

Model Question Paper-I with effect from 2023-24 (CBCS Scheme)

USN

--	--	--	--	--	--	--	--	--	--

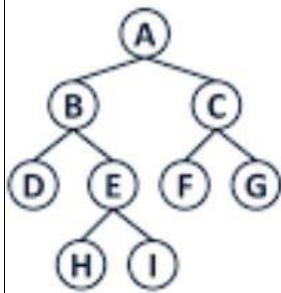
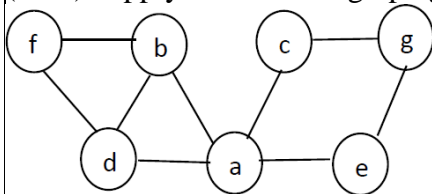
Third Semester B.E. Degree Examination Data Structures and Applications

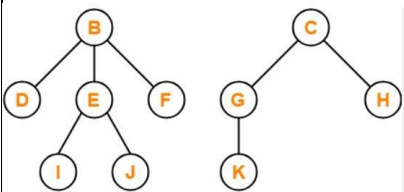
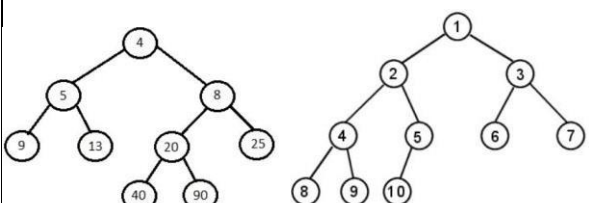
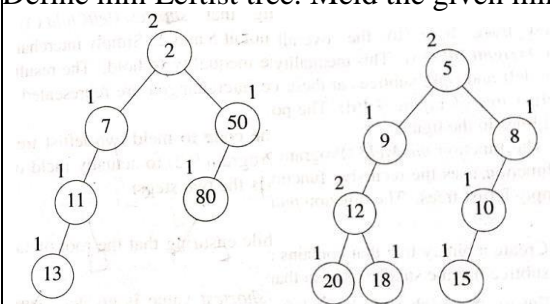
TIME: 03 Hours

Max. Marks: 100

Note: 01. Answer any **FIVE** full questions, choosing at least **ONE** question from each **MODULE**.

Module -1			*Bloom's Taxonomy Level	Marks
Q.01	a	Define data structures. With a neat diagram, explain the classification of data structures with examples.	L2	5
	b	What do you mean by pattern matching? Outline the Knuth Morris Pratt (KMP) algorithm and illustrate it to find the occurrences of the following pattern. P: ABCDABD S: ABC ABCDAB ABCDABCDABDE	L3	8
	c	Write a program in C to implement push, pop and display operations for stacks using arrays.	L3	7
OR				
Q.02	a	Explain in brief the different functions of dynamic memory allocation.	L2	5
	b	Write functions in C for the following operations without using built-in functions Compare two strings. Concatenate two strings. Reverse a string	L3	8
	c	Write a function to evaluate the postfix expression. Illustrate the same for the given postfix expression: ABC-D*+E\$F+ and assume A=6, B=3, C=2, D=5, E=1 and F=7.	L3	7
Module-2				
Q.03	a	Develop a C program to implement insertion, deletion and display operations on Linear queue.	L3	10
	b	Write a program in C to implement a stack of integers using a singly linked list.	L3	10
OR				
Q.04	a	Write a C program to implement insertion, deletion and display operations on a circular queue.	L3	10
	b	Write the C function to add two polynomials. Show the linked representation of the below two polynomials and their addition using a circular singly linked list P1: $5x^3 + 4x^2 + 7x + 3$ P2: $6x^2 + 5$ Output: add the above two polynomials and represent them using the linked list.	L3	10

Module-3																																																																			
Q. 05	a	Write recursive C functions for inorder, preorder and postorder traversals of a binary tree. Also, find all the traversals for the given tree. 	L3	8																																																															
	b	Write C functions for the following Search an element in the singly linked list. Concatenation of two singly linked list	L2	6																																																															
	c	Define Sparse matrix. For the given sparse matrix, give the linked list representation: $A = \begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$	L3	6																																																															
OR																																																																			
Q. 06	a	Write C Functions for the following i) Inserting a node at the beginning of a Doubly linked list Deleting a node at the end of the Doubly linked list	L3	8																																																															
	b	Define Binary tree. Explain the representation of a binary tree with a suitable example.	L2	6																																																															
	c	Define the Threaded binary tree. Construct Threaded binary for the following elements: A, B, C, D, E, F, G, H, I	L3	6																																																															
Module-4																																																																			
Q. 07	a	Design an algorithm to traverse a graph using Depth First Search (DFS). Apply DFS for the graph given below. 	L3	8																																																															
	b	Construct a binary tree from the Post-order and In-order sequence given below In-order: GDHBAEICF Post-order: GHDBIEFCA	L2	6																																																															
	c	Define selection tree. Construct min winner tree for the runs of a game given below. Each run consists of values of players. Find the first 5 winners. <table data-bbox="244 1787 748 2065"><tr><td>1</td><td>9</td><td>2</td><td>6</td><td>8</td><td>9</td><td>9</td><td>1</td></tr><tr><td>0</td><td></td><td>0</td><td></td><td></td><td></td><td>0</td><td>7</td></tr><tr><td>1</td><td>2</td><td>2</td><td>1</td><td>1</td><td>1</td><td>9</td><td>1</td></tr><tr><td>5</td><td>0</td><td>0</td><td>5</td><td>5</td><td>1</td><td>5</td><td>8</td></tr><tr><td>1</td><td>3</td><td>3</td><td>2</td><td>5</td><td>1</td><td>9</td><td>2</td></tr><tr><td>6</td><td>8</td><td>0</td><td>5</td><td>0</td><td>6</td><td>9</td><td>0</td></tr><tr><td colspan="3"></td><td>2</td><td colspan="4"></td></tr><tr><td colspan="3"></td><td>8</td><td colspan="4"></td></tr></table>	1	9	2	6	8	9	9	1	0		0				0	7	1	2	2	1	1	1	9	1	5	0	0	5	5	1	5	8	1	3	3	2	5	1	9	2	6	8	0	5	0	6	9	0				2								8					L2
1	9	2	6	8	9	9	1																																																												
0		0				0	7																																																												
1	2	2	1	1	1	9	1																																																												
5	0	0	5	5	1	5	8																																																												
1	3	3	2	5	1	9	2																																																												
6	8	0	5	0	6	9	0																																																												
			2																																																																
			8																																																																

OR				
Q. 08	a	Define Binary Search tree. Construct a binary search tree (BST) for the following elements: 100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145. Traverse using in-order, pre-order, and post-order traversal techniques. Write recursive C functions for the same.	L3	8
	b	Define Forest. Transform the given forest into a Binary tree and traverse using inorder, preorder and postorder traversal. 	L2	6
	c	Define the Disjoint set. Consider the tree created by the weighted union function on the sequence of unions: union(0,1), union(2,3), union(4,5), union(6,7), union(0,2), union(4,6), and union(0,4). Process the simple find and collapsing find on eight finds and compare which find is efficient.	L2	6
Module-5				
Q. 09	a	What is chained hashing? Discuss its pros and cons. Construct the hash table to insert the keys: 7, 24, 18, 52, 36, 54, 11, 23 in a chained hash table of 9 memory locations. Use $h(k) = k \text{ mod } m$.	L3	10
	b	Define the leftist tree. Give its declaration in C. Check whether the given binary tree is a leftist tree or not. Explain your answer. 	L2	5
	c	What is dynamic hashing? Explain the following techniques with examples: Dynamic hashing using directories Directory less dynamic hashing	L2	5
OR				
Q. 10	a	What is a Priority queue? Demonstrate functions in C to implement the Max Priority queue with an example. Insert into the Max priority queue Delete into the Max priority queue Display Max priority queue	L3	10
	b	Define min Leftist tree. Meld the given min leftist trees. 	L2	5
	c	Define hashing. Explain different hashing functions with examples. Discuss the properties of a good hash function.	L2	5

1a) Define data structures. With a neat diagram, explain the classification of data structures with examples.

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.

