
	int main()
	{
	    // Create an instance of Data
	    union Data data;
	
	    // Access union members one at a time
	    data.i = 10;
	    printf("Integer: %d\n", data.i);
	
	    data.f = 22.5;
	    printf("Float: %.2f\n", data.f);
	
	    strcpy(data.str, "Hello");
	    printf("String: %s\n", data.str);
	
	    return 0;
	}

1 c. What do you mean by pattern matching? Outline knuth, Morris, Pratt pattern matching algorithm.

Pattern Matching refers to the process of finding occurrences of a smaller string (the pattern) within a larger string (the text). The goal is to locate all positions in the text where the pattern appears. Efficient algorithms, like Knuth-Morris-Pratt (KMP), improve this process by avoiding redundant comparisons.

Knuth-Morris-Pratt (KMP) Algorithm:

The KMP algorithm preprocesses the pattern to create a Partial Match Table (LPS- Longest Prefix Suffix). This table is used to skip unnecessary comparisons in the text, ensuring the search is linear in time complexity, O(n+m), where n is the length of the text and m is the length of the pattern.

Steps of the KMP Algorithm:

1. **Build the LPS Array:**
    - Construct the Longest Prefix Suffix (LPS) array for the pattern.
    - LPS[i] contains the length of the longest prefix of the pattern that is also a suffix for the substring P[0:i].
2. **Search using the LPS Array:**
    - Traverse the text and compare characters with the pattern.
    - If there's a mismatch, use the LPS table to determine the next comparison point in the pattern, avoiding unnecessary checks.

Example:

Consider the following Text and pattern

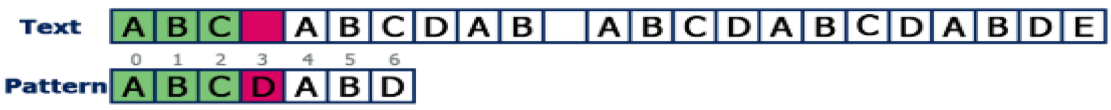
Text: ABC ABCDAB ABCDABCDABDE

Pattern: ABCDABD

LPS[ ] table for the above pattern is as follows

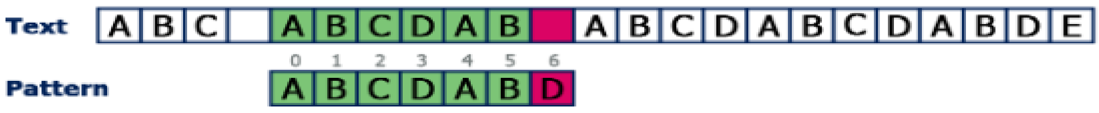
	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1 - Start comparing first character of Pattern with first character of Text from left to right



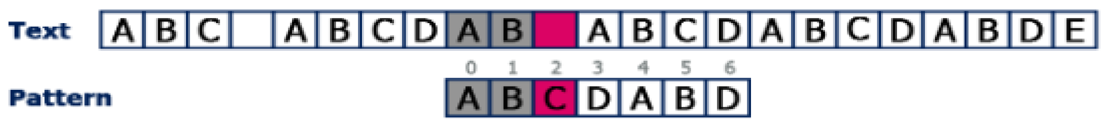
Here mismatch occurred at Pattern [3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

Step 2 - Start comparing first character of Pattern with next character of Text.



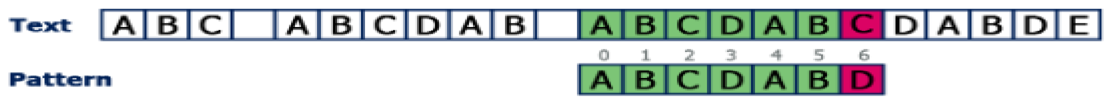
Here mismatch occurred at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values



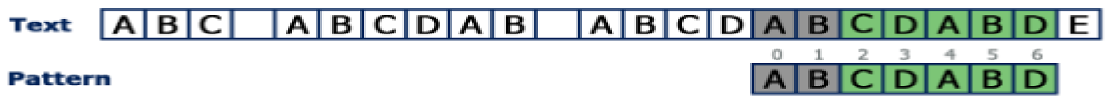
Here mismatch occurred at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occurred at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern [2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

2a. Define Stack. Give the implementation of Push (), POP () and display () functions by considering its empty and full conditions.

A **Stack** is a linear data structure that follows a particular order in which the operations are performed. The order may be **LIFO(Last In First Out)** or **FILO(First In Last Out)**.

```
#include <stdio.h>
#include <stdlib.h>
#define STACK_SIZE 5
int top = -1;
int stack [10];

// Function to insert an item into the stack
void push (int item)
{
        // Check for overflow of stack
        if ( top == STACK_SIZE - 1)
        {
                printf ("Stack Overflow");
                return;
        }
        // Insert an item into the stack
        stack [++top] = item;
}

// Function to delete an element from the stack
void pop ()
{
        // Check for underflow of stack
        if (_top== -1)
        {
                printf ("Stack Underflow");
                return;
        }
        printf ("Item deleted = %d ", stack[top--]);
        printf("\n");
}

// Function to display the contents of stack
void display ()
{
        int i;
        // Check for empty stack
        if (top== -1)
        {
                print ("Stack is empty");
                return;
        }
        printf("Stack: ");
        or (i = 0; i <= top; i++)
        printf(“ %d ”,  stack [i] );
        printf("\n");
}

void main()
{
        int choice, item;
        // Perform stack operations any number of times
        for (;;)
        {
                printf("1:Push 2:Pop 3:Display 4:Exit: ");
                scanf("%d", &choice);
                switch (choice)
                {
                        case 1: printf("Enter the item: ");
                                scanf("%d", &item);
                                push (item);
                                break;
                        case 2: pop ();
                                break;
                        case 3: display ();
                                break;
                        default: exit(0);
                }
        }
}
```

2 b. Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 6, 2, /, 3, -, 4, 2, *, +

A postfix expression is evaluated using a stack. The algorithm works as follows:

Algorithm:

1. Initialize a Stack: Create an empty stack to store operands.

2. Iterate through the Expression:

- For each token in the postfix expression:
- If the token is an operand (number), push it onto the stack.
- If the token is an operator:
    1. Pop the top two elements from the stack (operand2 and operand 1).
    2. Apply the operator: result = operand1 operator operand2.
    3. Push the result back onto the stack.

3. Final Result:

After processing all tokens, the value remaining in the stack is the result.

Given Postfix Expression: 6, 2, /, 3, -, 4, 2, *, +

Step-by-Step Evaluation:

Stack	**Op2**	**Op1**	**Result = Op1 $\oplus$ Op2**
6			
6 2	**2**	**6**	**Result = 6 / 2 = 3**
3 (Result)			
3 3	**3**	**3**	**Result = 3 – 3 = 0**
0 (Result)			
0 4			
0 4 2	**2**	**4**	**Result = 4 * 2 = 8**
0 8(Result)			
0 8	**8**	**0**	**Result = 8 + 0 = 8**

2 C. Write the Postfix form of the following using stack:

(i) A*(B*C+D*E) + F **(ii) (a + (b*c)/(d-e))**

(i) A*(B*C+D*E) + F

$A*(\underbrace{B*C}_{T1}+D*E) + F$ T1= B C *

$A*(T1+\underbrace{D*E}_{T2})+F$ T2= D E*

$A*(\underbrace{T1+T2}_{T3})+F$ T3= T1 T2 +

$\underbrace{A*T3}_{T4}+F$ T4= A T3 *

T4 + F

T4 F +

A T3 * F+

A T1 T2 + * F +

A T1 D E * + * F +

A B C * D E * + * F +

Postfix Expression is A B C * D E * + * F +

(ii) (a + (b*c)/(d-e))

$(a + (b * c) / \underbrace{(d - e)}_{T1})$ T1= d e -

$(a + (\underbrace{b * c)}_{T2} / T1)$ T2= b c *

$(a + \underbrace{T2 / T1}_{T3})$ T3= T2 T1 /

(a + T3)

a T3 +

a T2 T1 / +

a b c * T1 / +

a b c * d e - / +

Postfix Expression is a b c * d e - / +

3 a. What are the disadvantages of ordinary queue? Discuss the implementation of circular queue.

