

DISK SCHEDULING ALGORITHM

Experiment No: 7

Date: / /23

Aim: a) To implement First Come First Serve(FCFS) Disk Scheduling algorithm.

Theory:

The FCFS disk scheduling algorithm is one of the simplest disk scheduling algorithms. It operates on the principle of serving I/O requests in the order they arrive in the request queue. In other words, the request that arrives first is the one that gets serviced first.

Advantages of FCFS

Here are some of the advantages of First Come First Serve.

Every request gets a fair chance

No indefinite postponement

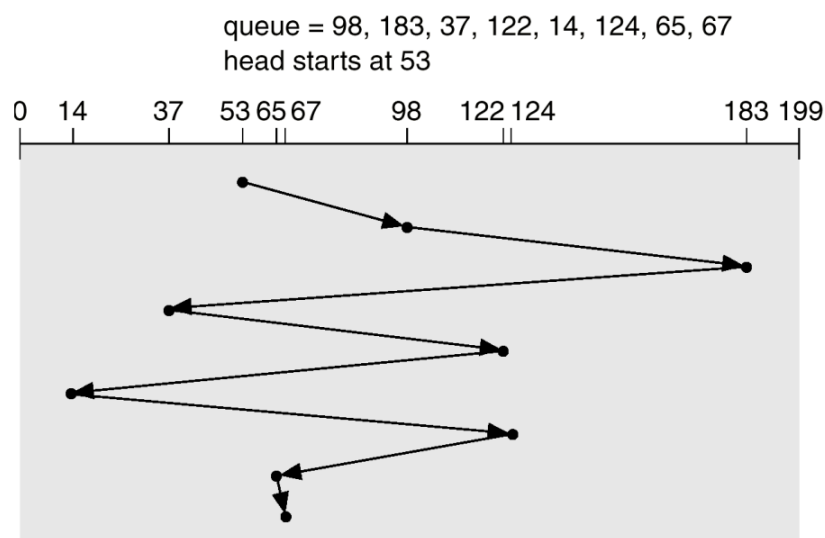
Disadvantages of FCFS

Here are some of the disadvantages of First Come First Serve.

Does not try to optimize seek time

May not provide the best possible service

Example:



Program

```
#include<iostream>
```

```
#include<vector>
```

```
using namespace std;
```

```
void FCFS(int n, int arr[], int head) {
```

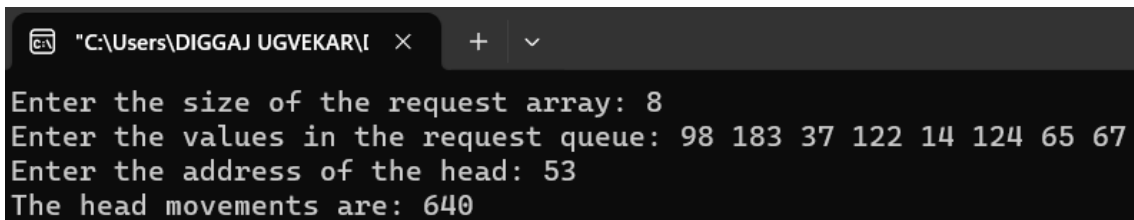
```
    int distance = 0;
```

// Traverse the array and keep adding the absolute difference to the distance to get the head movements

```
for (int i = 0; i < n - 1; i++) {  
    distance += abs(arr[i] - arr[i + 1]);  
}  
// Add the initial head movement to the first request  
distance += abs(head - arr[0]);  
cout << "The head movements are: " << distance;  
}  
int main() {
```

```
    // Take the seek sequence  
    int size, head;  
    cout << "Enter the size of the request array: ";  
    cin >> size;  
    int seek[size];  
    cout << "Enter the values in the request queue: ";  
    for (int i = 0; i < size; i++) {  
        cin >> seek[i];  
    }  
    cout << "Enter the address of the head: ";  
    cin >> head;  
    FCFS(size, seek, head);  
    return 0;  
}
```

Output



```
"C:\Users\DIGGAJ UGVEKAR\I" x + v  
Enter the size of the request array: 8  
Enter the values in the request queue: 98 183 37 122 14 124 65 67  
Enter the address of the head: 53  
The head movements are: 640
```

Conclusion

First In First Out Page replacement algorithm was studied and implemented successfully.

