EXPERIMENTS FROM OPERATING SYSTEMS

PROGRAM CODE:

```
#include<stdio.h>
struct Process
{
int pid;
int at, bt, ct, wt, tt;
};
typedef struct Process pr;
void main()
{
int n, i, j, t;
float sumwt, sumtt;
sumwt=0;
sumtt=0;
printf("\nEnter the number of processes: ");
scanf("%d", &n);
pr p[n], temp;
printf("\nEnter the process id, arrival time and burst time of each process:\n");
for (i=0 ; i<n ; i++)
scanf("%d %d %d", &p[i].pid, &p[i].at, &p[i].bt);
for (i=0 ; i<n-1 ; i++)
{
for (j=i+1 ; j<n ; j++)
{
if (p[i].at > p[j].at)
{
temp = p[i];
p[i] = p[j];
p[j] = temp;
}
}

}
t = p[0].at;
for (i=0 ; i<n ; i++)
{
p[i].wt = t - p[i].at;
t = t + p[i].bt;
p[i].ct = t;
p[i].tt = p[i].ct - p[i].at;
sumwt += p[i].wt;
sumtt += p[i].tt;
}
for (i=0 ; i<n-1 ; i++)
{
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.1(i)
FCFS

AIM:

Write a program to implement FCFS cpu scheduling.

ALGORITHM:

Step 1: Declare the structure Process to store the process attributes (pid, at, bt, ct, wt, tt).
Step 2: Define the main function.
Step 3: Initialize variables:
- n to store the number of processes.
- i and j for loops.
- t for time tracking.
- sumwt and sumtt for total waiting time and turnaround time.

Step 4: Prompt the user to enter the number of processes and store the input in n.
Step 5: Declare an array p[n] of type Process to hold the process information.
Step 6: Prompt the user to input the process ID, arrival time, and burst time for each process and store the values in the respective attributes (pid, at, bt) of the p array.
Step 7: Sort the processes based on their arrival time (at):
- Iterate over the array p with two nested loops (i and j).
- Compare the arrival times of processes. If p[i].at > p[j].at, swap the processes.

Step 8: Set t = p[0].at to initialize time with the first process's arrival time.
Step 9: For each process in the array:
- Calculate the waiting time (wt) as the difference between the current time (t) and the process's arrival time (at).
- Increment the current time (t) by the burst time (bt) of the process.
- Calculate the completion time (ct) as the updated time (t).
- Calculate the turnaround time (tt) as the difference between the completion time (ct) and the arrival time (at).
- Accumulate the waiting time and turnaround time into sumwt and sumtt respectively.

Step 10: Sort the processes again based on their process ID (pid) to maintain order in the final output:
- Iterate over the array p with two nested loops (i and j).
- Compare the process IDs. If p[i].pid > p[j].pid, swap the processes.

Step 11: Print the table header showing the fields: Process ID, Arrival Time,Burst

```
for (j=i+1 ; j<n ; j++)
{
if (p[i].pid > p[j].pid)
{
temp = p[i];
p[i] = p[j];
p[j] = temp;
}
}
}
printf("\nProcess ID\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround Time\n");

for (i=0 ; i<n ; i++)
printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].wt, p[i].tt);

printf("\nAverage Waiting Time = %.3f", (sumwt/n));
printf("\nAverage Turnaroung Time = %.3f\n", (sumtt/n));
}
```

OUTPUT:

Enter the number of processes: 4

Enter the process id, arrival time and burst time of each process:

1 0 4

2 1 3

3 2 3

4 2 4

Process ID	Arrival Time	Burst Time	Completion Time	Waiting Time	Turnaround Time
1	0	4	4	0	4
2	1	3	7	3	6
3	2	3	10	5	8
4	2	4	14	8	12

Average Waiting Time = 4.000

Average Turnaround Time = 7.500

Time, Completion Time, Waiting Time, and Turnaround Time.
Step 12: Print the values of pid, at, bt, ct, wt, and tt for each process.
Step 13: Calculate and print the average waiting time as sumwt/n and the average turnaround time as sumtt/n.

RESULT:

Successfully completed fcfs cpu scheduling program.

PROGRAM CODE:

```
#include<stdio.h>
struct Process
{
int pid;
int at, bt, ct, wt, tt;
};
typedef struct Process pr;
void main()
{
int n, i, j, t, b, pos;
float sumwt, sumtt;
sumwt=0;
sumtt=0;
printf("\nEnter the number of processes: ");
scanf("%d", &n);
pr p[n], temp;
printf("\nEnter the process id, arrival time and burst time of each process:\n");
for (i=0 ; i<n ; i++)
scanf("%d %d %d", &p[i].pid, &p[i].at, &p[i].bt);
for (i=0 ; i<n-1 ; i++)
{
for (j=i+1 ; j<n ; j++)
{
if (p[i].at > p[j].at)
{
temp = p[i];
p[i] = p[j];
p[j] = temp;
}
}

}
for (i=0 ; i<n ; i++)
{
if (i != 0)
t = p[i-1].ct;
else
t = p[i].at;
b = p[i].bt;
for (j=i ; j<n ; j++)
{
if ((p[j].at <= t) && (p[j].bt <= b))
{
pos = j;
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.1(ii)
SJF

AIM:

Write a program to implement SJF cpu scheduling.

ALGORITHM:

Step 1: Declare a structure Process with the following fields:
- pid: Process ID.
- at: Arrival time.
- bt: Burst time.
- ct: Completion time.
- wt: Waiting time.
- tt: Turnaround time.

Step 2: Initialize integer variables n, i, j, t, b, and pos. Initialize float variables sumwt and sumtt to store the total waiting and turnaround times, respectively.

Step 3: Prompt the user to enter the number of processes (n):
- Display the message: "Enter the number of processes: ".
- Read the value of n.

Step 4: Declare an array p[n] of type Process to store the process details and a temporary variable temp to assist with swapping during sorting.

Step 5: For each process (from i = 0 to i < n):
- Display the message: "Enter the process id, arrival time and burst time of each process:".
- Read the process ID, arrival time, and burst time for each process.

Step 6: Sort the processes by their arrival time using a nested loop:
- For each process i from 0 to n-1, compare it with all subsequent processes j (from i+1 to n).
- If the arrival time of p[i] is greater than p[j], swap the two processes.