Disadvantages of ordinary queue:

- Once queue is full, even if some elements are deleted from queue, it is not possible to insert the items into queue. The message "Queue is full" is displayed on the screen.
- Shifting queue elements towards left after every deletion, consume lot of time.
- Wastes memory space after dequeuing elements.
- Fixed size limits flexibility.
- No wrapping around for efficient memory use.
- Inefficient for dynamic resizing operations.
- Limited to basic linear functionality.
- The above disadvantages can be overcome using circular queues.

Implementation of circular queue:

```
#include <stdio.h>
#include <stdlib.h>
#define Q_SIZE 5
int front = 0, rear -1, count = 0;
int queue[10];

// Function to insert an item into circular queue
void insert_rear (int item)
{
        // Check for overflow of queue
        if (count === Q_SIZE)
        {
                printf ("Queue Overflow");
                return;
        }
        // Increment rear by 1
        rear = ( rear + 1) % Q_SIZE;
        // Insert an item into the queue
        queue[ rear] = item;
        // Update count by 1
        count++;
}
// Function to delete an element from queue
void delete_front()
{
        // Check for underflow of queue
        if (count == 0)
        {
                printf ("Queue Underflow");
                return;
        }
        // Delete the item from circular queue
        printf ("Item deleted :%d", queue[front]);
        // Increment front by 1
        front (front + 1)% Q_SIZE;
        // Update count by 1
        count = count - 1;
}
// Function to display the contents of queue
void display ()
{

        int i, temp;
        // Check for empty queue
        if (count=0)
        {
                printf("Queue is empty");
                return;
        }
        printf("Queue: ");
        temp = front;
        for (i = 1; i <= count; i++)
        {
                printf("%d", queue [temp]);
                temp = (temp +1)% Q_SIZE;
        }
        printf("\n");
}
void main()
{
        int choice, item;
        // Perform queue operations any number of times
        for (;;)
        {
                printf("1:Insert rear 2:Delete front 3:Display 4:Exit: ");
                scanf("%d", &choice);
        switch (choice)
                {
                    case 1: printf("Enter the item :");
                            scanf("%d", &item);
                            insert_rear (item);
                            break;
                    case 2: delete_front();
                            break;
                    case 3: display();
                            break;
                    default: exit(0);
            }
        }
}
```

3 b. Write a note on multiple stacks and priority queue.

Multiple Stacks

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, meaning the last element added is the first one to be removed. A multiple stack system refers to the concept of using multiple stacks within a single data structure or array, either to manage distinct types of data or to optimize memory usage. This approach is useful in scenarios where different stacks are needed for separate purposes, such as evaluating expressions, handling function calls, or managing different states.

Applications:

1. Memory management: Multiple stacks allow for efficient usage of memory, especially when the size of each stack is unknown or changes dynamically.
2. Expression evaluation: Different stacks can be used for different types of operands or operators during complex mathematical or logical expression evaluations.
3. Function calls: Multiple stacks can be used to maintain different contexts in recursive or multi-threaded applications.

Priority Queue

A priority queue is an abstract data type that stores elements along with their associated priorities. Elements are dequeued in order of their priority rather than the order they were enqueued. In a priority queue, the element with the highest priority is always dequeued first. This differs from a regular queue where elements are dequeued in a first-come, first-served manner (FIFO).

Applications:

1. Job Scheduling: In operating systems, priority queues are used to manage tasks with different priorities, such as in CPU scheduling or task management.
2. Graph Algorithms: Priority queues are integral to algorithms like Dijkstra’s and Prim’s for shortest path and minimum spanning tree calculations.
3. Event Simulation: In simulations, events may be processed based on their priorities (e.g. in discrete event simulation).
4. Huffman Encoding: Priority queues are used in algorithms like Huffman coding for data compression.

3 c. Define Queue. Discuss how to represent queue using dynamic arrays.

Definition : queue is a special type of data structure where elements are inserted from one end and are deleted from another end.

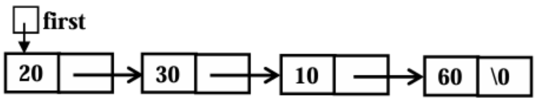
```
#include <stdio.h>
#include <stdlib.h>
int  Q_SIZE = 1;
int front = 0, rear -1;
int *queue;

// Function to insert item into queue
void insert_rear (int item)
{
        // Check for overflow of queue
        if(rear == Q_SIZE – 1)
        {
                Printf(“ Queue Overflow”)
                Q_SIZE++;
                Queue = realloc (queue, Q_SIZE * sizeof ( int ));
        }
        // Insert an item at the rear end of queue
        Queue [ ++ rear ] = item;
}

// Function to delete an item from queue
void delete_front()
{
        // Check for underflow of queue
        if (front > rear)
        {
                printf ("Queue Underflow");
                return;
        }
        printf ("Item deleted: %d ", queue[ front++]);

        // Reset to initial values
        if (front > rear) front = 0, rear = -1;
}

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

void main()
{
        int choice, item;
        queue = (int) malloc (Q_SIZE * sizeof (int));
        for (;;)
        {
                printf("1:Insert rear 2:Delete front 3:Display 4:Exit: ");
                scanf("%d", &choice);
                switch (choice)
                {
                        case 1: printf("Enter the item :");
                                scanf("%d", &item);
                                insert_rear (item);
                                break;
                        case 2: delete_front();
                                break;
                        case 3: display();
                                break;
                        default: exit(0);
                }
        }
}
```

4 a. What is a linked list? Explain the different types of linked lists with neat diagram.

A linked list is a data structure which is collection of zero or more nodes where each node is connected to one or more nodes. If each node in the list has only one link, it is called singly linked list. If it has two links one containing the address of the next node and other link containing the address of the previous node it is called doubly linked list. Each node in the singly list has two fields namely:

- ♦ info – This field is used to store the data or information to be manipulated
- ♦ link – This field contains address of the next node.

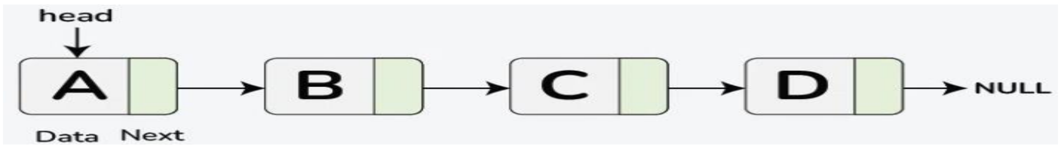
The pictorial representation of a singly linked list where each node is connected to the next node is shown below:



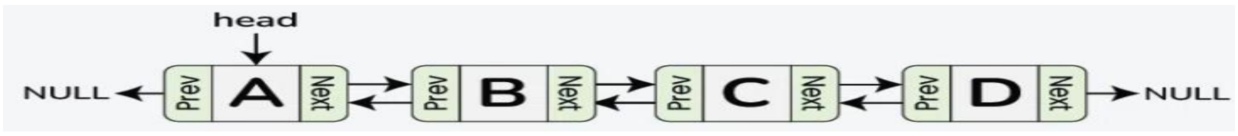
Different types of Linked Lists:

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

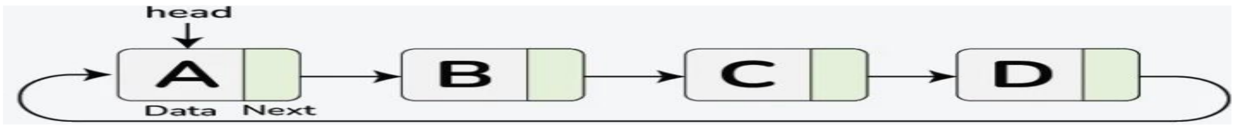
1. **Singly linked lists:**
   A **singly linked list** is a set of nodes where each node has two fields 'data' and 'link'. The 'data' field stores actual piece of information and 'link' field is used to point to next node. Basically the 'link' field stores the address of the next node.