DISK SCHEDULING ALGORITHM

Experiment No: 7

Date: / /23

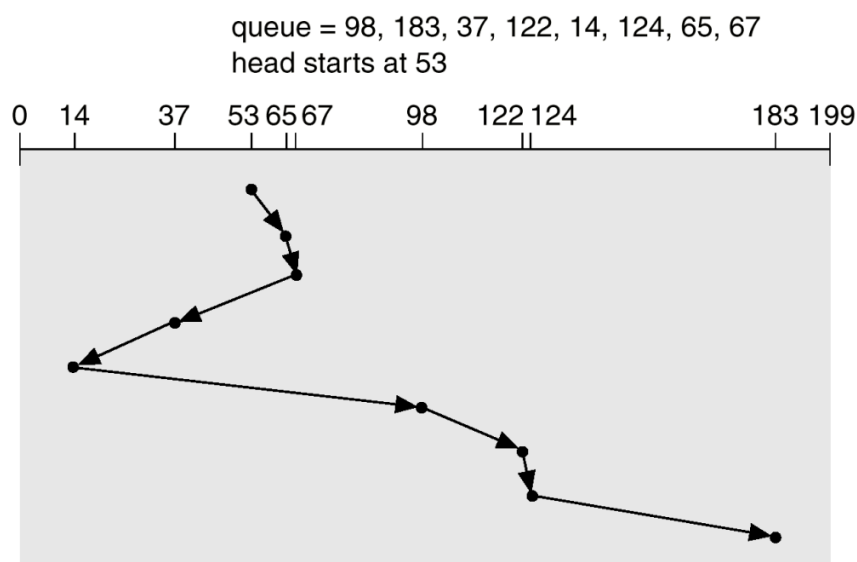
Aim: b) To implement Shortest Seek Time First (SSTF) Disk Scheduling algorithm.

Theory:

SSTF, which stands for Shortest Seek Time First, is a disk scheduling algorithm used in computer storage systems. The primary goal of disk scheduling algorithms is to minimize the time it takes to access and retrieve data from the disk. SSTF is a type of elevator algorithm that selects the request with the shortest seek time, which is the time it takes for the disk's read/write head to move to the desired track.

The main advantage of SSTF is that it minimizes the average seek time, resulting in faster data access times. However, one of its drawbacks is that it can lead to starvation of some requests if there is a constant stream of requests at or near the current head position, preventing requests at other locations from being serviced.

Example:



Program

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int sstf(vector<int> &requests, int current_head) {
    int total_seek_time = 0;
    int head=current_head;
```

```

vector<int> seekSequence;
while (!requests.empty()) {
    int min_seek = INT_MAX;
    int min_index = -1;
    for (int i = 0; i < requests.size(); i++) {
        int seek = abs(current_head - requests[i]);
        if (seek < min_seek) {
            min_seek = seek;
            min_index = i;
        }
    }
    seekSequence.push_back(requests[min_index]);
    total_seek_time += min_seek;
    current_head = requests[min_index];
    requests.erase(requests.begin() + min_index);
}

//prints the seek sequence
cout<<"seek sequence is: "<<head<<"-->";
for(int i=0;i<seekSequence.size();i++){
    cout<<seekSequence[i]<<"-->";
    if(i==seekSequence.size()-1){
        cout<<seekSequence[i];
    }
}

cout<<endl;
return total_seek_time;
}

int main() {
    int qsize,current_head;

    cout <<"enter the queue size:";
    cin>>qsize;
    vector<int> requests;

```

```

        requests.resize(qsize);

        cout<<"enter the values of request queue"<<endl;

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

            cin>>requests[i];

        }

        cout<<"enter the current pos of head"<<endl;

        cin>>current_head;

        int seek_time = sstf(requests, current_head);

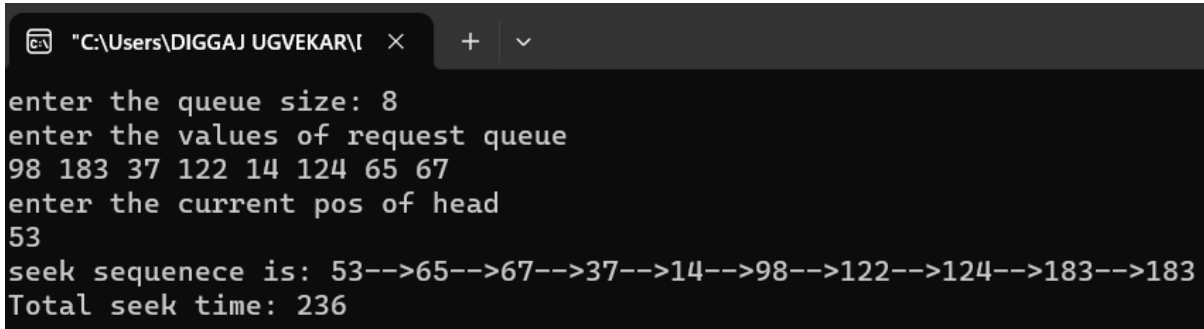
        cout << "Total seek time: " << seek_time << endl;

        return 0;