Step 7: Initialize the completion time (ct), waiting time (wt), and turnaround time (tt) for each process:
- For each process i (from 0 to n):
- If i = 0, set t to the arrival time of the first process (p[0].at).
- Otherwise, set t to the completion time of the previous process (p[i-1].ct).
- Set b to the burst time of the current process (p[i].bt).

Step 8: For each process j (from i to n):
- If the process p[j] has arrived (p[j].at <= t) and has the shortest burst time (p[j].bt <= b):

```
b = p[j].bt;
}
}
p[pos].ct = t + p[pos].bt;
p[pos].tt = p[pos].ct - p[pos].at;
p[pos].wt = p[pos].tt - p[pos].bt;
sumwt += p[pos].wt;
sumtt += p[pos].tt;
temp = p[pos];
p[pos] = p[i];
p[i] = temp;
}
for (i=0 ; i<n-1 ; i++)
{
for (j=i+1 ; j<n ; j++)
{
if (p[i].pid > p[j].pid)
{
temp = p[i];

p[i] = p[j];
p[j] = temp;
}
}
}
printf("\nProcess ID Arrival Time Burst Time Completion Time Waiting Time
Turnaround Time\n");

for (i=0 ; i<n ; i++)
printf("%5d%15d%14d%15d%15d%15d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].wt,
p[i].tt);

printf("\nAverage Waiting Time = %.3f", (sumwt/n));
printf("\nAverage Turnaroung Time = %.3f\n", (sumtt/n));
}
```

Update pos to j.
Set b to p[j].bt.

Step 9: Set the completion time (ct) for the process at pos:
p[pos].ct = t + p[pos].bt.

Step 10: Calculate the turnaround time (tt) and waiting time (wt) for the process at pos:

p[pos].tt = p[pos].ct - p[pos].at.
p[pos].wt = p[pos].tt - p[pos].bt.

Step 11: Add the waiting time and turnaround time of the process at pos to sumwt and sumtt.

Step 12: Swap the process at pos with the current process i to reflect its completion.

Step 13: Once all processes have been scheduled, sort them by their process ID using a nested loop:
For each process i from 0 to n-1, compare it with all subsequent processes j (from i+1 to n).
If the process ID of p[i] is greater than p[j], swap the two processes.

Step 14: Print the table headers: "Process ID Arrival Time Burst Time Completion Time Waiting Time Turnaround Time".

Step 15: For each process i (from 0 to n), print the process ID, arrival time, burst time, completion time, waiting time, and turnaround time.

Step 16: Calculate and print the average waiting time:
Average Waiting Time = sumwt / n.

Step 17: Calculate and print the average turnaround time:
Average Turnaround Time = sumtt / n.

OUTPUT:

Enter the number of processes: 4

Enter the process id, arrival time and burst time of each process:

1 0 5

2 1 5

3 1 3

4 3 6

Process ID	Arrival Time	Burst Time	Completion Time	Waiting Time	Turnaround Time
1	0	5	5	0	5
2	1	5	13	7	12
3	1	3	8	4	7
4	3	6	19	10	16

Average Waiting Time = 5.250

Average Turnaroung Time = 10.000

RESULT:

Successfully completed sjf cpu scheduling program.

PROGRAM CODE:

```
#include <stdio.h>

typedef struct{
   int pid;
   int at;
   int bt;
   int rm;
   int tat;
   int cmp;
   int wt;
   int inQ;

} Process;

int front=-1, rear=-1;


void printTable(int n, Process proc[n]){
   int avgT=0, avgW=0;
   printf("Process ID:\tArrival Time:\tBurst Time:\tCompletion Time:\tTurn Around
Time:\tWait Time:\n");
   for(int i=0; i<n; i++) {
      printf("%d\t\t%d\t\t%d\t\t%d\t\t\t%d\t\t\t%d\n", proc[i].pid, proc[i].at, proc[i].bt,
proc[i].cmp, proc[i].tat, proc[i].wt);
      avgT += proc[i].tat;
      avgW += proc[i].wt;
   }
   printf("Average Turn Around Time: %f\n", (float)avgT/n);
   printf("Average Waiting Time: %f\n", (float)avgW/n);
}

void sortProcess(int n, Process proc[n]){
   Process temp;
   for(int i=0; i<n-1; i++){
      for(int 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;
         }
      }
   }
}
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.1(iii)
ROUND ROBIN

AIM:

Write a program to implement RoundRobin cpu scheduling.

ALGORITHM:

Step 0: Start
Step 1: Initialize Data Structures:
Create an array of Process structs to represent the processes.
Initialize global variables front and rear to -1 to indicate an empty queue..
Step 2: Read Process Information:
Prompt the user to enter the number of processes n.
For each process, read its arrival time (at) and burst time (bt), and initialize its remaining time (rm) to its burst time. Also, assign a unique process ID (pid).
Step 3 Sort Processes by Arrival Time:
Sort the array of processes based on their arrival times using a sorting algorithm (e.g., bubble sort).
Step 4: Round Robin Execution:
Set the initial time to the arrival time of the first process.
Mark the first process as in the queue (inQ = 1) and enqueue it.
While not all processes have been completed:
If the queue is empty, increment the time and check for processes that have arrived and are not in the queue. Enqueue any found processes.
Dequeue a process from the queue. Calculate the execution time based on the remaining time and the time quantum. Update the process's remaining time accordingly.
Print the Gantt chart entry for the current process.
Check for processes that have arrived and are not in the queue, and enqueue them.
If the dequeued process's remaining time is less than or equal to zero, mark it as completed. Update its completion time, turn-around time, and waiting time. Increment the count of completed processes.
If the process still has work to do, mark it as in the queue and re-enqueue it.
Step 5 Print Results:
Print the completion time, turn-around time, and waiting time for each process.
Calculate and print the average turn-around time and average waiting time.
Step 6: Stop:

```
void enqueue(int n, Process q[1000], Process pr){
   if(front==-1) front=0;
   q[++rear] = pr;
}

Process dequeue(int n, Process q[1000]){
   if(front==rear){
      int k = front;
      front=-1;
      rear=-1;
      return q[k];
   }
   else{
      return q[front++];
   }

