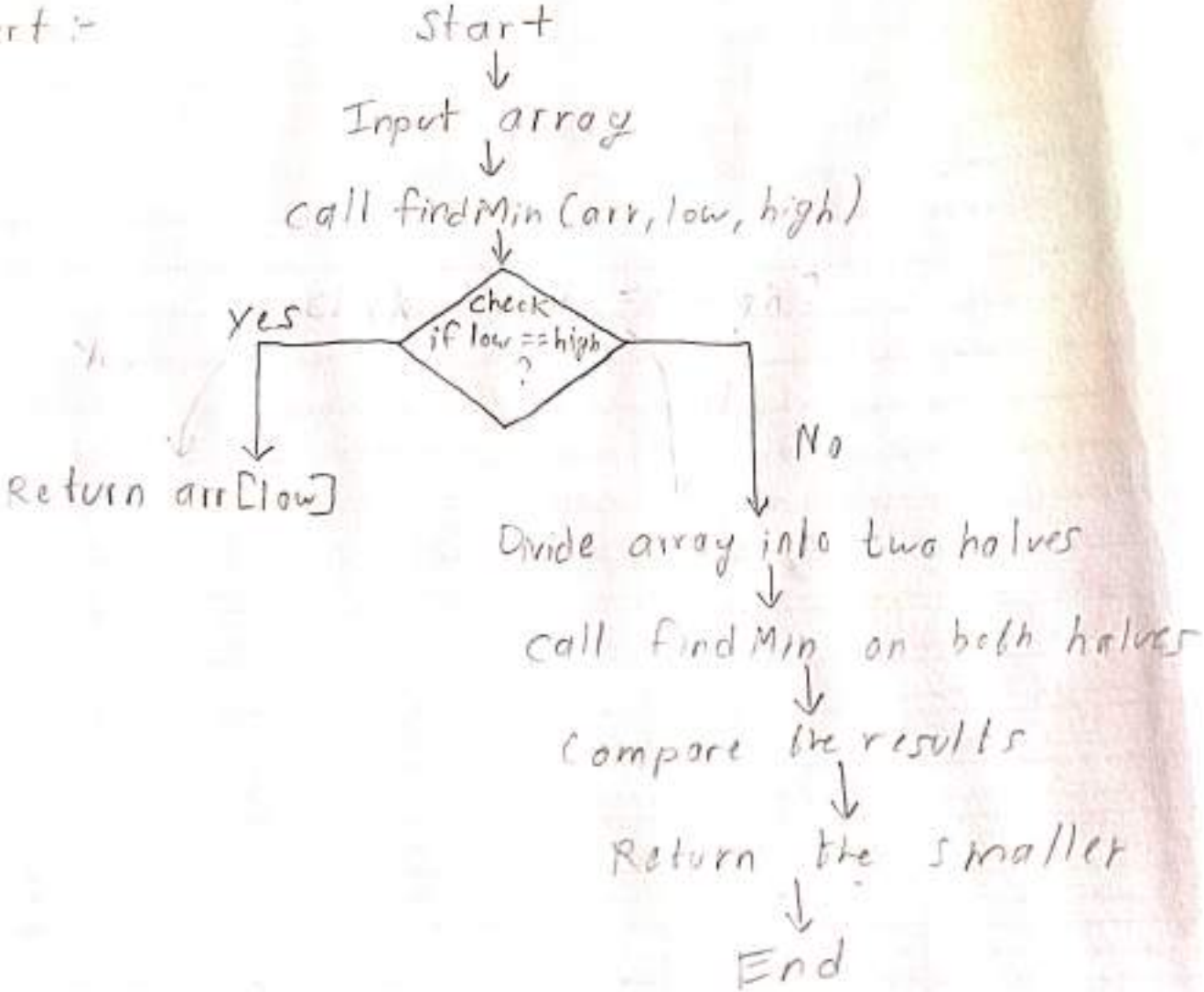
Practical No 1

Aim:- Write a program to implement the concept of divide and conquer strategy.

Flowchart :-



Output :- Minimum value in the array is :

Date

Practical No 1

Aim:- Write a program to implement the concept of Divide and conquer strategy

Theory:

Divide and conquer

Both merge sort and quicksort employ a common algorithmic paradigm based on recursion. This paradigm, divide and conquer, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1) Divide the problem into a number of subproblems that are smaller instances of the same problem.
2) Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3) Combine the solutions to the subproblems into the solution for the original problem.

Page No.

Date

Program

```
#include <stdio.h>
#define size 100
int min1, max2;
Void div_con(int *list, int n, int *min, int *max)
    {
        int mid, min2, max2;
        if (n == 1)
          {
            *min = *max = list[0];
          }
        else if (n == 2)
          {
            if (list[0] < list[1])
              {
                *min = list[0];
                *max = list[1];
              }
            else {
              {
                *min = list[1];
                *max = list[0];
              }
          }
        else
          {
            mid = n/2;
            div_con(list, mid, min, max);
```

Date

```
    div_con (list + mid, n - mid, &min2, &max2);
    if (min2 < *min) *min = min2;
    if (max2 > *max) *max = max2;
  }
}
int main()
  {
    int i, list[size], number;
    int min, max;

    printf("\n Input the number of elements
                    in the list : ");
    scanf("%d", &number);

    print f("\n Input the list elements: \n");
    for (i = 0; i < number; i++)
        scan f("%d", &list[i]);

    div_con (list, number, &min, &max);

    printf("\n Minimum element is : %d", min);
    printf("\n Maximum element is : %d", max);

    return 0;
}
```

Date

Algorithm

Step1:- Start

Step2:- Input the number of elements (`n') in the list.

Step3:- Input the list elements.

Step4:- Call the 'div_con' function with the list, 'n', and pointers to 'min' and 'max',

- Function 'div_con':

1) Base Case:
   - If 'n == 1', set 'min = max = list[0]',
   - If 'n == 2', compare two elements:
     - If 'list[0] < list[1]', set 'min = list[0]' and 'max = list[1]'.
     - Else, set 'min = list[1]' and 'max = list[0]',

2) Divide
   - If 'n > 2', find the middle index 'mid = n/2'

3) Conquer:-
   - Recursively apply 'div_con' to the left half 'list[0. mid-1]' and the right half 'list[mid. n-1]'.

4) Combine:
   - Set 'min' to the minimum of 'min' and 'min2' (from right half).

Page No.

Date :

·) Set 'max' to the maximum of 'max' and 'max2' (From right half).

Step5:- Output the 'min' and 'max' values.

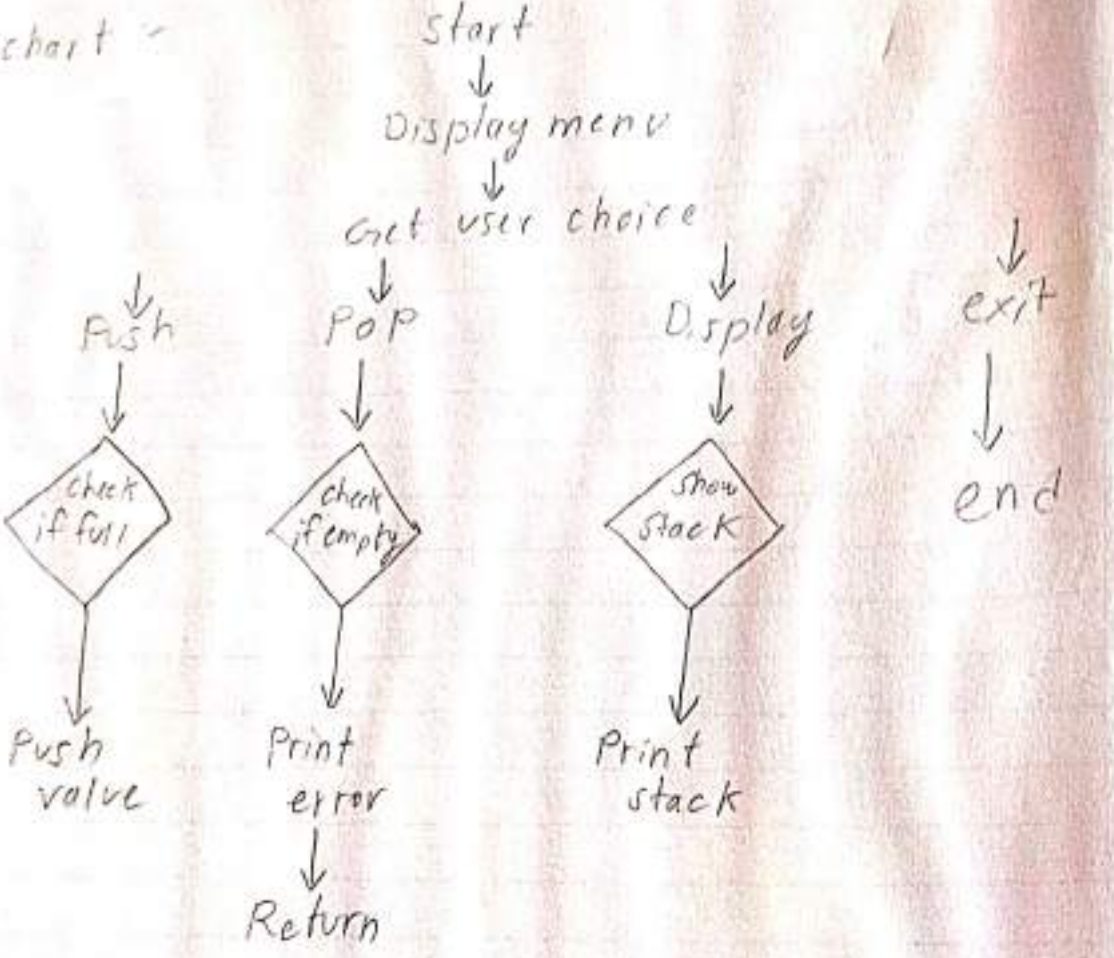
Step6:- End.

Result:- We have executed the program successfully.

Practical No

Aim :- Write a program to implement stack using an array.