    }

```

Output



```

C:\Users\DIGGAJ UGVEKAR\I >
enter the queue size: 8
enter the values of request queue
98 183 37 122 14 124 65 67
enter the current pos of head
53
seek sequence is: 53-->65-->67-->37-->14-->98-->122-->124-->183-->183
Total seek time: 236

```

Conclusion

SSTF disk scheduling algorithm algorithm was studied and implemented successfully.

DISK SCHEDULING ALGORITHM

Experiment No: 7

Date: / /23

Aim: c) To implement SCAN disk scheduling algorithm.

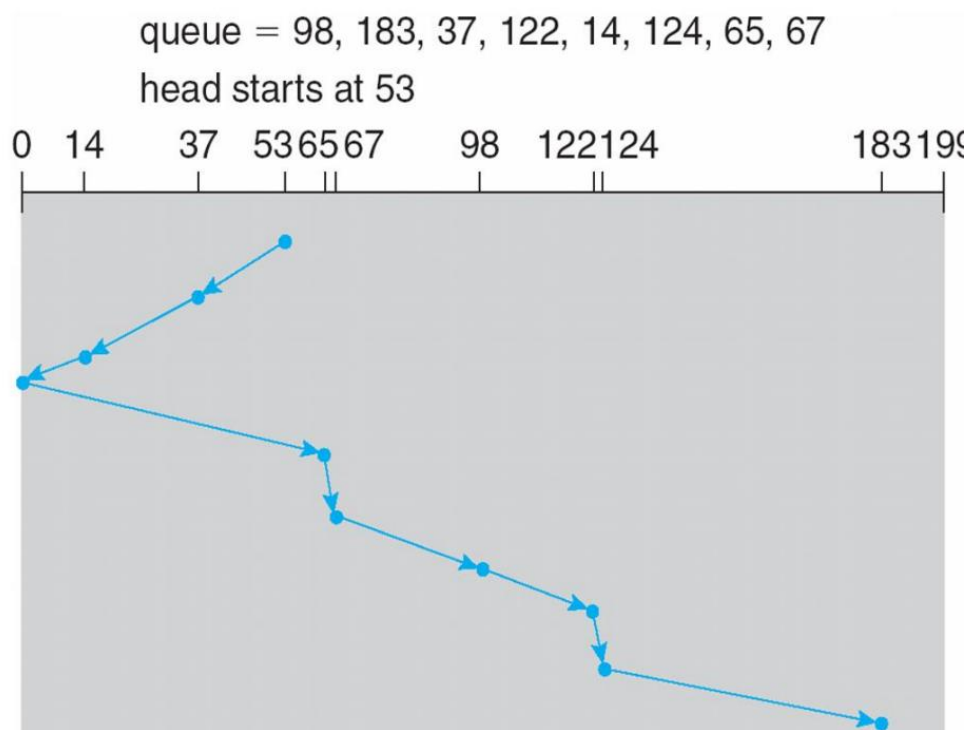
Theory:

SCAN, also known as Elevator is a disk scheduling algorithm commonly used in computer storage systems. The primary purpose of disk scheduling algorithms is to optimize the order in which disk I/O requests are serviced to minimize the total seek time and enhance overall system performance.

The SCAN algorithm is efficient and helps prevent starvation because it services requests in both directions. However, it may lead to some waiting time for requests that are located at the opposite end of the disk from the current position of the disk arm.

SCAN and C-SCAN are widely used in practice and strike a balance between simplicity and effectiveness for many disk scheduling scenarios.

Example:



Program

```
#include <bits/stdc++.h>

using namespace std;

int scan(vector<int> &rq, int current_head) {
    int head = current_head, distance;
    vector<int> seq;
    sort(rq.begin(), rq.end());
    auto it = lower_bound(rq.begin(), rq.end(), head);
    int mid = *it;
    int midPos;
    int var=1;
    while(var==1){
        if (head > *it) {
            midPos = it - rq.begin();
            var=0;
        } else {
            --it;
            midPos = it - rq.begin();
        }
    }

    for (int i = midPos; i >= 0; i--) {
        distance += abs(rq[i] - current_head);
        if(i==0){
            distance+=rq[0];
            current_head=0;
            seq.push_back(rq[i]);
            seq.push_back(0);
        }else{
```

```

        current_head = rq[i];
        seq.push_back(rq[i]);
    }
}

for(int i=midPos+1;i<rq.size();i++){
    distance += abs(rq[i] - current_head);
    current_head = rq[i];
    seq.push_back(rq[i]);
}

cout << "Seek sequence is: " << head << "-->";
for (int i = 0; i < seq.size(); i++) {

    if (i == seq.size() - 1) {
        cout << seq[i];
    }else{
        cout << seq[i] << "-->";
    }
}

cout << endl;

return distance; // Return the total seek distance
}

int main() {
//    vector<int> requests = {98, 183, 37, 122, 14, 124, 65, 67};
//    int current_head = 53;
    int qsize,current_head;

    cout <<"enter the queue size:";
    cin>>qsize;

```



```

        vector<int> requests;

        requests.resize(qsize);

        cout<<"enter the values of request queue"<<endl;
        for(int i =0;i<qsize;i++){
            cin>>requests[i];
        }

        cout<<"enter the current pos of head"<<endl;
        cin>>current_head;

