**Submitted by: Submitted To:**

Vanshika Mrs. Navneet Kaur

B.Tech CSE

4th SEM

1900295

**Operating System Lab**

**(ACCS – 16406)**

**1. Implement a C/C++ program to simulate the CPU scheduling algorithms to find turnaround time and waiting time of FCFS.**

#include<iostream>

using namespace std;

// Function to find the waiting time for all

// processes

void findWaitingTime(int processes[], int n,

int bt[], int wt[])

{

// waiting time for first process is 0

wt[0] = 0;

// calculating waiting time

for (int i = 1; i < n ; i++ )

wt[i] = bt[i-1] + wt[i-1] ;

}

// Function to calculate turn around time

void findTurnAroundTime( int processes[], int n,

int bt[], int wt[], int tat[])

{

// calculating turnaround time by adding

// bt[i] + wt[i]

for (int i = 0; i < n ; i++)

tat[i] = bt[i] + wt[i];

}

//Function to calculate average time

void findavgTime( int processes[], int n, int bt[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

//Function to find waiting time of all processes

findWaitingTime(processes, n, bt, wt);

//Function to find turn around time for all processes

findTurnAroundTime(processes, n, bt, wt, tat);

//Display processes along with all details

cout << "Processes "<< " Burst time "

<< " Waiting time " << " Turn around time\n";

// Calculate total waiting time and total turn

// around time

for (int i=0; i<n; i++)

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

// Driver code

int main()

{

//process id's

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

//Burst time of all processes

int burst\_time[] = {10, 5, 8};

findavgTime(processes, n, burst\_time);

return 0;

}

Processes Burst time Waiting time Turn around time

1 10 0 10

2 5 10 15

3 8 15 23

Average waiting time = 8.33333

Average turn around time = 16

**2. Implement a C/C++ program to simulate the CPU scheduling algorithms to find turnaround time and waiting time of SJF preemptive.**

// C++ program to implement Shortest Remaining Time First

// Shortest Remaining Time First (SRTF)  
#include <bits/stdc++.h>

using namespace std;

struct Process

{

int pid; // Process ID

int bt; // Burst Time

int art; // Arrival Time

};

// Function to find the waiting time for all

// processes

void findWaitingTime(Process proc[], int n,

int wt[])

{

int rt[n];

// Copy the burst time into rt[]

for (int i = 0; i < n; i++)

rt[i] = proc[i].bt;

int complete = 0, t = 0, minm = INT\_MAX;

int shortest = 0, finish\_time;

bool check = false;

// Process until all processes gets

// completed

while (complete != n)

{

// Find process with minimum

// remaining time among the

// processes that arrives till the

// current time`

for (int j = 0; j < n; j++)

{

if ((proc[j].art <= t) &&

(rt[j] < minm) && rt[j] > 0)

{

minm = rt[j];

shortest = j;

check = true;

}

}

if (check == false) {

t++;

continue;

}

// Reduce remaining time by one

rt[shortest]--;

// Update minimum

minm = rt[shortest];

if (minm == 0)

minm = INT\_MAX;

// If a process gets completely

// executed

if (rt[shortest] == 0)

{

// Increment complete

complete++;

check = false;

// Find finish time of current

// process

finish\_time = t + 1;

// Calculate waiting time

wt[shortest] = finish\_time -

proc[shortest].bt -

proc[shortest].art;

if (wt[shortest] < 0)

wt[shortest] = 0;

}

// Increment time

t++;

}

// Function to calculate turn around time

void findTurnAroundTime(Process proc[], int n, int wt[], int tat[])

{

// calculating turnaround time by adding

// bt[i] + wt[i]

for (int i = 0; i < n; i++)

tat[i] = proc[i].bt + wt[i];

}

// Function to calculate average time

void findavgTime(Process proc[], int n)

{

int wt[n], tat[n], total\_wt = 0,

total\_tat = 0;

// Function to find waiting time of all

// processes

findWaitingTime(proc, n, wt);