2. **Doubly linked list:**
   A **Doubly Linked List** (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.

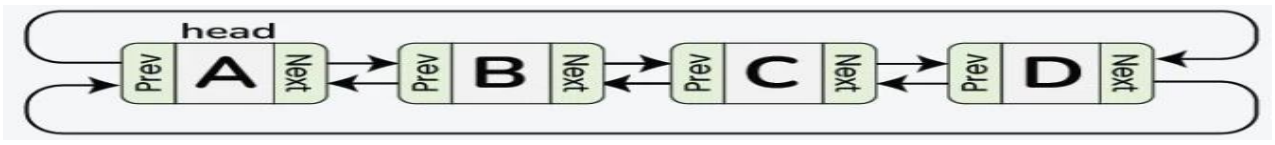


3. **Circular Linked List**
   C is a type of linked list in which the last node's next pointer points back to the first node of the list, creating **a circular** structure. This design allows for continuous traversal of the list, as there is no null to end the list**.**



4. Doubly Circular linked list

Doubly Circular linked list or a circular two-way linked list is a complex type of linked list that contains a pointer to the next as well as the previous node in the sequence. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node.



4 b. Give the structure definition for singly linked list (SLL). Write a C function to,

(i) **Insert on element at the front end of SLL.**

```
NODE insert_front(int item, NODE first)
{
        NODE temp;
        temp = getnode();
        temp->info = item;
        temp->link = first;
        return temp;
}
```

(ii) Delete a node at the beginning of SLL.

```
NODE delete_front(NODE first)
{
        NODE temp;

        if ( first == NULL )
        {
                printf("List is empty cannot delete\n");
                return NULL;
        }
        temp = first;
        temp = temp->link
        printf("Item deleted = %d\n",first->info);
        free(first);
        return temp;
}
```

4 C. Write a C-function to add two polynomials show the linked list representation of below two polynomials

$p(x)=3x^{14}+2x^{8}+1$

$q(x)=8x^{14}-3x^{10}+10x^{6}$

```
Node* addPolynomials(Node* poly1, Node* poly2)
{
        Node* result = NULL;
        while (poly1 != NULL && poly2 != NULL)
        {
                if (poly1->power > poly2->power)
                {
                        appendNode(&result, poly1->coeff, poly1->power);
                        poly1 = poly1->next;
                }
                else if (poly1->power < poly2->power)
                {
                        appendNode(&result, poly2->coeff, poly2->power);
                        poly2 = poly2->next;
                }
                else
                {
                        int sumCoeff = poly1->coeff + poly2->coeff;
                        if (sumCoeff != 0)
                        {
                                appendNode(&result, sumCoeff, poly1->power);
                        }
                        poly1 = poly1->next;
                        poly2 = poly2->next;
                }
        }
```

```
	while (poly1 != NULL)
	{
		appendNode(&result, poly1->coeff, poly1->power);
		poly1 = poly1->next;
	}

	while (poly2 != NULL)
	{
		appendNode(&result, poly2->coeff, poly2->power);
		poly2 = poly2->next;
	}
	return result;
}

int main()
{
	Node* poly1 = NULL;
	Node* poly2 = NULL;

	 // Constructing the first polynomial p(x) = 3x^14 + 2x^8 + 1
	appendNode(&poly1, 3, 14);
	appendNode(&poly1, 2, 8);
	appendNode(&poly1, 1, 0);

	// Constructing the second polynomial q(x) = 8x^14 - 3x^10 + 10x^6
	appendNode(&poly2, 8, 14);
	appendNode(&poly2, -3, 10);
	appendNode(&poly2, 10, 6);

	// Display the polynomials
	printf("First Polynomial: ");
	displayPoly(poly1);
	printf("Second Polynomial: ");
	displayPoly(poly2);

	// Adding the polynomials
	Node* result = addPolynomials(poly1, poly2);
	printf("Sum of Polynomials: ");
	displayPoly(result);

	return 0;
}
```

5 a. Write a C-function for the following operations on Doubly Linked List (DLL):

(i) addition of a node.

```
void addNode(Node** head, int data)
{
        Node* newNode = createNode(data);
        if (*head == NULL)
        {
                *head = newNode;
                return;
        }
        Node* temp = *head;
        while (temp->next != NULL)
        {
                temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
}
```

(ii) concatenation of two DLL.

```
Node* concatenateDLL(Node* head1, Node* head2)
{
        if (head1 == NULL) return head2;
        if (head2 == NULL) return head1;

        Node* temp = head1;
        while (temp->next != NULL)
        {
                temp = temp->next;
        }
        temp->next = head2;
        head2->prev = temp;

        return head1;
}
```

5 b. Write C functions for the following operations on circular linked list :

i. Inserting at the front of a list.

```
void insertAtFront(Node** head, int data)
{
        Node* newNode = createNode(data);
        if (*head == NULL)
        {
              newNode->next = newNode;
                *head = newNode;
        }
        else
        {
                Node* temp = *head;
                while (temp->next != *head)
                {
                        temp = temp->next;
                }
                // Insert the new node at the front
                temp->next = newNode;
                newNode->next = *head;
                *head = newNode;
       }
}
```

ii Finding the length of a circular list.

```
int findLength(Node* head)
{
        if (head == NULL)
        {
                return 0;
        }
        int count = 1;
        Node* temp = head->next;
        while (temp != head)
        {
                count++;
                temp = temp->next;
        }
        return count;
}
```

5 C. For the given sparse matrix, give the diagrammatic linked representation.

$$A= \begin{matrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{matrix}$$

To represent the given sparse matrix using a linked list or diagrammatic representation, we will follow these steps:

Step 1: Identify Non-Zero Elements
We need to locate all the non-zero elements in the matrix A and note their positions (row and column indices).

The non-zero elements in matrix A are:
- A[0][0] = 2
- A[1][0] = 4
- A[1][3] = 3
- A[3][0] = 8
- A[3][3] = 1
- A[4][2] = 6

Step 2: Create Linked List Representation
In a linked list representation for a sparse matrix, we typically use a structure that contains the row index, column index, and the value of each non-zero element.

We can represent each non-zero element as a node in the linked list. The structure for each node can be defined as follows:

- Row index
- Column index
- Value

The linked list for the non-zero elements will look like this:

1. Node: (0, 0, 2)
2. Node: (1, 0, 4)
3. Node: (1, 3, 3)
4. Node: (3, 0, 8)
5. Node: (3, 3, 1)
6. Node: (4, 2, 6)

Step 3: Diagrammatic Representation
We can visualize the linked list as follows:

(0, 0, 2) -> (1, 0, 4) -> (1, 3, 3) -> (3, 0, 8) -> (3, 3, 1) -> (4, 2, 6) -> NULL

Where each tuple represents (row index, column index, value) and the arrows indicate the next node in the linked list.

Final Answer
The linked list representation of the sparse matrix A is:
(0, 0, 2) -> (1, 0, 4) -> (1, 3, 3) -> (3, 0, 8) -> (3, 3, 1) -> (4, 2, 6) -> NULL

6 a. Discuss how binary tree are represented using,

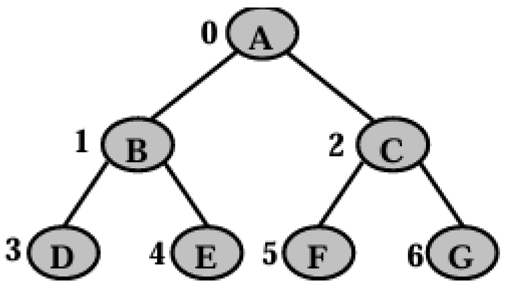
(i) Array (ii) Linked list

(i) Array Representation:

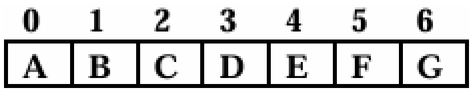
In an **array representation**, the tree is stored as a complete binary tree (filled level-by-level from left to right). Each node's position in the array i smapped based on its relationship to its parent and children.