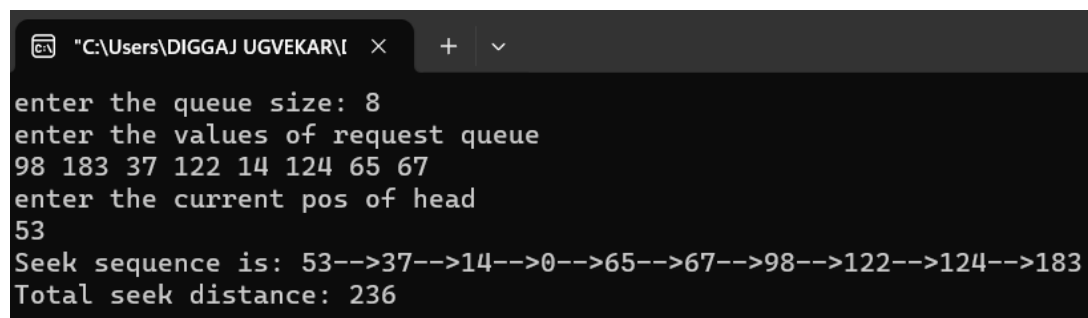
        int seek_distance = scan(requests, current_head);

        cout << "Total seek distance: " << seek_distance << endl;

        return 0;
    }

```

Output



```

C:\Users\DIGGAJ UGVEKAR\I
enter the queue size: 8
enter the values of request queue
98 183 37 122 14 124 65 67
enter the current pos of head
53
Seek sequence is: 53-->37-->14-->0-->65-->67-->98-->122-->124-->183
Total seek distance: 236

```

Conclusion

SCAN disk scheduling algorithm was studied and implemented successfully.

DISK SCHEDULING ALGORITHM

Experiment No: 7

Date: / /23

Aim: d) To implement C-SCAN disk scheduling algorithm.

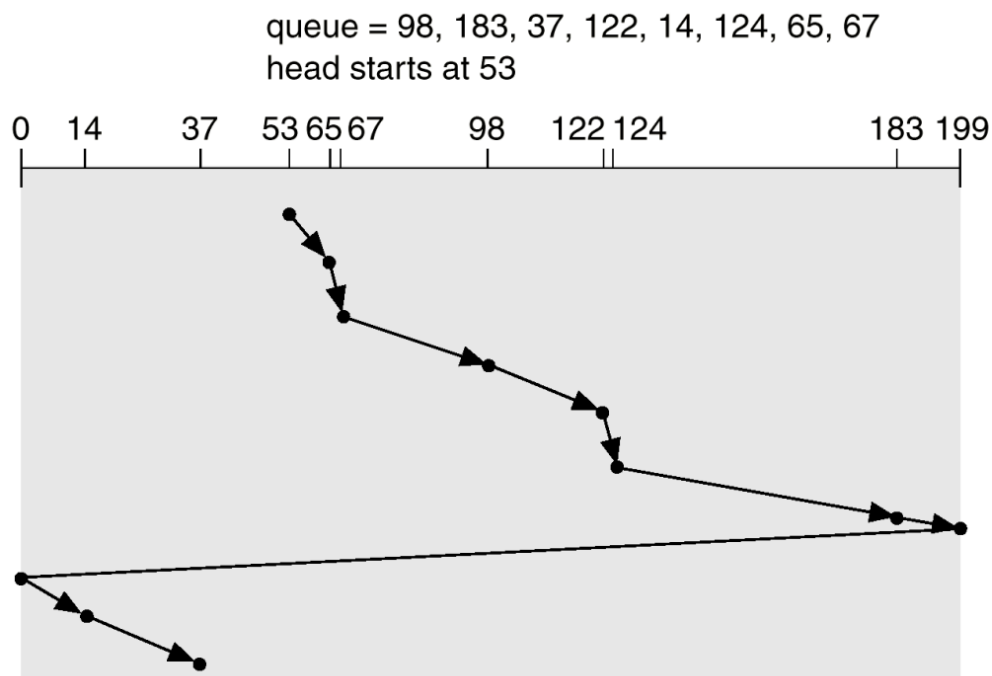
Theory:

C-SCAN, or Circular SCAN, is a disk scheduling algorithm that is a variation of the SCAN algorithm. It is designed to provide a more predictable and consistent service for disk I/O requests. C-SCAN minimizes waiting time for requests that are far from the current position of the disk arm.

C-SCAN is particularly useful in scenarios where the workload is predictable and there is a need to avoid prolonged waiting times for requests. Since the arm always moves in the same direction and immediately wraps around after reaching one end, it ensures a more consistent service for all requests, preventing any request from waiting indefinitely.

One drawback of C-SCAN is that it may introduce some additional seek time for requests that are near the "wrap-around" point, as they have to wait for the arm to complete a full revolution. However, overall, C-SCAN is a well-balanced algorithm for disk scheduling in certain scenarios

Example:



Program