Flowchart :-



Output :-

1) Push
2) Pop
3) Display
4) exit

Enter your choice: 1
Enter value to push = 10
10 pushed onto stack.

Date

Practical No 2

Aim :- Write a program to implement stack using an Array.

Theory :- Stack is an abstract data type with a bounded (predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

Algorithm :-

Step 1 :- Initialize 'top = -1'

Display menu:
-) "1. Push", "2. pop", "3. Display", "0. Quit".

Input choice from the user.

Switch choice:
- case 1:
  -) check if 'top < SIZE - 1'
  -) If true, increment 'top' and set 'stack[top] = data'
  -) Else, display "stack overflow".

Page No.

Date :

- Case 2:
  - Check if 'top >= 0'
  - If true, display 'stack[top]' and decrement 'top'.
  - Else, display "stack underflow"
- Case 3:
  - Check if 'top >= 0'
  - If true, display elements from 'stack[top]' to 'stack[0]'.
  - Else, display "stack is empty"
- Case 0: Exit the program

Step 5: Repeat until the user selects "0" to quit.

Date :

Program

```
# include <stdio.h>
# define SIZE 10

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

Void push (int data)
    {
      if (top >=SIZE -1)
        {
          printf ("\n stack overflow");
        }
      else
        {
          stack [++top] = data;
          printf ("\n Pushed %d into the stack",
                  data);
        }
    }
Void pop ()
    {
      if (top <0)
        {
          printf ("\n stack Underflow");
        }
      else {
          printf ("\n popped %d from the stack",
                  stack [top --1);
        }
    }
```

Page No.

Date

```
        case 2:
            pop();
            break;

        case 3:
            display();
            break;

        case 0:
            return 0;

        default:
            printf("\n Invalid choice");
        }
    }
}
```

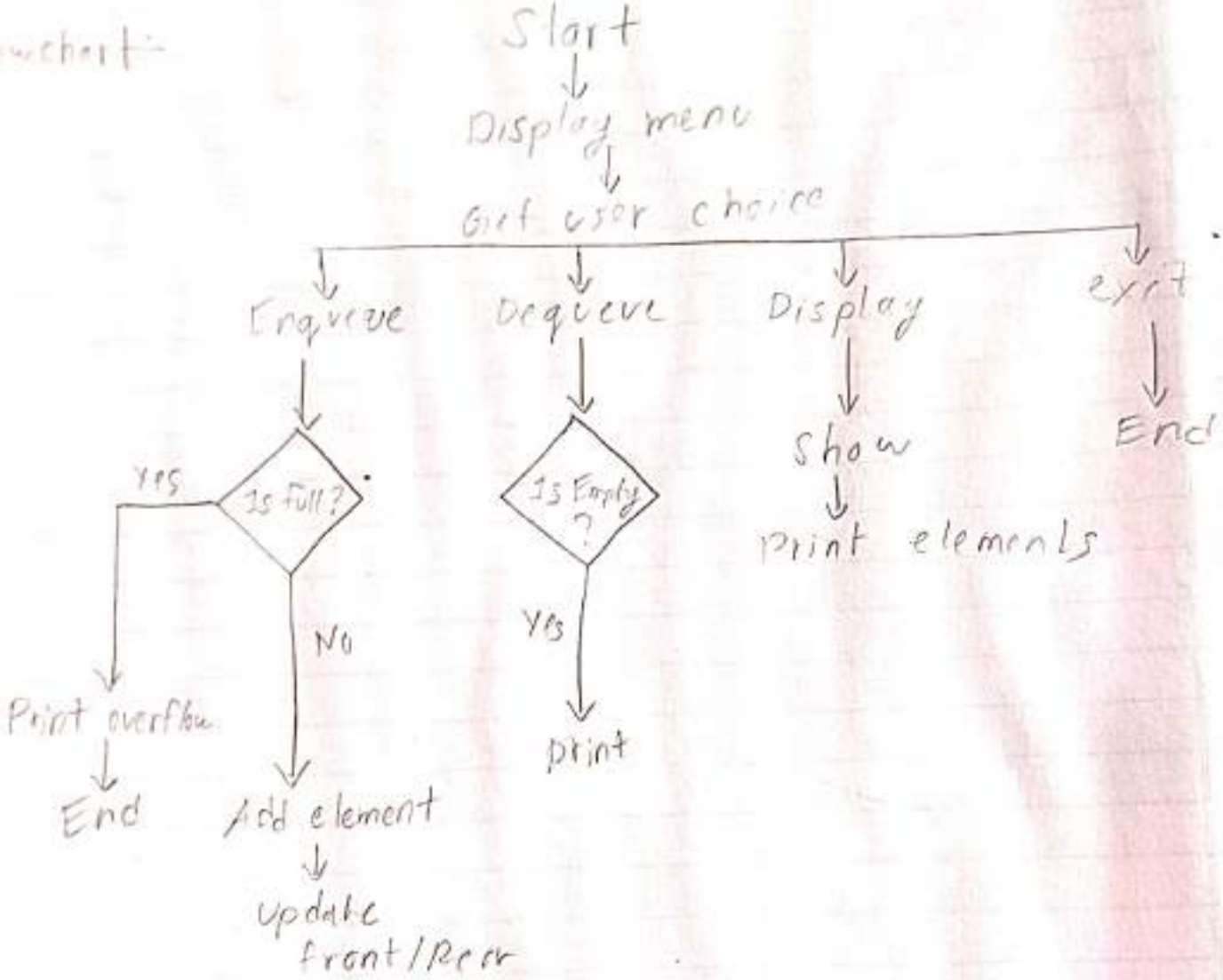
Result:- We have executed the program successfully.

Page No.

Practical No

Aim :- Write a program to implement Queue using an Array.

Flowchart :-



Output :-

1) Enqueue
2) Dequeue
3) Display
4) Exit

Enter your choice: 4

Date

Practical no. 5

Aim :- Write a program to implement Queue using an Array.

Theory :- Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one one end called REAR (also called tail), and the deletion of exisiting element takes place from the other end called as FRONT (also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first. The process to add an element into queue is called Enqueue and the process of removal of an element from queues called Dequeue.

Algorithm

Step 1 :- Initialize 'fron = -1' and 'rear = -1'.

Step 2 :- Display menu :
- 1. Insert
- 2. Delete
- 3. Display
- 4. Quit

Step 3 :- Input choice from the user.

Step 4 :- Switch choice :

Page No.

Date :

- Case 1 (Insert):
  - Check if 'rear' is equal to 'SIZE - 1':
    - If true, display "Queue Overflow"
    - If false, increment 'rear'.
  - If 'front == -1', set 'front = 0'.
  - Insert 'data' into 'queue[rear]'

- Case 2 (Delete)
  - Check if 'front == -1' or 'front > rear'
    - If true, display "Queue Underflow"
    - If false, display 'queue[front]' and increment 'front'.
    - If 'front > rear', reset 'front' and 'rear' to '-1'.

- Case 3 (Display):
  - Check if 'front == -1':
    - If true, display "Queue is empty"
    - If false, display all elements from 'queue[front]' to 'queue[rear]'

- Case 0: Exit the program

Step 5: Repeat until the user selects '0' to quit.

Date

Program :-

```
#include <stdio.h>
# define SIZE 10

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

Void insert_q (int data)
    {
      If (rear == SIZE - 1)
        {
          printf ("\n Queue Overflow");
        }
      else {
        if (front == -1) front = 0;
        queue [++rear] = data;
        printf ("Inserted %d into the queue", data);
        }
    }
Void del_q() {
    if (rear == SIZE - 1) {
        printf ("\n Queue Overflow");
      }
    else {
      if (front == -1) front = 0;
        queue [++rear] = data;
        printf ("\n Inserted %d into the queue,"
                                    data);
Void del_q () {
      if (front == -1 || front > rear)
        {
```

Date :

```
        printf ("\n Queue Underflow");
    }
    else {
        printf ("\n deleted %d from the
                queue", queue[front++]);
        if(front > rear) front = rear = -1;
    }
}
Void display () {
    if (front == -1) {
        printf("\n Queue is empty");
    }
    else {
        printf("\n Queue elements are :");
        for (int i = front; i <= rear; i++)
        {
            printf ("\n %d", queue[i]);
        }
    }
}
int main () {
    int choice, data;
    while (1) {
        printf ("\n \n1. Insert \n2. Delete \n3.
                Display \n0. Quit");
        printf ("\n Enter your choice : ");
        scanf ("%d", &choice);

    switch (choice) {
        case 1:
            printf (" Enter the element to insert:
            scanf ("%d", &data);
```

Page No.

Date :

```
            insert_q (data);
            break;

        case 2:
            del_q ();
            break;
        case 3:
            display ();
            break;

        case 0;
            return 0;

        default #;
            printf("\n Invalid choice");
        }
    }
}
```

Result: We have executed the program successfully.

Page No.

Date :

Practical No 6

Aim :- Write a program to implement insertion and Deletion on Singly link list.

Theory :- A singly linked list is a sequence of nodes where each node contains data and a reference to the next node. The list is accessed via a head pointer, and each node points to the next, with the last node pointing to 'null.'

1) Insertion
   - At Beginning: Create a new node, set its 'next' to the current head, and update the head.
   - At the end: Traverse to the last node and set its 'next' pointer to the new node.

2) Deletion
   - By value :- Traverse the list to find the node with the specified value. Adjust the 'next' pointer of the preceding node to bypass the node to be deleted. If the node to be deleted is the head, update the head pointer.

Date :

Algorithm :-

Step 1 :- Initialize : 'head = NULL'.

Step 2 :- Insert operation :

- Create a new node
- If inserting at the beginning, set its 'next' to 'head' and update 'head.'
- If inserting at the end, traverse to the last node, set its 'next' to the new node.

Step 3 :- Delete operation :

- If deleting from the beginning, update 'head' to 'head->next' and free the old head.
- If deleting from the end, traverse to the second last node, free the last node, set 'second_last->next' to 'NULL'.

Step 4 :- Display : Traverse from 'head', print each node's data.