Structure and Index Mapping:

1. **Root Node**: Stored at index 0 (or sometimes 1).
2. **Left Child**: The left child of a node at index i is stored at index 2i+1.
3. **Right Child**: The right child of a node at index i is stored at index 2i+2.
4. **Parent Node**: The parent of a node at index i is stored at ⌊i−1/2⌋.



Tree



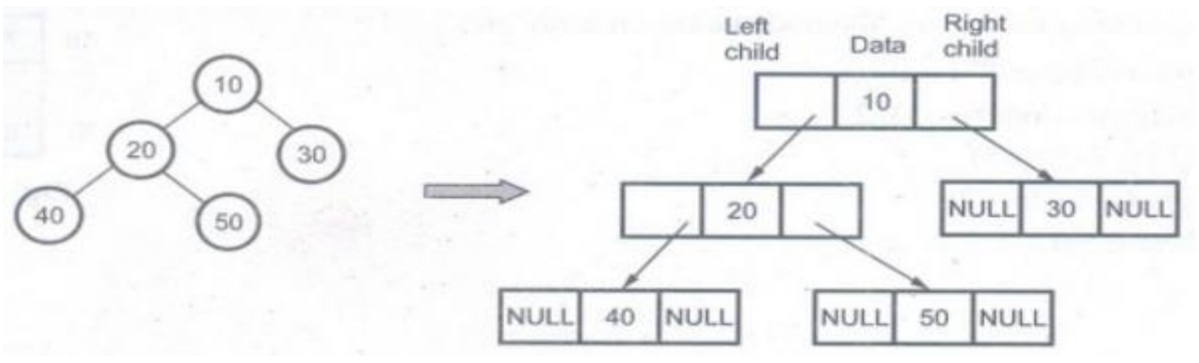
0	1	2	3	4	5	6
A	B	C	D	E	F	G

Array Representation

(ii) Linked list Representation:

Linked representation, a node in a tree has three fields:

- **info –** which contains the actual information
- **llink** – which contains address of the left subtree
- **rlink** – contains address of the right subtree.



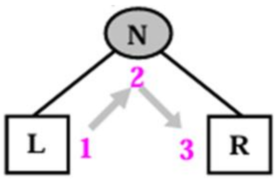
Tree Linked List Representation

6 **b. Discuss inorder, preorder, postorder and level order traversal with suitable recursive function for each.**

1. **Inorder Traversal (Left, Root, Right)**

   Definition: The inorder traversal of a binary tree can be recursively defined as follows:

   - Traverse the Left subtree in inorder [L]
   - Process the root Node [N]
   - Traverse the Right subtree in inorder [R]



Sequence: Left → Root → Right

Example: For a tree:

```
   1
  / \
 2   3
/ \
4   5
```

Inorder : 4, 2, 5, 1, 3

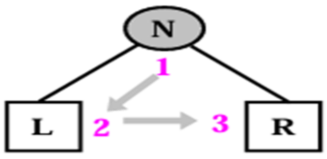
Recursive Function:

```
void inorder(NODE root)
{
        if ( root == NULL ) return;
        inorder(root->llink);
        printf("%d ",root->info);
        inorder(root->rlink);
}
```

2. Preorder Traversal (Root, Left, Right)

Definition: The preorder traversal of a binary tree can be recursively defined as follows:

- Process the root Node [N]
- Traverse the Left subtree in preorder [L]
- Traverse the Right subtree in preorder [R]



Sequence: Root → Left → Right.

Example: For a tree:

```
   1
  / \
 2   3
/ \
4   5
```

Preorder : 1, 2, 4, 5, 3

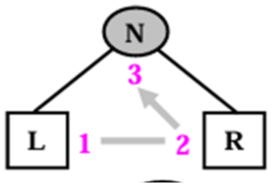
Recursive Function:

```
void preorder(NODE root)
{
        if ( root == NULL ) return;
        printf("%d ",root->info);
        preorder(root->llink);
        preorder(root->rlink);
}
```

3. Postorder Traversal (Left, Right, Root)

Definition: The postorder traversal of a binary tree can be recursively defined as follows:

- Traverse the Left subtree in postorder [L]
- Traverse the Right subtree in postorder [R]
- Process the root Node [N]



Sequence: Left → Right → Root.

Example: For a tree:

```
   1
  / \
 2   3
/ \
4   5
```

Postorder : 4, 5, 2, 3, 1

Recursive Function:

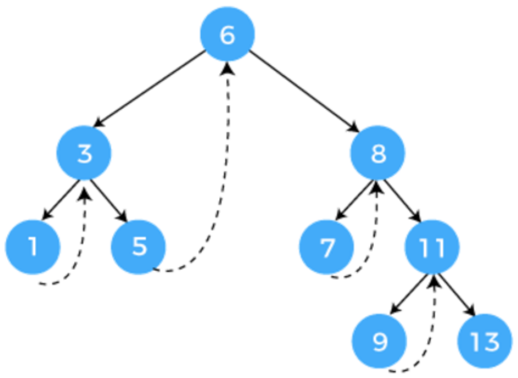
```
void postorder(NODE root)
{
        if ( root == NULL ) return;
        postorder(root->llink);
        postorder(root->rlink);
        printf("%d ",root->info);
}
```

6 C. Define Threaded Binary Tree. Discuss In-Threaded binary Tree.

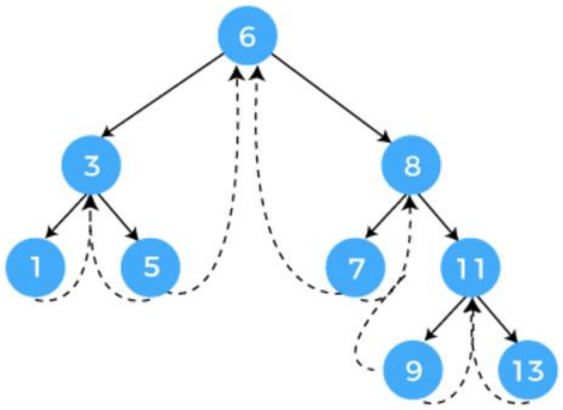
A Threaded Binary Tree is a variation of the binary tree where unused right and left child pointers (typically NULL in standard binary trees) are utilized to maintain references to the in-order predecessor or in-order successor of the node. This threading simplifies in-order traversal without using a stack or recursion, making it more memory-efficient.

Types of Threaded Binary Trees:
Threaded binary trees are categorized based on how the threading is implemented:

- **Single Threaded Binary Tree:**
  Each node is threaded either to its in-order predecessor or its in-order successor.



- **Double Threaded Binary Tree:**
  Each node has threads for both its in-order predecessor and in-order successor.



In-Threaded Binary Tree

An In-Threaded Binary Tree is a specific type of threaded binary tree in which:

1. The **left null pointer** of a node is used to store a thread to its in-order predecessor.
2. The **right null pointer** of a node is used to store a thread to its in-order successor.

Characteristics of an In-Threaded Binary Tree:

- The left thread points to the largest value smaller than the current node in the in-order traversal.
- The right thread points to the smallest value larger than the current node in the in-order traversal.
- In-order traversal can be performed efficiently without the need for a stack or recursion.

Advantages of In-Threaded Binary Trees

- **Efficient Traversal:** In-order traversal can be done in O(n) time with constant auxiliary space.
- **Memory Utilization:** Null pointers are utilized, reducing memory waste.
- **Simplified Algorithm:** Traversal and insertion/deletion algorithms are more straightforward compared to unthreaded trees when using the threading.