// Function to find turn around time for

// all processes

findTurnAroundTime(proc, n, wt, tat);

// Display processes along with all

// details

cout << "Processes "

<< " Burst time "

<< " Waiting time "

<< " Turn around time\n";

// Calculate total waiting time and

// total turnaround time

for (int i = 0; i < n; i++) {

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << proc[i].pid << "\t\t"

<< proc[i].bt << "\t\t " << wt[i]

<< "\t\t " << tat[i] << endl;

}

cout << "\nAverage waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

// Driver code

int main()

{

Process proc[] = { { 1, 6, 1 }, { 2, 8, 1 },

{ 3, 7, 2 }, { 4, 3, 3 } };

int n = sizeof(proc) / sizeof(proc[0]);

findavgTime(proc, n);

return 0;

}

Processes Burst time Waiting time Turn around time

1 6 3 9

2 8 16 24

3 7 8 15

4 3 0 3

Average waiting time = 6.75

Average turn around time = 12.75

**3. Implement a C/C++ program to simulate the CPU scheduling algorithms to find turnaround time and waiting time of SJF non preemptive.**

// C++ program to implement Shortest Job first with Arrival Time

#include<iostream>

using namespace std;

int mat[10][6];

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void arrangeArrival(int num, int mat[][6]);

{

for(int i=0; i<num; i++)

{

for(int j=0; j<num-i-1; j++)

{

if(mat[j][1] > mat[j+1][1])

{

for(int k=0; k<5; k++)

{

swap(mat[j][k], mat[j+1][k]);

}

}

}

}

}

void completionTime(int num, int mat[][6])

{

int temp, val;

mat[0][3] = mat[0][1] + mat[0][2];

mat[0][5] = mat[0][3] - mat[0][1];

mat[0][4] = mat[0][5] - mat[0][2];

for(int i=1; i<num; i++)

{

temp = mat[i-1][3];

int low = mat[i][2];

for(int j=i; j<num; j++)

{

{

low = mat[j][2];

val = j;

}

}

}

mat[val][3] = temp + mat[val][2];

mat[val][5] = mat[val][3] - mat[val][1];

mat[val][4] = mat[val][5] - mat[val][2];

for(int k=0; k<6; k++)

{

swap(mat[val][k], mat[i][k]);

}

}

int main()

{

int num, temp;

cout<<"Enter number of Process: ";

cin>>num;

cout<<"...Enter the process ID...\n";

for(int i=0; i<num; i++)

{

cout<<"...Process "<<i+1<<"...\n";

cout<<"Enter Process Id: ";

cin>>mat[i][0];

cout<<"Enter Arrival Time: ";

cin>>mat[i][1];

cout<<"Enter Burst Time: ";

cin>>mat[i][2];

}

cout<<"Before Arrange...\n";

cout<<"Process ID\tArrival Time\tBurst Time\n";

for(int i=0; i<num; i++)

{

cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\n";

}

arrangeArrival(num, mat);

completionTime(num, mat);

cout<<"Final Result...\n";

cout<<"Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround

Time\n";

for(int i=0; i<num; i++)

{

cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\t\t"<<mat[i][4]<<

"\t\t"<<mat[i][5]<<"\n";

}

}

Process ID Arrival Time Burst Time

1 2 3

2 0 4

3 4 2

4 5 4

Final Result...

Process ID Arrival Time Burst Time Waiting Time Turnaround Time

2 0 4 0 4

3 4 2 0 2

1 2 3 4 7

4 5 4 4 8

**4. Implement a C/C++ program to simulate the CPU Scheduling algorithms to find turnaround time and waiting time of Round Robin.**

// C++ program for implementation of RR scheduling

#include<iostream>

using namespace std;

// Function to find the waiting time for all

// processes

void findWaitingTime(int processes[], int n,

int bt[], int wt[], int quantum)

{

// Make a copy of burst times bt[] to store remaining

// burst times.

int rem\_bt[n];

for (int i = 0 ; i < n ; i++)

rem\_bt[i] = bt[i];

int t = 0; // Current time

// Keep traversing processes in round robin manner

/ until all of them are not done.