Step 5 :- Main loop : Based on user input

- '1' : insert at beginning
- '2' : insert at end.
- '3' : Delete from beginning
- '4' : Delete from end
- '5' : Display list
- '6' : exit.

Date :

Problem :-

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

typedef struct Node {
        int data;
        struct Node * next;
} Node;
Node* head = NULL;

Void insert(int data, int at_end) {
        Node** curr = & head;
        if(at_end) while (*curr) curr = &(*curr)
                                        ->next;
        *curr = (Node*) malloc(sizeof(Node));
        (*curr) -> data = data;
        (*curr) -> next = NULL;
    }
Void delete(int from_end) {
        Node** curr = & head;
        if (from_end) while (*curr && (*curr)->next)
                                curr = &(*curr)-> next;
        Node* temp = *curr;
        * curr = temp ? temp->next : Null;
        free (temp);
}

Void display () {
        for (Node* temp = head; temp; temp = temp->next)
          printf ("%d ->", temp->data);
          printf("NULL \n");
}
```

Date :

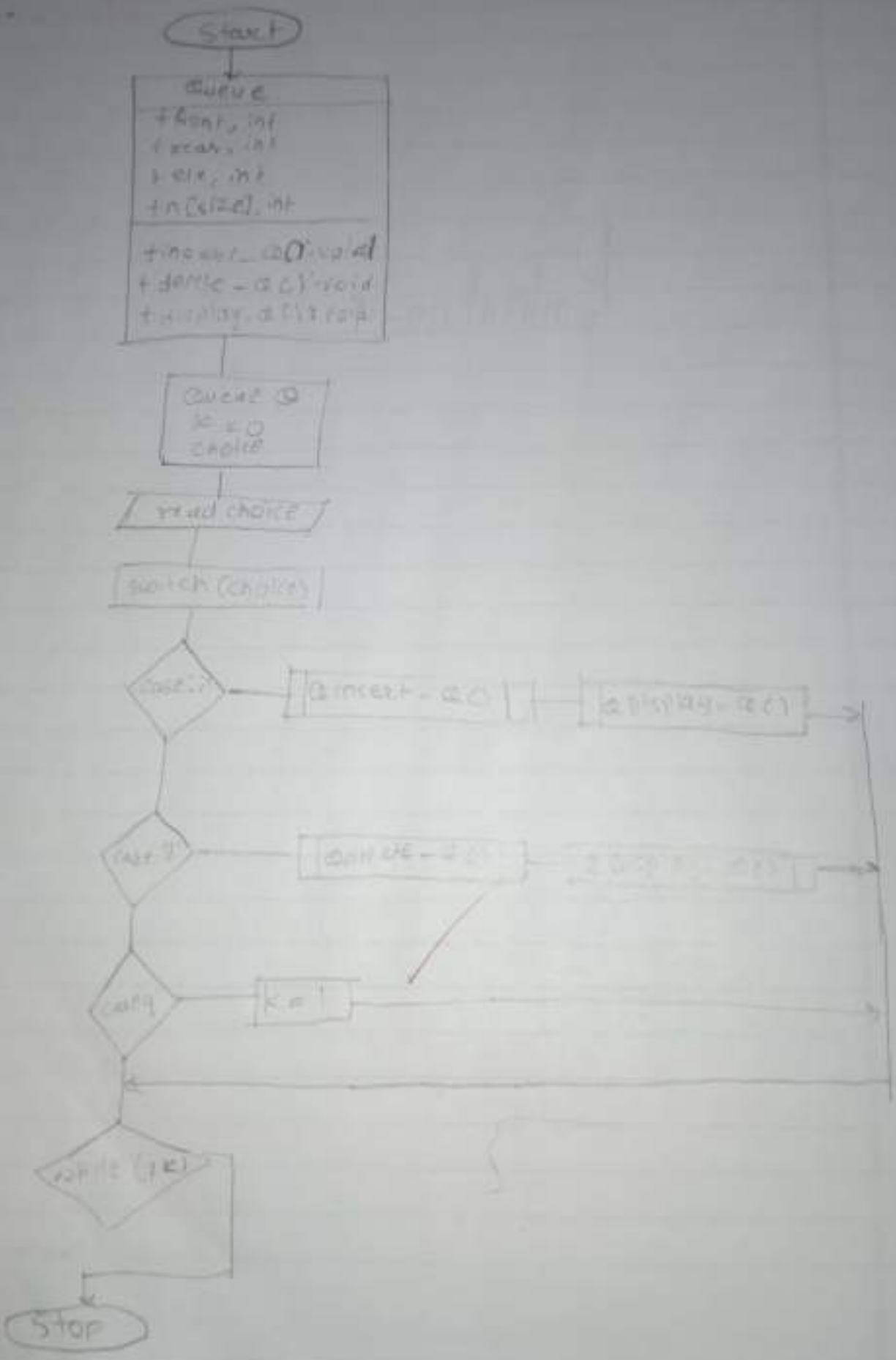
```
int main() {
    int choice, data;
    while(1) {
        printf("\n1. Insert at Beginning \n2. Insert
            at End \n3. Delete from Beginning
            \n4. Delete from End \n5.
            Display \n0. Quit \n choice:");
        scanf("%d", &choice);
        if(choice==0) break;
        if(choice==5) display();
        else if(choice==1||choice==2) {
            printf("Enter data:");
            scanf("%d", &data);
            insert(data, choice-1);
        }
        else if(choice==3||choice==4)
            delete(choice-3);
    }
    return 0;
}
```

Result: We have executed the program successfully.

Program no-4

Aim :- Write a program to implement queue using an array

Flowchart -



Date

Practical no-4

Aim :- Write a program to implement queue using an array

Theory :- Queue is also an abstract datatype or a linear data structure, in which the first element is inserted from one one end called REAR (also called tail) and the deletion of exist ing element takes place from the other end called as FRONT (also called head). This makes queue as FIFO data structure, which means that element inserted first and also removed first. The process to add an element into queue is called Enqueue and the process of removal of an element from queue called dequeue.

Algorithm

Step1 :- Initialize 'front = -1' and 'rear = -1'

Step2 - Display menu
1. Insert
2. Delete
3. Display
4. Quit

Step3 - Input choice from the user

Step4 - Switch choice

Case 1 (Insert)

(i) Check if 'rear' is equal to 'size -1':

(ii) If true, display "Queue Overflow".

(iii) If false, increment 'rear'

(iv) If 'front == -1', set 'front = 0'.

(v) Insert 'data' into queue[rear]

Case 2 (delete)

(i) check if 'front == -1' or 'front == rear'

(ii) If true display "Queue underflow"

Date

(iii) If false, display 'queue[front]' and increment 'front'.
(iv) If 'front > rear', reset 'front' and 'rear' to '-1'.

case 3 (display):
(i) Check if 'front == -1'
(ii) If true, display "queue is empty"
(iii) If false, display all elements 'queue[front]' to 'queue[rear]'

case 0:
Exit the program

Step 5 - Repeat until the user selects '0' to quit
Step 6 - End

Program :-

```
#include <stdio.h>
#define Size 10
int front = -1;
int rear = -1;
int queue[Size];
void insert_q(int data)
{
    If (rear = Size - 1)
    {
        printf("\n Queue overflow");
    }
    else {
        if (front == -1) front = 0;
        queue[rear + 1] = data;
        printf("Inserted %d into the queue", data);
    }
}
```

Date :

```
Void del_q () {
        if (front == -1 || front > rear)
        printf ("\n Queue underflow");
    }
    else {
        printf ("\n deleted %d from the queue", queue
                                [front + 1];
        if (front > rear)
            front = rear = -1;
        }
    }
Void display () {
        if (front == -1)
    {
    printf ("\n Queue is Empty");
    }
    else {
        printf ("\n Queue element are : ");
        for (int i = front; i <= rear; i++)
        {
            printf ("\n %d", queue[i]);
        }
    }
}
int main () {
    int choice, data;
    while () {
        printf ("\n 1. Insert 2. Delete 3.
                Display 0. Quit);
        printf ("\n Enter your choice : ");
        Scanf ("%d", & choice);
        Switch (choice) {
```

Output-

0 Quit
1. Insert
2. Delete
3. display

Enter your choice: 1.
Enter the element to insert: 52
= 52

Date