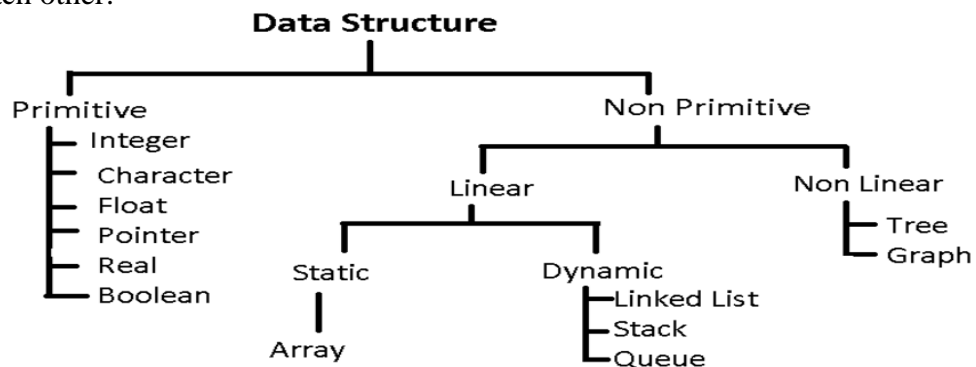


Figure: Classification of Data Structures

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) what do you mean by pattern matching? Outline the Knuth Morris Pratt (KMP) algorithm and illustrate it to find the occurrences of the following pattern.

Pattern: ABCDABD

String: ABC ABCDAB ABCDABCDABDE

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.

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

Text A B C A B C D A B A B C D A B C D A B D E

Pattern A B C D A B D

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.

Text A B C A B C D A B A B C D A B C D A B D E

Pattern A B C D A B D

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

Text A B C A B C D A B A B C D A B C D A B D E

Pattern A B C D A B D

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.

Text A B C A B C D A B A B C D A B C D A B D E

Pattern A B C D A B D

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.

Text A B C A B C D A B A B C D A B C D A B D E

Pattern A B C D A B D

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.

1c) Write a program in C to implement push, pop and display operations for stacks using arrays.

```
#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: ");
    for (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 a) Explain in brief the different functions of dynamic memory allocation

Dynamic memory allocation is the process of allocating memory during run time(execution time). Additional storage can be allocated whenever needed.

1. malloc():

Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.

Syntax:

ptr=(datatype *)malloc(sizeof(datatype); where,

- ptr is a pointer variable of type datatype
- datatype can be any of the basic datatype or user define datatype
- Size is number of bytes required.

Example:

```

int *p;
ptr=(int*)malloc(100*sizeof(int));

```

2. calloc ()

It stands for contiguous allocation. It is used to allocate multiple blocks of memory. It requires two parameters as number of elements and size of each element.

Syntax: ptr=(datatype *)calloc(n , sizeof(Datatype));

where, ptr is a pointer variable of type datatype datatype can be any of the basic datatype
n is number of blocks to be allocated size is number of bytes required

```

int *p;
ptr=(int*)calloc(100,sizeof(int));

```

3. realloc()

- It changes the size of block by deleting or extending the memory at end of the block.
- If memory is not available it gives complete new block.

Syntax:

ptr=(datatype *)realloc (ptr , sizeof(datatype); where,

- ptr is a pointer to a block previously allocated memory either using malloc() or calloc()
- Size is new size of the block.

```
int *p;
ptr=(int*)calloc(100,sizeof(int)); ptr=(int*)realloc(ptr,sizeof(int));
```

4. free()

This function is used to de-allocate(or free) the allocated block of memory which is allocated by using functions malloc(), calloc(), realloc().

Syntax:

```
free(ptr);
```

2b)Write functions in C for the following operations without using built-in functions

i)Compare two strings. ii) Concatenate two strings. iii) Reverse a string

i) Compare two strings

```
int scomp(char s1[ ], char s2[ ])
{
    int i, j;
    if(slen(s1) != slen(s2))
    {
        return 0;
    }
    for(i=0; s1[i] != '\0'; i++)
    {
        if(s1[i] != s2[i])
        {
            return 0;
        }
    }
    return 1;
}
```

ii) Concatenate two strings

```
int scat(char s1[ ], char s2[ ])
{
    int i, j;
    for(i = slen(s1), j=0; s2[j] != '\0'; i++, j++)
    {
        s1[i] = s2[j];
    }

    s1[i] = '\0';
    return 0;
}
```

iii) Reverse a string

```
void reverse()
{
    char str[100], temp;
    int i, j = 0;
    printf("Enter The String: ");
    gets(str);
    i = 0;
    j = strlen(str) - 1; while (i < j)
    {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp; i++;
        j--;
    }

    printf("\nReverse a String Is: %s\n\n", str);
}
```


2c) Write a function to evaluate the postfix expression. Illustrate the same for the given postfix expression: ABC-D*+E\$F+ and assume A=6, B=3, C=2, D=5, E=1 and F=7.

```
#include<stdio.h>
float compute(char symbol, float op1, float op2)
{
    switch (symbol)
    {
        case '+': return op1 + op2;
        case '-': return op1 - op2;
        case '*': return op1 * op2;
        case '/': return op1 / op2; case '$':
        case '^': return pow(op1,op2);
    }
}
void main()
{
    float s[20], res, op1, op2; int top, i;
    char postfix[20], symbol;
    printf("\nEnter the postfix expression:\n"); scanf ("%s", postfix);
    top=-1;
    for (i=0; i<strlen(postfix) ;i++)
    {
        symbol = postfix[i];
        if(isdigit(symbol))
            s[++top]=symbol - '0';
        else
        {
            op2 = s[top--];
            op1 = s[top--];
            res = compute(symbol, op1, op2);
            s[++top] = res;
        }
    }
    res = s[top--];
    printf("\nThe result is : %f\n", res);
}
```

To evaluate the given postfix expression ABC-D*+E\$F+ with the provided variable values A=6, B=3, C=2, D=5, E=1, and F=7, follow these steps:

Final Result:

Stack	Op2	Op1	Result = Op1 \oplus Op2
6			
6 3			
6 3 2	2	3	Result = 3 - 2 = 1
6 1 (Result) 5			
6 1 5	5	1	Result = 1 * 5 = 5
6 5 (Result)			
6 5	5	6	Result = 6 + 5 = 11
11 (Result)			
11 1	1	11	Result = 1 ^ 11 = 11
11			
11 7	7	11	Result = 11 + 7 = 18

The result of the postfix expression ABC-D*+E\$F+ is **18**.

3a Develop a C program to implement insertion, deletion and display operations on Linear queue.

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

#define Q_SIZE 5

int front = 0, rear = -1;
int queue[10];

// Function to insert an item into queue
void insert_rear ( int item )
{
    // Write the complete function
}

// Function to delete an element from queue
void delete_front ()
{
    // Write the complete function
}

// Function to display the contents of queue
void display ()
{
    // Write the complete function
}

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 );
        }
    }
}
```

// 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] );

    // Increment front by 1
    front = front + 1;
}
```

OR

```
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 " );
}
```

// Function to insert item into the queue

```
void insert_rear ( int item )
{
    // Check for overflow of queue
    if ( rear == Q_SIZE - 1 )
    {
        printf ( "Queue Overflow" );
        return;
    }
}
```

// Increment rear by 1
rear++;

// Insert an item into the queue
queue[rear] = item;

OR

// Insert an item into the queue
queue[++rear] = item;

3 b Write a program in C to implement a stack of integers using a singly linked list.

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
struct node
{
    int info;
    struct node *link;
};

typedef struct node* NODE;

//Function to get a new node from the operating system
NODE getnode()
{
    NODE x;
    x = ( NODE ) malloc(sizeof(struct node));
    if ( x == NULL )
    {
        printf("Out of memory\n");
        exit(0);
    }
    return x;
}

//Function to insert an item at the front end of the list
NODE insert_front(int item, NODE first)
{
    NODE temp;
    temp = getnode();
    temp->info = item;
    temp->link = first;
    return temp;
}

//Function to delete an item from the front end of the list
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;
}
```

```

//Function to display the contents of the list
void display(NODE first)
{
    NODE cur;
    if ( first == NULL )
    {
        printf("List is empty\n");
        return;
    }
    printf("The contents of singly linked list\n");

    cur = first;
    while ( cur != NULL )
    {
        printf("%d ",cur->info);
        cur = cur->link;
    }
    printf("\n");
}

void main()
{
    NODE first;
    int choice, item;
    first = NULL;
    for (;;)
    {
        printf("1:Insert_Front 2:Delete_Front\n");
        printf("3:Display 4:Exit\n");
        printf("Enter the choice\n");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter the item to be inserted\n");
                scanf("%d", &item);
                first = insert_front (item, first);
                break;
            case 2:
                first = delete_front(first);
                break;
            case 3:
                display(first);
                break;
            default:
                exit(0);
        }
    }
}

```

4 a Write a C program to implement insertion, deletion and display operations on a 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)
{
    if (count == Q_SIZE)
    {
        printf ("Queue Overflow");
        return;
    }
    rear = ( rear + 1 ) % Q_SIZE;
    queue[ rear ] = item;
    count++;
}