}

void roundRobin(int n, Process proc[n], int quanta){
   int completed=0, time=proc[0].at, wait;
   Process queue[1000];

   for(int i=0; i<n; i++) proc[i].inQ = 0;

   printf("\nGantt Chart: 0 ");

   enqueue(n, queue, proc[0]);
   proc[0].inQ = 1;
   while(completed<n){
      if(rear==-1 && front==-1){
         wait=1;
         time++;
         for(int i=0; i<n; i++){
            if(proc[i].at<=time && !proc[i].inQ){
               proc[i].inQ = 1;
               enqueue(n, queue, proc[i]);
            }
         }
      }
      else{
         Process pr = dequeue(n, queue);
         int exeTime = pr.rm<quanta ? pr.rm: quanta;
         pr.rm -= exeTime;
         if(wait){
           if(time!=0) printf("__ %d ", time);
```

```
            wait=0;
        }
        time += exeTime;
        printf("P%d %d ", pr.pid, time);

        //add to queue
        for(int i=0; i<n; i++){
            if(proc[i].at<=time && !proc[i].inQ){
                proc[i].inQ = 1;
                enqueue(n, queue, proc[i]);
            }
        }
        if(pr.rm <= 0){
            completed++;
            for (int i = 0; i < n; i++){
                if(proc[i].pid==pr.pid){
                    proc[i].cmp = time;
                    proc[i].tat = time - proc[i].at;
                    proc[i].wt = proc[i].tat - proc[i].bt;
                    break;
                }
            }
        }
        else{
            pr.inQ = 1;
            enqueue(n, queue, pr);
        }
      }
   }
   printf("\n");
}

int main(){
        int n, quanta;
        printf("Enter no. of process: ");
        scanf("%d", &n);
   Process proc[n];
        printf("Enter time quanta: ");
        scanf("%d", &quanta);

   printf("Enter (arrival time, burst time):\n");
   for(int i=0; i<n; i++){
      scanf("%d", &proc[i].at);
      scanf("%d", &proc[i].bt);
      proc[i].rm = proc[i].bt;
      proc[i].pid = i+1
```

```
    }

    sortProcess(n, proc);
    roundRobin(n, proc, quanta);
    printTable(n, proc);
}
```

OUTPUT:

Enter no. of process: 6

Enter time quanta: 5

Enter (arrival time, burst time):

0 7 1 4 2 15 3 11 4 20 4 9

Gantt Chart: 0 P1 5 P2 9 P3 14 P4 19 P5 24 P6 29 P1 31 P3 36 P4 12 P5 46 P6 50 P3 55 P4 56 P5 61 P5 66

Process ID:	Arrival Time:	Burst Time:	Completion Time:	Turn Around Time:	Wait Time:
1	0	7	31	31	24
2	1	4	9	84	
3	2	15	55	53	38
4	3	11	56	53	42
5	4	20	66	62	42
6	4	9	50	46	37

Average Turn Around Time: 42.166668

Average Waiting Time: 31.166666

RESULT:

Successfully completed roundrobin cpu scheduling program.

PROGRAM CODE:

```
#include<stdio.h>
struct Process
{
int pid;
int at, bt, ct, wt, tt, pri;
};
typedef struct Process pr;
void main()
{
int n, i, j, t, pos, prio;
float sumwt, sumtt;
sumwt=0;
sumtt=0;
printf("\nEnter the number of processes: ");
scanf("%d", &n);
pr p[n], temp;
printf("\nEnter the process id, arrival time, burst time and priority of each
process:\n");
for (i=0 ; i<n ; i++)
scanf("%d %d %d %d", &p[i].pid, &p[i].at, &p[i].bt, &p[i].pri);
for (i=0 ; i<n-1 ; i++)
{
for (j=i+1 ; j<n ; j++)
{
if (p[i].at > p[j].at)
{
temp = p[i];
p[i] = p[j];
p[j] = temp;
}
}

}
for (i=0 ; i<n ; i++)
{
if (i != 0)
t = p[i-1].ct;
else
t = p[i].at;
prio = p[i].pri;
for (j=i ; j<n ; j++)
{
if ((p[j].at <= t) && (p[j].pri == prio))
{
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.1(iv)
PRIORITY

AIM:

Write a program to implement priority scheduling.

ALGORITHM:

Step 1: Declare the structure Process to store process attributes (pid, at, bt, ct, wt, tt, pri).
Step 2: Define the main function.
Step 3: Initialize variables:
- n to store the number of processes.
- i, j, and pos for loops and position tracking.
- t for time tracking and prio to track the current priority.
- sumwt and sumtt to store the total waiting time and turnaround time.

Step 4: Prompt the user to enter the number of processes and store it in n.
Step 5: Declare an array p[n] of type Process to hold the process information.
Step 6: Prompt the user to input the process ID, arrival time, burst time, and priority for each process and store the values in pid, at, bt, and pri.
Step 7: Sort the processes based on their arrival time (at):
- Use two nested loops (i and j) to compare the arrival times of processes.
- Swap the processes if p[i].at > p[j].at.

Step 8: Start scheduling the processes. For each process p[i]:
- If i != 0, set t = p[i-1].ct (set time to the completion time of the previous process). If i == 0, set t = p[i].at (start time is the arrival time of the first process).
- Initialize prio to the priority of the current process p[i].

Step 9: Search for the process with the highest priority (smallest priority number) among the processes that have arrived (p[j].at <= t):
- If a process has the same priority as the current process and a shorter burst time, update pos to the index of that process.
- If a process has a higher priority (smaller value of pri), update pos to the index of that process and update prio.

