DATE :

EXPERIMENT NO – 1.1(i)
CPU SCHEDULING ALGORITHM - FCFS

AIM

To simulate the First Come First Serve scheduling algorithm to find turn around time and waiting time .

ALGORITHM

1. Start the program
2. **Define Process Structure:**
   - Create a structure named **Process** with attributes like **pid** (Process ID), **burst_time** (Burst time), **at** (Arrival Time), **tat** (Turnaround Time), and **waittime** (Waiting Time).
3. **Function fcfsScheduling:**
   - Inputs an array of **Process** structures (**proc[]**) and the total number of processes **n**.
   - Sorts the processes based on their Arrival Time in ascending order using the Bubble Sort algorithm:
     - Iterates through the array comparing the arrival time of adjacent processes.
     - Swaps the processes if their arrival time is not in the correct order.
   - Initializes variables for calculating total waiting time, total turnaround time, and the current time.
   - Iterates through the processes:
     - Checks if the current process's Arrival Time (**at**) is less than or equal to the current time (**time**):
       - If true, increments the time by the process's burst time (**burst_time**).
       - Calculates Turnaround Time (**tat**) and Waiting Time (**waittime**) for the process.
       - Updates the total turnaround time and total waiting time accordingly.
       - Moves to the next process.
       - If the Arrival Time condition is not met, increments time by 1.
   - Displays the details of each process: Process ID, Burst Time, Arrival Time, Waiting Time, and Turnaround Time.

OUTPUT

```
Enter the number of processes: 5

Enter process details (Process ID, Burst Time, Arrival Time):
Process 1: 4
9
0
Process 2: 3
6
2
Process 3: 5
4
3
Process 4: 1
8
0
Process 5: 2
11
3
Process         BT      AT      WT      TAT
4               9       0       0       9
1               8       0       9       17
3               6       2       15      21
5               4       3       20      24
2               11      3       24      35

Average Waiting Time: 13.60
Average Turnaround Time: 21.20
```

    - Calculates and displays the Average Waiting Time and Average Turnaround Time for all processes.
4. **Function main:**
    - Declares variables **n** and **i** to store the number of processes and loop counters, respectively.
    - Prompts the user to input the number of processes.
    - Creates an array **proc[]** of **Process** structures with a size of **n**.
    - Prompts the user to input details (Process ID, Burst Time, Arrival Time) for each process.
    - Calls the **fcfsScheduling** function passing the array of processes and the total number of processes.
5. Stop the program .

PROGRAM

```
#include<stdio.h>
struct Process {
  int pid;
  int burst_time;
  int at;
  int tat;
  int waittime;
};
void fcfsScheduling(struct Process proc[], int n) {
    int i, j;
    struct Process temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
                if (proc[j].at > proc[j + 1].at) {
                    temp = proc[j];
                    proc[j] = proc[j + 1];
                    proc[j + 1] = temp;
                }
        }
    }
   int total_waiting_time = 0;
   int total_turnaround_time = 0;
   int time=0;
   i=0;
```

```
    while (i<n) {
        if(proc[i].at<=time){
              time+=proc[i].burst_time;
              proc[i].tat=time-proc[i].at;
              proc[i].waittime=proc[i].tat-proc[i].burst_time;
              total_turnaround_time+=proc[i].tat;
              total_waiting_time+=proc[i].waittime;
              i++;
        }
        Else{
              time++;
        }
   }
   printf("Process\tBT\tAT\tWT \t TAT\n");
   for (i = 0; i < n; i++) {
       printf("%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].burst_time,
       proc[i].at,proc[i].waittime,proc[i].waittime+proc[i].burst_time);
   }
    printf("\nAverage Waiting Time: %.2f\n", (float) total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float) total_turnaround_time / n);
}
void main() {
 int n, i;
 printf("Enter the number of processes: ");
 scanf("%d", &n);
 struct Process proc[n];
 printf("Enter process details (Process ID, Burst Time, Arrival Time):\n");
 for (i = 0; i < n; i++) {
 printf("Process %d: ", i + 1);
 scanf("%d %d %d", &proc[i].pid, &proc[i].burst_time, &proc[i].at);
 }
 fcfsScheduling(proc, n);
}
```

<u>RESULT</u>

C program to implement FCFS scheduling algorithm has been executed successfully .

DATE :

EXPERIMENT NO – 1.1(ii)
CPU SCHEDULING ALGORITHM - SJF

AIM

To simulate the Shortest Job First scheduling algorithm to find turn around time and waiting time .

ALGORITHM

1. Start the program
2. **Include Necessary Libraries:** The algorithm starts by including necessary header files like **<stdio.h>** and **<limits.h>**.
3. **Define a Process Structure:** It defines a structure named **process** that holds various attributes of a process - arrival time (**at**), burst time (**bt**), waiting time (**wt**), turnaround time (**tat**), completion time (**ct**), and a status flag (**status**) indicating whether the process has been executed or not.
4. **Main Function:**
   - Declare variables **n**, **i**, **totalTAT**, **totalWT**, **completed**, and **currentTime**.
   - Prompt the user to enter the number of processes (**n**).
   - Create an array of **process** structures (**a[n]**).
   - Ask the user to input arrival time and burst time for each process while calculating the total burst time (**sum**) of all processes.
   - Initialize the status of each process as **0** (not executed) initially.
5. **Scheduling Algorithm:**
   - Enter a while loop that runs until all processes are completed.
   - Initialize variables **sJob** and **sBt**.
   - Iterate through each process and find the process with the shortest burst time (**sBt**) among the processes that have arrived (**at <= currentTime**) and have not yet been executed (**status=0**).
   - If a suitable process is found (**sJob != -1**), update its completion time, calculate its turnaround time and waiting time, update total turnaround time and waiting time, mark it as completed (**status=1**), and increment **completed**.
   - If no suitable process is found, increment **currentTime**.
6. **Calculate Averages and Display Results:**
   - Calculate the average turnaround time (**avgTAT**) and average waiting time (**avgWT**).
   - Display a table with process details including arrival time, burst time, completion time, turnaround time, and waiting time for each process.

OUTPUT

```
Enter the number of processes: 5
Enter the arrival time and burst time
2
6
5
2
1
8
0
3
4
4

PNo     AT      BT      CT      TAT     WT
P1      2       6       9       7       1
P2      5       2       11      6       4
P3      1       8       23      22      14
P4      0       3       3       3       0
P5      4       4       15      11      7

Average Waiting time = 5.200000
Average Turn Around time = 9.800000
```

- Print the calculated average waiting time and average turnaround time.

7. **End of Algorithm.**

PROGRAM

```
#include <stdio.h>
#include <limits.h>
typedef struct{
   int at,bt,wt,tat,ct,status;
}process;
int main()
{
   int n,i;
   int totalTAT=0,totalWT=0,completed=0,currentTime=0;
   printf("Enter the number of processes: ");
   scanf("%d",&n);
   process a[n];
   printf("Enter the arrival time and burst time\n");
   int sum=0;
   for(i=0;i<n;i++)
   {
      scanf("%d",&a[i].at);
      scanf("%d",&a[i].bt);
      sum+=a[i].bt;
      a[i].status=0;
   }
   while(completed<n)
   {
      int sJob=-1;
      int sBt = INT_MAX;
      for(int i=0;i<n;i++)
      {
         if(a[i].status==0 && a[i].bt<sBt && a[i].at<=currentTime)
         {
            sJob = i;
            sBt = a[i].bt;
         }
      }
      if(sJob==-1)
```

```
        {
           currentTime++;
        }
        else
        {
           a[sJob].ct=currentTime+a[sJob].bt;
           currentTime = a[sJob].ct;
           a[sJob].tat = a[sJob].ct-a[sJob].at;
           a[sJob].wt = a[sJob].tat-a[sJob].bt;
           totalTAT+=a[sJob].tat;
           totalWT+=a[sJob].wt;
           completed++;
              a[sJob].status=1;
        }
     }
  float avgTAT = (float)totalTAT/n;
     float avgWT = (float)totalWT/n;
     printf("\nPNo\tAT\tBT\tCT\tTAT\tWT\n");
     for(i=0;i<n;i++)
     {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n",i+1,a[i].at,a[i].bt,a[i].ct,a[i].tat,a[i].wt);
     }
     printf("\n");
     printf("Average Waiting time = %f\n",avgWT);
     printf("Average Turn Around time = %f\n",avgTAT);
  }
```

RESULT

C program to implement SJF scheduling algorithm has been executed successfully .