// Function to delete an element from queue
void delete_front()
{
    if (count == 0)
    {
        printf ("Queue Underflow");
        return;
    }
    printf ("Item deleted :%d", queue[front]);
    front = (front + 1) % Q_SIZE;
    count = count - 1;
}

// Function to display the contents of queue
void display ()
{
    int i, temp;
    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;
    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 b. Write the C function to add two polynomials. Show the linked representation of the below two polynomials and their addition using a circular singly linked list P1: $5x^3 + 4x^2 + 7x + 3$ P2: $6x^2 + 5$ Output: add the above two polynomials and represent them using the linked list.

```

// Function to add two polynomials
Node* addPolynomials(Node* p1, Node* p2)
{
    Node* result = NULL;
    Node* t1 = p1;
    Node* t2 = p2;

    do
    {
        if (t1->exp > t2->exp)
        {
            result = insertTerm(result, t1->coeff, t1->exp);
            t1 = t1->next;
        }
        else if (t1->exp < t2->exp)
        {
            result = insertTerm(result, t2->coeff, t2->exp);
            t2 = t2->next;
        }
        else
        {
            int sumCoeff = t1->coeff + t2->coeff;
            result = insertTerm(result, sumCoeff, t1->exp);
            t1 = t1->next;
            t2 = t2->next;
        }
    }
    while (t1 != p1 && t2 != p2);

    // Add remaining terms of p1
    while (t1 != p1)

```

```

    {
        result = insertTerm(result, t1->coeff, t1->exp);
        t1 = t1->next;
    }

    // Add remaining terms of p2
    while (t2 != p2)
    {
        result = insertTerm(result, t2->coeff, t2->exp);
        t2 = t2->next;
    }

    return result;
}

```

To represent and add the two polynomials using a circular singly linked list, follow these steps:

Circular singly linked list P1: $5x^3 + 4x^2 + 7x + 3$ P2: $6x^2 + 5$

Step 1: Define the polynomial structure

Each node in the circular singly linked list should store:

1. Coefficient: The coefficient of the term.
2. Exponent: The power of the variable.
3. Pointer: A link to the next node.

Step 2: Represent the polynomials

P1: Linked list representation:

Node 1: Coefficient = 5, Exponent = 3

Node 2: Coefficient = 4, Exponent = 2

Node 3: Coefficient = -7, Exponent = 1

Node 4: Coefficient = 3, Exponent = 0



P2: Linked list representation:

Node 1: Coefficient = 6, Exponent = 2

Node 2: Coefficient = 5, Exponent = 0



Step 3: Addition of polynomials

Combine the linked lists of and by:

1. Matching terms with the same exponent.
2. Adding their coefficients.
3. Creating a new linked list for the result.

Result:

Combined polynomial: Linked list representation:

Node 1: Coefficient = 5, Exponent = 3

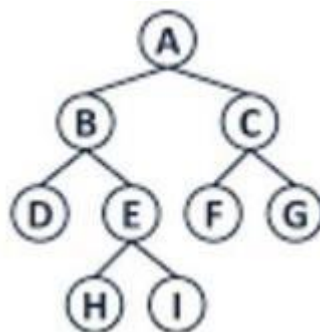
Node 2: Coefficient = 10, Exponent = 2

Node 3: Coefficient = -7, Exponent = 1

Node 4: Coefficient = 8, Exponent = 0



5 a. Write recursive C functions for inorder, preorder and postorder traversals of a binary tree. Also, find all the traversals for the given tree.



Function to traverse the binary tree in inorder	Function to traverse the binary tree in preorder	Function to traverse the binary tree in postorder
<pre> void inorder(NODE root) { if (root == NULL) return; inorder(root->llink); printf("%d ",root->info); inorder(root->rlink); } </pre>	<pre> void preorder(NODE root) { if (root == NULL) return; printf("%d ",root->info); preorder(root->llink); preorder(root->rlink); } </pre>	<pre> void postorder(NODE root) { if (root == NULL) return; postorder(root->llink); postorder(root->rlink); printf("%d ", root->info); } </pre>

In order : D B H E I A F C G

Pre order : A B D E H I C F G

Post order : D H I E B F G C A

5 b. Write C functions for the following i) Search an element in the singly linked list. ii) Concatenation of two singly linked lists.

// Function to search for an element in a singly linked list	// Function to concatenate two singly linked lists
<pre> void search(struct node *head,int key) { struct node *temp = head; while(temp != NULL) { if(temp->data == key) printf("key found"); temp = temp->next; } printf("key not found"); } </pre>	<pre> void Concat(struct Node *first, struct Node *second) { struct Node *p = first; while (p->next != NULL) { p = p->next; } p->next = second; second = NULL; } </pre>

5 c. Define Sparse matrix. For the given sparse matrix, give the linked list representation:

$$A = \begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

A sparse matrix is a matrix in which most of the elements are zero. In contrast, a dense matrix has most elements as non-zero. Sparse matrices are often used to save memory and computational resources when dealing with large matrices where many elements are zero.

For the given matrix A:

$$A = \begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

The non-zero elements are:

3 at position (1,3)

4 at position (1,5)

5 at position (2,3)

7 at position (2,4)

2 at position (4,2)

6 at position (4,3)

Linked List Representation

In a linked list representation of a sparse matrix, each non-zero element is represented as a node containing:

- The value of the element
- The row index
- The column index
- A pointer to the next node

The linked list for the given matrix would be:

1. Node: Value = 3, Row = 1, Column = 3, Next = Pointer to next node
2. Node: Value = 4, Row = 1, Column = 5, Next = Pointer to next node
3. Node: Value = 5, Row = 2, Column = 3, Next = Pointer to next node
4. Node: Value = 7, Row = 2, Column = 4, Next = Pointer to next node
5. Node: Value = 2, Row = 4, Column = 2, Next = Pointer to next node
6. Node: Value = 6, Row = 4, Column = 3, Next = Null (end of list)

Final Answer:

The linked list representation of the sparse matrix A is:

(3, 1, 3) -> (4, 1, 5) -> (5, 2, 3) -> (7, 2, 4) -> (2, 4, 2) -> (6, 4, 3) -> Null

6 a. Write C Functions for the following

- i) **Inserting a node at the beginning of a Doubly linked list**
- ii) **Deleting a node at the end of the Doubly linked list**

Inserting a node at the beginning	Deleting a node at the end
<pre> NODE insert_front(int item, NODE first) { NODE temp; temp = getnode(); temp->info = item; temp->link = temp->rlink = NULL; if (first == NULL) return temp; temp->rlink = first; first->llink = temp; return temp; } </pre>	<pre> NODE delete_rear(NODE first) { NODE cur, prev; if (first == NULL) { printf("List is empty \n"); return NULL; } if (first->rlink == NULL) { printf("item deleted = %d\n", first->info); free(first); return NULL; } cur = first; while(cur->rlink != NULL) cur = cur->rlink; prev = cur -> llink; prev->rlink = NULL; printf("item deleted = %d\n", cur -> info); free(cur); return first; } </pre>

6 b. Define Binary tree. Explain the representation of a binary tree with a suitable example.

A binary tree is a tree which has finite set of nodes that is either empty or consist of a root and two subtrees called left subtree and right subtree. A binary tree can be partitioned into three subgroups namely root, left subtree and right subtree.

- **Root** – If tree is not empty, the first node in the tree is called root node.
- **left subtree** – It is a tree which is connected to the left of root. Since this tree comes towards left of root, it is called left subtree.
- **right subtree** – It is a tree which is connected to the right of root. Since this tree comes towards right of root, it is called right subtree.

The storage representation of binary trees can be classified as shown below:

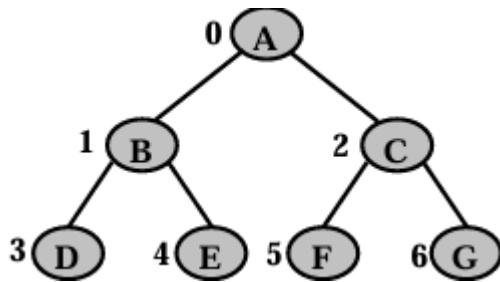
- Array representation (uses static allocation technique)
- Linked representation (uses dynamic allocation technique)

(i) Array Representation:

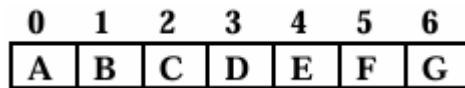
In an **array representation**, the tree is stored as a complete binary tree (filled level-by-level from left to right). Each node's position in the array is mapped based on its relationship to its parent and children.

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 $\lfloor i-1/2 \rfloor$.