Step 10: Update the selected process p[pos]:
- Set its completion time (ct) as t + p[pos].bt.
- Calculate the turnaround time (tt = ct - at).
- Calculate the waiting time (wt = tt - bt).
- Accumulate the waiting time and turnaround time into sumwt and sumtt.

```
if (p[i].bt > p[j].bt)
 pos = j;
 else
 pos = i;
 prio = p[j].pri;
 }
 else if ((p[j].at <= t) && (p[j].pri < prio))
 {
 pos = j;
 prio = p[j].pri;
 }
 }
 p[pos].ct = t + p[pos].bt;
 p[pos].tt = p[pos].ct - p[pos].at;
 p[pos].wt = p[pos].tt - p[pos].bt;
 sumwt += p[pos].wt;
 sumtt += p[pos].tt;
 temp = p[pos];
 p[pos] = p[i];
 p[i] = temp;

 }
 for (i=0 ; i<n-1 ; i++)
 {
 for (j=i+1 ; j<n ; j++)
 {
 if (p[i].pid > p[j].pid)
 {
 temp = p[i];
 p[i] = p[j];
 p[j] = temp;
 }
 }
 }
 printf("\nProcess ID Arrival Time Burst Time Completion Time Waiting Time
 Turnaround Time Priority\n");

 for (i=0 ; i<n ; i++)
 printf("%5d%15d%14d%15d%15d%15d%14d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct,
 p[i].wt, p[i].tt, p[i].pri);

 printf("\nAverage Waiting Time = %.3f", (sumwt/n));
 printf("\nAverage Turnaroung Time = %.3f\n", (sumtt/n));
 }
```

Step 11: Swap the selected process p[pos] with the process p[i] to ensure that the scheduled process is placed in the correct position.
Step 12: Repeat steps 8-11 for all processes until all processes are scheduled.
Step 13: Sort the processes based on their process IDs (pid) for final output:
    Use two nested loops (i and j) to compare the process IDs.
    Swap the processes if p[i].pid > p[j].pid.
Step 14: Print the table header showing the fields: Process ID, Arrival Time, Burst Time, Completion Time, Waiting Time, Turnaround Time, and Priority.
Step 15: Print the values of pid, at, bt, ct, wt, tt, and pri for each process.
Step 16: Calculate and print the average waiting time as sumwt/n and the average turnaround time as sumtt/n.

OUTPUT:

Enter the number of processes: 3

Enter the process id, arrival time, burst time and priority of each process:
1 0 4 1
2 1 5 3
3 1 3 2

Process ID	Arrival Time	Burst Time	Completion Time	Waiting Time	Turnaround Time	Priority
1	0	4	4	0	4	1
2	1	5	12	6	11	3
3	1	3	7	3	6	2

Average Waiting Time = 3.000
Average Turnaroung Time = 7.000

RESULT:

Successfully completed priority scheduling program.

PROGRAM CODE:

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

struct FileAllocation {
    char file_name[20];
    int start_block;
    int length;
};

int main() {
    int num_files;
    printf("Enter the number of files to be allocated: ");
    scanf("%d", &num_files);

    struct FileAllocation files[num_files];

    for (int i = 0; i < num_files; i++) {
        printf("Enter the name of the file %d: ", i + 1);
        scanf("%s", files[i].file_name);

        printf("Enter the start block of the file %d: ", i + 1);
        scanf("%d", &files[i].start_block);

        printf("Enter the length of the file %d: ", i + 1);
        scanf("%d", &files[i].length);
    }

    printf("\nFile Allocation Table\n");
    printf("%-10s %-12s %-10s\n", "File Name", "Start Block", "Length");

    for (int i = 0; i < num_files; i++) {
        printf("%-10s %-12d %-10d\n", files[i].file_name, files[i].start_block,
files[i].length);
    }

    return 0;
}
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.2(i)
SEQUENTIAL

AIM:

Write a program to implement sequential file allocation.

ALGORITHM:

Step 0: Start
Step 1: Declare a structure FileAllocation with three fields:
    file_name (to store the name of the file)
    start_block (to store the start block of the file)
    length (to store the length of the file in terms of blocks)
Step 2: Declare an integer num_files to store the number of files to be allocated.
Step 3: Prompt the user to enter the number of files to be allocated:
    Display the message: "Enter the number of files to be allocated: "
    Read the value into num_files.
Step 4: Declare an array of FileAllocation structures with size num_files to store the details of all the files.

Step 5: Initialize a loop that runs num_files times (from i = 0 to i < num_files) to input the file details.
    For each iteration:
        Display the message: "Enter the name of the file (i + 1): ".
        Read the file name into files[i].file_name.
        Display the message: "Enter the start block of the file (i + 1): ".
        Read the start block value into files[i].start_block.
        Display the message: "Enter the length of the file (i + 1): ".
        Read the length value into files[i].length.
Step 6: After collecting all the file details, print the heading:
    File Allocation Table
    Print the column names: "File Name", "Start Block", "Length"
Step 7: Initialize another loop that runs num_files times (from i = 0 to i < num_files) to print the file details.
    For each iteration:
        Print the values stored in files[i].file_name, files[i].start_block, and files[i].length.
Step 8: End

OUTPUT:

```
Enter the number of files to be allocated: 3
Enter the name of the file 1: file1
Enter the start block of the file 1: 2
Enter the length of the file 1: 8
Enter the name of the file 2: file2
Enter the start block of the file 2: 11
Enter the length of the file 2: 7
Enter the name of the file 3: file3
Enter the start block of the file 3: 20
Enter the length of the file 3: 4

File Allocation Table
File Name  Start Block  Length
file1          2          8
file2          11         7
file3          20         4
```

RESULT:

Successfully completed sequential file allocation program.