Example: In-Threaded Binary Tree Construction

Consider this binary tree:

```
    4
   / \
  2   6
 / \
1   3
```

If we make it an in-threaded binary tree:

Node 1's left thread points to NULL, and its right thread points to 2.

Node 2's left thread points to 1, and its right thread points to 3.

Node 3's left thread points to 2, and its right thread points to 4.

Node 4's left thread points to 3, and its right thread points to 6.

Node 6's left thread points to 4, and its right thread points to NULL.

7 a. Write a function to perform the following operations on Binary Search Tree (BST):

(i)Inserting an element into BST.

```
NODE insert(int item, NODE root)
{
        NODE temp,cur,prev;

        temp = getnode();
        temp->info = item;
        temp->llink = NULL;
        temp->rlink = NULL;

        if ( root == NULL ) return temp;

        prev = NULL;
        cur  = root;

        while ( cur != NULL )
        {
                prev = cur;
                        if (item < cur->info )
                        cur = cur->llink;
                else     /* or */
                        cur = cur->rlink;
        }
        if ( item < prev->info )
                        prev->llink = temp;
        else
                        prev->rlink = temp;

 return root;
}
```

(ii)Recursive search of a BST.

```
NODE search(int item, NODE root)
    {
        if ( root == NULL ) return root;
        if ( item == root->info ) return root;
        if ( item < root->info )
            return search(item, root->llink);
        return search(item, root->rlink);
    }
```

7 b. Discuss selection Trees with an example.

A Selection Tree is a complete binary tree used to efficiently determine the smallest (or largest) element from a collection of elements. It is widely used in k-way merging and sorting algorithms, particularly in external sorting. Each internal node of the tree stores the result of a comparison, while the leaf nodes represent the input elements.

Types of Selection Trees:

1. **Winner Tree:**

- The root contains the smallest element.
- Internal nodes represent the winners of comparisons between child nodes.

Example:

We have 4 numbers from 4 sorted arrays:
Array 1: [3], Array 2: [5], Array 3: [2], Array 4: [8]

- **Initial Comparison:**
  - ❖ Compare pairs:
    - o Array 1 (3) vs. Array 2 (5) → Winner: 3
    - o Array 3 (2) vs. Array 4 (8) → Winner: 2
  - ❖ Compare the winners: 3 vs. 2 → Winner: 2 (Root).
- **Tree Structure:**

```
   2
  / \
 3   2
/ \ / \
3  5 2  8
```

Result:

- The root (2) is the smallest.
- Replace 2 with the next number from Array 3 and repeat.

2. **Loser Tree:**

- The root contains the second smallest element.
- The smallest element remains at a leaf node, facilitating efficient updates during merging.

Example:

Using the same numbers (3, 5, 2, 8):

- **Initial Comparison:**
  - ❖ **Compare pairs:**
    - Array 1 (3) vs. Array 2 (5) → Winner: 3, Loser: 5
    - Array 3 (2) vs. Array 4 (8) → Winner: 2, Loser: 8
  - ❖ Compare the winners: 3 vs. 2 → Winner: 2 (Root), Loser: 3.
- **Tree Structure:**

```
      2
     / \
    3   8
   / \ / \
  5  3 2  8
```

Result:

- Root (2) is the smallest.
- Replace 2 with the next number from Array 3 and update the tree.

Applications of Selection Trees

- k-Way Merging: Merges k k sorted lists into one sorted list.
- External Sorting: Useful when sorting large datasets stored on external storage.
- Huffman Coding: Builds an optimal prefix code by repeatedly selecting the smallest elements.

Advantages

- Efficient for merging sorted arrays (k-way merge).
- Minimizes comparisons by organizing them hierarchically.
- Useful in external sorting and database applications.

Disadvantages

- Additional memory required for the tree structure.
- Overhead in updating the tree during repeated operations.

7 c. Explain transforming a forest into a binary tree with an example.

Transforming a forest into a binary tree is a process used to represent multiple trees (a forest) as a single binary tree. This transformation is helpful in simplifying tree operations.

Steps to Transform a Forest into a Binary Tree

- Left-Child Representation: The first child of each node becomes the left child in the binary tree.
- Right-Sibling Representation: The next sibling of a node becomes its right child in the binary tree.

Example

Let’s take a forest of two trees:

Tree 1:

```
   A
  / \
 B   C
```

Tree 2:

```
   D
  / \
 E   F
```

Step 1: Combine the Roots

Link the roots of the trees to form a sequence: A → D.

Step 2: Convert Each Tree

Tree 1:

- A's left child is its first child → B.
- B's right child is its sibling → C.

Converted:

```
  A
 /
B
 \
  C
```

Tree 2:

- D's left child is its first child → E.
- E's right child is its sibling → F.

Converted:

```
  D
 /
E
 \
  F
```

Step 3: Link the Converted Trees

Now, link A and D using the right-sibling rule:

```
     A
   /   \
  B     D
   \   /
    C E
       \
        F
```

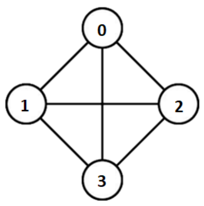
Final Binary Tree

The forest is now represented as a single binary tree.

Advantages

- Allows representation of multiple trees as a single structure.
- Simplifies operations like traversal and storage.

8 a. Define graph. Show the adjacency matrix and adjacency list representation 6 of the graph given below (Refer Fig. Q8 (a)).



A graph G is defined as a pair of two sets V and E denoted by G = (V, E), where V is set of vertices and E is set of edges.

A graph consists of vertices (nodes) and edges (connections between vertices). In this graph, vertices are labelled 0, 1, 2, and 3, and edges indicate which vertices are connected.

A graph is a collection of nodes (vertices) connected by edges. The graph given below has vertices labeled 0, 1, 2, and 3, with edges connecting them as follows:

- 0 is connected to 1, 2 and 3.
- 1 is connected to 0, 2 and 3.
- 2 is connected to 0, 1 and 3.
- 3 is connected to 0, 1 and 2.

Adjacency Matrix:

Adjacency Matrix: The adjacency matrix is a square matrix where the rows and columns represent the vertices. An entry in the matrix is 1 if there is an edge between the corresponding vertices and 0 otherwise.

	0	**1**	**2**	**3**
0	**0**	**1**	**1**	**1**
1	**1**	**0**	**1**	**1**
2	**1**	**1**	**0**	**1**
3	**1**	**1**	**1**	**0**

Adjacency List: The adjacency list is a collection of lists where each list corresponds to a vertex and contains the vertices to which it is connected.

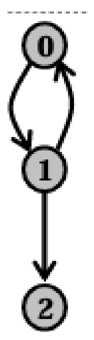
- 0: [1, 2, 3]
- 1: [0, 2, 3]
- 2: [0, 1, 3]
- 3: [0, 1, 2]

8 b. Define the following Terminologies with examples,

(i) Digraph

Definition: A graph G = (V, E) in which every edge is directed is called a directed graph. The directed graph is also called digraph.

Example: Consider the following graphs:



Here, graph G = (V, E) where

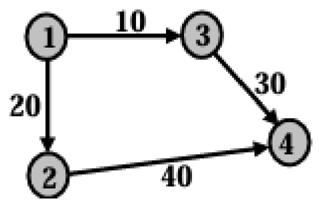
V = {0, 1, 2} is set of vertices

E = {<0, 1>, <1, 0>, <1, 2>} is set of edges

(ii) Weighted graph

Definition: A graph in which a number is assigned to each edge in a graph is called weighted graph. These numbers are called costs or weights. The weights may represent the cost involved or length or capacity depending on the problem.