Tree

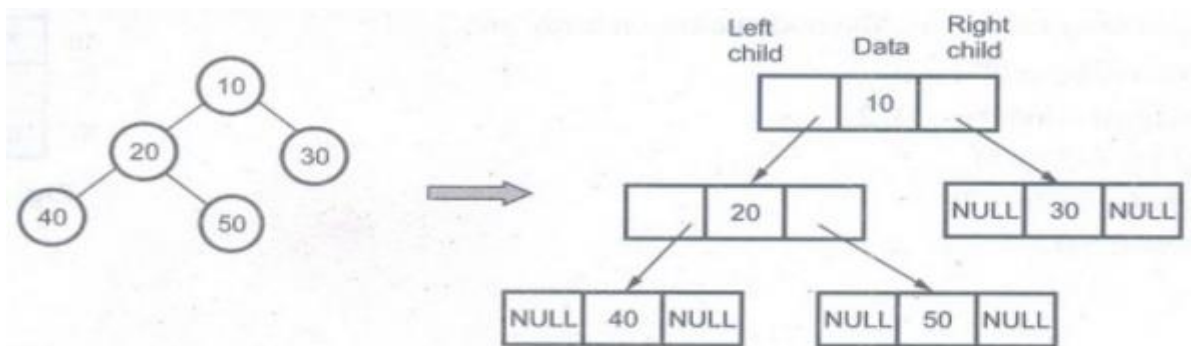


Array Representation

(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 c. Define the Threaded binary tree. Construct Threaded binary for the following elements: A, B, C, D, E, F, G, H, I.

A **Threaded Binary Tree** is a type of binary tree where the **null pointers** (or empty child pointers) are replaced with **threads** to make the tree traversal more efficient. The threading allows faster in-order, pre-order, and post-order traversals without needing a stack or recursion.

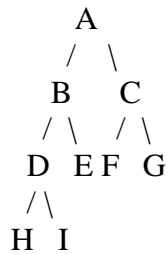
- **Right Threaded Binary Tree:** In this version of the tree, each node's right child is threaded to the node's **in-order successor** (the next node in an in-order traversal), if there is no right child.

- Left Threaded Binary Tree:** In this version of the tree, each node's left child is threaded to the node's **in-order predecessor** (the previous node in an in-order traversal), if there is no left child.

Elements: A, B, C, D, E, F, G, H, I

Let’s build a **Right Threaded Binary Tree** and a **Left Threaded Binary Tree**.

Assume the elements form a **complete binary tree**:

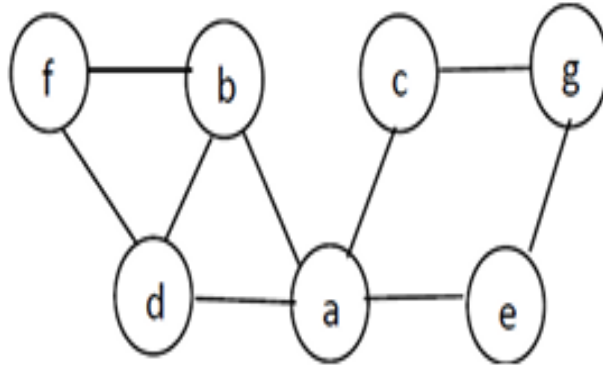


Inorder Traversal:

Inorder traversal is: H,D,I,B,E,A,F,C,G

<div> <div> Right Threaded Binary Tree </div> <div> Steps: <ul style="list-style-type: none"> H: Right child → D (successor). D: Right child → I (successor). I: Right child → B (successor). B: Right child → E (successor). E: Right child → A (successor). A: Right child → F (successor). F: Right child → C (successor). C: Right child → G (successor). G: Right child → null (no successor). </div> </div>		<div> <div> Left Threaded Binary Tree </div> <div> Steps: <ul style="list-style-type: none"> H: Left child → null (no predecessor). D: Left child → H (predecessor). I: Left child → D (predecessor). B: Left child → I (predecessor). E: Left child → B (predecessor). A: Left child → E (predecessor). F: Left child → A (predecessor). C: Left child → F (predecessor). G: Left child → C (predecessor). </div> </div>	
<div> <div> Threaded Trees Visualization Right Threaded </div> <div> <div> Binary Tree: <pre> graph TD A --> B A --> C B --> D B --> E C --> F C --> G D --> H D --> I </pre> </div> <div> Threads: <ul style="list-style-type: none"> - H -> D - D -> I - I -> B - B -> E - E -> A - A -> F - F -> C - C -> G </div> </div> </div>		<div> <div> Threaded Trees Visualization Left Threaded </div> <div> <div> Binary Tree: <pre> graph TD A --> B A --> C B --> D B --> E C --> F C --> G D --> H D --> I </pre> </div> <div> Threads: <ul style="list-style-type: none"> - D <- H - I <- D - B <- I - E <- B - A <- E - F <- A - C <- F - G <- C </div> </div> </div>	

- 7 a. Design an algorithm to traverse a graph using Depth First Search (DFS). Apply DFS for the graph given below.



Depth First Search: Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search

DFS Algorithm :

- Step 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
 Step 2 – If no adjacent vertex is found, pop up a vertex from the stack. (which do not have adjacent vertices.)
 Step 3 – Repeat Rule 1 and Rule 2 until the stack is empty

Now, the graph can be traversed using DFS as shown in following table

	Stack	$v = \text{adj}(s[\text{top}])$	Nodes visited S	pop(stack)
Initial step	a	-	a	
Stage 1	a	b	a, b	-
Stage 2	a, b	d	a, b, d	-
Stage 3	a, b, d	f	a, b, d, f	-
Stage 4	a, b, d, f	-	a, b, d, f	f
Stage 5	a, b, d	-	a, b, d, f	d
Stage 6	a, b	-	a, b, d, f	b
Stage 7	a	c	a, b, d, f	-
Stage 8	a, c	g	a, b, d, f, g	-
Stage 9	a, c, g	e	a, b, d, f, g, e	-
Stage 10	a, c, g, e	-	a, b, d, f, g, e	e
Stage 11	a, c, g	-	a, b, d, f, g, e	g
Stage 12	a, c	-	a, b, d, f, g, e	c
Stage 13	a ₁	-	a, b, d, f, g, e	a _{1,7}

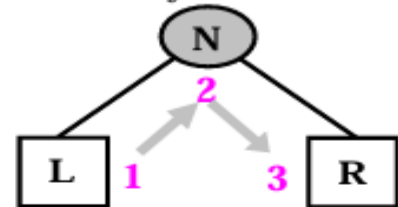
7 b. Construct a binary tree from the Post-order and In-order sequence given below

In-order: GDHBAEICF Post-order: GHDBIEFCA

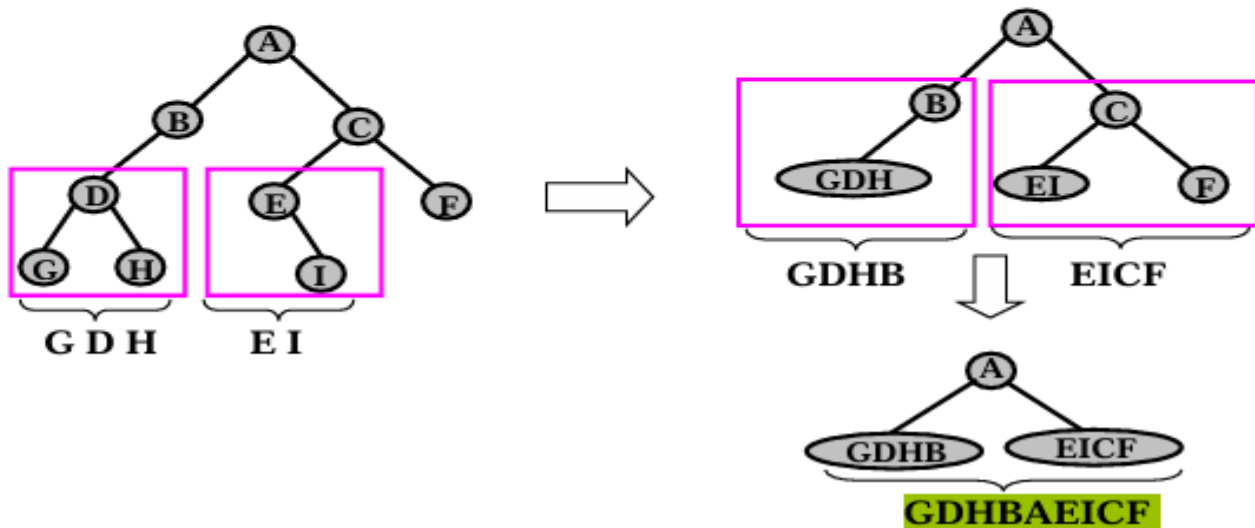
In-order: GDHBAEICF:

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

1. Traverse the **Left** subtree in inorder [**L**]
2. Process the root **Node** [**N**]
3. Traverse the **Right** subtree in inorder [**R**]



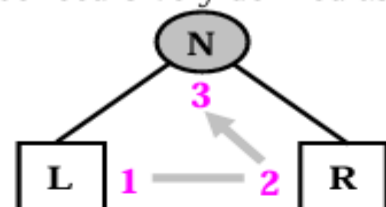
For example, let us traverse the following tree in preorder.