```
#include <bits/stdc++.h>

#include <iostream>

#include<vector>

using namespace std;

int scan(vector<int> &rq, int current_head,int limit) {

    int head = current_head, distance;

    vector<int> seq;

    sort(rq.begin(), rq.end());

    auto it = lower_bound(rq.begin(), rq.end(), head);

    int mid = *it;

    int midPos;

    int var=1;

    while(var==1){

        if (head < *it) {

            midPos = it - rq.begin();

            var=0;

        } else {

            ++it;

            midPos = it - rq.begin();

        }

    }

    for (int i = midPos; i < rq.size(); i++) {

        distance += abs(rq[i] - current_head);

        if(i == rq.size()-1){

            distance+=abs(rq[rq.size()-1]-limit)+limit;

            current_head=0;

            seq.push_back(rq[i]);

            seq.push_back(limit);

            seq.push_back(0);

        }else{
```

```

        current_head = rq[i];
        seq.push_back(rq[i]);
    }
}

for(int i=0;i<midPos;i++){
    distance += abs(rq[i] - current_head);
    current_head = rq[i];
    seq.push_back(rq[i]);
}

cout << "Seek sequence is: " << head << "-->";
for (int i = 0; i < seq.size(); i++) {
    if (i == seq.size() - 1) {
        cout << seq[i];
    }else{
        cout << seq[i] << "-->";
    }
}

cout << endl;

return distance; // Return the total seek distance
}

int main() {
//   vector<int> requests = {75, 90, 40, 135, 50, 170, 65, 10};
//   int current_head = 45;

    int qsize,current_head;

    cout <<"enter the queue size:";

    cin>>qsize;

    vector<int> requests;

    requests.resize(qsize);

    cout<<"enter the values of request queue"<<endl;

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

```

```

        cin>>requests[i];
    }
    cout<<"enter the current pos of head"<<endl;
    cin>>current_head;

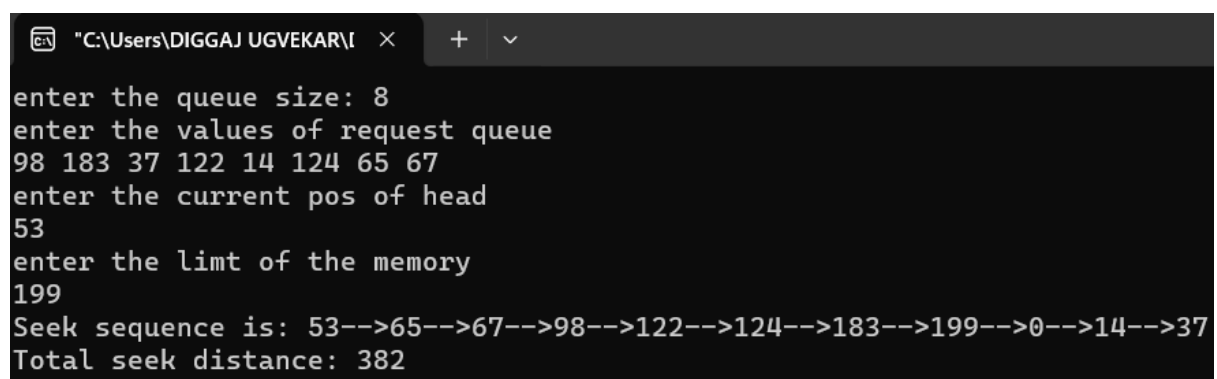
    int limit;
    cout<<"enter the limit of the memory"<<endl;
    cin>>limit;
    int seek_distance = scan(requests, current_head,limit);

    cout << "Total seek distance: " << seek_distance << endl;