Example:

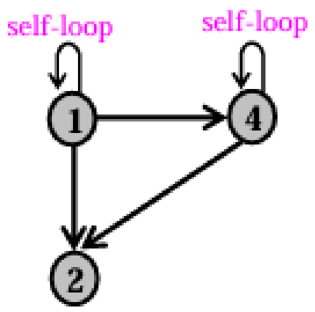


The following graph shown in figure the values 10, 20, 30 and 40 are the weights associated with four edges <1,3>, <1,2>, <3,4> and <2,4>.

(iii) Self-loop

Definition: A loop is an edge which starts and ends on the same vertex. A loop is represented by an ordered pair (i, i). This indicates that the edge originates and ends in the same vertex. A loop is also called self-edge or self-loop.

Example:

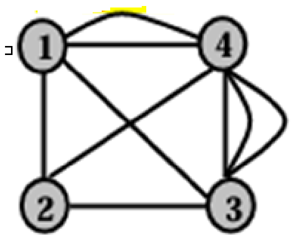


The given graph shown, there are two self-loops namely, <1 , 1> and <4 , 4> .

(iv) Parallel edges

Definition: A graph with multiple occurrence of the same edge between any two vertices is called multi-graph. It is also called Parallel edges.

Example: There are two edges between the nodes 1 and 4 and there are three edges between the nodes 4 and 3.



8 c. Explain in detail elementary graph operations.

Given an undirected graph, G = (V, £), and a vertex, v, in V(G) we wish to visit all vertices in G that are reachable from v, that is, ail vertices that are connected to v. There are two ways of doing this: depth first search and breadth first search.

Depth First Search:

Depth First Search is similar to a preorder tree traversal. We begin the search by visiting the start vertex, v. visiting consists of printing the node's vertex field. Next, we select an unvisited vertex, w, from v's adjacency list and carry out a depth first search on w. Eventually our search reaches a vertex, M, that has no unvisited vertices on its adjacency list. At this point, we remove a vertex from the stack and continue processing its adjacency list. Previously visited vertices are discarded; unvisited vertices are visited and placed on the stack. The search terminates when the stack is empty.

```
#define FALSE 0
 #define TRUE 1
 short int visited[MAX—VERTICES];
 void dfs(int v)
{
        node—pointer w;
        visited[v] = TRUE ;
         printf("%5d", v) ;
        for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex])
        dfs(w->vertex);
}
```

Program: Depth first search

Breadth First Search:

Breadth First Search resembles a level order tree traversal. Breadth first search starts at vertex v and marks it as visited. It then visits each of the vertices on v's adjacency list. When we have visited all the vertices on v's adjacency list, we visit all the unvisited vertices that are adjacent to the first vertex on v's adjacency list. To implement this scheme, as we visit each vertex we place the vertex in a queue. When we have exhausted an adjacency list, we remove a vertex from the queue and proceed by examining each of the vertices on its adjacency list. Unvisited vertices are visited and then placed on the queue; visited vertices are ignored. We have finished the search when the queue is empty.

The queue definition and the function prototypes used by bfs are:

```
typedef struct queue  *queue_pointer;
typedef struct queue
{
        int vertex;
        queue_pointer link;
};
void addq(queue_pointer *, queue_pointer *, int);
int deleteq(queue_pointer *);
```

```
void bfs(int v)
{
        node_pointer w;
        queue_pointer front, rear;
        front = rear = NULL;
        printf( “%5d”,v);
        visited[v] = TRUE ;
        addq(&front, &rear, v);
        while (front)
        {
                v = deleteq(&front);
                for (w = graph[v]; w; w = w—>link)
                if (!visited[w->vertex])
                {
                        printf("%5d", w->vertex);
                        addq(&frent,&rear,w->vertex);
                        visited[w->vertex] = TRUE ;
                )
        }
 }
```

Program: Breadth first search of a graph

9 a. What is collision? What are the methods to resolve collision? Explain linear probing with an example.

A **collision** occurs in a hash table when two or more elements hash to the same index or location in the table. Hash tables use a hash function to map keys to specific indices (or slots) in an array. If two keys map to the same index, a collision happens, as only one key-value pair can occupy each index.

When a collision occurs, the hash table must handle it properly to maintain efficient operations like insertion, deletion, and lookup.

Methods to Resolve Collisions

Several methods exist for handling collisions in hash tables. The primary methods include:

1. **Chaining**:
    - In chaining, each slot in the hash table contains a linked list (or other data structure) of entries that hash to the same index. When a collision occurs, the new entry is simply added to the list at the corresponding index.

    - **Example**: Inserting the value `10` at index `5`, then inserting `20` at index `5`. Both values will be linked in a list at index `5`.

2. **Open Addressing**:
    - In open addressing, when a collision occurs, the algorithm searches for another available slot within the table itself (instead of using a separate list). The most common methods of open addressing are:
        - **Linear Probing**
        - **Quadratic Probing**
        - **Double Hashing**
3. **Rehashing**:
    - Rehashing involves resizing the hash table and recalculating the hash function when the table reaches a certain load factor (the ratio of the number of elements to the table size). This reduces the likelihood of collisions.

Linear Probing

Linear probing is a collision resolution technique used in open addressing. When a collision occurs at a given index, linear probing searches for the next available slot in the hash table by checking subsequent positions in a linear sequence (i.e., it checks the next index, and if that is occupied, it checks the index after that, and so on).

How Linear Probing Works:

1. **Insertion**:
    - Compute the hash index using the hash function.
    - If the index is occupied, check the next index (increment by 1) until an empty slot is found.
    - Place the element in the first empty slot.
2. **Search**:
    - Compute the hash index for the key.
    - If the key is at that index, return the value.
    - If the key is not at the index, move to the next index and continue checking until the key is found or an empty slot is encountered (indicating the key is not in the table).
3. **Deletion**:
    - Find the key using the search method.
    - If found, delete the key and mark the slot as deleted (sometimes a special marker is used to indicate that the slot was previously occupied).

Suppose we have a hash table of size 7, and we want to insert the following keys: [50,700,76,85,92,73,101]

Step-by-Step Process:

1. **Insert 50**:
   - Hash index: 50%7=1.
   - Index 1 is empty, so insert 50 at index 1.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Value	**-**	**50**	**-**	**-**	**-**	**-**	**-**

2. **Insert 700**:
   - Hash index: 700%7=0.
   - Index 0 is empty, so insert 700 at index 0.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Value	**700**	**50**	**-**	**-**	**-**	**-**	**-**

3. **Insert 76**:
   - Hash index: 76%7=6.
   - Index 6 is empty, so insert 76 at index 6.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Value	**700**	**50**	**-**	**-**	**-**	**-**	**76**

4. **Insert 85**:
   - Hash index: 85%7=1.
   - Index 1 is occupied (collision with 50), so check the next index (2).
   - Index 2 is empty, so insert 85 at index 2.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Value	**700**	**50**	**85**	**-**	**-**	**-**	**76**

5. **Insert 92**:
   - Hash index: 92%7=1.
   - Index 1 is occupied, so check index 2 (also occupied).
   - Check index 3, which is empty, so insert 92 at index 3.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Value	**700**	**50**	**85**	**92**	**-**	**-**	**76**

6. **Insert 73**:
   - Hash index: 73%7=3.
   - Index 3 is occupied, so check the next index (4).
   - Index 4 is empty, so insert 73 at index 4.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Value	**700**	**50**	**85**	**92**	**73**	**-**	**76**

7. **Insert 73**:
   - Hash index: 73%7=3.
   - Index 3 is occupied, so check the next index (4).
   - Index 4 is empty, so insert 73 at index 4.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Value	**700**	**50**	**85**	**92**	**73**	**101**	**76**

Final Hash Table:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Value	**700**	**50**	**85**	**92**	**73**	**101**	**76**

9 b. Explain in detail, about static and dynamic hashing.

Hashing is a technique used to map keys to positions (or indices) in a hash table for fast data retrieval. The approach to managing the size and structure of the hash table defines whether the hashing method is **static** or **dynamic**.