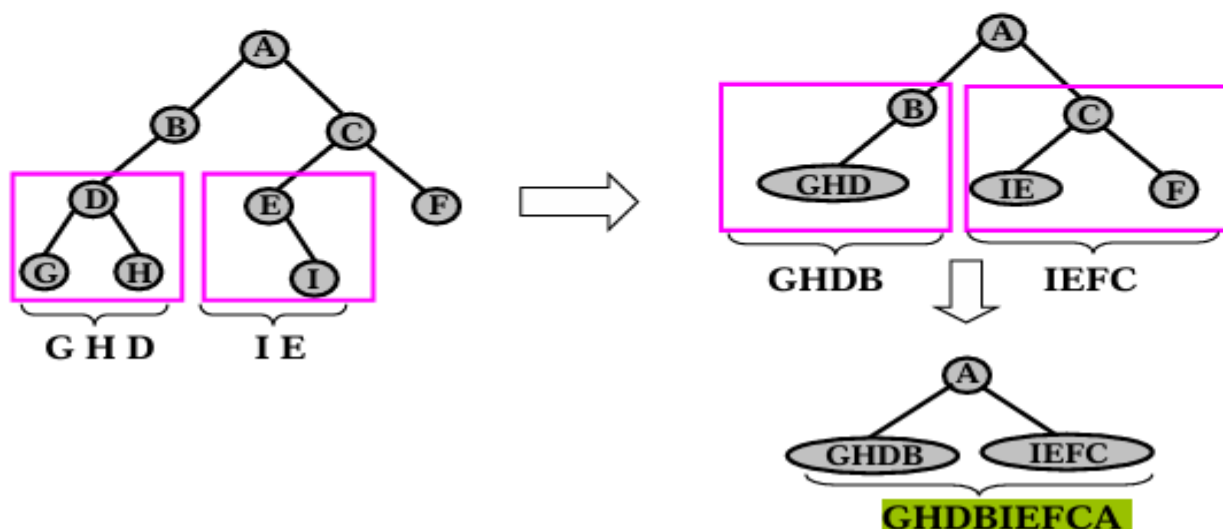
Post-order: GHDBIEFCA :

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

1. Traverse the **Left** subtree in postorder [**L**]
2. Traverse the **Right** subtree in postorder [**R**]
3. Process the root **Node** [**N**]



For example, let us traverse the following tree in preorder.



7 c. Define selection tree. Construct min winner tree for the runs of a game given below. Each run consists of values of players. Find the first 5 winners.

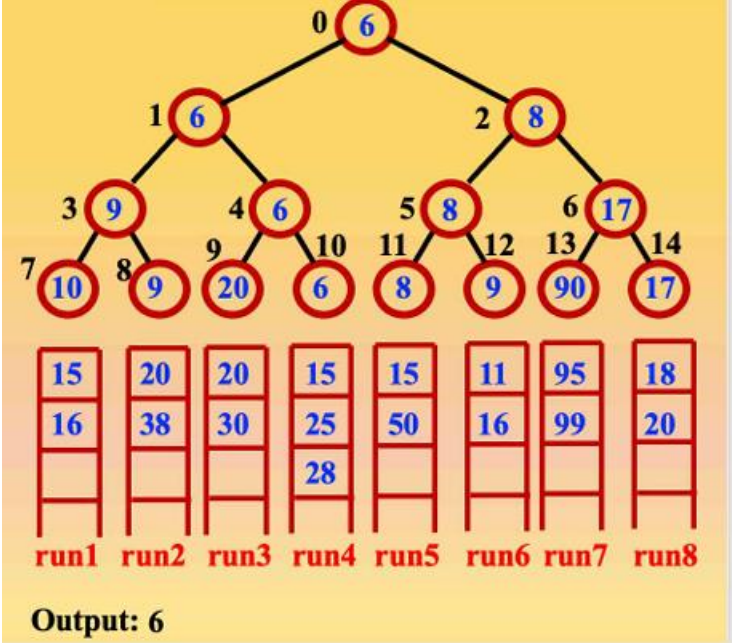
10	9	20	6	8	9	90	17
15	20	20	15	15	11	95	18
16	38	30	25	50	16	99	20
			28				

SELECTION TREE

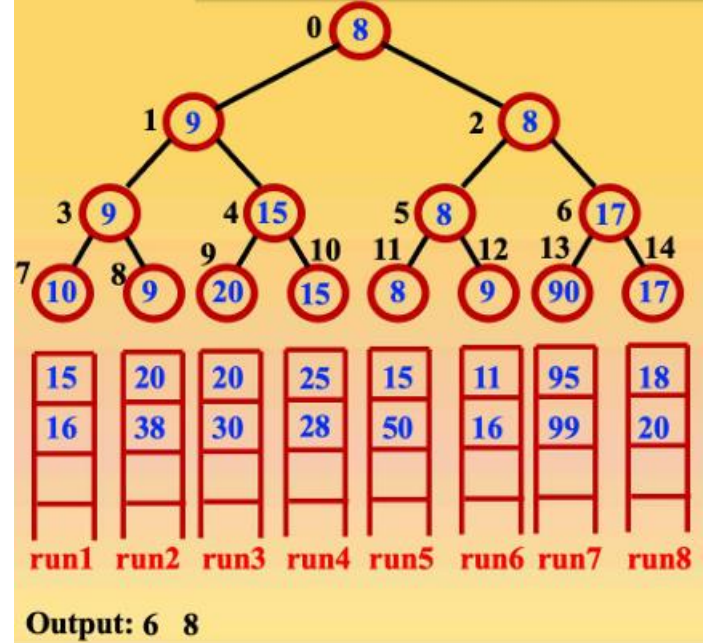
- This is also called as a tournament tree. This is such a tree data structure using which the winner (or loser) of a knock out tournament can be selected.
- There are two types of selection trees namely: winner tree and loser tree. WINNER TREE
- This is a complete binary tree in which each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree.

Min Winner Tree:

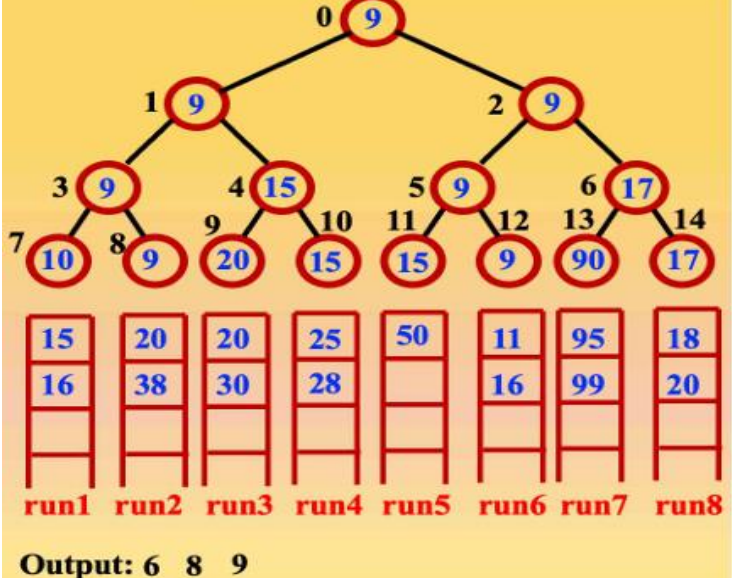
Step 1:



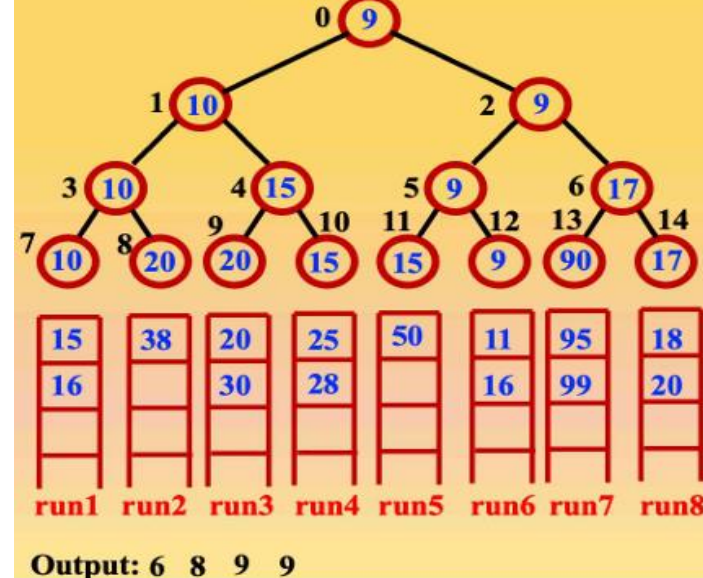
Step 2:

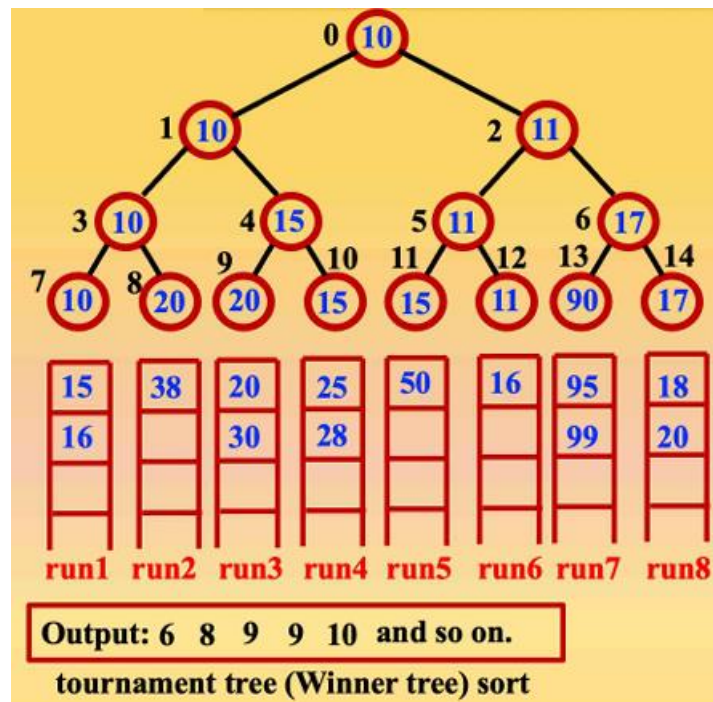


Step 3:



Step 4:





8 a. Define Binary Search tree. Construct a binary search tree (BST) for the following elements: 100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145.

Traverse using in-order, pre-order, and post-order traversal techniques. Write recursive C functions for the same.

A **Binary Search Tree (BST)** is a binary tree in which every node follows these properties:

1. The value of the left child node is less than the value of its parent node.
2. The value of the right child node is greater than the value of its parent node.
3. Both the left and right subtrees must themselves be binary search trees.

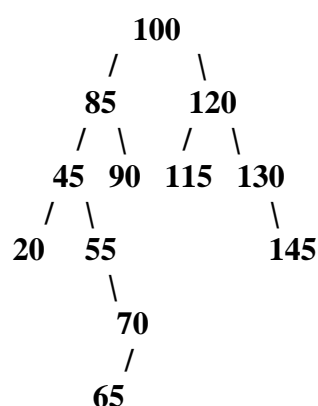
Construction of the BST:

The given elements are: 100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145.

To construct the BST:

1. Start with the first element (100) as the root.
2. Insert the remaining elements one by one, following the BST rules.

BST Structure:



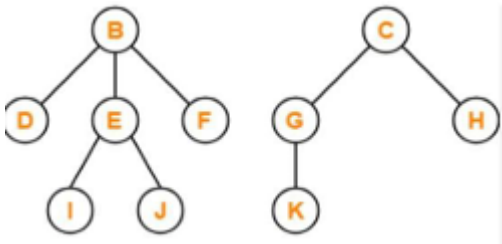
Let’s construct the Binary Search Tree (BST) with the given elements and traverse it using in-order, pre-order, and post-order traversal methods.

Function to traverse the binary tree in inorder	Function to traverse the binary tree in preorder	Function to traverse the binary tree in postorder
<pre>void inorder(NODE root) { if (root == NULL) return; inorder(root->llink); printf("%d ",root->info); inorder(root->rlink); }</pre>	<pre>void preorder(NODE root) { if (root == NULL) return; printf("%d ",root->info); preorder(root->llink); preorder(root->rlink); }</pre>	<pre>void postorder(NODE root) { if (root == NULL) return; postorder(root->llink); postorder(root->rlink); printf("%d ", root->info); }</pre>

Recursive Traversal Techniques:

- 1. **In-order Traversal:**
 - Traverse the left subtree.
 - Visit the root.
 - Traverse the right subtree.
 - Output: 20, 45, 55, 65, 70, 85, 90, 100, 115, 120, 130, 145
- 2. **Pre-order Traversal:**
 - Visit the root.
 - Traverse the left subtree.
 - Traverse the right subtree.
 - Output: 100, 85, 45, 20, 55, 70, 65, 90, 120, 115, 130, 145
- 3. **Post-order Traversal:**
 - Traverse the left subtree.
 - Traverse the right subtree.
 - Visit the root.
 - Output: 20, 65, 70, 55, 45, 90, 85, 115, 145, 130, 120, 100

8 b. Define Forest. Transform the given forest into a Binary tree and traverse using inorder, preorder and postorder traversal.



A **forest** is a collection of disjoint trees. Each tree in the forest is an independent hierarchical structure composed of nodes.

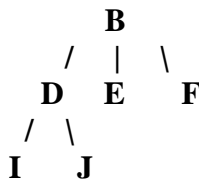
Transforming the Forest into a Binary Tree:

To transform a forest into a binary tree:

1. **Maintain the hierarchical structure:**
 - The first child of a node becomes its left child in the binary tree.
 - The subsequent siblings of a node are linked as right children in the binary tree.
2. **Repeat recursively** for each tree in the forest.

Given Forest:

- **Tree 1 (Root = B):**

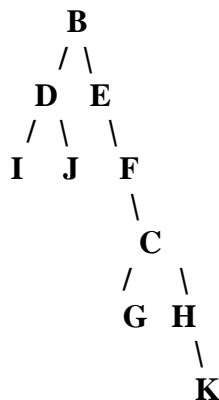


- **Tree 2 (Root = C):**



Transformed Binary Tree:

After transformation, the forest will look like this as a binary tree:



Traversals of the Binary Tree:

1. **In-order Traversal (Left, Node, Right):**
 - Traverse the left subtree, visit the root, then the right subtree.
 - **Output:** I, D, J, B, E, F, G, C, H, K
2. **Pre-order Traversal (Node, Left, Right):**
 - Visit the root, traverse the left subtree, then the right subtree.
 - **Output:** B, D, I, J, E, F, C, G, H, K
3. **Post-order Traversal (Left, Right, Node):**
 - Traverse the left subtree, then the right subtree, and visit the root.
 - **Output:** I, J, D, F, E, G, K, H, C, B

8c Define the Disjoint set. Consider the tree created by the weighted union function on the sequence of unions: union(0,1), union(2,3), union(4,5), union(6,7), union(0,2), union(4,6), and union(0,4). Process the simple find and collapsing find on eight finds and compare which find is efficient.

A **disjoint set** (also known as a union-find or merge-find set) is a data structure that keeps track of a partition of a set into disjoint subsets. It supports two primary operations:

1. **Union(x, y):** Merge the subsets containing xxx and yyy into a single subset.
2. **Find(x):** Determine which subset a particular element xxx belongs to. This can be used to check if two elements are in the same subset.

Weighted Union and Collapsing Find

1. **Weighted Union:** In this method, when merging two subsets, the smaller tree (in terms of size or rank) is always made a subtree of the larger one to keep the resulting tree as flat as possible. This minimizes the depth of the trees, making subsequent find operations efficient.
2. **Collapsing Find (Path Compression):** This optimization flattens the tree whenever a find operation is performed. It makes each node on the path from xxx to the root point directly to the root, reducing the depth of the tree.