while (1)

{

bool done = true;

// Traverse all processes one by one repeatedly

for (int i = 0 ; i < n; i++)

{

// If burst time of a process is greater than 0

// then only need to process further

if (rem\_bt[i] > 0)

{

done = false; // There is a pending process

if (rem\_bt[i] > quantum)

{

// Increase the value of t i.e. shows

// how much time a process has been processed

t += quantum;

// Decrease the burst\_time of current process

// by quantum

rem\_bt[i] -= quantum;

}

// If burst time is smaller than or equal to

// quantum. Last cycle for this process

else

{

// Increase the value of t i.e. shows

// how much time a process has been processed

t = t + rem\_bt[i];

// Waiting time is current time minus time

// used by this process

wt[i] = t - bt[i];

// As the process gets fully executed

// make its remaining burst time = 0

rem\_bt[i] = 0;

}

}

}

// If all processes are done

if (done == true)

break;

}

}

// Function to calculate turn around time

void findTurnAroundTime(int processes[], int n,

int bt[], int wt[], int tat[])

{

// calculating turnaround time by adding

// bt[i] + wt[i]

for (int i = 0; i < n ; i++)

tat[i] = bt[i] + wt[i];

}

// Function to calculate average time

void findavgTime(int processes[], int n, int bt[],

int quantum)

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

// Function to find waiting time of all processes

findWaitingTime(processes, n, bt, wt, quantum);

// Function to find turn around time for all processes

findTurnAroundTime(processes, n, bt, wt, tat);

// Display processes along with all details

cout << "Processes "<< " Burst time "

<< " Waiting time " << " Turn around time\n";

// Calculate total waiting time and total turn

// around time

for (int i=0; i<n; i++)

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

// Driver code

int main()

{

// process id's

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

// Burst time of all processes

int burst\_time[] = {10, 5, 8};

// Time quantum

int quantum = 2;

findavgTime(processes, n, burst\_time, quantum);

return 0;

}

Processes Burst time Waiting time Turn around time

1 10 13 23

2 5 10 15

3 8 13 21

Average waiting time = 12

Average turn around time = 19.6667

**5. Implement a C/C++ program to simulate the CPU Scheduling algorithms to find turnaround time and waiting time of Priority preemptive.**

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <string.h>

using namespace std;

struct process

{

int pid;

int arrival\_time;

int burst\_time;

int priority;

int start\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int response\_time;

};

int main()

{

int n;

struct process p[100];

float avg\_turnaround\_time;

float avg\_waiting\_time;

float avg\_response\_time;

float cpu\_utilisation;

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

int total\_response\_time = 0;

int total\_idle\_time = 0;

float throughput;

int burst\_remaining[100];

int is\_completed[100];

memset(is\_completed,0,sizeof(is\_completed));

cout << setprecision(2) << fixed;

cout<<"Enter the number of processes: ";

cin>>n;

for(int i = 0; i < n; i++) {

cout<<"Enter arrival time of process "<<i+1<<": ";

cin>>p[i].arrival\_time;

cout<<"Enter burst time of process "<<i+1<<": ";

cin>>p[i].burst\_time;

cout<<"Enter priority of the process "<<i+1<<": ";

cin>>p[i].priority;

p[i].pid = i+1;

burst\_remaining[i] = p[i].burst\_time;

cout<<endl;

}

int current\_time = 0;

int completed = 0;

int prev = 0;

while(completed != n)

{

int idx = -1;

int mx = -1;

for(int i = 0; i < n; i++)

{

if(p[i].arrival\_time <= current\_time && is\_completed[i] == 0)

{

if(p[i].priority > mx)

{

mx = p[i].priority;

idx = i;

}

if(p[i].priority == mx)

{

if(p[i].arrival\_time < p[idx].arrival\_time)

{

mx = p[i].priority;

idx = i;

}

}

}

}

if(idx != -1)

{

if(burst\_remaining[idx] == p[idx].burst\_time)