```
Case 1:
    printf ("Enter the element to insert: ");
    scanf ("%d", &data);
     insert_q (data);
     break;
Case 2:
     del_q();
     break;
case 3:
     display ();
     break;
Case 0:
     return 0;
default:
     printf ("\n Invalid choice ");
   }
 }
}
```

Result :- We have Executed the program Successfully

Viva Question

1. What is queue?

→ Queue is data structure in which insertion take place from one end called "FRONT and deletion take place from another end called "REAR" and it follow the principle of FIFO (first in first out).

2. How is the queue different from the stack?

→ Queue follows the principle of first in first out (FIFO) and insertion take place from one end

Date

deletion take place from another end while in stack the insertion and deletion take place from only one end and follows the principle of last in first out (LIFO).

3. What are the operations performed on queue?
→ The operations performed on queue are as follows:
1) Enqueue
2) Dequeue
3) peek

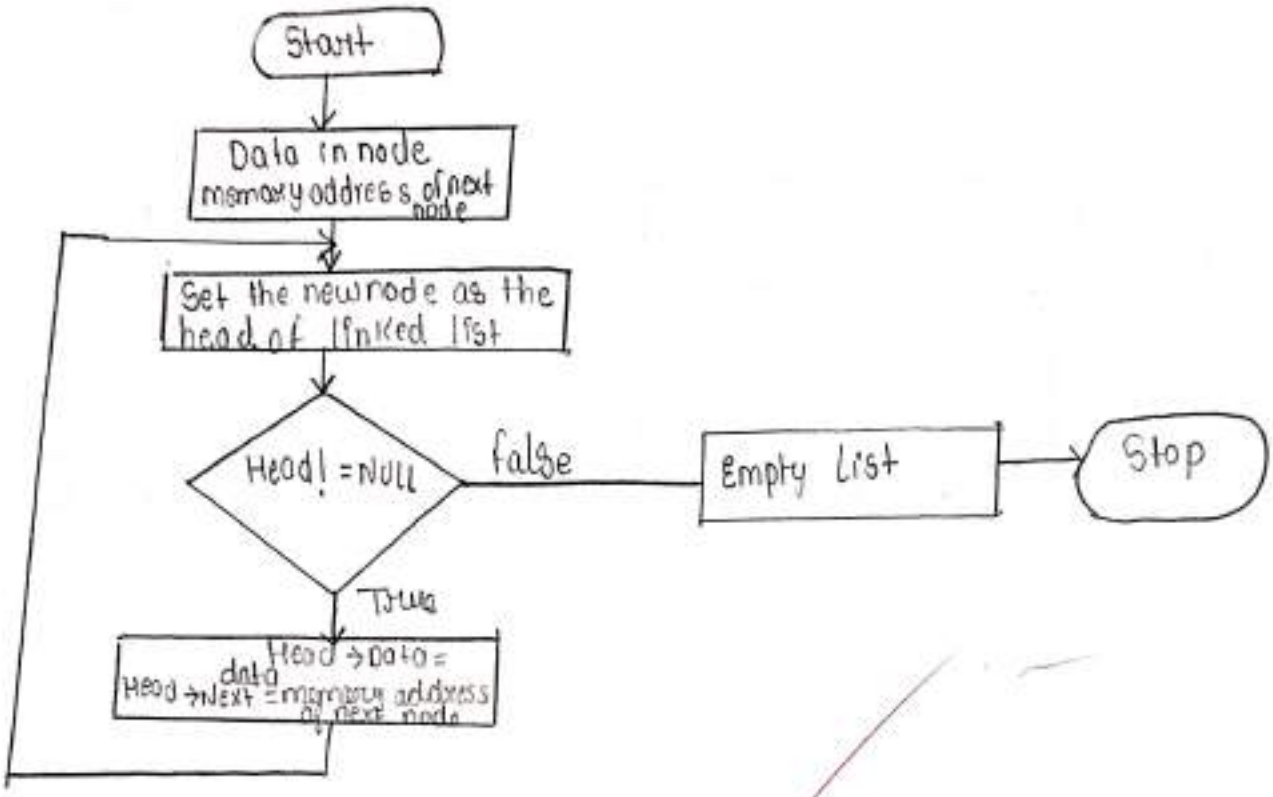
4. How do you represent the queue in computer memory?
→ We can represent queue in the form of an array using pointer: front and rear.

5. What are the disadvantage of queue?
→ The disadvantage of queue are as follow:-
1. No Random access
2. The operation such as insertion and deletion of element from the middle are time consuming
3. Restricted access
4. Bounded capacity
5. additional programming complexity

practical no-5

Aim :→ Write a program to implement insertion and deletion on Singly link list

Flowchart → insertion



Date

Practical no - 5

Aim :→ Write a program to implement insertion and deletion on Singly link list

Theory → A Singly linked list is a sequence of nodes where each node contains data and a reference to the next node. The list is accessed via a head pointer, and each node points to the next, with the last node pointing to 'null'.

1) Insertion
- At beginning - Create a new node, set its 'next' to the current head, and update.
- At the end - Traverse to the last node and set its 'next' pointer to the new node.

2) Deletion
- By value - Traverse the list to find the node with specified value. Adjust the 'next' pointer of the preceding node to by pass the node to be deleted. If the node to be deleted is the head, update the head pointer.

Algorithm -
Step1 - Start
Step2 - Initialize: 'head = NULL'
Step3 - Insert operation
- Create a new node
- If inserting at the beginning, set its 'next' to 'head' and update 'head'.
- If inserting at the end, traverse to the last node, Set its 'next' to the new node.

Step4 - Delete operation
- If deleting from the beginning, update 'head' to head → next 'and free the odd head.

Date :

- If deleting from the end, traverse to the second last node, free the last node set 'second_last → next' to 'NULL'

Step5 - Display : Traverse from 'head', print each node's data.

Step6 - main loop : Based on user input :-
(i) '1' : insert at beginning
(ii) '2' : Insert at end
(iii) '3' : Delete from beginning
(iv) '4' : Delete from end
(v) '5' : Display list
(vi) '0' : Exit

Program -

```
// linked list operation in c
#include <stdio.h>
#include <stdlib.h>
// Create a node
Struct Node {
    int data;
    Struct Node * next;
};
//insert at the beginning
void InsertAtbeginning (Struct node ** head_ref), int new_da
{
    //Allocate memory to a node.
Struct node * New_Node = (Struct Node*) malloc (sizeof (struc
                                                    Node));

//insert the data
    new_node -> data = new_data;
    new_node -> next = (* head_ref);
//move head to new node
```

Date :

```
(*head_ref) = new_node;
}
// Insert a node after a node
Void InsertAfter (Struct Node * prev_node, int new_data) {
    if (prev_node == NULL) {
        printf ("the given previous node cannot be Null");
    return;
    }
Struct node* new_node = (Struct Node *) malloc (sizeof (struc
                                                    Node));
    new_node -> data = new_data;
    new_node -> next = prev_node -> next;
    prev_node -> next = new_node;
}
// Insert the end
Void InsertAtEnd (Struct Node * * head_ref, int new_d
    {
        Struct Node * new_node = (Struct Node *) malloc (sizeof
                                    Struct Node));
        Struct Node * last = head_ref;
        New_node -> data = new_data;
        new_node -> next = Null;
        if (* head_ref == Null) {
            * head_ref = new_node;
        return;
    }
while (last -> next != Null) last = last -> next;
    Last -> next = new_node;
return;
}
// Delete a node
```

Date :

```
Void deleteNode (Struct Node ** head_ref, Int Key)
{
  Struct node * temp = * head_ref, * prev;
   if (temp != Null && temp -> data == Key)
      {
         * head_ref = temp -> next;
          free (temp);
          return;
      }
  // find the Key to delete
  While (temp != NULL && temp -> data != Key) {
      prev = temp;
       temp = temp -> next;
   }
  // If the Key is not present
      if (temp == NULL)
            return;
  // Remove the node
  prev -> next = temp -> next;
    free (temp);
}
// print the linKed list
  void printList (Struct Node * node) {
     While (node != Null) {
   printf ("%d", Node -> data);
     node = node -> next;
   }
}
// Driver program
 int main ( )
{
```

Output

Linked list : 3 2 5 1

After deleting an element : 2 5 1

Result - we have executed the code successfully.

Date

```
Struct Node * head = NULL;
Insert At End (& head, 1);
Insert At Beginning (& head, 2);
Insert At Beginning (& head, 3);
Insert At End (& head, 4);
Insert After (head → next, 5);
printf ("Linked list: ");
Print List (head);
printf ("\ After deleting an element: ");
delete Node (& head, 3);
Print List (head);
}
```

Result - We have executed the code Successfully

Viva question

1. What are the types of link list
→ • Singly Linked list
• Doubly Linked list
• Circular Linked list
• Circular doubly Linked list.

2. What are the advantage of link list ?
→ 1) Easy implementation
2) No space overhead
3) Easy insertion and deletion
4) Size can be change
5) Efficient memory utilizations

Date

3. What are the application of the Link list?
→ Application of Link list
1. Image viewer
2. Music player
3. Gps navigation system
4. Image processing
5. Sy file system

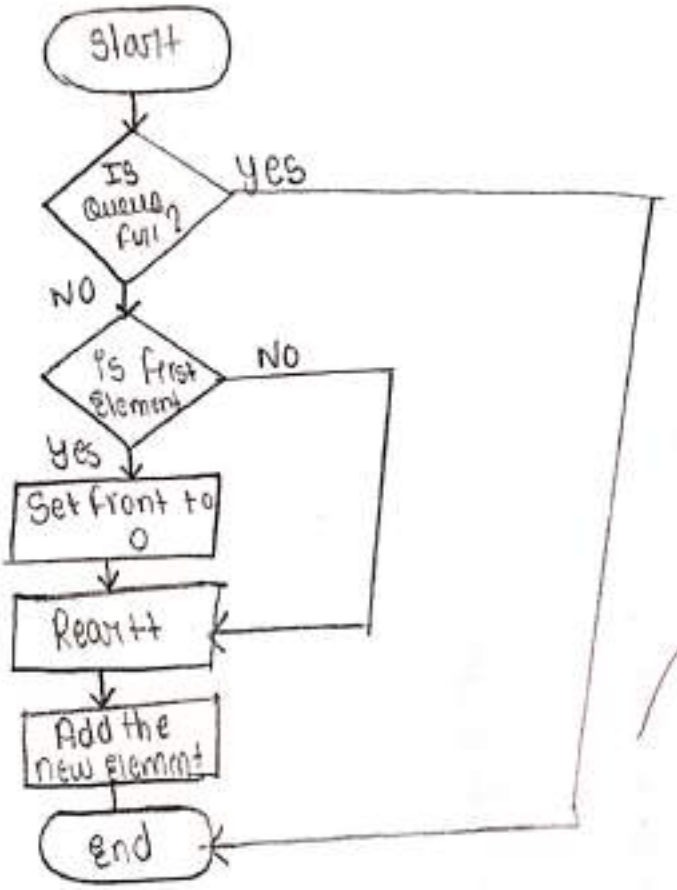
4. What is node?
→ A node is a basic unit that contain data and a reference the next node in the sequence. Nodes are building block of a linked list.

5. What do you mean by free pull?
→ free pull is said to be when the user use check for the free memory.

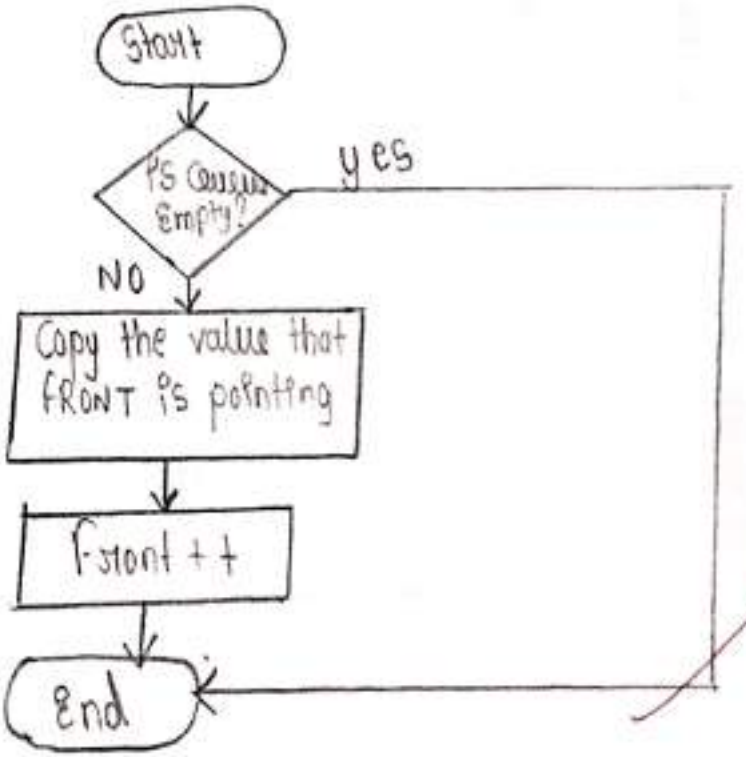
Practical No-6

Aim :- Write a program to implement Circular Queue

Flowchart :- Enqueue



Dequeue



Date

practical no-6

Aim :→ Write a program to implement Circular Queue

Theory → In a standard queue data structure the buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue the last is a linear data structure. It follows fifo principle. In circular queue the last node is connected back to the first node to make a circle. Circular linked list follows the First In First Out principle Elements are added at the rear end and the elements are deleted at front end of the queue. Both the front and the rear pointer points to the beginning of the array. It is also called as "Ring Buffer". Items can inserted and deleted from a queue in O(1) time.

Algorithm –

Insertion of element in Circular Queue.

Step1- Start

Step2 - [Check for the overflow]
If front = 0 and rear = (n-1)
Output "Overflow" & return

Step3 - [Insert element in the queue]
else if front = -1
Set front = 0 rear = 0
queue[rear] = element

Step4 - [check if the rear is at the end of the queue]
else if rear = (n-1)
rear = 0
queue[rear] = element

Step5 - [Insert the element]
else

Date

rear = rear + 1
queue[rear] = element
Step6- return
Step7- Stop

Deletion of an element from Circular Queue
Step1- Start
Step2- [Check for underflow]
If front = -1
output "Underflow" and return
Step3- [Name the element]
element = queue[front]
Step4- [Check whether the queue is Empty or not]
If front = rear
Set front = -1 rear = -1
Step5- [Check the front pointer position]
Else if front = (n-1)
front = 0
Else
front = front + 1
Step6- [Return the removed element]
return (element)
Step7 - Stop

Program -