Sequence of Union Operations

Given the sequence of unions:

1. union(0,1)
2. union(2,3)
3. union(4,5)
4. union(6,7)
5. union(0,2)
6. union(4,6)
7. union(0,4)

Steps:

- **Step 1:** union(0,1) Tree: 0→1
- **Step 2:** union(2,3) Tree: 2→3
- **Step 3:** union(4,5) Tree: 4→5
- **Step 4:** union(6,7) Tree: 6→7
- **Step 5:** union(0,2) Tree: 0→1, 0→2, 2→3
- **Step 6:** union(4,6) Tree: 4→5, 4→6, 6→7
- **Step 7:** union(0,4) Final tree: 0→1, 0→2, 2→3, 0→4, 4→5, 4→6, 6→7

Efficiency Comparison: Simple Find vs. Collapsing Find

1. **Simple Find:** For each find operation, you traverse the path from the node to the root. This can become inefficient as the tree depth increases.
2. **Collapsing Find:** During the traversal, each node along the path points directly to the root, flattening the tree. This makes future operations faster.

Eight Find Operations:

Suppose we perform find operations on all elements (0 to 7) after the union sequence.

- **Simple Find:** Each find operation traverses the tree, which may take $O(\text{tree depth})$. For this union sequence, some find operations could take $O(\log_{10} n)$ in the worst case.
- **Collapsing Find:** The first find operation for each element may take $O(\log_{10} n)$, but subsequent finds are nearly constant time, $O(1)$, because the tree is flattened.

Comparison

- **Simple Find:**
 - Time complexity: $O(\log_{10} n)$ per find in the worst case.
 - Total time for 8 finds: $O(8\log_{10} n)$.
- **Collapsing Find:**
 - Time complexity: Nearly $O(1)$ after the first find due to tree flattening.
 - Total time for 8 finds: Approximately $O(8)$.

9 a. What is chained hashing? Discuss its pros and cons. Construct the hash table to insert the keys: 7, 24, 18, 52, 36, 54, 11, 23 in a chained hash table of 9 memory locations. Use $h(k) = k \bmod m$

Chained hashing, also known as separate chaining, is a technique used to resolve collisions in hash tables. When two or more keys hash to the same index (known as a collision), chained hashing handles these collisions by maintaining a linked list of all elements that hash to the same index.

Pros of Chained Hashing:

1. **Simple Implementation:** Chained hashing is relatively easy to implement.
2. **Efficient Insertion and Deletion:** Insertion and deletion operations are efficient in chained hashing.
3. **Dynamic Data Sets:** Chained hashing allows for dynamic resizing of the hash table..
4. **Adaptability:** Chained hashing can handle a wide range of input data distributions

Cons of Chained Hashing:

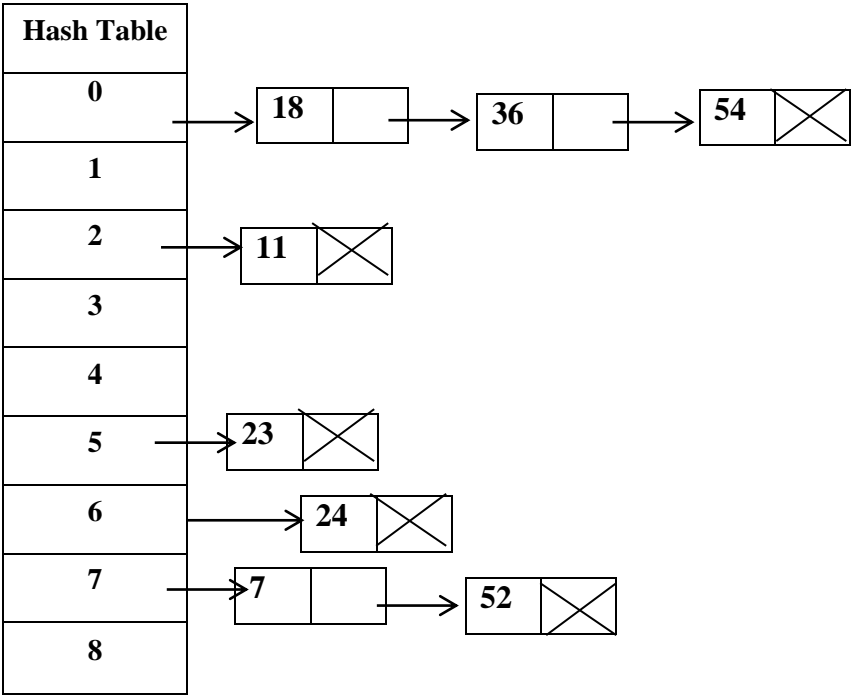
1. **Memory Overhead:** Chained hashing can have a higher memory overhead compared to other collision resolution techniques.
2. **Cache Inefficiency:** cache inefficiency, particularly for large hash tables with long linked lists.
3. **Performance Degradation:**
4. **Poor Worst-Case Performance:** Chained hashing does not guarantee constant-time performance for lookup operations in the worst case

Solution: 7, 24, 18, 52, 36, 54, 11, 23

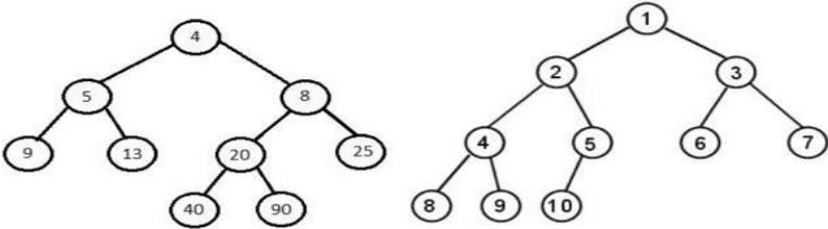
Table size 9. An element can be mapped to location using Key % 9

$7 \bmod 9 = 7$
 $24 \bmod 9 = 6$
 $18 \bmod 9 = 0$
 $52 \bmod 9 = 7$ (Collision Occurred)
 $36 \bmod 9 = 0$ (Collision Occurred)
 $54 \bmod 9 = 0$ (Collision Occurred)
 $11 \bmod 9 = 2$
 $23 \bmod 9 = 5$

Hash Table location	Mapped Element
0	18, 36, 54
1	
2	11
3	
4	
5	23
6	24
7	7, 52
8	



9 b. Define the leftist tree. Give its declaration in C. Check whether the given binary tree is a leftist tree or not. Explain your answer.



- A **leftist tree** is a special kind of binary tree that satisfies the following properties:
- Heap Property:** The value of each node is less than or equal to the values of its children.
 - Leftist Property:** For every node, the **null path length (NPL)** of the left child is greater than or equal to the NPL of the right child.

The **null path length (NPL)** of a node is the shortest distance from that node to a null child (a non-existent node). For a leaf node, the NPL is 0. For a null node, the NPL is -1.

C Declaration.

```
struct Node
{
    int key;
    struct Node* left;
    struct Node* right;
    int npl;
};
```

Explanation of the Given Binary Trees

1. Tree on the Left:

- **Heap Property:**
 - Root node 4 is less than 5 and 8.
 - Node 5 is less than 13 and 20, and so on.
 - **The heap property is satisfied.**
- **Leftist Property:**
 - Compute the NPL for each node and verify that the left subtree's NPL is greater than or equal to the right subtree's NPL:
 - For node 4: Left child 5 has NPL of 2, right child 8 has NPL of 1 ($2 \geq 1$).
 - For node 5: Left child 13 has NPL of 1, right child 20 has NPL of 0 ($1 \geq 0$).
 - This holds for all nodes.
 - **The leftist property is satisfied.**
- **Conclusion:** The left tree is a leftist tree.

2. Tree on the Right:

- **Heap Property:**
 - Root node 1 is less than 2 and 3, but node 2 is greater than 8 in its left subtree.
 - **The heap property is violated.**
- **Leftist Property:**
 - For node 2, the left child 8 has NPL 0, and the right child 9 has NPL 0 ($0 \geq 0$), so the leftist property holds.
 - The leftist property holds for other nodes.
- **Conclusion:** The right tree is **not** a leftist tree because the heap property is violated.

Final Answer:

- **Left Tree:** Is a leftist tree.
- **Right Tree:** Is **not** a leftist tree due to a violation of the heap property.

9 c. What is dynamic hashing? Explain the following techniques with examples: i) Dynamic hashing using directories ii) Directory less dynamic hashing.

Dynamic hashing is a technique used to manage hash tables efficiently, especially when the size of the table is unknown or changes dynamically as data is inserted or deleted.

Techniques of Dynamic Hashing

i) Dynamic Hashing Using Directories (Extendible Hashing)

This technique uses a **directory** to manage the buckets. The directory acts as an index to point to the buckets, and its size changes dynamically as data is added.

Process:

- 1. **Hash Function:** A binary hash function is used, which produces a bit string for each key (e.g., $h(k)$).
- 2. **Directory:**
 - The directory maintains pointers to the buckets.
 - Initially, it uses only the first few bits of the hash values to index into the directory.
 - If a bucket overflows, the directory doubles in size, using an additional bit from the hash to reassign keys and redistribute them into new buckets.
- 3. **Splitting Buckets:** When a bucket overflows:
 - A new bucket is created.
 - Keys in the original bucket are rehashed using one more bit of the hash function.
 - The directory size doubles, and pointers are updated.