{

p[idx].start\_time = current\_time;

total\_idle\_time += p[idx].start\_time - prev;

}

burst\_remaining[idx] -= 1;

current\_time++;

prev = current\_time;

if(burst\_remaining[idx] == 0) {

p[idx].completion\_time = current\_time;

p[idx].turnaround\_time = p[idx].completion\_time -

p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time -

p[idx].burst\_time;

p[idx].response\_time = p[idx].start\_time -

p[idx].arrival\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

total\_waiting\_time += p[idx].waiting\_time;

total\_response\_time += p[idx].response\_time;

is\_completed[idx] = 1;

completed++;

}

}

else

{

current\_time++;

}

}

int min\_arrival\_time = 10000000;

int max\_completion\_time = -1;

for(int i = 0; i < n; i++)

{

min\_arrival\_time = min(min\_arrival\_time,p[i].arrival\_time);

max\_completion\_time = max (max\_ completion\_ time,p[i] .completion\_ time);

}

avg\_turnaround\_time = (float) total\_turnaround\_time / n;

avg\_waiting\_time = (float) total\_waiting\_time / n;

avg\_response\_time = (float) total\_response\_time / n;

cpu\_utilisation = ((max\_completion\_time - total\_idle\_time) / (float)

max\_completion\_time )\*100;

throughput = float(n) / (max\_completion\_time - min\_arrival\_time);

cout<<endl<<endl;

cout<<"#P\t"<<"AT\t"<<"BT\t"<<"PRI\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<R

T\t"<<"\n"<<endl;

for(int i = 0; i < n; i++)

{cout<<p[i].pid<<"\t"<<p[i].arrival\_time<<"\t"<<p[i].burst\_time<<"\t"<<p[i]

.priority<<"\t"<<p[i].start\_time<<"\t"<<p[i].completion\_time<<"\t"<<p[i].t rnaround\_time<<"\t"<<p[i].waiting\_time<<"\t"<<p[i].response\_time<<"\t"<<"

\n"<<endl;}

cout<<"Average Turnaround Time = "<<avg\_turnaround\_time<<endl;

cout<<"Average Waiting Time = "<<avg\_waiting\_time<<endl;

cout<<"Average Response Time = "<<avg\_response\_time<<endl;

cout<<"CPU Utilization = "<<cpu\_utilisation<<"%"<<endl;

cout<<"Throughput = "<<throughput<<" process/unit time"<<endl;

}

Enter the number of processes: 5

Enter arrival time of process 1: 12

Enter burst time of process 1: 4

Enter priority of the process 1: 1

Enter arrival time of process 2: 5

Enter burst time of process 2: 1

Enter priority of the process 2: 2

Enter arrival time of process 3: 13

Enter burst time of process 3: 2

Enter priority of the process 3: 2

Enter arrival time of process 4: 3

Enter burst time of process 4: 4

Enter priority of the process 4: 2

Enter arrival time of process 5: 1

Enter burst time of process 5: 23

Enter priority of the process 5: 4

#P AT BT PRI ST CT TAT WT RT

1 12 4 1 31 35 23 19 19

2 5 1 2 28 29 24 23 23

3 13 2 2 29 31 18 16 16

4 3 4 2 24 28 25 21 21

5 1 23 4 1 24 23 0 0

Average Turnaround Time = 22.60

Average Waiting Time = 15.80

Average Response Time = 15.80

CPU Utilization = 97.14%

Throughput = 0.15 process/unit time

**6.Implement a C/C++ program to simulate the CPU scheduling algorithms to find turnaround time and waiting time of Priority non preemptive.**

#include<iostream>

using namespace std;

int main()

{

int a[10],b[10],x[10],pr[10]={0};

int waiting[10],turnaround[10],completion[10];

int i,j,smallest,count=0,time,n;

double avg=0,tt=0,end;

cout<<"\nEnter the number of Processes: ";

cin>>n;

for(i=0;i<n;i++)

{

cout<<"\nEnter arrival time of process: ";

cin>>a[i];