    return 0;
}

```

Output



```

C:\Users\DIGGAJ UGVEKAR\I
enter the queue size: 8
enter the values of request queue
98 183 37 122 14 124 65 67
enter the current pos of head
53
enter the limit of the memory
199
Seek sequence is: 53-->65-->67-->98-->122-->124-->183-->199-->0-->14-->37
Total seek distance: 382

```

Conclusion

C-SCAN disk scheduling algorithm was studied and implemented successfully.

DISK SCHEDULING ALGORITHM

Experiment No: 7

Date: / /23

Aim: e) To implement LOOK disk scheduling algorithm.

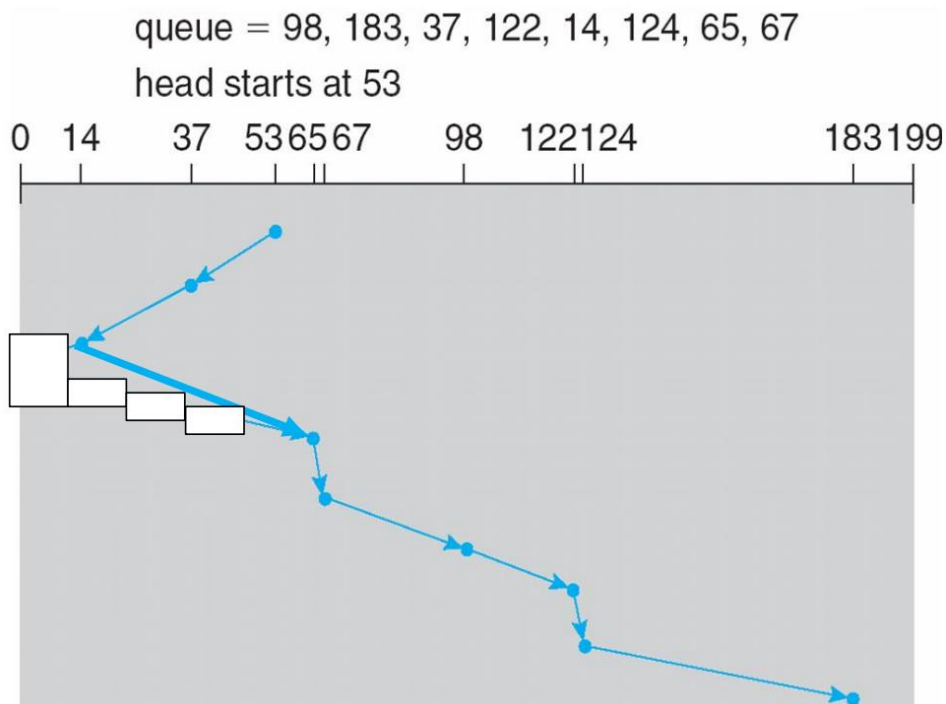
Theory:

LOOK is another disk scheduling algorithm that is used to optimize the order in which disk I/O requests are serviced. It is similar to the SCAN algorithm, but instead of always moving the disk arm to the end of the disk and reversing direction, LOOK only reverses direction if there are no requests pending in the current direction. This makes LOOK more efficient than SCAN in certain scenarios.

LOOK is designed to be more efficient than SCAN because it avoids unnecessary movement to the end of the disk when there are no requests in that direction. This can reduce the average seek time and improve overall disk performance. However, like SCAN, LOOK may still lead to some waiting time for requests that are far from the current position of the disk arm.

The effectiveness of LOOK depends on the nature of the disk I/O workload and access patterns. It is a good choice in scenarios where there is a mix of I/O requests distributed across the disk.

Example:



Program

```
#include <iostream>

#include <iomanip>

#include <algorithm>

using namespace std;

int total_head_movement = 0, n, current_head;

int request[30];

void right_display(){
    for (int i = 0; i < n; i++){
        if (request[i] < current_head)
            continue;
        else
            cout << request[i] << " ";
    }
    for (int i = n - 1; i >= 0; i--){
        if (request[i] > current_head)
            continue;
        else
            cout << request[i] << " ";
    }
}

void left_display(){
    for (int i = n - 1; i >= 0; i--){
        {
            if (request[i] > current_head)
                continue;
            else
                cout << request[i] << " ";
        }
    }
    for (int i = 0; i < n; i++)
```

```

    {
        if (request[i] < current_head)
            continue;
        else
            cout << request[i] << " ";
    }
}