Static Hashing

In static hashing, the size of the hash table is fixed when the hash table is created and does not change throughout its usage. All keys are mapped to one of the fixed number of slots in the hash table.

Characteristics of Static Hashing:

1. **Fixed Table Size**: The size of the hash table remains constant.
2. **Fixed Hash Function**: A single hash function is used to compute the index.
3. **Collision Handling**:
    - Collisions (two keys mapping to the same index) are handled using techniques such as:
        - **Chaining**: Each slot in the table points to a linked list of keys.
        - **Open Addressing**: Probes sequentially (e.g., linear probing, quadratic probing) to find an empty slot.

Advantages:

- Simple to implement.
- Efficient when the number of keys is known and does not exceed the table size.

Disadvantages:

- If the table is too small:
    - High collision rate leads to degraded performance.
- If the table is too large:
    - Wasted memory.

Example:

Consider a hash table of size 5 and keys {12,44,13,88,23}, with the hash function Index=Key%5.

Key	**Index**	**Collision Resolution (Chaining)**
12	2	Insert at index 2
44	4	Insert at index 4
13	3	Insert at index 3
88	3	Append to index 3 chain
23	3	Append to index 3 chain

Dynamic Hashing

Dynamic hashing adjusts the size of the hash table dynamically based on the number of keys. It allows the hash table to grow or shrink to accommodate the data without rehashing all the existing keys.

Characteristics of Dynamic Hashing:

1. **Scalable Table Size**: Hash table size grows or shrinks as needed.
2. **Hash Function Adjustments**: The hash function changes or adapts to map keys to the growing or shrinking table.
3. **Two Main Types**:
    - **Extendible Hashing**: Uses a directory of pointers to buckets. The directory doubles when needed.
    - **Linear Hashing**: Gradually increases the size of the hash table by splitting buckets incrementally.

Advantages:

- No need to predefine table size.
- Efficient handling of growing datasets.
- Minimizes collisions as the table expands.

Disadvantages:

- Implementation is more complex.
- Overhead of maintaining dynamic structures like directories in extendible hashing.

Example of Extendible Hashing:

1. Start with a hash table with 2 buckets and a single-bit hash:
   - Keys: {4,8,12,20}
   - Hash Function: Use the last bit of the key for initial placement.
2. Initial table (1-bit hash):
   - Bucket 0: {4,12}
   - Bucket 1: {8,20}
3. Insert 28:
   - Hash of 28 (last bit = 0) maps to Bucket 0. Bucket 0 is full.
   - Double the directory (2-bit hash) and split Bucket 0:
     - Bucket 00: {4,28}
     - Bucket 10: {12}

Comparison: Static vs Dynamic Hashing

Aspect	Static Hashing	Dynamic Hashing
Table Size	Fixed	Expands or shrinks as needed
Flexibility	Rigid	Flexible
Collision Handling	Uses chaining or open addressing	Reduces collisions dynamically
Performance	Degrades with collisions	Better performance for large data
Memory Usage	May waste space or lack space	Adapts to data size
Implementation	Simple	Complex

9 c. Discuss Leftist Trees with an example.

A **Leftist Tree** is a type of binary tree primarily used to implement priority queues. It is structured to keep its left subtree "heavier" or more "left-leaning" than its right subtree to minimize the work during merging operations.

Key Properties of Leftist Trees:

1. **Binary Tree**: A Leftist Tree is a binary tree where each node has at most two children.
2. **Heap Property**:
   - It follows the **min-heap property**: The key in any node is smaller than or equal to the keys of its children.

3. **Null Path Length (NPL)**:
   - The **Null Path Length (NPL)** of a node is the shortest path from the node to a leaf or a null child.
   - For a null node, NPL=−1.
   - For any non-null node, NPL=1+min(NPL of left child,NPL of right child).
4. **Leftist Property**:
   - The NPL of the left child is greater than or equal to the NPL of the right child for every node. This ensures the tree is left-leaning.
5. **Applications**:
   - Leftist Trees are commonly used for **priority queues** because of their efficient merging operations.

Operations on Leftist Trees:

1. **Merge**:
   - Combine two Leftist Trees into one while maintaining the Leftist and heap properties.
   - Algorithm:
     - Recursively merge the two trees starting from the roots.
     - Ensure the smaller key becomes the root.
     - Swap the left and right children if necessary to maintain the Leftist property.
2. **Insertion**:
   - Insert a new element by merging the new node with the existing tree.
3. **Deletion**:
   - Remove the root by merging its left and right subtrees.

Example

Let’s construct and operate on a Leftist Tree:

Step 1: Insert Elements

Insert the elements [3,10,8,21,14,17,23,26] one by one.

1. **Insert 3**:

   The tree contains only one node:

```
3 (NPL = 0)
```

2. **Insert 10**: Merge the new node (10) with the existing tree:

```
 3
  \
 10
```

   Swap the left and right children to maintain the Leftist property:

```
   3
  /
10
```

3. **Insert 8**: Merge 8 with the existing tree:

```
     3
    / \
  10   8
```

   The NPL of 10 (0) is not less than the NPL of 8 (0), so no swap is needed.

4. **Insert 21**: Merge 21 with the tree:

```
     3
    / \
  10   8
        \
        21
```

   Swap 8's children to maintain the Leftist property:

```
     3
    / \
  10   8
      /
     21
```

Step 2: Null Path Lengths

After building the tree, the **NPL** of each node is computed. The NPL helps maintain the Leftist property during merges.

Final Tree Example

For the given elements, the final Leftist Tree may look like this:

```
       3
      / \
     8   10
    / \
  21 14
  /
 17
```

- **Heap Property**: Each parent node is smaller than its children.
- **Leftist Property**: For every node, the NPL of the left child is greater than or equal to the NPL of the right child.

Advantages of Leftist Trees:

1. **Efficient Merging**:
   - Merging two Leftist Trees takes O(log n) time.
2. **Priority Queue Operations**:
   - Insert: O(log n)
   - Delete Min: O(log n)

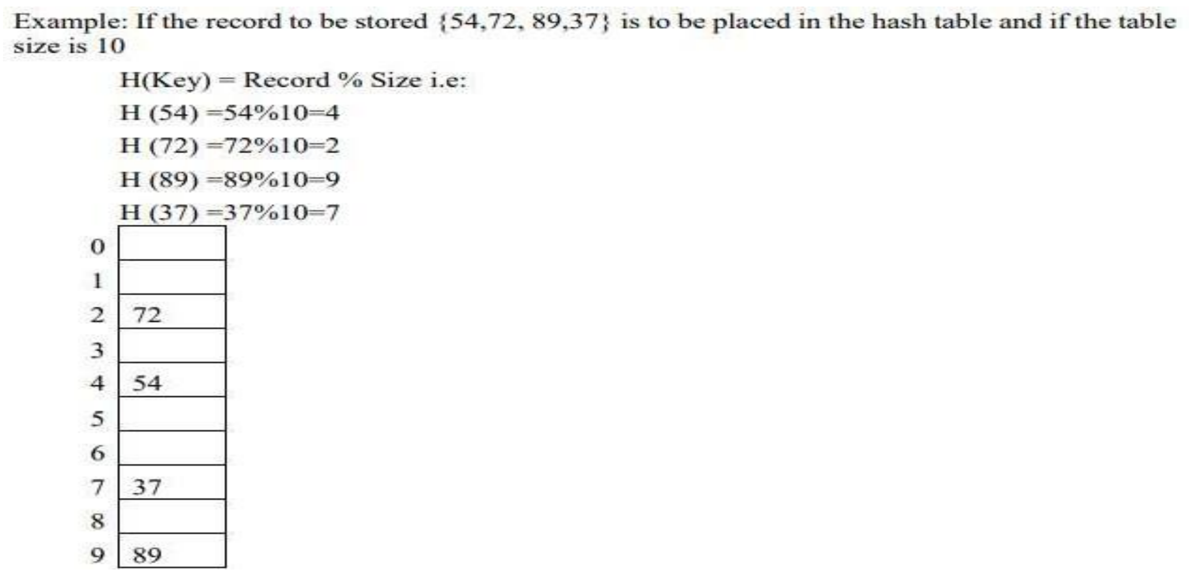
Applications:

- Implementing **priority queues**.
- Use cases where frequent merging of heaps is required, such as in event simulation or job scheduling systems.