PROGRAM CODE:

```
#include <stdio.h>

struct File {
    int start_address;
    int length;
    int num_blocks;
    int blocks[10];  // Assuming a maximum of 10 blocks per file
};

int main() {
    int num_files;
    printf("Enter the number of files to be added: ");
    scanf("%d", &num_files);

    struct File files[num_files];

    for (int i = 0; i < num_files; i++) {
        printf("Enter the start address of file[%d]: ", i);
        scanf("%d", &files[i].start_address);

        printf("Enter the length of the file[%d]: ", i);
        scanf("%d", &files[i].length);

        printf("Enter the number of blocks of file[%d]: ", i);
        scanf("%d", &files[i].num_blocks);

        printf("Enter the blocks allocated: ");
        for (int j = 0; j < files[i].num_blocks; j++) {
            scanf("%d", &files[i].blocks[j]);
        }
    }

    // Display file details
    printf("\nFile Details:\n");
    printf("%-10s %-12s %-12s %-12s\n", "File no", "no of blocks", "length", "start
addr");

    for (int i = 0; i < num_files; i++) {
        printf("%-10d %-12d %-12d %-12d\n", i, files[i].num_blocks, files[i].length,
files[i].start_address);
    }

    int choice, file_number;
    do {
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.2(ii)
INDEXED

AIM:

Write a program to implement indexed file allocation.

ALGORITHM:

Step 0: Start
Step 1: Define a structure File with the following fields:
start_address: to store the starting address of the file.
length: to store the length of the file.
num_blocks: to store the number of blocks allocated to the file.
blocks[]: an array to store the block numbers allocated to the file (with a maximum size of 10).
Step 2: Declare an integer variable num_files to hold the number of files to be added.
Step 3: Prompt the user to enter the number of files to be added:
Display the message: "Enter the number of files to be added: "
Read the value into num_files.
Step 4: Declare an array files of type File with size num_files to store the details of each file.
Step 5: Initialize a loop that runs num_files times (from i = 0 to i < num_files) to input the file details.
For each iteration:
Display the message: "Enter the start address of file[i]: "
Read the start address into files[i].start_address.
Display the message: "Enter the length of the file[i]: "
Read the length into files[i].length.
Display the message: "Enter the number of blocks of file[i]: "
Read the number of blocks into files[i].num_blocks.
Display the message: "Enter the blocks allocated: "
Initialize another loop that runs files[i].num_blocks times (from j = 0 to j < files[i].num_blocks) to read the block numbers.
For each iteration, read the block number into files[i].blocks[j].
Step 6: After collecting all file details, print the heading:
"File Details:"
Print the column names: "File no", "no of blocks", "length", "start addr".
Step 7: Initialize another loop that runs num_files times (from i = 0 to i < num_files) to print the file details.

```
        printf("\n1. Get the blocks\n2. Exit\n");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the file number: ");
                scanf("%d", &file_number);

                if (file_number < num_files) {
                    printf("Blocks allocated: ");
                    for (int j = 0; j < files[file_number].num_blocks; j++) {
                        printf("%d ", files[file_number].blocks[j]);
                    }
                    printf("\n");
                } else {
                    printf("Invalid file number!\n");
                }
                break;

            case 2:
                printf("Exiting...\n");
                break;

            default:
                printf("Invalid choice! Please select 1 or 2.\n");
                break;
        }
    } while (choice != 2);

    return 0;
}
```

OUTPUT:

```
Enter the number of files to be added: 3
Enter the start address of file[0]: 5
Enter the length of the file[0]: 7
Enter the number of blocks of file[0]: 4
Enter the blocks allocated: 3 4 5 6
Enter the start address of file[1]: 12
Enter the length of the file[1]: 6
Enter the number of blocks of file[1]: 3
Enter the blocks allocated: 7 8 9
Enter the start address of file[2]: 22
Enter the length of the file[2]: 3
Enter the number of blocks of file[2]: 2
```

For each iteration, print the file number, number of blocks, length, and start address from files[i].

Step 8: Declare an integer variable choice to hold the user's menu choice and an integer variable file_number to hold the selected file number.

Step 9: Initialize a do-while loop that continues until the user chooses to exit (i.e., while choice != 2):

Display the menu options:

1. Get the blocks
2. Exit

Read the user’s choice into choice.

Step 10: Use a switch statement to handle the user's choice:

Case 1:

Display the message: "Enter the file number: ".

Read the file number into file_number.

If file_number is valid (i.e., file_number < num_files):

Print "Blocks allocated: ".

Initialize a loop that runs files[file_number].num_blocks times to print the blocks allocated.

For each iteration, print the block number from files[file_number].blocks[j].

Print a newline after displaying the blocks.

Else, print "Invalid file number!".

Case 2:

Print "Exiting...".

Default Case:

Print "Invalid choice! Please select 1 or 2.".

Step 11: End the loop when the user chooses to exit (i.e., choice == 2).

Step 12: Return 0 to indicate successful termination of the program.

Enter the blocks allocated: 10 11

File Details:

File no	no of blocks	length	start addr
0	4	7	5
1	3	6	12
2	2	3	22

1. Get the blocks
2. Exit
1
Enter the file number: 1
Blocks allocated: 7 8 9

1. Get the blocks
2. Exit
2
Exiting...

RESULT:

Successfully completed indexed file allocation program.

PROGRAM CODE:

```
#include <stdio.h>
#define MAX_BLOCKS 100

int blocks[MAX_BLOCKS];

void allocateFile(int start, int num_blocks) {
    int count = 0;
    int block_num = start;

    while (count < num_blocks) {
        if (blocks[block_num] == 0) {
            blocks[block_num] = 1;
            printf("bno: %d ---->1\n", block_num);
            count++;
        }
        block_num++;
    }
}

int main() {
    int num_allocated, allocated_blocks[10], num_blocks, start_block, choice, i;

    for (i = 0; i < MAX_BLOCKS; i++) {
        blocks[i] = 0;
    }

    printf("Enter the number of blocks which are already allocated: ");
    scanf("%d", &num_allocated);

    printf("Enter the blocks which are already allocated: ");
    for (i = 0; i < num_allocated; i++) {
        scanf("%d", &allocated_blocks[i]);
        blocks[allocated_blocks[i]] = 1;
    }

    do {
        printf("\n1. Add file\n2. Exit\n");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the no of blocks to be allocated and starting block: ");
                scanf("%d %d", &num_blocks, &start_block);
```

Name: Anavadya N Lakshmi
Roll No: 10
Class: CS5A

Date:

EXPERIMENT 1.2(iii)
LINKED

AIM:

Write a program to implement linked file allocation.

ALGORITHM:

Step 1: Initialize blocks[] to 0 (all free).

Step 2: Input the number of pre-allocated blocks and mark them as allocated.

Step 3: Display menu options (Add file / Exit).

Step 4: Get user choice.

Step 5: If the user selects Add file, proceed to the next step. Otherwise, go to step 10.

Step 6: Input the number of blocks to allocate and the starting block.

Step 7: Allocate blocks starting from the given block number.

Step 8: After allocation, return to the main menu and repeat from Step 3.

Step 9: Handle invalid inputs by displaying an error message and going back to Step 4.

Step 10: If the user selects Exit, terminate the program.