DATE :

EXPERIMENT NO – 1.1(iii)
CPU SCHEDULING ALGORITHM – ROUND ROBIN

AIM

To simulate the Round Robin scheduling algorithm to find turn around time and waiting time .

ALGORITHM

1. Start the program
2. Initialize arrays and variables: Initialize arrays to store arrival time (at), burst time (bt), remaining time (rt), process IDs (id), waiting time (wt), turnaround time (tat), and a queue for the ready processes. Initialize front and rear of the queue.
3. Insert and Delete functions: Include functions to insert and delete elements into/from the queue.
4. Main function:
    - Take input for the number of processes (n), arrival time, and burst time for each process.
    - Input the time quantum (TQ).
5. Round Robin Scheduling:
    - Insert the first process into the ready queue and mark it as existing.
    - While the ready queue is not empty:
        - Remove a process from the queue.
        - If the remaining time of the process is greater than or equal to the time quantum (TQ), reduce its remaining time by TQ and increase the total time by TQ.
        - If the remaining time of the process is less than TQ, execute the remaining time and increase the total time accordingly.
        - Check for other processes that have arrived and insert them into the queue if they haven't been added yet.
        - Calculate turnaround time (TAT) and waiting time (WT) for each process.
        - Update the total turnaround time and waiting time.
6. Calculate averages: Calculate the average waiting time and average turnaround time for all processes.
7. Output: Display the process IDs, burst time, arrival time, waiting time, and turnaround time for each process. Also, print the average waiting time and average turnaround time for all processes.
8. Return: Return 0 to indicate successful execution.
9. Stop the program

OUTPUT

```
Enter the number of the process
3
Enter the arrival time and burst time of the process
AT      BT
0       10
1       8
2       7
Enter the time quantum
5
ID      BT       AT      WT      TAT
0       10      0       10      20
1       8       1       14      22
2       7        2      16      23
Average waiting time of the processes is : 13.333333
Average turn around time of the processes is : 21.666666
```

PROGRAM

```
#include<stdio.h>
int at[10],bt[10],rt[10],TQ,id[10],wt[10],tat[10];
// declaration of the ready queue
int queue[100];
int front=-1;
int rear=-1;
void insert(int n)
{
 if(front==-1)
 front=0;
 rear=rear+1;
 queue[rear]=n;
}
int delete()
{
 int n;
 n=queue[front];
 front=front+1;
 return n;
}
int main()
{
 int n,TQ,p,TIME=0;
 int temp[10],exist[10]={0};
 float total_wt=0,total_tat=0,Avg_WT,Avg_TAT;
 printf("Enter the number of the process\n");
 scanf("%d",&n);
 printf("Enter the arrival time and burst time of the process\n");
 printf("AT BT\n");
 for(int i=0;i<n;i++)
 {
 scanf("%d%d",&at[i],&bt[i]);
 id[i]=i;
 rt[i]=bt[i];
 }
 printf("Enter the time quantum\n");
 scanf("%d",&TQ);
```

```
insert(0);
exist[0]=1;
while(front<=rear)
{
p=delete();
if(rt[p]>=TQ)
{
rt[p]-=TQ;
TIME=TIME+TQ;
}
else
{
TIME=TIME+rt[p];
rt[p]=0;
}
for(int i=0;i<n;i++)
{
if(exist[i]==0 && at[i]<=TIME)
{
insert(i);
exist[i]=1;
}
}
if(rt[p]==0)
{
tat[p]=TIME-at[p];
wt[p]=tat[p]-bt[p];
total_tat=total_tat+tat[p];
total_wt=total_wt+wt[p];
}
else
{
insert(p);
}
}
Avg_TAT=total_tat/n;
Avg_WT=total_wt/n;
printf("ID BT AT WT TAT\n");
```

```
 for(int i=0;i<n;i++)
 {
 printf("%d  %d  %d  %d  %d\n",id[i],bt[i],at[i],wt[i],tat[i]);
 }
 printf("Average waiting time of the processes is : %f\n",Avg_WT);
 printf("Average turn around time of the processes is : %f\n",Avg_TAT);
 return 0;
}
```

<u>RESULT</u>

C program to implement Round Robin scheduling algorithm has been executed successfully .

DATE :

EXPERIMENT NO – 1.1(iv)
CPU SCHEDULING ALGORITHM - PRIORITY

AIM

To simulate the Priority scheduling algorithm to find turn around time and waiting time .

ALGORITHM

1. Start the program
2. **Input**: Receive the number of processes **n**.
3. Create a **process** struct with attributes: **id**, **at** (arrival time), **bt** (burst time), **ct** (completion time), **tat** (turnaround time), **wt** (waiting time), **rt** (remaining time), and **priority**.
4. Input the details of each process: id, arrival time, burst time, and priority.
5. Initialize variables **completed**, **totalWT**, **totalTAT**, and **currentTime** to keep track of completed processes and total waiting and turnaround times.
6. Perform scheduling until all processes are completed:
    - Inside the loop:
        - Initialize variables **sJob** and **sPriority**.
        - Find the suitable job based on priority that meets the following criteria:
            - Arrival time is less than or equal to the current time.
            - Priority is the highest among the available processes.
            - The process hasn't finished executing yet.
        - If no suitable job is found, increment the **currentTime**.
        - If a suitable job is found:
            - Execute the selected job by decrementing its remaining time.
            - Update **currentTime**.
            - If the remaining time for the job becomes zero:
                - Update completion time, turnaround time, and waiting time.
                - Increment the **completed** count.
7. Calculate average waiting time (**avgWT**) and average turnaround time (**avgTAT**).
8. Display process details and averages:
    - Print the details of each process: id, arrival time, burst time, priority, completion time, turnaround time, and waiting time.
    - Display the calculated average waiting time and average turnaround time.
9. **Output**: Display the process details and calculated averages.
10. Stop the program

OUTPUT

```
Enter the number of processes: 3
Enter the id, arrival time, burst time, and priority of 3processes:
1 0 5 3
2 3 3 1
3 5 2 2

ID      AT      BT      Prio    CT      TAT     WT
P1      0       5       3       10      10      5
P2      3       3       1       6       3       0
P3      5       2       2       8       3       1
Average Waiting Time = 2.000000
Average Turnaround Time = 5.333333
```

PROGRAM

```
#include <stdio.h>
typedef struct {
int id, at, bt, ct, tat, wt, rt, priority;
} process;
int main() {
int n, currentTime = 0;
float avgTAT, avgWT;
printf("Enter the number of processes: ");
scanf("%d", &n);
process a[n];
printf("Enter the id, arrival time, burst time, and priority of %dprocesses:\n", n);
for (int i = 0; i < n; i++) {
scanf("%d", &a[i].id);
scanf("%d", &a[i].at);
scanf("%d", &a[i].bt);
scanf("%d", &a[i].priority);
a[i].rt = a[i].bt;
}
int completed = 0;
int totalWT = 0;
int totalTAT = 0;
while (completed < n) {
int sJob = -1;
int sPriority = 99999;
for (int i = 0; i < n; i++) {
if (a[i].at <= currentTime && a[i].priority < sPriority &&
a[i].rt > 0) {
sJob = i;
sPriority = a[i].priority;
}
}
if (sJob == -1) {
currentTime++;
} else {
a[sJob].rt--;
currentTime++;
if (a[sJob].rt == 0) {
```

```
completed++;
a[sJob].ct = currentTime;
a[sJob].tat = currentTime - a[sJob].at;
a[sJob].wt = a[sJob].tat - a[sJob].bt;
totalTAT += a[sJob].tat;
totalWT += a[sJob].wt;
}
}
}
avgTAT = (float)totalTAT / n;
avgWT = (float)totalWT / n;
printf("\nID\tAT\tBT\tPrio\tCT\tTAT\tWT");
for (int i = 0; i < n; i++) {
printf("\nP%d\t%d\t%d\t%d\t%d\t%d\t%d", a[i].id, a[i].at, a[i].bt,
a[i].priority, a[i].ct, a[i].tat, a[i].wt);
}
printf("\nAverage Waiting Time = %f", avgWT);
printf("\nAverage Turnaround Time = %f\n", avgTAT);
return 0;
}
```

RESULT

C program to implement Priority scheduling algorithm has been executed successfully .