```
//C program to implement the Circular queue
#include<stdio.h>
// Defind the maximum size of queue
#define MAX_SIZE 5
// Declare the queue array and front, rear variable
int queue[MAX_SIZE];
int front = -1, rear = -1;
```

Date

```
// function to check if queue is full.
int isFull()
{
    // If the next position is front, the queue is full
    return (rear+1) % MAX_SIZE == front;
}
// function to check if the queue is empty
int isEmpty()
{
    // If the front hasn't been set, the queue is empty
    return front == -1;
}
// function to enqueue (insert) an element
void enqueue(int data)
{
    // If the queue is full, print an error message and
    return
    if (isFull()) {
        printf("Queue overflow \n");
        return;
    }
// If the queue is empty, set the front to first posi
    if (front == -1) {
        front = 0;
    }
// Add the data to the queue and move the rear poin
    rear = (rear+1) % MAX_SIZE;
    queue[rear] = data;
    printf("Element %d inserted \n", data);
}
// function to dequeue (remove) an element
int dequeue()
```

```
{
  // If the queue is empty, print an error message and
                    return -1
  if(isEmpty()){
            printf("Queue underflow\n");
            return -1;
  }
  // Get the data from the front of the queue
    int data = queue[front];
  // If the front and rear pointers are at the same
position, reset them
          if( front == rear) {
                front = rear = -1;
      }
      else
          {
            // otherwise, move the front pointer to the next
                position
            front = (front+1) % MAX_SIZE;
          }
      Return the dequeued data
          return data;
  }
  // function to display the queue elements
  void display ()
    {
      // If the queue is empty, printa message and
                        return
        if (isEmpty()) {
          printf("Queue is Empty\n");
        return;
```

Output

Element 10 inserted
Element 20 inserted
Element 30 inserted.
Queue elements : 10 20 30
Dequeued element : 10
Queue elements : 20 30

Result :- We have executed the program successfully

Date :

```
}
// print the Elements in the queue
    printf(" Queue Elements: ");
        int i = front;
        while (i != rear) {
        printf("%d", queue[i]);
        i = (i+1) % MAX_SIZE;
    // print the last element
    printf("%d\n", queue[rear]);
}
// main function
int main()
{
    // Enqueue some elements
        enqueue(10);
        enqueue(20);
        enqueue(30);
    // Display the queue
        display();
    // Dequeue an element and print it
    printf("Dequeued element: %d\n", dequeue());
    //  Display the queue again
        display();
    return 0;
}
```

Result - We have executed the program successfully

Viva question

1. What are the advantage of Circular queue as compared to Simple queue?

→ As the insertion in the queue from the rear end and in the case of Linear Queue of fixed size but in the case of circular queue the rear end moves from the last positi to the front position circularly. The insertion and deletion in the circular queue is easier than the insertion and deletion in linear queue

2. What is Circular queue

→ Circular queue is a linear data structure in which the operations are performed on FIFO [First in First out] princ and the last position is connected back to the first positi to make a circle.

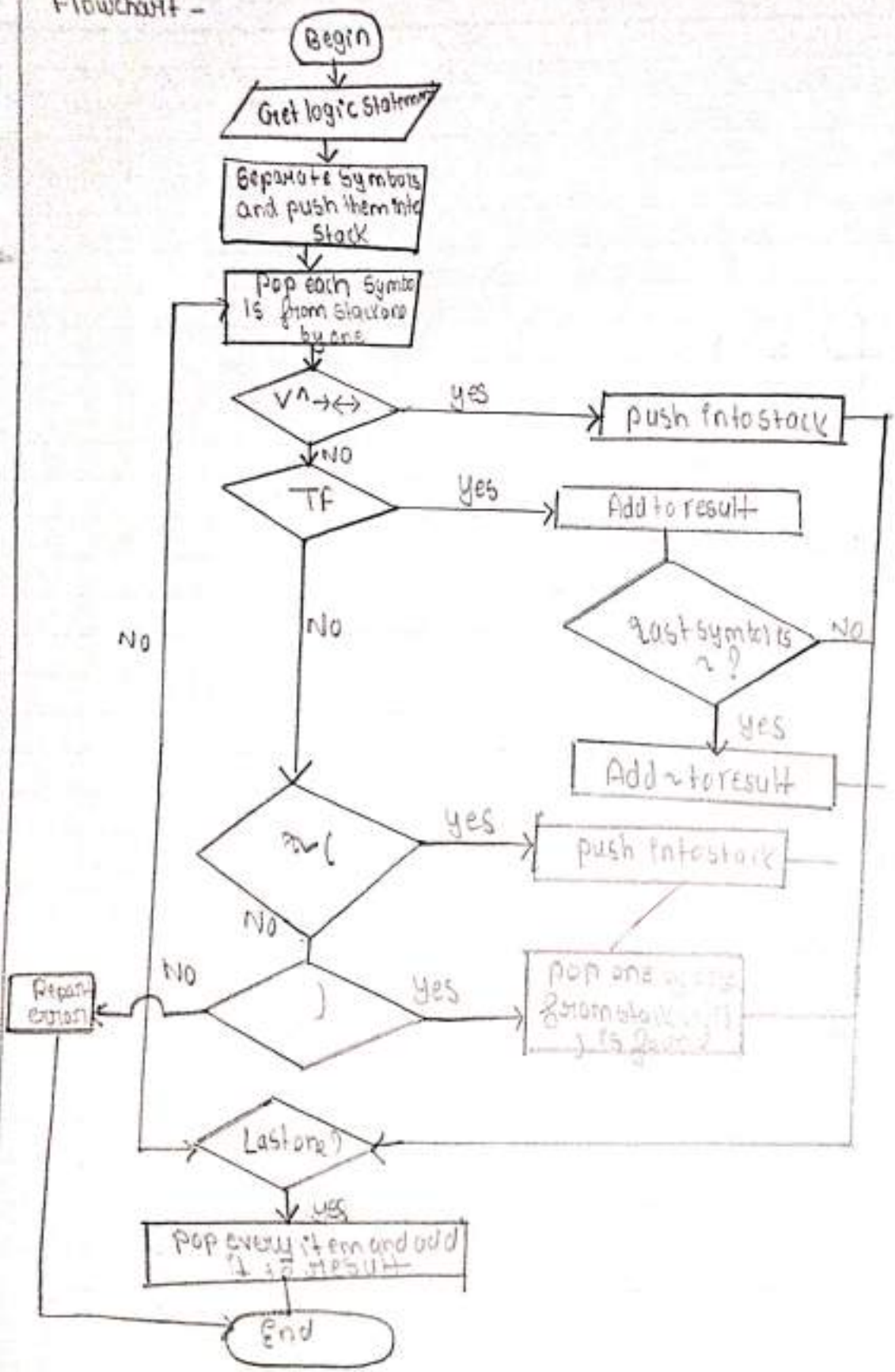
3. How do you represent it?

→ Representation of Circular Queue using arrays and linked list. You can use both 1-D array and linked list to implement it.

Practical no-7

Aim- Write a program to implement Infix to postfix expression Evaluation

Flowchart -



Date :

Practical no-7

Aim :→ Write a program to implement infix to postfix expression evaluation.

Theory :→ In order to convert infix to postfix expressions. We need to understand the precedence of operators first. precedence of operators There are five binary operators, called addition, subtraction, multiplication, division and exponentiation. We aware of some other binary operators as well. These require only one operand e.g - and +. There are rules on order of execution of operators in mathematics called precedence. firstly, the exponentiation operation is executed, followed by multiplication / division and at the end addition / subtraction is done. The order of precedence is (highest to lowest): exponentiation - multiplication/division *, / Addition / Subtraction +. For operator of same precedence, the left-to-right rule applies: A+B+C means (A+B)+C.
For exponentiation, the right-to-left rule applies :
A B C means A(BC)
We want to understand these precedence of operators and infix and postfix forms of expressions. A programmer can solve a problem where the program will be aware of the precedence rules and convert the expression from infix to postfix based on the precedence rules.

Algorithm - Suppose Q is an arithmetic expression written in infix notation.
Step1 - Start
Step2 - push "(" onto stack, add ")" to the end of Q
Step3 - Scan Q from left to right and repeat step4 to 7 for each element of Q until the stack is Empty.

Date :