```
            allocateFile(start_block, num_blocks);
            break;

        case 2:
            printf("Exiting...\n");
            break;

        default:
            printf("Invalid choice! Please enter 1 or 2.\n");
            break;
      }
    } while (choice != 2);

    return 0;
}
```

OUTPUT:

Enter the number of blocks which are already allocated: 3

Enter the blocks which are already allocated:

2 5 8

1. Add file

2. Exit

1

Enter the no of blocks to be allocated and starting block: 3 4

bno: 4 ---->1

bno: 6 ---->1

bno: 7 ---->1

1. Add file

2. Exit

1

Enter the no of blocks to be allocated and starting block: 2 9

bno: 9 ---->1

bno: 10 ---->1

1. Add file

2. Exit

2

Exiting...

RESULT:

Successfully completed linked file allocation program.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>
void main() {
 int RQ[100],i,n,TotalHeadMovement=0,initial;
 printf("Enter the no of requests: ");
 scanf("%d",&n);
 printf("Enter the requests sequence: ");
 for(i=0;i<n;i++) {
 scanf("%d",&RQ[i]);
 }
 printf("Enter initial head position: ");
 scanf("%d",&initial);

 for(i=0;i<n;i++) {
 TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
 initial=RQ[i];
 }

 printf("Total head moment is %d",TotalHeadMovement);
}
```

OUTPUT:

Enter the no of requests: 8

Enter the requests sequence: 98 183 37 122 14 124 65 67

Enter initial head position: 53

Total head movement is 612

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.3(i)
FCFS

AIM:

Write a program to implement FCFS disk scheduling algorithm.

ALGORITHM:

Step 0: Start
Step 1: Initialize Variables:
n: Number of disk requests.
RQ[]: Sequence of disk requests
TotalHeadMovement to 0, which will store the total distance the disk head moves.
initial to store the initial position of the disk head.
Step 2: Input the Number of Requests and the Sequence of Requests:
Read the number of disk requests n.
Read the sequence of disk requests RQ[].
Step 3: Input the Initial Head Position:
Read the initial position of the disk head initial.
Step 4: Process Each Disk Request:
For each request RQ[i] in the sequence:
• Calculate the absolute distance the disk head moves from its current position to the request position.
• Add this distance to TotalHeadMovement.
• Update the disk head's current position to the request position.
Step 5: Output the Total Head Movement:
Print the total head movement.
Step 6: Stop.

RESULT:

Successfully completed fcfs disk scheduling algorithm.

PROGRAM CODE:

```
include<stdio.h>
#include<stdlib.h>
void main() {
 int RQ[100],i,j,n,TotalHeadMovement=0,initial,size,move,temp,index;
 printf("Enter the no of requests: ");
 scanf("%d",&n);
 printf("Enter the requests sequence: ");
 for(i=0;i<n;i++) {
 scanf("%d",&RQ[i]);
 }
 printf("Enter initial head position: ");
 scanf("%d",&initial);
 printf("Enter total disk size: ");
 scanf("%d",&size);
 printf("Enter the head movement direction for high 1 and low 0: ");
 scanf("%d",&move);

 //Sort the request array
 for(i=0;i<n;i++) {
 for(j=0;j<n-i-1;j++) {
 if(RQ[j]>RQ[j+1]) {
 temp=RQ[j];
 RQ[j]=RQ[j+1];
 RQ[j+1]=temp;
 }
 } }

 for(i=0;i<n;i++) {
 if(initial<RQ[i]) {
 index=i;
 break;
 }
 }

 //if movement is towards high value
 if(move==1) {
 for(i=index;i<n;i++) {
 TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
 initial=RQ[i];
 }
 //last movement for max size
 TotalHeadMovement=TotalHeadMovement+abs(size-RQ[i-1]-1);
 initial=size-1;
 for(i=index-1;i>=0;i--) {
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT1.3(ii)
SCAN

AIM:

Write a program to implement SCAN disk scheduling algorithm.

ALGORITHM:

Step 0: Start
Step 1: Initialize Variables:
n: Number of disk requests.
RQ[]: Sequence of disk requests.
TotalHeadMovement to 0.
initial to store the initial position of the disk head.
size to store the total size of the disk.
move to indicate the direction of head movement (1 for high, 0 for low).
Step 2: Input the Number of Requests and the Sequence of Requests:
Read the number of disk requests n.
Read the sequence of disk requests RQ[].
Step 3: Input the Initial Head Position, Disk Size, and Head Movement Direction:
Read the initial position of the disk head initial.
Read the total disk size.
Read the head movement direction move.
Step 4: Sort the Request Array:
Sort the array RQ[] in ascending order.
Step 5: Find the Index of the First Request Greater Than the Initial Position:
Determine the index where the first request greater than the initial position is located in RQ[].
Step 6: Service Requests Based on Head Movement Direction:
If move is towards high values (1):
• Service requests from index to n-1.
• Move the head to the end of the disk (size - 1).
• Service the remaining requests from index-1 to 0.
If move is towards low values (0):
• Service requests from index-1 to 0.
• Move the head to the start of the disk (0).
• Service the remaining requests from index to n-1.
Step 7: Output the Total Head Movement:
Print the total head movement.
Step 8: Stop

```
 TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
  initial=RQ[i];
  }
  }

 //if movement is towards low value
  else{
  for(i=index-1;i>=0;i--) {
  TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
  initial=RQ[i];
  }
  //last movement for min size
  TotalHeadMovement=TotalHeadMovement+abs(RQ[i+1]-0);
  initial=0;
  for(i=index;i<n;i++) {
  TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
  initial=RQ[i];
  }
  }