DATE :

EXPERIMENT NO – 1.2(i)
SEQUENTIAL FILE ALLOCATION

AIM

To simulate the sequential file allocation strategy .

ALGORITHM

```
1. Start the program
2. Declare variables: n, i, j, x as integers, b[20], sb[20], t[20], c[20][20] as integer arrays
3. Read the number of files 'n' from the user
4. For i = 0 to n-1:
   a. Read the number of blocks occupied by file[i] from the user and store it in b[i]
   b. Read the starting block of file[i] from the user and store it in sb[i]
   c. Set t[i] = sb[i]
   d. For j = 0 to b[i]-1:
      i. Assign sb[i] to c[i][j] and increment sb[i] by 1
5. Display the table header: "Filename    Start block    Length"
6. For i = 0 to n-1:
   a. Display file details: file[i]+1, t[i], b[i]
7. Read the file name 'x' from the user
8. Display "File name is x" and "Length is b[x-1]"
9. Display "Blocks occupied:" followed by blocks stored in c[x-1][i] for i = 0 to b[x-1]-1
10. End
```

PROGRAM

```
#include<stdio.h>
void main()
{
 int n,i,j,b[20],sb[20],t[20],x,c[20][20];
 printf("Enter the no.of files:");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
 printf("Enter the number of blocks occupied by file%d:",i+1);
 scanf("%d",&b[i]);
 printf("Enter the starting block of file%d:",i+1);
 scanf("%d",&sb[i]);
```

OUTPUT

```
Enter the no.of files:3
Enter the number of blocks occupied by file1:3
Enter the starting block of file1:5
Enter the number of blocks occupied by file2:2
Enter the starting block of file2:9
Enter the number of blocks occupied by file3:3
Enter the starting block of file3:15
Filename        Start block      length
1                    5               3
2                    9               2
3                   15               3
Enter file name:2
File name is 2
Length is 2
Blocks occupied:   9  10
```

```
t[i]=sb[i];
for(j=0;j<b[i];j++)
{
c[i][j]=sb[i]++;
}
}
printf("Filename\tStart block\tlength\n");
for(i=0;i<n;i++)
printf("%d\t %d\t %d\n",i+1,t[i],b[i]);
printf("Enter file name:");
scanf("%d",&x);
printf("File name is %d\n",x);
printf("Length is %d\n",b[x-1]);
printf("Blocks occupied:");
for(i=0;i<b[x-1];i++)
printf("%4d",c[x-1][i]);
}
```

<u>RESULT</u>

Program to implement sequential file allocation strategy implemented successfully .

DATE :

EXPERIMENT NO – 1.2(ii)
INDEXED FILE ALLOCATION

AIM

To simulate the indexed file allocation strategy .

ALGORITHM

```
1. Start the program
2. Declare variables: n, m[20], i, j, sb[20], s[20], b[20][20], x
3. Display "Enter no. of files:"
4. Input the number of files n
5. Loop from i = 0 to n - 1:
      ➢ Display "Enter starting block and size of file i + 1:"
      ➢ Input starting block sb[i] and size s[i] of the file
      ➢ Display "Enter blocks occupied by file i + 1:"
      ➢ Input the number of blocks occupied by the file m[i]
      ➢ Display "Enter blocks of file i + 1:"
      ➢ Loop from j = 0 to m[i] - 1:
            • Input block numbers b[i][j] occupied by the file
6. Display "\nFile\t index\tlength"
7. Loop from i = 0 to n - 1:
      ➢ Display file index, starting block, and length: i + 1, sb[i], m[i]
8. Display "Enter file name:"
9. Input the file name x
10. Display "file name is: x"
11. Set i = x - 1
12. Display "Index is: sb[i]"
13. Display "Block occupied are:"
14. Loop from j = 0 to m[i] - 1:
      ➢ Display blocks occupied by file b[i][j]
15. End
```

PROGRAM

```
#include<stdio.h>
void main()
{
 int n,m[20],i,j,sb[20],s[20],b[20][20],x;
```

OUTPUT

```
Enter no. of files:2
Enter starting block and size of file1:2 4
Enter blocks occupied by file1:4
enter blocks of file1:7
6
8
4
Enter starting block and size of file2:3 5
Enter blocks occupied by file2:3
enter blocks of file2:1
5
9

File       index   length
1            2          4
2            3          3

Enter file name:2
file name is:2
Index is:3Block occupied are:  1  5  9
```

```
 printf("Enter no. of files:");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
 printf("Enter starting block and size of file%d:",i+1);
 scanf("%d%d",&sb[i],&s[i]);
 printf("Enter blocks occupied by file%d:",i+1);
 scanf("%d",&m[i]);
 printf("enter blocks of file%d:",i+1);
 for(j=0;j<m[i];j++)
 scanf("%d",&b[i][j]);
 }
 printf("\nFile\t index\tlength\n");
 for(i=0;i<n;i++)
 printf("%d\t%d\t%d\n",i+1,sb[i],m[i]);
 printf("\nEnter file name:");
 scanf("%d",&x);
 printf("file name is:%d\n",x);
 i=x-1;
 printf("Index is:%d",sb[i]);
 printf("Block occupied are:");
 for(j=0;j<m[i];j++)
 printf("%3d",b[i][j]);
}
```

RESULT

Program to implement indexed file allocation strategy implemented successfully .

DATE :

EXPERIMENT NO – 1.2(iii)
LINKED FILE ALLOCATION

AIM

To simulate the linked file allocation strategy .

ALGORITHM

1. Start the program
2. Structures Defined:
    - ➢ Block: Contains data and a pointer to the next block.
    - ➢ File: Contains a name and a pointer to the first block.
3. Function createFile:
    - ➢ Check if the number of files has not exceeded the limit.
    - ➢ Allocate memory for a new file.
    - ➢ Initialize the file's name and set its first block to NULL.
    - ➢ Store the file in the array of files and increment the file count.
    - ➢ Print a success message.
4. Function allocateBlock:
    - ➢ Allocate memory for a new block.
    - ➢ Set the block's data to the provided block number and its next pointer to NULL.
    - ➢ If the file doesn't have any blocks yet, assign the new block as the first block.
    - ➢ Otherwise, traverse the blocks of the file to the last one and append the new block.
    - ➢ Print a message indicating the block allocation.
5. Function displayFiles:
    - ➢ Check if no files are created; if so, print a message indicating the absence of files.
    - ➢ Otherwise, iterate through each file in the array:
        - Print the file name.
        - Traverse the blocks of the file and print their data.
        - Print a newline.
6. main Function:
    - ➢ Initialize an array to hold file pointers and other necessary variables.
    - ➢ Start an infinite loop to display a menu and take user input.
    - ➢ Provide options to:
        - Create a new file, taking its name as input.
        - Allocate a block to an existing file by index.
        - Display all created files and their allocated blocks.