int main()
{
    int block, direction;
    cout << "Enter the Block Size of the Disk: ";
    cin >> block;
    cout << "Enter the number of Disk Requests: ";
    cin >> n;
    cout << "Enter the Disk Requests to be Searched : ";
    for (int i = 0; i < n; i++)
        scanf("%d", &request[i]);
    sort(request, request + n);
    cout << endl;
    cout << "Enter the Current Disk Head Position: ";
    cin >> current_head;
    cout << "Enter Direction [Left - 0; Right - 1]: ";
    cin >> direction;
    cout << "Track sequence: " << current_head << " ";
    if (direction == 0)
    {
        total_head_movement += (current_head - request[0]);
        total_head_movement += (request[n - 1] - request[0]);
        left_display();
    }
}

```

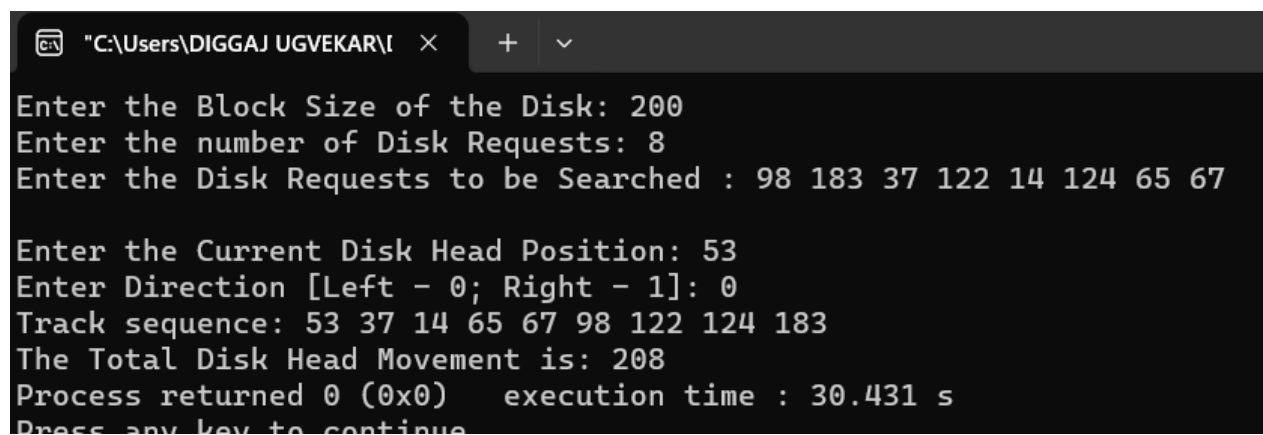


```

    }
else
{
    total_head_movement += (request[n - 1] - current_head);
    total_head_movement += (request[n - 1] - request[0]);
    right_display();
}
cout << endl
    << "The Total Disk Head Movement is: " << total_head_movement;
}

```

Output



```

C:\Users\DIGGAJ UGVEKAR\I >
Enter the Block Size of the Disk: 200
Enter the number of Disk Requests: 8
Enter the Disk Requests to be Searched : 98 183 37 122 14 124 65 67

Enter the Current Disk Head Position: 53
Enter Direction [Left - 0; Right - 1]: 0
Track sequence: 53 37 14 65 67 98 122 124 183
The Total Disk Head Movement is: 208
Process returned 0 (0x0)    execution time : 30.431 s
Press any key to continue

```

Conclusion

LOOK disk scheduling algorithm was studied and implemented successfully.

DISK SCHEDULING ALGORITHM

Experiment No: 7

Date: / /23

Aim: f) To implement C-LOOK disk scheduling algorithm.

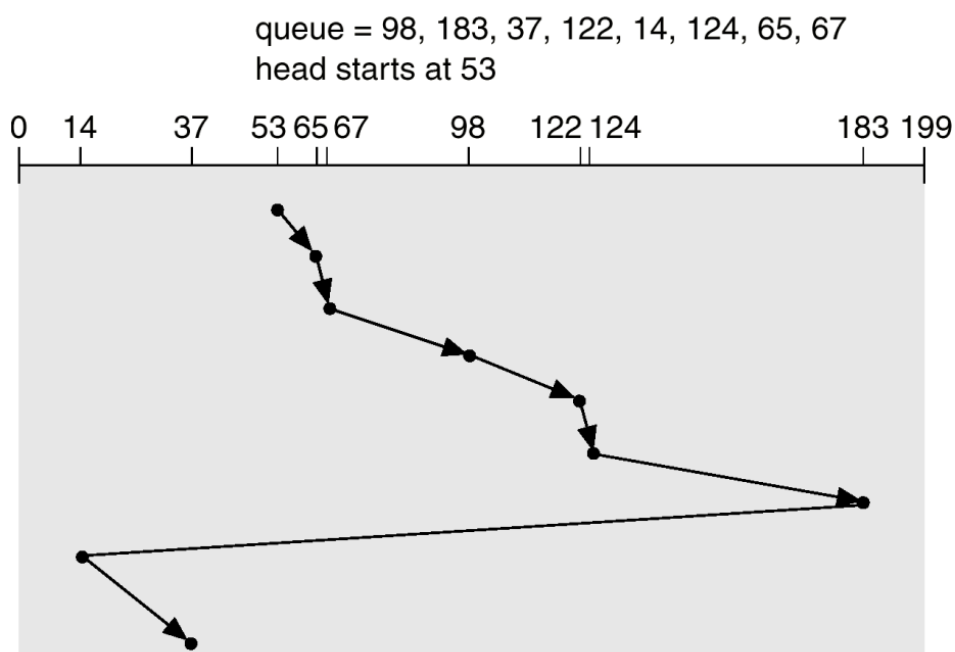
Theory:

C-LOOK, or Circular LOOK, is a disk scheduling algorithm that is a variation of the LOOK algorithm. Similar to C-SCAN, C-LOOK avoids the inefficiency of LOOK by not traversing the entire disk when there are no requests in the current direction. Instead, it immediately moves to the other end of the disk and continues servicing requests in that direction. C-LOOK aims to provide a more consistent and predictable service for disk I/O requests.

C-LOOK, like C-SCAN, is particularly useful in scenarios where there is a need to avoid prolonged waiting times for requests, providing a more consistent and fair service for all requests. The "wrap-around" behavior reduces the average seek time for requests that are located far from the current position of the disk arm.

One potential drawback is that, similar to C-SCAN, C-LOOK may introduce some additional seek time for requests that are near the "wrap-around" point. However, it is generally considered to be a well-balanced algorithm for certain disk scheduling scenarios.

Example:



Program