```
fget (infix [SIZE], SIZE, stdin);
infix_post (infix, post);
puts ("The post fix expression is %.s\n", post);
return 0;
}
void infix_post (char *i, char *p)
{
  char t;
  char stack [SIZE];
  while(*i)
  {
    if (*i == ' ' || *i == '\t')
    {
      i++;
      continue;
    }
    if (isdigit (*i) || isalpha (*i))
    { p[k++] = *i;
      i++;
    }
    if (*i == '(')
/* push left paranthesis onto stack */
    {
      push (stack, *i);
      i++;
    }
else if (*i == ')')
/*  {
      t = pop (stack);
      while (t != '(')
      {
```

Date :

```
        p[k++] = t;
        t = pop(stack);
    }
    i++;
}
else if (*i == '+' || *i == '-' || *i == '*' || *i == '/' || *i == '%')
{
    while (top1 != -1 && priority(stack[top]) >= priority(*i))
    {
        p[k++] = pop(stack);
    }
    push(stack, *i);
    i++;
}
}
while (top1 != -1) {
    p[k++] = pop(stack);
}
p[k] = '\0';
}
void push(char *stack, char t) {
    if (top == size-1) {
        printf("Stack is full\n");
    }
    else {
        stack[++top] = t;
    }
}
char pop(char *stack) {
    if (top == -1)
```

Date :

```
    {
      printf ("stack is empty \n");
      return '\0';
    }
  return stack [top--];
}
int priority (char t) {
    if (t=='*' || t=='/' || t=='%') {
          return 2;
    }
  else if (t=='+' || t=='-')
        {
          return 1;
        }
  return 0;
}
```

Result - We have executed the program Successfully

Viva questions

1) what do you mean by notation ?
Notation is a symbolic system used to represent mathematical expressions or operations.

2) what is reverse polish attention ?
→ Reverse polish notation (RPN) is a mathematical notation where operator follows their operands, eliminating the

Output -

Enter the infix Expression : A+B*C

The postfix Expression is : ABC*+

Result - We have successfully executed the program

Date :

Need for parenthesis.

3) Which operator is having the highest priority?

→ In Expression the operators * (multiplication), / (division) and % (modulus) generally have the highest priority over addition (+) and substraction (-).

practical - 8

Aim → Write a program to implement the Concept of binary tree.

Date:

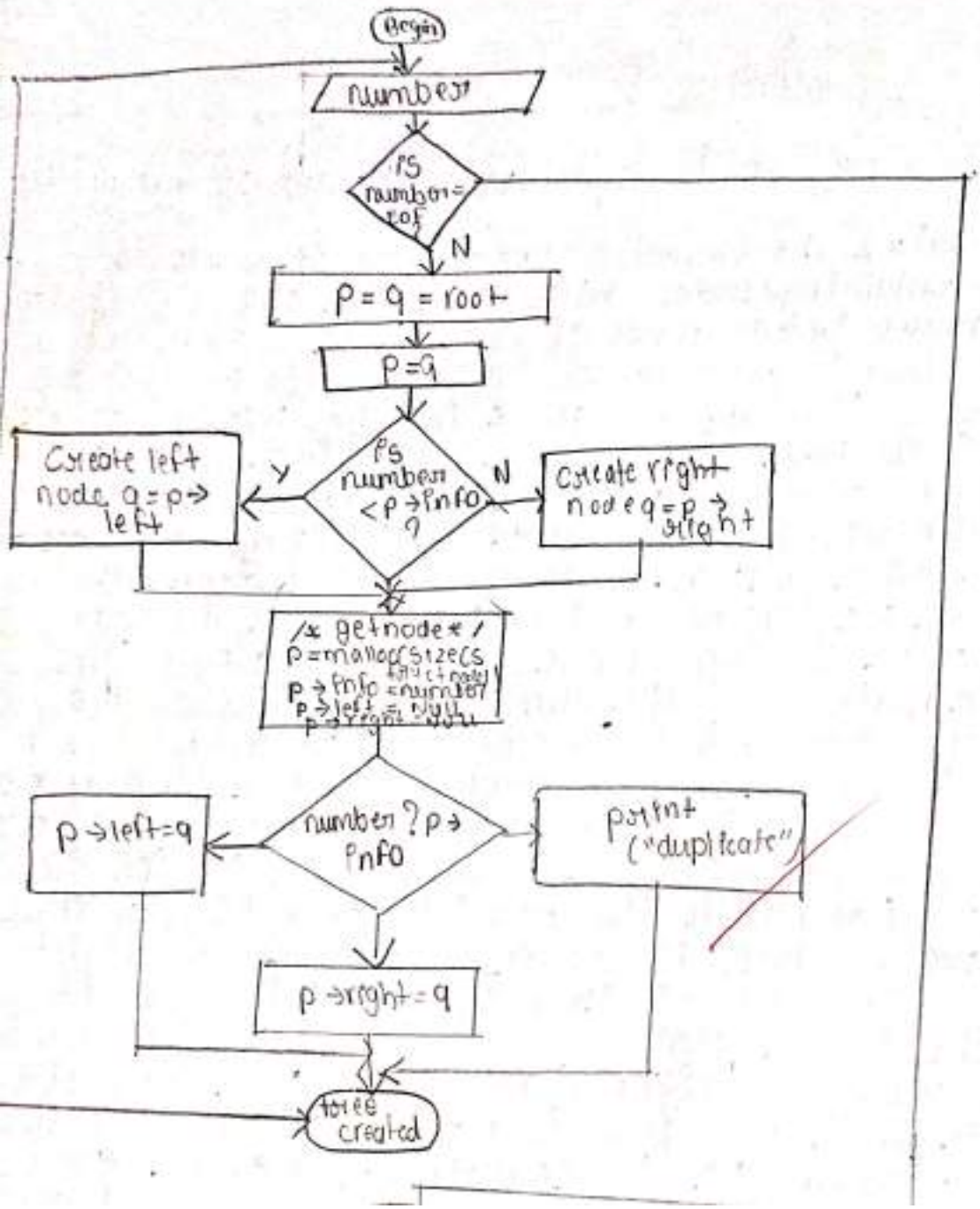
practical no - 8

Aim :- Write a program to implement the concept of binary tree

Theory - We extend the concept of linked data structure to structure containing nodes with more than one self referenced field. A binary tree is made of nodes, where each node contain a "left" reference a "right" reference, and a data element, the topmost node in the tree is called the root. Every node (including a root) in a tree is connected by a directed edge from exactly one other node, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or internal nodes. Nodes with the same parent are called sibling. More tree terminology. The depth of a node is the number of edges from the node to the deepest leaf. The height of a tree is a height of the root. A full binary tree is a binary tree in which each node has exactly zero or two children. A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree is very special tree, it provides the best possible ratio between the number of nodes and the height. The height of a complete binary tree with N nodes is at most O(log N). We can easily prove this by counting nodes on each tree, starting with the root, assuming that each level has the maximum number of nodes.

Algorithm -

Step - Start


Begin
number
IS number = 0 ?
N
p = q = root
p = q
Create left node q = p → left
Y
IS number < p → info q
N
Create right node q = p → right
/* getnode */
p = malloc(sizeof(struct node))
p → info = number
p → left = NULL
p → right = NULL
number ? p → info
p → left = q
printf ("duplicate")
p → right = q
tree created

Date :

Step2 - Define Struct tree
int val
Struct tree *left
Struct tree * right
Define Constant Size = 10
Step3- function Create_bt (arr, lb, ub)
if lb > ub then
return NULL
End if
Step4- mid = [lb + ub]/2
ALLOCATE Memory for Struct tree
S → val = arr[mid]
S → left = Create_bt (arr, lb, mid-1)
S → right = Create_bt (arr, mid+1, ub)
Returns S
End function
Step4 - function display (S, level)
if S1 = NULL THEN
display (S → right, level+1)
print "\n"
For i=0 To level DO
print " "
End FOR
PRINT S → val
display (S → left, level+1)
End if
End function
Step5- Main
Struct tree *t = Null
int arr [size]
int i=0
print "\n\t program to Create binary tree\n"

Date :

While i< SIZE
print "\n Enter any value : (-999 to exit): "
SCAN. arr [i]
if arr [i] === -999 THEN
BREAK
END IF
i++
END WHILE

Step6- T= Create_bt (arr, 0, i-1)
print "\n \t *** TREE ***** \n"
display (T, 1)
Return 0
END MAIN

Step7 - END

Program

```
#include <stdio.h>
#include <stdlib.h>
Struct tree {
        int val;
        Struct tree * left, * right;
    };
  # define SIZE 10
Struct tree * create_bt (int *, int, int);
void display (Struct tree *, int);
 int main() {
   Struct tree * T = NULL;
    int arr [SIZE], i = 0;
printf ("\n \t program to Create binary tree \n
  while (i< size)
printf ("\n Enter any value: (-999 to exit): "); <>
```

Output