OUTPUT

```
Linked File Allocation Simulation
1. Create a new file
2. Allocate a block to a file
3. Display files and allocated blocks
4. Exit
Enter your choice: 1
Enter the name of the new file: CSA
File 'CSA' created successfully.

Linked File Allocation Simulation
1. Create a new file
2. Allocate a block to a file
3. Display files and allocated blocks
4. Exit
Enter your choice: 2
Enter the index of the file (0-0): 0
Block 1 allocated to file 'CSA'.

Linked File Allocation Simulation
1. Create a new file
2. Allocate a block to a file
3. Display files and allocated blocks
4. Exit
Enter your choice: 3
List of files:
File 'CSA': Blocks -> 1

Linked File Allocation Simulation
1. Create a new file
2. Allocate a block to a file
3. Display files and allocated blocks
4. Exit
Enter your choice: 2
Enter the index of the file (0-0): 0
Block 2 allocated to file 'CSA'.

Linked File Allocation Simulation
```

        - Exit the program.
7. Menu Implementation in main:
    - Display a menu to the user.
    - Accept user choice using scanf.
    - Use a switch-case statement to perform actions based on the user's choice:
        - Create a file with the given name.
        - Allocate a block to a specified file (if files exist).
        - Display all files and their allocated blocks.
        - Exit the program if the user chooses.
8. Error Handling:
    - Check for invalid inputs (such as invalid file indices or invalid choices).
    - Display appropriate error messages.
9. Loop Continuity:
    - Keep the program running until the user chooses to exit.
10. End

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
struct Block {
  int data;
  struct Block* next;
};
struct File {
  char name[20];
  struct Block* firstBlock;
};
void createFile(struct File** files, int* numFiles, char* name) {
  if (*numFiles >= 50) {
    printf("Cannot create more files. Limit reached.\n");
    return;
  }
  struct File* newFile = (struct File*)malloc(sizeof(struct File));
  if (!newFile) {
    printf("Memory allocation failed.\n");
    return;
  }
```

```
1. Create a new file
2. Allocate a block to a file
3. Display files and allocated blocks
4. Exit
Enter your choice: 2
Enter the index of the file (0-0): 0
Block 3 allocated to file 'CSA'.

Linked File Allocation Simulation
1. Create a new file
2. Allocate a block to a file
3. Display files and allocated blocks
4. Exit
Enter your choice: 3
List of files:
File 'CSA': Blocks -> 1 2 3

Linked File Allocation Simulation
1. Create a new file
2. Allocate a block to a file
3. Display files and allocated blocks
4. Exit
Enter your choice: 4
```

```
    strcpy(newFile->name, name);
    newFile->firstBlock = NULL;
    files[*numFiles] = newFile;
    (*numFiles)++;
    printf("File '%s' created successfully.\n", name);
}
void allocateBlock(struct File* file, int blockNumber) {
    struct Block* newBlock = (struct Block*)malloc(sizeof(struct Block));
    if (!newBlock) {
        printf("Memory allocation failed.\n");
        return;
    }
    newBlock->data = blockNumber;
    newBlock->next = NULL;
    if (!file->firstBlock) {
        file->firstBlock = newBlock;
    } else {
        struct Block* current = file->firstBlock;
        while (current->next) {
            current = current->next;
        }
        current->next = newBlock;
    }
    printf("Block %d allocated to file '%s'.\n", blockNumber, file->name);
}
void displayFiles(struct File** files, int numFiles) {
    if (numFiles == 0) {
        printf("No files created yet.\n");
        return;
    }
    printf("List of files:\n");
    for (int i = 0; i < numFiles; i++) {
        struct File* file = files[i];
        printf("File '%s': Blocks -> ", file->name);
        struct Block* current = file->firstBlock;
        while (current) {
            printf("%d ", current->data);
            current = current->next;
```

```
        }
        printf("\n");
    }
}
int main() {
    struct File* files[50];
    int numFiles = 0;
    int blockNumber = 1;
    int choice;
    char fileName[20];
    while (1) {
        printf("\nLinked File Allocation Simulation\n");
        printf("1. Create a new file\n");
        printf("2. Allocate a block to a file\n");
        printf("3. Display files and allocated blocks\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the name of the new file: ");
                scanf("%s", fileName);
                createFile(files, &numFiles, fileName);
                break;
            case 2:
                if (numFiles == 0) {
                    printf("No files created yet. Please create a file first.\n");
                } else {
                    int fileIndex;
                    printf("Enter the index of the file (0-%d): ", numFiles - 1);
                    scanf("%d", &fileIndex);
                    if (fileIndex >= 0 && fileIndex < numFiles) {
                        allocateBlock(files[fileIndex], blockNumber);
                        blockNumber++;
                    } else {
                        printf("Invalid file index.\n");
                     }
                }
```

```
                break;
            case 3:
                displayFiles(files, numFiles);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }
    return 0;
}
```

RESULT

Program to implement linked file allocation strategy implemented successfully .

DATE :

EXPERIMENT NO – 1.3(i)
FILE ORGANIZATION – SINGLE LEVEL DIRECTORY

AIM

To simulate the single level directory file organization technique .

ALGORITHM

1. **Variables Initialization:**
   - Initialize variables: **nf** (number of files), **i** (loop iterator), **ch** (choice), **mdname** (directory name), **fname** (array to store file names).
2. **Input Directory Name:**
   - Prompt the user to input a directory name using **printf**.
   - Use **scanf** to read and store the directory name in the variable **mdname**.
3. **File Creation Loop:**
   - Start a do-while loop to repeatedly create files based on user input.
   - Inside the loop:
     - Prompt the user to enter the name of the file to be created using **printf**.
     - Read the file name entered by the user using **scanf** and store it in the variable **name**.
     - Iterate through the existing file names stored in the **fname** array to check if the entered file name already exists.
       - Use a **for** loop to check each element of the **fname** array.
       - Compare the entered file name (**name**) with existing file names (**fname[i]**) using **strcmp**.
       - If a match is found (**strcmp** returns 0), break the loop.
     - If the entered file name doesn't match any existing file name:
       - Store the new file name in the **fname** array at index **nf**.
       - Increment **nf** (number of files) to keep track of the number of files created.
     - If the entered file name matches an existing file name, display an appropriate message indicating the file already exists.
     - Prompt the user to choose whether to enter another file (yes - 1 or no - 0) using **printf**.
     - Read the user's choice using **scanf** and store it in the variable **ch**.
4. **Display Directory Name and File Names:**
   - After the loop exits (when the user chooses not to enter another file), display the directory name and the list of created file names.

OUTPUT

```
Enter the directory name: dir1
Enter file name to be created:file1
Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:file2
Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:file3
Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:file1
There is already file1
Do you want to enter another file(yes - 1 or no - 0):0
Directory name is:dir1
Files names are...
file1
file2
file3
```

- Print the directory name (**mdname**) using **printf**.
- Display a message indicating the list of file names using **printf**.
- Use a **for** loop to iterate through the **fname** array and print each file name (**fname[i]**) on a new line using **printf**.

5. **End of Program**

PROGRAM

```
#include<stdio.h>
#include<string.h>
void main()
{
int nf=0,i=0,ch;
char mdname[10],fname[10][10],name[10];
 printf("Enter the directory name: ");
scanf("%s",mdname);
 do
{
printf("Enter file name to be created:");
scanf("%s",name);
for(i=0;i<nf;i++)
{
if(!strcmp(name,fname[i]))
break;
}
if(i==nf)
strcpy(fname[nf++],name);
 else
printf("There is already %s\n",name);
printf("Do you want to enter another file(yes - 1 or no - 0):");
scanf("%d",&ch);
} while(ch==1);
printf("Directory name is:%s\n",mdname);
printf("Files names are...\n");
for(i=0;i<nf;i++)
printf("%s\n",fname[i]);
}
```

RESULT

Program to implement single level file organization technique has been executed successfully .

DATE :

EXPERIMENT NO – 1.3(ii)
FILE ORGANIZATION – TWO LEVEL DIRECTORY

AIM

To simulate the two level directory file organization technique .

ALGORITHM

1. Start of the program
2. Structure Definition:
   - Define a structure to hold directory names (dname), file names (fname), and file count (fcnt).
3. Variables Initialization:
   - Initialize variables: dir (an array of structures to hold directories and files), dcnt (directory count), i, ch, k, f, d.
4. Menu-driven System:
   - Start an infinite loop (while (1)) to display a menu and take user input.
   - Display a menu with options for directory and file operations: create directory, create file, delete file, search file, display, and exit.
   - Use scanf to accept the user's choice (ch).
5. Switch-case Statement:
   - Use a switch statement to perform actions based on the user's choice:
   - Case 1 (Create Directory):
     - Prompt the user to enter the name of the directory.
     - Read the directory name using scanf and store it in dir[dcnt].dname.
     - Initialize the file count (dir[dcnt].fcnt) for this directory to zero.
     - Increment dcnt (directory count) to keep track of the number of directories created.
     - Display a message indicating the directory creation.
   - Case 2 (Create File):
     - Prompt the user to enter the directory name.
     - Read the directory name using scanf and store it in the variable d.
     - Search for the directory in the dir array.
     - If found, prompt the user to enter the name of the file to create within that directory.
     - Read the file name using scanf and store it in the appropriate directory's file list (dir[i].fname[dir[i].fcnt]).
     - Increment the file count (dir[i].fcnt) for that directory.

OUTPUT