  printf("Total head movement is %d",TotalHeadMovement);
 }
```

OUTPUT:

Enter the no of requests: 8

Enter the requests sequence: 98 183 37 122 14 124 65 67

Enter initial head position: 53

Enter total disk size: 200

Enter the head movement direction for high 1 and low 0: 0

Total head movement is 236

RESULT:

Successfully completed scan disk scheduling algorithm.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>
int main() {
 int RQ[100],i,j,n,TotalHeadMovement=0,initial,size,move,index,temp;
 printf("Enter the number of Requests: ");
 scanf("%d",&n);
 printf("Enter the Requests sequence: ");
 for(i=0;i<n;i++) {
 scanf("%d",&RQ[i]);
 }
 printf("Enter initial head position: ");
 scanf("%d",&initial);
 printf("Enter total disk size: ");
 scanf("%d",&size);
 printf("Enter the head movement direction for high 1 and for low 0: ");
 scanf("%d",&move);
 /* logic for C-Scan disk scheduling */
 /*logic for sort the request array */
 for(i=0;i<n;i++) {
 for( j=0;j<n-i-1;j++) {
 if(RQ[j]>RQ[j+1]) {
 temp=RQ[j];
 RQ[j]=RQ[j+1];
 RQ[j+1]=temp;
 } } }

 for(i=0;i<n;i++) {
 if(initial<RQ[i]) {
 index=i;
 break;
 } }

 /* if movement is towards high value */
 if(move==1) {
 for(i=index;i<n;i++) {
 TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
 initial=RQ[i];
 }
/* last movement for max size*/
 TotalHeadMovement=TotalHeadMovement+abs(size-RQ[i-1]-1);
 /*movement max to min disk */
 TotalHeadMovement=TotalHeadMovement+abs(size-1-0);
 initial=0;
 for( i=0;i<index;i++) {
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.3(iii)
C-SCAN

AIM:

Write a program to implement C-SCAN disk scheduling algorithm.

ALGORITHM:

Step 0: Start
Step 1: Initialize Variables:
n: Number of disk requests.
RQ[]: Sequence of disk requests.
TotalHeadMovement to 0.
initial to store the initial position of the disk head.
size to store the total size of the disk.
move to indicate the direction of head movement (1 for high, 0 for low).
Step 2: Input the Number of Requests and the Sequence of Requests:
Read the number of disk requests n.
Read the sequence of disk requests RQ[].
Step 3: Input the Initial Head Position, Disk Size, and Head Movement Direction:
Read the initial position of the disk head initial.
Read the total disk size.
Read the head movement direction move.
Step 4: Sort the Request Array:
Sort the array RQ[] in ascending order using bubble sort.
Step 5: Find the Index of the First Request Greater Than the Initial Position:
Determine the index where the first request greater than the initial position is located in RQ[].
Step 6: Service Requests Based on Head Movement Direction:
If move is towards high values (1):
• Service requests from index to n-1.
• Move the head to the end of the disk (size - 1).
• Jump to the beginning of the disk (0).
• Service the remaining requests from 0 to index-1.
If move is towards low values (0):
• Service requests from index-1 to 0.
• Move the head to the beginning of the disk (0).
• Jump to the end of the disk (size - 1).
• Service the remaining requests from n-1 to index

```
TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
 initial=RQ[i];
 }
 }

 /* if movement is towards low value */
 else {
 for(i=index-1;i>=0;i--) {
 TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
 initial=RQ[i];
 }
 /* last movement for min size */
 TotalHeadMovement=TotalHeadMovement+abs(RQ[i+1]-0);
 /*movement min to max disk */
 TotalHeadMovement=TotalHeadMovement+abs(size-1-0);
 initial =size-1;
 for(i=n-1;i>=index;i--) {
 TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
 initial=RQ[i];
 }
 }
 printf("Total head movement is %d",TotalHeadMovement);
 return 0;
}
```

OUTPUT:

```
Enter the number of Requests: 8
Enter the Requests sequence: 98 183 37 122 14 124 65 67
Enter initial head position: 53
Enter total disk size: 200
Enter the head movement direction for high 1 and for low 0: 1
Total head movement is 382
```

Step 7: Output the Total Head Movement: Print the total head movement.
Step 8: Stop

RESULT:

Successfully completed CSCAN disk scheduling algorithm.

PROGRAM CODE:

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

#define MAX_FRAMES 3
#define MAX_PAGES 20

int frames[MAX_FRAMES];
int rear = -1;

void initialize() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        frames[i] = -1;
    }
}
void displayFrames() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        if (frames[i] != -1)
            printf("%d ", frames[i]);
        else
            printf("- ");
    }
    printf("\n");
}

void FIFO(int pages[], int n) {
    int page_faults = 0;
    int front = 0;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        int found = 0;


        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == page) {
                found = 1;
                printf("page %d already there\n",page);
                break;
            }
        }

        if (!found) {
            page_faults++;
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.4(i)
FIFO

AIM:

Write a program to implement FIFO page replacement algorithms.

ALGORITHM:

Step 1: Start.
Step 2: Initialize Variables:
Step 3: Set the number of frames (frames) and the incoming Stream array of page requests.
Initialize page Faults to 0.
Step 4: Initialize Frame Array:
Create an array temp of size frames and set all elements to -1 (empty).
Step 5: Process Each Page in the Incoming Stream:
For each page in incoming Stream:
Check if the page is already in the frames (temp array):
If found, it is a page hit.
If not found, it is a page fault:
Increment page Faults.
If there are empty frames, place the page in the next available frame.
If all frames are full, replace the oldest page using the FIFO strategy.
Step 6: Print the Current Frame Status:
After processing each page, print the current status of frames and the total number of page faults.
Step 7: Stop

```
            if (rear < MAX_FRAMES - 1) {
                rear++;
            } else {
                rear = 0;
            }
            frames[rear] = page;
            printf("page %d loaded in frame %d \n",page,rear);
        }