```
Enter any value (-999 to exit) =10
Enter any value (-999 to exit) = 20
Enter any value (-999 to exit) = 30
Enter any value (-999 to exit) = 40
Enter any value (-999 to exit) = 50
Enter
```

```
**** Tree ****

        50
      40
    30
20
  10
```

Date :

```
scanf("%d", &arr[i]);
if(arr[i] == -99)
    break;
i++;
}
T = Create_bt(arr, 0, i-1)
printf("\n\t ******* TREE *******\n");
display(T, 1);
return 0;
}
Struct tree * Create_bt(int *arr, int lb, int ub)
{
    if(lb > ub)
        return NULL;
    int mid = (lb + ub)/2
  Struct tree *S = (Struct tree *) malloc(size of (structure
  S -> val = arr[mid];
  S -> left = Create_bt(arr, lb, mid-1);
  S -> right = Create_bt(arr, mid+1, ub);
  return S;
}
void display(Struct tree *S, int level) {
    int i;
 if(S != NULL)
   display(S -> right, level+1);
   printf("\n");
 for(i=0; i<level; i++)
    printf("    ");
 printf("%d", S -> val);
 display(S -> left, level+1);
 }
}
```

Result - we have executed the program Successfully.

Date :

Result - We have executed the program Successfully.

Viva question

1. What do you mean by tree?

→ Tree is a hierarchical data structure consisting of nodes connected by edges, with one node designed as the root. It represents relationship and organizes data in parent-child hierarchy.

What is the binary tree?

→ A binary tree is a type of tree where each node has at most two children, known as the left and right children. This structure allows for efficient data management and traversal.

What is the importance of binary tree?

binary tree optimize search, insertion and deletion operation, making them foundational data structure like binary search trees and heaps.

What do you mean by strictly binary tree?

A strictly binary tree, a full binary tree is one where each node has either 0 or exactly 2 children, ensuring a consistent structure, this property simplifies certain algorithms and ensures balanced growth.

What are the properties of binary tree?

properties include high depth and the number of nodes at each level additionally, a binary tree can be balanced complete or full based on its structure.

Date

Practical no-9

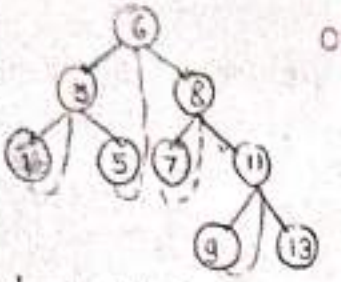
Aim :- Write a program to implement the concept of threaded binary tree.

Theory - Inorder traversal of a binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointer that would normally be NULL point to the inorder successor of the node (if it exists). There are two types of threaded binary trees. Single threaded: Where a NULL right pointer is made to point to the inorder successor (if successor exists) Double Threaded: Where both left and right NULL pointer are made to point to inorderpredeccesor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal. The threads are also useful for fast accessing ancestors of a node. Following diagram shows an example Single threaded Binary tree. The dotted lines represent thread. Representation of a Threaded Node following is c representation of a Single threaded node.

```
Struct node
{
    int data;
    node *left, *right;
    bool right Thread;
}
```

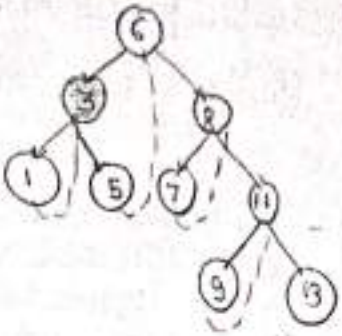
Since right pointer is used for two purposes, the boolean variable right Thread is used to indicate whether right pointer points to right child or inorder

Flowchart



output

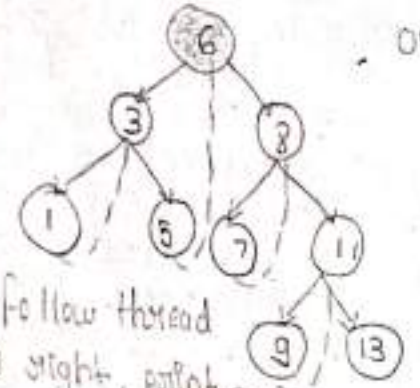
Start at leftmost node, print it



Output

1
3

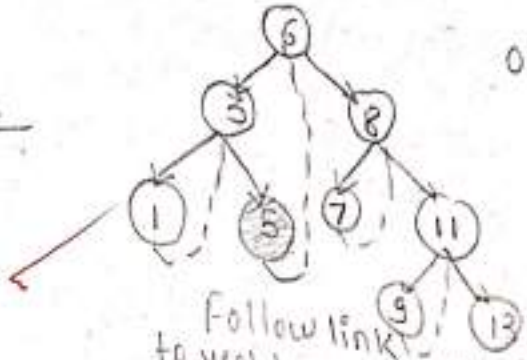
Follow thread to right, print node



Output

1
3
5
6

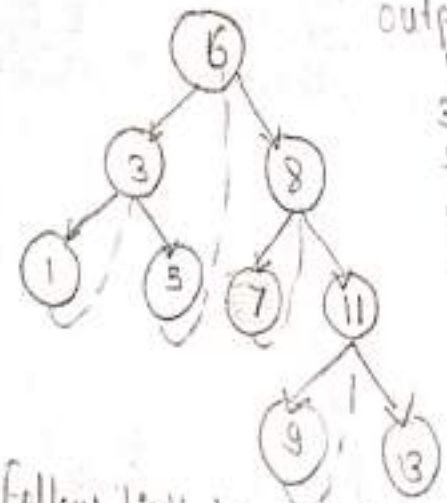
Follow thread to right, print node



Output

1
3
5

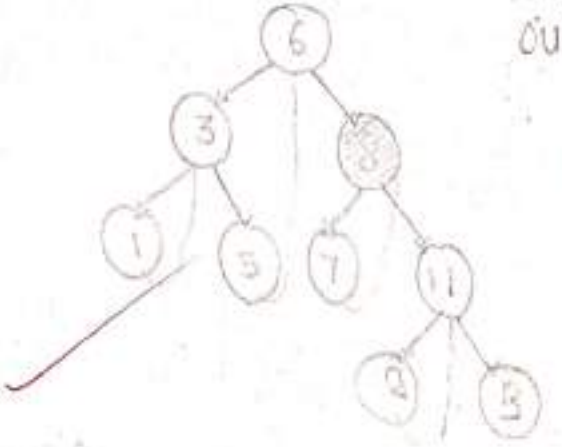
Follow link to right, go to leftmost node and print



Output

1
3
5
6
7

Follow link to right, go to leftmost node and print



Output

1
3
5
6
7
8

follow thread to right, print node

Date :

successor. Similarly, we can add left thread for a double threaded binary tree.

Algorithm

Initialize

- Create a structure tree with values, pointers to left and right children, and thread indicator (t.left, t.right)

Main function:

Initialize T = Null

If -999 is entered, exit

Insert the value into the tree using create_t()

Create tree:

If the tree is empty, Create root node.

Traverse the tree to find the correct position for th new code.

Insert the node in the appropriate position. Updating threaded links (t.left, t.right).

Inordered Traversal

Traverse the tree from the leftmost node.

print each node value in-order sequences using th when necessary.

END

program -

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
Struct tree {
        int val;
```

Date :

```
    int lleft, rright;
    Struct tree *left, *right;
};
Struct tree * Create_t (struct tree *, int);
void in-order_trav (Struct tree *);
int main() {
    int data;
    Struct tree * T = NULL;
printf("\n program for tree Creation:\n");
    While ( ) {
        printf("\n\n\t Enter any value :(-999 to exit):"
        Scanf("%d", &data);
        if (data == -99)
            break;
    T= Create_t (T, data);
}
printf("\n The Inorder Traversal of a Threaded Bina
                Tree :\n\n");
In_order_Trav (T);
    getch();
return 0;
}
Struct tree * Create_t (struct tree *S, int data)
    Struct tree *temp;
    int flag =0;
if (S == NULL)
    S = (Struct tree *) malloc (size of (Struct tree)
        S = val = data;
        S -> left = NULL;
        S -> right = NULL;
        S -> lleft = 1;
        S -> tright = t 1;
```

Date :

```
Return s;
}
Struct tree *q=s;
While (flag != 1)
        if(data < q->val && q->left==0){
            q = q->left;
        }
    else if (data > q->val && q->tright==0)
            {
                q=q->right;
            }
    else if
        if (data<q->val && q->lleft==1) {
            flag=1;
        }
    elseif (data>q->val && q->tright==1) {
            flag=1;
        }
    }
}
temp=(struct tree *)malloc(sizeof(struct tree));
temp->val=data;
temp->lleft=1;
temp->tright=1;
if (data<q->val && q->lleft==1) {
        temp->left = q->left;
        temp->right = q;
        q->left= temp;
        q->lleft=0;
    }
    return s;
}
```

Output -

```
Enter any value: (-999 to Exit): 50
Enter any value: (-999 to exit): 30
Enter any value: (-999 to exit): 70
Enter any value: (-999 to Exit): 20
Enter any value: (-999 to Exit): 40
Enter any value: (-999 to exit): 60
Enter any value: (-999 to exit): 80
Enter any value: (-999 to exit): -999

The In-order traversal of threaded Binary tree:
20 30 40 50 60 70 80
```

Result - We have executed the program successfully.

Date

```
Void In_order_trav (struct tree *s) {
        int flag = 0;
While (S1 = Null)
   While (s -> left == 0 && flag == 0) {
            S = S -> left;
        }
printf (" %.5d ", s -> val);
   if (S -> right == 0)
 {
      S = S -> right;
      flag = 0;
 }
else {
        S = S -> right;
        flag = 1;
     }
   }
}
```

Result - We have executed the program successfully.

drink 11/10/24

Viva question

1. What is threaded binary tree?

→ Threaded binary tree : A threaded binary tree is a type binary tree where the null pointer (that would normally point to child nodes) are replaced with pointers to the in-order predecessor or successor, facilitating fas in-order traversal.

Date :

Q2) What are its advantage and disadvantages ?

→ Advantage

- Allows efficient in-order traversal without recursion or a stack.
- Saves memory by reusing null pointers for threading.