```
1.Create Directory
2.Create File
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:1

Enter Name of Directory:dir1

Directory created
1.Create Directory
2.Create File
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:2

Enter Name of the Directory:dir1

Enter Name of the File to Create:file1

File created
1.Create Directory
2.Create File
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:2

Enter Name of the Directory:dir1

Enter Name of the File to Create:file2
```

        - Display a message indicating the file creation.
    - Case 3 (Delete File):
        - Prompt the user to enter the directory name.
        - Read the directory name using scanf and store it in the variable d.
        - Search for the directory in the dir array.
        - If found, prompt the user to enter the name of the file to delete within that directory.
        - Search for the file in the directory's file list.
        - If found, delete the file by shifting the elements in the array and decrementing the file count.
        - Display a message indicating the file deletion.
    - Case 4 (Search File):
        - Prompt the user to enter the directory name.
        - Read the directory name using scanf and store it in the variable d.
        - Search for the directory in the dir array.
        - If found, prompt the user to enter the name of the file to search within that directory.
        - Search for the file in the directory's file list.
        - Display a message indicating whether the file is found or not.
    - Case 5 (Display):
        - Display the list of directories and their respective files (if any).
        - If no directories exist, display a message indicating the absence of directories.
        - Iterate through each directory in the dir array and print its name along with the associated file names.
    - Default Case and Exit:
        - If the user selects an invalid option or chooses to exit, the program terminates using the exit(0) function.
6. End of Program

PROGRAM

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct
{
 char dname[10], fname[10][10];
 int fcnt;
```

```
File created
1.Create Directory
2.Create File
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:2

Enter Name of the Directory:dir1

Enter Name of the File to Create:file2

File created
1.Create Directory
2.Create File
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:2

Enter Name of the Directory:dir2

Directory dir2 Not Found!
1.Create Directory
2.Create File
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:1

Enter Name of Directory:dir2

Directory created
1.Create Directory
2.Create File
```

```
} dir[10];
void main()
{
 int i, ch, dcnt, k;
 char f[30], d[30];
 dcnt = 0;
 while (1)
 {
 printf("\n1.Create Directory\n2.Create File\n3.Delete File\n4.Search
File\n5.Display\n6.Exit\nEnter Your Choice:");
 scanf("%d", &ch);
 switch (ch)
 {
 case 1:
 printf("\nEnter Name of Directory:");
 scanf("%s", dir[dcnt].dname);
 dir[dcnt].fcnt = 0;
 dcnt++;
 printf("\nDirectory created");
 break;
 case 2:
 printf("\nEnter Name of the Directory:");
 scanf("%s", d);
 for (i = 0; i < dcnt; i++)
 if (strcmp(d, dir[i].dname) == 0)
 {
 printf("\nEnter Name of the File to Create:");
 scanf("%s", dir[i].fname[dir[i].fcnt]);
 dir[i].fcnt++;
 printf("\nFile created");
 break;
 }
 if (i == dcnt)
 printf("\nDirectory %s Not Found!", d);
 break;
 case 3:
 printf("\nEnter Name of the Directory:");
 scanf("%s", d);
```

```
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:2

Enter Name of the Directory:dir2

Enter Name of the File to Create:file3

File created
1.Create Directory
2.Create File
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:5

Directory       Files
dir1            file1     file2     file2
dir2            file3
1.Create Directory
2.Create File
3.Delete File
4.Search File
5.Display
6.Exit
Enter Your Choice:4

Enter Name of the Directory:dir1

Enter the Name of the File to Search:file2

File file2 Found
1.Create Directory
2.Create File
3.Delete File
```

```
for (i = 0; i < dcnt; i++)
{
if (strcmp(d, dir[i].dname) == 0)
{
printf("\nEnter Name of the File to Delete:");
scanf("%s", f);
for (k = 0; k < dir[i].fcnt; k++)
{
if (strcmp(f, dir[i].fname[k]) == 0)
{
printf("\nFile %s Deleted", f);
dir[i].fcnt--;
strcpy(dir[i].fname[k], dir[i].fname[dir[i].fcnt]);
goto jmp;
}
}
printf("\nFile %s Not Found!", f);
goto jmp;
}
}
printf("\nDirectory %s Not Found!", d);
jmp:
break;
case 4:
printf("\nEnter Name of the Directory:");
scanf("%s", d);
for (i = 0; i < dcnt; i++)
{
if (strcmp(d, dir[i].dname) == 0)
{
printf("\nEnter the Name of the File to Search:");
scanf("%s", f);
for (k = 0; k < dir[i].fcnt; k++)
{
if (strcmp(f, dir[i].fname[k]) == 0)
{
printf("\nFile %s Found", f);
goto jmp1;
```

```
4.Search File
5.Display
6.Exit
Enter Your Choice:6
```

```
 }
 }
 printf("\nFile %s Not Found!", f);
 goto jmp1;
 }
 }
 printf("\nDirectory %s Not Found!", d);
 jmp1:
 break;
 case 5:
 if (dcnt == 0)
 printf("\nNo Directories!");
 else
 {
 printf("\nDirectory\tFiles");
 for (i = 0; i < dcnt; i++)
 {
 printf("\n%s\t\t", dir[i].dname);
 for (k = 0; k < dir[i].fcnt; k++)
 printf("\t%s", dir[i].fname[k]);
 }
 }
 break;
 default:
 exit(0);
 }
 }
}
```

<u>RESULT</u>

Program to implement two level file organization technique has been executed successfully .

DATE :

EXPERIMENT NO – 1.3(iii)
FILE ORGANIZATION – HIERARCHICAL

AIM

To simulate the hierarchical file organization technique .

ALGORITHM

1. Start of program
2. **Structures Defined:**
    - **File**: Contains a name and content.
    - **Directory**: Contains a name, arrays for subdirectories and files, counters for the number of subdirectories and files.
3. **Function createDirectory:**
    - Check if the number of subdirectories in the parent directory has not reached the limit (**MAX_FILES**).
    - Allocate memory for a new directory.
    - Initialize the new directory's name, number of subdirectories, and number of files.
    - Add the new directory to the parent's subdirectory list and increment the count of subdirectories.
    - Print a message indicating successful directory creation or an error if the limit is reached.
4. **Function createFile:**
    - Check if the number of files in the parent directory has not reached the limit (**MAX_FILES**).
    - Create a new file structure with a given name and content.
    - Add the new file to the parent directory's file list and increment the count of files.
    - Print a message indicating successful file creation or an error if the limit is reached.
5. **Function listContents:**
    - Display the contents of a directory:
        - Print the directory's name.
        - Iterate through the subdirectories and print their names.
        - Iterate through the files and print their names.
6. **main Function:**
    - Create a root directory.

OUTPUT

```
Enter the name of the directory: dir1
Directory 'dir1' created.
Enter the name of a file: file1
Enter the content of the file: hello i am CSA
File 'file1' created.
Contents of directory 'root':
Subdirectories:
- dir1
Files:
- file1
```

- Prompt the user to input the name of a directory (**dirName**) and create it using **createDirectory**.
- Prompt the user to input the name of a file (**fileName**) and its content (**fileContent**), then create the file using **createFile**.
- Display the contents of the root directory using **listContents**.