Example:

- Start with a directory of size $2^1=2$ (using 1 bit of the hash).
- Keys: 5, 7, 12, 19 with $h(k)=k \bmod 4$.

Key	Binary Hash	Directory Index	Bucket
5	01	1	[5]
7	11	1	[5, 7]
12	00	0	[12]
19	11	1	Split: Bucket 1 (overflow)

ii) Directory-Less Dynamic Hashing (Linear Hashing)

This technique avoids using a directory. Instead, buckets are split sequentially as they overflow.

Process:

- 1. **Hash Function:** A series of hash functions $h_0(k), h_1(k), h_2(k)$, etc., are used, where each successive hash uses more bits of the key.
- 2. **Bucket Splitting:**
 - When a bucket overflows, only that bucket is split, and its contents are redistributed.
 - The decision to split is based on a **split pointer**, which moves sequentially from the first bucket to the last bucket.
- 3. **Progressive Splitting:**
 - Start with NN buckets and $h_0(k)$.

- When a bucket overflows, it is split into two buckets, and the new bucket uses $h_1(k)$
- The split pointer moves to the next bucket, ensuring gradual reorganization.

Example:

- Initial buckets: 0, 1 using $h_0(k)=k \bmod 2$.
- Keys: 2,3,5,7,11,13

Key	$h_0(k)=k \bmod 2$	Bucket
2	0	[2]
3	1	[3]
5	1	[3, 5]
7	1	Overflow

10 a. What is a Priority queue? Demonstrate functions in C to implement the Max Priority queue with an example. i) Insert into the Max priority queue ii) Delete into the Max priority queue iii) Display Max priority queue.

A **Priority Queue (PQ)** is a special type of queue where each element is associated with a **priority**. Elements with higher priority are dequeued before elements with lower priority. In a **Max Priority Queue**, the element with the highest priority (or largest value) is dequeued first. Priority queues are often implemented using **heaps**, which provide an efficient way to insert and delete elements based on priority.

// Function to insert an element into the Max Priority Queue

```
void insert(MaxPQ *pq, int value)
{
    if (pq->size == MAX_SIZE)
    {
        printf("Priority Queue is full!\n");
        return;
    }
    pq->arr[pq->size] = value;
    int index = pq->size;
    pq->size++;
    while (index > 0 && pq->arr[(index - 1) / 2] < pq->arr[index])
    {
        swap(&pq->arr[index], &pq->arr[(index - 1) / 2]);
        index = (index - 1) / 2;
    }
}
```

// Function to delete the maximum element (root) from the Max Priority Queue

```
int deleteMax(MaxPQ *pq)
{
    if (pq->size == 0)
```



```

{
    printf("Priority Queue is empty!\n");
    return -1;
}
int maxVal = pq->arr[0];
pq->arr[0] = pq->arr[pq->size - 1];
pq->size--;
heapify(pq, 0);
return maxVal;
}

```

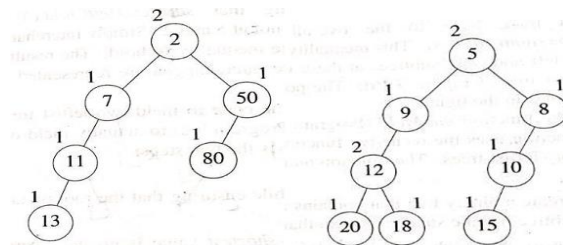
// Function to display the elements of the Max Priority Queue

```

void display(MaxPQ *pq)
{
    if (pq->size == 0)
    {
        printf("Priority Queue is empty!\n");
        return;
    }
    printf("Max Priority Queue: ");
    for (int i = 0; i < pq->size; i++)
    {
        printf("%d ", pq->arr[i]);
    }
    printf("\n");
}

```

10 b. Define min Leftist tree. Meld the given min leftist trees.



A **min leftist tree** is a leftist tree in which:

1. **Heap Property:** The value of each node is less than or equal to the values of its children (min-heap property).
2. **Leftist Property:** For every node, the null path length (NPL) of the left child is greater than or equal to the NPL of the right child.

Meld Operation in Min Leftist Trees

The **meld operation** combines two min leftist trees into a single min leftist tree. The steps for melding two min leftist trees are:

1. Merge Root Nodes:

- Compare the root keys of the two trees.
- The root with the smaller key becomes the root of the new tree.

2. Recursively Meld:

- Recursively meld the right subtree of the root with the other tree.

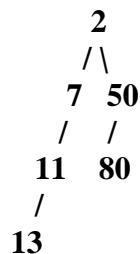
3. Update Structure:

- After merging, swap the left and right subtrees of the root if necessary to maintain the leftist property.

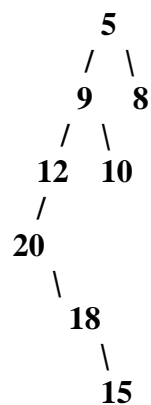
Steps to Meld the Given Trees

The two min leftist trees in the image are:

Tree 1:



Tree 2:



Step-by-Step Melding:

1. Compare Root Nodes:

- Tree 1's root: 2
- Tree 2's root: 5
- $2 < 5$, so 2 remains the root of the melded tree.

2. Recursively Meld 50 (Tree 1's right subtree) and Tree 2:

- Compare root of 50 with 5.

- $5 < 50$, so 5 becomes the root.

3. Meld 50 with Tree 2's right subtree (8):

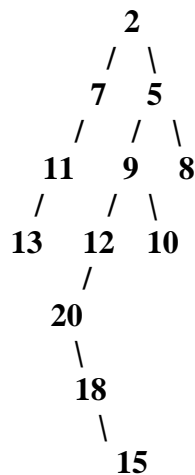
- Compare 50 and 8.
- $8 < 50$, so 8 becomes the root of the melded subtree.

4. Reorganize Subtrees to Maintain Leftist Property:

- After each meld step, update null path lengths (NPL) and swap left and right subtrees if necessary.

Resulting Melded Tree:

After following the steps, the final melded tree structure will look like this (simplified example):



This tree satisfies both the **heap property** and the **leftist property**.

10 c. Define hashing. Explain different hashing functions with examples. Discuss the properties of a good hash function.

Hashing is a technique used to map data (keys) to fixed-size values, typically integers, called **hash values** or **hash codes**. This mapping is performed by a **hash function**.

Different Hashing Functions

A **hash function** takes an input (key) and produces a fixed-size output (hash value). There are different types of hash functions depending on the requirements, such as:

1. Division Method:

- The simplest and most commonly used method.
- The hash value is computed by taking the modulus of the key with a prime number m (size of the table).

Hash function:

$$h(k) = k \bmod m$$

where k is the key, and mm is the size of the table (preferably a prime number).

Example:

- Given a table size of 10, hash function $h(k)=k \bmod 10$
- For key 15, the hash value is $h(15)=15 \bmod 10=5$

2. Folding Method:

- In this method, the key is divided into several parts, which are then added together to produce the hash value.

Hash function:

- Split the key into equal-sized parts, then sum the parts and take modulo mm .

Example:

- Given key 123456, split it into parts: 12, 34, 56.
- Sum the parts: $12+34+56=102$.
- Take $102 \bmod 10=2$.
- The hash value is 2.

3. Mid Square Method:

- In this method, the key is squared, and the middle digits of the result are extracted as the hash value.

Hash function:

$h(k)=\text{middle digits of } k^2$

Example:

- For key 23, square it: $23^2=529$.
- Extract the middle digit(s): 5.
- The hash value is 5.

Properties of a Good Hash Function

A **good hash function** should satisfy the following properties:

1. Deterministic:

- The same input should always produce the same hash value. This ensures consistency.

2. Uniform Distribution:

- The hash function should distribute the keys uniformly across the available hash values (buckets). This helps minimize collisions and ensures efficient lookups.

3. **Efficient:**

- The hash function should be computationally efficient, i.e., it should take constant time, $O(1)$, to compute the hash value.

4. **Minimize Collisions:**

- A collision occurs when two different keys produce the same hash value. A good hash function minimizes collisions as much as possible. This is particularly important for performance.

5. **Easy to Compute:**

- The hash function should be simple and fast to compute, especially for large datasets.

6. **Avalanche Effect:**

- A small change in the input should produce a large, unpredictable change in the hash value. This property is important for cryptographic hash functions.

7. **Sensitive to Input:**

- A small change in the input key should produce a significantly different hash value. This ensures that similar keys don't result in the same hash value.