Disadvantage

- Increased complexity in insertion and deletion operation due to the made to maintain threading.
- Slightly higher memory overhead for additional po

Q3 Implement the concept of threaded binary tree ?

→ Threaded binary trees are implemented to optimize traversal operation, making them faster and more memory-efficient compared to standard binary tr

Page No.

Date :

Practical no- 10

Aim :- Write a program to implement the Concept of depth first Search and breadth first Search

Theary :- Traversal of graphs and digraphs To traverse means to visit the vertices in some systematic order. you should be familiar with various traversal methods for trees : preorder - visit each node before its children. postorder: visit each node after its children. (for binary trees only): visit left subtree, node, right subtree. We also saw another kind of traversal, topological ordering, when I talked about shortest paths. Today, we'll see two other traversals: breadth first Search (BFS) and depth first Search (DFS). Both of these construct spanning trees with certain properties useful in other graph algorithm. We'll start by describing them in undirected graphs, but they are both also very useful for directed graphs. Breadth first Search This can be thought of as being like Dijkstra's algorithm for shortest path, but with every edge having the same length. However it is a lot simpler and does n't need any data structure we just keep a tree (the breadth first Search tree), a list of nodes to be added to the tree, and markings (Boolean variables) on the vertices to tell whether they are in the tree or list. breadth first Search : unmark all vertices choose some starting vertex x

```
mark s x
list L=X
tree T=X
while L nonempty
  choose some vertex v from front of list
  visit v
  for each unmarked neighbor w
```

Date :

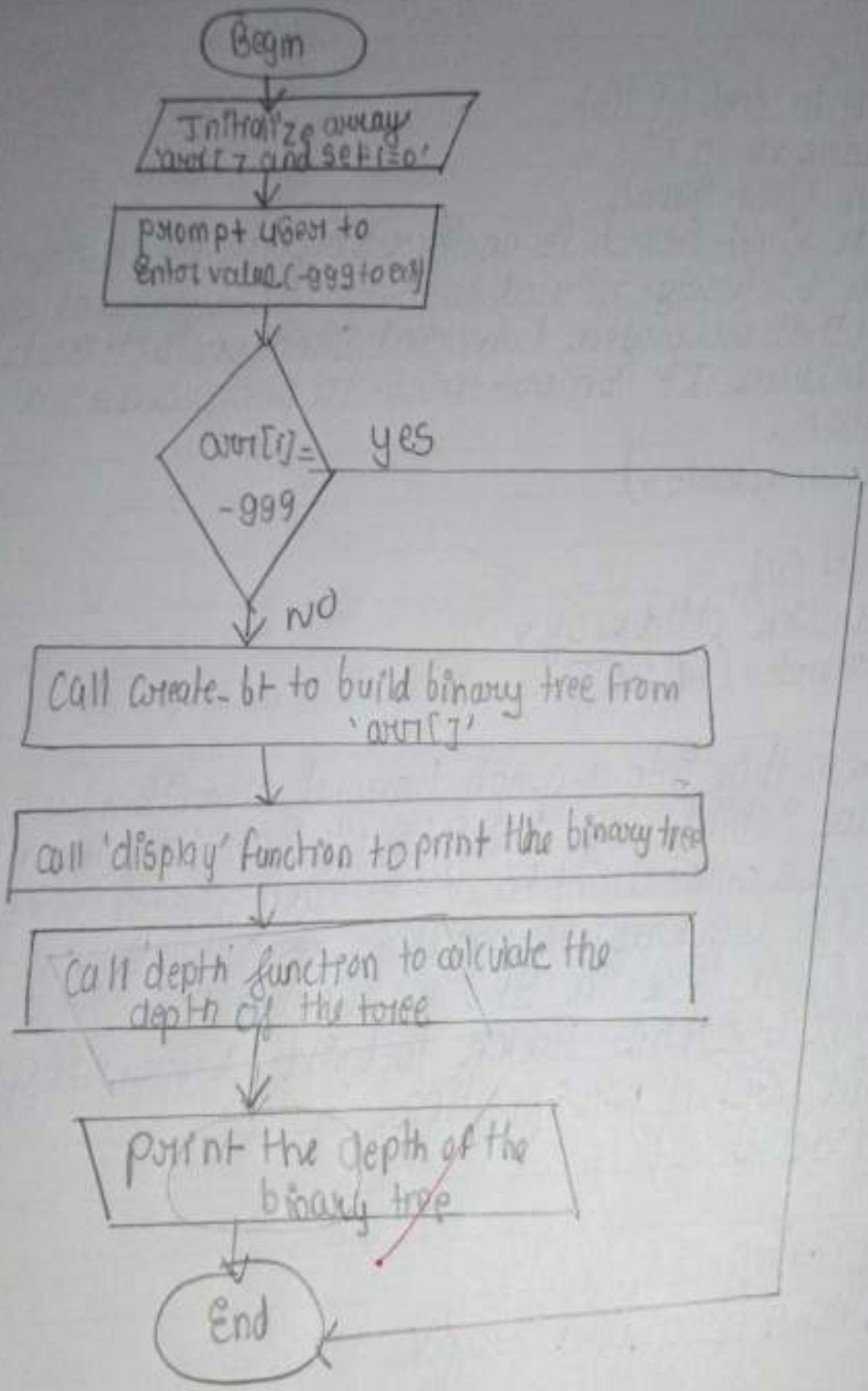
Mark w
add pi to end of list
add edge vw to T
Depth first Search
Depth first Search is another way of traversing graphs, which is closely related to preorder traversal of a tree. Recall that preorder traversal simply visit each node before its children. It is most easy to program as a recursive routine.

```
preorder (node v)
{
  visit (v);
  for each child w of v
   preorder (w);
}
```

To turn this into a graph traversal algorithm, we basically replace "child" by "neighbor". But to prevent infinite loops, we only want to visit each vertex once. Just like in BFS we can use mark to keep track of the vertex once. Just like in BFS we can use Also, Just like in BFS, we can use this search to build a spanning tree with certain useful properties.

```
dfs (vertex v)
{
  visit (v);
  for each neighbor w of v
   {
    dfs (w);
    add edge vw to tree T
   }
}
```

Flowchart


Begin
Initialize array
values and set i=0
prompt user to
enter value (-999 to exit)
arr[i]=
-999
yes
no
call create_bt to build binary tree from
'arr[]'
call 'display' function to print the binary tree
call 'depth' function to calculate the
depth of the tree
print the depth of the
binary tree
End

Date :

Algorithm - 1. main program
- Initialize arr[ ] of size 10.
- Read user input until -999 or array
- call Create_bt(arr, 0, size-1)
- Call display(T, 1)
- Call depth(T, 0)
- print depth of Tree

2. Create_bt(arr[], lb, ub);
- If lb > ub, return Null.
- Find middle element
- Create new node with middle element
- Recursively Create left Subtree.
- Recursively Create right Subtree.
- Return node

3. display(S, level)
- If S == NULL, return
- Recursively display right subtree.
- print Current node with indentation.
- Recursively display left Subtree.

depth(S, level)
- If S == NULL, return depth_bt.
- update depth_bt if current level exceeds depth_bt.
- Recursively calculate depth for left and right Subtrees
- Return maximum depth.

Date :

Program

```
#include <stdio.h>
#include <stdlib.h>
Struct tree{
        int val;
    Struct tree * left, * right;
};
# define SIZE 10
int depth_bt = 0;
Struct tree * create_bt (int *, int, int);
Void display (Struct tree *, int);
int depth (Struct tree *, int);
    int main() {
            Struct tree * T = Null;
          Int arr [SIZE], i = 0;
        printf("\n It program to Create Binary tree\n");
          While (i < SIZE) {
          printf ("\n Enter any value (-999 to exit) : ");
           Scanf ("%d", & arr [i]);
            if (arr [i] == -999)
                break;
           itt;
        }
    T = create_bt (arr 10, i-1);
    printf ("\n It **** Tree **** \n");
      display (T, 1)
      depth_bt = depth (T, 0);
printf ("\n Depth of binary Tree is : %d \n", depth
                                              bt)

Return 0;
}
```

Output

```
Enter any value (-999 to exit) : 10
Enter any value (-999 to exit) : 20
Enter any value (-999 to exit) : 30
enter any value (-999 to exit) : 40
```

**** Tree ****

```
      40
    30
  20
    10
```

Depth of binary tree is 2

Date

```
Struct tree * Create_bt (int arr[], int lb, int ub) {
    int (lb>ub)
        return NULL;
int   mid = (lb+ub)/2;
Struct tree *s = (Struct tree * malloc (size of (struct
                                          tree));

  s→val = arr[mid];
  s→left = Create_bt (arr, lb, mid-1);
  s→right = Create_bt (arr, mid+1, ub);
  return s;
}
Void display (struct tree * s, int level)
{
   int i;
{ if (s != NULL)

  display (s→right, level+1);
  printf ("\n");
  for (i=0; i<level; i++)
  printf ("  ")
  printf ("%d", s→val);
  display (s→left, level+1);
}
int depth (Struct tree *s, int level) {
    if (s != NULL) {
       if (level > depth_bt)
       depth_bt = level;
    depth (s→left, level+1);
    depth (s→right, level+1);
}
return depth_bt;
}
```


Page No.

Date :

Result - We have executed the program Successful.

Viva question

What is graph?

A graph is a collection of nodes connected by edges represent -ing relationship between pairs of objects.

Differentiate between DFS and BFS?

DFS - (Depth First Search) Explores as far as possible along a branch before backtracking, using a stack. It can be memory efficient but may get trapped in deep branches.

BFS. (Breadth First Search) explores all neighbors at the present depth prior to moving on to nodes at the next depth level using a queue guarantees the shortest path in unweight -ed graphs.

What are the different shortest path algorithm?

Dijkstra's algorithm

Bellman-ford algorithm

A* Search algorithm

Floyd - warshall algorithm

At the time of implementation of DFS which data structure is used?

A stack is used for implementation, either explicity or through recursion (call stack).