7. **Input Validation and Display:**
    - The program takes user input for directory and file creation.
    - It checks for limitations on the number of subdirectories and files.
    - Displays the created directory and its contents including subdirectories and files.
8. **End of program**

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_NAME_LENGTH 50
#define MAX_FILES 100
typedef struct File {
char name[MAX_NAME_LENGTH];
char content[1024];
} File;
typedef struct Directory {
char name[MAX_NAME_LENGTH];
struct Directory* subdirectories[MAX_FILES];
File files[MAX_FILES];
int num_subdirectories;
int num_files;
} Directory;
void createDirectory(Directory* parent, const char* name) {
if (parent->num_subdirectories < MAX_FILES) {
Directory* newDirectory = (Directory*)malloc(sizeof(Directory));
strcpy(newDirectory->name, name);
newDirectory->num_subdirectories = 0;
newDirectory->num_files = 0;
parent->subdirectories[parent->num_subdirectories++] = newDirectory;
 printf("Directory '%s' created.\n", name);
 } else
printf("Directory limit reached. Cannot create '%s'.\n", name);
```

```
}
void createFile(Directory* parent, const char* name, const char* content) {
 if (parent->num_files < MAX_FILES) {
File newFile;
strcpy(newFile.name, name);
strcpy(newFile.content, content);
 parent->files[parent->num_files++] = newFile;
printf("File '%s' created.\n", name);
 } else
 printf("File limit reached. Cannot create '%s'.\n", name);
}
void listContents(Directory* dir) {
 printf("Contents of directory '%s':\n", dir->name);
 printf("Subdirectories:\n");
 for (int i = 0; i < dir->num_subdirectories; i++) {
 printf("- %s\n", dir->subdirectories[i]->name);
 }
 printf("Files:\n");
 for (int i = 0; i < dir->num_files; i++) {
printf("- %s\n", dir->files[i].name);
 }
}
int main() {
 Directory root;
 strcpy(root.name, "root");
 root.num_subdirectories = 0;
 root.num_files = 0;
 char dirName[MAX_NAME_LENGTH];
 printf("Enter the name of the directory: ");
 scanf("%s", dirName);
 createDirectory(&root, dirName);
 char fileName[MAX_NAME_LENGTH];
 printf("Enter the name of a file: ");
 scanf("%s", fileName);
 char fileContent[1024];
 printf("Enter the content of the file: ");
 scanf("%s", fileContent);
 createFile(&root, fileName, fileContent);
```

```
 listContents(&root);
 return 0;
}
```

RESULT

Program to implement hierarchical file organization technique has been executed successfully .

DATE :

EXPERIMENT NO – 1.4(i)
DISK SCHEDULING ALGORITHM - FCFS

AIM

To simulate First Come First Serve disk scheduling algorithm .

ALGORITHM

1. Start of program
2. **Variable Initialization:**
    - Declare variables: **diskQueue** (an array to hold disk queue elements), **n** (size of the queue), **i** (loop iterator), **seekTime** (total seek time), **diff** (difference between disk positions).
3. **Input Handling:**
    - Prompt the user to enter the size of the queue (**n**) using **printf**.
    - Read the size of the queue from the user using **scanf**.
    - Prompt the user to enter the queue elements using **printf**.
    - Read the queue elements into the **diskQueue** array using a loop and **scanf**.
    - Prompt the user to enter the initial head position using **printf**.
    - Read the initial head position into **diskQueue[0]** using **scanf**.
4. **Disk Movement Calculation:**
    - Initialize **seekTime** to zero.
    - Print a header indicating the movement of cylinders using **printf**.
    - Iterate through the queue elements to calculate disk movements and seek times:
        - Calculate the absolute difference between consecutive disk positions (**diskQueue[i+1]** and **diskQueue[i]**) and store it in **diff**.
        - Add **diff** to the **seekTime** variable to accumulate total seek time.
        - Print the movement from one disk position to another along with the seek time using **printf**.
5. **Display Results:**
    - Print the total seek time using **printf**.
    - Calculate and print the average seek time by dividing the total seek time by the number of elements (**n**) in the queue using **printf**.
6. **End of Program:**

PROGRAM

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

OUTPUT

```
Enter the size of Queue: 6
Enter the Queue: 45
67
12
34
11
89
Enter the initial head position: 40

Movement of Cylinders
Move from 40 to 45 with seek time 5
Move from 45 to 67 with seek time 22
Move from 67 to 12 with seek time 55
Move from 12 to 34 with seek time 22
Move from 34 to 11 with seek time 23
Move from 11 to 89 with seek time 78

Total Seek Time: 205
Average Seek Time = 34.166668
```

```
int main() {
   int diskQueue[20], n, i, seekTime=0, diff;
   printf("Enter the size of Queue: ");
   scanf("%d", &n);
   printf("Enter the Queue: ");
   for(i=1;i<=n;i++) {
          scanf("%d",&diskQueue[i]);
   }
   printf("Enter the initial head position: ");
   scanf("%d", &diskQueue[0]);
   printf("\nMovement of Cylinders\n");
   for(i=0;i<n;i++) {
      diff= abs(diskQueue[i+1] - diskQueue[i]);
      seekTime+= diff;
      printf("Move from %d to %d with seek time %d\n", diskQueue[i], diskQueue[i+1], diff);
   }
   printf("\nTotal Seek Time: %d", seekTime);
   printf("\nAverage Seek Time = %f",(float) seekTime/n);
   printf("\n");
   return 0;
}
```

RESULT

Program to implement First Come First Serve disk scheduling algorithm has been implemented successfully .

DATE :

EXPERIMENT NO – 1.4(ii)
DISK SCHEDULING ALGORITHM - SCAN

AIM

To simulate SCAN disk scheduling algorithm .

ALGORITHM

1. Start of program
2. **Function Definitions:**
    - **scan(int Ar[20], int n, int start)**: Performs the disk arm movement scan.
    - **sort(int Ar[20], int n)**: Sorts the disk queue.
3. **Main Function:**
    - Declare variables: **diskQueue[20]** (to store disk queue), **n** (size of the queue), **start** (initial head position), **i**.
    - Input the size of the queue (**n**) from the user using **scanf**.
    - Input the queue elements into **diskQueue** using a loop and **scanf**.
    - Input the initial head position (**start**) using **scanf**.
    - Increment the size of the queue by 1 and set **diskQueue[0]** to the initial head position.
    - Sort the **diskQueue** using the **sort** function.
    - Perform the scan using the **scan** function with **diskQueue**, **n**, and **start** as parameters.
4. **scan Function:**
    - Declare variables: **i**, **pos**, **diff**, **seekTime**, **current**.
    - Find the position of the initial head position in the queue (**start**).
    - Iterate from the initial head position to the end of the queue, calculating seek times and printing movement details.
    - Then, iterate from the position before the initial head position to the start of the queue, calculating seek times and printing movement details.
    - Calculate and print the total seek time and average seek time based on the movements.
5. **sort Function:**
    - Sorts the **Ar** array in ascending order using the Bubble Sort algorithm.
6. **End of Program:**
    - The program concludes after performing the scan and displaying the movement details, total seek time, and average seek time.

OUTPUT

```
Enter the size of Queue: 6
Enter the Queue: 34
98
12
43
11
10
Enter the initial head position: 35

Movement of Cylinders
Move from 35 to 43 with seek time 8
Move from 43 to 98 with seek time 55
Move from 98 to 34 with seek time 64
Move from 34 to 12 with seek time 22
Move from 12 to 11 with seek time 1
Move from 11 to 10 with seek time 1

Total Seek Time: 151
Average Seek Time = 25.166666
```

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
void scan(int Ar[20], int n, int start);
void sort(int Ar[20], int n);
int main() {
   int diskQueue[20], n, start, i;
   printf("Enter the size of Queue: ");
   scanf("%d", &n);
   printf("Enter the Queue: ");
   for(i=1;i<=n;i++) {
          scanf("%d",&diskQueue[i]);
   }
   printf("Enter the initial head position: ");
   scanf("%d", &start);
   diskQueue[0] = start;
   ++n;
   sort(diskQueue, n);
   scan(diskQueue, n, start);
   return 0;
}
void scan(int Ar[20], int n, int start) {
   int i, pos, diff, seekTime=0, current;
   for(i=0;i<n;i++) {
      if(Ar[i]==start) {
         pos=i;
         break;
      }
   }
   printf("\nMovement of Cylinders\n");
   for(i=pos;i<n-1;i++) {
      diff = abs(Ar[i+1] - Ar[i]);
      seekTime += diff;
      printf("Move from %d to %d with seek time %d\n", Ar[i], Ar[i+1], diff);
   }
   current=i;
   for(i=pos-1;i>=0;i--) {
      diff = abs(Ar[current] - Ar[i]);
```

```
        seekTime += diff;
        printf("Move from %d to %d with seek time %d\n", Ar[current], Ar[i], diff);
        current=i;
    }
    printf("\nTotal Seek Time: %d", seekTime);
    printf("\nAverage Seek Time = %f",(float) seekTime/(n-1));
    printf("\n");
}
void sort(int Ar[20], int n) {
    int i, j, tmp;
    for(i=0;i<n-1;i++) {
        for(j=0;j<n-1-i;j++) {
            if(Ar[j]>Ar[j+1]) {
                tmp = Ar[j];
                Ar[j] = Ar[j+1];
                Ar[j+1] = tmp;
            }
        }
    }
}
```

RESULT

Program to implement SCAN disk scheduling algorithm has been implemented successfully .