}

for(i=0;i<n;i++)

{

cout<<"\nEnter burst time of process: ";

cin>>b[i];

}

for(i=0;i<n;i++)

{

cout<<"\nEnter priority of process: ";

cin>>pr[i];

}

for(i=0;i<n;i++)

x[i]=b[i];

pr[9]=-1;

for(time=0;count!=n;time++)

{

smallest=9;

for(i=0;i<n;i++)

{

if(a[i]<=time && pr[i]>pr[smallest] && b[i]>0 )

smallest=i;

}

time+=b[smallest]-1;

b[smallest]=-1;

count++;

end=time+1;

completion[smallest] = end;

waiting[smallest] = end - a[smallest] - x[smallest];

turnaround[smallest] = end - a[smallest];

}

cout<<"Process"<<"\t"<< "burst-time"<<"\t"<<"arrival-time"<< "\t" <<" waiting-time"<<"\t"<<"turnaround-time"<<"\t"<<"completion-time" <<" \t "<<"Priority"<<endl;

for(i=0;i<n;i++)

{

cout<<"p"<<i+1<<"\t\t"<<x[i]<<"\t\t"<<a[i]<<"\t\t"<<waiting[i]<<"\t\t"<<turnaround[i]<<"\t\t"<<completion[i]<<"\t\t"<<pr[i]<<endl;

avg = avg + waiting[i];

tt = tt + turnaround[i];

}cout<<"\n\nAverage waiting time ="<<avg/n;

cout<<" AverageTurnaround time ="<<tt/n<<endl;}

Enter the number of Processes: 5

Enter arrival time of process: 0

Enter arrival time of process: 1

Enter arrival time of process: 2

Enter arrival time of process: 3

Enter arrival time of process: 4

Enter burst time of process: 4

Enter burst time of process: 3

Enter burst time of process: 1

Enter burst time of process: 5

Enter burst time of process: 2

Enter priority of process: 2

Enter priority of process: 3

Enter priority of process: 4

Enter priority of process: 5

Enter priority of process: 5

Process burst-time arrival-time waiting-time turnaround-time completion-time Priority

p1 4 0 0 4 4 2

p2 3 1 11 14 15 3

p3 1 2 9 10 12 4

p4 5 3 1 6 9 5

p5 2 4 5 7 11 5

Average waiting time =5.2 Average Turnaround ti me =8.2

Process returned 0 (0x0) execution time : 32.070 s

Press any key to continue.

**7. Implement a C/C++ program to simulate page replacement algorithm of FIFO.**

// C++ implementation of FIFO page replacement

// in Operating Systems.

#include<bits/stdc++.h>

using namespace std;

// Function to find page faults using FIFO

int pageFaults(int pages[], int n, int capacity)

{

// To represent set of current pages. We use

// an unordered\_set so that we quickly check

// if a page is present in set or not

unordered\_set<int> s;

// To store the pages in FIFO manner

queue<int> indexes;

// Start from initial page

int page\_faults = 0;

for (int i=0; i<n; i++)

{

// Check if the set can hold more pages

if (s.size() < capacity)

{

// Insert it into set if not present

// already which represents page fault

if (s.find(pages[i])==s.end())

{

// Insert the current page into the set

s.insert(pages[i]);

// increment page fault

page\_faults++;

// Push the current page into the queue

indexes.push(pages[i]);

}

}

// If the set is full then need to perform FIFO

// i.e. remove the first page of the queue from

// set and queue both and insert the current page

else

{

// Check if current page is not already

// present in the set

if (s.find(pages[i]) == s.end())

{

// Store the first page in the

// queue to be used to find and

// erase the page from the set

int val = indexes.front();

// Pop the first page from the queue

indexes.pop();

// Remove the indexes page from the set

s.erase(val);

// insert the current page in the set

s.insert(pages[i]);

// push the current page into

// the queue

indexes.push(pages[i]);

// Increment page faults

page\_faults++;

}

}

}

return page\_faults;

}

// Driver code

{

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,

2, 3, 0, 3, 2};

int n = sizeof(pages)/sizeof(pages[0]);

int capacity = 4;

cout << pageFaults(pages, n, capacity);

return 0;

}

7

**8. Implement a C/C++ program to simulate page replacement algorithm of Optimal.**

// CPP program to demonstrate optimal page

// replacement algorithm.