```
#include <bits/stdc++.h>

#include <iostream>

#include<vector>

using namespace std;

int scan(vector<int> &rq, int current_head,int limit) {

    int head = current_head, distance;

    vector<int> seq;

    sort(rq.begin(), rq.end());

    auto it = lower_bound(rq.begin(), rq.end(), head);

    int mid = *it;

    int midPos;

    int var=1;

    while(var==1){

        if (head < *it) {

            midPos = it - rq.begin();

            var=0;

        } else {

            ++it;

            midPos = it - rq.begin();

        }

    }

    for (int i = midPos; i < rq.size(); i++) {

        distance += abs(rq[i] - current_head);

        current_head = rq[i];

        seq.push_back(rq[i]);

    }

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

        distance += abs(rq[i] - current_head);

        current_head = rq[i];

    }

}
```

```

        seq.push_back(rq[i]);
    }
    cout << "Seek sequence is: " << head << "-->";
    for (int i = 0; i < seq.size(); i++) {
        if (i == seq.size() - 1) {
            cout << seq[i];
        }else{
            cout << seq[i] << "-->";
        }
    }
    cout << endl;

    return distance; // Return the total seek distance
}

int main() {
//    vector<int> requests = {98,183,37,122,14,124,65,67};
//    int current_head = 53;

    int qsize,current_head;

    cout <<"enter the queue size:";

    cin>>qsize;

    vector<int> requests;

    requests.resize(qsize);

    cout<<"enter the values of request queue"<<endl;

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

        cin>>requests[i];

    }

    cout<<"enter the current pos of head"<<endl;

    cin>>current_head;

    int limit;

    cout<<"enter the limt of the memory"<<endl;

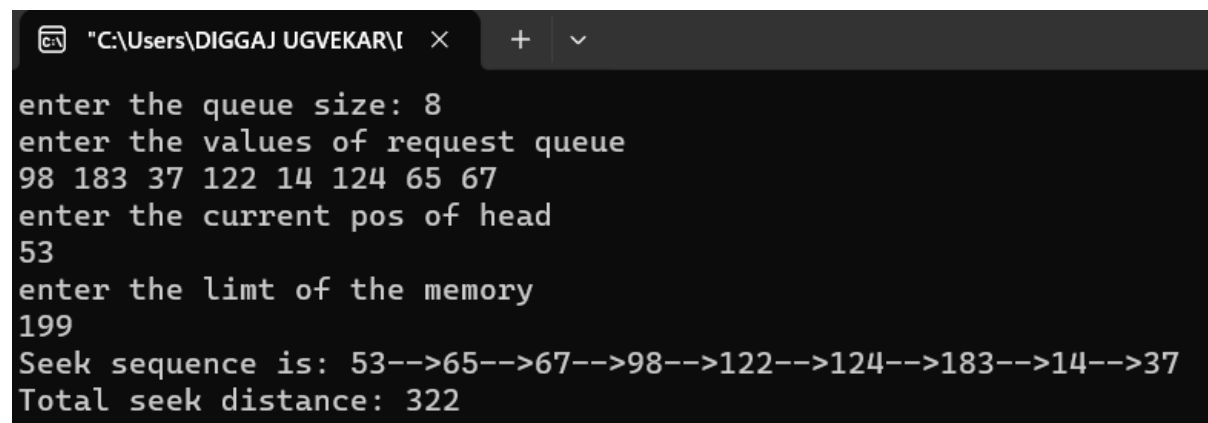
    cin>>limit;

    int seek_distance = scan(requests, current_head,limit);

```

```
    cout << "Total seek distance: " << seek_distance << endl;  
    return 0;  
}
```

Output



```
"C:\Users\DIGGAJ UGVEKAR\I  ×  +  v  
enter the queue size: 8  
enter the values of request queue  
98 183 37 122 14 124 65 67  
enter the current pos of head  
53  
enter the limit of the memory  
199  
Seek sequence is: 53-->65-->67-->98-->122-->124-->183-->14-->37  
Total seek distance: 322
```

Conclusion

C-LOOK disk scheduling algorithm was studied and implemented successfully.