DATE :

EXPERIMENT NO – 1.4(iii)
DISK SCHEDULING ALGORITHM – C SCAN

AIM

To simulate C - SCAN disk scheduling algorithm .

ALGORITHM

1. Start of program
2. **Include Libraries**: Import necessary libraries like **<stdio.h>** and **<stdlib.h>** for standard input/output and other essential functions.
3. **Function Declarations**:
    - **void cscan(int Ar[20], int n, int start)**: Function to implement the C-SCAN algorithm.
    - **void sort(int Ar[20], int n)**: Function to sort the disk queue in ascending order.
4. **Main Function** (**main()**):
    - Declare variables: **diskQueue[20]** for the disk queue, **n** for the size of the queue, **start** for the initial head position, **i** as an iterator, and **max** for the maximum size.
    - Ask the user to input the size of the queue and the queue elements.
    - Input the initial head position and adjust the queue accordingly.
    - Increment the size of the queue by one (**++n**) and call the **sort()** function to sort the disk queue.
    - Finally, call the **cscan()** function to perform the C-SCAN algorithm on the disk queue.
5. **C-SCAN Function** (**cscan()**):
    - Initialize variables: **i** as an iterator, **pos** to store the position of the initial head, **diff** for the difference between disk positions, **seekTime** to calculate total seek time, and **current** to keep track of the current head position.
    - Find the position of the initial head in the sorted disk queue.
    - Implement the movement of the disk head to the right and then to the left using a loop.
    - Calculate the seek time for each movement and print the details of the movement.
    - Display the total seek time and average seek time.
6. **Sorting Function** (**sort()**):
    - Sort the disk queue in ascending order using a bubble sort algorithm.
7. End of program

OUTPUT

```
Enter the size of Queue: 8
Enter the Queue: 21
90
54
37
29
51
19
31
Enter the initial head position: 20

Movement of Cylinders
Move from 20 to 21 with seek time 1
Move from 21 to 29 with seek time 8
Move from 29 to 31 with seek time 2
Move from 31 to 37 with seek time 6
Move from 37 to 51 with seek time 14
Move from 51 to 54 with seek time 3
Move from 54 to 90 with seek time 36
Move from 90 to 0 with seek time 0
Move from 0 to 19 with seek time 19

Total Seek Time: 89
Average Seek Time = 11.125000
```

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
void cscan(int Ar[20], int n, int start);
void sort(int Ar[20], int n);
int main() {
   int diskQueue[20], n, start, i, max;
   printf("Enter the size of Queue: ");
   scanf("%d", &n);
   printf("Enter the Queue: ");
   for(i=1;i<=n;i++) {
          scanf("%d",&diskQueue[i]);
   }
   printf("Enter the initial head position: ");
   scanf("%d", &start);
   diskQueue[0] = start;
   ++n;
   sort(diskQueue, n);
   cscan(diskQueue, n, start);
   return 0;
}
void cscan(int Ar[20], int n, int start) {
   int i, pos, diff, seekTime=0, current;
   for(i=0;i<n;i++) {
      if(Ar[i]==start) {
         pos=i;
         break;
      }
   }
   printf("\nMovement of Cylinders\n");
   for(i=pos;i<n-1;i++) {
      diff = abs(Ar[i+1] - Ar[i]);
      seekTime+= diff;
      printf("Move from %d to %d with seek time %d\n", Ar[i], Ar[i+1], diff);
   }
   current=0;
   printf("Move from %d to %d with seek time %d\n", Ar[i], current, 0);
   for(i=0;i<pos;i++) {
```

```
        diff = abs(Ar[i] - current);
        seekTime+= diff;
        printf("Move from %d to %d with seek time %d\n", current, Ar[i], diff);
        current = Ar[i];
    }
    printf("\nTotal Seek Time: %d", seekTime);
    printf("\nAverage Seek Time = %f",(float) seekTime/(n-1));
    printf("\n");
}

void sort(int Ar[20], int n) {
    int i, j, tmp;
    for(i=0;i<n-1;i++) {
        for(j=0;j<n-1-i;j++) {
            if(Ar[j]>Ar[j+1]) {
                tmp = Ar[j];
                Ar[j] = Ar[j+1];
                Ar[j+1] = tmp;
            }
        }
    }
}
```

RESULT

Program to implement C – SCAN disk scheduling algorithm has been implemented successfully .

DATE :

EXPERIMENT NO – 1.5(i)
PAGE REPLACEMENT ALGORITHM - FIFO

AIM

To simulate the FIFO page replacement algorithm .

ALGORITHM

1. Start of program
2. **Include Library**: Import the **<stdio.h>** library for standard input/output functions.
3. **Main Function** (**main()**):
    - Declare variables: **i**, **j**, **n**, **a[50]**, **frame[10]**, **no**, **k**, **avail**, and **count**.
    - Ask the user to input the number of pages (**n**) and the page numbers.
    - Input the number of frames (**no**).
    - Initialize the **frame** array with -1 (indicating an empty frame).
    - Set **j** to 0 for frame indexing and print headers for reference string and page frames.
    - Loop through each page number and implement the FIFO page replacement algorithm:
        - Check if the page is already in a frame (**avail == 1**).
        - If the page is not in any frame (**avail == 0**), perform page replacement:
            - Place the page in the current frame (**frame[j] = a[i]**).
            - Increment **j** (frame pointer) using modulo operation to simulate a circular queue.
            - Increment the **count** variable to track page faults.
            - Display the current state of the frames after the page replacement.
4. Display the total number of page faults (**Page Fault Is %d**).
5. Finally, return 0 to indicate successful completion of the program
6. End of program

PROGRAM

```
#include<stdio.h>
int main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0;
        printf("\n ENTER THE NUMBER OF PAGES:\n");
scanf("%d",&n);
```

OUTPUT

```
 ENTER THE NUMBER OF PAGES:
8

 ENTER THE PAGE NUMBER :
7
3
1
5
7
7
1
0

 ENTER THE NUMBER OF FRAMES :3
ref string         page frames
7                  7       -1       -1
3                  7       3        -1
1                  7       3        1
5                  5       3        1
7                  5       7        1
7
1
0                  5       7        0
Page Fault Is 6
```