10 a. Explain different types of HASH function with example.

There are 4 types of Hash Function:

1. **Division Method:** It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as $h(x) = x \bmod M$ The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M. Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values.



Example: If the record to be stored {54,72, 89,37} is to be placed in the hash table and if the table size is 10

H(Key) = Record % Size i.e:
H (54) =54%10=4
H (72) =72%10=2
H (89) =89%10=9
H (37) =37%10=7

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

2. **Mid-Square Method:**
   - Here, the key K is squared. A number ‘l’ in the middle of K^2 is selected by removing the digits from both ends. H(k)=l.
   - **Example 1:**
     Solution: Let key=2345, Its square is K^2 =574525

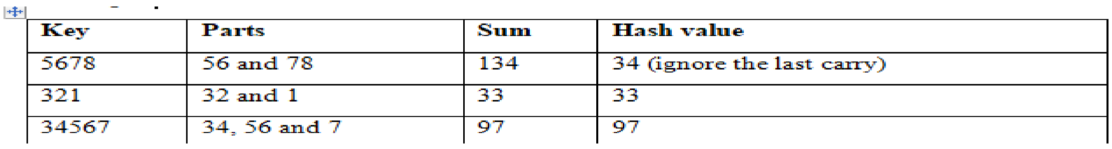
     H (2345) =45=>by discarding 57 and 25
   - **Example 2:** Calculate the hash value for keys 1234 using the mid-square method. The hashtable has 100 memory locations.
     Solution: Note that the hash table has 100 memory locations whose indices vary from 0 to 99.This means that only two digits are needed to map the key to a location in the hash table. When k = 1234, k^2 = 1522756, h (1234) = 27.

3. **Folding Method:**

   Step 1: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

   Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table.



Key	Parts	Sum	Hash value
5678	56 and 78	134	34 (ignore the last carry)
321	32 and 1	33	33
34567	34, 56 and 7	97	97

4. **Digit Analysis**

Here we make a statistical analysis of digits of the key, and select those digits (of fixed position) which occur quite frequently.

For example, if the key is : 9861234.

Here third and fifth position digits occur quite frequently, then we choose the digits in these positions from the key.

So we get, 62.

Reversing it we get 26 as the address.

10 b. Discuss AVL tree with an example. Write a function for insertion into an AVL Tree.

An **AVL Tree** (Adelson-Velsky and Landis Tree) is a type of self-balancing binary search tree where the difference in height (balance factor) of the left and right subtrees of any node is at most 1.

Key Features of AVL Trees:

1. **Balance Factor**:
   - Balance Factor of a node = Height of left subtree - Height of right subtree.
   - It must always be -1, 0, or 1 for an AVL tree.
2. **Self-balancing**:
   - After every insertion or deletion, the tree is rebalanced if needed using rotations.
3. **Rotations**:
   - **Single Rotations**:
     - Left Rotation (LL imbalance)
     - Right Rotation (RR imbalance)
   - **Double Rotations**:
     - Left-Right Rotation (LR imbalance)
     - Right-Left Rotation (RL imbalance)

Example of an AVL Tree:

Let's insert values: **10, 20, 30, 40, 50**.

1. **Insert 10:**

2. **Insert 20:**

```
    10
      \
       20
```

3. **Insert 30 (causes imbalance at root, single left rotation needed):**

```
  10               20
     \       =>     /  \
      20         10   30
        \
         30
```

4. **Insert 40:**

```
    20
   /  \
 10    30
         \
          40
```

5. **Insert 50 (causes imbalance at node 30, single left rotation needed):**

```
    20
   /  \
 10    40
      /  \
     30   50
```

Insertion Function for AVL Tree in Python:

```
void avl—insert(tree—pointer *parent, element x, int *unbalanced)

{

   if ( !* parent )
   {
         *unbalanced = TRUE ;
             *parent = (tree—pointer)
                malloc(sizeof(tree_node));
          if (IS—FULL(^parent))
         {
                fprintf(stderr, "The memory is full\n”);

                exit(1);
         }
             (*parent) -> left—child = (* parent )->right—child = NULL;
             ( * parent )->bf =  0;
         (* parent )->data= X;
       }
   else if (x.key <  (* parent )->data.key)
   {
         avl—insert (& ( *parent) ->left—child, x, unbalanced) ;
         if (* unbalanced)
         switch ((^parent)->bf)
         {
```

```
                    case -1: (* parent )->bf = 0;
                             *unbalanced = FALSE;
                             Break;
                        case 0 : (* parent )->bf = 1;
                             break;
                    case 1: : left_rotation(parent,unbalanced);
            }

    }

        else if (x.key > (* parent )->data.key)
        {
            avl—insert(&(*parent)->right —child, x, unbalanced);
                if (* unbalanced)
                switch((^parent)->bf)
            {
                    case 1 : (* parent )->bf = 0;
                             *unbalanced = FALSE;
                             Break;
                    case 0 : (* parent )->bf = -1;
                              break;
                        case -1: right_rotation(parent, unbalanced);
            }
        }
        else
        {
             *unbalanced = FALSE;
                  printf("The key is already in the tree");
        }

  }
```

Program : Insertion into an AVL tree

10 c. Define Red-black Tree, Splay tree. Discuss the method to insert an element into Red-Black tree.

Red-Black Tree

A **Red-Black Tree** is a self-balancing binary search tree (BST) that ensures the tree remains balanced by following specific rules.

Properties:

1. **Node Color**: Each node is either red or black.
2. **Root is Black**: The root node must always be black.
3. **Red Rule**: A red node cannot have a red parent or red child (no two consecutive red nodes).
4. **Black Height**: Every path from a node to its descendant leaf must contain the same number of black nodes.

5. **Leaf Nodes**: All leaf nodes (NIL or NULL) are black.

These rules help maintain a tree height of O(log⁡n)O(\log n)O(logn), ensuring efficient operations like search, insertion, and deletion.

Splay Tree

A **Splay Tree** is a self-adjusting binary search tree where recently accessed elements are moved to the root. This optimizes for sequences of operations that repeatedly access the same elements.

Key Features:

1. **Splaying**: The operation of moving an accessed node to the root using rotations.
   - **Zig Rotation**: Single rotation if the node is a child of the root.
   - **Zig-Zag Rotation**: Double rotation if the node is a grandchild and alternates in direction with its parent.
   - **Zig-Zig Rotation**: Double rotation if the node is a grandchild and is in the same direction as its parent.
2. **Amortized Complexity**: Each operation takes O(log⁡n)O(\log n)O(logn) on average, even though individual operations might take longer.

Insertion into a Red-Black Tree

To insert an element into a Red-Black Tree, we perform a standard BST insertion followed by "fix-up" operations to maintain Red-Black Tree properties.

Steps for Insertion:

1. **Standard BST Insertion**:
   - Insert the new node into the tree as you would in a regular BST.
   - Assign the new node the color **red**.
2. **Fix Violations**:
   - If the parent of the inserted node is black, no fix is needed.
   - If the parent is red (violates the Red Rule), perform one of the following:
     - **Case 1**: **Uncle is red**:
       - Recolor the parent, uncle, and grandparent.
       - Move the pointer to the grandparent and repeat.
     - **Case 2**: **Uncle is black**:
       - Perform rotations to fix the tree:
         - **Left-Left (LL)** or **Right-Right (RR)**: Single rotation.
         - **Left-Right (LR)** or **Right-Left (RL)**: Double rotation.
   - Ensure the root remains black.