        displayFrames();
    }

    printf("Total Page Faults: %d\n", page_faults);
}
int main() {
    int pages[MAX_PAGES];
    int n;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter the page reference sequence: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }
    initialize();
    FIFO(pages, n);
    return 0;
}
```

OUTPUT:

Enter the length of reference string: 20

Enter the reference string: 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6

Enter no of frames: 4

1 -1 -1 -1 PF No. 1

1 2 -1 -1 PF No. 2

1 2 3 -1 PF No. 3

1 2 3 4 PF No. 4

1 2 3 4

1 2 3 4

5 2 3 4 PF No. 5

5 6 3 4 PF No. 6

5 6 2 4 PF No. 7

5 6 2 1 PF No. 8

5 6 2 1

3 6 2 1 PF No. 9

3 7 2 1 PF No. 10

3 7 6 1 PF No. 11

3 7 6 1

3 7 6 2 PF No. 12

1 7 6 2 PF No. 13

1 7 6 2

1 3 6 2 PF No. 14

1 3 6 2

The number of page faults using FIFO are 14

RESULT:

Successfully completed fifo page replacement algorithm.

PROGRAM CODE:

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

#define MAX_FRAMES 3
#define MAX_PAGES 20

int frames[MAX_FRAMES];
int counter[MAX_FRAMES
void initialize() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        frames[i] = -1; // Initializing frames with -1 (empty)
        counter[i] = 0; // Initializing counter with 0
    }
}
void displayFrames() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        if (frames[i] != -1)
            printf("%d ", frames[i]);
        else
            printf("- ");
    }
    printf("\n");
}
int findLRU() {
    int max = counter[0];
    int lru_frame = 0;

    for (int i = 1; i < MAX_FRAMES; i++) {
        if (counter[i] > max) {
            max = counter[i];
            lru_frame = i;
        }
    }
    return lru_frame;
}
void LRU(int pages[], int n) {
    int page_faults = 0;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        int found = 0;
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == page) {
                found = 1;
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.4(ii)

LRU

AIM:

Write a program to implement LRU page replacement algorithm.

ALGORITHM:

Step 1: Start.
Step 2: Initialize Variables:
Set number of frames (no_of_frames).
Set number of pages (no_of_pages).
Create arrays frames (initialized to -1), pages, and time.
Initialize counter and faults to 0.
Step 3: Input Data:
Read no_of_frames and no_of_pages. Read the reference string into pages.
Step 4: Process Pages:
For each page in pages:
Check for Page Hit: If the page is in frames, update its access time.
Handle Page Fault:
If there is an empty frame, place the page there, update access time, and increment faults.
If no empty frame, find the least recently used page using find LRU, replace it, update access time, and increment faults.
Step 5: Print Frame Status:
Print the contents of frames after each page request.
Step 6: Print Total Page Faults:
Print the total number of page faults.
Step 7: Stop.

```
            printf("page %d is already there \n",page);
            break;
        }
    }

    if (!found) {
        int lru_frame = findLRU();
        printf("page %d is loaded in frame %d\n",page,lru_frame);
        frames[lru_frame] = page;
        counter[lru_frame] = 0;
        page_faults++;
    }
    for (int j = 0; j < MAX_FRAMES; j++) {
        counter[j]++;
    }
    for (int j = 0; j < MAX_FRAMES; j++) {
        if (frames[j] == page) {
            counter[j] = 0;
            break;
        }
    }
   displayFrames();
  }
  printf("Total Page Faults: %d\n", page_faults);
}

int main() {
  int pages[MAX_PAGES];
  int n;
  printf("Enter number of pages: ");
  scanf("%d", &n);
  printf("Enter the page reference sequence: ");
  for (int i = 0; i < n; i++) {
     scanf("%d", &pages[i]);
  }

  initialize();
  LRU(pages, n);

  return 0;
}
```

OUTPUT:

Enter number of pages: 7

Enter the page reference sequence: 1 3 0 3 5 6 3

page 1 is loaded in frame 0

1 - -

page 3 is loaded in frame 1

1 3 -

page 0 is loaded in frame 2

1 3 0

page is already there 3

1 3 0

page 5 is loaded in frame 0

5 3 0

page 6 is loaded in frame 2

5 3 6

page is already there 3

5 3 6

Total Page Faults: 5

RESULT:

Successfully completed lru page replacement algorithm.

PROGRAM CODE:

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

#define MAX_FRAMES 3
#define MAX_PAGES 20

int frames[MAX_FRAMES];
int counts[MAX_FRAMES];

void initialize() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        frames[i] = -1;
        counts[i] = 0;
    }
}
void displayFrames() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        if (frames[i] != -1)
            printf("%d ", frames[i]);
        else
            printf("- ");
    }
    printf("\n");
}
int findLFU() {
    int min = counts[0];
    int lfu_frame = 0;

    for (int i = 1; i < MAX_FRAMES; i++) {
        if (counts[i] < min) {
            min = counts[i];
            lfu_frame = i;
        }
    }

    return lfu_frame;
}
void LFU(int pages[], int n) {
    int page_faults = 0;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        int found = 0;
```

Name: Anavadya N Lakshmi Date:
Roll No: 10
Class: CS5A

EXPERIMENT 1.4(iii)

LFU

AIM:

Write a program to implement LFU page replacement algorithm.

ALGORITHM:

Step 1: Start.
Step 2: Initialize Variables:
Create arrays rs[] for the reference string, cntr[] for access frequencies, and a[] for frames.
Initialize page fault counter pf to 0.
Step 3: Input Data:
Read the number of page references (m).
Read the reference string into rs[].
Read the number of frames (f).
Step 4: Initialize Frames and Counters:
Set all elements of a[] to -1 (indicating empty frames).
Set all elements of cntr[] to 0 (indicating no access).
Step 5: Process Each Page in the Reference String:
For each page in rs[]:
Check for Page Hit:
If the page is in frames (a[]), increment its counter (cntr[]).
Handle Page Fault:
If the page is not in frames:
Find the frame with the least access frequency.
Replace the least frequently used page.
Reset its counter and increment the page fault counter (pf).
Print the current state of frames and page faults.
Step 6: Print Total Page Faults:
Print the total number of page faults (pf).
Step 7: Stop.

```
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == page) {
                found = 1;
                counts[j]++;
                printf("page %d is already in memory\n",page);
                break;
            }
        }

        if (!found) {
            int lfu_frame = findLFU();
            frames[lfu_frame] = page;
            counts[lfu_frame] = 1;
            page_faults++;
            printf("page %d is loaded in frame %d\n",page,lfu_frame);
        }

        //printf("Page %d : ", page);
        displayFrames();
    }

    printf("Total Page Faults: %d\n", page_faults);
}

int main() {
    int pages[MAX_PAGES];
    int n;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter the page reference sequence: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }

    initialize();
    LFU(pages, n);

    return 0;
}
```

OUTPUT:

Enter number of pages: 7

Enter the page reference sequence: 1 3 0 3 5 6 3

page 1 is loaded in frame 0

1 - -

page 3 is loaded in frame 1

1 3 -

page 0 is loaded in frame 2

1 3 0

page 3 is already in memory

1 3 0

page 5 is loaded in frame 0

5 3 0

page 6 is loaded in frame 0

6 3 0

page 3 is already in memory

6 3 0

Total Page Faults: 5

RESULT:

Successfully completed lfu page replacement algorithm.