```
          printf("\n ENTER THE PAGE NUMBER :\n");
          for(i=1;i<=n;i++)
          scanf("%d",&a[i]);
          printf("\n ENTER THE NUMBER OF FRAMES :");
          scanf("%d",&no);
for(i=0;i<no;i++)
          frame[i]= -1;
                    j=0;
                    printf("\tref string\t page frames\n");
for(i=1;i<=n;i++)
                    {
                              printf("%d\t\t",a[i]);
                              avail=0;
                              for(k=0;k<no;k++)
if(frame[k]==a[i])
                                        avail=1;
                              if (avail==0)
                              {
                                        frame[j]=a[i];
                                        j=(j+1)%no;
                                        count++;
                                        for(k=0;k<no;k++)
                                        printf("%d\t",frame[k]);
}
                              printf("\n");
}
                    printf("Page Fault Is %d",count);
                    return 0;
}
```

RESULT

C program to simulate the FIFO page replacement algorithm has been implemented successfully .

DATE :

EXPERIMENT NO – 1.5(ii)
PAGE REPLACEMENT ALGORITHM - LRU

AIM

To simulate the LRU page replacement algorithm .

ALGORITHM

1. Start of program
2. **Include Library**: The program includes the **<stdio.h>** library for standard input/output functions.
3. **Function Declarations**:
    - **void LRU(int[], int[], int[], int, int)**: Function implementing the LRU algorithm.
    - **int findLRU(int[], int)**: Function to find the least recently used frame.
4. **Main Function** (**main()**):
    - Declare variables: **i**, **pCount**, **fCount**, **pages[30]**, **frames[20]**, and **time[20]**.
    - Ask the user to input the number of frames (**fCount**) and the number of pages (**pCount**).
    - Initialize the **frames** array with null values (-1).
    - Input the reference string (pages to be referenced).
5. **LRU Function** (**LRU()**):
    - Initialize variables: **i**, **j**, **k**, **pos**, **flag**, **faultCount**, **counter**, and **queue** to manage the algorithm logic.
    - Loop through each page in the reference string:
        - Check if the page is already in the frames. If found, update the time of occurrence and print "Hit".
        - If the frame is empty and there's space available, insert the page and update the occurrence time.
        - If all frames are occupied, find the least recently used frame using the **findLRU()** function and replace it with the current page.
        - Print the frames after each page reference.
        - Keep track of the number of page faults.
6. **LRU Helper Function** (**findLRU()**):
    - Find the least recently used frame by iterating through the time array and returning the position of the frame with the minimum time of occurrence.
7. Display the total number of page faults (**Total Page Faults = %d**).
8. End of program

OUTPUT

```
Number of Frames : 3
Number of Pages : 9
Enter the reference string
2
7
9
3
2
0
9
2
7
Ref.String   |        Frames
-------------------------------
 2       |         2        -1        -1

 7       |         2         7        -1

 9       |         2         7         9

 3       |         3         7         9

 2       |         3         2         9

 0       |         3         2         0

 9       |         9         2         0

 2       |           Hit

 7       |         9         2         7

Total Page Faults = 8
```

PROGRAM

```
#include <stdio.h>
void LRU(int[], int[], int[], int, int);
int findLRU(int[], int);
int main() {
   int i, pCount, fCount, pages[30], frames[20], time[20];
   printf("Number of Frames : ");
   scanf("%d", &fCount);
   for (i = 0; i < fCount; ++i) {
      frames[i] = -1;
   }
   printf("Number of Pages : ");
   scanf("%d", &pCount);
   printf("Enter the reference string\n");
   for (i = 0; i < pCount; ++i) {
      scanf("%d", &pages[i]);
   }
   LRU(pages, frames, time, fCount, pCount);
   return 0;
}
void LRU(int pages[], int frames[], int time[], int fCount, int pCount) {
   printf("\nRef.String   |\tFrames\n");
   printf("-------------------------------\n");
   int i, j, k, pos, flag, faultCount, counter, queue;
   counter = 0, queue = 0, faultCount = 0;
   for (i = 0; i < pCount; ++i) {
      flag = 0;
      printf("  %d\t|\t", pages[i]);
      for (j = 0; j < fCount; ++j) {
         if (frames[j] == pages[i]) {
            flag = 1;
            counter++;
            time[j] = counter;
            printf("   Hit\n\n");
            break;
         }
      }
      if ((flag == 0) && (queue < fCount)) {
```

```
            faultCount++;
            counter++;
            frames[queue] = pages[i];
            time[queue] = counter;
            queue++;
        }
        else if ((flag == 0) && (queue == fCount)) {
            faultCount++;
            counter++;
            pos = findLRU(time, fCount);
            frames[pos] = pages[i];
            time[pos] = counter;
        }
        if (flag == 0) {
            for (k = 0; k < fCount; ++k) {
                printf("%d  ", frames[k]);
            }
            printf("\n\n");
        }
    }
    printf("Total Page Faults = %d\n\n", faultCount);
}
int findLRU(int time[], int fCount) {
    int k, min, pos;
    pos = 0;
    min = time[0];
    for (k = 1; k < fCount; ++k) {
        if (time[k] < min) {
            min = time[k];
            pos = k;
        }
    }
    return pos;
}
```

RESULT

C program for the simulation of the LRU page replacement algorithm has been implemented successfully .

DATE :

EXPERIMENT NO – 1.5(iii)
PAGE REPLACEMENT ALGORITHM - LFU

AIM

To simulate the LFU page replacement algorithm .

ALGORITHM

1. Start of program
2. Initialize necessary variables: arrays **q**, **p**, **b**, and **c2**, integers **c**, **c1**, **d**, **f**, **i**, **j**, **k**, **n**, **r**, and **t**.
3. Take user input for the number of pages (**n**), the reference string (**p[]**), and the number of frames (**f**).
4. Initialize the first element of **q[]** with the first page number (**p[0]**), print it, increment the count of page faults (**c**), and increment the index **k**.
5. Loop through the reference string (**p[]**) starting from the second element (**i = 1**):
    - ➢ Initialize **c1** as 0.
    - ➢ Check if the current page in **p[]** is not present in **q[]**:
        - Increment **c1** for each page in **q[]** that does not match **p[i]**.

c. If **c1** equals the number of frames (**f**), indicating a page fault:

- ➢ Increment the count of page faults (**c**).
- ➢ If there are available empty frames (**k < f**):
    - Place the page (**p[i]**) into an empty frame in **q[]**, print the frames, and increment **k**.
- ➢ If all frames are occupied (**k >= f**):
    - Calculate the distance till the next occurrence of each page in **q[]** in the remaining reference string.
    - Determine the page in **q[]** that will not be used for the longest duration (**b[]** and **c2[]**).
    - Replace the identified page in **q[]** with the current page (**p[i]**), print the frames.

6. After looping through all pages, print the total number of page faults (**c**).
7. End of program

PROGRAM

```
#include <stdio.h>
void main()
{
  int q[20], p[50], c = 0, c1, d, f, i, j, k = 0, n, r, t, b[20], c2[20];
```

OUTPUT

```
Enter no of pages: 15
Enter the reference string: 7 0 1 2 0 3 0 4 2 3 0 0 3 3 7
Enter no of frames: 3

        7
        7       1
        7       1       2
        0       1       2
        0       3       2
        0       3       4
        0       2       4
        3       2       4
        3       2       0
        3       7       0
```

```
printf("Enter no of pages: ");
scanf("%d", &n);
printf("Enter the reference string: ");
for (i = 0; i < n; i++)
   scanf("%d", &p[i]);
printf("Enter no of frames: ");
scanf("%d", &f);
q[k] = p[k];
printf("\n\t%d\n", q[k]);
c++;
k++;
for (i = 1; i < n; i++)
{
   c1 = 0;
   for (j = 0; j < f; j++)
   {
      if (p[i] != q[j])
         c1++;
   }
   if (c1 == f)
   {
      c++;
      if (k < f)
      {
         q[k] = p[i];
         k++;
         for (j = 0; j < k; j++)
            printf("\t%d", q[j]);
         printf("\n");
      }
      else
      {
         for (r = 0; r < f; r++)
         {
            c2[r] = 0;
            for (j = i - 1; j < n; j--)
            {
               if (q[r] != p[j])
```

```
                    c2[r]++;
                  else
                    break;
                }
              }
              for (r = 0; r < f; r++)
                b[r] = c2[r];
              for (r = 0; r < f; r++)
              {
                for (j = r; j < f; j++)
                {
                  if (b[r] < b[j])
                  {
                    t = b[r];
                    b[r] = b[j];
                    b[j] = t;
                  }
                }
              }
              for (r = 0; r < f; r++)
              {
                if (c2[r] == b[0])
                  q[r] = p[i];
                printf("\t%d", q[r]);
              }
              printf("\n");
            }
          }
        }
        printf("\nThe no of page faults is %d", c);
      }
```

RESULT

C program to simulate the LFU page replacement algorithm has been implemented successfully
.