#include <bits/stdc++.h>

using namespace std;

// Function to check whether a page exists

// in a frame or not

bool search(int key, vector<int>& fr)

{

for (int i = 0; i < fr.size(); i++)

if (fr[i] == key)

return true;

return false;

}

// Function to find the frame that will not be used

// recently in future after given index in pg[0..pn-1]

int predict(int pg[], vector<int>& fr, int pn, int index)

{

// Store the index of pages which are going

// to be used recently in future

int res = -1, farthest = index;

for (int i = 0; i < fr.size(); i++)

{

int j;

for (j = index; j < pn; j++)

{

if (fr[i] == pg[j])

{

if (j > farthest)

{

farthest = j;

res = i;

}

break;

}

}

// If a page is never referenced in future,

// return it.

if (j == pn)

return i;

}

// If all of the frames were not in future,

// return any of them, we return 0. Otherwise

// we return res.

return (res == -1) ? 0 : res;

}

void optimalPage(int pg[], int pn, int fn)

{

// Create an array for given number of

// frames and initialize it as empty.

vector<int> fr;

// Traverse through page reference array

// and check for miss and hit.

int hit = 0;

for (int i = 0; i < pn; i++)

{

// Page found in a frame : HIT

if (search(pg[i], fr))

{

hit++;

continue;

}

// Page not found in a frame : MISS

// If there is space available in frames.

if (fr.size() < fn)

fr.push\_back(pg[i]);

// Find the page to be replaced.

else

{

int j = predict(pg, fr, pn, i + 1);

fr[j] = pg[i];

}

}

cout << "No. of hits = " << hit << endl;

cout << "No. of misses = " << pn - hit << endl;

}

// Driver Function

int main()

{

int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };

int pn = sizeof(pg) / sizeof(pg[0]);

int fn = 4;

optimalPage(pg, pn, fn);

return 0;

}

No. of hits = 7

No. of misses = 6

**9. Implement a C/C++ program to simulate page replacement algorithm of LRU.**

//C++ implementation of above algorithm

#include<bits/stdc++.h>

using namespace std;

// Function to find page faults using indexes

int pageFaults(int pages[], int n, int capacity)

{

// To represent set of current pages. We use

// an unordered\_set so that we quickly check

// if a page is present in set or not

unordered\_set<int> s;

// To store least recently used indexes

// of pages.

unordered\_map<int, int> indexes;

// Start from initial page

int page\_faults = 0;

for (int i=0; i<n; i++)

{

// Check if the set can hold more pages

if (s.size() < capacity)

{

// Insert it into set if not present

// already which represents page fault

if (s.find(pages[i])==s.end())

{

s.insert(pages[i]);

// increment page fault

page\_faults++;

}

// Store the recently used index of

// each page

indexes[pages[i]] = i;

}

// If the set is full then need to perform lru

// i.e. remove the least recently used page

// and insert the current page

else

{

// Check if current page is not already

// present in the set

if (s.find(pages[i]) == s.end())

{

// Find the least recently used pages

// that is present in the set

int lru = INT\_MAX, val;

for (auto it=s.begin(); it!=s.end(); it++)

{

if (indexes[\*it] < lru)

{

lru = indexes[\*it];

val = \*it;

}

}

// Remove the indexes page

s.erase(val);

// insert the current page

s.insert(pages[i]);

// Increment page faults

page\_faults++;

}

// Update the current page index

indexes[pages[i]] = i;

}

return page\_faults;

}

// Driver code

int main()

{

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int n = sizeof(pages)/sizeof(pages[0]);

int capacity = 4;

cout << pageFaults(pages, n, capacity);

return 0;

}

6